A Mechanism for Scalable Event Notification and Delivery in Linux

by

Michal Ostrowski

A thesis

presented to the University of Waterloo

in fulfilment of the

thesis requirement for the degree of

M.Math

in

Computer Science

Waterloo, Ontario, Canada, 2000

I hereby declare that I am the sole author of this thesis.

I authorize the University of Waterloo to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the University of Waterloo to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

Previous research has shown that the strategy chosen by server applications for managing multiple simultaneous connections may have a significant impact on a server's behavior. This thesis examines commonly used mechanisms for communicating state and event information between the kernel and application and their effect on application structure and behavior. From this examination we find that there is potential for improving application scalability characteristics through the use of an improved event delivery and notification mechanism.

We propose a new mechanism for event delivery and notification in Linux that is designed with efficiency and scalability in mind. We proceed to apply this new mechanism to a web-server and present a series of experiments demonstrating that our web-server is capable of nearly doubling the throughput of a similar server based on polling mechanisms when the request rate exceeds what the server can handle.

Acknowledgements

I would like to thank the many people who directly and indirectly, knowingly and unknowingly made this thesis what it is.

I would like to thank my advisor, Tim Brecht, for allowing me to explore my interests, providing valuable focus to this work, listening to many crazy theories and ideas, and in general surviving the process of extracting thesis text out of operating system code.

The members of the Shoshin research group at the University of Waterloo also deserve my thanks for providing me with an enjoyable and stimulating work environment. In particular, I would like to thank Jay Black for his valuable comments, questions and support.

This research was also made possible by an equipment grant from the National Science and Engineering Research Council.

Finally, I would like to thank my friends and family for their patience, support and for providing welcome distractions from my work.

Contents

List of Figures

Chapter 1

Introduction and Motivation

1.1 Problem Statement

The aim of this thesis is to improve the ability of an operating system and web-server to efficiently handle a large number of simultaneous client connections through the use of a preemptive event-notification mechanism.

1.2 Motivation

Many applications are referred to as being "event-driven" by virtue of the fact that they perform actions in response to changes in their environment, that are often presented to them by the operating system. However, many such applications rely on a kernel-polling mechanism to learn of events of interest. Such applications are not truly driven by events, since events in the operating system and hardware do not trigger the application to respond immediately. In this thesis, we consider the design of a new mechanism in Linux for communicating event information from the kernel to an application that includes preemptive event-notification. Our goal is to enable truly event-driven applications with a mechanism that does not suffer from the limitations of existing mechanisms (e.g., POSIX signals and its limitation of delivering events one at a time).

This thesis presents and evaluates a new mechanism for propagating event information through the Linux kernel into applications. There are several types of applications that may benefit from a better event propagation mechanism such as garbage collectors [8, 19], persistent storage systems [32] and distributed shared memory [23]. We focus our discussion on the Linux operating system and simple "event-driven" web-servers and the kinds of events of interest to them (events providing information of activity on TCP/IP sockets). Such web-servers present a natural opportunity to evaluate our mechanism and despite the significant interest within the research community that this specific problem domain has generated [9, 10, 27, 18], we still see room for improvement.

1.2.1 Multiplexing Connections in Web-Servers

The Internet has seen phenomenal growth since its creation. This growth has seen systems initially designed to work in small academic or corporate environments being pressed into use in a world-wide public network with millions of users. The majority of today's Internet servers run on UNIX [24], an operating system originating in the 1970's. There is evidence to show that the mechanisms and interfaces of UNIX systems for managing multiple connections scale poorly with respect to the number of clients served by Internet-server applications [9, 10, 27].

Web-servers are essential to the World-Wide-Web [12]; their role being to service the requests of remote clients using the Hyper-Text Transfer Protocol (HTTP) [16]. Each client conducts an HTTP dialog with the server, requesting and receiving data over a TCP/IP connection. Each TCP/IP connection is represented in the web-server application with a file-descriptor that the web-server uses to communicate with the client machine [29].

As the rate of client requests increases, a web-server must deal with an increasing number of simultaneous connections. In order to handle these connections efficiently, the web-server must use some mechanism provided by the operating system to determine which connections (file-descriptors) require attention. We refer to such mechanisms as "multiplexing mechanisms"[1] since they enable the multiplexing of numerous connections within a single thread. A multiplexing mechanism allows for an application to switch among connections in order to avoid blocking and waiting for the completion of an I/O operation. The choice and performance of the multiplexing mechanism for a web-server has a significant impact on the web-server's ability to handle a large number of simultaneous connections [18].

In particular, it has been argued that web-servers based on an event-driven architecture have the best potential for scaling with higher client loads [18, 27, 7]. Such servers may use a small number of kernel-threads to take advantage of multiple processors, but otherwise eschew kernel-based multi-threading in favor of multiplexing multiple connections within a single kernel-thread. Event-driven application architectures have been implemented using a variety of Linux kernel mechanisms; `select()` [29], `poll()` [29], `/dev/poll` [27] or POSIX Real-Time Signals [21]. We examine these mechanisms and identify the advantages and disadvantages each presents in a web-server attempting to handle a large number of simultaneous client requests.

Even though POSIX Real-Time Signals [21] have been considered for use in web-servers [4, 27], they have not been used in a preemptive manner due to the inefficiencies of preemptive signal delivery on a large-scale. In this thesis, we develop a new event-notification mechanism designed to efficiently deliver large numbers of events to applications using preemptive notification. To examine the utility of this mechanism, we investigate the potential benefits of preemptive event-notification in web-servers since this seems to be a natural fit with an event-driven application architecture. Surprisingly little attention has been given to preemptive notification schemes in this problem domain [27, 10].

Our interest in preemptive event-notification arises from the fact that such a scheme is required to support a truly "event-driven" application. The alternative to preemptive event-notification is polling the kernel,[2] which requires that the application must explicitly recognize that it needs new events and ask the kernel to provide them. Such an approach is not truly event-driven since an operating system event does not drive the application; the application responds to the event but only of its own discretion.

1.3 Contributions

This thesis makes the following contributions:

- We present a new mechanism for event notification, called Scalable Event Notification and Delivery (SEND) for use in applications relying on frequent event notifications (we refer to scalability with respect to the frequency and number of events). SEND allows the operating system to deliver multiple events to an applications simultaneously in an efficient manner along with providing a preemptive notification. The efficiency of this mechanism arises from the use of special shared memory regions (shared between the kernel and application) that eliminate the need for system calls to perform frequent configuration operations (e.g., enabling and disabling of preemptive event notification) or obtaining new event data.

- In the process of developing the SEND mechanism we define the concepts of "event-delivery" and "event-notification". We show that it is beneficial to distinguish between these two concepts whenever we consider any mechanism that is used by applications to obtain event information. We demonstrate that by treating these concepts separately we are able to design mechanisms and applications capable of handling large numbers of events simultaneously.

[1]The term "concurrency strategy" is sometimes associated with this concept though we refrain from its use since we wish to include other mechanisms that may not necessarily include concurrency.

[2]Unless otherwise specified further references to "polling" refer to polling the operating system with a system call.

- Creating truly event-driven applications requires more of the operating system than simply providing an event notification mechanism. We identify problems in the Linux sockets interface that make it difficult for us to create a truly event-driven application. To solve these problems, we present an event-driven mechanism as a replacement for the `accept()` system call. With our mechanism we avoid certain race conditions and can remove all polling from our web-server thereby always maintaining purely event-driven semantics. We use this mechanism in conjunction with SEND to demonstrate that the manner in which an application schedules accept operations (e.g., calls to `accept()`) has a significant impact on web-server performance.
- In order to demonstrate the benefits of our mechanism, we compare the behavior of a web-server using the most efficient polling-based multiplexing mechanism available in Linux to the behavior of the same web-server modified to use our purely event-driven interface and mechanism. Our results show that when compared to a polling-based strategy, a purely event-driven web-server is capable of handling substantially higher client request rates without a substantial degradation in performance.

1.4 Outline

Chapter 2 discusses a variety of polling and event-driven multiplexing mechanisms, focusing on the scalability issues concerning each mechanism. From these discussions we arrive at a set of guidelines for use in our design of SEND, which is presented in Chapter 3. We discuss the approach to interactions between the operating system and applications that this mechanism presents and how we address potential system-security issues arising from the kernel allowing direct access by applications to some data structures. Chapter 4 describes the modifications required of an existing web-server to use SEND and present a series of experiments to show that this web-server is better suited to gracefully handling a high rate of client requests. Finally, Chapter 5 presents several potential avenues for future research.

Chapter 2

Background and Related Work

This chapter considers the characteristics of the multiplexing mechanisms available in Linux, beginning with a discussion of the traditional UNIX file-descriptor polling mechanisms found in Linux: `select()` and `poll()` [29]. This discussion is followed by an examination of relevant research into the characteristics and potential improvements to these mechanisms. Our belief is that a polling-based application design faces scalability concerns not present in applications based on purely event-driven multiplexing mechanisms with some form of preemption. The existing preemptive event-notification mechanism in Linux (POSIX signals) is not designed to process large numbers of events and thus, faces significant scalability concerns of its own. In response to these issues, in Chapter 3, we propose a generic, preemptive event-notification mechanism that avoids many of the problems associated with `select()`, `poll()` and POSIX signals.

2.1 Application Design Using Polling Mechanisms

This section discusses the basic principles of designing Internet server applications using polling-based multiplexing mechanisms. Using a polling mechanism, one would write an application similar to the pseudo-code depicted in Figure 2.1.

Figure 2.1 Pseudo-Code for a Polling-Based Web-Server

```

define interest_set // file descriptors the
                    // application is interested in

define active_set   // set of active file descriptors
loop
      active_set = poll_OS_for_state( interest_set )
      for each x in active_set
              work_on_file_descriptor( x )
      end loop
end loop
```

There are three different mechanisms available in Linux to write such applications: `select()` [29], `poll()` [29] and `/dev/poll` [1] [27, 5]. Using each of these mechanisms requires the following three basic steps:

1. The application informs the operating system which file descriptors it is interested in and for what purposes (reading and/or writing).

[1] This mechanism is not currently available with the standard Linux kernel, but is available as a separate patch.

2. The application utilizes a system call provided by the polling mechanism to query the operating system about the status of the file descriptors that the application has declared interest in.

3. The operating system communicates to the application the results of the query.

Although these steps are presented individually, the different mechanisms may combine them in various ways. The sample pseudo-code in Figure 2.1 combines all three steps on line 6. Of the three mechanisms available in Linux, only the `/dev/poll` mechanism separates these steps.

Before proceeding further we would like to point out a very important aspect of the application structure presented in Figure 2.1 that is crucial in understanding the behavior of such an application. Note that after the polling operation the application goes into a loop processing the file-descriptors that have been identified as being active by the operating system. After the polling operation the application has a "batch" of work to do and it proceeds to do it before polling again. Applications may use various strategies for deciding when to poll again, but once a polling operation has completed the application's course of action is determined up to the time of the next polling operation. We refer to the work that a polling-oriented design performs between polling operations as a "batch".

2.1.1 Polling with `select()`

The `select()` system call is the traditional polling mechanism in UNIX systems [29]. This call takes as parameters three bit-maps that indicate file descriptors the application is interested in and for what purposes. Each bit-map corresponds to one of three file-descriptor states identifying whether the file-descriptor is readable without blocking, writable without blocking or has an exception pending. When the k'th bit of a bit-map is asserted, it identifies the application's interest in the condition associated with that particular bit-map as it applies to file-descriptor k.

After `select()` returns, the bit-maps that were passed to the operating system as arguments have been modified to reflect the state of the file-descriptors that the application has declared interest in. If the k'th bit of a bit-map has been set by the application, the operating system keeps the bit set if the condition associated with the particular bit-map holds true, otherwise the operating system clears the bit. In this manner the application receives answers to the queries of file-descriptor state that it has posed to the operating system when making the `select()` call.

The `select()` system call blocks until the bit-maps it would return have at least one bit set among them. This behavior may be controlled through the specification of a time-out value that sets an upper bound on how long the call may block for. The application may specify a time-out of 0 to prevent the call from blocking. An additional parameter specifies the number of bits that the kernel must scan from each bit-map.

Figure 2.2 outlines the basic structure of a server application that using `select()`. In this example, as with the others presented throughout this chapter, we are not concerned with the specific actions the application performs on each file descriptor. Instead we focus on how the various file descriptors are managed. There are several points in this regard worth noting:

- Lines 20-22 show how we must copy the file-descriptor bit-maps prior to every `select()` call, since `select()` modifies its arguments. The copying costs may be substantial, growing with the size of the interest set (i.e., the number of connections) and are responsible for a significant amount of overhead associate with `select()` [9].

- Lines 28-33 show how new connections are created; by performing an `accept()` call when a listening socket is marked as readable. Note that we do not know how many `accept()` calls we can make without blocking at any time. As a result a web-server may set the socket on which `accept()` is to be called to be non-blocking and repeatedly call `accept()` until there are no more new connections (and `accept()` thus returns with the `EWOULDBLOCK` error condition). This is good practice because it empties the kernel's queue of accepted connections allowing new connection requests to succeed while we perform lines 34-39 (which may take some time).

Figure 2.2 Outline of a `select()`-Based Web-Server

```
/* Declare and initialize file-descriptor sets */
fd_set readable, writable, exception;
FD_ZERO(&readable); FD_ZERO(&writable); FD_ZERO(&exception);
int max_fd;

/* Configure a socket to accept new connections */
sock = socket(...);
bind(sock, ... );
listen(sock, ... );

/* Indicate interest in reading/accepting on ''sock'' */
FD_SET( &readable , sock );
max_fd = sock;

while (1) {
        int i;
        fd_set r,w,e;

        /* Prepare query parameters */
        memcpy(&r, &readable, sizeof(fd_set));
        memcpy(&w, &writable, sizeof(fd_set));
        memcpy(&e, &exception, sizeof(fd_set));

        /* Execute query , infinite timeout*/
        select(max_fd+1, &r, &w, &e, NULL);

        /* First check for new connections */
        if (FD_ISSET(&r,sock)) {
                new_sock = accept(sock, ...);
                FD_SET(&readable, new_sock);
                FD_SET(&writable, new_sock);
                max_fd = MAX(max_fd, new_sock);
        }

        for (i=0; i<max_fd+1; ++i) {
                if (FD_ISSET(&r, i))
                        do_read_action(i);
                if (FD_ISSET(&w, i))
                        do_write_action(i);
        }
}
```

Note this is a simplified outline and does not address issues such as how completed connections are cleaned up (in which case `max_fd` shrinks).

2.1.2 Polling with `poll()`

The `poll()` system call performs the same function as `select()` though it provides a different interface. When using `poll()`, the process's interest in file descriptors is not described with bit-maps. Instead the caller specifies an array of `struct pollfd` objects, one for each file descriptor that it is interested in. Figure 2.3 shows the interface presented by `poll()`.

Figure 2.3 Definition of `struct pollfd` and the `poll()` Interface

```

struct pollfd {
        int fd;           /* file descriptor */
        short events;     /* bit-map of conditions polled for */
        short revents;    /* bit-map of query results */
};

int poll(struct pollfd *fds,   /* query parameters and results */
         unsigned int num_fds, /* number of elements in ``fds'' */
         int timeout);         /* wait at most
                                *  ``timeout'' milliseconds */
```

When a call to `poll()` is made, the operating system examines the object represented by the file-descriptor of each element of `fds` to see if the conditions being queried for by the `events` field are met. The results of each of these queries are expressed as a bit-map and placed in the corresponding `revents` field. If the query result is empty, the call blocks for at most the amount of time specified by the `timeout` parameter or until the state of the operating system changes and the query result is non-empty. Figure 2.4 presents a re-write of the example shown in Figure 2.2 using `poll()` instead of `select()`.

The primary advantage of `poll()` over `select()` is that it copies back to the application information about a file-descriptor only if the file-descriptor is active (i.e., read or write operations can be made without blocking). Thus the application is presented with information concerning only those file descriptors that are ready. On the other hand, applications using `select()` must check the state of each file-descriptor after a `select()` call.

The disadvantage of using `poll()` is that under certain circumstances it may require more copying of data than `select()`. This depends on the number of file descriptors in the interest set and the proportion of those that are active; `poll()` performs fewer copies if the application's interest set is large but the number of file-descriptors that are active is small. The reason for this is that with `select()` each file descriptor is represented by 3 bits, whereas each `struct pollfd` is (on 32-bit architectures) 64-bits. It is up to programmers to determine which of these calls is more efficient for their program. Applications may be written to dynamically select which call to use based on an estimate of the copying overhead that each would incur (e.g., the `thttpd` web-server [26]).

2.2 Scalability of Polling Mechanisms

In the previous section, we outlined two closely related polling mechanisms; `select()` and `poll()`. These mechanisms have been shown to scale poorly with the number of file descriptors they are handling [9, 10, 27]. The fundamental problem with these mechanisms is that as the number of file descriptors being used by an application increases so does the amount of time taken by the application to configure the polling queries and interpret the results. Likewise, the amount of time required by the operating system to perform the query increases as well. The fundamental cause is that the work required on each polling operation by an application is proportional to the number of file descriptors instead of the number of useful results (that provide new data to the application) [9]. The overhead associated with these mechanisms is significant as Banga *et al.* [10] observe: two-thirds of the time spent executing inside an operating system may be spent performing `select()` operations on behalf of a web-proxy server. By understanding the causes of this

Figure 2.4 Outline of a `poll()`-Based Web-Server

```

/* Declare and initialize file-descriptor sets */
struct pollfd fds[MAX_NUM_FDS];
int max_fd = 1;

/* Configure a socket to accept new connections */
sock = socket(...);
bind(sock, ... );
listen(sock, ... );

/* Indicate interest in reading/accepting on ‘‘sock’’ */
fds[ 0 ].fd = sock; fds[ 0 ].events = POLLIN;

while (1) {
        int i;

        /* Execute query, infinite time-out value */
        poll(fds, max_fd, -1);

        /* First check for new connections */
        if (fds[0].revents & POLLIN) {
                new_sock = accept( sock, ... );
                fds[ max_fd ].fd = new_sock;
                fds[ max_fd ].events = POLLIN | POLLOUT ;
                ++max_fd;
        }

        /* Perform appropriate actions on other file-descriptors */
        for (i=1; i<max_fd+1; ++i) {
                if (fds[ i ].revents & POLLIN)
                        do_read_action(i);
                if (fds[ i ].revents & POLLOUT)
                        do_write_action(i);
        }
}
```

Note this is a simplified outline and does not address issues such as how completed connections are cleaned up (in that case `max_fd` shrinks).

scalability problem we can explain how these concerns are overcome by mechanisms such as SEND.

The first concern raised by Banga *et al.* [9, 10] is that `select()` and `poll()` are memory-less; they do not remember (between calls) what file descriptors the application is interested in. The costs incurred by the application to configure the input arguments, by the kernel to scan the query parameters, copying data between the kernel and application (in both directions) and the application to scan the results are incurred regardless of whether or not the application's interest set has changed.

Another more fundamental problem is that these mechanisms provide a means of querying the state of file descriptors rather than informing the application of events that have occurred that affect the file-descriptor state. Banga *et al.* [10] recognize this point and in response develop terminology to describe the differences. Under this terminology `select()` and `poll()` are referred to as being state-based mechanisms as opposed to event-based mechanisms. The distinction between the two is made as follows:

event-based: Informs applications that events have occurred on file-descriptors (e.g., informs the application that a packet of data has arrived on a socket). An event-based mechanism tells an application which file-descriptors have undergone a change in state since the last notification.

state-based: Tells the application the state of a file-descriptor (e.g., a socket has data that may be read by the application immediately). A state-based mechanism tells an application which file-descriptors match certain criteria, regardless of their state at the time of the last query or notification.

The difference between these two kinds of notification mechanisms is that a state-based mechanism provides an application with all of the information about its file-descriptors with each notification (e.g., call to `select()`). On the other hand, an event-based mechanism tells the application what changes have occurred to its file-descriptors since the last notification. The advantage of the event-based based approach is that if the application's interest set is large it incurs less overhead within the kernel and application since all of the work goes toward providing the application with information it does not have.

There are two subtle points to recognize about this terminology; it does not refer to how information is conveyed between the kernel and application and it does not specify the form in which information is conveyed. This terminology is used only to characterize how a mechanism determines whether it should or should not report a file-descriptor to an application. Thus, it is possible for an event-based mechanism to be a polling mechanism that conveys state information to an application, provided that it conveys information about only those file-descriptors for those events that the application is not yet aware of.

There is a important inefficiency in state-based multiplexing mechanisms that needs consideration. Operating systems such as Linux function in an event-driven environment: system calls, exceptions and interrupts. In order to provide a state-based mechanism for applications, the operating system must translate its own event-based view of its environment into a state-based view. As shown in Figure 2.5, this calls for a substantially different flow of information through the operating system. An "event-driven" web-server must then translate the state-based view of the multiplexing mechanism back into its own event-based semantics [10].

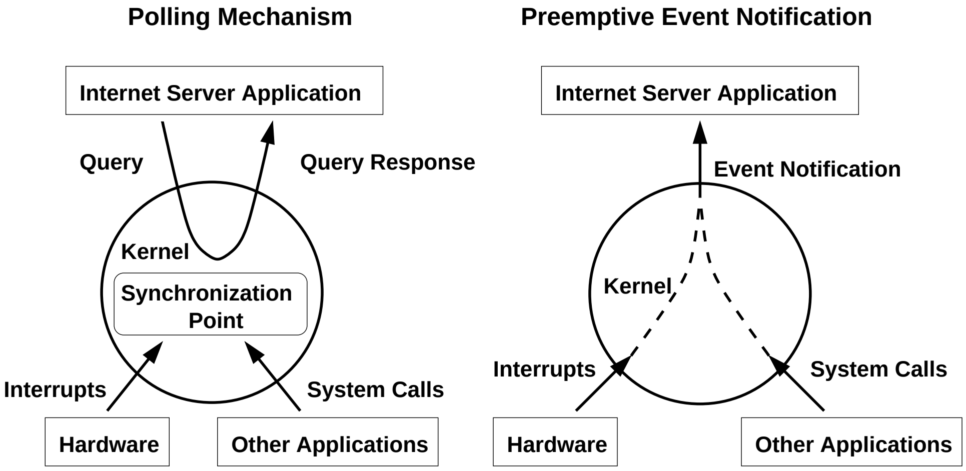
The translation of event-based semantics to state-based semantics and back serves no purpose except to conform to state-based polling interfaces. In this thesis, instead of accepting the inefficiencies of such mechanisms and interfaces, we consider event-based mechanisms that are not burdened by these inefficiencies. In developing SEND our goal is to develop a mechanism that allows for efficient propagation of event information through the operating system and into the application.

We also consider other event-based mechanisms; in the following sections we present another event-based mechanism proposed by Banga *et al.* [10] and a semantically equivalent mechanism known as `/dev/poll`. We also discuss POSIX signals, the standard event-based event-notification mechanism in UNIX/Linux. These discussions argue that despite the fact these are event-based mechanisms, there is room for improvement in these interfaces that we can exploit with our SEND mechanism.

2.3 Event-Based Polling

In light of the observations made in the previous sections, Banga *et al.* [10] have developed an event-based mechanism as an alternative to `select()` and `poll()`. This mechanism addresses the key problems pointed out in the preceding section; it is event-based and not memory-less (not requiring the application to re-specify its entire interest set with each query).

Figure 2.5 Data Flow in State-Based Polling and Preemptive Event Notification Mechanisms



This new mechanism (which was not given a name) consists of two system calls, `declare_interest()` and `get_next_event()`, that perform the following functions:

`declare_interest()`: This system call allows a process to create and manage an interest set within the kernel. An interest set is a set of file-descriptors and particular state changes for the descriptors that the application is interested in. For example, an application may use this call to inform the operating system that it is interested in being told when file-descriptor "5" may be read without blocking.

`get_next_event()`: This system call provides the application with a list of file-descriptors in a particular interest set along with bit-maps identifying the changes to their states that have occurred since the last invocation of
`get_next_event()`.

The mechanism proposed by Banga *et al.* is event-based and alleviates the key scalability concerns associated with `select()` and `poll()`. However, it is still a specialized polling mechanism. Banga *et al.* recognize this and argue that their approach is preferable to the preemptive POSIX signals mechanism [10] because

1. signal delivery is expensive due to kernel entry/re-entry costs,
2. preemptive notification requires a locking mechanism (that may introduce inefficiencies),
3. signals cannot be delivered in batches,
4. (as a consequence) applications lose the ability to schedule their responses to operating system events,
5. and memory requirements for storing event information are proportional to the event rate.

What these arguments illustrate is that the POSIX signals mechanism is not an appropriate mechanism on which to base a web-server's multiplexing strategy. What this thesis shows is that we can develop a generic, preemptive event-notification interface that does not suffer from these limitations.

Banga *et al.* implemented their mechanism in Digital UNIX. Unfortunately, the source code of this operating system and mechanism is unavailable to us for further examination, discussion and comparison. Provos and Lever [27] implement an improved form of the `/dev/poll` mechanism in Linux (the original implementation was for Solaris [5]). The `/dev/poll` mechanism is nearly equivalent functionally to the interface proposed by Banga *et al.* [27] and since `/dev/poll` is available for study, we consider the `/dev/poll` mechanism in its place. We discuss the details of how the `/dev/poll` mechanism functions in Section 2.3.1.

In the remainder of this section we discuss the high-level aspects of Banga *et al.*'s mechanism (and by implication Provos and Lever's `/dev/poll` implementation).

For completeness we should note that Windows NT implements a mechanism similar to the one proposed by Banga *et al.* [10] called "I/O Completion Ports" (IOCP) [17, 31]. This mechanism allows for threads to de-queue I/O completion events from a queue (a port) one at a time (instead of multiple events at a time). Although this is similar to the mechanism proposed by Banga *et al.* it is primarily intended for dispatching asynchronous I/O completion events among multiple (heavyweight) threads.

2.3.1 The `/dev/poll` Mechanism

`/dev/poll` is a mechanism very similar to `poll()`, however it remembers the set of file-descriptors an application is interested in inside the kernel, not requiring it to be repeatedly re-specified as with `select()` and `poll()`. As depicted in Figure 2.6, when an application opens `/dev/poll` it may write `struct pollfd` records to the resulting file descriptor, thus building up a file-descriptor interest set. A special `ioctl()` call (Figure 2.6, line 20), which resembles `poll()`, is used to query the operating system about the state of file-descriptors in the interest set associated with the particular `/dev/poll` file-descriptor. In essence, `/dev/poll` performs the same functionality as `poll()` but with the application's file-descriptor interest set being remembered by the operating system between queries by the application.

`/dev/poll` as originally implemented in Solaris as an improvement to `poll()` (a `poll()` mechanism that remembers the interest set between queries), though it is still a state-based mechanism. However, `/dev/poll` as implemented by Provos and Lever in Linux is an event-based mechanism when applied to file-descriptors for sockets (which is sufficient for our purposes). In this thesis, interest in the `/dev/poll` mechanism stems from the fact that it is an approximation of the mechanism proposed by Banga *et al.* and that it has been shown to be arguably the most efficient multiplexing mechanism for use in web-servers [27].

We say that `/dev/poll` is an approximation of Banga *et al.*'s mechanism because of the details of its implementation. `/dev/poll` as implemented by Provos and Lever scans the objects belonging to an interest set to see if a bit has been set indicating that an event has occurred for that object. If the bit is set, then the "poll" function (inside the kernel) particular to that file-descriptor is called, and this calculates the file-descriptor state to be reported back to the application. Since this implementation performs a "poll" call only if an event has occurred, the overhead of potentially thousands of calls to "poll" is avoided (unlike `select()` and `poll()` which always "poll").

A more appropriate implementation would maintain a list of all events, or of all file-descriptors that had events pending and this list, and only this list, would be scanned in the processing of a query from the application. This is how Banga *et al.* implement their mechanism and this avoids the need to check the flags of every file-descriptor. This difference in implementation of these mechanisms leads us to refer to `/dev/poll` as an approximation of Banga *et al.*'s mechanism. However, our experience with `/dev/poll` leads us to believe that the effect of this implementation issue is negligible.

2.4 POSIX Signals

The standard event-notification interface and mechanism found in Linux (and UNIX systems in general) is the POSIX Signals interface as specified by the POSIX.1b standard [21]. Signals were developed to provide an interrupt-like environment for user processes [24, 29]. The motivation behind the development of this interface is to provide a means for applications to be informed of exceptional conditions (such as changes in tty state) [24]. However, it is not intended to be used as the primary means by which an application (especially a large-scale, I/O intensive one) learns about changes in its environment.

2.4.1 Signals Basics

Signals are notifications of events within the operating system, which are delivered to the application via the interface defined by the POSIX standards. Signals are typically delivered to an application through the invocation of a signal-handler function, specified to the operating system with the `sigaction()` system call [29, 1]. An application may register a signal for each signal type defined by the operating system (with the exception of SIGKILL and SIGSTOP). Once handlers have been registered, an application uses

Figure 2.6 Outline of a `/dev/poll`-Based Web-Server

```
/* Declare and initialize file-descriptor sets */
struct pollfd fds[MAX_NUM_FDS];
struct dvpoll dp = {fds, 1, -1};
struct pollfd tmp;
int max_fd = 1;
int devpoll = open(‘‘/dev/poll’’, O_RDWR);
int i;

/* Configure a socket to accept new connections */
sock = socket(...);
bind(sock, ...);
listen(sock, ...);

/* Indicate interest in reading/accepting on ‘‘sock’’ */
tmp.fd = sock; tmp.events = POLLIN;
write(devpoll, &tmp, sizeof(struct pollfd));

while (1) {
        /* Execute query, infinite timeout value */
        dp.dp_nfds = max_fd;
        ioctl( devpoll, DP_POLL, &dp );

        /* First check for new connections */
        if (fds[0].revents & POLLIN) {
                new_sock = accept(sock, ...);
                /* Register interest in new file-descriptor */
                tmp.fd = new_sock;
                tmp.events = POLLIN | POLLOUT ;
                write(devpollfd , &tmp, sizeof(struct pollfd));
                ++max_fd;
        }

        /* Perform appropriate actions on other file-descriptors */
        for (i=1; i<max_fd+1; ++i) {
                if (fds[i].revents & POLLIN )
                        do_read_action(i);
                if (fds[i].revents & POLLOUT)
                        do_write_action(i);
        }
}
```

Note this is a simplified outline and does not address issues such as how completed connections are cleaned up (in that case `max_fd` shrinks).

the `sigprocmask()` [29, 1] system call to specify which signal handlers may or may not be invoked by the operating system (those that may not be delivered are considered "blocked"). If a signal is not blocked, the operating system may preempt the application and invoke the corresponding signal handler at any time.

When a signal handler is invoked, the set of blocked signals changes to the set associated with the signal handler when it was registered (via `sigaction()`) [1]. The modification of the set of blocked signals upon invocation of a signal-handler is necessary to avoid arbitrarily deep nesting of signals that could result in stack overflows. Furthermore, unless there is a means for the application to disable and enable preemption, there is little the application can do to deal with the concurrency issues raised by the preemptive invocation of its signal-handling code except to make system calls to change the set of blocked signals.

Consequently, this interface typically necessitates a system call be made each time a signal is delivered. These system calls impose extra overhead since each kernel entry requires the kernel to modify the processor's protection level, save and restore application context state and perform various checks for its own security. The goal of SEND is to reduce the number of kernel entries required in a preemptive event notification and delivery interface by amortizing the overhead of a preemptive event-notification over multiple events. We now demonstrate how and why this overhead is incurred by POSIX signals in Linux.

If the set of blocked signals change upon invocation of a signal-handler, it is most likely that the change is temporary for the duration of the execution of the signal-handler. However, the operating system cannot determine whether or not an application is running a signal-handler and hence must be told explicitly that a signal-handler has terminated. This is accomplished through a system call made upon completion of the signal-handler. When an application enters the kernel to reset the set of blocked signals, it gives the kernel the opportunity to deliver the next signal (if it is allowed to by the application's new signal mask).

2.4.2 Real-Time Signals and `phhttpd`

We now describe some advanced features of signals in Linux making them signals more suitable for file-descriptor management in web-servers. These features are usually referred to as "Real-Time Signals" (RT signals) defined by the POSIX.1b standard [21]. The use of these features is demonstrated in `phhttpd` [27, 4]. `phhttpd` is a web-accelerator, a simple web-server used to handle requests for static web-pages, handing-off other requests to a more robust server such as Apache [22].

RT signals provide an alternative mechanism for an application to receive signals. The `sigtimedwait()` system call [6] allows an application to poll the kernel's signal queue for signals and receive them without a preemptive signal-handler invocation. Retrieving signals in this manner does not affect the kernel-entry overhead since we must still make a kernel-entry per signal. However, it makes the application somewhat easier to write since this method eliminates preemption and thus, eliminates the associated concurrency issues. The `phhttpd` web-accelerator uses this system call instead of preemptive signal-handlers for these reasons.

Another significant feature of RT signals in Linux is that signal identifiers not reserved by the POSIX standard may be bound to file-descriptors (via a special `ioctl()` call). Once a file-descriptor is bound to an RT signal, a state change on that file-descriptor triggers that signal. The kernel then delivers to the application the equivalent of a `struct pollfd` describing the state of the file-descriptor when it delivers the signal. This binding is meant to be used as a form of interest sets, where multiple file-descriptors are bound to the same signal number that identifies the interest set. Furthermore, when that signal is delivered to the application it is delivered with a data-structure describing what file-descriptor is affected and what events occurred on that file descriptor (e.g., file-descriptor has been made readable).

2.4.3 Scalability of Signals-Based Applications

Signals were added to UNIX to provide a mechanism for informing applications of exceptional conditions and not as a primary means of communications between the kernel and applications [24]. As interest has grown in developing more sophisticated applications making use of event notification on a regular basis, the shortcomings of this interface have become apparent[30]. There are several issues that hamper the suitability of signals for use in large-scale Internet server applications:

- Only one signal can be delivered at a time (due to the definition of the signal-handler interface). Thus, delivering multiple signals requires that the kernel be entered to retrieve each signal. On the other hand, the `select()`, `poll()` and `/dev/poll` mechanisms we have previously considered allow for kernel entry overhead to be amortized over the notification concerning multiple file-descriptors.

In order for the application to obtain an accurate view of the state of its environment when only one signal is delivered at a time, it must ensure receipt of all signals queued in the kernel [27]. The time to process all queued signals from the kernel may be significant, due to the overhead associated with handling each signal. As a result, an Internet server application using this approach has less time to handle client requests.

If all signals are not immediately delivered to the application, some tricky situations may arise. Consider a situation where the operating system sends a signal to the application indicating that file-descriptor "6" is now readable without blocking. Let us suppose the signal is queued for delivery and before it is delivered the application closes file-descriptor "6" and performs an `accept()` call that re-allocates file-descriptor "6". If the signal is delivered to the application at this point the application has no way of knowing if the signal is for the old or the new file-descriptor.

- Modifying signal delivery behavior requires using a system call for each change. The actions performed by these system calls are essentially manipulations of a bit-map and so the need for system calls to perform these operations imposes a lot of overhead relative to the amount of real work done. As described in the preceding sections, modifying the set of blocked signals is an action that is performed frequently (usually on each signal-handler invocation).

- Kernel sub-systems do not usually keep track of signals once they issue them, having no way of knowing whether or not they have signals queued within the kernel for delivery to the application. This is because the signal mechanism takes control of the signal and becomes responsible for it. Suppose that a TCP/IP socket is communicating events about itself to an application using signals. Each packet that arrives for this socket generates a signal since the socket implementation does not know if past signals are queued or already delivered. The socket must assume that the signal it generates does in fact provide new information to the application. As a consequence, the number of signals queued for delivery depends on the rate at which events are generated in the kernel and is limited by the amount of memory available to the global signal buffer. This makes it likely that a high rate of events consumes all available signals from the kernels global signal pool.

  We approach this issue differently with SEND, by having sockets maintain ownership of the generated events as they are processed by SEND. Consequently, SEND acts on events rather than taking over responsibility for them; the kernel sub-systems that generate events may revoke or modify events after they have been posted. By making kernel sub-systems such as sockets responsible for their own events, we can coalesce events that provide duplicate information into one event and thus, allocate all memory for a socket's events at the time of socket creation.

The effect of the scalability concerns associated with signals is illustrated by Provos and Lever [27] in their comparison of the scalability of `phhttpd` with that of the `thttpd` web-server modified to use `/dev/poll`. In their experiments, Provos and Lever observe that the signals based `phhttpd` could only handle a client request rate of 85% of the request rate that a `/dev/poll` based `thttpd` could handle [27]. They provide two fundamental explanations for the observed behavior of `phhttpd` [27]:

- a high rate of client requests imposes significant overhead due to the inefficiencies of the signals mechanism and interface and thus;
- when the signal queue within the kernel overflowed (and no more signals could be queued) `phhttpd` resorted to using `poll()`, thus taking on the performance characteristics of a `poll()` based application.

These results demonstrate that it is crucial for such a server to be able to process events as quickly as they are generated. With SEND we present a mechanism that allows an application to have efficient and immediate access to event data. Internally, management of events with SEND keeps memory requirements proportional to the number of sockets, rather than the event rate, and therefore memory requirements are known at the time of socket creation. Since we know memory requirements in advance we do not run out of available events as events are generated.

2.5 The Exo-Kernel Approach

In recent years an alternative approach to operating systems design has been proposed in the form of the "exo-kernel" [15, 14, 3]. The exo-kernel philosophy is about reducing an operating system to a mechanism for multiplexing hardware and providing minimal support for abstractions on which applications are built. However, the exo-kernel approach does in fact provide an abstraction: since its goal is to multiplex hardware, it provides to applications a virtual machine environment within which applications (using libraries) implement specific interfaces and abstractions (e.g., UNIX emulation).

One of the features of an exo-kernel operating system is the notion of exception handling by applications. An application in an exo-kernel system can handle its own page-faults by having exception-handlers called by the exo-kernel. The mechanisms for supporting exception handling in exo-kernels are designed to be lightweight, allowing for efficient implementations of systems such as garbage collectors and persistent storage systems that rely on handling the hardware exceptions generated by an application due to memory operations.

The SEND mechanism proposed in this thesis aims toward implementing such a notification mechanism. SEND too has major design points in common with exo-kernel systems; the application is always responsible for restoring its previous execution state after it finishes handling an event-notification [15, 14, 3] and the kernel is able to directly manipulate data in an application's memory to aid in informing it of events. The challenge in this endeavor is to apply these exo-kernel-like principles into a traditional monolithic kernel.

Chapter 3

Design and Implementation

This chapter presents in detail the design of the Scalable Event Notification and Delivery (SEND) mechanism. The previous chapter discusses several mechanisms that applications may use to learn of operating system and external events of interest to them. SEND is a generic preemptive event-based mechanism that addresses the scalability concerns of POSIX signals to provide an efficient event-based mechanism to be used in place of traditional polling mechanisms (such as `select()`, `poll()` and `/dev/poll`). We begin with an overview of the major design features, why they are necessary, how they differentiate SEND from POSIX Signals and how one could write an application using SEND. This is followed by a discussion of the SEND interface illustrating how the SEND interface handles synchronization issues between kernel and application. Finally, we discuss how SEND is used to implement operating system functionality not otherwise available in Linux; an alternative event-driven mechanism for the `accept()` system call.

A Note About Terminology

Before going further, it is necessary to establish some of the terminology that used throughout the rest of this thesis. It is common among UNIX/Linux developers to refer to the POSIX Signals interface (and underlying implementation) simply as "signals". With this term, one usually associates the mechanism and interface being used to send certain events (called signals) to an application. Thus the term "signals" has a dual meaning; it is a specific set of events that an application may be notified of and it is the interface and mechanism by which this notification occurs. Using the first meaning, SEND delivers "events" to applications and some of these events may be "signals".

3.1 Overview of the SEND Design

In the previous chapters, we identify several problems with POSIX Signals, the standard preemptive event-notification interface in Linux (and UNIX). We use these concerns to define the design objectives of the design of our new preemptive event-notification mechanism, called Scalable Event Notification and Delivery (SEND).

As the name suggests, we make a distinction between the terms "event notification" and "event delivery", as follows:

event delivery: The process of copying event data into an application's address space, thus making the data available to the application. The process of copying signal information (when using the POSIX signals mechanism) onto the application's stack is an example of event-delivery.

event notification: The process by which an application's execution state becomes aware of event information delivered to it. An example of an event notification is a preemptive event-handler invocation, or an application making a system-call (such as `sigtimedwait()`) to extract event data from the operating system. Note that in both examples, event notification and event delivery occur at the same time. Also event notification need not be preemptive; polling is an event notification process as well.

Unlike POSIX signals, SEND allows for event delivery to occur without immediate event notification. The capability to control notification and delivery separately is an important feature of SEND, allowing for greater flexibility in application design. In particular, allowing for delivery to occur without notification is a surprisingly useful feature (as will be demonstrated Section 3.3).

Our primary goal in designing SEND is to develop a mechanism to deliver multiple event notifications simultaneously. In doing so, we make the most use out of each switch between kernel and user mode; each such transition being an opportunity to deliver multiple events to an application. By delivering multiple events per transition between the kernel and application, SEND reduces the overhead associated with delivering each event (relative to the signals mechanism) and allows the system to spend more time executing useful application code.

SEND delivers all data describing events to an application via a ring buffer (whose size, and therefore event capacity, is set by the application). Event data is communicated in the form of `struct event` objects, which are a modified form of `struct siginfo` objects used to communicate event data via the POSIX signals mechanism. When events are delivered into this buffer, the operating system *may* preemptively invoke an event-handler function within the application to notify it of the fact that events have been delivered to the application. There is only one event-handler function that the operating system calls and thus, unlike the signals mechanism, it is the responsibility of the application to de-multiplex the delivered events. This approach allows SEND to deliver multiple events to an application with one preemption. Since the stack is not used to deliver event data, the process of event delivery need not be tied to event notification (as is the case with the signals mechanism). As a consequence, it is possible to deliver events without immediate notification, allowing the application to handle events at a time convenient for it.

Applications using SEND control the key elements of SEND's behavior by manipulating a data structure shared between the application and the kernel, which is mapped into the address spaces of both. This data structure is called the "Event Control Block" (ECB) and contains fields that define the state of the ring buffer, the event-handler function, a pointer to the saved state of the application prior to event-handler invocation, bit-maps specifying what event types may be delivered to the application and which event types may trigger an event-handler invocation. A new system call for controlling SEND called `evtctl()` (similar in nature to `ioctl()`) is used to ask the kernel to create this mapping and return its address. The ECB allows the application to control the most frequently modified aspects of SEND (enabling and disabling delivery and notification) behavior without making system calls. In particular, SEND allows for an application to re-enable event notification upon completion of execution of the event-handler function without entering the kernel. With the application able to restore its prior state, in circumstances where applications are faced with high event rates, we have an expected cost of less than one kernel entry per event delivery. (The ECB is described in detail in Section 3.2.)

Our research has uncovered only one other type of application where kernel activity is affected by an application directly manipulating data in its own address space as is done with the ECB: providing a scheduler with hints as to what state a user-space thread is in to more effectively deal with lock contention issues [33]. Consequently, the ECB, is a rare example of a data structure that is shared between applications and the kernel. A promising avenue of future research is to investigate other operating system facilities that may benefit from such a shared-memory based interface.

This design overview addresses two of the key scalability concerns associated with POSIX signals (by reducing costs of controlling the mechanisms behavior and allowing delivery of multiple event simultaneously). However, it also raises some important issues that we devote most of the remainder of this chapter to:

Kernel Security

The SEND design exposes a kernel data structure (the ECB) to direct manipulation by an application. We must ensure that this does not threaten the stability and security of the operating system as a whole. Protection is guaranteed since the data within the ECB only affects the behavior of the operating system with respect to the current application, and only affects the behavior the operating system already allows the application to modify. (Consider that an application using POSIX signal is free to set the blocked signals mask to whichever value it chooses to.)

Concurrency Issues

Even though it may be safe for an application to modify the ECB with respect to the kernel's security, direct access by the application to the ECB raises the question of whether or not ECB modifications can be performed safely in an environment where the kernel may preempt the application at any time (to handle interrupts). To share the ECB between the kernel and application safely, each field of the ECB may be modified by the kernel or the application (and not both). In this manner we can ensure that data written by one of these entities is not lost by being overwritten by the other. The only requirement is that each field be modified atomically so that all read and write operations are performed consistently. If this requirement is not achieved, the kernel may not see the ECB the way the application intends it to be seen. Thus, it is in the application's best interest to respect this restriction. We discuss the particular details of these issues in Section 3.2.2.

Controlling Preemption Behavior

As mentioned above, all ECB fields manipulated by the application must be modified atomically. Thus, there needs to be a mechanism that allows the application to modify the set of blocked events with one operation. This capability is achieved by using two 32-bit bit-map within the ECB to specify the set of events that may be delivered and which trigger an event-handler invocation. (Events that may trigger an event-handler invocation may always be delivered.) We use a 32-bit bit-map since this is the largest size of bit-map that we can manipulate atomically on the x86 architecture, which we are working on (this may vary for other architectures with different word sizes). During the initial configuration of SEND, the application defines the set of events that are controlled by each bit in the bit-maps. This detail is discussed in Section 3.2.1.

Application State Restoration

One of the ways we reduce the number of system calls when using SEND is to have the application restore its own state after a preemptive event-handler invocation. In the course of doing so, one must not only restore the application's registers, but typically one needs to reset the bit-maps in the ECB controlling event delivery and notification behavior. In order to do this safely, one must ensure that the event-handler is not invoked in the midst of this restoration procedure. To deal with this situation, we allow the application to define a range of addresses within its code (referred to as the "critical-section"). When the application executes within the critical-section invocation of the event-handler is disabled, regardless of the settings of the ECB. The specific details of this mechanism are presented in Section 3.4.

Ring Buffer Overflow

Since we are using a ring buffer to deliver event data, we must consider what happens when the buffer is full and the operating system still has more events to deliver. SEND addresses this issue by defining a special event type (`EVT_OVERFLOW`) that is controlled with ECB bit-maps just as any other event, but delivery of which never consumes a ring buffer entry. The `EVT_OVERFLOW` event simply triggers the invocation of the event-handler when the ring buffer is full. The ECB also contains a field that informs the application how many events are queued for delivery within the operating system, allowing applications to determine inside the event-handler whether or not they should restore the application's state or extract more events from the kernel.

SEND Configuration

SEND allows for an application to control most of its behavior, blocking and unblocking event notification and delivery, without system calls. However, we provide a system call interface for some less frequently performed actions (where delay is not an issue). These actions include performing initial SEND configuration and defining the meaning of the event-controlling bit-maps. These are all actions that are performed rarely (usually only as the application is initializing itself) and so we are not greatly concerned about the overhead associated with performing such actions. To perform these tasks, SEND defines a new system call (`evtctl()`) that is similar in nature to the `ioctl()` system call and is used to perform all actions related to SEND that may require a system call.

The design outlined in this section addresses most of the concerns raised by Banga *et al.* with regards to POSIX signals as an event delivery mechanism for web-servers (Section 2.3). In particular, this design so far addresses all but item 5 (memory requirements being proportional to event rate). We address this last item in Section 3.6 by proposing alternative (kernel-internal) semantics for managing file-descriptor-related

Figure 3.1 Event Control Block Definition

```
struct event_control{
        u32 notify;         /*  u32 == unsigned 32-bit integer */
        u32 deliver;
        u32 head;
        u32 tail;
        void (*handler)(struct event_control *ecb);
        void* stack;
        struct evt_regs *regs;  /* struct evt_regs contains
                                 * saved values of all registers
                                 * prior to invocation of 'handler'
                                 */
        struct event eventbuf[]; /* each 'struct event' contains
                                  * individual event information
                                  * -- analogous to 'struct siginfo'
                                  */
};
```

events. These new semantics give sub-systems such as sockets more control over the events they generate, addressing the concerns about event managements (revocation) raised in Section 2.4.3.

In summary, with SEND we have an efficient mechanism for propagation of event information from the operating system into applications. In continuing our discussion, we first discuss the ECB, the key component of the SEND interface to be followed by the presentation of a sample application that uses SEND. Having presented SEND in sufficient detail we then address the particular issues and concerns about SEND that we have identified in this section.

3.2 The Event Control Block

The most important interface feature of SEND is the "Event Control Block" (ECB). The ECB defines the region of memory in which the application places the parameters that define SEND's behavior and provides a ring buffer that is used by the kernel to pass event information to the application. The structure of the ECB is determined by operating-system provided header files (Figure 3.1). The fields of the ECB serve the following purposes:

`deliver`:

Each event type is bound to an event group (see Section 3.2.1) and each event-group is represented by a bit in this 32-bit bit-map. This bit-map specifies the event groups for which events may be delivered to the application. If an event group's `deliver` bit is set, the events from that group are copied into the ECB's ring-buffer without triggering preemption of the application. The application can then check the ECB at a more convenient time to see if events have been delivered. This field is modified only by applications though it is initially set to 0 by the kernel.

`notify`:

This field is a 32-bit bit-map that identifies the events that may trigger a preemption (call to the event-handler by the operating system).

If an event group has its corresponding bit set in the `notify` bit-map, the bit for that group in the `deliver` bit-map is ignored. When a set of events is delivered to the application by the operating system, the operating system invokes the event handler function if any of the delivered events belong to a group whose bit in the `notify` bit-map is set. This field is modified only by applications though it is set to 0 by the kernel when the ECB is initialized, thus disabling all notifications.

`stack`:

This is the stack pointer that used when the event-handler is invoked in a preemption. If this field is set to NULL then SEND uses the application's current stack (as is done with the signals mechanism).

`handler`:

The address of the event-handler function invoked by the kernel when performing preemptive notifications. Since the kernel is delivering multiple events simultaneously, it does not know which specific handler to invoke (as in POSIX signals). Thus, a single handler is used and this handler must de-multiplex the single notification to event-specific handlers if it wishes to emulate a signals-like interface. This field is set by the application and is initialized by the operating system to NULL, thus disabling all preemption. The event-handler function is given one argument when invoked by the kernel: a pointer to the kernel-thread's ECB. This argument allows the function to identify its ECB, which is necessary in multi-threaded programs where each kernel thread must have its own ECB.

`regs`:

This is a pointer to a structure defining the register-state of the application prior to preemption. Once the application's event-handler function has finished, it can restore the application's state prior to preemption with the data pointed to by this field. The operating system sets this field just before invoking the event-handler function in the application.

`eventbuf`, `head`, `tail`:

The `eventbuf` array is used as a ring buffer to hold event information (Figure 3.2). The elements of this array are divided into two groups; those elements into which the kernel may deposit new event information and those from which the application will read that information. The `head` and `tail` field identify the range of array elements which are yet to be processed by the application. The `head` field identifies the ring-buffer entry where the next event will be delivered into and the `tail` field identifies the last event processed by the application. As new events are delivered, the kernel increments (and wraps around if necessary) the `head` field. Likewise, as the application consumes events it increments `tail`. Using this mechanism the kernel knows which elements of the buffer are free for delivery of events. Each element of the `eventbuf` buffer is a `struct event` object, which is an extension of `struct siginfo` object used in the signals mechanism (allowing us to easily deliver signals using SEND).

3.2.1 Event Groups

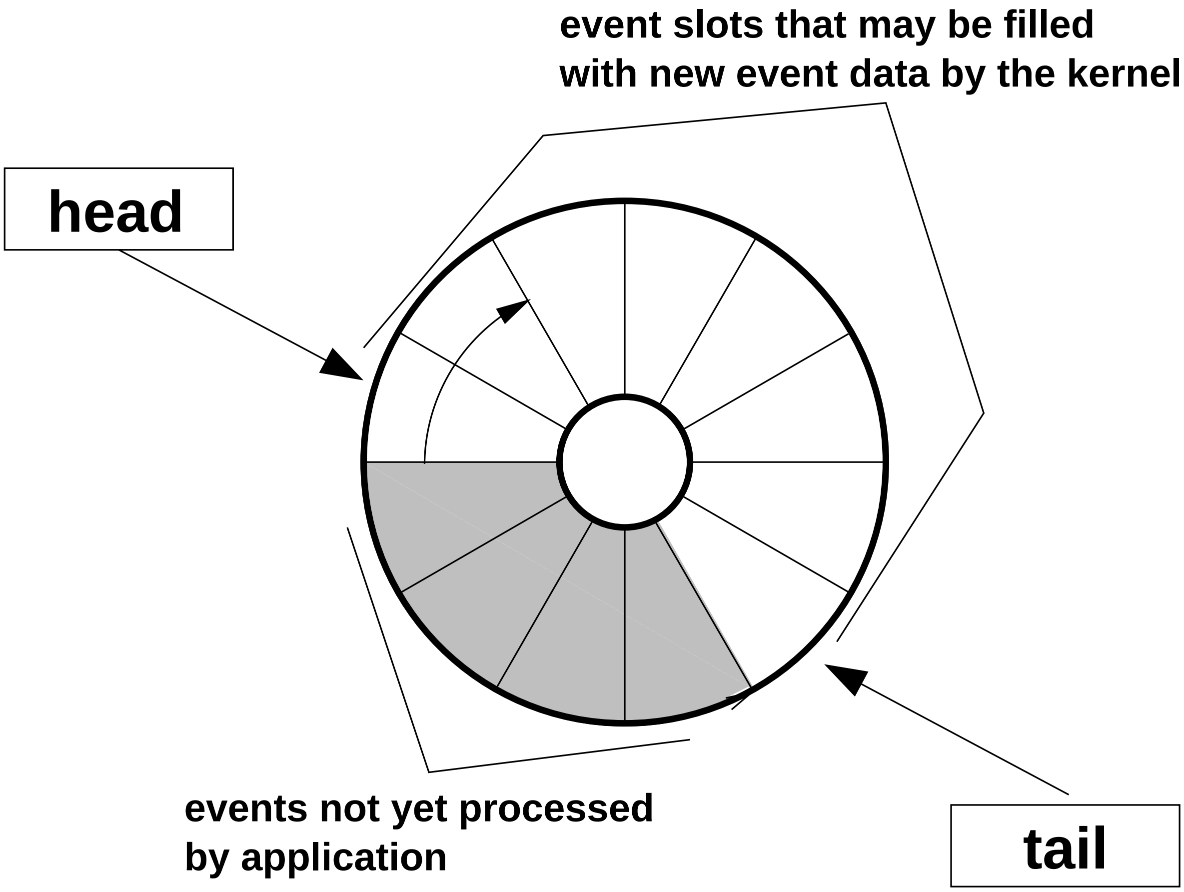
The POSIX signals implementation in x86 Linux also uses a bit-map to determine which signals should or should not be delivered. Each bit in this bit-map controls the behavior of the operating system with respect to one signal type. Each signal type has a signal number associated with it and thus, delivery of signal k is controlled by the value of the k'th bit of the bit-map. However, unlike the ECB bit-maps, this bit-map is maintained within the kernel and must be manipulated with system calls.

SEND uses two bit-maps inside the ECB to control the operating system's event notification and delivery behavior: `notify` and `deliver`. These bit-maps are both 32-bits in size, coinciding with the x86 architecture's word-size thereby allowing for the bit-maps to be read and set in their entirety with a single (atomic) instruction (see Section 3.2.2). If the application ensures that all manipulations of these bit-maps occur atomically, then no interrupt can occur in the midst of a manipulation and thus the kernel can never read a bit-map in an inconsistent state.

In SEND, event types (or signal numbers) are also mapped to bits in the bit-maps, but this is a mapping that is determined by the application. Each bit in a bit-map represents an "event group", which is a set of events (possibly signals) whose delivery or notification behavior is controlled by that bit. Event groups are identified by the bit that controls them, thus event group 0 is controlled by the lowest order bit in the bit-map. Allowing applications to define event groups in this manner accomplishes two things: it allows the application to prioritize different event types and usually makes a 32-bit bit-map sufficient.

It is because of this ability to dynamically set the meanings of the bits in these bit-maps that a 32-bit bit-map is sufficient. Applications such as web-servers are primarily interested in a small handful of event types and thus, need only a small number of bits to control these events. Of the other events (that

Figure 3.2 Ring Buffer of Event Data



an application is only marginal interest in), empirical evidence suggests that notification of these events is toggled simultaneously for all such events. SEND allows all such marginally interesting events to be assigned to the same event group (or to none at all, thus disabling them).

Unlike POSIX signals, SEND does not treat all bits in the bit-mask equally. Event types represented by low-order bits are delivered first by SEND and thus, the bit-maps provide a basic prioritization mechanism. Providing a prioritization mechanism allows for an application to give priority to exceptional events. This is important because SEND is designed for use in applications relying on the event-notification mechanism as the primary means of learning about changes in the environment. If we have a server application that processes thousands of file descriptors, there is a danger that a warning of an exceptional event would have to wait for thousands of file-descriptor events to be processed. The prioritization mechanism provided by the configurable bit-map mappings allows for the application to decide for itself which events are exceptional. This structure allows, for example, a server application using an advanced memory management system (garbage collection or persistent storage) to give higher priority to events related to memory management (e.g., SIGSEGV signals) than to events on sockets. Such prioritization is necessary since memory related events need to be dealt with immediately to allow the application to finish what it is doing with the current client connection it is handling (before it is able to deal with others).

Configuration of event groups is performed with the `evtctl()` system call. Figure 3.3 provides an example of how this is accomplished. In this example, we bind SIGINT and SIGTERM signals to event group 0, and SIGIO to event group 31.

Figure 3.3 Using `evtctl()` to Define Event Groups

```
int group;
/* SIGINT and SIGTERM should be processed with high priority,
 * assign to group 0
 */
group = 0;
evtctl(SEVTGROUP, SIGINT, &group, sizeof(int));
evtctl(SEVTGROUP, SIGTERM, &group, sizeof(int));

/* SIGIO is low priority */
group = 31;
evtctl(SEVTGROUP, SIGIO, &group, sizeof(int));
```

3.2.2 ECB Manipulation Issues

SEND relies on the operating system and application manipulating a shared data-structure (the ECB; Section 3.2). In this section we describe how this can be accomplished in a manner that is safe for both the application and operating system.

The signals mechanism relies on `sigprocmask()` [29, 1] to enable or disable the delivery of signals. One of the reasons for modifying the operating system's behavior using a system call is to ensure that the operating system's data structures are modified in a safe manner. In Linux, the kernel is not concerned with what signals an application wishes to block or unblock (applications are free to make mistakes in this regard), the only concern is that in modifying the blocked signal set the application does not corrupt other kernel data structures or cause synchronization problems. If these problems can be addressed, then we can make this kernel data available for direct manipulation by user-space programs.

The ECB exposes only data that may be manipulated safely by an application, addressing the first concern. However, this still leaves us with the issue of whether or not this can be done safely in an environment where the kernel may preempt the application at any time (to handle interrupts). To understand how this is possible, we need to first examine the relationship between application and the kernel and how they interact.

A typical Linux kernel-thread within the Linux kernel runs in one of two modes: in user mode ,in which application code runs, and kernel mode, in which the kernel executes system calls, exception handlers and interrupt handlers. An entry into kernel mode, for whatever reason, never switches the current thread; it only switches the mode of the current thread. We are thus guaranteed that only one of a thread's kernel and user-space components may be executing at any time. Thus, when a thread is running in kernel mode it can safely modify and examine the ECB, knowing that the user mode code is not running. This restriction on when the kernel can look at the ECB is not a problem for us since the only time we need to access ECB fields is just prior to switching from kernel mode to user mode; at a time when the current thread's ECB is in the current virtual address space (of the kernel code). A thread in user-mode may be interrupted (and switched to kernel-mode) at any time and as a consequence the user-mode component of a thread must ensure that the ECB is always in a valid state by using atomic instructions to manipulate it. Typically, this atomicity is accomplished by relying on the atomicity of word-size load and store operations of the CPU.

If an application using SEND fails to ensure that the ECB is in a consistent state at all times, then the ECB as perceived by it or by the kernel may not be what is intended. The effect of such mis-perceptions is equivalent to the application specifying the wrong set of signals to be blocked, if it were to use POSIX signals. In summary, it is the application's responsibility to ensure the ECB is managed properly and thus, must ensure that it obeys the following rules:

1. The `head`, `num_queued` fields and `eventbuf` ring-buffer are the only fields that are modified by the kernel (the kernel considers these to be write-only). These parts of the ECB should not be modified by the application. The kernel keeps a private copy of these fields and does not rely on the values in user-space, which may be corrupted by the application.

2. Applications may modify all other fields, however, all modifications must be made atomically since the operating system may preempt the application at any time due to an interrupt or exception. Strictly speaking, the atomicity requirement is that "load" and "store" operations performed on the ECB fields are performed atomically in order to ensure that fields are never partially set or partially read. (This requirement is typically met by using standard word load and store instructions.)

3. In cases where one wishes to modify several fields at once, all preemptive event notification must be disabled during the course of these operations. If notification is disabled, there is no need for the kernel to look at these other fields. (For example, if one wishes to modify `handler` and `stack` together, disable notification first.)

4. If an application is running with multiple kernel-threads, each thread must have its own ECB; an ECB may be modified only by the thread that configured it. The operating system does not make any guarantees as to the state of the ECB when observed by other threads.

3.3 SEND Example Application

This section presents an annotated example of a program using SEND. This application loops printing the message "In loop..." indefinitely. When the program receives a "SIGINT" signal, the signal is delivered via the SEND mechanism and the application prints "Received SIGINT" before resuming its original loop. Using SEND in such an application makes the application more complicated than it needs to be, but the goal of this example is to demonstrate how one would configure SEND without issues associated with more complicated applications (such as web-servers) to obscure the example. We discuss how to apply SEND in an Internet server application in Section 4.3.

Example SEND Application

Section 3.4 describes a mechanism called "critical-sections" that provides another mean of controlling application preemption. In this example, the impact of this mechanism is that if the application is executing code between `begin_cs()` and `end_cs()`, the kernel may not preemptively notify the application of any events. The role of this mechanism becomes apparent in the discussions following the presentation of this example.

```
/* Empty function to define start of critical section. */
void begin_cs(){};
void event_handler (struct event_control *ecb) {
    /* Notification is off due to critical section. */

    int old_notify = ecb->notify;
    /* Explicitly turn off notification to allow calls
     * to other functions  */
    ecb->notify = 0;

     /* Delivery is on, kernel may be changing ''head'' */
    while ( ecb->head > ecb->tail) {
        struct event * ce =
             &ecb->eventbuf[ecb->tail % EVENTBUF_SIZE];
        /* Invoke handler for specific type of event.
         * We are jumping out of critical section, must
         * rely on notify field for preemption control
         */

        /* evt_type specifies what kind of event it is.
         * Traditional signal events are identified by their
         * signal number
         */
```

```
        (*handle_event[ curr_event->evt_type ])(ce);
        ecb->tail = ecb->tail+1 % EVENTBUF_SIZE ;
    }

    ecb->notify = old_notify;

    /* Notification is off due to critical section. */

    /* Restore previous register state, thus jumping out
     * of the critical section, re-enabling notification
     * control via the notify field
     */
    restore_state(ecb->regs);

    /* This point never reached. */
}
/* Empty function to define end of critical section. */
void end_cs(){};


void sigint_handler (const struct event *e){
    printf(‘‘Received SIGINT\n’’);
}

int main () {

    /* Define ECB holding 100 events. */
    ecb = evtctl(DEF_ECB, 100);

    /* Define event handler stack.
     * Note: stack grows down, thus we provide a pointer
     * to the top of the memory region.
     */
    ecb->stack = malloc(8192) + 8192;

    /* Define SEND event handler */
    ecb->handler = event_handler;

    /* Define critical section. */
    evtctl(DEF_CRITSECT, begin_cs, end_cs);

    /* Bind SIGINT events to group 0. */
    evtctl(SETGROUP, SIGINT, 0);

    /* Set handler for SIGINT */
    handle_event[SIGINT] = sigint_handler;

    /* Enable notification and delivery for group 0,
     * which is represented by the lowest order bit */
    ecb->deliver = 0x00000001 ;
    ecb->notify  = 0x00000001 ;

    while(1){
        printf(‘‘Looping...\n’’);
```

```
77         }
78     }
79
```

We now explain the example code:

50: The application asks the kernel to configure SEND by creating a mapping for the ECB. In making this request the application also tells the kernel what capacity it desires for its ring-buffer. The kernel performs all memory allocation and mapping as well as checking for potential errors (such as an ECB already having been configured).

52-69: The application configures the ECB. This involves setting ECB fields to define the event-handler entry point and the stack pointer to be used by the event-handler.

72-73: The application enables notification and/or delivery by setting the appropriate ECB fields. The next time the kernel is entered (via system-call, exception or interrupt) it can observe these fields and deliver events if appropriate as it returns to user-mode. The application now proceeds to perform its intended tasks (or it may wait until an event is delivered).

In kernel: The kernel delivers an event and preemptively notifies the application. Just before it switches from kernel-mode to user-mode it sets the stack and instruction pointer to simulate a call to the event-handler function. The kernel also copies the register state of the application prior to preemption onto the application's original stack and sets the ECB field `regs` to point to this data.

11-25: The application's event-handler examines the ECB's ring buffer and handles the events contained within (however many it chooses). As the event-handler handles events, it increments the `tail` field freeing entries in the ring buffer for the kernel to deliver subsequent events into. Note that the kernel may be delivering more events throughout this loop. This is an important design feature of SEND; by separating control of notification and delivery we can deliver events to the application as it is handling other events.

27: The application sets the `notify` fields to allow the kernel to perform further preemption. At this point preemption by the operating system is disabled by the critical-section mechanism, allowing application state to be safely restored.

35: The application restores its state prior to preemption using data pointed to by the `regs` ECB field. The `restore_state()` function ultimately performs a jump out of the `event_handler()` function in order to restore the application's instruction pointer to its state prior to the event-handler being invoked. This jump causes the application to jump out of the critical-section and re-enables the `notify` field as the sole controller of preemptive notification.

The POSIX signals interface is much simpler and better suited to an application such as this. The various system calls in the POSIX signals interface provide larger functional units; this example application would require only a few lines of code if written using POSIX signals. SEND on the other hand forces a programmer using the raw SEND interface to be more aware of and involved in the process of configuring SEND and handling events. As a consequence, one could think of the distinction between POSIX signals and SEND as being analogous to the distinction between CISC and RISC processor architectures (respectively). Just as the details of a RISC architecture are hidden from the programmer by the use of high-level programming languages and compilers, it is conceivable that the details of the SEND interface could be hidden from programmers by a convenient yet efficient wrapper library. In fact, we see no reason why one could not emulate the POSIX signals interface in user-space on top of SEND.

The example presented in this section establishes most of the application infrastructure necessary to build an Internet server application using SEND. Note that the examples given in Chapter 2 (Figures 2.2, 2.4, 2.6) attempt to translate the semantics of a polling oriented kernel interface into an event-driven application architecture. The `do_read_action()` and `do_write_action()` functions in these examples are essentially event-handlers called when the application determines that an event of the appropriate type has occurred on the specified file-descriptor. Thus, to modify the example in this section to present a basic Internet server application requires adding the necessary system calls to configure sockets to generate appropriate events and modifying the event-handler function to properly de-multiplex these events. We discuss issues of implementing a web-server with SEND in Section 4.3.

3.4 Critical Sections

Given the simple SEND example, the necessity of the "critical sections" mechanism we can now be discussed. As discussed in Section 2.4.3, applications need some mechanism to prevent nested event-handler invocations (an event-handler preempting another invocation of the event-handler). There are two situations in which relying solely on the ECB's `notify` field is insufficient:

- When the event-handler is first invoked the `notify` field must be modified by the application to disable notification. However, the kernel could receive an interrupt and attempt to notify the application of a new event before this happens, resulting in a nested event-handler invocation.
- An application executing inside an event-handler wishing to restore its prior state (including the `notify` field) can only do so by setting the `notify` field and then restoring the instruction pointer. Thus, there exists an opportunity for the kernel to invoke the event-handler function just as it is restoring application state.

These situations are not a problem when using the signals mechanism. The signals mechanism allows for an application to specify the set of signals that are blocked during the execution of each signal handler. The signals mechanism also provides a means for simultaneously restoring application registers and signal notification behavior with the (implicit) `sigreturn()` system call [2]. Since SEND aims to eliminate this secondary kernel entry at the end of a preemption, a specialized mechanism must be developed to allow applications to safely enable and disable event notification inside the event-handler function.

The mechanism developed for SEND is to allow applications to register a "critical-section" of code. If the kernel finds the application's instruction pointer is inside a critical section, it refrains from invoking the event-handler; the critical section mechanism overrides the ECB's `notify` field. If we declare the region of memory in which the event-handler function resides as the critical section we can guard against nested invocation of the event-handler in the two situations outlined above.

Since SEND allows for event-delivery without event-notification, the kernel can still deliver events (depending on the setting of the `deliver` field) even though notification is turned off. Thus, an application need not worry about having event-delivery delayed because it is executing code that should not be preempted; applications are always free to check the ECB on their own to see if events have been delivered. This raises a key distinction between signals and SEND. When using signals applications must enter the kernel to get the next signal, whereas an application using SEND can check the ECB in a handful of instructions.

As shown in the example in Section 3.3, a SEND event-handler may repeatedly check if it has more events to handle (by comparing the `head` and `tail` fields) and if it determines that there are no further events to process, it commits itself to restoring the state of the application prior to the invocation of the event handler. An application may allow the kernel to deliver events while the event-handler is running and it is possible that an event may be delivered after the application has compared `head` and `tail` (and committed to restoring saved-application state) but before notification is enabled. The application would then be notified of the event upon the next invocation of the event-handler or if it were to check the ECB on its own.

Such an event-handler is "edge-triggered"; notifications may be delayed if events are delivered after the event-handler has committed to restoring the saved registers, but before it exits the critical section. Ideally, one would like to have a "level-triggered" event-handler that does not restore application state unless all delivered events have been processed. Such a "level-triggered" event-handler needs to allow itself to be preempted by itself (and be able to recover from such a situation). Figure 3.4 presents an outline of how such and event-handler may be written.

Figure 3.4 Pseudo-Code for a Level-Triggered Event-Handler

```
begin_critical_section:
void event_handler(){

    save notify field;
    if( preempting event handler )
        cleanup nested preemption;

    do
        clear notify field;
        process next event in ring buffer;
        update tail;
        restore notify field;

end_critical_section:
    while( head != tail );

    restore registers;
}
```

3.5 Non-Preemptive SEND

Although SEND has been designed to be used as a preemptive event-notification mechanism it is possible to use it in a non-preemptive manner. In doing so, we effectively transform SEND to simulate the mechanism proposed by Banga *et al.* [10], allowing us to evaluate the effect of preemptive notification in web-server performance.

One of the commands supported by the the `evtctl()` system call is the `WAIT_FOR_EVENT` command. This command causes the `evtctl()` system call to block until an event is ready to be delivered to the application in a manner similar to the `sigsuspend()` system call [1]. A bit-map whose bits have the same meaning as the `deliver` bit-map is passed to `evtctl()` to identify which event groups the application wishes to block for and this bit-map overrides the `deliver` bit-map for the duration of this system call. Additionally, `evtctl()` does not block if the ring buffer is not empty and upon completion of this system call the kernel does not invoke the application's event-handler. We have found that these last two properties of this mechanism make it easier to use, by reducing the volatility of application/kernel interactions. Figure 3.5 demonstrates how this mechanism is used.

This functionality in `evtctl()` permits us to simulate Banga *et al.*'s mechanism:

- Event groups are analogous to interest sets. Declaring interest in a particular file descriptor is accomplished by assigning the file descriptor to a particular event group.

- The analogue for the `get_next_event()` system call is `evtctl()` called with the `WAIT_FOR_EVENT` command, provided that event notification and delivery are otherwise disabled. If this is the case, the only opportunity for event information to enter the application is when the application calls `evtctl()` to wait for the next event.

In a true event-driven application (with preemptive notification), if the application has nothing left to do, it idles or blocks until the next event arrives. Thus, if we have a web-server that uses preemptive event-notification, we can transform it into one that uses our simulation of Banga *et al.*'s mechanism by running it with notification and delivery disabled and ensuring it blocks waiting for events when it has nothing left to do.

Figure 3.5 Waiting For Events

```
/* Disable delivery and notification. */
int old_deliver = ecb->deliver;
int old_notify  = ecb->notify;

ecb->deliver = 0;
ecb->notify  = 0;

/* If kernel has changed ‘‘head’’ without us knowing, then
 * make an evtctl() call that returns immediately. */
if (ecb->head == ecb->tail) {
        /* Wait for any event to be delivered. */
        evtctl(WAIT_FOR_EVENT, ~0);
}

/* Call event handler explicitly to handle events.
 * Handler must be smart enough to realize that it is
 * being called by the application and not preemptively
 * invoked by the kernel.
 */
event_handler(ecb);

/* Re-enable notification and delivery. */
ecb->notify  = old_notify;
ecb->deliver = old_deliver;
```

This example code demonstrates how to wait for events. Note that we disable delivery and notification before checking the ECB (to see if blocking is necessary). Since the kernel uses the bit-map passed into `evtctl()` to determine what events we can deliver, we avoid a potential race condition triggered by the kernel delivering all of its events as we are about to call `evtctl()`.

3.6 Related Kernel Internal and Interface Issues

In this section, we discuss some details of the internals of SEND in order to illustrate how kernel sub-systems (sockets in particular) interact with SEND. We describe how these new interaction semantics can be used to implement an event-driven alternative to the traditional `accept()` system call. Additionally, these semantics address the 5th concern raised by Banga *et al.* (memory requirements to represent event information — Section 2.3).

As illustrated in Section 2.4, mechanisms such as POSIX signals must deal with events that are queued, awaiting delivery to the application. The implementation of POSIX signals in Linux does not allow for sockets to modify or revoke signals after they have been issued. This limitation allows for the situation depicted in Section 2.4, where a signal notifying an application that a file-descriptor has data available for reading arrives after the file-descriptor has been closed.

The implementation of SEND allows kernel sub-systems (such as sockets) to have more control over the events sent applications. Events handled by SEND within the kernel may be modified or revoked while they are queued awaiting delivery. When a kernel sub-system generates an event to be delivered to an application via SEND it passes to SEND a `struct event_holder` object containing the `struct event` object for the event data to be delivered to the application. The `struct event` is eventually copied into the application's ring buffer of event data. The `struct event_holder` object also contains:

- Flags indicating the status of an event: inactive, queued for delivery or delivered;
- A pointer to a call-back function (within the kernel) that is called as the event is delivered;
- A flag indicating what is done with the `struct event_holder` object after delivery (nothing or the memory is to be freed).

In the common case, most of these features are unnecessary; when an event is generated it usually specifies that the memory is freed automatically by SEND after delivery. However, there are situations in which these features are useful.

The in-kernel call-back allows the sub-system that generated the event to be made aware when the event is about to be delivered, and allows it to perform actions with the kernel running in the context (address space) of the application that receives the event. This last point is important since events are usually generated inside interrupt handlers, at which time the current address space may belong to a process other than the one receiving the event. In such a situation, it is impossible to refer to an arbitrary process's virtual address space since the currently loaded page tables are for another process's virtual address space (though in-kernel addresses are mapped to the same, protected, virtual addresses in all address spaces). The call-back mechanism allows for the operating system to perform actions associated with an event in the appropriate process context.

Such functionality may be implemented by modifying the implementation of POSIX signals in Linux, but only by adding to the signals mechanism specific knowledge of the actions to be performed upon delivery of each event type. The implementation of this functionality in the SEND mechanism does not require the implementation to know anything about the particular actions associated with these events, making the SEND mechanism more general in this respect. The SEND mechanism is a conduit used by various kernel sub-systems to push events through to applications, unlike the POSIX signals mechanism, which takes full responsibility for signals once they have been posted. With SEND, sub-systems that generate events may maintain responsibility for managing them.

Thus, if a socket is using the SEND mechanism to deliver events,q it may update or revoke an event after it has been queued for delivery. Hence, if something happens to the socket while it has an event queued, it can modify the queued event to reflect all changes to its state. Furthermore, the socket can be notified by SEND when events are delivered, and thus, the socket can keep track of what information about is has been made available to the application. As a result, sockets require memory for only one event inside the kernel. We allocate the memory for this at the time the socket is configured to use SEND (by being bound to an event group – Section 3.6.2). Thus, when a new event is to be generated by a socket, allocating memory ofr the event is not an issue. With POSIX signals memory describing each signal is allocated from limited global pool. Once this pool is exhausted a server such as `phhttpd` must resort to polling (see Section 2.4.2) since the exhaustion of this resource only becomes apparent at the time a new event is to be generated (consequently this event information may be lost).

By shifting more responsibility for managing events from the notification and delivery mechanism (SEND) to the sub-systems (sockets) that generate events, we can more efficiently handle events. As a result, the number of events (describing socket activity) in the system is proportional to the number of file-descriptors not the event rate per file-descriptor since pending events may be modified while they are queued. Thus, the ratio of events to file-descriptors can be held constant.

3.6.1 Event-Driven `accept()`

In order to demonstrate the functionality associated with the implementation of the SEND mechanism we have implemented a mechanism called "auto-accept". In Linux, file-descriptors for new, incoming, TCP/IP connections are made by calling `accept()` on a socket that is listening for new connections. When the socket that is listening for new connections has successfully negotiated a connection, it is flagged as readable by `select()`, `poll()`, and `/dev/poll` indicating that an `accept()` call on that socket will not block. There are two problems with this approach: there is no way to know how many new connections are pending and the state of the new socket is unknown (and therefore must be polled). The second point is a problem for an event-driven application that must explicitly specify that it wants a socket to deliver events (via SEND or POSIX signals). After configuring a socket to generate events, the application must poll for the state of the socket to learn about what has happened to the socket between the time it was created and the time it was configured to generate events. If this is not done, the application is not aware of events for the socket that occurred before it was configured to generate events.

With SEND we can address both issues. The auto-accept mechanism is meant to replace the `accept()` system call and essentially amounts to an asynchronous `accept()` call. The auto-accept mechanism is enabled by a new `ioctl()` command (`SIOCAUTOACCEPT`) applied to the listening socket. Once this has been done, and `listen()` has been called on the socket, the negotiation of a new connection by the kernel will performs an accept operation (the actions usually performed by a call to `accept()`) and delivers the results to the application in a new event type (`EVT_IPACCEPT`). The actual accept operation (creation of the new socket) occurs at the time of the delivery and is triggered by the in-kernel call-back, which is called at that time. When an `EVT_IPACCEPT` event is delivered to an application, it provides the application with the same data it would have otherwise obtained by calling `accept()` (the new file-descriptor and the address of the remote end of the socket). Additionally, the `EVT_IPACCEPT` event carries along with it information about the current state of the new socket (e.g., is it readable?).

Section 3.6.2 describes extensions to `ioctl()` that allow us to have new sockets inherit the event generation properties of their parents (i.e., the socket used to create this socket via `accept()`). This capability avoids the need for us to explicitly configure new sockets to generate events. When combined with the auto-accept mechanism, new sockets that are created through accept operations are pre-configured to deliver event information and we always know their initial state (without polling). This mechanism provides a solution for both issues identified earlier in this section.

Our experience with the auto-accept mechanism reveal it performs two distinct roles: performing an accept operation without an explicit system call and altering the timing of this operation to occur at the time of event delivery. The experimental results and subsequent discussion presented in Chapter 4 reveal that the timing of the accept operation has a significant impact on web-server behavior when it is overloaded.

3.6.2 Making Sockets Event-Friendly

There are still some minor issues to address with regards to how sockets can interact with the SEND mechanism. Our goal in addressing these issues is to allow us to write a web-server that is completely event-driven and never needs to poll the kernel for any information.

Sockets and SEND Events

In applications using SEND, file-descriptors inform applications of events with a new event type called "`EVT_IOREADY`". We use this new event type instead of a `SIGIO` signal event to implement certain semantics not compatible with the semantics associated with `SIGIO`. An `EVT_IOREADY` has associated with it a bit-map describing the state of the file-descriptor (using the same format and meaning as the bit-map used for `poll()`). It is up to the application to keep track of socket states in order to identify the actual events that have occurred. Though it may seem odd to use events to communicate state information, our experience has been that communicating state as opposed to event information makes the kernel and application code simpler.

`EVT_IOREADY` events may be coalesced with other events of the same type (for the same socket) in order to reduce redundant delivery and notification. This means that if a socket generates two `EVT_IOREADY` events indicating that it has data to be read, and these events are generated before the first is delivered, then the two events are coalesced into one. If the application reads data from the socket buffer and empties it before a queued `EVT_IOREADY` event is delivered, that event is revoked by the socket that generated it.

The binding of this event to an event-group is performed by performing an `ioctl()` operation on the file descriptor. (`ioctl()` is used instead of `evtctl()` since we are manipulating the properties of a particular socket rather than the properties of the SEND mechanism.) We use the `F_SETSIG` command to specify the event-group to which EVT_IOREADY events from a particular file-descriptor are assigned. This `ioctl()` command was initially implemented in Linux to bind file-descriptors to specific Real-Time signals.

Inheritance of SEND Configuration

We also have added a new `ioctl()` command (`SIOCEVTCLONE`), that when applied to a socket that is accepting new connections causes the socket to apply to its children sockets the same SEND settings as it has. As a result, sockets for newly accepted connections inherit the SEND-related configuration of their parent and thus, do not need to be configured manually.

Chapter 4

Evaluation and Results

In this chapter, we evaluate our implementation of the SEND mechanism in Linux. We begin by presenting timing benchmarks that highlight the differences between the POSIX signals mechanism and SEND. We follow this with a discussion of how SEND can be applied to a web-server. During this discussion, we identify fundamental architectural semantics that are changed in modifying the web-server to use SEND; we later use these observations to explain the differences in behavior caused by these changes. We present a series of experiments that compare the behavior of our SEND-based web-server with the same web-server based on the `/dev/poll` mechanism. These experiments demonstrate that basing a web-server on SEND, as opposed to other mechanisms, dramatically improves its load handling capacity.

4.1 SEND vs. POSIX Signals

One of our goals in the design of SEND is to develop a more efficient and scalable event notification and delivery mechanism than the signals mechanism. We begin our evaluation of SEND by comparing it to the signals mechanism to identify any efficiency gains arising strictly from the change in interface (as opposed to changes in application behavior and semantics). We have designed an experiment that measures the overhead incurred by the delivery and notification of events to an application. Our experiments consist of the following steps:

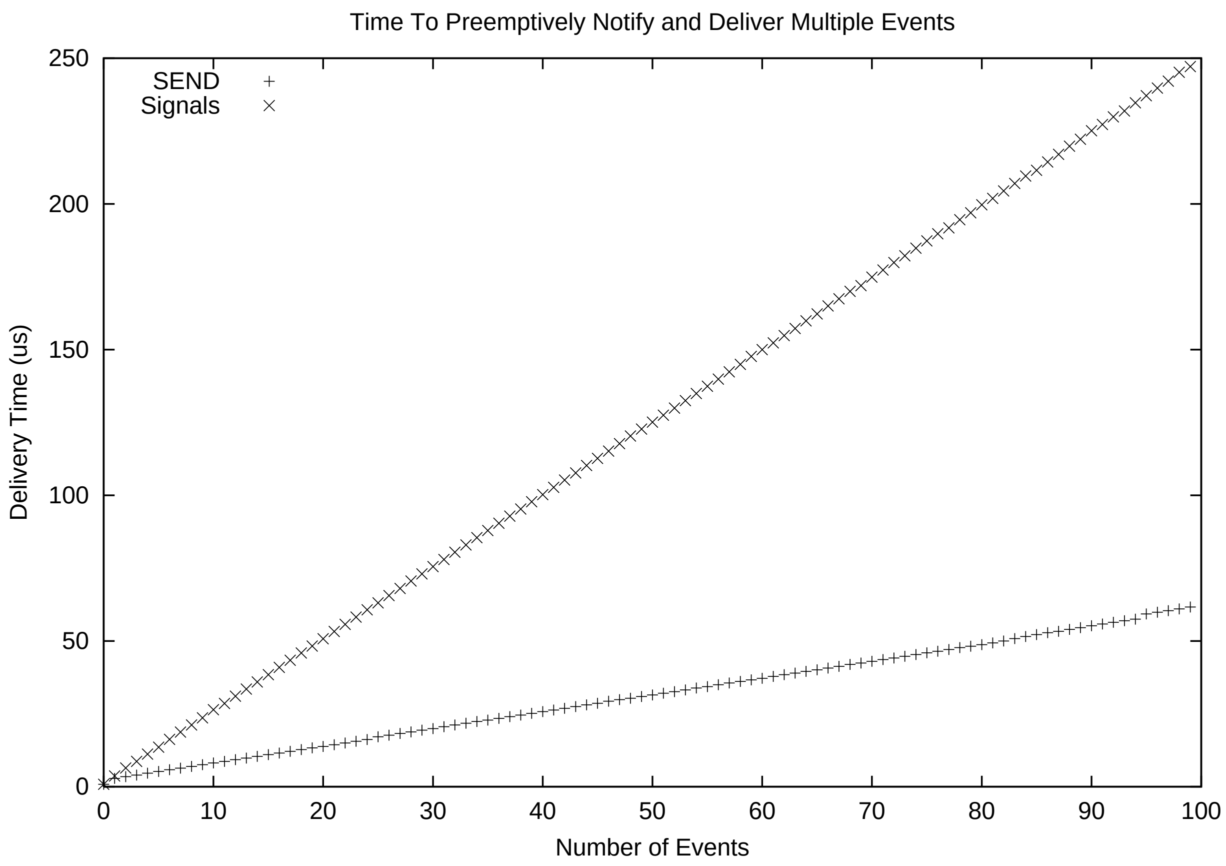
1. An arbitrary number of "SIGINT" signals are queued for delivery and notification (via the signals mechanism or via SEND). The number of signals is an experimental parameter.
2. Once the signals have been queued, the system time is recorded, indicating the start time of the experiment.
3. The application runs (in an idle loop) and the operating system delivers events to the application, preempting the idle loop with the event-handler or signal handler.
4. The application's handling of each "SIGINT" signal consists of incrementing a counter.
5. When the counter value indicates that all signals have been handled the application breaks out of its idle loop and records the system time. We then calculate the elapsed time required to handle all of the queued signals.

In this procedure we measure the amount of time needed to convey information about events from the operating system to the application. Note that we wish to restrict our timing to encompass only the time needed to process the signals and notify the application since we want measurements to reflect the difference in interfaces used by the application. In order to do this properly, we have implemented a special system call in Linux that performs steps 1 and 2. By implementing a special system call we can be certain that the measurement of the start time is taken immediately before the first SIGINT signal is delivered to the application. As a result, we know for certain that no signals have been delivered before the recording of the experiment's start time.

Theses experiments are conducted using Linux 2.4.0-test7 on a 550 MHz Pentium-III with 256 MB of RAM. We vary the number of signals to be handled from 0 to 100, repeating each instance of the experiment 1000 times and reporting the average. We find that during any particular experiment, only very rarely is the observed time to process the events affected by process context-switches, interrupts or page-faults occurring during the experiment. Thus, most of the time, the only operation measured by the elapsed time is how long it takes to actually deliver the event information into the application and perform a notification, as is intended. For these experiments, SEND is configured with a ring-buffer sized to hold 100 events (the entire ECB was 4096 bytes; 1 page); we find that a larger ring-buffer does not improve SEND's results any further. The SEND configuration in this experiment involves running with notification enabled only in the idle loop and delivery enabled at all times.

Figure 4.1 presents the results of these experiments. As expected, SEND was able to deliver the queued events in less time than the signals mechanism (requiring approximately one fifth of the time required of the signals mechanism). We thus deduce that in delivering signals, the POSIX signals mechanism spends four fifths of its time keeping track of application state (i.e., what signal handler, if any, is running?) and switching between kernel mode and user mode to de-queue the next signal. On the other hand, the time spent by SEND was primarily devoted to copying data into the ECB. Our implementation of SEND is not optimized in this regard; the delivery of each event involves copying approximately 40 bytes of data, the amount of data necessary to be able to fully describe any event.

Figure 4.1 SEND vs. POSIX Signals De-Queuing Time



In configuring these experiments, we found that increasing the ECB size to hold more than 100 events in the ring buffer did not improve the performance of the SEND mechanism. The most likely explanation for this is that we could not handle more than 100 events at a time before experiencing an interrupt that would bring the process into kernel mode. With the process in kernel mode, it is able to deliver more events to the application prior to switching to user mode and as a result the process cannot make use of a larger ring-buffer. This is not to say that a larger ring-buffer could not be useful for other applications, instead it is an indication that programmers using SEND should experiment with ring-buffer sizes to find one suitable for their application.

4.2 Demonstrating SEND in a Web-Server

Our intention is to evaluate the role of a purely event-driven kernel interface as a means for improving the behavior of web-servers in serving large numbers of clients (corresponding to a high request rate). In order to demonstrate the advantages presented by SEND to a web-server, we must have something to compare it against. To this end we demonstrate the advantages of a SEND-based design by modifying an existing web-server; `thttpd` [26]. Our choice of web-server is influenced by several factors:

1. `thttpd` is designed to work with `select()` (and `poll()`) and thus, is already design to work in an event-driven manner. (Note that the original design must translate the state-based semantics of `select()` and `poll()` into an event-based view of the world.)

2. The implementation of `thttpd` is modular and well-suited to the modifications we are interested in. The original design encapsulates the specific semantics of `select()` and `poll()` in an internal API. The transformation of `thttpd` to use SEND required only minor modifications to this API and isolated the majority of the `thttpd` changes to the implementation of this API.

3. Provos and Lever [27] used `thttpd` as the basis of their experiments (using `/dev/poll`). We ported their `/dev/poll` implementation to recent development Linux kernels (2.4.0-test7; for which our implementation of SEND was developed). Our port of `/dev/poll` is largely unchanged from the original code by Provos and Lever; most of the changes were required to adapt to new or modified kernel-internal interfaces. Our tests with Provos and Lever's implementation of `/dev/poll` for Linux 2.2.14 indicate that `thttpd` running with `/dev/poll` on Linux 2.4.0-test7 is capable of handling a higher rate of client requests than the version for Linux 2.2.14 (due to other improvements in the newer kernels).

We considered using `phhttpd`, which uses the signals mechanism as its multiplexing mechanism. At first this may seem like a better combination of application and multiplexing mechanisms against which we should make our comparison. However, Provos and Lever [27] demonstrate that this particular application is incapable of handling request rates as high as the `/dev/poll`-based `thttpd` can handle. An examination of the source code to `phhttpd` reaffirms this conclusion: `phhttpd` incurs substantial overhead in explicitly dequeuing signals one at a time (using `sigtimedwait()`), must configure each new socket to generate signals, and must poll sockets for information that may have generated signals prior to configuration.

Additionally, `phhttpd` resorts to using `poll()` if it cannot keep up with the rate at which events are being generated and it has been previously shown that the scalability characteristics of `select()` and `poll()` are poor [9, 10, 27]). This is especially important since Provos and Lever observe that `phhttpd` resorts to polling at request rates that were easily handled by the `/dev/poll`-based server, indicating that `phhttpd` is unable to extract signals from the kernel as fast as they are being generated. As a consequence, we have a strong indication that the concerns about the applicability of the signals mechanism that were raised in Section 2.4.3 are evident in this application.

The other alternative mechanism that requires consideration is the one proposed by Banga *et al.* [10] (Section 2.3). Their implementation is unavailable to us for direct comparison, however the `/dev/poll` mechanism as implemented by Provos and Lever [27] is a close approximation of the mechanism proposed by Banga *et al.* [10] (see Section 2.3.1). Furthermore, as described in Section 3.5, we can implement a mechanism nearly identical to the one proposed by Banga *et al.* [10] using SEND with preemptive notification disabled.

4.3 SEND in `thttpd`

Chapter 2 discusses the basic structure of Internet-server applications. Figure 2.1, which presents a pseudo-code outline of such an application structure, is in fact based on `thttpd`. Before presenting the application structure for a SEND-based web-server, there are two aspects of a polling-based design that we would like to point out:

- file-descriptors are handled in batches (performing a polling operation upon completion of the current batch) and,
- the order in which connections are handled is essentially determined before processing of the current batch begins.

Both of these points arise in a polling-based design new information about connection events or states, for the most part, enters the application only at the time of the polling operation. Our SEND-based design does not face this restriction since preemptive notification allows it to receive and incorporate new information into its activity scheduling as the information becomes available.

Figure 4.2 Outline of a SEND-Based `thttpd`

```
/* Simplified event-handler */
void event_handler () {
    for each new event{
        if (evt.type == EVT_IOREADY) {
            add_to_fd_queue(evt.fd, evt);
        } else if (evt.type == EVT_IPACCEPT) {
            add_to_newconn_queue(evt.fd, evt);
        }
    }
}
void main () {
    configure_send();
    ecb->notify = ecb->deliver = allow_notif_delivery;
    while (1) {
        /* Look at EVT_IPACCEPT event */
        q = dequeue_from_newconn_queue();
        if (q != NULL) {
            /* Create data-structures describing the new
             * connection.  No accept() call is needed. */
            next_conn = config_new_conn(q);
            if (no_data_to_read(next_conn))
                continue;
        } else {
            /* No new connection, process an old one. */
            next_conn = dequeue_from_fd_queue();
        }
        if (next_conn) {
            work_on_connection(next_conn);
        } else {
            ecb->notify = ecb->deliver = 0;
            if (ecb->head != ecb->tail) {
                 /* Nothing to do, block until next event.
                  * evtctl will not call event handler.
                  * This call blocks until head!=tail */
                 evtctl(WAIT_FOR_EVENT, ~0);
            }
            /* Call handler to process events */
            event_handler();
            ecb->notify = ecb->deliver = allow_notif_delivery;
        }
    }
}
```

Note that this figure presents an overly simplified view of the web-server. In particular, it excludes mechanisms to delete connections after they have timed-out and logic for manipulating SEND and the ECB in the event-handler function.

Figure 4.2 shows a pseudo-code outline of our SEND-based `thttpd`. Working in an event-driven environment offers to potential to avoid some inefficiencies introduced by translation between a polling kernel interface and event-driven application semantics. In order to explain the differences in behavior between SEND-based `thttpd` and `/dev/poll`-based `thttpd` observed later in this chapter, we now outline how these designs differ and how some functionality has been re-implemented in an event-driven environment:

- In the SEND-based version of this server, when the event-handler is invoked, it does not force the application to immediately handle the connections that triggered the latest events. Instead, the event-handler performs a scheduling operation (en-queuing the event data) which alters the behavior of the application once its state is restored. The event-handler in this case functions very much like a low-level interrupt-handler in an operating system; it does the bare minimum necessary to allow future events to occur safely and efficiently and leaves most of the computation to another execution context.

- We are using the event-driven alternative to `accept()` (Section 3.6.1) and thus, when we receive an event describing an incoming TCP connection, there is no need to perform an accept operation (by calling `accept()`). Consequently, when we de-queue a connection from the queue of new connections we only need to configure the application-internal data structures defining a connection when calling `config_new_conn()`. Note that the primary effect of the auto-accept mechanism is to alleviate the race condition described in Section 3.6.1 without performing a polling operation. The new socket is produced by the accept operation that occurs at the time of event delivery. This is nearly equivalent to the application receiving an event telling it that it may call `accept()` without blocking and then calling `accept()` *inside the event-handler.*

- Note that the first event for a connection is an `EVT_IPACCEPT` event and all subsequent events are `EVT_IOREADY` events and thus, we have two queues in the server; one for new connections and one for current connections. By using two separate queues we allow the application to give preference to new connections in order to be consistent with other `thttpd` variants. In fact, this design accomplishes this more effectively since we can react to new connections as soon as possible. Crovella *et al.* [13] have shown that connections active for some time are more likely to be high-latency connections or requests for larger files and web-server performance is likely to improve by giving priority to new connections, which are low latency or for small files.

- There is no concept of a current "batch" of connections to be handled as there is in a polling-oriented design (note the absence of an inner loop as in Figure 2.1). The event-handler may be constantly adding elements to the queues as the rest of the application works to remove them. Since there is no "batch" of work to do, there is no convenient way of keeping track of how long a connection has existed. Polling-based variants of `thttpd` keep track of time by checking at the completion of each batch. Since our SEND-based `thttpd` doesn't have a batch of requests to work with, it does not know when to check the time and thus, in order to avoid excessive polling (and maintain purely event-driven semantics) our server uses interval timers to keep track of time.

It is important to remember the last point since we demonstrate in Section 4.7 the switch from a polling "batch-oriented" design to a continuous, "incremental" approach to scheduling significantly alters the behavior of the application. In a batch-oriented design, if the request rate exceeds what the server can process, then the batch size grows significantly, potentially degrading the web-server's responsiveness.

4.4 Experimental Design

We now outline the experimental design used to demonstrate how we use our SEND mechanism to build a web-server that is better able to handle high rates of client requests.

Each experiment we conduct consists of applying a known workload to a web-server and observing the behavior of that server under the particular load. For the purposes of these experiments, a workload is a sequence of requests made by a client to a server for some set of files. In these experiments, a workload can be characterized by the total number of requests, the rate at which requests are made and the files that are requested. The number of requests is usually fixed and we have a small number of files to be requested. Therefore, our experiments will primarily focus on how changing the request rate affects the web-server.

A server is able to respond to requests at a particular rate (which we refer to as the "response rate"). We refer to the point beyond which the web-server cannot sustain a response rate equal to the request rate for a particular set of requested files as the "saturation point". The saturation point is a indication of the load bearing capacity of a web-server (under a particular workload). We are also interested in understanding the behavior of web-servers when the request rate exceeds the saturation point; a better web-server design reacts gracefully when it is overloaded (i.e., when the request rate exceeds the saturation point). If a web-server is capable of handling temporary overloads gracefully, we can safely let it run much closer to its saturation point for long periods of time knowing that a minor fluctuation in client activity does not affect it substantially.

Our basic experimental procedure is to apply a workload to a web-server over a range of request rates and to observe the results. The software we use to generate requests and measure the results is `httperf` [25]. We have chosen this particular client software for two reasons; it was used by Provos and Lever [27] in their experiments with the `/dev/poll`-based implementation of `thttpd` and we have found this software to be the best suited for reliably and predictably imposing very heavy loads on a web-server in our environment.

The request rate is not the only factor determining the intensity of a workload imposed on a web-server. We also consider the files being requested since this determines how much work the server must do per request (in particular due to file size). Requests for small files stress the operating system and web-server more than requests for larger files, which stresses the network and TCP/IP kernel sub-system. This is because the cost of serving a request consists of a per-request overhead component (e.g., accepting a new connection and processing the request) and a per-byte cost of transmitting the results back to the client. With smaller files the cost per byte is higher since the overhead costs are spread over fewer bytes.

We begin by examining simple workloads that request a file of size 1 KB, 2 KB, 4 KB, 6 KB or 8 KB. The results of these experiments demonstrate the effect of this parameter on web-server behavior and isolate the differences in web-server behavior caused by the changes in multiplexing mechanisms from the effects of network congestion. In Section 4.9, these experiments are reevaluated in the context of a more realistic workload.

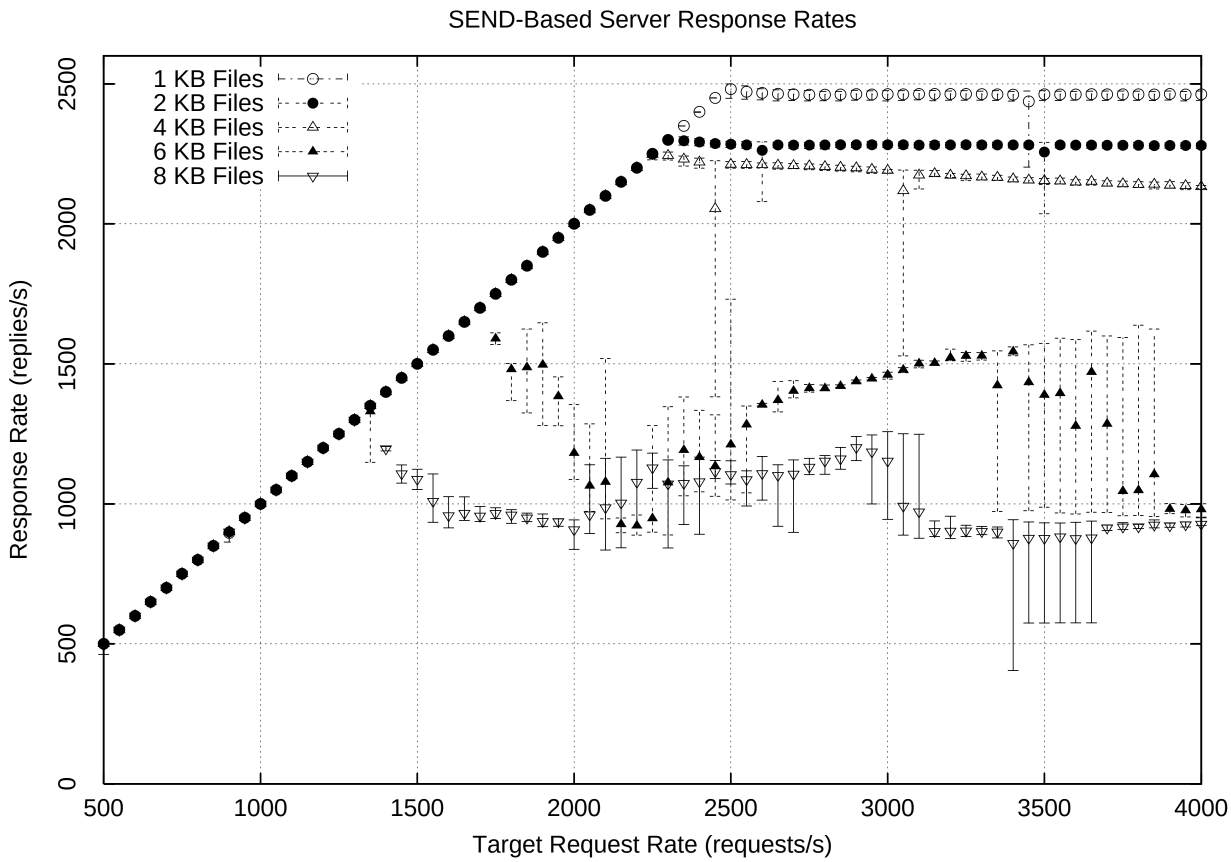
With the workload and request rate determined for each experiment we configure our load generator to make a total of 25,000 requests. This design is similar to the workload of 35,000 requests used by Provos and Lever [27]. Our preliminary experiments showed that reducing the number of requests per experiment did not noticeably affect the results. However, running 25,000 requests as opposed to 35,000 requests per experiment made it easier to manage the complete set of experiments since under very high request rates host running the `/dev/poll` server occasionally crashed (apparently due to problems in the TCP/IP stack). In almost all experiments, 25,000 requests are successfully fulfilled with only small set receiving a small number of "connection refused" errors. These errors occurred in experiments where network congestion can be detected; we have no indication that these few errors have any noticeable impact on our results.

Unless otherwise specified, in our experiments the web-server is running on a single-processor Pentium-III at 550 MHz with 256 MB RAM, with an Intel EtherExpress Pro/100 NIC connected to a switched 100 Mbps network. The client runs on a dual-processor version of the server machine. Aside from the `ssh` connections used to monitor and control the experiments, no other spurious network traffic occurred on the network. The operating system is based on RedHat 6.1 with a version 2.4.0-test6 of the Linux kernel. This kernel has been modified this kernel by applying patches to upgrade some components to 2.4.0-test7 in order to address noticeable bugs (particularly virtual-memory issues) and we have added support for SEND and `/dev/poll`.

Our decision to base this work on a development kernel stems from the fact that it is better able to tolerate the loads imposed on them by our experiments than the stable 2.2 kernel series. Our preliminary experiments involving clients running a stable 2.2 series kernel could not successfully complete the same range of experiments.

We decided to run our experiments with a single client running `httperf` connecting to our server machine. Preliminary tests showed that using a single client is sufficient to generate the required loads on our web-server. We observed that splitting the load generation duties among multiple client machines did not alter the aggregate measured behavior of the server. Since we could not discern any difference between having one client or several, we chose to use one in order to simplify the process of automating experiments.

Figure 4.3 SEND-Based Server Response Rates



4.5 /dev/poll vs. SEND: Response Rates

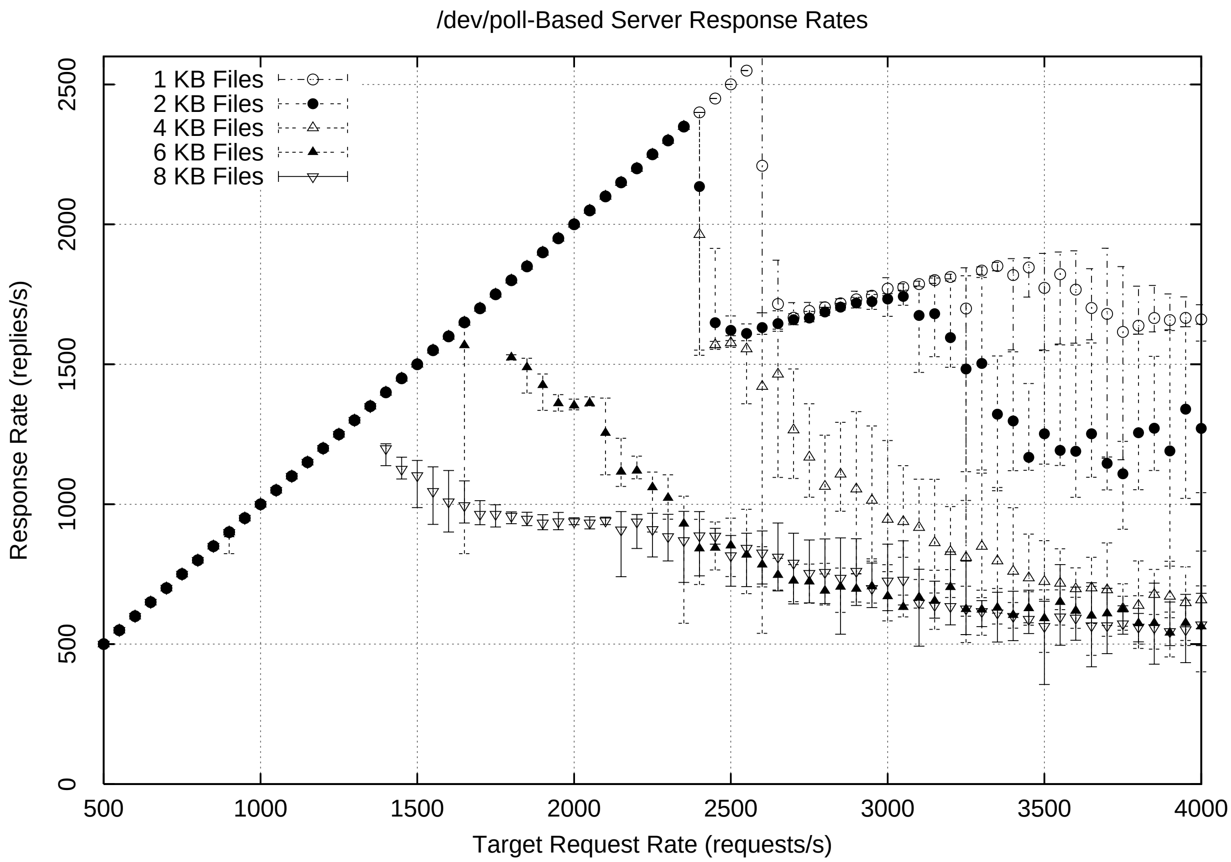
We now present the results of our experiments in which we examine the behavior of the /dev/poll-based and SEND-based servers under workloads based on file sizes of 1 KB, 2 KB, 4KB, 6KB and 8 KB. The response rates of the two servers under these different workloads are presented in Figures 4.3 and 4.4.

These figures indicate the SEND-based server is significantly better suited for handling high client request rates where network congestion is not a limiting factor, since it does not exhibit significant degradation in response rate for request rates in excess of the saturation point. The difference in behavior is substantial and requires explanation. As a starting point we draw the reader's attention to the following features of these figures:

- The saturation point of the SEND-based server is slightly lower than that of the /dev/poll-based server (under 5%). We believe this is due to the fact that the SEND-based server experiences slightly higher overhead per event delivery than the /dev/poll-based server. This situation is a consequence of a longer code path within the kernel required to generate and deliver a SEND event as opposed to a /dev/poll event. Furthermore, the SEND mechanism is apt to be invoked to deliver events more frequently than the /dev/poll mechanism (the polling semantics associated with /dev/poll force some events to be queued for delivery, whereas SEND events may be delivered immediately resulting in fewer events per delivery).

  This indicates that the two multiplexing mechanism exhibit comparable efficiency in delivering event data to applications. In order to explain the differences we must consider differences in application and system behavior that are introduced by switching from the /dev/poll mechanism to SEND.

Figure 4.4 `/dev/poll`-Based Server Response Rates

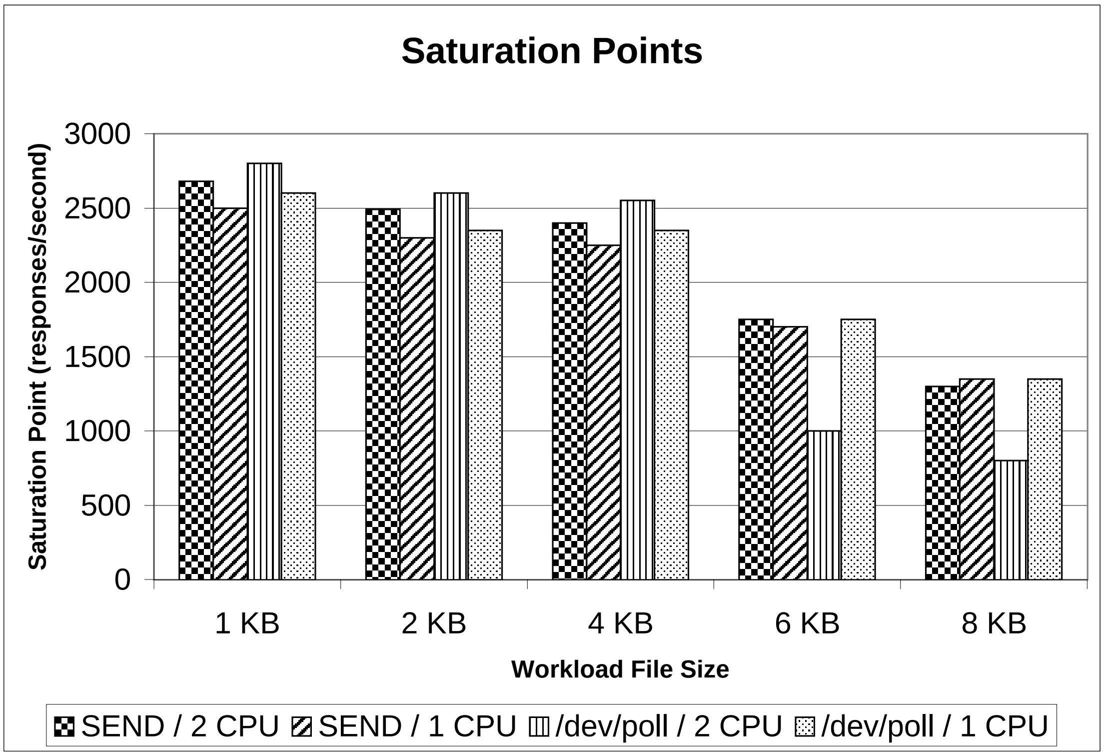


- With workloads based on 1 KB, 2 KB and 4 KB files the SEND-based server is capable of maintaining a response rate nearly equal to the saturation point at request rates in excess of the saturation point. On the other hand, the `/dev/poll`-based server experiences a substantial degradation in response rates. We suspect that the factor limiting the server response rate in these experiments is CPU availability. To test this claim we outfitted the server machine with 2 CPUs and determined the two servers' saturation points under the same range of workloads. The addition of a second CPU allows the kernel to split its execution time between the two CPUs thereby leaving slightly more CPU time for a single-threaded server. The results of these experiments are presented in Figure 4.5 and indicate that under these workloads (where we suspect the limiting factor is available CPU time) providing the servers with more CPU time increases their response rates, thereby confirming our hypothesis.

  Having identified that available CPU time is a factor limiting the performance of the web-servers under these workloads, we see no reason why the `/dev/poll` server should not be able to sustain its response rate as the request rate grows in excess of the saturation point. The fact that available CPU time is at issue, rather than available network bandwidth, leads us to conclude that we must focus on the differences between the multiplexing mechanisms in the two web-servers. These multiplexing mechanisms are pivotal in determining how these application use CPU time, and we must thus look to them to explain the discrepancies in performance profiles. We begin this discussion in Section 4.6.

- With workloads based on 6 KB and 8 KB files both server types exhibit a degradation of response rate when the request rate exceeds the saturation point. Figures 4.6 and 4.7 show the throughput attained by the servers. From these figures we see that when the workload consists of 6 KB and 8 KB files the saturation points correspond to throughput above 80 Mbps. These throughput figures

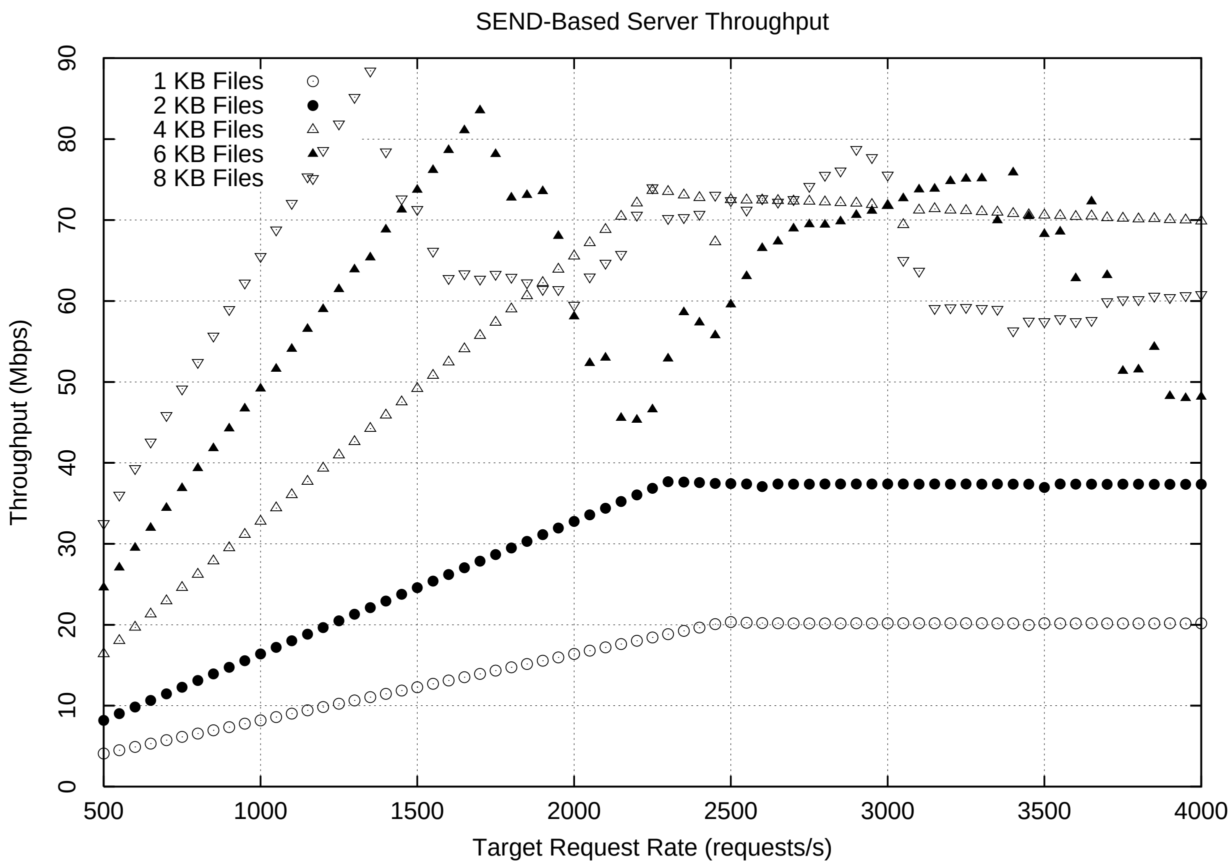
Figure 4.5 Effects of an Additional CPU on Saturation Points



A comparison of saturation points of the SEND-based and `/dev/poll`-based servers running on dual-processor and single-processor hosts (thereby providing the application with more CPU time by running some kernel processing on the other CPU). We suspect that the degradation in saturation points for the `/dev/poll`-based server under 6 KB and 8 KB workloads with 2 CPUs is a result of in-kernel lock-contention issues.

are application-level throughput (excluding connection initialization and TCP/IP header overhead) on a 100 Mbps network. If we account for connection initialization and TCP/IP header overhead, then in all likelihood, the network is nearly congested at these saturation points. Consequently, if we attempt to increase the request rate we are attempting to use network bandwidth in excess of what is available, leading to congestion. Therefore, the factor limiting the saturation point in these experiments is network bandwidth and it is our belief that the most likely explanation for the drop in response rates and throughput is an effect of TCP flow-control mechanisms invoked in response to congestion detection.

Figure 4.6 SEND-Based Server Throughput

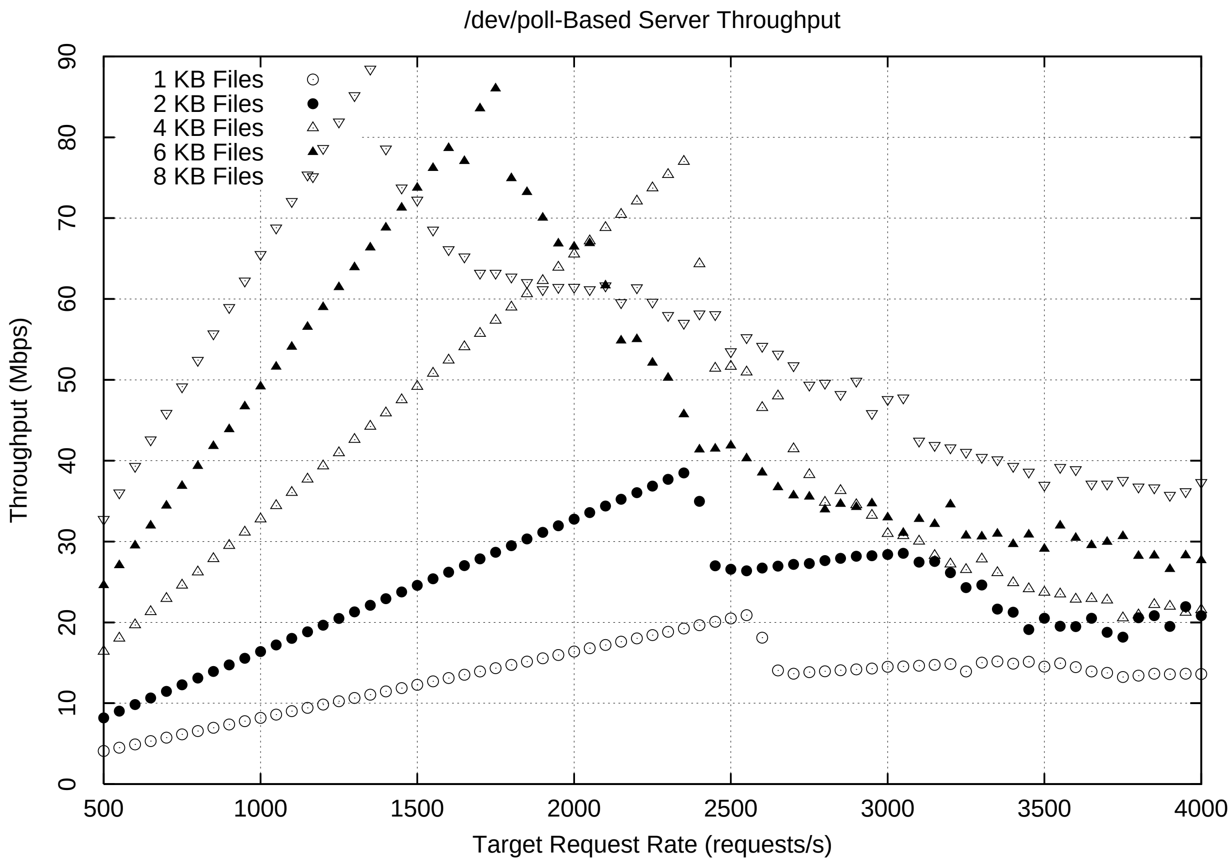


4.6 Response and Connection Times

The results presented in the previous section demonstrate that our SEND-based server is better able to handle request rates in excess of the saturation point for workloads where network congestion is not an issue. In this section we present results that illustrate why our SEND-based server is able to out-perform the `/dev/poll`-based server under such circumstances.

We begin our investigation by looking at the time required to obtain a response (as observed by the client). Figures 4.8 and 4.9 present the average connection and response times observed by the client workload generator. "Connection time" refers to the time needed to establish a TCP connection and "response time" refers to the time needed to receive a response to a request. Both of these times are measured from the point at which the request for a connection is first issued to the time all of the data has been received. Note

Figure 4.7 `/dev/poll`-Based Server Throughput



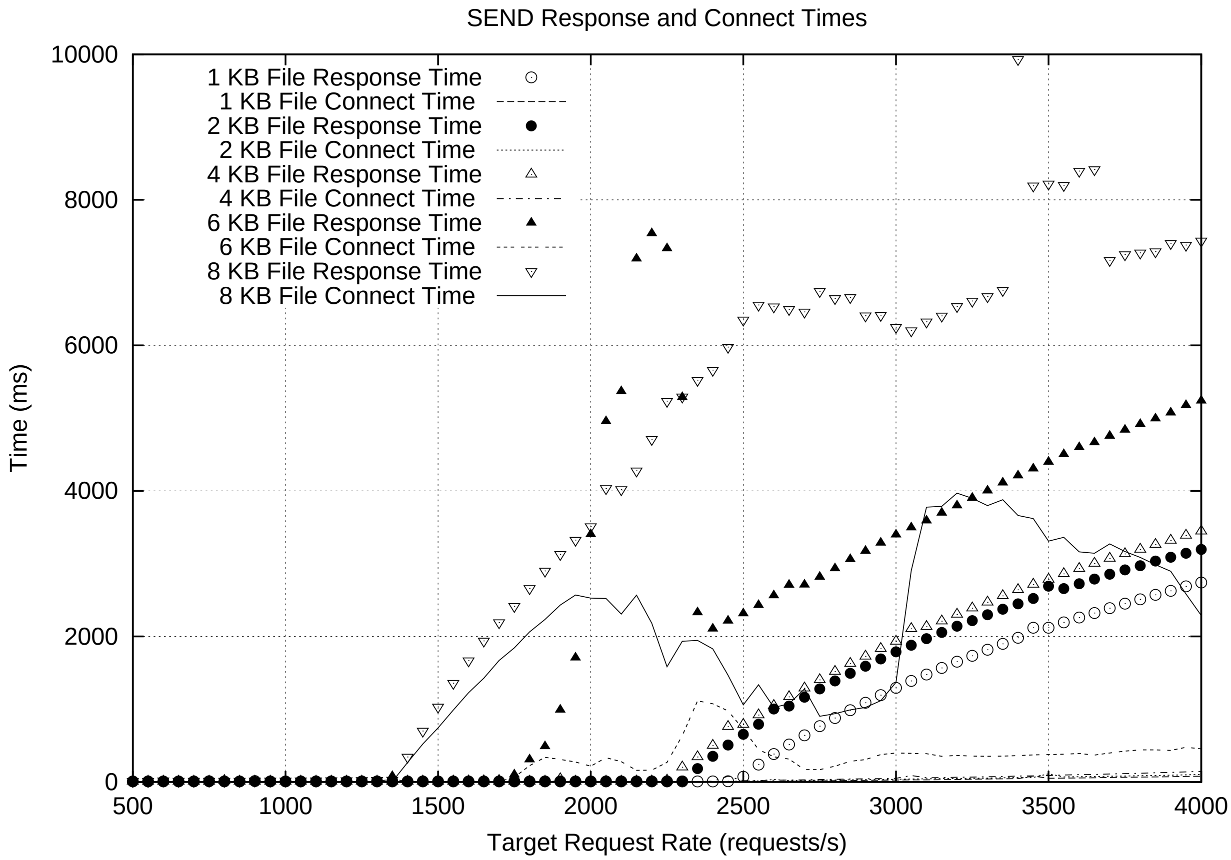
that the results presented in these figures are apt to vary as the total number of connections in the workload varies (thereby varying queuing delays).

Figures 4.8 and 4.9 show that when the saturation point for a particular workload is exceeded the average response time observed by the client increases. Generally, the average response times observed for the SEND-based server are equal to or slightly lower than the average response times for the `/dev/poll`-based server. This small discrepancy is not significant enough to explain the wide variation in response rates seen between Figures 4.3 and 4.4.

The response times observed for the two servers do not differ significantly, even though one may expect that the higher response rate achieved by the SEND-based server (Figures 4.3 and 4.4) would result in better response times. This indicates that the SEND-based server does a better job of multiplexing connections; handling multiple connections at the same time. This is naturally reflected in the throughput observations of Figures 4.6 and 4.7.

However, if we look at the connection times, we see that the SEND-based server maintains quite low average connection times for 1 KB, 2 KB and 4 KB workloads (where we have claimed there is no network congestion). On average, with the `/dev/poll`-based server connection times are substantially higher. The reason for this is that the SEND-based server reacts immediately to incoming connection requests (performing an "accept" operation via the auto-accept mechanism) and thus, clears the queue of accepted TCP connections. In doing so the SEND-based server prevents this queue from overflowing its length limit (its length is limited inside the Linux kernel to 128 pending connections). Since we are using preemptive event-notification and delivery, the SEND-based web-server receives delivery and notification of an `EVT_IPACCEPT` event describing a new connection as the kernel returns to the application after handling the interrupt that

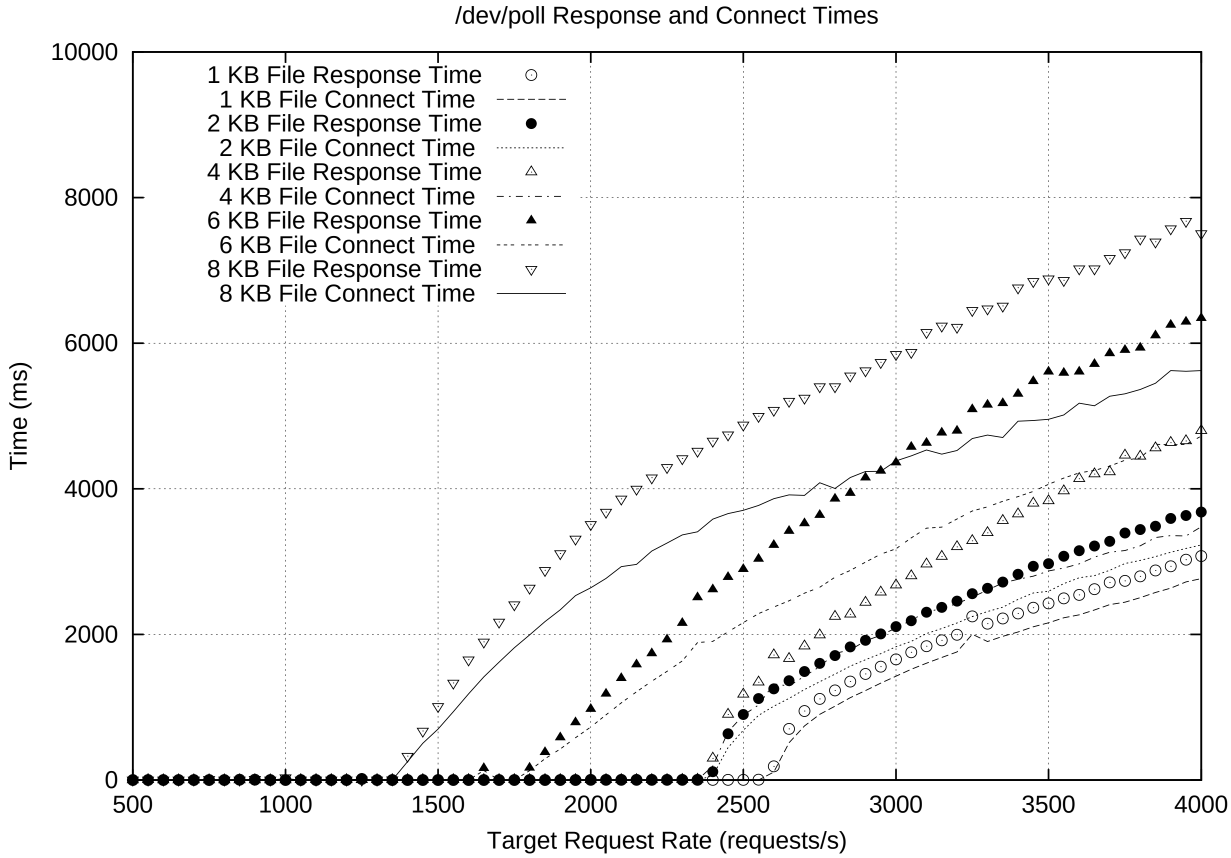
Figure 4.8 SEND-Based Server Mean Connection and Response Times



Note that the connection times for each of the workloads are substantially lower than the corresponding response times.

We suspect that the fluctuations in mean response time for the 6 KB and 8 KB workloads are a result of network congestion triggering TCP congestion avoidance mechanisms. Also, note how the fluctuations in connection time with the 6 KB and 8 KB workloads coincide with fluctuations observed in Figure 4.3. Our suspicion is that at some request rates some TCP connections back-off sufficiently to allow some other connections to proceed without experiencing congestions.

Figure 4.9 `/dev/poll`-Based Server Mean Connection and Response Times



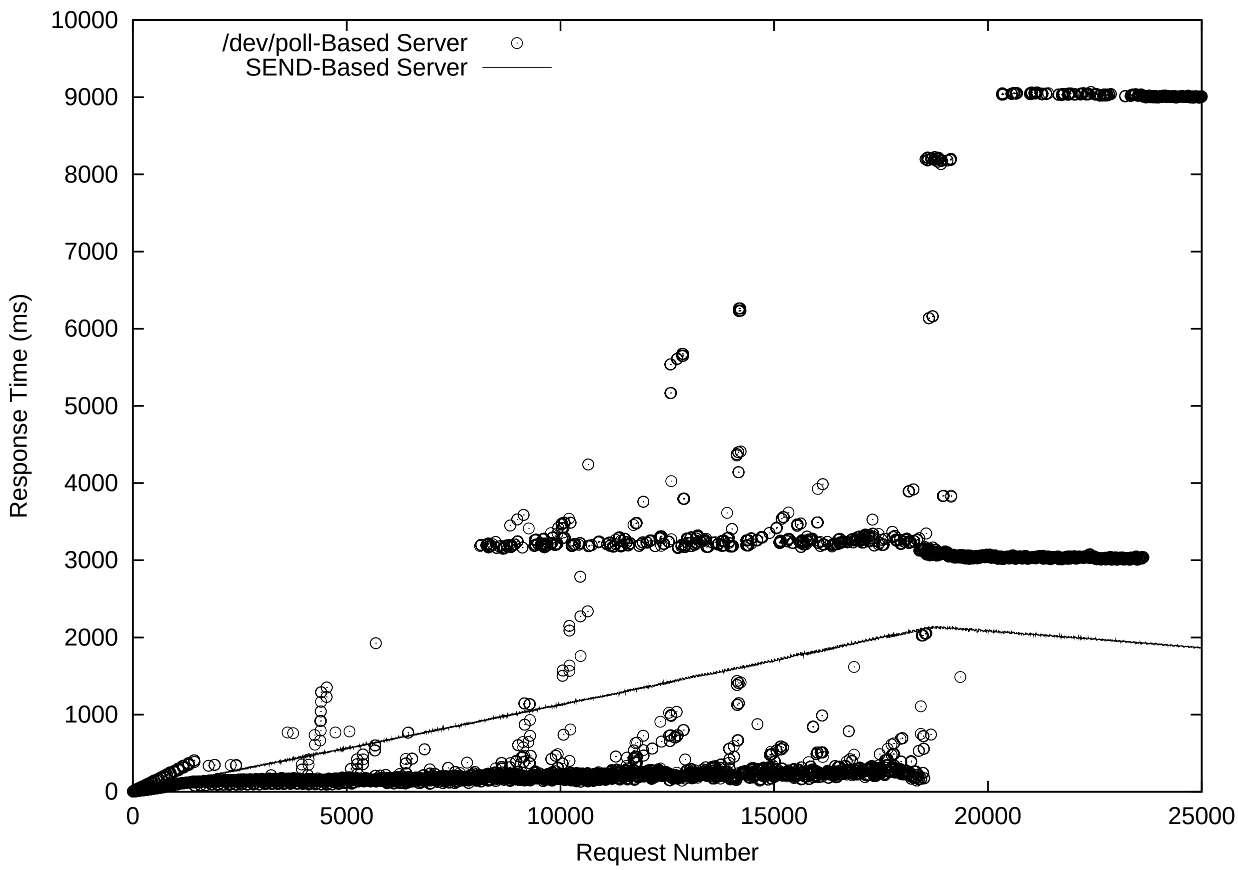
received the packet that completed the TCP three-way handshake. As a result, the in-kernel queue of incoming connections is cleared as soon as possible.

A polling server on the other hand cannot react to events in this manner since it must finish handling the current batch of events before polling again (and recognizing that it should call `accept()`). If a polling server cannot process requests as fast as it receives them, the number of connections that are waiting to be accepted at the time the server performs a polling operation grows. Eventually, this causes the queue for incoming connections (inside the kernel) to overflow. When this overflow occurs, the kernel drops packets for incoming connection requests, ultimately resulting in TCP congestion control mechanisms being invoked. This results in a prolonged connection time and consequently, a prolonged response time, thereby reducing the overall response rate.

To verify this claim, the web-server was run on a kernel modified to print an error message when the incoming connections queue overflows and a connection request had been dropped. We then ran several experiments with the `/dev/poll`-based server and found the kernel reporting these errors almost immediately when the response rate exceeded the saturation point. On the other hand, the SEND-based server never resulted in the incoming connections queue reaching its limit. Although printing such a message may substantially affect kernel behavior, we are only concerned with whether or not any queue overflow occurs at any time during an experiment. The printing of the first of these diagnostic messages during an experiment concludes the experiment (since we know a queue overflow has occurred). The resulting change in kernel behavior due to the need to print such a messages is therefore irrelevant.

One may think that the solution to this problem with the `/dev/poll`-based server is to increase the upper bound of the accepted connection queue maintained by the kernel. However, this does not solve the problem

Figure 4.10 Per-Connection Response Times (`/dev/poll` vs. SEND)



The x-axis identifies requests in order that they are completed, which may vary substantially from the order in which requests are initiated. The request rate is 3,000 requests per second.
A particular point on the graph indicates that the x'th connection that completed took y milliseconds.

because every time the server polls, there are now potentially more connections in this queue than before and it takes longer to process more connections. The queue still grows and eventually overflows, albeit a bit later. We experimented with this option and found that increasing the upper bound on this queue resulted in an even more dramatic decrease in server response rates above the saturation point (presumably since this increases the batch sizes the server is handling and thus, reduces responsiveness).

Finally, the discussion so far has only considered the average behavior observed over *all* connections. One may wonder whether the differences in behavior between the `/dev/poll`-based and SEND-based servers are due to the preferential treatment of certain connections (i.e., is the server being fair to all connections?). To consider this, we present Figure 4.10 illustrating the response times of individual connections from a specially instrumented **`httperf`** for a typical set of experiments (1 KB file workload, 3,000 requests/second). This figure demonstrates that our SEND-based server is capable of efficiently queuing and handling the connections it receives. Our SEND-based server does a better job of treating all connection requests equally whereas the `/dev/poll`-based server is capable of handling some requests very quickly but is apt to impose significant delays on other connections and in general appears to be more eratic than the SEND-based server. We believe the simpler, more well-behaved dynamics presented by our SEND-based approach are indicative of the fact that we are using a truly event-driven application architecture without the need to deal with polling semantics.

Figure 4.11 Differences Among SEND-Based Servers

Server	Delivery	Notification
Original SEND-based	Immediate	Preemptive
PS1	Batched	Batched
PS2	Immediate	Batched

"Batched" delivery and notification means that the action is performed in a batch oriented manner when the application has run out of things to do and polls the kernel or the ECB.
"Immediate" delivery means that events are delivered into the application's ECB as soon as possible (usually this means right after they are generated).

4.7 Variations on SEND-Based `thttpd`

By disabling preemptive event notification, as described in Section 3.5, we can use SEND to emulate the event-polling mechanism suggested by Banga *et al.* [10]. This allows us to consider another event-polling-based variant of `thttpd` by disabling preemptive event notification in our SEND-based `thttpd`. This section considers two new variants of our SEND-based `thttpd` for investigating the effects of batching of accept operations:

Polling SEND 1 (PS1): In this server, the `notify` and `deliver` fields are set to 0 at all times except when the application polls the kernel for new events using `evtctl()` (when it has run out of events to process). This server simulates the mechanism proposed by Banga *et al.* since all notification and delivery occurs when the server runs out of events to process and explicitly polls the kernel.

Polling SEND 2 (PS2): In this server, only the `notify` field is set to 0. The server is notified of new events only when it runs out of events to process and decides to look at the ECB on its own. However, delivery is enabled and thus the server only needs to poll the ECB for events already delivered (while the server was processing other events).

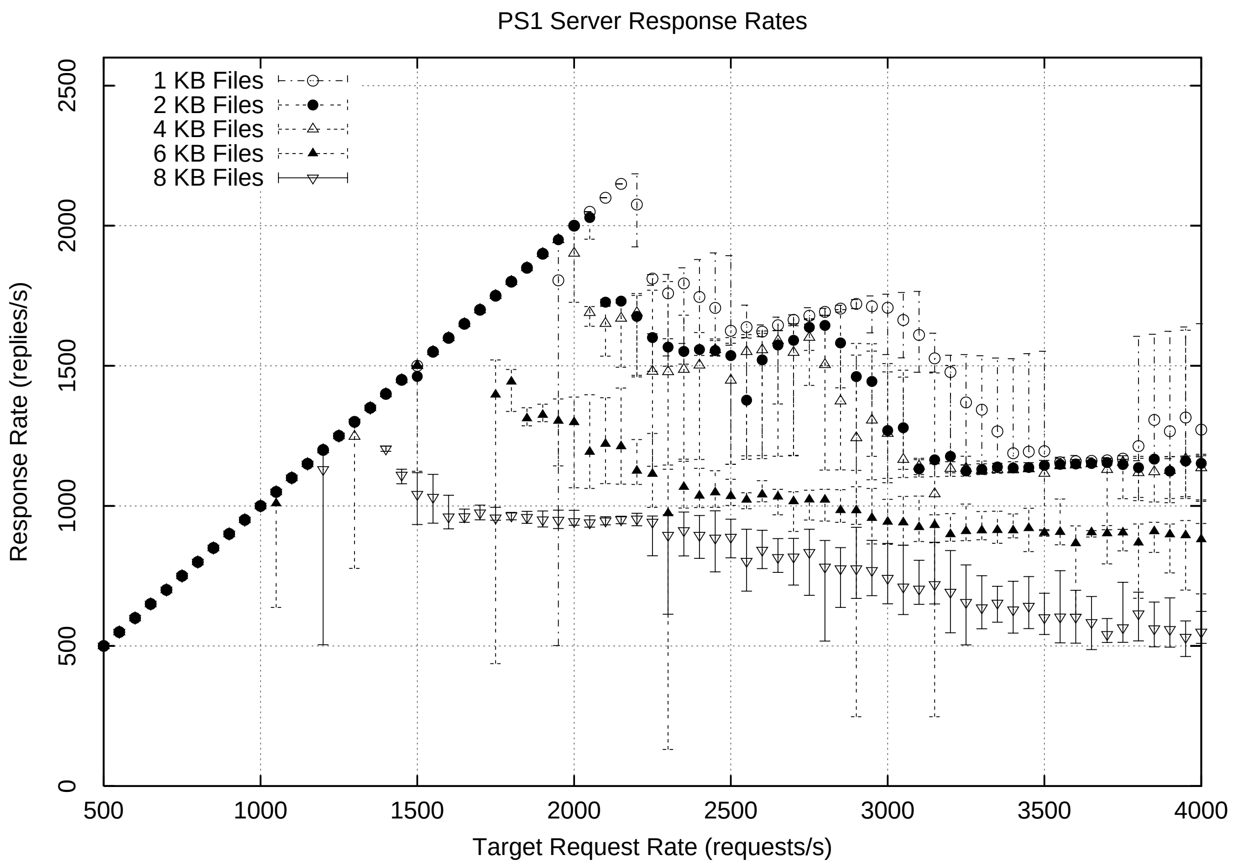
Our original SEND-based server has delivery and notification enabled for all events (almost all of the time). This structure means that as events are generated within the kernel they are passed on to the application as soon as possible and the application is made aware of these events as soon as possible. This is not so in PS1 and PS2, as summarized in Figure 4.11. In these servers, delivery and notification is potentially delayed and subject to batching effects. PS1 restricts the opportunity for event delivery to the explicit kernel polling operation performed via `evtctl()` (see Fig. 4.2), and all notification is restricted to the associated explicit call to the event-handler function (as depicted in Figure 3.5). In PS2 we restrict notification as in PS1, though we allow for event delivery to occur at any time.

What these difference amount to is a change in the timing of accept operations. Since accept operations (when using auto-accept) are performed at the time of delivery of `EVT_IPACCEPT` events, our original SEND-based server and PS2 have these operations performed as soon as possible. On the other hand, in PS1, the timing of these operations is subject to batching effects of a polling-based design.

It is very important to recognize that the only material difference between PS1 and PS2 is the timing of these accept operations. Consequently, if we have any differences in the behavior of PS1 versus PS2 these differences can all be attributed to the difference in timing of event delivery (delivery of `EVT_IPACCEPT` events in particular). Note that the difference in delivery timing between these two applications only affects the timing of the accept operations. Although we are allowing other events (`EVT_IOREADY` events in particular) to be delivered at any time, the delivery of such events has no effect on the application until notification occurs (and we know that PS1 and PS2 use the same semantics for notification). Thus, if we observe any differences between PS1 and PS2, it is due to the fact that these two applications use different semantics for scheduling accept operations.

It is important to recognize that PS1 and PS2 may behave differently in other ways that could affect their performance. In particular, these two servers see different batches of events every time they look for new events (especially with respect to `EVT_IOREADY` events). However, any such differences in behavior are a side-effect of the difference in timing of accept operations which is the ultimate cause of all differences between these two servers.

Figure 4.12 PS1 Server Response Rates



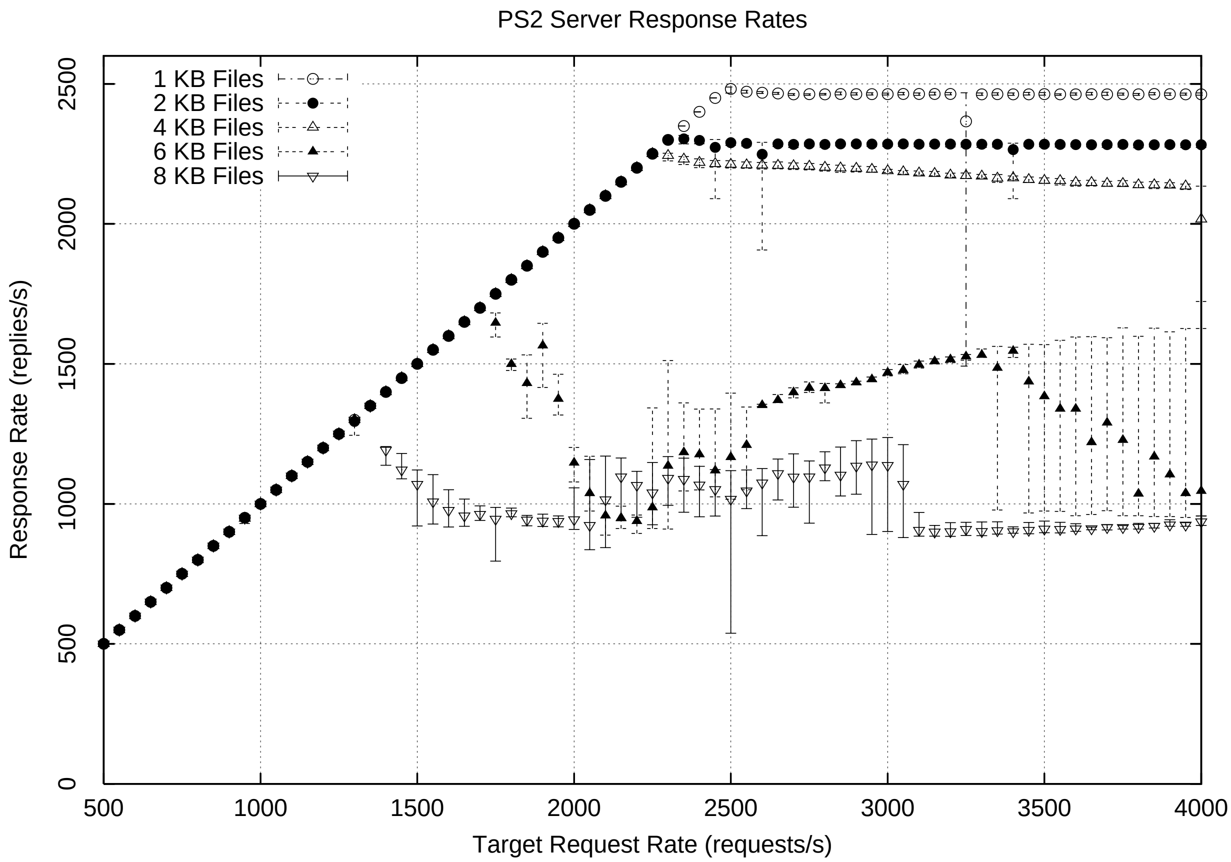
Figures 4.12 and 4.13 demonstrate the response rates attained by both servers (PS1 and PS2 respectively). Section 4.6 presents the argument that it is beneficial to a web-server to perform accept operations right away and this is confirmed by Figures 4.12 and 4.13. These figures show that PS2 maintains a performance profile similar to that of the original SEND-based server, whereas PS1's performance profile more closely resembles that of the `/dev/poll`-based server. Consequently, we believe that the differences in timing of `accept()` operations are the most likely explanation for the differences in behavior between the `/dev/poll`-based and SEND-based servers (depicted in Figures 4.4 and 4.3). However, there are too many differences between these two servers for us to claim this as a fact. (In particular, the `/dev/poll`-based server does not use the auto-accept mechanism, but all SEND-based servers do.)

It is interesting to note that the behaviors of PS2 and our original SEND-based server (Figure 4.3) are essentially identical. These two servers differ only in how they allow notification to occur and the similarity in their behavior suggests that it is characteristic of their accept operation scheduling semantics (i.e., perform accept operation as soon as possible). This is something which we do not think is possible with a strictly polling-based design.

4.8 Limitations of Polling

The issue of proper timing of operations illustrates a fundamental limitation of a polling-based approach to server design: there is no way for the application to know when it should try to accept new connections. A

Figure 4.13 PS2 Server Response Rates

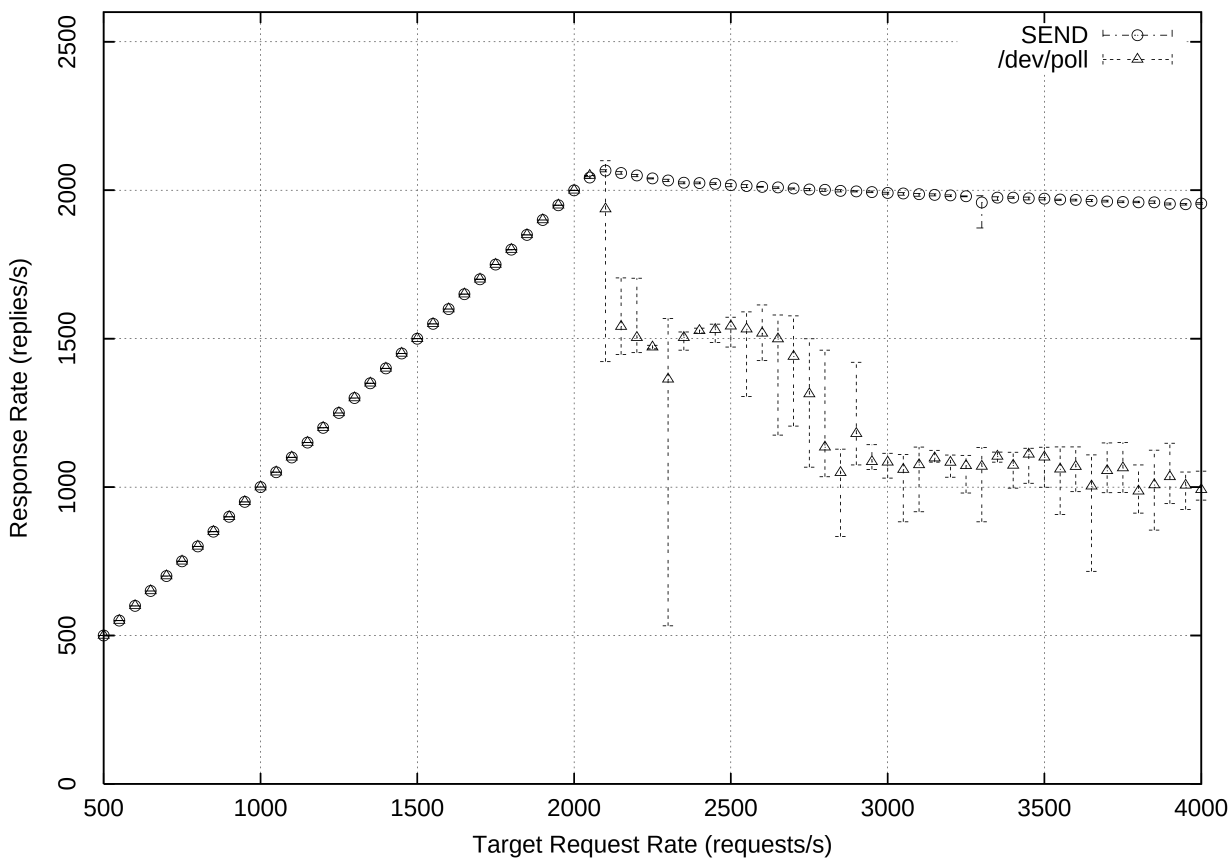


polling server must ensure that there is space in the incoming connections queue to allow for new connection requests to be queued while it performs other work. In order to do this, it must limit how much time is spent handling existing connections before polling to see if it should call `accept()`. Polling too frequently may be inefficient and polling infrequently results in larger batches, thereby reducing server responsiveness to events. Determining how often one should poll (or perform accept operations) cannot be done accurately without knowing in advance when this queue overflows. The fundamental problem is that between polling operations, a polling-based server cannot incorporate new information about its environment into its decision making process.

One may try to improve a polling-based server to more efficiently manage the frequency of polling and accept operations but this is at best only an approximation of the responsiveness attainable by a server using a mechanism such as SEND. Shortly before completion of this thesis, we were made aware of new results by Provos *et al.* [28] which indicating that improving this aspect of a polling-based server is possible. However, we still consider such techniques to be at best an approximation of the effect that can be attained with a purely event-driven design. The improvements proposed by Provos *et al.* [28] call for a web-server to drop connections if the batch size exceeds a pre-determined value in order to prevent the batch size from growing excessively. Though this provides a substantial improvement in response rates when the request rate exceeds the saturation point (and variability thereof), it still does not achieve the over-saturation stability observed in our SEND-based servers.

A server such as our SEND-based server allows the characteristics of the event being considered for delivery and notification to determine the behavior of the application and system in the immediate future. Only by doing so can we make a server truly event-driven. In a polling-based design application behavior in the present and in the immediate future is determined at the time of the last polling operation.

Figure 4.14 Response Rates with Sample Real-World Workload



4.9 Realistic Workload

Our experiments so far have been conducted with workloads based on several different file sizes. However, within each experiment the file size requested on each request is the same. This situation is not representative of the environment in which web-servers typically operate. Therefore, it is important to demonstrate that the behavioral differences among server architectures and multiplexing mechanisms that we have observed can be applied to a more realistic environment.

In order to create a workload more representative of what a real web-server is likely to see, we sampled a handful of commercial web-sites to obtain a collection of files of different sizes. We then configured `httperf` to cycle through this list of files to generate its request sequence. We examined our sample of files and found that the file-size distribution closely resembled the file-size distribution observed by Barford *et al.* [11] in their study of WWW traffic. Therefore, we believe that our sample is reasonably representative of a real-world distribution of requests with respect to file-size.

Figure 4.14 presents the results of these experiments and demonstrates behavior similar to that observed under synthetic workloads in the preceding sections. Consequently, we believe that the results we presented in this chapter can be applied to real-world situations.

Our series of experiments demonstrate that our SEND mechanism is an efficient means of propagating event information into applications. Although it makes little difference in determining the saturation point whether one uses `/dev/poll` or SEND as a basis of a web-server, there are nonetheless significant advantages to using SEND. A web-server based on SEND benefits from the ability to learn of and react to operating system events immediately. This property is unfeasible in a polling-based design and affirms our initial claims that a polling-based application cannot be truly event-driven.

Chapter 5

Conclusions and Future Work

In this chapter we present the conclusions drawn from the work presented in this thesis. In the process of developing the ideas and experiments in this thesis we have identified several avenues for potential research that are outlined in Section 5.2.

5.1 Conclusions

In this thesis we propose and implement a new mechanism (in Linux) for communicating information about events within the operating system to applications in a potentially preemptive manner. Our motivation is to create an efficient framework for truly event-driven applications. We believe existing mechanisms available in Linux (and UNIX in general) are inefficient, require polling semantics and/or provide state and not event information [10, 9, 27]. We believe our mechanism provides a novel approach to the design of interfaces for communicating event information between an operating system and applications and the interactions between the two.

We apply our mechanism to the problem of building a web-server that is better suited to handling high client request rates. In the process we demonstrate that:

- SEND is an efficient means of communicating event data from an OS to an application. We have shown that SEND presents substantially lower overhead costs when compared to the POSIX signals mechanism. These gains can be attributed to the fact that SEND requires substantially fewer kernel entries in the course of the application processing events (since it may deliver multiple events simultaneously and system calls are not required to enable or disable preemptive notification).
- In our experiments with the SEND-based web-server, we demonstrate that a similar saturation point is observed as with the `/dev/poll`-based server. However, the SEND-based server exhibited significantly better behavior when the server is overloaded (i.e., the request rate exceeds the saturation point). This demonstrates that the two mechanisms have comparable overhead costs and thus, the cost of using a generalized mechanism (SEND) as opposed to a specialized one (`/dev/poll`) is not significant.
- The SEND-based server is better suited to handling request rates in excess of the saturation point than web-servers based on a polling mechanism. Our experimental investigation reveals that the reason for this discrepancy is due to the ability of our server to force the system to react immediately to events within it. In particular, our SEND-based server is able to perform accept operations to create new sockets as soon as negotiation of an incoming TCP connection is completed. The differences in performance we observe between polling-based servers and truly event-driven servers demonstrate the potential hidden costs of translating between the semantics of polling mechanisms and those of event-driven application architectures.
- The SEND mechanism can function in several modes; ranging from preemptive event-notification to simulation of event-polling mechanisms. This flexibility demonstrates the robustness and generality of this mechanism and arises from the fact that we no longer depend on the stack and system calls

for communication between the application and kernel. This in turn stems from our recognition and separation of the functionality of event-delivery and event-notification.

In conclusion, we believe that a truly event-driven mechanism is a key component of applications designed to be "event-driven." We have demonstrated that applications (web-servers) can benefit from avoiding kernel-polling semantics in favor of having events drive application and operating system behavior.

5.2 Future Work

This section details possible future directions for this work. The SEND mechanism as described in this thesis, and the corresponding implementation is a first attempt at developing such a mechanism and interface. As we find more applications and place different demands on the mechanism, we expect to find it needs some fine-tuning. The following are some areas that should be given some further consideration:

Further Uses For Preemptive Notification:

We have chosen to demonstrate SEND in the context of a web-server and have thus focused our attention on two event types (`EVT_IPACCEPT` and `EVT_IOREADY`). However, we also see potential for other uses of this mechanism, with other specialized event types. In particular, we think that SEND would be useful for advanced user-space memory management systems; garbage-collection, persistent storage and distributed shared memory. The lower overhead of preemptive notifications presented by SEND may substantially improve the performance of such systems.

The basic prioritization mechanism provided by SEND (via event groups) may be of great value in such systems as it allows for memory-related events to be delivered ahead of other events. This would prevent a memory-related event, that must be processed immediately, from being delayed while the application handles other events of lesser importance.

Research in this area would involve identifying other kinds of operating system and hardware events that applications may be interested in. For example, we may want to consider having the operating system deliver events telling an application which pages have been paged in an out. Such information could be used by user-level threading systems or garbage collectors to adjust execution given the absence of certain pages from core memory. (One could extend this slightly to allow for asynchronous page-faults.)

Kernel/Application Shared Data Structures

In this thesis we demonstrate that it is possible for a kernel to share some data structures with its applications. The ECB is not simply data that is made available for applications to modify, but rather it is a control structure that affects the kernel's behavior. We have found little use of such techniques (shared kernel/application data structures) and consequently believe that such techniques have not been given the serious scrutiny they may deserve. We believe that our reliance on this technique as the basis of SEND demonstrates that a shared-memory based approach to operating system interface design bears further investigation, with the potential to significantly improve the performance, functionality and programmability of operating system interfaces and mechanisms.

One area where such an approach may be used is to provide applications with images (read-only) of their own page-tables, which can be provided with minimal expense. This would allow applications to potentially schedule their activity in a manner that reduces delays due to paging activity. Applications could use such information to advise the operating system of their interest in certain pages of memory in advance if these pages are not in core memory and could avoid touching those pages until they have been paged in. Such a technique would be of interest to writers of garbage collectors, persistent storage systems, distributed shared memory systems and user-level threading packages, which could schedule application activity to avoid blocking the application while it waits for the completion of paging activity.

POSIX-Signals Compatibility Library:

We claim that we could implement the POSIX signals interface on top of SEND. However, until we actually develop such a library we cannot be certain that this can be done with the exact interface as described in Chapter 3; some minor adjustments may be necessary.

Once such a library has been completed, it would be interesting to re-examine the viability of the POSIX signals API. While SEND is designed to address inefficiencies in the POSIX signals binary interface we have not concerned ourselves with providing an easy to use programmer interface. If we have applications that rely on the SEND binary interface, while allowing programmers to develop applications conforming to the POSIX signals API (translating between the two via the compatibility-layer library), we may be able to see most of the performance improvements seen by applications that use the raw SEND API. Such a library may allow applications written using POSIX signals and programmer familiar with POSIX signals (but not SEND) to take advantage of some of SEND's features.

Bibliography

[1] Linux man pages for `sigaction (3)`.

[2] Linux man pages for `sigreturn (3)`.

[3] MIT Exokernel Operating System Project. http://www.pdos.lcs.mit.edu/exo/.

[4] phhttpd home page. http://www.zabbo.net/phhttpd.

[5] Solaris 8 man pages for `poll(7d)`.

[6] Solaris 8 man pages for `sigtimedwait (3r)`.

[7] Signal driven I/O (thread), linux-kernel mailing list, November 1999.

[8] A.W. Appel, J.R. Ellis, and K. Li. Real-Time Concurrent Collection on Stock Multiprocessors. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 11–20, June 1988.

[9] Gaurav Banga and Jeffery C. Mogul. Scalable Kernel Performance for Internet Servers Under Realistic Loads. In *Proceedings of the USENIX 1998 Annual Technical Conference*, 1998.

[10] Gaurav Banga, Jeffrey C. Mogul, and Peter Druschel. A Scalable and Explicit Event Delivery Mechanism for UNIX. In *Proceedings of the USENIX 1999 Annual Technical Conference*, 1999.

[11] P. Barford, A. Bestavros, A. Bradley, and M. E. Crovella. Changes in Web Client Access Patterns: Characteristics and Caching Implications. *World Wide Web, Special Issue on Characterization and Performance Evaluation*, 2:15–28, 1999.

[12] T. Berners-Lee, R. Caillaiu, A. Luotonen, H.F. Nielse, and A. Secret. The World-Wide Web. *Communications ACM*, 37(8):76–82, August 1994.

[13] M. E. Crovella, R. Frangioso, and M. Harchol-Balter. Connection Scheduling in Web Servers. In *Proceedings of the 1999 USENIX Symposium on Internet Technologies and Systems*, Boulder, Colorado, October 1999.

[14] Dawson R. Engler. *The Exokernel Operating System Architecture*. PhD thesis, Massachusetts Institute of Technology, 1998.

[15] Dawson R. Engler, M. Frans Kasshoek, and James O'Toole Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 251–266, Copper Mountain Resort, Colorado, December 1995.

[16] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Request For Comments 2616: Hypertext Transfer Protocol – HTTP 1.1, June 1999.

[17] J. M. Hart. *Win32 System Programming*. Addison Wesly Longman, 1997.

[18] J.C. Hu, I. Pyatali, 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 2nd IEEE Global Internet Conference*, November 1997.

[19] Richard L. Hudson, J. Elliot, B. Moss, S. Subramoney, and W. Washburn. Cycles to Recycle: Garbage Collection on the IA-64. In *International Symposium on Memory Management*, Minneapolis, MN, October 2000.

[20] Intel Corporation. *Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference*, 1999.

[21] Andrew Josey, editor. *Go Solo 2: The Authorized Guide to Version 2 of the Single UNIX Specification*. The Open Group, May 1997. Includes "POSIX Realtime" and "POSIX Threads" by K. Gordon, J. Gwinn, J. Oblinger, and F. Prindle, as Chapters 9 and 10.

[22] Ben Laurie and Peter Laurie. *Apache: The Definitive Guide*. O'Reilly and Associates, February 1999.

[23] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[24] Marshall Kirk McKusick, Keith Bostic, and Michael J. Karels. *The Design and Implementation of the 4.4BSD Operating System*. Addison Wesley Publishing Company, 1996.

[25] D. Mosberger and T. Jin. `httperf` - A Tool for Measuring Web-Server Performance. In *SIGMETRICS Workshop on Internet Server Performance*, June 1998.

[26] Jef Poskanzer. thttpd. http://www.acme.com/software/thttpd.

[27] Nick Provos and Chuck Lever. Scalable Network I/O in Linux. In *Proceedings of USENIX Annual Technical Conference*, June 2000. FREENIX track.

[28] Niels Provos, Chuck Lever, and Stephen Tweedie. Analyzing the Overload Behavior of a Simple Web-Server. Technical report, Center for Information Technology Integration, Universit of Michigan, August 2000.

[29] W. Richard Stevens. *UNIX Network Programming*, volume 1. Prentice Hall PTR, 2nd edition, 1998.

[30] C. Thekkath and H. Levy. Hardware and Software Support for Efficient Exception Handling. In *Proceedings of the 6th International Conference on Architecture Support for Programming Languages*, October 1994.

[31] J. Vert. Writing Scalable Applications for Windows NT. http://www.microsoft.com/win32dev/base/SCALABIL.HTM, 1995.

[32] P.R. Wilson and S.V. Kakkad. Pointer Swizzling at Page Fault Time: Efficiently and Compatibly Supporting Huge Address Spaces on Standard Hardware. In *Proceedings of the 1992 International Workshop on Object Orientation in Operating Systems*, pages 364–377, September 1992.

[33] Robert W. Wisniewski, L. Kontothanassis, and Michael L. Scott. Scheduler Conscious Synchronization. *ACM Transactions on Computer Systems*, February 1997.