UNIVERSITY OF OSLO
Department of Informatics

Implementing and Testing the APEX I/O Scheduler in Linux

Master thesis

Andreas Jacbosen

February 1, 2007

Abstract

This thesis seeks to test an implementation of the APEX I/O scheduler to see how it compares to modern schedulers and whether it better serves mixed-media workloads. APEX is a scheduling framework that seeks to provide deterministic guarantees for storage service to applications. The implementation is done in Linux, a modern open source operating system kernel that includes a loadable scheduler framework. The implementation compares favorably with the existing schedulers on Linux, despite problems inherent in the assumptions made in the design of mixed-media schedulers about modern operating system environments.

Acknowledgements

Many thanks are due to those who have helped and supported the creation of this work. In no particular order, they are:

My supervisors, Pål Halvorsen and Carsten Griwodz.

My friends.

My family.

Thanks.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This work seeks to examine the suitability of available Linux I/O schedulers for use with real-time applications in mixed-media environments, and to implement and test a mixed-media scheduler in Linux. Schedulers have been an active field of research since the first platter based hard disk drives were introduced. Much work has focused on minimizing seek time to increase throughput. More recently, work on providing deterministic guarantees and varying service classes has become an active field of research. These schedulers are commonly known as mixed-media schedulers. While many mixed-media schedulers have been implemented and tested in simulations environments, none matching the requirements presented in later chapters have been implemented in real operating system kernels.

While early schedulers focused on work-conserving behavior to increase overall throughput and reduce seek times, modern applications present varying I/O requirements. For some applications the prompt dispatch of I/O operations is key. For example, interactive applications may be unable to respond to the user until one or more I/O operations are completed. A work-conserving scheduler may not take this into consideration when ordering requests for dispatch. Therefore, mixed-media schedulers seek to prioritize based not only on work-conserving parameters, but also to best serve the requirements of each individual application.

1.1 Mixed-media Workloads

Applications that perform both real-time and non-real-time I/O are collectively known as mixed-media applications. These applications are characterized by dynamically varying requirements. Other mixed-media applications might include News-on-Demand, interactive libraries, multimedia file servers and so on. These applications are growing in number as broadband Internet access becomes more common.

The mixed-media workload consists of a combination of ordinary I/O operations and I/O operations with special requirements. Whereas an ordinary I/O operation has no special requirements, other than that it be guaranteed to complete at some point in the future, special requirements might be a certain amount of bandwidth or prompt service. These requirements are tied to the type of service the I/O operations are operating on behalf of. An important class of services that have special I/O requirements are real-time workloads.

A real-time workload is a workload that must corresponds to some user's viewing or interaction, for instance a streaming video playback that must be serviced regularly so that the playback is not interrupted due to lack of data.

An important concept when dealing with real-time workloads is the *deadline* of the I/O request. The deadline is the point in time at which the request must be serviced in order to maintain some definition of value. The various types of deadlines is examined later.

A mixed-media scheduler must be deadline-aware in that it must take into account the deadline of real-time I/O requests. It must also manage ordinary requests in addition to these. If bandwidth guarantees are to be made, the scheduler must also provide some degree of isolation, so that added traffic in one class of requests does not disturb the bandwidth guarantees.

Providing deterministic guarantees is made difficult by the opaque nature of modern disk drives. Many modern disk drives include provisions for transparently remapping sectors, which makes it unreliable to predict seek directions in software. Internal scheduling algorithms are also implemented in most modern disk drives. These are often Circular Look or other similar, simple work-conserving schedulers.

Mixed-media applications are becoming more common, and the demand for mixed-media platforms to service them is increasing. Therefore, the implementation of a mixed-media scheduler as a general purpose scheduler in a common OS kernel is a natural next step.

1.2 Linux

The Linux kernel is an open source OS kernel in active development[47]. The open source nature of the kernel makes it a good candidate for research projects wishing to test new technology in a realistic environment. The Linux kernel stands out from other open source OS kernels, such as FreeBSD[1] or

[1] `http://www.freebsd.org`

Hurd[2], in that it has an exceptionally well-developed scheduler subsystem that implements recently researched devices.

While the schedulers in the Linux kernel follow modern research, and even partially implement service classes, they are not suitable for use in mixed-media environments. The I/O scheduler development in Linux has been primarily focused on fairness and overcoming deceptive idleness[21].

The Linux scheduling framework implements a two-level scheduling hierarchy, which is common to most mixed-media schedulers. This makes Linux a natural choice for implementation a mixed-media scheduler.

1.3 APEX

The Adaptive Disk Scheduler for Mixed-media Workloads (APEX) is a mixed-media scheduler designed for a Learning-on-Demand environment[25]. The Learning-on-Demand environment consists of a client that presents the learning material and a multimedia database management system (MMDBMS) server that provides content. The server must manage many different types of data, ranging from streaming audio or video presentations to still images and raw text.

The original presentation implemented APEX as a part of the MMDBMS, with full access to the disk. This specialized application is common for mixed-media I/O schedulers[7, 26, 42], but none have been implemented as part of a general purpose operating system (OS) kernel.

APEX details a two-level hierarchy of schedulers which act in conjunction to prioritize and optimize the dispatch of requests. The top level scheduler deals with the special requirements of each queue of requests, and the bottom level scheduler performs work-conserving to optimize throughput.

The original presentation of APEX also includes elements which are not implemented as part of this work. These include the queue manager, which enables each process to manipulate multiple queues with different requirements, and the admission control manager, which prevents more resources from being allocated than the block device can service. The possible ways to implement these on Linux are examined and found to be impeded by the generic structure of the I/O framework.

The APEX scheduler provides improvements over the existing schedulers in some cases, but the fundamental limitations of the low-level nature of the Linux scheduler framework makes a complete implementation a problematic task.

[2]`http://www.gnu.org/software/hurd/`

1.4 Contributions

This thesis details a partial implementation of APEX in Linux, with emphasis on the batch builder module of the mixed-media scheduler. Various tests are performed on the APEX implementation and other existing schedulers in Linux. The results show that existing schedulers are not suitable for use in mixed-media environments, and that APEX is better suited for these.

1.5 Structure

The structure of this thesis is as follows. Chapters 2 covers modern hard disk drives in detail, explaining how they help and hinder the work of schedulers. Chapter 3 examines the requirements presented by continuous media, which are the focus of the real-time component of APEX. An overview of research on schedulers is provided in chapter 4. Chapter 5 examines in detail the Linux kernel, with focus on the I/O framework. The schedulers currently implemented on Linux are covered in this chapter. The implementation of APEX in Linux is covered in chapter 6. Chapter 7 covers the test setup and evaluates the results of these. The final chapter offers a summary of the conclusions and some ideas for future work.

Chapter 2

Overview of Modern Hardware

While individual models differ in performance, modern hard disk drives have certain features in common. As shown in figure 2.1 on the following page and figure 2.2 on the next page they have one or more magnetic platters, a head for each (or one for each side of each) platter, a separate actuator arm for each head. The arms are mounted on a common pivot, so they move in unison. Each platter is divided into concentric tracks, which are in turn sectioned into sectors. Tracks are grouped by parallels on platters, which are called cylinders.

Each sector stores a set amount of bits, usually 512 bytes. When it is not specified otherwise in this text, sector may be assumed to mean a unit of 512 bytes. This is also the standard used internally in the Linux kernel, regardless of actual hardware sector size. Most disks are *zoned*. Since their sectors are of equal size, it is possible to fit more sectors into the outermost tracks. Tracks with equal number of sectors are said to be in the same zone.

The platters rotate in lock-step around a central spindle, at an almost constant speed. The rotation speed varies by model, though most comparable models keep similar speeds. 7200 rpm is normal for desktop IDE drives, and laptop drives are usually either 5400 rpm or 7200. SATA and SCSI drives are often 10000 or even 15000 rpm. This speed varies slightly during operation, but is nominally constant. The rotation frequency combined with the sector density determines the theoretical maximum transfer speed from a device. While older devices could not read continually from a track, modern hard drives can do so. Therefore, the theoretical maximum, often referred to by vendors as the *media transfer rate*, can be stated as in equation 2.1. (Where M is media transfer rate, s is the number of sectors in a track and r is the drive's rpm.)

$$M = s * r/60 \tag{2.1}$$

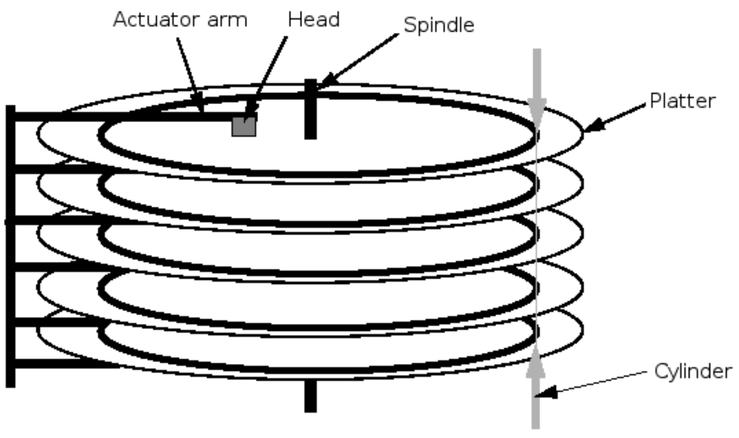

Figure 2.1: The side view of a hard disk drive[35]

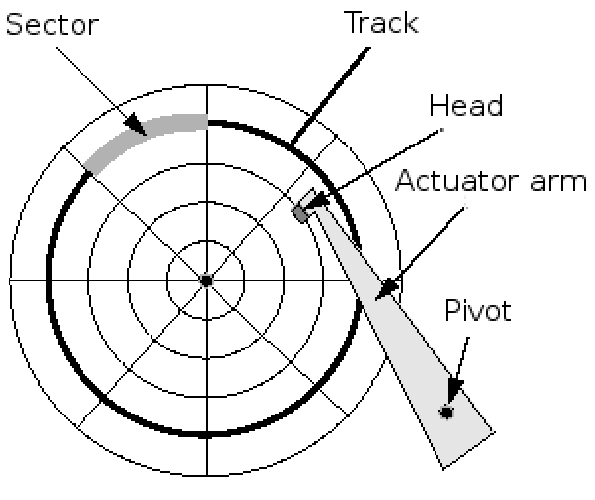

Figure 2.2: The top view of a hard disk drive[35]

2.1 Seek Delay

When servicing a request, a hard drive must perform four actions before the head is placed over the correct sector. First the controller must process the request. This entails processing such as translation of the request address from logical block addressing (LBA) format to the correct cylinder, head and sector. Controller overhead is typically in the range of 0.3-1.0ms [35].

Next an access request causes the arm to move the heads to the correct cylinder. The time required to perform this move is known as *seek time*. Seek time is primarily dependent on the size of the platter and the material composition of the arm. Larger platters imply longer seek times for two reasons. One is the obvious reason, that the seek head must travel further between tracks. To reduce seek times the arm moves very fast and has to be able to withstand extreme acceleration. On larger platters the arms must also be longer, thus causing more material stress. It is very important that the arms not deform during operation, since physical contact between the head and the platter will destroy data. Seek time may be reduced by reducing the size of the platter or by improving the physical properties of the material composite used for the actuator arms. Reducing the size of the platter has the dual advantage of allowing smaller arms that better withstand acceleration and shorter seek distances.

Decreasing the size of the platter has the disadvantage of reducing throughput, since absolute speed is reduced for smaller circumferences, and reducing storage space. The Seagate Cheetah 15k series is an example of a disk that makes this trade-off[36]. It compensates for the decreased throughput by increasing the rotational speed to 15000rpm.

The correct head is then chosen and must be further aligned to the track. Production differences in hard drives causes the tracks to have small offsets and imperfections that require tiny compensations by the arm to place the head correctly. The time required to do this is *settle time*. Settle time is approximately equal to the minimum seek time provided by hard drive manufacturers.

Finally the head must wait for the platter to rotate so that the correct sector is under it. The time this takes is known as *rotational delay*. In the worst case, the sector to be read has just passed under the head and the platter must complete a full rotation before the request can be processed. As is obvious, worst case rotational delay is defined by the rotational speed of the platters. For 5400 rpm drives, the time taken to complete one full rotation is approximately 11ms. On a 15000 rpm drive the same can be accomplished in approximately 4ms. While some research has been done to reduce rotational delay[19], modern block devices are addressed logically rather than by physical geometrical information, making it difficult, if not impossible, to know a sector's position in a track. Without this knowledge, schedulers cannot take into account rotational delay when choosing which request to service next.

Hard drive manufacturers usually supply three seek times for their devices. These times all include seek and settle time along with rotational delay. One is the time required for a full sweep, moving the read head from one edge of the drive to another. The average seek time is given as the time required to seek approximately a third of a full sweep, and is usually between 2-12ms[38, 37]. This number is sometimes achieved by dividing the time for a full sweep by three. The third time provided is usually called track-to-track seek. This covers requests where the seek moves to an adjacent track. This approximates to the settle time of the device. Typical track-to-track seeks vary between 0.2ms and 1.0ms. This might also approximate the time required to switch the active head, called a *head switch*. While head switches earlier were much quicker than track-to-track seeks, cylinder alignment is becoming worse as track density increases. Thus the time required for a head shift is approaching the time of a track-to-track seek.

Hard drive manufacturers often give separate, slightly lower numbers for reads than write requests. This is because reads may be attempted from the moment the arms are in the right cylinder, before the head is known to be correctly aligned to the track[35]. The error checking data on the device ensures that erroneous reads are discarded. Thus there is no penalty for making such speculative reads. The benefit here reduces the average rotational delay when performing read requests.

2.2 Access Time

The actual time spent reading or writing to a single sector is negligible compared to the time spent aligning the head correctly. It is a product of rotational speed and the number of sectors in the track.

$$a = r/s \tag{2.2}$$

From equation 2.2, where a is the access time for a single sector, r is the time required for one rotation and s is the number of sectors per track, so that a 7200 rpm disk with 400 sectors per track (a low estimate for a modern disk), will spend approximately 20μs. Even when accessing several sectors at a time, which is common behavior, this time is dwarfed by the seek delay.

2.3 Internal Optimizations

Modern hard disk drives perform many transparent operations to improve performance. They cache data, perform prefetching and often perform internal scheduling of operations. They may also perform internal scheduling to improve throughput and seek times. The scheduling algorithm used is usually SCAN or a variant thereof[35].

Internal hard drive caches are growing as prices for dual-ported[1] static random access memory (RAM) continues to drop. Current drives commonly have caches between 8-16MB. Older disks performed *on-arrival read-ahead* caching. (This is sometimes referred to as “zero-latency read.”) On-arrival caching was done by immediately beginning to read from a track before the desired sector was under the head. Thus the entire track would be in cache in case later requests from the host should need them. This has since been replaced by simple read-ahead caching, in which the drive continues to read after the last request from the host is served.

Read-ahead strategies can be aggressive, crossing track and cylinder boundaries. This type of policy would favor longer sequential accesses, while degrading random access performance. The read cache is segmented to support caching for multiple sequential read streams.

Write caching may be done in some cases as well. For example, some writes may simply immediately reported as complete, despite only having been written to cache. This policy is not without risks, as power failures may leave the filesystem in an inconsistent state. To avoid this, certain requests may be flagged as not allowed to be reported complete until they are actually written to disk. Another strategy is to provide nonvolatile memory, for example by using battery backed RAM. Current research is also focusing on using nonvolatile flash memory to improve caching[11], drives such as this are referred to as hybrid drives.

Immediate reporting is beneficial when data in the write cache is overwritten in place, a common a occurrence. The drive might also collect write requests until they can be efficiently scheduled so that they take less time to perform overtime.

Using command queueing to allow a host to send multiple requests to the disk, so that the controller may determine the best execution order, is a common strategy. On some modern parallel ATA devices and most Serial ATA 1.0 or SCSI devices, this is achieved with tagged command queueing (TCQ), whereas Serial ATA II devices use native command queueing[20]

[1]This special type of static RAM is needed to communicate with both the disk and bus interfaces.

(NCQ).

2.4 Summary

The primary consequence of the setup of modern hard drives is that they present themselves as a black box device. While they can be relied on to behave as expected most of the time, they provide no performance guarantees. This places severe limits on what a scheduler that aims to provide both statistical and absolute guarantees can achieve by mathematically manipulating the disk geometry.

Since the black box delivers improved and more predictable behavior amortized over several requests, a scheduler should instead perform high level optimizations. This way the disk may perform low level optimizations based on its intimate geometric knowledge.

APEX does this by sending requests to the hard drive in batches. By trusting the amortized performance to maintain the batch's total deadline, the variance in service time for single requests can safely be ignored. It is important that the scheduler supply sufficiently large batches that this variance can be evened out. This is examined further in section 4.4 on page 18.

Chapter 3

Continuous Media Requirements

There are many overviews of requirements for storage systems[15, 17, 50]. This chapter reviews some of these, and then looks at the specific requirements Lund identifies for I/O schedulers[25]. These are later used as criteria for judging the suitability of I/O schedulers.

3.1 General Continuous Media Requirements

Continuous media (CM) applications often require that disk accesses be processed and returned within a specific interval. The result of a late read is often that the user experiences a loss of quality in the media (such as a jitter in a video stream or a gap in audio). This can be worse than a missing frame of video or audio, since the lag is often more noticable than the simple omission of the frame would be, given that the next frame were delivered in time. This differs from standard applications that are more interested in completely reliable reads with high throughput than speedy reads. An example of the latter type of application might be a compiler or web server, whereas the former might be a multimedia database management system (MMDBMS), video streaming server, interactive environment or other CM application.

While video and audio streams differ in presentation, they are mathematically equivalent in respect to real-time requirements[15]. Some formats might also preclude the dropping of frames, since information is not encoded independently, but rather as the difference between the relevant frame and a previous one.

Data requests that become worth less over time can be said to be associated with a *deadline*. Deadlines can be divided into three classes[29]. Requests with *hard deadlines* absolutely must be met, or catastrophic failure occurs. Examples of this type of request could be topological data in a flight control system. Requests with *firm deadlines* become worthless after the deadline is exceeded, without causing a failure. Independent video or audio frames in a stream are examples of this might have firm deadlines. The final class is for requests with *soft deadlines.* These requests are not immediately worthless after the deadline is exceeded, but their value decreases and they may eventually become worthless.

Halvorsen et al.[17] identify three other requirements specific to systems designed for CM applications, in addition to existing OS requirements. The system must be able to service requests with *timeliness.* Different types of applications may require low latency, or regular requests each to be serviced within given deadlines. This requirement comes into conflict with disk bandwidth utilization, as seen in the implementation of EDF. The system must also be able to handle *large file sizes and high data rates.* Interactive environments, multimedia and similar are often presented in files orders of magnitude larger than text or graphics. The rate of playback is often higher as well. For example, MPEG-2 video, the format used for DVDs, has a data rate of 15Mbit/s, which at 25 frames per second is approximately 75KB per video frame. This is comparable to about 100 pages of plain text.

A CM application may also require *multiple data streams* to be available at the same time. A system must therefore also consider strong relations between multiple streams. An example of this is movie playback, which might involve tightly interleaved video and audio streams.

Since CM playback is often periodic, such as videos that display a given number of frames per second, requests can be processed in batches. Many schedulers take advantage of this fact. Aside from scheduling policies, it is also possible to perform data placement optimization to improve performance in some cases[15].

3.2 Specific Design Requirements

The five basic requirements that APEX is based on are presented in [25]. These aim to provide universal requirements for mixed-media systems. APEX is designed to fulfill all these requirements, and these are used as the basis for testing the implementation later.

Since CM applications often have low latency requirements, a scheduler must ensure *efficient reads.* It must not incur any significant overhead, and must also use geometry awareness to avoid causing excessive seekiness. Where possible the scheduler should aim to avoid overhead. While this might imply that inefficient algorithms have no place in a scheduler, the CPU runs so many orders of magnitude faster than hard drive requests that CPU overhead is not a primary concern.

The scheduler must also support *multiple service types and guarantee levels*. At the very least the scheduler must support both real-time and best effort services. Preferably there is support for deadline guarantees, low latency services, high throughput services and best effort. Within each of these service types there should be prioritization. For example, low latency applications may have different hard deadlines for their requests, or high throughput services may have different bandwidth requirements. The scheduler should also provide support for request dropping and similarly relevant functionality if possible. A mixed-media scheduler must be able to provide as many of these services as possible.

For the system to be able to maintain optimum efficiency regardless of active services, the scheduler must be *flexible*. It must adapt to the current requirements dynamically and be continually configurable to ensure that the type of services active does not negatively impact performance.

The scheduler must also provide *isolation* so that the active requirements of one service type may not adversely affect other types. Burstiness of best effort traffic should not cause failure of deadlines, throughput requirements or other quality of service (QoS) guarantees. Similarly, best effort requests should not be starved by higher classed services.

Finally, *work-conservation* is required. Due to variable conditions, the workload provided by CM applications can vary widely. Since it is impossible to continually distribute bandwidth between all active processes, the scheduler must ensure that work-conservation minimizes loss of efficiency. This way unused bandwidth can be used when available.

3.3 Summary

The common consequence of systems not designed to the requirements of CM playback is deteriorated user experience of quality. Audio playback might skip while video playback might not match the audio track. To avoid these adverse consequences, systems must be built around the requirements presented by CM applications. One part of this system framework is the I/O scheduler.

Chapter 4

Overview of I/O Scheduling Algorithms

Improving disk performance has been an active field of research for some time. Most early work focused on improving throughput while maintaining guarantees that requests are serviced eventually, known as *fairness*. As high speed networks and in particular the Internet provide an increasingly feasible frameworks for transporting information and multimedia on demand, the need to to optimize response time in addition to throughput has become apparent. While current scheduling algorithms provide guarantees that requests are serviced eventually, there are no guarantees on actual response times or throughput for specific processes.

4.1 Early Algorithms

The simplest algorithm for disk scheduling is First Come First Served (FCFS). This naive algorithm simply services each request in the order it appeared. The advantages of this algorithm is that it is simple to program and that it provides guarantees for fairness in that so long as the drive continues to process requests, any request queued is serviced in due time. A disadvantage of FCFS is that it spends much time seeking rather than reading, leading to poor throughput and long response times.

Shortest Seek First (SSF, sometimes called Shortest Seek Time First) sorts the access queue by seek distance or seek time (the difference between these two is non-trivial[19]). This improves throughput by ensuring that the drive spends less time seeking and improves response times similarly. The latter result might be considered unintuitive, since an SSF algorithm might continually service a series of requests on one area of a disk, while starving requests to other tracks that are far away. By adding a second criteria for choosing requests, based on the age of the candidate requests. If a request gets too old, it is automatically scheduled next, regardless of seek lengths. These long seeks can improve disk throughput by moving the disk head from an area where it is servicing increasingly infrequent and sparse requests, to an area where many, geometrically close requests are queued[18].

SCAN scheduling tracks the head back and forth across the surface, processing requests as it goes. This guarantees that each request is processed within two sequential scans of the disk surface. Since the seek distance remains the same regardless of how many requests are serviced, this greatly increases throughput compared with FCFS. The main benefit over SSF is a reduction in maximum response times [39]. Since this algorithm operates similarly to elevators, it is often called the elevator algorithm. Modern disk drives often run SCAN scheduling or variants thereof internally.

Many variations on the elevator algorithm exist. The following are a few which are relevant to the Linux kernel I/O scheduling layer.

Basic variations such as Circular SCAN (C-SCAN) improve fairness by only scanning in one direction, when it reaches the end of the disk it performs one long seek back to the other side. This improves performance for requests at the edges of the disk, where normal SCAN spends half as much time as on requests towards the middle.

The Look elevator optimizes SCAN by scanning only to the last request on that edge, rather than fully to the edge of the disk. This can cause delays for requests that arrive just as the head is turning, but gives an overall improvement in throughput and response time[50]. The C-Look algorithm works like C-SCAN in that it only scans in one direction. C-Look and C-Scan improve performance not only by increasing fairness, but by exploiting the prefetching nature of read-ahead caching policies.

VSCAN(R) seeks a compromise between SCAN and SSF, taking as a parameter R, a real designating the weighting towards SCAN behavior. VSCAN(0) is equivalent to SSF and VSCAN(1.0) is the same as SCAN (or a variant thereof) [13]. When active, the read head continues scanning in the same direction as the last seek so long as it has a request within its SCAN area, defined to be R x (total number of cylinders on the disk). Requests outside this area are scheduled in SSF order, where requests in the current SCAN direction have their distance reduced by the SCAN area. A good value of R is found to be 0.2, yielding both throughput and mean response time superior to SSF and SCAN.

4.1.1 Suitability of Early Algorithms for Multimedia Servers

The series of algorithms that seek to optimize throughput and response time without providing guarantees are, in order of optimality, increasingly useful for applications such as multimedia servers. However, without providing guarantees, they remain unsuitable. They inevitably violate deadlines, possibly consistently, in favor of applications without real-time constraints. As an example, an application replaying a compressed audio file to a user would require a low amount of bandwidth compared to a compiler piecing together a large program, but since the scheduler only guarantees fairness, the audio player has no guarantee that the frames are delivered in any timely fashion. The audio player can counter this by buffering the audio in main memory, but this is only a stop-gap measure. It may also, in a system with low amounts of main memory, lead to further disk accesses due to swapping.

4.2 Deadline-Aware Algorithms

Early work on deadline-aware algorithms focused on situations where the scheduler was concerned exclusively with deadline-aware processes. A common requirement of modern systems is that they be able to deal with multiple classes of I/O operations. Only the most basic of these are covered, focusing on the relevant areas of developments.

The most basic deadline-aware scheduling algorithm is the Earliest Deadline First (EDF) algorithm[1]. This algorithm holds a priority queue ordered by request deadline, and processes requests in this order. Since this does not take into account read head positions, EDF spends much time seeking and has bad drive utilization. Real-time scheduling problems have been solved using EDF previously, but these assume preemption, and do not take seek overhead into account[32].

Combining EDF with SCAN improves the throughput and response time performance of EDF while making SCAN deadline-aware. This has been done in SCAN-EDF[33] and D-SCAN[2]. In SCAN-EDF, requests are grouped by deadline, and requests with the same deadline are processed in SCAN order. This removes the FCFS element of EDF, improving throughput and response times. An overview[32] comparing SCAN-EDF to C-SCAN and EDF shows that C-SCAN can service the most concurrent streams, while also giving the worst response time to aperiodic requests without constraints. SCAN-EDF can service almost as many concurrent streams as C-SCAN, while maintaining an aperiodic response time similar to that of EDF. This confirms the result that combining scheduler algorithms improves balance between the various requirements. They also note that increasing request sizes increases the number of concurrent streams servicable, while also increasing response times.

Deadline SCAN (D-SCAN) and Feasible Deadline SCAN (FD-SCAN)[2] service the request with the earliest deadline first, performing a SCAN in the seek direction, servicing other requests as it proceeds. FD-SCAN differs from D-SCAN by only takes into accounts requests with deadlines it estimates it can meet.

4.3 Stream Oriented Scheduling Algorithms

Many schedulers exist that focus on servicing multiple continuous data streams. To achieve this they are built around the assumption that requests are periodic at stable data rates. Stream oriented schedulers typically dispatch requests in fixed-length rounds[25]. By focusing on fair distribution of bandwidth and periodic rounds, stream oriented scheduling algorithms provide optimizing behavior without any firm guarantees.

A non-comprehensive list of stream oriented scheduling algorithms includes Pre-seeking Sweep[14], Grouped Sweep Scheduling (GSS)[51], YFQ[9] and Greedy-but-Safe Earliest Deadline First (GS_EDF)[46].

4.4 Mixed-Media Scheduling Algorithms

This section provides an in-depth review of two mixed-media schedulers, along with a presentation of APEX and an overview of other mixed-media schedulers. It also examines how the available schedulers conform to the requirements. The two schedulers chosen for in-depth review are those that best conform to the requirements, Cello and MARS, and thus are most relevant.

Mixed-media schedulers seek to service both real-time and best effort requests on a type-specific basis. While real-time requests are treated in a deadline-aware manner, best effort requests are insured against starvation. This is usually accomplished by introducing a two-level scheduling hierarchy in which different request types are treated discretely at the top level and dispatched in batches to a lower level scheduler that performs work-conservation.

4.4.1 Cello

The Cello disk scheduling framework[42], meant for use together with the Symphony filesystem[41], introduces the idea of a two-level scheduler. Since it is tied directly to the filesystem the Cello scheduler framework can take advantage of extra metadata information. To assist with this, it implements several system calls for file handling, overriding the standard system calls. A process can specify service requirements on opening or creating a file, or it can specify a deadline when performing a read or write.

Cello operates with two levels of schedulers. The *class independent* scheduler is responsible for choosing which request classes are to be serviced. The *class specific* schedulers choose which request to service when called on by the class independent scheduler. Each class specific scheduler is designed to serve an *application class*. A process may be specified to be part of an application class during runtime, or a file may be specified to be part of an application class at opening or creation.

The class independent scheduler fulfills two functions. It distributes access to the disk among the specific classes according to some weight, and it exports an interface to the class specific schedulers that is used to determine where in the scheduled queue to place newly dispatched requests. The class independent scheduler performs rounds, operating on a coarse time scale with the goal of providing isolation for specific classes. Given the standard interval, each class is given a weight. This weight can be given in disk access time or bytes accessed. The class independent scheduler visits each class specific scheduler in turn, granting them access to the disk according to their weight. To avoid seek overhead violating deadlines, this is calculated into the cost of a request. As such, Cello relies on the ability to calculate both the seek time and rotational latency of a request.

Cello implements a given number of queues. One scheduled queue, for requests that are ready to be dispatched to hardware, and one pending queue for each application class. New requests are placed into the pending queues.

The class specific schedulers decide which requests to choose for dispatch from their own pending queues, and where in the scheduled queue they should be inserted. The class specific schedulers must take into account the deadlines and requirements of other requests in the scheduled queue. To accomplish this, the class specific schedulers import state information of the scheduled queue. The constraints are defined in terms of *slack*. Slack is the amount of time a request can be delayed from dispatch and still be serviced in time for any deadlines it is associated with.

Cello presents three class specific schedulers. One for interactive best effort applications, throughput-intensive best effort and soft real-time applications. They suggest that the best effort pending queues are sent to the scheduled queue in FIFO order. The interactive best effort requests should be inserted using a slack stealing technique. It does this by inserting them towards the beginning of the scheduled queue, but not so far towards the beginning that deadlines are violated. To minimize seek, these batches are sorted in SCAN or other work-conserving order. The throughput intensive best effort requests are suggested inserted late in the queue, so that low latency and deadline requests are not affected. Soft real-time requests would be dispatched to the scheduled queue in SCAN-EDF order. Placement in the scheduled queue would be done according to deadline.

Cello introduces several important ideas, in particular the two-level scheduler. It also introduces time slicing, which the Linux CFQ scheduler uses.

The use of seek time and rotational latency in calculating the request cost is somewhat problematic as shown in chapter 2 on page 5, but it is not unreasonable to assume that estimates can be done based on hardware geometry and specifications. Cello does not, however, take specific advantage of errors in dispatch time calculation.

Cello provides some work-conservation in that it redistributes unused disk bandwidth to queues with pending requests. It specifies multiple service types, and class specific schedulers for these. However, it is not possible to dynamically reallocate bandwidth for the queues. This also prevents the work-conservation from being optimal, since it is also distributed according to the weight of the queue.

4.4.2 MARS

Buddhikot et al.[10] presented MARS, a framework for modifying NetBSD[1]to function in a scalable, distributed fashion to serve multimedia on demand. MARS covers much more than just I/O scheduling, including mechanisms for zero-copy reads from disk to network and data layout. To differentiate between the MARS framework, the scheduler is referred to specifically as the MARS scheduler. Like many mixed-media schedulers, the MARS scheduler uses a two-level scheduling system. Each real-time process is associated with a *job queue*, which is internally scheduled according to an arbitrary algorithm, which may be FCFS, or a more advanced algorithm such as SCAN-EDF. There is also a normal request queue for requests which come from processes without real-time requirements.

[1]NetBSD is an implementation of 4.4 BSD, see `http://www.netbsd.org`. The implementation and evaluation of MARS was done with only one real-time queue with other limitations.

Requests are chosen from the job queues by the *job selector*. After the job selector has chosen requests, it sends the requests to the lower level driver which orders them and dispatches. The lower level driver could use an arbitrary scheduling algorithm, but in practice uses the elevator algorithm present in the NetBSD SCSI driver. This satisfies the requirement for efficiency.

The job selector in MARS employs a Deficit Round Robin (DRR) queueing algorithm[43]. While DRR was originally designed for network queueing, it is appropriate for fairly distributing resources between any kind of queue. In DRR, each queue is given a quantum service for each round. In the case of MARS, the quantum is given as a number of bytes each queue is allotted for requests. When servicing a queue, if the value of the quantum is greater than the size of the request, the request is removed from the queue for dispatch and the size is deducted from the remaining quantum. If the quantum is less than the size of the request, the deficit is carried over for the next round and service of the next queue begins.

The DRR algorithm gives MARS implicit work-conservation, since it shortens the round when there are unused service quantums. The class specific schedulers in MARS are not specified, which makes it difficult to apply requirement analysis. However, the lack of flexibility, in that it only specifies real-time and non-real-time queues and the lack of multiple service types is problematic.

4.4.3 APEX

APEX[2] was originally presented[25] as a user-space framework for scheduling requests from multimedia database management systems. It is based on requirements presented by a Learning-on-Demand (LoD) environment. As the most recent of the mixed-media schedulers, APEX utilizes previous results to optimize its performance and generalizations. It is a two-level scheduler, using a variant token-bucket algorithm to treat individual queues and a simple SCAN algorithm to optimize throughput. It aims to provide both deadline guarantees for requests and work-conservation.

Lund presents a scenario and system architecture which together define the context for APEX's requirements. The requirements APEX is based on are examined in chapter 3 on page 11. What follows provides a short examination of the context. The LoD-application is an approach to providing

[2]In cases where it is important to differentiate between the original APEX as provided by Lund, and the implementation of APEX on Linux, it is referred to as the *original presentation*. The exact differences are summarized in section 6.4 on page 73. In this chapter APEX may be assumed to refer to Lund's original scheduler.

interactive distance learning (IDL), a paradigm of instruction and learning that attempts to overcome limitations of distance and time. It is divided into a client and a server. The server is a centralized system which hosts teaching content. The client is used by the content consumer, who is situated at a remote location, to view the content provided by the server.

The LoD content can be plain text, plain text with pictures, multimedia presentations and so on. Multimedia presentations can be video, audio or a combination of the two. The LoD server must support both reading and writing of all these types of content. Content providers can create new content interactively, for example in response to questions by content consumers. The content providers must also be able to author the presentation of the material, by specifying various metadata relations between content objects. The server must also be able to perform content searches.

These specifications present a case for using a mixed-media I/O scheduler. An alternative could be to use multiple servers, with separate servers used for streaming multimedia content, but this provides complications of its own. For example, increased hardware costs and complications of access control.

The LoD server is implemented as an MMDBMS. APEX is integrated into this MMDBMS in an attempt to provide QoS guarantees to the LoD clients. Lund[25][3] provides an extensive overview of the MMDBMS and LoD data model, and readers with further interest in the matter may refer to that.

APEX consists of multiple components that work together to provide a complete framework for scheduling requests for the MMDBMS. These are the *request distributor*, *queue scheduler*, *batch builder*, *queue manager* and *bandwidth manager*. This breakdown is designed so that queue management and request management may be separated into discrete entities. Request management is handled by the request distributor, queue scheduler and batch builder. Queue management is handled by the queue manager and bandwidth manager.

The queue management module handles dynamic creation and destruction of queues, which enables APEX to be transaction-aware. It provides the admission control interface, which is used when processes wish to create new queues or reserve bandwidth for queues. Bandwidth reservation is not done upon queue creation, but instead deferred until a process begins a transaction on a queue with bandwidth reservation specified. In this way, bandwidth does not remain unused simply because a queue has not yet begun servicing requests.

The request management module handles treatment of individual requests. Distribution of requests to the correct queues is done by the request

[3]In chapter 3.

distributor. The queue scheduler component provides a scheduling policy specific to each queue. Different types of queues may be given different policies. The batch builder chooses requests from various queues to service based on bandwidth reservation, reorders them based on a scheduling policy and sends the batch to the disk driver.

The batch builder limits bandwidth usage by applying an extended token bucket model. Token buckets[28] are commonly used for traffic shaping and flow control on networks. For each connection a flow rate r and a bucket depth b are defined. The flow rate is the number of tokens the queue is allocated per second. The bucket depth represents a maximum number of tokens a queue may accrue when not using all allocated tokens. In APEX each token represents one database page of data.

When the batch builder performs a round, it visits each queue that has reserved bandwidth in turn. It checks whether the queue has tokens available and services a number of requests corresponding to this number of tokens. This phase continues until the batch size is met or until no queues have tokens left.

To ensure that best effort queues are not starved, the batch builder also performs a work-conservation phase. During this phase, all queues are served regardless of whether they have tokens available. This is an extension to the token bucket model, which provides added throughput compared to what a more tradition token bucket model would give. The order in which queues are serviced during this phase may be changed in order to serve varying priorities. The reference implementation is to serve them in the same order as the reservation based phase, simply adding consideration of best effort queues.

When building a batch, APEX chooses a round time based on the queue with the nearest next deadline. Using this time and the estimated time it takes to service a single request, the batch builder decides how many requests it can at most service in this round. The estimate is deliberately left high to insure against variance.

APEX implements a provision for low latency requests by allowing these to be inserted into the dispatch queue after the initial batch is built. During execution, if a new request arrives that is marked as low latency, it may immediately be inserted into the dispatch queue if the actual time of execution so far has been lower than the time it was estimated to take. Since the estimated time of execution is deliberately left high, this is usually the case.

4.4.4 Other Mixed-Media Schedulers

Cello, MARS and APEX have been given in-depth examinations, this section provides a summary of other mixed-media schedulers. Lund[25] provides an overview of these that this section follows.

Nerjes et al.[27] explore variations on a simple two-service scheduler. It supports continuous and discrete requests, with disk bandwidth distributed between the two. This is further tested and analyzed as FAMISH by Rompogiannakis et al.[34]. The only parameter that can be adjusted is the bandwidth distribution between the two service types, rendering the scheduler inflexible.

The scheduler presented by Kamel et al.[22] supports an arbitrary number of priority levels, and orders requests based on priority level and deadline. It drops requests that will obviously violate deadlines. The scheduler maintains a single SCAN sorted queue that new requests are inserted into under various conditions. New requests are not allowed to violate the deadline of a already inserted requests with higher priority, and insertion must minimize the overall number of deadlines violated. This scheduler has issues with starvation and isolation between classes.

In Ramakrishnan et al.[31] a scheduling and admission control algorithm are presented. The scheduling algorithm provides two service classes, real-time and best effort. The best effort requests are serviced in slack time. To prevent starvation of the best effort requests, a portion of the bandwidth is reserved for this class. This scheduler is not flexible, having only two service classes without any configurations according to varying needs.

The Fellini multimedia storage server[26] includes a scheduler that supports real-time and best effort service classes. The scheduler dispatches requests in fixed-length rounds, and the service classes are isolated from each other by using set fractions of each round. Unused bandwidth from the real-time queue is redistributed to best effort requests if necessary, rendering the scheduler partially work-conserving.

The scheduler provided for the Prism file server architecture[48], hereafter the Prism scheduler[49], has three service classes with admission control for each of these. The three service classes, periodic, interactive and aperiodic, can separately be granted throughput guarantees. The Prism scheduler combines requests in SCAN order to improve efficiency, but is only partially work-conserving because it only uses unused bandwidth to service interactive or aperiodic requests.

The ΔL scheduler[7] is a deadline-driven scheduler that provides deterministic guarantees to real-time tasks. It uses slack time to service best effort requests. The ΔL scheduler is similar to that presented by Ramakrishnan et

	Efficiency	**Multiple service types**	**Flexi-bility**	**Isolation**	**Work-conservation**
[27]	-	(X)	-	X	(X)
Fellini[26]	(X)	(X)	-	X	(X)
Cello[40]	(X)	X	(X)	X	(X)
MARS[10]	X	(X)	(X)	X	(X)
[27]	-	(X)	-	X	(X)
Prism[49]	(X)	(X)	(X)	X	(X)
[22]	(X)	(X)	-	-	(X)
ΔL[7]	-	(X)	-	X	(X)
DSSCAN[16]	(X)	(X)	-	-	(X)

Table 4.1: Summary of disk scheduler characteristics[25]. ("-" = not supported, "(X)" = partially supported, "X" = supported)

al.[31] and has similar drawbacks.

Deadline Sensitive SCAN[16] (DSSCAN) supports periodic and aperiodic real-time requests, interactive requests and best effort requests. The scheduler functions as SCAN, unless scheduling a request will cause another request to violate its deadline, in which case it schedules according to EDF. Interactive requests are supported by adding an FCFS service for them. The priority order then becomes EDF if a deadline is in danger of being violated, then FCFS if there are any interactive requests and finally SCAN order to increase throughput. While DSSCAN is work-conserving and efficient, there is no support for bandwidth allocation and heavy traffic starves all but the real-time requests so it does not provide isolation.

The various mixed-media schedulers presented provide different advantages and disadvantages. This overview covers a few of them in reference to the requirements presented in chapter 3 on page 11, and provides information about the most basic functionality of the mixed-media schedulers.

4.5 Suitability of Existing Schedulers

A comparison[25] of the mixed-media schedulers examined in this section to the requirements outlined in chapter 3 on page 11, shows that none of the preexisting schedulers fully provided the requirements. An overview of the schedulers and the requirements they support can be seen in table 4.1. The only two schedulers that partially support all five requirements are Cello and

MARS.

Aside from APEX, the mixed-media schedulers presented do not fully support the requirements. Cello and MARS are the two of the others that best support the requirements, but there are some shortcomings.

4.6 Summary

The early algorithms, both efficiency-oriented and deadline-aware, are not suitable for used in mixed-media environments. The efficiency-oriented algorithms do not take into account process-specific requirements. The deadline-aware algorithms do not provide enough efficiency, and do not redistribute bandwidth when there is no real-time traffic.

The mixed-media schedulers present several improvements. APEX, Cello and MARS best suit our requirements. Cello requires very detailed knowledge of the block device, which is not suitable for a general use OS scenario. MARS does not support multiple service types, and is only partially work-conserving. Therefore, APEX is the best choice for implementing a mixed-media scheduler in a general-purpose OS kernel.

Chapter 5

Examination of Linux I/O scheduling

This chapter provides an overview of the Linux block I/O architecture with focus on the elevator subsystem. It shows that the Linux kernel implements several I/O scheduling algorithms, known as noop, deadline, anticipatory and CFQ scheduling. Of these, only CFQ is suitable for MMDBMS and other real-time systems. The overview begins with a quick introduction to Linux, continued by a look at the block device interface, then examining each specific scheduler, and finally this is tied together with the system call interface the user sees.

5.1 Linux suitability

When choosing the OS for use in testing, it is natural to choose one of the more common open source OSes. Of the open source Unixes, Linux is the only to ship either a loadable elevator subsystem or a scheduler with per process priorities by default. While a set of patches providing a pluggable scheduler framework for FreeBSD[1] exists, it is not included in the mainline code nor is it updated for the most recent versions.

It is worth noting that the Linux elevator framework inherently supports the two-level hierarchy most mixed-media schedulers specify. All the Linux schedulers provide the high level, often per process, optimization. The elevator provides two API functions for schedulers to add requests to the dispatch queue. One of these is `elv_dispatch_sort`, which inserts the request in SCAN sorted order. The other is `elv_dispatch_add_tail` which appends the request to the end of the dispatch queue. The scheduler may therefore

[1] `http://www.freebsd.org`

choose whether to perform its own ordering, or whether to defer this to the elevator. Using TCQ or NCQ, the disk driver may then further optimize the requests.

Since Linux already has a loadable elevator framework, only small modifications are necessary to implement a new I/O scheduler. Some data structures must be declared at compile-time. In addition, the system call interface for reserving resources must be added. Adding new system calls to Linux is fairly simple, requiring only adding a system call number along with the function itself.

The open nature and active development of the Linux kernel, in particular with reference to I/O scheduling, makes it the preferred candidate for testing of a new I/O scheduler.

5.2 The Linux Kernel

The Linux kernel was begun as a hobby project in 1991 by Linus Torvalds when he desired a UNIX-like system to work on at home. It was released under the GNU public license, which enabled anyone to read the source code and make alterations. Today Linux is fully POSIX-compliant and aims for Single UNIX Specification compliance. Linus Torvalds continues to lead the Linux kernel development, acting as a supervisor and making the final call on which patches make it into the kernel mainline. Several dozen kernel developers, both volunteers and paid employees of various companies, regularly contribute patches and input on the various kernel subsystems. In total the number of credited contributors to the Linux kernel is in the hundreds[47]. Several OS distributions using the Linux kernel exist, such as Redhat[2], Debian[3] and Slackware[4].

The Linux kernel (hereafter the kernel or the Linux kernel) is a monolithic kernel with the ability to load object code (*modules*) into kernel memory at run time. While written in C with some assembler extensions, it implements several object oriented interfaces. This modularity makes extension of the kernel require little effort. However, since the kernel is monolithic, bugs in modules can be just as fatal as any other bugs in the kernel.

In addition to providing a system call application programming interface (API) exported to user-land, the kernel maintains an internal API of calls subsystems may make to each other. Unlike the system call API, which guarantees that once a call is implemented its visible behavior will never

[2]`http://www.redhat.com`

[3]`http://www.debian.org`

[4]`http://www.slackware.com`

change and that it will exist in all subsequent kernel versions, the kernel API changes continually. This sort of incremental improvement is typical of Linux development.

The kernel is, as of this writing, in development of version 2.6. Traditionally even numbered minor versions were the stable versions of the kernel, such as 2.2 and 2.4, with odd version numbers denoting a development version. After version 2.5 was declared stable and incremented to begin the 2.6 series, Torvalds decided that since the 2.5 kernel had for long periods of its development included superior features to the 2.4 kernel while being arguably as stable, this arrangement should be abandoned. Since then features have been incrementally added to the 2.6 kernel, incrementing a third version number for each stable release. In addition a fourth version number has been added for security patches. Currently only the newest stable version of the kernel is maintained, in addition to a branch known as the stable branch, which has its own team. This writing assumes the changes present in the 2.6.19 version of the kernel, which includes changes to the elevator subsystem that make it even more well-suited for expansion than previous.

The elevator subsystem is a good example of how the kernel provides object oriented interfaces to the higher levels that hide implementation details. Part of the focus of kernel development during the 2.5 cycle focused on improving the I/O scheduling. In this period the old Linus Elevator was phased out and replaced by a series of new schedulers. Improvement upon these, in particular the CFQ scheduler, has continued throughout 2.6 development. An overview of the schedulers available just prior to the release of the 2.6 kernel is available in [4].

While the kernel is primarily written in C, there are some special limits when programming in the kernel code-space compared to userspace. Foremost, the standard libraries are not available. Many of the important standard library functions have been recreated internally. The other important limitation is that using floating point operations is difficult enough that it is in practice avoided. This is because the kernel must normally catch a trap and do something (where something is an architecture dependant action) in response to a userspace program wishing to perform floating point instructions. Since the kernel cannot trap itself, using floating point in the kernel requires manually handling the various situations that arise.

The kernel keeps time using a unit known as *jiffies*, the length of which is defined at compile time. The value *HZ*, defines the number of jiffies per second. The variable `jiffies` is updated every time the clock interrupt is sent to the kernel and holds the current time since boot in jiffies.

Internally the Linux kernel does not differentiate between processes and threads, in the sense that a thread is merely a special kind of process. Instead the unit of execution is universally referred to as a task. Unless otherwise noted, these three terms are used interchangeably.

Overall, the Linux kernel provides a mature, stable development environment in which modern techniques can be tested. This chapter continues with examinations of the kernel internals that interact with the scheduler subsystem.

5.3 The Generic Block Device Layer

The kernel API interface for reading and writing to block devices is wholly provided by the source-file `block/ll_rw_blk.c`. Each block device driver must register a queue by calling `blk_init_queue` which returns a `request_queue_t` (which is a synonym for `struct request_queue`). This call registers a scheduler with the queue by making a call to `elevator_init_queue`.

The queue registers a few function pointers to determine heuristics for treating the queue. The defaults for these are found in `ll_rw_blk.c`, but the driver may choose to override them. These functions primarily concern device plugging and request merging, both of which are examined below.

5.3.1 Buffer Heads

To keep track of the relationship between buffered blocks in memory and on disk, `struct buffer_head` objects are created. Buffer heads were once also the unit of I/O in the filesystem and block layer. It has been replaced for this purpose, but buffer heads are still used for various other purposes. The comments preceding their definition in `include/linux/buffer_head.h` states that buffer heads are used for extracting block mapping, tracking state within a page and for wrapping block I/O submissions for backward compatibility. The latter is provided for filesystems that still use buffer heads rather than block I/O structures.

5.3.2 Block I/O Units

Each I/O request is contained in an instance of `struct bio`. This structure holds information about operations that are active. It holds a list of segments (continuous blocks of memory) which the data is copied to or from, information on whether the request is a read or write and what device the request concerns. The bio structure is designed to be easy and quick to perform split and merge operations on.

The block device layer defines sectors to be units of 512 bytes, regardless of what sector size the hardware device uses. It is the responsibility of the device driver to translate this.

Information about buffers in memory that the bio's I/O is copied to is stored in a list of `struct bio_vec` objects. This structure holds a pointer to the page, the length in bytes in represents and the offset in that page that the buffer resides.

5.3.3 Requests

The unit that Linux I/O schedulers deal with is the `struct request`, which tracks several block I/O units. The structure is defined in `include/linux/blkdev.h`. The information a request holds that an I/O scheduler uses include fields for inserting the request in lists. The I/O scheduler can also access the request's flags, which hold information about the request type and status. The request also holds the address of the request on the block device and the number of sectors the request covers. In addition the structure provides two void pointers for exclusive use by the scheduler.

The structural relationship between request and bio objects is shown in figure 5.1 on the next page.

While most literature on I/O scheduling deals with specific sector or block accessing, one can apply most conclusions from these, since a request represents contiguous sectors. A caveat here is that these sectors are only logically contiguous, and the block device may have remapped them. In this case the time spent by the request is unpredictable at best. However, since the same problems would arise in systems that handle sectors or blocks individually, it is safe to ignore them.

5.3.4 Plugging

When accessing block devices, it can be better to wait until multiple requests are ready, so that they can be SCAN sorted. The kernel block layer implements this with plugging and unplugging. When the device has no pending requests, it is plugged in a manner similar to a sink or a bath tub. When a single request arrives, the kernel checks whether the request is synchronous or asynchronous. An asynchronous request need not be serviced immediately, so the device remains plugged until more requests arrive or a short amount of time passes. Hopefully more requests will arrive, so that the requests can be serviced in optimized order. Since related synchronous requests by their nature cannot accumulate, these immediately unplug the block device.

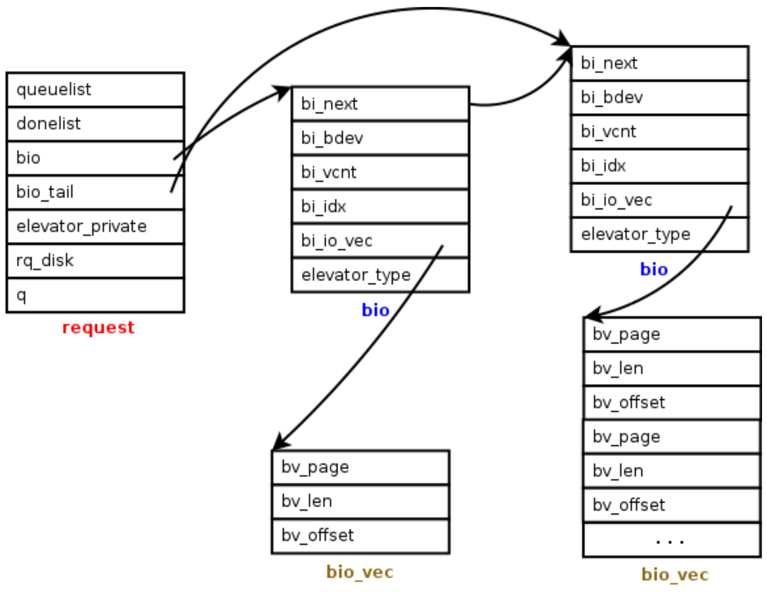

Figure 5.1: The relationship between request, bio and bio_vec structures[23].

5.4 Performing a Read Operation on Linux

This section provides an overview of the system call interface for performing synchronous requests to block devices, along with an in-depth look at how these requests are forwarded to the elevator layer. This section is based on various sources[8, 24, 23] along with the 2.6.19 version of the Linux kernel source[47].

The Linux kernel provides two different ways of performing block I/O from userspace. One is through the standard read and write system calls. The other is by mapping the file into memory through the mmap system call and reading from memory, which causes page faults. The standard read/write interface can be used in two ways, either for buffered or unbuffered (direct) I/O. The full call path of these two variations of the standard I/O interface for reads are examined. The call path for writes is very similar, with a slightly different entry point.

Linux system calls are performed using the system call handler. Each separate system call is denoted by a system call number, which is passed to the kernel when the system call handler is invoked. On the x86 architecture the system call handler is invoked using the interrupt instruction. On other architectures, similar provisions exist.

This section examines only the execution path from the `read` system call,

Function	**Source file, Line Nr.**
`sys_read`	`fs/read_write.c`, 356
`vfs_read`	`fs/read_write.c`, 255
`do_sync_read`	`fs/read_write.c`, 230
`generic_file_aio_read`	`mm/filemap.c`, 1140
`do_generic_file_read`	`include/linux/fs.h`, 1795
`do_generic_mapping_read`	`mm/filemap.c`, 877
`mpage_readpage`	`fs/mpage.c`, 427
`mpage_readpages`	`fs/mpage.c`, 386
`block_read_full_page`	`fs/buffer.c`, 1917
`page_cache_readahead`	`mm/readahead.c`, 464
`blockable_page_cache_readahead`	`mm/readahead.c`, 400
`get_block_t` (typedef)	`include/linux/fs.h`, 302
`do_mpage_readpage`	`fs/mpage.c`, 176
`submit_bh`	`fs/buffer.c`, 2640
`submit_bio`	`block/ll_rw_block.c`, 3106
`generic_make_request`	`block/ll_rw_block.c`, 2998
`__make_request`	`block/ll_rw_block.c`, 2832
`get_request_wait`	`block/ll_rw_block.c`, 2172
`get_request`	`block/ll_rw_block.c`, 2068
`blk_alloc_request`	`block/ll_rw_block.c`, 1971
`elv_set_request`	`block/elevator.c`, 797
`add_request`	`block/ll_rw_block.c`, 2608
`__elv_add_request`	`block/elevator.c`, 611

Table 5.1: The location in the source of various kernel API functions common for buffered I/O, as found in the modified version of the 2.6.19 kernel.

since this is representative of synchronous requests. The `mmap` type of I/O is not examined explicitly, since it is inappropriate for sequential access to large files. It differs from ordinary I/O in that requests to read are performed implicitly, by invoking page faults, rather than explicitly requesting positions in files. The pattern of access is similar enough that the important limitations presented by ordinary reads still hold.

5.4.1 Buffered I/O

The system call to read from a file is `sys_read`. It takes as arguments a file descriptor, a userspace address to a byte buffer and an integer denoting

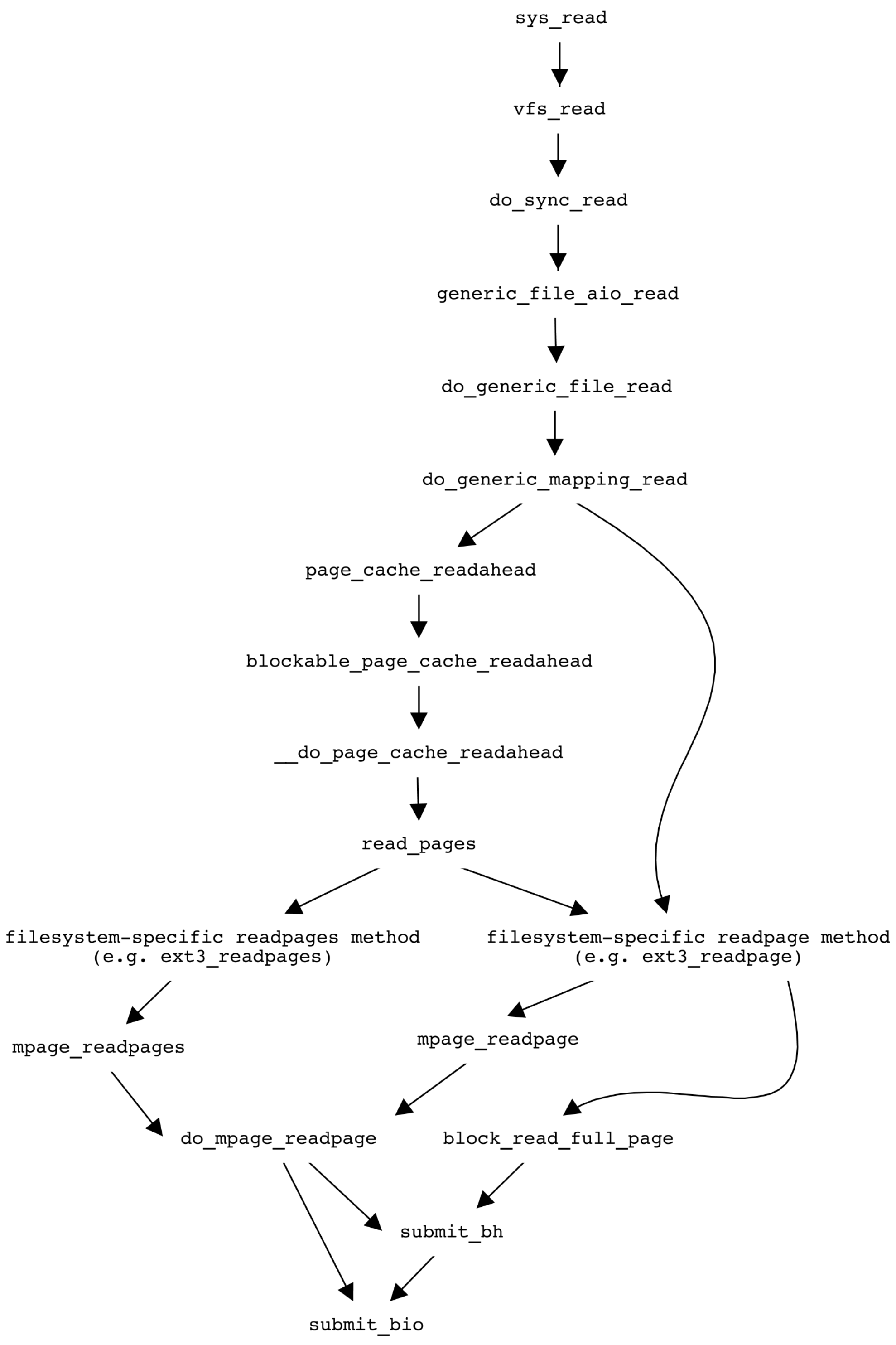

Figure 5.2: The call graph from `sys_read` to `submit_bio`.

Struct	**Source file, Line Nr.**
`struct file`	`include/linux/fs.h`, 720
`struct file_operations`	`include/linux/fs.h`, 1108
`struct kiocb`	`include/linux/aio.h`, 87
`struct address_space`	`include/linux/fs.h`, 430
`struct file_ra_state`	`include/linux/fs.h`, 705
`struct page`	`include/linux/mm_types.h`, 18
`struct buffer_head`	`include/linux/buffer_head.h`, 59
`struct bio`	`include/linux/bio.h`, 72
`struct request_ queue (request_queue_t)`	`include/linux/blkdev.h`, 386
`struct request`	`include/linux/blkdev.h`, 249
`struct request_list`	`include/linux/blkdev.h`, 151

Table 5.2: The location in the source of various kernel API structure definitions common for buffered I/O, as found in the modified version of the 2.6.19 kernel.

the number of bytes to read. The file descriptor is an integer that provides a lookup into a table of open files for the current user. The call returns an integer denoting the number of bytes that were actually read, or a negative value if there is an error. The order of the functions called is shown in figure 5.2 on the facing page. A list of the functions and their location in the source is shown in table 5.1 on page 33. Data structures and their locations are shown in table 5.2.

Using the file descriptor, the kernel retrieves the `struct file` for the open file. The `struct file` instance specifies which operations are possible on the open file. The instance also holds a `struct file_operations` which stores function pointers for functions specified for various operations on the file, such as reading, writing, opening, flushing and so on. Not all of these are always defined. The function call of interest is the read function. In the normal case, a standard file open for sequential access, this is `do_sync_read`. If the open file is a pipe or similar special construct, the read function is a pointer to a different function.

The file instance is then passed on to the virtual filesystem (VFS) layer using the `vfs_read` function. The VFS layer is an abstraction that provides a unified interface to all filesystems and block devices. It hides the underlying implementation of the filesystem and specifics of the block device from the userspace programs, so they function regardless of the underlying details. The can be seen in figure 5.3 on the next page. The arguments to `vfs_read`

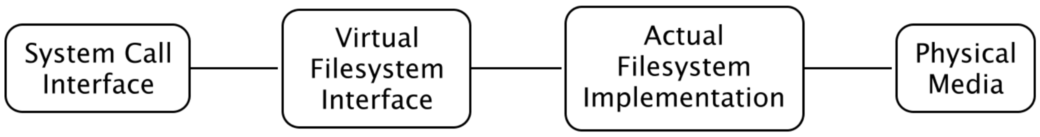
System Call Interface
Virtual Filesystem Interface
Actual Filesystem Implementation
Physical Media

Figure 5.3: The flow of information between the system call interface and the physical media goes via the virtual filesystem before the actual filesystem sees it.

are the same as `sys_read`, in addition to the position variable.

Once the operation is verified as legal, in that the user has correct permission to the file, that the buffer exists, that the requested position is within the file and so on, `vfs_read` either forwards the call to the read operation specified in the `struct file` instance's own read operation or calls `do_sync_read`. In most cases, these two are the same.

The function `do_sync_read` takes all the same arguments as `vfs_read`. At this point, the kernel creates an I/O control block (IOCB) to handle the request. This IOCB is represented by a `struct kiocb`. While kernel IOCBs are ostensibly for asynchronous I/O, the kernel may request that they be handled synchronously. This is done in `do_sync_read`. The kernel then attempts to perform an asynchronous read operation. If this returns stating that the operation has been queued, `do_sync_read` performs a wait operation until a callback reactivates it.

The asynchronous read operation is determined in the same way as the synchronous read operation, except that there is no default value. The filesystem must implement an asynchronous read operation, and the file object must have a valid function pointer stored for it. By default, and for most filesystems[5], this points to `generic_file_aio_read`. At this point the execution branches for requests that have been marked to bypass caches. The direct I/O path is examined in 5.4.2 on page 40, this section continues to examine buffered I/O.

The path of execution for a buffered read continues to `do_generic_file_read`, which reformats the arguments and forwards the call to `do_generic_mapping_read`. The arguments to this function are the file object, the position in the file, a read descriptor and a function pointer to a function that will copy the read data back to the userspace buffer. In addition, it takes a `struct address_space` and `struct file_ra_state` (file read-ahead state), but both of these are passed from the file object by `do_generic_file_read`.

The address space struct is part of the kernel's caching mechanism. An

[5]That is, all filesystems that can use the page cache directly.

address space is tied to an inode. A classical UNIX filesystem element, the inode is a descriptor that keeps track of accounting information about exactly one file[45]. The address space is the basic unit of the page cache. It holds the mapping from block devices to physical pages. This can represent any page-based object, be it a file or memory mapping of some sort.

When a page is not cached, it is read by the filesystem using the `readpage` operation in the address space's operations structure. This function takes two arguments, the file object being read and the `struct page` that specifies the location in memory to put the data that is read. It is ostensibly filesystem specific, but many filesystems simply forward this call to generic page reading functions.

It is worth noting that the reading happens in part via the read-ahead mechanisms in the kernel. This is invoked in `do_generic_mapping_read`, by way of the `page_cache_readahead` function. Thus a short digression here to explore the execution path of a read-ahead call is called for. The function `page_cache_readahead` controls the read-ahead window size management and submits read-ahead I/O. The read-ahead I/O submission is sent to `blockable_page_cache_readahead`, which in turn forwards the call to `__do_page_cache_readahead`. This function then allocates the amount of pages necessary to perform the I/O, then calls `read_pages` to perform the actual I/O. This function in turn calls the address space's relevant `readpages` operation, if the filesystem implements this function. Otherwise multiple calls to `readpage` are made.

The `readpages` function is similar in structure to the `readpage` call. It takes as its arguments a file object, an address space, a list of pages and the number of pages in that list. Most filesystems that implement `readpages` simply forward the call to `mpage_readpages`. Those that do not are typically networking filesystems or other specialized filesystems. Standard block device filesystems define `readpage` to either `mpage_readpage` or `block_read_full_page`.

None of the functions `mpage_readpages`, `mpage_readpage` and `block_read_full_page` accept a file object as an argument. This presents a problem when implementing APEX which is examined in detail in 6.4.1 on page 74. In addition to page information, the three functions all take a function pointer of type `get_block_t` as arguments. This function is filesystem specific and provides a translation from inode and file offset to block number on the relevant device.

The simplest of these three functions is `block_read_full_page`, which uses the `get_block_t` function to construct a `struct buffer_head` for I/O. This list is then passed to `submit_bh`, which does little more than translate the buffer head to the newer bio, represented by a `struct bio`. This is then passed to submit_bio. Both submit_bh and submit_bio accept two arguments, an integer designating the type of operation along with the buffer head and block I/O object, respectively. This point of execution is returned to later.

The functions mpage_readpage and mpage_readpages are very similar. Both construct struct bio objects for submission by calls to do_mpage_readpage. mpage_readpages differs by making multiple calls to create one large bio object, whereas mpage_readpage makes only a single call. The function do_mpage_readpage maps the disk blocks and creates the largest possible bio for submission. If it encounters any problems, it falls back on the buffer head-based read function, block_read_full_pages (see above). Problems that can cause this are a page which has buffers, a page which has a hole followed by a non-hole or a page which has non-contiguous blocks. These must be treated by the legacy code which handles buffer heads. Either way, the code path eventually results in one or more calls to submit_bio.

When a bio is set up and ready for I/O, it is sent to submit_bio. From this point and downwards no code uses buffer heads. This is part of the generic block device interface. The task of submit_bio is to prepare requests for the block device, but it defers this work to generic_make_request. See figure 5.4 on the facing page for an overview of call path from submit_bio and down to the elevator.

Like filesystems, individual block device drivers may register their own functions for handling various situations. One of these situations is generating requests. The struct request_queue_t holds the block device's queue, and specifies the function for this in make_request_fn. The default for this, which is used by standard block devices, such as physical IDE, SATA or SCSI disks, is __make_request.

The struct bio is sent from submit_bio to generic_make_request as the only argument to this function. The bio is then used to find the correct request_queue_t for the block device it refers to. Various checks are performed to ensure that the bio refers to legal areas of the disk, that the disk exists, that the bio is not too large for the device and so on. Finally, the bio is sent to the block device driver's preferred make_request_fn. This function takes as its parameters the request queue for the block device and the bio.

Standard block devices use the generic request generation function, __make_request. This function prepares the bio for the elevator. It first attempts to merge the bio into a preexisting request by calling elv_try_merge. If there are no preexisting requests, or no viable candidates for a merge, it calls get_request_wait to allocate resources for a new struct request.

The get_request_wait first attempts to allocate a request through get_

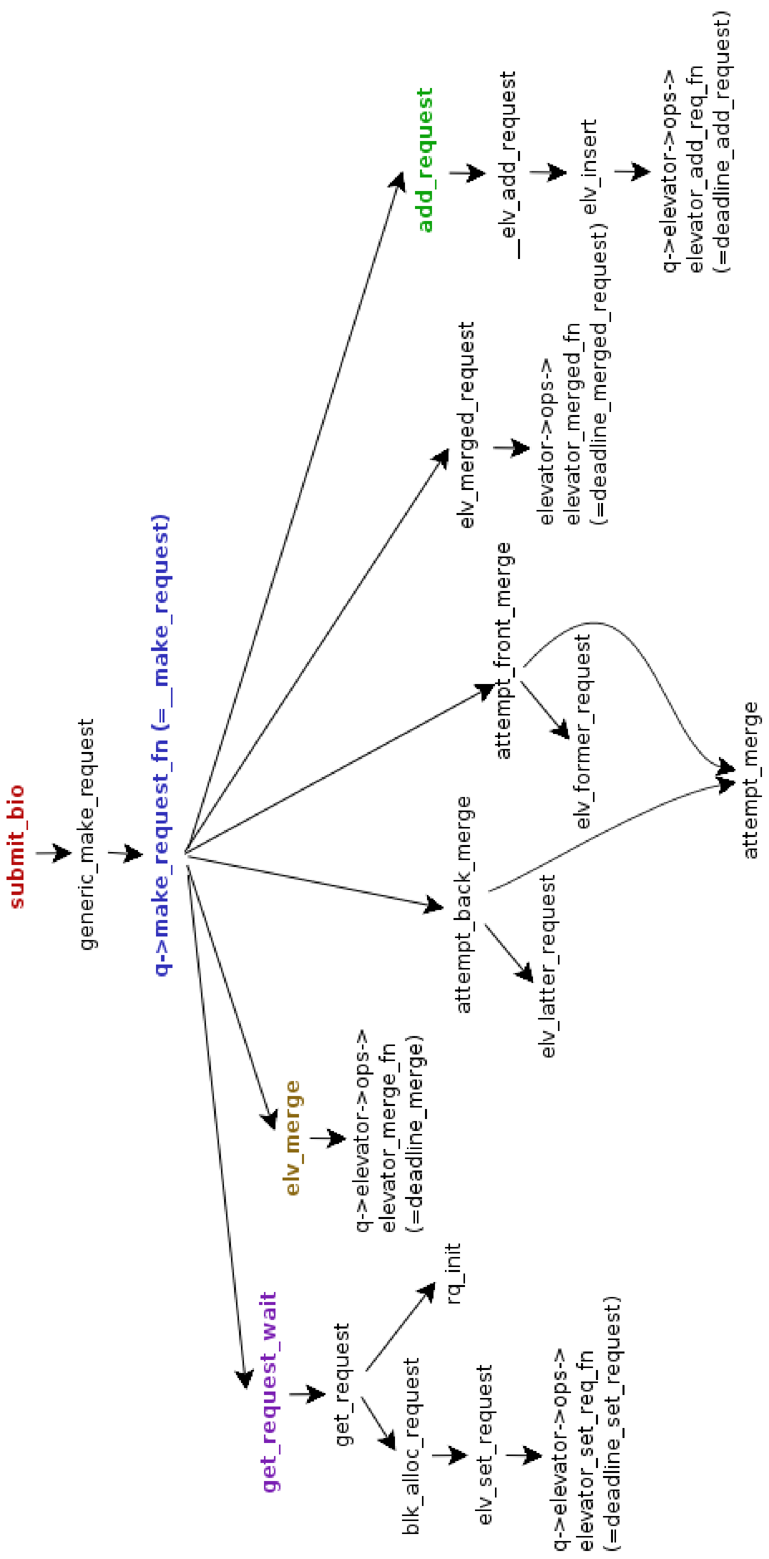

Figure 5.4: The call graph for request submission[23].

request. If resource allocation fails, it is either because the queue is full (congested), or because there is not enough memory available. Either way, the task calls the CPU scheduler to sleep until the struct request_list has resources freed. The struct request_list is a member of the request_queue_t that holds the cache used to allocate requests. If the process sleeps, it is granted special access to allocate requests to compensate for having deferred once awoken.

A successful call to get_request involves being allowed to allocate a new request by the request_list, then making a call to blk_alloc_request to perform the actual allocation from the cache. In addition to allocating memory for the struct request, the elevator may need to allocate some resources for the request. This is done by making a call to elv_set_request. If both these calls are successful, the request is allocated and control returns to __make_request.

Since the bio is not mergable, the request must now be added to the elevator for sorting and eventual dispatch. This is done by add_request, which performs some device driver related housekeeping, then calls __elv_add_request. In the normal case, this forwards the request to elv_insert which uses the scheduler's custom elevator_add_request_fn to perform a sorted insertion. Exceptions occur when special barrier requests are added, but for the vast majority of requests this is the execution path followed. The request is now in the scheduler, and will be dispatched to I/O when it is deemed appropriate.

From here, the process context returns up the call stack to do_sync_read, where it waits until the I/O is completed. An asynchronous request returns all the way to userspace.

This is how the buffered synchronous I/O request from userspace is processed, going from the high level system call, through the virtual filesystem, actual filesystem, to the block device layer and finally to the elevator. The access is turned into a kernel I/O control block, which is broken up into pages so that they can be accessed through the cache, and then rebuilt into buffer heads by the filesystem. The buffer heads are then turned into bios for handling by the block device layer. These are organized into requests, which are ordered by the elevator for dispatch. The elevator's functions are covered in section 5.5 on page 42.

5.4.2 Direct I/O

The previous section examines the execution path of buffered block device accesses. Some programs do not benefit from the kernel cache, or benefit more from managing their own cache. For these programs, direct I/O offers

Function	**Source file, Line Nr.**
`generic_file_direct_IO`	`mm/filemap.c`, 2393
`ext3_direct_IO`	`fs/ext3/inode.c`, 1619
`ext3_get_block`	`fs/ext3/inode.c`, 947
`blockdev_direct_IO`	`include/linux/fs.h`, 1819
`__blockdev_direct_IO`	`fs/direct-io.c`, 1180
`direct_io_worker`	`fs/direct-io.c`, 950
`do_direct_IO`	`fs/direct-io.c`, 796
`dio_bio_submit`	`fs/direct-io.c`, 351
Struct	**Source file, Line Nr.**
`struct dio`	`fs/direct-io.c`, 64

Table 5.3: The location in the source of various kernel API functions and structure declarations related to direct I/O, as found in the modified version of the 2.6.19 kernel.

a way to bypass the kernel's caching mechanisms. It also offers zero-copy I/O. This type of I/O improves performance by allowing the block device to copy directly to userspace memory, thus freeing the CPU from performing this task in the kernel, as would be the case in buffered I/O. The functions and data structures used in direct I/O are listed in table 5.3.

Direct I/O is used in particular by DBMS applications, under the reasoning that the atypical pattern of essentially random accesses to large files contradicts assumptions made in kernel read/write caching strategies. They also require better certainty that a write operation has been completed when the kernel reports it has. Write operations are normally asynchronous, meaning that processes can queue write operations and continue execution in parallel. For DBMS systems, it is often more important to guarantee data integrity. Direct I/O assists in this by returning synchronously from write calls only after the command has been committed to the hardware block device.

To perform direct I/O on a file in Linux, the application must open the file using the `open` system call modified by the `O_DIRECT` flag. On Linux, this requires that all operations are aligned to 512-byte boundaries. It is also possible to perform direct I/O directly on a block device. To perform raw direct I/O on a block device, the `open` system call is used with the device node as the file. For example, `open(‘‘/dev/hda’’, O_DIRECT)`, which opens the hard disk drive `hda`.

The execution path of a read on an open file object with the direct I/O flag enabled is similar to of a buffered until the `generic_file_aio_read` function. This function checks whether the direct I/O flag is enabled, and invokes the generic_file_direct_IO function to perform the read, rather than do_generic_file_read. This invokes the specific filesystem's own direct I/O call. The filesystem's direct I/O call takes the operation as an argument, and as such is common for both read and write operations. For the ext3 filesystem, this function is ext3_direct_IO, which in turn forwards the call to blockdev_direct_IO which forwards the call to __blockdev_direct_IO.

There are 10 arguments passed to __blockdev_direct_IO, including the kernel I/O control block for the access, the inode for the file being accessed, the struct block_device representing the block device being accessed and the filesystem's get_block_t function for direct I/O, which is ext3_direct_io_get_blocks. In __blockdev_direct_IO, some of these parameters are used to construct a struct dio, which represents the various stages of a direct I/O access. If arguments are valid and resource allocation is successful, the struct dio and some other arguments are forwarded to direct_io_worker.

The direct_io_worker function takes many of the same arguments as __blockdev_direct_IO and performs the main work for the direct I/O operation. It stores the current state of the call in the struct dio and calls do_direct_IO with the struct dio. This function prepares the struct dio for execution by mapping pages and device blocks to each other. This function has various helper functions, but in the end returns to direct_io_worker, which constructs block I/O units for the operation, and sends the dio object to dio_bio_submit. This function performs some accounting and then calls submit_bio.

5.5 The Linux Elevator Interface

The Linux kernel differentiates between the elevator and individual schedulers. The elevator refers to the whole request ordering subsystem, whereas a scheduler is the specific algorithm that decides request ordering. The elevator interface is provided by block/elevator.c. This file defines a set of interface functions which do minimal amounts of bookkeeping, forwarding almost all calls to relevant queue's scheduler. As of kernel version 2.6.19 it also handles back merges.

Since the kernel API is continually changing, the information given here may not be accurate for versions other than 2.6.19. While the kernel API is reasonably stable, there are no guarantees provided that exported symbols will remain available in later versions.

```
# cat /sys/block/hda/queue/scheduler
noop anticipatory deadline [cfq] apex
```

Figure 5.5: Reading the available and active schedulers.

```
# echo anticipatory > /sys/block/hda/queue/scheduler
# cat /sys/block/hda/queue/scheduler
noop [anticipatory] deadline cfq
```

Figure 5.6: Changing the scheduler on a block device.

5.5.1 Switching Scheduler on a Block Device

The generic nature of the elevator implementation enables switching the active scheduler on a block device at run-time. This is done using the sysfs interface. The sysfs interface is designed as a special filesystem presented as a tree hierarchy under the `/sys` directory. The `block` subdirectory covers block device specific variables, it holds one subdirectory per mounted block device on the system. These block devices are represented by their name under the `/dev` hierarchy. IDE disks are commonly represented by names prefixed with 'hd', whereas SCSI disks are commonly prefixed with 'sd'. For each block device that implements a queue, there is a directory `queue`. This directory holds a special file `scheduler` that contains information about the scheduler active on that block device.

To see a list of available schedulers, and the currently active scheduler, one reads the contents of the special `scheduler` file. Figure 5.5 shows an example of reading the list of available and active schedulers on the block device `hda`. The active scheduler is surrounded by square brackets, in this case it is `cfq`.

To set the scheduler, one writes the name of the desired scheduler to the file. An example is shown in figure 5.6. In this example the scheduler on `hda` is changed to Anticipatory.

5.5.2 The Elevator Interface Provided to Schedulers

The kernel elevator provides an interface to schedulers for handling requests and registering schedulers for use. These functions primarily cover bookkeeping, but two fundamental calls are provided. These are `elv_dispatch_add_tail` and `elv_dispatch_sort`. These two take requests and add them to a block device's dispatch queue. As the names imply, one appends the request to the end of the dispatch queue, while the other performs a sorted insertion. The appending version is provided for schedulers that perform strict SCAN ordered dispatch, such as the Deadline scheduler.

The other functions provided perform tasks such as registering or unregistering the scheduler for use, removing requests from the red-black tree used for merging or querying the red-black tree to find logically next or last requests. In addition, the scheduler has to perform reference counting on `struct io_context` via an interface provided by the elevator.

5.5.3 The Scheduler Interface Provided to the Block Layer

The elevator interface provides several function calls that the scheduler may implement. Some of these are obligatory. This overview is based on the specification provided in the kernel documentation[6], the elevator interface code itself, the block layer code that calls the elevator and the various schedulers that implement it.

- `elevator_merge_fn` is used to query the scheduler whether there are any requests appropriate for merge with a given bio. A scheduler must implement this if it wishes to implement front merges.
- `elevator_merge_req_fn` is called when two requests are merged. One request will not be seen again by the scheduler, so special resources allocated must be freed. This includes removing the request from any lists it has been added to as well as any dynamically allocated data structures.
- `elevator_merged_fn` is called when a request has been involved in a merge. This may be of interest if book keeping variables must be updated, lists resorted or similar.
- `elevator_dispatch_fn` is called to move ready requests into the dispatch queue. An argument is given that allows schedulers to defer moving requests to dispatch until later. Once dispatched, requests belong to the dispatch queue and the scheduler may no longer use them.
- `elevator_add_req_fn` adds a new request to the scheduler.
- `elevator_queue_empty_fn` checks whether the queue is empty. This is provided to the block layer for merging purposes.

- `elevator_former_req_fn` returns the request before the given request in SCAN sorted order. It is used by the block layer for merging purposes.

- `elevator_latter_req_fn` similarly returns the request after the given request in SCAN sorted order. Both these requests are implemented by a common elevator function after kernel 2.6.19.

- `elevator_completed_req_fn` is called when a request has been serviced by the hardware device. a

- `elevator_may_queue_fn` is used to allow a process to queue a new request even if the queue is congested.

- `elevator_set_req_fn` allocates the scheduler specific data for a request.

- `elevator_put_req_fn` frees the scheduler specific data for a request.

- `elevator_activate_req_fn` is called by the device driver when actual execution of a request starts.

- `elevator_deactivate_req_fn` is called by the device driver when it has decided to delay a request by requeueing it. A subsequent call to `elevator_activate_req_fn` will be made when it is again being serviced.

- `elevator_init_fn` is called to allocate scheduler specific data structures when the scheduler is activated for a device.

- `elevator_exit_fn` is called to free scheduler specific data structures when the scheduler is deactivated for a device.

- `trim` this undocumented method is used to clean up data structures in the block layer when a scheduler is unloaded from kernel memory.

5.5.4 Merging

To lessen overhead, the kernel may merge two sequential requests into a single, longer request. The two forms of merging are known as front merging and back merging. A front merge occurs when a new request arrives that can be added to the front of an existing request. This is rare and in most cases does not justify the overhead of testing every existing request for viability.

The more common case is back merging. Back merging is when a new request may be appended to an existing request. This case is so common and merging so beneficial that as of kernel version 2.6.19 the elevator layer implements this generically. Previous to this version, all schedulers bar the noop scheduler had their own implementations of back merging.

The implementation of back merging is done by keeping all existing requests in a red-black binary tree sorted on end sector. When a new request arrives, the elevator layer searches this red-black tree for an appropriate existing request. If one is found, a back merge is performed.

It is worth noting that unrelated merges are permitted. This might occasionally cause problems in scheduler accounting where process specific queues are maintained, such as CFQ and APEX.

In addition to reducing computational overhead, increasing request sizes improves throughput[32]. The request size is in practice limited by the block device.

5.5.5 Synchronous, Asynchronous and Metadata Prioritization

While much literature glosses over the differentiation of synchronous and asynchronous requests, it is a much more pressing issue in actual schedulers. Part of the reason for the rewrite of the block layer in the 2.5 kernel was due to asynchronous requests starving synchronous requests. An asynchronous request is a request that the issuing process does not need to block and wait for completion on. Most commonly these are write requests, since programs usually do not need to wait until a write request has been committed to disk to continue execution. The opposite is true for reads, which programs most often are dependent on to continue. (Most commonly to perform some sort of manipulation on the data read.) Consequently, the kernel developers refer to asynchronous requests starving synchronous requests as writes starving reads. In some cases this may be inaccurate, for example in DBMSes which require knowledge that writes have been committed to disk for integrity reasons. Unless otherwise specified reads and writes may be assumed to mean synchronous and asynchronous requests respectively.

All the new schedulers bar Noop implement differentiation between synchronous and asynchronous requests. This differentiation takes the form of prioritizing synchronous requests. For example, an asynchronous request will not preempt a synchronous request in CFQ. The deadline scheduler by default provides much longer timeouts for asynchronous requests than synchronous.

As of 2.6.19, the CFQ scheduler provides an extra level of differentiation, for requests for filesystem metadata. These requests are usually small, but precede larger I/O requests. Therefore they are cheap to prioritize, yet provide more efficient I/O by causing a process to be able to read when its round arrives instead of having to use a time-slice to read the metadata. Currently only the ext3, ext4 and gfs2 filesystems implement the metadata flag. Since CFQ is currently the default scheduler, more filesystems can be expected to implement this in the future. Prioritization of metadata reads is important, since both read and write requests depend on them[24].

Results relating to asynchronous starvation of synchronous requests have been presented[21]. These results have been applied to the Linux kernel and are used in the Anticipatory and CFQ schedulers. This is examined further in the next section.

5.6 The Linux I/O Schedulers

This section contains an overview of the Linux I/O schedulers, from the Linus Elevator used in the 2.4 version of the kernel, to the most recently implemented CFQ. These are loadable and unloadable at run-time, and have varying advantages and disadvantages.

5.6.1 Linus Elevator

The Linus Elevator was the only I/O scheduler available in the 2.4 version of the kernel. While it is no longer in use, it is examined here because it provided the basis for the Deadline scheduler, which in turn was used as the basis for the Anticipatory scheduler.

The Linus Elevator implements a C-Look algorithm with modifications to prevent process starvation. It performs both front and back merging. The request is then inserted into the queue in SCAN sorted order, unless a request is found that is older than a given threshold, in which case the new request is added to the tail of the queue. This is to prevent large sequences of requests on area of the disk from starving requests to other areas. This does not provide any guarantees, since the queue might already have very many requests prior to the request that has reached the end of its time limit.

Several weaknesses arise from this approach. In particular, it does not differentiate between synchronous and asynchronous requests. A long series of asynchronous requests can delay a few synchronous requests, causing applications to stall as they wait for disk access. This causes particularly bad problems when the system is swapping memory to and from the disk.

For these reasons the Linux Elevator was slated for replacement during the 2.5 kernel development. It was eventually replaced by a modular framework which supports multiple, loadable schedulers.

5.6.2 Noop

The Noop scheduler, defined in `block/noop-iosched.c` performs simple FCFS scheduling. Aside from the back merging performed by the elevator interface, the Noop scheduler itself does no ordering or treatment of requests. It maintains a single FIFO list, and delivers requests to the driver in the same order in which they arrived. This scheduler is designed for block devices in which access ordering does not make sense, such as flash storage devices or other random access devices, or very intelligent devices that can make better decisions than a kernel-space scheduler.

5.6.3 Deadline

The first new scheduler was designed as an iterative improvement on the Linus Elevator. The Deadline scheduler uses a C-Look queue in addition to two FIFO queues, one for synchronous requests and one for asynchronous. Requests in each FIFO queue are given expiration times, by default 500ms for reads and 5s for writes. Under normal conditions, requests are dispatched from the C-Look queue. However, if requests from either of the FIFO queues have expired, these are served instead.

The Deadline scheduler does not make any performance guarantees, contrary to what the name might imply. It instead provides some preference of reads over writes, which improves performance relative to the Linus Elevator and prevents writes from starving reads under some circumstances.

To guarantee that writes are not starved completely by reads, the scheduler defines a maximum number of times reads can be prioritized over writes when choosing from the FIFO queues. This ensures that the shorter expiration time of read requests does not completely prevent write requests from being serviced. The scheduler also attempts to fulfill requests from the C-Look queue when serving from the FIFO queues. This helps ensure that the scheduler does not degenerate to FCFS under high loads.

5.6.4 Anticipatory

The Anticipatory scheduler adds an *anticipation heuristic*, as proposed by Iyer[21] and the Linux kernel developers[44], to the Deadline scheduler. This heuristic attempts to solve the problem that arises when the Deadline scheduler has to deal with a small number of dependent reads while servicing a large stream of write requests. In this situation the reads are added to the queue and prioritized ahead of the writes. This may cause a long seek. The scheduler then resumes serving writes, causing a similarly long seek back to the writes. However, in most cases long sequences of reads are dependent, that is one cannot issue a new request before the last one is serviced. For example programs which loop over short reads, do some form of computation on the data and then perform a new read exhibit this sort of behavior.

To avoid this situation the anticipation heuristic is used to decide whether to wait for a new read request, or to continue processing requests. This wait is typically short, in the order of a few milliseconds. When new requests arrive, they are tested against the heuristic to see whether to continue waiting or begin dispatching the new requests. The heuristic considers factors such as which process the next request is from, how frequently a process has historically submitted requests or whether the next request is close geometrically on the platter. The scheduler records average think time, the wait between submission of requests, and the average seek distance of requests. The amount of time the scheduler waits is tunable and defaults to approximately 6.7ms $(1/150s)$.

To prevent reads and anticipatory delays from arbitrarily delaying write requests, the Anticipatory scheduler performs dispatching in batches, alternating between writing and reading. Both read and write batching is timed. Once the scheduler has spent more time than allotted serving a batch, it switches to serving the other type of requests. When serving a write batch the scheduler continually dispatches requests until the amount of time allotted for write batches expires. When serving a read batch the scheduler submits a single read at a time, and then uses the anticipation heuristic to decide whether to wait for a new request or continue dispatching.

Since requests are only chosen from the FIFO queues when serving the corresponding type of batch, Piggin[30] recommends that the FIFO expiration time associated with requests not exceed the expiration time associated with batch building. Since this would regularly cause requests to wait at least the time it takes to build the next batch (which is at least as long as the given batch expiration time), this follows naturally. The default expiration time of read batches is 500ms and write batches is 125ms.

When dispatching requests, if there are no expired requests on the FIFO queue, the Anticipatory scheduler chooses the request closest to the drive head's current position. The scheduler uses a VSCAN[13] heuristic when deciding how to define close in this context. This way the scheduler prefers to keep serving the queue in SCAN order, but seeks backwards if it is only a short distance. By default this distance is 1MB worth of sectors.

5.6.5 CFQ

Time-sliced completely fair queueing, *CFQ*, is a round-based scheduler that seeks to provide every process with at least some access to the disk. It is in many ways similar to APEX, but differs on certain key concepts, specifically deadlines and bandwidth allocation and implements rounds somewhat differently. The CFQ scheduler implements several service classes, for more on this see section 5.7 on the next page.

The CFQ scheduler provides each process with an exclusive queue. Whenever a process sends a request to the scheduler, it is placed in a FIFO list in the corresponding queue. Each queue, and therefore each process, is granted exclusive access to the disk for a short slice of time.

The CFQ scheduler is based on the assumption that all processes touch the disk occasionally[5]. Since it distributes disk time in equal time-slices, even processes which are fairly idle are granted reasonably prompt access to the disk. A single, I/O heavy process is unable to exclude requests from processes which send only a few requests, a common problem of SCAN based schedulers.

When called upon to dispatch requests, CFQ scheduler performs at least a partial round, in which it visits queues and activates each of their time-slices. As new requests arrive, a queue's time-slice may be preempted by the arrival of a request to a queue with higher priority. In addition, if a queue runs out of requests to serve while there is still time remaining in its slice, an anticipation heuristic is used to decide whether or not to end the slice and serve the next available queue. This heuristic is similar to that used in the Anticipatory scheduler. If a new request arrives to the active while it is waiting, it is immediately dispatched.

The exception to this is if an asynchronous request arrives while a queue is waiting. Since queues are only set to wait on the assumption that a synchronous request will arrive shortly, an asynchronous request implies that this assumption is wrong. The queue's time-slice is immediately expired and a new dispatch round is started.

As with the Anticipatory scheduler, the CFQ scheduler uses a VSCAN algorithm for determining which request to service out of a queue. For CFQ the default maximum backwards seek is 16MB.

$$time\_slice = base\_slice + \left(\frac{base\_slice}{SLICE\_SCALE} * (4 - ioprio)\right) \tag{5.1}$$

The priorities for real-time queues determine how long the per-round time-slice of the queue is. The formula for the time-slice is given in equation 5.1 on the facing page, where $base\_slice$ is the base slice for this type of queue, $SLICE\_SCALE$ is a constant defined to 5 and *ioprio* is the queue's priority. The base slice for the queue depends on whether the queue is for synchronous or asynchronous requests. If the queue is for synchronous requests the base slice is 100ms, in jiffies.

5.7 Suitability of Linux Schedulers

Of the Linux schedulers, only the CFQ scheduler is a suitable candidate for use with CM systems. It is the only scheduler which supports prioritizing of I/O requests. The other schedulers attempt to distribute I/O bandwidth fairly while optimizing for throughput and to a certain extent, latency. However, without prioritization they are unable to provide any service guarantees.

CFQ provides three different I/O classes and 8 levels of priority. The classes are real-time, best effort and idle. A process in the real-time class is given priority over best effort processes by being guaranteed a time-slice in every round. In addition, real-time queues are allowed to preempt idle and best effort queues. Unlike real-time and best effort queues, idle queues are not ranked internally by priority.

CFQ also provides efficient access in that it maintains the anticipation heuristic when servicing reads and dispatches requests in VSCAN order. However, CFQ does not provide actual performance guarantees. The testing phase reviews the quality of isolation in CFQ, in addition to how different priorities within the real-time class correspond to maintenance of deadlines and provided throughput.

CFQ also provides a low latency service, but this is only available to the filesystem in requesting metadata.

Reviewing the requirements as presented in chapter 3 on page 11, CFQ provides efficient disk reads through VSCAN request ordering. It provides partial support for multiple service types and guarantee levels through the ioprio interface, but lacks configuration options for specific deadlines or bandwidth allocation. CFQ provides flexibility by adapting the variables that control the anticipation heuristic dynamically, and by allowing administrator control of device-wide variables via the sysfs interface. Isolation is provided by allowing real-time requests to preempt best effort time-slices and by providing the anticipation heuristic to ensure that high priority interdependent reads are not continually interrupted by lower priority threads. However, since there are timeout values associated with requests, total isolation is not achieved as this would lead to starvation of best effort requests. CFQ is work-conserving by choosing not to set aside device time for high priority queues that do not have pending requests, instead letting them preempt other queues if requests arrive and they still have time left in their slice.

Since CFQ does not provide any absolute guarantees, it does not perform any admission control. Instead it aims to provide relative priorities with graceful performance degeneration as the number of queues increases. This is almost inevitable, since CFQ aims to provide a general scheduler that will run out of the box on any computer capable of running Linux.

A limitation of CFQ is that the real-time class functions as a low latency service class that is not well suited for large bandwidth processes. Enough of these processes entirely starve best effort processes if they provide a continual stream of synchronous I/O requests.

CFQ partially fulfills the requirements. In addition, it provides several provisions for prioritizing synchronous requests over asynchronous requests. There are also plans to incorporate bandwidth allocation[5]. CFQ is a modern I/O scheduler that follows the forefront of current research.

5.8 Summary

I/O Scheduling in Linux is modern and in continual development. To aid in this development, it has a well modularized generic block device layer with support for loadable schedulers. The schedulers deal in request structures, which hold the block I/O information that maps contiguous sectors to memory.

The execution path of I/O on Linux provides challenges for mixed-media schedulers that often have implicit requirements for high level information at the scheduler layer. Some of these challenges can be overcome, but others require substantial changes to the kernel API, some of which may not be suitable for general purpose use.

Chapter 6

Implementation of APEX in Linux

This chapter examines and justifies the extent and implementation of the APEX scheduler. The APEX implementation of the scheduler interface covers a bare minimum required to perform the required tests and comparisons to the existing schedulers. The implementation is built around the batch builder core as presented by Lund[25]. Since much of the code is necessarily bookkeeping or Linux specific the APEX implementation borrows, whenever possible, from existing kernel I/O schedulers.

6.1 Data Structures

The APEX scheduler introduces three data structures. They are the APEX I/O context, the APEX elevator data and the APEX queue. The APEX I/O context is defined as `apex_io_context`, as seen in figure 6.1 on the following page, and is a process specific structure found in `blkdev.h`. Every scheduler that wishes to hold data on a process specific level must introduce a custom I/O context structure. A pointer to one of these is then added to the general I/O context structure and housekeeping code is added to `ll_rw_blk.c` to keep this pointer in a correct state. The APEX I/O context holds a reference to the processes's queue, along with a reference back to process's generic I/O context. These two are used to gain access to data not immediately available in some context. The APEX I/O context also holds a reference to the active `struct apex_data` in `key`. Finally, for cleanup purposes the APEX I/O contexts are valid list elements[1].

[1]In the Linux kernel standard list implementation, a structure is made part of a list by holding a `list_head` instance[24].

```
struct apex_io_context {
    struct list_head list;
    struct apex_queue *apex_q;

    void *key;//unique identifier

    struct io_context *ioc;

    int class;//unused
    unsigned long deadline;//unused

    void (*dtor)(struct apex_io_context *);
    void (*exit)(struct apex_io_context *);
};
```

Figure 6.1: The process specific APEX I/O context information.

In the original presentation of APEX, separate request queues are generated when requested by applications. Due to difficulties presented in 6.4.1 on page 74, the implementation creates one queue per process. This queue is an instance of `struct apex_queue`, seen in figure 6.3 on page 56. Each queue holds a list of requests waiting for dispatch. It also holds a list of these requests sorted by sector number, in a red-black tree. A link back to the relevant device's APEX data is provided for availability in function calls where this is not a parameter. Since the queues are allocated dynamically, each queue is reference counted and the memory recovered when no further references are held. In practice this is done when the process exits.

The queues are organized into several lists, one for each type of queue and one for queues which do not currently have pending requests. To support this, a queue holds an array of `list_head` structures. Whenever a queue is emptied, it is placed on the list for waiting queues. When a new request is added to a waiting queue, it is moved to the list of active queues corresponding to its priority class. For consistency checking purposes each queue counts the number of associated requests allocated.

The queue structure holds data related to the modified token bucket implementation. There is a count of current tokens, `tokens`, along with the bucket depth, `max_tokens`. The `token_rate` is the number of jiffies between each time a queue is to be allotted a new token. Each time a queue is allotted a token, `last_token_update` is updated to reflect the current clock tick. The deadline and bandwidth allocated to a queue are recorded in the

```
struct apex_data {
    request_queue_t *queue;
    struct list_head active[APEX_NR_QUEUE_TYPES];

    struct list_head apex_ioc_list;

    /* Batch builder data*/
    int pending;
    int on_dispatch;

    unsigned long next_deadline;

    struct work_struct unplug_work;
    struct timer_list batch_delay_timer;
    int batch_delay_len;
    int threshold;
    int buildup;
    int delayed;
    int force_batch;
    int batch_active;

    int rnd_len;

    /* Per device tunables */

    unsigned long min_deadline;
    unsigned long default_deadline;

    /* For calculacing RQ cost, time in ms */
    int average_seek;
    int sectors_per_msec;
    int max_batch_len;

    int total_bandwidth;// (in pages/s)
    int available_bandwidth;
    int default_max_tokens;
};
```

Figure 6.2: The device specific elevator data structure.

```
struct apex_queue {
    struct apex_data *ad;
    struct list_head rq_queue;

    struct rb_root sort_list;

    atomic_t ref;
    int flags;

    struct list_head active[APEX_NR_QUEUE_TYPES];

    int allocated;

    int tokens;
    int max_tokens;
    int token_rate;//= nr of jiffies between arrival
    unsigned long last_token_update;

    int deadline;
    int bandwidth;
    int class;
};
```

Figure 6.3: The process specific request queue data structure.

corresponding variables. Each token represents a page worth of data, which on the test system is 4KB. While larger page sizes are desirable, this is the maximum page size Linux supports on the x86 architecture.

When a device registers a scheduler for use, an instance of that scheduler's elevator data is created. For APEX, this is `struct apex_data`, seen in figure 6.2 on page 55. This structure holds instance specific data. Some of this data is specifically tied to the device, such as the pointer to request queue structure associated with the device. In addition, it holds a reference to the list of active and waiting queues, and a list of all the APEX I/O contexts for processes that have performed I/O on this device.

For housekeeping purposes to assist with batch building, the number of requests pending dispatch and requests currently on dispatch are counted. The deadline (in jiffies) of the current batch being constructed is held in `next_deadline`, as is the remaining round length (in milliseconds).

Since APEX tries to predict the cost in time of dispatching requests, each device may have different tunable information. Depending on the cost of dispatching a single request, a device might require a higher minimum deadline. This is the case when the cost of dispatching a single request is high. In this case, the scheduler must be able to dispatch batches of at least a few requests in size to be able to amortize the the seek times, otherwise the batches would degenerate to EDF behavior. Requests with too short deadlines cannot decide the controlling deadline, as the consequences are the same. In addition to having a minimum deadline in `min_deadline`, a default deadline that is considered an optimal batch size for the device is recorded in `default_deadline`. Both deadlines are given in jiffies relative to the beginning of the batch dispatch.

To calculate the cost of a request, several device specific variables are presented. The average seek time on the device must be considered. This number does not need to be entirely accurate, but should present a good heuristic for deciding the cost of adding a request to the queue. An average seek that is too short creates batches so large that they regularly violate deadlines, whereas one that is too long degrades performance by creating unnecessarily small batches. Since requests coming to the scheduler can vary from single pages to whole tracks, the number of sectors must also be considered to contribute to the cost of a request. The cost of a single track is negligible, but several tracks might present a noticable delay. The variable `sectors_per_msec` specifies an estimate for how many milliseconds it will take to read a request from disk once the seek is completed.

Regardless of cost there must be an upper limit for the length of a batch. Otherwise, batches might become so large that requests that arrive after the batch has started don't have a chance of meeting their deadline. This is stored in `max_batch_len`.

To improve request cost amortization, APEX slightly delays batch building on the assumption that if a single request is ready for dispatch, more will soon follow. If this delay is shorter than the time it takes to perform a request, delaying will cost less than performing an incorrect choice. This mirrors the justification for waiting when anticipating further requests from a process which has completed an I/O operation[21].

The implementation of APEX records the estimated cost of the reservation class requests that are ready for dispatch in milliseconds. This cost estimate is stored in `buildup`. The buildup is compared to the batch delay threshold, stored in `threshold`. The batch delay threshold specifies how much buildup justifies dispatching a new batch. To prevent starvation when not enough requests arrive to trigger the condition, a maximum batch delay length is kept in `batch_delay_len`. If a delay lasts this long, the scheduler forces a dispatch of a new batch. This variable is currently static, but can be made to adapt to conditions. At the simplest this would entail being tunable by the user via the sysfs interface. It could also be made to adapt based on program behavior, such as counting the average amount of time from the first new reservation class request arrives, to the last one that was included in the same batch. Other possibilities exist and could be examined.

The dynamic event handler `struct timer_list` assists with batch delay. This `batch_delay_timer` is used to trigger a request to the kernel block driver to perform dispatch when the I/O buildup is not fast enough to meet the threshold within the maximum batch delay. The timer is created with two parameters, the function it calls when it is triggered and the `unsigned long` that it passes as argument to this function. For the batch delay timer these are `ad_batch_delay_timer` and the pointer address to the APEX elevator data.

To query the block device driver to perform dispatch, a `struct work_struct` is needed. A `struct work_struct` is an item of work that will be scheduled in the process context of the corresponding work list. In this case the kernel block I/O thread, `kblockd`. The work `unplug_work` is passed to the block device driver function `kblockd_schedule_work`. The work is associated with a function, `apex_kick_queue` which takes a pointer with no type as argument. The `unplug_work` uses the address of the block device's request queue in this pointer.

The flags `force_batch` and `delayed` assist with statefulness. The former specifies that a batch must be built the next time the block driver calls for requests to be dispatched. The `delayed` flag specifier's whether batch building has been delayed due to lack of buildup. If it is active the `batch_delay_timer` has been started, but is not stopped yet.

Average Seek	10ms
Sectors per ms	20
Maximum batch length	500ms
Minimum deadline	100ms
Default deadline	500ms
Total bandwidth	10k pages
Available bandwidth	10k pages
Force Batch Building Threshold	5 requests
Maximum Batch Building Delay	25 milliseconds

Table 6.1: Default values for the APEX elevator data

While admission control is not implemented, to do so each device must keep a record of how much bandwidth it can sustain and how much is currently reserved. This is done in the `total_bandwidth` and `available_bandwidth` variables.

The default values for the tunable variables in the APEX elevator data structure are shown in table 6.1. The average seek rate is slightly lower than that provided by the manufacturer, which takes into account that most requests will add time by being long. The sectors per millisecond value is 20. This is a high estimate. It allows for an extra seek for every 100KB or so in a request. This ensures that even if the disk geometry is different than expected, the request should finish within the estimated time. These two variables can be further fine-tuned. The maximum batch length is 500ms, which assumes that the most common deadline will be 1s. The maximum batch length should be half the most common expected deadline or less, so that a request that arrives just after the batch starts will not have its deadline violated by being scheduled in the next batch. The total bandwidth and available bandwidth are set to 40MB, but in practice this value currently means nothing. The force batch building threshold is set low to ensure that batches are built promptly even when there are few running reserved service class processes. The maximum batch building delay of 25ms approximately represents two average seeks, one forward and one backwards, which would be the expected cost of dispatching one request instead of waiting.

6.2 Functions

The functions the APEX implementation registers to the scheduler interface are shown in figure 6.4 on the next page. APEX does not implement all

```
static struct elevator_type elevator_apex = {
    .ops = {
        .elevator_merge_req_fn     = apex_merge_requests,
        .elevator_queue_empty_fn   = apex_queue_empty,
        .elevator_completed_req_fn = apex_completed_request,
        .elevator_add_req_fn       = apex_add_request,
        .elevator_dispatch_fn      = apex_dispatch,

        .elevator_former_req_fn    = elv_rb_former_request,
        .elevator_latter_req_fn    = elv_rb_latter_request,

        .elevator_put_req_fn       = apex_put_request,
        .elevator_set_req_fn       = apex_set_request,

        .elevator_init_fn          = apex_init_queue,
        .elevator_exit_fn          = apex_exit_queue,
        .trim = apex_trim,
    },
    .elevator_name = "apex",
    .elevator_owner = THIS_MODULE,
};
```

Figure 6.4: Using the ops structure in `elevator_type`, APEX registers the functions in the scheduler interface that it implements.

the interface functions. It focuses on the minimum necessary to perform an analysis of the batch building done in the APEX scheduler and compare the performance of this to that of the other schedulers in the Linux kernel. The functions not implemented are those related to customized merging of requests, along with specialized housekeeping functions for handling requests that are dequeued and requeued by the device driver.

The functions `apex_init_queue` and `apex_exit_queue` are called when the scheduler is loaded and unloaded to service a device respectively. As such, they serve as constructor and destructor functions. The constructor allocates kernel memory for an instance of `struct apex_data`, initiates the various lists and stores the default values in its member variables. In a complete version of the scheduler it would be correct to provide an interface to adjust these variables, preferably through the sysfs interface. The prototype implementation only uses the scheduler for testing on a single drive and hardcodes the desired values in the constructor. The default values are shown in table 6.1 on page 59. The request queue for the device is passed as a parameter to the constructor and a reference to this is stored in the APEX elevator data.

The destructor, `apex_exit_queue`, calls the function `kblockd_flush` to complete any ongoing dispatch of requests. It then acquires the request queue's spinlock and cleans up the remaining APEX I/O contexts by clearing the reference to them in the generic I/O contexts. Finally, the memory for the APEX elevator data is freed.

The function `apex_trim` is part of the I/O context housekeeping done when unloading a scheduler from memory. While this is implemented, APEX is never unloaded from memory due to the complex handling of synchronization necessary.

When a request is created from a bio, a call is made to allocate scheduler specific resources for that request (as is seen in figure 5.4 on page 39). In APEX, this call is `apex_set_request`. The call graph for this function can be seen in figure 6.5 on the following page. First the APEX I/O context of the current process is fetched using `apex_get_io_context`, which is described in 6.2.1 on page 67. The APEX I/O context may have a pointer to a valid APEX queue when it is returned to `apex_set_request`. If not, a call is made to `apex_alloc_queue` to create a new one. Once the queue is available, the count of requests allocated to this queue is incremented as is the reference counter. The memory address of the I/O context and queue are stored in the request's two special fields for private elevator data. If memory allocation for either the APEX I/O context or the queue fails, `apex_set_request` returns an error value.

The destructor companion to `apex_set_request` is `apex_put_request`.

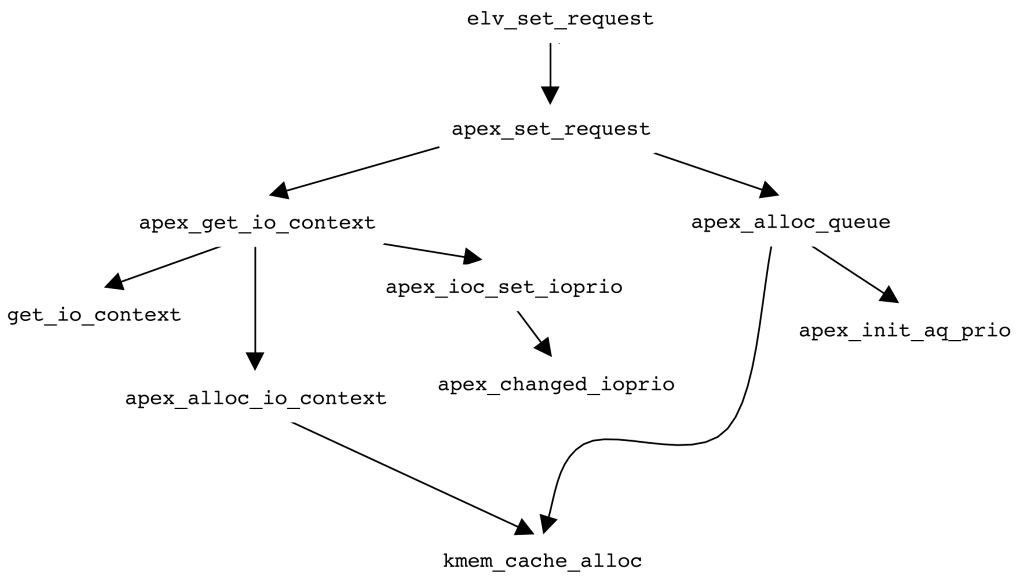

Figure 6.5:

It is called to free any scheduler specific resources when a request is being freed. It uses the private elevator data fields in the request structure to fetch the queue and original APEX I/O context of the request, calling their respective put functions.

After a request is initiated, a call is made to `apex_queue_empty` to see if there are any candidates for merging. If the call returns false, the elevator layer attempts to perform a back merge before it passes the request on to the scheduler. As seen in 5.5.4 on page 45, back merging is when a new request continues after a request that is already pending dispatch. This is done by adding requests into a red-black tree sorted on the last sector in the request. If a back merge is possible, the requests are merged. If not, the elevator queries the scheduler to see if it performs a merging operation on the new request. The implementation does not implement any extra merging to avoid added complexity, since most merges are back merges[3]

The function `apex_merge_requests` performs the necessary housekeeping on the scheduler specific data structures when two requests are merged. The functions `elv_rb_former_request` and `elv_rb_latter_request` are both for merging. Previous to the 2.6.19 version of the kernel, these functions were specific to the various schedulers, and it is therefore still part of the elevator interface.

Merges may happen to unrelated requests, that is requests that consist of bios that have different origins[6]. If this happens, the token count for one of the queues is incorrectly docked for another process's request. To ameliorate this APEX is liberal in allowing requests that exceed the current amount of remaining tokens a queue has, providing the queue has any tokens. More on this later.

After being checked for possible merges, a request is sent to the scheduler for processing. This is done with `apex_add_request`. This function first retrieves the correct queue for the request. If the queue is waiting for a new request, it is activated by appending it to the list of active queues that corresponds to its priority class and removing it from the waiting list.

If the queue is a resource reserving queue, then the APEX elevator data's build-up must be updates before the request is appended to the queue's request lists. The buildup is counted up and checked against the threshold. If dispatch has been previously delayed and this request brings the buildup over the threshold, then the batch delay timer is deleted and a dispatch is scheduled by calling `apex_schedule_dispatch`. This function calls upon the block layer to perform an unplugging of the relevant block device. This is not done immediately, but instead the next time the kernel block I/O thread is scheduled for CPU time.

Finally, the amount of pending requests is incremented, and the request is added to the queue's SCAN sorted list of requests and to the chronologically ordered FIFO queue. Once the request is added, it is ready for dispatch.

The call graph for APEX's batch building functions can be seen in figure 6.6 on the next page. When the device driver is unplugged by the kernel, as described in 5.3.4 on page 31, the driver makes a call to `elv_next_request` which returns a request from the elevator queue for dispatch. If the elevator queue is empty, it performs a call to `__elv_next_request` to notify the scheduler to insert requests into the elevator queue. For APEX, the function that is called is `apex_dispatch`, which returns the number of requests that were sent to the elevator queue.

When called, `apex_dispatch` takes two arguments, the `request_queue_t` for the device and the force flag. The force flag is used to indicate forced dispatch, in which case the scheduler may not defer dispatch. This is primarily used to drain queues when unmounting a device or switching schedulers. If the force flag is active, APEX calls `apex_force_dispatch` which drains all queues without regard to service classes or tokens. Dispatch of requests is detailed below.

If a batch is already active, a work-conserving phase is considered. The deadline of the batch must not have passed and there must not be resource reserving queues waiting for dispatch. If the conditions are met, APEX performs a work-conserving phase, which is handled by `__apex_wc_`

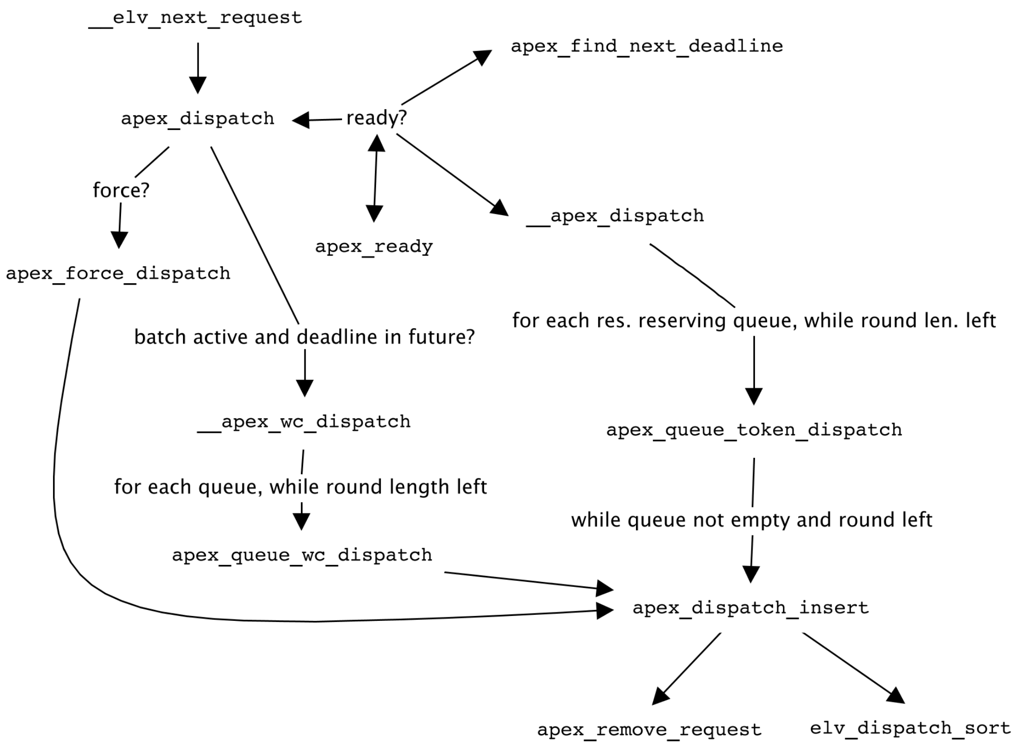

Figure 6.6: The call graph for dispatching requests on APEX.

dispatch. The work-conserving phase is performed over multiple calls to apex_dispatch so that the time spent is as accurate as possible. To facilitate this accuracy, the remaining round length is recalculated for each request. The actual phase is performed by iterating over all the active queue lists, calling apex_queue_wc_dispatch to dispatch a single request independent of tokens. The remaining round length is decremented by twice the request's calculated cost. Twice the calculated cost is used to guarantee that the work-conserving phase will finish before predicted. The exact implementation of the work-conserving phase is up to policy, and can be changed. As an example, to compensate for the lack of correct deadlines the implementation aggressively discriminates in favor of reserved resource queues.

If a batch is not already active, apex_dispatch checks whether it is ready to begin a new batch by calling apex_ready. If there is some build-up and the delay flag is not active, the batch delay timer is enabled in anticipation of more resource reserving requests arriving. Otherwise, the state is updated to reflect that a batch is active and being built. If buildup is over the threshold and the batch delay timer had been activated, it is now deactivated.

When performing batch building, the APEX dispatch function begins by calculating the controlling deadline. This is done in apex_find_next_deadline. This function calculates the default deadline, then iterates over all the active reservation priority queues checking if they have any controlling requests with deadlines prior to this. If they do, this deadline becomes the controlling deadline so long as it occurs after the minimum deadline. If it does not, it is ignored. This is a policy decision that depends on many factors. Another possible policy is to use the minimum deadline so that the already late request is delayed as little as possible. Since the implementation focuses on firm deadlines, it follows that being late has lowered the value of the request and the scheduler should instead seek to improve overall performance by dispatching more requests in the batch. This leads to an improvement in overall performance that helps fill buffers which decrease the impact of missed deadlines. While this implementation is $O(n)$, where n is the number of resource reserving queues, a priority queue can be maintained to lower this cost if the CPU overhead becomes a factor.

Once apex_dispatch has the controlling deadline is found, the round length in milliseconds is calculated. Milliseconds are used because they guarantee better granularity than jiffies, which may rate from 24 to over 1000 per second. The batch building is then done by calling __apex_dispatch.

The first phase of batch building is performed by iterating over the list of active reservation priority queues and calling apex_queue_token_dispatch to perform a token-limited dispatch on that queue. The token-limited dispatch function first updates the queue's token count and then iteratively dispatches requests from the queue while deducting tokens. The remaining round length is also decremented with the estimated cost of the request (corresponding to `t_es` in the original presentation of APEX). The token-limited dispatcher is done processing a queue when either the remaining round length falls to 0 or below, the queue is empty or the queue runs out of tokens. As mentioned above, APEX allows the queue to dispatch single requests which would take it to below 0 tokens. This is both for the previously mentioned reason of merges, and to ensure that queues with low token allotments are allowed to dispatch requests larger than their bucket depth. If this has occurred, the batch builder resets the queue's token count to 0.

There are two alternatives to resetting the the token count to 0 when a queue acquires a token deficit. One is to deny it service, which is impractical since the bucket depth may not be enough to handle the request. In this situation the resource reserving queue degenerates to being serviced in the work-conserving phase. The other is to allow the queue to accumulate negative token counts, giving it a temporarily increased bucket depth and corresponding number of tokens in advance to cover the cost of the request. This is problematic since the excess of the request might represent a partial read-ahead or wrong read-ahead. In the former case, the advantage of pre-spending the tokens is lost as the process waits for the next request to dispatch while the queue accumulates positive tokens again. In the case of a read-ahead miss, the process never sees the data that the queue paid for and again experiences a delay while the queue accumulates tokens.

The dispatch of a single request is done by `apex_dispatch_insert`, which first calls `apex_remove_request` to remove the request from all active data structures and, if the request is from a resource reserving queue, decrements the cost from the build-up. The request is then sent to `elv_dispatch_sort`, which inserts the request into the driver's dispatch queue in one-way elevator sorted order.

If `__apex_dispatch` does not dispatch any requests, which may happen if no resource reserving queues have tokens or if there are no pending resource reserved requests, then a work-conserving phase is immediately run instead.

There is an inherent round robin nature in the way the implementation of APEX builds batches. When a queue is activated, it is appended to the tail of the active list for its class. When a batch is built, it is always built from the beginning of the active list. When a queue is empty, it is removed from the active list. This way, queues are guaranteed to be serviced eventually.

When the device driver is completely finished with a request, `apex_completed_request` is called. Currently, this function only decrements the elevator data's count of requests being dispatched. However, it is conceivable that this function could also kick start batch building when it sees the number of dispatched requests is falling below a given threshold. This would more correctly follow the behavior specified in the original presentation of APEX. Since the delay in building the batch is negligible compared to the execution of the batch, this is not a major concern during testing. It is also worth noting that the synchronous nature of read requests might adversely come into play in this situation. There might be unnecessary delays for processes that have not received the result of their last read request until after the current batch is completely served. In this case, the process is not be able to issue new read requests until after the batch is begun, when it is too late.

6.2.1 Functions for Handling Data Structures

Since APEX I/O contexts and queues are dynamically allocated, constructor and destructor-like functions exist to handle them. Like many similar kernel constructs, APEX I/O contexts and queues are generated transparently on demand. The following is a process very similar to that performed in the other Linux schedulers that implement their own I/O contexts or their own queues (that is, the CFQ and Anticipatory schedulers). The call graph for allocation of APEX I/O contexts and queues can be seen in figure 6.5 on page 62.

APEX I/O contexts are generated by the `apex_get_io_context` function. This function first uses `get_io_context` to retrieve the current generic I/O context. If no such I/O context exists, one is be generated for the current context transparently. If the current I/O context does not have a valid APEX I/O context, which is always the case the first time the current I/O context is retrieved by the APEX scheduler, one needs to be allocated. The allocation function, `apex_alloc_io_context`, allocates a new APEX I/O context from a cache[2]. If the memory is successfully allocated, it is zeroed out (to prevent accidental access of stale data) and key data is stored. Specifically, a link to the APEX elevator data, and pointers to the functions to free and exit the APEX I/O context. The list head is also initialized. The destructor and exit function pointers for the APEX I/O context are set to `apex_free_io_context` and `apex_exit_io_context` respectively. The `apex_get_io_context` then sets the appropriate values for the APEX I/O context and adds it to the APEX elevator data's list of APEX I/O contexts. APEX I/O contexts are not reference counted, but are instead destructed when the process exits.

[2]The kernel has a special cache subsystem for structures that are frequently allocated and deallocated. For APEX these caches are allocated as part of the module initiation.

If the allocation was successful, the reciprocal links are stored between the APEX I/O context and generic I/O context. The I/O context is registered as having its I/O priority changed and the APEX I/O context is added to the list of contexts held by the elevator data. Finally, a check is performed to see if the flag indicating that the I/O priority of the generic I/O context has changed, as determined by the `ioprio_changed` member variable. If so, `apex_ioc_set_ioprio` is called to set the priority changed flag of APEX I/O context's queue. The `apex_ioc_set_ioprio` function takes the generic I/O context as argument, and passes the APEX I/O context to `apex_changed_ioprio` to perform the actual marking of the flag. If support for multiple block devices were added, the `apex_ioc_set_ioprio` function would need to iterate over the list of device specific APEX I/O contexts, but currently there is only one APEX I/O context per generic I/O context, so this is simplified. Since the priority is set to have been changed explicitly when the context is newly allocated, this test is always be triggered by new contexts.

APEX queues are allocated the first time a request is allocated in a given context. The allocation is done in `apex_alloc_queue`. In this function, memory is requested from the queue memory cache. If allocation succeeds, the memory is reset to prevent stale data from causing undefined behavior. The queue has various queues, which are all initialized. The link to the queue in the APEX I/O context is updated, as is the link to the elevator data in the new queue. The queue is flagged as best effort, and marked as having had its priority changed. The queue is added to the waiting list.

The final action of queue allocation is a call to `apex_init_aq_prio` to update the I/O priority for the queue. It is important that this function is called in the context of the process that issued the I/O request, so that the priority for the queue is set correctly. Therefore, it is only called from `apex_set_request`, either directly or via the queue allocation function. It is called directly each time a new request arrives to ensure that it is treated correctly.

The `apex_init_aq_prio` function begins by checking whether the flag indicating that the queue has changed priority is set. If it is not set, it returns immediately. It then copies the value of the new priority from the task structure for the current task. Once this is done, it clears all the class flags for the queue. The function then updates the class specific data, depending on the new class of the queue. If the queue is now a best effort queue, the class flag is updated and the queue's bandwidth is set to 0. If it is a resource reserving queue, the class flag is updated and the queue's deadline is set.

The bandwidth and bucket depth for the queue are updated by the `apex_update_bandwidth` function. This function recalculates the available bandwidth, and sets the queue's bandwidth and bucket depth. It also converts the bandwidth to token rate, which is the number of jiffies between each new token arrival for the queue.

Control then returns to `apex_init_aq_prio`. If the queue is active, it is moved from its old active list to the new class's active list. Finally, the flag indicating that the priority has changed is cleared and the function returns.

6.3 The `apex_ioprio_set` System Call and the APEX I/O Priority Framework

To manipulate the APEX scheduler settings from userspace, the `apex_set_ioprio` system call is provided. This section covers how the system call is added to Linux and its usage from userspace. It also shows why the existing I/O priority framework is unsuited for use with APEX.

Defining new system calls in Linux[24] is done by specifying a system call number for the function and creating the function itself. The system call number is architecture specific. The call is only implemented on the i386 architecture, but other implementations would be analogous. Each architecture specifies a system call table, for i386 it is located in `arch/i386/kernel/syscall_table.S`[3]. An excerpt from this file is seen in figure 6.7 on the following page. The entry specifies the name of the system call, which is prefixed by '`sys_`' as is convention. The system call number is implicit in the placement. The call number is denoted in comments alongside every 5th system call for convenience.

The next step is to define the system call number explicitly for the architecture. This is done in the architecture specific version of `unistd.h`. An excerpt from the i386 version of this file can be seen in figure 6.8 on the next page.

Finally, the system call itself is implemented. Since the system call must always be compiled into the kernel, it must reside in the main kernel tree. The APEX I/O priority system calls reside in the source file `block/apex_ioprio.c`, so that it is grouped with the block device layer and scheduler sources. To ensure that the source is compiled correctly the resulting object file, `apex_ioprio.o`, is added to the `Makefile` for the `block/` directory as can be seen in figure 6.9 on page 71. This way the source file is automatically compiled and linked into the kernel.

The actual system call is very similar to the `ioprio_set` system call implemented in `fs/ioprio.c`. This system call is used for controlling the existing I/O priority framework in Linux.

[3]For many architectures it is located in `entry.S` in the correspondingly same directory.

```
ENTRY(sys_call_table)
        .long sys_restart_syscall       /* 0 */
        .long sys_exit
        .long sys_fork
        .long sys_read
        .long sys_write
        .long sys_open          /* 5 */

. . .

        .long sys_tee                   /* 315 */
        .long sys_vmsplice
        .long sys_move_pages
        .long sys_getcpu
        .long sys_epoll_pwait
        .long sys_apex_ioprio_set       /* 320 */
```

Figure 6.7: An excerpt from the i386 system call table, as found in `arch/i386/kernel/syscall_table.S`. The new APEX I/O priority system call is at the bottom with system call number 320.

```
#define __NR_restart_syscall  0
#define __NR_exit             1
#define __NR_fork             2
#define __NR_read             3
#define __NR_write            4
#define __NR_open             5

. . .

#define __NR_tee              315
#define __NR_vmsplice         316
#define __NR_move_pages       317
#define __NR_getcpu           318
#define __NR_epoll_pwait      319
#define __NR_apex_ioprio_set  320
```

Figure 6.8: An excerpt from the i386 system call numbering, as found in `include/asm-i386/unistd.h`. The new APEX I/O priority system call is at the bottom with system call number 320.

```
#
# Makefile for the kernel block layer
#

obj-$(CONFIG_BLOCK) := elevator.o ll_rw_blk.o ioctl.o \
 genhd.o scsi_ioctl.o

obj-y += apex_ioprio.o

obj-$(CONFIG_IOSCHED_NOOP)      += noop-iosched.o
obj-$(CONFIG_IOSCHED_AS)        += as-iosched.o
obj-$(CONFIG_IOSCHED_DEADLINE)  += deadline-iosched.o
obj-$(CONFIG_IOSCHED_CFQ)       += cfq-iosched.o

obj-$(CONFIG_BLK_DEV_IO_TRACE)  += blktrace.o
```

Figure 6.9: The `Makefile` for the kernel block layer. The `apex_ioprio.c` source file is configured to always be compiled.

The `ioprio_set` system call takes three arguments, `which`, `who` and `ioprio`[12]. The `which` argument determines how the `who` argument is interpreted. It can be set to specify a process ID, a process group ID or a user ID. In the latter two cases all matching processes are modified. The `ioprio` parameter specifies the scheduling class and priority that will be assigned to matching processes. These two are internally encoded in 16 bits of information, with 3 set aside for class and 13 set aside for the priority information.

The I/O priority applies for reads and synchronous write requests. Write requests are made synchronous by passing either `O_DIRECT` or `O_SYNC` flag to the `open` system call. Asynchronous writes are issued outside the context of the process by the `pdflush` daemon kernel thread and therefore process specific priorities do not apply[4]. It also only applies to block devices using the CFQ scheduler, since no other schedulers utilize I/O priorities.

The `ioprio_set` system call is divided into two functions. The wrapper function `sys_ioprio_set`, which interprets the `which` and `who`, and `set_`

[4]Strictly speaking, the priorities of the pdflush thread apply, but these are created and destroyed dynamically, so while it is possible to set priorities for these threads, they are not guaranteed to apply.

task_ioprio which sets the I/O priority for a given task[5].

The wrapper function first extracts the class and priority from the ioprio argument. It then checks if the calling process has high enough access to set the real-time or idle classes. Exactly how high enough is defined depends on several factors, but on a standard Linux system, this requires the process to be owned by the superuser (root). The next step is to check the value of which. If which designates that a single process should be adjusted, who is either interpreted as meaning this process (for the value 0) or a process ID. The corresponding task is looked up and if it is valid, set_task_ioprio is called with the task and priority information as arguments.

If which is designated to mean a process group ID, a value of 0 in who again is interpreted to mean the current process group. The function then iterates over all tasks in that group and calls set_task_ioprio for each of them. If who is interpreted as a user ID, a value of 0 is taken to mean the current user. Otherwise a lookup is performed to fetch the correct user. The function then iterates over all tasks and calls set_task_ioprio for those owned by the given user.

The set_task_ioprio function first ensures that a process attempting to modify the I/O priority for a process owned by a different user is owned by the superuser. Otherwise it returns an error value. The task is then locked, and the member value ioprio is updated. If the task has a generic I/O context, the ioprio_changed value is set. The rest of the update is done by scheduler, which is covered in 6.2.1 on page 67.

It also worth noting that the Linux I/O priority framework supports inheritance of classes and priorities. When a process is cloned or forked, the child maintains the same class and priority as the parent.

The apex_ioprio_set system call is similar to the ioprio_set call that it is divided into the same two parts. The APEX I/O priority framework is also implemented by adding new fields to the task structure. Since this is the only structure that is guaranteed to exist for an active process[6], it is where the information must reside for processes to be able to control the class and suchlike of other processes.

While the general purpose I/O priority framework has obvious use for being able to set the priorities of other processes, since it is transparent and is thus eligible for use with any program, including those which do not implement their own I/O priority calls. That is, the program ionice enables the user to set the priority of other processes. The assumption here is that the user knows best what processes require the various priorities. This also

[5]Threads and processes are internally known as tasks in the kernel.

[6]I/O contexts are not generated until a process performs explicit I/O operations.

allows programs without provisions for using the I/O priority framework to use it. If a process were only to control its own I/O priority, then this information could be stored in the I/O context.

In addition to these factors, there is one more concern when deciding how to implement the new I/O priority framework. The old framework provides universal priorities. No matter what block device the process uses, so long as the device uses the CFQ scheduler, it is granted the same prioritized access. However, APEX provisions are designed to be transaction specific. It does not make sense to reserve an equal number of pages from every block device on a system.

The current implementation functions in the same manner as the original I/O priority framework, because of the underlying limitation that there is only one APEX I/O priority per process. As such, there is an assumption that APEX is only actively used for one block device per process.

The APEX I/O priority stores four pieces of information in three variables in the task structure. The class and deadline are stored in 16 bits, with the top 3 bits specifying the class and the bottom 13 used for the deadline. In addition, the token rate in pages per second and maximum bucket depth in pages are stored in separate variables.

When the relevant process is found, `apex_ioprio_set` calls `set_task_apex_ioprio`. This function then stores the above values in the task structure's variables. It also sets the process's generic I/O context's `ioprio_changed` variable. This variable forces both CFQ and APEX to reread the priority information stored in the task structure.

Since there is no way to propagate deadlines correctly along with each access (see 6.4.1 on the following page), the scheduler must make a general assumption based on the process's requirements. The process therefore specifies a general deadline for its requests, and the scheduler attempts to uphold this for each arriving request. Since neither the scheduler nor the process can know whether the filesystem must make preceding lookups before the actual data is requested, this only provides a heuristic for the scheduler that helps minimize request lateness.

6.4 Limitations in the Implementation

This section explores the limitations in the current implementation of APEX in Linux compared to the original presentation of the scheduler. It also seeks to justify the limitations either due to the difficulties presented by the Linux kernel internals or due to reduced relevance compared with more important features.

The difficulties in implementing the APEX scheduler as presented by Lund[25] stem primarily from the generic nature of the Linux kernel. While APEX is designed with the availability of a combination of userspace and block device information in mind, this does not always translate to the low level context of kernel I/O framework. Specifically, APEX is designed as part of an MMDBMS that functions on its own partition.

The implementation does not have a queue management module. This prevents processes from performing concurrent I/O at varying priorities. There is no admission control, since admission control is not necessary to test feasibility of the batch builder. There is also no low latency service class, since this has already been implemented in CFQ.

The primary limitation of the Linux scheduling framework is that it provides much less relevant metadata for each I/O request than APEX originally assumes to be present. Since APEX is designed to provide read and write primitives to an MMDBMS system, and simultaneously assumes direct access to the storage device, it is free to redefine system calls to require as much or as little extra information as necessary. This is not feasible in the Linux scheduling framework. In this context the only information provided is the logical block address and number of sectors in the request. It is not possible to know if the request even concerns a file I/O operation, it may be a memory swap operation.

The consequences of not being able to pass metadata to the scheduler is that it cannot be made transaction aware. It is also difficult to make provisions for dynamic creation of queues. Instead it assumes a single queue per process. Since process creation using the `clone` or `fork` system calls is cheap in Linux, it is still possible for processes to make some dynamic queue provisions in userspace. Some possible ways to implement dynamic queues are examined in 6.4.1.

APEX achieves efficient reads by building batches and having these sorted in SCAN order. The system call interface provided gives processes the ability to specify service requirements, for which the scheduler provides firm guarantees. These guarantees are weakened by the lack of admission control.

Some flexibility is achieved through a set of per device tunable variables. While an interface to these is not as yet written, it could feasibly be implemented via a new system call interface or via the sysfs interface.

6.4.1 Multiple Queues Per Process in Linux

This section explains why it is difficult to divide work from single processes into separate queues on Linux and present possible solutions. The problem is that there is very little contextual information available to the I/O scheduler. Section 5.4 on page 32, 5.3.2 on page 30 and 5.3.3 on page 31 show that the contextual information specific to a request when it arrives at the scheduler is limited. The process that issued the request and information related to the process is visible, but it is not possible to see what file the request is operating on.

The `struct request` and `struct bio` objects contain no valuable information about the context as such. However, one could conceive of a situation in which they did, allowing the scheduler to correctly sort requests into the correct queues. For this to be meaningful, queues would have to be created in advance and then requests tagged in some way that the scheduler could recognize.

Assuming that there is such a way to create and manipulate queues as the Queue Manager in the original presentation of APEX specifies, there are at least two conceivable ways to convey the information of what queue a request pertains to from userspace to the scheduler. One way would be to create a new system call that mirrors the access functions (`read` and `write`) and allows the user to specify which queue the access should use. However this would require adding a parameter to several function calls, which is very invasive, and would require rewriting the entire execution path to use special versions of the general purpose functions.

Piggybacking the information on existing data structures, with as little invasive change to the internal call interface as possible, would provide a less invasive way of making queue specific requests. The information could then be propagated to the `struct bio` when this is generated.

To do so, the information would need to be available in the two functions that create `struct bio` instances for submission to the block device layer, which are `block_read_full_page` and `do_mpage_readpage`. Unfortunately, both `block_read_full_page` and `mpage_readpage` take only two arguments, the page in memory to read to and the `get_block_t` function for the filesystem on the device.

Thus the information must be placed under the `struct page`. Placing it directly in the `struct page`, which represents a physical page in memory, is out of the question. One instance of this structure exists per physical page of memory on the system. Increasing its size would not only increase overall kernel memory usage, but cause additional overhead on all kernel memory operations. The page structure has two members that are candidates for holding this information. One is the buffer heads that might be linked in the `private` member. Unfortunately, buffer heads are not used consistently for I/O anymore. The only remaining candidate is the `struct address_space` that maps the inode's blocks to memory.

This presents some problems. A process does not always have sole owner-ship over a file's mapping. A file's mapping is copied from the inode's mapping when the process opens it. This saves memory by not duplicating the same file in cached memory, but it means that the `struct address_space` cannot be relied upon to hold the queue data. What would happen if two processes opened the same file using different queues? One could conjecture that they queue with the highest priority should be used, but this presents problems since the actual demands might be entirely different. One user might wish to copy the entire contents of the file to a different filesystem, whereas the other user wishes to purview the media contained in it. If the higher priority queue is used here, the user copying the data will continually exhaust the queue's tokens, potentially causing negative impact on the playback of the media.

Assuming the aforementioned set of system calls for manipulating and deciding which queue to currently use. These system calls would mimic those provided by Lund[25]. The queues would be held at the scheduler level, with one default queue always guaranteed to exist. At this point two alternatives present themselves. One is to add an additional system call that manipulates which queue is currently 'active' so that processes may change the queue prior to making I/O requests.

Improved performance could be achieved by creating a specialized read system call, which takes the normal read parameters in addition to a queue parameter. If the queue is valid, the system call would change which queue is active for the process, and then function as a normal call to read. Once the call returns, the queue could be reset, depending on policy. This way reading would still only be one system call.

Furthermore, the direct I/O execution path does provide some overlap between process specific file data structures and bio creation. Specifically, `direct_io_worker` takes a kernel I/O control block, `struct kiocb`, as one of its arguments. This is the function that constructs the `struct bio` objects which are sent to the block device layer. Since the kernel I/O control block has a link to the open file object, an extra field could be created in the `struct file` and `struct bio` that holds the queue tag for accesses to file. If `struct bio` were to include a queue tag, the scheduler would need to look through at least some of these per request to decide which queue to place the request in. It would then need to have some way of looking up the queue based on that tag. Several options present themselves, such as hash tables or sorted lists.

While few applications would require this degree of control, it is for precisely these systems that APEX is designed. It is worth noting that these changes to the system call interface would require rewriting some parts of these applications. The applications would need to keep a data structure mapping open file descriptors to the correct queues, so that they could activate the right queue when accessing the corresponding files. A different option is examined in section 6.6 on page 81.

6.4.2 Correctly Managing Admission Control

It is not possible to provide correct admission control without accessing device specific data structures from the I/O priority system call interface. Two options exist. One is to design a device specific system call that reserves the resources when the process begins I/O, the other is to design a device specific system call that immediately reserves the desired resources. What follows is an examination of how each of these could be implemented.

The first option, to create a system call that stores information about the desired resources, but waits to reserve until the process requests I/O, mirrors APEX's original design. In the original presentation queues are created with specific bandwidth requirements, but do not actually attempt reserve this bandwidth until a transaction is made active on that queue. If the resource are not available, the transaction fails to become active and an error is returned to the user.

To do this on Linux would require a system call to specify the resources to be reserved when the process becomes active, and an addition to the execution path of a read that checks whether the queue's resources can in fact be reserved. This cannot be added to the scheduler itself. The scheduler has four points of contact with the execution path of a synchronous request. The first is when a merge is attempted, but this can be discounted since the resources used by the merged request will be deducted from the original owner's reservations.

The second point of contact is when `get_request` calls `elv_may_queue`. This function can return three values, `ELV_MQUEUE_MAY`, `ELV_MQUEUE_NO` and `ELV_MQUEUE_MUST`. A process can be prevented from queuing a request by returning `ELV_MQUEUE_NO`, which causes the `get_request` to return a failure to `get_request_wait`. In this case, a failure is not returned to the process, but the process sleeps until requests have been served and freed. This presents a problematic situation. Assuming there are no requests currently allocated, and the processes that have reserved the resources do not issue any further requests, the callback to reactivate the process will never be performed. This situation would occur if the new process were to issue its first request just after all the processes which had previously reserved resources had finished their I/O, but not exited yet. Once any of these processes have exited, the resources are freed, and the new process may be able to reserve the resources it requires, but it will not be activated until an I/O operation is completed.

The solution to the above problem is to add a new call to reactivate waiting queues that is performed when reserved resources are freed. The call would be added in the following fashion. When a process sleeps due to not being able to allocate requests, it is placed on a wait queue. This queue is one of two wait queues stored in the block device's `request_queue_t`. The `request_queue_t` has an internal `request_list` which it uses for request allocation. The `request_list` holds the two wait queues, of type `wait_queue_head_t`, one for processes that are issuing a read request and one for writes. To reactivate sleeping processes on the wait queue, a call to `wake_up` is made. This function takes as its parameter the address to the wait queue from which the process is to be awoken.

A call to `wake_up` would need to be made when resources are freed in the scheduler. In APEX, this is done when the reference count of an `apex_queue` falls to 0. This happens in the function `apex_put_queue`, which calls `apex_update_bandwidth` which frees the bandwidth resources previously held by the queue. After the resources are freed, either in `apex_put_queue` or `apex_update_bandwidth`, a call to `wake_up` with the correct waiting queue as a parameter would need to be made.

The disadvantage of this form of admission control is that the process sleeps until the resources are available, rather than returning an error immediately. This presents a problem. The program is not able to alert a user that the desired resources were not available, thus the user experiences loss of quality. The user loses the option to attempt to view the desired item without reservation guarantees.

The third point of contact is when `blk_alloc_request` attempts to reserve scheduler specific resources when allocating a `struct request` by calling `elv_set_request`, which in turn forwards the call to the specific scheduler's `elevator_set_req_fn`. For APEX, this function is `apex_set_request`. Under the current implementation, this is where a process's queue is first created. If bandwidth resource reservation were done at queue creation time one could return an error value. In this case, `apex_set_request` would fail and return an error value. This would cause the same behavior in `get_request_wait` as returning `ELV_MQUEUE_NO` from `elv_may_queue`. In other words, these solutions are equivalent.

The fourth and final point of contact is when `add_request` calls `__elv_add_request`. This function does not return a value, so it cannot be used to return a negative result from the admission control mechanism.

To correctly manage admission control, a user feedback mechanism must be provided. This mechanism would warn a program attempting to reserve unavailable resources that an error has occurred, thus allowing the program to take some form of action based on this.

Reviewing the execution path of a read request as presented in section 5.4 on page 32, any function prior to `generic_file_aio_read` can be used to return an error value. After this function the execution paths of buffered and unbuffered reads diverge and do not share common functions again until `submit_bio`, which does not return any value. Since `generic_file_aio_read` is called after the filesystem specific read function, it is not actually guaranteed to be called. Prior to the first filesystem specific call, there are only two calls, `vfs_read` and `sys_read`.

In both of these the information available about the open file is given by the `struct file` object associated with the file descriptor passed to the `sys_read` call. It is possible to retrieve the request queue for the block device the file resides on from the information given in the `struct file` object. This is done by following a series of links either from the directory entry or the VFS mount structure.

Thus it is possible to rewrite either `sys_read` or `vfs_read` to perform the admission control, or provide a drop-in replacement for `sys_read` that performs it.

6.4.3 Low Latency Service Class

The low latency service class is provided for programs that have high levels of user interactiveness. These programs often have low bandwidth requirements, submitting requests only intermittently, but due to their interactive nature benefit from prompt handling of these requests. The framework for a low latency class for queues is already in place in the implementation.

The low latency class is not implemented because it does not represent original material. Under the original presentation of APEX, low latency requests are inserted into active batches when the remaining slack permits. The CFQ scheduler's treatment of filesystem metadata requests provides similar provisions by preempting any active time-slice to service these.

The current APEX implementation has a queue list set aside for low latency queues. It also silently accepts the low latency class as an I/O priority. However, they are currently given the same treatment as best effort requests.

Implementing support for the low latency service class could be done as follows. Whenever a request is added to a low latency queue with `apex_add_request`, a test is performed to see if a batch is currently being dispatched. If so, the block device driver is queried to make new calls to `apex_dispatch` and a flag is set in the APEX elevator data to indicate that the next dispatch is reserved for low latency requests. This function would then dispatch as many low latency requests as the batch slack provides room for.

6.5 Problematic Assumptions in Mixed-Media Schedulers

The design of APEX assumes more userspace context knowledge than the Linux kernel grants schedulers. This makes it difficult to provide a general purpose implementation. This section examines the problematic assumptions and what cause them to be so.

Mixed-media schedulers are often designed to use a two-level hierarchy of schedulers. In this hierarchy, there is the high level scheduler, which chooses which queues to service, and the low level scheduler which chooses how to order requests. How queues get their requests can differ from one mixed-media scheduler to another, but this general structure holds.

The faulty assumption is that the requests the high level tier sees correspond directly to the requests made by some process. Specifically, in APEX's case, that a single request represents a process's concept of a media access. For example, a process streaming continual media might attempt a read of 100KB for 1 second worth of video frames. Ideally, this read would be completed before a second has passed, so they can be displayed to the user without delay (assuming there are not any buffers). However, by the time the request reaches the scheduler, the generalized subsystems have disassembled this information and reassembled the request in other terms.

On Linux, primarily three factors contribute to this. When accessing a file sequentially, the kernel attempts to prefetch parts of the file so that the next time the process attempts to read, the data is already cached. This is a common practice in OS kernels. Thus, a request may not be associated with a real deadline, at least not yet. If the request is treated as if it does have a deadline, it will incorrectly preempt other requests with lower priority. If the request is treated as if it is not associated with a deadline, it may end up becoming delayed so much that when the process actually does need this data, it has to wait for the now low priority request to complete. Not only that, but if the request is granted resources, such as tokens, without the process ever actually making use of the data, the process will have lost the resource to speculation.

Another problem shown in section 5.4 on page 32 is that the request structures are created only just before they are handed to the scheduler. This means that the only reliable timestamp is created at the bottom of the execution path. The request might have been delayed due to having to wait for a read it depends on. In this case the delay is not insignificant, and it will not be reflected in the timestamp given on the request structure.

In some cases read accesses absolutely must be broken up into multiple parts. Block devices specify limits to the size of requests they accept. In this case, the actual requirement is for all the requests to be completed by the first deadline (which is probably later than the process actually requires). The scheduler, however, has no way to infer this. Some of the later requests may also depend on the earlier ones, in which case the deadline delay will propagate.

If the deadline were correct, the scheduler could still never know whether the requests it sees are enough to present the process with all the data it requires by the end of that deadline. So simply finding a way to propagate the deadline for requests correctly is not enough.

The consequence of these problems is that mixed-media schedulers are unable to provide certain types of deadline guarantees to programs.

6.6 Possible Improvements to the APEX Implementation

While the prototype of APEX implemented as part of this work contains many important elements from the original presentation, some improvements specific to the implementation are possible. Some limitations have been examined in this chapter, but this section examines the fundamental assumptions made in the implementation that shape the design and how they might be different.

The assumption that APEX should be a scheduler in the same way that CFQ or Deadline is, might be incorrect. Mixed-media schedulers are often arranged into two-tier scheduler hierarchies, and APEX is no exception to this. The top tier scheduler often assumes access to information that the Linux schedulers do not have access to. While it is possible to perform changes to the internal kernel API to pass this information down to the lowest layer, this change is not immediately compatible with the remaining kernel.

One other possibility is to implement a high level scheduling framework as part of the VFS layer of the kernel. With the resource information stored at this higher level, it is easier to implement queue management and admission control. Queues might be designated to correspond directly to open file objects, in which case a high degree of transparency could be achieved.

The current implementation of APEX does not assume any differentiation between synchronous and asynchronous requests, since this is not specified in the original presentation. It is clear from the previous work on I/O schedulers in the kernel that there is benefit from prioritizing synchronous I/O over asynchronous. One possibility would be to limit dispatch of asynchronous requests to the work-conserving phase.

The batch building phase currently dispatches all its requests immediately. Since the scheduler must wait for the block device driver to call for dispatch, this large dispatch prevents APEX from implementing a proper low latency service class. For a low latency service class to be possible, APEX would need to build the SCAN sorted batches internally and then dispatch these to the block device driver queue one at a time. This way a low latency request that arrives after a batch is built, but before it is completely serviced, could be inserted into the SCAN sorted batch and dispatched promptly.

Building batches internally could be used to circumvent the problems of synchronous request arrival. An anticipation heuristic that waits only when the controlling process has available tokens could be added to provide better service to processes that issue more than one read-ahead worth of requests in each I/O operation.

6.7 Summary

The implementation of APEX is done as a loadable Linux module that plugs into the elevator object of the generic block layer. It implements several functions and data structures to assist with its task. It has an elevator data structure, which holds the per device variables of the scheduler, and per process queues. It provides the block device driver with batches of requests built after the original presentation of APEX.

To assist the scheduler in correctly judging the requirements of processes, the `apex_ioprio_set` system call is provided. It allows processes to state how long deadlines for requests should be, along with their desired bandwidth and bucket depth for the token bucket model.

A limitation of the block device layer framework is that it is isolated from filesystem information, such as whether a request represents a read-ahead or data a process is immediately interested in. It is also not informed if there are more requests waiting to be built as part of the data the process is in. This negates some of the advantage of APEX's batch builder. It also prevents deadlines from work correctly when a process is interested in more than a single request.

The problematic assumptions in the original presentation of APEX that are not valid at the block device layer in Linux are not unique to that mixed-media scheduler. It is common for mixed-media schedulers to assume that they have full awareness of the extent and context of requests.

Chapter 7

Testing and Analysis of Results

This chapter examines how to test schedulers for the requirements as presented in chapter 3 on page 11 and shows the results of these tests. It presents the framework for testing and examines how this affects the methodology. The tests are performed in a series of scenarios, which each seeks to examine different aspects of the schedulers.

7.1 Hardware

Our tests are run on a consumer grade x86 machine. This type of hardware is common and cheap. As a consequence it is popular in modern systems. Where a single machine is not sufficient it is often cheaper to purchase multiple x86 machines than to buy specialized hardware. Systems as comprehensive as those that power the Google search engine have been built like this. Performing the tests on general hardware also helps focus on the limitations of the schedulers.

The hard disk drive used for testing is a Seagate Barracuda 7200.10. This

Size:	305 Gigabytes
Interface:	UltraDMA5 IDE
Seek (min/avg):	0.8 / 11.0 ms
Rotational Latency:	8.3ms
Rotational speed:	7200rpm
Sustained throughput:	70MB/s

Table 7.1: Performance details for the ST3320620A model of the Seagate 7200.10 series hard disk drive[38].

is a modern IDE interface drive. The specifications can be seen in table 7.1 on the previous page.

7.2 Operating System

The tests must ensure that the kernel does not cache the drive data which are being read when this is not appropriate. It would be appropriate in cases where multiple processes are reading from the same streams, but it would not be appropriate when it is cached due to having recently been created and written to the drive. One way to clear this cache is to perform a hardware reboot between tests. Another is to perform long sequences of reads on other files between writing the test data and beginning the tests.

The tests are run on the ext2 filesystem, which is a non-journaling version of the ext3 filesystem. Since journaling is used when writing, the difference is not expected to make an impact on the tests. The maximum filesystem block size for the ext2 filesystem is 4KB. The only Linux filesystem that is not limited to 4KB, is XFS, which is limited to the size of a page.

The size of a disk block has been shown to affect both latency and throughput[32] and the low filesystem block size can be expected to negatively impact the performance of real-time tasks by decreasing throughput. CFQ and Anticipatory counter-act some of this by implicitly increasing the request size through the anticipation heuristic.

7.3 Applications

This section examines the applications used for testing. It examines how the requirements from chapter 3 on page 11 are best tested. Testing focuses on various requirements under different circumstances. In addition to using an available application for testing disk drive performance, the specific nature of the requirements and testing environment necessitates using custom programs.

An existing filesystem and hard disk drive benchmarking suite is used to gain baseline information about the testing system and the schedulers. This suite is known as Bonnie++[1]. It tests throughput for reads and writes, along with seek times.

One of the factors that makes it difficult to use existing test suites is that the CFQ I/O priority interface is new and not commonly used. The informal tests used by the kernel developers are also inappropriate, since

[1]`http://www.coker.com.au/bonnie++/`

they often focus on balancing reads against writes rather than mixed-media requirements. Since the APEX I/O priority interface is entirely new it is not used in any existing software. A custom program is provided which uses both I/O priority systems.

To test the five requirements, the program runs several processes that attempt to read large amounts of data at varying priority levels. The tests are built around reservation based processes. To provide contention for the device resources, several processes performing a sequence of synchronous best effort requests are run. By varying the number of such processes, the schedulers's isolation can be tested. These can also use these to hamper the kernel and drive's caching facilities. The code for both the reservation based and best effort processes are available on the appended CD.

7.3.1 Bonnie++

The program Bonnie++ tests the throughput efficiency of each scheduler. Bonnie++ is a filesystem and hard disk drive benchmark suite based on on Bonnie[2]. The tests of interest are the raw throughput test known as sequential output and input. This test first creates several files of a given size, timing the bandwidth and CPU usage during writing. The second test is the rewrite test, which reads a given amount of the file then seeks back to the beginning of the read, changes the data and writes the new data back to the file. The rewrite is measured in bandwidth per second and CPU usage.

The sequential input test is performed by using the `read` library function to sequentially read the previously created files. Again, bandwidth and CPU usage are measured. The final test is random seeks, which is included for completeness. This test runs several processes in parallel performing a large number of `lseek` operations followed by reads. About 10% of the time, the data is dirtied and written back to the file. This test is designed to keep the disk continually seeking and is measured in seeks per second and CPU usage.

Bonnie++ provides important data about the schedulers and the test platform. In addition to testing the work-conserving abilities of the schedulers, it gives an idea about the maximum achievable bandwidth of the device and provides a baseline comparison of the schedulers.

[2]`http://www.textuality.com/bonnie/`

7.3.2 Reservation Based Reader

The reservation based processes are based on code used in SimStream[3]. The reservation based processes use the relevant system calls to set appropriate priorities. They then perform a series of reads while detailing reads that violate a given deadline. The length of the deadline, size of each read and priority information are all passed as arguments to the process. Two versions of the reservation based reader are created, one for use with APEX and one for use with CFQ. The source for these two versions is provided in the same file, with preprocessor variables determining which of the scheduler specific code paths should be compiled. The reader takes a series of arguments, as can be seen in table 7.2 on the next page.

After parsing and sanity checking the command-line options, the program sets its own I/O priority. In the APEX version, this is done using the custom `apex_ioprio_set` system call, as described in section 6.3 on page 69. The class, deadline, bandwidth and max bandwidth are passed as arguments. In the CFQ version, this is done using the `ioprio_set` system call. The class and priority are passed as arguments.

The program then opens the relevant files and enters a loop that reads the given amount of data and tests whether this took shorter or longer than the presented deadline. The pseudocode for this loop can be seen in figure 7.1 on page 88. If a deadline is violated, this is counted for later reference. When the file is read (in an amount specified by the option *size*), the program proceeds to log the number of violations, the amount of data read and the total time it took to read. The total time is calculated from just before the read loop is entered until just after it is completed.

When Anticipatory and Deadline run their tests, they use the CFQ version of the program, which sets I/O priority. These schedulers ignore the I/O priority of processes. Due to this, the terms *greedy reader*, *rate-limiting reader* and *randomly delayed reader* are used to refer to the reservation based reader for these schedulers. The name *resource reserving reader* is also used interchangeably with reservation based reader. In some tests, the reservation based reader is run without heightened priority, in which case it is referred to as the *foreground best effort reader*.

7.3.3 Background Best Effort Reader

The best effort processes read continually from a file, with random sleep intervals. These are meant to simulate varying levels of background activity.

[3]Original written by Erlend Birkedal, SimStream was used to test raw read response times of Linux schedulers.

Long Option	**Short Option**	**Description**
chunksize	c	Size in KB that will be used to time deadlines. Default is 512.
deadline	d	Time in microseconds for deadlines. Default is 5000.
logfile	l	Name of file to log results. Default is "SimStream.log."
size	s	Size in MB for the test file. Default is 100MB.
vbr	V	Turn on variable bitrate file reading.
delay	D	Turn on delay between calls to `read`. A value of 0 indicates that the program should read continually without delay. A value of 1 indicates that delays should ensure that the read rate is maintained at approximately 1 chunk per deadline. A value of 2 indicates random delays between 0 and 1 second.
class	x	IO priority class for use with APEX or CFQ scheduler, the numerical value might have different meanings depending on the scheduler. Default is best effort.
APEX Specific Options		
bandwidth	b	Average bandwidth in 4KB pages, this is the guarantee level forwarded to the APEX scheduler. Default is 0.
maxbw	m	Maximum bandwidth (bucket depth) to forward to the APEX scheduler. Default is 0.
CFQ Specific Options		
ioprio	i	IO priority internal to the CFQ class.

Table 7.2: Parameters for Reservation Based Readers

```
while(readsofar < totalsize){
    start=now()
    read chunksize amount of data
    stop=now()
    if(stop-start > deadline)
        deadline violated, count up
    if(not deadline violated and rate-limit delay)
        wait until deadline
    else if(random delay)
        wait randomly between 0s and 1s
    log read information
}
```

Figure 7.1: Pseudocode for the read loop in the reservation based reader.

Each read is 50KB. The background reader loops 10 times, each time performing a read, and then performs a sleep randomly distributed between 0 and 1 second. While this load does not simulate any specific workload, it places appropriate amounts of stress on the scheduler by providing bursty intervals of requests that attempt to disturb the reserved bandwidth queues.

The best effort reader is also written in C, and follows a structure similar to that of the reservation based reader albeit simplified since there are much fewer options, no special priorities and less bookkeeping.

The best effort reader takes three arguments. The first is the file to read. The second is a special file the reader uses to determine whether it should continue execution. The reader tests whether the file exists before it enters the read loop and continues execution only if it does. The last argument is the name of a file to log information to.

7.4 Test Scenarios

Our test scenarios are designed to test and compare various aspects of each scheduler. Each scenario specifies a given number of reservation based and best effort readers, along with various arguments controlling their behavior.

To find appropriate configurations for the tests, a series of preliminary tests are run, automatically pairing process priorities with deadlines and varying numbers of foreground and background processes. Initial results confirmed prior conclusions[25] that APEX, as a round-based scheduler, benefits from long round times so that request service time can be amortized.

In all cases the reservation based processes are set to read files 200MB long. The best effort readers are configured to read 256MB long files, starting back at the beginning every time the file is completely read. After best effort readers are started there is a short wait to ensure that they clear the filesystem cache as thoroughly as possible. The length of the file for the reservation based processes is chosen to give them enough time to present a representative pattern to the response times.

The schedulers tested are Anticipatory, Deadline, CFQ and APEX. The Noop scheduler is not tested, since it is an FCFS scheduler designed for devices that do not require scheduling. While neither Anticipatory and Deadline provide any explicit prioritization mechanisms, they are included for comparison. Tests running the CFQ scheduler use the ioprio system call interface to set real time readers to `IOPRIO_CLASS_RT`, with priority 0 (the best priority).

All logging is done to a separate disk to avoid disturbing the results.

The standard size of a read is set to 150KB with a deadline of 1 second, yielding a bitrate of 150KB/s. This bitrate is approximately that of a compressed video of reasonable quality. The popular online video hosting services YouTube[4] and Google Video[5] both support videos of this quality and higher. Google Video recommends that video and audio together total approximately 100KB/s[6], while YouTube recommends MPEG4 format at a resolution of 320x240 pixels[7] and variable bitrate MP3 audio. Uncompressed video at 320x240 pixels approximates about 235-280KB/s, depending on the framerate. It is not unreasonable to assume a compression ratio of 2:1 or better[8]

The read size of 150KB is also useful because it fits into one read-ahead window of maximum size, which is 256KB. This is important because of the synchronous nature of reads and how the SCAN-based schedulers dispatch requests. A read of more than a single read-ahead window is actually dispatched in multiple, synchronous requests. Thus a SCAN-based scheduler, such as APEX or Deadline, has to perform two sweeps to satisfy the read. Some of the tests run vary the size of the read to display the consequences of these situations.

For each test, response time of each read is measured in userspace. While it is possible to measure the response time in the kernel, it does not provide

[4]`http://www.youtube.com`

[5]`http://video.google.com`

[6]`https://upload.video.google.com/video_faq.html`

[7]`http://www.youtube.com/t/help_cat05`

[8]In fact, the Xvid FAQ claims to be able to achieve 200:1 ratio (`http://www.xvid.org/FAQ.14.0.html`).

the same degree of useful information, since the read may not represent actual data seen by any process (read-ahead miss) or it might be read from memory (read-ahead hit). The response time in userspace is also more important, since it is the endpoint for the data. With the response-times and read sizes, the number of violated deadlines, bandwidth and other useful statistics can be calculated.

The maximum delay is an important statistic, since it represents the amount of buffer space a continuous media program must fill to enable gapless playback. The amount of buffered data necessary is the maximum delay divided by the deadline.

7.4.1 Presentation of Results

The results for the tests will be presented as a statistical analysis of the response times of the reads, along with a statistical analysis of the bandwidths of the processes run and graphs plotting relevant data, specifically response times. The statistical analysis will include the mean, median and similar basic statistics. These together with the graphs show the response time patterns displayed by the various schedulers. The bandwidth measurements will assist in gauging the throughput efficiency of each scheduler.

The statistical analysis of the response times includes the maximum response time and percentile of response times within the deadline. This latter is included even in cases where best effort requests are measured for comparison purposes.

Since most of the tests run 50 or more programs, presenting the bandwidth of each process individually would require much space. Therefore, similar statistical analyses of the bandwidths is presented. In addition, the bandwidth of each process is summed into the aggregate bandwidth for the related processes. This does not represent an actual throughput, since the processes may not have finished running at the same. The aggregate bandwidth presents a comparison of the throughput efficiency of each scheduler, without saying anything about the maximum throughput possible.

The graphs show plots of the response time on the Y-axis and the total response times on the X-axis. A limitation of this is that the X-axis does not function as a correct comparison for programs that utilized different delays. It only functions as a time-line for comparison when none of the plotted programs added delays of their own. This limitation is primarily a problem in scenario 11, which directly compares rate-limiting delayed processes to randomly delayed processes.

Scheduler	**APEX**	**CFQ**	**Ant.**	**DL**
Write bandwidth (in KB/s):	53593	55632	56441	56673
Write CPU usage:	38%	43%	44%	44%
Rewrite bandwidth (in KB/s):	20786	23610	26329	19696
Rewrite CPU usage:	12%	14%	17%	12%
Read bandwidth (in KB/s):	56114	51187	52005	56256
Read CPU usage:	14%	12%	13%	14%
Random seeks per second:	157	155.3	167.1	154.8
Random seek CPU usage:	0%	1%	1%	0%

Table 7.3: The results of Bonnie++ on each scheduler, running with 4096MB worth of files.

7.5 Scenario 1: Bonnie++

In this scenario, Bonnie++ is run with files totalling 4096MB. The tests Bonnie++ run are detailed in 7.3.1 on page 85.

This test shows the total bandwidth throughput of each scheduler. It also shows some information about the seek behavior of each scheduler.

7.5.1 Results

The results of the tests are shown in table 7.3. The write bandwidth of each scheduler is approximately the same, with APEX slightly below the rest. The difference is under 6% from the highest result and further testing shows that it is within the bounds of natural variance. The CPU usage during all the write tests are comparable. This usage is dominated by the kernel's `pdflush` daemon, which writes dirty pages back to disk.

The rewrite test shows slightly higher bandwidth for CFQ and Anticipatory. The lower bandwidth for APEX can be explained by the lack of prioritization of reads over writes. Since the prototype is not designed to test optimization of synchronous and asynchronous commands, this is natural. The Deadline scheduler's performance can be explained by deceptive idleness causing it to preemptively choose to write when it might be more advantageous to wait for a new read. It follows from this that the performance by CFQ and Anticipatory is due to the use of the anticipation heuristic in both cases. The higher CPU usage for these two is explained by the higher bandwidth.

The read bandwidth, which is the focus of this scenario, shows that APEX and Deadline have a slightly higher maximum read throughput than CFQ

Reservation based readers:	50
Read delay:	None
Size of Read:	150KB
Deadline:	1000ms
Read delay:	None
Background best effort readers:	50
Scheduler specific configuration:	
APEX	
Bandwidth allocated (each client):	37 pages/s
Bucket depth (each client):	37 pages

Table 7.4: Scenario 2: Configuration

and Anticipatory. This is explained by the non-work-conserving nature of the anticipation heuristic. Like Deadline, APEX never delays calls to dispatch requests when there are no resource reserving queues active. The CPU usage again reflects bandwidth.

The random seek test shows slightly higher performance for Anticipatory, which can be explained by correct anticipation and delaying of writes. CFQ's time-sliced nature works against it in this test.

Due to the variance in results, the benchmarks produced by Bonnie++ are used only has rough guidelines. The tests do not achieve the sustained bandwidth reported by the manufacturer, which can be attributed to filesystem and kernel overhead.

Scenario 1 provides a best-case read throughput which further tests can be measured against. It shows that the work-conserving nature of APEX and Deadline give them higher potential read throughput.

7.6 Scenario 2

Scenario 2 is the first of a series of tests run using the reservation based reader to test the real-time performance of each scheduler. The first test is run with 50 reservation based readers without any read delay. Scenario 2 provides further insight into the fundamental behavior of each scheduler. It also provides a baseline against which to compare the other scenarios in this series. The configuration details for the tests can be seen in table 7.4.

This test is run twice for CFQ, once each time with the reservation based readers running at priorities 0 and 7. These are the highest and lowest priorities, respectively. For real-time processes, the priorities determine how

Scheduler	**APEX**	**CFQ pri0**	**CFQ pri7**
Mean response time:	327922	107193	123848
Standard Deviation:	270203	949378	481472
1st Quartile response time:	434	1544	1523
Median response time:	510576	1595	1593
3rd Quartile response time:	545143	2830	2839
Minimum response time:	71	67	68
Maximum response time:	1219817	14635339	3513422
Percentile within deadline:	99.45%	98.76%	94.01%

Table 7.5: Scenario 2: Analysis of response times for APEX and CFQ (priority 0 and 7). All results except percentile in μs.

Scheduler	**Anticipatory**	**Deadline**
Mean response time:	128650	720495
Standard Deviation:	939101	592671
1st Quartile response time:	1539	426
Median response time:	1593	1110069
3rd Quartile response time:	2829	1209765
Minimum response time:	67	69
Maximum response time:	17020258	2700360
Percentile within deadline:	98.18%	40.35%

Table 7.6: Scenario 2: Analysis of response times for Anticipatory and Deadline. All results except percentile in μs.

long the allocated time-slice for the queue is. The formula for time-slice length is given equation 5.1 on page 50. Following the description from 5.6.5 on page 50, the time-slice for priority 0 queues is calculated to be 180ms and for priority 7 queues to be 40ms. Running scenario 2 once for each of these priorities shows the effect of adjusting the length of the time-slice for each request.

7.6.1 Results

Table 7.5 and table 7.6 show that APEX has the best results in Scenario 2 by having the highest percentile of operations serviced within the deadline and the lowest maximum response time. CFQ at priority 0 and Anticipatory show similar performances, with high maximum response times, but at least 75%

Scheduler	**APEX**	**CFQ pri0**	**CFQ pri7**
Mean bandwidth:	456.94	1400.20	1208.61
Standard Deviation:	0.88	21.90	16.05
1st Quartile bandwidth:	456	1374	1197
Median bandwidth:	457	1402	1204
3rd Quartile bandwidth:	457	1422	1219
Minimum bandwidth:	456	1365	1190
Maximum bandwidth:	460	1432	1256
Aggregate bandwidth:	22847	70009	60430

Table 7.7: Scenario 2: Analysis of bandwidth for each resource reserving process under APEX and CFQ (priority 0 and 7). All numbers in KB/s.

Scheduler	**Anticipatory**	**Deadline**
Mean bandwidth:	1168.40	208.01
Standard Deviation:	56.19	0.11
1st Quartile bandwidth:	1150	207
Median bandwidth:	1157	207
3rd Quartile bandwidth:	1170	208
Minimum bandwidth:	1137	207
Maximum bandwidth:	1528	208
Aggregate bandwidth:	58420	10400

Table 7.8: Scenario 2: Analysis of bandwidth for each resource reserving process under Anticipatory and Deadline. All numbers in KB/s.

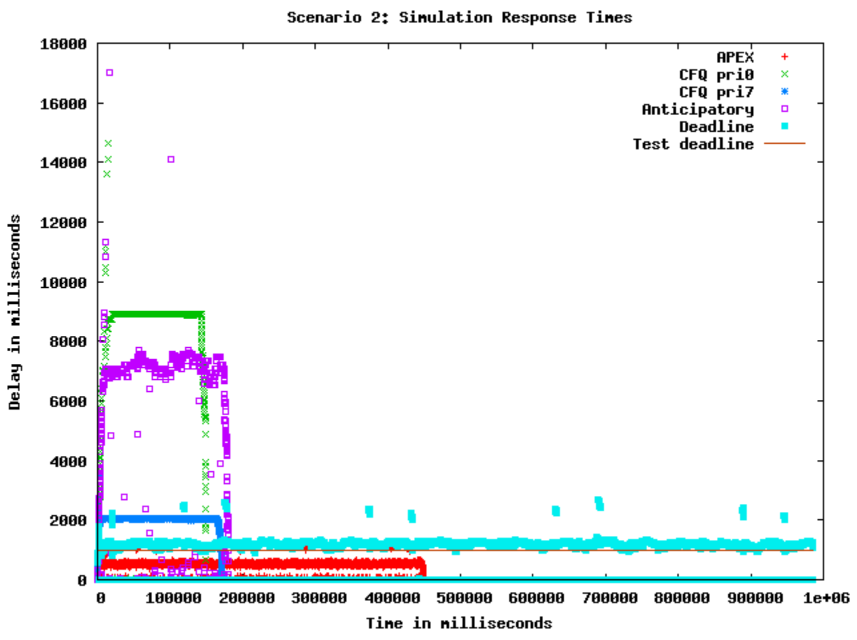

Figure 7.2: Scenario 2: Response times for all resource reserving readers under each scheduler setting.

of operations serviced within a few milliseconds. CFQ at priority 7 maintains fewer deadlines, but has a lower maximum response time. Deadline performs worst, with the highest mean response and fewest deadlines maintained.

The plotted response times in figure 7.2 show APEX to have response times clustered around 400-600ms. The higher mean response time, but low standard deviation shown by APEX are both due to the batching nature of the scheduler. The greedy nature of the readers in this test ensures that every batch is full.

Both priority settings of the CFQ scheduler show similar mean delay, but higher standard deviation for priority 0. Comparing the two in figure 7.2 shows that both tests display some requests with exceptionally high response times, but that these are higher for the priority 0 test. This is explained by the time-sliced nature of the CFQ scheduler. When running with priority 0, the processes are allocated longer time-slices, giving them longer exclusive access to the device, which the greedy nature of the reader configuration in this scenario ensures. The maximum delay displayed is the delay each process experiences while the other 49 real-time processes are being serviced. This is easily calculated. Since the time-slice is 180ms for each priority 0 process, a round of 50 processes with take 9s to complete. This fits very well with the plotted points for the CFQ priority 0 test.

In the priority 7 test, the processes are allocated shorter time-slices, which translates to shorter delays for the other processes. However, the processes must stop and wait for the remaining processes more often. This is also visible in percentile of requests that maintained the desired deadline. Here the priority 0 processes maintained the deadline for more requests. Given that the time-slice is 40ms for each priority 7 queue, this gives about 2s for a complete round. Again, the cluster of points plotted around 2s fit.

The Anticipatory scheduler provides a low mean response time, but at the cost of the longest delays. This can be explained by the anticipation heuristic, which will strongly favor continued service of each process, since they perform consistently sequential accesses with low 'think time'. This situation is similar to that under CFQ, but the Anticipatory scheduler does not have the time-slicing provisions to cut off this exclusive access. Thus the maximum response times for Anticipatory are scattered around 6-8s

For both CFQ and Anticipatory, most (at least 75%) of the requests are serviced in approximately 3ms or less. This is explained by the anticipation heuristic. Each process experiences very low response time for most of its requests, due to the greedy nature of the resource reserving reader in this scenario. The scheduler continually rewards the process for promptly issuing new requests each time previous requests have been serviced.

The response time for the first quartile of requests is much lower for APEX and Deadline than the other schedulers. Response time as low as 426μs indicates a buffered read. By not implementing an anticipation heuristic, APEX and Deadline enable the filesystem to perform more read-ahead operations, resulting in more cache hits. During a series of anticipation reads, no other process is given any access to the device, preventing them from performing any action. During batched dispatch, filesystem read-ahead commands for many processes can be added at the same time.

Table 7.7 on page 94 and table 7.8 on page 94 show the bandwidth granted each resource reserving process run. APEX provides approximately equal amounts of bandwidth to each process. This is a consequence of the round based nature of the scheduler. Since each process has the same token rate, they are granted the same share of the token-based batch building phase. In addition, the work-conserving phase dispatches equal amounts of requests from each queue until the expected round length is met. The round based, scanning nature of Deadline gives it similar performance characteristics, but with worse bandwidth, since it shares the bandwidth equally with the background best effort readers.

The increased exclusive access provided by CFQ at priority 0 gives it the best mean bandwidth. By avoiding seeks, the CFQ scheduler maximizes the bandwidth potential of the device. The cost of this is the long response times. CFQ at priority 7 provides high bandwidth throughput as well, but lower than priority 0 since it seeks to satisfy other processes's requests more often. The variation in both these cases is down to which processes complete I/O first.

The bandwidth granted by Anticipatory varies more than the other schedulers. The processes with most bandwidth are scheduled more often than the remaining processes. Since Anticipatory chooses the next request by SCAN or FIFO order when it ends anticipation, the process chosen will be biased by the positioning on disk or request arrival.

CFQ and Anticipatory provide much better throughput than the scanning schedulers. The background best effort requests are not counted in the bandwidth overviews, but both CFQ and APEX completely starved the background best effort readers. Despite this, Anticipatory showed better throughput for the greedy readers without any prioritization framework. This is due to the implicit prioritization of greedy requests, since they are rewarded by the scheduler when anticipation waits promptly receive new requests. The improved throughput comes at the cost of the higher maximum response time.

Disks with higher rotation rates or faster seeks will provide overall lower response times for APEX and Deadline by increasing throughput or decreasing rotational delay and seek time. The maximum delay of CFQ and Anticipatory would not be affected by this, since this is not determined by throughput or seek time. Increased rotation rates would increase throughput, which would provide better service to each process for the duration of it's time-slice or exclusive access to the Anticipatory scheduler. Since CFQ and Anticipatory both use the anticipation heuristic to avoid seeking whenever possible, they would not benefit as much from faster seek times.

The performance of CFQ and Anticipatory in this test is highly dependant on the sequential and defragmented nature of the files being read. Fragmented files will cause the anticipation heuristic to anticipate new reads less often leading to more seeking. It is not unreasonable to assume that files are defragmented on a reasonably static filesystem, but for active filesystems with many small files that are often appended to, this is not reasonable.

Scenario 2 shows that the scanning schedulers, APEX and Deadline, provide more reliable service in high contention situations than the anticipating schedulers. CFQ and Anticipatory show themselves to be better in situations with short, bursty traffic. This is not suited for serving large, high bandwidth multimedia streams or file servers, but better suited for interactive situations, such as web servers or games. Both CFQ and APEX showed they could maintain deadlines for a large number of streams, but CFQ occa-

Reservation based readers:	50
Read delay:	None
Size of Read:	150KB
Deadline:	1000ms
Read delay:	None
Background best effort readers:	100
Scheduler specific configuration:	
APEX	
Bandwidth allocated (each client):	37 pages/s
Bucket depth (each client):	37 pages

Table 7.9: Scenario 3: Configuration

sionally provided very high delays when time-slices ran out. For CFQ priority 7, which is the lowest priority, actually provided better amortized service by seeking more often than priority 0. Anticipatory gives implicit priority to processes that deliver continual streams of synchronous requests.

7.7 Scenario 3

Scenario 3 shows the consequences of exhausting the available resources for the allocation of new requests. The request queue structure in the Linux block device layer specifies a limit to the number of requests that can be concurrently allocated, and when that limit is met processes must wait for further dispatches before they may queue requests. Scenario 3 shows the consequence of reaching that limit. The configuration of the scenario is similar to that of scenario 2, except that the number of background readers is increased to reach the default limit of 128 active requests. The configuration is shown in table 7.9.

The request limit is important because it shows how the performance of the schedulers changes when the system is put under pressure. The request limit is typically low to prevent situations where swapping must be done to perform I/O, which would cause much worse deterioration in performance.

7.7.1 Results

The results for scenario 3 are shown in table 7.10 on the facing page. APEX performs slightly worse than in scenario 2, with higher mean response times and larger variability. More deadlines are violated and the maximum re-

Scheduler	**APEX**	**CFQ pri0**	**Antic.**	**DL**
Mean response time:	568389	182186	246830	2008861
Standard Deviation:	319660	543430	956127	713584
1st Quartile response time:	292772	1721	1773	1633274
Median response time:	634301	1790	1845	1750012
3rd Quartile response time:	725856	2819	2964	1857902
Minimum response time:	74	68	67	75
Maximum response time:	2243521	3014956	11550718	5605657
Percentile within deadline:	91.44%	90.14%	93.64%	0.09%

Table 7.10: Scenario 3: Analysis of response times. All results except percentile in μs.

Scheduler	**APEX**	**CFQ pri0**	**Antic.**	**DL**
Mean bandwidth:	263.82	823.05	636.97	74.63
Standard Deviation:	1.56	4.73	142.29	0.12
1st Quartile bandwidth:	262	819	503	74
Median bandwidth:	263	822	584	74
3rd Quartile bandwidth:	264	825	772	74
Minimum bandwidth:	262	819	477	74
Maximum bandwidth:	270	839	890	74
Aggregate bandwidth:	13190	41152	31848	3731

Table 7.11: Scenario 3: Analysis of bandwidth for each resource reserving process. All numbers in KB/s.

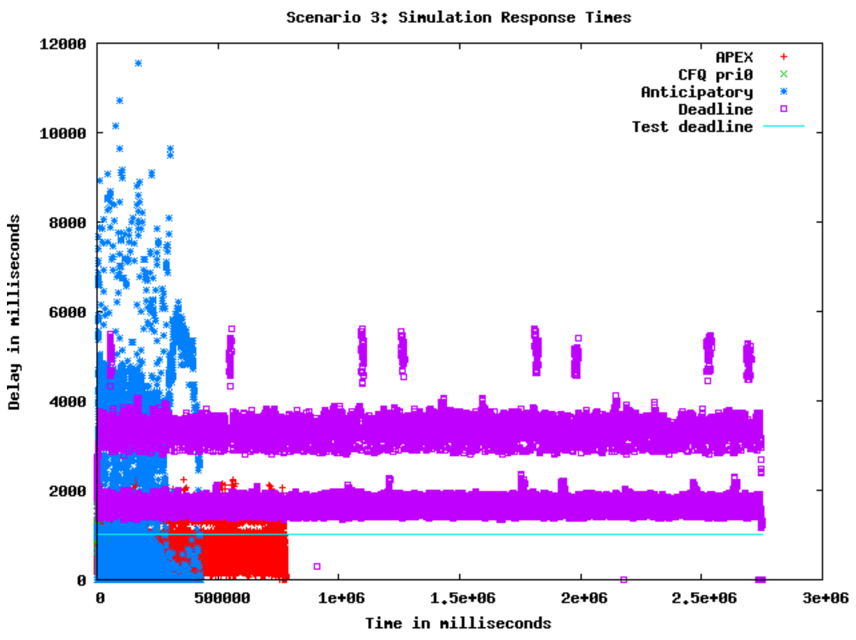

Figure 7.3: Scenario 3: Response times for all resource reserving readers under each scheduler setting.

sponse time is increased. For CFQ, the maximum response time has dropped, reflecting the performance of the priority 7 test in scenario 2. Anticipatory has a higher mean response time than in scenario 2, but a lower maximum with more deadlines violated. This change is similar to that exhibited by CFQ. Deadline shows a higher mean response time and higher max, with almost no deadlines upheld.

The results in scenario 3 reflect in part a limitation provided by the block device layer. Both CFQ and APEX changed performance profile, without providing better service to the background best effort readers. Both CFQ and APEX starved the background best effort readers while the resource reserving processes ran in both scenario 2 and 3. However, in scenario 3 the number of requests in the scheduler would regularly reach the limit imposed by the block device layer. Each request queue specifies a maximum number of requests that it will hold. While the scheduler may override this by returning a special value to `elv_may_queue`, it is not recommended to do this without reason. CFQ does so if the concerned request is associated with a queue anticipating new requests.

The default maximum number of requests a request queue allows is 128. Since this test runs 150 processes all attempting synchronous I/O, this number will quickly be met. Since the requests are under the size of a read-ahead window (256KB), only one request will be queued per process. Since CFQ and APEX starve the background best effort readers, they will continually hold 100 of these requests. APEX does not have any special behavior implemented for full request queues, so it will be forced to dispatch suboptimal batches. The delay for the requests causes the deadline to run down before the scheduler can see the request. This causes batches to contain fewer requests, which prevents APEX from efficiently amortizing the request service time.

For CFQ, the results are different. This is the result of two factors. When a process is denied resources when attempting to allocate a request, it sleeps until resources are made available. When it is awoken, the block device layer gives it special access to circumvent limitations for 20ms or 32 requests, whichever comes first. This works in synergy with CFQ's time-slicing system. However, instead of being given 180ms, the block device layer's limit of 20ms to circumvent the request limit controls the length of the time-slices. This causes much more seeking and performance degenerates accordingly. While the single highest delay for CFQ is approximately 3s, there is a longer string of response times around 2s that are not visible under the Deadline response times in figure 7.3 on the preceding page.

For Anticipatory and Deadline, the situation is different since they do not starve the background best effort readers in the same way. For Anticipatory, the resource limit causes the anticipation heuristic to fail more often, as background requests with random delays accumulate while the anticipating processes dominate I/O, locking out other greedy readers.

In figure 7.3 on the facing page, Deadline shows two separate batches, clustered just under 2s and 4s respectively. This reflects the resource limit preventing Deadline from filling effectively filling the SCAN queue.

Table 7.11 on page 99 shows the bandwidth of all the schedulers is adversely affected by the resource limit. This is due to the excessive seeking caused by the request queue being filled.

Both CFQ and APEX experienced adverse effects from the lack of resources available in this test. Both these schedulers maintained their prioritization policies despite the lack of resources. When further tests demand it, the maximum number of requests is increased through the sysfs interface.

7.8 Scenario 4

Scenario 4 tests how each scheduler handles I/O access consistently broken into multiple request objects. The maximum request size of the disk drive

Reservation based readers:	35
Size of Read:	600KB
Deadline:	1000ms
Read delay:	None
Background best effort readers:	35
Scheduler specific configuration:	
APEX	
Bandwidth allocated (each client):	150 pages/s
Bucket depth (each client):	150 pages

Table 7.12: Scenario 4: Configuration

Scheduler	**APEX**	**CFQ pri0**	**Antic.**	**DL**
Mean response time:	856573	300307	351074	1894232
Standard Deviation:	192274	1297773	1242389	435029
1st Quartile response time:	705568	7676	7673	1546326
Median response time:	732186	8642	8643	1672597
3rd Quartile response time:	1050084	8957	8970	2290977
Minimum response time:	387	1520	405	382
Maximum response time:	1469137	13076693	9048994	3331697
Percentile within deadline:	62.11%	95.06%	92.83%	0.28%

Table 7.13: Scenario 4: Analysis of response times. All results except percentile in μs.

used is 512KB[9], thus read requests of 600KB must be broken into multiple `struct request` instances to be serviced. While two requests would be sufficient, the limit on the read-ahead window more commonly breaks operations into requests representing 256KB.

Since admission control is not implemented, the number of readers is reduced so that APEX is able to maintain the necessary throughput per process to meet the bandwidth requirement. Thirty-five resource reserving processes was found to be an appropriate number for this test, with an equal number of background best effort readers. The configuration is shown in table 7.12.

Scheduler	APEX	CFQ pri0	Antic.	DL
Mean bandwidth:	697.96	1989.24	1704.44	315.99
Standard Deviation:	1.19	26.74	59.83	0.23
1st Quartile bandwidth:	696	1969	1678	316
Median bandwidth:	698	1988	1692	316
3rd Quartile bandwidth:	698	2007	1706	316
Minimum bandwidth:	696	1950	1665	315
Maximum bandwidth:	698	2089	1988	316
Aggregate bandwidth:	24428	69623	59655	11059

Table 7.14: Scenario 4: Analysis of bandwidth for each resource reserving process. All numbers in KB/s.

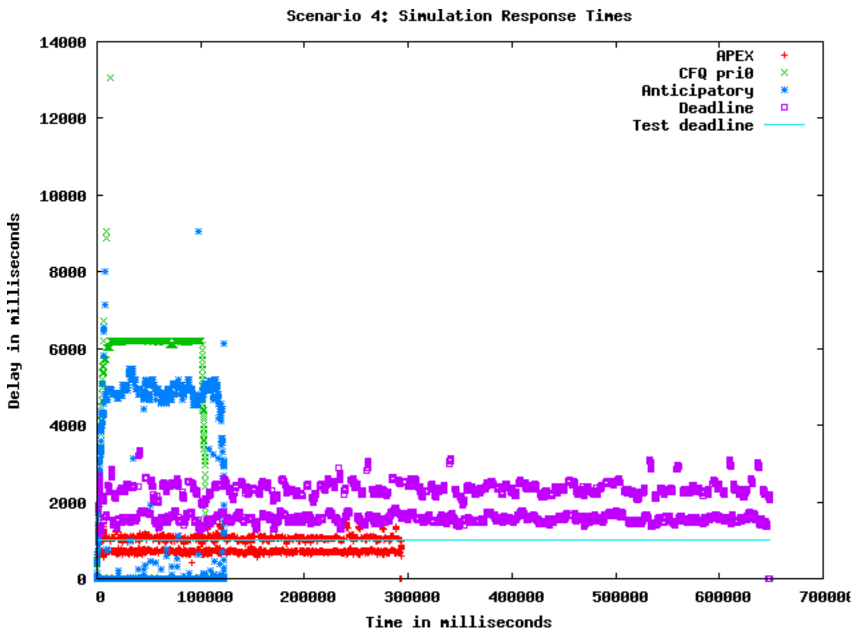

Figure 7.4: Scenario 4: Response times for all resource reserving readers under each scheduler setting.

7.8.1 Results

As section 6.5 on page 80 predicts, APEX has difficulties coping with I/O operations broken into multiple request objects. Table 7.13 on page 102 shows that the mean response time is close to the deadline, at over 850ms, and only around 60% of the deadlines are maintained. CFQ and Anticipatory maintain their performance from previous tests. Deadline suffers similarly to APEX, only worse, since it is unable to prioritize.

It is visible by the clustering around 700ms and just over 1s in figure 7.4 on the previous page that APEX services operations in two or three rounds. Since each round represents approximately 256KB read for each process, no operations will be completed after only a single round, but if there is some cached data, two rounds may be enough. This is shown by the cluster of responses at approximately 700ms. This is a disadvantage of APEX's scanning nature in the context of the Linux kernel. Since read-aheads are delivered synchronously, it cannot correctly amortize the performance of requests across single operations. That is, a small number of processes that issue large requests cause more seeking than a high number of processes that issue small requests.

Since APEX cannot differentiate between the first and final request in an I/O operation, it fails to correctly prioritize. The implemented work-conserving policy attempts to compensate for this by not running a work-conserving phase if there are pending resource reserved requests. This functions well for the resource reserving requests since they can dispatch full requests if they only have some tokens. While reworking these policies is possible, they will not change the fundamental problem. One could prevent resource reserved queues from dispatching without enough tokens, and compensate by running work-conserving phases to read from resource reserved queues first, but this would not improve latency.

CFQ and Anticipatory both take advantage of the sequential nature of arriving requests, as they did in scenario 2. For CFQ, the cluster of long delays is just over 6s, which fits with the calculation of 180ms per process, as $34 * 180ms = 6120ms$. The clustering of high response times for Anticipatory is correspondingly lowered.

Deadline gives poor performance in this test, maintaining almost no deadlines. Since it is a scanning scheduler, it suffers the same problem as APEX, that each operation takes multiple rounds to complete. This is exacerbated by the lack of prioritization, which causes the background best effort readers to prolong the rounds.

[9] This is reported at boot-time by the kernel hardware detection probe, and is available at runtime via the sysfs interface.

Table 7.14 on page 103 shows that APEX delivers about 15% more bandwidth to each process than it requires to maintain the desired throughput. The aggregate bandwidth is slightly higher than scenario 2, which can be explained by each process necessitating dispatch of requests to the drive for every operation. This causes higher throughput by dispatching more requests per scan round.

CFQ and Anticipatory maintain the approximate bandwidth throughput from scenario 2. CFQ is distributes the bandwidth more evenly than Anticipatory, since it has explicit criteria for fairness in the time-slicing system. While both CFQ and APEX starve their background readers completely, Anticipatory maintains a high throughput while providing intermittent, albeit very service to these. Thus the aggregate bandwidth reported for APEX and CFQ represents all the bandwidth usage during the test, while Anticipatory and Deadline are slightly under reported. Despite this, Anticipatory records over twice the bandwidth used by APEX. This is due to the anticipation heuristic allowing the scheduler to avoid seeking.

While the aggregate bandwidth is representative, it does not represent any single achieved throughput, since the time the tests are recorded over may vary. That is, some processes finishing earlier than others will record higher bandwidth, but the total bandwidth for the period is determined by the process that took the longest.

The limitations provided by the read-ahead model in Linux prevents the scanning schedulers from correctly amortizing I/O operations over 256KB in size. CFQ and Anticipatory are designed with this in mind and therefore perform similarly when dispatching I/O operations above and below this limit.

7.9 Scenario 5

The goal of scenario 5 is to see how the schedulers perform when increasing the size of each read and correspondingly increasing the deadline. The bandwidth required is maintained. Table 7.15 on the next page shows the size of the read and deadline have been doubled compared to scenario 1. The other settings are maintained.

7.9.1 Results

The analysis of the results from scenario 5 are shown in table 7.16 on the following page. APEX performs well, maintaining all the deadlines. CFQ performs similar to scenario 2, with high maximum response times and most

Reservation based readers:	50
Size of Read:	300KB
Deadline:	2000ms
Read delay:	None
Background best effort readers:	50
Scheduler specific configuration:	
APEX	
Bandwidth allocated (each client):	37 pages/s
Bucket depth (each client):	37 pages

Table 7.15: Scenario 5: Configuration

Scheduler	**APEX**	**CFQ pri0**	**Antic.**	**DL**
Mean response time:	669103	215032	259958	1465095
Standard Deviation:	231610	1339237	1326322	504813
1st Quartile response time:	537289	3191	3192	1177017
Median response time:	562120	4325	4319	1243520
3rd Quartile response time:	610836	4624	4690	1311398
Minimum response time:	395	1561	431	391
Maximum response time:	1699660	20018519	18071166	3657985
Percentile within deadline:	100.00%	97.53%	96.40%	78.04%

Table 7.16: Scenario 5: Analysis of response times. All results except percentile in μs.

Scheduler	**APEX**	**CFQ pri0**	**Antic.**	**DL**
Mean bandwidth:	447.95	1393.55	1157.31	204.71
Standard Deviation:	0.93	22.61	39.83	0.22
1st Quartile bandwidth:	447	1374	1137	204
Median bandwidth:	447	1393	1150	204
3rd Quartile bandwidth:	449	1412	1157	204
Minimum bandwidth:	447	1365	1125	204
Maximum bandwidth:	451	1432	1402	205
Aggregate bandwidth:	22397	69677	57865	10235

Table 7.17: Scenario 5: Analysis of bandwidth for each resource reserving process. All numbers in KB/s.

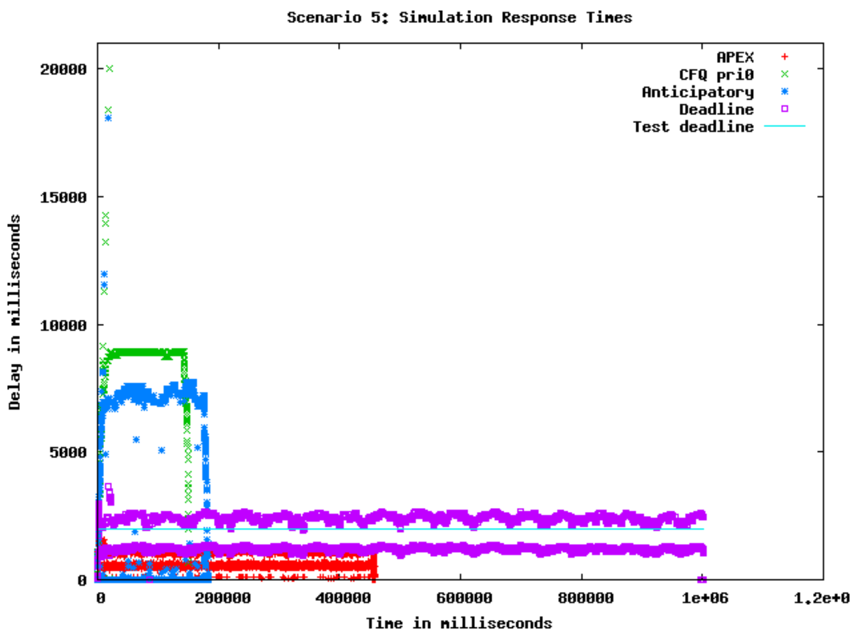

Figure 7.5: Scenario 5: Response times for all resource reserving readers under each scheduler setting.

requests serviced within a few milliseconds. The same holds for Anticipatory. Deadline copes better with longer deadlines, maintaining almost 80% of the deadlines.

APEX is able to maintain the deadline for all the requests because it can perform the two scan sweeps needed within 2s. The I/O operations in this test require two sweeps to service in the worst case because they are broken into two requests. The maximum size of a read-ahead is 256KB, which is not enough to fit the whole 300KB for the read. In most cases, parts of the read will already be cached, which is shown by the thick clustering of response times around 500-600ms in figure 7.5. There is another clustering around 1s, which is partially hidden under the Deadline response times. This clustering is formed by operations that required two requests to service.

CFQ again shows the delay to be determined by the sum of time-slices for the remaining processes in the same class and priority. The median response time is, however, slightly longer than in scenario 2, reflecting the longer reads. Anticipatory shows similar results to CFQ. The clustering of maximum response times is around 6-8s, which is the same as in scenario 2. Again Anticipatory shows that it implicitly prioritizes greedy readers.

Deadline shows better results in scenario 5 than in scenario 2, primarily

Reservation based readers:	50
Size of Read:	75KB
Deadline:	500ms
Read delay:	None
Background best effort readers:	50
Scheduler specific configuration:	
APEX	
Bandwidth allocated (each client):	36 pages/s
Bucket depth (each client):	36 pages

Table 7.18: Scenario 6: Configuration

due to the longer deadline. Figure 7.5 on the preceding page shows that response times for Deadline are separated into two groups. The lower group, around the 1s mark, shows the operations which are partially cached from previous rounds. The clustering just over 2s shows the results which required two rounds to service. The rounds are both longer than those exhibited by APEX because Deadline cannot separate out the best effort requests from the background readers.

The bandwidth shown in table 7.17 on page 106 is similar to the bandwidth from scenario 2 for all four schedulers.

Scenario 5 supports the conclusions from scenario 2. The anticipation-based schedulers provide better throughput than the scanning schedulers, but this comes at the cost of very high maximum response times. The scanning schedulers benefit from the increased request deadline, because missed deadlines are closer to the deadline than the anticipating schedulers. This makes them more suited to delivering continual multimedia streams, since less buffering is required to maintain continual playback.

7.10 Scenario 6

The goal of scenario 6 is to see how the schedulers respond to decreasing the size of each read, while also decreasing the deadline. The bandwidth required is maintained. Table 7.18 shows the size of the read and deadline have been halved compared to scenario 1. The other settings are maintained.

The number of pages for APEX is reduced by 1 to 36 due to integer math rounding down. Since the implementation of APEX uses token rate per jiffy and the test machine is configured with $HZ = 1000$, this does not make any difference. Both 1000/37 and 1000/36 are rounded down to 27.

Scheduler	APEX	CFQ pri0	Antic.	DL
Mean response time:	175809	53540	64643	369419
Standard Deviation:	262661	672064	668683	551695
1st Quartile response time:	133	114	113	133
Median response time:	240	1401	1400	241
3rd Quartile response time:	543753	1479	1477	1119506
Minimum response time:	71	68	67	70
Maximum response time:	1075064	14439291	11651019	2333044
Percentile within deadline:	69.64%	99.37%	99.07%	68.92%

Table 7.19: Scenario 6: Analysis of response times. All results except percentile in μs.

Scheduler	APEX	CFQ pri0	Antic.	DL
Mean bandwidth:	426.22	1398.68	1161.31	202.99
Standard Deviation:	0.63	22.05	40.34	0.10
1st Quartile bandwidth:	425	1374	1137	202
Median bandwidth:	425	1402	1150	202
3rd Quartile bandwidth:	426	1422	1170	202
Minimum bandwidth:	425	1365	1107	202
Maximum bandwidth:	428	1432	1383	203
Aggregate bandwidth:	21311	69933	58065	10149

Table 7.20: Scenario 6: Analysis of bandwidth for each resource reserving process. All numbers in KB/s.

7.10.1 Results

Scenario 6 shows that APEX performs worse when dealing with shorter round times. Almost 70% of the results violate the deadline, despite at least half the results being serviced from the read cache. CFQ and Anticipatory both perform better than in scenario 5, violating almost no deadlines. Deadline performs worse than in scenario 5, but not much, since the smaller request size allows more requests to be serviced from read-ahead.

In table 7.19 APEX shows at least half the I/O operations served from memory. This is a natural consequence of the operation size compared to the maximum read-ahead window. Approximately 3 operations of 75KB fit into one read-ahead window of 256KB. Therefore, a single operation triggers the read of enough data to service the next two. This explains the number

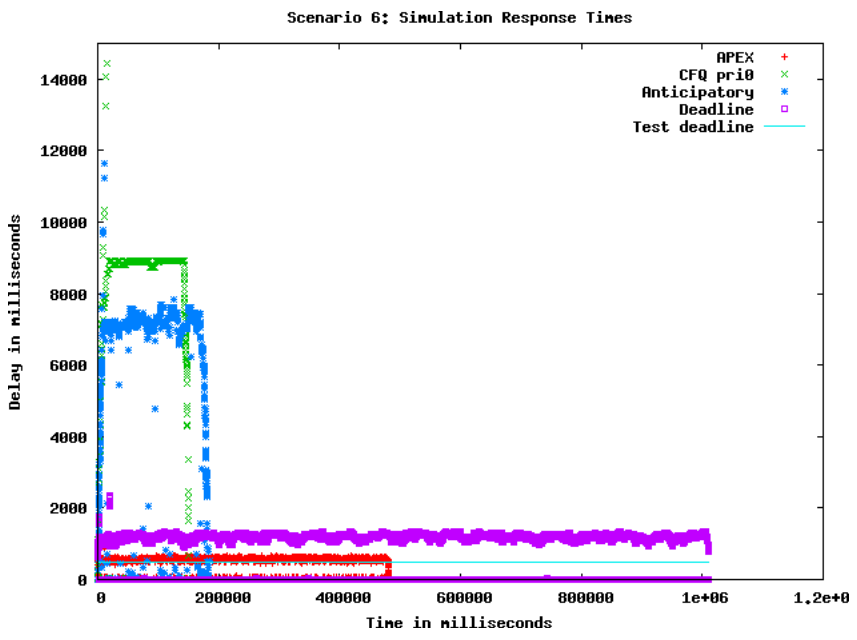

Figure 7.6: Scenario 6: Response times for all resource reserving readers under each scheduler setting.

of deadlines maintained. Figure 7.6 shows that the bulk of requests that are not serviced from the buffer, are serviced around or just past the deadline. This follows from scenario 2, since the size and number of the requests being sent to the block device are approximately the same. The processes are only noticing that delay in every few operations, since this is when the data must actually be fetched from disk.

In contrast to APEX, CFQ and Anticipatory seemingly benefit from the lower operation size by maintaining a higher percentage of the deadlines. In fact, the number of deadlines violated is approximately the same, but since their violations are constrained by the scheduler rather than the request pattern in this case, and there are more I/O operations, the percentage drops. That is, CFQ is still delaying due to expiring time-slices, clustering high response results around 9s, but since there are more operations counted per time-slice, the percentage of missed deadlines is lower. The results for Anticipatory are similar. The anticipation heuristic maintains the I/O stream for each process approximately as long, but the length of this period is the same, more operations are processed.

Neither CFQ nor Anticipatory shows as many operations served from the read cache as APEX or Deadline. Only the first quartile response time in-

Reservation based readers (total):	100
- CBR readers:	50
- VBR readers:	50
Read delay:	Rate-limiting
Background best effort readers:	50
Size of Read:	150KB
Deadline:	1000ms
Scheduler specific configuration:	
APEX	
Bandwidth allocated (each client):	37 pages/s
Bucket depth (each client):	37 pages

Table 7.21: Scenario 7: Configuration

dicates a read from memory, whereas the median time indicates a wait for the drive. This is due to the aggressive dispatch by the schedulers, preventing merging from building up full 256KB requests and the exclusive access framework preventing background reads from happening while a process is waiting.

Deadline shows a situation similar to that shown by APEX. While the cluster of requests shown in figure 7.6 on the preceding page is grouped around 1s, well after the deadline, about 2/3 of the operations are serviced from memory.

Table 7.20 on page 109 shows the bandwidth provided by each scheduler to the processes. Again CFQ and Anticipatory deliver the most bandwidth. While the mean response time for APEX and Deadline is lower than in previous tests, the bandwidth is approximately the same, since the length and number of seeks necessary to serve all the requests is about the same.

Despite providing enough bandwidth to serve all the requests, APEX and Deadline are unable to maintain more than 70% of the deadlines due to the length of the scaning rounds. CFQ and Anticipatory show the same behavior as in scenarios 2, 4 and 5, which reflects their design parameters. The scanning schedulers are shown to benefit from longer deadlines, which follows from their strategy of batching as many requests as possible when dispatching.

7.11 Scenario 7

Scenario 7 runs 100 resource reserving readers with the delay setting active. The configuration can be seen in table 7.21 on the previous page. The delay configured is the type that instructs the readers to pause until the deadline is reached. This way the readers attempt to maintain maintain their desired bitrate with failures causing playback to be delayed. In addition, half the resource reserving readers run with the variable bitrate setting enabled.

This scenario tests the schedulers under periodic loads. The resource reserving requests no longer continually saturate the scheduler with requests. Since this access pattern affects the results in previous tests, the new access patterns reveal new information.

In continuous media playback situation, a violated deadline might require that the playback software read more aggressively for a short while to replenish buffers. This is not assumed to be the case in scenario 7. When a deadline is violated, the reader simply proceeds to immediately issue the next request, rather than permit any further delay.

Half the resource reserving readers are run with the VBR setting enabled. This will regularly vary their reads from approximately half the bitrate to approximately one and a half times the bitrate. This challenges the schedulers by making the I/O profile of the processes less predictable.

The number of pages allocated each process remains at 37 pages/s, with the same bucket depth. This means the VBR processes does not have enough pages to guarantee service during peak bandwidth. However, since the requests stay within 256KB, this does not affect the results, as is shown by scenarios 8 and 9 later in this chapter.

Since this test runs more than 128 processes, the maximum number of requests the queue will allocate is increased. This increase is enough to ensure that the results are not impacted by lack of available resources.

7.11.1 Results

Table 7.22 on the facing page and table 7.23 on the next page show that APEX performs well for both the CBR and VBR processes, maintaining almost all the deadlines and in fact servicing at least the third quartile of the operations from memory. CFQ shows worse performance than in previous tests, with under 90% of deadlines maintained. It services at least half the requests from memory, but shows higher third quartile results, with slightly worse results for VBR than CBR. The situation for Anticipatory is similar, with an even higher mean response time than CFQ. Deadline performs worst of all four, with the highest mean and fewest deadlines maintained.

Scheduler	**APEX**	**CFQ pri0**	**Antic.**	**DL**
Mean response time:	31739	357791	405176	601228
Standard Deviation:	83873	520848	666529	681641
1st Quartile response time:	460	370	376	383
Median response time:	474	594	555	559226
3rd Quartile response time:	911	627772	660276	916720
Minimum response time:	109	93	91	92
Maximum response time:	1626078	3014685	5029095	3396970
Percentile within deadline:	99.95%	87.04%	81.58%	79.22%

Table 7.22: Scenario 7: Analysis of response times for CBR readers. All results except percentile in μs.

Scheduler	**APEX**	**CFQ pri0**	**Antic.**	**DL**
Mean response time:	32457	455313	404364	604835
Standard Deviation:	95544	621558	664103	688370
1st Quartile response time:	344	382	377	385
Median response time:	467	832	553	556989
3rd Quartile response time:	826	818079	661547	921423
Minimum response time:	117	92	92	92
Maximum response time:	2087653	3115490	5807096	3460739
Percentile within deadline:	99.88%	80.63%	81.51%	79.07%

Table 7.23: Scenario 7: Analysis of response times for VBR readers. All results except percentile in μs.

Scheduler	**APEX**	**CFQ pri0**	**Antic.**	**DL**
Mean bandwidth:	149.71	127.03	120.45	117.02
Standard Deviation:	0.00	1.18	0.75	0.15
1st Quartile bandwidth:	149	126	119	116
Median bandwidth:	149	126	120	117
3rd Quartile bandwidth:	149	127	120	117
Minimum bandwidth:	149	124	119	116
Maximum bandwidth:	149	128	121	117
Aggregate bandwidth:	7485	6351	6022	5851

Table 7.24: Scenario 7: Analysis of bandwidth for each CBR resource reserving process. All numbers in KB/s.

Scheduler	**APEX**	**CFQ pri0**	**Antic.**	**DL**
Mean bandwidth:	134.56	121.95	120.52	116.65
Standard Deviation:	0.00	1.23	0.72	0.13
1st Quartile bandwidth:	134	121	120	116
Median bandwidth:	134	122	120	116
3rd Quartile bandwidth:	134	122	121	116
Minimum bandwidth:	134	119	119	116
Maximum bandwidth:	134	124	122	116
Aggregate bandwidth:	6728	6097	6026	5832

Table 7.25: Scenario 7: Analysis of bandwidth for each resource reserving process. All numbers in KB/s.

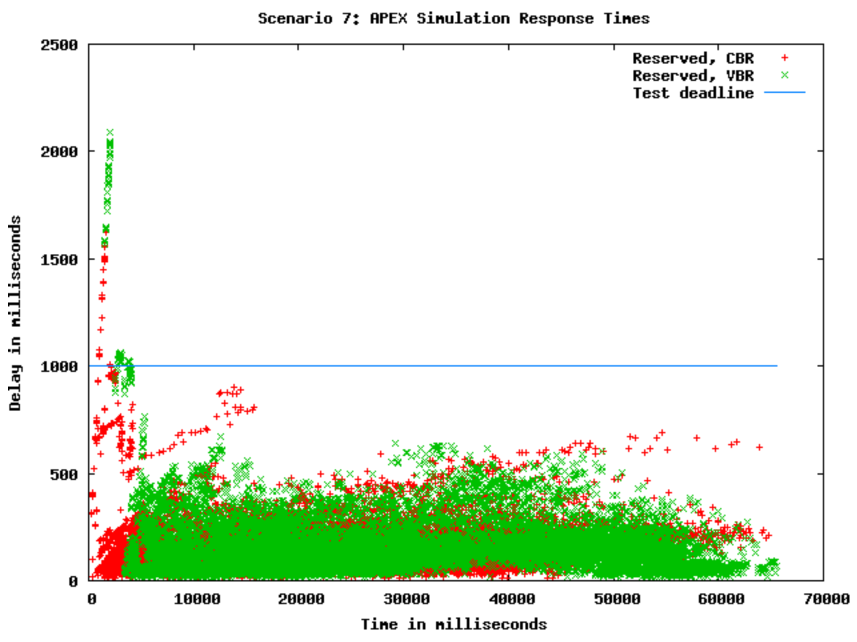

Figure 7.7: Scenario 7: Response times for CBR and VBR readers under APEX

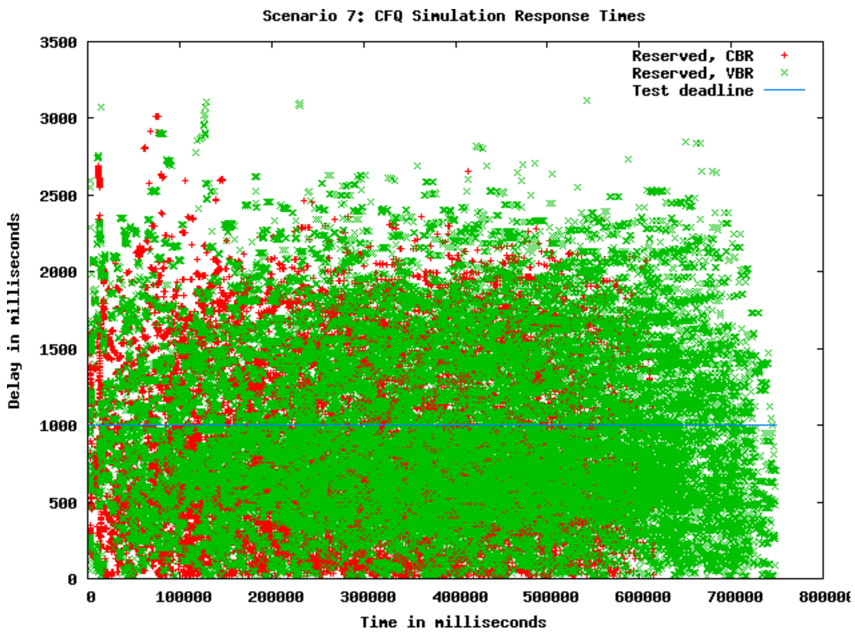

Figure 7.8: Scenario 7: Response times for CBR and VBR readers under CFQ

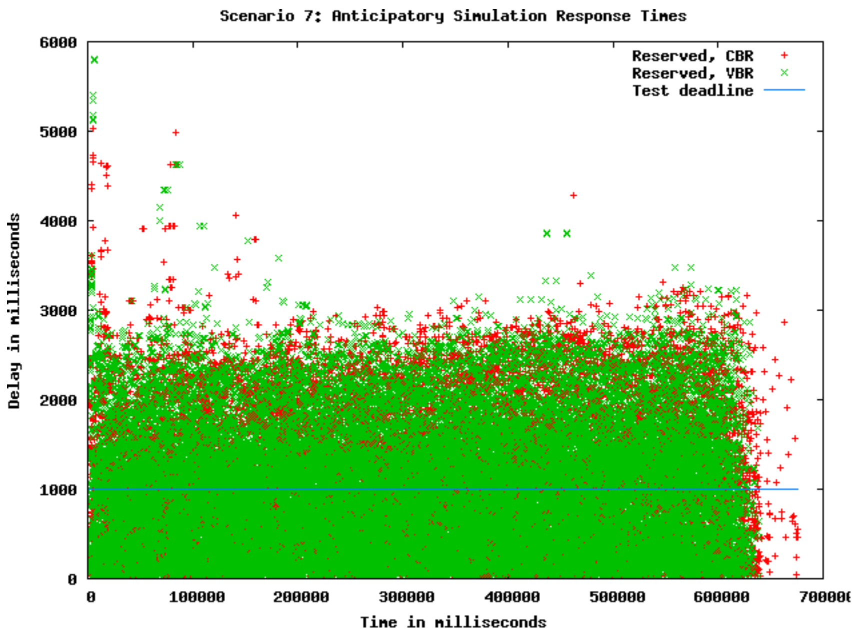

Figure 7.9: Scenario 7: Response times for CBR and VBR readers under Anticipatory

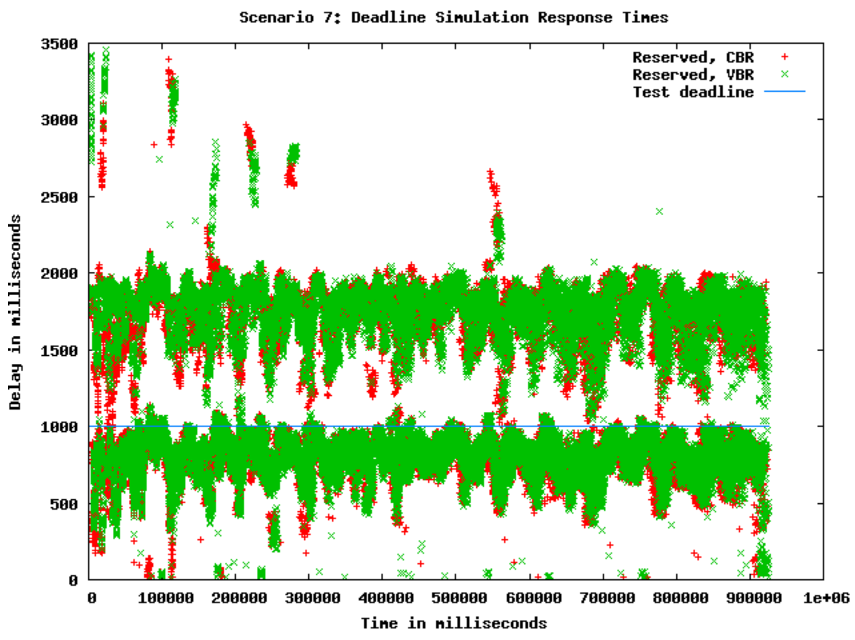

Figure 7.10: Scenario 7: Response times for CBR and VBR readers under Deadline

Overall, there is a slight bias against the VBR processes and in favor of the CBR processes. This is particularly apparent in the case of APEX and CFQ, since they perform per-process resource granting. The results for VBR are slightly worsened because the VBR processes are started after the CBR processes. This biases the results in favor of the CBR processes due to them being granted time-slices or dispatch first. This is shown particularly in figure 7.7 on page 114, where there is a visible spike at the beginning of dispatch, which is dominated by VBR operations. Once the scheduling rounds even out, this differentiation disappears.

APEX provides a mean response time of around 30ms in this scenario. While whole batches are not dispatched in 30ms, as can be seen in figure 7.7 on page 114, the read-ahead mechanisms are given more time to perform read-ahead operations. Since the files are read sequentially, these operations result in cache hits. The read-aheads benefit from the prioritized access provided to the resource reserving processes by APEX. The third quartile response time in both the CBR and VBR cases shows that over 75% of the I/O operations are satisfied by the read-ahead cache under APEX, which accounts for the low mean.

CFQ displays reasonably low mean delays well within the deadline for both CBR and VBR requests, but as shown by the standard deviation and the plotted response times in figure 7.8 on page 115, there is high variability. The anticipation heuristic and exclusive device access do not serve the scheduler well in this case. Unlike previous cases, CFQ does not starve the background thrashers in this scenario. Despite the increase in number of resource reserving readers, there are now chances for the background readers to be picked for service when none of the resource reserving readers are active.

The results of the Anticipatory scheduler are similar, with a few major differences. Since the Anticipatory scheduler does not implement prioritization, the background readers disturb the results more often. This causes longer response times in the average cases. The worst cases are caused by the initial series of failed anticipations, as is visible in figure 7.9 on page 115. This figure shows the maximum response times are all during the first series of reads. Overall, however, the starvation prevention mechanisms provide earlier dispatch for the Anticipatory scheduler than CFQ.

In this scenario, the anticipation heuristic has a negative impact on performance for CFQ and Anticipatory. The reader will delay reading after a promptly delivered request. Since CFQ and Anticipatory both assume continuous reading, the resource reserving reader will 'trick' them, by only performing short bursts of reads when it has been denied disk access for a long period of time (any period greater than or equal to the deadline). These bursts will be promptly dispatched, but the resource reserving reader will then wait before performing any further I/O. The read-ahead mechanisms will cause another request to be dispatched, but once this is completed, the CFQ and Anticipatory scheduler will both anticipate in vain. For Anticipatory this period is by default 6ms, and for CFQ this is 8ms. Not every anticipation will wait for this full period, since the heuristic adjusts for behavior such as this. However, some waiting is done, and this impacts performance.

Deadline shows two SCAN batches required to dispatch all the requests in figure 7.10 on the facing page. The first batch finishes within the deadline, but any requests not included in this batch do not maintain the deadline. Combined with the one round within the deadline, read-ahead ensures that about 80% of the requests maintain the deadline.

Since the resource reserving readers issue delays, they do not attain the desired bandwidth in under any of the schedulers. The bandwidth for the CBR readers is shown in table 7.24 on page 113, and the VBR readers in table 7.25 on page 114. This bandwidth is impacted every time a deadline is violated, since the reader does not attempt to make up for time lost due to a delay by reading more aggressively until the loss compensated for.

APEX provides the highest bandwidth throughput due to violating the fewest deadlines. APEX causes a slight differentiation in bandwidth provided to CBR and VBR requests. It does not provide worse performance for the

Scenario 8	
Bandwidth allocated (each client):	37 pages/s
Bucket depth (CBR clients):	37 pages
Bucket depth (VBR clients):	60 pages
Scenario 9	
Bandwidth allocated (CBR clients):	37 pages/s
Bandwidth allocated (VBR clients):	60 pages/s
Bucket depth (CBR clients):	37 pages
Bucket depth (VBR clients):	60 pages

Table 7.26: Scenario 8 and 9: Configuration of APEX token bucket parameters

VBR readers so much as it provides better performance for the CBR readers. The VBR readers are able to service more requests from the read-ahead cache, thus giving the more consistent CBR readers better access to the device when they are inactive. When the VBR bitrate is high, so is contention for the device. Since access is granted in a round-robin fashion, the more reliably periodic readers come out better. More data on how long the processes slept would be beneficial to explain the remaining difference.

CFQ provides slightly higher performance for CBR requests for the same reason. The process queues are granted their time-slices in a round-robin system. Since the CBR requests more frequently have pending requests when their time-slice is activated, they are serviced more promptly.

Anticipatory and Deadline provide no distinguishable difference between the bandwidth allocated to either types of readers.

APEX gives the best performance for resource reserving processes when dealing with periodic requests that fit within single read-ahead windows. Streaming audio or low quality video servers are examples of this type of data.

7.12 Scenario 8 and 9

The goal of scenarios 8 and 9 is to test whether the token bucket model in APEX provides stronger guarantees for VBR processes when the bucket depth and reserved bandwidth are increased. Both scenarios run with the same settings as given in table 7.21 on page 111 excepting the bucket depth and bandwidth allocation, which are seen in table 7.26. In scenario 8 only the bucket depth is increased, so that processes are able to build up sufficient

Scheduler	**APEX in 7**	**APEX in 8**	**APEX in 9**
Mean response time:	32457	34187	31757
Standard Deviation:	95544	98600	92815
1st Quartile response time:	344	342	344
Median response time:	467	466	467
3rd Quartile response time:	826	830	830
Minimum response time:	117	117	116
Maximum response time:	2087653	2048823	2079380
Percentile within deadline:	99.88%	99.88%	99.91%

Table 7.27: Analysis of response times for CBR readers under APEX in scenarios 7, 8 and 9. All results except percentile in μs.

Scheduler	**APEX in 7**	**APEX in 8**	**APEX in 9**
Mean bandwidth:	134.56	134.51	134.56
Standard Deviation:	0.00	0.04	0.01
1st Quartile bandwidth:	134	134	134
Median bandwidth:	134	134	134
3rd Quartile bandwidth:	134	134	134
Minimum bandwidth:	134	134	134
Maximum bandwidth:	134	134	134
Aggregate bandwidth:	6728	6725	6727

Table 7.28: Analysis of bandwidth for the VBR resource reserving processes under APEX in scenarios 7, 8 and 9. All numbers in KB/s.

tokens to perform prompt dispatch on their highest bitrate requests. In scenario 9 the minimum bandwidth is also increased, which should provide deterministic guarantees for the highest bitrate requests.

Since these tests are specific to APEX's token bucket model, they run only on the APEX scheduler.

7.12.1 Results

In table 7.27 the analysis of the VBR processes is shown. This table includes the results from the VBR processes scenario 7 for comparison. The mean delay in scenario 7 and 9 is lower than scenario 8, but the variability is so low that it is likely just experimental variability. Though not shown, the situation is similar in the CBR results. The response times for scenarios

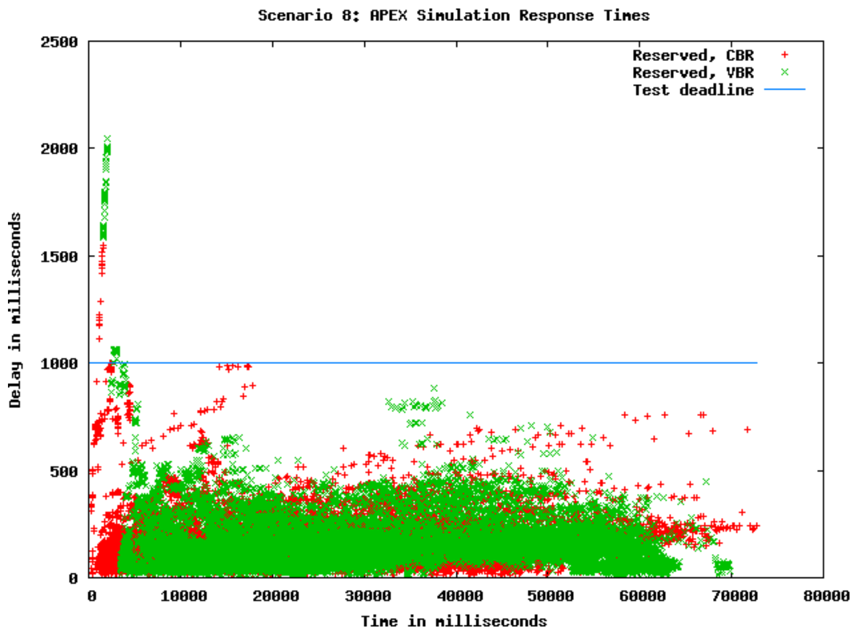

Figure 7.11: Scenario 8: Response times for VBR readers under APEX

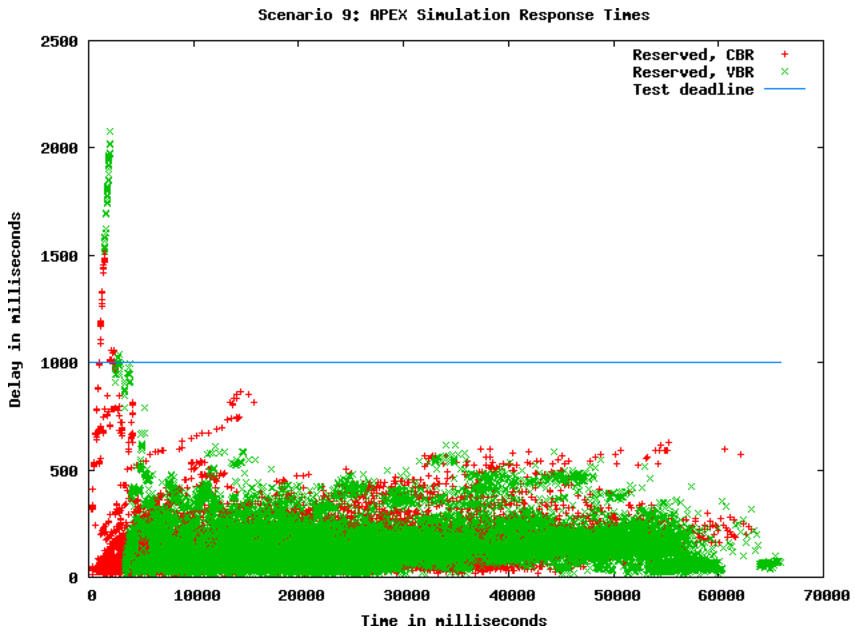

Figure 7.12: Scenario 9: Response times for CBR and VBR readers under APEX

8 and 9, shown in figure 7.11 and figure 7.12 on the facing page, display patterns very similar to that from scenario 7.

The bandwidth for the VBR results are shown in table 7.28 on page 119. The results are almost exactly the same. Again, the results for CBR are not shown, but they are just as similar.

Regardless of when a request arrives, APEX is likely to have tokens available for that queue. This is because requests are issued synchronously, so there is a delay of at least one read between the last request dispatched and the next batch building phase. Since there are several resource reserving queues active in these scenarios, it is more likely that an entire batch round is dispatched between request arrivals. Since new tokens arrive at a rate of one per 27ms[10] in scenario 7, there should always be at least one token available when a request completes. Since there is always at least one token available, the policy of APEX allowing single requests that cost more tokens than a queue has available means that a queue is almost guaranteed to be considered for service during every batch building phase.

The results from scenario 8 and 9 show that the policy of allowing queues to dispatch requests that are more costly than they have tokens is improving response times for resource reserving requests under APEX. The cost of this policy for best effort requests is shown in scenario 10. These tests should be redone with a variability that varies the number of read-ahead windows necessary to complete an operation, but the results are not expected to illuminate any new facts. Operations that require multiple batches to dispatch will have corresponding delays, regardless of the token rate, since the requests arrive to the scheduler synchronously.

7.13 Scenario 10

Scenario 10 runs 100 reserved resource readers, but half of them are run without actually reserving any resources. For the APEX and CFQ schedulers, these processes are run at the best effort priority. These processes are hereafter referred to as the foreground best effort readers. The resource reserving processes are set to read greedily in this test. The foreground best effort readers are set to random delay after they complete a read request.

The goal of scenario 10 is to assess the performance of schedulers on both reserving and best effort processes. Specifically by observing how prioritized, greedy processes impact non-greedy best effort processes. The configuration is shown in table 7.29 on the following page.

[10] $1000ms/37 = 27ms$

Reservation based readers:	50
Reader delay:	None
Foreground best effort readers:	50
Foreground best effort reader delay:	Random
Background best effort readers:	0
Size of Read:	150KB
Deadline:	1000ms
Scheduler specific configuration:	
APEX	
Bandwidth allocated (each reservation based client):	37 pages/s
Bucket depth (each reservation based client):	37 pages

Table 7.29: Scenario 10: Configuration

Scheduler	**APEX**	**CFQ pri0**	**Antic.**	**DL**
Mean response time:	421034	99887	115182	793774
Standard Deviation:	346572	957338	844761	654284
1st Quartile response time:	418	1483	1498	425
Median response time:	653082	1548	1553	1235596
3rd Quartile response time:	701819	2752	2770	1319176
Minimum response time:	69	66	65	68
Maximum response time:	1559733	75770020	11758214	2792952
Percentile within deadline:	99.33%	98.73%	98.08%	40.04%

Table 7.30: Scenario 10: Analysis of response times for resource reserving readers. All results except percentile in μs.

While neither implement prioritization, Anticipatory and Deadline are included in this scenario to test the impact of greedy readers on non-greedy readers.

7.13.1 Results

Table 7.30 shows that the results for the resource reserving processes are close to those in scenario 2. This is to be expected since the scenario is similar, with 50 resource reserving processes reading as fast as they can.

Table 7.31 on the facing page shows that the maximum response times for the best effort readers under APEX and CFQ indicate complete starvation of the processes. Apart from this, they both deliver good performances, since

Scheduler	**APEX**	**CFQ pri0**	**Antic.**	**DL**
Mean response time:	482969	57271	122934	425833
Standard Deviation:	15593287	1705988	808702	507921
1st Quartile response time:	450	453	456	454
Median response time:	480	470	476	56262
3rd Quartile response time:	79736	28868	16788	880198
Minimum response time:	100	106	72	108
Maximum response time:	576718141	140948491	38470211	2992992
Percentile within deadline:	99.93%	99.98%	98.13%	80.13%

Table 7.31: Scenario 10: Analysis of response times for foreground best effort readers. All results except percentile in μs.

Scheduler	**APEX**	**CFQ pri0**	**Antic.**	**DL**
Mean bandwidth:	356.15	1506.49	1313.58	188.85
Standard Deviation:	0.63	88.11	130.36	0.25
1st Quartile bandwidth:	355	1473	1280	188
Median bandwidth:	356	1539	1312	188
3rd Quartile bandwidth:	356	1551	1347	188
Minimum bandwidth:	355	1296	1119	188
Maximum bandwidth:	358	1750	1878	189
Aggregate bandwidth:	17807	75324	65678	9442

Table 7.32: Scenario 10: Analysis of bandwidth for each resource reserving process. All numbers in KB/s.

Scheduler	**APEX**	**CFQ pri0**	**Antic.**	**DL**
Mean bandwidth:	153.90	274.73	244.14	163.54
Standard Deviation:	0.24	18.39	2.24	0.15
1st Quartile bandwidth:	153	282	242	163
Median bandwidth:	153	283	244	163
3rd Quartile bandwidth:	154	284	245	163
Minimum bandwidth:	153	237	238	163
Maximum bandwidth:	154	285	249	163
Aggregate bandwidth:	7694	13736	12206	8176

Table 7.33: Scenario 10: Analysis of bandwidth for each foreground best effort process. All numbers in KB/s.

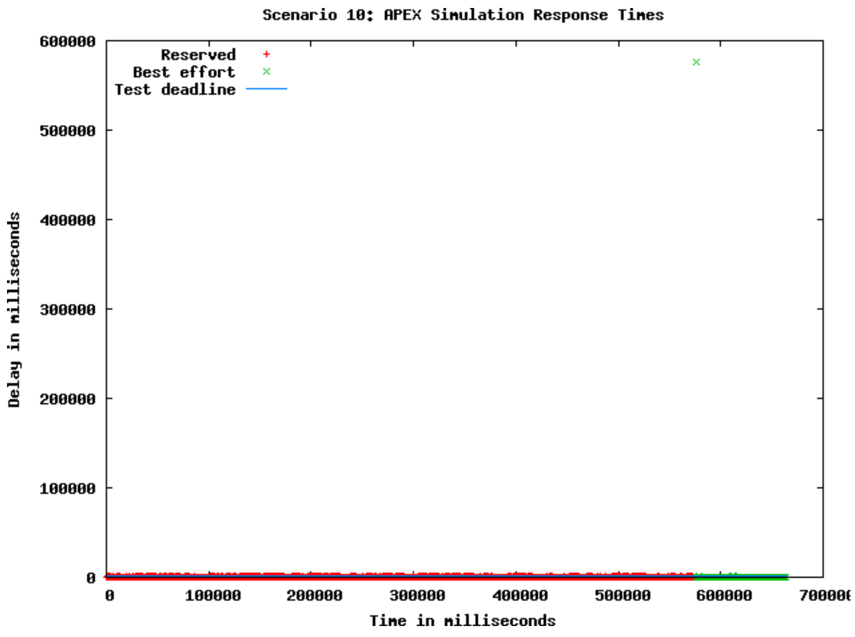

Figure 7.13: Scenario 10: Response times for reserving and best effort readers under APEX

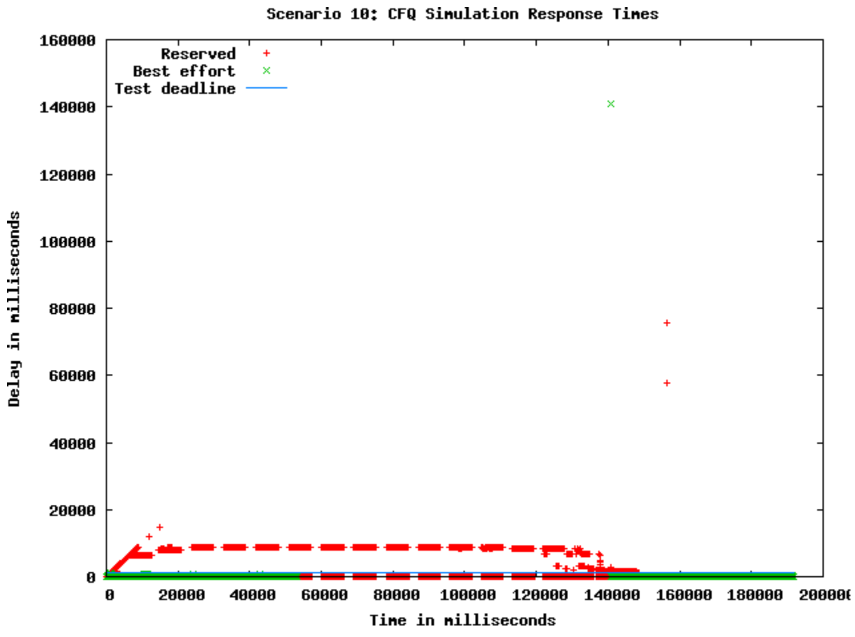

Figure 7.14: Scenario 10: Response times for reserving and best effort readers under CFQ

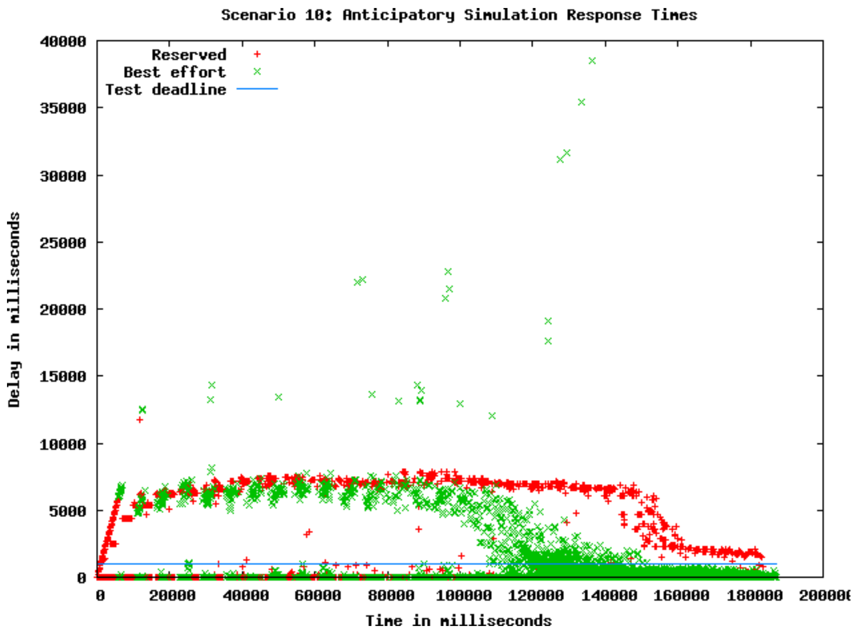

Figure 7.15: Scenario 10: Response times for reserving and best effort readers under Anticipatory

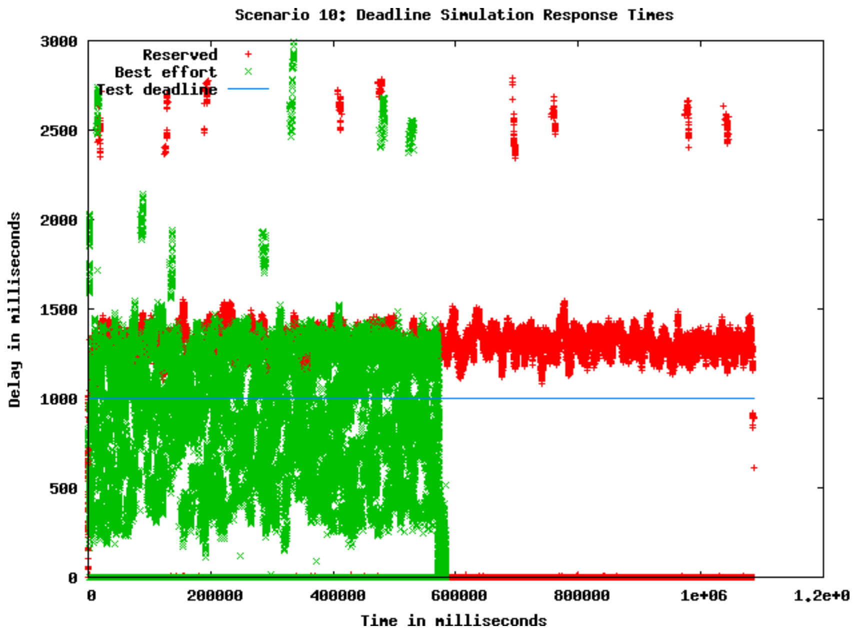

Figure 7.16: Scenario 10: Response times for reserving and best effort readers under Deadline

the best effort readers don't contend with the resource reserving processes once they begin execution. Anticipatory and Deadline show good performance for the foreground best effort processes, with Deadline performing better for these than for the resource reserving processes.

APEX completely starves the best effort readers for the execution time of the resource reserving processes, as shown in figure 7.13 on page 124. Since APEX's aggregate throughput is less than CFQ's, this is a longer time. The starvation is due to the policy of not performing a work-conserving phase when there are pending requests on reserved queues. Since this policy is adjustable, it is possible to reduce the amount of starvation for best effort processes on APEX when there is continual reserved resource traffic.

CFQ completely starves the best effort readers for the execution time of the resource reserving processes. During logging there are some accounting errors, and some of the foreground best effort processes do not begin logging correctly until after the resource reserving processes has exited. This is shown in figure 7.14 on page 124, which incorrectly shows some best effort processes running at the beginning of the graph. There are also some outlier response times for resource reserving processes associated with this logging problem.

Anticipatory again shows implicit prioritization of greedy readers in figure 7.15 on the previous page. This figure shows scattered reserved resource reads with heightened response times, along with a tight grouping of very low response times along the bottom. This corresponds with the third quartile response time of approximately 3ms and over 98% of the reads within the deadline. The consequences are visible in the best effort reads being consistently grouped along the same high cluster, but with only intermittent service at low response times. This is reflected by the median response time for the best effort readers, which at under 1ms shows that most reads are done from memory. The read-ahead causes short bursts of requests for the best effort processes, but once these are serviced, the processes must wait as long as the greedy readers to again be serviced. The greedy readers are prioritized in that they receive longer service time once they are picked for dispatch. Once the greedy readers have completed, the response times start dropping. This is inverted on the figure, since the delay for the best effort requests is not accounted for. There are also scatterings of higher response time best effort requests for Anticipatory. These requests are simply unlucky processes, which are seldom picked for service.

Deadline displays the response time pattern characteristic of saturated SCAN-related schedulers in figure 7.16 on the preceding page. The mean response time of the foreground best effort reader is close to 400ms shorter than the greedy reader. This is due to the read-ahead mechanism of the kernel combined with the random delay to the foreground best effort reader.

The delay is randomly distributed between 0 and 1s, averaging to 500ms. The read-ahead mechanism will issue the next read immediately following the completion of the previous one, thus the total delay on the read will be the same as for the greedy readers, but the foreground best effort readers will not start counting until an average of 500ms after the request is issued. In the figure, this is visible in the tight clustering of greedy reader response times, and the random distribution below this.

The bandwidth results for the resource reserving processes shown in table 7.32 on page 123 show similar throughputs to scenario 2. In table 7.33 on page 123 APEX and CFQ show low bandwidths due to completely starving these processes for the period during which the resource reserving processes ran. Anticipatory shows lower aggregate bandwidth for the randomly delayed readers than the greedy readers. This illustrates how much processes gain by promptly issuing new reads under the Anticipatory scheduler. Deadline shows slightly lower bandwidth for the randomly delayed readers than the greedy readers. While the randomly delayed readers experience lower response times due to read-ahead, they must in actuality longer than the greedy readers for their operations to complete.

The aggregate bandwidth in this test give some information as to how efficient each scheduler is relative to the maximum sustained throughput reported for the disk used in testing. The disk was reported to have a maximum throughput of around 70MB/s. APEX is limited to about 18MB/s during the run of the resource reserving processes. Deadline provides about the same, just over 17MB/s, across both the greedy and randomly delayed readers. Only CFQ and Anticipatory come close to the reported 70MB/s. CFQ's reported 75MB/s is high since the processes finish at varying times. The aggregate bandwidth for Anticipatory is similarly higher than the maximum throughput, but it also displays a bandwidth several times higher than APEX or Deadline.

APEX and CFQ completely starve best effort requests when saturated with requests from reserved resource processes. This is inappropriate for a mixed-media scheduler when the requirements of the reserved processes could be met while also dispatching at least some best effort requests. The policy chosen to not run APEX's work-conserving phase when there are pending reserved requests should be reexamined if operations can be relied on to fit within a single read-ahead window. Anticipatory rewards greedy processes for enabling efficient use of the block device, without delaying other processes much more than greedy processes. Deadline provides equivalent service to greedy and randomly delayed processes thanks to read-ahead.

Reservation based readers:	50
Reservation based reader delay:	Rate-limiting
Foreground best effort readers:	50
Foreground best effort reader delay:	Random
Background best effort readers:	0
Size of Read:	150KB
Deadline:	1000ms
Scheduler specific configuration:	
APEX	
Bandwidth allocated (each client):	37 pages/s
Bucket depth (each client):	37 pages

Table 7.34: Scenario 11: Configuration

7.14 Scenario 11

Scenario 11 is configured similarly to scenario 10, except that the resource reservation readers are configured to use the rate-limiting delay setting. With this setting the resource reserving readers will simulate periodic requests by sleeping so that each request takes approximately 1 deadline, in this case 1s, to complete. The foreground best effort readers are still configured to random delays.

This scenario examines the performance impacts of periodic resource reserving requests on best effort traffic, and the performance of the schedulers in servicing periodic requests with high amounts of best effort traffic. The full configuration can be seen in table 7.34.

For Anticipatory and Deadline the resource reserving readers are be referred to as the rate-limited readers in this scenario, since they do not reserve any resources.

7.14.1 Results

The results in table 7.35 on the facing page show that APEX performs best for the resource reserving readers in this scenario, maintaining the deadline for almost all the requests. CFQ and Anticipatory perform well, maintaining around 96% of the deadlines. The Deadline scheduler holds the highest mean and highest maximum delay. Table 7.36 on the next page shows that CFQ provides better mean response time than APEX for best effort requests. Deadline shows overall good response times for the randomly delayed best effort operations.

Scheduler	**APEX**	**CFQ pri0**	**Antic.**	**DL**
Mean response time:	26776	163497	125590	225965
Standard Deviation:	68230	320698	294293	369596
1st Quartile response time:	408	451	456	455
Median response time:	428	478	481	36943
3rd Quartile response time:	663	145155	55695	265778
Minimum response time:	104	103	111	102
Maximum response time:	2189359	2175784	2348360	2389587
Percentile within deadline:	99.99%	96.11%	96.21%	89.99%

Table 7.35: Scenario 11: Analysis of response times for resource reserving requests. All results except percentile in μs.

Scheduler	**APEX**	**CFQ pri0**	**Antic.**	**DL**
Mean response time:	614860	320553	271629	451598
Standard Deviation:	724781	372850	403638	478634
1st Quartile response time:	422	471	462	469
Median response time:	224873	182945	866	364801
3rd Quartile response time:	1196064	574907	475856	858947
Minimum response time:	73	102	109	111
Maximum response time:	4933037	2487295	2727230	35299666
Percentile within deadline:	67.78%	93.80%	91.40%	83.39%

Table 7.36: Scenario 11: Analysis of response times for foreground best effort requests . All results except percentile in μs.

Scheduler	**APEX**	**CFQ pri0**	**Antic.**	**DL**
Mean bandwidth:	149.79	148.61	148.27	147.03
Standard Deviation:	0.05	0.22	0.28	0.22
1st Quartile bandwidth:	149	148	148	146
Median bandwidth:	149	148	148	147
3rd Quartile bandwidth:	149	148	148	147
Minimum bandwidth:	149	148	147	146
Maximum bandwidth:	149	149	149	147
Aggregate bandwidth:	7489	7430	7413	7351

Table 7.37: Scenario 11: Analysis of bandwidth for each resource reserving process. All numbers in KB/s.

Scheduler	**APEX**	**CFQ pri0**	**Antic.**	**DL**
Mean bandwidth:	135.57	184.69	196.54	159.04
Standard Deviation:	0.20	1.12	2.42	0.59
1st Quartile bandwidth:	135	184	195	158
Median bandwidth:	135	184	196	158
3rd Quartile bandwidth:	135	185	197	159
Minimum bandwidth:	135	182	193	157
Maximum bandwidth:	136	186	203	160
Aggregate bandwidth:	6778	9234	9826	7951

Table 7.38: Scenario 11: Analysis of bandwidth for each foreground best effort processes. All numbers in KB/s.

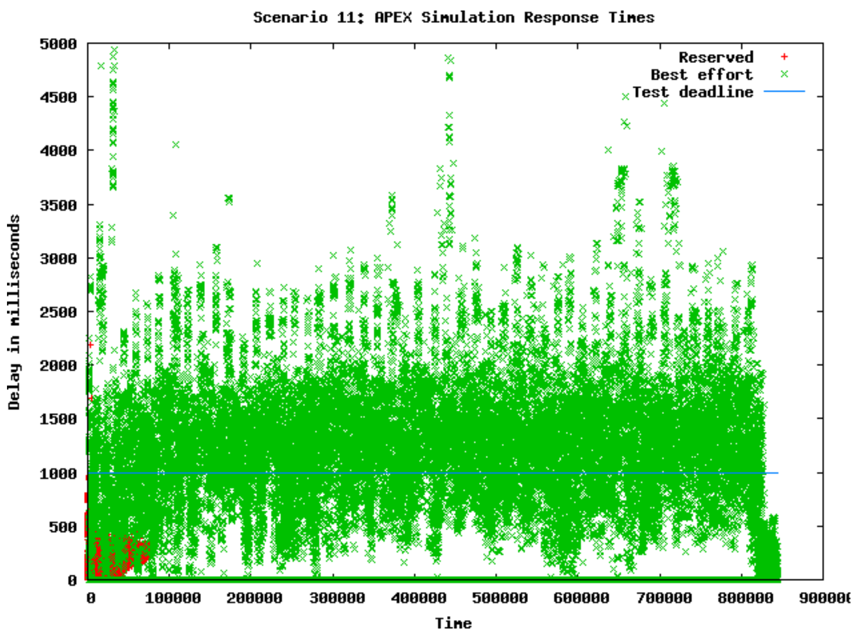

Figure 7.17: Scenario 11: Response times for reserving and best effort readers under APEX

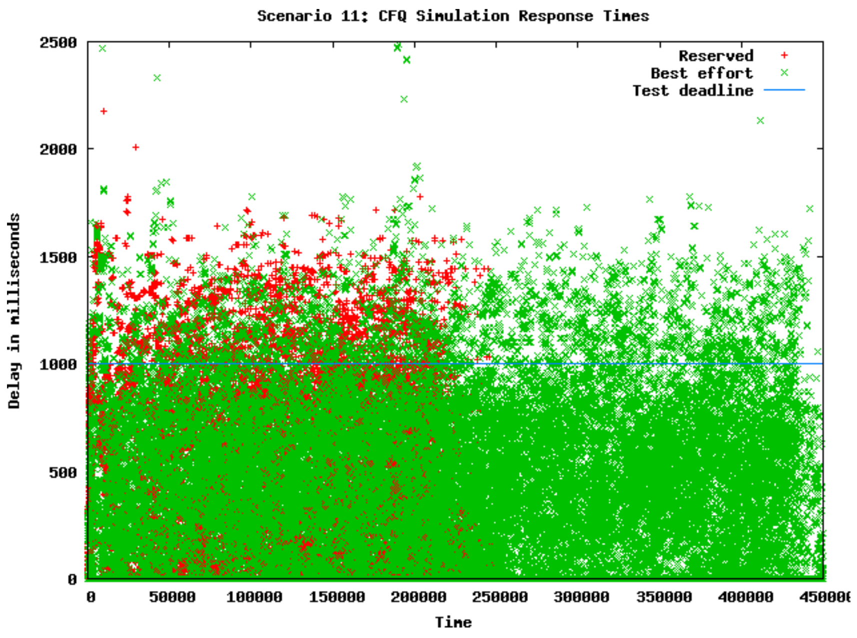

Figure 7.18: Scenario 11: Response times for reserving and best effort readers under CFQ

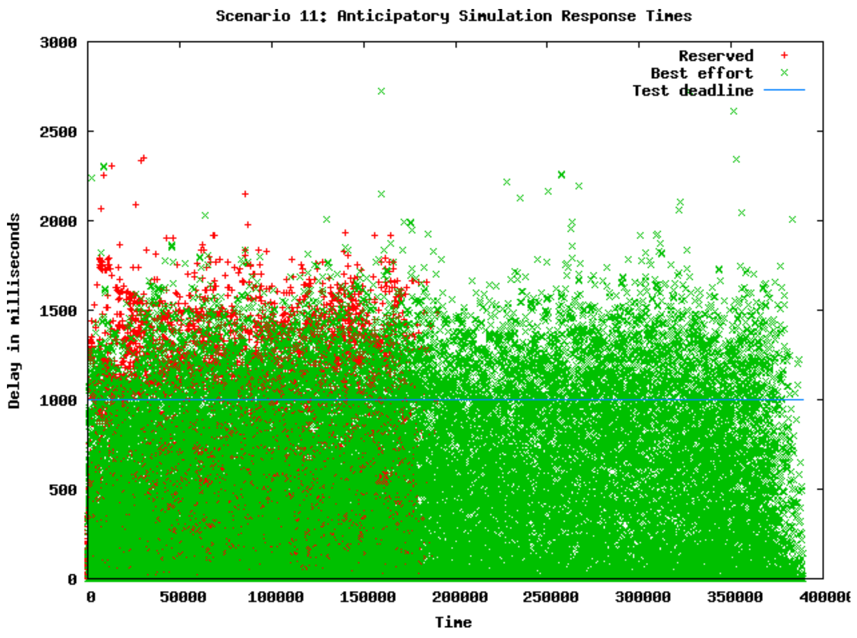

Figure 7.19: Scenario 11: Response times for reserving and best effort readers under Anticipatory

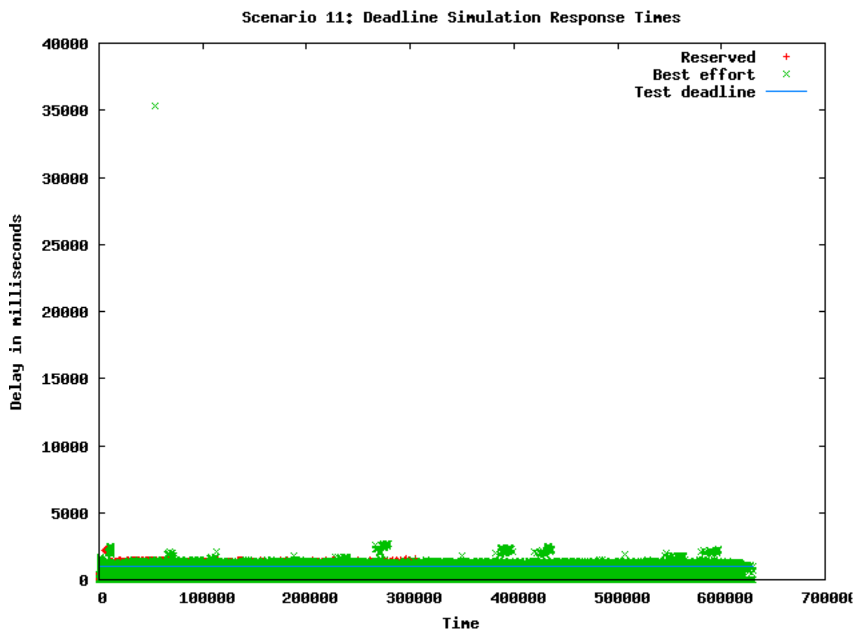

Figure 7.20: Scenario 11: Response times for reserving and best effort readers under Deadline

Figure 7.17 on page 130 shows that APEX maintains approximately the same service for the best effort requests throughout the test. While the resource reserving processes run, they hold priority to the block device, thus the best effort requests are only dispatched when there are windows available for work-conserving phases.

The third quartile of response times for the reservation based reader under APEX shows that at least 75% of the reservation based operations are served by the read-ahead cache. This means that delay to the best effort operations is caused by read-ahead operations.

CFQ provides better service to best effort readers than APEX, but at the cost of more deadlines broken for the reserved processes. This is shown in figure 7.18 on the previous page. Here the distributions of the response times for both the resource reserving and foreground best effort processes show similar distributions. The best effort processes spend more time reading, which reflects the higher median response time. The best effort processes are more often preempted by requests arriving for higher priority queues and they delay longer between reads. The former is a consequence of CFQ's prioritization mechanism, and the latter causes the anticipation heuristic to cut off access to the device sooner.

Real-time queues delayed by active best effort dispatches cause the following behavior. When no real-time queues are dispatching requests, for example when they are sleeping because their requests are promptly dispatched, a real-time queue's idle slice will expire, handing over control of the device to a best effort queue. This best effort queue will then dispatch its one request. If a real-time request arrives just after this best effort request has been dispatched, it will have to wait for the request to be completed before CFQ grants it control of the device. This negates CFQ's anticipation heuristic for best effort queues, and causes seekiness by allowing only one request to be serviced at a time for best effort queues. Since the real-time queues are issuing only a few requests before the resource reserving processes sleep, CFQ behaves more like a prioritized round robin scheduler with more seeky behavior than shown in previous tests.

The Anticipatory scheduler performs similarly to CFQ in this scenario, as shown by the distribution in figure 7.19 on page 131. The distribution is similar to the corresponding graph for CFQ. The lack of a time-slicing system causes Anticipatory to less often pick the randomly delayed processes for dispatch. The anticipation heuristic biases against these in favor of the rate-limited delay. The rate-limited delaying processes are at some point serviced by a late response, which causes them to immediately dispatch a new request. This behavior is rewarded by the anticipation heuristic with continued access to the device. This goes on until the process is sleeping and the last read-ahead completes. This way, every late response time by Anticipatory triggers behavior that is rewarded by extra access. This extra access denies the randomly delaying processes device time.

This test shows how Deadline can temporarily starve requests when high amounts of traffic to one end of the device arrive continually. The one way elevator can get stuck on one side of the platter if there are continual streams of requests on that side of the platter. To prevent this, Deadline implements an expire time for requests. If a request expires, Deadline will allow seeking backwards to satisfy it. However, if the FIFO queue is filled with requests on the same side of the platter, the behavior will continue to starve requests to the other side of the platter until the FIFO is empty. This explains the behavior in figure 7.20 on the preceding page, where some randomly delayed requests have response times clustered above the main SCAN belt. The single delay at 35s is an anomaly.

Table 7.37 on page 129 shows the analysis of the bandwidth of the resource reserving processes. APEX manages to maintain the highest mean bandwidth. Since this bandwidth follows the rate of deadline failure, Deadline delivers the lowest bandwidth. Table 7.38 on page 130 shows that CFQ and Anticipatory gave the highest aggregate bandwidth to the randomly delaying best effort readers. This is explained by the anticipation heuristic increasing the throughput by avoiding seeks when read-ahead requests arrive.

APEX performs best for the reservation based readers, but there is a cost in response times for the randomly delaying readers. CFQ and Anticipatory provide worse throughput when dealing with programs that do not continually issue new I/O operations. APEX is better suited for providing data for unbuffered streams.

7.15 Summary

APEX performs well, so long as the I/O operations performed fit into a number of dispatch rounds that can be executed within the deadline. Otherwise the synchronous arrival of requests prevent it from correctly dispatching requests in one round. It has trouble with the token bucket model when the processes issue requests continually. While the conceptual translation from the original presentation to the implementation of APEX is problematic due to the lack of metadata, APEX functions well as a prioritized varied-length round-based scheduler with mixed-media capabilities. The policy for the work-conserving phase does not provide full work-conservation, but in practice this can be adjusted by a tunable variable.

Of the existing Linux schedulers, Anticipatory and CFQ are oriented towards providing high throughput at the cost of occasional high response times. This is not appropriate in cases of continuous media playback, where the minimum buffer required and therefore also start-up delay equals the longest response time. An admission control device allowing some sort of limitation to the maximum response time might be appropriate for CFQ.

Since APEX relies on a scanning algorithm for efficiency, it does not provide efficiency as well as Anticipatory and CFQ do in the best case. Since it is not unreasonable to expect MMDBMS systems to order data according to metadata requirements, such as ordering sequential data consecutively on the disk, it is not unreasonable to assume that an anticipation heuristic would provide better efficiency.

Chapter 8

Conclusions

This thesis focuses on implementing and testing the APEX mixed-media scheduler in the Linux kernel as a part of the elevator framework. Requirements for continuous media environments are presented and existing mixed-media scheduler designs are examined for suitability. APEX is found to best match the requirements. The Linux internals relevant to the implementation are examined and difficulties related to implementing a high level scheduler in the Linux kernel are examined. A partial implementation of APEX focused on the batch builder is implemented and tested.

The tests focused on requirements of mixed-media environments, and show that the existing Linux I/O schedulers are not suited for these types of environments. While the implementation of APEX is better, the lack of high level information at the scheduler makes its task difficult. When read operations fit into one read-ahead window, APEX behaves much like the original implementation details. Otherwise the synchronous nature of read operations provides a series of problems. The implementation of APEX does not provide full efficiency or work-conservation, and the flexibility is limited by the lack of the queue manager.

8.1 Future Work

Implementing the high level scheduler above or as part of the virtual filesystem in Linux is a possible solution to many of the problems encountered during the development of this work. This will make queue management easier, in that it can exist implicit to the file handle model. Admission control is also made possible by returning values directly from the `open` system call.

Adding a token bucket model to CFQ to improve the starvation under real-time loads and provide bandwidth guarantees is another possible improvement to current work. A new service class could also be added, which is more suitable for periodic requests associated with deadlines, since the current real-time class is in actuality a low latency service.

A high level scheduler could also be employed in concert with CFQ, since the maximum delay is fairly predictable. In this case, admission control could specify the maximum allowable response time, rather than a bandwidth guarantee.

Appendix A

Source Code

The source code of the readers used in testing, the modified version of Linux and APEX module are available on the appended CD. The file `README` describes the directory structure.

Bibliography

[1] Robert K. Abbott and Hector Garcia-Molina. Scheduling real-time transactions: a performance evaluation. In *VLDB '88: Proceedings of the 14th International Conference on Very Large Data Bases*, pages 1–12, San Francisco, CA, USA, 1988. Morgan Kaufmann Publishers Inc.

[2] Robert K. Abbott and Hector Garcia-Molina. Scheduling i/o requests with deadlines: A performance evaluation. In *IEEE Real-Time Systems Symposium*, pages 113–125, 1990.

[3] Jens Axboe. [PATCH] block/elevator updates + deadline i/o scheduler. Email list posting, 07 2002. `http://lwn.net/Articles/5862/`.

[4] Jens Axboe. Linux block io - present and future. In *Proceedings of the Ottawa Linux Symposium 2004*, pages 51–61, July 2004.

[5] Jens Axboe. [PATCH][CFT] time sliced cfq ver18. Email list posting, 12 2004. `http://lwn.net/Articles/116496/`.

[6] Jens Axboe, Suparna Bhattacharya, and Nick Piggin. *Notes on the Generic Block Layer Rewrite in Linux 2.5*, 1 2002. `Documentation/block/biodoc.txt`.

[7] Peter Bosch, Cwi, peterb@cwi. nl, and Sape J. Mullender. Real-time disk scheduling in a mixed-media file system. In *RTAS '00: Proceedings of the Sixth IEEE Real Time Technology and Applications Symposium (RTAS 2000)*, page 23, Washington, DC, USA, 2000. IEEE Computer Society.

[8] Neil Brown et al. *The Linux Virtual File-system Layer*, 1.6 edition, 12 1999. `http://www.cse.unsw.edu.au/~neilb/oss/linux-commentary/vfs.html`.

[9] John Bruno, Jose Brustoloni, Eran Gabber, Banu Ozden, and Abraham Silberschatz. Disk scheduling with quality of service guarantees. In

ICMCS '99: Proceedings of the IEEE International Conference on Multimedia Computing and Systems Volume II-Volume 2, page 400, Washington, DC, USA, 1999. IEEE Computer Society.

[10] Milind M. Buddhikot, Xin Jane Chen, Dakang Wu, and Guru M. Parulkar. Enhancements to 4.4 bsd unix for efficient networked multimedia in project mars. In *ICMCS '98: Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, page 326, Washington, DC, USA, 1998. IEEE Computer Society.

[11] T. Coughlin and C. Associates. Putting Portable Storage in Perspective. *Time (Point above which HDDs become less expensive than Flash Memory)*, 1:7.

[12] Eduardo Madeira Fleury, Michael Kerrisk, and Jens Axboe. *ioprio_get, ioprio_set - get/set I/O scheduling class and priority*, 6 2006. Linux version 2.6.13 and later.

[13] Robert Geist and Stephen Daniel. A continuum of disk scheduling algorithms. *ACM Trans. Comput. Syst.*, 5(1):77–92, 1987.

[14] D. James Gemmell and Jiawei Han. Multimedia network file servers: multi-channel delay sensitive data retrieval. In *MULTIMEDIA '93: Proceedings of the first ACM international conference on Multimedia*, pages 243–250, New York, NY, USA, 1993. ACM Press.

[15] D. James Gemmell, Harrick M. Vin, Dilip D. Kandlur, P. Venkat Rangan, and Lawrence A. Rowe. Multimedia storage servers: A tutorial. *Computer*, 28(5):40–49, 1995.

[16] K. Gopalan and T. Chiueh. Real-time disk scheduling using deadline sensitive SCAN. Technical Report 92, Experimental Computer Systems Labs, Dept. of Computer Science, SUNY at Stony Brook, Stony Brook, NY, USA, Jan 2001.

[17] Pål Halvorsen, Carsten Griwodz, Vera Goebel, Ketil Lund, Thomas Plagemann, and Jonathan Walpole. Storage system support for continuous-media applications, part 1: Requirements and single-disk issues. *IEEE Distributed Systems Online*, 05(1), 2004.

[18] Micha Hofri. Disk Scheduling: FCFS vs. SSTF Revisited. *Commun. ACM*, 23(11):645–653, 1980.

[19] Lan Huang and Tzi cker Chiueh. Implementation of a rotation latency sensitive disk scheduler, 2000.

[20] A. Huffman and J. Clark. Serial ATA Native Command Queuing. 2003.

[21] Sitaram Iyer and Peter Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *18th ACM Symposium on Operating Systems Principles*, October 2001.

[22] I. Kamel, T. Niranjan, and S. Ghandeharizedah. A Novel Deadline Driven Disk Scheduling Algorithm for Multi-Priority Multimedia Objects. *Data Engineering, 2000. Proceedings. 16th International Conference on*, pages 349–361, 2000.

[23] Ravi Kiran. Linux Block Device Architecture. WWW page, 5 2006. `http://www.geocities.com/ravikiran_uvs/articles/blkdevarch.html`.

[24] Robert Love. *Linux Kernel Development, Second Edition.* Novell Press, 2005.

[25] K. Lund. *Adaptive Disk Scheduling in a Multimedia DBMS.* PhD thesis, Department of Informatics, University of Oslo, 2003.

[26] C. Martin, P. Narayanan, B. Ozden, R. Rastogi, and A. Silberschatz. The fellini multimedia storage server, 1996.

[27] G. Nerjes, P. Muth, M. Paterakis, Y. Romboyannakis, P. Triantafillou, and G. Weikum. Scheduling strategies for mixed workloads in multimedia informationservers. *Research Issues In Data Engineering, 1998. Continuous-Media Databases and Applications. Proceedings. Eighth International Workshop on*, pages 121–128, 1998.

[28] H. Ohnishi, T. Okada, K. Noguchi, and T. NTT. Flow Control Schemes and Delay/Loss Tradeoff in ATM Networks. *Selected Areas in Communications, IEEE Journal on*, 6(9):1609–1616, 1988.

[29] G. Özsoyoglu and RT Snodgrass. Temporal and Real-time Databases: A Survey. *Knowledge and Data Engineering, IEEE Transactions on*, 7(4):513–532, 1995.

[30] Nick Piggin. *Anticipatory IO scheduler*, 9 2003. `Documentation/block/as-iosched.txt`.

[31] KK Ramakrishnan, L. Vaitzblit, C. Gray, U. Vahalia, D. Ting, P. Tzelnic, S. Glaser, and W. Duso. Operating System Support for a Video-On-Demand File Service. *Multimedia Systems*, 3(2):53–65, 1995.

[32] A. L. Narasimha Reddy and James C. Wyllie. I/o issues in a multimedia system. *Computer*, 27(3):69–74, 1994.

[33] A. L. Narasimha Reddy and Jim Wyllie. Disk scheduling in a multimedia I/O system. In *MULTIMEDIA '93: Proceedings of the first ACM international conference on Multimedia*, pages 225–233, New York, NY, USA, 1993. ACM Press.

[34] Y. Rompogiannakis, G. Nerjes, P. Muth, M. Paterakis, P. Triantafillou, and G. Weikum. Disk Scheduling for Mixed-media Workloads in a Multimedia Server. In *MULTIMEDIA '98: Proceedings of the sixth ACM international conference on Multimedia*, pages 297–302, New York, NY, USA, 1998. ACM Press.

[35] Chris Ruemmler and John Wilkes. An Introduction to Disk Drive Modeling. *IEEE Computer*, 27(3):17–28, 1994.

[36] Seagate Global Product Marketing. 2.5-Inch Enterprise Disc Drives: Key to Cutting Data Center Costs. Technical Report TP-534, Seagate Technology LLC, Feb 2005.

[37] Seagate Technology LLC. *Seagate Cheetah 15K.4 SCSI Product Manual*, 5 2005. Fetched December 2006.

[38] Seagate Technology LLC. *Seagate Barracuda 7200.10 PATA Product Manual*, 05 2006. Fetched December 2006.

[39] M. Seltzer, P. Chen, and J. Ousterhout. Disk Scheduling Revisited. In *Proceedings of the USENIX Winter 1990 Technical Conference*, pages 313–324, Berkeley, CA, 1990. USENIX Association.

[40] Prashant Shenoy and Harrick M. Vin. Cello: A Disk Scheduling Framework for Next Generation Operating Systems*. *Real-Time Syst.*, 22(1-2):9–48, 2002. Based on preliminary version presented in the proceedings ACM SIGMETRICS'98 conference.

[41] Prashant J. Shenoy, Pawan Goyal, Sriram Rao, and Harrick M. Vin. Symphony: An Integrated Multimedia File System. Technical Report CS-TR-97-09, 1, 1998.

[42] Prashant J. Shenoy and Harrick M. Vin. Cello: A Disk Scheduling Framework for Next Generation Operating Systems. Technical report, Austin, TX, USA, 1998.

[43] M. Shreedhar and George Varghese. Efficient Fair Queueing Using Deficit Round-robin. *IEEE/ACM Trans. Netw.*, 4(3):375–385, 1996.

[44] Jeffrey B. Siegal. Re: Strange scheduling behavoir in SMP (kernel 2.2.14). Email list posting, 1 2000. `http://www.cs.rice.edu/~ssiyer/r/antsched/linux.html`.

[45] Andrew S. Tanenbaum. *Modern Operating Systems.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[46] Tsun-Ping J. To and Babak Hamidzadeh. Dynamic real-time scheduling strategies for interactive continuous media servers. *Multimedia Syst.*, 7(2):91–106, 1999.

[47] Linus Torvalds and the Linux Kernel Developers. The Linux Kernel (2.6.19). Source code, 1991-2006. `http://www.kernel.org`.

[48] Ravi Wijayaratne. *Prism: A File Server Architecture for Providing Integrated Services.* PhD thesis, 2001. Chair-Ricardo Bettati and Chair-A. L. Reddy.

[49] Ravi Wijayaratne and A. L. Narasimha Reddy. Providing QoS guarantees for disk I/O. *Multimedia Syst.*, 8(1):57–68, 2000.

[50] Bruce L. Worthington, Gregory R. Ganger, and Yale N. Patt. Scheduling Algorithms for Modern Disk Drives. *SIGMETRICS Perform. Eval. Rev.*, 22(1):241–251, 1994.

[51] Philip S. Yu, Mon-Song Chen, and Dilip D. Kandlur. Design and Analysis of a Grouped Sweeping Scheme for Multimedia Storage Management. In *Proceedings of the Third International Workshop on Network and Operating System Support for Digital Audio and Video*, pages 44–55, London, UK, 1993. Springer-Verlag.