Traffic Shaping System Calls Using Threading by Appointment

Christoph M. Kirsch and Harald Röck

Department of Computer Science
University of Salzburg
Austria

Technical Report T009
August 2005
http://cs.uni-salzburg.at/pubs/reports/T009.pdf

Abstract

Threading by Appointment (TAP) is a concurrent programming model that combines automatic stack management (thread-based) with system call queueing (event-driven). However, unlike conventional threads, TAP threads invoke system calls by appointment only, and, unlike events, appointments have a duration, which is determined by the TAP runtime system transparently through a POSIX-compliant interface. The TAP mechanism essentially implements system call queueing using a given TAP policy that consists of a strategy to make appointments, i.e., enqueue system calls, and a logical clock to begin and end appointments, i.e., dequeue system calls. We propose a TAP policy that resembles traffic shaping in network routers, where system calls are treated as network packets. The policy distinguishes system calls for network and disk I/O, and gives priority to system calls invoked by short-running, interactive threads rather than long-running, bulk threads. We have implemented a high-performance, user-space TAP library and benchmarked the library using a multi-threaded, POSIX-compliant web server.

1 Introduction

Threading by appointment (TAP) [9] is an attempt to combine the convenience of automatic stack management (threads) with the efficiency of system call queueing (events). Previous attempts such as Capriccio [15] and State Threads [13] have also used related event-driven mechanisms to improve the performance of an integrated thread and I/O management while maintaining the convenience of thread-based APIs. Similarly, the TAP library offers a conventional thread-based API on top of a high-performance user-space implementation. More importantly, however, TAP generalizes (1) common thread semantics and (2) the notion of events. Unlike conventional threads, TAP threads can only make progress by appointment. In particular, a TAP thread cannot simply invoke a system call but needs an appointment to do so (similar to registering a one-time event handler). Appointments are made by the TAP runtime system or by the TAP threads using TAP-specific API calls. Unlike events, appointments may have a duration to accommodate preemptive I/O and locking mechanisms. In our current implementation, we use the latest nonblocking network and asynchronous disk APIs of the 2.6 Linux kernel. As an example, suppose that a TAP thread T has an appointment to read from disk, which is done by an appropriate TAP wrapper call. When T attempts to invoke the wrapper call, T is blocked and the wrapper call is queued. Until the appointment begins, wrapper calls of other threads with earlier appointments may be executed. When the appointment for T begins, a request for reading asynchronously from disk is submitted to the kernel. The TAP runtime system will eventually collect the read data and make it available to T. Finally, at the end of the appointment, T is released again and can continue to execute. The end of the appointment does not necessarily coincide with the availability of the requested data. In other words, T may not be released immediately after the read data becomes available because, e.g., other wrapper calls should be completed first. We use appointments to control explicitly the order, time, and duration of system calls and, more generally, any communication among threads and between threads and the I/O subsystem. This is the key principle of threading by appointment, which we call *logical timing*.

Appointments are made according to a TAP policy, which consists of an appointment strategy and an appointment clock. The TAP system follows the appointment strategy to enqueue system calls, i.e., to determine queue and position in the queue. The appointment clock dequeues system calls, i.e., determines the time instant when appointments begin and end. We propose a TAP policy that resembles traffic shaping in network routers where system calls are treated as network packets. Traffic shaping controls volume, throughput, and latency of network traffic. Traffic shaping typically involves a queueing discipline similar to an appointment strategy and clock, and a classification scheme. Our policy processes network- and disk-related system calls in separate queues, and classifies system calls invoked by short-running, interactive threads to have a higher priority than system calls invoked by long-running, bulk threads. With an obvious preference for web servers and related applications, a thread is considered more interactive than another if it invokes an accept call on a socket more often. Our policy attempts to improve the latency of interactive threads while maintaining high throughput for all threads.

In Section 2, we introduce the mechanisms that provide the infrastructure for threading by appointment. In Section 3, we describe the notion of TAP policies, in general, and traffic shaping system calls, in particular. Related work is discussed in Section 4. We have implemented a high-performance, user-space TAP library described in Section 5 and benchmarked the library using a multi-threaded, POSIX-compliant web server. See Section 6 for the results. We feel that traffic shaping system calls is an interesting application of threading by appointment but we also believe that there are other interesting aspects of TAP that we discuss in Section 7. We conclude the paper in Section 8.

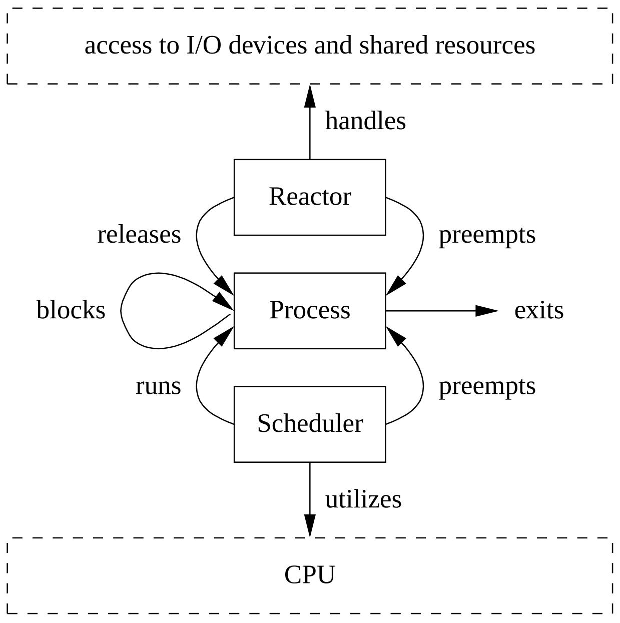


Figure 1: The RSP model

2 Threading by Appointment: Mechanism

The TAP mechanism is based on a simple process model called the *reactor-scheduler-process* (RSP) model [8]. RSP models the control flow in the scheduling core of an operating system, i.e., RSP models task and I/O management but not stack management in the sense of [1]. RSP abstracts from any data-related aspects such as memory management and protection.

2.1 Process Model

An *RSP process* (or process, for short) consists of process and system code. A process can only communicate with other processes or hardware devices through *system code*, which, e.g., performs all process I/O and access to shared resources. *Process code*, on the other hand, implements the actual functionality of a process but has no means to communicate. For example, system calls are clearly system code in the RSP model whereas code that adds two local variables is process code. Figure 1 depicts the process management in the RSP model. The *reactor* handles the *release* of process and system code to the *scheduler*, which handles the *execution* of released process and system code. The reactor controls when transitions from process to system code and vice versa may or may not occur. In other words, during the execution of a process, every transition from process to system code and vice versa involves, at least logically, an invocation of the reactor. Event-driven systems often use a state machine to control the invocation of callbacks. The concept of the reactor is related to such a state machine although the reactor may not be implemented by a state machine.

Figure 2 shows the process states in the RSP model. Suppose that there is a process currently executing process code. In this case, the process is in the green *running* state. There are three possible outcomes: the process attempts to execute system code, which will *always* block the process and thus put it into the red *blocked* state, or the process exits the system, e.g., because an error occurred, or the reactor or scheduler preempted the process and put it into the yellow *released* state. Note that we have not yet implemented the preemption transition. Suppose that the process attempts to execute system code. The process blocks and the reactor is invoked, which can choose to release the system code to the scheduler now or some time later. The reactor or the scheduler need to take care that the system code does not interfere with other already released system code using, e.g., conventional locking techniques. If the reactor decides to wait with the release, it may or may not release other processes, and eventually relinquish control to the scheduler. The purpose of the reactor is to control the *time* when a process *attempts* to begin (and end) the execution of system code. In other words, the reactor, not the process, controls when the process may attempt the execution of system code. Note that the time when the reactor decides to

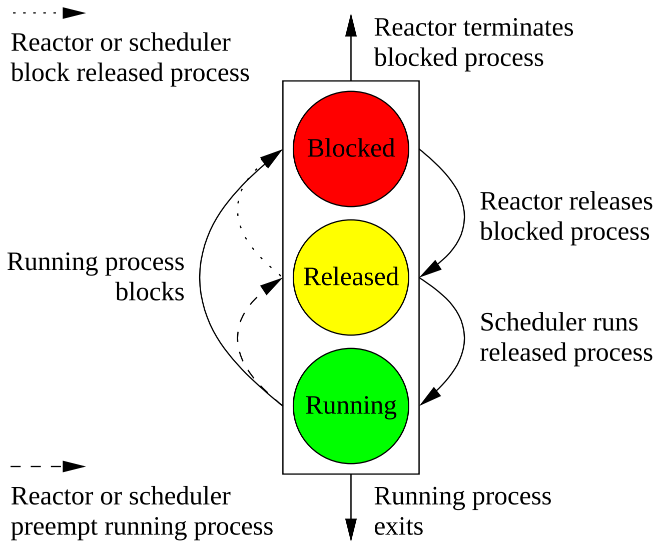

Reactor or scheduler block released process
Reactor terminates blocked process
Blocked
Reactor releases blocked process
Running process blocks
Released
Scheduler runs released process
Running
Reactor or scheduler preempt running process
Running process exits

Figure 2: The process states in the RSP model

release a process can be completely decoupled from the process' progress. For example, a real-time version of the reactor may even decide to release system code of a process that is not yet blocked. This case would correspond to a deadline violation of the process and would require special handling. Suppose that the reactor decides to release the system code of the blocked process now. When the system code eventually completes execution, the process again blocks and the reactor is invoked. The reactor can choose to release the process, i.e., the subsequent process code, now or some time later, or terminate the process, e.g., because an error occurred in the system code. Suppose that the reactor chooses to release the process now and then relinquishes control to the scheduler. In general, there can be multiple processes in the yellow, released state from which the scheduler can choose to run as many processes as there are CPUs. When the scheduler decides to run our example process again, the process has completed a full cycle through all process states in the RSP model. Note that the scheduler and the reactor also have the option to block a released process, i.e., move it into the red blocked state but we have not yet implemented this transition.

The reactor is the key component in the RSP model. The reactor determines the externally relevant, i.e., reactive system behavior by decoupling process scheduling from shared resource management (shared resources other than the CPU). The scheduler in the RSP model merely accommodates the reactor by executing processes proactively as fast as possible. The processes perform computation but rely on the reactor to communicate with anything outside their sphere of control. The reactor therefore effectively determines process composition and system semantics.

2.2 System Call Queueing

TAP is a thread-based instantiation of the RSP model [9]. A *TAP thread* is an RSP process, where system calls are system code and all other thread code is process code. TAP threads can only invoke system calls by appointment. Therefore, in order to access a shared resource such as an I/O device or shared memory, a TAP thread must have an appointment with that resource in advance. Only *during* the time of the appointment, the thread can access the resource, even if the resource is available earlier. If the thread attempts to invoke the according system call before the beginning of the appointment, the thread is blocked (in the sense of the RSP model) until the beginning of the appointment. If, on the other hand, the appointment has already begun or even ended, we speak of a *broken* appointment. If the thread attempted to invoke the system call before the beginning of the appointment but the system call did not return before the end of the appointment, we also speak of a broken appointment. Note that broken appointments are only possible if the notion of time for appointments is decoupled from the threads' progress, e.g., when using a real-time clock for appointments. However, with the TAP policies in this paper, appointments cannot be broken because appointments can only begin and end if the according threads are blocked. Nevertheless, we have introduced the notion of broken appointments here to emphasize the conceptual independence of appointment times and thread progress.

The *TAP reactor* (or ambiguously reactor, for short) makes the appointments and releases TAP threads. The *TAP scheduler* (or ambiguously scheduler, for short) executes released TAP threads cooperatively until they reach the next system call. A preemptive scheduler is future work. Each system call in a TAP thread must be replaced by an appropriate *TAP wrapper call* that invokes the reactor before and after the wrapped system call. Each wrapper call involves the following four steps:

1. (*Process Code to System Code Transition*) The calling thread switches context to the reactor, which first checks if the thread actually has an appointment with this wrapper call. If not, the reactor terminates the thread. Otherwise, the reactor checks if the thread has broken the appointment, i.e., has invoked the wrapper call after the appointment has begun or ended. If yes, the reactor terminates the thread. Otherwise, the reactor puts the thread into the blocked state.

2. (*Begin Appointment*) At the beginning of the appointment, the reactor may release the thread to the scheduler, which will eventually dispatch the thread for execution. Then, the thread will execute system code, i.e., invoke a system call.

3. (*System Code to Process Code Transition*) When the system call returns, the thread switches context to the reactor, which checks if the thread has broken the appointment, i.e., the appointment has ended before the system call returned. If yes, the reactor terminates the thread. Otherwise, the reactor puts the thread into the blocked state.

4. (*End Appointment*) At the end of the appointment, the reactor may release the thread to the scheduler, which will eventually dispatch the thread for execution. Then, the thread will execute process code, i.e., any thread code other than system calls.

In our implementation of the TAP wrapper calls for network and disk I/O, step 2 is actually done by the TAP I/O subsystem instead of the threads and the scheduler, i.e., the necessary system calls are invoked by the I/O subsystem on behalf of the threads. Nevertheless, we prefer to look at the mechanism first from a more logical point of view such as the above, and only then consider the details and optimizations. Note that aggressive optimizations are possible if tailored to a given TAP policy. For example, a thread may invoke a wrapper call exactly at the time of its appointment, which may happen if appointments are made according to a policy that considers the threads' progress. In this case, the reactor does not have to block the thread. See Section 5 for more details.

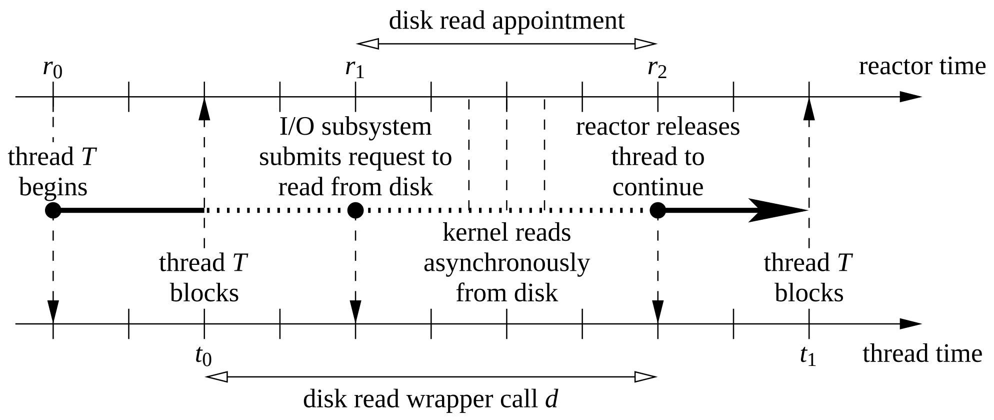
TAP effectively performs *system call queueing*. The TAP reactor controls the order and time of system calls using a given TAP policy. However, before we introduce the concept of TAP policies in more detail, we describe the TAP I/O subsystem as part of the TAP mechanism.

2.3 The TAP I/O Subsystem

The TAP library uses nonblocking network and asynchronous disk calls of the Linux 2.6 kernel to implement I/O. The TAP I/O subsystem of the library interfaces the TAP reactor and scheduler to the kernel. The I/O subsystem is technically the most involved part of the library. In the following two sections, we explain the principles rather than the details of disk and network I/O in the TAP library. Technical details can be found in Section 5.

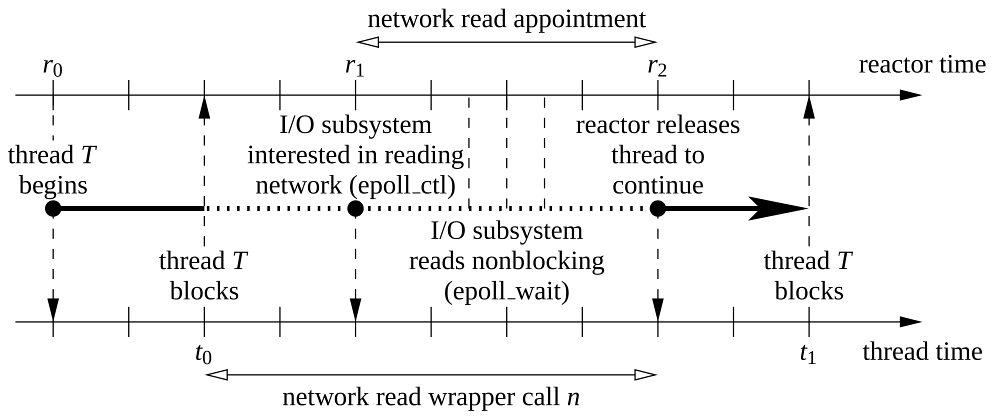
2.3.1 Disk

There are TAP wrapper calls for disk read, write, open, and close operations. All calls require appointments. As an example, Figure 3 shows a TAP thread T invoking the TAP wrapper call d to read from disk. There are two timelines: the reactor determines when time advances on the reactor timeline while the thread (and, in fact, the scheduler) determines when time advances on the thread timeline. For example, the time instant r_0 on the reactor timeline is under control of the reactor because thread creation also requires an appointment. Thus there is also a TAP wrapper call for thread creation. The time instant t_0 on the scheduler timeline is under control of T because the time when T runs into d and therefore blocks depends on T. The two timelines are an important feature of the RSP model and thus the TAP system. The time of events on the reactor timeline are under exclusive control of the reactor. The reactor timeline is a system-level timeline in the sense that it incorporates the system's progress, i.e., the progress of *all* threads, and not of just one thread.



disk read appointment
r_0
r_1
r_2
reactor time
I/O subsystem submits request to read from disk
reactor releases thread to continue
thread T begins
kernel reads asynchronously from disk
thread T blocks
thread T blocks
t_0
t_1
thread time
disk read wrapper call d

Figure 3: The execution of the TAP wrapper call d to read from disk



network read appointment
r_0
r_1
r_2
reactor time
I/O subsystem interested in reading network (epoll_ctl)
reactor releases thread to continue
thread T begins
I/O subsystem reads nonblocking (epoll_wait)
thread T blocks
thread T blocks
t_0
t_1
thread time
network read wrapper call n

Figure 4: The execution of the TAP wrapper call n to read from network

For now, let us assume that an appointment with d from r_1 to r_2 has been made for T at r_0 after T has been created but before T begins executing. Note that r_1 and r_2 are logical time instants whose physical time may or may not be known at r_0. When T invokes d at t_0, the reactor puts T in the blocked state. The system may now perform other work such as executing other threads or I/O. At r_1, the I/O subsystem only submits a request to read asynchronously from disk to the Linux kernel but the reactor does not release T. In other words, the I/O subsystem invokes the submit system call on behalf of T and thus bypasses the TAP scheduler for efficiency reasons. At r_2, the I/O subsystem checks if the disk read has actually completed. If not, we say that the appointment with T has been *broken* by the resource, i.e., the disk, and T is terminated. Similar to appointments broken by threads, appointments cannot be broken by resources with the TAP policies of this paper. In other words, the appointment is guaranteed to end after the disk read is complete. The wrapper call d is complete at r_2 and the reactor releases T to the scheduler.

The TAP wrapper calls for disk I/O essentially provide preemptive disk access and hide the complex API of asynchronous disk I/O from the user of the TAP library. However, we do not intend to argue in favor of or against asynchronous disk I/O but see the presented approach more like an example of how to map TAP to existing I/O mechanisms. For network I/O, we had to use another approach since asynchronous network I/O is not available in the Linux 2.6 kernel.

2.3.2 Network

There are TAP wrapper calls for socket accept, read, write, and close operations. All calls require appointments. As an example, Figure 4 shows a TAP thread T invoking the TAP wrapper call n to read from a socket. Until the time instant r_1, the system behaves just like in the case of the TAP wrapper call that reads from disk. At r_1, the

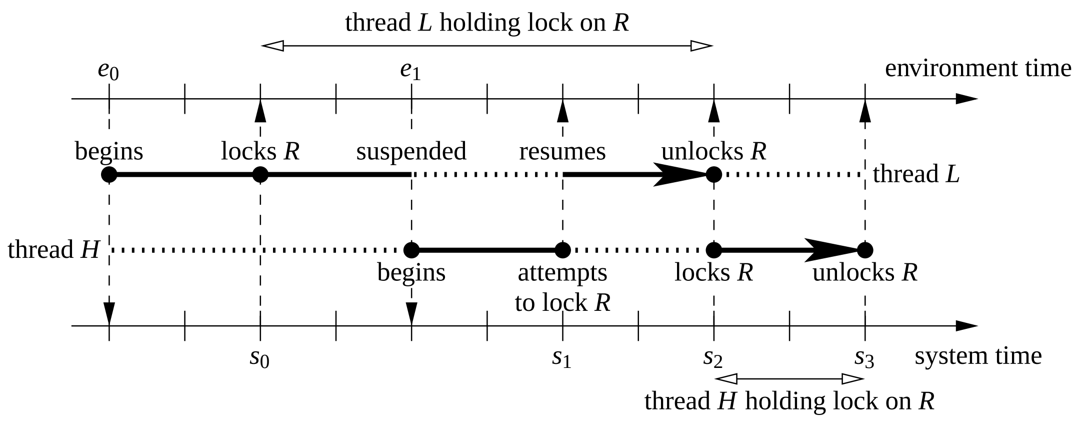


Figure 5: Two conventional threads L and H accessing a shared resource R

I/O subsystem invokes a nonblocking system call (epoll_ctl) on behalf of T that expresses the thread's interest in reading from the given socket. Similar to the disk read, the reactor does not release T now. Unlike the disk read, however, the I/O subsystem will then try to read the socket itself since there is no mechanism for asynchronous network access in the Linux 2.6 kernel. During the time of the appointment and whenever no thread executes, the I/O subsystem checks if it can read from the socket without blocking (epoll_wait). If yes, the I/O subsystem reads from the socket on behalf of T. Otherwise, it will try again later. In other words, the TAP I/O subsystem effectively implements an asynchronous mechanism to access the network. At r_2, the reactor checks if the network read has actually completed, i.e., if the appointment with T has been broken by the network. Again, however, appointments cannot be broken by resources with the TAP policies of this paper. The wrapper call n is complete at r_2 and the reactor releases T to the scheduler.

The TAP wrapper call for accept on a socket requires special care since the time of an incoming connection is difficult to predict. For consistency and thus generic handling, TAP threads need an appointment to invoke the wrapper call for accept. However, as soon as a thread invokes the wrapper call, the thread is blocked but the appointment is discarded. Now, the TAP I/O subsystem checks on behalf of the thread for incoming connections. If there is an incoming connection, the I/O subsystem informs the reactor to make a so-called *instantaneous* appointment for the thread, i.e., a nonpreemptive, logical zero time appointment. At the time of the appointment, the I/O subsystem invokes the actual accept system call on behalf of the thread and the reactor thereafter releases the thread to the scheduler.

Similar to disk I/O, the TAP wrapper calls for network I/O essentially provide preemptive network access and hide the complex API of nonblocking network I/O from the user of the TAP library. The next section introduces a mechanism for locking shared memory in the TAP system.

2.4 TAP Locking

The TAP library in its current implementation uses a cooperative scheduler that dispatches threads to run until completion, i.e., to run without interruption until the next TAP wrapper call is reached. A TAP thread is therefore guaranteed to have exclusive access to shared memory between any two wrapper calls but not across a wrapper call. We have implemented a conventional locking mechanism to overcome this limitation and to be able to run POSIX-compliant web-servers on the TAP library. However, TAP actually gives rise to interesting alternative forms of locking, which we have not yet implemented but feel worth mentioning here in order to provide more insight into TAP.

Figure 5 shows the single-processor execution of two conventional threads L and H accessing a shared resource R using a lock on R. Upon the occurrence of some event in the environment of the system, thread L begins executing at e_0. After some time, L successfully locks R at s_0. Similar to reactor and thread time, we distinguish environment and system time here in order to emphasize who controls the time of an event. At e_1 an environment event occurs and triggers the thread H to begin executing. The scheduler chooses H over L because, for example, H has a higher priority than L. At s_1, H attempts to lock R but fails since L still holds the lock. As a consequence, L resumes its execution while H waits for L to release the lock. This situation is known as priority inversion,

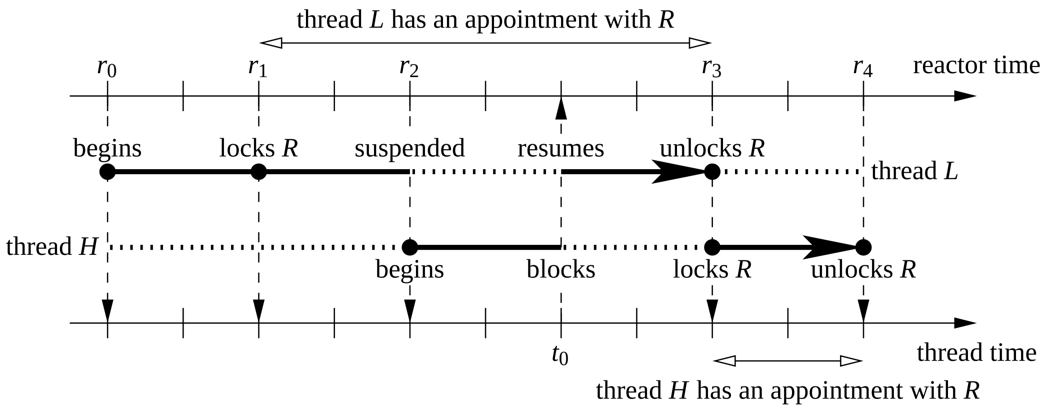


Figure 6: Two TAP threads L and H accessing a shared resource R

which may lead to starvation of L and H if there is another thread M with a priority higher than L's priority. In this case, M may get access to the CPU indefinitely even though H may have a higher priority than M's priority (M stands for medium priority). Note that M may not even have an interest in R at all. In a real-time system, H may then miss a deadline. Priority inversion is traditionally addressed by making the scheduler smarter using techniques such as priority inheritance or priority ceiling protocols. The idea is to increase the priority of L to at least H's priority temporarily from s_1 until s_2. Then, L gets the CPU, even in the presence of a thread like M, and may execute until it unlocks R at s_2. Immediately thereafter, H is released and scheduled to run. This time H successfully attempts to lock R and continues to run until it unlocks R at s_3.

The disadvantage of priority inheritance and similar techniques is the increased complexity and nondeterminism of the scheduler and therefore the overall system. The root of the problem, however, is that these techniques only offer a symptomatic solution based on priorities. Preemptive yet atomic access to shared resources and priorities are incompatible concepts since the notion of preemptive yet atomic access is a stateful concept whereas priorities are not. Locking based on TAP solves the problem by using appointments, which are stateful and therefore semantically richer than priorities. Note that TAP locking has advantages, even in the absence of real-time constraints, because it allows the TAP system to balance I/O activity and access to shared memory as a whole.

Figure 6 shows the single-processor execution of the two threads L and H using TAP locking. There are two important differences to the conventional locking shown in Figure 5: firstly, all time instants except t_0 are now under the reactor's control, and secondly, H does not even get to attempt to lock R at t_0 but is blocked by the reactor since an appointment for H with R has already been made to occur after the appointment for L with R is finished. The relative urgency of the threads is expressed by appointments rather than priorities avoiding priority inversion altogether. It is now up to the TAP policy to make appointments appropriately. For example, in the presence of real-time constraints, the appointment for L could be made short to express the fact that L is *less* important than H and therefore gets only short access to R. Then, a simple scheduling strategy such as earliest deadline first gives priority to L but only while L has access to R because the end of L's appointment imposes a deadline on L that is earlier than the beginning of H's appointment, which is the next deadline for H. This example demonstrates that appointments model urgency requirements more accurately than priorities, in particular, in the presence of shared resources.

The resulting observable behavior in the previous example is essentially the same as with conventional locking and, e.g., priority inheritance, but it is achieved with quite different means. TAP locking shifts complexity from the scheduler (priorities) to the reactor (appointments). TAP locking looks into the *future* to handle access to shared resources rather than conventional locking, which looks into the *past*. As a consequence, an important aspect of TAP are the policies that determine how appointments are made. The following section introduces TAP policies in more detail.

3 Threading by Appointment: Policy

A *TAP policy* consists of an appointment strategy and an appointment clock. The TAP reactor follows the *appointment strategy* to enqueue system calls, i.e., to determine the queue as well as the position in the queue. For example, at the end of an appointment for a TAP thread before the thread is released to the scheduler, the reactor can make an appointment for the next wrapper call. We call such a policy *continuous* because it guarantees that a TAP thread always has an appointment. The current version of the TAP mechanism only supports continuous policies and, in fact, allows a TAP thread to have at most one appointment. Thus a TAP thread always has exactly one appointment at any time in our current implementation. More than one appointment per thread is future work.

The *appointment clock* dequeues system calls, i.e., actually determines the time instant when an appointment begins and ends. The TAP mechanism currently supports non-real-time clocks that begin appointments as soon as the *next-appointed* threads become blocked, and end appointments as soon as the according I/O operations have completed. Therefore, appointments cannot be broken in the current implementation. Note that, similar to conventional threading libraries, such a system executes as fast as possible and never idles unless there is nothing to do. In other words, the current implementation may reorder system calls arbitrarily but does not otherwise delay them. However, we are also interested in *idling* appointment clocks, which may begin or end an appointment later than its *inflection point*, i.e., the earliest time when the appointment cannot be broken anymore. Idling appointment clocks may limit system call throughput similar to bandwidth throttling in traffic shaping routers.

3.1 Multiple System Call Queues

Given a continuous TAP policy, suppose a TAP thread has reached the end of its current appointment and the reactor needs to make a new appointment for the thread. In general, however, the reactor does not know which wrapper call will be invoked next by the thread. There are at least four options: (1) the thread (i.e., the programmer) tells the reactor which wrapper call it *intends* to invoke, (2) the thread's compiler *analyzes* the thread statically and tells the reactor which wrapper call the thread may invoke, (3) the reactor *monitors* the thread at runtime and caches its call behavior, or (4) the reactor makes an appointment for an *unknown* wrapper call. Option (1) may result in highest runtime performance but is less convenient than the other options and requires a non-POSIX API. Options (2) and (3) may not be able to predict the next wrapper call because of branching but are more convenient than option (1). The compiler of the Capriccio project [15] relates closely to option (2). The compiler computes a control-flow graph for a given program whose nodes represent potentially blocking calls invoked by the program. Option (4) may result in the lowest runtime performance but is convenient, very simple to implement, and avoids a non-POSIX API. So far, we have only implemented option (4) using a separate queue for unknown wrapper calls. We call an appointment for an unknown wrapper call a *commit appointment* because at the time of the appointment the thread is supposed to commit to invoking a known wrapper call for which a regular appointment can be made. Commit appointments are always instantaneous, i.e., take logical zero time.

Let us go back to the TAP thread that has reached the end of its current appointment. In our TAP implementation, the reactor now makes a commit appointment for the thread and then releases the thread to the scheduler. When the thread reaches the next wrapper call, the reactor is invoked and makes an appointment for that wrapper call. There are two queues, one for network calls and one for disk calls, in order to support policies that distinguish such calls. In general, the new appointment may begin now or some time later. If it begins now the system was actually not able to exploit any information to improve performance until the appointment begins because the wrapper call was unknown up to now. Nevertheless, we have only implemented this restricted form because of simplicity and because the system can still freely determine when the appointment will end. In sum, there are three queues in our current implementation: DISK, NET, and CPU, where CPU contains the unknown wrapper calls with commit appointments. Subsequently, we may also refer to DISK, NET, and CPU as *calendars*.

3.2 Traffic Shaping System Calls

Traffic shaping is a technique used in network routers to control volume, throughput, and latency of network traffic. Since Version 2.2 the Linux kernel also features traffic shaping of network packets. Traffic shaping typically involves a *queueing discipline* and a *classification scheme*. The leaky-bucket and token bucket algorithms are popular queueing disciplines. For example, the leaky-bucket algorithm shapes bursty traffic into a steady stream of traffic. Conceptually, arriving packets are placed (enqueued) in a bucket of a given size with a hole in the bottom. Packets drain (are dequeued) through the hole into the network at a given rate. Packets are discarded if the bucket is full. The token bucket algorithm is similar to the leaky-bucket algorithm but allows bursty traffic to continue transmitting while there are tokens in the bucket, see, e.g., [14] for more details. Traffic shaping often uses, in addition to a queueing discipline, a classification scheme to analyze, classify, and then enqueue incoming packets resulting in non-FIFO queueing. For example, interactive traffic such as telnet and ssh connections may be enqueued in front of any bulk traffic such as long-running ftp downloads.

We now introduce a TAP policy for traffic shaping system calls. The difference to traditional traffic shaping including the existing Linux implementations is that we shape the flow of system calls and not of network packets. Traffic shaping system calls is on a logically higher level, closer to the application than traffic shaping network packets. Therefore, the TAP system can potentially obtain a more comprehensive picture of the requirements on and the utilization of network, disk, and memory (with TAP locking). For example, a single incoming network packet may trigger multiple disk accesses by the thread that processes the packet. Such a correlation of network and disk access through a thread is easily observed by the TAP system but is harder to establish on a lower level such as the I/O subsystem. However, it is difficult to obtain experimental evidence that traffic shaping system calls rather than network packets performs better because it is not obvious how to relate the involved configuration parameters in a fair way. The design of an appropriate benchmark is part of our ongoing work.

We propose the following appointment strategy for traffic shaping system calls. At the end of an appointment, a new commit appointment is made in the CPU calendar that will begin after all existing CPU appointments have ended. When the commit appointment begins, a new appointment is made in either the DISK or NET calendar depending on whether the current wrapper call is a disk or network call. This appointment begins immediately but may end after other appointments in the respective calendar have ended. The appointment clock dequeues appointments from the calendars in round-robin fashion with a given ratio. For example, a cycle begins with five CPU appointments dequeued and processed at the first time instant, followed by four DISK appointments at the next instant, and completed by one NET appointment at the third instant. A time instant occurs as soon as all next-appointed threads are blocked and the according I/O operations have completed. The current implementation does not, however, include anything related to bandwidth throttling, e.g., the leaky-bucket algorithm for system calls. This is future work.

Appointment strategy and appointment clock together correspond to a queueing discipline in the context of traffic shaping network packets. However, an important difference is that appointments may have a duration to address the fact that executing system calls, unlike processing network packets, may take considerable time and even require preemptive scheduling. This is also the reason why we chose the notion of an appointment clock rather than something implicit or unrelated to time. System calls and system code, in general, are inherently more complex objects than network packets. The notion of appointments is an attempt to capture the temporal aspects of that complexity.

The TAP policy for traffic shaping system calls also includes the following classification scheme that is particularly suitable for multi-threaded web servers. The scheme gives priority to system calls invoked by short-running, supposedly interactive threads. Each thread has a priority represented by an integer value. An accept wrapper call on a socket resets the thread's priority to the highest priority. Any other wrapper call decreases the thread's priority by one until the lowest priority is reached. The commit appointments for unknown wrapper calls are enqueued according to the priority of the invoking threads. In other words, the longer a thread runs, i.e., does not invoke an accept wrapper call on a socket, the lower its priority for getting appointments, not CPU time, becomes. This scheme is a rather simple approximation of more general monitoring techniques that might be possible to analyze and cache patterns of system call invocations. However, the scheme turned out to be quite successful on some web server workloads. More involved thread monitoring is future work.

4 Related Work

We relate threading by appointment to existing thread-based and event-driven concurrency models, and to concurrency mechanisms that incorporate notions of planned futures. Threading by appointment generalizes the notion of logical execution time (LET) programming first proposed in the real-time language Giotto [7]. The LET of a task is given by the programmer and determines for how long the task executes in real time to produce its result, independently of any platform characteristics such as CPU utilization and performance. An LET program can statically and dyncamically be checked if its tasks do in fact execute within their LETs for a given platform, which is typically characterized by a scheduling strategy and worst-case execution times for all tasks. If a task needs less time than its LET, its output is buffered and only made available exactly when its LET has elapsed. TAP can be seen as a more flexible form of the LET model suitable for thread-based concurrent programming.

The key difference between thread-based and event-driven models is that thread-based models provide automatic stack management and typically preemptive task management whereas event-driven models require manual stack management and usually provide cooperative task management [1]. In the web server domain, event-driven systems such as the Zeus web server are known to have superior performance on cached workloads, i.e., if disk access is limited, whereas thread-based systems such as the Apache web server perform better on disk-bound workloads [5]. In terms of the RSP model, event-driven systems typically implement just the reactor by an efficient event and I/O multiplexer. A scheduler is usually not required since the multiplexer executes the callbacks (tasks) itself non-preemptively. The implementation of thread-based systems, on the other hand, typically requires a more complex reactor that handles I/O multiplexing and thread synchronization, and a non-trivial preemptive scheduler. We observe two recent trends that share the same goal of providing superior performance on cached and disk-bound workloads but approach it from opposite directions: (1) thread-based models are implemented by simpler event-driven reactors and cooperative schedulers [13] using asynchronous I/O for disk access [15], while (2) event-driven models are enhanced by delegating disk access to thread-based models such as helper threads [5] or thread pools [17], which effectively resemble the techniques that implement asynchronous I/O. The first approach has the apparent advantage of maintaining the illusion of sequential control flow in threads through automatic stack management, which, however, vanishes with the use of asynchronous I/O techniques on the application level. The control flow in applications running on top of event-driven systems is already known to be hard to comprehend, which, in the presence of thread-based enhancements, gets even more difficult. Both trends, while successful at improving performance, suffer from increased complexity on the application level. Threading by appointment is an attempt to address this problem. The TAP reactor is a mediator between system and threads using appointments and calendars, which allows us to optimize system performance while maintaining a given programming abstraction. For example, TAP can completely hide nonblocking and asynchronous I/O as well as traffic shaping of system calls under a POSIX-compliant interface. A general question in this context is how to design other, in particular, TAP-specific, concurrent programming interfaces that allow for even more systems-level optimizations without changing the original interfaces' semantics.

The notion of appointments also relates to so-called futures first introduced in MultiLisp [12]. A future is a placeholder for an expression that is to be evaluated concurrently to the current thread of execution. An appointment is an agreement between the current thread and the runtime system to meet at a future time to execute code accessing shared resources other than the CPU. Appointments can therefore be seen as a means to implement futures. Other work inspired by futures such as batched futures used in databases to reduce crossdomain call overhead [3], and safe futures used in Java to designate opportunities for concurrency [16], are related to appointments in the same rather abstract sense.

5 Implementation

The implementation consists of three parts: a C library, a posix interface and a web-server on top of the library. We describe only the main parts of the implementation and leave the technical details, e.g. bootstrapping, context switching, and the mapping of the system data-structures, to the source code documentation.

5.1 Library

The library implements our cooperative threading model "Threading by Appointment"[9]. It is divided in three layers: a small layer for context switching, a threading layer with the reactor, and an interface which corresponds to `pthread`. The context switching layer is based on `makecontext/setcontext/swapcontext` of the glibc, but with our own implementation in assembler. As a compiler option, however, we can fall back to the standard implementation. The threading layer is a simple cooperative user-space threading system with a manager thread, the reactor. The reactor, is the main part of the library. When a thread yields, the reactor takes over and decides what is done next. It executes the system-calls on behalf of the threads and observes the completion of events of the threads and the underlying operating system, in our case Linux. This layer is separated in several modules. We use an I/O module which combines asynchronous disc and nonblocking network access into one interface. A thread module which handles thread creation, deletion and stack management. Additionally we need an appointment module which calculates the threads for the next appointments and inserts threads into the calendar. The scheduler, on the other hand, dispatches the threads of the run queue. We don't have any preemption in the library yet; the threads yield at wrapper calls. The library interface is not `pthread` compliant, but we provide an additional posix interface which overwrites the system-call stubs of `read`, `write`, `open`, `close`, `accept`, `socket`, `bind`, `listen`, and most of the `pthread_*` functions. These wrappers invoke the TAP calls.

Listing 1: The reactor thread

```
... /* init data structures */
while(active threads){
    if(current_thread is not blocked)
        release current_thread;

    delete_zombies();

    while(run_queue is empty){
        if(appointment_ready()){
            appointment_clock()
            if(run_queue not empty)
                break;
        }

        threads = poll();
        forall(thread in threads){
            if(thread on next appointment)
                move thread from next to ready queue;
        }
    }
    scheduler();
}
```

5.1.1 The Reactor Thread

The reactor is the main part of the implementation. It operates on several lists or queues of threads. There is a run-queue which holds the released threads scheduled for execution, a zombie queue with threads to delete, and an IO queue of threads with finished IO operations. Additionally we need queues to determine the appointment clocks: a list of threads which are ready and on the next appointment(`ready_queue`) and a list for threads which are on the next appointment but not yet ready(`next_queue`). An appointment is ready when the `next_queue` is empty and the `ready_queue` is not empty, hence we check for this condition before we start polling for finished IO.

The reactor thread, subsequently called reactor, is outlined in Listing 1. Note that this is not all of the reactor in the sense of section 2, it is the part of the reactor which runs when a thread yields. The reactor will always continue where the scheduler left of, thus at the end of the while loop. As long as there are released threads we want to execute them, hence we start the scheduler immediately. If all threads are blocked we check upcoming appointments and poll for finished IO. Threads are released according to the current calendar strategy and at an appointment clock, hence we check the run-queue again before we poll for finished IO. The `poll()` is a call to the IO subsystem and it returns a list of threads which are blocked but the IO they are waiting for is finished and they could potentially be released. Note that the system-calls are already done by the IO module. Although some threads could continue we don't release them immediately. We release threads only at and appointment clock, which depends not only on the completion of the IO of one thread but also on the progress of other threads.

A calendar strategy consists of two elements: one is responsible for inserting threads into the calendar, i.e. making appointments, the other one calculates the threads for the next appointment. The latter is called at an appointment clock. A thread is ready for the next appointment if the IO the thread is waiting for is finished or it run into a wrapper call for which the thread has an appointment. Otherwise the thread is not ready, and the next appointment clock is delayed until all threads for the next appointment are ready. On an appointment a thread is released to the scheduler or the IO operation is submitted. The latter is at the beginning of a wrapper call and the release marks the end of a wrapper call. Calendars are implemented as priority lists with discrete priority levels. Each level consists of a separate list and a lower level represents a higher priority, i.e. we start looking for threads at level 0.

5.1.2 IO Module

The IO module abstracts the different interface of non-blocking IO(NIO) and asynchronous IO(AIO). It implements a simpler `submit` and `poll` interface. Logically `poll` is part of the reactor and `submit` is part of the wrapper

Listing 2: AIO Polling

```
events = getevents();
forall(event in events){
    thread = parse_event(event);
    if(thread is ok)
        insert thread in io_queue;
}
```

calls but we put them in one module, which handles all IO related calls. For network IO we use the new `epoll` interface. The submission is done with `epoll_ctl` and polling with `epoll_wait`. If `epoll_wait` returns a ready file-descriptor we execute the correct non-blocking system-call immediately. On the other hand AIO is done with `io_submit` to submit one or more disc requests(only `read` and `write` is supported) and `io_getevents` to get the finished IO operations. The difference between NIO and AIO is, with NIO we can only check if a future IO operation on a specified file-descriptor will potentially block, whereas with AIO we submit the IO request to the OS and later check its completion. Thus, for NIO we still have to execute the read/write system call, whereas AIO does it in background and we ask for completed operations. An additional feature of AIO is the possibility to submit several requests with one call, i.e. to batch disc access. We batch disc requests and submit all AIO operations of an appointment with one system-call. There is no comparable feature for epoll, however in [6] `epoll_ctlv` is introduced and evaluated. This new call would combine several `epoll_ctl` calls to one system-call, but it is not supported by and hence we do not use it. To support batched AIO submissions we do not execute the submits immediately. On an appointment clock we extract the information for the system-call of the threads which haven and appointment. With this data we fill the necessary data-structures(e.g. the `struct iocb`) and put them in an array from where we submit the batched requests at the end of an appointment clock.

A limitation of AIO on Linux is that it works only on files opened with `O_DIRECT`. Therefore we bypass the disc cache and access the disc directly. Furthermore we can only read or write multiplies of 512 bytes and the user-space buffers for these operations must be aligned to 512 byte boundaries. However, with this configuration we don't test or benchmark the caching of the system, but rather the performance of invoking blocking system-calls.

A known problem is the incompatibility of AIO notification and epoll. If both are used within one process it is not possible to wait for a finished AIO operation and ready socket-descriptor with one call. More precisely, Linux provides two system calls `io_getevents` for finished AIO notification and `epoll_wait` for socket ready notification. Both calls provide the possibility to wait for at least a certain amount of events and to block the process as long as none occurs. However, it is not possible to wait for both, hence we have to call `io_getevents` and `epoll_wait` in a loop until one of them returns at an event. Therefore our implementation needs a lot of CPU when no events are ready. A small improvement is to poll for finished AIO only if something is pending. Another possibility would be to use separate kernel threads for event polling. This, however, would introduce the need for conventional threading and synchronization techniques which we want to avoid at this state of development. We assume we will have to implement it for performance reasons in the near future, because our latest tests show that a variable disc load with a big set of different files increases the time between submission of IO and notification, and hence we spend much time in polling and need too many CPU cycles. In comparison to a kernel thread implementation, e.g. NPTL, which does not need any CPU because all threads are basically blocked most of the time. Performance evaluation and improvements are future work.

The IO subsystem provides the poll call which checks for finished IO operations. This call does not block and returns immediately a list of threads. We have to poll first for NIO and concatenate the threads returned with the list of threads return from AIO poll. Both calls are logically equivalent in structure, hence we describe only AIO polling. The `parse_event` is different for AIO and NIO not only in the used data-structures but for NIO we have to do the additional system-call.

5.1.3 Wrapper Calls

The wrapper calls are the interface and entry point from the application to the TAP system. The logical description can be found in section 2. The actual implementation uses a slightly different approach. We try to minimize the number of context switches and the number of times a thread runs through the scheduler. Therefore, we store all necessary information for the system call in the thread context at the beginning of a wrapper call. The thread is blocked right after the information is saved and the reactor and the IO subsystem do the next steps. The thread is

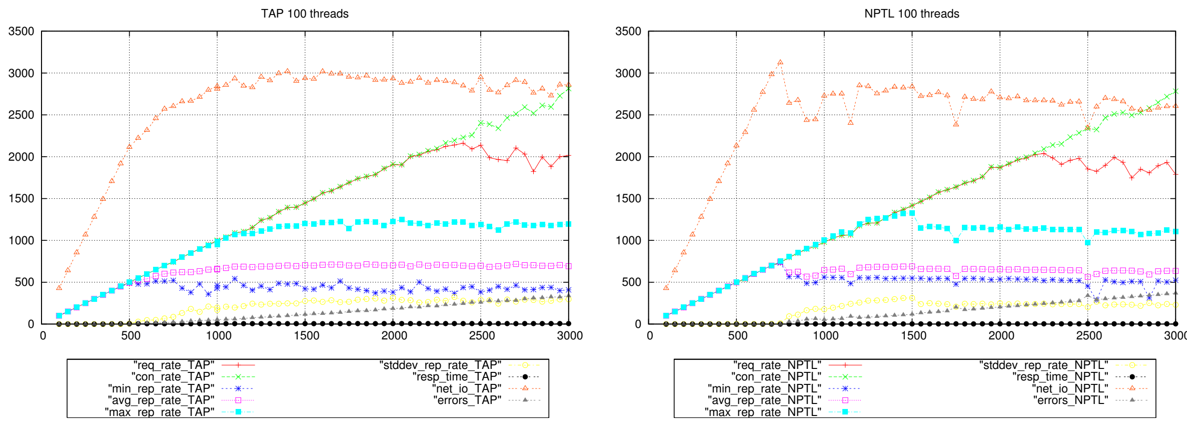


Figure 7: TAP and NPTL

released when all operations are done and the end of the appointment comes up. This eliminates context switches at appointment clocks.

5.2 Web-server

We use a pthread compliant web-server to test and benchmark our implementation. It is a small server without any caching. The server is linked against our library and against the new Native POSIX Thread Library(NPTL) for Linux [4]. Our web-server implements only a subset of HTTP version 1.0; just the basic GET method is supported. The web-server is intended to show the capabilities of our approach, hence it is a very simple implementation without any fancy optimizations. The main thread creates a listen socket and starts several threads, which handle all incoming connections.

6 Experiments

6.1 Configuration

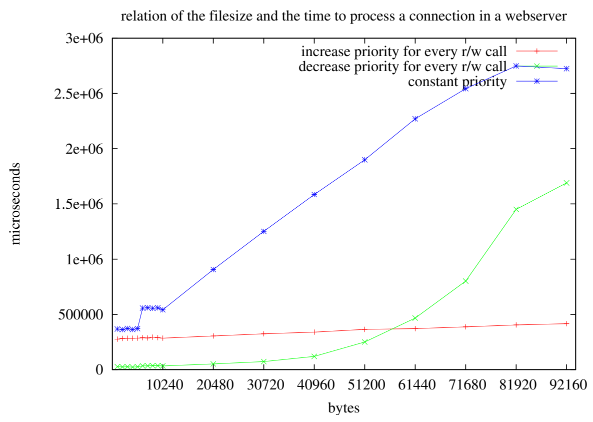
Two identical machines are used for the tests and benchmarks. These computers are dual Intel Xeon CPUs with 3GHz each and 2GB RAM, running Linux kernel 2.6.8 connected via Gbit Ethernet. Additionally we use a laptop with a 1.4 GHz Intel Pentium M and 512MB RAM as a second client.

The clients and server should run in sync and packet queuing at the server should be reduced as much as possible. The measurements of the benchmark should reflect the behavior of the web-server application and not the performance and behavior of the kernels TCP/IP implementation. The server kernel is configured to reset connections if the listening service is too slow. This should reduce queuing in the kernel, and result in better and more comparable benchmarking values of the application. The server uses only one CPU by deactivating SMP support in the kernel. The benchmarking tool on the client is `httperf`[11], distributed among the clients with `autobench`[10]. We use httperf to generate load on the server and to evaluate the behavior of the library under full load. It is used with the option `close-with-reset`, which forces the connections to be reset when closed. With this option the number of necessary TCP ports on the clients is reduced, because the TIME_WAIT state of the TCP protocol is omitted and port numbers can be reused sooner. Another option is `hog` which deactivates the port range limitation to the ephemeral ports and allows httperf to use as many ports as possible. This option is necessary for our configuration, otherwise we would run out of sockets and the client would be overloaded instead of the server.

6.2 Results

6.2.1 TAP and NPTL

Figure 7 compares our small non-caching web-server linked against TAP and NPTL. The *x* axis represents the number of connections per second. We retrieve always the same file of size 4244 bytes, and we read and send packets of size 512 bytes. Hence we need 10 read calls from disc(9 plus one to determine `EOF`) and 10 write calls



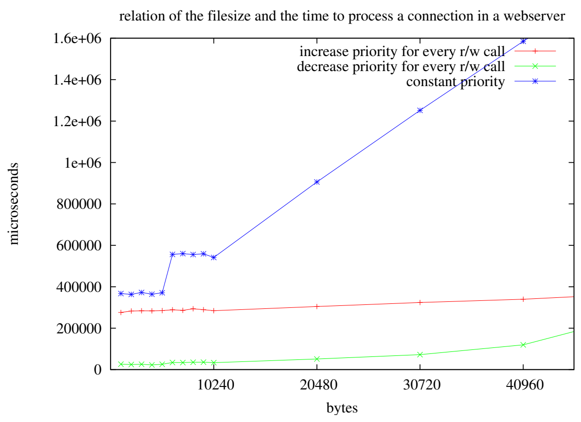


Figure 8: The connection time per file size with different traffic shaping algorithms

to the network(9 to send the file content and one for the HTTP header). This setting is a stress test regarding to the amount of system-calls the server can handle. The performance of our library is comparable to the NPTL implementation with this settings, however the behavior is slightly different. The NPTL version as a significant peek at 750 connections and the IO throughput drops from 3100 to about 2700 bytes per second. Whereas the server with TAP does not have such a peek; it slowly converges to around 2900 bytes per second. The reason for the slightly better performance of TAP is most probably due to the user-space implementation and the smaller overhead for the kernel. However, we don't want to argue about a better performance at this point of development. This figure should only show that the current implementation, although in a early version, is comparable to existing systems under certain conditions.

6.2.2 Improve Latency

For this benchmark we request different files of various size. The files are spread across different directories, and every directory contains files between 100 bytes and 900 kilobytes. The complete set is about 17 gigabyte. This setup reduces caching in the operating system to a minimum. The previous configuration accessed the disc already(because of `O_DIRECT`), however the file is cached in the disc and the disc head does not move. This can be observed by the sound of the disc and the time it takes between a disc submit and a finished notification. For the following benchmarks we use subsets of the 17 GB set. 18 random files of size between 1 and 90 kilobytes are constantly requested. Starting by 40 conn/sec we increase the connection rate every 30 seconds for additional 40 connections until we reach 600 connections per second. The file-buffer in the server is 5kb, hence a thread needs between 2 and 19 read and write calls respectively. We log the file size and the time to process a connection in the server. From this data we can calculate the average connection time per file size. Figure 8 plots the data obtained on the server. The left figure shows the curves for all file-sizes whereas the right one focuses on small files of the same benchmark. We considered only successful connections, i.e. the complete file is sent, in this figures. We see immediately the different behavior of the threads. TAP with constant priorities is basically traffic shaping deactivated, i.e. we don't change the priority of running threads. The latency, or the time to process a connection(the time between a successful `accept` and the `close` on the socket) is steadily increasing with the file size. If we enable traffic shaping and try to improve latency, i.e. decrease priority for longer running threads, we get low latency for small files and an exponential higher latency for big files. Long connections will starve eventually and the socket is reset by the client. On the other hand if we increase the priority for running threads we get a constant connection time, because already established connections get a higher priority. In figure 10 we see the data collected at the clients for different traffic shaping algorithms. The left column shows all data obtained and the right one focuses on the request rate and the response time. In these figures we relate the performance of the server to the number of attempted connections per second. We reach a saturation between 150 and 200 connections per seconds where the network IO stabilizes and the response time increases. In figure 9 we see the number of successful connections per file-size for this benchmark. The starvation of large files is obvious and there is still some future work for optimizations.

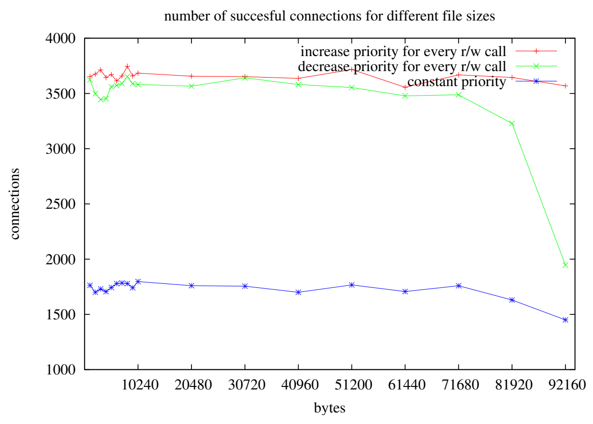


Figure 9: number of successful connections

7 Capabilities

Threading by appointment is a concurrent programming model that rests on the principle of logical timing. In most conventional thread-based models, system progress is implicitly determined by a large number of factors such as CPU performance and load. In contrast, TAP uses the notion of a reactor that explicitly controls system progress based on a given policy. Logical timing has system-wide effects. TAP therefore has a number of interesting ramifications on programming abstractions, system analysis and optimization, kernel design, multiprocessing, and real-time computing.

Programming Abstractions. Multi-threaded programming is difficult as are other forms of concurrent programming such as event-driven programming. Software developers are faced with a fundamental trade-off between concurrency and protection. More use of protection mechanisms such as semaphores means fewer opportunities for concurrency or even starvation such as deadlock. Less protection increases concurrency but may be unsafe. Yet, due to the large number of legal ways (interleavings) to execute, e.g., a system of threads, system analysis detecting starvation and guaranteeing memory safety does not scale to the level of practical relevance. TAP, on the other hand, requires either the runtime system or the threads (using a yet to be defined, TAP-specific API) to plan ahead and provide more information on resource requirements, which, in turn, creates opportunities for analysis. Nevertheless, while TAP is a first attempt to combine the advantages of thread-based programming (automatic stack management) and event-driven programming (reactor), defining TAP-specific programming abstractions remains a challenge.

System Analysis and Optimization. An important advantage of multi-threaded programming is composability. For example, threads can be created, then executed, and finally deleted at any time. However, thread composition is semantically a rather weak concept since order and time of interaction among threads as well as between threads and the system is usually unknown. More restrictive notions of thread composition based on threading by appointment give rise to system analysis that may help elevating per-thread program analysis to the system level. Instead of the traditional compose-and-error approach, TAP-based programs may be amenable to static as well as dynamic analysis, and execute only when passing the analysis.

A related problem is system optimization. An interesting question is, given some TAP-based programming abstraction, how to design TAP policies that improve system performance without changing the semantics of the given programming abstraction. For example, reordering system calls to improve performance may, to a certain extent, be done without changing relevant system semantics similar to the out-of-order execution of machine instructions by a microprocessor.

Kernel Design. We have implemented the TAP library in user space in order to avoid the complexity of kernel-space software development. However, given our encouraging benchmarks, we plan to work on a kernel-space implementation to study systems entirely based on threading by appointment. A kernel-space implementation

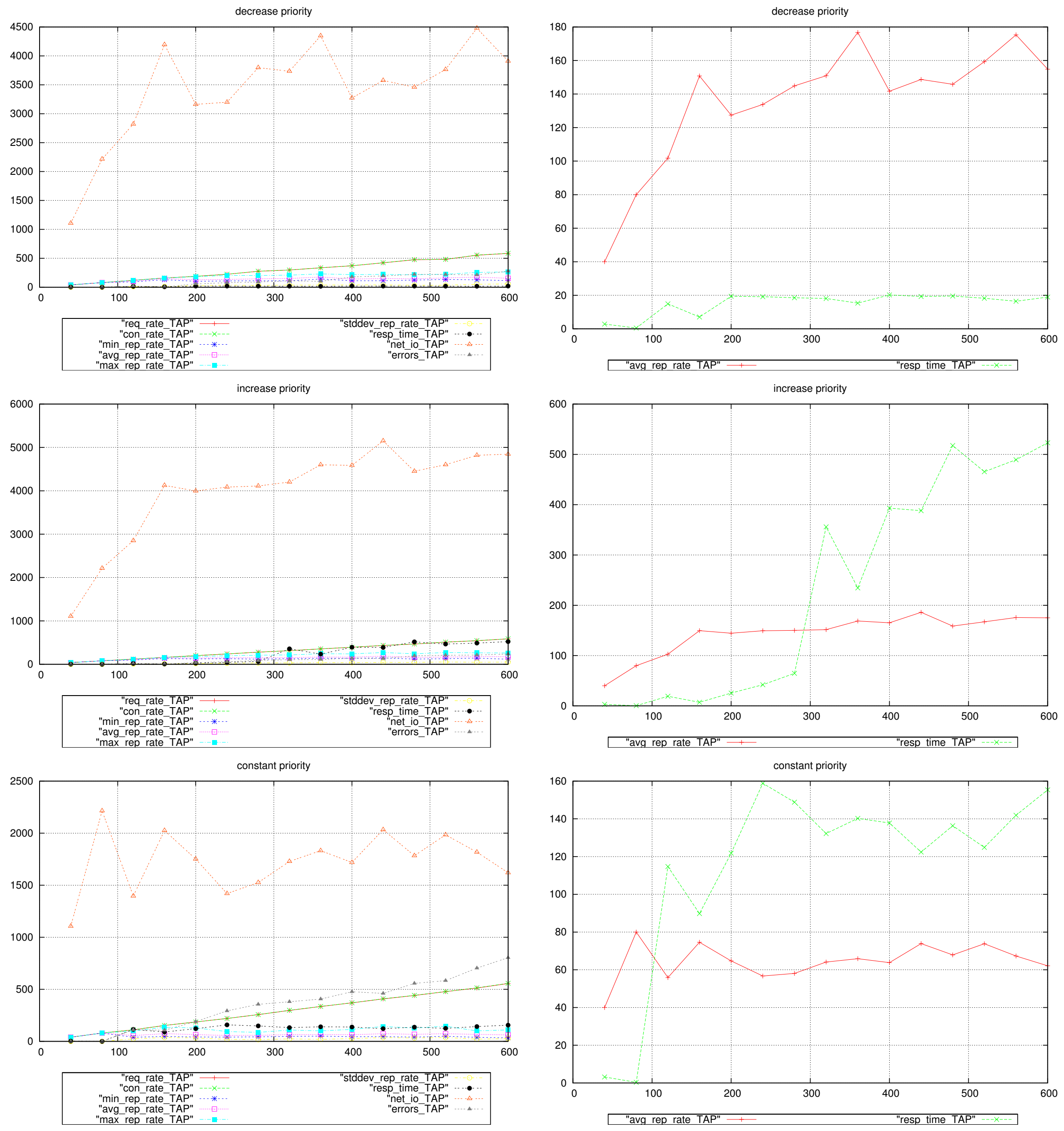


Figure 10: Data collected on the clients

will allow us to fully integrate reactor, scheduler, and I/O subsystem, and better understand their interactions and dependencies.

The TAP library uses a cooperative scheduler that simplifies the overall design of the system. However, a preemptive scheduler, besides improving system responsiveness, raises interesting issues such as TAP locking in general and an integration of TAP locking with the TAP I/O subsystem although such an integration does not require a kernel-space implementation. See Section 2.4 for more details on TAP locking.

Multiprocessing. Threading by appointment may help to improve resource sharing of threads executing concurrently on a multiprocessor system. For example, a TAP policy for multiprocessor systems could reorder system calls to maintain the invariant that each processor executes system code that is mutually exclusive with respect to usage of shared resources. The TAP policy could make appointments such that, for some amount of time, the first processor executes network-related system code, the second processor executes disk-related system code, and all other processors execute system code working on disjoint portions of memory. When the time has elapsed, the role of each processor changes while maintaining the invariant.

Real-Time Computing. Threading by appointment is originally inspired by the notion of logical execution time (LET) programming first proposed in the real-time language Giotto [7], see Section 4 for more details. Real-time computing can be supported by a TAP policy that uses a real-time appointment clock similar to the LET concept. In this case, however, appointments can be broken because a thread may not complete or I/O may not be available on time for the next appointment. This situation corresponds to a deadline violation in conventional real-time systems. We plan to use TAP in complex systems, e.g., [2], running a mixture of non-real-time and real-time applications since conventional threads suffer from a number of problems in the presence of real-time constraints such as priority inversion and deadlock.

8 Conclusion

We presented and evaluated the concurrent programming model TAP that combines automatic stack management with system call queueing. TAP was implemented as a users pace threading library with a pthread interface. With this library we demonstrated that traffic shaping system calls influences the responsiveness and performance of an application, in our case a multi-threaded web-server. With our simple monitoring scheme we were able to reduce the latency for small files at the expense of the latency of large files.

Acknowledgements. We thank Petru Flueras for implementing the POSIX-compliant API of the TAP library.

References

[1] A. Adya, J. Howell, M. Theimer, W.J. Bolosky, and J.R. Douceur. Cooperative task management without manual stack management or, event-driven programming is not the opposite of threaded programming. In *Proc. USENIX*, 2002.

[2] D.F. Bacon, P. Cheng, D. Grove, M. Hind, V.T. Rajan, E. Yahav, M. Hauswirth, C.M. Kirsch, D. Spoonhauer, and M.T. Vechev. High-level real-time programming in Java. In *Proc. ACM International Conference on Embedded Software (EMSOFT)*. ACM Press, 2005.

[3] Phillip Bogle and Barbara Liskov. Reducing cross domain call overhead using batched futures. In *Proc. 9th annual conference on Object-oriented programming systems, language, and applications (OOPSLA)*, pages 341–354, New York, NY, USA, 1994. ACM Press.

[4] Ulrich Drepper and Ingo Molnar. The native posix thread library for linux. http://people.redhat.com/drepper/nptl-design.pdf, 2005.

[5] P. Druschel, V.S. Pai, and W. Zwaenepoel. Flash: An efficient and portable web server. In *Proc. USENIX*, 1999.

[6] Louay Gammo, Tim Brecht, Amol Shukla, and David Pariag. Comparing and evaluating epoll, select, and poll event mechanisms. In *Proceedings of 6th Annual Linux Symposium*, 2004.

[7] T.A. Henzinger, C.M. Kirsch, and B. Horowitz. Giotto: A time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1):84–99, January 2003.

[8] C.M. Kirsch. Principles of real-time programming. In *Proc. International Workshop on Embedded Software (EMSOFT)*, volume 2491 of *LNCS*, pages 61–75. Springer, 2002.

[9] C.M. Kirsch. Threading by appointment. Technical Report T003, University of Salzburg, September 2004.

[10] Julian T. J. Midgley. autobench - automates the benchmarking of web servers using httperf. http://www.xenoclast.org/autobench/.

[11] David Mosberger and Tai Jin. httperf - a tool for measuring web server performance. *SIGMETRICS Perform. Eval. Rev.*, 26(3):31–37, 1998.

[12] Jr. Robert H. Halstead. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985.

[13] G. Shekhtman and M. Abbott. State threads for Internet applications. http://state-threads.sourceforge.net/docs/st.html.

[14] Andrew Tanenbaum. *Computer Networks*. Prentice Hall, 3rd edition, 2002.

[15] R. von Behren, J. Condit, F. Zhou, G.C. Necula, and E. Brewer. Capriccio: Scalable threads for internet services. In *Proc. Symposium on Operating System Principles (SOSP)*, 2003.

[16] Adam Welc, Suresh Jagannathan, and Antony Hosking. Safe futures for java. In *Proc. 20th annual conference on Object-oreinted programming systems, language, and applications (OOPSLA)*. ACM Press, 2005.

[17] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Proc. Symposium on Operating System Principles (SOSP)*, 2001.