CHAPTER

Performance

6

CHAPTER CONTENTS

Principles of Computer System Design: An Introduction

DOI: 10.1016/B978-0-12-374957-4.00015-3

OVERVIEW

The specification of a computer system typically includes explicit (or implicit) performance goals. For example, the specification may indicate how many concurrent users the system should be able to support. Typically, the simplest design fails to meet these goals because the design has a *bottleneck*, a stage in the computer system that takes longer to perform its task than any of the other stages. To overcome bottlenecks, the system designer faces the task of creating a design that performs well, yet is simple and modular.

This chapter describes techniques to avoid or hide performance bottlenecks. Section 6.1 presents ways to identify bottlenecks and the general approaches to handle them, including exploiting workload properties, concurrent execution of operations, speculation, and batching. Section 6.2 examines specific versions of the general techniques to attack the common problem of implementing multilevel memory systems efficiently. Section 6.3 presents scheduling algorithms for services to choose which request to process first, if there are several waiting for service.

6.1 DESIGNING FOR PERFORMANCE

Performance bottlenecks show up in computer systems for two reasons. First, limits imposed by physics, technology, or economics restrict the rate of improvement in some dimensions of technology, while other dimensions improve rapidly. An obvious class of limits are the physical ones. The speed of light limits how fast signals travel from one end of a chip to the other, how many memory elements can be within a given latency from the processor, and how fast a network message can travel in the Internet. Many other physical limits appear in computer systems, such as power and heat dissipation.

These limits force a designer to make trade-offs. For example, by shrinking a chip, a designer can make the chip faster, but it also reduces the area from which heat can be dissipated. Worse, the power dissipation increases as the designer speeds up the chip. A related trade-off is between the speed of a laptop and its power consumption. A designer wants to minimize a laptop's power consumption so that the battery lasts longer, yet customers want laptops with fast processors and large, bright screens.

Physical limits are only a subset of the limits a designer faces; there are also algorithmic, reliability, and economic limits. More limits mean more trade-offs and a higher risk of bottlenecks.

The second reason bottlenecks surface in computer systems is that several clients may share a device. If a device is busy serving one client, other clients must wait until the device becomes available. This property forces the system designer to answer questions such as which client should receive the device first. Should the device first perform the request that requires little work, perhaps at the cost of delaying the request that requires a lot of work? The designer would like to devise a scheduling plan that doesn't starve some clients in favor of others, provides low turnaround time

for each individual client request, and has little overhead so that it can serve many clients. As we will see, it is impossible to maximize all of these goals simultaneously, and thus a designer must make trade-offs. Trade-offs may favor one class of requests over another and may result in bottlenecks for the unfavored classes of requests.

Designing for performance creates two major challenges in computer systems. First, one must consider the benefits of optimization in the context of technology improvements. Some bottlenecks are intrinsic ones; they require careful thinking to ensure that the system runs faster than the performance of the slowest stage. Some bottlenecks are technology dependent; time may eliminate these, as technology improves. Unfortunately, it is sometimes difficult to decide whether or not a bottleneck is intrinsic. Not uncommonly, a performance optimization for the next product release is irrelevant by the time the product ships because technology improvements have removed the bottleneck completely. This phenomenon is so common in computer design that it has led to formulation of the design hint: *when in doubt use brute force*. Sidebar 6.1 discusses this hint.

Sidebar 6.1 Design Hint: When in Doubt use Brute Force This chapter describes a few design hints that help a designer resolve trade-offs in the face of limits. These design hints are hints because they often guide the designer in the right direction, but sometimes they don't. In this book we cover only a few, but the interested reader should digest *Hints for computer system design* by B. Lampson, which presents many more practical guidelines in the form of hints [Suggestions for Further Reading 1.5.4].

The design hint "when in doubt use brute force" is a direct corollary of the *d(technology)/dt* curve (see Section 1.4). Given computing technology's historical rate of improvement, it is typically wiser to choose simple algorithms that are well understood rather than complex, badly characterized algorithms. By the time the complex algorithm is fully understood, implemented, and debugged, new hardware might be able to execute the simple algorithm fast enough. Thompson and Ritchie used a fixed-size table of processes in the UNIX system and searched the table linearly because a table was simple to implement and the number of processes was small. With Joe Condon, Thompson also built the Belle chess machine that relied mostly on special-purpose hardware to search many positions per second rather than on sophisticated algorithms. Belle won the world computer chess championships several times in the late 1970s and early 1980s and achieved an ELO rating of 2250. (ELO is a numerical rating systems used by the World Chess Federation (FIDI) to rank chess players; a rating of 2250 makes one a strong competitive player.) Later, as technology marched on, programs that performed brute-force searching algorithms on an off-the-shelf PC conquered the world computer chess championships. As of August 2005, the Hydra supercomputer (64 PCs, each with a chess coprocessor) is estimated by its creators to have an ELO rating of 3200, which is better than the best human player.

A second challenge in designing for performance is maintaining the simplicity of the design. For example, if the design uses different devices with approximately the same high-level function but radically different performance, a challenge is to abstract devices such that they can be used through a simple uniform interface. In this chapter, we see how a clever implementation of the READ and WRITE interface for memory can transparently extend the effective size of RAM to the size of a magnetic disk.

6.1.1 Performance Metrics

To understand bottlenecks more fully, recall that computer systems are organized in modules to achieve the benefits of modularity and that to process a request, the request may be handed from one module to another. For example, a camera may generate a continuous stream of requests containing video frames and send them to a service that digitizes each frame. The digitizing service in turn may send its output to a file service that stores the frames on a magnetic disk.

By describing this application in a client/service style, we can obtain some insights about important performance metrics. It is immediately clear that in a computer system such as this one, four metrics are of importance: the capacity of the service, its utilization, the time clients must wait for request to complete, and throughput, the rate at which services can handle requests. We will discuss each metric in turn.

6.1.1.1 Capacity, Utilization, Overhead, and Useful Work

Every service has some *capacity*, a consistent measure of a service's size or amount of resources. *Utilization* is the percentage of capacity of a resource that is used for some given workload of requests. A simple measure of processor capacity is cycles. For example, the processor might be utilized 10% for the duration of some workload, which means that 90% of its processor cycles are unused. For a magnetic disk, the capacity is usually measured in sectors. If a disk is utilized 80%, then 80% of its sectors are used to store data.

In a layered system, each layer may have a different view of the capacity and utilization of the underlying resources. For example, a processor may be 95% utilized but delivering only 70% of its cycles to the application because the operating system uses 25%. Each layer considers what the layers below it do to be *overhead* in time and space, and what the layers above it do to be *useful work*. In the processor example, from the application point of view, the 25% of cycles used by the operating system is overhead and the 70% is useful work. In the disk example, if 10% of the disk is used for storing file system data structures, then from the application point of view that 10% used by the file system is overhead and only 90% is useful capacity.

6.1.1.2 Latency

Latency is the delay between a change at the input to a system and the corresponding change at its output. From the client/service perspective, the latency of a request is the time from issuing the request until the time the response is received from the service.

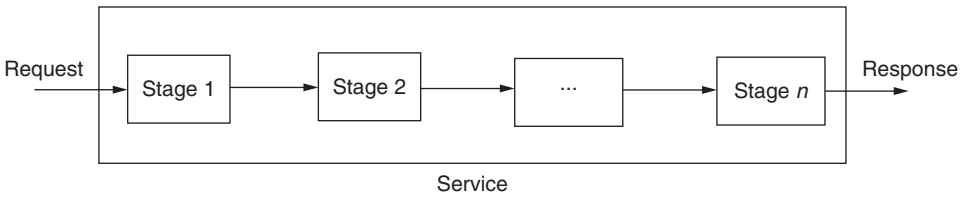


FIGURE 6.1

A simple service composed of several stages.

This latency has several components: the latency of sending a message to the service, the latency of processing the request, and the latency of sending a response back.

If a task, such as asking a service to perform a request, is a sequence of subtasks, we can think of the complete task as traversing stages of a pipeline, where each stage of the pipeline performs a subtask (see Figure 6.1). In our example, the first stage in the pipeline is sending the request, the second stage is the service digitizing the frame, the third stage is the file service storing the frame, and the final stage is sending a response back to the client.

With this pipeline model in mind, it is easy to see that latency of a pipeline with stages A and B is greater than or equal to the sum of the latencies for each stage in the pipeline:

$$latency_{A+B} \geq latency_A + latency_B$$

It is possibly greater because passing a request from one stage to another might add some latency. For example, if the stages correspond to different services, perhaps running on different computers connected by a network, then the overhead of passing requests from one stage to another may add enough latency that it cannot be ignored.

If the stages are of a single service, that additional latency is typically small (e.g., the overhead of invoking a procedure) and can usually be ignored for first-order analysis of performance. Thus, in this case, to predict the latency of a service that isn't running yet but is expected to perform two functions, A and B, with known latencies, a designer can approximate the joint latency of A and B by adding the latency of A and the latency of B.

6.1.1.3 Throughput

Throughput is a measure of the rate of useful work done by a service for some given workload of requests. In the camera example, the throughput we might care about is how many frames per second the system can process because it may determine what quality camera we want to buy.

The throughput of a system with pipelined stages is less than or equal to the minimum of the throughput for each stage:

$$throughput_{A+B} \leq minimum(throughput_A, throughput_B)$$

Again, if the stages are of a single service, passing the request from one stage to another usually adds little overhead and has little impact on total throughput. Thus, for first-order analysis that overhead can be ignored, and the relation is usually close to equality.

Consider a computer system with two stages: one that is able to process data at a rate of 1,000 kilobytes per second and a second one at a rate of 100 kilobytes per second. If the fast stage generates one byte of output for each byte of input, the overall throughput must be less than or equal to 100 kilobytes per second. If there is negligible overhead in passing requests between the two stages, then the throughput of the system is equal to the throughput of the bottleneck stage, 100 kilobytes per second. In this case, the utilization of stage 1 is 10% and that of stage 2 is 100%.

When a stage processes requests serially, the throughput and the latency of a stage are directly related. The average number of requests a stage handles is inversely proportional to the average time to process a single request:

$$throughput = \frac{1}{latency}$$

If all stages process requests serially, the average throughput of the complete pipeline is inversely proportional to the average time a request spends in the pipeline. In these pipelines, reducing latency improves throughput, and the other way around.

When a stage processes requests concurrently, as we will see later in this chapter, there is *no* direct relationship between latency and throughput. For stages that process requests concurrently, an increase in throughput may *not* lead to a decrease in latency. A useful analogy is pipes through which water flows with a constant velocity. One can have several parallel pipes (or one fatter pipe), which improves throughput but doesn't change latency.

6.1.2 A Systems Approach to Designing for Performance

To gauge how much improvement we can hope for in reducing a bottleneck, we must identify and determine the performance of the slowest and the next-slowest bottleneck. To improve the throughput of a system in which all stages have equal throughput requires improving *all* stages. On the other hand, improving the stage that has a throughput that is 10 times lower than any other stage's throughput may result in a factor of 10 improvement in the throughput of the whole system. We might determine these bottlenecks by measurements or by using simple analytical calculations based on the performance characteristics of each bottleneck. In principle, the performance of any issue in a computer system can be explained, but sometimes it may require substantial digging to find the explanation; see, for example, the study by Perl and Sites on Windows NT's performance [Suggestions for Further Reading 6.4.1].

One should approach performance optimization from a systems point of view. This observation may sound trivial, but many person-years of work have disappeared in optimizing individual stages that resulted in small overall performance improvements. The reason that engineers are tempted to fine-tune a single stage is that

optimizations result in some measurable benefits. An individual engineer can design an optimization (e.g., replacing a slow algorithm with a faster algorithm, removing unnecessary expensive operations, reorganizing the code to have a fast path, etc.), implement it, and measure it, and can usually observe some performance improvement in that stage. This improvement stimulates the design of another optimization, which results in new benefits, and so on. Once one gets into this cycle, it is difficult to keep the *law of diminishing returns* in mind and realize that further improvements may result in little benefit to the system as a whole.

Since optimizing individual stages typically runs into the law of diminishing returns, an approach that focuses on overall performance is preferred. The iterative approach articulated in Section 1.5.2 achieves this goal because at each iteration the designer must consider whether or not the next iteration is worth performing. If the next iteration identifies a bottleneck that, if removed, shows diminished returns, the designer can stop. If the final performance is good enough, the designer's job is done. If the final performance doesn't meet the target, the designer may have to rethink the whole design or revisit the design specification.

The iterative approach for designing for performance has the following steps:

1. Measure the system to find out whether or not a performance enhancement is needed. If performance is a problem, identify which aspect of performance (throughput or latency) is the problem. For multistage pipelines in which stages process requests concurrently, there is no direct relationship between latency and throughput, so improving latency and improving throughput might require different techniques.

2. Measure again, this time to identify the performance bottleneck. The bottleneck may not be in the place the designer expected and may shift from one design iteration to another.

3. Predict the impact of the proposed performance enhancement with a simple back-of-the-envelope model. (We introduce a few simple models in this chapter.) This prediction includes determining where the next bottleneck will be. A quick way to determine the next bottleneck is to unrealistically assume that the planned performance enhancement will remove the current bottleneck and result in a stage with zero latency and infinite throughput. Under this assumption, determine the next bottleneck and calculate its performance. This calculation will result in one of two conclusions:

   a. Removing the current bottleneck doesn't improve system performance significantly. In this case, stop iterating, and reconsider the whole design or revisit the requirements. Perhaps the designer can adjust the interfaces between stages with the goal of tolerating costly operations. We will discuss several approaches in the next sections.

   b. Removing the current bottleneck is likely to improve the system performance. In this case, focus attention on the bottleneck stage. Consider brute-force methods of relieving the bottleneck stage (e.g., add more memory). Taking

advantage of the $\frac{d(technology)}{dt}$ curve may be less expensive than being clever. If brute-force methods won't relieve the bottleneck, be smart. For example, try to exploit properties of the workload or find better algorithms.

4. Measure the new implementation to verify that the change has the predicted impact. If not, revisit steps 1–3 and determine what went wrong.

5. Iterate. Repeat steps 1–5 until the performance meets the required level.

The rest of this chapter introduces various systems approaches to reducing latency and increasing throughput, as well as simple performance models to predict the resulting performance.

6.1.3 Reducing Latency by Exploiting Workload Properties

Reducing latency is difficult because the designer often runs into physical, algorithmic, and economic limits. For example, sending a message from a client on the east coast of the United States to a service on the west coast is dominated by the speed of light. Looking up an item in a hash table cannot go faster than the best algorithm for implementing hash tables. Building a very large memory that has uniform low latency is economically infeasible.

Once a designer has run into such limits, the common approach is to reduce the latency of some requests, perhaps even at the cost of increasing the latency for other requests. A designer may observe that certain requests are more common than other requests, and use that observation to improve the performance of the frequent operations by splitting the staged pipeline into a *fast path* for the frequent requests and a *slow path* for other requests (see Figure 6.2). For example, a service might remember the results of frequently asked requests so that when it receives a repeat of a recently handled request, it can return the remembered result immediately without having to recompute it. In practice, exploiting non-uniformity in applications

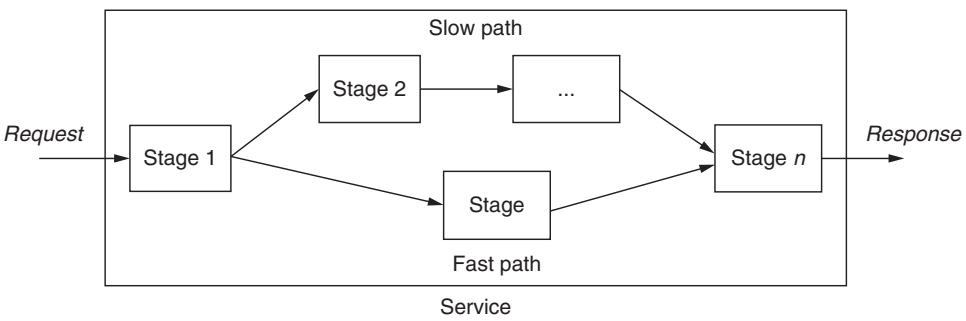


FIGURE 6.2

A simple service with a slow and fast path.

Sidebar 6.2 Design Hint: Optimize for the Common Case A cache (see Section 2.1.1.3) is the most common example of optimizing for the most frequent cases. We saw caches in the case study of the Domain Name System (in Section 4.4). As another example, consider a Web browser. Most Web browsers maintain a cache of recently accessed Web pages. This cache is indexed by the name of the Web page (e.g., http://www.Scholarly.edu) and returns the page for that name. If the user asks to view the same page again, then the cache can return the cached copy of the page immediately (a fast path); only the first access requires a trip to the service (a slow path). In addition to improving the user's interactive experience, the cache helps reduce the load on services and the load on the network. Because caches are so effective, many applications use several of them. For example, in addition to caching Web pages, many Web browsers have a cache to store the results of looking up names, such as "www.Scholarly.edu", so that the next request to "www.Scholarly.edu" doesn't require a DNS lookup.

The design of multilevel memory in Section 6.2 is another example of how well a designer can exploit non-uniformity in a workload. Because applications have locality of reference, one can build large and fast memory systems out of a combination of a small but fast memory and a large but slow memory.

works so well that it has led to the design hint *optimize for the common case* (see Sidebar 6.2).

To evaluate the performance of systems with a fast and slow path, designers typically compute the average latency. If we know the latency of the fast and slow paths, and the frequency with which the system will take the fast path, then the average latency is:

$$AverageLatency = Frequency_{fast} \times Latency_{fast} + Frequency_{slow} \times Latency_{slow} \quad (6.1)$$

Whether introducing a fast path is worth the effort is dependent on the relative difference in latency between the fast and slow path, and the frequency with which the system can use the fast path, which is dependent on the workload. In addition, one might be able to change the design so that the fast path becomes faster at the cost of a slower slow path. If the frequency of taking the fast path is low, then introducing a fast path (and perhaps optimizing it at the cost of the slow path) is likely not worth the complexity. In practice, as we will see in Section 6.2, many workloads don't have a uniform distribution of requests, and introducing a fast path works well.

6.1.4 Reducing Latency using Concurrency

Another way to reduce latency that may require some intellectual effort but that can be effective is to parallelize a stage. We take the processing that a stage must do for a single request and divide that processing up into subtasks that can be performed concurrently. Then, whenever several processors are available they can be assigned to run

those subtasks in parallel. The method can be applied either within a multiprocessor system or (if the subtasks aren't too entangled) with completely separate computers.

If the processing parallelizes perfectly (i.e., each subtask can run without any coordination with other subtasks and each subtask requires the same amount of work), then this plan can, in principle, speed up the processing by a factor n, where n is the number of subtasks executing in parallel. In practice, the speedup is usually less than n because there is overhead in parallelizing a computation—the subtasks need to communicate with each other, for example, to exchange intermediate results; because the subtasks do not require an equal amount of work; because the computation cannot be executed completely in parallel, so some fraction of the computation must be executed sequentially; or because the subtasks interfere with each other (e.g., they contend for a shared resource such as a lock, a shared memory, or a shared communication network).

Consider the processing that a search engine needs to perform in order to respond to a user search query. An early version of Google's search engine—described in more detail in Suggestions for Further Reading 3.2.4—parallelized this processing as follows. The search engine splits the index of the Web up in n pieces, each piece stored on a separate machine. When a front end receives a user query, it sends a copy of the query to each of the n machines. Each machine runs the query against its part of the index and sends the results back to the front end. The front end accumulates the results from the n machines, chooses a good order in which to display them, generates a Web page, and sends it to the user. This plan can give good speedup if the index is large and each of the n machines must perform a substantial, similar amount of computation. It is unlikely to achieve a full speedup of a factor n because there is parallelization overhead (to send the query to the n machines, receive n partial results, and merge them); because the amount of work is not balanced perfectly across the n machines and the front end must wait until the slowest responds; and because the work done by the front end in farming out the query and merging hasn't been parallelized.

Although parallelizing can improve performance, several challenges must be overcome. First, many applications are difficult to parallelize. Applications such as search have exploitable parallelism, but other computations don't split easily into n mostly independent pieces. Second, developing parallel applications is difficult because the programmer must manage the concurrency and coordinate the activities of the different subtasks. As we saw in Chapter 5, it is easy to get this wrong and introduce race conditions and deadlocks. Systems have been developed to make development of parallel applications easier, but they are often limited to a particular domain. The paper by Dean and Ghemawat [Suggestions for Further Reading 6.4.3] provides an example of how the programming and management effort can be minimized for certain stylized applications running in parallel on hundreds of machines. In general, however, programmers must often struggle with threads and locks, or explicit message passing, to obtain concurrency.

Because of these two challenges in parallelizing applications, designers traditionally have preferred to rely on continuous technology improvements to reduce application latency. However, physical and engineering limitations (primarily the problem of heat dissipation) are now leading processor manufacturers away from making processors

faster and toward placing several (and soon, probably, several hundred or even several thousand, as some are predicting [Suggestions for Further Reading 1.6.4]) processors on a single chip. This development means that improving performance by using concurrency will inevitably increase in importance.

6.1.5 Improving Throughput: Concurrency

If the designer cannot reduce the latency of a request because of limits, an alternative approach is to *hide* the latency of a request by overlapping it with other requests. This approach doesn't improve the latency of an individual request, but it can improve system throughput. Because hiding latency is often much easier to achieve than improving latency, it has led to the hint: ***instead of reducing latency, hide it*** (see Sidebar 6.3). This section discusses how one can introduce concurrency in a multistage pipeline to increase throughput.

To overlap requests, we give each stage in the pipeline its own thread of computation so that it can compute concurrently, operating much like an assembly line (see Figure 6.3). If a stage has completed its task and has handed off the request to the next stage, then the stage can start processing the second request while the next stage processes the first request. In this fashion, the pipeline can work on several requests concurrently.

An implementation of this approach has two challenges. First, some stages of the pipeline may operate more slowly than other stage. As a result, one stage might not be able to hand off the request to the next stage because that next stage is still working on a previous request. As a result, a queue of requests may build up, while other stages might be idle. To ensure that a queue between two stages doesn't grow without bound, the stages are often coupled using a bounded buffer. We will discuss queuing in more detail in Section 6.1.6.

The second challenge is that several requests must be available. One natural source of multiple requests is if the system has several clients, each generating a request. A single client can also be a source of multiple requests if the client operates asynchronously. When an asynchronous client issues a request, rather than waiting for the response, it continues computing, perhaps issuing more requests. The main challenge

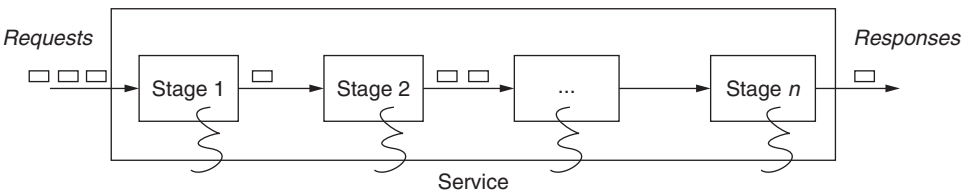


FIGURE 6.3

A simple service composed of several stages, with each stage operating concurrently using threads.

Sidebar 6.3 Design Hint: Instead of Reducing Latency, Hide it Latency is often not under the control of the designer but rather is imposed on the designer by physical properties such as the speed of light. Consider sending a message from the east coast of the United States to the west coast at the speed of light. This takes about 20 milliseconds (see Section 7.1 [online]); in the same time, a processor can execute millions of instructions. Worse, each new generation of processors gets faster every year, but the speed of light doesn't improve. As David Clark, a network researcher, put it succinctly: "One cannot bribe God." The speed of light shows up as an intrinsic barrier in many places of computer design, even when the distances are short. For example, dies are so large that for a signal to travel from one end of a chip to another is a bottleneck that limits the clock speed of a chip.

When a designer is faced with such intrinsic limits, the only option is to design systems that hide latency and try to exploit performance dimensions that do follow d(technology)/dt. For example, transmission rates for data networks have improved dramatically, and so if a designer can organize the system such that communication can be overlapped with useful computation and many network requests can be batched into a large request, then the large request can be transferred efficiently. Many Web browsers use this strategy: while a large transfer runs in the background, users can continue browsing Web pages, hiding the latency of the transfer.

in issuing multiple requests asynchronously is that the client must then match the responses with the outstanding requests.

Once the system is organized to have many requests in flight concurrently, a designer may be able to improve throughput further by using *interleaving*. The idea is to make n instances of the bottleneck stage and run those n instances concurrently (see Figure 6.4). Stage 1 feeds the first request to instance 1, the second request to instance 2, and so on. If the throughput of a single instance is t, then the throughput using interleaving is $n \times t$, assuming enough requests are available to run all instances

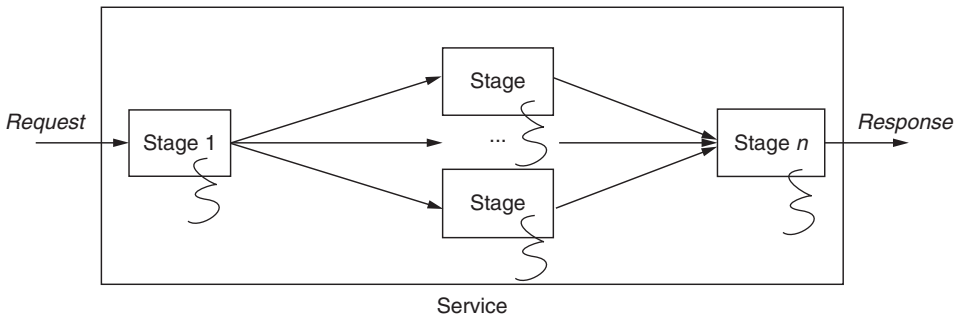


FIGURE 6.4

Interleaving requests.

concurrently at full speed and the requests don't interfere with each other. The cost of interleaving is additional copies of the bottleneck stage.

RAID (see Section 2.1.1.4) interleaves several disks to achieve a high aggregate disk throughput. RAID 0 stripes the data across the disks: it stores block 0 on disk 0, block 1 on disk 1, and so on. If requests arrive for blocks on different disks, the RAID controller can serve those requests concurrently, improving throughput. In a similar style one can interleave memory chips to improve throughput. If the current instruction is stored in memory chip 0 and the next one is in memory chip 1, the processor can retrieve them concurrently. The cost of this design is the additional disks and memory chips, but often systems already have several memory chips or disks, in which case the added cost of interleaving can be small in comparison with the performance benefit.

6.1.6 Queuing and Overload

If a stage in Figure 6.3 operates at its capacity (e.g., all physical processors are running threads), then a new request must wait until the stage becomes available; a queue of requests builds up waiting for the busy stage, while other stages may run idle. For example, the thread manager of Section 5.5 maintains a table of threads, which records whether a thread is runnable; a runnable thread must wait until a processor is available to run it. The stage that runs with an input queue while other stages are running idle is a bottleneck.

Using queuing theory* we can estimate the time that a request spends waiting in a queue for its turn to be processed (e.g., the time a thread spends in the ready queue). In queuing theory, the time that it takes to process a request (e.g., the time from when a thread starts running on the processor until it yields) is called the *service time*. The simplest queuing theory model assumes that requests (e.g., a thread entering the ready queue) arrive according to a random, memoryless process and have independent, exponentially distributed service times. In that case, a well-known queuing theory result tells us that the average queuing delay, measured in units of the average service time and including the service time of this request, will be $1/(1-\rho)$, where ρ is the service utilization. Thus, as the utilization approaches 1, the queuing delay will grow without bound.

This same phenomenon applies to the delays for threads waiting for a processor and to the delays that customers experience in supermarket checkout lines. Any time the demand for a service comes from many statistically independent sources, there will be fluctuations in the arrival of load and thus in the length of the queue at the bottleneck stage and the time spent waiting for service. The rate of arrival of requests for service is known as the *offered load*. Whenever the offered load is greater than the capacity of a service for some duration, the service is said to be *overloaded* for that time period.

*The textbook by Jain is an excellent source to learn about queuing theory and how to reason about performance in computer systems [Suggestions for Further Reading 1.1.2].

In some constrained cases, where the designer can plan the system so that the capacity just matches the offered load of requests, it is possible to calculate the degree of concurrency necessary to achieve high throughput and the maximum length of the queue needed between stages. For example, suppose we have a processor that performs one instruction per nanosecond using a memory that takes 10 nanoseconds to respond. To avoid having the processor wait for the memory, it must make a memory request 10 instructions in advance of the instruction that needs it. If every instruction makes a request of memory, then by the time the memory responds, the processor will have issued 9 more. To avoid being a bottleneck, the memory therefore must be prepared to serve 10 requests concurrently.

If half of the instructions make a request of memory, then on average there will be five outstanding requests. Thus, a memory that can serve five requests concurrently would have enough capacity to keep up. To calculate the maximum length of the queue needed for this case depends on the application's pattern of memory references. For example, if every second instruction makes a memory request, a fixed-size queue of size five is sufficient to ensure that the queue never overflows. If the processor performs five instructions that make memory references followed by five that don't, then a fixed-size queue of size five will work, but the queue length will vary in length and the throughput will be different. If the requests arrive randomly, the queue can grow, in principle, without limit. If we were to use a memory that can handle 10 requests concurrently for this random pattern of memory references, then the memory would be utilized at 50% of capacity, and the average queue length would be $(1/(1-0.5) = 2$. With this configuration, the processor observes latencies for some memory requests of 20 or more instruction cycles, and it is running much slower than the designer expected. This example illustrates that a designer must understand non-uniform patterns in the references to memory and exploit them to achieve good performance.

In many computer systems, the designer cannot plan the offered load that precisely, and thus stages will experience periods of overload. For example, an application may have several threads that become runnable all at the same time and there may not be enough processors available to run them. In such cases, at least occasional overload is inevitable. The significance of overload depends critically on how long it lasts. If the duration is comparable to the service time, then a queue is simply an orderly way to delay some requests for service until a later time when the offered load drops below the capacity of the service. Put another way, a queue handles short bursts of too much demand by time-averaging with adjacent periods when there is excess capacity.

If overload persists over long periods of time, the system designer has only two choices:

1. *Increase the capacity of the system.* If the system must meet the offered load, one approach is to design a system that has less overhead so that it can perform more useful work or purchase a better computer system with higher capacity. In computer systems, it is typically less expensive to buy the next generation of

the computer system that has higher capacity because of technology improvements than trying to squeeze the last ounce out of the implementation through complex algorithms.

2. *Shed load.* If purchasing a computer system with higher capacity isn't an option and system performance cannot be improved, the preferred method is to shed load by reducing or limiting the offered load until the load is less than the capacity of the system.

One approach to control the offered load is to use a bounded buffer (see Figure 5.5) between stages. When the bounded buffer ahead of the bottleneck stage is full, then the stage before it must wait until the bounded buffer empties a slot. Because the previous stage is waiting, its bounded buffer may fill up too, which may cause the stage before it to wait, and so on. The bottleneck may be pushed all the way back to the beginning of the pipeline. If this happens, the system cannot accept any more input, and what happens next depends on how the system is used.

If the source of the load needs the results of the output to generate the next request, then the load will be self-managing. This model of use applies to some interactive systems, in which the users cannot type the next command until the previous one finishes. This same idea will be used in Chapter 7 [on-line] in the implementation of self-pacing network protocols.

If the source of the load decides not to make the request at all, then the offered load decreases. If the source, however, simply holds on to the request and resubmits it later, then the offered load doesn't decrease, but some requests are just deferred, perhaps to a time when the system isn't overloaded.

A crude approach to limiting a source is to put a *quota* on how many requests a source may have outstanding. For example, some systems enforce a rule that an application may not create more than some fixed number of active threads at the same time and may not have more than some fixed number of open files. If a source has reached its quota for a given service, the system denies the next request, limiting the offered load on the system.

An alternative to limiting the offered load is reducing it when a stage becomes overloaded. We will see one example of this approach in Section 6.2. If the address spaces of a number of applications cannot fit in memory, the virtual memory manager can swap out a complete address space of one or more applications so that the remaining applications fit in memory. When the offered load decreases to normal levels, the virtual memory manager can swap in some of the applications that were swapped out.

6.1.7 Fighting Bottlenecks

If the designer cannot remove a bottleneck with the techniques described above, it may be possible instead to fight the bottleneck using one or more of three different techniques: batching, dallying, and speculation.

6.1.7.1 Batching

Batching is performing several requests as a group to avoid the setup overhead of doing them one at a time. Opportunities for batching arise naturally at a bottleneck stage, which may have a queue of requests waiting to be processed. For example, if a stage has several requests to send to the next stage, the stage can combine all of the messages into a single message and send that one message to the next stage. This use of batching divides the overhead of an expensive operation (e.g., sending a message) over the several messages. More generally, batching works well when processing a request has a fixed delay (e.g., transmitting the request) and a variable delay (e.g., performing the operation specified in the request). Without batching, processing n requests takes $n \times (f + v)$, where f is the fixed delay and v is the variable delay. With batching, processing n requests takes $f + n \times v$.

Once a stage performs batching, the potential arises for additional performance wins. Batching may create opportunities for the stage to avoid work. If two or more write requests in a batch are for the same disk block, then the stage can perform just the last one.

Batching may also provide opportunities to improve latency by *reordering* the processing of requests. As we will see in Section 6.3.4, if a disk controller receives a batch of requests, it can schedule them in an order that reduces the movement of the disk arm, reducing the total latency for the batch of requests.

6.1.7.2 Dallying

Dallying is delaying a request on the chance that the operation won't be needed, or to create more opportunities for batching. For example, a stage may delay a request that overwrites a disk block in the hope that a second one will come along for the same block. If a second one comes along, the stage can delete the first request and perform just the second one. As applied to writes, this benefit is sometimes called *write absorption*.

Dallying also increases the opportunities for batching. It purposely increases the latency of some requests in the hope that more requests will come along that can be combined with the delayed requests to form a batch. In this case, dallying increases the latency of some requests to improve the average latency of all requests.

A key design question in dallying is to decide how long to wait. There is no generic answer to this question. The costs and benefits of dallying are application and system specific.

6.1.7.3 Speculation

Speculation is performing an operation in advance of receiving a request on the chance that it will be requested. The goal is that the results can be delivered with less latency and perhaps with less setup overhead. Speculation can achieve this goal in two different ways. First, speculation can perform operations using otherwise idle resources. In this case, even if the speculation is wrong, performing the additional operations has no downside. Second, speculation can use a busy resource to do an

operation that has a long lead time so that the result of the operation can be available without waiting if it turns out to be needed. In this case, speculation might increase the delay and overhead of other requests without benefit because the prediction that the results may be needed might turn out to be wrong.

Speculating may sound bewildering because how can a computer system predict the input of an operation if it hasn't received the request yet, and how can it predict if the result of the operation will be useful in the future? Fortunately, many applications have request patterns that a system designer can exploit to predict an input. In some cases, the input value is evident; for example, a future instruction may add register 5 to register 9, and these register values may be available now. In some cases, the input values can be predicted accurately; for example, a program that asks to read byte n is likely to want to read bytes $n + 1$, $n + 2$, and so on, too. Similarly, for many applications a system can predict what results will be useful in the future. If a program performs instruction n, it will likely soon need the result of instruction $n + 1$; only when the instruction n is a JMP will the prediction be wrong.

Sometimes a system can use speculation even if the system cannot predict accurately what the input to an operation is or whether the result will be useful. For example, if an input has only two values, then the system might create a new thread and have the main thread run with one input value and the second thread with the other input value. Later, when the system knows the value of the input, it terminates the thread that is computing with the wrong value and undoes any changes that thread might have made. This use of speculation becomes challenging when it involves shared state that is updated by different thread, but using techniques presented in Chapter 9 [on-line] it is possible to undo the operations of a thread, even when shared state is involved.

Speculation creates more opportunities for batching and dallying. If the system speculates that a read request for block n will be followed by read requests for blocks $n + 1$ through $n + 8$, then the system can batch those read requests. If a write request might soon be followed by another write request, the system can dally for a while to see if any others come in and, if so, batch all the writes together.

Key design questions associated with speculation are when to speculate and how much. Speculation can increase the load on later stages. If this increase in load results in a load higher than the capacity of a later stage, then requests must wait and latency will increase. Also, any work done that turns out to be not useful is overhead, and performing this unnecessary work may slow down other requests. There is no generic answer to this design question; instead, a designer must evaluate the benefits and cost of speculation in the context of the system.

6.1.7.4 Challenges with Batching, Dallying, and Speculation

Batching, dallying, and speculation introduce complexity because they introduce concurrency. The designer must coordinate incoming requests with the requests that are batched, dallied, or speculated. Furthermore, if the requested operations share variables, the designer must coordinate the references to these variables. Since coordination is difficult to get right, a designer must use these performance-enhancing

techniques with discipline. There is always the risk that by the time the designer has worked out the concurrency problems and the system has made it through the system tests, technology improvements will have made the extra complexity unnecessary. Problem set *14* explores several performance-enhancing techniques and their challenges with a simple multithreaded service.

6.1.8 An Example: The I/O Bottleneck

We illustrate design for performance using batching, dallying, and speculation through a case study involving a magnetic disk such as was described in Sidebar 2.2. The performance problem with disks is that they are made of mechanical components. As a result, reading and writing data to a magnetic disk is slow compared to devices that have no mechanical components, such as RAM chips. The disk is therefore a bottleneck in many applications. This bottleneck is usually referred to as the *I/O bottleneck*.

Recall from Sidebar 2.2 that the performance of reading and writing a disk block is determined by (1) the time to move the head to the appropriate track (the seek latency); (2) plus the time to wait until the requested sector rotates under the disk head (the rotational latency); (3) plus the time to transfer the data from the disk to the computer (the transfer latency).

The I/O bottleneck is getting worse over time. Seek latency and rotational latency are not improving as fast as processor performance. Thus, from the perspective of programs running on ever faster processors, I/O is getting slower over time. This problem is an example of problems due to incommensurate rates of technology improvement. Following the ***incommensurate scaling rule*** of Chapter 1, applications and systems have been redesigned several times over the last few decades to cope with the I/O bottleneck.

To build some intuition for the I/O bottleneck, consider a typical disk of the last decade. The average seek latency (the time to move the head over one-third of the disk) is about 8 milliseconds. The disks spin at 7,200 rotations per minute, which is one rotation every 8.33 milliseconds. On average, the disk has to wait a half rotation for the desired block to be under the disk head; thus, the average rotational latency is 4.17 milliseconds.

Bits read from a disk encounter two potential transfer rate limits, either of which may become the bottleneck. The first limit is mechanical: the rate at which bits spin under the disk heads on their way to a buffer. The second limit is electrical: the rate at which the I/O channel or I/O bus can transfer the contents of the buffer to the computer. A typical modern 400-gigabyte disk has 16,383 cylinders, or about 24 megabytes per cylinder. That disk would probably have 8 two-sided platters and thus 16 read/write heads, so there would be $24/16 = 1.5$ megabytes per track. When rotating at 7,200 revolutions per minute (120 revolutions per second), the bits will go by a head at $120 \times 1.5 = 180$ megabytes per second. The I/O channel speed depends on which standard bus connects the disk to the computer. For the Integrated Device Electronics (IDE) bus, 66 megabytes per second is a common number in practice; for

the Serial ATA 3 bus the limit is 3 gigabytes per second. Thus, the IDE bus would be the bottleneck at 66 megabytes per second; with a Serial ATA 3 bus, the disk mechanics would be the bottleneck at 180 megabytes per second.

Using such a disk and I/O standard, reading a 4-kilobyte block chosen at random takes:

average seek time + average rotation latency + transmission of 4 kilobytes
= 8 + 4.17 + (4 / (180 × 1024)) × 1000 milliseconds
= 8 + 4.17 + 0.02 milliseconds
= 12.19 milliseconds

The throughput for reading randomly chosen blocks one by one is:

= 1000/12.19 × 4 kilobytes per second
= 328 kilobytes per second

The main opportunity to handle the I/O bottleneck is to drive the disk at the transfer rate (180 megabytes per second) instead of the rate of seeks and rotations (327 kilobytes per second). This strategy is an example of hiding latency (moving the disk arm) by exploiting throughput (the high transfer rate between computer and disk).

Consider the following prototypical program, which processes a large input file sequentially and produces an output file sequentially:

```
1    in ← OPEN ("in", READ)              // open "in" for reading
2    out ← OPEN ("out", WRITE)           // open "out" for reading
3
4    while not ENDOFFILE (in) do
5        block ← READ (in, 4096)         // read 4 kilobyte block from in
6        block ← COMPUTE (block)         // compute for 1 millisecond
7        WRITE (out, block, 4096)        // write 4 kilobyte block to out
8    CLOSE (in)
9    CLOSE (out)
```

If we think of this application as a pipeline, then there are the following stages: (1) the file system, which reads data from a disk in response to a READ (line 5); (2) the application, which computes new data using the data read (line 6); and (3) the file system, which writes the new data to the disk (line 7).

If the application is organized naively, without batching, dallying, and speculation, the average time to go around the loop is equal to the latency of the three stages. The latencies of the two file system stages are dominated by the latency of the disk operations, and thus we can approximate the average latency of the loop as follows:

reading 4 kilobytes + 1 millisecond of computation + writing 4 kilobytes
= 12.19 + 1 + 12.19 milliseconds
= 25.38 milliseconds

In practice, the latency might be lower because this calculation assumes that each disk access involves an average seek time, but if the file system has allocated the blocks near each other on the disk, the disk might have to perform only a short seek.

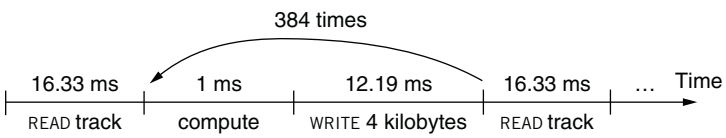
How can we improve the performance of this program? The program reads the file only once, and thus a cache cannot improve the latency of reading a block. The only alternative is to hide the latency of read and write operations. The simplest optimization is to overlap the reading and writing of blocks with the computation on line 6. Let's start with reading.

When the application READs a block, the file system can speculate that the application will read a few blocks following the requested block. This speculation can improve performance for our application if we combine it with two further optimizations. First, we modify the file system to lay out the blocks of a file contiguously. Second, we modify the file system to prefetch an entire track of data on each read. Our prototypical application is perfect for prefetching, since the whole data set is read sequentially.

These optimizations eliminate rotational delay before reading can start. An entire track can be read in:

average seek time + 1 rotational delay
= 8 + 8.33 milliseconds
= 16.33 milliseconds

With 1.5 megabytes (1,536 kilobytes) per track, the file system issues one read request per 384 (1536/4) loop iterations, and we have the following timing diagram:



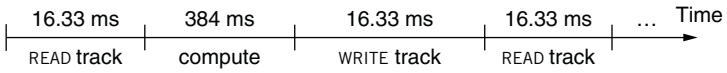
The average time for 384 iterations is:

= reading 1536 kilobytes + 384 × (1 millisecond of computation + writing 4 kilobytes)
= 16.33 + 384 × (1 + 12.19) milliseconds
= 16.33 + 5065 milliseconds
= 5081 milliseconds

Thus, the average time for a loop iteration is 5081/384 = 13.23 milliseconds, a substantial improvement over 25.38 milliseconds.

We can improve the performance of writing blocks by dallying and batching write requests. We modify WRITE to use a buffer of blocks in RAM (see Figure 6.5). The WRITE call stores the updated block into this buffer and returns immediately, and the application thread can continue. When the buffers fill up, the file system can batch the blocks in the buffer and combine them into a single disk request, which the disk can process in parallel with the processor running the application. Batching allows the

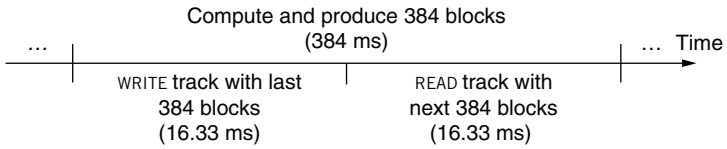
disk controller to execute writes to adjacent sectors with no rotational delay. Because blocks are written contiguously in our example, the file system may take 384 contiguous writes and batch them together to form a complete track write. These optimizations result in the following timing diagram:



This optimization reduces the average time around the loop:

= (16.33 + 384 + 16.33)/384 milliseconds
= 1.09 milliseconds

If we modify the file system to prefetch the next track before the application calls the 385th READ, we can overlap computation and I/O completely. If we modify the file system to read the next track after it has processed, say, half of the last track read, then we obtain the following timing diagram for each block of 384 loop iterations, other than the first one:



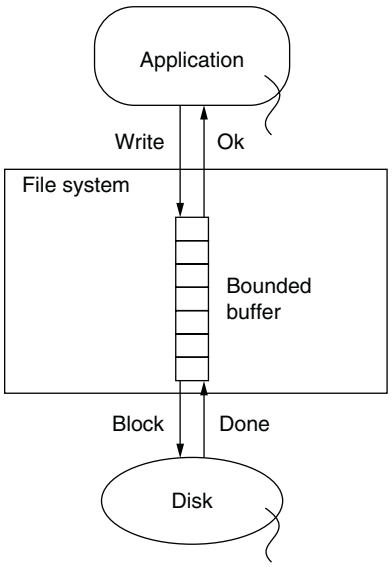


FIGURE 6.5
Using a buffer to delay writes.

Now the system overlaps computation with I/O completely, the average time around the loop is 1 millisecond, and the application is now bottlenecked by computation, rather than by I/O.

The optimizations take advantage of the facts that the application processes the input file sequentially and that the file system allocates blocks for the output contiguously on disk. However, even for applications that process blocks not in the order in which they are laid out on the disk, these optimizations can be beneficial. The file system, for example, can reorder the disk requests for a batch in the order of their track number, thereby minimizing disk arm movement, and thus improving performance for the whole batch of requests. (To understand what a good algorithm is for disk scheduling, we need to think more broadly about scheduling requests in computer systems, which is the topic of Section 6.3.)

The analysis assumes a simple performance model for the disk; for a more in-depth discussion of the performance of disks, see Suggestions for Further Reading 6.3.1. The analysis also assumes a single disk; using several disks can offer opportunities for improving performance. For example, RAIDs have several disks (see Section 2.1.1.4), which allows the file system to interleave read and write requests instead of serving them one by one, providing additional opportunities for increasing performance. Finally, practical, alternative storage technologies are emerging, which change the trade-offs. For example, designing a high performance storage system with Flash disks provides new opportunities and new challenges (see, for example, Suggestions for Further Reading 6.3.4).

A buffer without write-through can provide substantial performance improvements but can lose on reliability. If the computer system fails before the file system has written out data to the disk, some data is lost. The basic problem is how long to delay before forcing the data to the disk. The longer the file system delays writes, the larger the opportunity for higher performance will be, but the greater the probability that data will be lost if, for example, the power fails and the volatile RAM resets.

There are at least four choices as to when the WRITE request to the disk can be issued:

- Before WRITE returns to the caller (write-through).
- On an explicit *force* request from the user (user-controlled write).
- When the file is closed (another kind of user-controlled write).
- When a certain number of write requests have been accumulated or when some fixed time has passed since the last write request. This option can be a bad idea if one needs to control the order of writes.

A buffer without write-through also introduces some other complexities, mostly related to reliability in the face of system failures. First, if the file system batches several write requests in a single disk request, then the disk may write the blocks in an order different from the order issued by the file system to reduce seek time. Thus, the disk may not reflect a consistent state if the system crashes halfway through the batched write request. Second, the disk controller may also use a buffer without write-through. The file system may think the data has been stored reliably on disk when, in fact, the disk controller is caching it. We shall see systematic ways of controlling the problem caused by caches without write-through in Chapter 9 [on-line]; a nice application of these systematic ways to design a high-performance and robust file system is given by Ganger and Patt [Suggestions for Further Reading 6.3.3]. In general, here we have a good example that increased performance comes at the cost of increased complexity, as illustrated by Figure 1.1.

The prototypical application represents one particular workload for which the techniques described above improve performance well. Improving the performance of the prototypical application is challenging because it doesn't reuse a block. Many applications read and write a block multiple times, and in that case additional techniques are available to improve performance. In particular, in that case it is worthwhile for the file system to maintain a cache of recently read blocks in RAM. If an

application reads a block that is already in the cache, then the file system doesn't have to perform any disk operations.

Introducing a cache leads to additional coordination constraints. The file system may have to coordinate WRITE and READ operations with outstanding disk requests. For example: a READ operation may force the removal of a modified block from the cache to make space for the block to be read. But the file system cannot throw out a modified block until it has been written it to the disk, so the file system must wait until the write request of the modified block has completed before proceeding with the READ operation.

Understanding for what workloads a cache works well, learning how to design a cache (e.g., which block to throw out to make space for a new block), and analyzing a cache's performance benefits are sophisticated topics, which we discuss next. Problem set *16* explores these issues, as well as topics related to scheduling, in the context of a simple high-performance video server.

6.2 MULTILEVEL MEMORIES

The previous section described how to address the I/O bottleneck by using two types of digital memory devices: a RAM chip and a magnetic disk, which have different capacities, costs, and speeds. A system designer would like to have a single memory device that is both as large and as fast as the application requires, and that at the same time is affordable. Unfortunately, application requirements often exceed one or another of these three parameters—a memory device that is both fast enough and large enough is usually too expensive—so the designer must make some trade-offs. The usual trade-off is to use more than one memory device, for example, one that is fast but expensive (and thus necessarily too small), and another that is large and cheap (but slower than desired). But fitting an application into such an environment adds the complexity of deciding which parts of the application should use the small, fast memory and which parts the large, slow one. It may also increase maintenance effort if the memory configuration changes.

One might think that improvements in technology may eventually make a brute-force solution economical—someday the designer can just buy a memory that is both large and fast enough. But there are two problems with that thought: one practical and one intrinsic. The practical problem is that historically the increase in memory size has been matched by an equal increase in problem sizes. That is, the data that people want to manipulate has grown along with memory technology.

The intrinsic problem is that memory has a trade-off between latency and size. This trade-off becomes clear when we consider the underlying physics. Even if one has an unlimited budget to throw at the design problem, the speed of light interferes. To see why, imagine a processor that occupies a single point in space, with memory clustered around it in a sphere, using the densest packing that physics allows. With this packing, some of the memory cells will end up located quite near the processor, so the latency (that is, the time required for access to those cells, which requires a

propagation of a signal at the speed of light from the processor to the bit and back) will be short. But because only a few memory cells can fit in the space near the processor, most memory cells will be farther away, and they will necessarily have a larger latency. Put another way, for any specified minimum latency requirement, there will be some memory size for which at least some cells must exceed that latency, based on speed-of-light considerations alone. Moreover, the geometry of spheres (the volume of a shell of radius r grows with the square of r) dictates that there must be more high-latency cells than low-latency ones.

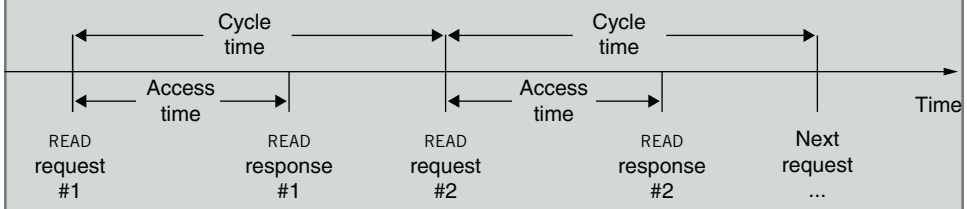
In practical engineering terms, available technologies also exhibit analogous packing problems. For example, the latency of a memory array on the same chip as the processor (where it is usually called an L1 cache) is less than the latency of a separate memory chip (which is usually called an L2 cache), which in turn is less than the latency of a much larger memory implemented as a collection of memory chips on a separate card. The result is that the designer is usually forced to deal with a composite memory system in which different component memories have different parameters of latency, capacity, and cost. The challenge then becomes that of achieving overall maximum performance by deciding which data items to store in the fastest memory device, which can be relegated to the slower devices, and deciding if and when to move data items from one memory device to another.

6.2.1 Memory Characterization

Different memory devices are characterized not just by dimensions of capacity, latency, and cost, but also by cell size and throughput. In more detail, these dimensions are:

- *Capacity*, measured in bits or bytes. For example, a RAM chip may have a capacity from a few to tens of megabytes, whereas magnetic disks have capacities measured in scores or hundreds of gigabytes.
- *Average random latency*, measured in seconds or processor clock cycles, for a memory cell chosen at random. For example, the average latency of RAM is measured in nanoseconds, which might correspond to hundreds of processor clock cycles. (On closer examination, RAM `READ` latency is actually more complicated—see Sidebar 6.4.) Magnetic disks have an average latency measured in milliseconds, which corresponds to millions of processor clock cycles. In addition, magnetic disks, because of their mechanical components, usually have a much wider variance in their latency than does RAM.
- *Cost*, measured in some currency per storage unit. The cost of RAM is typically measured in cents per megabyte, while the cost of magnetic disks is measured in dollars per gigabyte.
- *Cell size,* measured as the number of bits or bytes transferred in or out of the device by a single `READ` or `WRITE` operation. For example, the cell size of RAM is typically a few bytes, perhaps 4, 8, or 16. The cell size of a magnetic disk is typically 512 bytes or more.

Sidebar 6.4 RAM Latency Performance analysis sometimes requires a better model of random access memory latency for READ operations. Most random access memory devices actually have two latency parameters of interest: *cycle time* and *access time*. The distinction arises because the physical memory device may need time to recover from one access before it can handle the next one. For example, some memory READ mechanisms are destructive: to READ a bit out, the memory device literally smashes the bit and examines the resulting debris to determine the value that the bit had. Once it determines that value, the memory device writes the bit back so that its value can again be available for future READs. This write-back operation typically cannot be overlapped with an immediately following READ operation. Thus, the *cycle time* of the memory device is the minimum time that must pass between issuance of one READ request and issuance of the next one. However, the result of the READ may be available for delivery to the processor well before the cycle time is complete. The time from issuance of the READ to delivery of the response to the processor is known as the *access time* of the memory device. The following figure illustrates.



- *Throughput*, measured in bits per second. RAM can typically transfer data at rates measured in gigabytes per second, while magnetic disks transfer at the rate of hundreds of megabytes per second.

The differences between RAM and magnetic disks along these dimensions are orders of magnitude in all cases. RAM is typically about five orders of magnitude faster than magnetic disk and two orders of magnitude more expensive. Many, but not all, of the dimensions have been improving rapidly. For example, the capacity of magnetic disks has doubled, and cost has fallen by a factor of 2 every year for the last two decades, while the average latency has improved by only a factor of 2 in that same 20 years. Latency has not improved much because it involves mechanical operations as opposed to all-electronic ones, as described in Sidebar 2.2. This incommensurate rate of technology improvement makes effective memory management a challenge to implement well.

6.2.2 Multilevel Memory Management using Virtual Memory

Because larger latency and larger capacity usually go hand in hand, it is customary and useful to describe the various available memory devices as belonging to different levels, with the fastest, smallest device being at the highest level and slower, larger

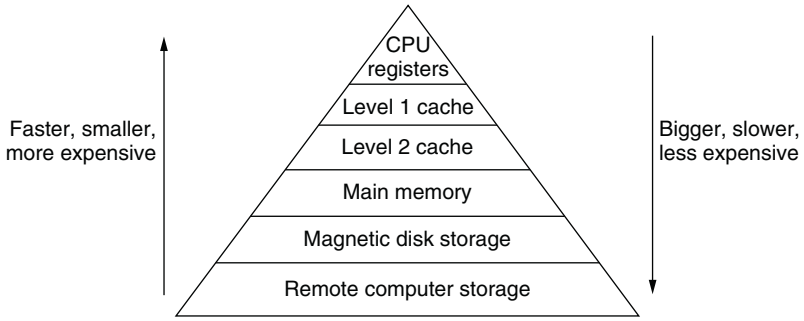


FIGURE 6.6

A multilevel memory pyramid.

devices being at lower levels. A memory system constructed of devices from more than one level is called a *multilevel memory*. Figure 6.6 shows a popular way of depicting the multiple levels, using a pyramid, in which higher levels are narrower, suggesting that their capacity is smaller. The memories in the top of the hierarchy are fast and expensive, and they are therefore small; the memories at the bottom of the hierarchy are slow and inexpensive; and so they can be much bigger. In a modern computer system, an information item can be in the registers of the processor, the L1 cache memory, the L2 cache memory, main memory, a RAM disk cache, on a magnetic disk, or even on another computer that is accessible through a network.

Two quite different ways can be used to manage a multilevel memory. One way is to leave it to each application programmer to decide in which memory to place data items and when to move them. The second way is automatic management: a subsystem independent of any application program observes the pattern of memory references being made by the program. With that pattern in mind, the automatic memory management subsystem decides where to place data items and when to move them from one memory device to another.

Most modern memory management is automatic because (1) there exist automatic algorithms that have good performance on average and (2) automatic memory management relieves the programmer of the need to conform the program to specifics of the memory system such as the capacities of the various levels. Without automatic memory management, the application program explicitly allocates memory space within each level and moves data items from one memory level to another. Such programs become dependent on the particular hardware configuration for which they were written, which makes them difficult to write, to maintain, or to move to a different computer. If someone adds more memory to one of the levels, the program will probably have to be modified to take advantage of it. If some memory is removed, the program may stop working.

As Chapter 2 described, there are two commonly encountered memory interfaces: an interface to small-cell memory to which threads refer using `READ` and `WRITE` operations, and an interface to large-cell memory to which threads refer using `GET` and

PUT operations. These two interfaces correspond roughly to the levels of a multilevel memory; higher levels typically have small cells and use the READ/WRITE interface, while lower levels typically have large cells and use the GET/PUT interface.

One opportunity in the design of an automatically managed multilevel memory system is to combine it with a virtual memory manager in such a way that the small-cell READ/WRITE interface appears to the application program to apply to the entire memory system. This creates what is sometimes called a *one-level store*, an idea first introduced in the Atlas system.* Put another way, this scheme virtualizes the entire memory system around the small-cell READ/WRITE interface, thus hiding from the application programmer the GET/PUT interface as well as the specifics of latency, capacity, cost, cell size, and throughput of the component memory devices. The programmer instead sees a single memory system that appears to have a large capacity, a uniform cell size, a modest average cost per bit, and a latency and throughput that depend on the memory access patterns of the application.

Just as with virtualization of addresses, virtualization of the READ/WRITE memory interface further exploits the design principle *decouple modules with indirection*. In this case, indirection allows the virtual memory manager to translate any particular virtual address not only to different physical memory addresses at different times but also to addresses in a different memory level. With the support of the virtual memory manager, a *multilevel memory manager* can then rearrange the data among the memory levels without having to modify any application program. By adding one more feature, the *indirection exception*, this rearrangement can become completely automatic. An indirection exception is a memory reference exception that indicates that memory manager cannot translate a particular virtual address. The exception handler examines the virtual address and may bind or rebind that value before resuming the interrupted thread.

With these techniques, the virtual memory manager not only can contain errors and enforce modularity, but it also can help make it appear to the program that there is a single, uniform, large memory. The multilevel memory management feature can be slipped in underneath the application program *transparently*, which means that the application program does not need to be modified.

Virtualization of widely used interfaces creates an opportunity to transparently add features and thus evolve a system. Since by definition many modules use a widely used interface, the transparent addition of features beneath such an interface can have a wide impact, without having to change the clients of the interface. The memory interface is an example of such a widely used interface. In addition to implementing single-level stores, here are several other ways in which systems designers have used a virtual memory manager with indirection exceptions:

- *Memory-mapped files*. When an application opens a file, the virtual memory manager can map files into an application's address space, which allows the application to read and write portions of a file as if they were located in RAM. Memory-mapped files extend the idea of a single-level store to include files.

*T. Kilburn, D.B.J. Edwards, M.J. Lanigan, and F.H. Sumner. One-level storage system. *IRE Transactions on Electronic Computers, EC-11*, 2 (April 1962), pages 223–235.

- *Copy-on-write*. If two threads are working on the same data concurrently, then the data can be stored once in memory by mapping the pages that hold the data with only READ permissions. If one of the threads attempts to write a shared page, the virtual memory hardware will interrupt the processor with a permission exception. The handler can demultiplex this exception as an indirection exception of the type copy-on-write. In response to the indirection exception, the virtual memory manager transparently makes a copy of the page and maps the copy with READ and WRITE permissions in the address space of the threads that wants to write the page. With this technique, only changed pages must be copied.
- *On-demand zero-filled pages*. When an application starts, a large part of its address space must be filled with zeros—for instance, the parts of the address space that aren't preinitialized with instructions or initial data values. Instead of allocating zero-filled pages in RAM or on disk, the virtual memory manager can map those pages without READ and WRITE permissions. When the application refers to one of those pages, the virtual memory hardware will interrupt the processor with a memory reference exception. The exception handler can demultiplex this exception as an indirection exception of the type zero-fill. In response to this zero-fill exception, the virtual memory manager allocates a page dynamically and fills it with zeros. This technique can save storage in RAM or on disk because the parts of the address space that the application doesn't use will not take up space.
- *One zero-filled page*. Some designers implement zero-filled pages with a copy-on-write exception. The virtual memory manager allocates just one page filled with zeros and maps that one page in all page-map entries for pages that should contain all zeros, but granting only READ permission. Then, if a thread writes to this read-only zero-filled page, the exception handler will demultiplex this indirect exception as a copy-on-write exception, and the virtual memory manager will make a copy and update that thread's page table to have WRITE permission for the copy.
- *Virtual shared memory*. Several threads running on different computers can share a single address space. When a thread refers to a page that isn't in its local RAM, the virtual memory manager can fetch the page over the network from a remote computer's RAM. The remote virtual memory manager unmaps the page and sends the content of the page back. The Apollo DOMAIN system (mentioned in Suggestions for Further Reading 3.2.1) used this idea to make a collection of distributed computers look like one computer. Li and Hudak use this idea to run parallel applications on a collection of workstations with shared virtual memory [Suggestions for Further Reading 10.1.8].

The virtual memory design for the Mach operating system [Suggestions for Further Reading 6.1.3] provides an example design that supports many of these features and that is used by some current operating systems.

The remainder of this section focuses on building large virtual memories using automatic multilevel memory management. To do so, a designer must address some

challenging problems, but, once it is designed, application programmers do not have to worry about memory management. Except for embedded devices (e.g., a computer acting as the controller of a microwave oven), nearly all modern computer systems use virtual memory to contain errors, enforce modularity, and manage multiple memory levels.

6.2.3 Adding Multilevel Memory Management to a Virtual Memory

Suppose for the moment that we have two memory devices, one that has a READ/WRITE interface, such as a RAM, and the second that has a GET/PUT interface, such as a magnetic disk. If the processor is already equipped with a virtual memory manager such as the one illustrated in Figure 5.20, it is straightforward to add multilevel memory management to create a one-level store.

The basic idea is that at any instant, only some of the pages listed in the page map are actually in RAM (because the RAM has limited capacity) and the rest are on the disk. To support this idea, we add to each entry of the page map a single bit, called the *resident* bit, in the column identified as *r*? in Figure 6.7. If the resident bit of a page is TRUE, that means that the page is in a block of RAM and the physical address in the page map identifies that block. If the resident bit of a page is FALSE, that means that the page is not currently in any block of RAM; it is instead on some block on the disk.

In the example, pages 10 and 12 are in RAM, while page 11 is only on the disk. Thus, references to pages 10 and 12 can proceed as usual, but if the program tries to refer to page 11, for example, with a LOAD instruction, the virtual memory manager must take some action because the processor can't refer to the disk with READ/WRITE operations. The action it takes is to alert the multilevel memory manager that it needs

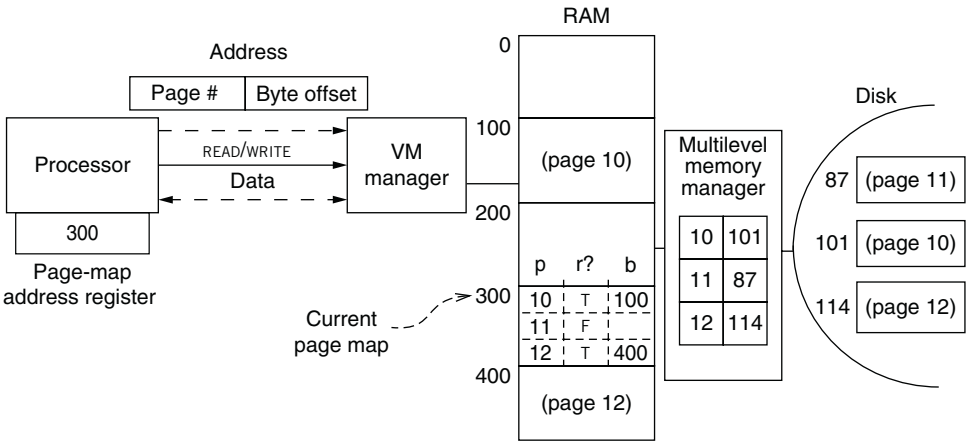


FIGURE 6.7

Integrating a virtual memory manager with a multilevel memory manager. The virtual memory manager is typically implemented in hardware, while the multilevel memory manager is typically implemented in software as part of the operating system.

```
7.1   if resident of page_table[page] = FALSE then      // check if page is resident
7.2       signal MISSING_PAGE (page)                    // no, signal a missing-page exception
7.3   else
7.4       block ← page_table[page].address              // Index into page map
8         physical ← block + offset                     // Concatenate block and offset
9         return physical                               // return physical address
```

FIGURE 6.8

Replacement for lines *7–9* of procedure TRANSLATE of Chapter 5, to implement a multilevel memory.

to use the GET/PUT interface of the disk to copy that page from the disk block into some block in the RAM where the processor can directly refer to it. For this purpose, the multilevel memory manager (at least conceptually) maintains a second, parallel map that translates page numbers to disk block addresses. In practice, real implementations may merge the two maps.

The pseudocode of Figure 6.8 (which replaces lines *7–9* of the version of the TRANSLATE procedure of Section 5.4.3.1) illustrates the integration. When a program makes a reference to a virtual memory address, the virtual memory manager invokes TRANSLATE, which (after performing the usual domain and permission checks) looks up the page number in the page map. If the requested address is in a page that is resident in memory, the manager proceeds as it did in Chapter 5, translating the virtual address to a physical address in the RAM. If the page is not resident, the manager signals that the page is missing.

The pseudocode of Figure 6.8 describes the operation of the virtual memory manager as a procedure, but to maintain adequate performance a virtual memory manager is nearly always implemented in hardware because it must translate every virtual address the processor issues. With this page-based design, the virtual memory manager interrupts the processor with an indirect exception that is called a *missing-page exception* or a *page fault*.

The exception handler examines the value in the program counter register to determine which instruction caused the missing-page exception, and it then examines that instruction in memory to see what address that instruction issued. Next, it calls SEND (see Figure 5.30) with a request containing the missing page number to the port for the multilevel memory manager. SEND invokes ADVANCE, which wakes up a thread of the multilevel memory manager. Then, the handler invokes AWAIT on behalf of the application program's thread (i.e., with the stack of the thread that caused the exception). The AWAIT procedure yields the processor.

The multilevel memory manager receives the request and copies pages between RAM blocks and disk blocks as they are needed. For each missing-page exception, the multilevel memory manager first looks up that page in its parallel page map to determine the address of the disk block that holds the page. Next, it locates an unused block in RAM. With these two parameters, it issues a GET for the disk block that holds the

page, writing the result into the unused RAM block. The multilevel memory manager then informs the virtual memory manager about the presence of the page in RAM by writing the block number in the virtual memory manager's page map and changing the resident bit to TRUE, and makes the thread that experienced the missing-page exception runnable by calling ADVANCE.

When that thread next runs, it backs up the program counter found in its return point so that after the return to user mode the application program will reexecute the instruction that encountered the missing page. Since that page is now resident in RAM and the multilevel memory manager has updated the mappings of the virtual memory manager, this time the TRANSLATE function will be able to translate the virtual address to a physical address.

If all blocks in RAM are occupied with pages, the multilevel memory manager must select some page from RAM and remove it to make space for the missing page. The page selected for removal is known colloquially as the *victim*, and the algorithm that the multilevel memory manager uses to select a victim is called the *page-removal policy*. A bad choice (for example, systematically selecting for removal the page that will be needed by the next memory access) could cause the multilevel memory system to run at the rate that pages can be retrieved from the disk, rather than the rate that words can be retrieved from RAM. In practice, a selection algorithm that exploits a property of most programs known as *locality* can allow those programs to run with only occasional missing-page exceptions. The locality property is discussed in Section 6.2.5, and several different page removal policies are discussed in Section 6.2.6.

If the selected page was modified while it was in RAM, the multilevel memory manager must PUT the modified page back to the disk before issuing a GET for the new page. Thus, in the worst case, a missing-page exception results in two accesses to the disk: one to PUT a modified page back to the disk and one to GET the page requested by the missing-page exception handler. In the best case, the page in RAM has not been modified since being read from disk, so it is identical to the disk copy. In this case, the multilevel memory manager can simply adjust the virtual memory page-map entry to show that this page is no longer resident, and the number of disk accesses needed is just the one to GET the missing page. This scheme maintains a copy of *every* virtual memory page on the disk, whether or not that page is also resident in RAM, so the disk must be larger than the RAM and the effective virtual memory capacity is equal to the space allocated for virtual memory on the disk.

A concern about this scheme is that it introduces what sometimes are called *implicit I/Os*. The multilevel memory manager performs I/O operations beyond the ones performed by the application (which are then called *explicit I/Os*). Given that a disk is often an I/O bottleneck (see Section 6.1.8), these implicit I/Os may risk slowing down the application. Problem set *15* explores some of the issues related to implicit I/Os in the context of a page-based and an object-based single-level store.

To mitigate the I/O bottleneck for missing-page exceptions, a designer can exploit concurrency by implementing the multilevel memory manager with multiple threads. When a missing-page exception occurs, the next available multilevel memory manager thread can start to work on that missing page. The thread begins a GET operation

and waits for the GET to complete. Meanwhile, the thread manager can assign the processor to some other thread. When the GET completes, an interrupt notifies the multilevel memory manager thread and it completes processing of the missing-page exception. With this organization, the multilevel memory manager can overlap the handling of a missing-page exception with the computation of other threads, and it can handle multiple missing-page exceptions concurrently.

A quite different, less modular organization is used in many older systems: integrate the multilevel memory manager with the virtual memory manager in the kernel, with the goal of reducing the number of instructions required to handle a missing-page exception, and thus improving performance. Typically, when integrated, the multilevel memory manager runs in the application thread in the kernel, thus reducing the number of threads and avoiding the cost of context switches. Most such systems were designed decades ago when instruction count was a major concern.

Comparing these two organizations, one benefit of the modularity of a separate multilevel memory manager is that several multilevel memory managers can easily coexist. For example, one multilevel memory manager that reads and writes blocks to a magnetic disk to provide applications with the illusion of a large memory may

Sidebar 6.5 Design Hint: Separate Mechanism from Policy If a module needs to make a policy decision, it is better to leave the policy decision to the clients of the module so that they can make a decision that meets their goals. If the interface between the mechanism and policy module is well defined, then this split allows the schedule policies to be changed without having to change the implementation of the mechanism. For example, one could replace the page-removal policy without having to change the mechanism for handling missing-page exceptions. Furthermore, when porting the missing-page exception mechanism to another processor, the missing-page handler may have to be rewritten, but the policy module may require no modifications.

Of course, if a change in policy requires changes to the interface between the mechanism and policy modules, then both modules must be replaced. Thus, the success of following the hint is limited by how well the interface between the mechanism and policy module is designed. The potential downsides of separating mechanism and policy are a loss in performance due to control transfers between the mechanism and policy module, and increased complexity if flexibility is unneeded. For example, if one policy is always the right one, then separating policy and mechanism may just be unnecessary complexity.

In the case of multilevel memory management, separating the missing-page mechanism from the page replacement policy is mostly for ease of porting because the least recently used page-replacement policy (discussed in Section 6.2.5) works well in practice for most applications.

coexist with another multilevel memory manager that provides memory-mapped files. These different multilevel memory managers can be implemented as separate modules, as opposed to being integrated together with the virtual memory manager. Separating the multilevel memory manager from the virtual memory manager is an example of the design hint *separate mechanism from policy*, discussed in Sidebar 6.5. The Mach virtual memory system is an example of a modern, modular design [Suggestions for Further Reading 6.1.3].

If the multilevel managers are implemented as separate modules from the virtual memory manager, then the designer has the choice of running the multilevel manager modules in kernel mode or as separate applications in user mode. For the same reasons that many deployed systems are monolithic kernel systems (see Section 5.3.6), designers often choose to run the multilevel manager modules in kernel mode. In a few systems, the multilevel managers run as separate user applications with their own address spaces.

One question that requires some careful thought is what to do if a multilevel memory manager encounters a missing-page exception in its own procedures or data. In principle, there is no problem with recursive missing-page exceptions as long as the recursion bottoms out. To ensure that the recursion does bottom out, it is necessary to make sure that some essential set of pages (for example, the pages containing the instructions and tables of the interrupt handler and the kernel thread manager) is never selected for removal from RAM. The usual method is to add a mark to the page-map entries for those essential pages saying, in effect, "Don't remove this page." Pages so marked are commonly said to be *wired down*.

6.2.4 Analyzing Multilevel Memory Systems

Multilevel memories are common engineering practice. From the processor perspective, stored instructions and data traverse some pyramid of memory devices such as the one that was illustrated in Figure 6.6. But when analyzing or constructing a multilevel memory, we do so by analyzing each adjacent pair of levels individually as a two-level memory system, and then stacking the several two-level memory systems. (One reason for doing it this way is that it seems to work. Another is that no one has yet figured out a more satisfactory way to analyze or manage a three- or more-level memory as a single system.)

Devices that function as the fast level in a two-level memory system are called *primary devices*, and devices that function as the slow level are called *secondary devices*. In virtual memory systems, the primary device is usually some form of RAM; the secondary device can be either a slower RAM or a magnetic disk. Web browsers typically use the local disk as a cache that holds pages of remote Web services. In this case, the primary device is a magnetic disk; the remote service is the secondary device, which may itself use magnetic disks for storage. The multilevel memory management algorithms described in the remainder of this section apply to both of these different configurations, and many others.

A cache and a virtual memory are two similar kinds of multilevel memory managers. They are so similar, in fact, that the only difference between them is in the name space they provide for memory cells:

- The user of a *cache* identifies memory cells using the name space of the secondary memory device.
- The user of a *virtual memory* identifies memory cells using the name space of the primary memory device.

Apart from that difference, designers of virtual memories and caches choose policies for multilevel memory management from the same range of possibilities.

The pyramid of Figure 6.6 is typically implemented with the highest level explicitly managed by the application, a cache design at some levels and a virtual memory design at other levels. For example, a multilevel memory system that includes all six levels of the figure might be organized something like the following:

1. At the highest level, the registers of the processor are the primary device, and the rest of the memory system is the secondary device. The application program (as constructed by the compiler code generator) explicitly loads and stores the registers to and from the rest of the memory system.

2. When the processor issues a READ or WRITE to the rest of the memory system, it provides as an argument a name from the main memory name space, but this name goes to a primary memory device located on the same chip as the processor. Since the name is from the lower level main memory name space, this level of memory is being managed as a cache, commonly known as a "level 1 cache" or "L1 cache".

3. If the named cell is not found in the level 1 cache, a multilevel memory manager looks in its secondary memory, an off-chip memory device, but again using the name from the main memory name space. The off-chip memory is thus another example of a cache, this one known as a "level 2 cache" or "L2 cache".

4. The level 2 cache is now the primary device, and if the named memory cell is not found there, the next lower multilevel manager (the one that manages the level 2/main memory pair) looks in its secondary device—the main memory—still using the name from the main memory name space.

5. At the next level, the main memory is the primary device. If an addressed cell is not in main memory, a virtual memory manager invokes the next lower level multilevel memory manager (the one described in Section 6.2.3, that manages movement between main and disk memory) but still using the name from the main memory name space. The multilevel memory manager translates this name to a disk block address.

6. The sequence may continue down another layer; if the disk block is not found on the (primary) local disk, yet another multilevel memory manager may retrieve it from some remote (secondary) system. In some systems, this last memory pair is managed as a cache, and in others as a virtual memory.

It should be apparent that the above example is just one of a vast range of possibilities open to the multilevel memory designer.

6.2.5 Locality of Reference and Working Sets

It is not obvious that an automatically managed multilevel memory system should perform well. The basic requirement for acceptable performance is that all information items stored in the memory must not have equal frequency of use. If every item is used with equal frequency, then a multilevel memory cannot have good performance, since the overall memory will operate at approximately the speed of the slowest memory component. To illustrate this effect, consider a two-level memory system. The average latency of a two-level memory is:

$$AverageLatency = R_{hit} \times Latency_{primary} + R_{miss} \times Latency_{secondary} \tag{6.2}$$

The term R_{hit} (known as the *hit ratio*) is the frequency with which items are found in the primary device, and R_{miss} is $(1 - R_{hit})$. This formula is a direct application of Equation 6.1, (in Section 6.1) which gives the average performance of a system with a fast and slow path. Here the fast path is a reference to the primary device, while the slow path is a reference to the secondary device.

If accesses to every cell of the primary and secondary devices were of equal frequency, then the average latency would be proportional to the number of cells of each device:

$$AverageLatency = \frac{S_{primary}}{S_{primary} + S_{secondary}} \times T_{primary} + \frac{S_{secondary}}{S_{primary} + S_{secondary}} \times T_{secondary} \tag{6.3}$$

where S is the capacity of a memory device and T is its average latency. In a multilevel memory, it is typical that $T_{primary} \ll T_{secondary}$ and $S_{secondary} \gg S_{primary}$ (as, for example, with RAM for primary memory and magnetic disk for secondary memory), in which case the first term is much smaller than the second, the coefficient of the second term approaches 1, and $AverageLatency \approx T_{secondary}$. Thus, if accesses to every cell of primary and secondary are equally likely, a multilevel memory doesn't provide any performance benefit.

On the other hand, if the frequency of use of some stored items is significantly higher than the frequency of use of other stored items, even for a short time, automatically managed multilevel memory becomes feasible. For example, if, somehow, 99% of accesses were directed to the faster memory and only 1% to the slower memory, then the average latency would be:

$$AverageLatency = 0.99 \times T_{primary} + 0.01 \times T_{secondary} \tag{6.4}$$

Thus if the primary device is L2 cache with 1 nanosecond latency and the secondary device is main memory with 10 nanoseconds latency, the average latency becomes 0.99 + 0.10 = 1.09 nanoseconds, which makes the composite memory,

with a capacity equal to that of the main memory, nearly as fast as the L2 cache. For a second example, if the primary device is main memory with 10 nanoseconds latency and the secondary device is magnetic disk with average latency of 10 milliseconds, the average latency of the multilevel memory is

$$0.99 \times 10 \text{ nanoseconds} + 0.01 \times 10 \text{ milliseconds} = 100.0099 \text{ microseconds}$$

That latency is substantially larger than the 10 nanosecond primary memory latency, but it is also much smaller than the 10 millisecond secondary memory latency. In essence, a multilevel memory just exploits the design hint *optimize for the common case*.

Most applications are not so well behaved that one can identify a static set of information that is both small enough to fit in the primary device and for which reference is so concentrated that it is the target of 99% of all memory references. However, in many situations most memory references are to a small set of addresses for significant periods of time. As the application progresses, the area of concentration of access shifts, but its size still typically remains small. This concentration of access into a small but shifting locality is what makes an automatically managed multilevel memory system feasible. An application that exhibits such a concentration of accesses is said to have *locality of reference*.

Analyzing the situation, we can think of a running application as generating a stream of virtual addresses, known as the *reference string*. A reference string can exhibit locality of reference in two ways:

- *Temporal locality*: the reference string contains several closely spaced references to the same address.
- *Spatial locality*: the reference string contains several closely spaced references to adjacent addresses.

An automatically managed multilevel memory system can exploit temporal locality by keeping in the primary device those memory cells that appeared in the reference string recently—thus applying speculation. It can exploit spatial locality by moving into the primary device memory cells that are adjacent to those that have recently appeared in the reference string—a combination of speculation and batching (because issuing a `GET` to a secondary device can retrieve a large block of data that can occupy many adjacent memory cells in the primary device).

There are endless ways in which applications exhibit locality of reference:

- Programs are written as a sequence of instructions. Most of the time, the next instruction is stored in the memory cell that is physically adjacent to the previous instruction, thus creating spatial locality. In addition, applications frequently execute a loop, which means there will be repeated references to the same instructions, creating temporal locality. Between loops, conditional tests, and jumps, it is common to see many instruction references directed to a small subset of all the instructions of an application for an extended time. In addition, depending on the conditional structure, large parts of an application program may not be exercised at all.

- Data structures are typically organized so that a reference to one component of the structure makes references to physically nearby components more likely. Arrays are an example; reference to the first element is likely to be followed shortly by reference to the second. Similarly, if an application retrieves one field of a record, it will likely soon retrieve another field of the same record. Each of these examples creates spatial locality.
- Information processing applications typically process files sequentially. For example, a bank audit program may examine accounts one by one in physical storage order (creating spatial locality) and may perform multiple operations on each account (creating temporal locality).

Although most applications naturally exhibit a significant amount of locality of reference, to a certain extent the concept also embodies an element of self-fulfilling prophecy. Application programmers are usually aware that multilevel memory management is widely used, so they try to write programs that exhibit good locality of reference in the expectation of better performance.

If we look at an application that exhibits locality of reference, in a short time the application refers to only a subset of the total collection of memory cells. The set of references of an application in a given interval Δt is called its *working set.* In one such interval, the application may execute a procedure or loop that operates on a group of related data items, causing most references to go to the text of the procedure and that group of data items. Then, the application might call another procedure, causing most references to go to the text and related data items of that procedure. The working set of an application thus grows, shrinks, and shifts with time.

If at some instant the current working set of an application is entirely stored in the primary memory device, the application will make no references to the secondary device. On the other hand, if the current working set of an application is larger than the primary device, the application (or at least the multilevel memory manager) will have to make at least some references to the secondary device, and it will therefore run more slowly. An application whose working set is much larger than the primary device is likely to cause repeated movement of data back and forth between the primary and secondary devices, a phenomenon called *thrashing*. A design goal is to avoid, or at least minimize, thrashing.

6.2.6 Multilevel Memory Management Policies

Equipped with the concepts of locality of reference and working set, we can now examine the behavior of some common multilevel memory management policies, algorithms that choose which stored objects to place in the primary device, which to place in the secondary device, and when to move a stored object from one device to the other. To make the discussion concrete, we will analyze multilevel memory management policies in the context of a virtual memory system with two levels: RAM (the primary device) and a magnetic disk (the secondary device), in which the stored objects are pages of uniform size. However, it is important to keep in mind that the

same analysis applies to any multilevel memory system, whether organized as a cache or a virtual memory, with uniform or variable-sized objects, and any variety of primary and secondary devices.

Each level of a multilevel memory system can be characterized by four items:

- *The string of references directed to that level.* In a virtual memory system, the reference string seen by the primary device is the sequence of page numbers extracted from virtual addresses of both instructions and data, in the order that the application makes references to them. The reference string seen by the secondary device is the sequence of page numbers that were misses in the primary device. The secondary device reference string is thus a shortened version of the primary device reference string.
- *The bring-in policy for that level.* In a virtual memory system, the usual bring-in policy for the primary device is *on-demand*: whenever a page is used, bring it to the primary device if it is not already there. The only remaining policy decision is whether or not to bring along some adjacent pages. In a two-level memory system there is no need for a bring-in policy for the secondary device.
- *The removal policy for that level.* In the primary device of a virtual memory system, this policy chooses a page to evict (the victim) to make room for a new page. Again, in a two-level memory system there is no need for a removal policy for the secondary device.
- *The capacity of the level.* In a virtual memory system, the capacity of the primary level is the number of primary memory blocks, and the capacity of the secondary level is the number of secondary memory blocks. Since the secondary memory normally contains a copy of every page, the capacity of the multilevel memory system is equal to the capacity of the secondary device.

The goal of a multilevel memory system is to have the primary device serve as many references in its reference string as possible, thereby minimizing the number of references in the secondary device reference string. In the example of the multilevel memory manager, this goal means to minimize the number of missing-page exceptions. One might expect that increasing the capacity of the primary device would guarantee a reduction (or at least not an increase) in the number of missing-page exceptions. Surprisingly, this expectation is not always true. As an example, consider the *first-in, first-out (FIFO) page-removal policy*, in which the page selected for removal is the one that has been in the primary device the longest. (That is, the first page that was brought in will be the first page to be removed. This policy is attractive because it is easy to implement by managing the pages of the primary device as a circular buffer.) If the reference string is 0 1 2 3 0 1 4 0 1 2 3 4, and the primary device starts empty, then a primary device with a capacity of three pages will experience nine missing-page exceptions, while a primary device with a capacity of four pages will experience ten missing-page exceptions, as shown in Tables 6.1 and 6.2:

Table 6.1 FIFO Page-Removal Policy with a Three-Page Primary Device

Time	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	
Reference string	**0**	**1**	**2**	**3**	**0**	**1**	**4**	**0**	**1**	**2**	**3**	**4**	
Primary device contents	–	0	0	0	3	3	3	4	4	4	4	4	Pages brought in
	–	–	1	1	1	0	0	0	0	0	2	2	
	–	–	–	2	2	2	1	1	1	1	1	3	
Remove	–	–	–	0	1	2	3	–	–	0	1	–	
Bring in	0	1	2	3	0	1	4	–	–	2	3	–	9

Table 6.2 FIFO Page-Removal Policy with a Four-Page Primary Device

Time	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	
Reference string	**0**	**1**	**2**	**3**	**0**	**1**	**4**	**0**	**1**	**2**	**3**	**4**	
Primary device contents	–	0	0	0	0	0	0	4	4	4	4	3	Pages brought in
	–	–	1	1	1	1	1	1	0	0	0	0	
	–	–	–	2	2	2	2	2	2	1	1	1	
	–	–	–	–	3	3	3	3	3	3	2	2	
Remove	–	–	–	–	–	–	0	1	2	3	4	0	
Bring in	0	1	2	3	–	–	4	0	1	2	3	4	10

This unexpected increase of missing-page exception numbers with a larger primary device capacity is called *Belady's anomaly*, named after the author of the paper that first reported it. Belady's anomaly is not commonly encountered in practice, but it suggests that when comparing page-removal policies, what appears to be a better policy might actually be worse with a different primary device capacity. As we shall see, one way to simplify analysis is to avoid policies that can exhibit Belady's anomaly.

The objective of a multilevel memory management policy is to select for removal the page that will minimize the number of missing-page exceptions in the future. If we knew the future reference string, we could look ahead to see which pages are about to be touched. The optimal policy would always choose for removal the page not needed for the longest time. Unfortunately, this policy is unrealizable because it requires predicting the future. However, if we run a program and keep track of its reference string, afterwards we can review that reference string to determine how many missing-page exceptions would have occurred if we had used that optimal policy. That result can then be compared with the policy that was actually used to determine how close it is to the optimal one. This unrealizable policy is known as the *optimal (OPT) page-removal policy*. Tables 6.3 and 6.4 show the result of the OPT page-removal policy applied to the same reference string as before.

Table 6.3 The OPT Page-Removal Policy with a Three-Page Primary Device

Time	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	
Reference string	**0**	**1**	**2**	**3**	**0**	**1**	**4**	**0**	**1**	**2**	**3**	**4**	
Primary device contents	–	0	0	0	0	0	0	0	0	0	2	3	Pages brought in
	–	–	1	1	1	1	1	1	1	1	1	1	
	–	–	–	2	3	3	3	4	4	4	4	4	
Remove	–	–	–	2	–	–	3	–	–	0	2	–	
Bring in	0	1	2	3	–	–	4	–	–	2	3	–	7

Table 6.4 The OPT Page-Removal Policy with a Four-Page Primary Device

Time	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	
Reference string	**0**	**1**	**2**	**3**	**0**	**1**	**4**	**0**	**1**	**2**	**3**	**4**	
Primary device contents	–	0	0	0	0	0	0	0	0	0	0	3	Pages brought in
	–	–	1	1	1	1	1	1	1	1	1	1	
	–	–	–	2	2	2	2	2	2	2	2	2	
	–	–	–	–	3	3	3	4	4	4	4	4	
Remove	–	–	–	–	–	–	3	–	–	–	0	–	
Bring in	0	1	2	3	–	–	4	–	–	–	3	–	6

It is apparent from the number of pages brought in that, at least for this reference string, the OPT policy is better than FIFO. In addition, at least for this reference string, the OPT policy gets better when the primary device capacity is larger.

The design goal thus becomes to devise page-removal algorithms that (1) avoid Belady's anomaly, (2) have hit ratios not much worse than the optimal policy, and (3) are mechanically easy to implement.

Some easy-to-implement page-removal policies have an average performance on a wide class of applications that is close enough to the optimal policy to be effective. A popular one is the *least-recently-used (LRU) page-removal policy.* LRU is based on the observation that, more often than not, the recent past is a fairly good predictor of the immediate future. The LRU prediction is that the longer the time since a page has been used, the less likely it will be needed again soon. So LRU selects as its victim the page in the primary device that has not been used for the longest time (that is, the "least-recently-used" page). Let's see how LRU fares when it tackles our example reference string:

Table 6.5 The LRU Page-Removal Policy with a Three-Page Primary Device

Time	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	
Reference string	**0**	**1**	**2**	**3**	**0**	**1**	**4**	**0**	**1**	**2**	**3**	**4**	
Primary device contents	–	0	0	0	0	0	0	0	0	0	0	3	Pages brought in
	–	–	1	1	2	1	1	1	1	1	1	1	
	–	–	–	2	3	3	3	4	4	4	2	2	
Remove	–	–	–	1	–	2	3	–	–	4	0	1	
Bring in	0	1	2	3	–	1	4	–	–	2	3	4	9

Table 6.6 The LRU Page-Removal Policy with a Four-Page Primary Device

Time	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	
Reference string	**0**	**1**	**2**	**3**	**0**	**1**	**4**	**0**	**1**	**2**	**3**	**4**	
Primary device contents	–	0	0	0	0	0	0	0	0	0	0	0	Pages brought in
	–	–	1	1	1	1	1	1	1	1	1	1	
	–	–	–	2	2	2	2	4	2	2	2	2	
	–	–	–	–	3	3	3	3	4	4	4	3	
Remove	–	–	–	–	–	–	2	–	–	–	4	0	
Bring in	0	1	2	3	–	–	4	–	–	–	3	4	8

For this reference string, LRU is better than FIFO for a primary memory device of size 4 but not as good as the OPT policy. And for both LRU and the OPT policy the number of page movements is monotonically non-decreasing with primary device size; these two algorithms avoid Belady's anomaly, for a non-obvious reason that will be explained in Section 6.2.7.

Most useful algorithms require that the new page be the only page that moves in and that only one page move out. Algorithms that have this property are called *demand algorithms.* FIFO, LRU, and some algorithms that implement the OPT policy are demand algorithms. If any other page moves in to primary memory, the algorithm is said to use *prepaging*, one of the topics of Section 6.2.9.

As seen above, LRU is not as good as the OPT policy. Because it looks at history rather than the future, it sometimes throws out exactly the wrong page (the page movement at reference #11 in the four-page memory provides an example). For a more extreme example, a program that runs from top to bottom through a virtual memory that is larger than the primary device will always evict exactly the wrong page. Consider a primary device with capacity of four pages that is part of a virtual

memory that contains five pages being managed with LRU (the letter "F" means that this reference causes a missing-page exception):

Reference string	0	1	2	3	4	0	1	2	3	4	0	1	2
4-page primary device	F	F	F	F	F	F	F	F	F	F	F	F	F

If the application repeatedly cycles through the virtual memory from one end to the other, each reference to a page will result in a page movement. If we start with an empty primary device, references to page 0 through 3 will result in page movements. The reference to page 4 will also result in a page movement, in which LRU will remove page 0, since page 0 has been used least recently. The next reference, to page 0, will also result in a page movement, which leads LRU to remove page 1, since it has been used least recently. As a consequence, the next reference, to page 1, will result in a page movement, replacing page 2, and so on. In short, every access to a page will result in a page movement.

For such an application, a *most-recently-used (MRU) page-removal policy* would be better. MRU chooses as the victim the most recently used page.

Let's see how MRU fares on the contrived example that gave LRU so much trouble:

Reference string	0	1	2	3	4	0	1	2	3	4	0	1	2
4-page primary memory	F	F	F	F	F				F				F

The initial references to pages 0 through 3 result in page movements that fill the empty primary device. The first reference to page 4 will also result in a page movement, replacing page 3, since page 3 has been used most recently. The next reference, to page 0, will *not* result in a missing-page exception since page 0 is still in the primary device. Similarly, the succeeding references to page 1 and 2 will not result in page movements. The second reference to page 3 will result in a page movement, replacing page 2, but then there will be three references that do not require page movements. Thus, with the MRU page-removal policy, our contrived application will experience fewer missing-page exceptions than with the LRU page-removal policy: once in steady state, MRU will result in one page movement per loop iteration.

In practice, however, LRU is surprisingly robust because past references frequently are a reasonable predictor of future references; examples in which MRU does better are uncommon. A secondary reason why LRU works well is that programmers assume that the multilevel memory system uses LRU or some close approximation as the removal policy and they design their programs to work well under that policy.

6.2.7 Comparative Analysis of Different Policies

Once an overall system architecture that includes a multilevel memory system has been laid out, the designer needs to decide two things that will affect performance:

- How large the primary memory device should be
- Which page removal policy to use

These two decisions can be—and in practice often are—supported by an analysis that begins by instrumenting a hardware processor or an emulator of a processor to maintain a trace of the reference string of a running program. After collecting several such traces of typical programs that are to be run on the system under design, these traces can then be used to simulate the operation of a multilevel memory with various primary device sizes and page-removal policies. The usual measure of a multilevel memory's performance is the hit ratio because it is a pure number whose value depends only on the size of the primary device and the page-removal policy. Given the hit ratio and the latency of the primary and secondary memory devices, one can immediately estimate the performance of the multilevel memory system by using Equation 6.2.

In the early 1970s, a team of researchers at the IBM Corporation developed a rapid way of doing such simulations to calculate hit ratios for one class of page-removal policies. If we look more carefully at the "primary device contents" rows of Tables 6.3 and 6.4, we notice that at all times the optimal policy keeps in the three-page memory a subset of the pages that it keeps in the four-page memory. But in FIFO Tables 6.1 and 6.2, at times 8, 9, 11, and 12, this *subset property* does not hold. This difference is no accident; it is the key to understanding how to avoid Belady's anomaly and how to rapidly analyze a reference string to see how a particular policy will perform for any primary device size.

If a page-removal policy can somehow maintain this subset property at all times and for every possible primary device capacity, then a larger primary device can never have more missing-page exceptions than a smaller one. Moreover, if we consider a primary device of capacity n pages and a primary device of capacity $n + 1$ pages, the subset property ensures that the larger primary device contains exactly one page that is not in the smaller primary device. Repeating this argument for every possible primary device size n, we see that the subset property creates a total ordering of all the pages of the multilevel memory system. For example, suppose a memory of size 1 contains page A. A memory of size 2 must also contain page A, plus one other page, perhaps B. A memory of size 3 must then contain pages A and B plus one other page, perhaps C. Thus, the subset property creates the total ordering $\{A, B, C\}$. This total ordering is independent of the actual capacity chosen for the primary memory device.

The IBM research team called this ordering a "stack" (in a use of that word that has no connection with push-down stacks), and page-removal policies that maintain the subset property have since become known as *stack algorithms*. Although requiring the subset property constrains the range of algorithms, there are still several different, interesting, and practical algorithms in the class. In particular, the OPT policy, LRU, and MRU all turn out to be stack algorithms. When a stack algorithm is in use, the virtual memory system keeps just the pages from the front of the ordering in the primary device; it relegates the remaining pages to the secondary device. As a consequence, if $m < n$, the set of pages in a primary device of capacity m is always a subset of the set of pages in a primary device of capacity n. Thus a larger memory will always be able to satisfy all of the requests that a smaller memory could—and with luck some additional requests. Put another way, the total ordering ensures that if a particular reference hits in a primary memory of size n, it will also hit in every memory larger than n. When a

Table 6.7 Simulation of the LRU Page-Removal Policy for Several Primary Device Sizes

Time	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	
Reference string	**0**	**1**	**2**	**3**	**0**	**1**	**4**	**0**	**1**	**2**	**3**	**4**	
Stack contents after reference	0	1	2	3	0	1	4	0	1	2	3	4	Number of moves in
	–	0	1	2	3	0	1	4	0	1	2	3	
	–	–	0	1	2	3	0	1	4	0	1	2	
	–	–	–	0	1	2	3	3	3	4	0	1	
	–	–	–	–	–	–	2	2	2	3	4	0	
Size 1 in/out	0/–	1/0	2/1	3/2	0/3	1/0	4/1	0/4	1/0	2/1	3/2	4/3	12
Size 2 in/out	0/–	1/–	2/0	3/1	0/2	1/3	4/0	0/1	1/4	2/0	3/1	4/2	12
Size 3 in/out	0/–	1/–	2/–	3/0	0/1	1/2	4/3	–/–	–/–	2/4	3/0	4/1	10
Size 4 in/out	0/–	1/–	2/–	3/–	–/–	–/–	4/2	–/–	–/–	2/3	3/4	4/0	8
Size 5 in/out	0/–	1/–	2/–	3/–	–/–	–/–	4/-	–/–	–/–	–/–	–/–	–/–	5

stack algorithm is in use, the hit ratio in the primary device is thus guaranteed to be a non-decreasing function of increasing capacity. Belady's anomaly cannot arise.

The more interesting feature of the total ordering and the subset property is that for a given page-removal policy an analyst can perform a simulation of all possible primary memory sizes, with a single pass through a given reference string, by computing the total ordering associated with that policy. At each reference, some page moves to the top of the ordering, and the pages that were above it either move down or stay in their same place, as dictated by the page-removal policy. The simulation notes, for each primary memory device size of interest, whether or not these movements within the total ordering also correspond to movements between the primary and secondary memory devices. By counting those movements, when it reaches the end of the reference string the simulation can directly calculate the hit ratio for each potential primary memory size. Table 6.7 shows the result of this kind of simulation for the LRU policy when it runs with the reference string used in the previous examples. In this table, the "size n in/out" rows indicate which pages, if any, the LRU policy will choose to bring into and remove from primary memory in order to satisfy the reference above. Note that at every instant of time, the "stack contents after reference" are in order by time since last usage, which is exactly what intuition predicts for the LRU policy.

In contrast, when analyzing a non-stack algorithm such as FIFO, one would have to perform a complete simulation of the reference string for each different primary device capacity of interest and construct a separate table such as the one above for each memory size. It is instructive to try to create a similar table for FIFO.

In addition, since the reference string is available, its future is known, and the analyst can, with another simulation pass (running backward through the reference string), learn how the optimal page-removal policy would have performed on that same string for every memory size of interest. The analyst can then compare the OPT result with various realizable page-removal candidate policies.

Table 6.8 The Optimal Page-Removal Policy for All Primary Memory Sizes

Time	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	
Reference string	**0**	**1**	**2**	**3**	**0**	**1**	**4**	**0**	**1**	**2**	**3**	**4**	
Stack contents after reference	0	1	2	3	0	1	4	0	1	2	3	4	Number of pages removed
	–	0	0	0	3	0	0	4	0	0	0	0	
	–	–	1	1	1	3	1	1	4	4	4	3	
	–	–	–	2	2	2	3	3	3	3	2	2	
	–	–	–	–	–	–	2	2	2	1	1	1	
Size 1 victim	–	0	1	2	3	0	1	4	0	1	2	3	11
Size 2 victim	–	–	1	2	–	3	1	–	1	2	3	4	10
Size 3 victim	–	–	–	2	–	–	4	–	–	2	3	–	7
Size 4 victim	–	–	–	–	–	–	4	–	–	2	–	–	6
Size 5 victim	–	–	–	–	–	–	–	–	–	–	–	–	5

Sidebar 6.6 OPT is a Stack Algorithm and Optimal To see that OPT is a stack algorithm, consider the following description of OPT, in terms of a total ordering:

1. Start with an empty primary device and an empty set that will become a total ordering. As each successive page is touched, note its depth d in the total ordering (if it is not yet in the ordering, set d to infinity) and move it to the front of the total ordering.
2. Then, move the page that was at the front down in the ordering. Move it down until it follows all pages already in the ordering that will be touched before this page is needed again, or to depth d, whichever comes first. This step requires knowing the future.
3. If $d > m$ (where m is the size of the primary memory device), step 1 will require moving a page from the secondary device to the primary device, and step 2 will require moving a page from the primary device to the secondary device.

The result is that, if the algorithm removes a page from primary memory, it will always choose the page that will not be needed for the longest time in the future. Since the total ordering of all pages is independent of the capacity of the primary device, OPT is a stack algorithm. Therefore, for a particular reference string, the set of pages in a primary device of capacity m is always a subset of the set of pages in a primary device of capacity $m + 1$. Table 6.8 illustrates this subset property.

Proof that the optimal page removal policy minimizes page movements, and that it can be implemented as an on-demand stack algorithm, is non-trivial. Table 6.8 illustrates that the statement is correct for the reference string of the previous examples. Sidebar 6.6 provides the intuition of why OPT is a stack algorithm and optimal. The

interested reader can find a detailed line of reasoning in the 1970 paper by the IBM researchers [Suggestions for Further Reading 6.1.2] who introduced stack algorithms and explained in depth how to use them in simulations.

6.2.8 Other Page-Removal Algorithms

Any algorithm based on the LRU policy requires updating recency-of-usage information on every memory reference, whether or not a page moves between the primary and secondary devices. For example, in a virtual memory system every instruction and data reference of the running program causes such an update. But manipulating the representation of this usage information may itself require several memory references, which escalates the cost of the original reference. For this reason, most multilevel memory designers look for algorithms that have approximately the same effect but are less costly to implement. One elegant approximation to LRU is the *clock page-removal algorithm.*

The clock algorithm is based on a modest hardware extension in which the virtual memory manager (implemented in the hardware of the processor) sets to TRUE a bit, called the *referenced* bit, in the page table entry for a page whenever the processor makes a reference that uses that page table entry. If at some point in time the multilevel memory manager clears the referenced bit for every page to FALSE, and then the application runs for a while, a survey of the referenced bits will reveal which pages that application used. The clock algorithm consists of a systematic survey of the referenced bits.

Suppose the physical block numbers of the primary device are arranged in numerical order in a ring (i.e., the highest block number is followed by block number 0), as illustrated in Figure 6.9. All referenced bits are initially set to FALSE, and the system begins running. A little later, in Figure 6.9, we find that the pages residing in blocks 0, 1, 2, 4, 6, and 7 have their referenced bits set to TRUE, indicating that some program touched them. Then, some program causes a missing-page exception, and the system invokes the clock algorithm to decide which resident page to evict in order to make room for the missing page. The clock algorithm maintains a pointer much like a clock arm (which is why it is called the clock algorithm). When the virtual memory system needs a free page, the algorithm begins moving the pointer clockwise, surveying the referenced bits as it goes:

1. If the clock arm comes to a block for which the referenced bit is TRUE, the algorithm sets the referenced bit to FALSE and moves the arm ahead to the next block. Thus,

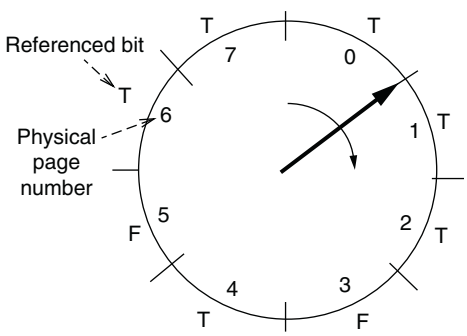


FIGURE 6.9

Example operation of the clock page-removal policy.

the meaning of the referenced bit becomes "The processor has touched the page residing in this block since the last pass of the clock arm."

2. If the clock arm comes to a block for which the referenced bit is FALSE, that means that the page residing in this block has not been touched since the last pass of the clock arm. This page is thus a good candidate for removal, since it has been used less recently than any page that has its referenced bit set to TRUE. The algorithm chooses this page for eviction and leaves the arm pointing to this block for the next execution of the algorithm.

The clock algorithm thus removes the page residing in the first block that it encounters that has a FALSE referenced bit. If there are no such pages (that is, every block in the primary device has been touched since the previous pass of the clock arm), the clock will move all the way around once, resetting referenced bits as it goes, but at the end of that round it will come again to the first block it examined, which now has a FALSE referenced bit, so it chooses the page in that block. If the clock algorithm were run starting in the state depicted in Figure 6.9, it would choose to remove the page in block 3, since that is the first block in the clockwise direction that has a FALSE referenced bit.

The clock algorithm has a number of nice properties. Space overhead is small: just one extra bit per block of the primary device. The extra time spent per page reference is small: forcing a single bit to TRUE. Typically, the clock algorithm has to scan only a small fraction of the primary device blocks to find a page with a FALSE referenced bit. Finally, the algorithm can be run incrementally and speculatively. For example, if the designer of the virtual memory system wants to keep the number of free blocks above some threshold, it can run the policy ahead of demand, removing pages that haven't been used recently, and stop moving the arm as soon as it has met the threshold.

The clock algorithm provides only a rough approximation to LRU. Rather than strictly determining which page has been used least recently, it simply divides pages into two categories: (1) those used since the last sweep and (2) those not used since the last sweep. It then chooses as its victim the first page that the arm happens to encounter in the second category. This page has been used less recently than any of the pages in the first category, but is probably not the least-recently-used page. What seems like the worst-case scenario for the clock algorithm would be when all pages have their referenced bit set to TRUE; then the clock algorithm has no information on which to decide which pages have recently been used. On the other hand, if every page in the primary device has been used since the last sweep, there probably isn't a much better way of choosing a page to remove.

In multilevel memory systems that are completely implemented in hardware, even the clock algorithm may involve too much complexity, so designers resort to yet simpler policies. For example, some processors use a *random removal policy* for the translation look-aside buffer described in Chapter 5. Random removal can be quite effective in this application because

- its implementation requires minimal state to implement.
- if the look-aside buffer is large enough to hold the current working set of translations, the chance that a randomly chosen victim turns out to be a translation that is about to be needed is relatively small.
- the penalty for removing the wrong translation is also quite small—just one extra reference to a slightly slower random access memory.

Alternatively, some processor cache managers use a completely stateless policy called *direct mapping* in which the page chosen for eviction is the one located in block n modulo m, where n is the secondary device block number of the missing page and m is the number of blocks in the primary device. If the compiler optimizer is aware that the processor uses a direct mapping policy, and it knows the size of the primary device, it can minimize the number of cache misses by carefully positioning instructions and data in the secondary device.

When page-removal policies are implemented in software, designers can use methods that maintain more state. One popular software policy is least-frequently-used, which tracks how often a page is used. Complete coverage of page-removal polices is beyond the scope of this book. The reader is encouraged to explore the large literature on this topic.

6.2.9 Other Aspects of Multilevel Memory Management

Page-removal policies are only one aspect of multilevel memory management. The designer of a multilevel memory manager must also provide a bring-in policy that is appropriate for the system load and, for some systems, may include measures to counter thrashing.

The bring-in policy of all of the paging systems described so far is that pages are moved to the primary device only when the application attempts to use them; such systems are called *demand paging* systems. The alternative method is known as *prepaging*. In a prepaging system, the multilevel memory manager makes a prediction about which pages might be needed and brings them in before the application demands them. By moving pages that are likely to be used before they are actually requested, the multilevel memory manager may be able to satisfy a future reference immediately instead of having to wait for the page to be retrieved from a slower memory. For example, when someone launches a new application or restarts one that hasn't been used for a while, none of its pages may be in the primary device. To avoid the delay that would occur from bringing in a large number of pages one at a time, the multilevel memory manager might choose to prepage as a single batch all of the pages that constitute the program text of the application, or all of the data pages that the application used on a previous execution.

Both demand paging and prepaging make use of speculation to improve performance. Demand paging speculates that the application will touch other bytes on the page just brought in. Prepaging speculates that the application will use the prepaged pages.

A problem that arises in a multiple-application system is that the working sets of the various applications may not all simultaneously fit in the primary device. When

that is case, the multilevel memory manager may have to resort to more drastic measures to avoid thrashing. One such drastic measure is *swapping*. When an application encounters a long wait, the multilevel memory manager moves all of its pages out of the primary device in a batch. A batch of writes to the disk can usually be scheduled to go faster than a series of single-block writes (Section 6.3.4 discusses this opportunity). In addition, swapping an application completely out immediately provides space for the other applications, so when they encounter a missing-page exception there is no need to wait to move some page out. However, to do swapping, the multilevel memory manager must be able to quickly identify which pages in primary memory are being used by the application being swapped out, and which of those pages are shared with other applications and therefore should not be swapped out.

Swapping is usually combined with prepaging. When a swapped-out application is restarted, the multilevel memory manager prepages the previous working set of that application, in the hope of later reducing the number of missing-page exceptions. This strategy speculates that when the program restarts, it will need the same pages that it was using before it was swapped out.

The trade-offs involved in swapping and prepaging are formidable, and they resist modeling analysis because reasonably accurate models of application program behavior are difficult to obtain. Fortunately, technology improvements have made these techniques less important for a large class of systems. However, they continue to be applicable to specialized systems that require the utmost in performance.

6.3 SCHEDULING

When a stage is temporarily overloaded in Figure 6.3, a queue of requests builds up. An important policy decision is to determine which requests from the queue to perform first. For example, if the disk has a queue of disk requests, in which order should the disk manager schedule them to minimize latency? For another example, should a stage schedule requests in the order they are received? That policy may result in high throughput, but perhaps in high average latency for individual requests because one client's expensive request may delay several inexpensive requests from other clients. These questions are all examples of the general question of how to schedule resources. This section provides an introduction to systematic answers to this general question. This introduction is sufficient to tackle resource scheduling problems that we encounter in later chapters but scratches only the surface of the literature on scheduling.

Because the technology underlying resources improves rapidly in computer systems, some scheduling decisions become irrelevant over time. For example, in the 1960s and 1970s when several users shared a single computer and the processor was a performance bottleneck, scheduling the processor among users was important. With the arrival of personal computers and the increase in processing power, processor scheduling became mostly irrelevant because it is no longer a performance bottleneck in most situations, and any reasonable policy is good enough. On the other hand, with massive Internet services handling millions of paying customers, the issue of scheduling has increased in importance. The Internet exposes Web sites to extreme

variations in load, which can result in more requests than a server can handle at an instant of time, and the service must make a choice in which order to handle the queued requests.

6.3.1 Scheduling Resources

Computer systems make scheduling decisions at different levels of abstraction. At a high level of abstraction, a Web site selling goods might allocate more memory and processor time to a user who always buys goods than to a user who never buys goods but just browses the catalog. At a lower level of abstraction, a bus arbiter must decide to which processor's memory reference to allocate a shared bus.

Although in these examples allocation decisions are made at different levels of abstraction, the scheduling problem is similar. From the perspective of scheduling, a computer system is a collection of entities that require the use of a set of resources, and *scheduling* is the set of policies and dispatch mechanisms to allocate resources to entities. Examples of entities include threads, address spaces, users, clients, services, and requests. Examples of resources include processor time, physical memory space, disk space, network capacity, and I/O-bus time. Policies to assign resources to entities include dividing the resources equally among the entities, giving one entity priority over another entity, and providing some minimum guarantee by performing admission control on the number of entities. The *scheduler* is the component that implements a policy.

Designing the right policy is difficult because there are usually gaps between the high-level goal and the available policy, between the chosen policy and mechanism to dispatch, and between the chosen mechanism and its actual implementation. We discuss each of these challenges in turn.

The desired scheduling policy might incorporate elements of the environment in which the computer system is used but that are difficult to capture in a computer system. For example, how can a Web site identify a high-value customer (that is, one who is likely to make a large purchase)? The high-value user might never have bought before at this site, or it may be difficult to associate an anonymous catalog-browsing request with a particular previous customer. Even if we could identify the request with a particular customer, the request may traverse several modules of the Web site, some of which may have no notion of users. For example, the database that contains information about prices and goods might be unable to prioritize requests from an important customer.

If we can construct the right policy, then there is the challenge of identifying the mechanism to implement the policy. One module might implement a scheduling policy, but because another module is not aware of it, the policy is ineffective. For example, we might desire to give the text editor high priority to provide a good interactive experience to users. We can easily change the thread scheduler to give the thread running the editor higher priority than any other runnable thread. However, how does the bus arbiter, shared file service, or disk scheduler know that a memory, file, or disk request on behalf of the editor should have higher priority than other disk or memory requests? Worse, the disk scheduler is likely to delay operations to batch

disk requests to achieve high throughput, but this decision may result in bad interactive performance for the text editor because its requests are delayed.

The final challenge is getting the actual implementation of the mechanism right. Sidebar 6.7 on receive livelock provides an example of how easy it is for two schedulers to interact badly. It illustrates that to design a computer system that doesn't collapse under overload is a challenge and requires that a designer carefully think through all implementation decisions.

The list of challenges in designing and implementing schedulers is formidable, but fortunately sophisticated schedulers are often not a requirement for computer systems. Airlines use sophisticated and complex scheduling algorithms because they deal with genuinely expensive and scarce resources (such as airplanes, landing slots, and fuel) and situations in which the peak load can be far larger than usual load (e.g., travel around family holidays). Usually, in a computer system few resources are truly scarce, and simple policies, mechanisms, and implementations suffice.

The rest of Section 6.3 introduces some common goals for a scheduler in a computer system, describes some basic policies to achieve these goals, and presents a case study of scheduling a disk arm. Along the way, the section points out a few scheduling pitfalls, such as receive livelock and priority inversion.

6.3.2 Scheduling Metrics

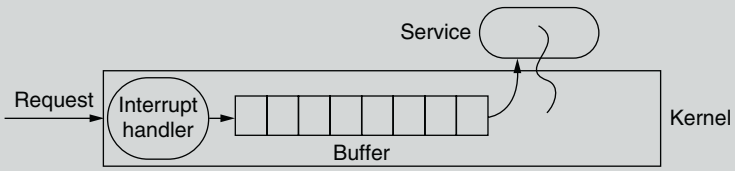
To appreciate possible goals for a scheduler, consider the thread scheduler from the previous chapter. It chooses a thread from a set of runnable threads. In the implementation of the thread manager in Figure 5.24, the scheduler picks the threads in the order in which they appear in the thread table. This scheduling policy is one of many possible policies.

By slightly restructuring the thread scheduler, it could implement different policies easily. A more general implementation of the thread manager would follow the design hint *separate mechanism from policy* (see Sidebar 6.5). This implementation would separate the dispatch mechanism (the mechanisms for suspending and resuming a thread) from scheduling policy (selecting which thread to run next) by putting them into their own procedures, so that a designer can change the policy without having to change the dispatch mechanism.

A designer may want to change the policy because there is no one single best scheduling policy. "Best" might mean different things in different situations. For example, there is tension between achieving good overall efficiency and providing good service to individual requests. From the system's perspective, the two important measures for "best" are throughput and utilization. With a good scheduler, throughput grows linearly with offered load until throughput hits the capacity of the system. A good scheduler will also ensure that a system doesn't collapse under overload conditions. Finally, a good scheduler is efficient: it doesn't consume many resources itself. A scheduler that needs 90% of the processor's time to do its job is not of much value.

Applications achieve high throughput by being immediately scheduled when a request arrives and processing it to completion, without being rescheduled.

Sidebar 6.7 Receive Livelock When a system is temporarily overloaded, it is important to have an effective response to the overload situation. The response doesn't have to be perfect, but it must ensure that the system doesn't collapse. For example, suppose that a Web news server can handle 1,000 requests per second, but a short time ago there was a big earthquake in Los Angeles and requests are arriving at the rate 10,000 per second. The goal is to successfully serve (perhaps a random) 10% of the load, but if the designer isn't careful, the server may end up serving 0%. The problem is called *receive livelock*, and it can arise if the server spends too much of its time saying "I'm too busy" and as a result never gets a chance to serve any of the requests. Consider the following simple interrupt-driven Web service with a bounded buffer:



When a request arrives on the network device, the device generates an interrupt, which causes the interrupt handler to run. The interrupt handler copies the request from the device into a bounded buffer and reenables interrupts so that it can receive the next request. The service has a single thread, which consumes requests from the bounded buffer. When the service is overloaded and requests arrive faster than the service can process them, then the system as described reaches a state where it serves no requests at all because it experiences receive livelock.

Consider what happens when requests arrive much faster than the service can process them. While the service thread is processing a request, the processor receives an interrupt from the network device and the interrupt handler runs. The interrupt handler copies the request into the buffer, notifies the service thread, and returns, reenabling interrupts. As soon as the handler reenables interrupts, the arrival of another request may interrupt the processor again, invoking the interrupt handler. The interrupt handler goes through the same sequence as before until the buffer fills up; then it has no other choice than to discard the request and return from the interrupt, reenabling interrupts. If the network device has another request available, it will interrupt the processor immediately again; the interrupt handler will throw the request away and return. This sequence of events continues indefinitely as long as requests arrive faster than the time for the interrupt handler to run. We have receive livelock: the service never runs, and as a result the number of requests processed by the service per second drops to zero; to users the Web site appears to be down!

The problem here is that the processor's internal scheduler interacts badly with the thread scheduler. Conceptually, the processor schedules the main thread and the

(*Sidebar continues*)

interrupt thread, and the thread manager schedules the main processor thread among the service thread and any other threads. The processor scheduler gives absolute priority to the interrupt thread, scheduling it as soon as an interrupt arrives; the main thread never gets a chance to run the thread manager, and as a result the service thread never receives the processor. This problem occurs when some processing must be performed *outside* of the interrupt handler. One could contemplate moving all processing into interrupt handlers. This approach has its own problems (as discussed in Section 5.6.4) and negates the modularity advantages of using threads. However, once the problem is stated as a scheduling problem, a solution is available.

The solution [Suggestions for Further Reading 6.4.2] is to modify the scheduling policy so that the service thread gets a chance to run when requests are available in the bounded buffer. This policy can be implemented with a slight modification to the interrupt handler. If the bounded buffer fills up, the interrupt handler should not reenable interrupts as it returns. When the service thread has drained the bounded buffer, say, to only half full, it should reenable interrupts. This policy ensures that the network device doesn't discard requests unless the buffer is full (i.e., there is an overload situation) and sees that the service thread gets a chance to process requests, avoiding livelock.

It is still possible that requests may be discarded. If the network device receives a request but it cannot generate an interrupt, the device has no other choice than to discard the next request. This situation is unavoidable: if the network can generate a higher load than the capacity of the service, the device must shed load. The good news is that under overload the system will at least process some requests rather than none at all.

For example, any time a thread scheduler starts a thread, but then preempts it to run another thread, it is delaying the preempted thread. Thus, for an application to achieve high throughput, a scheduler must minimize the number of preemptions and the number of scheduling decisions. Unfortunately, this system-level goal may conflict with the needs of individual threads.

Each individual request wants good service, which typically means good response: it starts soon and completes quickly. There are several ways of measuring a request's response:

- *Turnaround time*. The length of time from when a request arrives at a service until it completes.
- *Response time*. The length of time from when a request arrives at a service until it starts producing output. For interactive requests, this measure is typically more useful than turnaround time. For example, many Web browsers optimize for this metric. Typically, a browser displays an incomplete Web page as soon as the browser receives parts of it (e.g., the text) and fills in the remainder later (e.g., images).

- *Waiting time*. The length of time from when a request arrives at a service until the service starts processing the request. This measure is better than turnaround time, since it captures how long the thread must wait even though it is ready to execute. The ideal waiting time is zero seconds.

More sophisticated measures are also possible by combining the performance of all requests using some of these measures and some way of combining. For example, one can compute *average waiting time* as the average of waiting times of all requests. Similarly, one can calculate the sum of the waiting times, the variance in response time, and so on.

In an interactive computer system, many requests are on behalf of a human user sitting in front of a display. Therefore, the perception of the user is another measure of the goodness of the service that a request receives. For example, an interactive user may tend to perceive a high variance in response time to be more annoying than a high mean. On the other hand, a response time that is faster than the human reaction time may not improve the perception of goodness.

Sometimes a designer desires a scheduler that provides some degree of *fairness*, which means that each request obtains an equal share of the shared service. A scheduler that starves a request to serve other requests is an unfair scheduler. An unfair scheduler is not necessarily a bad scheduler; it may have higher throughput and better response time than a fair scheduler.

It is easy to convince oneself that designing a scheduler that optimizes for fairness, throughput, and response time all at the same time is an impossible task. As a result, there are many different scheduling algorithms; each one of them optimizes along different dimensions.

6.3.3 Scheduling Policies

To illustrate some basic scheduling algorithms, we present a number of them in the context of a thread manager. The objective is to share the processor efficiently among multiple threads. For example, when one thread is blocked waiting for I/O, we would like to run a different, runnable thread on the processor. These threads might be running different programs on a shared computer, or a number of threads that cooperate to implement a high-performance Web service on a dedicated computer.

Since threads typically go through a cycle of running and waiting (e.g., waiting for user input, a client request, or completion of disk request), it is useful to model a thread as a series of *jobs*. Each job corresponds to one burst of activity.

We survey a few different algorithms to schedule these jobs. Many textbooks, lecture notes, and papers explore these algorithms in more detail, and our description is based on this literature. Although the algorithms are described in the context of a thread manager for a single processor, the algorithms are generic and apply to other contexts as well. For example, they work equally well for multiprocessors and have the same pros and cons, but they are harder to illustrate when several jobs run concurrently. The algorithms also apply to disk-arm scheduling, which we shall discuss in Section 6.3.4.

6.3.3.1 First-Come, First-Served

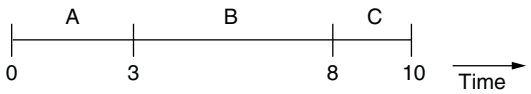
At a busy post office, customers may be asked to take a ticket with a number as they walk in and wait until the number on their ticket is called. Typically, the post office allocates the numbers in strict increasing order and calls the numbers in that order. This policy is called a *first-come, first-served (FCFS) scheduler* and some thread managers use it too.

A thread manager can implement the first-come, first-served policy by organizing the ready list as a first-in, first-out queue. The manager simply runs the first job on the queue until it finishes; then the manager runs the next job, which is now the first job, and so on. When a job becomes ready, the scheduler simply adds it to the end of the queue.

To illustrate and analyze the behavior of a scheduling policy, the literature uses sequences of job arrivals, in which each job has a specific amount of work. We adopt one particular sequence, which illustrates the differences between the scheduling algorithms that we cover. This sequence is the following:

Job	Arrival Time	Amount of Work
A	0	3
B	1	5
C	3	2

Given a specific sequence, one can draw a timeline that depicts when the thread manager dispatches jobs. For the above sequence and the first-come, first-served policy this timeline is as follows:



Given this timeline, one can fill out a table that includes finish time and waiting times, and make some observations about a policy. For the above timeline and the first-come, first-served policy this table is as follows:

Job	Arrival Time	Amount of Work	Start Time	Finish Time	Waiting Time Till Job Starts	Wait Time Till Job is Done
A	0	3	0	3	0	3
B	1	5	3	8	2	7
C	3	2	8	10	5	7
	Total waiting				7	

From the table we can see that for the given job sequence, the first-come, first-served policy favors the long jobs A and B. Job C waits 5 seconds to start a job that takes 2 seconds. Relative to the amount of work, job C is punished the most.

Because first-come, first-served can favor long jobs over short jobs, a system can get into an undesirable state. Consider what happens if we have a system with one thread that periodically waits for I/O but mostly computes and several threads that perform mostly I/O operations. Suppose the scheduler runs the I/O-bound threads first. They will all quickly finish their jobs and go start their I/O operations, leaving the scheduler to run the processor-bound thread. After a while, the I/O-bound threads will finish their I/O and queue up behind the processor-bound thread, leaving all the I/O devices idle. When the processor-bound thread finishes its job, it initiates its I/O operation, allowing the scheduler to run the I/O-bound threads.

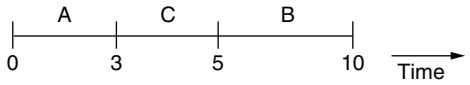
As before, the I/O-bound threads will quickly finish computation and initiate an I/O operation. Now we have the processor sitting idle, while all the threads are waiting for their I/O operations to complete. Since the processor-bound thread started its I/O first, it will likely finish first, grabbing the processor and making all the other threads wait before they can run. The system will continue this way, alternating between periods when the processor is busy and all the I/O devices are idle with periods when the processor is idle and all the threads are doing I/O in a convoy, which is why the literature sometimes refers to this case as a *convoy effect*. The main opportunity for having threads is missed, since in this convoy scenario the system never overlaps computation with I/O.

This scenario is unlikely to materialize in practice because workloads are unlikely to have exactly the right mix of computing and I/O threads that would produce a sequence of scheduling decisions that lead to a situation where I/O isn't overlapped at all with computation. Nevertheless, it has inspired researchers to think about policies other than first-come, first-served.

6.3.3.2 Shortest-Job-First

The undesirable scenario with the first-come first-served policy suggests another scheduler: a *shortest-job-first scheduler*. Whenever the time comes to dispatch a job, the scheduler chooses the job that has the shortest expected running time. Shortest-job-first requires that the scheduler has a prediction of the running time of a job before running it. In the general case, it is difficult to make predictions of the running time of a job, but in practice there are special cases that can work.

Let's assume we know the running time of a job beforehand and see how a shortest-job-first scheduler performs on the example sequence:



As we can see, job C runs before job B because when the scheduler runs after job A completes, it picks C instead of B, since job C has just entered the system and needs less time than job B. Here is the complete table for the shortest-job-first policy:

Job	Arrival Time	Amount of Work	Start Time	Finish Time	Waiting Time Till Job Starts	Waiting Time Till Job is Done
A	0	3	0	3	0	3
B	1	5	5	10	4	9
C	3	2	3	5	0	2
	Total waiting				4	

Job B's waiting time has increased, but relative to the amount of work it has to do, it has to wait less than job C did under the first-come, first-served policy. The *total* amount of waiting time for the shortest-job-first policy decreased compared to the first-come, first-served policy (4 versus 7).

The shortest-job-first policy has one implementation challenge: how do we know the amount of work a job has to do? In some cases, we may be able to decide before running the job whether or not this is a short job. For example, if we have two requests for reading different sectors on the disk and the disk arm is close to one of them, then the request that requires moving the arm to the closer track is the shorter job.

If we cannot decide without executing a job whether or not the job is short, we can make some forward progress by assuming that jobs fall in different classes: a thread that is interactive has mostly short jobs, while a thread that is computationally intensive is likely to have mostly long jobs. This suggests that if we track the past behavior of a thread, then we might be able to predict its future behavior. For example, if a thread just completed a short job, we might predict that its next job also will be short. We can make this idea more precise by basing our prediction on all past jobs of a given thread. One way of doing so is using an *Exponentially Weighted Moving Average (EWMA)* (see Sidebar 7.6 [on-line]). Of course, past behavior may be a weak indicator of future behavior.

A disadvantage of the shortest-job-first policy versus the first-come first-served policy is that shortest-job-first may lead to *starvation*. Several threads that consist entirely of short jobs and that together present a load large enough to use up the available processors may prevent a long job from ever being run. In practice, as we will see in Sections 6.3.3.4 and 6.3.4, the shortest-job-first policy can be combined with other policies to avoid starvation.

6.3.3.3 Round-Robin

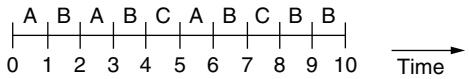
One of the issues with shortest job first is identifying which jobs are short and which are long. One approach is to make all jobs short by breaking long jobs up into a number of smaller jobs using preemptive scheduling. A preemptive scheduling policy stops a job after a certain amount of time so that the scheduler can pick

another job, resuming the preempted one at some time later. As we discussed in Chapter 5, preemptive scheduling also has the benefit that it enforces modularity; a programming error cannot cause a job to never release the processor.

A simple preemptive scheduling policy is *round-robin scheduling.* A round-robin scheduler maintains a queue of runnable jobs as before. It selects the first job from this queue, as in the first-come first-serve policy, but stops the job after some period of time, and selects a new job. Some time later the scheduler will select the stopped job again and run it again for no longer than the fixed period of time, and so on, until the job completes.

Round-robin can be implemented as follows. Before running the job, the round-robin scheduler sets a timer with a fixed time value, called a *quantum*. When the timer expires, it causes an interrupt and the interrupt handler calls YIELD. This call gives control back to the scheduler, which moves the job to the end of the queue and selects a new job from the front of the queue. The quantum should be long enough that most short jobs complete without being interrupted, and it should be short enough that most long jobs do get interrupted so that short jobs can get to run sooner.

Let's look at how a round-robin scheduler with a quantum of 1 second performs on the example sequence:



At time 0, only A is in the queue of runnable jobs, so the scheduler selects it. At time 1, B is in the queue so the scheduler selects B and appends A to the end of the queue, since it is not done. At time 2, A is at the front, so the scheduler selects A and appends B to the end of the queue. At time 3, the scheduler appends C to the end of the queue after B. Then, the scheduler selects B, since it is at the front of the queue, and appends A after C. At time 4, the scheduler appends B to the end of the queue and selects C to run. At time 5, the scheduler appends C to the end of the queue and selects A. At time 6, A is done, and the scheduler selects B, and so on.

This timeline results in the following table:

Job	Arrival Time	Amount of Work	Start Time	Finish Time	Waiting Time Till Job Starts	Waiting Time Till Job is Done
A	0	3	0	6	0	6
B	1	5	1	10	1	9
C	3	2	5	8	2	5
	Total waiting				3	

As can been seen in this example, compared to first-come, first-served and shortest-job-first, round-robin results in the worst performance to complete an individual job, measured in total time elapsed since start. This is not surprising because a round-robin scheduler forces long jobs to stop after a quantum of time.

Round-robin, however, has the shortest total waiting time because with round-robin jobs start earlier: every job runs no longer than a quantum before it is stopped and the scheduler selects another job.

Round-robin favors jobs that run for less than a quantum at the expense of jobs that are more than a quantum long, since the scheduler will stop a long job after one quantum and run the short one before returning the processor to the long one. Round-robin is found in many computer systems because many computer systems are interactive, have short jobs, and a quick response provides a good user experience.

6.3.3.4 Priority Scheduling

Some jobs are more important than others. For example, a system thread that performs minor housekeeping chores such as garbage collecting unused temporary files might be given lower priority than a thread that runs a user program. In addition, if a thread has been blocked for a long time, it might be better to give it higher priority over threads that have run recently.

A scheduler can implement such policies using a *priority scheduling policy*, which assigns each job a priority number. The dispatcher selects the job with the highest priority number. The scheduler must have some rule to break ties, but it doesn't matter much what the rule is, as long as it doesn't consistenly favor one job over another.

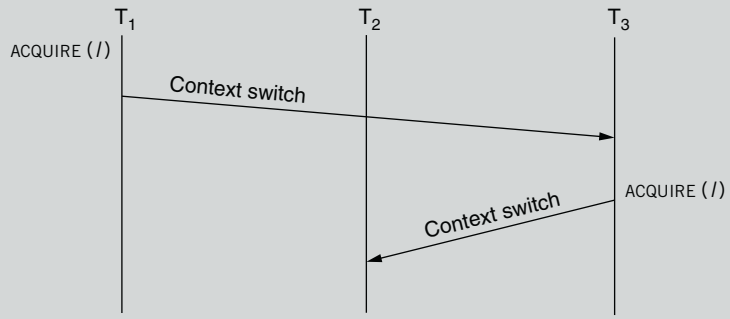
A scheduler can assign priority numbers in many different ways. The scheduler could use a predefined assignment (e.g., systems jobs have priority 1, and user jobs have priority 0) or the priority could be computed using a policy function provided by the system designer. Or the scheduler could compute priorities *dynamically*. For example, if a thread has been waiting to run for a long time, the scheduler could temporarily boost the priority number of the thread's job. This approach can be used, for example, to avoid the starvation problem of the shortest-job-first policy.

A priority scheduler may be preemptive or non-preemptive. In the preemptive version, when a high-priority job enters while a low-priority job is running, the scheduler may preempt the low-priority job and start the high-priority job immediately. For example, an interrupt may notify a high-priority thread. When the interrupt handler calls NOTIFY, a preemptive thread manager may run the scheduler, which may interrupt some other processor that is running a low-priority job. The non-preemptive version would not do any rescheduling or preemption at interrupt time, so the low-priority job would run to completion; when it calls AWAIT, the scheduler will switch to the newly runnable high-priority job.

As we make schedulers more sophisticated, we have to be on the alert for surprising interactions among different schedulers. For example, if a thread manager that provides priorities isn't carefully designed, it is possible that the highest priority thread obtains the least amount of processor time. Sidebar 6.8, which explains priority inversion, describes this pitfall.

Sidebar 6.8 Priority Inversion Priority inversion is a common pitfall in designing a scheduler with priorities. Consider a thread manager that implements a preemptive, priority scheduling policy. Let's assume we have three threads, T_1, T_2, and T_3, and threads T_1 and T_3 share a lock *l* that serializes references to a shared resource. Thread T_1 has a low priority (1), thread T_2 has a medium priority (2), and thread T_3 has a high priority (3).

The following timing diagram shows a sequence of events that causes the high-priority thread T_3 to be delayed indefinitely while the medium priority thread T_2 receives the processor continuously.



Let's assume that T_2 and T_3 are not runnable; for example, they are waiting for an I/O operation to complete. The scheduler will schedule T_1, and T_1 acquires lock *l*. Now the I/O operation completes, and the I/O interrupt handler notifies T_2 and T_3. The scheduler chooses T_3 because it has the highest priority. T_3 runs for a short time until it tries to acquire lock *l*, but because T_1 already holds that lock, T_3 must wait. Because T_2 is runnable and has higher priority than T_1, the thread scheduler will select T_2. T_2 can compute indefinitely; when T_2's time quantum runs out, the scheduler will find two threads runnable: T_1 and T_2. It will select T_2 because T_2 has a higher priority than T_1. As long as T_2 doesn't call WAIT, T_2 will keep the processor. As long as T_2 is runnable, the scheduler won't run T_1, and thus T_1 will not be able to release the lock and T_3, the high priority thread, will wait indefinitely. This undesirable phenomenon is known as *priority inversion*.

The solution to this specific example is simple. When T_3 blocks on acquiring lock *l*, it should temporarily lend its priority to the holder of the lock (sometimes called *priority inheritance*)—in this case, T_1. With this solution, T_1 will run instead of T_2, and as soon as T_1 releases the lock its priority will return to its normal low value and T_3 will run. In essence, this example is one of interacting schedulers. The thread manager schedules the processor and locks schedule references to shared resources. A challenge in designing computer systems is recognizing schedulers and understanding the interactions between them.

(*Sidebar continues*)

The problem and solution have been "discovered" by researchers in the real-time system, database, and operating system communities, and are well documented by now. Nevertheless, it is easy to fall into the priority inversion pitfall. For example, in July 1997 the Mars Pathfinder spacecraft experienced total systems resets on Mars, which resulted in loss of experimental data collected. The software engineers traced the cause of the resets to a priority inversion problem*.

*Mike Jones. What really happened on Mars? *Risks Forum 19,* 49 (December 1997). The Web page http://research.microsoft.com/~mbj/Mars_Pathfinder/Mars_Pathfinder.html includes additional information, including a follow-up by Glenn Reeves, who led the software team for the Mars Pathfinder.

6.3.3.5 Real-Time Schedulers

Certain applications have *real-time* constraints; they require delivery of results before a specified deadline. A chemical process controller, for instance, might have a valve that must be opened every 10 seconds because otherwise a container overflows. Such applications employ *real-time schedulers* to guarantee that jobs complete by the stated deadline.

For some systems, such as a chemical plant, a nuclear reactor, or a hospital intensive-care unit, missing a deadline might result in disaster. Such systems require a *hard real-time scheduler.* For these schedulers, designers must carefully determine the amount of resources each job takes and design the complete system to ensure that all jobs can be handled in a timely manner, even in the worst case. Determining the amount of resources necessary and the time that a job takes, however, is difficult. For example, a system with a cache might sometimes run a job fast (when the job's references hit in the cache) and sometimes slow (when the job's references miss in the cache). Therefore, designers of hard real-time systems make the time a job takes as predictable as possible, either by turning off performance-enhancing techniques (e.g., caches) or by assuming the worst case performance. Typically, designers turn off interrupts and poll devices so that they can carefully control when to interact with a device. These techniques combined increase the likelihood that the designer can estimate when jobs will arrive and for how long they will run. Once the amount of resources and time required for each job are estimated, the designer of a hard real-time system can compute the schedule for executing all jobs.

For other systems, such as a digital music system, missing a deadline occasionally might be just a minor annoyance; such systems can use a *soft real-time scheduler.* A soft real-time scheduler attempts to meet all deadlines but doesn't guarantee it; it may miss a deadline. If, for example, multiple jobs arrive simultaneously, all have 1 second of work, and all have a deadline in 1 second, all jobs except one will miss their deadlines. The goal of a soft real-time scheduler is to avoid missing deadlines but to accept that it might happen when there is more work than there is time before the deadline to do the work.

One popular heuristic for avoiding missing deadlines is the *earliest-deadline-first scheduler*, which keeps the queue of jobs sorted by deadline. The dispatcher runs the first job on the queue, which is always the one with the closest deadline. Most students and faculty follow this policy: work first on the homework or paper that has the earliest deadline.This scheduling policy minimizes the total (summed) lateness of all the jobs.

For soft real-time schedulers that have a given set of jobs that must execute at periodic intervals, we can develop scheduling algorithms instead of just heuristics. Systems with periodic jobs are quite common. For example, a digital video recorder must process a picture frame every 1/30th of a second to make the output look like a movie.

To develop a scheduler for such a system, the total amount of work to be done by the periodic jobs must be less than the capacity of the system. Consider a system with n periodic jobs i that happen with a period of P_i seconds and that each requires C_i seconds.The load of such a system can be handled only if:

$$\left(\sum_{i=1}^{n} \frac{C_i}{P_i}\right) \leq 1$$

If the total amount of work exceeds the system's capacity at any time, then the system will miss a deadline. If the total amount of work is less than the capacity, the system may still miss a deadline occasionally because for some short interval of time the total amount of work to be done is greater than the capacity of the system. For example, a periodic interrupt may arrive at the same time that a periodic task must run. Thus, the condition stated is a necessary condition but not a sufficient one.

A good algorithm for dynamically scheduling periodic jobs is the *rate monotonic scheduler.* In the design phase of the system, the designer assigns each job a priority that is proportional to the frequency of the occurrence of that job. For example, a job that needs to run every 100 milliseconds receives a priority 10, and a job that needs to run every 200 milliseconds receives a priority 5. At runtime, the scheduler always runs the highest priority job, preempting a running job if necessary.

6.3.4 Case Study: Scheduling the Disk Arm

Much work has been done on thread scheduling, but since processors are no longer a usual performance bottleneck, thread scheduling has become less important. As explained in Section 6.1, however, disk arm scheduling is important because the mechanical disk arm creates an I/O bottleneck. The typical goal of a disk arm scheduler is to optimize overall throughput as opposed to the delay for each individual request.

When a disk controller receives a batch of disk requests from the file system, it must decide the order in which to process these requests. At first glance, it might appear that first-come first-served is a fine choice for scheduling the requests, but unfortunately that choice is a bad one.

To see why, recall from Section 6.1 that if the controller moves the disk arm, it reduces the transfer rate of the disk because seeking from one track to another takes time. However, the time required to do a seek depends on how many tracks the arm must cross. A simple, but adequate, model is that a seek from one track to another track that is n tracks away takes $n \times t$ seconds, where t is roughly constant.

Consider a disk controller that is on track 0 and receives four requests that require seeks to the tracks 0 (the innermost track), 90, 5, and 100 (outermost track). If the disk controller performs the four requests in the order in which it received them (first-come first-served), then it will seek first to track 0, then to 90, back to 5, and then forward to 100, for a total seek latency of $270t$:

Request	Movement	Time
Seek 1	$0 \rightarrow 0$	$0t$
Seek 2	$0 \rightarrow 90$	$90t$
Seek 3	$90 \rightarrow 5$	$85t$
Seek 4	$5 \rightarrow 100$	$95t$
Total		$270t$

A much better algorithm is to sort the requests by track number and process them in the sorted order.The total seek latency for that algorithm is $100t$:

Request	Movement	Time
Seek 1	$0 \rightarrow 0$	$0t$
Seek 2	$0 \rightarrow 5$	$5t$
Seek 3	$5 \rightarrow 90$	$85t$
Seek 4	$90 \rightarrow 100$	$10t$
Total		$100t$

In practice, disk scheduling algorithms are more complex because new requests arrive while the disk controller is working on a set of requests. For example, if the disk controller is working on requests in the order of track number (0, 5, 90, and 100), it finishes 5, and receives a new request for track 1, which request should it perform next? It can go back and perform 1, or it can keep going and perform 90 and 100. The first choice is an algorithm that is called shortest seek first; the second choice is called the *elevator algorithm*, named after the algorithm that many elevators execute

to transport people from floor to floor in buildings. With shortest-seek-first, the total seek time is 108*t*:

Request	Movement	Time
Seek 1	$0 \rightarrow 0$	$0t$
Seek 2	$0 \rightarrow 5$	$5t$
Seek 3	$5 \rightarrow 1$	$4t$
Seek 4	$1 \rightarrow 90$	$89t$
Seek 5	$90 \rightarrow 100$	$10t$
Total		$108t$

With the elevator algorithm, the total seek latency is 199*t*:

Request	Movement	Time
Seek 1	$0 \rightarrow 0$	$0t$
Seek 2	$0 \rightarrow 5$	$5t$
Seek 3	$5 \rightarrow 90$	$85t$
Seek 4	$90 \rightarrow 100$	$10t$
Seek 5	$100 \rightarrow 1$	$99t$
Total		$199t$

Many disk controllers use a combination of the shortest-seek-first algorithm and the elevator algorithm. When processing requests, for a while they use the shortest-seek algorithm to choose requests, minimizing seek time, but then switch to the elevator algorithm to avoid starving requests for more distant tracks. For example, if the controller performs the request for track 1 first, starts seeking into the direction of 90, but at track 5 another request for track 1 comes in, then shortest-seek-first would go back to track 1. Since this sequence of events may repeat forever, the disk controller may never serve the request for tracks 90 and 100. By bounding the time that disk controllers perform shortest-seek-first and then switching to the elevator algorithm, requests for the distant tracks will also be served. This method is fine for disk systems, since the primary objective is to maximize total throughput, and thus delaying one request over another is acceptable. In a building, however, people do not want to have long delays, and therefore for buildings the elevator algorithm is better.

EXERCISES

6.1 Suppose a processor has a clock rate of 100 megahertz. The time required to retrieve a word from the cache is 1 nanosecond, and the time required to retrieve a word not in the cache is 101 nanoseconds.

6.1a Determine the hit rate needed such that the average memory latency equals the processor cycle time.

1988–1–4a

6.1b Keeping the same memory devices but considering processors with a higher clock rate, what is the maximum useful clock rate such that the average memory latency equals the processor cycle time, and to what hit rate does it correspond?

1988–1–4b

6.2 A particular program uses 100 data objects, each 10^5 bytes long. The objects are contiguously allocated in a two-level memory system using the LRU page replacement policy with a fast memory of 10^6 bytes and a page size of 10^3 bytes. The program always makes 1,000 accesses to randomly selected bytes in one object, then moves on to another randomly selected object (with probability 0.01 it could be the same object), makes 1,000 accesses to randomly selected bytes there, and so on.

6.2a Ignoring any memory accesses that might be needed for fetching instructions, if the program runs long enough to reach an equilibrium state, what will the hit ratio be?

1987–1–5a

6.2b Will the hit ratio go up or down if the page size is changed from 10^3 words to 10^4 words, with all other memory parameters unchanged?

1987–1–5b

6.3 OutTel corporation has been delivering j786 microprocessors to the computer industry for some time, and Metoo systems has decided to get into the act by building a microprocessor called the "clone786+", which differs from the j786 by providing twice as many processor registers. Metoo has simulated many programs and concluded that this one change reduces the number of loads and stores to memory by an average of 30%, and thus should improve performance, assuming of course that all programs—including its popular microkernel operating system—are recompiled to take advantage of the extra registers. Why might Metoo find the performance improvement to be less than their simulations predict? If there is more than one reason, which one is likely to reduce performance the most?

1994–1–6

6.4 Mike R. Kernel is designing the OutTel P97 computer system, which currently has one page table in hardware. The first tests with this design show excellent performance with one application, but with multiple applications, performance is awful. Suggest three design changes to Mike's system that would improve

performance with multiple applications, and explain your choices briefly. You cannot change processor speed, but any other aspect of the system is fair game.

1996–1–3

6.5 Ben Bitdiddle gets really excited about remote procedure call and implements an RPC package himself. His implementation starts a new service thread for each arriving request. The thread performs the operation specified in the request, sends a reply, and terminates. After measuring the RPC performance, Ben decides that it needs some improvement, so Ben comes up with a brute-force solution: he buys a much faster network. The transit time of the new network is half as large as it was before. Ben measures the performance of small RPCs (meaning that each RPC message contains only a few bytes of data) on the new network. To his surprise, the performance is barely improved. What might be the reason that his RPCs are not twice as fast?

1995–1–5c

6.6 Why might increasing the page size of a virtual memory system increase performance? Why might increasing the page size of a virtual memory system decrease performance?

1993–2–4a

6.7 Ben Bitdiddle and Louis Reasoner are examining a 3.5-inch magnetic disk that spins at 7,500 RPM, with an average seek time of 6.5 milliseconds and a data transfer rate of 10 megabytes per second. Sectors contain 512 bytes of user data.

6.7a On average, how long does it take to read a block of eight contiguous sectors when the starting sector is chosen at random?

6.7b Suppose that the operating system maintains a one-megabyte cache in RAM to hold disk sectors. The latency of this cache is 25 nanoseconds, and for block transfers the data transfer rate from the cache to a different location in RAM is 160 megabytes per second. Explain how these two specifications can simultaneously be true.

6.7c Give a formula that tells the expected time to read 100 randomly chosen disk sectors, assuming that the hit ratio of the disk block cache is h.

6.7d Ben's workstation has 256 megabytes of RAM. To increase the cache hit ratio, Ben reconfigures the disk sector cache to be much larger than one megabyte. To his surprise he discovers that many of his applications now run slower rather than faster. What has Ben probably overlooked?

6.7e Louis has disassembled the disk unit to see how it works. Remembering that the centrifuge in the biology lab runs at 36,000 RPM, he has come up with a bright idea on how to reduce the rotational latency of the disk. He suggests speeding it up to

96,000 RPM. He calculates that the rotation time will now be 625 microseconds. Ben says this idea is crazy. Explain Ben's concern.

1994–3–1

6.8 Ben Bitdiddle has proposed the simple neat and robust file system (SNARFS).* Ben's system has no on-disk data structures other than the disk blocks themselves, which are *self-describing*. Each 4-kilobyte disk block starts with the following 24 bytes of information:

- *fid* (File-ID): a 64-bit number that uniquely defines a file. A *fid* of zero implies that the disk block is free.
- *sn* (Sequence Number): a 64-bit number that identifies which block of a file this disk block contains.
- *t* (Time): The time this block was last updated.

In addition, the first block of a file contains the file name (string), version number, and the *fid* of its parent directory. The rest of the first block is filled with data. Setting the directory *fid* to zero marks the entire file free.

Directories are just files. Each directory should contain only the *fid* of its parent directory. However, as a "hint" directories may also include a table giving the mapping from name to *fid* and the mapping from *fid* to blocks for some of the files in the directory.

To allow fast access, three in-memory (virtual memory) structures are created each time the system is booted:

- MAP: an in-memory hash table that associates a (*fid*, *sn*) pair with the disk block containing that block of that file
- FREE: a free list that represents all of the free blocks on disk in a compact manner
- RECYCLE: a list of blocks that are available for reuse but have not yet been written with a *fid* of 0

6.8a Each read or write of a disk block results in one disk I/O. What is the minimum number of disk I/Os required in SNARFS to create a new file containing 2 kilobytes of data in an existing directory? If the system crashes (i.e., the contents of virtual memory are lost) after these I/Os are completed, the file should be present in the appropriate directory after recovery.

6.8b Ben argues that the in-memory structures can easily be rebuilt after a crash. Explain what actions are required to rebuild MAP, FREE, and RECYCLE at boot time.

1995–3–4a…c

6.9 Ben Bitdiddle has written a "style checker" intended to uncover writing problems in technical papers. The program picks up one sentence at a time, computes intensely for a while to parse it into nouns, verbs, and the like, and then looks up

*Credit for developing *Ex.* 6.8 goes to William J. Dally.

the resulting pattern in an enormous database of linguistic rules. The database was so large that it was necessary to place it on a remote service and do the lookup with a remote procedure call.

Ben is distressed to find that the RPCs have such a long latency that his style checker runs much more slowly than he hoped. He wonders if adding multiple threads to the client could speed up his program.

6.9a Ben's checker is running on a single-processor workstation. Explain how multiple client threads could reduce the time to analyze a technical paper.

6.9b Ben implements a multithreaded style checker and runs a series of experiments with various numbers of threads. He finds that performance does indeed improve when he adds a second thread and again when he adds a third. But he finds that as he adds more and more threads the additional performance improvement diminishes, and finally adding more threads leads to reduced performance. Give an explanation for this behavior.

6.9c Suggest a way of improving the style checker's performance without introducing threads. (Ben is allowed to change only the client.)

1994–1–4a…c

6.10 Threads in a new multithreaded Web browser periodically query a nearby World Wide Web server to retrieve documents. On average, a browser's thread performs a query every N instructions. Each request to the server incurs an average round-trip time of T milliseconds before the answer returns.

6.10a For $N = 2{,}000$ instructions and $T = 1$ millisecond, what is the smallest number of such threads that would be required to keep a single 100 million instructions every second (MIPS) processor 100% busy? Assume that the context switch between threads is instantaneous and that the scheduler is optimal.

6.10b But context switches are not instantaneous. Assume that a context switch takes C instructions to perform. Recompute the answer to 6.10a for $C = 500$ instructions.

6.10c What property of the application threads might cause the answers of parts 6.10a and 6.10b to be incorrect? That is, why might more threads be required to keep the processor running the browser busy?

6.10d What property of the actual computer system might make the answers of 6.10a and 6.10b gross overestimates?

1995–1–4a…d

6.11 What are the advantages of using the clock algorithm as compared with implementing LRU directly?

A. Only a single bit per object or page is required.

B. Clock is more efficient to execute.

C. The first object or page to be purged is the most recently used one.

2001–1–4

6.12 Louis Reasoner found the mention of prepaging systems in Section 6.2.9 to be so intriguing that he has devised a version of OPT that uses prepaging. Here is a description of Louis's prepage-OPT:

- Knowing the reference string, create a total ordering of the pages in which each page is in the order in which the application will next make reference to it. Then, prepage the front of the stack into the primary memory.
- After each page reference, rearrange the ordering so that every page is again in the order in which the application will next make reference to it. Thus, in contrast with LRU, which maintains an ordering since most recent use, prepage-OPT maintains an ordering of next use.

To do this rearrangement requires moving exactly one page, the one that was just touched, down in the ordering to the depth d where it will next be used. All of the pages that were above depth d move up one position. A page that will never be used again is assigned a depth of infinity and moves to the bottom of the stack.This rearrangement scheme ensures that the first page of the ordering is always the page that will be used next.

- If $d > m$ (where m is the size of the primary memory device), the operation of the second bullet will result in a page being moved from the secondary memory to the primary memory. Since the reference string has not yet demanded this page, this movement anticipates a future need, another example of prepaging.

6.11a Is prepage-OPT a stack algorithm? Why or why not?

6.11b For the reference string in the example of Table 6.3, develop a version of that table (or of Table 6.8 if that is more appropriate) that shows what page movements occur with prepage-OPT for each memory size. Assume that the first step of the run is to preload the primary memory with pages from the front of the ordering.

6.11c Is prepage-OPT better or worse than demand-OPT?

2006-0-1

Additional exercises relating to Chapter 6 can be found in the problem sets beginning on page 425.