
# Introduction

## Practice Exercises

**1.1** The three main puropses are:

- To provide an environment for a computer user to execute programs on computer hardware in a convenient and efficient manner.
- To allocate the separate resources of the computer as needed to solve the problem given. The allocation process should be as fair and efficient as possible.
- As a control program it serves two major functions: (1) supervision of the execution of user programs to prevent errors and improper use of the computer, and (2) management of the operation and control of I/O devices.

**1.2** Generally, operating systems for batch systems have simpler requirements than for personal computers. Batch systems do not have to be concerned with interacting with a user as much as a personal computer. As a result, an operating system for a PC must be concerned with response time for an interactive user. Batch systems do not have such requirements. A pure batch system also may have not to handle time sharing, whereas an operating system must switch rapidly between different jobs.

**1.3** The four steps are:

a. Reserve machine time.

b. Manually load program into memory.

c. Load starting address and begin execution.

d. Monitor and control execution of program from console.

**1.4** Single-user systems should maximize use of the system for the user. A GUI might "waste" CPU cycles, but it optimizes the user's interaction with the system.

**1.5** The main difficulty is keeping the operating system within the fixed time constraints of a real-time system. If the system does not complete a task in a certain time frame, it may cause a breakdown of the entire system it is running. Therefore when writing an operating system for a real-time system, the writer must be sure that his scheduling schemes don't allow response time to exceed the time constraint.

**1.6** **Point**. Applications such as web browsers and email tools are performing an increasingly important role in modern desktop computer systems. To fulfill this role, they should be incorporated as part of the operating system. By doing so, they can provide better performance and better integration with the rest of the system. In addition, these important applications can have the same look-and-feel as the operating system software.

**Counterpoint**. The fundamental role of the operating system is to manage system resources such as the CPU, memory, I/O devices, etc. In addition, it's role is to run software applications such as web browsers and email applications. By incorporating such applications into the operating system, we burden the operating system with additional functionality. Such a burden may result in the operating system performing a less-than-satisfactory job at managing system resources. In addition, we increase the size of the operating system thereby increasing the likelihood of system crashes and security violations.

**1.7** The distinction between kernel mode and user mode provides a rudimentary form of protection in the following manner. Certain instructions could be executed only when the CPU is in kernel mode. Similarly, hardware devices could be accessed only when the program is executing in kernel mode. Control over when interrupts could be enabled or disabled is also possible only when the CPU is in kernel mode. Consequently, the CPU has very limited capability when executing in user mode, thereby enforcing protection of critical resources.

**1.8** The following operations need to be privileged: Set value of timer, clear memory, turn off interrupts, modify entries in device-status table, access I/O device. The rest can be performed in user mode.

**1.9** The data required by the operating system (passwords, access controls, accounting information, and so on) would have to be stored in or passed through unprotected memory and thus be accessible to unauthorized users.

**1.10** Although most systems only distinguish between user and kernel modes, some CPUs have supported multiple modes. Multiple modes could be used to provide a finer-grained security policy. For example, rather than distinguishing between just user and kernel mode, you could distinguish between different types of user mode. Perhaps users belonging to the same group could execute each other's code. The machine would go into a specified mode when one of these users was running code. When the machine was in this mode, a member of the group could run code belonging to anyone else in the group.

Another possibility would be to provide different distinctions within kernel code. For example, a specific mode could allow USB device drivers to run. This would mean that USB devices could be serviced without having to switch to kernel mode, thereby essentially allowing USB device drivers to run in a quasi-user/kernel mode.

**1.11** A program could use the following approach to compute the current time using timer interrupts. The program could set a timer for some time in the future and go to sleep. When it is awakened by the interrupt, it could update its local state, which it is using to keep track of the number of interrupts it has received thus far. It could then repeat this process of continually setting timer interrupts and updating its local state when the interrupts are actually raised.

**1.12** The Internet is a WAN as the various computers are located at geographically different places and are connected by long-distance network links.

# Operating System Structures


## Practice Exercises

**2.1** System calls allow user-level processes to request services of the operating system.

**2.2** The five major activities are:

a. The creation and deletion of both user and system processes

b. The suspension and resumption of processes

c. The provision of mechanisms for process synchronization

d. The provision of mechanisms for process communication

e. The provision of mechanisms for deadlock handling

**2.3** The three major activities are:

a. Keep track of which parts of memory are currently being used and by whom.

b. Decide which processes are to be loaded into memory when memory space becomes available.

c. Allocate and deallocate memory space as needed.

**2.4** The three major activities are:

- Free-space management.
- Storage allocation.
- Disk scheduling.

**2.5** It reads commands from the user or from a file of commands and executes them, usually by turning them into one or more system calls. It is usually not part of the kernel since the command interpreter is subject to changes.

**2.6** In Unix systems, a *fork* system call followed by an *exec* system call need to be performed to start a new process. The *fork* call clones the currently

executing process, while the *exec* call overlays a new process based on a different executable over the calling process.

**2.7** System programs can be thought of as bundles of useful system calls. They provide basic functionality to users so that users do not need to write their own programs to solve common problems.

**2.8** As in all cases of modular design, designing an operating system in a modular way has several advantages. The system is easier to debug and modify because changes affect only limited sections of the system rather than touching all sections of the operating system. Information is kept only where it is needed and is accessible only within a defined and restricted area, so any bugs affecting that data must be limited to a specific module or layer.

**2.9** The five services are:

a. **Program execution**. The operating system loads the contents (or sections) of a file into memory and begins its execution. A user-level program could not be trusted to properly allocate CPU time.

b. **I/O operations**. Disks, tapes, serial lines, and other devices must be communicated with at a very low level. The user need only specify the device and the operation to perform on it, while the system converts that request into device- or controller-specific commands. User-level programs cannot be trusted to access only devices they should have access to and to access them only when they are otherwise unused.

c. **File-system manipulation**. There are many details in file creation, deletion, allocation, and naming that users should not have to perform. Blocks of disk space are used by files and must be tracked. Deleting a file requires removing the name file information and freeing the allocated blocks. Protections must also be checked to assure proper file access. User programs could neither ensure adherence to protection methods nor be trusted to allocate only free blocks and deallocate blocks on file deletion.

d. **Communications**. Message passing between systems requires messages to be turned into packets of information, sent to the network controller, transmitted across a communications medium, and reassembled by the destination system. Packet ordering and data correction must take place. Again, user programs might not coordinate access to the network device, or they might receive packets destined for other processes.

e. **Error detection**. Error detection occurs at both the hardware and software levels. At the hardware level, all data transfers must be inspected to ensure that data have not been corrupted in transit. All data on media must be checked to be sure they have not changed since they were written to the media. At the software level, media must be checked for data consistency; for instance, whether the number of allocated and unallocated blocks of storage match the total number on the device. There, errors are frequently process-

independent (for instance, the corruption of data on a disk), so there must be a global program (the operating system) that handles all types of errors. Also, by having errors processed by the operating system, processes need not contain code to catch and correct all the errors possible on a system.

**2.10** For certain devices, such as handheld PDAs and cellular telephones, a disk with a file system may be not be available for the device. In this situation, the operating system must be stored in firmware.

**2.11** Consider a system that would like to run both Windows XP and three different distributions of Linux (e.g., RedHat, Debian, and Mandrake). Each operating system will be stored on disk. During system boot-up, a special program (which we will call the **boot manager**) will determine which operating system to boot into. This means that rather initially booting to an operating system, the boot manager will first run during system startup. It is this boot manager that is responsible for determining which system to boot into. Typically boot managers must be stored at certain locations of the hard disk to be recognized during system startup. Boot managers often provide the user with a selection of systems to boot into; boot managers are also typically designed to boot into a default operating system if no choice is selected by the user.


# Processes

## Practice Exercises

**3.1** a. A method of time sharing must be implemented to allow each of several processes to have access to the system. This method involves the preemption of processes that do not voluntarily give up the CPU (by using a system call, for instance) and the kernel being reentrant (so more than one process may be executing kernel code concurrently).

b. Processes and system resources must have protections and must be protected from each other. Any given process must be limited in the amount of memory it can use and the operations it can perform on devices like disks.

c. Care must be taken in the kernel to prevent deadlocks between processes, so processes aren't waiting for each other's allocated resources.

**3.2** The CPU current-register-set pointer is changed to point to the set containing the new context, which takes very little time. If the context is in memory, one of the contexts in a register set must be chosen and be moved to memory, and the new context must be loaded from memory into the set. This process takes a little more time than on systems with one set of registers, depending on how a replacement victim is selected.

**3.3** Only the shared memory segments are shared between the parent process and the newly forked child process. Copies of the stack and the heap are made for the newly created process.

**3.4** The "exactly once" semantics ensure that a remore procedure will be executed exactly once and only once. The general algorithm for ensuring this combines an acknowledgment (ACK) scheme combined with timestamps (or some other incremental counter that allows the server to distinguish between duplicate messages).
The general strategy is for the client to send the RPC to the server along with a timestamp. The client will also start a timeout clock. The client

will then wait for one of two occurrences: (1) it will receive an ACK from the server indicating that the remote procedure was performed, or (2) it will time out. If the client times out, it assumes the server was unable to perform the remote procedure so the client invokes the RPC a second time, sending a later timestamp. The client may not receive the ACK for one of two reasons: (1) the original RPC was never received by the server, or (2) the RPC was correctly received—and performed—by the server but the ACK was lost. In situation (1), the use of ACKs allows the server ultimately to receive and perform the RPC. In situation (2), the server will receive a duplicate RPC and it will use the timestamp to identify it as a duplicate so as not to perform the RPC a second time. It is important to note that the server must send a second ACK back to the client to inform the client the RPC has been performed.

**3.5** The server should keep track in stable storage (such as a disk log) information regarding what RPC operations were received, whether they were successfully performed, and the results associated with the operations. When a server crash takes place and a RPC message is received, the server can check whether the RPC had been previously performed and therefore guarantee "exactly once" semanctics for the execution of RPCs.


# Threads

## Practice Exercises

**4.1** (1) A Web server that services each request in a separate thread. 2) (A parallelized application such as matrix multiplication where (different parts of the matrix may be worked on in parallel. (3) An (interactive GUI program such as a debugger where a thread is used (to monitor user input, another thread represents the running (application, and a third thread monitors performance.

**4.2** (1) User-level threads are unknown by the kernel, whereas the kernel is aware of kernel threads. (2) On systems using either M:1 or M:N mapping, user threads are scheduled by the thread library and the kernel schedules kernel threads. (3) Kernel threads need not be associated with a process whereas every user thread belongs to a process. Kernel threads are generally more expensive to maintain than user threads as they must be represented with a kernel data structure.

**4.3** Context switching between kernel threads typically requires saving the value of the CPU registers from the thread being switched out and restoring the CPU registers of the new thread being scheduled.

**4.4** Because a thread is smaller than a process, thread creation typically uses fewer resources than process creation. Creating a process requires allocating a process control block (PCB), a rather large data structure. The PCB includes a memory map, list of open files, and environment variables. Allocating and managing the memory map is typically the most time-consuming activity. Creating either a user or kernel thread involves allocating a small data structure to hold a register set, stack, and priority.

**4.5** Yes. Timing is crucial to real-time applications. If a thread is marked as real-time but is not bound to an LWP, the thread may have to wait to be attached to an LWP before running. Consider if a real-time thread is running (is attached to an LWP) and then proceeds to block (i.e. must perform I/O, has been preempted by a higher-priority real-time thread, is waiting for a mutual exclusion lock, etc.) While the real-time thread is

blocked, the LWP it was attached to has been assigned to another thread. When the real-time thread has been scheduled to run again, it must first wait to be attached to an LWP. By binding an LWP to a real-time thread you are ensuring the thread will be able to run with minimal delay once it is scheduled.

**4.6** Please refer to the supporting Web site for source code solution.


# CPU Scheduling

## Practice Exercises

**5.1** *n!* (*n* factorial = $n \times n-1 \times n-2 \times \ldots \times 2 \times 1$).

**5.2** Preemptive scheduling allows a process to be interrupted in the midst of its execution, taking the CPU away and allocating it to another process. Nonpreemptive scheduling ensures that a process relinquishes control of the CPU only when it finishes with its current CPU burst.

**5.3**
a. 10.53
b. 9.53
c. 6.86

Remember that turnaround time is finishing time minus arrival time, so you have to subtract the arrival times to compute the turnaround times. FCFS is 11 if you forget to subtract arrival time.

**5.4** Processes that need more frequent servicing, for instance, interactive processes such as editors, can be in a queue with a small time quantum. Processes with no need for frequent servicing can be in a queue with a larger quantum, requiring fewer context switches to complete the processing, and thus making more efficient use of the computer.

**5.5**
a. The shortest job has the highest priority.
b. The lowest level of MLFQ is FCFS.
c. FCFS gives the highest priority to the job having been in existence the longest.
d. None.

**5.6** It will favor the I/O-bound programs because of the relatively short CPU burst request by them; however, the CPU-bound programs will not starve because the I/O-bound programs will relinquish the CPU relatively often to do their I/O.

**5.7** PCS scheduling is done local to the process. It is how the thread library schedules threads onto available LWPs. SCS scheduling is the situation where the operating system schedules kernel threads. On systems using either many-to-one or many-to-many, the two scheduling models are fundamentally different. On systems using one-to-one, PCS and SCS are the same.

**5.8** Yes, otherwise a user thread may have to compete for an available LWP prior to being actually scheduled. By binding the user thread to an LWP, there is no latency while waiting for an available LWP; the real-time user thread can be scheduled immediately.


# Process Synchronization

## Practice Exercises

**6.1** The system clock is updated at every clock interrupt. If interrupts were disabled—particularly for a long period of time—it is possible the system clock could easily lose the correct time. The system clock is also used for scheduling purposes. For example, the time quantum for a process is expressed as a number of clock ticks. At every clock interrupt, the scheduler determines if the time quantum for the currently running process has expired. If clock interrupts were disabled, the scheduler could not accurately assign time quantums. This effect can be minimized by disabling clock interrupts for only very short periods.

**6.2** Please refer to the supporting Web site for source code solution.

**6.3** These operating systems provide different locking mechanisms depending on the application developers' needs. Spinlocks are useful for multiprocessor systems where a thread can run in a busy-loop (for a short period of time) rather than incurring the overhead of being put in a sleep queue. Mutexes are useful for locking resources. Solaris 2 uses adaptive mutexes, meaning that the mutex is implemented with a spin lock on multiprocessor machines. Semaphores and condition variables are more appropriate tools for synchronization when a resource must be held for a long period of time, since spinning is inefficient for a long duration.

**6.4** Volatile storage refers to main and cache memory and is very fast. However, volatile storage cannot survive system crashes or powering down the system. Nonvolatile storage survives system crashes and powered-down systems. Disks and tapes are examples of nonvolatile storage. Recently, USB devices using erasable program read-only memory (EPROM) have appeared providing nonvolatile storage. Stable storage refers to storage that technically can *never* be lost as there are redundant backup copies of the data (usually on disk).

**6.5** A checkpoint log record indicates that a log record and its modified data has been written to stable storage and that the transaction need not to be redone in case of a system crash. Obviously, the more often checkpoints

are performed, the less likely it is that redundant updates will have to be performed during the recovery process.

- System performance when no failure occurs—If no failures occur, the system must incur the cost of performing checkpoints that are essentially unnecessary. In this situation, performing checkpoints less often will lead to better system performance.
- The time it takes to recover from a system crash—The existence of a checkpoint record means that an operation will not have to be redone during system recovery. In this situation, the more often checkpoints were performed, the faster the recovery time is from a system crash.
- The time it takes to recover from a disk crash—The existence of a checkpoint record means that an operation will not have to be redone during system recovery. In this situation, the more often checkpoints were performed, the faster the recovery time is from a disk crash.

**6.6** A transaction is a series of read and write operations upon some data followed by a commit operation. If the series of operations in a transaction cannot be completed, the transaction must be aborted and the operations that did take place must be rolled back. It is important that the series of operations in a transaction appear as one indivisible operation to ensure the integrity of the data being updated. Otherwise, data could be compromised if operations from two (or more) different transactions were intermixed.

**6.7** A schedule that is allowed in the two-phase locking protocol but not in the timestamp protocol is:

| step | $T_0$ | $T_1$ | Precedence |
|---|---|---|---|
| 1 | **lock-S**($A$) | | |
| 2 | **read**($A$) | | |
| 3 | | **lock-X**($B$) | |
| 4 | | **write**($B$) | |
| 5 | | **unlock**($B$) | |
| 6 | **lock-S**($B$) | | |
| 7 | **read**($B$) | | $T_1 \rightarrow T_0$ |
| 8 | **unlock**($A$) | | |
| 9 | **unlock**($B$) | | |

This schedule is not allowed in the timestamp protocol because at step 7, the W-timestamp of $B$ is 1.

A schedule that is allowed in the timestamp protocol but not in the two-phase locking protocol is:

| step | $T_0$ | $T_1$ | $T_2$ |
|---|---|---|---|
| 1 | **write**($A$) | | |
| 2 | | **write**($A$) | |
| 3 | | | **write**($A$) |
| 4 | **write**($B$) | | |
| 5 | | **write**($B$) | |

This schedule cannot have lock instructions added to make it legal under two-phase locking protocol because $T_1$ must unlock ($A$) between steps 2 and 3, and must lock ($B$) between steps 4 and 5.


# Deadlocks

## Practice Exercises

**7.1**

- Two cars crossing a single-lane bridge from opposite directions.
- A person going down a ladder while another person is climbing up the ladder.
- Two trains traveling toward each other on the same track.
- Two carpenters who must pound nails. There is a single hammer and a single bucket of nails. Deadlock occurs if one carpenter has the hammer and the other carpenter has the nails.

**7.2** An unsafe state may not necessarily lead to deadlock, it just means that we cannot guarantee that deadlock will not occur. Thus, it is possible that a system in an unsafe state may still allow all processes to complete without deadlock occurring. Consider the situation where a system has 12 resources allocated among processes $P_0$, $P_1$, and $P_2$. The resources are allocated according to the following policy:

| | Max | Current | Need |
|---|---|---|---|
| $P_0$ | 10 | 5 | 5 |
| $P_1$ | 4 | 2 | 2 |
| $P_2$ | 9 | 3 | 6 |

Currently there are two resources available. This system is in an unsafe state as process $P_1$ could complete, thereby freeing a total of four resources. But we cannot guarantee that processes $P_0$ and $P_2$ can complete. However, it is possible that a process may release resources before requesting any further. For example, process $P_2$ could release a resource, thereby increasing the total number of resources to five. This allows process $P_0$ to complete, which would free a total of nine resources, thereby allowing process $P_2$ to complete as well.

**7.3** **Answer:** This is probably not a good solution because it yields too large a scope. It is better to define a locking policy with as narrow a scope as possible.

```
for (int i = 0; i < n; i++) {
   // first find a thread that can finish
   for (int j = 0; j < n; j++) {
      if (!finish[j]) {
         boolean temp = true;
         for (int k = 0; k < m; k++) {
            if (need[j][k] > work[k])
               temp = false;
         }

         if (temp) { // if this thread can finish
            finish[j] = true;
            for (int x = 0; x < m; x++)
               work[x] += work[j][x];
         }
      }
   }
}
```

**Figure 7.1** Banker's algorithm safety algorithm.

**7.4** Figure 7.1 provides Java code that implement the safety algorithm of the banker's algorithm (the complete implementation of the banker's algorithm is available with the source code download).
As can be seen, the nested outer loops—both of which loop through $n$ times—provide the $n^2$ performance. Within these outer loops are two sequential inner loops which loop $m$ times. The big-oh of this algorithm is therefore $O(m \times n^2)$.

**7.5** An argument for installing deadlock avoidance in the system is that we could ensure deadlock would never occur. In addition, despite the increase in turnaround time, all 5,000 jobs could still run.
An argument against installing deadlock avoidance software is that deadlocks occur infrequently and they cost little when they do occur.

**7.6** Starvation is a difficult topic to define as it may mean different things for different systems. For the purposes of this question, we will define starvation as the situation whereby a process must wait beyond a reasonable period of time—perhaps indefinitely—before receiving a requested resource. One way of detecting starvation would be to first identify a period of time—$T$—that is considered unreasonable. When a process requests a resource, a timer is started. If the elapsed time exceeds $T$, then the process is considered to be starved.
One strategy for dealing with starvation would be to adopt a policy where resources are assigned only to the process that has been waiting the longest. For example, if process $P_a$ has been waiting longer for resource $X$ than process $P_b$, the request from process $P_b$ would be deferred until process $P_a$'s request has been satisfied.
Another strategy would be less strict than what was just mentioned. In this scenario, a resource might be granted to a process that has waited less than another process, providing that the other process is not starving.

However, if another process is considered to be starving, its request would be satisfied first.

**7.7** a. Deadlock cannot occur because preemption exists.

b. Yes. A process may never acquire all the resources it needs if they are continuously preempted by a series of requests such as those of process *C*.

**7.8** Yes. The *Max* vector represents the maximum request a process may make. When calculating the safety algorithm we use the *Need* matrix, which represents *Max* — *Allocation*. Another way to think of this is *Max* = *Need* + *Allocation*. According to the question, the *Waiting* matrix fulfills a role similar to the *Need* matrix, therefore *Max* = *Waiting* + *Allocation*.

**7.9** No. This follows directly from the hold-and-wait condition.


# Main Memory

## Practice Exercises

**8.1** A logical address does not refer to an actual existing address; rather, it refers to an abstract address in an abstract address space. Contrast this with a physical address that refers to an actual physical address in memory. A logical address is generated by the CPU and is translated into a physical address by the memory management unit(MMU). Therefore, physical addresses are generated by the MMU.

**8.2** The major advantage of this scheme is that it is an effective mechanism for code and data sharing. For example, only one copy of an editor or a compiler needs to be kept in memory, and this code can be shared by all processes needing access to the editor or compiler code. Another advantage is protection of code against erroneous modification. The only disadvantage is that the code and data must be separated, which is usually adhered to in a compiler-generated code.

**8.3** Recall that paging is implemented by breaking up an address into a page and offset number. It is most efficient to break the address into X page bits and Y offset bits, rather than perform arithmetic on the address to calculate the page number and offset. Because each bit position represents a power of 2, splitting an address between bits results in a page size that is a power of 2.

**8.4** a. Logical address: 16 bits

b. Physical address: 15 bits

**8.5** By allowing two entries in a page table to point to the same page frame in memory, users can share code and data. If the code is reentrant, much memory space can be saved through the shared use of large programs such as text editors, compilers, and database systems. “Copying” large amounts of memory could be effected by having different page tables point to the same memory location.

However, sharing of nonreentrant code or data means that any user having access to the code can modify it and these modifications would be reflected in the other user's "copy."

**8.6** Since segment tables are a collection of base–limit registers, segments can be shared when entries in the segment table of two different jobs point to the same physical location. The two segment tables must have identical base pointers, and the shared segment number must be the same in the two processes.

**8.7** Both of these problems reduce to a program being able to reference both its own code and its data without knowing the segment or page number associated with the address. MULTICS solved this problem by associating four registers with each process. One register had the address of the current program segment, another had a base address for the stack, another had a base address for the global data, and so on. The idea is that all references have to be indirect through a register that maps to the current segment or page number. By changing these registers, the same code can execute for different processes without the same page or segment numbers.

**8.8**

a. Protection not necessary, set system key to 0.

b. Set system key to 0 when in supervisor mode.

c. Region sizes must be fixed in increments of 2k bytes, allocate key with memory blocks.

d. Same as above.

e. Frame sizes must be in increments of 2k bytes, allocate key with pages.

f. Segment sizes must be in increments of 2k bytes, allocate key with segments.

# Virtual Memory


## Practice Exercises

**9.1** A page fault occurs when an access to a page that has not been brought into main memory takes place. The operating system verifies the memory access, aborting the program if it is invalid. If it is valid, a free frame is located and I/O is requested to read the needed page into the free frame. Upon completion of I/O, the process table and page table are updated and the instruction is restarted.

**9.2** a. *n*

b. *p*

**9.3** a. Stack—good.

b. Hashed symbol table—not good.

c. Sequential search—good.

d. Binary search—not good.

e. Pure code—good.

f. Vector operations—good.

g. Indirection—not good.

**9.4**

| Rank | Algorithm | Suffer from Belady's anomaly |
|---|---|---|
| 1 | Optimal | no |
| 2 | LRU | no |
| 3 | Second-chance | yes |
| 4 | FIFO | yes |

**9.5** The costs are additional hardware and slower access time. The benefits are good utilization of memory and larger logical address space than physical address space.

**9.6**

| | | |
|---|---|---|
| effective access time | = | 0.99 × (1 μsec + 0.008 × (2 μsec) |
| | | + 0.002 × (10,000 μsec + 1,000 μsec) |
| | | + 0.001 × (10,000 μsec + 1,000 μsec) |
| | = | (0.99 + 0.016 + 22.0 + 11.0) μsec |
| | = | 34.0 μsec |

**9.7** a. 50

b. 5,000

**9.8**

| Number of frames | LRU | FIFO | Optimal |
|---|---|---|---|
| 1 | 20 | 20 | 20 |
| 2 | 18 | 18 | 15 |
| 3 | 15 | 16 | 11 |
| 4 | 10 | 14 | 8 |
| 5 | 8 | 10 | 7 |
| 6 | 7 | 10 | 7 |
| 7 | 7 | 7 | 7 |

**9.9** You can use the valid/invalid bit supported in hardware to simulate the reference bit. Initially set the bit to invalid. On first reference a trap to the operating system is generated. The operating system will set a software bit to 1 and reset the valid/invalid bit to valid.

**9.10** No. An optimal algorithm will not suffer from Belady's anomaly because —by definition—an optimal algorithm replaces the page that will not be used for the longest time. Belady's anomaly occurs when a page-replacement algorithm evicts a page that will be needed in the immediate future. An optimal algorithm would not have selected such a page.

**9.11** a. **FIFO**. Find the first segment large enough to accommodate the incoming segment. If relocation is not possible and no one segment is large enough, select a combination of segments whose memories are contiguous, which are "closest to the first of the list" and which can accommodate the new segment. If relocation is possible, rearrange the memory so that the first *N* segments large enough for the incoming segment are contiguous in memory. Add any leftover space to the free-space list in both cases.

b. **LRU**. Select the segment that has not been used for the longest period of time and that is large enough, adding any leftover space to the free space list. If no one segment is large enough, select a combination of the "oldest" segments that are contiguous in memory (if relocation is not available) and that are large enough. If relocation is available, rearrange the oldest *N* segments to be contiguous in memory and replace those with the new segment.

**9.12** a. Thrashing is occurring.

b. CPU utilization is sufficiently high to leave things alone, and increase degree of multiprogramming.

c. Increase the degree of multiprogramming.

**9.13** The page table can be set up to simulate base and limit registers provided that the memory is allocated in fixed-size segments. In this way, the base of a segment can be entered into the page table and the valid/invalid bit used to indicate that portion of the segment as resident in the memory. There will be some problem with internal fragmentation.

# File-System Interface


## Practice Exercises

**10.1** Deleting all files not specifically saved by the user has the advantage of minimizing the file space needed for each user by not saving unwanted or unnecessary files. Saving all files unless specifically deleted is more secure for the user in that it is not possible to lose files inadvertently by forgetting to save them.

**10.2** Some systems allow different file operations based on the type of the file (for instance, an ascii file can be read as a stream while a database file can be read via an index to a block). Other systems leave such interpretation of a file's data to the process and provide no help in accessing the data. The method that is "better" depends on the needs of the processes on the system, and the demands the users place on the operating system. If a system runs mostly database applications, it may be more efficient for the operating system to implement a database-type file and provide operations, rather than making each program implement the same thing (possibly in different ways). For general-purpose systems it may be better to only implement basic file types to keep the operating system size smaller and allow maximum freedom to the processes on the system.

**10.3** An advantage of having the system support different file structures is that the support comes from the system; individual applications are not required to provide the support. In addition, if the system provides the support for different file structures, it can implement the support presumably more efficiently than an application.
The disadvantage of having the system provide support for defined file types is that it increases the size of the system. In addition, applications that may require different file types other than what is provided by the system may not be able to run on such systems.
An alternative strategy is for the operating system to define no support for file structures and instead treat all files as a series of bytes. This is the approach taken by UNIX systems. The advantage of this approach is that it simplifies the operating system support for file systems, as the

system no longer has to provide the structure for different file types. Furthermore, it allows applications to define file structures, thereby alleviating the situation where a system may not provide a file definition required for a specific application.

**10.4** If arbitrarily long names can be used then it is possible to simulate a multilevel directory structure. This can be done, for example, by using the character "." to indicate the end of a subdirectory. Thus, for example, the name *jim.java.F1* specifies that *F1* is a file in subdirectory *java* which in turn is in the root directory *jim*.
If file names were limited to seven characters, then the above scheme could not be utilized and thus, in general, the answer is *no*. The next best approach in this situation would be to use a specific file as a symbol table (directory) to map arbitrarily long names (such as *jim.java.F1*) into shorter arbitrary names (such as *XX00743*), which are then used for actual file access.

**10.5** The purpose of the `open()` and `close()` operations is:

- The `open()` operation informs the system that the named file is about to become active.
- The `close()` operation informs the system that the named file is no longer in active use by the user who issued the close operation.

**10.6** Two possible applications are:

a. Print the content of the file.

b. Print the content of record *i*. This record can be found using hashing or index techniques.

**10.7** a. One piece of information kept in a directory entry is file location. If a user could modify this location, then he could access other files defeating the access-protection scheme.

b. Do not allow the user to directly write onto the subdirectory. Rather, provide system operations to do so.

**10.8** a. There are two methods for achieving this:

i. Create an access control list with the names of all 4990 users.

ii. Put these 4990 users in one group and set the group access accordingly. This scheme cannot always be implemented since user groups are restricted by the system.

b. The universal access to files applies to all users unless their name appears in the access-control list with different access permission. With this scheme you simply put the names of the remaining ten users in the access control list but with no access privileges allowed.

**10.9**

- *File control list*. Since the access control information is concentrated in one single place, it is easier to change access control information and this requires less space.
- *User control list*. This requires less overhead when opening a file.

# File-System Implementation


## Practice Exercises

**11.1** The results are:

| | Contiguous | Linked | Indexed |
|---|---|---|---|
| a. | 201 | 1 | 1 |
| b. | 101 | 52 | 1 |
| c. | 1 | 3 | 1 |
| d. | 198 | 1 | 0 |
| e. | 98 | 52 | 0 |
| f. | 0 | 100 | 0 |

**11.2** There would be multiple paths to the same file, which could confuse users or encourage mistakes (deleting a file with one path deletes the file in all the other paths).

**11.3** In case of system crash (memory failure) the free-space list would not be lost as it would be if the bit map had been stored in main memory.

**11.4**
- **Contiguous**—if file is usually accessed sequentially, if file is relatively small.
- **Linked**—if file is large and usually accessed sequentially.
- **Indexed**—if file is large and usually accessed randomly.

**11.5** This method requires more overhead then the standard contiguous allocation. It requires less overhead than the standard linked allocation.

**11.6** Caches allow components of differing speeds to communicate more efficiently by storing data from the slower device, temporarily, in a faster device (the cache). Caches are, almost by definition, more expensive than the device they are caching for, so increasing the number or size of caches would increase system cost.

**11.7** Dynamic tables allow more flexibility in system use growth — tables are never exceeded, avoiding artificial use limits. Unfortunately, kernel

structures and code are more complicated, so there is more potential for bugs. The use of one resource can take away more system resources (by growing to accommodate the requests) than with static tables.

**11.8** VFS introduces a layer of indirection in the file system implementation. In many ways, it is similar to object-oriented programming techniques. System calls can be made generically (independent of file system type). Each file system type provides its function calls and data structures to the VFS layer. A system call is translated into the proper specific functions for the target file system at the VFS layer. The calling program has no file-system-specific code, and the upper levels of the system call structures likewise are file system-independent. The translation at the VFS layer turns these generic calls into file-system-specific operations.

# Mass Storage Structure


## Practice Exercises

**12.1** a. $t = 0.95 + 0.05L$

b. FCFS 362.60; SSTF 95.80; SCAN 497.95; LOOK 174.50; C-SCAN 500.15; (and C-LOOK 176.70). SSTF is still the winner, and LOOK is the runner-up.

c. $(362.60 - 95.80)/362.60 = 0.74$ The percentage speedup of SSTF over FCFS is 74%, with respect to the seek time. If we include the overhead of rotational latency and data transfer, the percentage speedup will be less.

**12.2** In a single-user environment, the I/O queue usually is empty. Requests generally arrive from a single process for one block or for a sequence of consecutive blocks. In these cases, FCFS is an economical method of disk scheduling. But LOOK is nearly as easy to program and will give much better performance when multiple processes are performing concurrent I/O, such as when a Web browser retrieves data in the background while the operating system is paging and another application is active in the foreground.

**12.3** The center of the disk is the location having the smallest average distance to all other tracks. Thus the disk head tends to move away from the edges of the disk. Here is another way to think of it. The current location of the head divides the cylinders into two groups. If the head is not in the center of the disk and a new request arrives, the new request is more likely to be in the group that includes the center of the disk; thus, the head is more likely to move in that direction.

**12.4** Most disks do not export their rotational position information to the host. Even if they did, the time for this information to reach the scheduler would be subject to imprecision and the time consumed by the scheduler is variable, so the rotational position information would become incorrect. Further, the disk requests are usually given in terms

of logical block numbers, and the mapping between logical blocks and physical locations is very complex.

**12.5** How would use of a RAM disk affect your selection of a disk-scheduling algorithm? What factors would you need to consider? Do the same considerations apply to hard-disk scheduling, given that the file system stores recently used blocks in a buffer cache in main memory?
Disk scheduling attempts to reduce the overhead time of disk head positioning. Since a RAM disk has uniform access times, scheduling is largely unnecessary. The comparison between RAM disk and the main memory disk-cache has no implications for hard-disk scheduling because we schedule only the buffer cache misses, not the requests that find their data in main memory.

**12.6** A system can perform only at the speed of its slowest bottleneck. Disks or disk controllers are frequently the bottleneck in modern systems as their individual performance cannot keep up with that of the CPU and system bus. By balancing I/O among disks and controllers, neither an individual disk nor a controller is overwhelmed, so that bottleneck is avoided.

**12.7** If code pages are stored in swap space, they can be transferred more quickly to main memory (because swap space allocation is tuned for faster performance than general file system allocation). Using swap space can require startup time if the pages are copied there at process invocation rather than just being paged out to swap space on demand. Also, more swap space must be allocated if it is used for both code and data pages.

**12.8** Truly stable storage would never lose data. The fundamental technique for stable storage is to maintain multiple copies of the data, so that if one copy is destroyed, some other copy is still available for use. But for any scheme, we can imagine a large enough disaster that all copies are destroyed.

**12.9**

a. The disk spins 120 times per second, and each spin transfers a track of 80 KB. Thus, the sustained transfer rate can be approximated as 9600 KB/s.

b. Suppose that 100 cylinders is a huge transfer. The transfer rate is total bytes divided by total time. Bytes: 100 cyl * 20 trk/cyl * 80 KB/trk, i.e., 160,000 KB. Time: rotation time + track switch time + cylinder switch time. Rotation time is 2000 trks/120 trks_per_sec, i.e., 16.667 s. Track switch time is 19 switch_per_cyl * 100 cyl * 0.5 ms, i.e., 950 ms. Cylinder switch time is 99 * 2 ms, i.e., 198 ms. Thus, the total time is 16.667 + 0.950 + 0.198, i.e., 17.815 s. (We are ignoring any initial seek and rotational latency, which might add about 12 ms to the schedule, i.e. 0.1%.) Thus the transfer rate is 8981.2 KB/s. The overhead of track and cylinder switching is about 6.5%.

c. The time per transfer is 8 ms to seek + 4.167 ms average rotational latency + 0.052 ms (calculated from 1/(120 trk_per_second * 160

sector_per_trk)) to rotate one sector past the disk head during reading. We calculate the transfers per second as 1/(0.012219), i.e., 81.8. Since each transfer is 0.5 KB, the transfer rate is 40.9 KB/s.

d. We ignore track and cylinder crossings for simplicity. For reads of size 4 KB, 8 KB, and 64 KB, the corresponding I/Os per second are calculated from the seek, rotational latency, and rotational transfer time as in the previous item, giving (respectively) 1/(0.0126), 1/(0.013), and 1/(0.019). Thus we get 79.4, 76.9, and 52.6 transfers per second, respectively. Transfer rates are obtained from 4, 8, and 64 times these I/O rates, giving 318 KB/s, 615 KB/s, and 3366 KB/s, respectively.

e. From 1/(3+4.167+0.83) we obtain 125 I/Os per second. From 8 KB per I/O we obtain 1000 KB/s.

**12.10** For 8 KB random I/Os on a lightly loaded disk, where the random access time is calculated to be about 13 ms (see Exercise 12.9), the effective transfer rate is about 615 MB/s. In this case, 15 disks would have an aggregate transfer rate of less than 10 MB/s, which should not saturate the bus. For 64 KB random reads to a lightly loaded disk, the transfer rate is about 3.4 MB/s, so five or fewer disk drives would saturate the bus. For 8 KB reads with a large enough queue to reduce the average seek to 3 ms, the transfer rate is about 1 MB/s, so the bus bandwidth may be adequate to accommodate 15 disks.

**12.11** Since the disk holds 22,400,000 sectors, the probability of requesting one of the 100 remapped sectors is very small. An example of a worst-case event is that we attempt to read, say, an 8 KB page, but one sector from the middle is defective and has been remapped to the worst possible location on another track in that cylinder. In this case, the time for the retrieval could be 8 ms to seek, plus two track switches and two full rotational latencies. It is likely that a modern controller would read all the requested good sectors from the original track before switching to the spare track to retrieve the remapped sector and thus would incur only one track switch and rotational latency. So the time would be 8 ms seek + 4.17 ms average rotational latency + 0.05 ms track switch + 8.3 ms rotational latency + 0.83 ms read time (8 KBis 16 sectors, 1/10 of a track rotation). Thus, the time to service this request would be 21.8 ms, giving an I/O rate of 45.9 requests per second and an effective bandwidth of 367 KB/s. For a severely time-constrained application this might matter, but the overall impact in the weighted average of 100 remapped sectors and 22.4 million good sectors is nil.

**12.12** Two bad outcomes could result. One possibility is starvation of the applications that issue blocking I/Os to tapes that are not mounted in drives. Another possibility is thrashing, as the jukebox is commanded to switch tapes after every I/O operation.

**12.13** Tapes are easily removable, so they are useful for off-site backups and for bulk transfer of data (by sending cartridges). As a rule, a magnetic hard disk is not a removable medium.

**12.14** a. For 512 bytes, the effective transfer rate is calculated as follows.
ETR = transfer size/transfer time.
If X is transfer size, then transfer time is ((X/STR) + latency).
Transfer time is 15ms + (512B/5MB per second) = 15.0097ms.
Effective transfer rate is therefore 512B/15.0097ms = 33.12 KB/sec.
ETR for 8KB = .47MB/sec.
ETR for 1MB = 4.65MB/sec.
ETR for 16MB = 4.98MB/sec.

b. Utilization of the device for 512B = 33.12 KB/sec / 5MB/sec = .0064 = .64
For 8KB = 9.4%.
For 1MB = 93%.
For 16MB = 99.6%.

c. Calculate .25 = ETR/STR, solving for transfer size X.
STR = 5MB, so 1.25MB/S = ETR.
1.25MB/S * ((X/5) + .015) = X.
.25X + .01875 = X.
X = .025MB.

d. A disk is a random-access device for transfers larger than K bytes (where K > disk block size), and is a sequential-access device for smaller transfers.

e. Calculate minimum transfer size for acceptable utilization of cache memory:
STR = 800MB, ETR = 200, latency = $8 * 10^{-9}$.
200 (XMB/800 + 8 X $10^{-9}$) = XMB.
.25XMB + 1600 * $10^{-9}$ = XMB.
X = 2.24 bytes.
Calculate for memory:
STR = 80MB, ETR = 20, L = $60 * 10^{-9}$.
20 (XMB/80 + $60 * 10^{-9}$) = XMB.
.25XMB + $1200 * 10^{-9}$ = XMB.
X = 1.68 bytes.
Calculate for tape:
STR = 2MB, ETR = .5, L = 60s.
.5 (XMB/2 + 60) = XMB.
.25XMB + 30 = XMB.
X = 40MB.

f. It depends upon how it is being used. Assume we are using the tape to restore a backup. In this instance, the tape acts as a sequential-access device where we are sequentially reading the contents of the tape. As another example, assume we are using the tape to access a variety of records stored on the tape. In this instance, access to the tape is arbitrary and hence considered random.

**12.15** a. Assume that a disk drive holds 10 GB. Then 100 disks hold 1 TB, 100,000 disks hold 1 PB, and 100,000,000 disks hold 1 EB. To store 4 EBwould require about 400 million disks. If a magnetic tape holds

40 GB, only 100 million tapes would be required. If an optical tape holds 50 times more data than a magnetic tape, 2 million optical tapes would suffice. If a holographic cartridge can store 180 GB, about 22.2 million cartridges would be required.

b. A 3.5" disk drive is about 1" high, 4" wide, and 6" deep. In feet, this is 1/12 by 1/3 by 1/2, or 1/72 cubic feet. Packed densely, the 400 million disks would occupy 5.6 million cubic feet. If we allow a factor of two for air space and space for power supplies, the required capacity is about 11 million cubic feet.

c. A 1/2" tape cartridge is about 1" high and 4.5" square. The volume is about 1/85 cubic feet. For 100 million magnetic tapes packed densely, the volume is about 1.2 million cubic feet. For 2 million optical tapes, the volume is 23,400 cubic feet.

d. A CD-ROM is 4.75" in diameter and about 1/16" thick. If we assume that a holostore disk is stored in a library slot that is 5" square and 1/8" wide, we calculate the volume of 22.2 million disks to be about 40,000 cubic feet.


# I/O Systems

## Practice Exercises

**13.1** Three advantages: Bugs are less likely to cause an operating system crash
Performance can be improved by utilizing dedicated hardware and hard-coded algorithms
The kernel is simplified by moving algorithms out of it
Three disadvantages: Bugs are harder to fix—a new firmware version or new hardware is needed
Improving algorithms likewise require a hardware update rather than just a kernel or device-driver update
Embedded algorithms could conflict with application's use of the device, causing decreased performance.

**13.2** It is possible, using the following algorithm. Let's assume we simply use the busy-bit (or the command-ready bit; this answer is the same regardless). When the bit is off, the controller is idle. The host writes to data-out and sets the bit to signal that an operation is ready (the equivalent of setting the command-ready bit). When the controller is finished, it clears the busy bit. The host then initiates the next operation. This solution requires that both the host and the controller have read and write access to the same bit, which can complicate circuitry and increase the cost of the controller.

**13.3** Polling can be more efficient than interrupt-driven I/O. This is the case when the I/O is frequent and of short duration. Even though a single serial port will perform I/O relatively infrequently and should thus use interrupts, a collection of serial ports such as those in a terminal concentrator can produce a lot of short I/O operations, and interrupting for each one could create a heavy load on the system. A well-timed polling loop could alleviate that load without wasting many resources through looping with no I/O needed.

**13.4** A hybrid approach could switch between polling and interrupts depending on the length of the I/O operation wait. For example, we

could poll and loop N times, and if the device is still busy at N+1, we could set an interrupt and sleep. This approach would avoid long busy-waiting cycles. This method would be best for very long or very short busy times. It would be inefficient it the I/O completes at N+T (where T is a small number of cycles) due to the overhead of polling plus setting up and catching interrupts.
Pure polling is best with very short wait times. Interrupts are best with known long wait times.

**13.5** DMA increases system concurrency by allowing the CPU to perform tasks while the DMA system transfers data via the system and memory buses. Hardware design is complicated because the DMA controller must be integrated into the system, and the system must allow the DMA controller to be a bus master. Cycle stealing may also be necessary to allow the CPU and DMA controller to share use of the memory bus.

**13.6** Consider a system which performs 50% I/O and 50% computes. Doubling the CPU performance on this system would increase total system performance by only 50%. Doubling both system aspects would increase performance by 100%. Generally, it is important to remove the current system bottleneck, and to increase overall system performance, rather than blindly increasing the performance of individual system components.

**13.7** The STREAMS driver controls a physical device that could be involved in a STREAMS operation. The STREAMS module modifies the flow of data between the head (the user interface) and the driver.


# Protection

## Practice Exercises

**14.1** An access list is a list for each object consisting of the domains with a nonempty set of access rights for that object. A capability list is a list of objects and the operations allowed on those objects for each domain.

**14.2** This would be useful as an extra security measure so that the old content of memory cannot be accessed, either intentionally or by accident, by another program. This is especially useful for any highly classified information.

**14.3** $D_j$ is a subset of $D_i$.

**14.4** $A(x,y)$ is a subset of $A(z,y)$.

**14.5** The contents of the stack could be compromised by other process(es) sharing the stack.

**14.6** Hierarchical structure.

**14.7** Add an integer counter with the capability.

**14.8** Reference counts.

**14.9** In earlier chapters we identified a distinction between kernel and user mode where kernel mode is used for carrying out privileged operations such as I/O. One reason why I/O must be performed in kernel mode is that I/O requires accessing the hardware and proper access to the hardware is necessary for system integrity. If we allow users to perform their own I/O, we cannot guarantee system integrity.

**14.10** A capability list is considered a "protected object" and is accessed only indirectly by the user. The operating system ensures the user cannot access the capability list directly.


# Security

No Practice Exercises

# Distributed System Structures


## Practice Exercises

**16.1** Cost. A fully connected network requires a link between every node in the network. For a WAN, this may be too costly as communication links between physically distant hosts may be expensive.

**16.2** A token ring is very effective under high sustained load, as no collisions can occur and each slot may be used to carry a message, providing high throughput. A token ring is less effective when the load is light (token processing takes longer than bus access, so any one packet can take longer to reach its destination) or sporadic.

**16.3** All broadcasts would be propagated to all networks, causing a *lot* of network traffic. If broadcast traffic were limited to important data (and very little of it), then broadcast propagation would save gateways from having to run special software to watch for this data (such as network routing information) and rebroadcast it.

**16.4** There is a performance advantage to caching name translations for computers located in remote domains: repeated resolution of the same name from different computers located in the local domain could be performed locally without requiring a remote name lookup operation. The disadvantage is that there could be inconsistencies in the name translations when updates are made in the mapping of names to IP addresses. These consistency problems could be solved by invalidating translations, which would require state to be managed regarding which computers are caching a certain translation and also would require a number of invalidation messages, or by using leases whereby the caching entity invalidates a translation after a certain period of time. The latter approach requires less state and no invalidation messages but might suffer from temporary inconsistencies.

**16.5** Circuit switching guarantees that the network resources required for a transfer are reserved before the transmission takes place. This ensures that packets will not be dropped and their delivery would satisfy quality of service requirements. The disadvantage of circuit switching is that

it requires a round-trip message to set-up the reservations and it also might overprovision resources, thereby resulting in suboptimal use of the resources. Circuit switching is a viable strategy for applications that have constant demands regarding network resources and would require the resources for long periods of time, thereby amortizing the initial overheads.

**16.6** One such issue is making all the processors and storage devices seem transparent across the network. In other words, the distributed system should appear as a centralized system to users. The Andrew file system and NFS provide this feature: the distributed file system appears to the user as a single file system but in reality it may be distributed across a network.
Another issue concerns the mobility of users. We want to allow users to connect to the "system" rather than to a specific machine (although in reality they may be logging in to a specific machine somewhere in the distributed system).

**16.7** For the same operating system, process migration is relatively straightforward, as the state of the process needs to migrate from one processor to another. This involves moving the address space, state of the CPU registers, and open files from the source system to the destination. However, it is important that identical copies of the operating system are running on the different systems to ensure compatibility. If the operating system are the same, but perhaps different versions are running on the separate systems, then migrating processes must be sure to follow programming guidelines that are consistent between the different versions of the operating system.
Java applets provide a nice example of process migration between different operating systems. To hide differences in the underlying system, the migrated process (i.e., a Java applet) runs on a virtual machine rather than a specific operating system. All that is required is for the virtual machine to be running on the system the process migrates to.

**16.8** Three common failures in a distributed system include: (1) network link failure, (2) host failure, (3) storage medium failure. Both (2) and (3) are failures that could also occur in a centralized system, whereas a network link failure can occur only in a networked-distributed system.

**16.9** No. Many status-gathering programs work from the assumption that packets may not be received by the destination system. These programs generally *broadcast* a packet and assume that at least some other systems on their network will receive the information. For instance, a daemon on each system might broadcast the system's load average and number of users. This information might be used for process migration target selection. Another example is a program that determines if a remote site is both running and accessible over the network. If it sends a query and gets no reply it knows the system cannot currently be reached.

**16.10** Typically distributed systems employ a coordinator process that performs functions needed by other processes in the system. This would

include enforcing mutual exclusion and—in this case of a ring—replacing a lost token.

A scheme similar to the ring algorithm presented in Section 18.6.2 can be used. The algorithm is as follows:

The **ring algorithm** assumes that the links are unidirectional and that processes send their messages to the neighbor on their right. The main data structure used by the algorithm is the **active list**, a list that contains the priority numbers of all active processes in the system when the algorithm ends; each process maintains its own active list.

a. If process $P_i$ detects a coordinator failure, it creates a new active list that is initially empty. It then sends a message *elect*($i$) to its right neighbor, and adds the number $i$ to its active list.

b. If $P_i$ receives a message *elect*($j$) from the process on the left, it must respond in one of three ways:

   i. If this is the first *elect* message it has seen or sent, $P_i$ creates a new active list with the numbers $i$ and $j$. It then sends the message *elect*($i$), followed by the message *elect*($j$).

   ii. If $i \neq j$, that is, the message received does not contain $P_i$'s number, then $P_i$ adds $j$ to its active list and forwards the message to its right neighbor.

   iii. If $i = j$, that is, $P_i$ receives the message *elect*($i$), then the active list for $P_i$ now contains the numbers of all the active processes in the system. Process $P_i$ can now determine the largest number in the active list to identify the new coordinator process.

**16.11** One technique would be for B to periodically send a *I-am-up* message to A indicating it is still alive. If A does not receive an *I-am-up* message, it can assume either B—or the network link—is down. Note that an *I-am-up* message does not allow A to distinguish between each type of failure. One technique that allows A better to determine if the network is down is to send an *Are-you-up* message to B using an alternate route. If it receives a reply, it can determine that indeed the network link is down and that B is up.

If we assume that A knows B is up and is reachable (via the *I-am-up* mechanism) and that A has some value $N$ that indicates a normal response time, A could monitor the response time from B and compare values to $N$, allowing A to determine if B is overloaded or not.

The implications of both of these techniques are that A could choose another host—say C—in the system if B is either down, unreachable, or overloaded.

# Distributed File Systems


No Practice Exercises

# Distributed Coordination


No Practice Exercises

# Real-time Systems


No Practice Exercises

# Multimedia Systems


No Practice Exercises

# The Linux System


## Practice Exercises

**21.1** There are two principal drawbacks with the use of modules. The first is size: module management consumes unpageable kernel memory, and a basic kernel with a number of modules loaded will consume more memory than an equivalent kernel with the drivers compiled into the kernel image itself. This can be a very significant issue on machines with limited physical memory.

The second drawback is that modules can increase the complexity of the kernel bootstrap process. It is hard to load up a set of modules from disk if the driver needed to access that disk itself a module that needs to be loaded. As a result, managing the kernel bootstrap with modules can require extra work on the part of the administrator: the modules required to bootstrap need to be placed into a ramdisk image that is loaded alongside the initial kernel image when the system is initialized.

In certain cases it is better to use a modular kernel, and in other cases it is better to use a kernel with its device drivers prelinked. Where minimizing the size of the kernel is important, the choice will depend on how often the various device drivers are used. If they are in constant use, then modules are unsuitable. This is especially true where drivers are needed for the boot process itself. On the other hand, if some drivers are not always needed, then the module mechanism allows those drivers to be loaded and unloaded on demand, potentially offering a net saving in physical memory.

Where a kernel is to be built that must be usable on a large variety of very different machines, then building it with modules is clearly preferable to using a single kernel with dozens of unnecessary drivers consuming memory. This is particularly the case for commercially distributed kernels, where supporting the widest variety of hardware in the simplest manner possible is a priority.

However, if a kernel is being built for a single machine whose configuration is known in advance, then compiling and using modules

may simply be an unnecessary complexity. In cases like this, the use of modules may well be a matter of taste.

**21.2** Thread implementations can be broadly classified into two groups: kernel-based threads and user-mode threads. User-mode thread packages rely on some kernel support—they may require timer interrupt facilities, for example—but the scheduling between threads is not performed by the kernel but by some library of user-mode code. Multiple threads in such an implementation appear to the operating system as a single execution context. When the multithreaded process is running, it decides for itself which of its threads to execute, using non-local jumps to switch between threads according to its own preemptive or non-preemptive scheduling rules.

Alternatively, the operating system kernel may provide support for threads itself. In this case, the threads may be implemented as separate processes that happen to share a complete or partial common address space, or they may be implemented as separate execution contexts within a single process. Whichever way the threads are organized, they appear as fully independent execution contexts to the application.

Hybrid implementations are also possible, where a large number of threads are made available to the application using a smaller number of kernel threads. Runnable user threads are run by the first available kernel thread.

In Linux, threads are implemented within the kernel by a clone mechanism that creates a new process within the same virtual address space as the parent process. Unlike some kernel-based thread packages, the Linux kernel does not make any distinction between threads and processes: a thread is simply a process that did not create a new virtual address space when it was initialized.

The main advantage of implementing threads in the kernel rather than in a user-mode library are that:

- kernel-threaded systems can take advantage of multiple processors if they are available; and
- if one thread blocks in a kernel service routine (for example, a system call or page fault), other threads are still able to run.

A lesser advantage is the ability to assign different security attributes to each thread.

User-mode implementations do not have these advantages. Because such implementations run entirely within a single kernel execution context, only one thread can ever be running at once, even if multiple CPUs are available. For the same reason, if one thread enters a system call, no other threads can run until that system call completes. As a result, one thread doing a blocking disk read will hold up every thread in the application. However, user-mode implementations do have their own advantages. The most obvious is performance: invoking the kernel's own scheduler to switch between threads involves entering a new protection domain as the CPU switches to kernel mode, whereas switching between threads in user mode can be achieved simply by saving and restoring the main CPU registers. User-mode threads may also con-

sume less system memory: most UNIX systems will reserve at least a full page for a kernel stack for each kernel thread, and this stack may not be pageable.

The hybrid approach, implementing multiple user threads over a smaller number of kernel threads, allows a balance between these tradeoffs to be achieved. The kernel threads will allow multiple threads to be in blocking kernel calls at once and will permit running on multiple CPUs, and user-mode thread switching can occur within each kernel thread to perform lightweight threading without the overheads of having too many kernel threads. The downside of this approach is complexity: giving control over the tradeoff complicates the thread library's user interface.

**21.3** The primary impact of disallowing paging of kernel memory in Linux is that the non-preemptability of the kernel is preserved. Any process taking a page fault, whether in kernel or in user mode, risks being rescheduled while the required data is paged in from disk. Because the kernel can rely on not being rescheduled during access to its primary data structures, locking requirements to protect the integrity of those data structures are very greatly simplified. Although design simplicity is a benefit in itself, it also provides an important performance advantage on uniprocessor machines due to the fact that it is not necessary to do additional locking on most internal data structures.

There are a number of disadvantages to the lack of pageable kernel memory, however. First of all, it imposes constraints on the amount of memory that the kernel can use. It is unreasonable to keep very large data structures in non-pageable memory, since that represents physical memory that absolutely cannot be used for anything else. This has two impacts: first of all, the kernel must prune back many of its internal data structures manually, instead of being able to rely on a single virtual-memory mechanism to keep physical memory usage under control. Second, it makes it infeasible to implement certain features that require large amounts of virtual memory in the kernel, such as the /tmp-filesystem (a fast virtual-memory-based file system found on some UNIX systems).

Note that the complexity of managing page faults while running kernel code is not an issue here. The Linux kernel code is already able to deal with page faults: it needs to be able to deal with system calls whose arguments reference user memory that may be paged out to disk.

**21.4** The primary advantages of shared libraries are that they reduce the memory and disk space used by a system, and they enhance maintainability.

When shared libraries are being used by all running programs, there is only one instance of each system library routine on disk, and at most one instance in physical memory. When the library in question is one used by many applications and programs, then the disk and memory savings can be quite substantial. In addition, the startup time for running new programs can be reduced, since many of the common

functions needed by that program are likely to be already loaded into physical memory.

Maintainability is also a major advantage of dynamic linkage over static. If all running programs use a shared library to access their system library routines, then upgrading those routines, either to add new functionality or to fix bugs, can be done simply by replacing that shared library. There is no need to recompile or relink any applications; any programs loaded after the upgrade is complete will automatically pick up the new versions of the libraries.

There are other advantages too. A program that uses shared libraries can often be adapted for specific purposes simply by replacing one or more of its libraries, or even (if the system allows it, and most UNIXs including Linux do) adding a new one at run time. For example, a debugging library can be substituted for a normal one to trace a problem in an application. Shared libraries also allow program binaries to be linked against commercial, proprietary library code without actually including any of that code in the program's final executable file. This is important because on most UNIX systems, many of the standard shared libraries are proprietary, and licensing issues may prevent including that code in executable files to be distributed to third parties.

In some places, however, static linkage is appropriate. One example is in rescue environments for system administrators. If a system administrator makes a mistake while installing any new libraries, or if hardware develops problems, it is quite possible for the existing shared libraries to become corrupt. As a result, often a basic set of rescue utilities are linked statically, so that there is an opportunity to correct the fault without having to rely on the shared libraries functioning correctly.

There are also performance advantages that sometimes make static linkage preferable in special cases. For a start, dynamic linkage does increase the startup time for a program, as the linking must now be done at run time rather than at compile time. Dynamic linkage can also sometimes increase the maximum working set size of a program (the total number of physical pages of memory required to run the program). In a shared library, the user has no control over where in the library binary file the various functions reside. Since most functions do not precisely fill a full page or pages of the library, loading a function will usually result in loading in parts of the surrounding functions, too. With static linkage, absolutely no functions that are not referenced (directly or indirectly) by the application need to be loaded into memory.

Other issues surrounding static linkage include ease of distribution: it is easier to distribute an executable file with static linkage than with dynamic linkage if the distributor is not certain whether the recipient will have the correct libraries installed in advance. There may also be commercial restrictions against redistributing some binaries as shared libraries. For example, the license for the UNIX "Motif" graphical environment allows binaries using Motif to be distributed freely as long as they are statically linked, but the shared libraries may not be used without a license.

**21.5** Using network sockets rather than shared memory for local communication has a number of advantages. The main advantage is that the socket programming interface features a rich set of synchronization features. A process can easily determine when new data has arrived on a socket connection, how much data is present, and who sent it. Processes can block until new data arrives on a socket, or they can request that a signal be delivered when data arrives. A socket also manages separate connections. A process with a socket open for receive can accept multiple connections to that socket and will be told when new processes try to connect or when old processes drop their connections.

Shared memory offers none of these features. There is no way for a process to determine whether another process has delivered or changed data in shared memory other than by going to look at the contents of that memory. It is impossible for a process to block and request a wakeup when shared memory is delivered, and there is no standard mechanism for other processes to establish a shared memory link to an existing process.

However, shared memory has the advantage that it is very much faster than socket communications in many cases. When data is sent over a socket, it is typically copied from memory to memory multiple times. Shared memory updates require no data copies: if one process updates a data structure in shared memory, that update is immediately visible to all other processes sharing that memory. Sending or receiving data over a socket requires that a kernel system service call be made to initiate the transfer, but shared memory communication can be performed entirely in user mode with no transfer of control required.

Socket communication is typically preferred when connection management is important or when there is a requirement to synchronize the sender and receiver. For example, server processes will usually establish a listening socket to which clients can connect when they want to use that service. Once the socket is established, individual requests are also sent using the socket, so that the server can easily determine when a new request arrives and who it arrived from.

In some cases, however, shared memory is preferred. Shared memory is often a better solution when either large amounts of data are to be transferred or when two processes need random access to a large common data set. In this case, however, the communicating processes may still need an extra mechanism in addition to shared memory to achieve synchronization between themselves. The X Window System, a graphical display environment for UNIX, is a good example of this: most graphic requests are sent over sockets, but shared memory is offered as an additional transport in special cases where large bitmaps are to be displayed on the screen. In this case, a request to display the bitmap will still be sent over the socket, but the bulk data of the bitmap itself will be sent via shared memory.

**21.6** The performance characteristics of disk hardware have changed substantially in recent years. In particular, many enhancements have been introduced to increase the maximum bandwidth that can be achieved on a disk. In a modern system, there can be a long pipeline between the

operating system and the disk's read-write head. A disk I/O request has to pass through the computer's local disk controller, over bus logic to the disk drive itself, and then internally to the disk, where there is likely to be a complex controller that can cache data accesses and potentially optimize the order of I/O requests.

Because of this complexity, the time taken for one I/O request to be acknowledged and for the next request to be generated and received by the disk can far exceed the amount of time between one disk sector passing under the read-write head and the next sector header arriving. In order to be able efficiently to read multiple sectors at once, disks will employ a readahead cache. While one sector is being passed back to the host computer, the disk will be busy reading the next sectors in anticipation of a request to read them. If read requests start arriving in an order that breaks this readahead pipeline, performance will drop. As a result, performance benefits substantially if the operating system tries to keep I/O requests in strict sequential order.

A second feature of modern disks is that their geometry can be very complex. The number of sectors per cylinder can vary according to the position of the cylinder: more data can be squeezed into the longer tracks nearer the edge of the disk than at the center of the disk. For an operating system to optimize the rotational position of data on such disks, it would have to have complete understanding of this geometry, as well as the timing characteristics of the disk and its controller. In general, only the disk's internal logic can determine the optimal scheduling of I/Os, and the disk's geometry is likely to defeat any attempt by the operating system to perform rotational optimizations.


# Windows XP

## Practice Exercises

**22.1** A 32/64 bit preemptive multitasking operating system supporting multiple users. (1) The ability automatically to repair application and operating system problems. (2) Better networking and device experience (including digital photography and video).

**22.2** Design goals include security, reliability, Windows and POSIX application compatibility, high performance, extensibility, portability and international support. (1) Reliability was perceived as a stringent requirement and included extensive driver verification, facilities for catching programming errors in user-level code, and a rigorous certification process for third-party drivers, applications, and devices. (2) Achieving high performance required examination of past problem areas such as I/O performance, server CPU bottlenecks, and the scalability of multithreaded and multiprocessor environments.

**22.3** (1) As the hardware powers on, the BIOS begins executing from ROM and loads and executes the bootstrap loader from the disk. (2) The NTLDR program is loaded from the root directory of the identified system device and determines which boot device contains the operating system. (3) NTLDR loads the HAL library, kernel, and system hive. The system hive indicates the required boot drivers and loads them. (4) Kernel execution begins by initializing the system and creating two processes: the system process containing all internal worker threads, and the first user-mode initialization process: SMSS. (5) SMSS further initializes the system by establishing paging files and loading device drivers. (6) SMSS creates two processes: WINLOGON, which brings up the rest of the system, and CSRSS (the Win32 subsystem process).

**22.4** (1) The HAL (Hardware Abstraction Layer) creates operating system portability by hiding hardware differences from the upper layers of the operating system. Administrative details of low-level facilities are provided by HAL interfaces. HAL presents a virtual-machine interface that is used by the kernel dispatcher, the executive and device drivers.

(2) The kernel layer provides a foundation for the executive functions and user-mode subsystems. The kernel remains in memory and is never preempted. Its responsibilities are thread scheduling, interrupt and exception handling, low-level processor synchronization, and power failure recovery. (3) The executive layer provides a set of services used by all subsystems: object manager, virtual memory manager, process manager, local procedure call facility, I/O manager, security monitor, plug-and-play manager, registry, and booting.

**22.5** Objects present a generic set of kernel mode interfaces to user-mode programs. Objects are manipulated by the executive-layer object manager. The job of the object manager is to supervise the allocation and use of all managed objects.

**22.6** The process manager provides services for creating, deleting, and using processes, threads and jobs. The process manager also implements queuing and delivery of asynchronous procedure calls to threads. The local procedure call (LPC) is a message-passing system. The operating system uses the LPC to pass requests and results between client and server processes within a single machine, in particular between Windows XP subsystems.

**22.7** The I/O manager is responsible for file systems, device drivers, and network drivers. The I/O manager keeps track of which device drivers, filter drivers, and file systems are loaded and manages buffers for I/O requests. It furthermore assists in providing memory-mapped file I/O and controls the cache manager for the whole I/O system.

**22.8** Environmental subsystems are user-mode processes layered over the native executable services to enable Windows XP to run programs developed for other operating systems. (1) A Win32 application called the virtual DOS machine (VDM) is provided as a user-mode process to run MS-DOS applications. The VDM can execute or emulate Intel 486 instructions and also provides routines to emulate MS-DOS BIOS services and provides virtual drivers for screen, keyboard, and communication ports. (2) Windows-on-windows (WOW32) provides kernel and stub routines for Windows 3.1 functions. The stub routines call the appropriate Win32 subroutines, converting the 16-bit addresses into 32-bit addresses.

**22.9** Support is provided for both peer-to-peer and client-server networking. Transport protocols are implemented as drivers. (1) The TCP/IP package includes SNMP, DHCP, WINS, and NetBIOS support. (2) Point-to-point tunneling protocol is provided to communicate between remote-access modules running on Windows XP servers and other client systems connected over the internet. Using this scheme, multi-protocol virtual private networks (VPNs) are supported over the internet.

**22.10** The NTFS namespace is organized as a hierarchy of directories where each directory uses a B+ tree data structure to store an index of the file names in that directory. The index root of a directory contains the top level of the B+ tree. Each entry in the directory contains the name and file reference of the file as well as the update timestamp and file size.

**22.11** In NTFS, all file-system data structure updates are performed inside transactions. Before a data structure is altered, the transaction writes a log record containing redo and undo information. A commit record is written to the log after a transaction has succeeded. After a crash the file system can be restored to a consistent state by processing the log records, first redoing operations for committed transactions and undoing operations for transactions that did not successfully commit. This scheme does not guarantee that user file contents are correct after a recovery, but rather that the file-system data structures (file metadata) are undamaged and reflect some consistent state that existed before the crash.

**22.12** User memory can be allocated according to several schemes: virtual memory, memory-mapped files, heaps, and thread-local storage.

**22.13** (1) Virtual memory provides several functions that allow an application to reserve and release memory, specifying the virtual address at which the memory is allocated. (2) A file may be memory-mapped into address space, providing a means for two processes to share memory. (3) When a Win32 process is initialized, it is created with a default heap. Private heaps can be created that provide regions of reserved address space for applications. Thread management functions are provided to allocate and control thread access to private heaps. (4) A thread-local storage mechanism provides a way for global and static data to work properly in a multithreaded environment. Thread-lock storage allocates global storage on a per-thread basis.

# Influential Operating Systems


No Practice Exercises