RICE UNIVERSITY

Operating system support for server applications

by

Gaurav Banga

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Doctor of Philosophy

Approved, Thesis Committee:

Dr. Peter Druschel (Chair),
Assistant Professor,
Computer Science

Dr. Willy Zwaenepoel,
Noah Harding Professor,
Computer Science; and
Professor,
Electrical and Computer Engineering

Dr. Edward W. Knightly,
Assistant Professor,
Electrical and Computer Engineering

Houston, Texas

May, 1999

ABSTRACT

Operating system support for server applications

by

Gaurav Banga

General-purpose operating systems provide inadequate support for large-scale servers. Server applications lack sufficient control over scheduling and management of machine resources, which makes it difficult to enforce priority policies, and to provide robust and controlled service. For example, server applications cannot provide differentiated quality of service to requests from different clients.

The root cause of these problems is a fundamental mismatch between the original design assumptions underlying the resource management mechanisms of current general-purpose operating systems, and the behavior of modern server applications. In particular, the notions of protection domain and resource principal coincide in the process abstraction of current operating systems. Moreover, these operating systems provide insufficient control to an application over the resources that are consumed inside the kernel on behalf of the application. These aspects of current operating systems prevent a server process that manages large numbers of network connections, for example, from properly allocating system resources among those connections.

This dissertation addresses the lack of operating system support for fine-grained resource management in large-scale server systems. It starts by characterizing the nature of the mismatch between the design assumptions of current general-purpose operating systems, and the behavior of server applications. The traditional design of core operating system abstractions and APIs is reevaluated in the light of the requirements of server applications. This reevaluation leads to a set of novel operating system abstractions and APIs that serve to provide effective support for server applications.

To my mother.

ACKNOWLEDGEMENTS

First, I would like to thank Peter Druschel, my advisor and the chairman of my thesis committee, for his invaluable guidance and technical advice throughout the course of my graduate studies. Peter never failed to support my work and gave me a lot of freedom in choosing what I wanted to work on. His experience and fairness coupled with his honest and critical reviewing of my work was absolutely invaluable to me. I would also like to thank the other members of my committee, Willy Zwaenepoel and Ed Knightly, for their comments and suggestions, which greatly helped in improving the quality of this work. I am also indebted to Jeff Mogul for his comments and advice which were instrumental in crystallizing many aspects of this research.

Many people at Rice provided the supportive environment in which this work was performed. I am thankful to Vivek Pai, Johnny Chen and Mohit Aron— the other students in my group who provided constantly-available sounding boards for discussion and constructive criticism. Shriram Krishnamurthi provided "outside" criticism which ensured global sanity in my work. Special thanks to Iva Jean Jorgensen, Rhonda Guajardo, and Ann Lugg for all their help in my jostles with the workings of Rice.

Finally, thanks to my wife, Gita, for providing constant support during the course of this work.

Contents

Illustrations

Chapter 1

Introduction

The exponential growth of the World Wide Web (WWW) [167] and the widespread interest in sharing information and computing resources through networking has caused network servers to become one of the most important applications of large computer systems. Not so long ago, most computers were isolated entities and computing essentially involved processing of local data. Today, most computers are networked and a large fraction of the processing performed by a computer involves the use of services that are provided by other computers.

Computer systems that provide specialized services for other computers are called servers. A service typically provides access to some resource being managed by the server. These resources can be the physical computing resources of the server, e.g., disk space and CPU cycles, or more abstract, such as specific functionality implemented by the server, or some information stored in the server. Examples of server systems include file servers, which provide access to disk space resources; mail servers, which provide functionality to transmit and receive electronic mail; and WWW servers, which provide access to information as well as a variety of other resources. Specially constructed applications, called server applications, run on server systems to implement their functionality.

With the increased reliance on remote services, the user-perceived speed of computing depends on the ability of server systems to provide efficient service to their clients. Moreover, the scope of computing, i.e., the variety and range of tasks that can accomplished with a computer, depends on the functionality and robustness of the servers accessible to the computer. Unfortunately, despite significant increases in the raw speeds of both the server hardware and the physical network, the performance of server systems has increased only modestly. In addition, servers lack many qualities that are necessary to provide the robust behavior and rich functionality that the emerging Internet based global information infrastructure will require.

The fundamental reason behind the poor performance and modest functionality of servers is a mismatch between the original design assumptions underlying the interfaces and algorithms of the operating system (OS) that runs on server systems, and the behavior of server applications [113, 135, 148]. In particular, the OS fails to provide server applications with efficient access to the underlying hardware, resulting in poor performance. Also, the operating system does not provide server applications with fine-grained control over scheduling and management of server machine resources; this makes it difficult to provide scalable, robust and controlled service, and limits the functionality of servers.

This dissertation is concerned with addressing the lack of appropriate support for large-scale server applications in current general-purpose operating systems. In particular, it focuses on developing fine-grained resource management facilities for general-purpose operating systems. It starts by characterizing the nature of the mismatch between existing operating system features and the characteristics of modern server applications. The traditional design of core operating system abstractions is reevaluated in the light of the requirements of server applications. This reevaluation leads to a set of novel operating system abstractions and application programming interfaces (APIs) that serve to provide effective support for server applications.

1.1 Mismatch between existing OS services and the requirements of server applications

The mismatch between the design assumptions of current operating system interfaces and algorithms and their use by modern server applications is a result of a difference between the basic nature of server applications and that of traditional applications. Current general-purpose operating systems, e.g., UNIX, were originally designed to efficiently support large timesharing systems. Applications that ran on such systems were largely independent of each other and spent most of their time either in user mode, or blocked inside the OS kernel waiting for the completion of an operation on some relatively slow I/O device.

In these systems, the operating system's primary performance-critical task was to efficiently and safely multiplex the system's resources, i.e., CPU, memory and I/O devices, between the various application processes. The OS also tried to ensure fairness between processes by carefully controlling their total CPU and memory consumption. Since the cost of I/O operations was relatively high, overall utilization of I/O devices was more important than fairness in allocation of I/O resources to applications. As a result, classical OS algorithms for controlling I/O typically aimed at high

I/O throughput, while providing relatively weak fairness guarantees to application processes (e.g., merely assuring the absence of starvation).

In order to keep I/O devices fully utilized, I/O processing in these systems was generally performed at a higher priority* than most other system activities. Nevertheless, since these devices were relatively slow, I/O processing only consumed an insignificant fraction of the system's CPU cycles. Because of this reason, I/O services provided by the operating system were designed for programming simplicity, rather than CPU efficiency. The various I/O subsystems of the operating system were also designed with primary considerations other than CPU efficiency, such as efficiency of memory usage and modularity.

In contrast, modern server applications usually consist of a number of cooperating processes running on a dedicated machine. (In some types of servers, the server application is just a single process.) These applications often manage a large number of simultaneous I/O streams and are thus particularly I/O intensive. Since modern I/O devices, especially network I/O devices, have significantly higher speeds (relative to CPU and memory speeds) than the I/O devices of classical timesharing systems, I/O processing is computationally expensive. This fact, coupled with the I/O intensive nature of server applications, causes server processes to spend significant time executing inside the I/O subsystems of the OS kernel, i.e., server applications are *OS intensive*. These characteristics of server applications lead to various problems, the most important of which are briefly outlined below:

- The I/O intensive nature of server applications coupled with high costs of I/O processing in modern computer systems interacts poorly with the strictly-highest priority of I/O processing in current operating systems. An excess of service requests from clients can monopolize the resources of a server, leading to instability and severely degraded overload performance [111].
- The nature of the I/O subsystems and copy based I/O interfaces of current operating systems leads to poor locality of kernel code [50, 51, 128]. As modern computer systems rely heavily on caches to bridge the gap between CPU and memory speeds and provide good performance, poor locality of kernel code

*In this dissertation, the term "priority" is used in a loose sense to refer to the relative precedence of an activity. The use of "priority" is not intended to restrict the discussion to systems whose CPU schedulers use numeric priorities in their operation.

combined with the OS intensive nature of server applications leads to poor server performance.

- Many features of modern operating systems were designed without considerations of scaling to large sets of kernel resources being used by a single process; $\mathcal{O}(N)$ behavior that was acceptable with $N = 20$ is problematic when $N = 10,000$. For example, the select() system call, which is used heavily by some single-process server applications to monitor multiple I/O streams for activity, scales poorly with the total number of streams being considered, severely limiting server scalability [16, 15, 13].

- Server applications are often forced to use relatively heavyweight and general-purpose OS facilities for want of lightweight OS features with the desired semantics. For example, some WWW servers use a dedicated process to handle each client request. The rationale for this usage is that a process is the natural unit to encompass an independent computation. Unfortunately, however, such application architectures suffer from unnecessary overheads of frequent context-switching and communication between protection domains, and inefficient memory usage.

- The I/O interfaces of current operating systems do not support prioritization of I/O streams. This fact, in conjunction with the manner in which the I/O subsystems of current operating systems perform I/O processing, precludes server applications from providing certain important functionality to their clients, such as differentiated quality of service for requests from different clients.

A lot of recent work by the operating systems research community has focused on improving the performance of server applications. In particular, there has been a lot of research to adapt current operating systems to I/O intensive applications. These efforts have attempted to improve the performance of I/O subsystems by eliminating excess data movement [54, 130, 6, 158, 118], increasing cache effectiveness [33, 115], improving I/O subsystem structure [87, 38, 55, 113, 51] and removing operating system involvement in the I/O path [50, 53, 17, 88, 87].

Researchers and operating system vendors have also worked towards optimizing many kernel mechanisms that are used heavily by server applications. These include in_pcblookup() [101], which is used inside the kernel to classify an incoming packet to a connection, select() [15], which provides an event notification and delivery mechanism, and `ufalloc()` [15], which is the I/O endpoint descriptor allocation routine. Some recent research has also focused on improving server control mechanisms [16, 13, 78], and the overload behavior of servers [111].

The architecture of server application has also undergone a significant change, with server application authors trying to make better use of existing operating system services. For example, the original process-per-connection architecture [119, 8] of Internet servers is being superseded by single-process architectures [31, 174, 161, 11, 88], which avoid the overhead of inter-process context-switching and communication. There have also been efforts to reduce the total number of system calls executed by server applications by the use of innovative techniques [78, 130, 131, 118].

1.2 The resource management problem

While the work cited above has been fruitful, it has generally treated the operating system's core abstractions and its application programming interface (API) as a constant. This has frustrated efforts to solve thornier problems of server robustness and effective control over resource consumption. In particular, current systems generally provide poor control to server applications over system resources that are consumed inside the kernel, e.g., during I/O processing. This causes servers to suffer from poor overload behavior, as was mentioned before, and makes them vulnerable to various types of denial of service attacks, where a malicious client manages to consume all of the server's resources. The susceptibility of servers to such attacks has important implications for their robustness and security.

Moreover, this lack of control over kernel resource consumption precludes the design of server systems that can provide certain important functionality. For example, consider the following scenarios:

- A server may want to provide clients with differentiated service depending on a variety of factors. This may include, for example, consideration of any access charges paid for the service by a corporate client vis-a-vis no charges paid by an individual client.

- The administrators of a corporate network may desire to prioritize network traffic in the corporate network based on the degree of relevance to the objectives of the corporation. For example, it may be desired that strictly business-related traffic have higher priority than the traffic generated by employees surfing Web sites for personal entertainment. Similarly, the administrators of a corporate Web server may want to give a higher priority to requests from inside the corporate network, than to those from outside.

- A Web hosting service may want to provide differentiated quality of service (QoS) for requests to Web pages, perhaps based on the amount of money paid by the owner of the particular document being fetched [3]. Many Web hosting sites provide individual users with free service while requiring corporations to pay a fee. These sites may desire to offer QoS guarantees to their paying clients.

- A server may wish to limit the resource usage of particular classes of requests. For example, it may be desired that all requests for dynamic resources (CGI requests) to a Web server are limited to a fixed share of the server system's CPU resources.

- A Rent-A-Server host [164] may like to control the resource consumption of its various guest servers. For example, it may be desired that the total resource usage of each guest server is limited to no more than a fixed (or relative) fraction of the system's resources under full load. Moreover, this resource partition should be dynamic; any unused resources allocated to a underutilized guest server should be made available to other guest servers that want them.

Unfortunately, although such situations already exist today, and will only get more pervasive over time, servers running on current operating systems cannot provide the needed functionality effectively.

The reason behind the inability of server applications to address these scenarios correctly is the resource management subsystem of classical operating systems, i.e., the set of mechanisms that control how system resources are allocated to applications. The original assumptions underlying these mechanisms interact poorly with the requirements of modern server applications. There are three aspects of this poor interaction:

1. **Inappropriate kernel processing model.** In current general-purpose operating systems, resource management does not extend to the execution of significant parts of kernel code. In particular, the OS generally does not control or properly account for resources consumed during I/O processing. For example, most operating systems do network protocol processing in the context of

software interrupts, which execute at a statically higher priority than any application process, and whose execution is either charged to the unlucky process running at the time of the interrupt, or to no process (if the system was idle at the time of the interrupt).

2. **Inflexible scope of resource principals.** The resource management mechanisms of current operating systems are tied to the assumption that a process is what constitutes an independent activity†. Processes are the resource principals, i.e., those entities between which the resources of the system are to be shared.

3. **Inappropriate APIs.** Many operating system APIs, whose purpose is to allow application control over the scheduling of system activities and kernel resource consumption, are inadequate from the standpoint of server applications. The semantics of these APIs makes it difficult for applications to specify policies for kernel resource consumption.

The misaccounting of I/O processing in current systems, and the resulting lack of control over kernel resource consumption, coupled with the statically high priority of I/O processing, leads to many scheduling anomalies. These anomalies are manifested as priority inversion, decreased throughput and resource starvation of applications. Furthermore, these factors cause a server system to become unstable in the face of overload from the network.

The treatment of processes as resource principals by current operating systems is very problematic for server systems. The reason for this is as follows: As mentioned earlier, modern high-performance server applications use a single process to perform many independent activities in order to avoid the overhead of switching and communicating between different protection domains. For example, a Web server may manage hundreds or even thousands of simultaneously active independent network connections, all within the *same* process. Much of the resource consumption associated with these connections occurs in kernel mode, and is therefore beyond the direct control of the application. The operating system, however, considers the server process as a single resource principal; it cannot distinguish between the various independent

†I will use the term *independent activity* to denote a unit of computation for which the application wishes to perform separate resource allocation and accounting; for example, the processing associated with a single client request.

connections *within* this process. Thus, it is impossible for the server application to control which connections are given priority in resource consumption.

Finally, the lack of support in current operating system APIs for applications to specify appropriate priority policies for kernel activities, by itself, can cause significant priority inversion in the handling of client requests by a server. For example, the socket bind() API provided by current operating systems does not allow server applications to distinguish between connection requests from different sources. This can cause a misbehaving client to consume an unfair share of the resources of the server system, and thus adversely affect the acceptance and subsequent handling of requests from other well-behaved clients. The implementation and interfaces of other APIs, such as the select() system call, suffer from scalability problems, which also lead to priority inversion.

This mismatch between the design assumptions of the resource management facilities of current operating systems and their use by server applications leads to what will be referred to in this dissertation as *the resource management problem* of current operating systems. Stated in a single sentence, this problem concerns the lack of adequate operating system support for applications to exercise fine-grained and flexible control over resource consumption at all levels of the system.

Note that this resource management problem is different from pure performance problems in that using any amount of faster hardware will not help. Thus, an increase in the raw performance of server hardware will not solve this problem. Because of this fact, the resource management problem is critically important, and is likely to become more so in the future.

1.3 Special-purpose vs. general-purpose operating systems

The recent work towards improving server performance, which was cited earlier in this chapter, falls into two broad categories: One approach has been to try to keep existing operating system abstractions and interfaces intact, and incorporate all performance enhancements into the framework of current general-purpose operating systems, such as UNIX [140] and Windows NT [44]. The biggest advantage of this *compatible* approach is that it is easy to deploy any performance enhancing changes to the operating system. Its chief disadvantage lies in that the nature of many existing operating system interfaces and abstractions limit the type of performance enhancements applicable, and the degree to these modifications are successful in solving performance problems. Examples of this approach include the improvements to in_pcblookup() and select(), and the efforts to adapt server architectures to make better use of existing operating system facilities.

The other approach is far more radical and is best represented by the the exokernel work [87, 88]. The proponents of this approach advocate a customized operating system tailored specifically for servers. In a server operating system based on the exokernel, the application controls essentially all of the network protocol stack, including the device drivers, through a combination of library code and a novel kernel architecture. The exokernel provides a similar interface to the storage system. This allows the application to directly control the algorithms and data structures used to implement network communication and file I/O. A prototype Web server system [88] that the exokernel project has built uses many of the optimizations cited above to achieve an order of magnitude performance improvement over a server running on a conventional operating system.

The exokernel approach is a specific example of an *extensible operating system*. Other extensible systems, e.g., SPIN [21] and VINO [145], allow applications to customize OS behavior by safely downloading code into the OS kernel, and can also be used to implement a special-purpose server operating system. In addition, several commercial special-purpose operating systems exist that are not based on extensible kernels. An example of such a system is Network Appliance's *Data ONTAP* [120] operating system, which is the basis of their *NetCache* [121] and *Web Filer* [122] products. A common design goal of the various extensible operating system approaches, which distinguish them from such older, less sophisticated special-purpose systems, is to allow for reuse of core operating system functionality without reducing the flexibility needed to implement functionality in a system-specific manner.

The main advantage of the special-purpose server operating system approach is that it achieves much better performance than the compatible approach. The mismatch between server application requirements and operating system services, which was described earlier in this chapter, can be avoided by design. The biggest disadvantage of this approach is that it is incompatible with existing operating system interfaces and abstractions, and is thus hard to deploy. For example, although a *Web Filer* serves static documents much faster than any Web server running on top of a general-purpose operating system, it cannot serve requests for dynamically created documents by itself. Requests for dynamic resources are typically handled by third party components of a Web server that usually need a complete general-purpose operating system environment to execute. While a system like *Web Filer* can conceivably be enhanced to support a suitable subset of some general-purpose operating system environment, the complexity and variety of these third party components cause this problem to quickly degenerate into the problem of developing and maintaining a new general-purpose operating system, which supports all standard OS interfaces.

The resource management problem stated in the previous section is equally relevant to both general-purpose and special-purpose operating systems. While theoretically, the exokernel approach allows the application to directly control the resource consumption for all associated network communication and file I/O, there has been no work or specific proposals to develop the algorithms and abstractions that are needed to provide fine-grained resource management.

1.4 Thesis statement and contributions

The hypothesis of this dissertation is that current operating systems lack sufficient support for allowing server applications to exercise tight control over resource consumption in the system. This support is imperative for an application to explicitly allocate resource consumption among the independent activities that it manages. Effective resource management is infeasible if the operating system's view of an independent activity differs from that of the application, and if the system fails to account for large chunks of resource consumption. Yet it is crucial for a server to support accurately differentiated QoS among its clients, and to prevent overload from denial of service attacks, and to give its existing connections priority over new ones.

It is therefore necessary to construct a set of new operating system features, i.e., mechanisms, policies and interfaces, to support the specific resource management requirements of server applications. These mechanisms need to perform complete resource accounting at all levels of the system and allow control over resource usage at a very fine level of granularity. Without such explicit support, any attempt to solve the problems described in Section 1.2 will be necessarily ad hoc, and inherently limited in its effectiveness.

The goal of this dissertation is to develop support for fine-grained resource management in general-purpose operating systems. Towards this end, the contributions here are the analysis of the specific requirements of server applications and the creation of novel operating system abstractions and APIs as part of a concerted design to meet the requirements of server applications.

Specifically, this dissertation presents:

- **An analysis of the requirements of server applications.** This analysis characterizes the poor interaction between existing operating system services and the behavior of server applications, and quantifies its performance implications. It also identifies the specific resource management requirements of server applications.

- **A new model for resource management in operating systems.** This model is based on a new operating system abstraction called a *resource container* [14]. A resource container encompasses all system resources that a server uses to perform a particular independent activity, such as servicing a particular client connection. A container abstracts the system's notion of a resource principal at a very fine level of granularity. All user and kernel level processing for an activity is performed in the resource context of the corresponding container. The new resource management model allows fairly arbitrary interrelationships between protection domains, threads and resource containers, and can therefore support a wide range of resource management scenarios.

- **A novel I/O subsystem architecture** based on *lazy receiver processing* [10, 52]. This architecture allows kernel I/O processing for an activity to be charged to the appropriate resource container, and scheduled at the priority of the container. As a result, processing performed on behalf of an independent activity at any level of the system is correctly attributed to the activity.

- **A new set of operating system APIs** that allow the application to exercise complete control over how resources are consumed at all levels of the system. These APIs essentially allow the application to fully exploit the capabilities of the resource container mechanism.

- **An implementation and a rigorous quantitative evaluation** of the new abstractions and APIs presented here in the context of a commercial general-purpose operating system. This evaluation shows that resource containers are indeed effective in correctly handling a large variety of complex resource management scenarios in server systems.

The focus in this dissertation is to develop the new server-centric operating system services described above in the context of a widely-used general-purpose operating system. In keeping with the spirit of the term "general-purpose", the new abstractions and APIs introduced in this dissertation are targeted at supporting a broad range of server and server-like applications. However, a substantial portion of the research presented here, particularly the core resource container mechanism, is directly applicable to special-purpose systems, such as those based on the exokernel.

In the remainder of this dissertation, I will focus on WWW servers as a concrete example of a large-scale server application. The performance of such servers is particularly interesting, since they must often scale to thousands or millions of users. This focus is necessary in order to allow the discussion to include sufficient details to be meaningful. I would like to stress, however, that the core ideas described in this dissertation are applicable to a wide variety of server applications, as defined in general terms in the second paragraph of this chapter.

1.5 Dissertation overview

The rest of this dissertation is organized as follows: Chapter 2 presents background information for understanding this dissertation. Specifically, it describes the architecture of a typical high-performance WWW server application. It also presents a brief overview of how network processing is performed in commonly used general-purpose operating systems.

Chapter 3 presents a discussion of how current operating systems fail to meet the requirements of modern server applications. This description also quantifies the performance implications of these issues, and presents an outline of the specific resource management facilities that are necessary to properly support server applications.

In Chapter 4, the basic design of the resource container abstraction is presented. This chapter briefly describes the basic operation of a system based on resource containers, and the relationship between classical operating system abstractions, such as processes and threads, and resource containers. This chapter also describes some new APIs that allow the application to use the facilities provided by the resource container mechanism.

Chapter 5 describes the I/O subsystem architecture that is used in the new resource management model, providing many details of the working of a system based on resource containers. The mechanics of CPU scheduling in a container based system is the subject of Chapter 6. This discussion explores the design space of how resource containers can be used with various CPU scheduling algorithms.

The use of resource containers in server applications is described in Chapter 7. Several examples of how resource containers can be used to tightly control resource consumption in servers are presented. It is also shown how the fine-grained resource management capabilities of resource containers can be used to implement higher-level functionality, such as the ability to perform prioritized handling of clients.

Chapter 8 describes a prototype implementation of resource containers in the context of Digital UNIX [41], which is a widely-used commercial operating system. Issues that arise when mapping the general resource container mechanism onto a real operating system, including a number of important policy issues, are addressed here.

A quantitative evaluation of the ideas presented in this dissertation is described in Chapter 9. This evaluation attempts to substantiate the discussion in Chapter 7 by showing that resource containers are indeed capable of enforcing very general and fine-grained resource control policies in server systems.

Chapter 10 presents related work. It discusses how resource containers are related to other resource management abstractions that have been described in the literature. It also places the work of this dissertation in perspective with respect to the recent efforts to support quality of service in the Internet.

Finally, Chapter 11 summarizes the contributions and points out limitations of this dissertation. It also identifies future directions of research.

Chapter 2

Background

This chapter presents background information for understanding the rest of this dissertation. It begins by describing the architecture of a typical high-performance server application. It also traces how this architecture has evolved over time, in response to observations about the interaction between server application behavior and the nature of operating system services. As mentioned in Chapter 1, the discussion focuses on WWW servers. At a fundamental level, however, the issues discussed here also apply to other types of servers, such as mail servers and file servers. The use of a UNIX-like API is assumed; nevertheless, most of this discussion is valid for servers based on non-UNIX general-purpose operating systems, such as Windows NT [44].

This chapter also presents an overview of how commonly used general-purpose operating systems perform network protocol processing. To simplify the discussion, the focus is on the TCP/UDP/IP protocol suite, and on BSD-derived UNIX systems [102]. The working of other protocol suites, such as in System V-derived UNIX [9] systems and Windows NT [44], is similar.

2.1 Architecture of a high-performance WWW server

On the World Wide Web, clients and servers interact via the use of the Hypertext Transfer Protocol (HTTP) [20, 64], which follows a *request/response* paradigm. A HTTP client, e.g., a Web browser, needs to establish a TCP [134] connection to a WWW server before it can download a Web document stored at the server. The server application *listens* on a well-known port (usually port 80) for new connection requests. When a new connection request arrives, the operating system kernel delivers the connection to the server application via the `accept()` system call. The server application then waits for the client to send a request for data on this connection. The first two versions of HTTP (0.9 and 1.0) allowed for only a single request to be sent on a TCP connection, and the connection had to be torn down and reestablished before the next HTTP transaction could take place. In HTTP/1.1 [64], several requests may be sent serially over a single persistent TCP connection.

```
while(1) {
     /* wait for new connection */
     new_so = accept(listen_so, &client, &len);

     /* allocate connection descriptor, buffers etc. */
     conn_descr = alloc_conn_descrip(new_so);

     /* HTTP processing */
     /* read request and parse it */
     read_request(conn_descr);
     parse_request(conn_descr);

     /* obtain the document being requested */
     /* a Web server reads this from the local file system;
      * a proxy server gets this over the network
      */
     obtain_response(conn_descr);

     /* send HTTP response to client */
     send_response(conn_descr);

     /* done with HTTP processing, close connection */
     close_conn(conn_descr);
}
```

Figure 2.1 Basic pseudo-code for an HTTP server.

After the request arrives at the server, it is parsed, and then the server application returns a response on the same connection. Web servers typically obtain the response from the local file system, while proxy servers obtain responses from other servers; however, both kinds of server may use a memory cache to speed retrieval. A pseudo-code fragment that illustrates the inner-loop of a simple (pre-HTTP/1.1) Web server is shown in Figure 2.1. Stevens [155] describes the basic operation of HTTP servers in more detail.

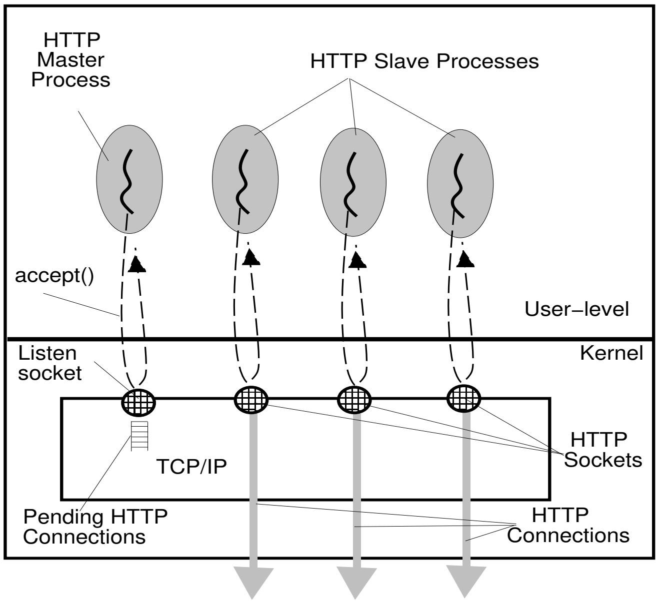

Figure 2.2 A process-per connection server with a master process.

The architecture of HTTP servers has undergone radical changes. Early servers forked a new process to handle each HTTP connection, following the classical UNIX model. The forking overhead quickly became a problem, and subsequent servers (such as the NCSA httpd [119]), used a set of pre-forked processes. In this model, which is shown in Figure 2.2, a master process accepts new connections and subsequently passes them to the pre-forked worker processes, which perform the remainder of HTTP processing.

The next innovation eliminates the master process. Instead, each pre-forked server process calls accept() directly to accept new connection requests. The Apache server [8] has this architecture, which is illustrated in Figure 2.3. Although it is not shown in the figure, all processes call accept() directly to remove new connections from the listen socket. Essentially, each server process runs the code shown in Figure 2.1.

Multi-process servers can suffer from large context-switching and inter-process communication (IPC) overheads [31, 142], so many recent servers use a single-process architecture. With only one process, there is little need for context-switching or IPC. In the event-driven model (Figure 2.4), the server uses a single thread to manage all of its connections. (In a multiprocessor server, as many threads as CPUs are used.) The server process uses the select() system call [153] to simultaneously wait for events

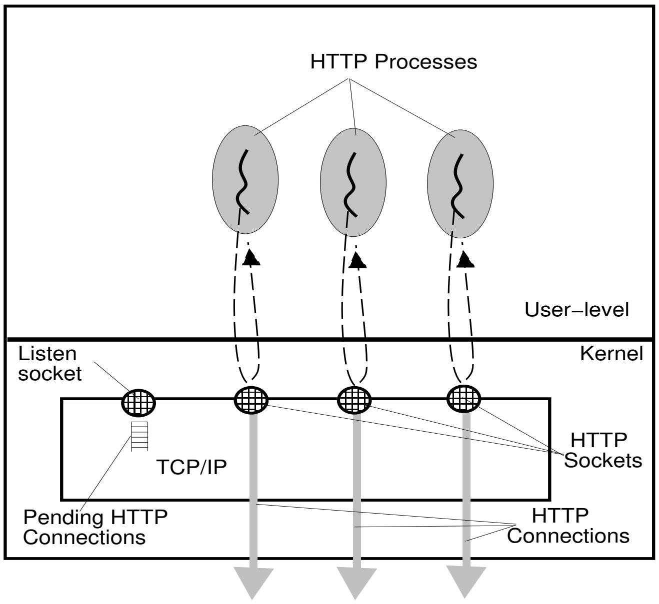

Figure 2.3 A process-per connection server without a master process.

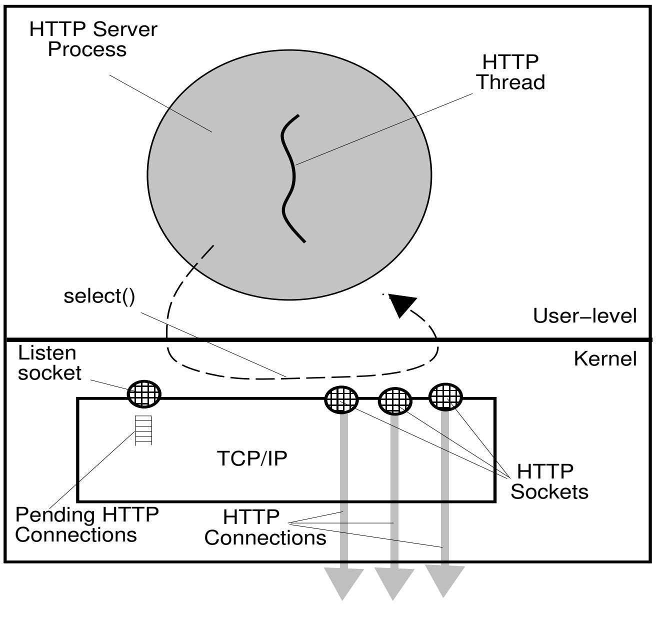

Figure 2.4 A single-process event-driven server.

on all connections that it is handling*. Squid [31, 151], Zeus [174], thttpd [161] and several research servers [11, 88] all use an event-driven architecture.

Pseudo-code for a select() based event-driven server is shown in Figures 2.5 and 2.6. Each call to select() specifies up to three bitmaps corresponding to the application's interest in read, write or exception events on the specified descriptors. (By setting the ith bit in a select bitmap, the application indicates interest in the corresponding type of event on descriptor i.) In Figure 2.5, two bitmaps, readfs and writefs, are used; this server application has no interest in exception events.

When select() delivers one or more events, the server's main loop invokes handlers for each ready connection. Figures 2.5 and 2.6 depict this as a scan through the bitmaps returned by select(). If the listen socket is ready for reading, the server calls accept() and initializes a new client connection. If some other socket (established connection) is ready, the server attempts to read the remainder of the request from the connection. If a socket is ready to be written to, the server attempts to write a chunk of the remainder of the server's response.

In the single-process multi-threaded model (Figure 2.7), each connection is assigned to a unique thread. These can either be user-level threads or kernel threads. The thread scheduler is responsible for time-sharing the CPU between the various server threads. Idle threads accept new connections from the listening socket. The front-end of the *AltaVista* search engine [4] has this architecture [27].

Multi-threaded servers have some practical advantages over event-driven servers. Current general-purpose operating systems lack efficient support to allow an application to wait for events on a large number of connections [15, 13, 16]. This leads to performance problems as delays and packet losses inherent in wide area networks cause WWW servers to often manage a large number of simultaneous connections [16, 12]. The lack of non-blocking I/O support in many current general-purpose operating systems also limits the performance of event-driven servers. In these systems, a single page-fault or disk read can cause an event-driven server process to be suspended for tens to hundreds of milliseconds. This prevents any progress, even on unrelated connections that could be handled without additional I/O, for long periods of time.

Other operating systems, which support non-blocking I/O, suffer from a lack of integration of services for disk and network I/O. For example, the POSIX aio

*System V UNIX based servers use the poll() system call [136] to perform this function; Windows NT based servers use either of two *Win32* API calls: WaitForMultipleObjects() or GetQueuedCompletionStatus() [139, 44].

```
while(1) {
    /* wait for event */
    init_event_sets(readfs, writefs);
    n_events = select(nfd, readfs, writefs, NULL, timo);

    /* now handle returned events */
    for(i = 0; i < MAXFDS; i++) {
        /* socket readable event? */
        if(FD_ISSET(i, readfs)) {
            /* event on listen socket? */
            if(IS_LISTEN_SOCKET(i)) {
                /* new request */
                new_so = accept(listen_so, &client, &len);

                /* allocate connection descriptor, buffers */
                CONN_DESCR[i] = alloc_conn_descrip(new_so);
            } else
            /* event on established connection? */
            if(CONN_DESCR[i]->state == ST_READING) {
                /* socket readable, read more of request */
                read_request(CONN_DESCR[i]);

                /* parse request, if complete change state */
                if(parse_request(CONN_DESCR[i]) == OK) {
                    CONN_DESCR[i]->state = ST_READING;
                    obtain_response(CONN_DESCR[i]);
                }
            }
        } else /* socket writable event? (see Figure 2.6) */
```

Figure 2.5 Pseudo-code for an event-driven Web server.

```
        /* socket writable event? (cont'd from Figure 2.5) */
        if((FD_ISSET(i, writefs)) &&
           (CONN_STATE[i]->state == ST_WRITING)) {
            /* socket writable */
            /* write a chunk of the remainder of the HTTP
             * response to the socket
             */
            write_response(CONN_STATE[i]);

            /* if response complete, close socket */
            if(CONN_STATE[i]->done) {
                /* if the socket LINGER option is used,
                 * this handler is a lot more complicated
                 */
                close(i);
                CONN_STATE[i] = NULL;
            }
        }
    }
}
```

Figure 2.6 Pseudo-code for an event-driven Web server (cont'd).

API [159] for non-blocking disk I/O does not work with sockets. Moreover, select() cannot be used to detect aio completion events. This forces a server application to use cumbersome methods to wait simultaneously for network and disk events. Multi-threaded servers might perform better than event-driven servers on such systems. Also, it is hard to exploit the parallelism of a multiprocessor system without at least some use of multiple kernel threads.

On the other hand, efficient support for huge pools of threads is not always available. This can result in poor performance because each connection that a multi-threaded server handles is processed by a dedicated thread, and, as mentioned above, the total number of these connections, and thus the corresponding threads, can get very large in busy servers [16, 12]. Moreover, the necessity for synchronization and thread-switching in a thread based system vis-a-vis none in a event-driven system has a negative impact on performance [127]. Also, an event based program can use a sin-

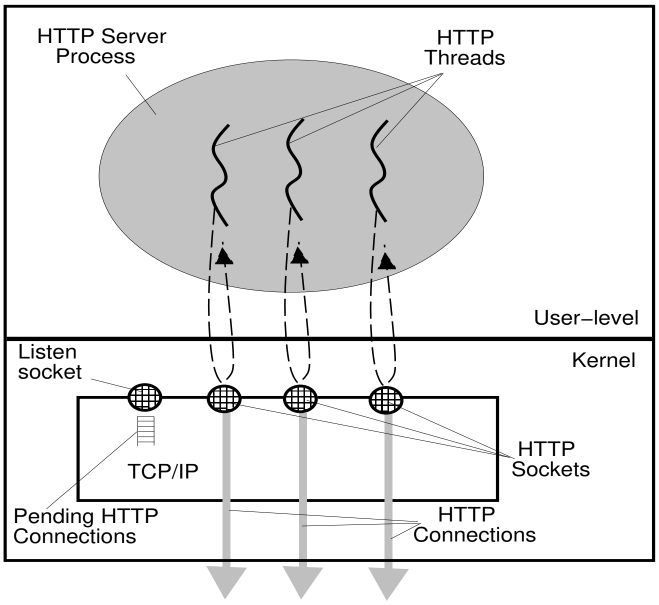

Figure 2.7 A single-process multi-threaded server.

gle execution stack; a thread based program uses multiple stacks, potentially putting more pressure on the data caches and the TLB, and thus degrading performance significantly [71, 30, 72, 60].

To get around these problems, some servers, such as the Inktomi *Traffic Server* [80], use a hybrid approach [163, 71]. In such servers, a moderate-size pool of threads, each of which is an event-driven state machine, is multiplexed among many connections. Events are used to control the assignment of connections to threads. AMPED [131] is another hybrid server architecture that uses an event-driven core and helper processes for I/O. Multiple processes are used instead of threads in order to get around the lack of support for kernel-level threads in many commonly-used general-purpose operating systems.

So far, I have assumed the use of static documents (or "resources", in HTTP terms). These are documents that are stored in the local filesystem of a Web server, and are served by simply reading the corresponding disk file and transmitting its contents over the client's TCP connection. HTTP also supports requests for dynamic resources, for which responses are created on demand, perhaps based on client-provided arguments. For example, a query to a Web search engine such as *AltaVista* [4] resolves to a dynamic resource.

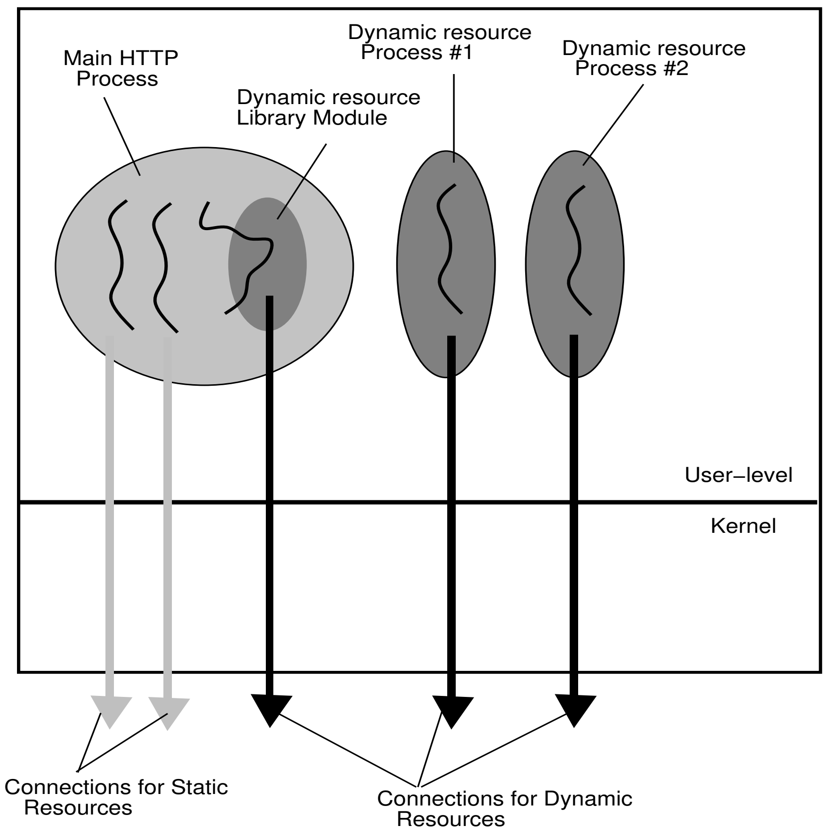

Figure 2.8 A typical high-performance Web server.

Dynamic responses are typically created by auxiliary third-party programs, which run as separate processes to provide fault isolation and modularity. To simplify the construction of such auxiliary programs, standard interfaces (such as CGI [29] and FastCGI [61]) have been defined to support communication between Web servers and these programs. The older interface, CGI, creates a new process for each request to a dynamic resource; the newer FastCGI allows persistent service processes for dynamic resource creation. Microsoft and Netscape have also defined library based interfaces [106, 123] to allow the construction of third-party modules that reside in the main server process, if fault isolation is not required; this minimizes overhead.

In summary, modern high-performance HTTP servers are implemented as a small set of processes. One main server process services requests for static documents; dynamic responses are created either by library code within the main server process, or, if fault isolation is desired, by auxiliary processes communicating with the main server process via a standard interface. Figure 2.8 depicts this situation. This is ideal, in theory, because the overhead of switching context between protection domains is incurred only if absolutely necessary. However, structuring a server as a small set of processes poses numerous important problems, as shown in the next chapter.

2.2 Overview of network processing

As mentioned in Chapter 1, server applications are particularly network-intensive and spend a substantial fraction of their execution time inside the operating system kernel. Thus, the manner in which operating systems perform network processing has significant performance implications for servers, as shall be seen throughout this dissertation. To provide the necessary background for these discussions, I will now present a brief overview of network processing in commonly used general-purpose operating systems. As mentioned earlier, I will focus on the working of the TCP/UDP/IP protocol suite in BSD-derived UNIX systems [102].

In BSD, network processing is performed by code that resides in the kernel. Figure 2.9 illustrates the BSD networking architecture. The network code consists of three distinct parts. The top part of this code consists of the *socket layer*. This code layer implements the sockets API, which forms part of the kernel's system call interface. User programs access the TCP/UDP/IP protocol suite using this API. In the socket layer, the kernel maintains the state of each communication endpoint in a data structure which is called a socket. At the user level, sockets are named via file descriptors.

The bottom part of the networking code is comprised of the network interface device driver. The various interrupt handlers of the device driver implement the hardware-specific portion of network processing.

The major part of network processing is performed in the middle part of the networking code, which includes the TCP, UDP and the IP protocol code. This middle part executes in the context of what is referred to as a software interrupt. A software interrupt is similar to a hardware interrupt, except it indicates a software event instead of a hardware event. For example, the network device driver signals the arrival of a new IP packet to the IP protocol code by posting a software interrupt. A software interrupt is handled by a software interrupt handler which executes asynchronously in the kernel. Software interrupt handlers have lower priority than hardware interrupt

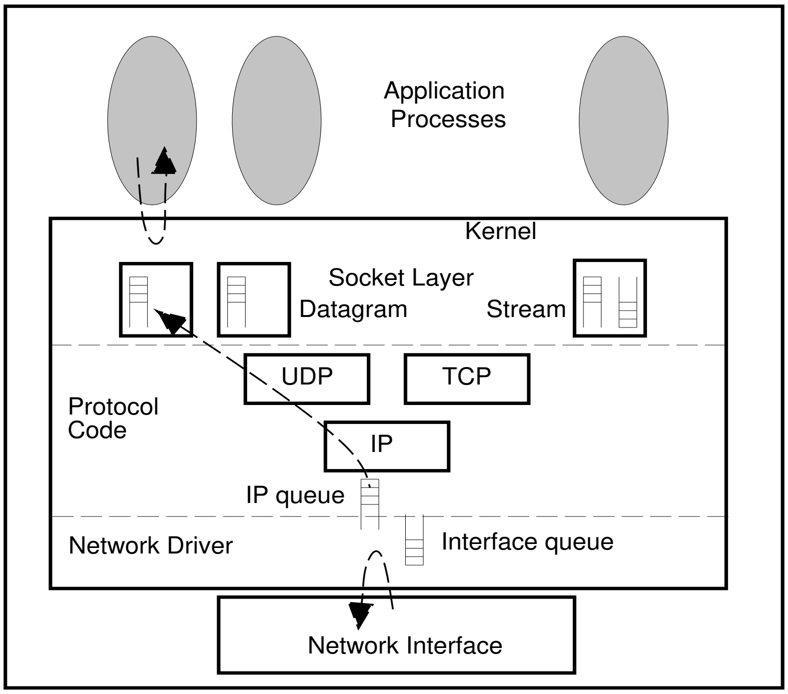

Figure 2.9 BSD network subsystem architecture

handlers, but higher priority than any application processing. I will now proceed to describe protocol processing in more detail.

On the receiving side, the arrival of a network packet is signaled by an hardware interrupt. The interrupt handler encapsulates the packet in an *mbuf*, queues the packet in the *IP queue*, and posts a software interrupt.

In the context of this software interrupt, the packet is processed by IP. After potential reassembly of multiple IP fragments, UDP's or TCP's input function is called, as appropriate. Finally—still in the context of the software interrupt—the packet is queued on the *receive socket queue* of the socket that is bound to the packet's destination port. Since the software interrupt has higher priority than any user process, whenever a user process is interrupted by a packet arrival, the protocol processing for that packet occurs *before* control returns to the user process. Also, as software interrupts have lower priority than hardware interrupts, the reception of subsequent packets can interrupt the protocol processing of earlier packets.

When an application process performs a receive system call[†] on the socket, the packet's data is copied from the mbufs into the application's address space. The mbufs are then dequeued and deallocated. This final processing step occurs in the context of the user process performing a system call.

On the sending side, data written to a socket by an application is copied into newly allocated mbufs. For datagram sockets (UDP), the mbufs are then handed to UDP and IP for transmission. After potential fragmentation, the resulting IP packets are then transmitted, or—if the interface is currently busy—placed in the driver's interface queue. All of these actions are executed in the context of the user process that performed the send system call on the socket. Packets queued in the interface queue are removed and transmitted in the context of the network interface's interrupt handler.

For stream sockets (TCP), the mbufs are queued in the socket's *outgoing queue*, and TCP's output function is called. Depending on the state of the TCP connection and the arguments to the send call, TCP makes a logical copy of all, some, or none of the queued mbufs, processes them for transmission, and calls IP's output function. The resulting IP packets are then transmitted or queued on the interface queue. Again, this processing occurs in the context of the application process performing a system call. As for UDP packets, data is removed from the interface queue and transmitted in the context of the network interface's interrupt handler.

Processing of any remaining data in the socket queue typically occurs in the context of a software interrupt. If TCP receives an acknowledgment, more data from the socket queue may be sent in the context of the software interrupt that was posted to process the incoming acknowledgment. Or, data may be sent in the context of a software interrupt that was scheduled by TCP to indicate a timeout. Data is not removed from the socket queue until its reception was acknowledged by the remote receiver.

CPU time consumed during the processing of network I/O is accounted for as follows. Any processing that occurs in the context of a user process performing a system call is charged to that process as system time. CPU time spent in software or hardware interrupt handlers is charged to the user process *that was interrupted*. Note that

---

[†]I use the term *receive system call* to refer to any of the five system calls available in UNIX to read data from a socket. The term *send system call* is used analogously to refer to system calls that write data to a socket.

in general, the interrupted process may be unrelated to the network communication that caused the interrupt.

A single pool of memory buffers (mbufs) is used to store all network data buffered in the kernel. The amount of data stored in any socket queue is limited on a per-socket basis. Most versions of UNIX impose a hard limit (approximately 16-64KB) on the size of any individual socket queue. Typically, larger limits apply to the IP and interface queues.

Before ending this discussion, let us consider in some detail the sequence of steps by which a Web server application accepts a new connection. Figure 2.10 illustrates these steps. The server application begins by creating a service socket. Subsequently, it invokes the `bind()` system call to name the socket, and the `listen()` system call to indicate to the protocol code in the kernel that it is ready to receive new connection requests on the service socket. After this, the server application executes the `accept()` system call to dequeue a new client connection from the listen socket.

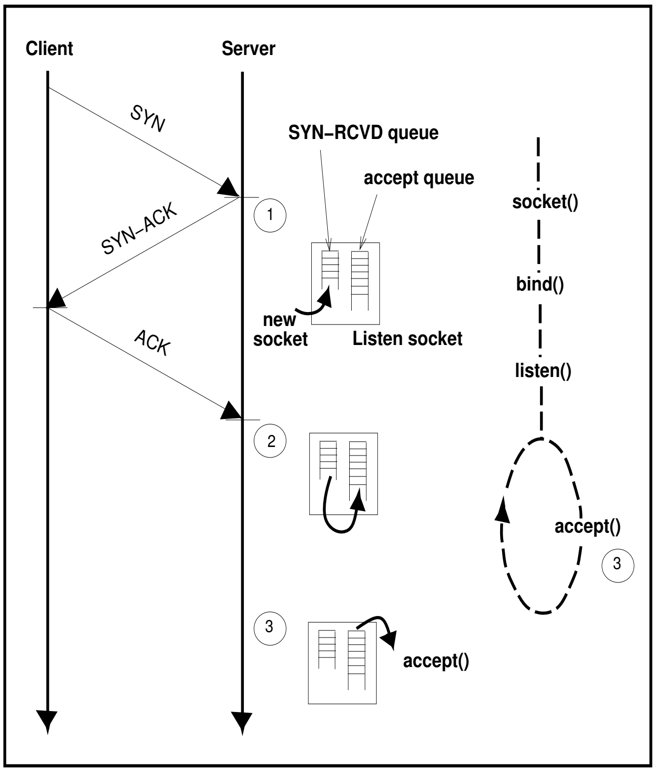

Figure 2.10 HTTP connection establishment timeline

When a connection establishment request (TCP SYN packet) from a client is received at the server (Figure 2.10, position 1), the server's TCP responds with a SYN-ACK TCP packet, creates a socket for the new, incomplete connection, and places it in the listen socket's SYN-RCVD queue. Later, when the client responds with an ACK packet (position 2), the server's TCP removes the socket created above from the SYN-RCVD queue and places it in the listen socket's queue of connections awaiting acceptance (accept queue). Each time the Web server process executes the `accept()` system call (position 3), the first socket in the accept queue of the listen socket is removed and returned.

In most UNIX based TCP/IP implementations, the kernel variable `somaxconn` limits the maximum *backlog* on a listen socket. This backlog is an upper bound on the sum of the lengths of the SYN-RCVD and accept queues. In the context of the discussion above, the server's TCP drops incoming SYN packets (Figure 2.10, position 1) whenever this sum exceeds a value of 1.5 times the backlog[‡].

In the next chapter, problems that arise because of the network processing architecture described above will be discussed.

[‡]In the System V Release 4 flavors of UNIX (e.g., Solaris) this sum is limited by $1 \times backlog$ rather than $1.5 \times backlog$.

Chapter 3

Shortcomings of current resource management facilities

This chapter discusses how the resource management frameworks of existing operating systems fail to effectively support server applications. It begins by describing the limitations of the process-centric resource allocation model of current operating systems. Several application scenarios are examined where the equivalence of protection domain and resource principal, as manifested in the process abstraction, proves to be problematic. We also discuss important problems that arise because of the manner in which current operating systems perform network processing.

Subsequently, the chapter discusses certain shortcomings of some operating system APIs from a resource management perspective. Specifically, we expose inadequacies in the event specification and delivery mechanisms, and in the socket `bind()` API. Throughout this chapter, a number of quantitative examples illustrate the performance implications of the issues in question.

3.1 The process-centric nature of operating systems

State of the art operating systems use sophisticated means of controlling the resources consumed by application processes. An operating system's scheduling and memory allocation policies attempt to provide fairness among resource principals, as well as graceful behavior of the system under various load conditions.

Most operating systems treat a process, or a thread within a process, as the schedulable entity. The process is also the "chargeable" entity for the allocation of resources, such as CPU time and memory. Recall from Chapter 1 that the term "independent activity", as used in this dissertation, denotes a unit of computation for which the application wishes to perform resource allocation and accounting independent of other computations. A basic design premise of such process-centric systems is that a process is the unit that constitutes an independent activity.

In these systems, a process also provides a virtual address space for a computation. This gives the process abstraction a dual function: it serves both as a protection domain and as a resource principal. As protection domains, processes provide isolation between applications. As resource principals, processes provide the operating system's resource management subsystem with accountable entities between which the system's resources are shared. This equivalence between protection domains and resource principals, however, is not always appropriate, as is discussed further below.

3.2 The distinction between protection domains and activities

This section argues that an operating system needs to clearly distinguish between the units of resource management and protection isolation. Several situations are considered where the natural boundaries of resource principals do not coincide with processes.

3.2.1 Classical applications

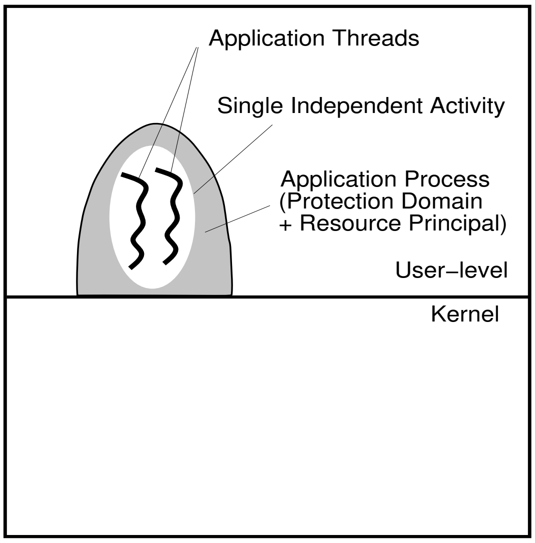

Figure 3.1 A classical application.

A classical application uses a single process to perform an independent activity. This process may use multiple threads to perform its computation. For such applications, the desired units of isolation and resource consumption are identical, and the process abstraction suffices. Figure 3.1 shows a mostly user-mode application, using one process to perform a single independent activity. Examples of such applications include a program to solve a numerical analysis problem, a text editor, a spreadsheet application, and a compiler.

In a network-intensive application, however, much of the processing is done in the kernel. The process is the correct unit for protection isolation, but it does not encompass all of the associated resource consumption; in commonly used operating systems, the kernel generally does not control or properly account for resources consumed during the processing of network traffic on behalf of an application.

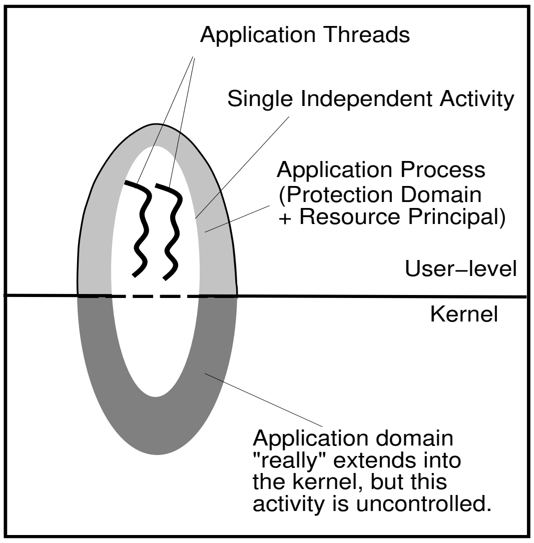

Figure 3.2 A classical network-intensive application.

As described in Section 2.2, most operating systems do protocol processing in the context of software interrupts, whose execution is either charged to the unlucky process running at the time of the interrupt, or to no process at all. Moreover, network processing for a process does not happen at the priority of the process; instead, all network processing in the system happens at a statically higher priority than any application processing. The performance implications of these issues are discussed in detail in Section 3.4. For the purposes of the current discussion, it suffices to note that system resources consumed in network processing are generally beyond the control of the application. The relationship between the application, process, resource principal and independent activity entities for a typical network-intensive application is shown in Figure 3.2.

Application	% CPU (user mode)	% CPU kernel
ftp transferring data at 92 Mbps	10	90
Netscape downloading at 320 KBps	31	69
Squid proxy handling 120 req/sec	22	78
thttpd Web server handling 400 req/sec	16	84
gcc compiling NFS mounted files	79	21
RealPlayer playing video at 30 frames/sec.	32	68

Table 3.1 User mode/kernel level resource usage for network-intensive applications.

Table 3.1 shows the percentage of CPU time spent in executing user-level and kernel code in some common network-intensive applications. These measurements were performed with the applications running on a 500Mhz Alpha (21164) processor based machine running Digital UNIX v4.0D. As can be seen, a substantial portion of the resource usage of these applications is incurred in the kernel. This large chunk of application resource usage is either misaccounted or unaccounted for, and not controlled by the resource management subsystem of the OS. As a result, the OS could, for example, allow a network-intensive application to get a significantly larger share of system resources than another, equal-priority, mostly-user mode application, as long as the user-mode resource usage of the two applications is the same.

3.2.2 Multi-process applications

Some applications are split into multiple protection domains. There are several motivating factors for this: One common reason is to provide fault isolation between different components of the application, which might come from different vendors. In this manner, a bug in one component of the application is prevented from crashing the entire application.

Another reason why a multi-process application architecture is used is to get around limitations imposed by operating systems on the total number of kernel resources that can be used by a single process. For example, most operating systems

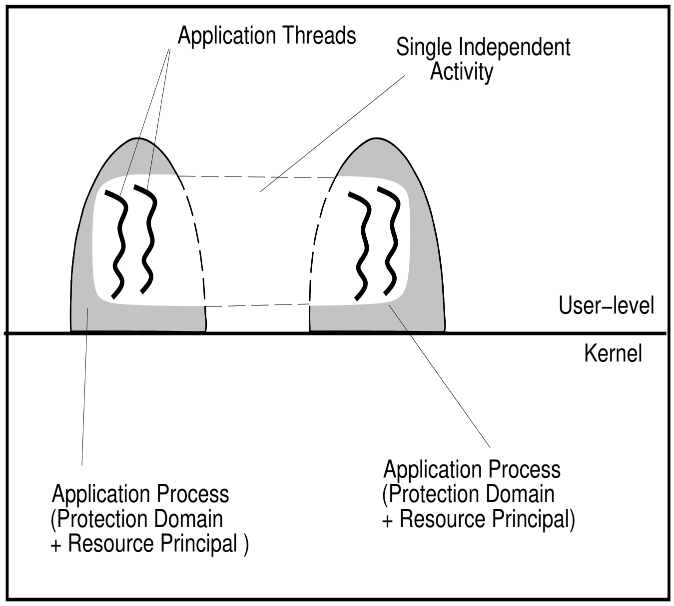

Figure 3.3 A classical multi-process application.

limit the total number of simultaneously open files (or sockets) that a given process may have. If an application needs to have more than these many files open simultaneously, it needs to be structured as a collection of cooperating processes. Similarly, if an operating system supports only a single kernel thread in each process, the programmer must split up his application into multiple processes in order to exploit any underlying machine parallelism.

One very common reason for why programs are split into multiple processes is for reasons of modularity and ease of composition of functionality. Many applications are structured as a collection of sub-programs, each of which is a also a complete application. An example of this is the program creation tool-chain in UNIX. This consists of the make program, the pre-processor(s), the various compiler stages, the assembler and the linker, all of which run as separate processes.

Applications that use multiple processes may still perform a single independent activity. A mostly user-mode multi-process application trying to perform a single independent activity is shown in Figure 3.3. In such an application, the desired unit of protection (the process) is different from the desired unit of resource management, which consists of all the processes of the application. This difference between the application's and the system's notion of the correct unit of resource consumption may cause a multi-process application to get an unfair share of the total resources of the system. This happens because the multi-process application appears to the operating system as multiple resource principals, and thus gets a larger share of system resources than any other single-process application.

For example, consider a scenario where a mostly user-mode three-process application belonging to *user A* runs in conjunction with two other mostly user-mode single-process applications belonging to *users B* and *C*. Assume that all processes are CPU-bound. In most UNIX systems, the three-process application of *user A* will get 60% of the system's CPU cycles, because it presents three out of the five resource principals that the operating system sees. This number is in contrast to the 33.33% "fair value" that we would normally consider as desirable.

3.2.3 Server applications

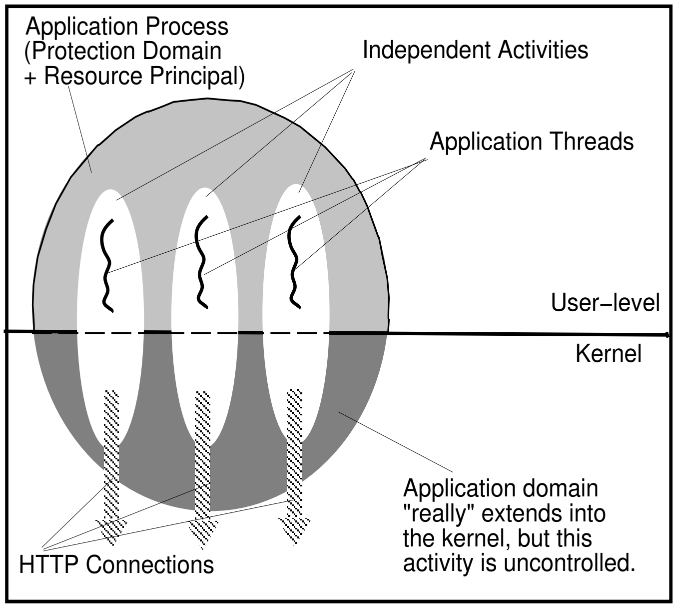

Figure 3.4 A single-process multi-threaded server.

In yet another scenario, an application consists of a single process performing multiple independent activities. Such applications use a single protection domain in order to reduce context-switching and IPC overhead. For these applications, the correct unit of resource management is smaller than a process: it is the set of all resources being used by the application to accomplish a single independent activity. Figure 3.4 shows, as an example, a single-process multi-threaded Internet server.

Unfortunately, in such servers, the operating system sees only one resource principal – the single server process. Since a significant portion of the activity of such a server takes place inside the kernel and is beyond the control of the server application, the server cannot control the individual resource usage of the various connections that it handles. Thus, it cannot provide differentiated quality of service (QoS) to its clients. Similarly, it cannot differentiate between requests to different resources and cannot prioritize their handling.

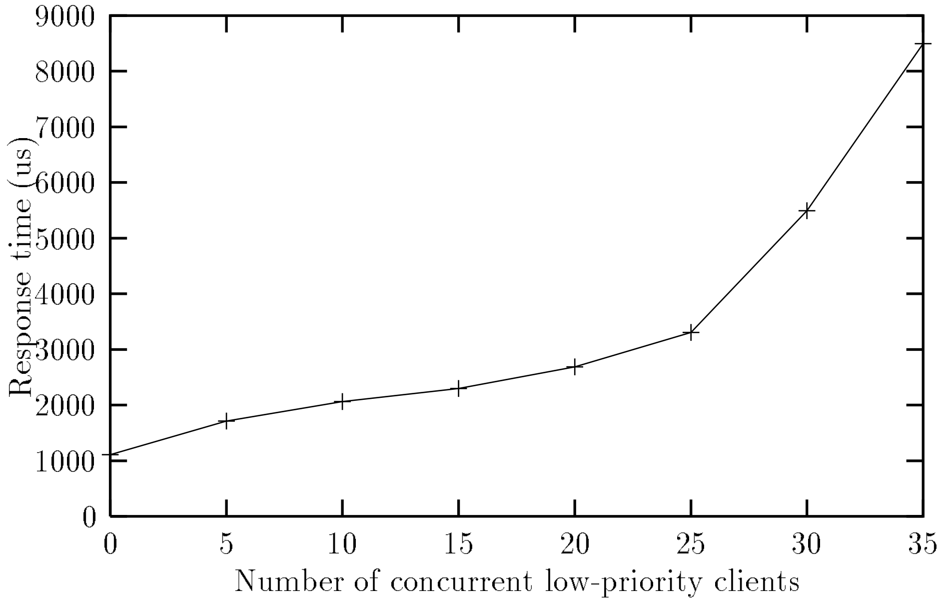

Figure 3.5 Variation of the response time seen by a high-priority client with concurrent load.

Figure 3.5 shows the results of a quantitative experiment designed to measure the ability of a single-process server to provide elevated quality of service to a high-priority, request source. In this experiment, an increasing number of low-priority clients were used to saturate a server, while a single high-priority client made requests of the server. These measurements were also performed on a 500Mhz Alpha (21164) processor based machine running Digital UNIX v4.0D. The server used was a slightly modified version of the `thttpd` [161] server. All requests were for the same (static) 1KB file.

The server application attempted to give preference to requests from the high-priority client by handling events on the socket corresponding to this client's connection before events on other sockets. The figure shows that, despite this preferential treatment, the response time experienced by the high-priority client increases sharply when there are enough low-priority clients to begin to saturate the server. As mentioned before, this happens because the server's operating system does not differentiate between the kernel-mode resource usage of the higher-priority client and that of the other clients. The protocol code in the kernel processes packets in order of arrival, and is oblivious of any priorities that the application may have assigned to connections.

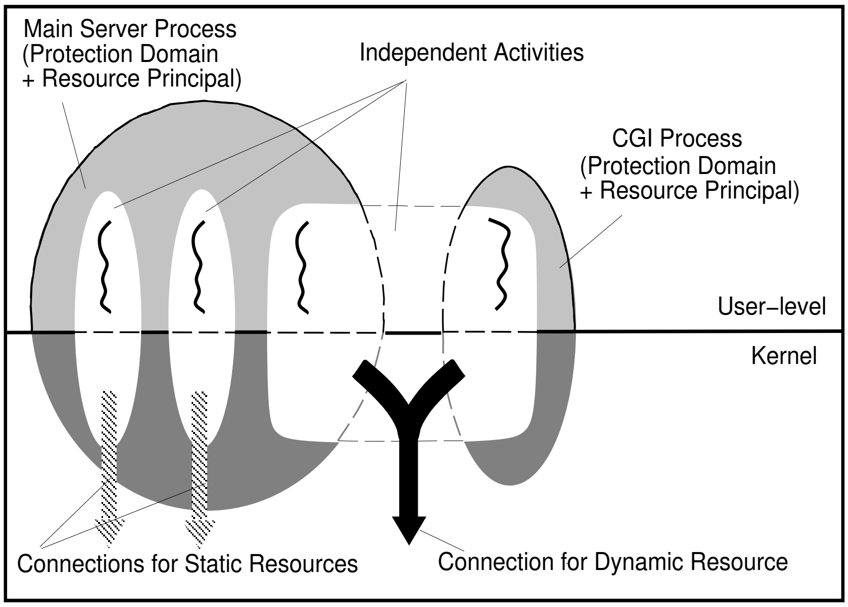

Figure 3.6 Independent activities in a Web server.

Real-world Internet servers typically combine the last two scenarios, as was described in Section 2.1. A single process usually manages all of the server's connections, but additional processes are employed when modularity or fault isolation is necessary. In this case, the desired unit of resource management includes part of the activity of the main server process, and also the entire activity of, for example, a CGI process.

This situation is illustrated in Figure 3.6, which shows a Web server handling two connections for static resources, and one request for a dynamic resource. Notice, how the independent activity corresponding to the connection serving the dynamic resource spans two processes.

In this situation, the problems in the last two scenarios are compounded. First, the Web server application cannot provide differentiated service to the connections (for static resources) that it handles in the main server process. Moreover, it cannot limit the resource usage of the various requests for CGI documents. Each CGI request is handled at the server by a separate process, which appears to the system as a distinct resource principal. Thus, CGI request processing can get up to $(n-1)/n$ of the CPU resources of the system where n is the total number of server processes in the system (1 main server process and $n-1$ CGI processes). The main server process is left with only $1/n$ of the CPU cycles of the system.

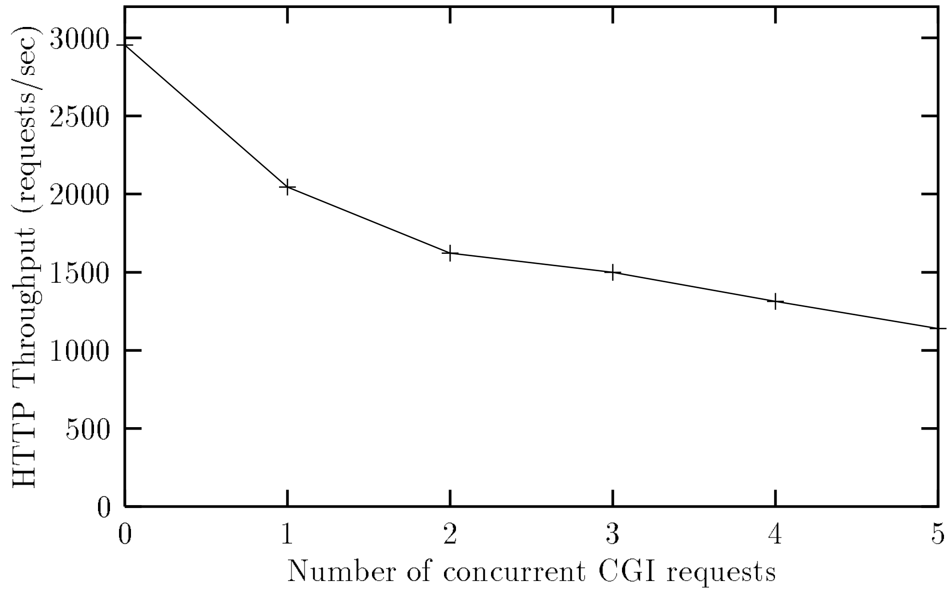

Figure 3.7 Throughput with competing CGI requests.

Figure 3.7 shows the throughput degradation of a Web server in the presence of concurrent requests for CGI documents. In this experiment, the same server system was used as in the previous experiment. The throughput of the Web server (for cached, 1 KB static documents) was measured while increasing the number of concurrent

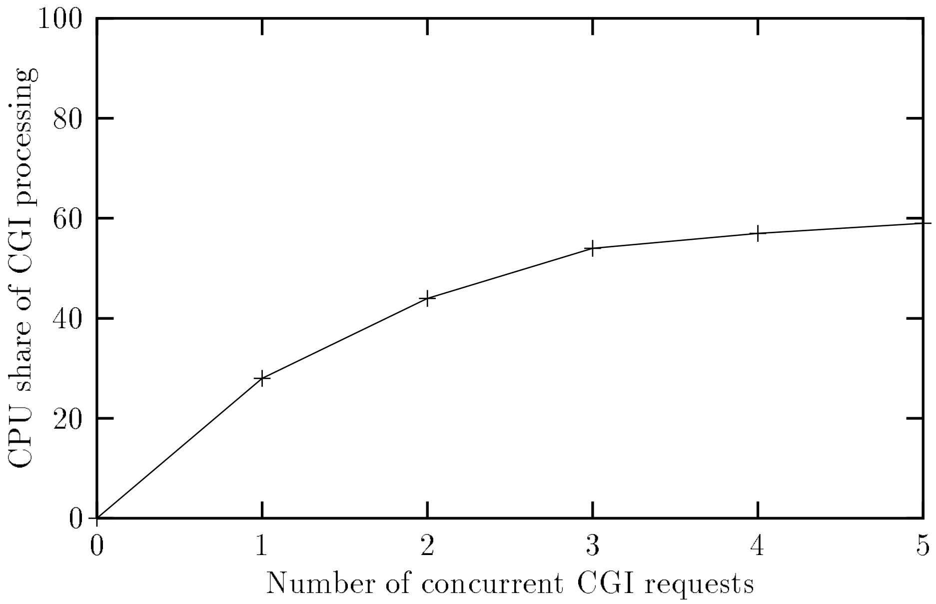

Figure 3.8 CPU share of CGI requests.

requests for a dynamic (CGI) resource. Each CGI request process consumed about 2 seconds of CPU time.

The effective throughput of the server for the static documents falls drastically as the number of concurrent CGI requests is increased. As mentioned above, this happens because the increasing number of concurrent CGI requests causes the Web server to use an increasing number of CGI processes to handle these requests. The server's kernel sees an increasingly larger number of resource principals, and gives a decreasingly smaller CPU share to the main server process handling static requests, which continues to appear to the operating system as a single resource principal.

The CPU share obtained by the CGI processes is shown in Figure 3.8. For example, with only 4 concurrent CGI requests, the main server process gets only 40% of the CPU, and the static-request throughput drops to 44% of its maximum. Notice that the main server process actually gets slightly more of the CPU than does each CGI process. This happens because of misaccounting for network processing.

Excessive CPU usage by CGI processes may interfere with the provision of response time guarantees for certain critical static documents being served by a server system. In addition, the main memory space consumed by CGI processes may decrease the available size of the file-cache for static documents. This may also cause an increase in the response time of requests for static documents, as the corresponding files now need to be fetched from disk more often. All of these effects may prevent the enforcement of business-critical resource usage policies by the server. Also, a large number of concurrent CGI requests may be used by a malicious user as part of a denial of service attack on the server.

3.2.4 Summary

To summarize the discussion, the equivalence between "process" and "resource principal" is appropriate for classical programs, because they seldom need to control the rate of progress of distinct activities within themselves. Even when this control is important, it can be adequately accomplished using user-level mechanisms, because kernel resource utilization forms an insignificant part of the total resource usage of the process. This is not true for a server application, which is kernel-intensive, and needs to ensure the controlled, potentially prioritized progress of its various independent activities.

3.3 Thread-centric operating systems

In some operating systems, e.g., Sun Microsystem's Solaris Operating Environment [149], threads assume some of the role of a resource principal. In these systems, CPU usage is charged to individual threads rather than to their parent processes. This allows threads to be scheduled either independently, or based on the combined CPU usage of the parent process' threads. The process is still the resource principal for the allocation of memory and other kernel resources, such as sockets and protocol buffers.

I would like to stress that it is not sufficient to simply treat threads as the resource principals. The processing for a particular connection (activity) may involve multiple threads, not always in the same protection domain (process). As described earlier, a request for a dynamic HTTP document is often handled by a thread in the main Web server process, in co-operation with another thread in a separate CGI process. Alternatively, a single thread may be multiplexed between several connections, as is the case in a event-driven Web server. Treating threads as resource principals in both these cases leads to problems similar to those described in the previous section.

3.4 Lack of integration of network processing with global resource management

As mentioned repeatedly in this chapter, traditional systems provide little control over the system resources consumed by network-intensive applications. The manner in which current operating systems perform network processing, as was described in Section 2.2, leads to several important problems. Specifically, four aspects of current network subsystems are problematic.

1. **Inaccurate accounting**. System resources consumed in processing network traffic are either misaccounted to the wrong process or not accounted for at all. This leads to inaccurate accounting, and therefore inaccurate scheduling.

2. **Eager receiver processing**. Much of the network processing is performed in response to the arrival of packets, with highest priority given to the capture and storage of packets in main memory; second highest priority is given to the protocol processing of packets; and, lowest priority is given to the applications that consume the messages. A packet arrival always interrupts a presently executing application; this happens regardless of the state or the scheduling priority of the receiving application of the incoming packet.

3. **Lack of effective load shedding**. Packet dropping as a means to resolve receiver overload happens mostly at the level of the socket queues. At that stage, significant host CPU resources have already been invested in the dropped packet. This and the last aspect of current network subsystems can directly lead to starvation or livelock [137].

4. **Lack of traffic separation**. Because of the shared nature of the IP queue, the single shared buffer pool, and the FIFO algorithm for processing packets, excess incoming traffic destined for one application (socket) can lead to delay and loss of packets destined for another application (socket). In effect, there can be significant interference between independent network streams.

As we saw in Section 3.2, these issues are very important for Internet server applications because these applications are particularly network-intensive. Moreover, some server applications (e.g., WWW servers) are often deployed in "unfriendly" environments where the server has little control over the request load it may be subject to, and can easily be overloaded by the practically unlimited client population.

The adverse effects of the inaccurate accounting of the resources consumed in network processing have already been briefly discussed in Section 3.2. Two quantitative examples of the poor overload performance of servers that arise because of the issues described above are presented below:

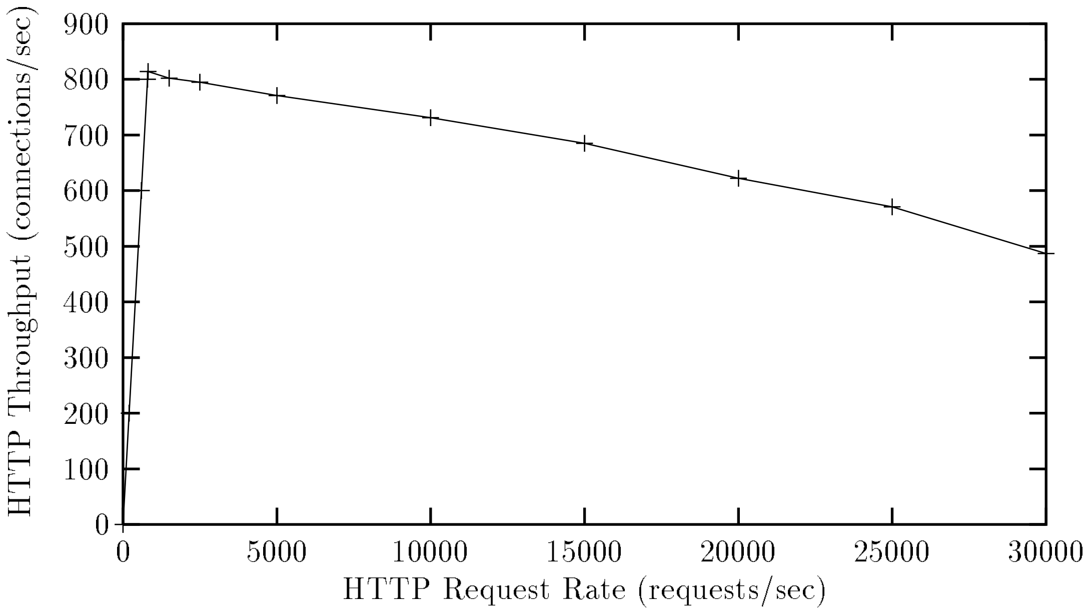

Figure 3.9 Overload performance of the Apache server.

This experiment attempts to quantify the variation in the throughput of the Apache [8] Web server under overload. As before, these measurements were performed on a 500Mhz Alpha (21164) processor based machine running Digital UNIX v4.0D. In this experiment, all requests were for the same 1KB file which was always present in the main memory file cache of the server.

Figure 3.9 shows the results of this experiment. As can be seen, the server saturates at about 800 requests/second. Subsequently, as the load on the server increases, the server's throughput suffers a significant degradation. At 30,000 requests/second, the throughput of the server is only about 59% of its maximum throughput.

This decline in throughput occurs because of the following: When the server is saturated, it cannot keep up with the arrival rate of connection requests, and the accept queue of its listen socket grows in size. Unfortunately, this feedback is used only in a limited manner by the protocol code to shed load. Excess TCP connection establishment requests (SYN packets) are discarded at the full SYN-RCVD queue of the listen socket, i.e., after significant resources have already been expended in processing these packets. Moreover, as the kernel gives strictly higher-priority to the processing of network traffic in preference to any user-level processing, this "useless" processing of SYN packets happens at the cost of the some user-level processing, and thus decreases the rate at which the server's established connections are handled.

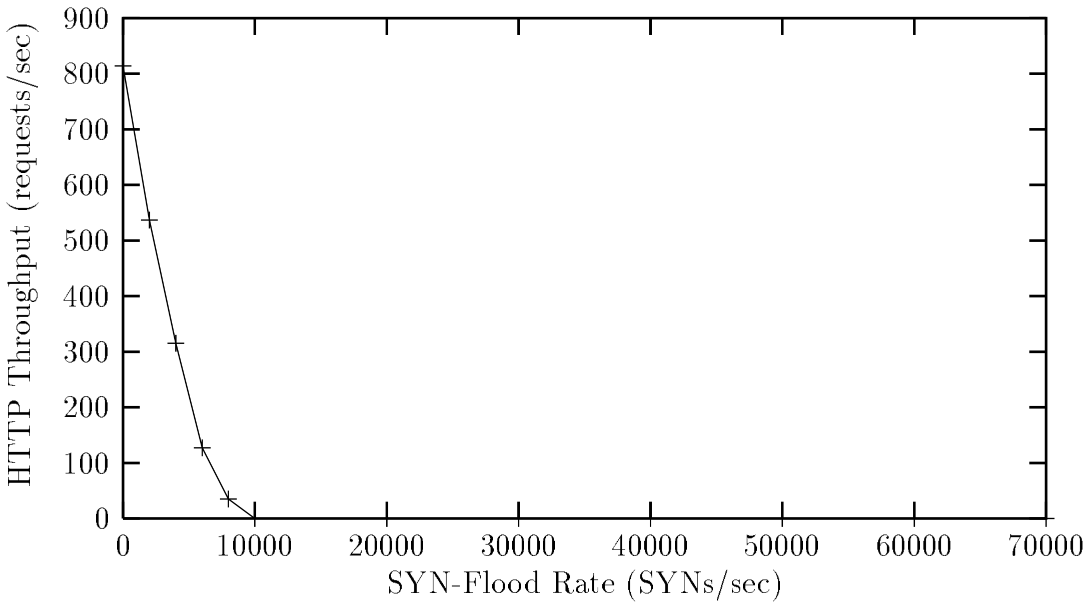

Figure 3.10 Performance of the Apache server under a denial of service attack.

Figure 3.10 shows the throughput of the Apache server under a SYN-flood based denial of service attack [143, 18, 42] from a malicious client. In this experiment, a client sent bogus SYN packets at a very high rate to the Web server, but subsequently did not participate in the TCP three-way handshake for any SYN packet. Thus, it never established any connections with the server and its SYN packets ultimately timed-out of the SYN-RCVD queue of the server's listen socket.

The throughput of the server for concurrent requests from a set of well-behaved clients was measured. These clients made requests as fast as the server could handle them; thus they attempted to saturate the server but not to overload it. Again, all requests from the well-behaved hosts were for the same cached 1KB file.

As we can see, the throughput of the system degrades severely under the denial of service attack. At about 10,000 SYNs/second, the throughput of the server is effectively zero. This drastic throughput drop occurs because the excessive SYN packets occupy an increasingly large number of slots in the SYN-RCVD queue of the Web server's listen socket, and at higher SYN rates, also in the IP queue. This causes an increasingly larger number of packets from the well-behaved clients to get dropped. Because of TCP's exponential backoff in the face of packet losses, the effective throughput of the Web server falls significantly.

This throughput degradation is an extreme example of interference between a ill-behaved and non-flow-controlled stream of SYN packets (those from the malicious client) and the well-behaved streams from the other clients. The effect of this interference is catastrophic; it literally brings the server to its knees and prevents it from performing any useful work. In other, less drastic situations, the throughput of a high-priority stream may be adversely affected by a misbehaved lower-priority stream.

3.5 The need for explicit resource principals

The fundamental concern of this dissertation is to allow an application to explicitly allocate resources among the independent activities that it manages. This is not possible if the operating system's notion of an independent activity differs from that of the application, or if the system does not correctly and completely account for resource consumption at all levels of the system. The need to do fine-grained resource management is critical for a server to support prioritized handling of its clients, to ensure stable overload behavior, and to prevent misbehaving or malicious clients from monopolizing all resources of the server system.

As we have seen, with a single-process server, traditional operating systems see only one resource principal – the process. Moreover, this process-centered resource principal does not extend into the kernel. This prevents the application from controlling consumption of kernel CPU time (and other kernel resources) by various network connections *within* this resource principal. The application cannot control the order in which the kernel delivers its network events; nor, in most systems, can it control whether it receives network events before other processes do.

It is this lack of a carefully defined concept of resource principal, independent from other abstractions such as process or thread, that precludes the application control that is desired. In the next few chapters, I will describe a new resource management model that is based on the *resource container*, an explicit operating system abstraction that encompasses an independent activity. These chapters will also show how kernel processing can be performed in the context of the correct resource principal, and thus be correctly accounted for and controlled.

3.6 The need for hierarchical principals

Often an application needs to restrict the total resource consumption of a subsystem without constraining (or even understanding) how the subsystem allocates and schedules resources among its various independent sub-activities. As a motivating example, consider a Rent-A-Server host [164], which provides an environment for supporting multiple "guest" Web servers, each of which provides Web service for a different organization. Such a service may be useful for a variety of reasons, which include, for instance, the need to temporarily increase service capacity by a large factor, in order to handle seasonal bursts of traffic (as for the Internal Revenue Service's Web server around April 15).

The administrators of a Rent-A-Server host may wish to limit the total resource usage of each guest server to a fixed fraction of the total resources of the host. Also, each guest server may itself want to control the relative resource consumption of its various client connections. Alternatively, the guest server may wish to treat connection aggregates, e.g., all connections from a particular set of clients, as resource principals. In general, the Rent-A-Server administrator may not be aware of the exact resource management policies that each guest server wishes to implement.

To support such scenarios correctly, there is a need for the operating system to support an hierarchy of resource principals. Specifically, we need that a parent resource principal be able to constrain and schedule the total resource consumption of its child sub-principals. Given this support, a Rent-A-Server system's administrator can specify a parent container for each guest server. Each individual server can then create child containers in an arbitrary fashion to distribute the resources allocated to its parent container between its various independent activities, according to its specific resource allocation policies.

At a more fundamental level, the notion of *independence* that defines the extent of an "independent activity" (and thus a resource principal) is relative to the context being considered. From an operating system's standpoint, individual users, or projects, are independent and should be treated as the resource principals of the system. From each user's standpoint, however, individual applications or sets of applications are independent, and should constitute the principals for resources allocated to the user. An application might itself be performing several independent activities, leading to further levels of the consideration for defining independent activities.

Thus, it is necessary for an operating system to support an hierarchy of resource principals in order to control fairness or priority between its principals at various levels of "independence", and to enforce a resource usage containment relationship between parent principals and their children. In current operating systems, not only is the notion of protection domain intrinsically tied to a that of a resource principal (and thus an independent activity), there is no support for multiple notions of "independence". The resource container abstraction, to be described in the next chapter, provides such support.

3.7 API issues

I will now describe the shortcomings of some operating system APIs from the perspective of a server application that wishes to exercise fine-grained control over resource consumption. Again, the focus is on UNIX APIs; similar issues arise with other general-purpose operating systems, such as Windows NT.

In the following discussion, we first consider select(), which provides event-driven server applications with a control mechanism. Multi-threaded servers make a more traditional use of operating system control mechanisms; the thread scheduling mechanism itself provides the control mechanism for such servers. Thus, the use of multi-threaded servers does not introduce any significant API issues as far as control mechanisms are concerned.

Subsequently, we examine the socket bind() API, that allows a server application to provide its services at a well-known port. As we shall see, the current bind() API is problematic for both event-driven as well as multi-threaded servers.

3.7.1 Shortcomings of select()

A UNIX program uses the select() system call to wait for asynchronous events on socket descriptors. A process can indicate interest in three types of events on a descriptor: events that make the descriptor *readable*, those that make it *writable*, and *exception* events. This information is passed to the kernel using three bitmaps. In each bitmap the kth bit indicates interest in events of that type for the kth descriptor. These bitmaps are value-result parameters, and the returned bitmaps indicate the sets of ready descriptors.

In the kernel, select() is implemented as follows: During a call to select(), the kernel starts by checking, for each descriptor in the input bitmaps, whether that descriptor is available for I/O. If none are available, select() blocks. Later, when a protocol processing module's state changes to make a descriptor readable or writable, the blocked process is woken up. Then, select() creates its output bitmaps and returns to user space. Wright and Stevens provide a detailed discussion of the classical BSD implementation of select() [171].

Several aspects of select() interact poorly with its use by server applications [13]. First, each call to select() has cost proportional to the total number of descriptors that the application is interested in. This is problematic, because as mentioned in Chapter 2, busy WWW servers often manage a large number of simultaneous connections [15]. The poor scalability of select() with the total number of input descriptors is rooted in the semantics of its API; there is an inherent need to scan each descriptor in the input bitmaps during each call to select(), leading to a per-call overhead that is intrinsically proportional to the total number of descriptors being selected. In previous work [15], I have shown how the traditional implementation of select() can be optimized. The algorithmic complexity of this system call, however, cannot be improved without changing the semantics of its API.

From the standpoint of this dissertation, select()'s most severe drawback is that it does not support any notion of priority for a socket. There is no way for an application to specify that events on a particular socket should be delivered to the application without delaying to process additional events on other, lower-priority, sockets. This shortcoming, which is exacerbated by the poorly scaling nature of select(), can lead to significant priority inversion in a busy server.

As a concrete example of the problematic nature of select(), consider a server application that manages two sets of connections with different priorities. This situation is similar to the experiment whose results are shown in Figure 3.5. Assume that the server is handling five connections, one of which belongs to the higher-priority set. Now, suppose that a number of packets for these connections arrive in quick succession. First, protocol processing for these packets happens in arrival order; the protocol code in the kernel has no notion of connection priority. As described earlier in this chapter, this leads to some priority inversion in the delivery of events to the server application.

Furthermore, the select() mechanism also contributes to priority inversion in request processing. After protocol processing, the server application, which is blocked inside select(), is woken up. Unfortunately, however, select() does not return to user space until after it has scanned the status of the sockets associated with *all* connections the server is managing (or at least all sockets associated with ready connections, if we consider my improved select() implementation [15]). Moreover, before the return to user space, select() also creates its output bitmaps and copies them out to the application's address space; this has cost proportional to the number of descriptors being selected. Both of these factors cause the latency of event delivery to be proportional to the total number of connections being handled by a server. Thus the delivery of higher-priority events is delayed by the presence of concurrent low-priority connections. This phenomenon is responsible for at least some of the increase in response time in Figure 3.5.

Measurements that I have performed as part of my previous work on improving the implementation of select() [15] indicate that busy servers spend up to 60% of their execution time inside select(). An example profile of a busy real-world select() based proxy server obtained by using Digital's Continuous Profiling Infrastructure (DCPI) [7] is shown in Table 3.2. These measurements were performed on Compaq Computer Corporation's Palo Alto proxy server. The proxy system was a 500Mhz AlphaStation equipped with 512MB of RAM and running Digital UNIX 4.0B. The proxy software was Network Appliance's *NetCache* [121] product.

For this system, 35.27% of the CPU cycles are consumed by the various kernel functions that implement select(). An additional 12.64% of the system's CPU is consumed by the user-level commSelect() function, which collates the data returned by select(). These numbers should give the reader a sense of the overhead of select(). Most of this overhead occurs in scanning linear data structures while delivering an event to the application, and these scans directly lead to priority inversion in event delivery.

In summary, current event mechanisms do not support the prioritized delivery of events. Therefore, there is a need to develop a new event mechanism and API which allows the prioritization of event sources. In Chapter 4, we will see how the priorities of event sources are specified in the new resource management model of this dissertation, and describe a new priority-aware event API.

CPU %	Procedure	Mode
10.77%	all idle time	
89.23%	all non-idle time	
35.27%	all select functions	kernel
13.51%	selscan()	kernel
12.56%	soo_select()	kernel
7.48%	undo_scan()	kernel
1.64%	select()	kernel
12.64%	commSelect()	user
1.74%	all TCP functions	kernel
1.49%	malloc-related #1	user
1.39%	malloc-related #2	user
1.09%	mutex_unblock()	user
1.03%	read_io_port()	kernel
0.95%	bcopy()	kernel
0.94%	memGrep()	user

Profile on 1998-04-16 from 10:00 to 11:00 PDT
mean load: 54 requests/sec.
peak load: ca. 98 requests/sec

Table 3.2 Profile of the Digital UNIX kernel on Compaq's Palo Alto proxy server

3.7.2 Limitations of the bind() API

The bind() system call is used by a server application to assign a name to a service endpoint, i.e., a listen socket. Clients can send connection requests to the server by directing their requests to this named socket. The bind() call supports a number of name spaces; this allows it to work with a variety of network protocols and address families. For TCP/IP listen sockets, bind() names are specified as sockaddr_in structures, each of which contains a <*local IP address, local port*> tuple.

Recall from Section 2.2 the sequence of steps by which a Web server accepts a new client connection. An incoming connection establishment request (TCP SYN packet) causes the server's TCP to queue a socket denoting the incomplete connection in the SYN-RCVD queue of the listen socket. When the three-way TCP handshake is completed for this connection, the server TCP moves this socket from the listen socket's SYN-RCVD queue to its accept queue, where its waits for the server application to dequeue it via the accept() system call.

This operation is problematic because it does not distinguish between connection requests from different clients. Requests from all clients arrive at the same listen socket, and are queued in the single SYN-RCVD queue of the socket. Also, all established connections await acceptance by the server application in the FIFO accept queue of the listen socket. This can lead to priority inversion: A high-priority connection request may be dropped if all slots in the SYN-RCVD queue of the server's listen socket are occupied, perhaps by lower-priority connection requests. Also, higher-priority connections may have to wait behind lower-priority requests in the accept queue. This priority inversion precludes a server application from effectively performing prioritized handling of requests.

Furthermore, the processing performed for all connections prior to acceptance by the server application is charged to the same resource principal, i.e., the principal owning the server's listen socket. Thus, the resource consumption of a particular client connection during the connection establishment phase cannot be independently controlled. The priority of the principal owning the listen socket can be changed, but this affects the priority of processing connection requests from *all* clients. Because of this, a server cannot protect itself from a SYN-flood based denial of service attack from a malicious client [143, 18, 42]. In this attack, SYN packets sent by a malicious client occupy all slots in the SYN-RCVD queue of the server's listen socket. Almost all legitimate requests are dropped, leading to the extreme throughput degradation that we saw in Figure 3.10.

The fundamental cause of this problem lies in the nature of the bind() name space for TCP/IP listen sockets. An application cannot specify client addresses in a socket name, and therefore cannot create a different listen socket for a specific client or set of clients. This prevents the assignment of connection requests from different clients to separate sockets, each with its own set of queues, and whose resource consumption could be charged to distinct resource principals with independently controllable priorities. This also prevents the isolation of a misbehaving client, such as one engaged in a SYN-flood based denial of service attack, which could otherwise be handled by assigning all requests from it to a socket owned by a very low priority resource principal.

This shortcoming of the current bind() name space for listen sockets argues for a new class of socket names that allow for clients to be associated with listen sockets. Such a name space is presented in the next chapter.

Chapter 4

A new model for resource management

This chapter describes a new model for fine-grained resource management in general-purpose operating systems. This model is based on a new abstraction, called a *resource container* [14], which explicitly identifies the operating system's resource principal.

The discussion in this chapter is structured as follows: To begin with, a description of the basic nature of a resource container, and the relationship between containers and classical operating system abstractions, such as processes and threads. Following this, the usefulness of resource containers in enabling flexible and fine-grained resource management for a variety of resource types is illustrated. Sections 4.4 through 4.8 describe the operation of a container based system in some detail. Finally, in Section 4.10, some new APIs are presented that are necessary for server applications to make effective use of the fine-grained resource management capabilities of resource containers.

Most of the discussion in this chapter is applicable to any general-purpose operating system. In a number of places, however, it has been necessary to introduce system-specific details in order to make the discussion meaningful. In these cases, we assume the context of a UNIX-like operating system. While differing in specific low-level details, these sub-discussions are still valid in their essence for other general-purpose operating systems, such as Windows NT.

4.1 Resource containers

A resource container is an abstract operating system entity that logically contains all the system resources being used by an application to perform a particular independent activity. For a given HTTP connection managed by a Web server, for example, these resources include CPU time devoted to the connection, and kernel objects such as sockets, protocol control blocks, and network buffers that are used by the connection.

Containers have attributes; these are used to provide scheduling parameters, resource limits, and network QoS values. The exact nature of these attributes depends on the specific set of resource management policies being used in the concerned system; the container mechanism can work with a broad range of such policies. A practical implementation requires an access control model for containers and their attributes. This is discussed in detail later in Section 4.7.

The kernel carefully accounts for the system resources, such as CPU time and memory, consumed by a resource container. The system's schedulers can access this usage information and use it to control how they schedule resource requests made by threads associated with the container. The application process can also access this usage information, and might use it, for example, to adjust the container's priority. Scheduling of resources in a container based system is discussed in Section 4.3.1.

As discussed in Chapter 3, current operating systems implicitly treat processes as the resource principals, while ignoring many of the kernel resources they consume. By introducing an explicit abstraction for resource containers, a clear distinction is made between protection domains and resource principals. Resource containers can have fairly arbitrary relationships with protection domains. For example, a container can be contained entirely, along with other resource containers, within one process. Alternatively, a container can span across several protection domains, encompassing all or some parts of their computation. The new resource management model also provides for fuller accounting of kernel resource consumption. This provides the flexibility necessary for servers to handle complex resource management problems.

4.2 Containers, processes and threads

In classical systems, there is a fixed association between threads* (which are the executable entities of the system) and resource principals (which are either the threads themselves, or the processes containing the threads). The resource consumption of a thread is charged to the associated resource principal, and this information is used by the system when scheduling threads.

With resource containers, the binding between a thread and a resource principal is dynamic, and under the explicit control of the application; this is called the thread's *resource binding*. The kernel charges the thread's resource consumption to this con-

*In this chapter the term "thread" is used to refer to a kernel-level thread, unless otherwise qualified.

tainer. Multiple threads, perhaps from multiple processes, may simultaneously have their resource bindings set to a given container.

A thread starts with a default resource container binding, which is inherited from its creator. The application can rebind the thread to another container as the need arises. For example, a thread that is time-multiplexed between several connections changes its resource binding as it switches from handling one connection to another, to ensure correct accounting of resource consumption.

In the new resource model, the system scheduler makes CPU resources directly available to resource containers instead of threads. Of course, since threads are in fact the executable entities of the system, a CPU allocation to a resource container must ultimately be mapped onto a runnable thread that is associated with the container. This function is performed by a second-level, potentially container-specific, scheduler. It is this second-level scheduler's responsibility to keep track of the runnable threads that are available in various protection domains to the container, and to schedule these threads based on the specific requirements of the independent activity corresponding to this container. This issue is discussed in detail in Chapter 6.

To support this two-level scheduling, the new resource model defines a binding, called a *scheduler binding*, between each resource container and the set of threads that are available to it. A newly created container has a scheduler binding set that includes just the thread that created it. Subsequently, threads can be added to and deleted from this scheduler binding set by using explicit system calls. A thread that handles multiple containers, such as the single thread of an event-driven server, is present in the scheduler binding sets of all the containers that it handles.

4.3 Controlling resource usage with resource containers

The basic purpose of resource containers is to allow an application to control resource consumption at a fine level of granularity. These resources include CPU cycles, memory pages, disk bandwidth and other kernel resources such as sockets, socket buffers, protocol control blocks and file-cache pages. This section describes the use of resource containers to control usage of each of these resource types.

4.3.1 CPU cycles

CPU schedulers make their decisions using information about both the desired allocation of CPU time, and potentially the recent history of actual usage. For example, the traditional UNIX scheduler uses numeric process priorities (which indicate desired behavior) modified by time-decayed measures of recent CPU usage; lottery scheduling [169, 168] uses lottery tickets, which are abstract measures of CPU resources, to represent the allocations. In systems that support threads, the allocation for a thread may be with respect only to the other threads of the same process ("process contention scope"), or it may be with respect to all of the threads in the system ("system contention scope").

As described before, resource containers allow an application to associate scheduling information with an activity, rather than with a thread or process. This allows the system's scheduler to provide resources directly to an activity, no matter how it might be mapped onto threads.

The container mechanism supports a large variety of scheduling policies, including priorities, guaranteed CPU shares, or CPU usage limits. The allocation and accounting attributes appropriate to the scheduling model are associated with each resource container in the system. Chapter 6 describes in detail the integration and use of resource containers with classical scheduling frameworks.

4.3.2 Physical memory

Like CPU cycles, the use of physical memory by activities can also be conveniently controlled by the resource container mechanism. The system can charge the memory usage of an application to the appropriate resource containers instead of attributing it to the processes being used by the application. The memory management subsystem can then try to ensure fairness between the various resource containers, or handle them in a prioritized fashion (if so desired).

Unfortunately, policies that are deployed in current general-purpose operating systems for controlling the consumption of resources other than CPU cycles are able to do so only in a very coarse manner. In particular, memory management policies are able to ensure only a limited degree of fairness to their principals (processes). This is reviewed in more detail in the discussion below.

Physical memory is used by an application process in two ways, i.e., to back virtual pages of the process' user-level address space, and to provide cache pages for disk files being used by the process. In current general-purpose operating systems, e.g., UNIX, physical memory usage for the user-level pages of processes is controlled by a global, two-level, demand-paging and replacement algorithm. Page frames are allocated to applications as needed[†] until the total number of free page frames reaches a low-water mark. Subsequently, a global page-replacement algorithm selects allocated page frames, which have not been used recently, from processes and evicts them. This page-replacement method is usually a variant of the clock algorithm [43, 56], which approximates global LRU. If a process has no pages left in memory, it is temporarily swapped out.

The operation of allocation schemes for cache pages in the kernel's filesystem cache is similar. Two algorithms are commonly used. In the first method, a fixed percentage of the system's memory is set aside for the filesystem cache. This memory pool is managed using a LRU page replacement algorithm. In the other scheme, cache pages come from the same physical memory pool as page frames that back the user-level address spaces of processes. Filesystem cache pages compete with the user-level pages of processes in the operation of the page-replacement algorithm.

The "global" nature of classical paging and file-caching schemes is often problematic. For example, an application that has large memory (or file-cache) requirements can adversely affect the performance of other processes by using up an unfair fraction of the system's memory resources. The global page-replacement algorithm will ultimately penalize this application by replacing more pages from it than from other applications. However, this adjustment usually happens at a time-scale that is too large to completely isolate other processes from this application. Similarly, an application that is aware of its memory access pattern cannot use this information to implement a specialized paging mechanism (e.g., with some prefetching) that would result in better performance; all applications are forced to use pure demand-paging, which performs reasonably for many applications but is certainly not optimal in all application scenarios.

Several researchers have addressed the need for application-specific page replacement [103, 144, 75, 90, 59, 165] and file-caching [28] algorithms. A number of policies have been developed to provide better isolation between applications, and allow for prioritization of memory consumption. However, in all this work, the unit of what constitutes an independent application activity is still a single process, or as in the case of the work of Verghese et al. [165], a group of processes. From the discussion in Chapter 3, we know that an independent activity in a system does not always correspond to a process (or even a group of processes). This lack of equivalence between

[†]The need of a page is detected by a page fault to a virtual address.

the system's and the server application's notion of the unit of memory-management can lead to various problems related to unfairness and priority inversion.

For example, a request for a large (say 50MB) file can clobber the entire filesystem cache of a Web server, and thus adversely effect the performance of other, less resource-intensive requests being serviced by the same process. Similarly, requests for dynamic HTTP resources, which typically cause CGI processes to be spawned, can consume a significant fraction of the system's memory, and leave insufficient memory for efficiently serving other requests being handled by the server.

With resource containers, the boundaries of independent activities can be correctly specified for the purposes of memory management. The various policies developed by the application-specific paging and file-caching work cited above may be used to provide isolation between containers by simply using containers as the policies' principals in place of processes.

Given such support, an activity with a small memory footprint could, for example, pre-fetch all its required pages in all the protection domains that it uses and pin them down (i.e., prevent them from being paged out for the duration of the activity), and therefore isolate itself from the behavior of more memory-intensive applications. Similarly, a larger activity (which also potentially spans multiple protection domains) may obtain a performance benefit from pre-fetching some of its pages (and pinning down the most frequently-used of these) and demand-paging the rest. Each activity could use a different page replacement algorithm and thus improve its performance if its memory access pattern does not match simple LRU page-replacement. The specific memory management policy in use is responsible for ensuring fairness between the various activities by controlling, for example, the total number of pages that an activity's container can pin down.

4.3.3 Disk bandwidth

Resource containers can also be used to control the disk bandwidth used by the various independent activities of a system. Policies that ensure fairness of disk bandwidth allocation between processes can be modified to consider containers as their resource principals. This provides applications, such as servers, with complete control over the handling of disk requests for the various connections being served by them.

Traditional operating systems schedule disk requests based only on the current head position of the disk (using some variant of the classical C-SCAN algorithm [156, 157]). Outstanding disk requests are sorted by block number and serviced in order as the disk head moves across the disk, from the first cylinder to the last. When the head reaches the request closest to the end of the disk, it goes back to the first cylinder and starts again. This minimizes seek time while preventing starvation.

Since the blocks of a file are often laid our contiguously on disk, the writing of a large file, such as a core dump, can temporarily monopolize all the bandwidth of the disk. This can delay the disk requests of other, higher priority activities for a substantial amount of time.

There has been some recent work towards developing disk scheduling mechanisms that allow the prioritization of disk requests [165, 168]. These mechanisms try to ensure some degree of fairness between processes while still keeping seek overhead low. For example, an algorithm proposed by Verghese et al. [165]) tracks the decayed disk bandwidth usage of each principal. Normally, disk requests are serviced as in C-SCAN [156, 157]. If the disk bandwidth usage of some principal exceeds the average disk bandwidth usage of all principals by a specified factor, then the disk I/O requests from this principal may be delayed by the disk scheduler until a more equitable disk bandwidth distribution results.

These algorithms, however, still consider only a single process or a group of processes as the resource principal. Thus, they suffer from the same fundamental problems as the efforts to manage CPU and memory resources, as described earlier.

With resource containers, the work cited above can be conveniently used to control disk bandwidth usage at an appropriately fine level of granularity. The operation of the specific policy remains unchanged; the role of a process, or a group of processes, is simple performed by a resource container. In Verghese's algorithm, for example, the system can charge a disk I/O operation to the container to which the thread performing the I/O is resource-bound. The system can also track the decayed disk bandwidth usage of each resource container, and aim to limit the factor by which the total disk bandwidth usage of any container exceeds the average bandwidth usage of all containers.

4.3.4 Other kernel resources

Besides CPU cycles, physical memory and disk bandwidth, the performance of server applications also depends critically on correct allocation of *soft* kernel resources such as sockets, socket buffers and protocol control blocks between independent activities.

In conjunction with appropriate policies, resource containers can be used to correctly account for, and control the allocation of such kernel resources.

In current operating systems, the kernel maintains global pools of physical memory blocks for each of such resources. Resources are allocated to applications until they are exhausted. When this happens, resource allocation requests either fail, or the application making the request is blocked until the request can be fulfilled.

As in the case of physical memory, global allocation of such soft resources is problematic for server systems. A low-priority client connection can hog large amounts of kernel resources in a server and thus adversely affect the performance of higher-priority connections. For example, consider a TCP connection between a Web server and client system. This connection needs a certain size of socket buffer space at the server for being able to achieve the maximum transfer bandwidth available on the network path between the server and the client. If this space is not available, only a fraction of this available bandwidth may be realized. The excessive use of socket buffer space by low priority connections may therefore limit the throughput of a higher-priority connection.

Resource containers enable the controlled allocation of kernel resources on a per-connection basis. The kernel resource consumption of the thread handling a particular connection is charged to the correct resource container. The various soft resource allocators may take into account the resource management parameters of the resource container that the calling thread is resource-bound to, to make decisions about whether to fulfill or to delay allocation requests.

As for other resources, policies are needed to control how allocation decisions are made for kernel resources. This has been the subject of recent work [146]. My prototype implementation of resource containers, which will be described in detail in Chapter 8, allows any such policy to be easily deployed in the system. The detailed study of policy issues related to soft resources is, however, beyond the scope of this dissertation.

4.4 The resource container hierarchy

The resource containers of a system form a hierarchy. The root container of this hierarchy tree corresponds to all the resources of the system. Its children containers correspond to various independent activities being performed by the system. Each of these containers can potentially have a container sub-hierarchy rooted at it; containers in these sub-hierarchies correspond to the various independent sub-activities of the activity corresponding to the root container of each sub-hierarchy.

The resource usage of a child container is constrained by the scheduling parameters of its parent container. For example, if a parent container is guaranteed at least 70% of the system's resources, then it and its child containers are collectively guaranteed 70% of the system's resources. The hierarchical structure of resource containers makes it easy, for example, to implement fixed-share CPU scheduling classes, and to enforce a rich set of priority policies.

The motivation behind structuring containers in a hierarchical fashion, as was discussed in Section 3.6, is to allow an activity to limit the total resource usage of a sub-activity, while still allowing the sub-activity to partition resources awarded to it amongst its own sub-activities in accordance to its specific resource distribution policies. By allowing containers to be structured in a hierarchical fashion, multiple notions of "independence" that define the boundaries of independent activities can be supported.

The exact rules concerning the structure of the resource container hierarchy, e.g., what containers can have child containers, and what limits are implied on the scheduling parameters of a child container given the resource consumption constraints on its parent, depend on the exact scheduling framework being used. This will be discussed further in Chapter 6 in the context of CPU scheduling.

4.5 Naming of resource containers

Containers are visible to an application process as file descriptors. An operation that creates a new container returns a new file descriptor that provides access to the new container. Similarly, an operation that causes an application process to gain access to an already existing container, e.g., by IPC with another application process that created the container (see below), also returns a file descriptor. These descriptors can be used as parameters to the various kernel-supported operations on resource containers.

While a new name space could have been defined for allowing processes to name resource containers, the use of file descriptors allows us to use several existing system calls to manipulate resource containers. For example, the UNIX `sendmsg()` system call can be used to transfer file descriptors, and hence resource containers, between protection domains.

One disadvantage of using file descriptors to name containers that a number of operations on descriptors, e.g., select(), have no sensible semantics for container descriptors. Practically, this has an implication that in an implementation of resource containers in UNIX, the code for a large number of existing system calls needs to be revisited and "special-cased" to return an appropriate error if a container descriptor is incorrectly passed in as an argument to the system call.

4.6 Operations on resource containers

The resource container mechanism includes these operations on containers:

Creating a new container: A process can create a new resource container at any time (and may have multiple containers available for its use). A default resource container is created for a new process as part of a fork(), and the first thread of the new process is bound to this container. Since containers are visible to the application as file descriptors, they are inherited by a new process after a fork().

Set a container's parent: A process can change a container's parent container (or set it to "no parent"). In general, this operation may cause a change in the share of system resources that the container is entitled to, as limited by the resource share of its new parent. For example, a container guaranteed 30% of the CPU cycles of the system may suffer a decrease in its net CPU entitlement if its new parent has a residual capacity (after prior allocation by its existing children) that is less than this 30% value. Likewise, a container whose parent container is set to "no-parent" is not part of the system's scheduling hierarchy, and thus gets no resources. If later, it is made part of the hierarchy at any level, it will again begin to get an appropriate share of system resources.

Container release: Processes release their references to containers using close(); once there are no such descriptors, and no threads with resource bindings, to the container, it is destroyed. If the parent P of a container C is destroyed, C's parent is set to "no parent."

Sharing containers between processes: Resource containers can be passed between processes, analogous to the transfer of descriptors between UNIX processes (the sending process retains access to the container). When a process receives a reference to a resource container, it can use this container as a resource context for its own threads. This allows an application to move or share a computation between multiple protection domains, regardless of the container inheritance sequence.

Container attributes: An application can set and read the attributes of a container. Attributes include scheduling parameters, memory allocation limits, and network QoS values.

Container usage information: An application can obtain the resource usage information charged to a particular container. This allows a thread that serves multiple containers to timeshare its execution between these containers based on its particular internal scheduling policy.

The following operations control the relationship between containers, threads, sockets, and files:

Resource-binding a thread to a container: A process can set the resource binding of a thread to a container at any time. Subsequent resource usage by the thread is charged to this resource container. A process can also obtain the current resource binding of a thread.

Scheduler-binding a thread to a container: An application can add a thread to (and delete one from) the scheduler binding set of a resource container at any time. Runnable threads that are in the scheduler binding set of a container are candidates for execution when the scheduler awards the container with CPU quanta.

Binding a socket or file to a container: A process can bind the descriptor for a socket or file to a container; subsequent kernel resource consumption on behalf of this descriptor is charged to the container. A descriptor may be bound to at most one container, but many descriptors may be bound to one container.

The prototypes of system calls that expose these facilities to applications are described in Table 4.1. The implementation of these system calls and associated kernel functions is described in Section 8.3. Appendix A contains detailed man pages for these system calls.

prototype	description
int rc_create(void)	create a container
int set_rc_parent(int rc1, int rc2)	set rc1's parent to rc2
int close(int rcd)	regular close() system call –release a container
int sendmsg(int rc, struct msghdr *msg, int flag)	regular sendmsg() call –transfer a container descriptor
int recvmsg(int rc, struct msghdr *msg, int flag)	regular recvmsg() call –transfer a container descriptor
int get_rc_opt(int rcd, struct rcopt *rco)	get container attributes
int set_rc_opt(int rcd, struct rcopt *rco)	set container attributes
int get_rc_usage(int rcd, struct rcusage *rcu)	get resource usage info. for container
int get_rc(void)	get a descriptor for the current thread's resource binding
int set_res_binding(int rcd)	set current thread's resource binding to container rcd
int add_to_sched_binding(int rcd)	add current thread to container rcd's scheduler binding set
int delete_from_sched_bindin)	delete current thread from container scheduler binding set
int set_fd_rc(int rc1, int rc2)	bind the file or socket rc1 to resource container rc2

Table 4.1 Resource container related system calls.

4.7 Access control of resource containers

The basis of the access control model of resource containers is the capability-like nature of a file descriptor. If an application has a reference to an object in the form of a file descriptor, it can perform operations on it. Otherwise, the application cannot even name the object, much less operate on it. Thus, an application can access only those resource containers that it has references to, in the form of file descriptors.

Each reference to a container also has an access control attribute. The reason for this is that the access control model that we desire is somewhat stricter than the all-or-nothing model provided by vanilla file descriptors. In particular, we would like to distinguish between the following rights associated with containers:

- **User rights**, i.e., the ability to charge resource usage to a particular resource container, and to create subordinate containers which are children of this container in the resource container hierarchy (or change the parent of an existing container to this particular container by using the set_rc_parent() system call). These rights also include the ability to pass a reference to a container to another protection domain

- **Owner rights**, i.e., the ability to change the scheduling parameters and other attributes of a container, to change the parent of a container, in addition to all user rights.

User rights allow normal usage of the resource container mechanism by an independent activity within the overall resource usage limits of the top-level container of the activity. Owner rights are needed to configure the attributes of a resource container. Working together, these two classes of rights allow an activity to limit the total resource usage of a sub-activity.

For example, we can arrange things such that a parent activity configures the scheduling parameters of the top-level container of each of its sub-activities. Each sub-activity itself has only user rights to its top-level container. Thus, each sub-activity can name its top-level container, and use it as the target of a set_rc_parent() operation involving a container that it created. Any sub-activity, however, cannot change the scheduling parameters of its top-level container, and thus cannot affect the total resource usage of its containers. This total resource consumption can only be changed by the parent activity.

This access control system is easily implemented by a protected bit-value associated with each reference to a container. This bit-value is set by the kernel when the reference is created. A creator of a resource container has owner rights to it. When the owner of a container performs an operation that causes the creation of a new references to the container, a particular attribute of this container controls the access control rights of this new reference. If a new reference to a container is created as a result of an operation performed by a thread that only has user rights to the container, then this new reference has only user rights to the container.

The kernel performs an access control check before each operation on a resource container. Operations that use a container reference with insufficient privileges are aborted with an appropriate error message.

4.8 Kernel execution method for I/O processing

Resource containers are effective only if kernel processing on behalf of an application is performed in the resource context of the appropriate container. This kernel activity consists primarily of network processing and disk I/O processing. As discussed in Chapters 2 and 3, most current systems do network processing in the context of a software interrupt, and may fail to charge the costs of this processing to the correct resource principal.

In the new resource management model, this problem is addressed by using an execution method that attributes all kernel processing to the correct resource container. For the protocol processing of received network packets, this model works by associating arriving packets with the receiving container as early as possible. This allows the kernel to charge the cost of received-packet processing to the correct principal. Similarly, processing involved in the transmission of packets is also charged to the appropriate container, and performed at the priority of the container.

Similarly, in current operating systems, some portions of disk I/O processing are also performed in interrupt context, and are thus misaccounted for and uncontrolled [165]. For correct operation of the new resource management model, like network processing, disk I/O processing also needs to be performed at the priority of the correct resource container, and charged to it.

The CPU resource consumption of disk I/O processing is, however, insignificant in comparison to network processing. The reason for this is the slow speed of disks, as compared to network devices. Measurements that I have performed in the context of a real-world busy caching proxy server (i.e., Compaq's Palo Alto Web proxies) and with commonly-used Web servers in the laboratory indicate that this overhead is less than 5% of the system's total CPU consumption. Thus, any misaccounting of CPU resources consumed during disk I/O is not likely to affect server performance significantly.

Instead, the more critical problem here is the schedule of disk requests generated by the disk I/O processing code, as was discussed in Section 4.3.3. The schedule of disk operations should support some level of fairness or priority to principals that perform disk I/O, while still keeping the cost of disk I/O operations (e.g., the seek overhead) acceptable.

As briefly discussed in Section 4.3.3, recent research has shown how this problem can be addressed [165, 168]. Therefore, in this dissertation, we will focus on mechanisms that perform network I/O processing in the resource context of the correct principal. A new kernel execution method for network processing that is designed to meet this goal will be described in detail in the next chapter.

4.9 Relationship to other resource management mechanisms

Resource containers are in some ways similar to many resource management mechanisms that have been developed in the context of multimedia and real-time operating systems [66, 73, 85, 105, 116]. Resource containers are distinguished from these other mechanism by their generality, and their direct applicability to existing general purpose operating systems. See Chapter 10 for a comprehensive discussion of this related work.

4.10 API issues

Resource containers provide the basis for fine-grained resource management in a server system. The container mechanism by itself does not, however, automatically provide effective fine-grained control over resource consumption to applications. The operating system's system call interface must participate in providing applications with facilities to use the resource management capabilities afforded by resource containers. In Section 3.7, we saw the shortcomings of `select()` and `bind()` in the context of a server application from a resource management standpoint. I will now present some new APIs, as replacement for these two system calls. These APIs allow the application to use the capabilities of the resource container mechanism in enforcing priority policies for event sources, and for connection request processing of requests from various clients.

4.10.1 A new priority-aware event mechanism

To support the notion of priority of event sources, I have developed two new APIs to allow applications to specify interest in event sources and receive event notifications. One tells the kernel what event sources an application is waiting on for events. The other provides event notifications to the application, preserving the application's priority assignments to event sources. The `select()` system call's interface merges both functions; splitting them into separate calls increases flexibility while avoiding the inherent unscalability of select(), and thus preventing priority inversion in the event delivery process.

The key idea behind the design of this new event mechanism is as follows: Once an application becomes interested in events on a descriptor, it is likely to remain interested in this descriptor for a lengthy period. The kernel maintains an INTERESTED set for each thread that persists across many system calls. The INTERESTED set consists of the descriptors for which the thread is interested in events.

The first API allows the application to inform the kernel when this period begins and ends, rather than (as in select()) passing this information repeatedly. The declare_interest() system call asserts an application's interest in events on a set of one or more descriptors. The revoke_interest() system call indicates that it is no longer interested in events on a set of descriptors. For example, when a server accepts a new connection, from which it will read a request message, it calls declare_interest() with the new socket as an input parameter. Similarly, when a proxy cache starts an asynchronous I/O operation on a disk file, it calls declare_interest() with the disk file descriptor as a parameter. This indicates interest in the disk I/O completion event.

prototype	description
int declare_interest(int fd, event_t ev)	declare interest in a descriptor
int revoke_interest(int fd, event_t ev)	revoke interest in a descriptor
int get_next_event(int n, event_t *evarr, struct timeval *timo)	get up to n pending events

Table 4.2 Event-related system calls.

When an event (e.g., received packet or completed disk operation) arrives for a descriptor in the INTERESTED set of a thread, the kernel adds the descriptor to a SIGNALED_EVENTS set it maintains for the thread. My second new API, the get_next_event() system call, allows the application to obtain this set of descriptors with pending events. The application may either ask for the entire set, or for a limited number of descriptors. This allows a multiprocessor application to distribute event processing across the CPUs: each thread can ask for just one pending event, leaving the rest for other CPUs. Alternatively, a thread can request multiple events, amortizing the cost of this system call across several events. Table 4.2 summarizes the new event APIs.

There are other viable alternatives to the proposed get_next_event() system call as an event delivery mechanism. For example, event notifications can be structured as upcalls [36], with the server process specifying handlers for various types of events directly to the operating system kernel. Similarly, the kernel could indicate events to the server process by using a *pending-events* queue in a memory region shared with the application process. While these approaches might perform better than one that uses a system call, I feel that the system call approach is closer to the APIs provided by current general-purpose operating systems. For this reason, this approach might be easier to use, and reason about, than one based on upcalls or shared memory. Moreover, it avoids the need for explicit synchronization on the programmer's part unlike the other two approaches. The last is important, since one of the primary advantage of an event-driven approach vis-a-vis a multi-threaded one is that the former does not require the complexity of explicit synchronization.

The resource container mechanism allows the application to assign priorities to event sources. The kernel delivers descriptors, via get_next_event(), in priority order, allowing the application to entirely postpone its processing of low-priority events. The kernel also uses an efficient data structure, such as a priority queue, to represent the SIGNALED_EVENTS set.

4.10.2 A new bind() API

As was discussed in Section 3.7.2, a server may wish to assign different priorities to requests from different sources, even for processing that occurs in the kernel before the application sees the connection. This could be used to defend against some denial-of-service attacks, and could also be used by an ISP to provide an enhanced class of service to users who have paid a premium. As described in Section 3.7.2, the nature of the current bind() name space for TCP/IP sockets (i.e., sockaddr_in) prevents the implementation of this functionality.

To support this prioritization, I have developed a new sockaddr name space (sockaddr_in_filter) that includes a "filter" specifying a set of foreign addresses, in addition to the usual Internet address and port number. The application uses the bind() system call to bind multiple server sockets, each with the same <*local-address, local-port*> tuple but with different filters. The network subsystem uses these filters to assign requests from a particular client, or set of clients, to the socket with a matching filter.

By associating a different resource container with each socket, the server application can assign different priorities to different sets of clients, prior to listening for and accepting new connections on these sockets. This new name space also supports complement filters, to allow the application to specify the desire to accept connections at a particular socket *except* from certain clients.

When a listen socket queue becomes full (which causes an incoming SYN packet to get dropped), a signal is sent to the processes that have a reference to the socket. The new system also provides a new system call, get_listen_queue_info(), to let the application examine the identity of the connection request packets in a listen queue. This allows a server to discover that it is under a denial of service attack from a particular client. It can then take steps to address the situation.

In Chapter 7, I will present some concrete examples on how the new bind() name space is used. Chapter 8 will describe the implementation of this new name space in my prototype system.

4.11 Summary

This brings us to the end of this introductory chapter about resource containers. We discussed the container mechanism, and saw how containers can be used to ensure control over the resource consumption of an independent activity at a fine level of granularity, even when the activity spans multiple protection domains. We also discussed two new APIs that allow applications to make use of the fine-grained resource management capabilities of resource containers.

The next four chapters will continue to describe the new resource management model in more detail. In Chapter 5, we will describe LRP, which is the execution method for performing kernel network processing for an application in a system based on resource containers. Chapter 6 will show how resource containers can be used with various CPU scheduling policies. The use of resource containers in server applications is the subject of Chapter 7. Finally, Chapter 8 will describe a prototype implementation of resource containers in Digital UNIX, a widely-used commercial operating system.

Chapter 5

Lazy receiver processing

This chapter describes Lazy receiver processing (LRP), a kernel execution method for network I/O processing that is used in the container based resource management model introduced in the previous chapter. As described in Chapter 2, current operating systems perform network processing in the context of software interrupts. This eager receiver processing (ERP) method causes system resources consumed during protocol processing to be unaccounted for and uncontrolled, as was explained in Section 3. In that discussion, it was also seen that this aspect of current operating systems is the root cause of many performance problems of server systems.

LRP is a general mechanism for performing kernel network I/O processing in the resource context of the concerned application. In effect, LRP integrates the system resources consumed in network processing into the system's global resource management framework. Thus, LRP attempts to eliminate the poor performance of servers, which is caused by uncontrolled network processing, by design.

This chapter is structured as follows: it begins with a brief overview of the basic design of LRP. Subsequently, the detailed working of LRP in the context of the TCP/UDP/IP protocol suite is presented. Finally, we discuss the effect of LRP on the dynamics of the TCP protocol. Implementation issues concerning LRP are described in Chapter 8.

5.1 Basic design

Recall from Section 2, current general-purpose operating systems perform a substantial portion of network processing in the context of software interrupts. The execution of these software interrupts occurs at a higher priority than any application processing. Moreover, the kernel does not distinguish between different connections while performing protocol processing of incoming packets. Packets destined for all connections are queued for the protocol code in the same FIFO queue (IP queue), and processed by the protocol code in FIFO order. Furthermore, resources consumed during the execution of software interrupts are charged to whatever process happens to be running at the time of the interrupt, or to no process at all; i.e., the resource consumption of a significant fraction of network processing is miss-accounted.

As mentioned before, the goal of LRP is to extend the system's global resource accounting and management mechanisms to include resources consumed in network processing. To this end, it is desired that the system performs kernel processing of network packets that are destined for a resource principal, or being transmitted by a principal, at the priority of the principal. Since many general-purpose operating systems use recent resource usage as feedback to modify priority, it is also desired that the resources consumed in network processing are charged to the appropriate principal. To achieve its goals, LRP uses the following combination of techniques:

1. The system's global IP queue is replaced with a per-socket queue. This per-socket queue forms part of a per-socket data structure called a *network interface (NI) channel*. When a socket is bound to a local port (either implicitly or explicitly by means of a `bind()` system call), an NI channel is created. Also, when a connected stream socket is created, it is allocated its own NI channel.

2. The network interrupt handler demultiplexes incoming packets according to their destination socket, and places the packet directly on the appropriate receive queue. Packets destined for a socket with a full receiver queue are silently discarded (early packet discard). A hardware based approach may also be used; here the demultiplexer forms part of the network interface firmware. This demultiplexing operation is discussed further in Section 5.2.

3. Receiver protocol processing is performed at the priority of the receiving principal. Moreover, whenever the protocol semantics allow it, receiver protocol processing is performed lazily, in the context of the user thread performing a receive system call. This is in contrast to current systems where all receiver processing is done in software interrupt context, and at a statically high priority, as was described in Chapter 2.

4. All sender protocol processing is also performed at the priority of the sending principal. In current systems, some sender processing, such as the processing of outgoing UDP packets, is performed at the priority of the sending principal. As described in Section 2, however, current systems often process outgoing TCP

packets in the context of software interrupts caused by received acknowledgment packets.

In their full generality, these techniques work with any reasonable notion of what constitutes a resource principal. For example, a system that considers processes as principals can use LRP as an execution method for network processing [10, 52]. In the following discussion, resource containers are the resource principals of the system.

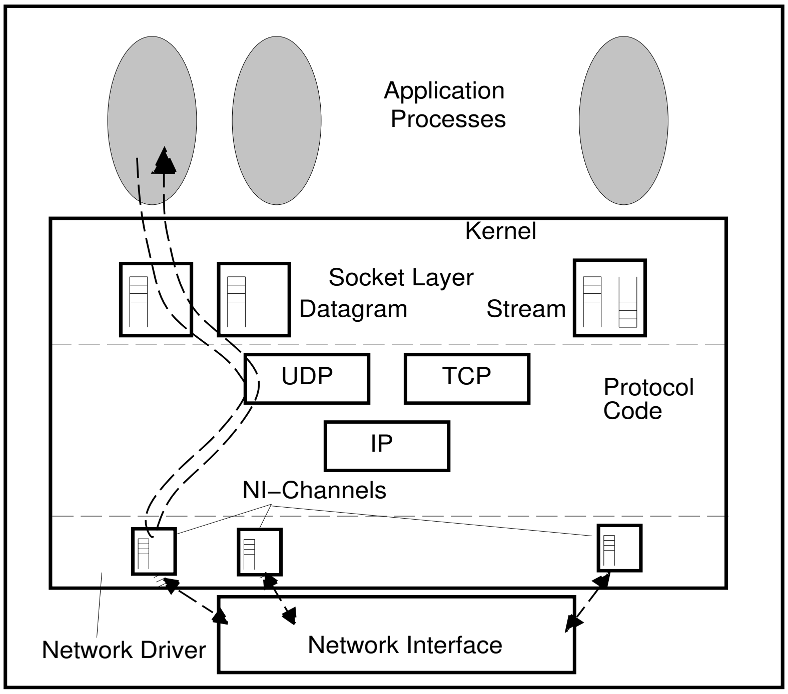

Figure 5.1 The architecture of a network subsystem based on LRP

Figure 5.1 illustrates the architecture of a network subsystem based on LRP.

There are several things to note about the behavior of a LRP based network subsystem. First, protocol processing for a packet in many cases does not occur until the application requests the packet in a receive system call. Packet processing no longer interrupts the running thread at the time of the packet's arrival, unless the receiving principal has higher scheduling priority than the currently executing principal. This arrangement avoids inappropriate context switches and can increase performance.

Second, the network interrupt handler separates (demultiplexes) incoming traffic by destination socket and places packets directly into per-socket receive queues. Combined with the receiver protocol processing at application priority, this provides feedback to the network interface about application's ability to keep up with the traffic arriving at a socket. This feedback is used as follows: Once a socket's receive queue fills, the demultiplexer discards further packets destined for the socket until applications have consumed some of the queued packets. Thus, an efficient demultiplexer can effectively shed load without consuming significant host resources. As a result, the system has stable overload behavior and increased throughput under high load.

Third, the demultiplexer's separation of received traffic, combined with the receiver processing at the priority of the appropriate resource principal, eliminates interference among packets destined for separate principals. Thus, the delivery latency of a packet cannot be influenced by a subsequently arriving packet of equal or lower priority. In addition, the elimination of the shared IP queue greatly reduces the likelihood that a packet is delayed or dropped because traffic destined for a different socket has exhausted shared resources.

Finally, CPU time spent in receiver protocol processing is charged to the resource principal that receives the traffic. As mentioned before, this correct accounting is important since the recent CPU usage of a resource principal influences the priority that the scheduler assigns to the principal. In particular, it ensures fairness in the case where the various resource principals receive high volumes of network traffic.

It is important to note that both of the techniques described above—lazy protocol processing at the priority of the receiver, and early demultiplexing—are necessary to achieve the goals of LRP. Lazy protocol processing trivially depends on early demultiplexing. To see this, observe that the receiver resource container of an incoming packet must be known to determine the time and priority at which the packet should be processed.

Conversely, early demultiplexing by itself is not sufficient to meet the goals of LRP. Consider a system that combines BSD's traditional eager protocol processing with early demultiplexing. Packets are dropped immediately in case their destination socket's receive queue is full. One would expect this system to remain stable under overload, since traffic arriving at an overloaded endpoint is discarded early. Unfortunately, the system is still defenseless against overload from incoming packets that do not contain valid user data. For example, a flood of control messages, such as TCP FIN packets, or corrupted data packets can still cause livelock. This is because processing of these packets does not result in the placement of data in the socket queue, thus defeating the only feedback mechanism that can effect early packet discard.

In addition, early demultiplexing by itself lacks LRP's benefits of reduced context switching and fair resource allocation [10, 52], since it shares BSD's resource accounting and eager processing model. Early demultiplexing and lazy processing are discussed further below.

5.2 Early demultiplexing

As mentioned above, the LRP execution method requires the demultiplexing of incoming packets as early as possible to the correct socket. Essentially we need a function that executes early in the receiver processing path and classifies network packets to their destination NI channel. For TCP/IP packets, this corresponds to a demultiplexing operation that is based on the *<protocol, local-address, local-port, foreign-address, foreign-port>* tuple embedded in the packet. This operation is similar to the in_pcblookup() function used by the BSD TCP/IP code to classify processed IP datagrams to the correct TCP or UDP socket; it is different because it operates on raw, physical-level packets, such as Ethernet frames, instead of processed IP datagrams. In effect, the demultiplexing operation locates the correct resource container that needs to be charged for the remainder of the processing of a particular packet.

Early demultiplexing can be performed in a variety of ways. It can be performed in hardware (i.e., in the network interface itself) or in software, in the context of the network interrupt handler. The hardware based approach involves the implementation of the demultiplexing function in the firmware of the network adapter, and a control-information transfer API between the adapter and the operating system. The NI channel queues are shared with the network interface. The network interface demultiplexes incoming packets according to their destination socket, and places the packet directly on the appropriate NI channel queue. Packets destined for a socket with a full receiver queue are thus silently discarded without even causing an interrupt on the host CPU.

For the software based approach, the demultiplexer inherently introduces some host overhead for each received packet. The demultiplexer, along with the rest of the network interface hardware interrupt handler, is the only part of network processing code that executes outside of the system's priority-controlled resource management model. A host interrupt and subsequent software-demultiplexing operation always happens, and suspends the currently executing principal, even if the receiving socket has a full receive queue in its NI channel. Thus, the demultiplexing function must be fast and should have a small data cache footprint; the efficiency of the demultiplexer directly controls the overload stability, and hence the robustness, of the system.

To implement a software based demultiplexer, a *packet-filter* [99, 112] may be used. Due to its interpreted nature, however, it is likely to be too inefficient. A custom, compiled demultiplexer [166, 83, 173] can also be used. An approach based on the DPF (Dynamic packet filter), as described by Engler and Kaashoek [58], is likely to be the most efficient.

I have used a software based, compiled demultiplexer in my prototype implementation. This will be described in more detail in Section 8. For my implementation, the overhead of software demultiplexing stays within acceptable limits even at the highest rates of incoming network packets that I have been able to generate in my experimental testbed.

5.3 Lazy processing for UDP

Receiver protocol processing in LRP is performed at the priority of, and charged to the resource container that contains the receiving socket. This ensures correct accounting of system resources consumed in network processing and allows the scheduler to enforce the priority policies desired by the application.

For unreliable, datagram-oriented protocols like UDP, network processing proceeds as follows: The transmit side processing remains largely unchanged. Packets are processed by the UDP and IP code in the context of the user application performing the send system call. Then, the resulting IP packet(s) are placed on the interface queue.

On the receiving side, the demultiplexer determines the destination socket of incoming packets and places them on the corresponding channel queue. If that queue is full, the packet is discarded. With demultiplexing in the NI, if the queue was previously empty, and a state flag indicates that interrupts are requested for this socket, the NI generates a host interrupt. As mentioned earlier, with software demultiplexing, a host interrupt always occurs.

When a user thread calls a receive system call on a UDP socket, the system checks the associated channel's receive queue. If the queue is non-empty, the first packet is removed; else, the thread is blocked waiting for an interrupt from the NI. After removing a packet from the receive queue, IP's input function is called, which will in turn call UDP's input function. Eventually the processed packet is copied into the application's buffer. All these steps are performed in the context of the user thread performing the system call.

It is important to realize that LRP does *not* increase the latency of UDP packets. The only condition under which the delivery delay of a UDP packet could increase under LRP is when a host CPU is idle between the time of arrival of the packet and the invocation of the receive system call that will deliver the packet to the application. This case can occur on multiprocessor machines, and on a uniprocessor when the only runnable application blocks on an I/O operation (e.g., disk) before invoking the receive system call. To eliminate this possibility, an otherwise idle CPU should always perform protocol processing for any received packets. This task is easily accomplished by means of a kernel thread with minimal priority that checks NI channels and performs protocol processing for any queued UDP packets.

5.4 Lazy processing for TCP

Protocol processing is slightly more complex for a reliable, flow-controlled protocol such as TCP. As in the original architecture, data written by an application is queued in the socket queue. Some data may be transmitted immediately in the context of the user thread performing the send system call. The remaining data is transmitted in response to arriving acknowledgments, and possibly in response to timeouts.

The main difference between UDP and TCP processing in the LRP architecture is that receiver processing cannot be performed only in the context of a receive system call, due to the semantics of TCP. Because TCP is flow controlled, transmission of data is paced by the receiver via acknowledgments. Achieving high network utilization and throughput requires timely processing of incoming acknowledgments. If receiver processing were performed only in the context of receive system calls, then at most one TCP congestion window of data could be transmitted between successive receive system calls, resulting in poor performance for many applications.

The solution is to perform receiver processing for TCP sockets asynchronously when required. Packets arriving on TCP connections can thus be processed even when the application thread is not blocked on a receive system call. Unlike in conventional architectures, however, this asynchronous protocol processing does not take strict priority over application processing. Instead, the processing is scheduled at the priority of the application container that contains the associated socket, and CPU usage is charged back to that container.

Under normal conditions, the application has a sufficiently high priority to ensure timely processing of TCP traffic. If an excessive amount of traffic arrives at the socket, the application's priority will decay as a result of the high CPU usage. Eventually, the protocol processing can no longer keep up with the offered load, causing the channel receiver queue to fill and packets to be dropped by the NI. In addition, protocol processing is disabled for listening sockets that have exceeded their listen backlog limit, thus causing the discard of further SYN packets at the NI channel queue.

A variety of methods can be used to implement asynchronous protocol processing (APP). The exact choice of mechanism depends on the facilities provided by the particular operating system kernel in question. For instance, a dedicated per-process kernel thread can be used. While performing protocol processing for a particular connection, this thread can set its resource binding to the container corresponding to this connection. This ensures that the resources consumed by the network processing thread while performing protocol processing for this connection are charged to the correct container. If there is pending protocol processing for multiple containers, the priority (or other scheduling parameters) of these containers determines the order in which they are serviced by the network processing thread.

Since protocol processing always runs to completion, no state needs to be retained between activations. Therefore, it is not necessary to assign a private runtime stack to the APP thread; a single per CPU stack can be used instead. A non-threaded kernel might use a more ad hoc mechanism to perform this processing, such as the signal context of an application [10].

5.5 Other protocol processing

Processing for certain network packets cannot be directly attributed to any application. In the TCP/IP suite, this includes processing of some ARP, RARP, ICMP packets, and IP packet forwarding. In an ERP system, processing of such packets is performed in the context of a software interrupt. For the reasons described in

Chapter 2, an excess of such traffic can also monopolize all the CPU resources of a system.

In LRP, this processing is charged to "daemon" containers that act as proxies for a particular protocol. These daemon containers can be structured as children of a particular "master" daemon container in the resource container hierarchy. Thus, the total CPU usage of the daemon containers can be limited to an administrator-specified fraction of the system resources under full load.

As a concrete example, consider the activity of IP forwarding. In LRP, an IP forwarding daemon is charged for CPU time spent on forwarding IP packets, and its priority controls resources spent on IP forwarding. The IP daemon competes with other containers for CPU time. This ensures that excess IP forwarding traffic cannot monopolize the CPU resources of a system. Moreover, as described above, the administrator of a machine can set the priority of the IP forwarding daemon and thus has complete control over the CPU cycles consumed in forwarding when the machine is overloaded and CPU resources are under severe contention.

5.6 Effect of LRP on TCP dynamics

We will now examine the effect of LRP on TCP's performance. Since TCP's transmission of data depends on timely processing of acknowledgment packets, a concern is whether LRP's lazy processing of packets has an adverse effect on TCP dynamics. This issue also arises in user-level network subsystems [91, 17, 50, 53, 57, 96, 160], which, like LRP, perform early demultiplexing of incoming packets and protocol processing at the priority of the receiver application. Like LRP, user-level network subsystems can delay protocol processing.

The following discussion presents an analysis of the effect of LRP's lazy processing of packets on TCP's performance. Before this discussion, however, a brief review of the operation of TCP is presented, which provides the necessary background for understanding the subsequent discussion.

5.6.1 Background

TCP [134] provides reliable data streams between hosts. Besides ensuring loss-free and in-sequence delivery of data, TCP also provides flow and congestion control. In the following discussion, I will briefly describe TCP's flow and congestion control algorithms. For more details, the interested reader is referred to Stevens [152]. TCP's congestion control has also been described in detail by Van Jacobson [81].

Flow control

The aim of TCP's flow control is to pace the transmission of data by the sender at the rate at which the receiver application can consume data. Ideally, we want the sender to put new data into the network whenever the receiver application reads from its receive buffer. TCP implements this by sending a control packet, which is called an ACK, back to the sender whenever the receiver application consumes data. This ACK packet allows the sender to transmit as much additional data as the receiver application has consumed.

For good throughput, a TCP sender transmits up to a window of data before waiting for an ACK. This window is usually much larger than a single packet. The use of a window allows for overlap of communication and receiver computation, i.e., the transfer proceeds even when the receiver application is not reading data from its receive buffer. For maximal overlap of computation and communication, the sender's window should be at least as large as the bandwidth of the path between the sender and the receiver times the largest interval during which the receiver application does not read from its receive buffer. In TCP's terminology, this window is referred to as the *flow control* window.

The receiver TCP needs to have enough buffer space to hold up a full window of data transmitted by the sender TCP. The receiver TCP informs the sender TCP of the available buffer space it has by advertising the unused space in its receive buffer each time it sends an ACK. The sender uses this *window-advertisement* to control its window size. In the subsequent discussion, we will refer to ACKs that allow the sender TCP to transmit new data as *window-updating* ACKs.

In practice, not all ACKs are window-updating ACKs. A receiver TCP may also send back an ACK to the sender when it reads a packet from the network interface and queues it for the receiver application after processing it*. This ACK may be sent even before the receiver application has read the data. Its purpose is to indicate to the sender that a packet has left the network. An ACK sent when TCP reads a packet

*TCP usually delays this type of ACK transmission by a maximum of 200ms, or until TCP has two unacknowledged segments; this allows TCP to piggyback ACKs on data packets traveling in the reverse direction.

from the network interface driver[†] typically shrinks the flow control window, as the available space in the receiver's network buffer decreases. Later, when the application reads data from this buffer, space becomes available and a window-updating ACK is sent.

Congestion control

To avoid overloading the network, a sender TCP does not immediately transmit a full flow control window's worth of data after a new TCP connection is established. Instead, TCP starts off by transmitting a small number of packets and gradually increases its sending window, and thus the sending rate, based on feedback from the network. More precisely, TCP's sending window is computed as the minimum of the receiver's advertised flow control window and a dynamically varying *congestion window*. This congestion window is TCP's measure of the maximum amount of unacknowledged data that it allows itself to inject into the network, based on its estimate of the available network bandwidth. The use of a gradually increasing congestion window helps the sender TCP to avoid over-running the buffers of the routers on the path to the receiver.

Let us picture the path between the sender and the receiver applications as a pipe. The "thickness" of this pipe is determined by the bandwidth of the network path between the sender and the receiver. The "length" of this pipe is determined by the network delay of this path. The goal of TCP's congestion control algorithm is to keep this pipe full by putting new data at one end as soon as some data leaves at the other end, assuming, of course, that the sender application can generate data at this rate. At the beginning of a transfer, TCP aims to fill up this pipe as quickly as possible. If congestion occurs, this means a thinner pipe is available for the transfer. TCP's aim then is to promptly adjust its sending window to the reduced pipe thickness. After congestion passes, TCP's aim is to quickly increase its sending window (and hence its sending rate) to match the increased pipe thickness.

To implement congestion control, TCP takes the receipt of an ACK to mean that a packet that it had sent earlier has successfully reached the receiver, and thus left the network. An ACK sent by a receiver TCP when it reads a packet from the network interface driver signals this event. As long as ACKs are received in order, the

[†]I am glossing over the fact that TCP actually reads from IP, and IP reads from the network interface driver; the ommision of this detail does not affect the the results of this discussion.

congestion window is increased. Missing and duplicate ACKs may indicate congestion and TCP uses such events to cut down its congestion window, and thus its sending rate.

When the sender-to-receiver pipe is saturated, the sending window (and thus the sending rate) is limited by the flow control window. At this point, only ACKs that advertise a non-zero flow control window allow the sender TCP to transmit more data. In particular, ACKs sent when TCP reads data from the network interface driver advertise a zero valued flow control window and therefore do not allow the sender to transmit new data. Thus, when the congestion window is less than the receiver's advertised flow control window, the transfer rate is determined by the congestion control algorithm. When the congestion window is equal to the flow control window, the speed of the transfer is governed by the flow control algorithm.

With this background, I will now proceed with an analysis of the effect of LRP on TCP dynamics.

5.6.2 Effect of lazy packet processing on TCP

There are four potential effects of LRP on TCP dynamics. The first effect arises because of correct accounting of protocol processing in LRP, compared to BSD. This accounting decreases the amount of CPU resources available for protocol processing, and thus potentially the throughput of a sender-bound or a receiver-bound data transfer. Lazy processing of ACKs potentially affects the operation of both TCP's flow and congestion control algorithms. Finally, LRP causes some burstiness in the transmission of data. This increased burstiness has the potential to adversely affect network performance. Each of these effects is discussed in detail below.

Effect of correct accounting

In LRP, system resources consumed during network processing are correctly charged to the concerned application. As a result of this, an application performing network I/O is generally charged a higher CPU usage in a LRP system than in a classical ERP system. Since the fair-share CPU schedulers of general-purpose operating systems use recent CPU usage to determine scheduling priority, this causes a network-intensive application to have lower priority under LRP than under ERP. Consequently, a network-intensive application receives just its fair share of the CPU in a LRP system. This is in contrast to an ERP system, where such applications may get a "more-than-fair-share" of the CPU.

This effect may lead to a lower throughput in a data transfer if the receiver or sender application is the bottleneck in the transfer. This is because LRP potentially decreases the amount of CPU available to the receiver or sender application for protocol processing to its correct fair share. Notice that this effect is both correct and desirable. If the scheduler wants to favor applications that perform network I/O, it can do so easily by using an appropriate and explicit policy. LRP provides the mechanism for the scheduler to be in control of all CPU consumption. This control is not possible under ERP, where network-intensive applications are favored independent of the desired scheduling policy. In Section 9.2.10, we will experimentally quantify this effect.

Effect on flow control

LRP does not affect the performance of TCP's flow control algorithm. The reasons for this are discussed below: First, recall from Section 5.6.1, that once the sending window of a TCP sender is equal to the receiver's advertised flow control window (i.e., the transfer is limited by flow control), the sending rate is controlled by the receipt of ACKs that are sent when the receiver application reads data from its receive buffer.

In a LRP system, each time the receiver application reads data from its receive buffer, the receiver TCP immediately sends an ACK that indicates to the sender that this data has been read. As mentioned earlier, such ACKs are called window-updating ACKs, and allow the sender TCP to transmit more data. Window-updating ACKs are never delayed by LRP; in this respect, the behavior of a LRP system is operationally identical to that of an ERP system. Thus, the transmission of data is essentially paced by the rate at which the receiver application consumes data, which is identical to the situation in an ERP system.

As mentioned before, a receiver TCP may also send an ACK to the sender when a packet is processed. LRP may delay the transmission of such ACKs because it performs protocol processing only when the receiver application has sufficient priority. However, when the transfer is limited by the flow control algorithm, the timing of such ACKs has no effect on the speed of the data transfer. This reason for this is that when the sending window is equal to the flow control window, such ACKs do not allow the sender to transmit any more data than it is already allowed to send. Each

ACK advertises a receiver flow control window equal to the old advertised window minus the size of the acknowledged packet, which shrinks the sending window.

To summarize, LRP has no effect on the operation and performance of TCP's flow control algorithm. LRP does not delay any ACKs that control the sending rate of a sender TCP, if this rate is being determined by the flow control algorithm. Section 9.2.10 will present some experiments to verify the qualitative argument presented above.

Effect on congestion control

We will now consider the effect of LRP on TCP's congestion control algorithm. As stated in Section 5.6.1, for achieving maximal network utilization, a sender TCP attempts to put new data at one end of the sender-to-receiver pipe as soon as it knows that data has left this pipe at the other end. Since sender transmission is driven by the receipt of ACKs, it is important for these ACKs to be processed in a timely fashion.

Before we discuss the effect of LRP's lazy ACK processing on TCP's congestion control, let us define a receiver process to be *runnable* if any packets destined for it can be processed immediately. In LRP, this is possible if this process is currently executing on the CPU or has a priority that is sufficiently high for the process to be scheduled immediately after a packet destined for it arrives at the receiver host. If none of these conditions is true, the receiver process is *non-runnable*. Similarly, a sender process is runnable if it is currently running on the CPU or will be scheduled immediately after an ACK for an outstanding packet transmitted by it arrives at the sender; otherwise it is non-runnable.

Now consider a non-runnable receiver application. Assume that the receiver's progress is limited by the rate of data arrival, i.e., the sender-to-receiver pipe has not yet filled up. In LRP, no ACKs are sent back to the sender while this receiver application waits to regain the CPU. Thus, during this time the sender does not transmit new data, causing the sender-to-receiver pipe to stall. This is undesirable because the network bandwidth of the sender-to-receiver path is underutilized during this time period. A similar effect arises if the sender process becomes non-runnable when the transfer is limited by congestion control.

This phenomenon does not happen in ERP because protocol processing is performed in software interrupt context. ACKs are sent back even when the receiver is non-runnable. At the sender, new data is transmitted even if the sender application is non-runnable (as long as the sending application's send buffer contains untransmitted data).

In practice, the effect of LRP on TCP's congestion control depends on the CPU scheduling policy being used. Consider first, a fair-share feedback-scheduled system, e.g., UNIX, where recent CPU usage determines application priority. In such a system, if the sender cannot transmit enough data to keep the receiver busy, the receiver application will block and gain priority. Since LRP does not delay protocol processing if the priority of the receiver application is sufficient, the receiver will perform eager processing and cause the immediate transmission of ACKs to the sender, just like in ERP. Thus, the sender TCP will transmit data the same rate as in ERP.

A similar effect happens on the sender side; if the transfer is limited by the network, the sender blocks and gains priority. As a result it processes ACKs eagerly and transmits new data into the sender-to-receiver pipe immediately after the ACKs are received, just like in ERP.

These effects happen on any system where recent CPU usage determines scheduling priority, and are a natural consequence of a scheduler's feedback property. This feedback property is what makes the UNIX scheduler good for a system where interactive applications coexist with CPU-bound applications. In a sense, network applications are like interactive processes. If they do not consume significant CPU cycles they have higher priority than other applications and are scheduled eagerly.

In other types of systems, such as in one that uses a rate-monotonic scheduler [94], this automatic receiver priority adjustment does not happen. As a result, we may see poor transfer throughput if the transfer is limited by congestion control. To see this, consider a rate-monotonic LRP system where a receiver application requests $10ms$ of CPU in every $100ms$ timeslot. (Assume that the round-trip time of the path between the sender and the receiver is much less than $100ms$.) Initially, let the congestion window at the sender be equal to 1. During the first $100ms$ the sender sends one packet before blocking. This causes the receiver, which is blocked on a "receive" call, to wake up, process this packet and subsequently block for want of data. Since this takes very little CPU time, the receiver does not use its allocated $10ms$ fully. In the next round, the congestion window is 2 and so the receiver has more data to process but it still blocks for want of data before its $10ms$ timeslice expires. This situation continues until the congestion window is large enough for the receiver to remain busy throughout its $10ms$ timeslice. It may take many hundreds of milliseconds to get to this state.

In contrast, with a rate-monotonic ERP receiver system, the congestion window is increased rapidly by ACKs that are sent eagerly. There is enough data to keep the receiver busy soon after its first timeslice.

The performance degradation described above happens because the receiver application in the rate-monotonic system takes a naive view of network processing. Instead of requesting $10ms$ every $100ms$ to account for receiving and processing 100 packets every $100ms$, the receiver application should request 100μs every $1ms$ to more closely match the timing corresponding to packet arrivals at the receiver system. It should not expect network processing to happen on time for free as in the ERP model. CPU time requests, at a granularity that matches individual packet processing times, will solve this slow increase of the congestion window.

Two caveats are in order here about the above discussion. First, note that there may be a transient degradation in TCP's throughput due to the time granularity at which the receiver's (or sender's) scheduler (including a feedback based scheduler) makes its scheduling decisions. For example, this may cause a delay between the instant that a sender TCP cuts its transmission rate on detecting congestion, and the instant that the corresponding receiver system's scheduler increases the priority of the receiver application. In the experiments described in Section 9.2.10, I have observed this issue to have little impact on performance.

Secondly, in some situations it may be that the receiver application is non-runnable when the transfer is limited by congestion-control, even in a receiver system that has a feedback based fair-share scheduler. This may happen, for example, when the receiver application performs other computationally intensive tasks besides participating in this data transfer. (These tasks could include, for instance, the handling of other network connections.) In this scenario, packets that arrive at the receiver system when the receiver is non-runnable are not ACKed eagerly. As a result, the sender-to-receiver pipe stalls until the receiver application becomes runnable again. A similar situation can arise with a sender application that performs other, computationally intensive, tasks besides participating in data transfer.

The reason for this effect is again the fact that the receiver (or sender) application takes a naive view of network processing. The application uses the CPU resources made available to it for performing computationally intensive sub-activities, while expecting protocol processing to happen on time and for free. The application needs to isolate its various sub-activities from one another and ensure that sufficient resources are available for performing protocol processing. To achieve this, the application should bind its sub-activities to different resource containers, and assign appropriate scheduling parameters to each of these containers. Once this isolation is in place, the feedback based priority adjustment described above will happen for each of the resource containers that correspond to a data-transfer sub-activity.

To summarize this discussion, if a transfer is limited by congestion control, then LRP's effect on TCP dynamics depends on the particular scheduler being used. In a fair-share feedback-scheduled system, such as UNIX, there is an automatic adjustment of the receiver's priority when the sender does not send enough data to keep the receiver busy. As a result, the receiver application performs eager processing and the sender TCP is able to transmit data at the same rate as in ERP. A similar adjustment happens on the sender side. This adjustment does not happen if a computationally intensive sub-activity shares a resource principal with a data-transfer sub-activity. This automatic adjustment also does not happen in a non-feedback scheduled system and the application writer needs to make requests for CPU time at a granularity that closely matches individual packet arrival times. A full evaluation of TCP's interaction with scheduler dynamics is beyond the scope of this dissertation. In Section 9.2.10, the claims made above about LRP's effect on TCP dynamics with respect to fair-share feedback-scheduled systems will be experimentally verified.

Increased transmission burstiness

A final concern about LRP's effect on TCP dynamics is the increased burstiness of data transmission under LRP. This can happen because of two reasons. First, recall from Section 5.6.1 that a TCP sender transmits new data paced by the arrival of ACKs. Thus, the sending rate is determined by the spacing of the incoming ACKs. However, if the sender is not runnable when ACKs arrive at the sender host, this spacing is lost. Subsequently, when the sender becomes runnable and gets the CPU, it transmits a burst of packets corresponding to the ACKs accumulated during its inter-timeslice period.

At the receiver, a similar effect occurs if packets have to wait before being processed (because the receiver is non-runnable when they arrive at the receiver host). Subsequently, a burst of ACKs is sent after the receiver starts runnning again. This ACK burstiness (called *ACK compression*) induces packet burstiness at the sender.

Independent of LRP, ACK compression has been observed to occur because of queuing delays in network routers [147, 108]. ACK compression can also occur if a receiver application reads data from its receive buffer at a rate that is greater than the rate at which this data arrived at the receiver host. (This may happen, for example, if the receiver application reads data in bursts interleaved with periods of other computational activity.) In this case, the spacing of the generated window-updating ACKs indicates the rate of progress of the receiver and not the rate of data arrival at the receiver host. Whenever it occurs, ACK compression causes the spacing of ACKs that reach a sender to be different from the spacing on the bottleneck link in the transfer path.

Both of these phenomena are undesirable because they induce instantaneous sending rates that are much greater than the path bandwidth, and this increases the likelihood of congestion in the network. In practice, however, LRP does not increase burstiness in most situations, at least in the context of a feedback-scheduled system. To see this, assume that the congestion window at the sender is fully open. The sending rate is controlled only by window-updating ACKs that are sent when the application reads data from the receive buffer. As described earlier, such ACKs are never delayed by LRP, and no increase in ACK burstiness (and hence packet burstiness) occurs. The spacing between ACKs that do not contain window updates is altered, but this has no effect as such ACKs do not cause transmission of new data in this scenario.

If data transfer is limited by congestion control, the receiver application is unlikely to be (busy) CPU-bound, as we saw in the previous section. In this case, a LRP system effectively behaves as an ERP system and spacing between ACKs (even window-updating ACKs) is conserved.

Similarly, if congestion window at the sender is fully open, all data is sent in the context of the write system call. Thus, when the sender is non-runnable, it has no data in its send buffer. Any ACKs that arrive during this period do not cause any data to be transmitted. If the sender's congestion window is not fully open, then it blocks in the write system call and gains priority. In this case, LRP does not delay protocol processing and the sender behaves exactly as in ERP.

Notice that the above argument is also dependent on the feedback property of a scheduler. The automatic adjustment of the CPU priority of a sender or receiver based on the size of the congestion window causes burstiness of non-window updating ACKs to only happen when the compressed ACKs will not cause any data transmission. As with congestion-control, this feedback based adjustment does not happen if the data-transfer activity shares a resource principal with a computationally intensive sub-activity. Also, a non-feedback based scheduler, such as a rate-monotonic scheduler, may cause ACK compression if the granularity of a network receiver or sender application's CPU requests do not match packet arrival times. A detailed study of this effect is outside the scope of this dissertation.

Quantitative evaluation

The qualitative discussion above has been experimentally verified and the quantitative results will be presented in detail in Section 9.2.10. All experiments were conducted in the context of fair-share feedback-scheduled hosts. A preview of the experiments conducted and their salient results is presented below:

- **Effect on accounting and flow control**. A set of experiments were conducted where 24MB of data was transfered between two hosts. The transfer was configured so as to be limited by flow control. It was observed that under ERP, the network receiver process gets an unfairly large share of the receiver's CPU. Under LRP, all processes get a fair share of the CPU. The net transfer throughput is lower in the LRP case because fewer CPU cycles are available to perform protocol processing. During the active timeslices of the network receiver process the network is fully utilized; the total number of active timeslices of the network receiver process per unit of time are, however, lower in the LRP case as a result of a fairer distribution of CPU. It was also confirmed that LRP does not cause any window-updating ACKs to be delayed.

- **Effect on congestion control**. A set of experiments were conducted where a data transfer was performed between two hosts separated by a long-delay link. Artificial packet losses to simulate congestion were introduced. For a number of typical scenarios, it was confirmed that when the transfer is limited by congestion control, LRP does not delay any ACK packets. LRP's effect on the transmission of duplicate ACKs is as if the network path between the sender and the receiver is slightly longer.

Thus, all experiments confirm the arguments made earlier in this section.

5.7 Summary

To summarize this chapter, the detailed working of LRP, a new network subsystem architecture that performs network protocol processing in the resource context of the concerned application, was discussed. The effect of LRP on TCP dynamics was also described in detail. Chapter 8 will present a prototype implementation of LRP. In Chapter 9, the performance of the LRP network subsystem will be quantitatively characterized. This will experimentally substantiate the discussion in this chapter about LRP's effect on TCP's algorithms.

Chapter 6

Resource containers and CPU scheduling

This chapter describes the mechanics of CPU scheduling in a system based on resource containers. It describes the use of a number of different CPU scheduling policies with containers. We begin by considering decay-usage scheduling, which is the most commonly used CPU scheduling algorithm in current general-purpose operating systems. It is shown that it is straightforward to transform a decay-usage scheduling framework that considers processes (or threads) as resource principals, to one that considers resource containers as its principals.

Next, the chapter discusses how priority based decay-usage scheduling can be extended to a hierarchical scheduling framework that supports fixed share CPU allocation as well as regular, UNIX-style timesharing.

Finally, we discuss the use of some recently developed CPU scheduling algorithms, i.e., Lottery scheduling [169, 168] and Start-time fair queue (SFQ) scheduling [70], in a container based system. These algorithms are substantially more powerful than decay-usage scheduling for a variety of reasons that include, for example, the fact that these algorithms can support fairness in proportional share allocation at much smaller timescales than is possible with decay-usage scheduling. We show that the resource container hierarchy is analogous to the hierarchical scheduling structures used by the Lottery scheduling and SFQ scheduling algorithms. Thus, these algorithms can be easily applied to schedule a resource container hierarchy.

6.1 Decay-usage scheduling

Decay-usage scheduling is a mechanism for CPU allocation that is based on priority and recent usage. The aim of this algorithm is to ensure equitable allocation of CPU between principals, while still providing good response time for interactive processes. This scheduling algorithm is deployed in many environments, such as System V [9] and BSD UNIX [102], and the Mach operating system [1, 24].

Decay-usage scheduling is motivated by two concerns: fairness and performance. Fairness is achieved by awarding CPU quanta to processes that have received few quanta in the recent past. This also improves performance, since I/O-bound processes, which accumulate CPU time slower than compute-intensive processes, are dispatched preferentially allowing disk activity to be overlapped with CPU. In addition, since shorter jobs tend to be scheduled first, this improves response time as it approximates a shortest-job-first policy (which is known to minimize response time).

Specifically, a decay-usage scheduler operates as follows: The scheduler always gives the CPU to the runnable thread with the highest priority. The priority of a thread is computed by considering recent CPU usage and a starting base priority. A statistical accounting mechanism charges CPU usage to processes (or threads) by observing at every statistical clock (statclock) tick, the particular process (or thread) that is running. This information is used by the CPU scheduler, which is invoked periodically by the scheduling clock interrupt (hardclock), to modify priority values. The scheduling clock and the accounting clock should ideally be distinct, and use different frequencies, to avoid scheduling bias.

The detailed operation of a decay usage scheduler is shown in pseudo-code form in Figures 6.1 and 6.2. This code assumes the use of three pieces of information about processes and threads.

- p_threadpri - Thread priority
- p_cpu - Decayed CPU usage of a thread's process
- p_basepri - External adjustment to priority.

```
For the thread that has just finished running:
     p_threadpri = K * p_cpu + p_basepri

Dispatch the thread with the smallest p_threadpri
```

Figure 6.1 Working of a decay-usage scheduler– end of quantum adjustments

This code also assumes three constants, which depend on the particular system. These are K, which converts from units of CPU usage to units of priority; T, the fixed interval at the end of which CPU usage is decayed; and D (> 1), the factor by which p_cpu is decayed. In some variants of UNIX, e.g., BSD, D is a function of the load of the system, i.e., the number of runnable processes in the system.

Additionally, this code assumes that threads are scheduled based on the total CPU usage of all sibling threads that belong to the thread's process, i.e., the process is the system's resource principal.

On every statclock tick, the system charges CPU time to the running process by updating p_cpu. Figure 6.1 shows the scheduling computations that take place at the end of a CPU quantum (triggered by a scheduling clock tick or a voluntary thread block). The operating system updates the CPU usage of the executing thread's process and the priority of the executing thread.

```
for (all processes) {
    p_cpu = p_cpu/D
    for (all threads of this process) {
         p_threadpri = K * p_cpu + p_basepri
    }
}
Dispatch the thread with the smallest p_threadpri
```

Figure 6.2 Working of a decay-usage scheduler– end of decay-cycle adjustments

Figure 6.2 shows the computation at the end of each decay period, i.e., at every T scheduling clock time units. The CPU usage of all processes is decayed by the factor D and the priority of all threads in the system is re-computed.

6.2 Decay-usage scheduling of resource containers

In a system based on resource containers, the scheduler considers resource containers instead of processes (or threads) as its resource principals. Every thread's CPU usage is charged to the resource container to which the thread is resource-bound at the instance of a statclock tick. Priority values are associated with resource containers instead of threads. The scheduler gives the CPU to the highest priority *runnable* resource container, i.e., the highest priority container that has an unblocked thread associated with it. This gives CPU quanta to threads associated with the container. At the end of each decay cycle, the scheduler decays the CPU usage of each resource container in a manner similar to what was described in the previous section.

As mentioned in Section 4.3.1, if multiple runnable threads are associated with a resource container to which the system decides to give a CPU quantum (i.e., multiple runnable threads exist in the container's scheduler binding set), a second-level, potentially application-specific, scheduler decides which thread gets to run. When a resource container's quantum is mapped to a thread, its resource binding is set to this container. If a thread blocks before its resource container's CPU quantum runs out, the second-level scheduler has the option to give the CPU to another runnable thread associated with the container. In effect, the second-level scheduler of a container round-robins the CPU share of the resource container between the various threads available to it. Of course, these threads can themselves be prioritized, and these priorities can also be decayed using recent CPU usage as a metric; this allows control over the relative rates of execution of these threads.

When a thread invokes the operating system to change its resource binding, the scheduler is invoked to see if this thread needs to be preempted in order to run a higher-priority activity than the target of the resource binding change call. If no other activity has a higher priority, the current thread continues to run, but with a different resource binding.

As was briefly described in Section 4.3.1, the system keeps track of the threads available to each resource container in the scheduler binding set of the container. An application can add a thread to this set by invoking the system call add_to_sched_binding(), and can delete a thread from this set by calling delete_from_sched_binding(). Thus, an application can effectively share a single thread, T, between a large number of containers. Whenever, the scheduler awards a quantum to one of these containers, T gets to run. When T executes on behalf of a particular container, its resource usage is charged to that container.

When a CPU quantum is awarded to a container, its thread that is allocated this quantum may have been executing code corresponding to some other resource container that shares this thread when it was last preempted. In the resource container programming model, it is the application thread's responsibility to periodically check and ensure that its resource binding, which indicates what activity is currently being billed for resource consumption, is consistent with the actual code being executed. In the above example, the application thread will quickly discover the inconsistency between the activity being executed and its resource binding, and switch to executing code corresponding to the correct activity.

6.3 Priority based hierarchical scheduling

In server environments, it is often necessary to restrict the resource usage of an activity, or a set of activities to a fixed fraction of the system's total resources. For example, a Web server may want to restrict the total CPU usage of requests for dynamic documents to exceed no more than 20% of the system's total cycles. Such scheduling requirements are referred to as *service-rate guarantees*.

Service-rate guarantees are hard to achieve with a pure decay-usage scheduling algorithm. There have, however, been several efforts to build priority based weighted fair share schedulers [77, 89]. These approaches monitor CPU usage of principals and dynamically adjust conventional priorities to push actual usage closer to entitled shares. There have also been efforts to use decay-usage scheduling with appropriate base priorities to achieve service-rate guarantees [76]. Any of these approaches can be easily used in a resource management framework where resource containers, instead of processes (or threads), are the resource principals.

Since we would like to consider service-rate guarantee for activities as well as groups of activities, the desired scheduling algorithm needs to support a hierarchically structured set of resource principals. The following discussion describes such a scheduling framework.

This scheduling framework keeps track of the CPU share entitlement of each resource container in the hierarchy. Containers can either ask the scheduler for CPU share guarantees, or choose to timeshare the residual CPU share of their parent that is left over after any CPU share allocations by sibling containers. The CPU share guarantee that a resource container can ask for is limited by the residual capacity of its parent that is left after prior allocations. Thus, a container can ask for a CPU share guarantee only if its parent has a CPU share guarantee; timeshare containers can only have timeshare containers as children.

As an example, consider Figure 6.3 which depicts a simple scheduling hierarchy. In the figure, containers with circular icons are timeshare containers. Rectangular

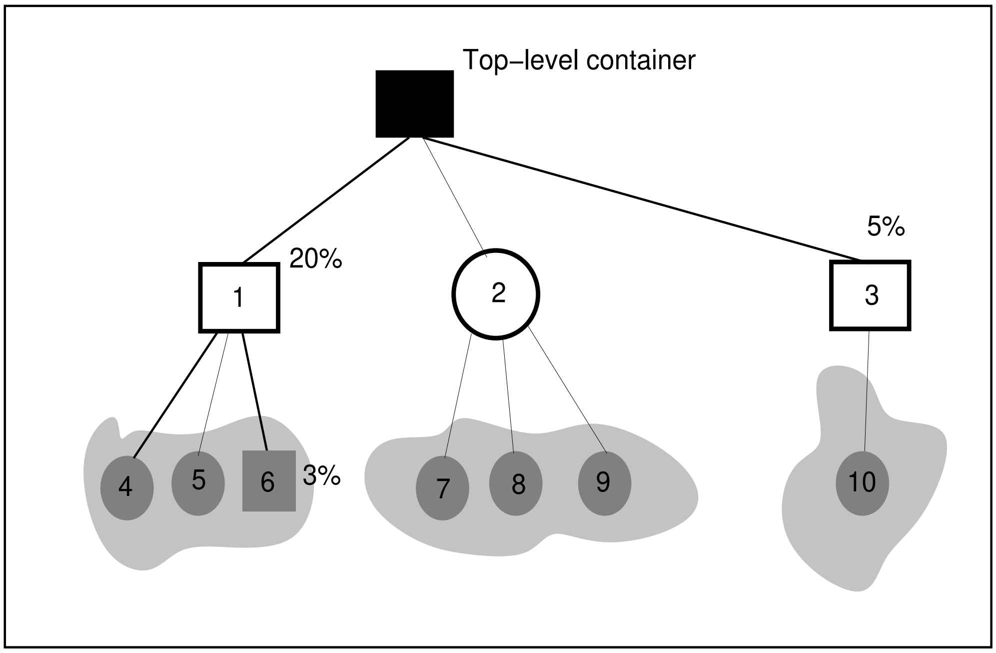

Figure 6.3 An example scheduling hierarchy.

icons correspond to guaranteed-share containers. In this example hierarchy, 20% of the system's CPU has been allocated to container 1, and 5% of the CPU has been allocated to container 3. This leaves 75% of the CPU to be shared between the timeshare children of the top-level container, in this example only container 2. The CPU share of comtainer 2 is itself shared equally between its children, i.e., containers 7, 8 and 9, each of which gets 25% of the CPU. Container 6 has a 3% reservation, which leaves containers 4 and 5 to share 17% of the CPU evenly.

Timeshare containers can also have relative weights, which determines what fraction of their parent's residual CPU share they get. For example, in Figure 6.3, if containers 4 and 5 have weights 2 and 1, they get 11.39% and 5.61% of the CPU respectively. The CPU share of each container determines its decay-usage parameters.

The hierarchical scheduling algorithm schedules the resource container hierarchy in a recursive fashion. First, a top-level node, which is a direct child of the system's root container, is scheduled. If this node is a leaf-level container, the container's second-level scheduler is invoked to schedule its threads. If not, the main scheduler runs again to schedule CPU between this container's children. The recursive operation continues until the bottom of the container hierarchy is reached.

In more detail, the operation of the scheduler is as follows: At the top-level, the kernel awards a CPU quantum to the highest priority runnable resource container subtree whose root is a direct child of the system's root container. As described before, a runnable container is one that has a unblocked thread available to run its computation. To generalize this for a container subtree in the resource container hierarchy, a runnable container subtree is one whose root container has an unblocked thread directly associated with it, or a runnable child container subtree.

When a container subtree rooted at resource container C is awarded a CPU quantum, an intra-node scheduling operation is performed to decide which child container of C is actually allocated this quantum. The scheduler now considers any children of C that have CPU-share guarantees, and C itself, for allocation of CPU. In this scheduling decision, the highest priority runnable container (or container subtree) gets to use the quantum. If this container (or container subtree) does not use up the quantum, another scheduling decision is made to award the residual to one of the other containers (or container subtrees).

If a quantum is allocated to container C in this intra-node scheduling operation, the scheduler checks if C has any unblocked threads in its scheduler binding set. If so, the container's second-level scheduler is invoked to choose the correct thread to run. If no such thread exists, or if all of these threads yield or block before the quantum expires, the CPU is now eligible to be partitioned between the child timeshare container subtrees of C. The scheduler partitions CPU quanta allocated to C between its timeshare child container subtrees in exactly the same manner as the system's total CPU cycles are partitioned at the top-level.

The scheduler computes the priority of each container (at all levels of the hierarchy) using the equation:

$$p\_containerpri = K(W * p\_cpu) + p\_basepri$$

As before, K converts from units of CPU usage to units of priority. The scheduling parameter W is chosen so as to reflect the CPU share that this container is entitled to. The value of W is proportional to the inverse CPU fraction that this container is entitled to. The intuition behind this is as follows: If the values of W for two resource containers, A and B, are in the ratio of 2:1, then the priorities of these containers increase in the ratio of 2:1 during a decay cycle. Thus, these containers, and as a result the threads bound to these containers, get CPU cycles in the ratio of 1:2.

The exact values of *p_basepri* and the normalization factor included in W are chosen so as to allow the priority values of all containers to increase continuously throughout a decay cycle without hitting the maximum priority value limit; this can happen because priorities are implemented as fixed (usually 8) bit values. (Once a priority value hits its maximum value, the corresponding resource principal continues to get CPU cycles at the same rate at which any other principal whose priority has reached the maximum value.) In particular, small weight values increase the range of control (the ability to have large differences in service rates), but decrease the granularity of control (the degree to which service rates can be allocated in small increments).

When any changes are made to the resource container hierarchy (i.e., the addition or deletion of a node, or the movement of a node within the hierarchy, or any change to the scheduling parameters of a node), the scheduler performs a re-computation of the exact CPU share that each node is entitled to. For example, if node 7 in Figure 6.3 goes away, containers 8 and 9 now get 37.5% of the CPU each. After any CPU share re-computations, the scheduler adjusts the decay-usage parameters of each container that has been affected. The scheduler also performs a re-computation of the priorities of the affected containers.

Most changes to the hierarchy, or to the scheduling parameters of individual nodes, require only local re-computation. For example, in Figure 6.3, if container 7 goes away, the changes to the scheduling parameters are restricted to node 2. Hence, the computation involved in keeping track of the CPU entitlement of each resource container does not carry significant overhead.

Priority based schemes to achieve service-rate guarantees suffer from several fundamental problems. The first problem stems from the fixed representation of priorities (e.g., 8 bits) in any real implementation. This limits the granularity of control and the range of control that a scheduling algorithm based on priorities can simultaneously provide. A larger range of control causes the granularity of control to become coarser, and a finer granularity of control decreases the range of control [76].

Secondly, priority based schemes can, in general, achieve service rate guarantees in time-scales that range from a few tens of seconds to a few minutes [169]. This makes priority based scheduling somewhat unsuitable for controlling the CPU consumption of activities that last for a time period in the sub-second range.

For these reasons, many novel schemes to support service-rate guarantees for a hierarchically structured set of resource principals have been described in the literature [25, 67, 70, 74, 169, 168]. These schemes do not depend on the notion of priority, and so are not restricted in their performance by the fundamental limitations of priority based schemes. In the next two sections, I will present a discussion of how resource containers can be incorporated into two of such scheduling frameworks, i.e., Lottery scheduling [169, 168] and Start-time fair queuing (SFQ) scheduling [70].

6.4 Lottery scheduling

Lottery scheduling [169, 168] is a randomized resource scheduling mechanism that allows good control over the relative execution rates of computations. In lottery scheduling, resource rights are represented by lottery tickets. Each resource allocation, such as a CPU quantum, is determined by holding a lottery. The resource is granted to the application with the winning ticket. Thus resources are allocated to competing applications in proportion to the number of tickets they hold.

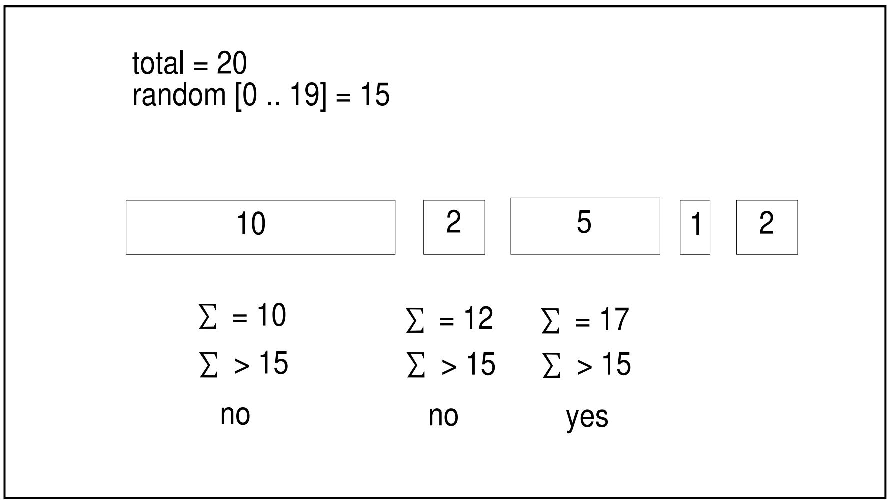

Figure 6.4 An example lottery.

For example consider the situation shown in Figure 6.4. Five clients compete in a lottery with a total of 20 tickets. The fifteenth ticket is randomly drawn, and the client list is searched for the winner. A running ticket sum is accumulated until the winning ticket value is reached. In this example, the third client is the winner.

An important aspect of lottery scheduling is the notion of currency. A currency is used to denominate tickets within each trust boundary. Currencies are backed by tickets that are denominated in more primitive currencies. The system maintains an exchange rate between each local currency and the base currency. The currency abstraction is useful for naming, sharing and protecting resource rights.

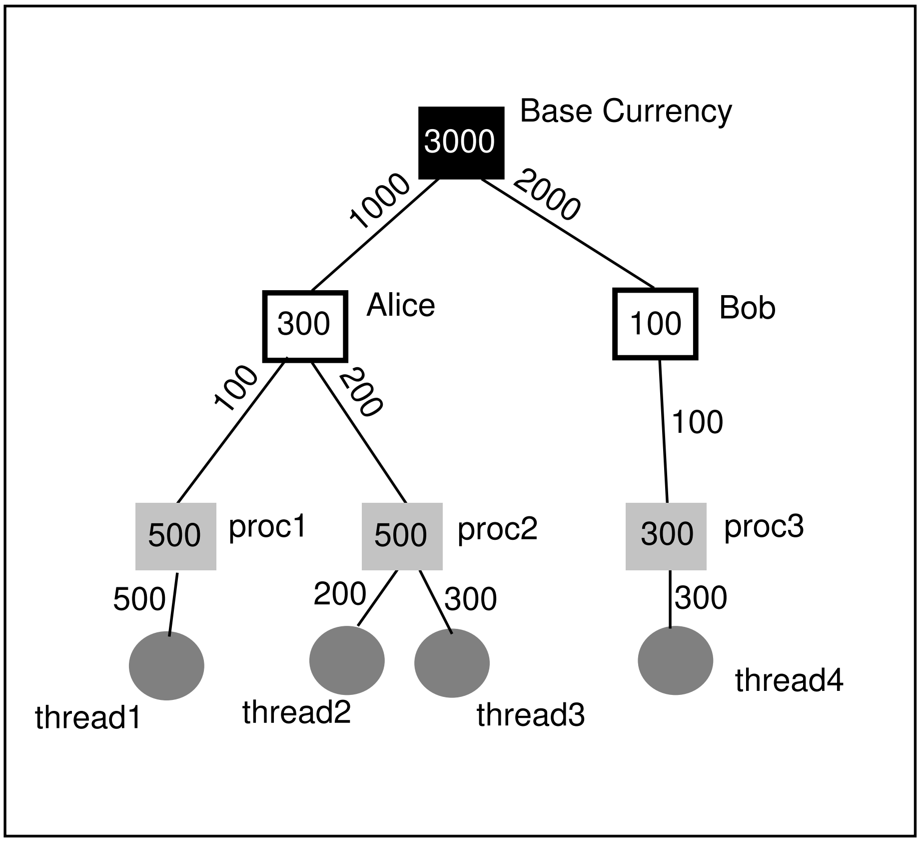

Figure 6.5 An example currency hierarchy.

The computation in a lottery scheduling framework can be described by use of a currency hierarchy. For an example, consider Figure 6.5, which shows two users competing for computing resources. A rectangular icon depicts a currency. A circular icon depicts a thread. The figure shows that user Alice is executing two processes: *proc1* has one thread and *proc2* has two threads. User Bob is executing one single-threaded process, *proc3*.

The numbers within the currency icons correspond to the total value of the currency. For example, there are in all 3000 units of base currency available. The numbers on the tree edges show backing tickets (in units of the parent currency) for a child currency. For instance, the currency *Alice* is backed by 1000 units of base currency. The current resource values in base units for the threads are *thread1* = 333.33, *thread2* = 266.66, *thread3* = 400 and *thread4* = 2000;

As we can see, currencies abstract computation units which are groups of processes. The currency hierarchy allows a group of processes to be considered together for the purpose of resource scheduling. In this sense a currency is similar to a resource container. A currency is, however, not as general a construct as a resource container because a currency hierarchy is tied to the process as its basic building block. In particular, currencies by themselves cannot be used to model scenarios such as the one described in Figure 3.6.

The lottery scheduling algorithm can, nevertheless, be easily used to schedule the resource container hierarchy. Each node in the container hierarchy can be associated with a currency. The scheduling attributes of containers and threads can be modeled by currency values and backing tickets. Each lottery gives awards a CPU quantum to a particular leaf-level container in the resource container hierarchy, which is ultimately mapped to a thread associated with the container.

Changes in the resource management policies of the system can be handled exactly as described by Waldspurger and Weihl [169, 168]. For example, in Figure 6.5 (considering the non-leaf nodes of the currency tree to be resource containers), assume that Alice starts a new independent activity, *proc4*, which has the same relative importance as *proc2*. This can be handled by inflating Alice's currency by 200 units and using these 200 units to back *proc4*'s node in the currency/container hierarchy. The various implementation-related optimizations discussed by Waldspurger and Weihl [169, 168] are also fully applicable for a lottery scheduling implementation of a framework based on resource containers.

6.5 SFQ scheduling

The Start-time fair queuing (SFQ) algorithm described by Goyal et al. [70] was designed with the explicit goal of supporting a hierarchical framework of resource principals. This algorithm has more recently been shown to be a member of a family of algorithms that can be used to schedule a hierarchy of principals [19]. The following discussion will, however, be limited to the SFQ algorithm.

In the SFQ framework, the scheduling requirements of threads are specified through a tree structure. Each thread belongs to exactly one leaf node of this tree. A leaf node represents a collection of threads, and hence an application class of the system. An interior node represents a collection of application classes.

Each node in the tree has a weight that determines the percentage of its parent node's CPU allocation that should be allocated to it. Specifically, the CPU received by node i is given by:

$$CPU_i = (\frac{r_i}{\sum_{j=1}^{n} r_j}) * CPU_k$$

where $r_1, r_2, ..., r_n$ denote weights of the children of node k, which is the parent of node i.

Each node has a scheduler associated with it. Leaf node schedulers are responsible for scheduling threads. Interior node schedulers of a node schedule its child nodes. Thus, the scheduling of threads occurs hierarchically. The root node schedules one of its children; the child node, in turn, schedules one of its children, and so on and so forth until a leaf node schedules a thread.

To schedule the interior nodes of this framework, we need an algorithm which can provide weighted fair-share scheduling of each node's children in the presence of variations in the total CPU bandwidth available to the node. The SFQ algorithm was designed to address this requirement.

In SFQ, the scheduler assigns a start tag to every entity that is to be scheduled. Entities are scheduled in increasing order of start tags. To define how a start tag is computed, let entities be scheduled for variable length quantums at a time. Also, let q_f^j and l_f^j denote the j^{th} quantum of schedulable entity f and its length (in units of instructions), respectively. Let $R(q_f^j)$ denote the time at which the j^{th} quantum is requested. The SFQ algorithm is defined as follows:

1. When a quantum q_f^j is requested by a schedulable entity f, the entity is stamped with start tag S_f, computed as:

   $$S_f = max(v(R(q_f^j)), F_f)$$

   where $v(t)$ is the virtual time at instant t (see below), and F_f is the finish tag of entity f (initially 0). After the j^{th} quantum finishes, this is incremented as:

   $$F_f = S_f + \frac{l_f^j}{r_f}$$

where r_f is the weight of thread f.

2. The virtual time is 0 when the scheduler begins operation. The virtual time at instant t, $v(t)$ is defined to be the start tag of the entity in service at time t, if the CPU is busy at t. Otherwise, $v(t)$ is defined to be the maximum finish tag assigned so far.

3. As mentioned before, entities are scheduled in increasing order of start tags. Ties between entities are broken arbitrarily.

The nodes of the SFQ scheduling hierarchy are analogous to resource containers in that both denote aggregates of computational activities. A resource container is a mechanism that provides a flexible and hierarchical notion of a resource principal. SFQ is a policy for hierarchical scheduling. Thus, the two complement each other; the SFQ algorithm can be easily be adapted to schedule the resource container hierarchy. CPU quanta are awarded to leaf-level containers instead of threads in exactly the same manner as in the vanilla SFQ algorithm. As described earlier in this chapter, a second-level scheduler associated with each container maps a CPU quantum awarded to the container to a thread associated with it. Container scheduling parameters can be expressed as weights in the manner that was described in Section 6.3.

One powerful property of the SFQ algorithm is that it can be used in conjunction with an arbitrary scheduling algorithm that undertakes to manage the scheduling of the hierarchy below a particular node in the tree. For example, a rate-monotonic scheduler [94] can be used to schedule all activities in the system that have hard-real-time requirements. This can be achieved by placing all such activities below a particular node in the scheduling hierarchy and letting a rate-monotonic scheduler manage this node. This property of the SFQ algorithm makes it particularly advantageous for scheduling the resource container hierarchy.

6.6 Summary

To summarize this chapter, resource containers can be naturally used with a variety of scheduling frameworks. These include classical UNIX-style decay-usage scheduling algorithms and a version of it that supports hierarchically structured principals, as well as newer algorithms such as Lottery scheduling and Start-time fair queuing. In a sense, the container mechanism is completely orthogonal to the scheduling mechanism used in a particular operating system. Any scheduling algorithm that can schedule traditional resource principals, such as processes and threads, can be used to schedule resource containers.

Chapter 7

The use of resource containers

This chapter describes how resource containers can be used by a server application to provide robust and controlled behavior. To begin with, it describes how resource containers can control the resource consumption of requests in a Web server. Several example server designs are considered, and resource containers are shown to be sufficiently flexible and powerful to correctly control resource consumption in each type of server.

Next, this chapter discusses how containers can be used in a Web server to control the resource consumption of various classes of requests. It illustrates the use of resource containers to protect a server from a SYN-flood based denial of service attacks. It also describes the use of containers in controlling the resource usage of guest servers in a Rent-A-Server host.

Following this, we consider the use of resource containers in controlling resource consumption in multi-user server systems on a per-user basis. Finally, the issues of accurate billing and capacity planning in server systems are visited. The subsequent discussion shows how resource containers make these previously ad hoc and error-prone tasks relatively straightforward.

7.1 Usage in a single-process multi-threaded server

First, consider a single-process multi-threaded Web server. Recall from Section 2.1 that this type of server uses a dedicated thread to handle each HTTP connection. The server maintains a pool of idle threads, each of which wait for a new connection to appear at the listen socket. When this happens, one thread returns from the `accept()` system call with a socket associated with this new connection.

In the new resource model, the server thread handling a connection creates a new resource container for the connection immediately after accepting the connection. It sets its resource binding to this newly created container, and also binds the socket associated with the new connection to this container. Any subsequent resource con-

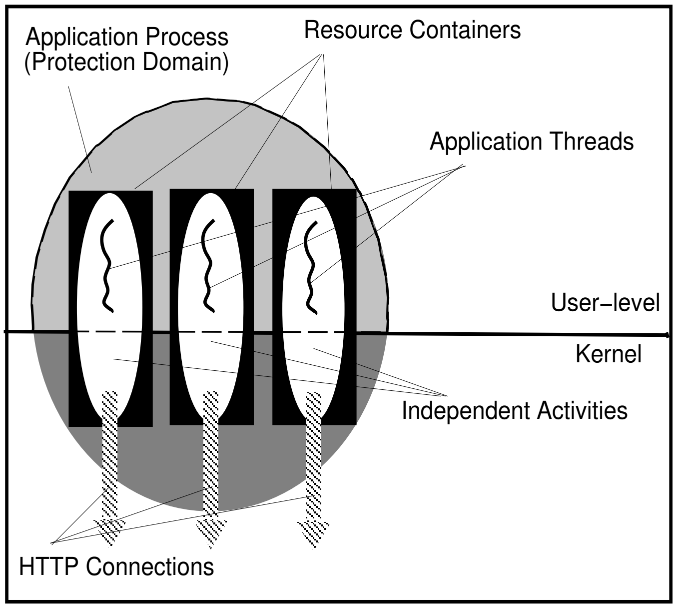

Figure 7.1 Resource containers in a multi-threaded Web server.

sumption for the connection is charged to the connection's resource container. This situation is shown in Figure 7.1. Figure 7.2 shows pseudo-code for the main loop of a multi-threaded server that uses resource containers.

The server application may also group all concurrent connections from the same client into one resource container. This causes all of a client's connections to be considered together for the purpose of resource consumption at the server. Depending on the specific objectives of the server, this may be more desirable than a container-per-connection arrangement.

If a particular connection (e.g., a long file transfer) consumes a lot of system resources, this usage is charged to the associated resource container. As a result, the scheduling priority of this container will decay, leading to preferential scheduling of containers associated with other connections. In this manner, fairness is ensured between the server's connections.

To implement prioritized handling of connections, the server can assign appropriate priorities to the containers corresponding to the connections. This causes the kernel's network processing code to prioritize the processing of packets for the server's connections accordingly. The system's CPU scheduling subsystem also assigns user-

```
while(1)  {
    /* done with previous connection, wait for connection */
    new_so = accept(listen_so, &client, &len);

    /* configure new connection */
    new_rc = rc_create();
    set_res_binding(new_rc);
    set_fd_rc(new_so, new_rc);

    /* do remaining HTTP processing */
                    .
                    .
                    .
    /* done with HTTP processing */
}
```

Figure 7.2 Pseudo-code for a container-aware multi-threaded Web server.

level CPU cycles to the various containers, and hence to the corresponding threads, in conformance with these priorities.

7.2 Usage in a single-process event-driven server

Next, consider an event-driven Web server that runs on a uniprocessor machine. From Section 2.1, we know that this type of server uses a single thread to handle all of its connections. In the new resource model, an event-driven server also creates a new resource container for each new connection immediately after accepting the connection, as described in the previous section. It may also group all connections from a given client into a client-specific container.

Later, when the server performs processing for a particular connection, it sets the resource binding of its thread to the container associated with this connection. It also adds the thread to the scheduler binding set of this connection's container. This ensures that resources that the server application consumes while handling a connection are correctly charged to the container associated with the connection. Figure 7.3 depicts this situation. Figure 7.4 shows pseudo-code for the

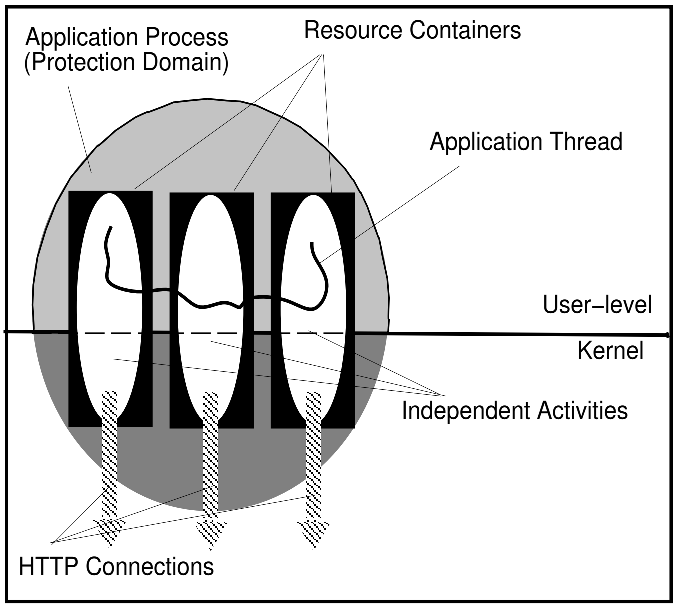

Figure 7.3 Containers in an event-driven Web server.

main loop of a event-driven server that uses resource containers. This code uses the get_next_event()/declare_interest() event API described in Section 4.10.1.

If a connection consumes a lot of resources, this is reflected in the resource usage counters of the corresponding container. The server application can obtain this usage information, and use it to adjust the container's priority, which controls how the protocol code in the kernel consumes resources on behalf of this connection. The application can also use this information to control how it expends resources for the connection at user level.

Prioritized handling of connections can be achieved in a manner that is similar to what was described for multi-threaded servers in the previous section. With an event-driven server, however, container priorities do not by themselves ensure user-level server code to be executed in a prioritized manner. Instead, all user-level cycles are made available to the single server thread, which must then perform request processing in priority order. It is not that the resource container mechanism is not important or necessary here; on the contrary, containers provide correct accounting of the resource consumption of each activity (including the component of resource usage that happens inside the kernel): information which is imperative for the user-level

```
while(1) {
    /* previous event handled, wait for next event */
    n_events = get_next_event(MAX_EVENTS, &event_info)

    /* handled returned events */
    for(i = 0; i < n_events; i++) {
        if(IS_LISTEN_SOCKET(event_info[i].fd)) {
            new_so = accept(listen_so, &client, &len);

            /* configure connection */
            new_rc = rc_create();
            set_res_binding(new_rc);
            set_fd_rc(new_so, new_rc);
            declare_interest(new_so, READ_INTEREST);
        } else if(....) {
            /* handle events on other types of sockets */
        }
    }
}
```

Figure 7.4 Pseudo-code for a container-aware event-driven Web server.

code to perform prioritized request processing. Containers also allow for the kernel portion of request processing to be performed in a prioritized manner.

For a event-driven server on a multi-processor, the situation is a hybrid of the scenarios described in this section and in Section 7.1. The server uses as many threads as processors, each of which handles a subset of the connections being handled by the server. Each thread multiplexes itself between the resource containers corresponding to the connections that it handles.

7.3 Usage in a hybrid server

In a hybrid Web server, such as Flash [131], a main event-driven server process handles requests for static resources that are cached in the main memory of the server system. Requests for static resources that require disk I/O, or requests for dynamic resources are handled by helper processes.

Resource containers can be for controlling resource consumption even in such a model. A container is created for each request. Any processing for the request in the main server process is charged to this container. If processing for a request is moved to a helper process, a reference to the corresponding container is also moved and can continue to control the resource usage of the request. The ability of containers to control resource consumption of an independent activity no matter how the activity is distributed across protection domains is further demonstrated in the next section.

7.4 Controlling resource usage of CGI processing

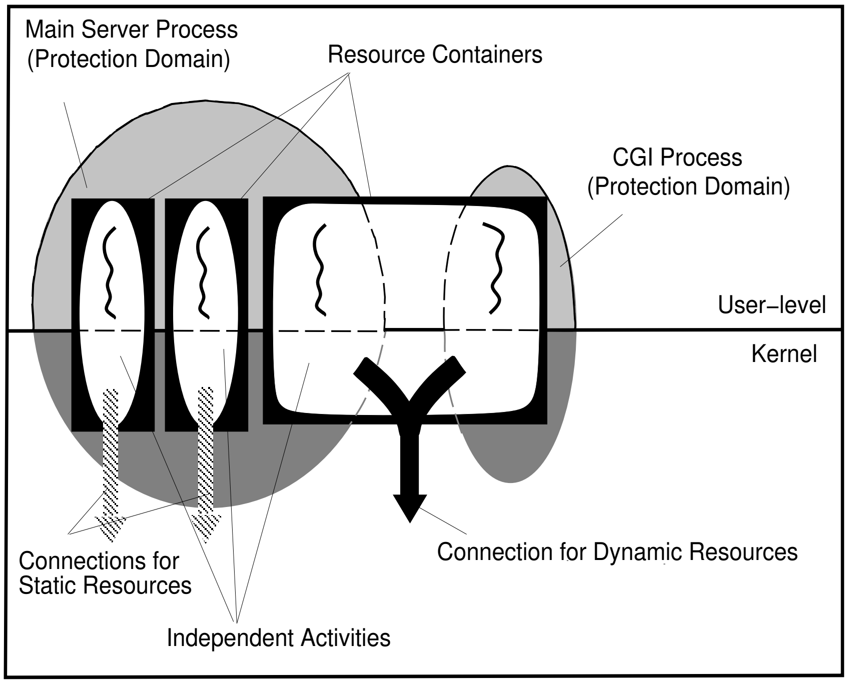

Figure 7.5 The use of containers for CGI processing.

Both event-driven and multi-threaded servers, when handling a request for a dynamic (CGI) document, can pass the corresponding connection's container to the CGI process. This may either be done by inheritance, for traditional CGI using a child process, or explicitly, when persistent CGI server processes are used. If the processing for the dynamic resource is performed in a module within the main server process itself, the application simply resource binds the thread that performs this processing to the appropriate container. In this manner, server system resources consumed in CGI processing are correctly accounted for and controlled, irrespective of how, and in what processes, the CGI processing is actually performed.

Figure 7.5 shows a multi-threaded server handling two connections for static resources and one connection for a dynamic document. This server uses classical non-persistent CGI. Notice how the resource container corresponding to the CGI request encompasses the resource consumption of the entire CGI process as well as the main server thread assigned to this request.

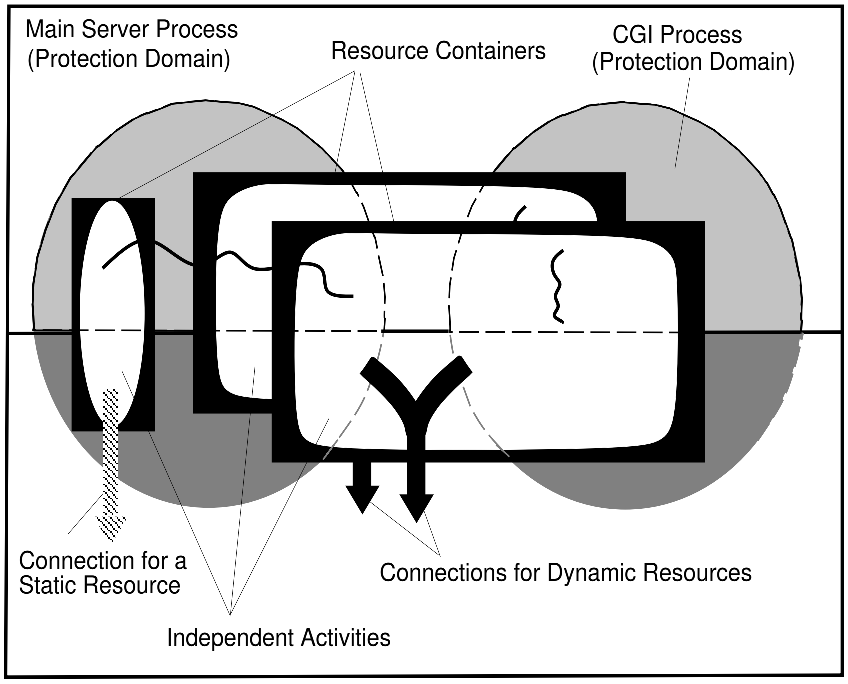

Figure 7.6 Containers and persistent CGI servers.

A different scenario that uses persistent CGI is depicted in Figure 7.6, where an event-driven server is handling one request for a static document and two requests for dynamic documents. The persistent CGI server is itself multi-threaded. The static request's CGI container is completely contained within the main server process. The two dynamic request containers span across the main server process and the CGI process. In even more complicated scenarios, e.g., where the CGI resource is created by the co-operation of multiple CGI processes, the CGI container can be passed to all protection domains as necessary.

7.5 Controlling resource usage of request classes

A Web server's administrator may wish to restrict the total resource consumption of certain classes of requests, such as CGI requests, requests from certain hosts, or requests for certain resources. The server application can implement this by creating a container for each such class, setting its attributes appropriately (e.g., limiting the total CPU usage of the class), and then creating the resource container for each individual request as the child of the corresponding class-specific container.

As an example, let us consider a hypothetical corporate Web server whose administrators wish to enforce a particular resource usage policy. Assume that it is desired that CGI requests get no greater than 25% of the machine's resources under full load. Some clients from the IP network 129.42.x.x are known to particularly malicious, so the administrators want to give lowest priority to requests from these hosts. Finally, the administrators want to reserve 5% of the system's capacity under full load for any administrative tasks that may need to performed.

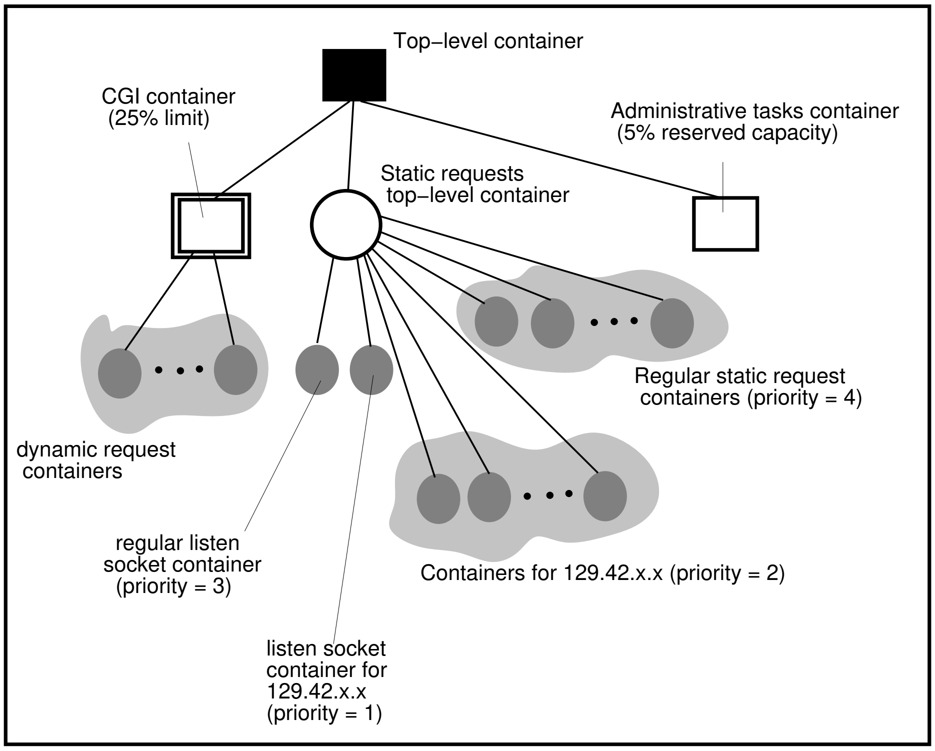

Figure 7.7 A resource container hierarchy for the server of Section 7.5.

To implement these policies, the server can use the resource container hierarchy shown in Figure 7.7. The pseudo-code for initializing this hierarchy is shown in Figure 7.8. As we can see, this code is a relatively straightforward description of Figure 7.7. It may in fact even be possible to automate this process by the use of a configuration file, which describes the server's container hierarchy in a suitable language, and a parser for implementing the hierarchy.

The code that classifies connections and assigns scheduling attributes to their associated containers for the hierarchy of Figure 7.7 is shown in Figure 7.9. This code assumes an event-driven control model for the server, but does not include a conditional statement that changes the parent of a connection's container to cgi_rc. The missing code-fragment is invoked if the server reads and parses a request for a dynamic resource.

7.6 Protection against SYN-flooding

In Chapter 3, we briefly discussed the SYN-flood based denial of service attack [143, 18, 42]. As described earlier, in this attack a malicious client sends bogus TCP SYN packets to a server at a very high rate. The client does not participate in the three-way TCP connection handshake, and so its connection requests eventually time out of the SYN-RCVD queue of the server's listen socket. By occupying all slots in the SYN-RCVD queue, however, the malicious client is able to prevent the server from handling any legitimate requests as all SYN packets from well-behaved clients get dropped.

To defend itself against a SYN-flood based denial-of-service attack from a specific set of clients, a server application can create a separate listen socket whose bind() filter matches this set. It can then bind this socket to a resource container with a low priority. This activity is triggered as follows: When a server application gets a signal from the kernel indicating that a particular listen socket's queue is full, it can use the new get_listen_queue_info() system call described in Section 4.10.2 to isolate which client, or set of clients, is responsible for the SYN-flood. It can then use this information to limit resource usage by the malicious set of clients.

Subsequently, the kernel will classify incoming connection requests from this client set to this low priority container. This has two effects: First, bogus connection requests are placed in a separate queue and therefore do not cause legitimate requests to be needlessly dropped. Secondly, the isolation of bogus requests to a low-priority

```
/* hierarchy initialization */
/* create CGI container */
cgi_rc = rc_create();
rcopt.attr_val = 25;
rcopt.attr_type = SCHED_CPU_LIMIT;
set_rc_opt(cgi_rc, &rcopt);

/* create administration container */
admin_rc = rc_create();
rcopt.attr_val = 5;
rcopt.attr_type = SCHED_CPU_RESERVE;
set_rc_opt(admin_rc, &rcopt);

/* create container for all other processing */
static_rc = rc_create();

/* create and initialize listen socket containers */
regular_listen_rc = rc_create();
bad_net_listen_rc = rc_create();
set_rc_parent(regular_listen_rc, static_rc);
set_rc_parent(bad_net_listen_rc, static_rc);
rcopt.attr_val = 3;
rcopt.attr_type = SCHED_CPU_PRI;
set_rc_opt(regular_listen_rc, &rcopt);
rcopt.attr_val = 1;
rcopt.attr_type = SCHED_CPU_PRI;
set_rc_opt(bad_net_listen_rc, &);

/* listen socket initialization, assume they have been created */
set_fd_rc(regular_listen_so, regular_listen_rc);
set_fd_rc(bad_net_listen_so, bad_net_listen_rc);
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = INADDR_ANY;
addr.sin_port = htons(80);
addr.filter.f_addr = INADDR_ANY;
addr.filter.f_mask = 0xFFFFFFFF;
bind(regular_listen_so, &addr, addrlen);
addr.filter.f_addr = atoin(``129.42.1.1'');
addr.filter.f_mask = 0xFFFF0000;
bind(bad_net_listen_so, &addr, addrlen);
```

Figure 7.8 Pseudo-code for initializing the container hierarchy of Figure 7.7.

```
     .
     .
while(1) {
        /* previous event handled */
        /* so wait for event */
        n_events = get_next_event(MAX_EVENTS, &event_info)

        for(i = 0; i < n_events; i++) {
            if(IS_LISTEN_SOCKET(event_info[i].fd)) {
                  new_so = accept(listen_so, &client, &len);

                  /* new connection accepted */
                  new_rc = rc_create();
                  set_rc_parent(new_rc, static_rc);
                  set_res_binding(new_rc);
                  set_fd_rc(new_so, new_rc);
                  declare_interest(new_so, READ_INTEREST);

                  /* check if client is blacklisted  */
                  if(IN_BAD_NET(client)) {
                        rcopt.attr_val = 2;
                        rcopt.attr_type = SCHED_CPU_PRI;
                        set_rc_opt(new_rc, &rcopt);
                  } else {
                        rcopt.attr_val = 4;
                        rcopt.attr_type = SCHED_CPU_PRI;
                        set_rc_opt(new_rc, &rcopt);
                  }
            } else if(....) {
               /* handle events on other types of sockets */
                          .
                          .
            }
        }
}
     .
     .
```

Figure 7.9 Code for classifying connections for the server of Section 7.5

container allows the server to give preference to processing of packets destined for existing connections, and new connection requests from well-behaved clients, over the processing of bogus requests.

Of course, this technique requires the network infrastructure to reject spoofed source addresses, a problem currently being addressed by the network operators community [125].

7.7 Controlling resource consumption of guest virtual servers

Resource containers also allow a Rent-A-Server host [164] to control the resource consumption of its guest virtual servers. Recall from Section 3.6, that this type of server provides an environment for supporting multiple Web servers, each of which provides the Web presence for a distinct organization.

Consider a scenario where a Rent-A-Server machine hosts N guest servers. In current operating systems, each guest server, which might consist of many processes, can appear to the system as numerous resource principals. The number may vary dynamically, and have little relation to the fraction of the Rent-A-Server host's CPU resources that its administrator wishes to allow each guest server. In particular, a sophisticated guest server can gain an unfair share of the Rent-A-Server system's resources.

Resource containers allow the Rent-A-Server host to put "resource sand-boxes" around each guest virtual server. The master server process running on the Rent-A-Server host can create N resource containers and limit the resource consumption of each to a desired value. These containers can then be provided as the initial containers of the master processes of each guest server. Any containers created subsequently by a guest server are descendents of its initial container. The resource usage of all containers of the guest server is thus limited by the share of system resources assigned to its initial container. This effectively limits the total resource usage of the guest server.

Furthermore, because the resource container hierarchy is recursive, each guest server can itself control how its allocated resources are re-divided among competing connections. Each server can itself support scenarios as complicated as the example in Section 7.5 by using an appropriate sub-hierarchy of its main resource container. Figure 7.10 shows an example hierarchy for a Rent-A-Server host.

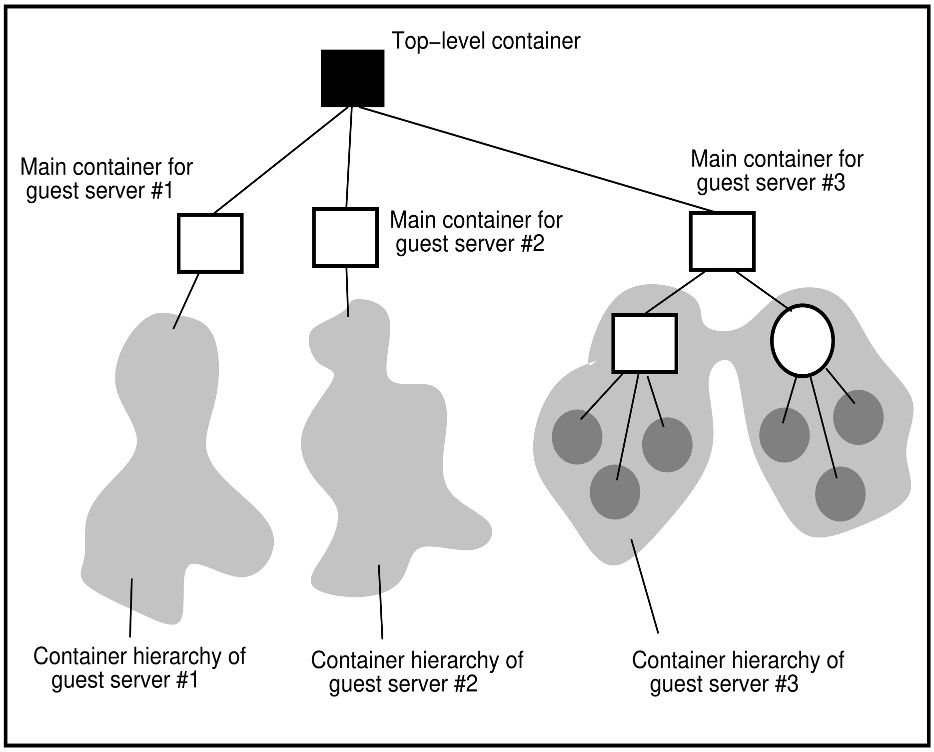

Figure 7.10 A resource container hierarchy for a Rent-A-Server host.

7.8 Use in multi-user server systems

Traditional multi-user server systems, such as departmental "CPU cycle" servers, can also use the resource container mechanism to control resource consumption. In current operating systems, the treatment of processes as resource principals causes system resources to be partitioned between users in proportion to the total number of processes that each user presents to the operating system. Moreover, a particularly network-intensive application, such as `ftp`, will have most of its resource consumption misaccounted, and will thus allow the corresponding user to obtain substantial system resources for "free".

Resource containers allow a multi-user server to correctly account for, and control resource consumption on a per-user basis. The method to do this accounting is similar to the one used in the previous section. Essentially, we can arrange things such that the resource containers corresponding to every process that belongs to a user are the children of a per-user top-level container. The scheduling parameters of these top-level containers can be controlled by the administrator of the machine.

Since the resource usage of child containers is constrained by their parent containers, this causes the various processes of a user to be effectively "resource sand-boxed" within the main container of that user. This "resource sand-boxing" potentially extends to all resources, including CPU, memory, disk bandwidth and soft kernel resources. Moreover, if appropriately flexible allocation policies are used for each of the system resource types, each user may be able to use all the resources of the system when there is no contention, and no more than its specified share, in the presence of competing users.

7.9 Use in accurate billing and capacity planning

Because resource containers enable precise accounting for the costs of an activity, they may be useful to administrators simply for sending accurate bills to customers, and for use in capacity planning.

The ability to generate accurate bills in a multi-user system can be implemented in a manner similar to that described in the previous section. Essentially, at some level in the resource container hierarchy of a system, we need to group all activities that belong to a user under a per-user top-level container. This is relatively straightforward to implement, and can subsequently be used to provide precise measures of each user's resource usage.

The more interesting of these issues is the use of containers in capacity planning. To make this discussion concrete, let us consider the problem of configuring the parameters of the server in Section 7.5. Initially, the server's administrator will design the system's container hierarchy and configure its parameters based on prior experience, desired policies and expectations of future load. For example, in Section 7.5, the decision to give lowest priority to clients from the IP network 129.42.x.x was determined by prior experience. Later, the administrator can tune the hierarchy based on further operational experience.

Now suppose after a few days of operation, the administrator of this server discover that requests for static documents have unacceptable response time. With current operating systems, the administrator of the system will be left with no option but to buy bigger/faster server hardware. With a system based on resource containers, however, the administrator can tune the resource share of requests for static documents relative to the resource shares of other types of requests and obtain acceptable performance that meets the desired level of service. For instance, in the above example, it may suffice to give no service to requests from hosts in 129.42.x.x if this brings the response time for static documents within acceptable limits.

Even if this resource re-allocation is not successful in meeting desired service standards, the resource container mechanism may help a server administrator to learn about the bottleneck factor in his system. For example, by experimenting with the memory usage bounds of requests for static documents, an administrator may find out that the server is memory-bound additional memory needs to be deployed to support a larger file-cache for static documents.

Once a process of periodic system resource usage monitoring and capacity planning is in place, the container mechanism can help an administrator to decide when and how to upgrade his server hardware. For example, resource usage measurements might indicate that the system is CPU-bound but memory is still plentiful. This will suggest a CPU upgrade leaving the amount of server memory unchanged. Additionally, by supporting an explicit container hierarchy, the container mechanism can help an administrator in evolving the resource usage policies of the server.

7.10 Summary

To summarize this chapter, resource containers can be used to support a broad range of complicated resource management scenarios in server systems. For the most part, simple modifications to existing server code suffice to ensure controlled behavior. Containers also help in accurate billing and capacity planning. Chapter 9 will quantitatively substantiate the discussion in this chapter by showing that an implementation of resource containers does indeed lead to performance that this section promises.

Chapter 8

Implementation

This chapter describes a prototype implementation of the container based resource management model. I built this prototype by modifying version 4.0D of Digital UNIX [41], which is a widely used general-purpose operating system developed and sold by Compaq Computer Corporation.

The chapter begins by presenting background information about the internal architecture of the Digital UNIX kernel. This is followed by a brief discussion of the basic philosophy that guided this implementation effort. Next, a description of the modifications made to various components of the Digital UNIX kernel for supporting the resource container abstraction is presented. Finally, this chapter describes a number of changes that were made to other components of the operating system, i.e., system tools and libraries, in order to allow applications to conveniently use resource containers.

8.1 Background information

Digital UNIX (DUNIX) [41], previously called OSF/1, is a widely used commercial operating system based on BSD [102] and Mach [1, 24] with support for some System V [9] APIs. The DUNIX kernel is essentially a BSD system built on top of a Mach core. It uses Mach mechanisms such as pre-emptable kernel threads, tasks, IPC ports and two-level virtual memory to implement BSD's kernel abstractions and services. Unlike the user-level BSD emulator [69] that ships with the standard Mach distribution, the DUNIX kernel is tightly integrated with the Mach core in the same address space. The derivation of Digital UNIX from Mach has been described extensively in the literature [107, 23, 95, 2, 126].

Digital UNIX follows the process-centric resource management model that was described in Chapter 3. Each DUNIX process is a protection domain and also (partly) a resource principal. A DUNIX process can have multiple kernel threads. These threads are scheduled directly by the kernel based on their individual priority values and recent CPU usage, and thus are the system's practical resource principals as far as CPU cycles are concerned. (The system does impose a per-process limit on the total CPU time expended by all the threads of a process, but this limit has little practical significance.) The process is, however, the resource principal for other resources, such as memory and various soft kernel resources.

CPU scheduling support in Digital UNIX takes the form of a multi-level scheduling framework. The top-level scheduler partitions CPU between threads belonging to three scheduling categories, fixed-priority round-robin, fixed-priority FIFO and regular timesharing. Second-level schedulers partition the CPU cycles made available to each scheduling category between threads that belong to the category. The kernel also supports the notion of a scheduling class within each scheduling category. Applications can be assigned to a scheduling class and resource usage limits for a configurable unit of time can be specified for each class. This allows the system to limit the total CPU fraction obtained by an application or a set of applications.

Digital UNIX implements BSD-style networking. The TCP/IP protocol code is derived from the 4.3 BSD networking code and has additional support for some newer TCP features such as window-scaling and timestamps [82], and IP support for multicast routing [47]. Network processing is performed in the context of a single system-wide dedicated kernel thread*. This thread, like a software interrupt, runs at a statically higher priority than any user-level processing. The services of the network subsystem are exposed through the socket API.

Digital UNIX implements a unified file-cache/virtual memory system. A small statically sized fraction of the system's memory is reserved for caching filesystem metadata. The rest of the memory is shared dynamically between memory pages that back user-level address spaces of processes and file-cache pages. Page replacement is performed using an approximate LRU scheme.

Digital UNIX also supports symmetric-multiprocessor (SMP) machines. The DUNIX kernel uses a combination of simple spin locks, complex sleep based locks, elevated interrupt priority levels (system priority levels, i.e., SPLs, in UNIX terminology), and funneling to guarantee synchronized access to system resources and data structures. A locking hierarchy is used to prevent deadlock. The CPU scheduler supports soft affinity of threads; a thread is preferentially scheduled on the processor on which the thread last ran. A load balancing algorithm keeps the number of runnable

*In a multi-processor machine, the system uses as many kernel threads as there are processors.

threads spread evenly across the available processors. The SMP aspects of DUNIX have also been described in detail in the literature [48, 98].

8.2 Implementation philosophy

The basic philosophy that guided this prototype implementation of resource containers was as follows: First, I wanted to minimize interface and semantic changes to the services provided by the Digital UNIX kernel. This was a primary concern because I wanted to run existing applications without any changes whatsoever on the new system. It was desired that an application that was not aware of resource containers should behave exactly as it did under a vanilla Digital UNIX system.

Moreover, I wanted to evaluate the benefits of resource containers in the context of real server applications, such as existing, commonly used Web servers, instead of toy applications and micro-benchmarks. I did not, however, want to have to go through the effort of changing application code significantly. Because of these concerns, the default behavior of the new system (even in the presence of explicit support for resource containers in the kernel) is the behavior of vanilla Digital UNIX. In addition, the semantics and syntax of all of the new kernel services of this prototype are very similar to those of existing kernel interfaces, even sometimes at the cost of being somewhat inefficient.

For example, this prototype implementation uses a signal to notify a server application of the fact that its listen socket queue is full. The application needs to make a subsequent system call to discover the identity of the listen socket before calling get_listen_queue_info() to retrieve more information about the listen queue. I did not change the implementation and semantics of signals to allow the identity of the listen socket to be communicated to the application as part of the event notification.

Another implementation goal was to minimize changes to the basic structure of the kernel. The reason for this was to be able to demonstrate the fundamental benefits of resource containers without having to filter out the secondary effects of other major changes to the kernel architecture. For this reason, it was also desired that the implementation of resource containers be SMP-safe, so that any real overheads of the resource container mechanism, such as those arising from fine-grained locking, could be evaluated.

Last but not least, although I did attempt to implement resource containers efficiently, my system has not yet been through a long cycle of evaluation and improvement, as has vanilla Digital UNIX, which is a fairly mature commercial product. Thus, further optimizations are very likely to be of benefit. This fact should be kept in mind when reading the description of the quantitative performance of the new system. I will now proceed to describe in detail the implementation of my prototype.

8.3 Implementation of the resource container abstraction

Resource containers were implemented as first-class kernel objects, along with the necessary functions to implement various operations on containers. The C structure describing a resource container object is shown in Figure 8.1.

A resource container object maintains five sets of information. The first set of data is necessary for the implementation of the container mechanism. Specifically, this includes a reference count, a spin lock, a pointer to the parent of the container, a list of pointers to its children containers (if any), and the scheduler binding set of the container.

The reference count allows a resource container to be garbage collected when there are no references left to it. The code that manipulates container references excludes cyclic reference chains by design, as is the case with many other kernel objects in DUNIX, to avoid the accumulation of uncollectable garbage. Some pointers to a container do not count as references to it. For example, a pointer from a socket object to the container that it belongs to is not a "true" reference to the container as far as the container reference counting mechanism is concerned. Instead, a socket is considered to be part of its owning container and must be destroyed, or disposed off differently (e.g., charged to another container), when the container is destroyed.

The spin lock of a container implements exclusive access to a container's internal data by the various functions that manipulate containers. The parent and children pointers help in implementing the resource container hierarchy. The scheduler binding set lists those threads that are eligible for receiving CPU quanta awarded to this container.

The second part of a container object comprises the actual system resources that are bound to the container. These include the sockets, protocol control blocks (PCBs) and NI channels belonging to the container, and the physical memory pages allocated to the container. Figure 8.1 shows this as four "resource list" structures. Each resource list structure also contains summary information about the resources allocated to the list. When a container is destroyed, these kernel resources must either be de-

```
struct resource_container {
    /* pure mechanism data */
    int ref_count;
    simple_lock_data_t rc_lock;
    struct resource_container *parent;
    struct rc_list children;
    struct thread_list sched_binding;

    /* resources inside the container */
    struct resource_list socket_list;
    struct resource_list pcb_list;
    struct resource_list ni_channel_list;
    struct resource_list mem_page_list;

    /* scheduler-specific information */
    int rc_cpu_limit;
    int rc_cpu_reserve;
    int rc_cpu_pri;
    int rc_mem_user_pages;
    int rc_max_sockbufspace;
    struct sched_info;

    /* access control */
    int rc_rights_inherit;

    /* statistics */
    struct rc_stats stats;
    int propagate_stats;
};
```

Figure 8.1 The resource container structure.

stroyed, or charged to some other container. In my implementation, any sockets, PCBs or NI channels that cannot be destroyed immediately (e.g., because of protocol semantics) are charged to a specific system-wide "dead data" container. When protocol semantics allow, this dead data is deleted. In a more sophisticated implementation, multiple dead data containers at various levels in the container hierarchy, and with different policies for the handling of dead data, may be used.

The third part of a container object consists of information specific to the scheduling of resources. It includes CPU usage limit data, reserved CPU share information and CPU priority values, and utilization limits for the various other system resource types. The system's resource schedulers use this information in their resource scheduling decisions. Since this prototype's CPU scheduler maps CPU resource policy parameters into weights and base priority values, this information is also stored as part of a container's data. This data is shown as the sched_info substructure in Figure 8.1. Although, in this prototype, CPU scheduler data is the only piece of data that is part of sched_info, this substructure can, in general, also contain any other information specific to other resource schedulers. The sched_info structure is described in more detail in Section 8.4.

The fourth piece of data in a container object is the rc_rights_inherit field. Recall from Section 4.7, that when a new reference to a container is created, user or owner rights may be associated with the new reference. The rc_rights_inherit field controls whether a new reference for a resource container inherits owner access rights to the container.

Finally, the stats field of a container object contains information about the past resource usage of this container and its late children (i.e., containers that were children of this container but do not exist anymore). The propagate_stats field controls whether these statistics are propagated to the parent container when the container is destroyed.

The operations on resource containers described in Section 4.6 are implemented by a number of kernel functions, which provide the core functionality to operate on container objects. Associated system calls expose these facilities to applications. In the following discussion, a brief description of the implementation of these kernel functions and system calls is presented:

1. **Container creation:** The kernel creates a new resource container either as a result of an explicit rc_create() system call by a thread, or implicitly as part of a fork() system call. The rc_create() call is like socket() or open(), and returns a file descriptor that points to the newly created container.

In a fork(), a new container is created to serve as the default resource context of the new process. No descriptor names this container; however, any application thread of the new process can subsequently obtain a descriptor that points to this container by using the get_rc() system call.

During the create operation, a piece of kernel memory to store the container data structure is created and initialized. The parent of a newly created container is set to the resource container that the creating thread is resource-bound to. The scheduler binding of a container created by a rc_create() call is initialized to contain just the creating thread. The scheduler binding of a new container created as part of a fork() is set to the single thread of the new process.

Internally, the DUNIX kernel treats a resource container object in a manner similar to how it treats socket and vnode[†] objects. Each container that a process can name (i.e., has a descriptor to) is present in the process' file table as a file object. These file[‡] objects have type DTYPE_RC, and point to resource container objects. This situation is similar to how file objects of types DTYPE_VNODE and DTYPE_SOCKET point to vnode and socket objects. The kernel also keeps track of one bit of information for every resource container descriptor that describes the access control rights of the descriptor. This information is maintained in a re-sizable bitmap of 64 bit words. As described in Section 4.7, the kernel uses these bits to verify the authority of the calling thread while performing various operations on resource containers.

The pointer from a file object to a container object is "worth" one reference to the container, and thus effects a container's reference count. Anonymous containers (i.e., containers that do not have a file descriptor naming them in any process) have references to them in the form of thread resource-binding pointers. Thus, to exist, a container object should either have at least one thread that is resource-bound to it, or a file descriptor reference to it in any process.

2. **Container release:** Application processes explicitly release their references to resource containers using the close() system call. This causes the file object corresponding to this container in the processes' file table to be destroyed. The

[†]A vnode is a data structure that describe a traditional UNIX file.

[‡]These file objects are referred to as open file objects in many systems.

associated pointer in the file object to the resource container object is also destroyed, and the reference count in the container object is decremented.

Implicitly, a process releases a reference to a container when it is terminated. This happens because when a process dies, all objects that it references via its file table are dereferenced. Also, when a thread changes its resource binding, a reference to the resource container that the thread was bound to before the binding change operation is destroyed.

As described in Section 4.6, once there are no file descriptors pointing to a container, and no threads with resource bindings to a container, its reference count falls to zero, and it is destroyed. When this happens, the parent pointers of all the child containers of this resource containers are set to denote "no parent." Also, if the propagate_stats flag is set appropriately, the container's resource usage statistics are propagated to its parent.

3. **Set a container's parent:** Using the set_rc_parent() system call, a process can change a container's position in the resource container hierarchy. This operation requires owner access rights to the container in question, as mentioned in Section 4.7. An application can also disconnect a container from the rest of the resource container hierarchy by specifying a parent container descriptor of -1 in a call to set_rc_parent().

   In handling a call to set_rc_parent(), the parent and child pointers of the affected containers are modified, and the scheduling parameters of these containers are made consistent. (See Section 8.4 below for a more detailed discussion of these parameters.) The kernel tries to conserve any relative CPU share of the container whose parent is being changed vis-a-vis its old parent's CPU entitlement. If the kernel cannot provide the same relative share of its parent's resources to the container in the new context, the best possible share is provided.

   A disconnected container, i.e., a container with a NULL parent pointer other than the system's root container, is not part of the system's scheduling hierarchy. Thus, this container gets no resources in the operation of the system's schedulers. A disconnected container can, however, be incorporated into the resource container hierarchy by a call to set_rc_parent().

4. **Transfer of containers between processes:** The sendmsg() and recvmsg() system calls allow resource containers to be passed between processes via a

UNIX domain socket. This facility is implemented in a manner that is similar to the transfer of file and socket descriptors in Digital UNIX. A new file object, which points to the container object being transferred, is created in the file table of the process that calls recvmsg(). A newly allocated file descriptor corresponding to this file table entry is returned in the recvmsg() call.

5. **Manipulating container attributes:** An application can set the attributes of a resource container by using the set_rc_opt() system call. Similarly, it can read the attributes of a container by using the get_rc_opt() system call. These system calls manipulate the following fields of a resource container object: rc_cpu_limit, rc_cpu_reserve, rc_cpu_pri, rc_mem_user_pages, rc_max_sockbufspace rc_rights_inherit, and propagate_stats.

6. **Container usage information:** An application can obtain the resource usage information of a container via the get_rc_usage() system call. This transfers summary data from the sched_info structure of a container object to the user application. A related system call get_rc_stats() allows an application to obtain the contents of the stats field of a container object.

7. **Resource-binding operations:** A thread can use the set_res_binding() system call to change the container that is charged for its subsequent resource consumption. Operationally, this changes a pointer in the thread's control block to point to the specified container. As mentioned above, this pointer reference increases the reference count field of the specified container. The reference count field of the container that was the thread's resource binding before this call to set_res_binding(), is accordingly decremented.

   As part of this system call, the kernel also performs a priority check to verify if the calling thread should continue to run. It may be that the newly resource-bound container does not have the highest priority of all eligible runnable containers. This situation causes the running thread to be preempted. The CPU is then given to another container that has sufficient priority.

   The kernel takes a call to set_res_binding() to mean that the currently running activity (corresponding to the currently active resource container) cannot make any further progress before an external event (such as the receipt of a network packet) happens. Thus, the CPU scheduler subsequently does not consider this yielding container as "runnable" until this container receives an event (see

Section 8.4). However, a thread may sometimes wish to change its resource binding even though the current activity is still to be considered "runnable". For this reason, the kernel provides another system call, change_res_binding(), which is operationally identical to set_res_binding() except that it has no effect on the scheduling state of the yielding container.

An application can also obtain a descriptor that points to the current resource binding of a thread by performing the get_rc() system call. This call creates a new file object which points to the resource container that the thread is resource-bound to. Subsequently, a file descriptor corresponding to this newly created file object is returned to user space.

8. **Scheduler-binding operations:** An application thread can add itself to the scheduler binding set of a resource container by using the add_to_sched_binding() system call. Similarly, it can delete itself from a container's scheduler binding set by using the delete_from_sched_binding() call. These calls affect the sched_binding field of a container object. As a result of a add_to_sched_binding() operation, a thread becomes eligible to receive CPU quanta awarded to the concerned container. My current implementation considers all threads that are scheduler-bound to a container to have equal priority. A more sophisticated implementation may support relative weights for these threads.

9. **Binding a socket or file to a container:** A process can change the ownership of a socket or file to any container by using the set_fd_rc() system call. Subsequently, any kernel resource consumption on behalf of this kernel object is charged to this container. Moreover, an event on this object is now considered as an event for this container, and may change its scheduler state (see below).

Detailed man pages for the system calls mentioned above are presented in Appendix A.

8.4 Changes to the CPU scheduler

I modified DUNIX's CPU scheduler to treat resource containers as its resource principals and implement the hierarchical decay-usage scheduling algorithm described in Section 6.3. A resource container can obtain a fixed-share guarantee from the scheduler (within the CPU usage restrictions of its parent container), or can choose to time-share the CPU resources granted to its parent container with its sibling containers. Only containers with fixed-share guarantees can have child containers with fixed-share guarantees.

To implement the hierarchical scheduling algorithm, the scheduler maintains a sched_info structure as part of each resource container object. This structure is shown in Figure 8.2. The structure includes fields for keeping track of the CPU time consumed by the container, parameters for the priority computation described in Section 6.3, the computed container priority, information for scheduling the container hierarchy rooted at the container, and some information used by the second-level thread scheduler.

As described in Section 6.3, the scheduler operates in a hierarchical manner. At each level of the resource container hierarchy, the scheduler awards a CPU quantum to the highest priority top-level runnable resource container subtree, S_h, that is rooted at container C_h. This quantum is further allocated to a container within S_h as follows: First, C_h and its children, $G_1 ... G_k$, which have a CPU share guarantee, compete for this quantum. This intra-node scheduling operation considers the normal priorities of $G_1 ... G_k$, and the adjusted priority of C_h, adj_pri. (C_h's normal priority (sched_pri) corresponds to the complete subtree S_h. C_h's adjusted priority corresponds to just the node C_h.) If the quantum is not used up fully by the container (or container subtree) to which it is allocated, another intra-node scheduling operation is performed to allocate the residual.

If a quantum is allocated to C_h, then if C_h has unblocked threads associated with it, the quantum is dispersed between these threads by the C_h's second level scheduler. Otherwise, or if these threads do not consume the complete quantum before blocking, the timeshare children container subtrees of C_h, $T_1 ... T_j$ are considered for being awarded the quantum (or what remains of it). For this, a runnable child subtree of C_h, S_{h2}, that has the highest (normal) priority at this level is picked. The scheduler operation continues down the resource container hierarchy recursively.

The above discussion assumed the notion of an unblocked thread associated with a container. Formally, a container has an unblocked thread if there exists a thread that:

1. is present in the scheduler binding set of the container

2. is runnable

```
struct sched_info {
    /* timing data for the container */
    timer_data_t  user_timer;
    timer_save_data_t user_timer_save;
    timer_data_t  system_timer;
    timer_save_data_t system_timer_save;

    /* raw and weighted cpu usage for subtree */
    unsigned int   cpu_delta;
    unsigned long  sched_delta;

    /* scheduler operation parameters for subtree */
    unsigned int  weight_divisor;
    unsigned int  weight_multiplier;
    int  base_priority;
    unsigned int  max_priority;  /* maximum priority */
    unsigned int  sched_pri;    /* normal priority */

    /* recursive scheduling info */
    unsigned int  adj_cpu_delta;
    unsigned long adj_sched_delta;
    unsigned int  adj_weight_divisor;
    unsigned int  adj_weight_multiplier;
    int  adj_base_priority;
    unsigned int  adj_max_priority;
    unsigned int  adj_pri;    /* adjusted priority */
    unsigned int sigma_w_children;
    struct subtree_scheduling_info *ssi; /* scheduling
                                           queues, etc. */

    /* second-level scheduler data */
    thread_t last_thread;
};
```

Figure 8.2 The sched_info structure.

3. either has a event pending for it, or has not yielded the CPU by calling set_res_binding() since it was last resource-bound to this container.

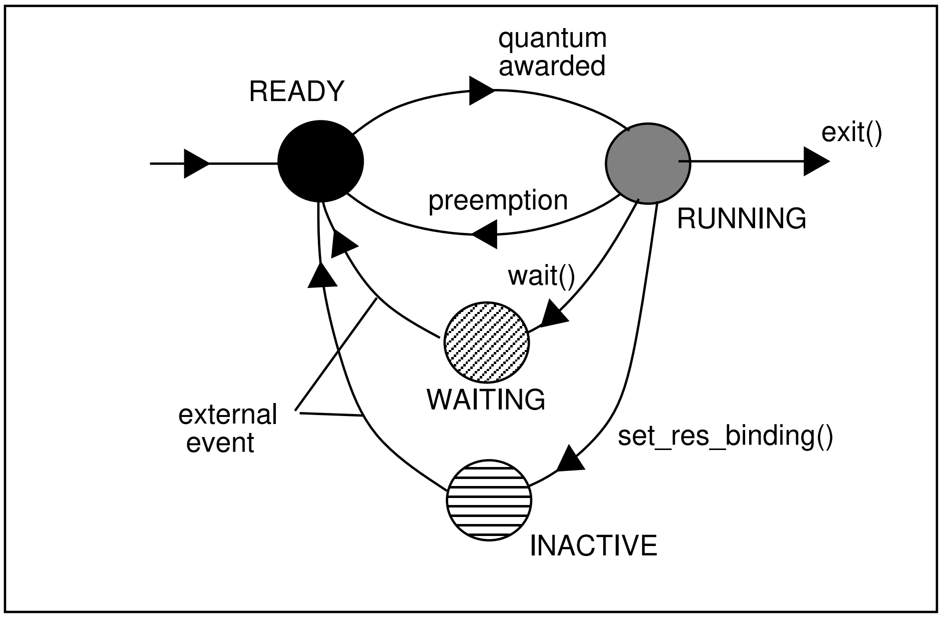

Figure 8.3 Resource container scheduling states.

The first two of these conditions should be obvious. To understand the third condition, let us consider the scheduling state transition graph of a resource container, shown diagrammatically in Figure 8.3. Notice the presence of a new state, INACTIVE, which is not present in classical scheduling state diagrams. When a thread that is resource-bound to a particular resource container changes its resource binding, it effectively performs a yield operation for the container and, if there are no other threads resource-bound to it, moves it to the INACTIVE state. Subsequently, when an event happens on a resource, such as a socket, that is owned by this container, the container's state is changed to READY. The last condition in the enumerated list above excludes runnable threads in the scheduler binding set of an INACTIVE container, i.e., one that is waiting for an event, from being considered as unblocked.

As described in Section 6.3, the priority (sched_pri) of the container subtree, S, rooted at resource container C is computed by decaying the recent CPU usage of all the containers in S, and adjusting by a base priority. The weight that is used to decay CPU usage at each level, i.e. the fraction weight_multiplier/weight_divisor, is configured based on the fraction of the CPU resources of the parent of C that the container subtree S is entitled to. To begin with, these values are so chosen such that weight_divisor/weight_multiplier times 10 is the percentage value of CPU cycles that the container subtree is entitled to. As the operation of the system proceeds, the weight value of some container subtree, S, may get so large that the S's priority value reaches its maximum value before the end of a decay cycle. In this case, the weights of all container subtrees that are children of S's parent P are normalized by scaling by a constant value. This operation is performed by a trigger function that is invoked when the above condition is noticed.

The adjusted priority of container C, which is used in the intra-node scheduling algorithm described above, is computed by decaying the CPU usage of C alone by the adjusted weight fraction adj_weight_multiplier/adj_weight_divisor. This adjusted fraction reflects the residual CPU share of C and its timeshare children after any CPU-share guarantees of the child subtrees of C have been accounted for. If the weights of the children of C are normalized, the adjusted weight of C is also normalized.

The CPU share of a timeshare child of C, T_i, is fraction $w_i/(\sum_1^n w_i)$ of the residual allocation of C. Here, $w_1 ... w_n$ are the relatives weights of the timeshare children of C, $T_1 ... T_n$. This corresponds to a weight fraction, $(\sum_1^n w_i)/w_i$ for the subtree T_i. For efficiency of computation, $\sum_1^n w_i$ is stored in the kernel object C as sigma_w_children. This allows a rapid update to this value when a timeshare child of C is created or destroyed.

Second-level thread scheduling within a container is performed as follows: The second-level scheduler of a container gives the CPU to the various runnable threads in the scheduler binding set of the container in a prioritized round-robin fashion. Each thread has a priority, which is decayed according to recent CPU usage in a manner similar to the regular Digital UNIX's vanilla thread scheduling algorithm. If a container, and hence a thread associated with it, was preempted at the end of the last CPU quantum awarded to the container, preference is given to the preempted thread when re-starting the round-robin operation at the beginning of the next quantum.

8.5 Changes to the network subsystem

I changed Digital UNIX's TCP/IP subsystem to implement LRP-style network processing, treating resource containers as resource principals. Specifically, a software based early demultiplexer to associate Ethernet frames with their destination sockets was implemented. The software interrupt for performing protocol processing was replaced by per-process dedicated threads which perform packet processing lazily[§]. I also changed the kernel allocators for PCBs, sockets and socket buffer space to respect per-container limits on the usage of these resources. The implementation of early demultiplexing and lazy processing is discussed further below.

8.5.1 Early demultiplexing

To implement early demultiplexing, the single IP queue of Section 2.2 was replaced with per-socket NI channels. The kernel data structure corresponding to a NI channel is shown in Figure 8.4. This structure includes a pointer to the corresponding protocol control block, a receive queue that holds unprocessed IP packets, a lock that helps in maintaining data consistency, and a pointer to the owning container.

```
struct ni_channel {
    struct inpcb *inp;
    struct ifqueue receive_queue;
    simple_lock_data_t ni_channel_lock;
    struct resource_container *owner;
};
```

Figure 8.4 A NI channel.

The network interface's interrupt handler demultiplexes incoming packets according to their incoming socket, and places the packet directly on the appropriate NI channel's receive queue. Packets destined for a socket with a full receiver queue are silently discarded (early packet discard).

The operation of the demultiplexing function is as follows: First, a packet is demultiplexed based on its *protocol* field, i.e., TCP packets are separated from UDP packets. Subsequently, a different algorithm for each protocol demultiplexes packets to the correct socket. For UDP, a simple hash table lookup that uses the *<local-address, local-port>* tuple as its key is used.

[§]The issue of why per-process threads are used instead of per-RC threads is discussed later in this section.

For TCP, however, the algorithm is a little more involved. TCP packets on established connections are demultiplexed based on exact matches of their <*local-address, local-port, foreign-address, foreign-port*> tuples with the corresponding parameters of the sockets of these connections. On the other hand, TCP packets for half-open connections, such as SYN packets destined for a listen socket, are demultiplexed based on partial matches with the <*local-address, local-port, client-addr, CIDR-mask*> tuples of listen sockets (see Section 8.6.2 below). To implement this, the demultiplexer uses two hash tables, one for established connections and one for listen sockets. When a packet arrives, the established connections are searched first using the complete <*local-address, local-port, foreign-address, foreign-port*> tuple as a key. If a match is not found, the other hash table is also searched. In this case, the <*local-address, local-port*> sub-tuple is used as a key. The other, <*client-addr, CIDR-mask*> sub-tuple is partially matched by using a full, second-level search in the appropriate hash bucket.

Besides queueing a demultiplexed packet to the appropriate socket, the demultiplexer queues a hint about the identity of this socket on a per-process queue. It also wakes up a per-process network processing thread (see below). Subsequent processing for the packet is performed by this thread.

This demultiplexing function can efficiently demultiplex all packets in the TCP/IP protocol family, including IP fragments. In rare cases, an IP fragment does not contain enough information to allow demultiplexing to the correct endpoint. This happens when the fragment containing the transport header of a fragmented IP packet does not arrive first. In this case, the offending packet is placed on a special receive queue reserved for this purpose. The IP reassembly function checks this receive queue when it misses fragments during reassembly.

My demultiplexing function is also self-contained, and has minimal requirements on its execution environment (non-blocking, no dynamic memory allocation, no timers). As such, it can be be readily integrated in a smart network interface's firmware to implement network interface based demultiplexing [10, 52].

8.5.2 Lazy processing

I changed the Digital UNIX kernel to use a per-process kernel thread to perform network processing. When it creates a new process, the modified DUNIX kernel also creates a new thread to perform protocol processing for this process. The lifetime of this thread is the same as that of the process. This thread executes at the priority of its associated container, as is discussed below.

The network processing thread of each process waits for events to be signaled on a private per-process synchronization object. As described above, these synchronization objects are signaled by the early demultiplexer when a packet arrives for a socket referenced by the process. After being woken up, the network processing thread dequeues the hint posted by the demultiplexer about the identity of the socket for which the new packet has arrived. Subsequently, the thread changes its resource binding to the resource container that this receiving socket belongs to: this ensures correct accounting. It then dequeues the unprocessed Ethernet frame from the socket's NI channel and performs protocol processing for the frame through IP and TCP. This processing may also result in any of the user-level threads of this application being woken up.

The network thread synchronizes with the demultiplexer and the other regular threads of the process using the same synchronization primitives that are used by the vanilla Digital UNIX network processing kernel thread. Thus, this new implementation of network processing does not increase synchronization overhead.

It may seem that the use of a dedicated per-process thread for protocol processing introduces the overhead of context-switching between the regular thread(s) of a process and this network processing thread. In addition, there is a potential for an increase in the cache and TLB pressure due to the stack of the network processing thread. This may be especially important for single-threaded programs whose thread management overhead is doubled in the new system. In this prototype implementation, however, I have observed this overhead to be so small as to have no observable impact on system performance.

For a packet destined to a socket shared between multiple processes, the use of a kernel thread belonging to one of the processes to perform protocol processing may seem somewhat arbitrary. Note that system resources consumed during this processing are charged correctly to the appropriate resource container. However, it may be that successive packets for a shared socket are processed by a number of different threads that belong to the various processes that hold a reference to the socket, while the data is always consumed by a single process. This may increase kernel-thread switching overhead unnecessarily. The use of per-RC threads instead of per-process threads for protocol processing also suffers from the same disadvantage. In a more sophisticated implementation of LRP, this may be addressed by allowing an application to specify exactly which process' kernel thread is responsible for performing protocol processing for a shared socket. Since shared sockets are the exception, rather than the rule, the current implementation does not include this enhanced level-of-control over protocol processing of shared sockets.

As with vanilla Digital UNIX, on an SMP machine, multiple network processing threads are required to utilize the parallelism available in the hardware. The regular DUNIX kernel uses as many network processing threads as there are processors in the SMP machine. The current prototype uses as many threads on a per-process basis as there are processors in the system.

8.6 New APIs

I will now briefly describe the implementation of the new event API and the new bind() name space introduced in Section 4.10.

8.6.1 The new event API

The new event API of Section 4.10.1 was implemented by adding the three new system calls described in Table 4.2 and associated support routines to the Digital UNIX kernel. The following changes were made to the kernel: The kernel maintains an INTERESTED set and a SIGNALED_EVENTS set for each thread of a process. When a thread calls declare_interest(), the kernel adds a tuple consisting of the socket's descriptor and the event type the application is interested in to the thread's INTERESTED set. This tuple is removed when the thread calls revoke_interest().

Currently, the implementation of the event API supports only network sockets. The declare_interest()/revoke_interest() API uses the event_interest structure shown in Figure 8.5. The event_t structure has a subsystem specific implementation.

```
struct event_interest {
    int fd;          /* file descriptor */
    event_t event; /* event types */
};
```

Figure 8.5 The event_interest structure.

The current implementation supports three types of events, SOCKET_READABLE, SOCKET_WRITABLE and SOCKET_EXCEPTION. These correspond to the three socket event types that are supported by select(). For these events, the event_t structure needs to contain just the event type information. A more sophisticated implementation can easily support other events such as DISK_READ_COMPLETE, DISK_WRITE_COMPLETE, DISK_RANGE_READ_COMPLETE, etc. For these events, the event_t structure will also contain some other information, e.g., the range specifier for a disk read, besides the usual event type identity.

The kernel maintains the INTERESTED set of each thread as three bitmaps, one to signify application interest in reading from a socket, one for signifying application interest in writing to a socket and the last to signify interest in exception events on a socket. The SIGNALED_EVENTS set of each thread is implemented as a heap based priority queue. This allows the efficient prioritization of event delivery to the application via the get_next_event() system call. The protocol processing code performs a sorted insert into this queue when it posts an event on a socket. The application thread calling get_next_event() performs a sorted delete from this queue.

8.6.2 bind()

I will now describe the implementation of the new sockaddr_in_filter name space for the bind() system call, which was described in Section 4.10.2. As discussed earlier, the sockaddr_in_filter name space allows the specification of the address of a listen socket as a <*local-address, local-port*> tuple in conjunction with a filter that can specify the set of clients whose requests should be directed to this socket.

In this implementation, such filters are specified as tuples consisting of a template address and a CIDR network mask [138]. The data structures of the new sockaddr_in_filter name space and the filters is shown in Figure 8.6. The data structure representing an Internet protocol control block (PCB) object was modified to include a in_addr_filter object as part of the PCB's address.

Since the bind() implementation transparently passes a socket address through sobind() and a protocol specific bind operation all the way to in_pcbbind(), this was the only function that had to be change in the implementation of the new version of the system call. I changed in_pcbbind() to understand addresses of the type sockaddr_in_filter, and to fill in the address field of the PCB appropriately. An early switch identifies addresses of the type sockaddr_in and falls over to the original in_pcbbind()

```
struct sockaddr_in_filter {
    u_char  sin_len;
    u_char  sin_family;
    u_short sin_port;
    struct  in_addr sin_addr;
    struct  in_addr_filter filter;
    char    sin_zero[8];
};


struct in_addr_filter {
    u_int32 f_addr;
    u_int32 f_mask;
};
```

Figure 8.6 The sockaddr_in_filter and in_addr_filter structures.

code. Thus, my bind() implementation transparently supports old style socket addresses also.

As described in Section 8.5.1, the early demultiplexer classifies TCP SYN packets to the correct listen socket based on a partial match with the client address field of a packet and the <*template-address, CIDR-mask*> tuples associated with each listen socket. Multiple matches are broken on the length of the matching CIDR mask field. Conflicting CIDR equal-length CIDR masks are excluded in in_pcbbind() by design. The in_pcblookup() function was also changed to understand the sockaddr_in_filter name space.

8.7 Changes to system tools and libraries

To make the new system usable, a number of tools and libraries were modified to understand resource containers. Specifically, some system monitoring tools, i.e., ps and top were modified to understand resource containers. The ps command was modified to optionally print the resource containers that each process had a reference to in its descriptor table, and to print the container that each thread of a process was resource-bound to (with the -m option).

The top command was changed to optionally print information about the system's top resource containers instead of the top processes. The key criterion can be chosen from a variety of container parameters, such as recent CPU usage, allocated memory pages and allocated socket buffer space. In a situation where no application in the system is container-aware, i.e., with only unmodified applications running, this listing is simply the regular top listing with the addresses of each process' resource container listed instead of their process ids.

I also implemented a new tool, rsched, to control the scheduling of resource containers. rsched is somewhat similar to the existing nice and renice commands. It allows a user with appropriate privileges to change the scheduling parameters of a resource container. A detailed man page is presented in Appendix A.

A resource consumption control system operating on a per-user basis was also implemented. This system is similar to what was described in Section 7.8. It is based on the idea of using a top-level resource container to control the total CPU utilization of all the processes of a user. Every resource container created by an application belonging to a user is created as some descendent of the user's top-level container in the resource container hierarchy.

This control system consists of a urcd daemon process, which is responsible for keeping track of the top-level resource container of each user, and a modified login program. The changed login program essentially queries the urcd daemon to obtain a reference to the top-level resource container of the user logging in. (urcd creates a new resource container for this user if the user has not logged in recently.) A configuration file, /etc/urcdtab, allows the system administrator to specify priority policies on a per-user basis.

Two new commands, rcstat and useracc, was also implemented as part of this resource control system. The rcstat command prints the resource container hierarchy on a per-user or a system-wide basis. It can also optionally print details about a specific container. This command uses the knlist and the kmem interfaces provided by the Digital UNIX kernel to explore the resource container hierarchy starting at the system's top-level container.

The useracc command allows the recent resource usage of a particular user's processes to be displayed. This information is tracked by the urcd daemon process at a very fine level of granularity. Detailed resource usage information about a user in terms of, for example, total CPU usage, total memory usage, average memory per-process, total processes spawned, total system calls, etc. can be obtained. The urcd daemon obtains this information from the statistics fields of the resource container and process kernel objects. A statistics file, /etc/user_stats is used by urcd to preserve this information across system reboots and crashes of urcd. Detailed man pages for these new commands are presented in Appendix A.

The standard C library, libc, was also modified to provide assembly language stubs for the new system calls described in this dissertation.

Chapter 9

Performance

I performed several experiments to evaluate the performance of my prototype implementation of resource containers. The purpose of these experiments was to verify that resource containers are indeed an effective way for a server application to control resource consumption and provide robust and controlled service. I also evaluated the performance of the new event API described in Section 4.10.1, and quantitatively characterized the effect of the LRP network subsystem architecture on the dynamics of TCP. This chapter describes these experiments and their results in some detail.

This chapter begins with a brief description of the experimental setup that was used for the experiments. Subsequently, it describes the actual experiments that were performed.

9.1 Experimental environment

In all experiments, the server system was a Digital Personal Workstation 500au (500Mhz 21164, 8KB I-cache, 8KB D-cache, 96KB level 2 unified cache, 2MB level 3 unified cache, SPECint95 = 12.3, 128MB of RAM), running Digital UNIX 4.0D. (Most experiments were conducted first with vanilla Digital UNIX, and then repeated with the prototype resource container (RC) system.) The client systems were 166MHz Pentium Pro PCs, with 64MB of memory, and running FreeBSD 2.2.5. The client machines were connected to the server via a private 100Mbps switched Fast Ethernet.

The server software used in most experiments was a single-process event-driven program derived from `thttpd` [161]. I started from a modified version of `thttpd` with numerous performance improvements, and changed it to optionally use the new APIs proposed in this dissertation. In some experiments the Apache server [8] was used. The client machines used the S-Client software [12].

9.2 Experimental results

To begin with, we consider two control experiments whose purpose is to quantify the base performance of the server system in the experimental testbed described above, and to show that resource containers do not add significant overhead to the operation of the server.

Subsequently, some experiments are presented, which demonstrate how the use of resource containers by a server allows it to have stable overload behavior and control resource consumption throughout the server system. After this, we quantitatively evaluate the scalability of the new event API. Finally, a set of experiments are described that aim at characterizing the effect of LRP on TCP's performance.

9.2.1 Baseline throughput

The first experiment aims at quantifying the baseline performance of the experimental server system. I measured the throughput of my HTTP server running on the unmodified kernel. When handling requests for small files (1 KByte) that were always present in the file system cache, the server achieves a rate of 2954 requests/second using vanilla connection-per-request HTTP (i.e., with a new TCP connection being used for every request). When using persistent-connection HTTP, the server achieves a service rate of 9487 requests/second. Both of these measurements were made when the server CPU is saturated and increasing the request rate did not increase the server's HTTP transaction rate. Therefore, these measurements correspond to per-request CPU costs of $338\mu s$ and $105\mu s$, respectively.

Using Digital's Continuous Profiling Infrastructure [7], it was measured that the system spends 87% of its time in kernel mode in the connection-per-request HTTP case above. In the persistent-connection case, the system spends 88.5% of its time in kernel mode.

9.2.2 Costs of new primitives

Next, the costs of primitive operations on resource containers were measured. For each new primitive, a user-level program invoked the corresponding system call 10,000 times, measured the total elapsed time, and divided to obtain a mean "warm-cache" cost. The results, in Table 9.1, show that all such operations have costs much smaller than that of a single HTTP transaction. This implies that the use of resource containers should add negligible overhead.

Operation	Cost (μs)
create resource container	2.36
destroy resource container	2.10
change thread's resource binding	1.04
add thread to container's scheduler binding	1.03
delete thread from container's scheduler binding	1.91
obtain container resource usage	2.04
set/get container attributes	2.10
move container between processes	3.15
obtain handle for existing container	1.90
set the owner of a socket to container	1.10

Table 9.1 Cost of resource container primitives.

This was verified by measuring the throughput of the HTTP server running on the modified kernel. In this test, the Web server process created a new resource container for each HTTP request. The throughput of the system remained effectively unchanged.

9.2.3 Prioritized handling of clients

My next experiment tested the effectiveness of resource containers in enabling prioritized handling of clients by a Web server. In the target scenario, the server's administrator wanted to differentiate between two classes of clients (for example, based on payment tariffs).

This experiment used an increasing number of low-priority clients to saturate a server, while a single high-priority client made requests of the server. All requests were for the same (static) 1KB file, with one request per connection. The response time perceived by the high-priority client was measured.

Figure 9.1 shows the results. The y-axis shows the response time seen by the high-priority client (T_{high}). The dotted curve shows how (T_{high}) varies when using the unmodified kernel. This curve was also shown in Figure 3.5. As described in Section 3.2.3, the server application attempts to give preference to requests from the high-priority client by handling events on its socket, which are returned by `select()`, before events on other sockets. The figure shows that, despite this preferential treatment, (T_{high}) increases sharply when there are enough low-priority clients to begin

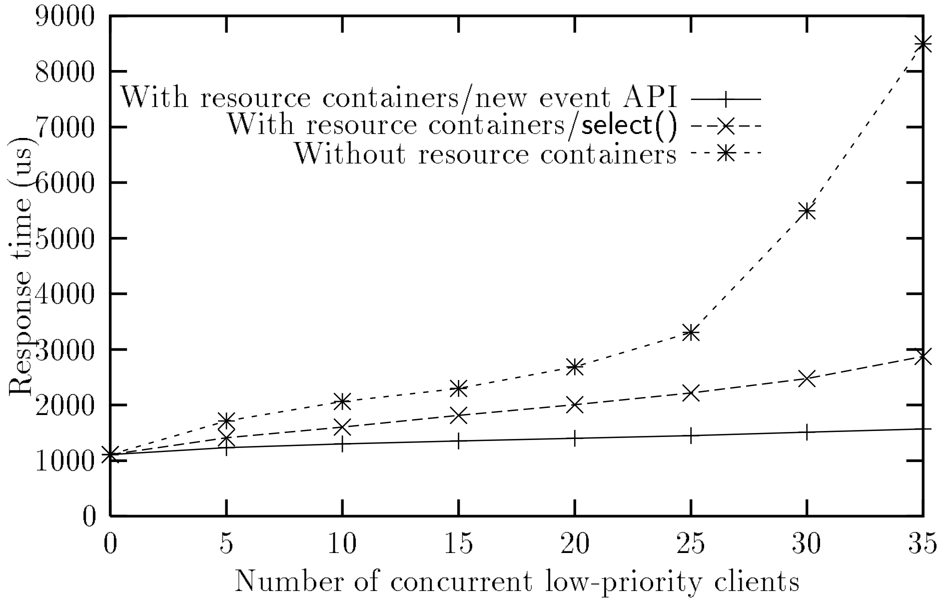

Figure 9.1 How T_{high} varies with load.

to saturate the server. This happens because most of the request processing occurs inside the kernel, and so is uncontrolled.

The dashed and the solid curves in Figure 9.1 show the effect of using resource containers. Here, the server uses two containers, with different priorities, assigning the high-priority requests to one container, and the low-priority requests to another. The solid curve, labeled "With resource container/new event API", shows the variation in T_{high} when the server uses the new event API to wait for events. In this case, T_{high} increases only very slightly as the number of low-priority clients increases: resource containers, in conjunction with the new container aware and scalable event API, succeed in allowing the application to control resource consumption at almost all levels of the system. The remaining slight increase in T_{high} comes from the cost of packet-arrival interrupts from low-priority connections (since the system cannot defer low-priority packets before examining their headers).

The dashed curve, labeled "With resource container/select()", shows the effect of using select() instead of the new event API to wait for events. T_{high} increases at a slightly higher rate than with the new API. The reason for this increase is the inherent priority inversion introduced by the poorly scaling select() interface. Note that this increase is likely to be more significant in a real-world server than is shown in Figure 9.1 because of the much larger number of concurrent connections that are likely to be active at the real-world server.

9.2.4 Controlling resource usage of CGI processing

Chapter 2 described how requests for dynamic resources are typically handled by processes other than the main Web server process. In a system that timeshares the CPU equally between processes, these back-end (CGI) processes may gain an excessive share of the CPU, which reduces the throughput for static documents. Each CGI request essentially presents a different resource principal to the OS, and this decreases the CPU share available to the main server process, as was discussed in Section 3.2.3. I constructed an experiment to show how a server can use resource containers to explicitly control the CPU consumption of CGI processes.

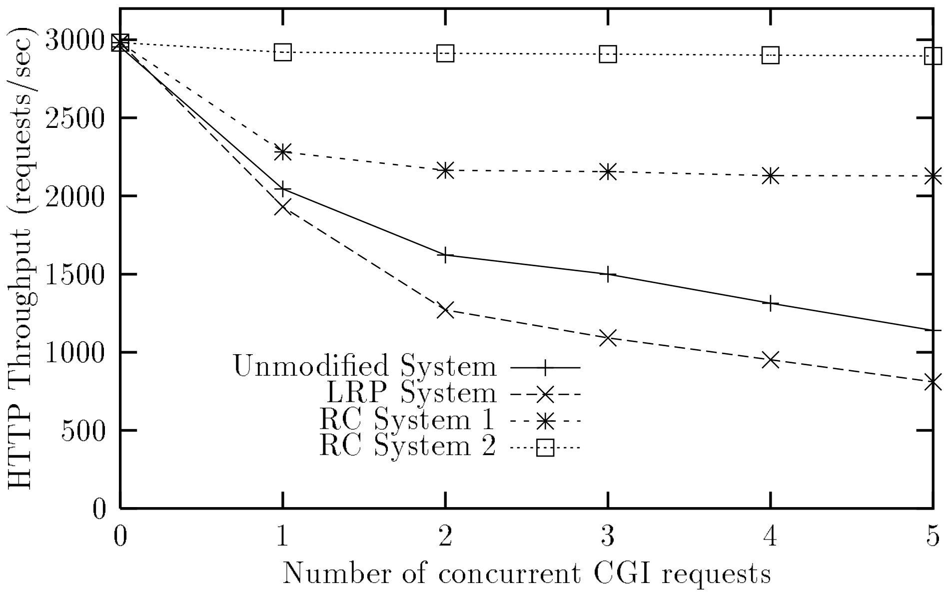

Figure 9.2 Throughput with competing CGI requests.

In this experiment, I measured the throughput of the Web server (for cached, 1 KB static documents) while increasing the number of concurrent requests for a dynamic (CGI) resource. Each CGI request process consumed about 2 seconds of CPU time.

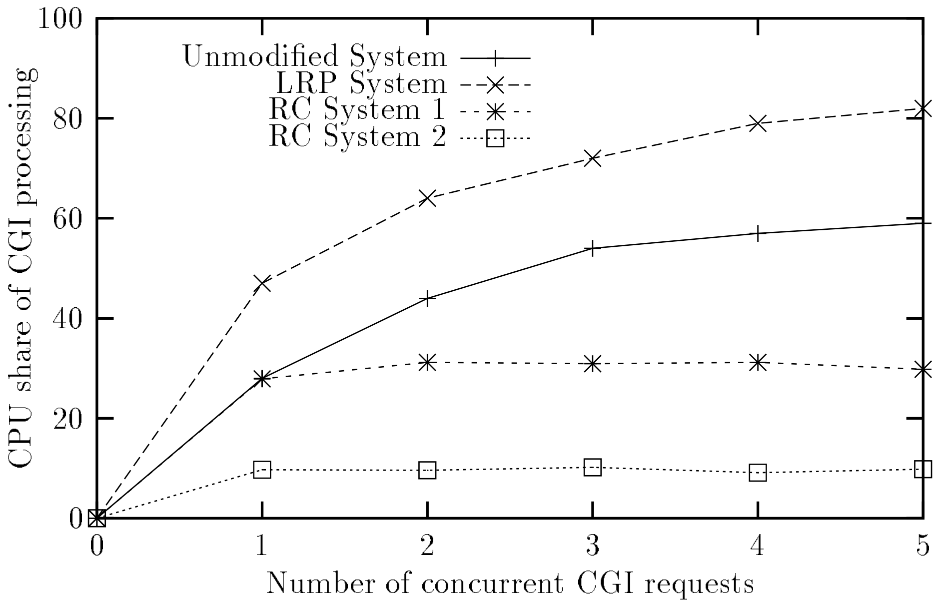

Figure 9.3 CPU share of CGI requests.

The results are shown in the curve labeled "Unmodified System" in Figure 9.2. This curve was also shown in Figure 3.7.

As we saw in Section 3.2.3, with an increase in the number of concurrent CGI requests, the server's CPU is shared among a larger set of processes, and the main Web server process' share decreases; this sharply reduces the throughput for static documents. For example, with only 4 concurrent CGI requests, the main Web server process itself gets only 40% of the CPU, and the static-request throughput drops to 44% of its maximum.

Additionally, as discussed earlier, the main server process actually gets slightly more of the CPU than each CGI process, because of misaccounting for network processing. This result is shown in Figure 9.3, which plots the total CPU time used by all CGI processes.

In Figures 9.2 and 9.3, the curves labeled "LRP System" show the performance of a LRP version of Digital UNIX, which still considers the process as the system's resource principal. LRP fixes the misaccounting, so the main server process shares the CPU equally with other processes. This further reduces the throughput for static documents.

To measure how well resource containers allow fine-grained control over CGI processes, the HTTP server was modified so that each container created for a CGI request was the child of a specific "CGI-parent" container. This CGI-parent container was restricted to a maximum fraction of the CPU (recall that this restriction includes resources consumed by its children). In Figures 9.2 and 9.3, the curves labeled "RC System 1" show the performance when the CGI-parent container was limited to 30% of the CPU; the curves labeled "RC System 2" correspond to a limit of 10%.

Figure 9.3 shows that the desired CPU usage limits are enforced almost exactly. Figure 9.2 shows that the CGI-parent container effectively forms a "resource sand-box" around the CGI processes, and so the throughput of static requests remains almost constant as the number of concurrent CGI requests increases from 1 to 5. By resource sand-boxing CGI requests, a server can completely control the partitioning of system resources between requests for static documents and those for dynamic documents. A server administrator can also use this functionality to make the system immune to denial-of-service attacks that use an excess of concurrent CGI requests to monopolize the server system's resources.

Note that the Web server could additionally impose relative priorities among the CGI requests, by adjusting the resource limits on each corresponding child container.

9.2.5 Overload behavior

Next, I conducted an experiment to quantify the improvement in the behavior of a HTTP server that uses resource containers under severe overload. First, the overload behavior of the Apache [8] Web server was measured with the server running on vanilla Digital UNIX, and while running the resource container prototype system. As in Section 3.4, all requests were for the same 1KB file, which was always present in the main memory file cache of the server.

The results of this experiment are shown in Figure 9.4. The solid curve shows the performance of the server under the unmodified Digital UNIX system. This was repeated from Figure 3.9. As we can see, the throughput of the Web server drops sharply as the rate of HTTP requests increases beyond the saturation point of the server. The reason for this phenomenon is the eager processing of SYN packets in software interrupt context in the DUNIX system. This processing takes priority over the handling of already established TCP connections, and therefore an increasing

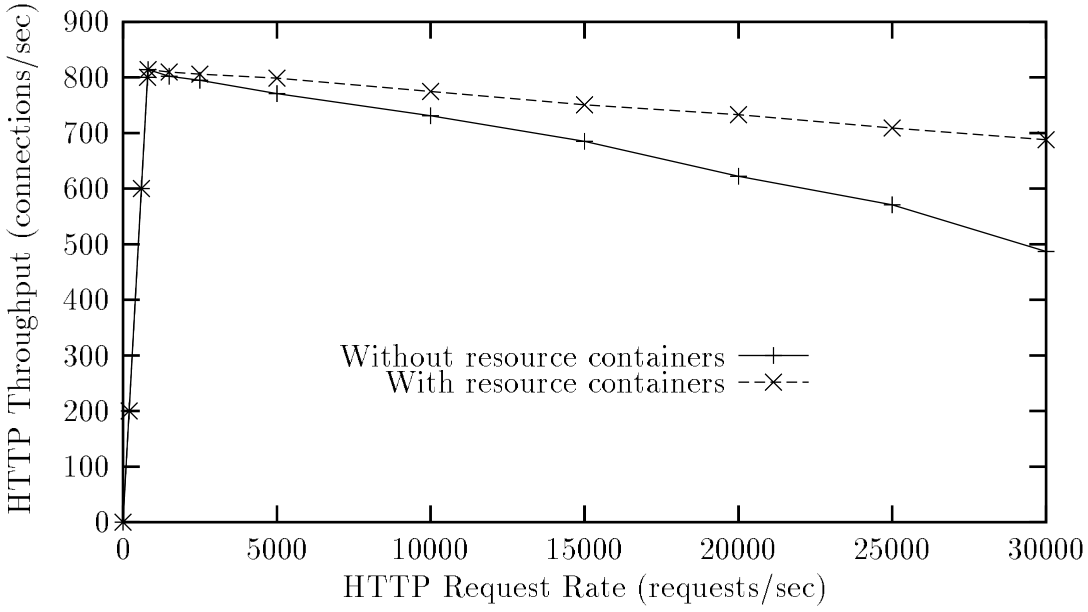

Figure 9.4 Overload performance of the Apache server.

rate of SYN packets decreases the number of CPU cycles available to the server for performing useful work.

The dashed curve shows the performance of the Web server running on the RC system. In this case the throughput of the server decreases at a much slower rate than with the original system. The reason for this is follows: Once the server is saturated, the HTTP processes cannot dequeue requests from the server's listen socket as fast as they arrive. Hence the receive queue in the NI channel corresponding to the listen socket starts to grow and eventually becomes full. Once this happens, most SYN packets that arrive are dropped by the early demultiplexer before significant system resources have been invested into processing them. SYN packets are processed through the TCP/IP code only as fast as the Apache processes can establish new connections. Thus, the throughput of the Web server remains relatively undegraded. The slight degradation that does occur happens because of the interrupt overhead of incoming SYN packets.

Figure 9.5 shows the overload performance of the modified `thttpd` server on vanilla Digital UNIX, and on the new system. As with Apache, the throughput of the server under overload falls significantly over the range of the curve (45%) under the

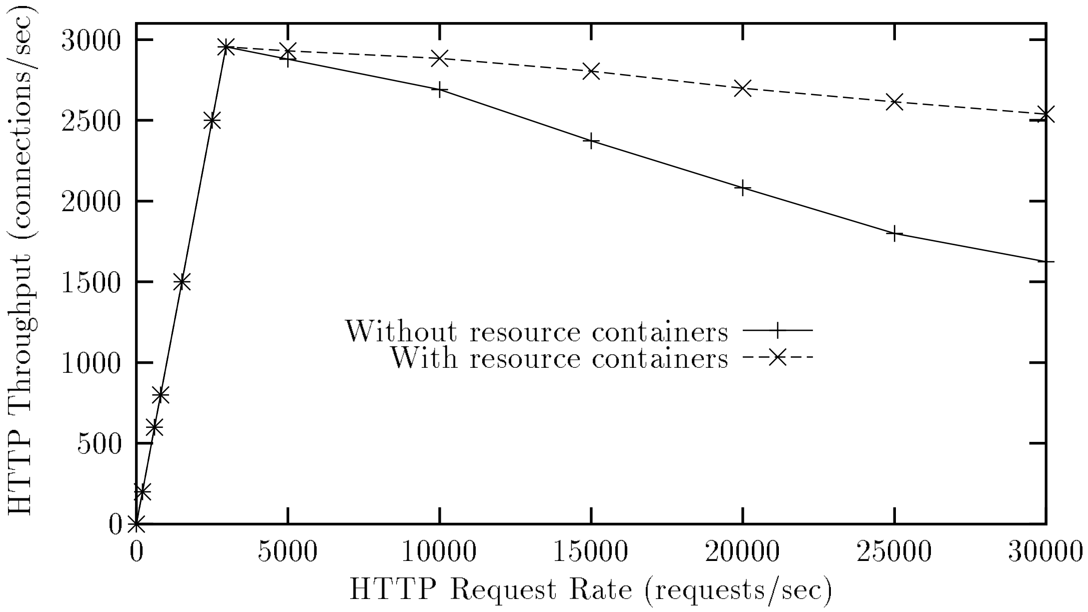

Figure 9.5 Overload performance of the modified `thttpd` server.

unmodified Digital UNIX system. In the new system there is only a small (14%) degradation over the range of the curve. The slight difference between the shape of the curves in Figure 9.4 and those in Figure 9.5 is because of the different process architecture of the two Web servers. As explained in Chapter 2, Apache is a process-per-connection server, while `thttpd` is a single-process server.

9.2.6 Immunity against SYN-flooding

Next, I constructed an experiment to verify that resource containers, combined with the filtering mechanism described in Section 4.10.2, allow a server to protect itself against denial-of-service attacks using SYN-flooding [143, 18, 42]. In this experiment, a set of "malicious" clients sent bogus SYN packets to the server's HTTP port, at a high rate. The server's throughput for requests from well-behaved clients (for a cached, 1 KB static document) was measured.

Figure 9.6 shows that the throughput of the unmodified system falls drastically as the SYN-flood rate increases. The throughput is effectively zero at about 10,000 SYNs/second. Note that even a slow T1 link is capable if carrying almost 5000

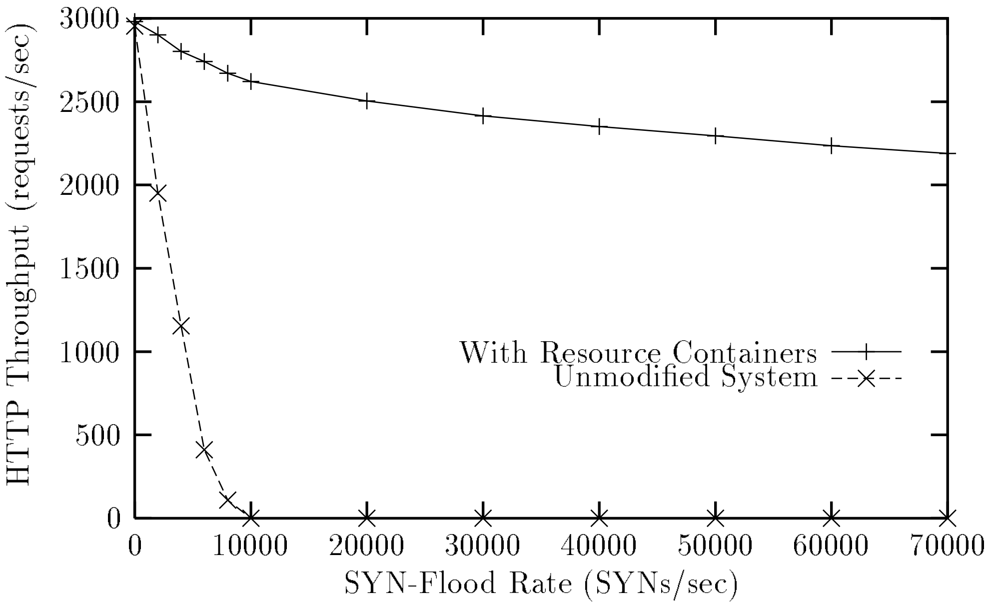

Figure 9.6 Server behavior under SYN-flooding attack.

SYNs/second. Thus, a SYN-flood can literally bring almost every server quickly to its knees.

I modified the kernel to notify the application, via a signal, when it drops a SYN (due to queue overflow). The HTTP server was also modified to obtain information about the identity of misbehaving clients from the kernel by using the get_listen_queue_info() system call described in Section 4.10.2. Additionally, some code was added to the HTTP server to allow it to isolate the misbehaving client(s) to a low-priority listen-socket, using the filter mechanism described in Section 4.10.2.

With these modifications, even at 70,000 SYNs/second, the useful throughput remains at about 73% of maximum. This remaining degradation results from the interrupt overhead of the SYN flood.

9.2.7 Isolation of virtual servers

Section 9.2.4 shows how resource containers allow "resource sand-boxes" to be put around CGI processes. This approach can be used in other applications, such as controlling the total resource usage of guest servers in a Rent-A-Server [164] environment. My next experiment attempted to demonstrate how resource containers can solve this problem.

In this experiment, the Rent-A-Server server application created 3 top-level containers and restricted their CPU consumption to fixed CPU shares, i.e., 10%, 25% and 65%. Each container was then used as the root resource container for a guest server. The first of these was an Apache Web server [8], the second was a Zeus server [174], and the third was a Squid proxy server [151]. The situation was similar to that described in Figure 7.10. None of the servers was aware of the resource container mechanism, i.e., I used binaries constructed from the original source code.

Subsequently, three sets of independent clients running on distinct sets of client machines placed varying request loads on these servers; the requests included CGI resources. The experiment was conducted for 15 minutes, and the total CPU time consumed by each server was determined.

Server	total CPU time (seconds)	% CPU
Apache	86	9.56
Zeus	235	26.11
Squid	579	64.33

Table 9.2 Performance of virtual guest servers.

The results of this experiment are shown in Table 9.2. As can be seen, the total CPU time consumed by each guest server almost exactly matched its allocation. Note, this result is in spite of the fact that these servers have different architectures (Apache has a process-per-connection architecture, while the other two servers are single-process event-driven servers), and none of the servers is aware of the resource container mechanism.

9.2.8 Performance of a per-user resource control system

I also conducted an informal experiment to verify the performance of the per-user resource consumption control system described in Section 8.7. A test DUNIX system was configured with four users, each of which was allocated a fixed percentage of the system's CPU resources. Each user then ran a mix of jobs, consisting of both compute-intensive as well as interactive applications. Things were so arranged that each user ran at least one job that consumed any available CPU cycles.

The total CPU resource usage of each user was measured at the end of a time period of 4 hours. As expected, each user got a total CPU share that was in proportion to the fraction allocated to it. Moreover, each user's interactive applications showed the same degree of responsiveness as they do on a regular DUNIX system.

9.2.9 Scalability of the new event API

The next experiment was designed to evaluate the scalability of the new event API proposed in Section 4.10.1. As noted in Section 3.7.1, the poor scalability of the select() mechanism interacts poorly with the large number of simultaneous connections induced at a Web server in WAN environments. Besides degrading throughput, poor scalability leads to indirect priority inversion. The aim of this experiment was to evaluate server scalability, and hence the ability to handle clients in a prioritized manner in real-world environments, directly.

In this experiment, a number of well-behaved clients made requests from the server for a cached 1KB document as fast as the server could serve them. It was ensured that sufficient concurrent clients were being used to saturate the server at all times. Following the method described in [15], the effect of large WAN delays was simulated as follows: A dummy HTTP client process was set up on a client machine. This process opened a large number (100-2000) of connections to the Squid server but subsequently made no requests on these connections. This process will be referred to as the *load-adding client*. The throughput of the server, as experienced by the well-behaved clients, was measured as a function of the total number slow connections being simulated by the load-adding client. The modified thttpd server was used in this experiment.

The results of this experiment are depicted in Figure 9.7. The various curves show the variation of server throughput with an increasing number of simultaneous clients for a Digital UNIX kernel with different implementations of select(), and with the new event API. To factor out the known scalability problems of the file descriptor allocation algorithm (ufalloc()) in UNIX, all systems used the scalable ufalloc() implementation [15] of Digital UNIX.

The curve labeled "with classical select()" shows the degradation of server performance with the classical implementation of select(). The server system suffers a 31% degradation over the range of the curve. The curve labeled "with scalable select()" shows the performance of the scalable select() implementation [15] of Digital

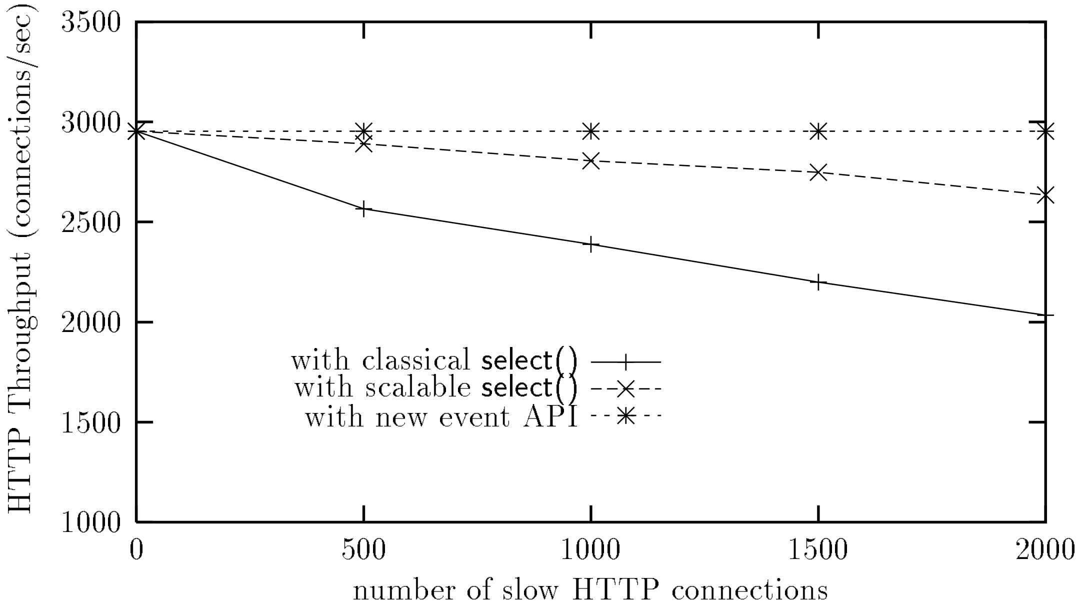

Figure 9.7 Scalability of the new event API.

UNIX. Although this system performs much better than the classical system, server throughput still degrades by about 11% over the range of the curve.

The curve labeled "with new event API" shows the performance of the system with the new event API. There is no observable effect of the number of simultaneously present connections on the throughput of the server. The system scales almost perfectly; this result indicates that the new event API does not cause any scalability-related priority inversion in the delivery of network events to the server application.

To verify the superior performance of the new event API over select() independently, I obtained DCPI [7] profiles of the original Digital UNIX system (with classical select()), and the system with the new event API. These profiles were obtained when 50 clients generated requests for the server at a controlled rate of 800 requests/second. The load-adding client simulated 750 slow connections. The profiles are shown in Table 9.3 and Table 9.4.

As can be seen in Table 9.3, 28.92% of the system's time is spent inside kernel functions that implement select(). Another 4.19% of the system's time is spent inside do_select(), which is the user-level function that calls select() and collates the infor-

CPU %	Procedure	Mode
16.19%	all idle time	kernel
28.92%	*all kernel* select *functions*	kernel
10.61%	soo_select()	kernel
9.32%	selscan()	kernel
6.92%	undo_scan()	kernel
2.07%	select()	kernel
4.19%	do_select()	user
1.83%	_Xsyscall()	kernel
1.71%	_doprnt()	user
1.21%	memset()	user
1.15%	cache_lookup()	kernel
1.11%	namei()	kernel
1.08%	read_io_port()	kernel
1.01%	syscall()	kernel

750 cold connections, 50 hot connections,
800 requests/second

Table 9.3 Example profile for unmodified kernel

mation returned by it*. In total, the select() mechanism accounts for more than 33% of the system's CPU cycles.

The CPU profile of the new system (Table 9.4) is drastically different. The do_select() function consumes only 0.98% of the system's CPU. Even this CPU share is still greater than the 0.70% of CPU that is consumed by all kernel event functions together. This dramatic decrease in the cost of the event mechanism results in the 54.23% idle time now available in the system. This situation is in contrast to the 16.19% system idle time in Table 9.3. Increased idle time, which is a result of the significantly improved scalability of the new event API vis-a-vis select(), also improves the response time of many other system activities because the Digital UNIX kernel performs many latency hiding activities, such as zeroing out memory pages for later use, when it has idle cycles.

*The original thttpd contains the main loop that calls select() in the function main(), which also does some other work besides calling select(). To be able to pin-point the user-level costs of using select() exactly, I reorganized the code to use a new function do_select() to call select() and collate the information it returns.

CPU %	Procedure	Mode
54.23%	all idle time	kernel
1.96%	_Xsyscall()	kernel
1.80%	_doprnt()	user
1.75%	memset()	user
1.38%	cache_lookup()	kernel
1.22%	namei()	kernel
1.22%	read_io_port()	kernel
1.14%	syscall()	kernel
1.14%	_XentInt()	kernel
1.09%	malloc()	kernel
1.09%	free()	kernel
0.98%	do_select()	user
0.70%	*all kernel event API functions*	kernel
0.35%	get_next_event()	kernel
0.21%	declare_interest()	kernel
0.14%	revoke_interest()	kernel

750 cold connections, 50 hot connections,
800 requests/second

Table 9.4 Example profile for new kernel

Another secondary performance effect of the event API, which was verified at higher request rates that saturate the system, is an improvement in the d-cache behavior of the system. The new event API, unlike select(), does not clobber the d-cache while performing a linear scan through data structures corresponding to hundreds of connections. This results in significantly higher maximum throughput for the new server system.

9.2.10 Effect of LRP on TCP dynamics

I will now describe some experiments that were performed to evaluate the effect of the LRP network subsystem architecture on the dynamics of TCP. As mentioned in Section 5.6, there is a concern that the delayed processing of packets in LRP causes an adverse effect on the performance of TCP. The discussion in Section 5.6 argued qualitatively that LRP does not have a negative effect on the operation of TCP's flow-control algorithm. We also saw that the LRP's effect on TCP's congestion control depends on the particular CPU scheduler in use. We argued that with feedback based fair-share schedulers, as are found in most general-purpose operating systems, there is an automatic adjustment of priority which causes LRP to have no significant effect on congestion-control dynamics in most situations. In the following discussion, the claims made in Section 5.6 are quantitatively verified.

Effect on accounting and flow-control

To begin with, I conducted an experiment to quantify the effect of correct accounting in LRP on the operation of TCP's algorithms. As discussed in Section 5.6, with a UNIX fair-share scheduler, the correct accounting of protocol processing causes a decrease in the number of CPU cycles available to the sender and receiver applications. The following discussion quantifies the effect of this reduction.

In this experiment, TCP's throughput was measured under a number of different scenarios. Specifically, 24MB of data were transferred between a sender application running on a Pentium Pro client system and a receiver application running on the Alpha server system. The receiver system was first configured to run vanilla Digital UNIX, and then the measurements were repeated with the receiver system running the RC prototype system. Additionally, the transfer throughput was measured in the presence of, and then in the absence of, a matrix-multiplication (matmult) process on the receiver system. To observe the timing of the data flowing on the network, I used tcpdump [112] and some analysis tools similar to those described by Mogul [108].

The purpose of the matmult process was to present a competing resource principal (besides the receiver application) to the receiver machine's operating system. With this competing principal, the RC system's[†] correct accounting mechanism comes into play since it indirectly affects how CPU cycles are distributed between the two principals.

To factor out the effect of TCP's congestion control algorithms, the sender's TCP was configured to use an initial send window equal to the receiver's advertised flow control window. It was also ensured that no packets were lost; thus TCP's congestion control algorithm was never triggered into operation. To fully expose the effect of receiver scheduling, the receiver application's receive buffer was deliberately kept small (4KB) in order to cause the receiver application to stall if ever the sender was

[†]In this section we use the terms "LRP" and "RC system" interchangably.

delayed in its transmission of data. The results of this experiment are reported in Table 9.5.

In the absence of matmult, the transfer is network-bound as there are plenty of CPU cycles available for protocol processing on the receiver system. The receiver application always blocks on a receive system call and remains at sufficiently high priority for protocol processing to be performed eagerly. Because of this, the throughput of the vanilla Digital UNIX system and the RC system are identical.

When matmult runs in the background on the receiver system, there is a reduction in throughput of both systems. This reduction happens because the receiver CPU is now shared between the receiver application and the matmult process. As a result, the transfer sometimes becomes limited by the receiver CPU. Instrumentation indicated that the receiver process was almost always network-bound, and blocked on a receive system call, but occasionally (after the receiver application had consumed sufficient CPU cycles) had its priority lowered and was timesliced-off the CPU.

This reduction in throughput is greater in the RC system because protocol processing in LRP is always charged to the network receiver, and never to matmult, which is not the case in vanilla Digital UNIX. In the RC system, the CPU is split more fairly between the two applications and less CPU is available to do protocol processing. Indeed, detailed event tracing confirmed that the receiver process is taken off the CPU (because its timeslice expires) for 30% of the system's time in the RC system. This is in contrast to the corresponding number of 21% for the unmodified Digital UNIX system. The complementary effect is that while in vanilla Digital UNIX the matmult process gets only 37% of the CPU, in the RC system it gets 44% of the CPU. Tracing also confirmed that during the active timeslices of the receiver, the network utilization was the same for both systems. This fact is discussed further below.

Figure 9.8 and Figure 9.9 show 120ms fragments of the TCP sequence number plots for packets seen by the receiver machine while running vanilla Digital UNIX, and

System	throughput (Mbps) (no matmult)	throughput (Mbps) (with matmult)	CPU share of matmult
vanilla Digital UNIX	95	74	37%
RC prototype system	95	66.5	44%

Table 9.5 Effect of LRP on accounting and flow control

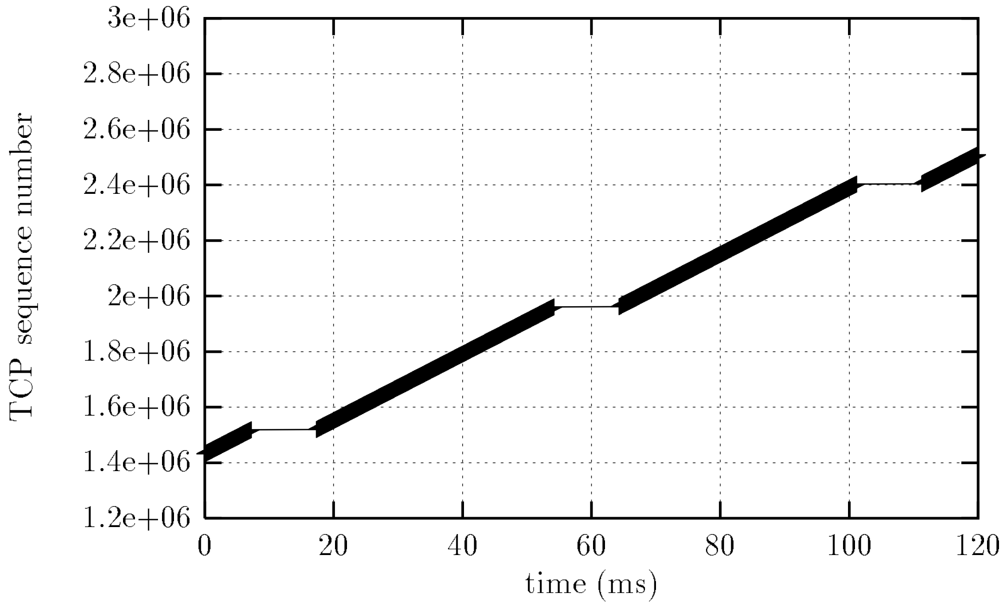

Figure 9.8 Sequence number plot #1 for vanilla Digital UNIX.

with the RC system. The increasing part of the curves indicate time when the receiver machine is receiving data. The flat part of the curves correspond to time periods when the receiver system does not receive any data. When the receiver application is running on the CPU, it reads from its socket buffers and window-updating ACKs are sent back to the sender. Consequently, during this time, the data transfer proceeds at a rate that is bounded by the speed at which the receiver application consumes data. This corresponds to the increasing portions of the sequence number plots.

When the receiver application is timesliced off the CPU, it does not read any data. No window-updating ACKs are sent, and the data transfer stalls. This corresponds to the flat parts of the sequence number plots. During each of these $10ms$ timeslices, matmult gets exclusive use of the CPU. No packets arrive for the receiver application during matmult's timeslice, except a few (1 or 2) packets immediately at the beginning of the timeslice.

As can be seen, the slope of the sequence number curve in its increasing segments is the same for both systems. This indicates similar network utilization for both receiver systems when the receiver application is active on the CPU. We can also see that the receiver application has more "off-CPU" timeslices per unit time in the RC

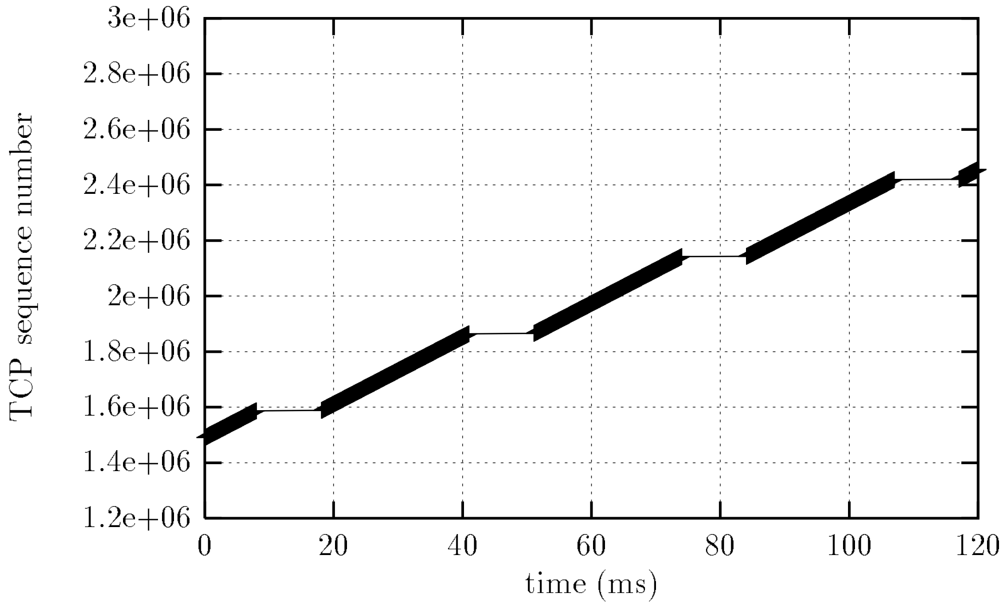

Figure 9.9 Sequence number plot #1 for RC system.

system; this is a result of the more equitable distribution of CPU in the RC system. As indicated in Section 5.6, this effect is both correct and desirable.

Furthermore, the sequence number plot of the RC system indicates that when the receiver application is active, i.e., not timesliced off the CPU, each packet that it consumes causes the receiver TCP to transmit an ACK, and subsequently the sender transmits a new packet. There is no observable additional delay in the RC system (vis-a-vis the vanilla DUNIX system) between the instant that the receiver application consumes data and the time that a corresponding window-updating ACK is transmitted. This confirms the claim made in Section 5.6 that LRP does not delay the transmission of window-updating ACKs and therefore has no adverse effect on the operation of the flow control algorithm.

I repeated this experiment with a small modification— the buffer size of the receiver was increased to 256K. In this case, it was observed that the sender keeps transmitting data at its maximal rate when the receiver application is first context-switched out. This causes the receiver application to have a sufficient number of unconsumed packets in its receive buffer to remain CPU-bound during its next CPU timeslice. From this point, the receiver application and matmult share alternate $10ms$

CPU timeslices. The receiver TCP generates window-updating ACKs only during the active timeslices of the receiver application. This causes a corresponding series of time periods when the sender receives these ACKs and generates new data. In the timeslices of matmult, the sender is mostly inactive.

The transfer bandwidth achieved in this experiment by the unmodified Digital UNIX system is 67Mbps. The RC system achieves a somewhat lower rate of 59Mbps. The complementary effect is that the matmult process gets 43% of the system's CPU in the vanilla Digital UNIX system, and almost exactly 50% of the CPU in the RC system. This result is shown in Table 9.6.

Detailed analysis of the sequence number plots in this experiment confirmed that there is no qualitative difference between the behavior of the sender TCP in the two cases. As in the previous experiment, the RC system does not delay window-updating ACKs. The redistribution of CPU on the receiver simply decreases the amount of CPU resources available to perform network processing in the RC system. As discussed in Chapter 5, this is the desirable behavior. Note that the ratio of transfer bandwidth to receiver application CPU share is almost identical for the two systems (1.175 for Digital UNIX vis-a-vis 1.18 for the RC system).

Effect of LRP on TCP's congestion control

In Section 5.6, we discussed the effects of lazy packet processing on the operation of TCP's congestion control algorithm. In that discussion, we argued that the nature of the end-host's scheduler is the real determinant of the precise behavior of TCP's congestion control under LRP. We showed that, in general, an application in a LRP system must arrange for the scheduler to provide it with CPU cycles at a granularity that matches individual packet arrival and transmission times. This is necessary because all protocol processing now happens in the resource context of the application,

System	throughput (Mbps) (no matmult)	throughput (Mbps) (with matmult)	CPU share of matmult
vanilla Digital UNIX	95	67	43%
RC prototype system	95	59	50%

Table 9.6 Effect of LRP on accounting and flow control (II)

and the timing constraints on packet processing (which are necessary for achieving good network throughput) are the responsibility of the application.

We also argued that for a system with a feedback based fair-share scheduler, LRP does not have any significant effect on the operation of TCP's congestion control in most situations. When the progress of the receiver application is limited by the rate of data arrival, the end-hosts perform eager processing. As a result, no ACKs are delayed and the operation of the congestion control algorithm is unchanged from its operation under ERP. When the congestion window is fully open, LRP may cause some ACK packets to get delayed, but this effect is inconsequential as the system is now limited by the operation of the flow control algorithm, i.e., by the speed at which the receiver application consumes data. In the experiments described below, we evaluate the effect of LRP on TCP's congestion control algorithm in systems that use a feedback based fair-share scheduler.

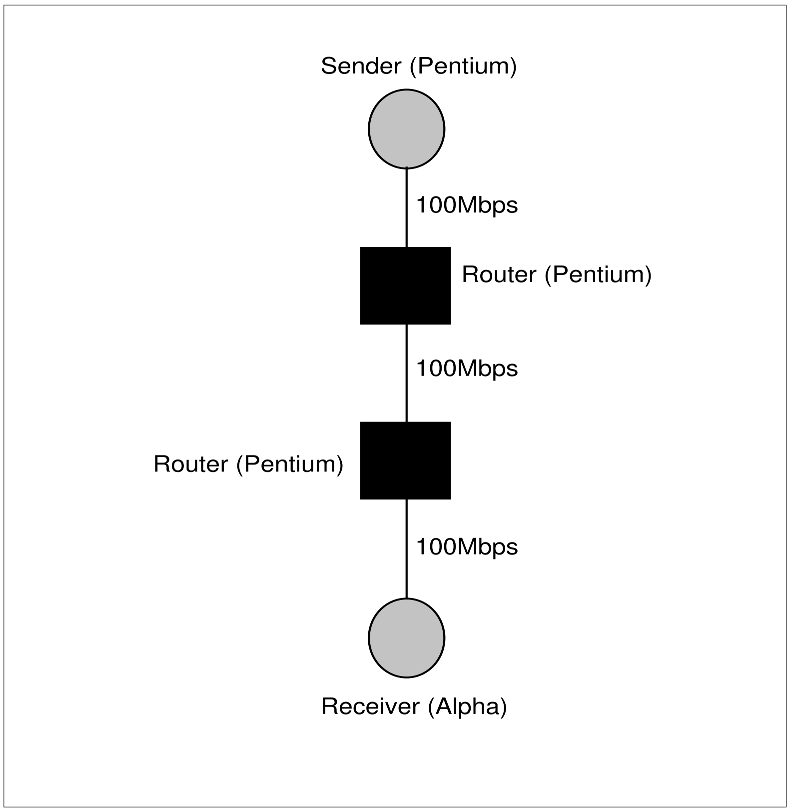

Figure 9.10 Testbed for congestion control experiments

To evaluate the effect of lazy protocol processing on TCP's congestion control, a testbed was set up where a Pentium Pro client and the Alpha server were connected to each other using a simple network with an optional bottleneck link (Figure 9.10). The sender machine (Pentium Pro) was connected to a router on a 100Mbps path. This router was connected to another router using a 100Mbps path. Finally, the second router was connected to the receiver machine (Alpha) using a 100Mbps link. Each router was also a Pentium Pro machine.

The kernel on both routers was changed to allow the addition of an optional delay in packet forwarding. Specifically, the IP forwarding code was modified to delay the transmission of each outgoing packet by a configurable time period [12]. This allowed for the simulation of the large round-trip delays of WANs in the path between the sender and the receiver systems.

A series of experiments was conducted on this testbed. First, a delay of 50ms (100ms round-trip) was introduced on the router-router path. The throughput of a 24MB file transfer was measured with the receiver system either running a vanilla Digital UNIX kernel or the prototype RC kernel. As in the flow control experiments, a matmult process was optionally run in the background on the receiver system. A large socket receive buffer (1024KB) was used for the receiver application.

In the absence of matmult, the transfer is always network-bound. This fact is true for the receiver running on the unmodified Digital UNIX system, and also with the RC system. After the initial congestion window increase phase (slow-start in TCP terminology), both transfers proceed at network speed and achieve an effective throughput of 95Mbps, as in the previous experiment (Table 9.5).

When matmult is present, it was observed that when the congestion window is not fully open, both systems behave identically since the receiver is not CPU-bound. The RC system never delays TCP processing and therefore the transmission of ACKs is never delayed. With both systems, the rate of increase of the congestion window is similar. Subsequently, when the congestion window opens fully, both systems become CPU-bound. With vanilla Digital UNIX, the transfer speed (after factoring out the initial slow-start phase) is 70Mbps. For the RC system, the throughput achieved is 59Mbps. In the Digital UNIX system, matmult gets 41% of the CPU; in the RC system, matmult gets almost exactly 50% of the CPU.

Figure 9.11 shows a TCP sequence number plot fragment for the Digital UNIX system. This fragment corresponds to the phase of the transfer after the slow start. As we can see, there are 14.8ms long periods of increasing sequence numbers for

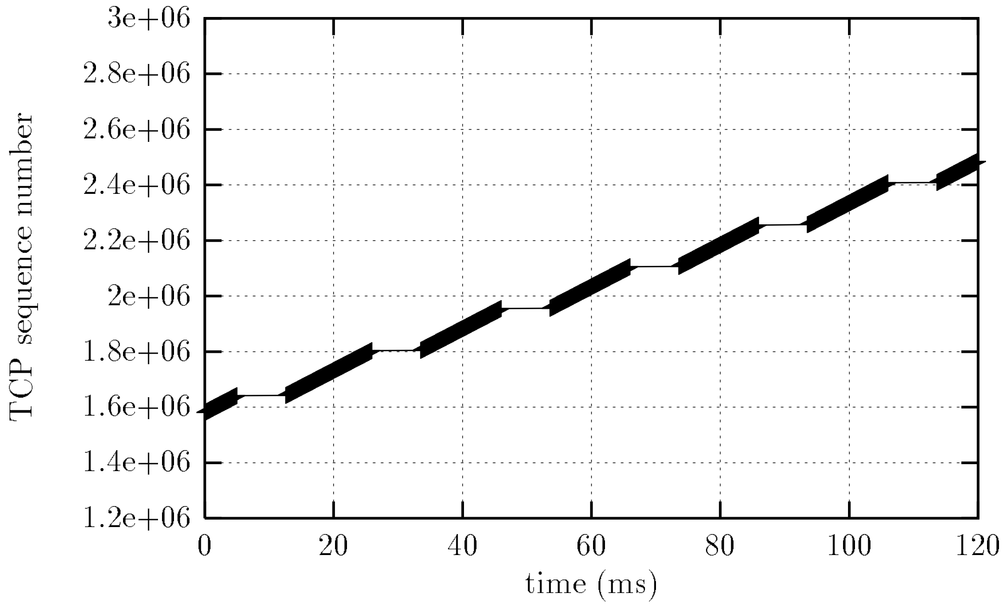

Figure 9.11 Sequence number plot #2 for vanilla Digital UNIX.

each $10ms$ active timeslice of the receiver application. Some packets received by the receiver machine are processed through TCP/IP during the active timeslices of matmult leaving it with only 41% of the system's CPU. The behavior of this plot is identical to what was described for the flow control experiments.

This time period of $14ms$ is longer than the $10ms$ timeslice of the receiver application because of the following: During its active timeslices, the instantaneous data processing bandwidth of the receiver application is about 140Mbps ($2 \times$ 70Mbps); thus ACKs that the receiver TCP sends out during the receiver application's active timeslices are somewhat compressed. Since the sender can transmit packets at a maximum rate that is limited by the 100Mbps link bandwidth, it needs $14.8ms$ to send data that the receiver application can consume in $10ms$. TCP processing for data that arrives during the timeslice of matmult consumes some CPU cycles from the matmult process. This explains why matmult gets only 41% of the total CPU in spite of getting exactly half of the CPU timeslices.

The corresponding plot for the RC system is shown in Figure 9.12. Each increasing sequence number segment is about $12.5ms$ long. Again, this is because the instantaneous data processing bandwidth of the receiver application during its active

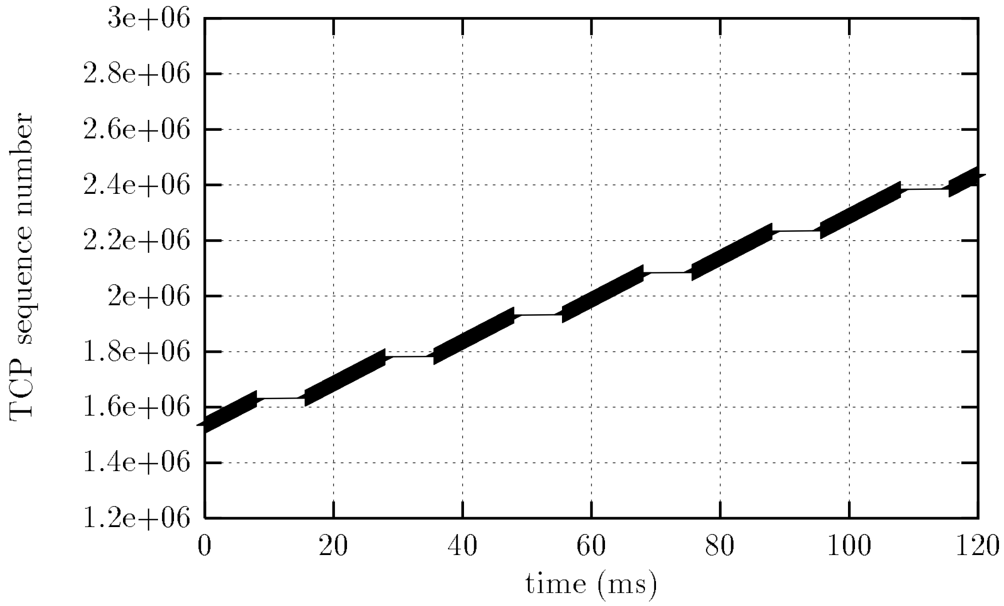

Figure 9.12 Sequence number plot #2 for RC system.

timeslice (118Mbps = 2 × 59Mbps) is larger than the link bandwidth. ACKs sent by the receiver during each of its $10ms$ timeslices cause the sender to transmit new data which requires $12.5ms$ worth of transmission time.

Unlike in Figure 9.11, however, data packets that arrive at the receiver during the timeslice of matmult are not processed immediately; this causes matmult to get almost exactly 50% of the system's CPU since it gets exactly half of all $10ms$ timeslices. This also accounts for why the receiver application's instantaneous data processing bandwidth during its active timeslice is somewhat lower in the RC system than in the unmodified Digital UNIX system. The Digital UNIX system effectively gets some TCP processing for free in the CPU timeslices of matmult.

It should be noted that in the experiment above, we see ACK compression. In the Digital UNIX case the consumption of data by the receiver application in its active timeslices generates window-updating ACKs that allow the sender to transmit new data at 140Mbps. In the LRP case, the compressed ACKs allow the sender to transmit at 118Mbps. In both cases the ACKs allow the sender to transmit data at a rate that is greater than the bandwidth of all links in the sender-to-receiver path. This ACK compression is caused by flow control effects (see Section 5.6). During its active timeslices, the receiver application reads data (and thus generates ACKs) at a much higher rate than the rate at which this data was received by the receiver host. Notice that the degree of ACK compression induced by LRP is no more than what happens in the vanilla Digital UNIX case. However, I have not experimentally studied the effect of LRP on ACK compression (and transmission burstiness) in any detail.

To factor out the effect of LRP's correct accounting as manifested in the redistribution of CPU on the receiver, I repeated this experiment with matmult confined to a resource container that had been assigned a somewhat lower priority than the container of the receiver application. Specifically, matmult's container was configured to get no more than 41% of the system's CPU. This corresponds to the CPU share that matmult gets in the vanilla Digital UNIX case described above.

The throughput of the data transfer was measured after factoring out slow-start. The measured value was 73.5Mbps: very close to the 70Mbps value observed above with vanilla Digital UNIX. This result reinforces the claim that for a typical receiver application running a feedback based fair-share scheduled system, LRP has no effect on TCP's congestion control dynamics except what results from a more equitable distribution of CPU on the end-hosts.

Detailed instrumentation indicated the following: In steady-state, i.e., after the slow-start phase has passed, matmult got roughly 4 timeslices for every six timeslices that the receiver application got: a CPU distribution very close to what we desired. The fact that the RC system managed 73.5Mbps with 59% of the system's CPU vis-a-vis the vanilla Digital UNIX system's 70Mbps value with the same amount of CPU is because of the slightly higher efficiency of protocol processing under the LRP execution model [10, 52]. A DCPI [7] profile of the receiver system confirmed this; the receiver system spent slightly lesser time in protocol processing code in the RC case as compared to the vanilla Digital UNIX case (for a given data transfer rate).

Next, packet losses in the network were simulated by having a router drop packets under user control. As described in Section 5.6.1, on detecting congestion TCP cuts its sending rate, and then tries to recover it by gradually increasing its congestion window. I wanted to determine the effect of this changing sending rate on the behavior of the RC system relative to the original Digital UNIX system, and the resulting effect on throughput.

The results of this experiment are shown in Figure 9.7. Two packet drop scenarios were considered: in the first scenario, 1 packet in every 100,000 was dropped; in the

System	*scenario #1* throughput (Mbps)	*scenario #1* CPU share of matmult	*scenario #2* throughput (Mbps)	*scenario #2* CPU share of matmult
vanilla Digital UNIX	62	48%	12	89%
RC prototype system	52	56%	12	89%

Table 9.7 Effect of LRP on congestion control

second, 1 packet in every 1000 was dropped. As can be seen, there is a drop in transfer throughput from the previous experiment for both receiver systems because of packet losses. However, this drop in throughput is similar for both systems. For the first packet drop scenario, the throughput drop for the Digital UNIX system is 11.4% and the throughput drop for the RC system is 11.9%. For the second drop scenario, both systems achieve a throughput of 12Mbps.

This drop in throughput is entirely due to the effect of the sender TCP lowering its transmission rate for some time when it detects packet loss. Extensive examination of tcpdump traces showed no observable qualitative difference between the behavior of the two systems in the face of the sender TCP adjusting its sending rate on detecting losses.

For the RC receiver case, whenever the sending rate was lowered, the sender could not keep the receiver busy and the receiver application ceased to be CPU-bound. Subsequently, the priority of the receiver application was elevated and protocol processing was performed eagerly, as in the Digital UNIX system. The sender's transmission rate increased at the same rate as in the vanilla Digital UNIX case to the point where the sender system became bound by the receiver application's speed again.

In the RC system, the receiver application did get CPU-bound somewhat earlier (82-143ms, mean 98ms) than in the Digital UNIX system. As discussed before, this is because of correct accounting of CPU cycles to the receiver application. The RC receiver system's kernel does not allow the receiver application to consume more than 50% of the system's CPU cycles for any substantial period of time.

One might expect that because a RC system with a CPU-bound receiver application does not process TCP packets in the timeslice periods of matmult, it may delay sending duplicate acknowledgments that trigger a fast-retransmit [81]. Thus, some packet losses may be detected and recovered later with an RC receiver, than with a vanilla Digital UNIX receiver. However, an examination of tcpdump traces in the above described experiment around time instances where packets are re-transmitted by the fast-retransmit algorithm indicates the following: With a RC system as the receiver machine, the sender notices duplicate ACKs for a single lost packet between 4.3 and $10.8ms$ (mean $8.1ms$) later than with the vanilla Digital UNIX system as the receiver. Thus, the net effect of lazy packet processing on fast-retransmit is the same as if the sender-receiver path had a longer delay by a time period equal to the timeslice period ($10ms$) on the receiver. Since typical variance in WAN delays is much more than $10ms$, this distinction has no real significance.

To summarize, in the above described experiments, lazy packet processing does not seem to have a significant effect on TCP's congestion control beyond what is due to correct accounting. Whenever the receiver's progress is limited by the rate of data arrival, the feedback based adjustment of the receiver's priority causes the receiver application to gain priority and perform eager processing. As a result, in all these experiments the dynamics of ACK transmission under LRP are similar to the ERP scenario.

As indicated earlier in Section 5.6, this adjustment does not necessarily happen for other types of CPU schedulers. In such systems, it may be necessary for applications to request CPU quanta to match individual packet transmission times; the application is now responsible for meeting any timing constraints that are necessary to achieve good data transfer throughput. Even in fair-share feedback-based systems, it may be necessary for the receiver (and the sender) applications to isolate the activity of data transfer from other computationally intensive activities to benefit from feedback-based priority adjustment. These issues need more research. A full study of the interaction of TCP with scheduler dynamics is, however, beyond the scope of this dissertation.

9.3 Summary

To summarize this chapter, I showed that the use of resource containers does not add significant overhead to the operation of servers. Containers allow server applications to implement prioritized handling of clients. Containers also allow resource consumption to be effectively controlled in a large variety of complex situations such as those involving CGI processes and in Rent-A-Server environments. It was shown that a container based system is stable under overload. Moreover, the use of resource containers also allows a server to protect itself against SYN-flood based denial-of-service attacks.

This chapter also quantitatively verified the ability of the new event API to deliver events in a prioritized manner. The near-perfect scalability of this API was also demonstrated. Finally, a study of TCP's dynamics under the new system was described. It was shown that lazy receiver processing does not significantly affect TCP's dynamics for UNIX fair-share scheduled systems.

Chapter 10

Related work

In recent years, operating system researchers and vendors have devoted much effort to improving the performance of I/O-intensive applications. A substantial portion of this work has focused specifically on improving Internet server performance. The work presented in this dissertation draws from some of this related research, and complements other parts.

This chapter presents an overview of related work. It begins by briefly describing the various efforts aimed at improving Internet server performance. Next, it presents an overview of work that has been done on developing fine-grained resource management mechanisms in the context of experimental real-time and multimedia operating systems. We briefly describe the nature of this related work, and show its relationship to the new resource management model presented in this dissertation.

We also discuss some recent efforts that attempt to provide differentiated quality-of-service in the context of the Internet. It is shown that resource containers are critical to the accomplishment of this goal, and complement the work done so far towards the realization of this goal.

10.1 Improving the performance of I/O-intensive applications and Internet servers

A lot of work has been done on improving the raw I/O processing speed of operating systems. As mentioned in Chapter 1, most of this work has focused on improving message latency and on delivering the network's full bandwidth to application programs [6, 17, 33, 38, 50, 51, 53, 54, 55, 113, 115, 130, 158].

More recently, researchers have started to look specifically at the performance of Internet servers on general-purpose operating systems. One early experience that lead to published results was the 1994 California election server [109, 110]; another early study was performed at NCSA [100]. Operating system vendors responded to complaints of performance problems by improving various kernel mechanisms, especially by replacing BSD's linear-time PCB lookup algorithm [101], and by changing certain kernel parameter values. Vendors also provided tuning guides for systems being used as Web servers [40].

There have been also some proposals for enhancing the UNIX system call interface for supporting I/O in Internet servers more efficiently. These include proposals to support zero-copy I/O [130, 158], and to reduce the number of system calls in the typical I/O path [78, 118].

In response to observations about the large context-switching overhead of process-per-connection servers, recent servers [31, 121, 151, 161, 174] have used event-driven architectures. Measurements of these servers under laboratory conditions indicate an order of magnitude performance improvement [31, 142]. However, some studies of such servers under real workloads, e.g., Maltzahn et al. [97], Fox et al. [68], indicate significant state-management overhead and poor scalability. The fundamental reason behind these problems is the poor scalability of the select() (and the poll()) system call.

Addressing these problems has been the subject of some of my recent research work [15, 13]. Specifically, I have developed an implementation of select() that scales significantly better than the vanilla implementation [15]. However, the inherently-linear interface of select() fundamentally limits scalability, and, as described in Section 3.7.1, prevents a server application from providing differentiated handling of connections. The event API of Section 4.10.1, which was first briefly described in [13], is an attempt to provide a scalable, and priority-aware, replacement for select().

Other approaches for increasing server performance employ multiple machines. In this area, there has been some work that has focused on using multiple server nodes in parallel [68, 35, 79, 46, 129], or sharing memory across machines [45, 62, 93].

10.2 Resource management issues in servers

Resource management issues in network servers were first studied in the context of the overload behavior of operating systems [110, 111]. Mogul and Ramakrishnan [111] devised and evaluated a set of techniques for improving the overload behavior of an interrupt-driven network architecture. These techniques avoid receiver livelock by temporarily disabling hardware interrupts and using polling under conditions of overload. Disabling interrupts limits the interrupt rate and causes early packet discard by the network interface. Polling is used to ensure progress by fairly allocating resources among receive and transmit processing, and multiple interfaces.

The LRP network subsystem architecture [52, 10] described in Chapter 5 attempts to eliminate undesirable resource starvation effects, such as livelock, by design. As described in in this dissertation, LRP drives a principal's resource context into the kernel. As a result, the system's usual resource management mechanisms, which ensure fairness between traditional, mostly user-level, principals, are now able to account for, and control kernel-mode resource consumption.

Architecturally, the design of *user-level network subsystems* [91, 17, 50, 53, 57, 96, 160] is similar to that of LRP. Unlike LRP's goals, however, the main goal of these efforts is to achieve low communication latency and high bandwidth by removing protection boundaries from the critical send/receive path, and/or by enabling application-specific customization of protocol services. However, such subsystems share with LRP the combination of the techniques of early demultiplexing and protocol processing in the context of the application. Thus, user-level subsystems also support a strong equivalence between a process and a resource principal, and should have similar behavior from a resource management standpoint.

As described in Chapter 3, the natural boundary of a resource principal in a modern server application does not usually coincide with a process. Thus, a system that treats processes as principals, even if it is based on LRP or a user-level network subsystem, is not effective in controlling resource consumption in such environments. A more general notion of a resource principal is needed; the resource container abstraction described in this dissertation provides the needed mechanism by separating the notion of a resource container from that of a protection domain.

10.3 Operating system mechanisms for resource management

A wealth of mechanisms have been developed by the operating systems research community to support fine-grained resource management, and resource management at a granularity other than a process. Most of this work has been done in the context of experimental operating systems, particularly hard and soft real-time operating systems; other efforts have been in the context of mainframe environments and, more recently, in multiprocessor compute server system environments. All of these abstractions offer some point solutions to the general problem of resource management. I will now discuss these mechanisms, and their relationship to resource containers, in some detail below.

10.3.1 The paths of Scout

The Scout operating system [116, 114] has explicit support for a *path* abstraction, representing an I/O channel (such as a TCP connection) through a multi-layered system. A path encapsulates the specific attributes of an I/O channel, and allows access to these attributes across layers. Paths have been used to improve execution speed in a network subsystem by enabling cross-layer optimizations [115]. Paths have also been used to implement fine-grained resource management in network appliances, such as network-attached TVs [114] and Web server appliances [150].

Paths are similar to resource containers; however, resource containers are more general as they can encompass several otherwise "unconnectable" paths. Essentially, resource containers, in contrast to paths, allow the application to treat the resources consumed by several I/O channels as being part of the same activity. Moreover, unlike paths, resource containers can be hierarchically structured, as described in Section 4.4, allowing server applications to support very general resource management scenarios. While theoretically, the path model could perhaps be extended to support hierarchies of paths, there have been no specific efforts in this direction.

In addition, the binding between a resource container and the kernel resources that can be associated with it is more dynamic and flexible than the association between a path and Scout kernel entities. This is because paths are specified at kernel build time. They can be instantiated, extended, optimized and associated with execution entities (threads) at run time during the path creation phase. However, the association between a path and the system resources associated with it cannot be arbitrarily changed *after* the path creation phase. For instance, we cannot change the binding between a path and a kernel resource, such as a socket, *after* the path creation phase. Also, unlike resource containers which can encompass arbitrary sets of resources at run-time, the composition of a path is limited by the router graph specified at kernel build

Finally, Scout is a special-purpose operating system built from scratch to efficiently support network appliances. Scout's paths imply a fundamental redesign of the entire operating system. The path abstraction is not available in general-purpose operating systems, and there has been no work to make paths work with current operating system interfaces.

10.3.2 Processor capacity reserves

Mercer et al. [105, 104] designed an operating system abstraction called a *processor capacity reserve* in the context of the Real-Time Mach operating system [162]. Reserves insulate programs from the timing and execution characteristics of other programs. An application can reserve system resources, and the operating system ensures that these resources will be available to threads associated with the reserve when needed.

At a superficial level, reserves are similar to resource containers. Like resource containers, a reserve provides a thread with a resource context. A thread can be associated with several reserves and vice versa. Reserves can be passed between protection domains. Thus, reserves can be used to correctly charge resources consumed in the execution of a task distributed across protection domains to the same resource principal.

However, the work of Mercer et al. was primarily concerned with issues involved in the specification of resource reservations based on the timing constraints of programs, and the operating system mechanisms needed to enforce these reservations. RT Mach is a micro-kernel based system where all network I/O processing is performed at user level [91]. Unlike resource containers, this work does not address the problem of controlling resources consumed by the kernel in performing network processing on behalf of applications. Thus, the issues involved in integrating explicit resource principal abstractions with kernel processing were not addressed. There has been no attempt to integrate reserves with the socket system call interface. This difference is very important for real-world Internet servers as most general-purpose operating systems on which they run are typically monolithic in nature and perform network processing inside the kernel.

Finally, unlike this dissertation, the processor capacity reserve work did not address the structuring of reserves in a hierarchical framework, or its use with a hierarchical scheduling framework.

10.3.3 Migrating threads

The *migrating threads* [66] of Mach [1, 24] and AlphaOS [39], and the *shuttles* of Spring [73] allow the resource consumption of a thread (or a shuttle) performing a particular independent activity to be charged to the correct resource management entity, even when the thread (or shuttle) moves across protection domains. This capability is similar to some of the functionality provided by resource containers; however, it is not as general because a thread is still tied to a single resource principal. Thus, these systems cannot correctly handle situations where a single thread is desired to be associated with multiple independent activities, as is the case in an event-driven Internet server.

Perhaps more fundamentally, Mach and Spring are also micro-kernel based systems. Thus, migrating threads and shuttles, like the real-time work of Mercer et al., do not address how the resource context of an application can be extended into the kernel. These abstractions are not applicable to the problem of controlling resources consumed by an application in kernel based network processing.

10.3.4 Activities of Rialto

The *activity* abstraction [85] of the Microsoft's experimental Rialto real-time operating system [84] is also somewhat similar to a resource container. The chief differences, as in the case of processor capacity reserves, are related to the fact this work does not address the control of resources in kernel based network processing. Also, an activity in Rialto can be associated with multiple threads, but not the other way around, which limits its generality.

In addition, Rialto is an experimental object-oriented operating system, and presents a somewhat different context from the traditional monolithic kernels that this dissertation has focussed on. Rialto also aims at a real-time environment [86], rather than my concern with resource prioritization in a general-purpose system.

10.3.5 Resource groups of Opal

The Opal single-address space operating system [32] supports the notion of a *resource group*, which is Opal's notion of a resource principal. Like a resource container, a resource group is completely decoupled from the notion of a protection domain. The resource groups work was, however, concerned with flexible accounting and resource control in an experimental single-address space environment, while this dissertation has targeted the same issues in a (multi-address-space) general-purpose system. As a result, a large number of issues that have been addressed by this work, such control over resource usage in kernel based network processing, have no parallels in that work, and vise-versa.

10.3.6 Nemesis

The Nemesis operating system [92] provides accurate accounting of resource usage by running the application and most of the kernel code in the same address space. This allows fine-grained control over resource consumption. However, Nemesis employs a radically different operating system structure than the general-purpose systems that this dissertation focuses on. Moreover, unlike this work, Nemesis mainly focuses on real-time issues and not on prioritization of requests in a server environment.

10.3.7 The exokernel approach

As mentioned in Section 1, Kaashoek et al. [87, 88] advocate a customized operating system for servers, which is built from the ground-up. An exokernel based server operating system allows the application to control resource consumption for activities at all levels of the system. The application writer controls all of the code of the server operating system making many optimizations possible which cannot be applied in classical systems.

The exokernel approach to operating system development is a radical departure from how current operating systems are implemented. For this reason, this approach represents a point in the design space of server oriented operating systems which is less likely to be of immediate utility. Moreover, implementing the Web server system on a exokernel brings several software engineering issues related to the problem of developing and maintaining a complicated library operating system to the domain of the Web server developer. Nevertheless, it seems feasible to implement the resource container abstraction as a feature of an exokernel library operating system, since the exokernel delegates most resource management to user code.

10.3.8 Resource allocation schemes in mainframe systems

A number of resource management schemes in the context of mainframe operating systems [49, 141, 34] focused on resource management at a granularity other than a process. These systems tried to achieve a balance between fairness to the user and overall system utilization. Long-term fairness for users was achieved through the allocation of quotas (sometimes called budgets) in conjunction with a mechanism that accounted for the resources consumed by all applications of each user. The resource management system stopped a job once a user's resource quota was exhausted. Short-term fairness was achieved only at a process level through the use of a scheduling scheme similar to the decay-usage scheduling method described in Chapter 6.

In a sense, these approaches allow the system to consider a group of processes (all the applications of a user) as a single resource principal. In this respect, they share some of the same goals as my work. However, these schemes are not as general because they are able to guarantee fairness between users only for time-scales that were on the order of tens of minutes or more, i.e., significantly larger than is possible with the resource container approach. Moreover, these systems do not support scenarios where the natural boundary of a resource principal is a subpart of a process, as in an event-driven Internet server.

10.3.9 Reservation domains and software performance units

The *reservation domains* [26] of the Eclipse operating system allow the system to control the total resource consumption of a set of processes together. A reservation domain provides resource guarantees to a group of processes allowing a variety of QoS requirements to be met. The current implementation of reservation domains focuses on real-time issues.

Similarly, the *software performance units (SPUs)* of Verghese et al. [165] also allow the resource consumption of a group of processes to be considered together for the purpose of resource scheduling. The implementation of SPUs described in [165] considers controlled allocation of CPU, memory and disk bandwidth in a large symmetric multiprocessor based compute server environment. The SPU abstraction allows fairness between resource principals to be achieved at much shorter time-scales than the mainframe scheduling mechanisms described above.

Like resource containers, reservation domains and software performance units allow a resource principal to encompass a number of protection domains. Unlike resource containers, however, both abstractions do not address scenarios, such a single-process Internet server, where the natural extent of a resource principal is more complicated.

10.4 Providing differentiated quality of service in the Internet

Recently, there have been a number of efforts to provide multiple quality of service (QoS) levels in the context of Internet servers. These include research that attempts to develop QoS-aware Internet servers [3, 132] and efforts to provide quality of service support in the network [22, 172, 176]. In the following discussion, I will briefly describe these approaches and their relationship to resource containers.

10.4.1 Differentiated QoS in Internet servers

Almeida et al. attacked the problem of providing QoS support in a Web server running on a widely available general-purpose operating system [3]. The authors mapped QoS requirements onto scheduling priorities, experimenting both with a user-level implementation, and with a slightly modified Linux kernel scheduler. They used the Apache server [8], and so followed the process-per-connection model. This approach allowed them to provide differentiated service to HTTP requests in different QoS classes, albeit with some limitations on effectiveness. However, the authors did not evaluate how accurately their system allocated kernel-mode time. Their implementations did not attempt to set priorities for processing of received packets, or to differentiate between existing connections and new connection requests. Moreover, it is not clear how their approach can be extended to single-process server architectures.

Pandey et al. [132] also describe the construction of Web servers that attempt to provide differentiated quality of service to clients. This work proposes a QoS model for Web servers which supports the reservation of server resources as either a fixed percentage of the total resources of the server system, or as a rate/bandwidth guarantee. This QoS model can be easily enforced by a system based on resource containers.

The authors also describe the implementation of a distributed QoS-aware Web server based on the NCSA `httpd` [119]. This implementation uses only user-level mechanisms to provide differentiated quality of service. Thus, this approach is also limited in its effectiveness to provide multiple levels of service by the fact that the kernel resource utilization of server applications is not controlled. Moreover, to date, this work addresses only servers that have a process-per-connection architecture.

10.4.2 QoS support in the network

Purely end-host mechanisms, such as the resource container abstraction described in this dissertation (or the application-level efforts described in the previous subsection), cannot provide differentiated quality of service in an end-to-end network stream by themselves. This is because such end-host mechanisms cannot, in general, control what happens to packets inside the network. Explicit network-level mechanisms are needed in order to support QoS on an end-to-end basis.

Several recent proposals have tried to accommodate different levels of service in the network layer of the Internet. One such proposal is to extend IP [133] for integrated services [172]. This approach calls for receivers to request a guaranteed service commitment from the network via the RSVP protocol [176].

A second proposal is to extend IP to support differentiated services [22]. In this method, high priority traffic will take precedence over existing traffic on a per-packet basis. Routers built to support this approach will respect packet priorities in their queuing and forwarding algorithms.

There have also been several other, more general, studies to provide bandwidth reservation for network quality of service [63, 37, 175, 5]. In addition, many researchers have also proposed complete QoS architectures for managing quality of service specification and negotiation in a general setting [124, 170, 117, 65]. The work described in this dissertation complements this QoS research by providing an effective and flexible mechanism for controlling resource consumption in end-hosts and network gateways.

Chapter 11

Conclusion and future work

The objective of this dissertation is to addresses the lack of appropriate support for server applications in existing operating systems. This problem stems from a mismatch between the original assumptions underlying the design of the resource management facilities of current operating systems and the fine-grained control over system resources that is desired by modern server applications. This dissertation introduces a set of new resource management facilities that provide applications with control over resource consumption at all levels of the system, without sacrificing the basic structure of traditional general-purpose operating systems.

11.1 Contributions and limitations

The contribution of this thesis is a set of mechanisms that facilitate fine-grained resource management in server systems. Specifically, this dissertation proposed:

- A new model for resource management in operating systems based on a new abstraction called a *resource container*. By encompassing system resources that an application uses to perform a particular independent activity at any level of the system, regardless of how the activity is distributed across protection domains, resource containers enable fine-grained control over system resource consumption. All processing for an activity is charged to the appropriate resource container, and scheduled at the priority of the container. This allows appropriate policies to control how system resources are to be shared between independent activities.

- A new network subsystem architecture for performing kernel network processing in the context of the concerned principal. This architecture, which is based on lazy receiver processing, is key for a server system to have stable overload behavior, and to allow the new resource management model to be effective.

- A new set of operating system APIs that allow an application to use the capabilities of the resource container mechanism, and implement powerful functionality which was not possible before, such as the ability to prioritize kernel-intensive network connections within the same protection domain or the ability to protect a server from certain types of denial-of-service attacks.
- An implementation of containers in the context of a real, widely used, general-purpose operating system and a rigorous quantitative evaluation that shows the viability of the resource container mechanism in solving complex resource management problems.

The resource container mechanism and the new APIs proposed in this dissertation result from the following key observations about existing operating systems:

- The coincidence of the system's notion of protection domain and resource principal in the process abstraction of existing operating systems is problematic for many modern applications. These problems arise because the natural boundaries of a resource principal in such applications are often not the same as those of a process, or even a group of processes.
- Most operating systems allow the application very little control over kernel processing; this interacts poorly with server applications because such applications are particularly system-intensive. Explicit mechanisms are needed that allow the application to control resource usage at all levels of the system.
- Several operating system APIs interact poorly with the requirements of server applications, because these APIs were not designed for server applications.

This dissertation focuses on providing mechanisms that are needed to address the resource management problems of server applications. Example policies, that allow the demonstration of the usefulness of the container mechanism in solving complex resource management problems in server applications are also considered.

The widespread use of server applications has also introduced a host of complementary problems, such as poor memory system performance in server systems, poor scalability of OS mechanisms and interfaces, the choice of a server execution model, the design of a QoS architecture for Internet servers, etc. This work does not directly address these problems. Complementary research has addressed these issues [130, 158, 78, 118, 31, 131, 16, 15, 13, 132, 3, 22, 172].

11.2 Future work

In terms of future work, the work described in this dissertation has logical continuations along several dimensions. First, the resource container mechanism can be extended to control resource consumption across machine boundaries. Along with a more sophisticated, and machine-independent, identification and funding mechanism, resource containers can be used to encompass the computation of an independent activity across system and network boundaries. This will allow for flexible resource management in distributed environments like workstation farms and cluster-based network servers.

This dissertation exposed a number of issues related to the interaction between the scheduling of protocol processing in end-hosts and TCP's dynamics. When resources consumed in protocol processing are correctly accounted for and controlled by an end-system's scheduler, the application becomes responsible for timing constraints that are essential for good protocol performance. This issue needs further research: it remains to be worked out what new facilities a system's scheduler and resource management system should provide to applications whose performance depends critically on their interaction with protocol dynamics.

Resource containers can be used as a sand-boxing mechanism in Java virtual machine (JVM) implementations. A JVM is in many ways similar to a classical network server application running on a general-purpose operating system. A JVM hosts multiple untrusted activities, often within a single protection domain, and may perform a significant amount of computation for each activity inside the operating system kernel. The use of containers in this environment is a fertile area for new research.

Finally, there is a significant amount of engineering effort that is needed in order to tap the full capabilities of resource containers. Specifically, there is a need to develop policies which can be used in conjunction with resource containers for effective management of diverse resources, such as memory and disk bandwidth. The exact nature of these policies is often very application-specific, and different types of server applications need their own specialized policies.

Appendix A

Man pages

This Appendix contains detailed man pages for the new system calls and application-level programs described in this dissertation. The man pages are divided into four groups. The first set covers system calls that directly manipulate resource containers, i.e., those listed in Table 4.1. The second set contains the system calls of the new event API described in Section 4.10.1. Then, the man page for the new `bind()` system call is presented. Finally, man pages for the new resource container manipulating commands that were described in Section 8.7 are presented.

NAME

rc_create — create a new resource container.

SYNOPSIS

#include <sys/rc.h>

int rc_create(void)

DESCRIPTION

rc_create() creates a new resource container and returns a descriptor.

RETURN VALUES

A -1 is returned if an error occurs, otherwise the return value is a descriptor referencing the resource container.

ERRORS

The rc_create() call fails if:

[EMFILE]	The per-process descriptor table is full.
[ENFILE]	The system file table is full.
[ENOBUFS]	Insufficient buffer space is available. The resource container cannot be created until sufficient resources are freed.

SEE ALSO

set_rc_parent(1), close(1), sendmsg(1), recvmsg(1), get_rc_opt(1), set_rc_opt(1), get_rc_usage(1), get_rc_stats(1), get_rc(1), set_res_binding(1), add_to_sched_binding(1), delete_from_sched_binding(1), set_fd_rc(1), declare_interest(2), revoke_interest(2), get_next_event(2), bind(3).

NAME

set_rc_parent — set the parent of a resource container

SYNOPSIS

#include <sys/rc.h>

int **set_rc_parent**(int rc1, int rc2)

DESCRIPTION

set_rc_parent() sets the parent container of container rc1 to rc2.

After the call, any scheduler related attributes of rc1 now become relative to the scheduler attributes of rc2. For example, if rc1 was originally guaranteed 10% of the CPU share of its parent container, an attempt is now made to give it 10% of the CPU share of rc2. If prior resource commitments of rc2 (e.g., a 95% CPU share aggregate allocation to the already existing children of rc2) do not allow this, the best allowable guarantees are provided.

RETURN VALUES

A -1 is returned if an error occurs, otherwise 0 is returned.

ERRORS

The **set_rc_parent()** call fails if:

[EBADF]	rc1 is not a valid descriptor.
[EBADF2]	rc2 is not a valid descriptor.
[ENOTRC]	rc1 is not a resource container.
[ENOTRC2]	rc2 is not a resource container.

SEE ALSO

rc_create(1), close(1), sendmsg(1), recvmsg(1), get_rc_opt(1), set_rc_opt(1), get_rc_usage(1), get_rc_stats(1), get_rc(1), set_res_binding(1), add_to_sched_binding(1), delete_from_sched_binding(1), set_fd_rc(1), declare_interest(2), revoke_interest(2), get_next_event(2), bind(3).

NAME

close — delete a descriptor

SYNOPSIS

#include <unistd.h>

int close(int rcd)

DESCRIPTION

The close() system call deletes a descriptor from the per-process object reference table. The semantics of close() for resource container descriptors are very similar to those for other descriptors. As for other objects, such as files and sockets, if this is the last reference to the resource container, the container will be destroyed. Note, however, if a thread is resource-bound to a particular container, a reference to the container exists in thread's control block. Thus, if a thread closes a descriptor that corresponds to its resource binding, the descriptor is deleted but the container will stay around till it has no more references to it.

When a process exits, all associated resource container descriptors are freed.

When a process forks, all descriptors for the new child process reference the same resource containers as they did in the parent before the fork. If a new program is then to be run using execve(), the new program would normally inherit these descriptors. As for files and sockets, the resource container descriptors can be rearranged with dup2() or deleted with close() before the execve() is attempted, but if some of these descriptors will still be needed if the execve fails, it is necessary to arrange for them to be closed if the execve succeeds. Like for other descriptors, this can be achieved by using the fcntl() system call to set the CLOSE_ON_EXEC flag of the descriptor.

RETURN VALUES

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and the global integer variable errno is set to indicate the error.

ERRORS

The close() system call fails if:

[EBADF] rcd is not a valid descriptor.

[EINTR] An interrupt was received.

SEE ALSO

rc_create(1), set_rc_parent(1), sendmsg(1), recvmsg(1), get_rc_opt(1), set_rc_opt(1), get_rc_usage(1), get_rc_stats(1), get_rc(1), set_res_binding(1), add_to_sched_binding(1), delete_from_sched_binding(1), set_fd_rc(1), declare_interest(2), revoke_interest(2), get_next_event(2), bind(3).

NAME

sendmsg, recvmsg — transfer a resource container descriptor between processes

SYNOPSIS

#include <sys/types.h>
#include <sys/socket.h>
#include <sys/rc.h>

int **sendmsg**(int s, const struct msghdr *msg, int flags)

int **recvmsg**(int s, const struct msghdr *msg, int flags)

DESCRIPTION

sendmsg() is used to transmit a message to another process over a UNIX domain socket. **recvmsg()** is the corresponding system call used in the receiving process. Messages supported are regular data messages, as well as descriptors for objects, including resource containers. For details on how the message msg needs to be structured to do this, consult the UNIX manual pages for **sendmsg()** and **recvmsg()** or Stevens [153, 154].

RETURN VALUES

The **sendmsg()** call returns the number of characters sent, or -1 if an error occurred. The **recvmsg()** call returns the number of characters received, or -1 if an error occurred.

ERRORS

See the UNIX **sendmsg()** and **recvmsg()** man pages for details.

SEE ALSO

rc_create(1), set_rc_parent(1), close(1), get_rc_opt(1), set_rc_opt(1), get_rc_usage(1), get_rc_stats(1), get_rc(1), set_res_binding(1), add_to_sched_binding(1), delete_from_sched_binding(1), set_fd_rc(1), declare_interest(2), revoke_interest(2), get_next_event(2), bind(3).

NAME

get_rc_opt, set_rc_opt — get/set the options of a resource container

SYNOPSIS

#include <sys/rc.h>

int **get_rc_opt**(int rcd, struct rcopt *rco)

int **set_rc_opt**(int rcd, struct rcopt *rco)

DESCRIPTION

get_rc_opt and set_rc_opt allow the manipulation of a resource container's options. For each container, the kernel maintains the a set of attributes as defined below:

```
int rc_cpu_limit;
int rc_cpu_reserve;
int rc_cpu_pri;
int rc_mem_user_pages;
int rc_max_sockbufspace;
int rc_rights_inherit;
int propogate_stats;
```

rc_cpu_limit specifies any CPU usage limits for a container as a % fraction of the systems' total CPU cycles. rc_cpu_reserve specifies a CPU share reservation for a container. rc_cpu_pri is a priority value relative to other time-share sibling containers of a particular container. rc_mem_user_pages specifies the maximum number of user-level page frames that the container is entitled to. rc_max_sockbufspace limits the maximum amount of memory that can be used as socket buffer space by the container. rc_rights_inherit controls whether owner access rights are inherited by a new descriptor reference to the resource container. propogate_stats controls whether the resource usage statistics associated with a container are propogated to its parent container when the container in question is destroyed. All of these attributes are maintained in the resource_container structure described n Figure 8.1.

The rcopt structure is defined as follows:

```
struct rcopt {
   int attr_val;
   int attr_type;
};
```

The current prototype implementation supports the following values of attr_type: SCHED_CPU_LIMIT, SCHED_CPU_RESERVE, SCHED_CPU_PRI, MEM_USE R_PAGES, MAX_SOCKBUFSPACE, RIGHTS_INHERIT and PROPOGATE_STATS. These correspond to control over rc_cpu_limit, rc_cpu_reserve, rc_cpu_pri, rc_mem_user_pages, rc_max_sockbufspace, rc_rights_inherit and propagate_stats attributes respectively.

Future versions may support more attributes; this is why the structure rcopt is so defined. If a rc_options structure consisting of just the attributes described above was passed during each call to these system calls, older programs would cease to work if this structure was to change.

RETURN VALUES

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and the global integer variable errno is set to indicate the error.

ERRORS

The get_rc_opt() system call fails if:

[EBADF]	rcd is not a valid descriptor.
[ENOTRC]	rcd is not a resource container.
[EFAULT]	rco is not a valid address.

The set_rc_opt() system call fails if:

[EBADF]	rcd is not a valid descriptor.
[ENOTRC]	rcd is not a resource container.
[EFAULT]	rco is not a valid address.
[EACCES]	the reference rcd does not have sufficient access control privileges to set the desired options.
[ESCHED]	the desired scheduling objectives cannot be granted because of prior resource commitments of rcd's parent container.
[EBADPAGEVAL]	the desired user page frame count limit cannot be allowed.
[EBUFSPACE]	the desired socket buffer space limit cannot be allowed.

SEE ALSO

rc_create(1), set_rc_parent(1), close(1), sendmsg(1), recvmsg(1), get_rc_usage(1), get_rc_stats(1), get_rc(1), set_res_binding(1), add_to_sched_binding(1), delete_from_sched_binding(1), set_fd_rc(1), declare_interest(2), revoke_interest(2), get_next_event(2), bind(3).

NAME

get_rc_usage — obtain the resource usage of a container

SYNOPSIS

#include <sys/rc.h>

int **get_rc_usage**(int rcd, struct rcusage *rcu)

DESCRIPTION

get_rc_usage gets the resource usage of a resource container. In the current prototype, the following structure is produced by summarizing resource usage information that the kernel maintains for each resource container:

```
struct resource_usage {
   unsigned cpu_ticks;
   unsigned user_pages;
   unsigned sockbufspace;
};
```

These fields correspond to CPU usage, number of user-level page frames being used and the total socket buffer space being used by a container. Other types of resources may be supported in future versions.

The rcusage structure is defined as follows:

```
struct rcusage {
   int usage_val_type;
   int value;
};
```

The current prototype supports usage_val_type values corresponding to the various fields of the resource_usage structure described above.

RETURN VALUES

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and the global integer variable errno is set to indicate the error.

ERRORS

The get_rc_usage() system call fails if:

[EBADF] rcd is not a valid descriptor.

[ENOTRC] rcd is not a resource container.

[EFAULT] rcu is not a valid address.

SEE ALSO

rc_create(1), set_rc_parent(1), close(1), sendmsg(1), recvmsg(1), get_rc_opt(1), set_rc_opt(1), get_rc_stats(1), get_rc(1), set_res_binding(1), add_to_sched_binding(1), delete_from_sched_binding(1), set_fd_rc(1), declare_interest(2), revoke_interest(2), get_next_event(2), bind(3).

NAME

get_rc_stats — obtain resource usage statistics of a container

SYNOPSIS

#include <sys/rc.h>

int get_rc_stats(int rcd, struct rcusage *rcu)

DESCRIPTION

get_rc_stats gets the resource usage statistics of a resource container and its late children (see Section 8.3). In the current the kernel maintains the following structure for each container:

```
struct rc_stats {
   unsigned cpu_ticks;
   unsigned user_pages;
   unsigned sockbufspace;
};
```

As with the information returned by get_rc_usage(), these fields correspond to CPU usage, number of user-level page frames being used and the total socket buffer space being used by a container. Again, other types of resources may be supported in future versions. The operation of this system call is similar to the get_rc_usage() call.

RETURN VALUES

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and the global integer variable errno is set to indicate the error.

ERRORS

The get_rc_stats() system call fails if:

[EBADF] rcd is not a valid descriptor.

[ENOTRC] rcd is not a resource container.

[EFAULT] rcu is not a valid address.

SEE ALSO

rc_create(1), set_rc_parent(1), close(1), sendmsg(1), recvmsg(1), get_rc_opt(1), set_rc_opt(1), get_rc_usage(1), get_rc(1), set_res_binding(1), add_to_sched_binding(1), delete_from_sched_binding(1), set_fd_rc(1), declare_interest(2), revoke_interest(2), get_next_event(2), bind(3).

NAME

get_rc — get a descriptor for the current thread's resource binding

SYNOPSIS

#include <sys/rc.h>

int get_rc(void)

DESCRIPTION

This system call allows a process to obtain a descriptor for the resource container that the currently executing thread is resource bound to.

RETURN VALUES

A descriptor for the resource container in question is returned.

ERRORS

The get_rc() call fails if:

[EMFILE] The per-process descriptor table is full.

[ENFILE] The system file table is full.

SEE ALSO

rc_create(1), set_rc_parent(1), close(1), sendmsg(1), recvmsg(1), get_rc_opt(1), set_rc_opt(1), get_rc_usage(1), get_rc_stats(1), set_res_binding(1), add_to_sched_binding(1), delete_from_sched_binding(1), set_fd_rc(1), declare_interest(2), revoke_interest(2), get_next_event(2), bind(3).

NAME

set_res_binding, change_res_binding — set resource binding of a thread with or without scheduling state change

SYNOPSIS

#include <sys/rc.h>

int set_res_binding(int rcd)

int change_res_binding(int rcd)

DESCRIPTION

set_res_binding() sets the resource binding of the currently executing thread to resource container rcd. Subsequently, any resource consumption of this thread will be charged to container rcd. As described in Section 8.3, this moves the currently active resource container into the INACTIVE state, from where it moves to the READY state only after an event arrives for it. change_res_binding() is operationally identical to set_res_binding() except it does not entail a scheduling state change.

RETURN VALUES

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and the global integer variable errno is set to indicate the error.

ERRORS

The set_res_binding() system call fails if:

[EBADF] rcd is not a valid descriptor.

[ENOTRC] rcd is not a valid resource container.

SEE ALSO

rc_create(1), set_rc_parent(1), close(1), sendmsg(1), recvmsg(1), get_rc_opt(1), set_rc_opt(1), get_rc_usage(1), get_rc_stats(1), get_rc(1), add_to_sched_binding(1), delete_from_sched_binding(1), set_fd_rc(1), declare_interest(2), revoke_interest(2), get_next_event(2), bind(3).

NAME

add_to_sched_binding, delete_from_sched_binding — add/remove current thread to/from the scheduler binding set of a resource container

SYNOPSIS

#include <sys/rc.h>

int add_to_sched_binding(int rcd)

int delete_from_sched_binding(int rcd)

DESCRIPTION

add_to_sched_binding() and delete_from_sched_binding() allow the manipulation of the scheduler binding set of a resource container. After a thread is added to a container's scheduler binding set, it becomes eligible to receive CPU quanta awarded to the container by the system's CPU scheduler. The exact share of CPU obtained by the thread is dependent on the specific policies being used by the second-level scheduler associated with the container.

When the application wants to disassociate a thread from a container, it can call delete_from_sched_binding(). This happens automatically when a thread is destroyed.

RETURN VALUES

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and the global integer variable errno is set to indicate the error.

ERRORS

Both system calls fails if:

[EBADF]	rcd is not a valid descriptor.
[ENOTRC]	rcd is not a valid resource container.

SEE ALSO

rc_create(1), set_rc_parent(1), close(1), sendmsg(1), recvmsg(1), get_rc_opt(1), set_rc_opt(1), get_rc_usage(1), get_rc_stats(1), get_rc(1), set_res_binding(1), set_fd_rc(1), declare_interest(2), revoke_interest(2), get_next_event(2), bind(3).

NAME

set_fd_rc — bind a file or socket to a given resource container

SYNOPSIS

#include <sys/rc.h>

int set_fd_rc(int fd, int rcd)

DESCRIPTION

set_fd_rc() binds the file or socket fd to the resource container rcd.

After this call, any kernel processing corresponding to the object denoted by fd is charged to rcd.

RETURN VALUES

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and the global integer variable errno is set to indicate the error.

ERRORS

The set_fd_rc() system call fails if:

[EBADF]	fd is not a valid descriptor.
[EBADF2]	rcd is not a valid descriptor.
[ENFILE]	fd is not a valid file descriptor.
[ENSOCK]	fd is not a valid socket descriptor.
[ENOTRC]	rcd is not a valid resource container.

SEE ALSO

rc_create(1), set_rc_parent(1), close(1), sendmsg(1), recvmsg(1), get_rc_opt(1), set_rc_opt(1), get_rc_usage(1), get_rc_stats(1), get_rc(1), set_res_binding(1), add_to_sched_binding(1), delete_from_sched_binding(1), declare_interest(2), revoke_interest(2), get_next_event(2), bind(3).

NAME

declare_interest, revoke_interest — declare/revoke interest in events on a given file or socket

SYNOPSIS

#include <sys/events.h>

int **declare_interest**(int fd, event_t event)

int **revoke_interest**(int fd, event_t event)

DESCRIPTION

The **declare_interest()** system call asserts an application's interest in events on the object pointed to by descriptor fd. The **revoke_interest()** system call indicates that the current thread is no longer interested in events on the object denoted by descriptor fd. Essentially, these calls define a per-thread INTERESTED set of kernel objects.

Currently, the implementation of these system calls supports only network sockets. Three types of events, SOCKET_READABLE, SOCKET_WRITABLE and SOCKE-T_EXCEPTION, are supported. These correspond to the three socket event types that are supported by select(1).

RETURN VALUES

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and the global integer variable errno is set to indicate the error.

ERRORS

Both system calls fails if:

[EBADF]	fd is not a valid descriptor.
[EBADEV]	event is a bad event type for object fd

SEE ALSO

rc_create(1), set_rc_parent(1), close(1), sendmsg(1), recvmsg(1), get_rc_opt(1), set_rc_opt(1), get_rc_usage(1), get_rc_stats(1), get_rc(1), set_res_binding(1), add_to_sched_binding(1), delete_from_sched_binding(1), set_fd_rc(1), get_next_event(2), bind(3).

NAME

get_next_event — get the next pending event(s) for objects in the current thread's INTERESTED set.

SYNOPSIS

#include <sys/events.h>

int **get_next_event**(int nevents, event_t *ev_array, struct timeval *timeout)

DESCRIPTION

get_next_event() returns up to nevents pending events for objects in the calling thread's INTERESTED set. These events are returned in the ev_array array.

If no event happens for a period equal to timeout, the call returns anyway. A NULL timeout value causes **get_next_event()** to block indefinitely.

RETURN VALUES

Upon successful completion, the number of active events returned. Otherwise, a value of -1 is returned and the global integer variable errno is set to indicate the error.

ERRORS

The **get_next_event()** system call fails if:

[EFAULT]	ev_array is a bad address or is not large enough.
[EWOULDBLOCK]	the timeout expired.
[EINTR]	an interrupt occurred.

SEE ALSO

rc_create(1), set_rc_parent(1), close(1), sendmsg(1), recvmsg(1), get_rc_opt(1), set_rc_opt(1), get_rc_usage(1), get_rc_stats(1), get_rc(1), set_res_binding(1), add_to_sched_binding(1), delete_from_sched_binding(1), set_fd_rc(1), declare_interest(2), revoke_interest(2), bind(3).

NAME

bind — binds a name to a socket

SYNOPSIS

#include <sys/socket.h>

int **bind**(int so, struct sockaddr *addr, size_t addr_len);

DESCRIPTION

The **bind()** system call assigns an address to an unnamed socket. Sockets created with the **socket()** function are unnamed; they are identified only by their address family.

For full details about **bind()**, refer to the UNIX **bind(3)** man page. Here I will simply concentrate on the new sockaddr_in_filter name space.

The sockaddr_in_filter name space is described by the following structure:

```
struct sockaddr_in_filter {
   int sin_len;
   int sin_family;
   int sin_port;
   struct in_addr sin_addr;
   struct in_addr_filter filter;
   char sin_zero[8];
}
```

where the in_addr_filter structure is defined as:

```
struct in_addr_filter {
   u_int32 f_addr;
   u_int32 f_mask;
}
```

The meaning of the fields in the above structures are described in detail in Section 8.6.2. Essentially, once a filter has been specified for a particular socket, only packets that match the filter are demultiplexed to the socket.

RETURN VALUES

Upon successful completion, the bind() function returns a value of 0. If the bind() function fails, a value of -1 is returned and errno is set to indicate the error.

ERRORS

See the UNIX man page for details.

SEE ALSO

rc_create(1), set_rc_parent(1), close(1), sendmsg(1), recvmsg(1), get_rc_opt(1), set_rc_opt(1), get_rc_usage(1), get_rc_stats(1), get_rc(1), set_res_binding(1), add_to_sched_binding(1), delete_from_sched_binding(1), set_fd_rc(1), declare_interest(2), revoke_interest(2), get_next_event(2).

NAME

rsched — control the scheduling attributes of a command

SYNOPSIS

rsched <pid> <descriptor> <parameter> <value>

DESCRIPTION

The rsched command allows the manipulation of the scheduling parameters of a container by a user with appropriate privileges. The target container is identified by the (<pid>, <descriptor>) tuple. The <parameter> argument is one of CPU_LIMIT, CPU_RESERVE, CPU_PRI, MEM_USER_PAGES and MAX_SOCKBUFSPACE. These correspond to the similarly named fields in the structure that describes a resource container in the kernel. The <value> parameter specifies the new value of the <parameter> being changed.

The rsched command operates by using the ptrace() and the set_rc_opt() system calls. Thus, to use rsched on a given (<pid>, <descriptor>) tuple, the user must have the necessary privilege to execute the ptrace() system call on the process with the indicated pid.

ERRORS

rsched displays "Operation successful" if the manipulation operation succeeds. Otherwise, an appropriate error message is printed.

SEE ALSO

set_rc_opt(1).

NAME

urcd — per-user resource control daemon

SYNOPSIS

urcd [-f <config-file>] [-d]

DESCRIPTION

The urcd daemon keeps track of the top-level resource container of each user. The login program queries urcd to get a descriptor for the top-level container of the user logging in. The urcd process creates a new container for the user if one does not exist (because this user has not logged in recently).

The urcd process reads /etc/urcdtab at startup to read administrator-specified relative priorities and other resource usage parameters for each user. An alternative configuration file can be specified by -f option. The -d option enables debugging messages which are printed on the system console.

The urcd process periodically writes out the resource consumption information in each active user's top-level container to the file /etc/user_stats. This information can be queried by the useracc command.

FILES

/etc/urcdtab, /etc/user_stats.

SEE ALSO

urcdtab(4), useracc(4).

NAME

urcdtab — urcd configuration file

SYNOPSIS

/etc/urcdtab

DESCRIPTION

The /etc/urcdtab file contains descriptive information about a user's total resource usage parameters. The exact syntax of this information is as follows. Each line in /etc/urcdtab has the form:

```
<username> <parameter> <value>
```

where <username> is some login id, <parameter> is one of CPU_LIMIT, CPU_RESERVE, CPU_PRI, MEM_USER_PAGES and MAX_SOCKBUFSPACE corresponding to the similarly named attributes in the in-kernel structure describing a resource container, and ¡value¿ is the actual value of the <parameter> being specified. Multiple lines for the same <username> are allowed; the last line in /etc/urcdtab for a given (<username>, <parameter>) tuple takes precedence.

The /etc/urcdtab file can also contain a line of the form:

```
<username> TIMEOUT <value>
```

which specifies the time in minutes after which a top-level container of a user is destroyed after the last process belonging to the user terminates. A <value> of -1 indicates that the top-level container is kept around till the next system reboot.

The urcd process reads /etc/urcdtab at startime, and when a new login happens.

SEE ALSO

urcd(4).

NAME

rcstat — print information about resource container or container hierarchy

SYNOPSIS

rcstat rcstat <user id> rcstat <pid> <descriptor>

DESCRIPTION

The rcstat command prints information about the resource container hierarchy on a per-user or a system-wide basis. In the first form, rcstat prints information for the complete hierarchy of the system. In the second form, the container hierarchy of a particula user is printed. In the last form, detailed information about the specified container is displayed.

NAME

useracc — show a user's resource usage information.

SYNOPSIS

useracc [<user>]

DESCRIPTION

The useracc command displays a resource usage summary for a particular user. This information is obtained by querying the urcd daemon process.

SEE ALSO

urcd(4).

Bibliography

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the USENIX 1986 Summer Conference*, July 1986.

[2] L. W. Allen, H. G. Singh, K. G. Wallace, and M. B. Weaver. Program Loading in OSF/1. In *Proceedings of the USENIX '91 Conference*, Dallas, TX, Jan. 1991.

[3] J. Almeida, M. Dabu, A. Manikutty, and P. Cao. Providing Differentiated Quality-of-Service in Web Hosting Services. In *Proceedings of the Workshop on Internet Server Performance (held in conjunction with ACM SIGMETRICS 1998 conference)*, Madison, WI, June 1998.

[4] Compaq Computer Corporation, *AltaVista*. http://www.altavista.com/.

[5] D. P. Anderson, R. G. Herrtwich, and C. Schaefer. SRP: A Resource Reservation protocol for Guaranteed-Performance Communication in the Internet. Technical Report TR-90-006, International Computer Science Institute, Berkeley, CA, Feb. 1990.

[6] E. W. Anderson and J. Pasquale. The Performance of the Container Shipping I/O System. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, Copper Mountain, CO, Dec. 1995. ACM.

[7] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: Where have all the cycles gone? In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, St. Malo, France, Oct. 1997.

[8] Apache. http://www.apache.org/.

[9] M. J. Bach. *The Design of the UNIX Operating System.* Prentice-Hall, Englewood Cliffs, NJ, 1986.

[10] G. Banga. The Design and Implementation of a New Network Subsystem Architecture for Server Systems. Master's thesis, Rice University, Jan. 1998.

[11] G. Banga, F. Douglis, and M. Rabinovich. Optimistic Deltas for WWW Latency Reduction. In *Proceedings of the USENIX 1997 Annual Technical Conference*, Anaheim, CA, Jan. 1997.

[12] G. Banga and P. Druschel. Measuring the Capacity of a Web Server under Realistic Loads. *World Wide Web (Special issue on World Wide Web Characterization and Performance Evaluation*, 1999. To Appear.

[13] G. Banga, P. Druschel, and J. C. Mogul. Better operating system features for faster network servers. In *Proceedings of the Workshop on Internet Server Performance (held in conjunction with the ACM SIGMETRICS 1998 conference)*, Madison, WI, June 1998.

[14] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, New Orleans, LA, Feb. 1999.

[15] G. Banga and J. C. Mogul. Scalable kernel performance for Internet servers under realistic loads. In *Proceedings of the USENIX 1998 Annual Technical Conference*, New Orleans, LA, June 1998.

[16] G. Banga, J. C. Mogul, and P. Druschel. A scalable and explicit event delivery mechanism for UNIX. In *Proceedings of the USENIX 1999 Annual Technical Conference*, Monterey, CA, June 1999.

[17] A. Bas, V. Buch, W. Vogels, and T. von Eicken. U-Net: A user-level network interface for parallel and distributed computing. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, Copper Mountain, CO, Dec. 1995.

[18] S. M. Bellovin. Security Problems in the TCP/IP Protocol Suite. *Computer Communication Review*, 19(2):32–48, Apr. 1989.

[19] J. C. Bennett and H. Zhang. Hierarchical Packet Fair Queueing Algorithms. *IEEE/ACM Transactions on Networking*, 5(5):675–689, Oct. 1997.

[20] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext Transfer Protocol – HTTP/1.0. RFC 1945, May 1996.

[21] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, Copper Mountain, CO, Dec. 1995. ACM.

[22] D. Black, S. Blake, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An Architecture for Differentiated Services. Work in Progress (Internet Draft), May 1998.

[23] D. Black, J. Carter, G. Feinberg, R. MacDonald, J. V. Sciver, P. Wang, S. Mangalat, and E. Shienbrood. OSF/1 Virtual Memory Improvements. In *Proceedings of the Second USENIX Mach Symposium*, Monterey, CA, Nov. 1991.

[24] D. L. Black, D. B. Golub, D. P. Julin, R. F. Rashid, R. P. Draves, R. W. Dean, A. Forin, J. Barrera, H. Tokuda, G. Malan, and D. Bohman. Microkernel operating system architecture and Mach. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, Seattle, WA, Apr. 1992.

[25] A. C. Bomberger and N. Hardy. The KeyKos Nanokernel Architecture. In *Proceedings of the USENIX Workshop on Micro-kernels and Other Kernel Architectures*, Seattle, WA, Apr. 1992.

[26] J. Bruno, E. Gabber, B. Ozden, and A. Silberschatz. The Eclipse Operating System: Providing Quality of Service via Reservation Domains. In *Proceedings of the USENIX 1998 Annual Technical Conference*, New Orleans, LA, June 1998.

[27] M. Burrows. Personal communication, Mar. 1998.

[28] P. Cao. *Application Controlled File Caching and Prefetching*. PhD thesis, Princeton University, Jan. 1996.

[29] The Common Gateway Interface. http://hoohoo.ncsa.uiuc.edu/cgi/.

[30] S. Chandra, B. Richards, and J. Larus. Teapot: Language Support for Writing Memory Coherence Protocols. In *Proceedings of the 1996 ACM SIGPLAN Symposium on Programming Language Design and Implementation*, Philadelphia, PA, May 1996.

[31] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A Hierarchical Internet Object Cache. In *Proceedings of the USENIX 1996 Annual Technical Conference*, San Diego, CA, Jan. 1996.

[32] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska. Sharing and protection in a single-address-space operating system. *ACM Transactions on Computer Systems*, 12(4):217–252, Nov. 1994.

[33] J. B. Chen and B. N. Bershad. The impact of operating system structure on memory system performance. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, Asheville, NC, Dec. 1993.

[34] D. Chess and G. Waldbaum. The VM/370 resource limiter. *IBM Systems Journal*, 20(4):424–437, 1981.

[35] Cisco Systems Inc. LocalDirector. http://www.cisco.com/.

[36] D. D. Clark. The structuring of systems using upcalls. In *Proceedings of the Tenth ACM Symposium on Operating System Principles*, Orcas Island, WA, Dec. 1985.

[37] D. D. Clark, S. Shenker, and L. Zhang. Supporting real-time applications in an integrated services packet network: Architecture and mechansim. In *Proceedings of the ACM SIGCOMM 1992 Conference*, Baltimore, MD, Aug. 1992.

[38] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. In *Proceedings of the ACM SIGCOMM 1990 Symposium*, Philadelphia, PA, Sept. 1990.

[39] R. K. Clark, E. D. Jensen, and F. D. Reynolds. An architectural overview of the alpha real-time distributed kernel. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, Seattle, WA, Apr. 1992.

[40] Compaq Computer Corporation, Tuning Parameters for Web Servers. http://www.digital.com/info/internet/document/ias/tuning.html.

[41] Compaq Computer Corporation, Digital UNIX. http://www.unix.digital.com/.

[42] Computer Emergency Response Team (CERT). TCP SYN Flooding and IP Spoofing, CERT Advisory CA-96.21. http://www.cert.org/ftp/cert_advisories-/CA-96.21.tcp_syn_flooding, Sept. 1996, Last updated: Aug. 1998.

[43] F. J. Corbato. A Paging Experiment with the Multics System. Project MAC Memo MAC-M-384, July 1968.

[44] H. Custer. *Inside Windows NT.* Microsoft Press, Redmond, WA, 1993.

[45] M. Dahlin, R. Yang, T. Anderson, and D. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, Monterey, CA, Nov. 1994.

[46] O. P. Damani, P.-Y. E. Chung, Y. Huang, C. K. intala, and Y.-M. Wang. ONE-IP: Techniques for hosting a service on a cluster of machines. *Computer Networks and ISDN Systems*, 29:1019–1027, 1997.

[47] S. E. Deering. Host Extensions for IP Multicasting. RFC 1112, Aug. 1989.

[48] J. M. Denham, P. Long, and J. A. Woodward. DEC OSF/1 Symmetric Multiprocessing. *Digital Technical Journal*, 6(3):29–43, Summer 1994.

[49] P. Denning. Third generation computer systems. *ACM Computing Surveys*, 3(4):261–275, Dec. 1971.

[50] P. Druschel. Operating systems support for high-speed networking. Technical Report TR 94-24, Department of Computer Science, University of Arizona, Oct. 1994.

[51] P. Druschel, M. B. Abbott, M. A. Pagels, and L. L. Peterson. Network subsystem design. *IEEE Network*, 7(4):8–17, July 1993.

[52] P. Druschel and G. Banga. Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, Seattle, WA, Oct. 1996.

[53] P. Druschel, B. S. Davie, and L. L. Peterson. Experiences with a high-speed network adaptor: A software perspective. In *Proceedings of the ACM SIGCOMM 1994 Conference*, London, UK, Aug. 1994.

[54] P. Druschel and L. L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, pages 189–202, Dec. 1993.

[55] P. Druschel, L. L. Peterson, and N. C. Hutchinson. Beyond micro-kernel design: Decoupling modularity and protection in Lipto. In *Proceedings of the Twelth International Conference on Distributed Computing Systems*, Yokohama, Japan, June 1992.

[56] M. C. Easton and P. A. Franaszek. Use Bit Scanning in Replacement Decisions. *IEEE Transactions on Computing*, 28(2):133–141, Jan. 1973.

[57] A. Edwards, G. Watson, J. Lumley, D. Banks, C. Calamvokis, and C. Dalton. User-space protocols deliver high performance to applications on a low-cost Gb/s LAN. In *Proceedings of the ACM SIGCOMM 1994 Conference*, London, UK, Aug. 1994.

[58] D. Engler and M. F. Kaashoek. DPF: Fast, flexible message demultiplexing using dynamic code generation. In *Proceedings of the ACM SIGCOMM 1996 Conference*, Palo Alto, CA, Aug. 1996.

[59] D. R. Engler, S. K. Gupta, and M. F. Kaashoek. AVM: Application-Level Virtual Memory. In *Proceedings of 1995 Workshop on Hot Topics in Operating Systems*, Orcas Island, WA, May 1995.

[60] D. R. Engler, D. K. Lowenthal, and G. R. Andrews. Shared Filaments: Efficient Fine-Grain Parallelism on Shared-Memory Multiprocessors. Technical Report TR 93-13a, Department of Computer Science, University of Arizona, Tucson, AZ, 1993.

[61] Open Market. FastCGI Specification. http://www.fastcgi.com/.

[62] M. J. Feeley, W. E. Morgan, F. H. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath. Implementing global memory management in a workstation cluster. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, Copper Mountain, CO, Dec. 1995.

[63] D. Ferrari and D. C. Verma. A Scheme for Real-Time Channel Establishment in Wide-Area Networks. *IEEE Journal on Selected Areas in Communication*, 8(3):368–379, Apr. 1990.

[64] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2068, Jan. 1997.

[65] S. Floyd and V. Jacobson. Link-sharing and Resource Management Models for Packet Networks. *IEEE/ACM Transactions on Networking*, 3(4):365–386, Aug. 1995.

[66] B. Ford and J. Lepreau. Evolving Mach 3.0 to a migrating thread model. In *Proceedings of the USENIX 1994 Winter Conference*, San Francisco, CA, Jan. 1994.

[67] B. Ford and S. Susarala. CPU Inheritance Scheduling. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, Seattle, WA, Oct. 1996.

[68] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based scalable network services. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, St. Malo, France, Oct. 1997.

[69] D. Golub, R. Dean, A. Forin, and R. Rashid. UNIX as an Application Program. In *Proceedings of the USENIX 1990 Summer Conference*, Anaheim, CA, June 1990.

[70] P. Goyal, X. Guo, and H. M. Vin. A Hierarchical CPU Scheduler for Multimedia Operating Systems. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, Seattle, WA, Oct. 1996.

[71] D. Grunwald. Personal communication, Mar. 1998.

[72] D. Grunwald and R. Neves. Whole-Program Optimization for Time and Space Efficient Threads. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, Oct. 1996.

[73] G. Hamilton and P. Kougiouris. The Spring nucleus: A microkernel for objects. In *Proceedings of the USENIX 1993 Summer Conference*, Cincinatti, OH, June 1993.

[74] N. Hardy. The KeyKos Architecture. *Operating System Review*, Sept. 1985.

[75] K. Harty and D. R. Cheriton. Application-Controlled Physical Memory using External Page-Cache Replacement. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, Oct. 1992.

[76] J. L. Hellerstein. Achieving service rate objectives with decay usage scheduling. *IEEE Transactions on Software Engineering*, 19(8):813–825, Aug. 1993.

[77] G. J. Henry. The fair share scheduler. *AT&T Bell Laboratories Technical Journal*, 63(8):1845–1857, Oct. 1984.

[78] J. C. Hu, I. Pyrali, and D. C. Schmidt. Measuring the impact of event dispatching and concurrency models on web server performance over high-speed networks. In *Proceedings of the Second Global Internet Conference*, Phoenix, AZ, Nov. 1997.

[79] IBM Corporation. IBM interactive network dispatcher. http://www.ics.raleigh.ibm.com/ics/isslearn.htm/.

[80] Inktomi Corporation, *Traffic Server*. http://www.inktomi.com/products/traffic/.

[81] V. Jacobson. Congestion avoidance and control. In *Proceedings of the ACM SIGCOMM 1988 Conference*, Stanford, CA, Aug. 1988.

[82] V. Jacobson, R. T. Braden, and D. A. Borman. TCP Extensions for High Performance. RFC 1323, May 1992.

[83] D. B. Johnson and W. Zwaenepoel. The Peregrine High Performance RPC system. *Software—Practice and Experience*, 23(2):201–221, Feb. 1993.

[84] M. B. Jones, J. S. B. III, A. Forin, P. J. Leach, D. Rosu, and M.-C. Rosu. An overview of the rialto real-time architecture. In *Proceedings of the Seventh ACM SIGOPS European Workshop*, Connemara, Ireland, Sept. 1996.

[85] M. B. Jones, P. J. Leach, R. P. Draves, and J. S. Barrera. Modular real-time resource management in the Rialto operating system. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, Orcas Island, WA, May 1995.

[86] M. B. Jones, D. Rosu, and M.-C. Rosu. CPU reservations and time constraints: Efficient, predictable scheduling of independent activities. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, St. Malo, France, Oct. 1997.

[87] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. Briceno, R. Hunt, D. Mazieres, T. Pinckney, R. Grimm, J. Janotti, and K. Mackenzie. Application performance and flexibility on Exokernel systems. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, St. Malo, France, Oct. 1997.

[88] M. F. Kaashoek, D. R. Engler, G. R. Ganger, and D. A. Wallach. Server Operating Systems. In *Proceedings of the Seventh ACM SIGOPS European Workshop*, Connemara, Ireland, Sept. 1996.

[89] J. Kay and P. Lauder. A fair share scheduler. *Communications of the ACM*, 31(1):44–55, Jan. 1989.

[90] K. Krueger, D. Loftesness, A. Vahdat, and T. Anderson. Tools for the Development of Application-Specific Virtual Memory Management. In *Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, Washington, DC, Oct. 1993.

[91] C. Lee, K. Yoshida, C. Mercer, and R. Rajkumar. Predictable communication protocol processing in real-time Mach. In *Proceedings of Second IEEE Real-time Technology and Applications Symposium*, Boston, MA, June 1996.

[92] I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Braham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communication*, 14(7):1280–1297, Sept. 1996.

[93] H. Levy, G. Voelker, A. Karlin, E. Anderson, and T. Kimbrel. Implementing Cooperative Prefetching and Caching in a Globally-Managed Memory System.

In *Proceedings of the ACM SIGMETRICS '98 Conference*, Madison, WI, June 1998.

[94] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM*, 20(1):46–61, Jan. 1973.

[95] S. LoVerso, N. Paciorek, A. Langerman, and G. Feinberg. The OSF/1 UNIX Filesystem (UFS). In *Proceedings of the USENIX 1991 Conference*, Dallas, TX, Jan. 1991.

[96] C. Maeda and B. N. Bershad. Protocol service decomposition for high-performance networking. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, Asheville, NC, Dec. 1993.

[97] C. Maltzahn, K. J. Richardson, and D. Grunwald. Performance Issues of Enterprise Level Web Proxies. In *Proceedings of the ACM SIGMETRICS 1997 Conference*, Seattle, WA, June 1997.

[98] S. Mangalat and D. Bolinger. Parallelizing Signal Handling and Process Management in OSF/1. In *Proceedings of the USENIX 1991 Conference*, Dallas, TX, Nov. 1991.

[99] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Proceedings of the USENIX 1993 Winter Conference*, San Diego, CA, Jan. 1993.

[100] R. E. McGrath. Performance of Several HTTP Demons on an HP 735 Workstation. http://www.ncsa.uiuc.edu/InformationServers/Performance/V1.4/report.html, Apr. 1995.

[101] P. E. McKenney and K. F. Dove. Efficient Demultiplexing of Incoming TCP Packets. In *Proceedings of the ACM SIGCOMM 1992 Conference*, Baltimore, MD, Aug. 1993.

[102] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System.* Addison-Wesley, Reading, MA, 1996.

[103] D. McNamee and K. Armstrong. Extending the Mach External Pager Interface to Accomodate User-Level Page Replacement Policies. In *Proceedings of the USENIX Mach Symposium*, Burlington, VT, Oct. 1990.

[104] C. W. Mercer. *Operating System Resource Reservation for Real-Time and Multimedia Applications*. PhD thesis, Carnegie Mellon University, June 1997.

[105] C. W. Mercer, S. Savage, and H. Tokuda. Processor Capacity Reserves for Multimedia Operating Systems. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, Boston, MA, May 1994.

[106] Microsoft Corporation, ISAPI Overview. http://www.microsoft.com/msdn/sdk/platforms/doc/sdk/internet/src/isapimrg.htm.

[107] D. W. Mitchell. Mach Resource Control in OSF/1. In *The Second USENIX Mach Symposium Conference Proceedings*, Monterey, CA, Nov. 1991.

[108] J. C. Mogul. Observing TCP Dynamics in Real Networks. In *Proceedings of the ACM SIGCOMM 1992 Conference*, pages 281–292, Aug. 1993.

[109] J. C. Mogul. Network behavior of a busy web server and its clients. Technical Report WRL 95/5, Compaq Western Research Laboratory, Palo Alto, CA, 1995.

[110] J. C. Mogul. Operating System Support for Busy Internet Servers. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, Orcas Island, WA, May 1995.

[111] J. C. Mogul and K. K. Ramakrishnan. Eliminating Receive Livelock in an Interrupt-driven Kernel. *ACM Transactions on Computer Systems*, 15(3):217–252, Aug. 1997.

[112] J. C. Mogul, R. F. Rashid, and M. J. Accetta. The Packet Filter: An Efficient Mechanism for User-level Network Code. In *Proceedings of the Eleventh ACM Symposium on Operating System Principles*, Austin, TX, Nov. 1987.

[113] A. B. Montz, D. Mosberger, S. W. O'Malley, L. L. Peterson, and T. A. Proebsting. Scout: A communications-oriented operating system. In *Proceedings of 1995 Workshop on Hot Topics in Operating Systems*, Orcas Island, WA, May 1995.

[114] D. Mosberger. *Scout: A Path-based Operating System.* PhD thesis, University of Arizona, Tucson, July 1997.

[115] D. Mosberger, L. Peterson, P. Bridges, and S. O'Malley. Analysis of Techniques to Improve Protocol Latency. In *Proceedings of the ACM SIGCOMM 1996 Conference*, Palo Alto, CA, Aug. 1996.

[116] D. Mosberger and L. L. Peterson. Making paths explicit in the scout operating system. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, Seattle, WA, Oct. 1996.

[117] K. Nahrstedt and J. M. Smith. The QoS Broker. *IEEE Multimedia*, 2(1):53–67, Spring 1995.

[118] E. Nahum, T. Barzilai, and D. Kandlur. Evaluating High Performance Socket APIs on AIX. *submitted for publication*, Feb. 1998.

[119] NCSA httpd. http://hoohoo.ncsa.uiuc.edu/.

[120] Network Appliance Inc., *Data ONTAP*. http://www.netapp.com/level3/ont-ap.html.

[121] Network Appliance Inc., *NetCache*. http://www.netapp.com/level3/netcach-e/datasheet.html.

[122] Network Appliance Inc., *Web Filer*. http://www.netapp.com/level3/webfile-r.html.

[123] Netscape Inc., Netscape Server API. http://www.netscape.com/newsref/std/s-erver_api.html.

[124] C. A. Nicolaou. *A Distributed Architecture for Multimedia Communication Systems*. PhD thesis, University of Cambridge, 1991.

[125] North American Network Operators Group (NANOG). Mailing List Archives, Thread #01974. http://www.merit.edu/mail.archives/html/nanog/threads.h-tml#01974, Apr. 1998.

[126] Open Software Foundation. *The Design of the OSF/1 Operating System*. Prentice-Hall, Englewood Cliffs, NJ, 1993.

[127] J. Ousterhout. Why Threads Are A Bad Idea (for most purposes). Invited talk at the USENIX 1996 Annual Technical Conference. http://www.scriptics.com/people/john.ousterhout/threads.ps.

[128] J. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Proceedings of the USENIX 1990 Summer Conference*, Anaheim, CA, June 1990.

[129] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-aware request distribution in cluster-based network servers. In *Proceedings of the 8th Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct. 1998.

[130] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: A unified I/O buffering and caching system. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, New Orleans, LA, Feb. 1999.

[131] V. S. Pai, P. Druschel, and W. Zwanaepoel. Flash: An efficient and portable Web server. In *Proceedings of the USENIX 1999 Annual Technical Conference*, Monterey, CA, June 1999.

[132] R. Pandey, J. F. Barnes, and R. Olsson. Supporting Quality of Service in HTTP servers. In *Proceedings of the Seventeenth Annual SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Puerto Vallarta, Mexico, June 1998.

[133] J. B. Postel. DARPA Internet Protocol Specification. RFC 791, Sept. 1981.

[134] J. B. Postel. Transmission Control Protocol. RFC 793, Sept. 1981.

[135] C. Pu, T. Autry, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang. Optimistic Incremental Specialization: Streamlining a Commercial Operating System. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, Copper Mountain, CO, Dec. 1995.

[136] S. A. Rago. *UNIX System V Network Programming.* Addison-Wesley, Reading, MA, 1993.

[137] K. K. Ramakrishnan. Scheduling issues for interfacing to high speed networks. In *Proceedings of the IEEE Global Telecommunications Conference*, Orlando, FL, Dec. 1992.

[138] Y. Rekhter and T. Li. An Architecture for IP Address Allocation with CIDR. RFC 1518, Sept. 1993.

[139] J. Richter. *Advanced Windows.* Microsoft Press, Redmond, WA, 1997.

[140] D. M. Ritchie and K. Thompson. The Unix time-sharing system. *Communications of the ACM*, 17(7):365–375, July 1974.

[141] R. Schardt. An MVS tuning approach (OS problem solving). *IBM Systems Journal*, 19(1):102–119, 1980.

[142] S. E. Schechte and J. Sutaria. A Study of the Effects of Context Switching and Caching on HTTP Server Performance. http://www.eecs.harvard.edu/~stuart/Tarantula/FirstPaper.html.

[143] C. L. Schuba, I. Krsul, M. Kuhn, E. Spafford, A. Sundaram, and D. Zamboni. Analysis of denial of service attacks on TCP. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, Los Alamitos, CA, May 1997.

[144] S. Sechrest and Y. Park. User-Level Physical Memory Management for Mach. In *Proceedings of the Second USENIX Mach Symposium*, Monterey, CA, Nov. 1991.

[145] M. Seltzer, Y. Endo, C. Small, and K. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, Seattle, WA, Oct. 1996.

[146] J. Semke and J. M. M. Mathis. Automatic TCP Buffer Tuning. In *Proceedings of the ACM SIGCOMM 1998 Conference*, Vancouver, British Columbia, Sept. 1998.

[147] S. Shenker, L. Zhang, and D. Clark. Some Observations on the Dynamics of a Congestion Control Algorithm. *ACM Computer Communication Review*, 20(4):30–39, Oct. 1990.

[148] C. Small and M. I. Seltzer. Vino: An Integrated Platform for Operating Systems and Database Research. Technical Report TR-30-94, Department of Computer Science, Harvard University, Cambridge, MA, 1994.

[149] Sun Microsystems Inc., Solaris 7 Operating Environment. http://www.sun.com/solaris/.

[150] O. Spatscheck and L. L. Petersen. Defending Against Denial of Service Attacks in Scout. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, New Orleans, LA, Feb. 1999.

[151] Squid. http://squid.nlanr.net/Squid/.

[152] W. Stevens. *TCP/IP Illustrated Volume 1.* Addison-Wesley, Reading, MA, 1994.

[153] W. R. Stevens. *UNIX Network Programming.* Prentice-Hall, Englewood Cliffs, NJ, 1990.

[154] W. R. Stevens. *Advanced Programming in the UNIX environment.* Addison-Wesley, Reading, MA, 1992.

[155] W. R. Stevens. *TCP/IP Illustrated Volume 3.* Addison-Wesley, Reading, MA, 1996.

[156] T. Teorey. Properties of Disk Scheduling Policies in Multiprogrammed Computer Systems. In *Proceedings of the AFIPS Fall Joint Conference*, 1972.

[157] T. J. Teorey and T. B. Pinkerton. A comparative analysis of disk scheduling policies. *Communications of the ACM*, 15(3), Mar. 1972.

[158] M. N. Thadani and Y. A. Khalidi. An efficient zero-copy I/O framework for UNIX. Technical Report SMLI TR-95-39, Sun Microsystems Laboratories, Inc., May 1995.

[159] The Open Group, Woburn, MA. The Single UNIX Specification, Version 2 - 6 Vol set for UNIX 98, 1997.

[160] C. Thekkath, T. Nguyen, E. Moy, and E. Lazowska. Implementing network protocols at user level. In *Proceedings of the ACM SIGCOMM 1993 Symposium*, Ithaca, NY, Sept. 1993.

[161] thttpd. http://www.acme.com/software/thttpd/.

[162] H. Tokuda, T. Nakajima, and P. Rao. Real-Time Mach: Toward a Predictable Real-Time System. In *Proceedings of the USENIX Mach Symposium*, Burlington, VT, Oct. 1990.

[163] B. Totty. Personal communication, Mar. 1998.

[164] A. Vahdat, E. Belani, P. Eastham, C. Yoshikawa, T. Anderson, D. Culler, and M. Dahlin. WebOS: Operating System Services For Wide Area Applications. In *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*, Chicago, IL, July 1998.

[165] B. Verghese, A. Gupta, and M. Rosenblum. Performance Isolation: Sharing and Isolation in Shared-Memory Multiprocessors. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct. 1998.

[166] T. von Eicken, A. Basu, and V. Buch. Low-latency communication over ATM networks using active messages. *IEEE Micro*, 15(1):46–53, Feb. 1995.

[167] The World Wide Web Consortium. http://www.w3.org/.

[168] C. A. Waldspurger. *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Managament.* PhD thesis, Massachusetts Institute of Technology, Sept. 1995.

[169] C. A. Waldspurger and W. E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, Monterey, CA, Nov. 1994.

[170] L. C. Wolf and R. G. Herrtwich. The System Architecture of the Heidelberg Transport System. *ACM Operating Systems Review*, 28(2):51–64, Apr. 1994.

[171] G. R. Wright and W. R. Stevens. *TCP/IP Illustrated Volume 2*. Addison-Wesley, Reading, MA, 1995.

[172] J. Wroclawski. The use of RSVP with IETF Integrated Services. RFC 2210, Sept. 1997.

[173] M. Yuhara, B. N. Bershad, C. Maeda, and J. E. Moss. Efficient packet demultiplexing for multiple endpoints and large messages. In *Proceedings of the USENIX 1994 Winter Conference*, San Francisco, CA, Jan. 1994.

[174] Zeus. http://www.zeus.co.uk/.

[175] H. Zhang. *Service Disciplines for Packet-Switching Integrated-Services Networks*. PhD thesis, University of California, Berkeley, 1993.

[176] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala. RSVP: A New Resource ReSerVation Protocol. *IEEE Network*, Sept. 1993.