A Solutions Manual for

Operating Systems
—A Concept-Based Approach

Second Edition

Dhananjay M. Dhamdhere

Chapter 1

Introduction

Problem 1: (a) A user expects 'immediate' response to his computational request. A sure way to achieve it is not to have any other programs in the system, which affects efficient use of the system. (b) A user expects a large part of a disk to be reserved for his exclusive use, which limits the number of users of the system, which in turn affects efficient use of the system.

Problem 2: Fairness in the use of CPU can be achieved by preempting the CPU. This way a program cannot dominate use of the CPU leading to poor service to other programs. Similarly memory may be preempted by removing a program from memory so that another program can be loaded in memory and given an opportunity to execute. Section 2.5.3.1 discusses priority based preemption of the CPU to achieve efficient use of a system.

Problem 3: Students should be able to answer this question based on their knowledge of computer architecture. Informally, we say a device is *position-sensitive* if its operation involves a notion of position and depends on the position it was in when last used. Preemption of a device and its use by another program interferes with its position. Hence position-sensitive devices cannot be preempted. Examples are sequential devices such as printers, tapes and cartridges, etc. A device is not position-sensitive if a position has to be explicitly specified in any command required to use it. Memory, CPU and disks are not position sensitive. Hence they can be preempted. However, disks are not preempted; programs simply use different areas on a disk.

To be able to resume execution of a program 'as if preemption had not occurred', the OS should be able to restore the situation in the program as at the time of preemption. It is easy to restore the CPU and memory to the situation when preemption occurred by loading back correct values in the CPU registers (this is discussed in Section 2.1.1) and in memory locations. However, it is not possible to position the magnetic tape cartridge that was in use at the time of preemption at same point as at

the time of preemption.

Problem 4: When a file is to be printed, a virtual printer is created for it and the file is sent to the virtual printer. The print server keeps track of the order in which various virtual printers were created during operation of the system, and arranges to map them on the real printer for printing the file.

Problem 5: In partitioned resource allocation, a 'resource partition' is assigned to a program when its execution is to be initiated. The resource partition contains all the resources the user's program may require. Hence the program does not get stuck or delayed midway during its execution due to lack of required resources. In a pool based allocation strategy, a program may be stuck midway during its execution. Hence the first part of this statement. However, some of the resources in a resource partition may be wasted because the program to which the partition is allocated may not use them.

Problem 6: Pool-based allocation requires a resource request to be made when a resource is needed, hence it incurs allocation overhead every time a resource is required. If several resource requests are made by a program, this may cause high allocation overhead. The primary benefit of pool-based resource allocation is that resources can be allocated only when needed, which avoids resource idling. It also enables more programs to be run at any time—if pool based allocation is not used, resources allocated to some programs would remain idle while other programs would not be able to obtain the resources they need for their execution.

Pool based allocation provides higher efficiency by reducing resource idling. It provides high user convenience by permitting a program to request and obtain all the resources within a system for its own use, compared to a limited set of resources that can be allocated to it in partitioned resource allocation.

Problem 7: Swapping incurs the overhead of I/O operations involved in (a) copying a program's instructions and data onto a disk when a program is swapped out, and (b) copying the same back into memory when the program is swapped in.

In systems having small memory, the memory size restricts the number of programs that can be serviced simultaneously. This effect can lead to idling of the CPU because it does not have enough work to perform. Swapping increases the number of programs that can be serviced simultaneously, which may lead to higher CPU efficiency because the CPU now has more work to perform.

Problem 8: A sequence of programs primarily facilitates user convenience. In its absence, a user would have to periodically check whether a program has terminated, and then decide whether it is meaningful to execute the next program. These user actions would consume time. A sequence of programs saves this time for the user. It also saves the system time that would have been wasted if the user was slow in performing these checks and initiating the next program.

A virtual device can be created whenever desired, hence execution of a user's

program does not have to be delayed until a device required by it becomes available. It increases the number of programs that can be active simultaneously, thereby improving system efficiency.

Problem 9: Resource consumption can be computed in terms of the amount of time over which a resource is allocated to a program. A dormant program has two kinds of resources tied to it: resources automatically allocated to it by the OS, e.g., memory, and resources allocated at its own request, e.g., I/O devices. Both kinds of resources remain idle when a program is idle. (Note that some 'invisible' resources are also allocated to a program. These resources include data structures used by the OS to control execution of a program.)

Problem 10: A command processor is obviously a component of an OS. Compiler for a concurrent programming language helps in creating an executable form for a program, which may consist of a set of co-executing programs, however it does not execute the computations by itself. Hence it is not an OS component.

The ssh program helps to set up execution of a program in a remote computer; however, it does not handle user programs the same way as a command processor, so it is not an 'OS component'. A file system helps in handling resources like I/O devices, hence it is an OS component. However, a user may own a few I/O devices connected to a computer system, and use his own file system to access these devices. Such a file system cannot be called an 'OS component'.

Problem 11: The physical view can be shown in two parts. In the first part, the compiler of language L exists in the memory of the computer system. It reads the program written in language L and outputs a ready-to-execute form of the program called a *binary program*. The second part of the view resembles Figure 1.3(b). The ready-to-execute form of the program exists in memory. It reads its data and executes its instructions on the CPU to produce results.

Problem 12: The purpose of this problem is to make the student think of the practical arrangement involved in use of files. Students should be encouraged to think of as many details as possible. The physical view would show the program in memory, information about the file maintained by the file system, the file stored on a disk, and some part of it existing in memory, which is being processed by the program.

Problem 13: A few representative computing environments are as follows:

An embedded environment	The computer is a part of a larger system, e.g., an appliance, and controls a part of its functioning.
A workstation environment	A single user uses a computer system.
A server environment	A computer system is used simultaneously by a group of users.

Chapter 2

Overview of Operating Systems

Problem 1: The following instructions should be privileged: (a) Put the CPU in privileged mode, (b) Load bound registers, (d) Mask off some interrupts, and (e) forcibly terminate an I/O operation.

(Item (a) is actually meaningless. The privileged/non-privileged mode of the CPU is governed by contents of the 'privileged mode (P)' field of the PSW. An instruction called 'Load PSW' is provided to load new contents into PSW fields, including in the P field. This instruction is privileged. The new mode of the CPU may be privileged or non-privileged depending on the new content of the P field.)

Problem 2: The 'privileged mode (P)' field of the PSW indicates whether the CPU is in the privileged mode. The kernel forms an 'initial PSW' for a program when the program is to be initiated. The PC field of this PSW points to the first instruction of the program, and its P field is set to 'off' (i.e., 0) to indicate that the CPU is in the non-privileged mode. Let a program $prog_1$ be in execution when an interrupt occurs. Contents of the PSW are saved in some kernel data structure corresponding to $prog_1$ (later in Chapter 3 we call it the PCB). A new PSW is loaded into the CPU by the interrupt action. The P field of this PSW is 'on' (i.e., 1), so that the CPU is in the privileged mode. The saved PSW of $prog_1$ is loaded back into the CPU when $prog_1$ is to be resumed. This way, the CPU enters the privileged mode when the kernel gets control for execution, and the CPU enters the non-privileged mode when a user program is in execution (See Example 2.3).

Problem 3: This statement is valid. In a time sharing or multiprogramming system, execution of several programs is overlapped on the CPU. So when a program is scheduled, the cache contains (some) instructions and data of the programs that have executed on the CPU recently and only a small part of the cache may contain data and instructions of the scheduled program from the last time it executed on the CPU. Hence the program would have a poor cache hit ratio initially. On the other hand,

if a program is executed by itself, the cache would be occupied only by its own instructions and data and those of the kernel. Hence the program would experience a higher cache hit ratio than in time sharing or multiprogramming.

Problem 4: The P field should be 'on' (i.e., 1), so that the interrupt processing routine would be executed in the privileged mode. Contents of the IC field are irrelevant. The MPI field should contain memory allocation information that gives the kernel a privilege to access the entire memory. The CC field is irrelevant. The PC field should point at the first instruction of the interrupt processing routine. Contents of the IM field control the occurrence of interrupts in the system. See Problems 6 and 7 for a related discussion.

Problem 5: The answer is contained in Example 2.1. When an interrupt occurs, the CPU state is saved by the interrupt processing routine. If control is returned to the program immediately after the kernel processes the interrupt, the CPU state will be restored before the program resumes execution. Thus, the CC field will contain the correct condition code, so the branch instruction would execute correctly. (If the program is not immediately rescheduled, then the kernel moves the saved PSW information into the CPU at some future moment when it decides to resume the program. Thus, in either case, the CC and PC fields are restored to their contents as at the time of the interrupt before the program's execution is resumed.)

Problem 6: Masking off all interrupts provides a guarantee that new interrupts cannot occur while the kernel is processing an interrupt; new interrupts will remain pending. This guarantee simplifies coding of the kernel. However, this scheme has the drawback that the kernel will not be able to process higher priority interrupts until after processing of the current interrupt is completed. Hence the kernel's response to certain important conditions in the system may be delayed.

Problem 7: If another interrupt occurs while the kernel is processing one interrupt, either the new interrupt could be ignored until processing of the current interrupt is completed, or it could be accepted for processing. In the latter case, nested interrupt servicing would occur as shown in Figure 2.10—after processing the new interrupt, the kernel would resume processing of the previous interrupt. Thus, interrupts would be processed in a last-in-first-out (LIFO) manner.

Use of a stack for the saved PSW information helps in an obvious manner. The PSW of the interrupted program is pushed onto this stack whenever an interrupt occurs. The kernel now starts processing the interrupt. When a second interrupt occurs, the PSW of the interrupt processing routine that was processing the first interrupt would be pushed on the stack. After finishing processing of the second interrupt, the kernel loads the information at the top of the stack into the PSW. This way, the first interrupt processing routine is resumed when the kernel finishes processing the second interrupt. When processing of the first interrupt is completed, the kernel returns

to the program that was in execution when the first interrupt occurred.

Problem 8: System calls that make resource requests would lead to suspension of the program if the request cannot be satisfied straightaway. Systems calls that request operations on files would typically lead to blocking until the desired operation, such as opening or closing of a file, has been completed successfully. Communication-related system calls for receiving a message or setting up a connection would similarly lead to blocking.

Problem 9: When a software interrupt signifying a system call occurs, the interrupt hardware transfers control to the kernel. After processing the interrupt, the kernel passes control to a user program. Both these actions cause switching between programs. This overhead can be reduced by making a single request to obtain a large chunk of memory, instead of making several requests for small areas of memory.

Problem 10: When the program tries to read the sixth data card, the kernel encounters a /* statement and realizes that the program is trying to read past its data. Hence the kernel decides to abort the program. It now reads and ignores all cards until the next // JOB card is encountered. The next job is processed starting with this card.

Problem 11: Throughput of a job in a batch processing system is the number of jobs processed per unit time. In an underloaded system, all jobs get processed, so the total time taken to process them is the same irrespective of the sequence in which they are processed. In an overloaded system, some jobs do not get processed. So the sequence in which jobs are processed is important for throughput of the system. When separate batches of short and long jobs are formed, some batch(es) of long jobs may not get processed at all. Hence the throughput would be better than when short and long jobs are put in the same batch.

Problem 13: Actions contributing to overhead of a multiprogramming system are: Allocation of memory and I/O devices, handling of system calls, handling of interrupts, and scheduling of programs.

Problem 14: A wrong classification of a CPU-bound program as an I/O-bound program would give the program a high priority. When scheduled, this program would execute for a long time. Hence it would monopolize the CPU and would not let any other program—especially an I/O-bound program—execute. Hence response times to other programs would suffer and throughput of the system would reduce. The throughput *vs.* degree of multiprogramming characteristic would be flatter than shown in Figure 2.18—throughput would not increase much with the degree of multiprogramming. If many programs are wrongly classified in this manner, fewer I/O bound programs would be serviced at any time, so throughput would stagnate at a lower value.

If an I/O-bound program is classified as a CPU-bound program, it would have a low priority. Hence its response time would suffer. If many programs are wrongly classified in this manner, the system would be used less efficiently because the CPU

may not have enough work to do; however, throughput of the system would not differ much from that shown in Figure 2.18.

Problem 15: The high priority program executing on the CPU makes a system call to request initiation of the I/O operation. The interrupt processing routine of the kernel analyses the system call and passes control to the routine that handles initiation of I/O operations. The routine knows that the program cannot execute until the I/O operation is completed, hence it invokes the scheduler to schedule another program for execution.

Problem 16: The following actions are common to both parts (a) and (b): Let program P be executing when the occurrence of an interrupt diverts the CPU to an interrupt processing routine. The interrupt processing routine realizes that an I/O operation is complete. It finds the program that had initiated the I/O operation, say, program Q, and notes that the program's execution can now be resumed. Now the kernel decides which process to schedule for execution.

(a) Program Q has a lower priority that program P, hence the kernel will schedule program P. Thus, program P resumes its execution as if nothing had happened. (b) Program Q has a higher priority than program P, hence Q is the highest priority program that can execute. So the kernel schedules Q.

Problem 18: (a) Since the program has the highest priority, each iteration of the loop would consume exactly $50+200 = 250$ msec. Hence the elapsed time is 12.5 seconds.

(b) For $n = 3$, the program would receive the first CPU burst between 150 and 200 msec. It would be followed by 200 msec of I/O operations. Since the CPU requirements of the other three programs amount to 150 msec in each iteration, this program would receive the CPU the moment its I/O operation completes. Hence its behavior is similar to that in case (a) except that it is offset by 150 msec. Hence its elapsed time is 12.65 seconds. The situation is similar for $n = 4$, except that it receives its first CPU burst between 200 and 250 msec. Hence its elapsed time is 12.70 seconds. For $n = 5$, the other programs keep the CPU fully occupied, so this program does not get a chance to execute.

Problem 19: The progress coefficient is obtained by dividing the elapsed time of each case of part (b) by the elapsed time of part (a). In the first two cases, it is $\frac{12.5}{12.65}$ and $\frac{12.5}{12.7}$, respectively, and it is 0 in the third case.

Problem 20: A CPU-bound program has a low priority, hence the statement is valid.

Problem 21: (a) Since the degree of multiprogramming does not change, there may be substantial periods of time when the CPU has no work. Hence the throughput may be limited by the availability of memory. (b) Expanding the memory increases m, but the throughput would be limited by the speed of the CPU, hence the throughput may not double. (c) Assuming a proper mix of programs, this proposal is the best, as it

does not have the drawbacks of proposals (a) and (b).

Problem 22: (a) False. Some program P_k with $b^k_{io} < b^h_{io}$ and a high enough priority may receive more than one CPU burst while the highest priority program is engaged in performing its I/O, so CPU idling may not occur even if this condition is met. (For example, the following sequence of events may occur sometime after P_h starts its I/O operation: some program P_k may receive a CPU burst, start an I/O operation, finish it, receive a second CPU burst and start a second I/O operation, etc., before P_h's I/O operation completes.) The following statement would be valid: $b^h_{io} > \Sigma_{j \neq h}(b^j_{cpu}) \wedge b^j_{io} > b^h_{io}$ for all $j \neq h$.

(b) True. The first condition ensures that no program can receive two CPU bursts while a higher priority program is engaged in an I/O operation. The second condition ensures that the CPU will be idle, which ensures that even the lowest priority program will receive an opportunity to use the CPU.

Problem 23: (a) False. Some of the programs $P_{i+1} \ldots P_{m-1}$ may be engaged in I/O (because they received a CPU burst recently), hence they may not need a CPU burst. This will permit P_m to receive a CPU burst even if $b^i_{io} < \Sigma_{j=i+1 \ldots m-1}(b^j_{cpu})$.
(b) True. $b^i_{cpu} > b^j_{io}$ for all $j > i$ ensures that by the time program P_i finishes its CPU burst, all of programs $P_{i+1} \ldots P_m$ would have finished their I/O operations. Now $b^i_{io} < \Sigma_{j=i+1 \ldots m-1}(b^j_{cpu})$ would ensure that program P_m would be starved of CPU attention.

Problem 24: An important point is that when $t_p > \delta$ a program has to be scheduled more than once before it can produce a response, so its response time may be larger than when $t_p < \delta$. Example 4.5 in Chapter 4 illustrates this point.

Problem 25: The situation mentioned in the problem arises if $t_p < \delta$ for one person while $t_p > \delta$ for another person (see Problem 24).

Problem 26: False. Consider a situation in which all programs had received a CPU burst recently, and are engaged in performing I/O operations. In a time sharing system, the programs would receive their next CPU bursts in the order in which they complete their I/O operations. However, in a multiprogramming system a program receives the CPU if it is not engaged in performing an I/O operation and all higher priority programs are engaged in I/O operations. The lower priority programs do not influence this decision irrespective of whether they are engaged in performing an I/O operation. Thus, a few high priority programs may monopolize the CPU and starve low priority programs.

Problem 27: (a) Swapping incurs overheads in terms of CPU time to make swapping decisions and perform page-in and page-out operations. However, it effectively increases the number of programs that can be handled simultaneously by the system, which increases the system utilization. Hence the answer to this question depends on whether the CPU and I/O subsystem are underutilized in the system. If they

are, swapping increases the efficiency of system utilization; otherwise, it does not. (b) Yes. (c) No, because a program that has been swapped out to a disk should be swapped back into memory before it is due to be scheduled to use the CPU. Due to the priority-driven nature of multiprogramming scheduling, it is not possible to predict when a program may be scheduled next.

Problem 28: A program needs to exist in memory when its I/O operation is in progress (so that data being read by an input operation can be deposited into memory and data to be output can be obtained from memory). Hence the time sharing system would transfer a program to the being swapped-out list when (a) the program is preempted at the end of a time slice, or (b) when a program that relinquishes the CPU due to an I/O operation completes the I/O operation. A program would be transferred to the being swapped-in list when the kernel knows that the program is likely to be scheduled in near future.

Problem 29: There is some hope of meeting the response requirement when the time sharing system has an average response time of 0.2 seconds; however, it is not guaranteed.

Problem 30: $t_C < \delta$ and $(n-1) \times (t_v + \delta) + (t_v + t_C) + t_I < t_D$. It is possible to meet the response requirement even when $t_C > \delta$, but the condition is somewhat more complex.

Problem 31: In a multiprogramming system, an I/O-bound activity is given a higher priority than a non I/O-bound activity to achieve high throughput. However, in a real time system, a higher priority is assigned to an activity that is more critical, irrespective of whether it is an I/O-bound activity. Hence, an I/O activity may be assigned a lower priority.

Problem 32: Example 3.1 is a motivating example for this problem. When several processes are created in the application, the I/O operations of one process can overlap with CPU processing in another process.

Problem 33: (a) The alarm raising functions, functions (b) and (d), are hard real time functions. Function (e) might also be considered to be a real time function. It is a soft real time function.

(b) and (c) The application has very light CPU and I/O requirements. Hence there is no need to create multiple processes. The fuel level and engine conditions should be detected by sensors of an electronic or electro-mechanical kind, and should raise interrupts. Functions (b) and (d) would be performed as an aspect of interrupt processing. A timer interrupt should be raised periodically to activate function (e). Speed and distance should be inputs that lead to special interrupts. A single process will handle these interrupts and control the display. Interrupt priorities should be defined such that engine conditions have the highest priority, followed by fuel level. Masking

of interrupts should be used to ensure that high priority functions are not delayed.

Problem 34: The probability of failure of a CPU should be 1 in 100.

Problem 35: The three parts of the computation executed in three different computer systems may need to interact with one another to share data or coordinate their activities, hence they may not be able to execute simultaneously all the time. It leads to a speed-up that is < 3.

Chapter 3

Processes and Threads

Problem 1: This situation can arise due to several reasons. The most obvious one is that a *running* → *blocked* or *running* → *ready* transition in one process causes a *ready* → *running* transition for another process due to scheduling. A *running* → *terminated* transition for a process causes *blocked* → *ready* transitions for processes waiting for its termination. A *blocked* → *ready* transition by a process that has a higher priority than the *running* process causes a *running* → *ready* transition for the running process and a *ready* → *running* transition for the high priority process.

Problem 2: (a),(b) If the kernel is able to satisfy the request to receive a message or allocate memory, it can simply dispatch the process. Otherwise, it blocks the process and schedules another process. (c) The kernel can provide the status information to the requesting process and simply dispatch the process. If the request is invalid, the kernel can simply return an error code and dispatch the process or it can abort the process and perform scheduling to select another process for use of the CPU. (e) Termination of a child process may release a resource, which would activate a process that is waiting for its termination. If so, the kernel should perform scheduling to select a process for use of the CPU; otherwise, it can simply dispatch the requesting process.

Problem 3: The *blocked swapped* → *ready swapped* transition can occur if the request made by the *blocked swapped* process is satisfied, e.g., if it was blocked for a message and a message is sent to it. A process that is blocked for an I/O operation may be swapped out if its I/O operation cannot be started straightaway. Such a process would make the transitions *blocked swapped* → *blocked* → *ready* → *ready swapped* → *ready*.

Problem 4: A kernel need not perform scheduling after handling an interrupt if the interrupt did not cause a change in state for any process, e.g., if a request is satisfied

straightaway.

Problem 5: Let us assume that event processing is performed by disabling all interrupts. Hence events are processed strictly one at a time. If scheduling is performed after every event, the number of scheduling actions equals the number of events; however, as seen in the previous problem, scheduling need not be performed if an event does not cause a state change for any process. Let N_i be the number of times an event E_i occurs, and P_i be the probability that it causes a state change. The number of scheduling actions is given by $\Sigma_i(N_i \times P_i)$.

If other events are permitted to occur while an event is being processed, then the number of scheduling actions could be even smaller—scheduling is performed after completing processing of an 'outermost' event only if the event, or any nested events, had caused a state change.

Problem 6: A Unix process can wait only for termination of its children. Hence a process cannot wait for a process that is not its child. Also, Unix lacks a feature by which a process can wait until a specific child process terminates. A process issuing a *wait* is activated when *any* of its child processes terminate (see Section 3.5.1). If the terminating process is not the one it wishes to wait on, it would have to once again issue a *wait*. This is cumbersome.

The *proc* structure of a process lives on even after the process terminates. This feature is needed so that the parent process can check the termination status of the process.

Problem 8: SIGINT is raised due to actions of a user; SIGFPE, SIGILL, SIGSEGV and SIGSYS are raised due to execution of instructions; SIGXCPU is raised due to a timer interrupt; SIGKILL and SIGXFSZ are raised due to actions of a process; while SIGCHLD is raised when a process dies or is suspended.

Problem 9: Section 3.5.1 describes how the Unix kernel arranges to pass control to a process when a signal is sent to it. A similar arrangement can be used when a process is in the *blocked* state, or in one of the *running* or *ready* states when a signal arrives, In the former case, the kernel should change the state of the process to *ready*, arrange to gain control at the end of execution of the signal handler (which may be a handler provided by a process, or the default handler of the kernel), and change the state of the process back to *blocked*. This provision is not needed in the other two cases.

Problem 10: The reservations system can be implemented by allocating the reservations data base as a resource to a process, and creating threads in this process to perform reservations. This arrangement would enable the threads to share the reservations data base without involving the kernel and the overhead of switching between requests would be low. Instead, if reservations are performed by creating several processes, it would be necessary to provide for sharing of the reservations data base. Switching overhead would also be higher.

Problem 11: A user-level thread should avoid making any system call that can lead

to blocking. This is easier said than done, because certain system calls are inherently blocking in nature, e.g., waiting until an I/O operation completes. The kernel must provide non-blocking calls to overcome this difficulty, e.g., an 'I/O request' system call that simply passes on an I/O request to the kernel, and a 'check I/O status' call that returns with a status flag indicating whether an I/O operation has completed.

Whenever a user thread wishes to wait for occurrence of an event, it makes a call on the threads library. The library uses a non-blocking call to check the status of the event and block the thread if the event has not occurred. It now schedules another thread to execute. It also periodically keeps checking the status of the event and changes the state of the thread to *ready* when the event occurs. If no thread is in the *ready* state, the library would block itself, i.e., it would block the process, by making a system call to indicate that it wishes to wait until one of several events occur—these are the individual events on which the threads are blocked. The kernel would activate the process when one of these events occurs. The threads library would check which event has occurred and activate the corresponding thread.

Problem 12: This statement is incorrect. Consider an example. User-level threads provide concurrency because only one thread of a process can be in execution even if several processors exist in the system. However, this arrangement also provides speedup possibilities because the I/O of one thread in a process can be overlapped with execution of another thread of the same process.

Problem 14: The number of user-level threads in a process would depend on the number of logically concurrent units of execution in a process. This arrangement would provide concurrency between these units. In a web-based application server, a user thread can be created for each request. Ideally, you need only as many LWPs as the number of CPUs; however, more LWPs would be created due to the following reason: The Solaris kernel permits a user thread to make blocking calls. In a uniprocessor system, creating only one LWP for an application may deny parallelism between threads due to this reason, so it is desirable to create more LWPs.

Problem 15: (a) A kernel-level thread provides parallelism but incurs higher overhead. If the system contains more than one CPU, creation of kernel-level threads for CPU-bound activities would exploit the parallelism provided in the system; however, if the system contains a single CPU, it is better to use user-level threads and curtail the overhead.

(b) If The process contains a CPU-bound activity and the system has > 1 CPU, a kernel-level thread would have been created in the process. In this situation, creating a kernel-level thread for the I/O-bound activity would exploit the parallelism provided by multiple CPUs. However, if a process does not contain a CPU-bound activity, creating kernel-level threads for the I/O-bound activities incurs higher overhead but does not provide any benefit.

Problem 16: The application server can create a new thread under the mentioned

conditions (if request queue size exceeds some number). It is useful to destroy unwanted threads as they consume kernel resources. This can be done by sending signals to the threads. Each thread must provide a signal handler that performs a 'terminate me' system call.

Problem 17: Two groups of processes should be created, each with 5 processes in it. However, a process *can* send signals that are not in the same process group.

Problem 18: A race condition leads to a result that is different from the results produced by two operations a_1, a_2 performed in a strict sequence, i.e., either in the sequence a_1, a_2 or in the sequence a_2, a_1. Let a query that is an evaluation of some expression e, be performed concurrently with an operation a_1. Operations in this reservations system increment *nextseatno* by 1. Let $e = 5$ before a_1 is performed and $e = 6$ after a_1. The result would be one of the two. Hence it is the equivalent of performing the operations e, a_1 or a_1, e.

In general a race condition occurs if more than one operation concurrently updates a value. A race condition can also occur if one operation updates a value in more than one step and another operation merely reads the value. For example, if operation a_1 first increments *nextseatno* by 2 and then decrements it by 1, e might return a value 7 instead of 5 or 6. This result does not conform to either the sequence e, a_1 or a_1, e.

Problem 19: Applying Definition 3.6, we find ($\{P_1, P_5, P_6\}$, A), i.e., processes P_1, P_5 and P_6 require data access synchronization due to use of array B. Other synchronizations are (Note that 'read' and 'find' imply a write!): ($\{P_1, P_2\}$, A), ($\{P_2, P_4\}$, A_{max}), ($\{P_3, P_4\}$, X), and ($\{P_4, P_6\}$, Y). Control synchronization pairs are: (P_1, P_2), (P_1, P_5), (P_2, P_4), (P_3, P_4), (P_4, P_6), and (P_5, P_6).

When we analyze these requirements, we find that all data access synchronization requirements are incorporated by the control synchronizations! Actually, this is obvious from the problem specification itself.

Chapter 4

Scheduling

Problem 1: In a user-centric view, a user who initiated a CPU-bound process may expect fast response; however, an effort to provide it will affect system throughput. Assigning a low priority to a CPU-bound process will ensure good system throughput but will affect response time of the process.

Problem 3: (a) False. A short request finishing earlier may have a larger weighted turn around that a long request finishing later. Consider 3 requests with execution times 2,3 and 10. (b) True. Consider an I/O-bound request and a CPU-bound request. In RR scheduling, the I/O bound request is put at the end of the scheduling queue when it finishes its I/O operation. In LCN scheduling, it would be scheduled soon after it finishes its I/O operation because it would not have consumed much CPU time.

Problem 4: Assume that all requests arrive at the same time instant. Let the requests be completed in the order $P_1, P_2 \ldots P_n$. Now interchange processes P_i and P_{i+1} for some value of i and show that the average turn-around time is either the same (if the execution requirements of these processes are identical), or increases due to the interchange. Hence the above order provides the minimum value of the average turn-around time.

Problem 5: (a) STG favors short processes, HRN does not. STG suffers from starvation, HRN does not. (b) LCN favors newly arrived processes, HRN does not. LCN suffers from starvation, HRN does not. (c) If the time slice is smaller than the duration of a typical I/O operation, the RR policy provides better service to an I/O-bound process; otherwise, it provides better service to a CPU-bound process. However, the HRN policy would provide equal service to both I/O-bound and CPU-bound processes.

Problem 6: Performance is better if response-ratio scheduling is implemented such

that the process with the highest response ratio is always being executed. Hence very small values of t are attractive. However, the overhead would be unacceptably high. At the other extreme, very large values of t imply low overhead but a very crude implementation of HRN since the process with highest response ratio may not always be in execution. Hence performance in terms of throughput may suffer. Also, a newly arrived process would be denied an opportunity to execute for some time after arrival.

Problem 7: (a) Following Example 4.5 we assume that a response is produced at the end of the CPU burst. Hence the response times of processes are $53 \times n$ msec, for $n = 1 \ldots 10$, for the first request. Hence the average response time is 53×5.5 msec = 291.5 msec. For a subsequent request, the response times are $10 \times 53 - 200$ = 330 msec. If the loops execute a large number of times, the average response time would be approx. 330 msec. (b) The response times are $2 \times 10 \times 23 + 13 \times n$, for $n = 1 \ldots 10$. For a subsequent request, the response times are $2 \times 10 \times 23 + 10 \times 13 - 200 = 390$ msec. The average response time is thus approx. 390 msec.

Problem 8: The interrupt handler in the kernel passes control to the event handler for timer interrupts (see Figure 4.6). This event handler invokes the context save mechanism, changes the state of the process to *ready*, and passes control to the scheduler. The scheduler selects a process for execution and invokes the process dispatching mechanism to pass to resume operation of the selected process.

Problem 9: CTSS uses a multilevel adaptive scheduling policy, hence a process at scheduling level i gets scheduled only if all processes in higher priority levels are in the *blocked* state. It is not possible to predict when this situation will arise and how long it will last. Processes at every level may also get promoted or demoted dynamically. All these aspects make it impossible to predict the average response time of processes at level i.

Problem 10: Let the current value of the time slice be δ. Some processes are classified as I/O-bound processes because their CPU burst is smaller than δ seconds; other processes are classified as CPU-bound processes. When the time slice is reduced, some processes that were earlier classified as I/O-bound processes would be reclassified as CPU-bound processes. It would improve the response time of I/O-bound processes because there would be fewer I/O-bound processes, but increase the scheduling overhead and thereby reduce the throughput of the system. Increasing the time slice would have the converse effect.

The purpose of varying the time slice is to adapt dynamically to the nature of processes to obtain a balance of good response times and low overhead. The effect of varying the time slice is best seen when most processes are CPU-bound or I/O-bound. In the former case, the time slice would be set at a large value, which would incur low overhead; in the latter case, it would be set at a small value, which would

provide good response times.

Problem 11: (a) Unix scheduling considers the amount of CPU attention received by a process and favors processes that have not received much CPU attention recently. However, it differs from LCN in that it uses a decay function, such that only the CPU time received by a process in recent past is considered for scheduling purposes. Consequently, it does not suffer from starvation, whereas the LCN policy suffers from starvation.

(b) HASP uses two scheduling levels meant for I/O-bound processes and CPU-bound processes. An I/O-bound process is demoted if it uses up the time slice completely. A CPU-bound process is promoted if it fails to use up the time slice completely. Hence the HASP approach resembles two-level adaptive scheduling. However, HASP does not use a fixed definition of scheduling levels since the time slice is varied to maintain the correct mix of processes.

Problem 12: The starting deadline of a process is given by

$$SD_i = D_{application} - \Sigma_{k \in descendant(i)}\ x_k - x_i$$

$D_{application}$ is 25 seconds, so the starting deadlines of the processes are 6, 13, 11, 16, 14 and 20 seconds, respectively.

Problem 13: An interactive process would not have used too much CPU time recently. Hence it will enjoy a higher effective scheduling priority compared to a non-interactive process, which would have used more CPU time. However, it is not so when a non-interactive process starves.

Problem 14: If processes do not perform I/O operations, a process P_i that was a recent selection from g_i would have used CPU time recently. The kernel considers only the recent CPU time used by a process, so the scheduling priority of process P_i would be smaller than that of any other process in the group. Hence some other process would be scheduled before P_i is scheduled again. (This argument assumes that the effect of CPU utilization by a process does not decay completely by the time the system makes another selection from the group. It also assumes that at least one other process of the group is in *ready* state.)

Problem 15: The resulting scheduling policies can be described as follows: (a) Most completed process next, (b) Process that has spent the most time in the *ready* state next, (c) Most interactive process next, and (d) Least completed process next. If priority of a process is set to 0 every time it is scheduled, the scheduling policies have the following properties: (a) Favors CPU-bound processes, (b) Performs FCFS in the ready queue, (c) Favors I/O-bound processes, and (d) Favors least completed process when last scheduled, i.e., it favors I/O-bound processes.

Problem 16: (a) Yes. (b) No. (c) Yes.

Problem 18: The mean wait time for FCFS is $\frac{W_0}{1-\rho}$ (see Table 4.16), and $W_0 = \frac{\alpha}{\omega^2}$.

Thus, mean wait time is $\frac{\alpha/\omega^2}{1-\alpha/\omega} = \frac{\alpha}{\omega\times(\omega-\alpha)}$. Now, α = 20, and mean wait time = 2 seconds. Hence, ω = approx. 20.5, and size of each request = 1/20.5 seconds = approx. 48.7 msec.

Problem 19: Following Example 4.17, the ready queue should have at least 6 entries in it.

Problem 20: Analogous to Problem 18, the mean execution rate is approx. 5.25.

Problem 21: From Table 4.16, for small service time (t), the mean wait time is $W_0 + \frac{\rho^2}{1-\rho} \times \frac{t}{2}$. Since $\omega = 8$, $1/\omega = 0.125$. So let t be 0.05 seconds. Hence $\rho = \frac{5}{8} =$ 0.625. The mean turn-around time is the mean wait time + t = approx. 0.095 + 0.05 seconds = approx. 0.145 seconds.

Chapter 5

Memory Management

Problem 1: Allocation and deallocation of program controlled dynamic data (PCD data) is under control of a process, hence PCD data may *not* be allocated and deallocated in the LIFO order.

Problem 2: False. If dynamic memory allocation is used, memory would be allocated to array A during execution of the program. Due to the overhead of memory allocation, program P would require more execution time when dynamic memory allocation is used. If the array is declared in a function, its allocation and deallocation may be performed several times during execution of P.

Problem 3: When dynamic memory allocation is used, memory is allocated and freed during the execution of a program, so a program's peak demand for memory when dynamic memory allocation is used may be smaller than the amount of memory allocated to it when static memory allocation is used.

Problem 4: For any allocation request, the best fit allocator always creates the smallest sized free area. On the contrary, the worst fit allocator always creates the largest sized free area. This fact avoids the main problem in best fit allocation—that free areas become too small to be used. Worst fit allocation is also better than first fit allocation in the same respect; however, it would incur higher overhead than first fit.

Knuth (1973) remarks that the first fit method stayed in action longer than the best fit method for the same memory size, i.e., the best fit method reached a stage when it could not satisfy a request for memory but the first fit method could satisfy it. This fact indicates that the free area sizes in the best fit approach became impractically small earlier than in the first fit approach. We can expect the worst fit approach to do as well as, and probably better than, the first fit approach. Accordingly, the memory utilization factor may be better for the worst fit approach rather than for the first fit

approach.

Problem 5: When free blocks are merged, a block has to be deleted from the free list and a new block has to be inserted. Deletions is faster when doubly linked lists are used than when singly linked lists are used.

Problem 6: Let a block b_i have only one tag situated at its start, and let block b_j, which follows b_i in memory, be released. Now, the allocator knows a_{b_j}, the start address of b_j, and it has to check the status of block b_i to decide whether b_i can be merged with b_j. If the size of b_i is known to be s_{b_i}, the allocator can access the tag of b_i from the memory location with the address $a_{b_j} - s_{b_i}$. If the size of b_i is not known, it cannot know where b_i's tag is located. Use of two boundary tags avoids this difficulty because the tag at b_i's end immediately precedes the first byte of b_j.

Problem 7: Internal fragmentation is larger in the powers-of-two allocator because splitting is not performed during allocation. External fragmentation may be larger in a powers-of-two allocator because merging is not performed while freeing memory.

Problem 8: Boundary tags are an overkill. Since the address of the buddy of a block can be computed as described in Section 5.4.1, it would be adequate to have the tag at the starting boundary irrespective of whether the block would be merged with its left neighbor or its right neighbor.

A tag is stored in a block itself, which reduces the amount of memory from the block that can be allocated. It can lead to larger internal fragmentation—a request for a memory area that is an exact powers of 2 in size would lead to allocation of a block whose size is the next power of 2.

Problem 9: Lookup of the status tag of a block is efficient, but slower than when boundary tags are used. However, the status tags are stored separately from the memory used for allocation, hence this method does not have the fragmentation problem created by storing status tag(s) in a block itself (see Problem 8).

Problem 10: Assuming $120 > 2^t$, 9 splits have been performed. 4 coalesce operations would be performed when a block of 120 bytes is freed.

Problem 11: (a) Obvious. (b) 11.

Problem 12: Execution efficiency of the buddy system depends on the overhead caused by (a) operations needed to check the allocation status of the buddy of a block that is being freed, and (b) splits and merges performed during allocation. The Fibonacci buddy system would incur a larger overhead due to factor (a) because determination of a buddy's address is not as simple as in the binary buddy system. In the general case, it is not possible to determine whether the overhead due to factor (b) is any different in the two buddy systems.

Memory efficiency in the buddy system is determined by internal fragmentation. The Fibonacci buddy system is likely to incur less internal fragmentation than the binary buddy system because sizes of blocks are spaced closer together in the Fi-

bonacci buddy system than in the binary buddy system.

Problem 13: If a buddy system allocator is used for managing large memory areas, it would waste a lot of memory due to internal fragmentation. Hence a first fit allocator is used for large areas to ensure good memory utilization.

The advantage of using the buddy system for small areas is that it would merge some of the free areas. This way the system may stay in operation longer before it reaches a state where it cannot satisfy a request.

Problem 14: For the fifty percent rule to be applicable, the number of free areas of memory should increase by 1, decrease by 1, or stay the same, when a memory area is freed. Hence the fifty percent rule does not apply to the buddy system allocator, the powers-of-two allocator, and the slab allocator.

Problem 15: Buddy or powers-of-two allocators, or any of their variants, would not be suitable for allocating areas of standard sizes because these allocators always allocate a block whose size is a power of 2. The slab allocator would be a good choice for allocating standard sized areas.

The requirement of no internal fragmentation for areas of other sizes restricts the choice to a first fit or best fit/worst fit allocator. A first fit allocator would be the obvious choice because it incurs a lower execution overhead and performs better in practice than the best fit allocator.

Problem 16: To start with, only one block of 64K bytes exists. Since slack is not supposed to be < 0, this block must be a globally free block. Hence slack is 0 and the class of 64K byte blocks is in the accelerated state. Slack is 0 for every other class of blocks, hence these classes are also in the accelerated state. When blocks of sizes 2K, 11K, 120 and 20K bytes are allocated, blocks of sizes 8K, 4K, 1K, 512, 256 and 128 bytes are free. These blocks are marked as globally free. Now, slack is again 0 for these classes of blocks and the classes continue to be in the accelerated state. N = 1 for the class of blocks with size 128 bytes. Hence this class of blocks is in the reclaiming state.

When the block of 128 bytes is freed, it is considered to be globally free and it is merged with its buddy. The new block of 256 bytes is considered to be globally free. The state of the class remains accelerated and the block is merged with its buddy. Similar actions are performed for classes of blocks with sizes 256, 512 and 1K bytes.

Problem 17: No split or coalesce actions take place during its execution. Since at least one free block of each size is available, the block sizes allocated are identical to those in Problem 10. When the block of 128 bytes is released, it is simply added to the free list of blocks of 128 bytes.

Problem 18: (a) True because they need to relocate themselves during execution. (b) Not true. Linkers perform two functions—relocation and linking. The first function is performed by a self-relocating program itself. However, linking would still have

to be performed by a linker.

Problem 19: The best way to achieve it in the self-relocating program is by using the subroutine call instruction:

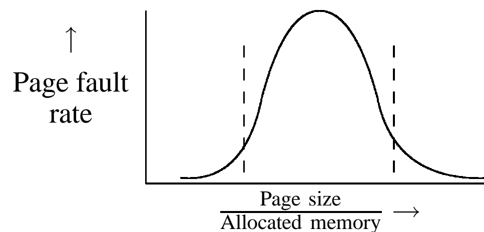
BSUB l_1, AREG

The BSUB instruction branches to the subroutine which starts on the instruction with label l_1 and loads the return address, i.e., address of the next instruction in AREG. This way, the program obtains information about the load time address of one of its instructions. It uses this information for relocating itself. The subroutine has a null body—it merely returns to the calling program.

Chapter 6

Virtual Memory

Problem 1: This problem can be discussed at great length in a class or tutorial to advantage.



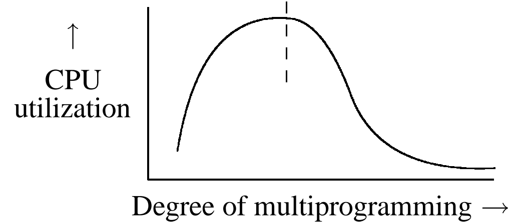
Let S be the size of a process, s the size of a page, and a the size of memory allocated to the process. For small values of s, the nature of the curve can be explained as discussed in Section 6.2.2.1. When page sizes are small, a large number of pages exist in memory. Hence proximity regions of several previous instructions are in memory, and the page fault rate is low. When the page size is increased, fewer pages fit in memory. So fewer proximity regions are now covered and more page faults occur. Hence the page fault rate rises. As the page size becomes very large, the current locality of a process can fit into fewer pages, hence the nature of the curve depends on whether these pages can reside in memory simultaneously. If so, the page fault rate would plummet as shown in the graph; otherwise, the page fault rate would increase as the page size is increased. For example, for the process segment

```
for (i = 1, i<200, i++)
      a[i] = b[i] + c[i] + d[i] + e[i]
```

the current locality is spread across 6 pages if the code and the arrays exist in different pages. For $s = \frac{S}{6}$, and $a < S$, the page fault rate could be significant since fewer than 6 pages exist in memory. Also, when s is small, a very small page may cover only a small part of the proximity regions of interest. Hence the page fault rate would rise as page size is decreased. From these arguments it appears that the curve may be

applicable only in the region enclosed by the dashed lines.

Problem 2:



The left half of the curve can be explained as in the discussion of variation of throughput with the degree of multiprogramming in Section 2.5.3.1. All processes have sufficient memory allocated to them, hence throughput rises with an increase in the degree of multiprogramming. However, the CPU utilization peaks and drops off as the degree of multiprogramming increases because processes do not have sufficient amount of memory. Eventually, the system enters the region of thrashing, so CPU utilization nosedives.

Problem 3: Let TLB contain only pairs of the form (page #, frame #) as shown in Figure 6.8. Let an entry for page p_i be made in the TLB while process P_1 is in operation. Let process P_1 be preempted and process P_2 be scheduled. Now, if P_2 refers to page p_i in its logical address space, the VM hardware would use contents of the TLB entry of p_i, which was made while P_1 was in operation, to complete the address translation. This way, process P_2 could access code or data of P_1. This can be prevented in several ways. Each TLB entry can be made to contain a triple (process id, page #, frame #). Alternatively, the TLB can be flushed whenever scheduling is performed.

Problem 4: Following Section 6.2.3.1, $pr_2 = 0.85$, $pr_1 = 0.98$, $t_a = 100$ nanoseconds, $t_t = 10$ nanoseconds, $t_{pfh} = 2$ msec. Hence effective memory access time

$$= 0.85 \times (10+100) + (0.98\text{-}0.85) \times (10+200)$$
$$+ (1-0.98) \times (10+100+2\times10^6+10+200) \text{ nanosec}$$
$$= 93.5 + 27.3 + 0.02 \times 2{,}000{,}320 = 120.8 + 40{,}006 \text{ nanosec} = 40.1 \text{ microsecond}$$

Problem 5: We first develop the formula for computing the effective access time for a two-level page table. Following Problem 4, access to a page in the first level page table (i.e., access to a PT page) consumes 40.1 microseconds. The PT page contains the entry for the page to be accessed. During access to this page, the same probabilities concerning its being present in memory and its entry being present in the TLB apply, hence this access would consume another 40.1 microseconds. Thus, the effective access time when a two-level page table is used is 80.2 microseconds. Effective access times for three and four-level page tables are 120.3 and 160.4 microseconds, respectively.

Problem 6: Approach (a) can be implemented by disabling address translation when the CPU is executing kernel instructions. However, approach (a) is not feasible be-

cause it ties up a large amount of memory. Approach (b) is practical and is used in several OSs. Windows 2000 uses approach (c).

Problem 7: A multi-dimensioned array is typically accessed in nested loops, each loop controlling one subscript in an array reference. The innermost loop controls the consecutive array references made during execution. Hence this loop should be such that these references lie in a few pages, possibly in a single page. Consider a program

```
for (i=1;i<100;i++)
    for (j=1;j<100;j++)
        x = a[i][j] ...
```

Successive references in this program will therefore be to elements in same row of the array. These elements should lie in the same page. For languages that allocate arrays in column-major manner, the array elements accessed in consecutive array references will be located in different pages, leading to a higher page fault frequency. In such cases, the page fault frequency can be reduced by enclosing the loop for `i` inside the loop for `j`.

Problem 8: The trick is that a page that will be removed by LRU should be accessed again immediately in the page reference string. Accordingly, the page reference string $1,2,3,4,5,6,1,2,3,6$ will suffice.

Problem 9: When some pages are frequently referenced but others are not, the second-chance algorithm favors the frequently referenced pages. It behaves like the FIFO replacement algorithm if reference bits of all pages are 1. So differences between FIFO and the second-chance algorithm arise when the second-chance algorithm passes over a page for replacement. It will do worse than FIFO for a page reference string if such an action leads to more page faults. As an example, consider a process that has been allocated 4 page frames and has the reference string $1,1,2,3,4,5,2,3,4$. Use of the second-chance algorithm leads to 8 page faults because all page references except the second reference to page 1 cause page faults. FIFO replacement causes only 5 page faults because none of the second references to a page cause a page fault.

Problem 10: (a) d, (b) r.

Problem 11: The proposed page replacement policy replaces the farthest referenced page at every page fault. Let P denote this policy. We show its optimality by showing that there is an optimal policy that makes exactly the same page replacement decisions as P and hence has the same number of page faults.

We do this by induction. Let O be an optimal policy that makes exactly the same page replacement decisions as P for the prefix $r_1, r_2, .., r_i$ of a page reference string $r_1, r_2, ..., r_n$. The base case is for $i = 0$, i.e., before a process is initiated. No page references have been processed and no pages of the process exist in memory, hence P and O behave identically for the first i page references and the contents of memory are the same after i page references. Consider the page reference r_{i+1}, which

refers to, say, page s. If page s is in memory, both P and O will behave the same. Let reference to page s generate a page fault. Suppose p is the page removed from memory by P and q is the page removed from memory by O.

Consider a new policy O' which removes page p at reference $i+1$ and retains page q. Note that the contents of memory are identical in O and O', except for pages p, q. Let $j > i+1$ be the earliest time at which one of the following events occurs:

1. Policy O removes page p from memory
2. Page p is referenced again, i.e., r_j is a reference to page p
3. No more references remain in the page reference string.

At time j, we make the contents of memory identical under policies O and O' as follows:
Case 1: Since page p was not referenced again until this point, under policy O' memory contains a page which is not in memory under policy O. Remove it from memory, making memory contents under the two policies identical. (This page may not be page q if q was referenced again).
Case 2: Again there is a page in memory under O' which is not in memory under O at this time (since O retained p in memory but O' did not). Replace this page by p. Now the memory contents are identical.
Case 3: Neither of pages p, q is referenced again. In this case, the number of page faults generated by both O and O' will be identical.

Now we claim that the page-faults under policy O' cannot be more than the page-faults under O. A page-fault can occur under O' but not under O only if page p is referenced, and only for the first reference to p after $i+1$. The memory contents are made identical at this time. So there can be at most one such fault. But by the choice of p, q must have been referenced at least once before time j. Now the first reference to q will cause a page fault under O but not under O'. Therefore there are at least as many faults under policy O as under O'. Hence O' is an optimal policy that makes exactly the same page replacement decisions as P for all references upto $i+1$.

When none of the pages are modified, a page replacement involves only a page-in operation; however, replacement of a modified page involves a page-out operation as well. Since the 'preempt farthest reference' policy does not differentiate between modified and unmodified pages, it does not lead to the minimum number of page-in and page-out operations.

Problem 12: For an algorithm possessing the stack property, $\{P_i\}_n^k \subseteq \{P_i\}_m^k$ for $n < m$. Hence the number of page faults when m page frames are allocated to a process cannot exceed the number of page faults when of n page frames are allocated to it.

Problem 13: If LRU does not possess the stack property, $\{P_i\}_n^k \not\subseteq \{P_i\}_m^k$ for $n < m$. However, this is a contradiction given the notion of 'least recently used'.

Problem 14: Unlike other page replacement policies, optimal page replacement does

not give a unique sequence of page replacement decisions. In other words, for a given reference string and memory allocation, more than one sequence of page replacement decisions may be optimal. For example, consider

(a) 4*,3*,2*,1*,1,1,2,2,3,3,3,3,4*,4,4,4	$4 \to 1, 2 \to 4$	$\{P_i\}_3^{13} = \{1,3,4\}$
(b) 4*,3*,2*,1*,1,1,2,2,3,3,3,3,4*,4,4,4	$4 \to 1, 1 \to 4$	$\{P_i\}_3^{13} = \{2,3,4\}$
(c) 4*,3*,2*,1*,1,1,2,2,3*,3,3,3,4*,4,4,4	$4 \to 2, 3 \to 1$	$\{P_i\}_2^{13} = \{1,4\}$
	$2 \to 3, 3 \to 4$	

where the '*' in 1* indicates that a page fault occurred during the reference to page 1, and $4 \to 1$ indicates that page 4 is replaced by page 1. (a) and (b) are two possible results of optimal page replacement with $n = 3$ since they both suffer 5 page faults. (c) depicts optimal page replacement for $n = 2$. Since several optimal replacements are possible for any value of n, comparison of arbitrarily chosen optimal replacements for $n = 3$ and $n = 2$ may not exhibit the stack property. For example, (b) and (c) violate the stack property. However, for different values of n, there also exist optimal page replacement decisions that exhibit the stack property, e.g., (a) and (c). It should be possible to develop a formal proof of this property.

The clock algorithm does not possess the stack property because it makes an arbitrary decision when all pages have their reference bits *on*.

Problem 15: (a) Let S_1^k and S_2^k be the set of pages in memory after the kth page reference when the window sizes are Δ_1 and Δ_2, respectively. From the definition of a working set, $S_1^k \subseteq S_2^k$ for all k. Hence $pfr_1 \geq pfr_2$ for all values of k.
(b) The memory allocation for a process is re-determined when the working set is recomputed, i.e., every n instructions. During the next n instructions, page replacement will be performed within the memory allocated to a process. Hence properties of the page replacement algorithm will determine whether $pfr_1 \geq pfr_2$ (e.g., $pfr_1 \geq pfr_2$ if LRU page replacement is used).

Problem 16: To reduce the degree of multiprogramming, the virtual memory handler selects a process for suspension. It now frees the page frames occupied by pages of the to-be-suspended process, and mark these pages as 'not resident in memory'. The freed page frames are distributed among processes whose recently computed working set sizes are larger than their present allocations. The virtual memory handler would use page tables and the free frames list, and a table to maintain information about working sets of processes.

Problem 18: These changes occur due to the nature of the computation in a process. Consider a program containing two loops. The first loop performs heavy numerical analysis of the values in a small array, whereas the second loop simply sums up the elements in a large array. The number of distinct pages referenced in the first loop is small, so the process has a small working set while executing the first loop. The process refers to a large number of pages in the second loop, so it has a large working

set while executing the second loop.

Problem 19: Reduction in the memory allocated to a process would lead to a higher page fault rate. It would lead to loading of more processes, and further reduction of the memory allocated to a process. This trend will culminate in thrashing.

Problem 20: The working set allocator determines the working set of a process periodically. Let the working set of a process P_i determined at time t_1 have n_1 pages in it. The allocator now sets the allocation for P_i at n_1 page frames and keeps it at n_1 page frames until it determines the working set of P_i once again, say, at time t_2. The page fault rate of P_i would increase sharply if, during the interval $t_1 - t_2$, its 'actual' working set size increases to a value that is much larger than n_1. Thrashing would arise if all processes face this situation simultaneously. However, the next time the working set of a process is determined, its memory allocation is adjusted accordingly (and the degree of multiprogramming is varied if necessary). Hence thrashing would be short-lived.

Problem 21: Compared to the one-handed clock algorithm, the two-handed clock algorithm uses more recent information about accesses to a page. Hence it is likely to make a better-quality decision about page replacement. Consider pages P_i and P_j whose reference bits were set to 'off' recently by the one-handed clock algorithm. If page P_i is not referenced for some time, and then referenced a few times, whereas page P_j is referenced a few times and then not referenced for some time, the one-handed clock algorithm will not be able to differentiate between them, but the two-handed clock algorithm will be able to.

Problem 22: The Windows case study in Section 6.9 contains some interesting points concerning shared pages. However, we will consider the problem in a concept-based manner. (a) The page fault arises because the page is marked 'not in memory' in the page table of a process. However, the page may be a shared page that exists in memory due to its use by some other process. If so, the virtual memory handler should simply 'connect' the existing copy of the page to the process that faced the page fault. If not, it should perform a page-in operation. If a page-out operation is needed to facilitate loading of the page, the virtual memory handler should mark the page being replaced as 'not in memory' in page tables of all processes that were using it. (b) The page is marked 'not present' in the page table of the process; however, it may still be retained in memory due to use by other processes. The virtual memory handler may maintain a count of processes that have a particular page in their working sets. The count would be manipulated appropriately. The page would be considered for replacement only if its count becomes zero.

Problem 23: Protection in the logical address spaces can provide different access privileges to shared pages. Protection and page reference information is maintained in the page table. This arrangement complicates page replacement of shared pages. Protection in the physical address space does not suffer from this problem; however,

it cannot support different access privileges to processes sharing the pages.

Problem 24: It is easy to show that the rules are equivalent to those used by a working set allocator as follows: In parts (b) and (c), the least recently used page is the '$-w$' page. Removing or replacing it amounts to 'shifting' the working set window over the page reference string. Thus, these steps handle removal of a page from the working set. Step (d) handles addition of a page to the working set.

These rules can be used to show that the page fault rate may be higher for smaller values of w.

Problem 25: If segment and byte id's are numeric, sharing of segments has a restriction found in sharing of pages—a segment must occupy identical positions in the logical address spaces of all processes sharing it. This restriction is not necessary when segment and byte id's are symbolic.

Problem 26: (a) Shared pages should be treated separately. (b) Due to a change in the locality of a process, some segment referenced in the past may not be a part of the current locality of a process. Use of LRU within a segment has the drawback that pages of such segments would not be replaced. It can be eliminated by employing a working set for each individual segment, so that a segment which is not a part of the current locality would not have any of its pages in memory. When processes share a segment, page replacement for the shared segment should be performed using the union of working sets of all processes sharing the segment.

Problem 27: This statement is valid unless page replacement is performed locally within a segment.

Problem 28: This feature cannot be handled by the methods described in Section 6.2.4 because the page containing the address of the data area would not have been prefetched and I/O fixed in memory.

There is no difficulty in handling the simplified form of self-describing I/O since pages participating in an I/O operation (i.e., pages of the I/O area starting at address *bbb*) would be preloaded and an I/O fix would be put on them.

Problem 29: The temporary memory area exists in the logical address space of a process. When the process is initiated, the temporary area can be mapped into adjoining page frames and its pages can be I/O fixed in memory. When an I/O operation is to be initiated, the addresses involved in it can be 'translated' into addresses inside this area. It is not necessary to load the pages participating in the I/O operation into the memory and I/O fix them, which simplifies I/O management.

There are three drawbacks of this approach. First, memory has to be committed to the temporary area. Second, data has to be moved in memory, which would consume CPU time. Third, due to use of the temporary area for all I/O operations, in a multithreaded application only one thread of a process can be performing an I/O

operation at any time.

Problem 30: It is useful to see why this approach is attractive: The swap space is under control of the virtual memory handler, so it can optimize paging performance by organizing the swap space appropriately. If a code page is not loaded into the swap space, it would have to be accessed from the file system, which may be less efficient.

Copying a code page from the file system into the swap space incurs overhead; the optimization seeks to reduce the overhead by incurring it incrementally only for the code pages used during operation of a process. However, due to the incremental nature of the optimization, its per-page overhead may be higher than the per-page overhead of copying the entire code into the swap space before initiating a process. Hence, this optimization yields the desired benefits only if a small fraction of the code pages of the process are used during its operation; it may be counter-productive if the process uses most of its code pages during its operation. Details of how Unix and Windows implement this optimization can be found in relevant sections of the text.

Problem 31:

(a) Increase the degree of multiprogramming. Increasing the size of memory will also help. (b) Thrashing is the likely cause, so increasing the size of memory will help more than using a faster disk. (c) It may be useful to increase the degree of multiprogramming by adding some I/O-bound programs. (d) No single action is likely to improve the system performance.

Chapter 7

File Systems

Problem 1: Copies of a file may become inconsistent and stale information may be used accidentally. Hence it is better to set up links from every user's home directory to `data`. However, a user is more likely to delete an unwanted file than to delete a link to a file that is no longer required. Hence dangling references may arise when the owner of `data` deletes it.

Problem 2: Let us assume that exactly one disk operation is required to access a disk block, and that a disk block either contains exactly one record or the complete index. If the 10,000 records were stored in a sequential file, an average of 5000 disk operations would be required to access a record in the file. If the records are stored in an index sequential file, one disk operation is required to read the index (we assume that the index is read every time a record is to be accessed), and an average of 50 disk operations are required to access the required record. Thus, only 51 disk operations are required to access a record from the index sequential file.

Problem 3: The index sequential file of Figure 7.6 contains 6 records on each track. Let the average number of records in the overflow area of a track be *#overflow*. Following an analysis analogous to that in Problem 2, reading of a record will need searches in the higher level index and the track index followed by a search to locate the record in the track. If the higher level index and a track index fit into one disk block each, $(2+\frac{6+\#overflow}{2})$ disk operations are needed to access a record. Hence access efficiency for the file will improve if the file is rewritten as a new file that does not contain any overflow records.

Problem 4: Care should be taken that links do not give rise to dangling references when a file is deleted.

Problem 5: Several issues discussed in conjunction with contiguous memory allocation (see Chapter 5) arise when contiguous disk space is allocated to a file. A file may

outgrow the disk space allocated to it. To reduce its possibility, one may allocate a large disk area to a file. However, this approach leads to internal fragmentation without really eliminating the problem. Older operating systems, e.g., OS/360, allocated several *extents* to a file, where an extent is one contiguous disk area.

The Amoeba approach avoids the problem of internal fragmentation. Its drawback is the overhead in terms of CPU time and I/O time.

Problem 6: Yes. Linked allocation interferes with index sequential and direct file organization. Indexed allocation introduces an element of indirection; however, it does not introduce significant overhead.

Problem 7:

(a) The FMT contains pointers to 12 records. Each record can be read in 3 msec, and requires 5 msec of CPU processing. Records 13–5000 have to be accessed by first reading an index block to obtain the address of a record. Hence access to each of the records requires (3+3) msec, and its processing requires 5 msec. Hence elapsed time of the process is $= (3+5) \times 12 + (3+3+5) \times 4988$ msec $= 54.964$ seconds. (b) 12 records are accessed directly. Each index block contains 1024 records, so 1024 records can be accessed using an index block that is in memory. This is done 4 times. 892 records are accessed using the fifth index block. Hence elapsed time $= (3+5) \times 12 + 4 \times (3 + (3+5) \times 1024) + (3 + (3+5) \times 892) = 40.015$ seconds.

Problem 9: One disk operation is required to access a 'good' disk block. Three disk operations are required for a bad disk block: one to read the bad block, one to read the bad blocks table and one more to read the alternate disk block. The average number of disk blocks required to read a record $= 0.98 \times 1 + 0.02 \times 3$ disk operations $= 1.04$ disk operations. So bad blocks degrade the performance by 4 percent.

To minimize performance degradation, the bad blocks table can be kept in memory all the time. This would reduce the degradation to 2 percent. To completely eliminate the degradation due to bad blocks, the file system can first consult the table to check whether the block to be accessed is a bad block and accordingly access either the block or its alternate block.

Problem 10:

It is not enough to make a single validation check while opening a file; a check has to be made at every file operation to ensure that the operation is consistent with the file processing mode indicated in the 'open' statement. The best way to implement it is to add a new field in the FCB to record the file processing mode mentioned in the 'open' statement and check consistency of each file operation with this information.

Problem 11:

(a) These entries are not adequate if a relative filename is indicated. FCBs should be created for all directories in the path from the root to the current directory. (b) A new field can be added to each directory; it would contain the address of its parent

directory. If the size of a directory increases, the file system can use the pointer to access the parent directory and update the child directory's entry in it. This field also helps when a relative pathname is used to access a file.

Problem 12: (a) Cascaded mounting as described in the question should be feasible. Its implementation requires only minor changes in the algorithms used by the file system to (1) update a directory and (2) allocate disk space for a file in the file system in which it exists. Cascaded mounting is straightforward in a conventional, i.e., non-distributed OS; however, it is complex in a distributed system. Section 19.6.1 discusses mounting in the SUN network file system (NFS).

(b) Multiple mounting is easy to implement; however, it gives rise to many semantic issues. For example, some file X in a multiply mounted file system may be opened using two pathnames. Are these files to be treated as different files or copies of the same file? These semantics should be clearly documented. (See Section 7.9 for a discussion of file sharing.)

Problem 14: A full back-up records data in all files in the system at the time when the back-up is taken. An incremental back-up contains data of only those files that have been modified since the last full or incremental back-up. An audit trail contains copies of data that have been modified since the start of file processing by a process. It can be used in conjunction with full and incremental back-ups to restore a file to its state just prior to a failure, as follows: First restore all files in the system using the last full back-up. After that, restore all files whose contents are found in incremental back-ups taken since the last full back-up. Finally, process the audit trail for files that were being processed at the point of failure and update the files accordingly.

Problem 15: The stable storage technique protects data existing on the disk against loss due to faults. A file system can use the stable storage technique to protect that part of its control data which is stored on a disk. However, it alone is not adequate to ensure integrity of the file system because the file system's control data that exists in memory would be lost when a fault occurs. Hence, to benefit from use of the stable storage technique, the file system must store all of its control data on a disk using the stable storage technique.

Problem 16: This statement is correct.

Problem 17: If a failure occurs after Step 3, block d_1 would not have been updated. Thus, data pointed to by d_1 is not lost. However, mix-up in the data of two files can still occur as discussed in Section 7.10.1.

Problem 18: The states of files recorded in a full back-up should be mutually consistent; otherwise, mix-ups of files analogous to that discussed in Section 7.10.1 may occur. To ensure mutual consistency, states of files should be recorded at the *same* instant of time. It is not possible while file processing is in progress. Recording of

an incremental back-up faces an identical difficulty.

Problem 19: The file allocation table of Unix (see Figure 7.31) should be used appropriately.

Problem 20: (a) FCBs for *infile* and *outfile* would be created when a process is created. These files are assigned to the user's terminal by default. When a user redirects these files in a command through use of the operators '<' and '>', the FCB contents of *infile* or *outfile* are changed to point at the files mentioned in the command, and the old contents of FCB are saved. The files mentioned in the command are now opened. When the execution of the command completes, the Unix shell would close the files used in the command and restore old contents of the FCBs so that subsequent commands would use the files *infile* and *outfile*.

(b) At a '>>' command, the FCB contents are changed as mentioned in (a). In addition, the FCB field containing 'address of the next record to be processed' is set to point at the record following the last record in the file.

Problem 21: Recall the structure of the Unix file system's free list from Section 7.12.3. When a file is deleted, all disk blocks whose addresses appear in its file allocation table, which we call FMT in our conceptual view, should be entered in the free list. Each block in the Unix free list is an index block, which uses a single level of indirection. Hence double and triple indirection blocks should be 'straightened out' into lists of single indirection blocks while entering into the free list.

Problem 22: This statement is substantially true. In a conventional OS, user processes request services of the kernel through system calls, and the kernel executes the appropriate event handler—essentially, some kernel process executes the necessary actions. In Unix, when a user process makes a system call, the user process itself proceeds to execute the event handling code. Hence several processes could be executing kernel code. Unix avoids interference between these processes by using locks on crucial data structures. A process about to use a data structure waits until it can set the lock. It resets the lock when it finishes using the data structure.

Data structures do not need to be locked in a conventional OS if a system process performing event handling is guaranteed to finish its processing without being interrupted. It can be achieved by disabling the interrupts when a kernel process is executing; however, in this approach the kernel cannot respond to high priority interrupts immediately. An alternative arrangement is as follows: When an event occurs, the event handler can check whether a similar event is under processing. If so, it can enter the new event into a request queue and defer to the event processing in progress. The queued request would be handled after the current event's handling is completed.

Chapter 8

Security and Protection

Problem 1: Tampering, stealing and denial of service cannot be prevented by encryption. In most cases, tampering can be detected because decryption of a tampered encrypted message does not yield an intelligible output; however, tampering itself is not prevented. Stealing and denial of service cannot be prevented for obvious reasons.

Problem 2: Encryption can ensure secrecy and integrity of data, but not its privacy. (Again integrity is ensured only if tampering can be detected, which is not the case for unknown/arbitrary data unless a message digest of data is maintained along with the data (see Section 8.6.1)).

Problem 3: Execution of a program whose setuid bit is 'on' is aborted if, for any file accessed by the program, the program owner's access privileges exceed those of the user who executes the program.

Problem 4: For high security, it is not a good idea to commit a function D_k, even if feasible, to the code in the OS; actually, such a function should not even exist. Hence the validity check is performed by encrypting the password presented by a user with the stored value of E_k(password) of the user.

Problem 5: Authentication yields the user id of the user. When a user process tries to access a file, the system uses the user id of the user to find whether the user has the necessary access privileges for the file. (For efficiency reasons, authentication is performed when a user logs in, rather than when a process created by the user tries to open a file.)

Problem 6: See Problem 10 of Chapter 7.

Problem 7: This scheme does not facilitate withdrawal of an access privilege easily—either k or both E and k would have to be changed, and their new values would have

to be provided to users who are to continue holding of an access privilege. It also provides a coarse granularity of protection; however, it is of a different kind than the coarse granularity discussed in this chapter: All users who are authorized to access a file may do so in exactly the same manner, i.e., it is not possible to provide a read access to one user and a read-write (i.e., modify) access to another user.

Problem 8: Providing access privileges to a set of users by sharing a password has the following drawbacks: It is cumbersome to provide different kinds of access privileges to users. For example, the owner of a file can associate three passwords with a file—one provides a read access privilege, another provides a write privilege and the third one provides an execute privilege. The file owner has to pass between one and three passwords to other users to let them use the file. A process created by a user has to present an appropriate password to the file system when it wishes to perform a file operation.

This scheme shares the problems faced by capability systems in revoking access privileges granted to some users: Revocation of the access privilege granted to a user is possible only by changing the password(s) associated with a file and distributing new passwords to other users.

Problem 9: When a system uses C-lists, the kernel can search through the C-lists of users to find which users hold a capability for a specific object. An analogous procedure can be used if capability segments are used. In a tagged architecture, it is cumbersome for the kernel to find which users hold a capability for an object—it would have to search through the entire address space of a user to find objects with the tag ‘capability’, and then check the object id to determine whether the object is the capability for the desired object.

When software capabilities are used, it is impossible to perform a capability review by searching, because a capability is indistinguishable from any other object. If capability review is important, the system’s design would have to provide for it. To start with, an object manager can save a copy of every capability it creates for a user. The kernel would now have to track how capabilities are passed to other processes. However, when a review is to be performed, the kernel would still not be able to know which of the passed capabilities have since been destroyed.

Problem 10: This system would be vulnerable to unauthorized use of capabilities if it permits use of a capability that simply matches some capability saved by it. It is so because anyone can create a bit string that matches the owner capability for an object and pass it to the system for verification. Since it will match an owner capability, the system would permit its use! This weakness is plugged by putting a random number in every capability. This idea would make it impossible to fabricate a valid capability.

To revoke a capability, the random number in the copy of the capability stored in the object manager is simply changed.

Problem 11: When capability-based addressing is used, an instruction indicates the

address of a capability, either in the C-list or in a capability segment. The CPU uses the capability as follows: It first loads the capability in some hardware register, then interprets the capability and accesses the object indicated in it in accordance with the instruction. Capability registers or a capability cache is used to speed up this process of accessing and interpreting a capability. The capability register is like a CPU register; its use is under control of a process. The process can load a capability in some capability register, say cr_i, and use the capability register to designate the capability. It is most effective if an object is likely to be accessed frequently. A capability cache would provide similar benefits.

Problem 12: A pair (node id, object id) can be used instead of simply an object id.

Problem 13: (a) An inode contains the user and group ids of a file's owner. (b) /etc/group is an ascii file containing group name, group password, a numerical group id and the list of users belonging to the group. To check whether a user belongs to the group to which a file's owner belongs, the system can check whether the user's id exists in the list of users belonging to the file owner's group.

Chapter 9

Process Synchronization

Problem 1: The upper bound is 100. The lower bound is not so obvious; it is only 2! The explanation is as follows: The statement `sum:=sum+1;` is implemented by a `load-add-store` sequence of instructions. Let one of the processes, say process P_1, perform a `load` and get delayed for a long time (presumably because its time slice elapses), and let process P_2 finish 49 iterations of its loop. The value of `sum` is now 49. Now, let process P_2 get delayed and let process P_1 finish its first iteration. The value of `sum` is now 1. Let P_2 start on its 50th iteration, perform a `load` instruction and get delayed (so it has a 1 in a CPU register). Now, let P_1 finish all iterations of its loop. The value of `sum` is now 50. Now let process P_2 finish its last iteration. The value of `sum` is now 2!

Problem 3: If $turn = 2$, P_1 is trying to enter CS and P_2 is not interested in entering CS, c_2 will be 1. Hence P_1 enters CS straightaway.

Problem 4: (a) Let a deadlock exist. Hence from P_1's code $c_1 = 0$ and $turn = 2$. From P_2's code $c_2 = 0$ and $turn = 1$; however, this is a contradiction. Hence a deadlock cannot exist.
(b) Let a livelock exist, i.e., both processes defer to one another indefinitely. It implies that both processes are stuck in their **while** loops. However, variable *turn* either has the value 1 or 2. Accordingly, one of the processes would change its status flag, i.e., either c_1 or c_2, to 1, hence the other process would exit from its **while** loop. This is a contradiction, hence a livelock cannot exist.

Problem 5: The bounded wait condition can be violated only if both processes are simultaneously interested in entering CS and one of them enters CS, exits, again wishes to enter CS and enters ahead of the other process arbitrarily many times. However, when both processes wish to enter, the value of *turn* decides which process would get in. When this process exits its CS, it changes its own flag to *false*, hence the other process, which was stuck in its **while** loop, can enter CS irrespective of the

value of *turn*. Thus, a process can enter the CS only once ahead of the other process if the other process is also interested in entering the CS. Hence the bounded wait condition is satisfied.

Problem 6: This scheme has two problems: It suffers from lack of mutual exclusion and from violation of the progress condition. Both *flag*[0] and *flag*[1] are *false* initially. The first process that wishes to enter the CS can do so straightaway because both flags are *false*. If the other process wishes to enter while the first process is still inside its CS, it will also enter its CS. Thus, both processes can enter their CSs simultaneously.

Now, let both processes exit their CSs. Both flags will be set to *true*. Let the value of *turn* be *i*, where *i* is either 0 or 1, and let process P[1-*i*] wish to enter its CS once again. Now, it will loop until the other process wishes to use its CS, enters, and exists its CS; which violates the progress condition.

Problem 7: We have starvation as follows: Let one of the processes, say process P_0, wish to enter its CS. It will set *flag*[0] to *true*, *turn* to 1, and enter the **while** loop. It will iterate indefinitely in its **while** loop because its governing condition is *true*. If P_1 also wishes to enter its CS, it will set *flag*[1] to *true* and *turn* to 0. The change in the value of *turn* does not affect P_0, because *flag*[1] has been changed to *true*. Hence P_0 continues to loop. Process P_1 will also loop indefinitely because *flag*[0] is *true*.

Problem 8: After completing use of a CS, a process sets *turn* to the next process in modulo order that is interested in entering its CS (see the **while** loop after CS). If no other process is interested, it would encounter its own flag and stop looking for another process wishing to enter CS. It would now exit the **while** loop and set *turn* to point at itself. If it wishes to enter CS before any other process, it would be able to enter straightaway. If some other process wishes to enter its CS, it would execute the **repeat** loop in which it would check whether any other process ahead of it in the modulo order wishes to enter its CS. If not, it would enter its own CS.

Problem 9: Processes wishing to enter their CSs choose a number using the *max* function and enter their CSs in the order imposed by the $<$ relation on the pair (*number*, *process id*). However, preemption of a process while it is choosing a number may interfere with it as follows: Process P_j starts choosing a number before P_i, but it is preempted after choosing a number and before putting it in *number*$[j]$. When P_i chooses a number, it obtains the same number. If P_i went on to perform the second **while** loop before P_j put its number in *number*$[j]$, P_i might enter the CS even if its pair does not satisfy the $<$ relation with P_j's pair, which would be incorrect. The first **while** loop prevents it from happening.

Problem 10: In this solution, an actual produce or consume action takes place inside a CS. This feature permits only one produce or consume action to be in progress. When many producers and consumers exist, this feature can be changed through the following provisions: A producer first produces a record in a temporary area

and then enters the **while** loop to find an empty buffer. When it finds one, it puts the produced record into the buffer. Analogous provisions are made in a consumer. This way produce and consume actions take place outside a CS, which permits many producers and consumers to produce and consume simultaneously.

Problem 11: Consider a situation analogous to that depicted in Table 9.3.

Problem 12: Semaphore *empty* is initialized to n, which provides an n-way concurrency in the produce actions by producers. However, a producer process updates *prod_ptr* only after producing. It is therefore possible for another producer process to start producing in the same buffer in which the previous producer has produced, or is producing. This will lead to loss of information produced by producer processes. Putting the statement *buffer*[*prod_ptr*] := ... in a critical section would solve this problem, but it would disallow any concurrency between the produce actions of producer processes. So rewrite the **repeat** loop in the producer as:

```
{ Produce information in some variable named xyz }
wait(empty);
wait(prod_mutex);
my_ptr := prod_ptr;
prod_ptr := prod_ptr+1 mod n;
signal(prod_mutex);
buffer[my_ptr] := xyz;
signal(full);
```

where *prod_mutex* is a binary semaphore initialized to 1. Analogous changes are made in the code for a consumer process.

Problem 13: Following the hint given in the problem itself, define a binary semaphore *rw_permission* that is initialized to 1. A reader process checks whether reading is already in progress. If so, it goes on to read straightaway; otherwise, it performs a *wait*(*rw_permission*). When it finishes reading, it checks whether any other readers are engaged in reading. If not, it performs a *signal*(*rw_permission*) which would wake up a writer process waiting to write, if any, or enable reading and writing in future. A writer process uses semaphore *rw_permission* instead of the semaphore *writing*. It enters a CS on *mutex*, increments *totwrite*, and exits the CS. It now performs a *wait* on *rw_permission*, writes, and performs a *signal* on *rw_permission*. (Other code in the write process, including the **while** loop, should be deleted.)

Problem 14: Two changes are needed to provide writer priority: (1) A reader starts reading if no writer is writing. This is to be changed such that a reader starts reading only when no writer is either writing or waiting. Hence the first **if** statement should be changed to use the condition *totwrite* = 0. (2) After finishing a write operation, the writer should check whether a writer is waiting to write. If so, it must activate the writer. Else it must activate waiting readers, if any. For this purpose, the **while** and the second **if** statements should be interchanged—in fact, the **while** should be put in

the **else** clause of the **if**. The **if** statement need not check for *runread* = 0, it should simply check whether $totwrite > runwrite$.

Problem 15: Let a writer be writing when a reader wishes to start reading. The reader will perform a *wait*(*reading*) inside the CS on *mutex*, and get blocked. When the writer finishes, it will perform a *wait*(*mutex*) in order to update relevant counters and decide whether to activate a writer or reader(s). However, it will be blocked on the *wait* operation since a reader is inside the CS. Now we have a deadlock situation.

Problem 16: The *wait* operation on semaphores does not specify the order in which processes blocked on the operation will be activated. This feature can lead to a situation in which a process remains blocked on a *wait* operation while some other process executes *wait* operations, gets blocked and gets activated an arbitrary number of times. It leads to a violation of the bounded wait property. Activation of processes blocked on a *wait* operation in a FIFO order would ensure that this will not happen.

Problem 17: The key point is implementation of FIFO behavior. (a) It is best to let each process use a semaphore for self-scheduling (see Section 9.8.3). When a process P_i makes a resource request, the *request_resource* procedure would either perform a *signal* operation on the scheduling semaphore of P_i, or put the name of P_i's scheduling semaphore in a queue of pending requests. The release operation would perform a *signal* operation on the first semaphore in the queue. (b) A similar scheme can be used with conditional critical regions, the only difference being use of an **await** condition on a scheduling flag called *proceed*[*i*] for each process P_i. The flag would be manipulated by the request and release procedures. These procedures would also manipulate the queue of pending requests.

Problem 18: The changes are analogous to those in Problem 14. Introduce a count *totwrite* and statements to increment and decrement its value. A reader should be allowed to read only if *totwrite* = 0. A writer should wait on the condition (*run_read* = 0 **and** *run_write* = 0). However, the condition *run_write* = 0 is redundant because writing is performed in the CCR on variable *read_write*, so the condition is trivially true when any writer evaluates it. After performing a write, a writer should enable a waiting writer, if present. If not, it should enable waiting readers, if any.

Problem 19: In Figure 9.18, the actions to check availability of forks and to lift the forks are performed in a CS. It ensures that if a philosopher finds that both forks are available, she will lift both of them before any other philosopher is permitted to check availability of forks, which avoids a race condition involving neighboring philosopher processes. If the action of lifting the forks is moved outside the **while** loop, the following situation could arise: Let forks put between philosophers P_{i-1}, P_i, P_{i+1}, and P_{i+2} be available. Process P_i finds that two forks are available to it and exits its **while** loop. However, before it can pick up the forks, process P_{i+1} also finds two forks to be available to it and decides to pick them. Now, we have a race condition on the

fork put between processes P_i and P_{i+1}.

Problem 20: Deadlocks do not arise if *some* philosopher is able to pick both left and right forks. (a) If all philosophers pick up their left forks, the philosopher who has a vacant seat on her right will be able to pick up her right fork as well. When she finishes eating, the philosopher on her left will be able to pick up her right fork and so on, hence a deadlock will not arise. Analogously, one philosopher will be able to eat even when all philosophers first pick up their right forks. (b) Yes. Presumably a left-handed philosopher first picks up the left fork and then the right fork, while a right-handed philosopher does the converse. Analogout to part (a), it ensures that some philosopher will be able to eat.

Problem 21: (a) Deadlocks can be prevented by requiring a philospher process to lift both forks simultaneously. The condition in **await** statements can take care of this. However, this solution permits only one philosopher to lift the forks at any time; philosophers cannot do it simultaneously. This problem can be alleviated by making each philosopher process do the eating outside the CCR.
(b) It is not easy to implement lifting of both forks simultaneously because a blocked philosopher process should be activated only when both forks become available. Instead, we can permit philosopher processes to pick up one fork at a time and use resource ordering to prevent deadlocks. Let all forks be numbered in an anticlockwise direction. Now for most philosophers the left fork has a smaller number than the right fork. Only one philosopher has a smaller numbered right fork. Let each philosopher know the numbers of his/her left and right fork and ask for them in ascending order by number. Each fork can be implemented as a condition variable. Following Problem 20, Part (b), no deadlocks can arise.

Problem 22: A consumer waits on *buff_full* only if *full* is 0. Hence a producer needs to performs *buff_full*.**signal** after incrementing *full* only if the new value of *full* is 1. Similarly, a consumer should perform *buff_empty*.**signal** only if the new value of *full* is $n-1$.

Problem 23: From the code for a process, delete the statements to perform an I/O operation and invoke procedure *IO_complete*. Procedure *IO_request* of *Disk_Mon_type* contains the statement *proceed*[*proc_id*].**wait**. It should be followed by code that performs the I/O operation on the track *track_no*, which should be followed by the code that is present in procedure *IO_complete*. Procedure *IO_complete* is no longer needed.

Problem 24: (a) The reservations data would become data of the monitor. *Bookings* and *Cancellations* would be monitor procedures. (b) The monitor provides mutual exclusion between executions of its procedures; no additional synchronization would be necessary. Use of CR or CCR would yield a similar implementation.

Problem 25: The monitor should have a variable to hold the value n, an appropriate number of condition variables as explained in the following, and a data structure to

hold pending debits. It has two procedures named *debit* and *credit*. Every banking transaction is assumed to be a process—we call it a transaction process. A process invokes the procedures *credit* or *debit* with its own id and the amount to be credited or debited.

Procedure *debit* carries out a feasible debit and returns. It enters an infeasible debit in the pending debits structure and executes a **wait** statement on an appropriate condition variable. Procedure *credit* ignores a credit if $balance > n$. Otherwise, it performs the credit operation. Now, as many pending debits should be performed as possible. One approach to implementing it is to have a condition variable for each of the transaction processes and maintain a list of condition variables of those processes whose debits were delayed. After a credit operation, each of the blocked processes would be activated through a **signal** operation on its condition variable. It would check whether its debit can be performed and remove its condition variable from the linked list if it is the case. However, this approach needs a condition variable for each transaction, which is infeasible if the number of banking transactions is not known. To avoid this problem, the monitor could provide a finite set of condition variables, associate a condition variable dynamically with a transaction process if its debit is delayed, and store this association in its own data structures. An alternative approach is to use a single condition variable on which all transaction processes would wait, maintain a count of the number of processes blocked on it, and arrange to execute those many **signal** statements following a credit operation. In both approaches, a process that is activated would execute a **wait** statement once again if its debit still cannot be performed. It amounts to a busy wait situation. It can be avoided in the first approach by activating a process only when its debit operation can be performed; however, it cannot be avoided in the second approach.

Problem 26: We have one barber process and several customer processes. A deadlock arises if the barber goes to sleep and customers expect him to wake up and admit a customer, whereas the barber expects a customer to wake him. The barber process uses 3 procedures: *Check_and_admit_customer*, *finish_customer* and *sleep*. A customer process uses 2 procedures: *check_and_wake_barber* and *wait*. A monitor can be defined to contain these procedures and relevant data. Monitor procedures would use **wait** and **signal** statements to implement the necessary synchronization.

This is a very interesting problem; however, many of the complications vanish when a monitor is used.

Problem 27: This is a very interesting and important exercise for practice in concurrent programming. In fact, when I offer an undergraduate course in OS, I often give this exercise as a laboratory assignment. It brings out many interesting points concerning concurrency and deadlock possibilities.

The monitor for the interprocess communication system (minus the deadlock detection feature) is as follows :

Monitor
type

```
    msg_ptr = ↑ msg;
    txt = array [1..200] of character;
    msg = record
          text : txt;
          destination : integer;
          ptr : msg_ptr;
          end;
var
    message_pool : array [1..20] of msg;
    free_list : msg_ptr;
    wake_up : array [1..4] of condition;
    free_space : condition;
    in_messages : array [1..4] of msg_ptr;

Procedure entry send (msg_txt : txt; destination : integer);
begin
    if free_list = null then free_space.await;
    else
        {Remove a message unit from free_list}
        {Enter details of the message in the message unit}
        {Link the message unit to the list starting on in_messages[destination]}
        wake_up[destination].signal;
end;
Procedure entry receive (msg_area : txt; proc_id : integer);
begin
    if in_messages[proc_id] = null then wake_up[proc_id].await;
    else
        {Copy message from the message unit pointed by in_messages[proc_id]}
        {Delink the message unit from list starting on in_messages[proc_id]}
        {Link the message unit to free_list}
        free_space.signal;
end;
begin
    {Link all elements in message_pool to free_list}
    {Set all elements of in_messages array to null}
end.
```

Deadlock handling requires the program to check whether all processes are involved in indefinite waits. It becomes more interesting if some processes may complete before others. These features are not discussed here.

Problem 28: A 'wake me' request is entered in a requests data structure in a sorted order according to time. The earliest of these times is entered in a variable called *next_event_at*. The procedure handling the *elapsed_time* signal updates the time and

compares it with *next_event_at*. If the current time matches the time indicated in *next_event_at*, it wakes the process that had made the request.

Problem 29: Let process P_i invoke a procedure of monitor A. While this procedure is in execution P_i holds mutual exclusion over A. Let this procedure call a procedure of monitor B and get blocked on entry to B because some procedure of B invoked by some process P_j is active. Effectively P_i holds mutual exclusion over A and needs mutual exclusion over B. Let the procedure of monitor B that was invoked by P_j call a procedure of monitor A. It would be blocked on entry to A. Effectively P_j holds mutual exclusion over B and and needs mutual exclusion over A. This is a deadlock situation.

Problem 30: (a) Mutual exclusion over a monitor can be implemented using a binary semaphore. **wait** and **signal** operations would be performed on the semaphore when a monitor procedure is invoked and finishes its execution, respectively. (b) Monitor procedures are executed in a mutually exclusive manner, so why should they be re-entrant? The reason is that execution of a procedure would be suspended if it executes a **wait** statement. Some other process may now call it and commence its execution. Hence each monitor procedure should be reentrant. See Section 5.11.3.2 for details.

Problem 31: Rules for this readers–readers system are fairly obvious: Readers can concurrently read one part of data, however no reader can read the other part of data at the same time. When the last reader finishes reading one part of data, the other part of data can be loaded and all readers wishing to read the other part can be enabled.

The simplest way to implement this readers–readers system is to use the notion of the 'current' part of data. Any reader wishing to read it can commence reading straightaway. Other readers have to wait. When the last reader exits from the current part of data, the other part of data can become the 'current' part, if any readers are waiting for it. All blocked readers wishing to read it will be activated now. When no readers are active, the data required by a newly arriving process can be made the 'current' part and it can be allowed to read. One can adapt a solution of the 2-process CS problem from Section 9.2 for this purpose. However, it is simplest to use a monitor with two procedures called *start_read* and *end_read*. Procedure *start_read* blocks a reader that wishes to read a part of the data other than its 'current' part, while procedure *end_read* activates blocked readers when the last reader for the 'current' part departs.

Problem 32: Let the bridge be East–West. A procedure can control entry to the bridge from East, and another procedure can control entry from West. Similarly two procedures control exit at the West and East ends of the bridge. Procedures for entry from East and exit at West cooperate to maintain counts of vehicles waiting to enter, and vehicles traveling in the East–West direction. Similar arrangement exists for the West–East direction. Procedures for entry to the bridge check all this data and decide

whether the vehicle process that invoked it can start crossing the bridge.

Problem 33: An additional counter to count how many vehicles have started crossing the bridge in the current phase would be needed. Rules for its use are obvious.

Chapter 10

Message Passing

Problem 1: Lack of memory for creating either IMCB or ECB is an exceptional condition within the kernel itself. Following the discussion of kernel memory allocation in Section 5.10, the kernel can allocate additional memory to create these data structures. If it is not possible due to some reason, the kernel must reattempt creation of these data structures whenever it releases memory allocated to some other data structures. It is best to keep a list of unsatisfied requirements and check this list whenever a kernel data structure is released.

Problem 2: In asymmetric communication, the sender names the receiver but the receiver does not name the sender. This feature calls for several changes in the scheme of Figure 10.5. If a *receive* request cannot be satisfied straightaway, it anticipates a *send* to be performed by some process, hence an ECB is created for a *send*. However, identity of the prospective sender is not known at this point. Let process P_j now send to P_i. The kernel must relate this event to that mentioned in the ECB. The search for a matching ECB in Steps S_2 and S_3 and the creation of ECB in Step S_4 should be modified to incorporate this requirement. Analogous changes are needed in Steps R_2 and R_3.

Problem 3: This idea has to be handled with care. A process sends a message to its own mailbox to avoid getting blocked. If genuine messages do exist for it, its dummy message would be added at the end of the message queue. If the process sees the dummy message immediately, it should realize that there are no genuine messages waiting to be delivered to it; however, if it sees the dummy message sometime in future it should realize that the dummy message is dated and that genuine messages may exist further down in the message queue. A simple method to check whether a dummy message is dated is to number each dummy message sent to a mailbox and note the number in the last dummy message. Any dummy message with a smaller number is an old message that should be discarded. This change can be incorporated

in the reservations server.

Problem 4: *Send* and *receive* calls must mention the message queue ids. All steps of Figure 10.5 must be applied to IMCBs and ECBs for the message queue. Additional provision is required to create message queues.

Problem 5: A simple method is to create an ECB for the time-out event. The process that issues the *receive* request now waits on one of two events—the time-out event and the expected *send* event. The event that occurs earlier would activate the process. The event handling schematic should be modified to incorporate this feature—it should make use of the ECB of the event that occurs earlier, and discard the other ECB.

Problem 6: A deadlock can arise when each process is blocked either on a *receive* request or on a *send* request due to lack of memory to create an IMCB. Deadlock due to the latter cause can be avoided in various ways: One way is not to create an IMCB if a matching ECB exists. The message can now be transferred directly from P_j's memory to P_i's memory. This method also reduces the overhead of message passing by avoiding the need to copy a message twice—once into an IMCB and later into the memory area of the receiver.

Chapter 11

Deadlocks

Problem 1: See Sections 11.5 and 11.6.

Problem 2: The change implies that a process is permitted to ask for a unit of resource class R_k only if $rank_k \geq rank_i$. So a hold-and-wait condition can arise as follows: Let a system contain n units of a resource class R_k. Let processes P_1 and P_2 be allocated u_1 and u_2 units of R_k, respectively. Now, processes P_1 and P_2 request some more resource units of R_k that cannot be granted. It leads to a circular wait condition. Hence deadlocks can arise.

Problem 3: If the processes are allocated two resource units each, a deadlock would arise if each of them asks for one more resource unit. No deadlocks would arise if the system contains seven resource units because at least one process can be allocated its maximum requirement of 3 resource units. When this process completes, resources released by it can be allocated to the other processes, so all processes can complete.

Problem 4: A process P_i is permitted to wait only for an older process. Thus, it is not possible to have a circular wait-for relationship. Hence deadlocks cannot arise. Completion of every process in finite time can be shown as follows: The oldest process in the system is guaranteed any resource it needs, hence it is guaranteed to complete. Now, the second oldest process can complete, if not already done, and so on.

Problem 5: Let a system contain two *blocked* processes P_i and P_j and a sufficient number of free resource units such that either process P_i or process P_j can be allocated those resource units and made *running* during deadlock analysis. Let it be the case that the resource units should be first allocated to P_i and not to P_j, such that both processes can finish their operation. However, it is a contradiction because the system already has sufficient number of free resource units to satisfy the requirements of either process P_i or process P_j, and so it will have a larger number of free units when

either of them finishes. Hence the result of deadlock analysis cannot depend on the order in which processes are transferred from *Running* to *Finished*.

Problem 6: State $(6,1,2)$ is not safe because the outstanding requirements would be $(2,6,3)$ while only 1 unallocated unit exists in the system. Hence no process is guaranteed to finish. $(5,1,1)$ is a safe state because 3 unallocated units exist while the outstanding requirements are $(3,6,4)$. Hence processes can finish in the sequence P_1, P_2, P_3 or P_1, P_3, P_2.

Problem 7: (a) Yes. (b) Yes. (c) Yes. (d) No. (e) No.

Problem 8: (a) Yes. (b) (i) No. (ii) Yes. (iii) Yes.

Problem 9: If P_3's outstanding requirement is 2 units and the system has only 2 unallocated units, these units can be granted to P_3 because P_3 can complete. However, no process will be able to complete if the outstanding requirement of P_1 and P_2 is more than 2 units and 1 unit is allocated to one of them. Hence such an allocation would not be made. In effect, the 2 unallocated units would be 'reserved' for allocation to P_3 until P_3 finishes.

Problem 10: The system contains 8 and 11 resources of R_1 and R_2, respectively.

	Allocated		Maximum	
	R_1	R_2	R_1	R_2
Process P_1	2	3	5	5
Process P_2	3	5	4	5
Process P_3	2	2	2	3

If the (1,0) request by process P_1 is granted, processes can finish in the order P_3, P_2, P_1. However, if P_1 is further allocated (0,1) no process can finish. Hence P_1 would not be allocated (0,1). If P_1 makes a (0,1) request to start with and the request is granted, processes can finish in the order P_2, P_1, P_3. Hence the request is granted. (The trick is to let P_1 need more than (1,1) resources to finish, and have two other processes that require (1,0) and (0,1) resources to finish.)

Problem 11: The Banker's algorithm grants a request if the new allocation state would be safe; otherwise, it keeps the request pending until it can be granted. Thus, at every allocation, the algorithm ensures that there is at least one sequence of allocations through which each process in the system can be granted its maximum requirement and can complete. Given finite execution times of processes, each process in such a sequence must complete in finite time.

Problem 12: When a process makes a multiple request, none of the requested resources are allocated to it unless all requested resources can be allocated. However, the safety of allocating each of the requested resources individually does not ensure safety of allocating *all* of them. Hence this scheme is incorrect. The solution of

Problem 10 contains a counter-example.

Problem 13: (a) Under this condition, $\Sigma max_{i,s} \leq n$. Hence all processes are guaranteed their maximum requirement. (b) Similar to (a). (c) Let each of the n processes be allocated one unit less than its maximum requirement. Hence the number of allocated units is $\Sigma_i \; max_{i,s} - n$, which is $\leq exist_s + n - 1 - n$, i.e., $\leq exist_s - 1$. Hence 1 resource unit is available using which one of the processes can finish and release its resources. Now, at least one other process can finish, and so on until all processes have finished.

Problem 14: (a) No. (b) No. Consider $k = 2$ in Problem 3, where the system contains 6 resource units and 3 processes each requiring 3 units have been allocated 2 units each.

Problem 15: (a) Yes. Since a multiple-resource Banker's algorithm is used, the state must have been safe before the new process arrived, i.e., the four processes in the system can complete in finite time. The newly arrived process can surely be allocated all its required resources after the other processes complete and it, too, can complete. (b) Yes. Aborting of a process does not interfere with completion of other processes in the system.

Problem 16: A deadlock implies existence of a circular wait-for relationship. Let processes P_i and P_j wait for one another in the deadlock situation. When the processes are restarted, the deadlock will recur if processes make their resource requests in exactly the same order as before. However, it will not recur if one of them can obtain all the required resources. It can happen if operation of the processes is interleaved in a different manner; it will happen if the processes are executed one at a time!

Problem 17: Several deadlock possibilities exist. If all processes that have produced some printer output get blocked for disk space while producing more printer output, none of them can proceed and release the disk space allocated to it. Such a deadlock can be prevented by handing over spooled output to a printer before a process finishes (as in THE operating system—see Section 11.5). A wait-for relationship also develops if a process is blocked due to lack of disk space while creating a file and a few processes are blocked while creating printer output. A set of processes blocked while creating files also face indefinite waits; however, this situation cannot be called a deadlock because there is no reason to expect any process to delete a file—it is simply an indefinite wait. The kernel can resolve this situation by adding more disk space.

Problem 18: If the set *Blocked* is non-empty when Algorithm 11.1 finishes, processes included in the *Blocked* set satisfy the hold-and-wait and mutual wait-for conditions. Hence the algorithm does not detect a phantom deadlock. However, if processes are permitted to withdraw their resource requests using a time-out, the mutual wait-for condition is not indefinite. Detection of phantom deadlocks can be

prevented by including a process that has specified a time-out interval in a request to be in the *Running* set rather than in the *Blocked* set.

Problem 19: (a) A deadlock can arise if both gates have to be closed simultaneously for a train to pass, e.g., if length of a train exceeds the distance between gates. Let one gate be closed. If the traffic jam caused by it extends past the second crossing point, the second gate cannot be closed. This situation is a deadlock. There is no difference in the deadlock possibilities even if traffic is one way. However, a deadlock would not arise if both train and road traffic are one-way in the same direction, e.g., if both run from north to south, because the gates would naturally be closed in an order which would not cause a traffic jam that blocks the other gate.
(b) Deadlocks can be prevented in several ways: (i) Close both gates at once. (ii) Do not permit stalled road traffic to clutter any gate. (iii) Close one gate, allow train to pass it, then close the other gate and allow the train to pass it. (iv) If traffic is one-way, always first close the gate that is reached first by an approaching car. Then close the other gate.

Problem 20: The numbering of forks creates a situation in which either $n-1$ philosophers have a smaller numbered left fork but one philosopher has a smaller numbered right fork, or the converse. This situation eliminates deadlocks because every philosopher cannot pick up the left fork and wait for the right fork, or conversely.

Problem 21: Consider the RRAG of the system. (a) Process P_j must satisfy one of the two conditions: (i) it exists in the auxiliary set (AS), or (ii) some process P_k included in the knot set has at least two out-edges (by the definition of a knot, these out-edges are also involved in cycles), and all cycles in which P_k is involved are not broken when P_j is aborted. (b) Process P_i must be in the knot set (KS) and none of the processes in the knot set has more than one out-edge.

Problem 22: In an SISR system, no process or resource node has more than one out-edge. Under this condition, a resource knot has exactly one cycle in it, which is the necessary and sufficient condition for SISR systems. In an MISR system, a knot is a necessary and sufficient condition. Since a process node can have only one out-edge, a knot is also a resource knot and vice versa, so a resource knot is a necessary and sufficient condition for a deadlock. In an SIMR system, a cycle is a necessary and sufficient condition. Since a resource node has only one out-edge, a single cycle is a resource knot, hence a resource knot is a necessary and sufficient condition. In an MIMR system a resource knot is a necessary and sufficient condition.

Problem 23: See Section 11.7 for characterization of deadlocks in SISR, MISR and SIMR systems. In an MIMR system, a process may make a multiple request. Let each request be for a single instance resource class. In this case, the multiple request gives rise to AND edges in the WFG. From SIMR systems we know that a deadlock exists if any one of a set of AND edges is involved in a cycle. If one of the requested resource classes has multiple instances, the requesting process has a

set of OR edges pointing to each holder of that class. These edges collectively have an AND relationship with edges representing other requests in the multiple request. From deadlocks in MISR systems, we know that each edge in a set of OR edges must be involved in a cycle for a deadlock to exist. Thus, in general, a process is involved in a deadlock under two conditions: (a) if some out-edge that individually has an AND relationship with some other edges is involved in a cycle, or (b) if each out-edge in at least one set of OR edges is involved in a cycle.

Problem 24: A WFG based generic approach can be developed as follows: Let a process P_i be in a deadlock. WF_i is the set of processes to which process P_i has an out-edge. If out-edges of P_i have an AND relationship with one another, only one of them needs to be in a cycle for a deadlock to arise; however, several of them may be involved in cycles. *All* cycles involving them must be broken to eliminate the deadlock. It can be achieved by aborting process P_i, by aborting those of the processes in WF_i that are involved in cycles, or by aborting some other processes such that all cycles involving P_i are broken. If all out-edges of P_i have an OR relationship among them, each of them must be involved in a cycle, but only one of the cycles needs to be broken to eliminate the deadlock. Hence the general approach is as follows: Abort a process that is involved in the maximum number of cycles due to AND out-edges. Repeat this step if a deadlock persists.

Chapter 12

Implementation of File Operations

Problem 1: Buffering and blocking of records is beneficial for sequential files. It is also useful when index sequential files are used in the sequential mode. However, both the techniques are not useful for direct files or for random access to records in index sequential files because adjoining records are rarely accessed. In fact, for direct and index sequential files, blocking and buffering introduce unnecessary overhead.

Problem 2: (a) Typically update files are recorded on disk devices. There are two reasons for this. In tapes, it is not possible to align the start and end of a record's boundaries precisely when the record is rewritten. The only practical method to use a tape for an update file is to read the entire file and then write the entire updated file as a new file. Another reason for using disks is that records in update files are often processed randomly. (b) For reasons mentioned in the solution of Problem 1, blocking and buffering are useful only if the file would be accessed sequentially.

Problem 3: We assume that implementation of noncontiguous allocation in the file system either uses a linked allocation of disk blocks, or keeps some of the index blocks in the FMT of a file in memory to ensure efficient file processing. In both cases, it is beneficial to ensure that consecutive records of a sequential file are stored in adjacent disk blocks so that they can be accessed efficiently using data staggering techniques such as head skewing, cylinder skewing, or sector interleaving. It is hard to ensure it while allocating disk blocks to files that are frequently updated, hence the notion of cylinder groups should be used while allocating blocks to a file to reduce disk head movement.

Problem 4:

inf	Arrangement of sectors	t_{io} for sectors (msec)	Average t_{io} (msec)	Peak throughput (Kbytes/sec)
0	1,2,3,4,5,6,7,8	9,9,9,9,9,9,9,9	9 msec	111.1 Kbytes
1	1,5,2,6,3,7,4,8	10,10,10,11,10,10,10,9	10 msec	100.0 Kbytes
2	1,4,7,2,5,8,3,6	3,3,3,3,3,3,3,3	3 msec	333.3 Kbytes
3	1,3,5,7,2,4,6,8	4,5,4,5,4,5,4,9	5 msec	200.0 Kbytes
4	1,6,3,8,5,2,7,4	5,5,5,5,5,5,5,5	5 msec	200.0 Kbytes

Problem 5: A disk cache is advantageous if same records are likely to be used more than once, or if a disk block contains >1 record. A RAM disk is useful only for scratch files as files in a RAM disk are lost when the system is shut down or re-booted.

Problem 7: The SCAN algorithm schedules I/O operations in the order of increasing track numbers. Given a set of requests, the scheduler for a biased disk would find that the seek times of requests are proportional to the track numbers. Hence requests would be scheduled in the order of increasing track number, *provided no new requests arrive*.

Problem 8: (a) Since both blocking and buffering are employed, wait time for a record would be minimized only if $(t_{io})_{lr} < t_c + t_p$. This condition is satisfied for a blocking factor(m) of 3.

(b) When the blocking factor is 5, the number of records present in memory at different times are as follows:

Before reading which record in a block	**No. of next few records in memory**
1st record	5
2nd record	4
3nd record	3
4nd record	7∗
5nd record	6∗

∗: Reading of the next block was completed before the the process accessed this record in the current block

Hence the minimum number of records in memory at any time is 3.

Problem 9: Optimal blocking factor = 4, and number of buffers = 2. However, since the process may require 5 records together, following Problem 8 we use a blocking factor of 8. (i) Blocking factor = 6, number of buffers = 2. (ii) Blocking factor = 11. However, it causes a large memory commitment for buffers. Instead, blocking factor

= 4 and number of buffers = 3 can be used. Since the optimal blocking factor is used, 2 buffers would be always full in the steady state.

Problem 10: Use Eq. (12.5) and consider the fact that only one FMT block exists in memory at any time, so a new FMT block would have to be loaded before being used to obtain the address of a disk block containing file data.

Problem 11: This statement is invalid. I/O wait can be eliminated only if $t_{io} < t_c + t_p$.

Problem 12: From Figure 12.17, we see that the FS action for open issues I/O requests for all I/O buffers. Hence if an open is issued sufficiently in advance of the first read operation, at least one buffer would be full and the I/O wait can be completely eliminated. For example, in Problem 8, the elapsed time of the program would reduce by 40 msec. However, if the file is not opened, the process would encounter an I/O wait until the first record is read in. The onset of the steady state would also be delayed, causing a performance penalty when the process wishes to manipulate several records together (see Ex. 12.7).

Problem 13: Disk scheduling would affect effectiveness of I/O buffering in an obvious manner. A program would issue a read on the next disk block, expecting it to be read into memory before it needs the data in the block; however, the disk scheduling policy in force may not schedule the read operation immediately. Thus, the program may suffer I/O waits even if $t_{io} < t_c + t_p$.

Problem 14: (a) Buffering would be effective despite disk scheduling if the access times of all disk blocks are equal, which is not the case in practice. FIFO scheduling would perform better than other disk scheduling techniques only if (i) a program is executed all by itself, and (ii) a file has been allocated disk blocks with consecutive disk block addresses. If this is not the case, it is difficult to predict which scheduling policy will perform the best. (b) Consider a file that is processed using n buffers. The access method issues a read on all n buffers when the file is opened. When sector interleaving is used, the disk space allocation policy would allocate sectors on the same track for these blocks. Most disk scheduling policies would schedule I/O operations on these disk blocks one after another. Hence sector interleaving would be effective in reducing the I/O waits. However, subsequent I/O operations on the file would be issued intermittently, and would compete with I/O operations issued by other processes for the kernel's attention. Hence sector interleaving may not be effective. Analogously, other data staggering techniques would also be effective during access to the first few disk blocks allocated to a file, but would not be effective during access to the other disk blocks allocated to it.

Problem 15: The data transfer rate of the tape is $2 \times 1000 \times 8$ bytes per second = 16K bytes per second. Hence the transfer time per record is 400/16000 second = 1/40 second = 25 msec. (a) Each record occupies 400/8 mm = 50 mm = 5 cm. The inter-record gap is 0.5 cm. Hence the length of the tape occupied by the file is $4999 \times 5.5 + 5$ cm. The total I/O time is = $5000 \times (5 + 25)$ msec. (b) Each block occupies

$4 \times 5 = 20$ cm. The length of the tape occupied by the file is: $1250 \times (20 + 0.5)$ cm. The total I/O time is $1250 \times (5 + 4 \times 25)$ msec.

Problem 16: No. Due to blocking of the file, it is not possible to know which record of the 'current' block the process was engaged in processing when the failure occurred. Due to buffering, the data in buffers would be lost, which includes data that was read into buffers but has not yet been delivered to the process, as well as data that was written into buffers by the process but has not yet been recorded in the file.

Problem 17: We assume that if blocking and buffering are not used, a program is processed as a program with a blocking factor of 1. Consider a file containing n_r records, and a blocking factor of m. The speed-up factor when blocking is used without buffering is

$$= \frac{n_r \times (t_a + t_x + t_c + t_p)}{\frac{n_r}{m} \times (t_a + m \times (t_x + t_c + t_p))} = \frac{n_r \times (t_a + t_x + t_c + t_p)}{n_r \times (\frac{t_a}{m} + (t_x + t_c + t_p))} = \frac{(t_a + t_x + t_c + t_p)}{(\frac{t_a}{m} + (t_x + t_c + t_p))}$$

This factor can exceed 2. Consider $t_a = 10 \times (t_x + t_c + t_p)$ and $m = 5$. Now, the speed-up factor would be 3.67.

Problem 18: Refer to Problem 17 and Eq. (12.7).

Problem 19: Data in the Unix buffer cache would be lost if the system crashes. A programmer can issue `flush` after every file update to make sure that its updated data does not exist for long in the cache. It would reduce the possibility of loss of data due to a failure; however, some data (generated in its last update) remains vulnerable as it would be lost if a failure occurs before the process can issue a `flush`.

Problem 20: The `lseek` command can be used to position the disk heads to read the designated disk block. This way the process suffers a smaller I/O wait when it issues the command to read the block. The file system should convert the offset mentioned in the `lseek` command into the pair (disk block #, byte #) where 'disk block #' is the serial number of the disk block in the file. Using FMT, this number is converted to the actual disk address and the disk heads are positioned over it. Note that `lseek` would involve accessing indirect blocks in the FMT.

In a time sharing system, it is futile to position disk heads in this manner, because an I/O operation for another process can move the heads before the process that issued the `lseek` command actually reads the block. Hence a file system should try to read the data into a disk or file cache when a `lseek` command is given for an input file.

Chapter 13

Synchronization and Scheduling in Multiprocessor Operating Systems

Problem 1: It is best to use spin locks for 3 reasons: (1) The processor remains with the process executing the supervisor. (2) A spinning processor can attend to interrupts while it spins. (3) Kernel execution is more efficient because spin locks involve less process switching.

Problem 2: Two most interesting situations are: (1) To nudge a processor to do process scheduling. It is required when interrupt processing performed by a processor activates a process that has a higher priority than the process being executed by another processor. (2) To activate a processor that is sleeping on a sleep lock.

Problem 3: The hands-off hint is used by a thread to relinquish its processor to another thread, where such a hand-over would speed-up operation of an application. A thread spinning on a spin lock should hand over its processor to the thread that currently holds the lock. Thus, it is useful only for *local* purposes. However, the software scheme of Section 13.4.2 is useful for NUMA or NORMA architectures, wherein the thread holding the lock and a thread spinning for it may be located in different nodes. Hence the hands-off hint would be of marginal utility only.

Problem 4: (a) Yes, partly. If higher priority processes are not *ready*, all processes of an application would be scheduled one after another, thus giving the benefits of affinity scheduling. (b) Yes, under the condition mentioned in Part (a), processes of an application would be scheduled simultaneously.

Problem 5: No. Spin and sleep locks are used when (almost) every process has a

CPU available to it. Hence a process holding a lock is not likely to be preempted.

Problem 6: Spin and sleep locks are superior to queued locks for applications with fine synchronization granularity, because they incur a lower overhead than queued locks. Queued locks might be preferred for applications with a coarse synchronization granularity because blocking of a process frees the processor on which it was executing.

Chapter 14

Structure of Operating Systems

Problems 1 and 2:

Event handler	Activated by	Information passed
Process creation	system call	priority of new process
Await process completion	system call	process id
I/O management	I/O interrupt,	device id, operation details
	system call	device id, operation details
File management	system call	File name, operation details
Message passing	system call	Id of program to send to/receive from

Problem 3: Any OS function that changes the state of a process requires use of the mechanism 'manipulate scheduling lists'. Thus, the process management function uses this mechanism when it creates a process or changes its priority, when it blocks a process due to interaction with other processes or due to a resource request, and when it activates a blocked process because its request is granted. The memory management function uses this function when it decides to swap-out or swap-in a process. I/O and file management, and message passing use this mechanism in a similar manner.

Chapter 15

Distributed Systems

Problem 1: As described in Section 15.2, parallelism in a cluster is qualitatively different from that in an SMP. Processors do not share memory. Accordingly, they should not use spin or sleep locks. It is best to use message passing to achieve both sharing of data (rather than data access synchronization) and control synchronization.

Problem 2: Aspects of blocking and activation of processes would be implemented by procedures *send_request*, *receive_reply* and *receive_ack*. These procedures would contain appropriate **wait** and **signal** statements for this purpose. These procedures would also implement buffering of received requests and recognition of duplicates. The time-out feature in *receive_reply* and *receive_ack* can be implemented as follows: Each process has a *time-out_occurred* flag. When the process executes a *receive_reply* or *receive_ack* statement, it would be blocked on a **wait** statement. When a time-out occurs for the process, its *time-out_occurred* flag would be set and a **signal** statement would be executed to activate it. To facilitate this arrangement, the monitor would maintain a list of processes that are awaiting acknowledgments. It would contain a procedure called *clock_tick* that is invoked at every clock tick by a signal handler. This procedure would execute **signal** statements for processes whose time-outs have occurred.

Problem 3: The obvious issue here is the trade-off between response times and overhead of retransmission of requests or acknowledgments.

Problem 4: The sender uses a counter to assign sequence numbers to requests. A reply contains the sequence number of the request. In a blocking version of the RRA protocol, a reply is an outdated one if the sequence number in it is smaller than the current value in the sender's counter. A duplicate reply to the most recent request would arrive after the process has been activated. It can be simply discarded. In the non-blocking version of the protocol, the sender can maintain a list of sequence ids

of its messages for which replies have not been received. A reply would be discarded if its sequence number is not present in this list.

Problem 5: A request with a smaller sequence number may be received after some request with a larger sequence number. Hence a numerical comparison of sequence numbers cannot be used to detect duplicate requests. The protocol must check whether the request was received and processed earlier by searching for a reply with a matching sequence number in the buffer. If it is found, the request is a duplicate request, so its buffered reply is simply resent. Otherwise, the request is processed.

Problem 6: This protocol does not retransmit requests or replies, so both requests and replies may be lost. The acknowledgment would have helped the receiver to release a buffer containing a reply; however, it is a redundant feature here because messages are not retransmitted, so replies need not be buffered.

Problem 7: If the destination process blocks until it receives an acknowledgment of its reply, it will need only one reply buffer (without the modification, it would have required one reply buffer for every sender process). This feature simplifies handling of duplicate requests and acknowledgments; however, it causes delays in processing of requests by other processes. The semantics of the RRA protocol do not change due to this modification.

Problem 8: (a) System on the left: (i) 0, and (ii) 0. System on the right: (i) 0, and (ii) 3. Explanation: The sender and receiver sites must not fail, hence the system cannot tolerate any site faults. faults in all links of a site would isolate the site. If the system is divided into three parts, the left and right parts which are similar, and the central part, we find that a link fault can occur in each of the parts without disrupting interprocess communication. (b) The figure on the left: Nodes with only one link would be isolated by single link faults. Hence these six nodes must have a copy each. Other nodes do not need to have a copy. The figure on the right: Only two copies are needed; one in a node in the left half of the figure, and the other in a node in the right half of the figure.

Problem 9: d is the largest number of links in any shortest path in the system. However, there is no guarantee that communication between a pair of nodes would take place along the shortest path between them, because the routing algorithm would consider (i) traffic densities, and (ii) faults in links. Hence the maximum communication delay may not be $d \cdot \delta$. In fact, it could be as high as $(n-1) \cdot \delta$, where n is the number of nodes in the system.

Problem 10: RPC is less flexible, more efficient and more secure. All these features arise due to the fact that the remote procedure is compiled and installed at a remote site. Remote evaluation is more flexible because the id of the remote site at which code should be executed can be specified dynamically. The code to be executed may have to be compiled or interpreted in the remote site. The security aspects of its execution in the remote site would depend on capabilities of the software environment

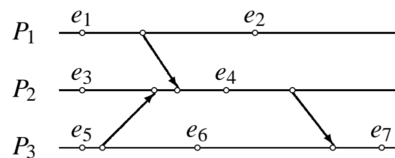
at the remote site.

Chapter 16

Theoretical Issues in Distributed Systems

Problem 1: This situation would arise if several events occurred in P_3, none of them preceding any event in P_1 or P_2 until P_3 sent message m_3. (This condition includes messages sent by P_3 to P_1 or P_2, provided they are not yet received by P_2.) An OS can ensure tighter synchronization of clocks by requiring processes to send dummy messages to one another at short intervals.

Problem 2: (a)



(b) e_1 precedes e_2, e_4, e_7; e_3 precedes e_4, e_7; and e_5 precedes e_4, e_6, e_7.
(c) e_1 is concurrent with e_3, e_5, e_6. e_2 is concurrent with all events except e_1. e_3 is concurrent with e_1, e_2, e_5, e_6. e_4 is concurrent with e_2, e_6. e_5 is concurrent with e_1, e_2, e_3. e_6 is concurrent with e_1, e_2, e_3, e_4. e_7 is concurrent with e_2.

Problem 3: A relevant observation for all proposals is that receipt of a message by a process contributes to clock synchronization only if the clock of the receiving process is adjusted. In the following, we assume that this condition is satisfied at all receive events. (a) True only if the receipt of messages from some common process P_g occurred recently. (b) This alternative is better than alternative (a) provided processes P_g and P_j received the messages in the correct order (P_g received from P_i before sending to P_j). (c) Only if P_j received P_i's message recently. (d) True. (e) True. It is better than (d) because it does not require an assumption regarding synchronization of clocks—if both processes received the messages recently, at least one of them would

synchronize its clock. (None of them would synchronize its clock if the clocks were perfectly synchronized; however this situation is unlikely.) (f) True.

Problem 4: Consider precedences of events in the processes of Problem 2. Relation (16.1) would put an order on all pairs of concurrent events as well. The primary advantage is that a total order is defined on events. This order can be used for purposes such as FCFS allocation. The disadvantage is that the order imposed on concurrent events may be 'unfair' as it may be different from the actual order in which the events occurred.

Problem 5: The purpose of rule (16.2) is to define an order on concurrent events. Due to rules R3 and R4, the time-stamp of every event is unique, hence no pair of events will satisfy condition (ii) proposed in the problem, viz. $pvts(e_i).local\ time = pvts(e_j).local\ time$. Hence the proposed relation does not impose a total order.

Problem 6: The logical clock of the site used by user A must be faster than the logical clock of the site used by user B. (Note that a phone call does not synchronize logical clocks!)

Problem 7: (a) In Figure 16.3, which depicts use of logical clocks, $e_{23} < e_{12}$, whereas in Figure 16.4, which depicts use of vector clocks, $e_{23} \not< e_{12}$.

(b) When vector clocks are used, if t_i, t_j are time-stamps of events e_i, e_j, respectively, $t_i < t_j$ only if (i) t_i and t_j are time-stamps of the send and receive events of a message, respectively, or (ii) by rules of transitiveness of precedence. Under these conditions, $t_i < t_j$ also when logical clocks are used.

(c) Let t_i, t_j be time-stamps of events e_i, e_j in processes P_i, P_j. When logical clocks are used, $t_i < t_j$ in two situations: (i) Process P_i sent a message to P_j, e_i is either the message send event or an event that precedes it in P_i, and e_j is either the message receive event or an event that succeeds it in P_j. Now, $t_i < t_j$ when vector clocks are used as in part (b). (ii) Events e_i, e_j are concurrent. In this case, $t_i[k] < t_j[k]$ for some k, and $t_i[l] > t_j[l]$ for some l. Hence $t_i \not> t_j$.

Problem 8: Given that e_i, e_j are concurrent events and $vts(e_i)[k] < vts(e_j)[k]$, there must exist some h such that $vts(e_i)[h] > vts(e_j)[h]$. Now, (a) $vts(e_l)[g] = vts(e_j)[g]$ for $g \neq k$, so we have $vts(e_i)[h] > vts(e_l)[h]$. (b) $vts(e_l)[k] > vts(e_j)[k]$, so $vts(e_l)[k] > vts(e_i)[k]$. From (a) and (b), e_i, e_l are concurrent events.

Problem 9: Consider a channel Ch_{ij} between processes P_i and P_j in a system in which state recording is in progress. Let process P_i send the following messages: m_1, marker, m_2, m_3. If the channel is not FIFO, P_j may receive m_1, m_2, marker, m_3. If this is the first marker received by P_j (see Step 2(a)), the recorded process states would be: P_i has sent message m_1, whereas P_j has received messages m_1 and m_2! The FIFO property of channels prevents it from happening.

Problem 10: State of a channel Ch_{ij} is recorded as empty if P_i records its state and sends a marker to P_j, and P_j records its state on receiving the marker. If P_j has already

received a marker along some other channel, it might record the state of Ch_{ij} as being non-empty (see Step 2(b)). Hence a system in which each process has exactly one incoming channel will have a transit-less state. A system with a star topology with edges pointing outwards is a good example.

Problem 11: Let P_i initiate a state recording. Since $\delta < \sigma$, process P_j would have sent at least one message $m*$ along Ch_{ji} before it receives a marker along channel Ch_{ij} and sends a marker along Ch_{ji}. When P_i receives the marker, it would record the state of Ch_{ji} as message $m*$. Hence the recorded state is not transit-less.

Problem 12: The theoretical background for this problem is as follows: Let state recording commence when the system is in state S_i and let it complete when the system is in state S_f. Several sequences of system states $S_i \ldots S_f$ are possible. It has been shown that the recorded state is some valid state in some sequence of states from S_i to S_f. If only one sequence of states $S_i \ldots S_j$ is possible in the system, then it is guaranteed that the recorded state is a state in which the system would have existed at some instant of time.

Consider a system containing processes $P_1 \ldots P_n$. Processes behave as follows: A process P_i sends a message to P_{i+1} after it receives a message from P_{i-1}. Let P_1 initiate state recording after sending a message. It is clear that the recorded state will be one of the actual states the system was in. In this case, it will be the state in which each $P_i, i \neq 1$ has received a message from P_{i-1} and P_1 has sent one message.

A more general example is a system in which concurrent events do not occur. A system in which only one message is in transit at any time is an obvious example of such a system. The above system is a special case of this system.

Problem 13: The marker in channel Ch_{42} is delayed and delivered after event e_{23}. The marker in Ch_{32} would reach P_2 first. P_2's state would be recorded now and markers would be sent to P_1 and P_3. When the marker along Ch_{42} reaches P_2, it would record the state of Ch_{42}. It would be empty in this case. Hence the state would be identical to the state in Table 16.4.

Problem 14: The recorded states are as follows: (a) No messages have been sent or received by any process and no messages are in transit. (b) Process P_2 has sent message m_{21} and process P_1 has received message m_{21}. Process P_3 has sent message m_{32} and process P_4 has sent message m_{43}. Process P_3 has either received message m_{43}, or m_{43} is contained in channel Ch_{43}. Ch_{32} contains m_{32}. All other channels are empty.

Problem 15: Since channels are not FIFO, receipt of a marker by a process does not imply that it has received all messages that were sent before the marker as the marker could have overtaken some messages. The algorithm must find the messages that were overtaken by the marker and record them as being present in the channel. If a process that has not received a marker receives a message with an *after marker* flag before it receives the marker, it must be the case that the message has overtaken the

marker sent earlier along the same channel. Similarly, it could have overtaken some messages that were sent before or after the marker. The algorithm must find such messages and record the state of the channel as consisting of these messages. Thus the distinction between Steps 2(a) and 2(b) of the Chandy–Lamport algorithm vanishes as far as recording the state of the channel along which a marker or a message with an *after marker* flag is received.

The following scheme can be used to record the state of a channel: Messages passed along a channel carry sequence id's starting with 1. A marker indicates the sequence id of the message that immediately precedes it. An *after marker* flag carries the same information. The process receiving messages along a channel keeps track of the id's of received messages. Let a process receive (a) a marker that indicates n messages were sent before the marker, or (b) a message with an *after marker* flag containing the information that n messages were sent before the marker was sent. The receiving process records the state of the channel as consisting of all those messages with sequence id's $\leq n$ that have not been received yet. (Refer to the bibliography of Chapter 16 for pointers to related literature.)

Chapter 17

Distributed Control Algorithms

Problem 1: A distributed mutual exclusion algorithm aims at enabling a process to enter a mutual exclusion region. Liveness of distributed mutual exclusion implies that any process desiring to use a mutual exclusion region gets to do so in a finite amount of time. Leader election simply elects one process as a leader. Its liveness implies that a leader will be elected and its id would be made known to all other processes in a finite amount of time.

Problem 2: Let a process P_i, which made a request for CS entry after another process P_j, enter the CS ahead of it. However, it would need a 'go ahead' reply from P_j, which would not send such a reply before using the CS itself because its own request for CS entry precedes P_i's request in time. Hence we have a contradiction.

Let processes P_i and P_j get into a deadlock because each of them awaits a 'go ahead' reply from the other. This, too, is a contradiction if time-stamps are distinct.

Problem 3: Consider two processes that wish to enter CS. Each process will send request messages to all other processes in the system. Hence they will receive each other's request. In the modified algorithm, they will not respond to each other's request messages since they themselves wish to enter CS. Hence none of them will be able to collect $n-1$ replies, so a deadlock exists.

Problem 4: Safety of distributed mutual exclusion implies that not more than one process can be in a mutual exclusion region at any time. The rule: for all P_i, P_j : $R_i \cap R_j \neq \phi$ ensures that no two processes can get permissions from processes in their request sets at the same time.

Problem 5: In the system of Figure 17.4, let process P_5 be in CS and let processes P_4, P_1 and P_2 make requests for CS entry in that order. The request queue of P_5 will be as shown in Figure 17.4(a)—it will have P_3 and P_1 in it. P_3's request queue will have P_4, P_2 in it. After finishing use of its CS, P_5 will send the token to P_3 and also

make a request because its request queue is not empty. P_3 will send the token to P_4 and make a request. In due course, P_4 will send the token to P_3, which will send it to P_2. Thus, P_2 will enter CS ahead of P_1.

Problem 7: The first marker received by a process is the engaging query in the Chandy–Lamport algorithm. A subsequent marker received by it is a non-engaging query.

The recorded state can be collected as follows: When a process receives a non-engaging query, it sends a reply containing the recorded state of the channel on which the non-engaging query traveled. A process collects the information received in all replies, adds its own recorded state and the state of the channel on which it received an engaging query, and sends this information in its reply to an engaging query. The initiator puts together the information received in all replies, and adds its own state. The result is the recorded state of the system.

Problem 8: In both wait-or-die and wound-or-wait, young processes are killed to ensure absence of deadlocks. If such processes retain their original time-stamps, they have a smaller probability of being killed again. However, killing of processes degrades their response times. They might miss their deadlines due to this reason. It is particularly true if a process has a 'tight' deadline because the process would have to complete its operation while it is still young.

Problem 9: In both wait-or-die and wound-or-wait, young processes are killed to ensure absence of deadlocks. If a process is given a new time-stamp when it is re-initiated, it starts off young once again, so it may be killed repeatedly and never become old enough to obtain the resources it needs and complete its operation. Hence the schemes do not possess the liveness property.

Problem 10: The proposal to use an edge chasing algorithm to detect communication deadlocks works well when symmetric naming is employed because a receiver names the process from which it wishes to receive a message and waits for a message from it. However, in asymmetric naming, a receive request does not mention a sender's name. In this case, the receiver waits for a message from anyone of the processes in the system. This relationship is analogous to the wait-for relationship of a process that requests a resource unit of a multiple instance resource class in an MISR system, hence a knot is a necessary and sufficient condition for a deadlock (see Section 11.7). However, an edge-chasing algorithm cannot detect knots as easily as it can detect a cycle, so it is not suitable when asymmetric naming is employed.

Problem 12: The *detect* rule requires that a process having identical public and private labels must wait for a process with a matching public label. If the *inc* function is not used, this condition may not hold even if a cycle exists. Consider the following situation for some n: For $i = 1, \ldots n$, the public and private labels of process P_i are both i initially. During subsequent operation, the system reaches a state wherein each process $P_i, i = 1 \ldots n-1$ waits for process P_{i+1}. In this state, the public and private

labels of P_n are both n and the public label of each of the processes $P_1, P_2, \ldots, P_{n-1}$ has been changed to n due to the *transmit* rule. Now, let process P_n wait for a resource held by process P_1, and let its public and private labels be set to a new value $m < n$. Now, the public label of P_n will be changed to n by the *transmit* rule. Thus, none of the processes has its public and private labels equal, so none of them can detect deadlock.

Problem 13: Every active process and every message carries some credit. Hence obvious.

Problem 14: The algorithm prevents a situation wherein a receiver node becomes a sender node because too many processes are transferred to it. However, it does not contain any provision to control the overhead of trying to find a receiver. Hence its overhead could become arbitrarily large under high load, which makes it unstable at high load.

Problem 15: The 'good' thing in distributed scheduling is that a load imbalance is attended to by the algorithm. A 'bad' thing is instability. Hence liveness implies that a load imbalance is attended to eventually by the system and safety implies that the algorithm does not suffer instability. Interestingly, liveness cannot be achieved unless safety is achieved, however safety does not guarantee liveness (because imbalances would persist if safety is achieved by limiting the overhead).

Chapter 18

Recovery and Fault Tolerance

Problem 1: As described in Section 18.3, a synchronous checkpoint is a consistent checkpoint because it contains consistent process states. This fact implies that each message recorded as received by a process is recorded as sent by its sender. However, strong consistency also requires that no messages be in transit in the recorded state. A synchronous checkpoint does not possess this property.

Problem 2: Each process takes a checkpoint after sending a message. Hence no message becomes an orphan when a process rolls back to its last checkpoint. An orphan message would arise if a process has to roll back farther than its last checkpoint. However, such a rollback would be needed only if some message received before the last checkpoint becomes an orphan; this situation can never arise. However, the sending of a message and taking of a checkpoint should be performed atomically—either both take place or none does.

Problem 3: Yes. Consider a process P_i that takes a checkpoint, receives a message, sends another message $m*$ and fails. When P_i rolls back to its last checkpoint, $m*$ would become an orphan. A process that had received $m*$ would have to be rolled back to its latest checkpoint, so a message sent by it after that checkpoint would become an orphan, and so on.

Problem 4: Yes, use of quorums determined by Eq. (18.2) can lead to a deadlock—consider $Q_r + Q_w = n + 2$, and some reader and writer processes lock $Q_r - 1$ and $Q_w - 1$ copies. To prevent deadlocks while still satisfying Eq. (18.2), one can use $Q_r + Q_w = n + 1$. For an analogous reason, Eq. (18.1). should be satisfied by using 2 $\times$ $Q_w = n + 1$.

Problem 5: No. Rules 1 and 4 of Section 18.4.2 cannot be satisfied by the Maekawa-style request sets.

Problem 6: (a) No, because process P_i may have to roll back more than once. (b) No.

Process P_i may have received some messages. It would have to roll back if one of these messages becomes an orphan, so C_{ij} might be needed during recovery. (c) Yes. Under these conditions, rolling back process P_i from C_{ij+1} to C_{ij} does not nullify the action of receiving any messages that have become orphans, hence such a roll back would not contribute to recovery—if process P_i has to be rolled back from C_{ij+1} due to the domino effect, it would always have to be rolled back to C_{ij-n} for some $n > 0$. (d) No. Let P_k be the process that had sent a message m_k that is recorded as received in checkpoint C_{ij} of process P_i. The condition that P_k has taken a checkpoint after sending m_k is not adequate for preventing a roll back of P_i as follows: P_k might have received some message $m*$ before taking the checkpoint after sending m_k. If $m*$ becomes an orphan, P_k will be forced to roll back. That would make m_k an orphan, so P_i would have to roll back. Hence a previous checkpoint of P_i should be preserved.

Chapter 19

Distributed File Systems

Problem 2: As discussed in Section 19.3 and illustrated in Figure 19.2, a session is restricted to a single site in the system, i.e., all clients participating in a session exist in the same site. Hence it is best to maintain the version of the file being accessed by the clients in the same site. This arrangement appears similar to file caching in several ways but differs from conventional file caching in two important respects. More than one session may be in progress in the same site, hence more than one copy of the file may be needed in one site. However, by definition, different sessions on the same file access different versions of the file, so there is no need to maintain consistency between file copies used by sessions in the same or different sites. Hence session semantics does not suffer response degradation due to coherence traffic as in the case of file caching. This fact makes implementation of session semantics more efficient than file caching. The copy of the file in the filing system needs to be updated only once at the end of the session. Whether file versions created by different sessions should be maintained as distinct versions or whether they should be reconciled in some manner depends on the version of session semantics adopted by a specific DFS.

Problem 3: Maintaining file buffers at the client site is bad for Unix semantics because considerable coherence traffic would be generated by the need to maintain coherence between file buffers maintained at various sites. Use of file buffers in the server site suits Unix semantics for obvious reasons.

Maintaining file buffers at the client site suits session semantics. However, session semantics supports multi-image mutable files and even clients in the same site may access different images of a file. This feature complicates the DFS design. Use of file buffers at the server site does not suit session semantics for obvious reasons.

Problem 4: (a) Crash of a stateful server disrupts file processing activities completely—data held in buffers and states of file processing activities are lost. Recovery

is simplest if the file server provides atomic transactions; otherwise, a client would have to create back-ups of files involved in a file processing activity and manually roll back files to the back-ups when a failure occurs. (b) Failure of a stateless server does not disrupt file processing because state information is not lost. The client merely notices a slow response. However, as mentioned in Section 19.4.3, a failure may have undesirable side-effects like duplicate actions, or misleading or ambiguous warnings if the server processes file creation or deletion requests more than once.

Problem 5: When clients access different files, DFS threads handling client requests need synchronization while accessing DFS data structures like directories and the free list, but do not need synchronization while accessing data stored in files. However, when clients access the same file dynamically, the threads must cooperate to implement the file sharing semantics.

Problem 6: The issues to be handled by a file recovery procedure depend on the exact arrangement used by the DFS for handling file updates. Recovery is simpler if file updates are made only on the primary copy of a file—file processing can continue if the failed node contained a non-primary copy of the file, whereas writes would have to be delayed if the failed node contained the primary copy. If quorum-based file replication is used, node failures can be simply ignored if fault tolerant quorums are used as described in Section 18.4.2; otherwise, it would be necessary to wait until a failed site comes up before resuming an application.

Where failure of a site holding a replicated file does not disrupt file processing, the copy may be out-of-date by the time the site is operational again. It is therefore necessary to 'restore' its copy of the file to the current state before letting it participate in file processing. This task can be quite complicated unless the primary copy approach is used. It is best to obtain a write quorum of files and use the information found in it to restore the file in the failed node.

Problem 7: The stateless file server does not maintain any state information. Therefore it is not aware of which clients cache any part of a file. Hence file caching, if at all implemented, must be completely handled by a client herself, so cache validation must be performed by the client. Statelessness of the server also implies several complications and compromises concerning file sharing semantics. See Section 19.6.1 for a discussion of caching in the Sun NFS.

Problem 8: This statement is correct. File caching resembles maintenance of file buffers at the client site. Therefore, following the discussion of Problem 3, file caching does not integrate well with Unix semantics. It integrates well with session semantics if the DFS makes arrangements to cache different versions of a file in a site if the site contains several concurrent sessions on the file.

Chapter 20

Distributed System Security

Problem 1: $n = 77$. Hence $p = 11$ and $q = 7$. e should be relatively prime to $(11 - 1) \times (7 - 1)$, i.e., to 60. Let e be 71. Now $d \times 71 \bmod 60 = 1$. Let d be 11. Hence either (1) 11 is the public key and 71 is the private key (or vice versa), or (2) (11, 77) is a public key and (71, 77) is the private key.

Problem 2: Let an intruder attempt impersonation by tampering with the message from a process to the KDC in Protocol (20.3). The intruder modifies the original message 'P_i, P_j' to 'P_i, P_k' with the intention of impersonating P_i while communicating with P_k. However, KDC's reply to the modified message is $E_{V_i}(P_k, SK_{i,k}, E_{V_k}(P_i, SK_{i,k}))$. P_k expects that $E_{V_k}(P_i, SK_{i,k})$ should be passed to it when P_i initiates communication with it. However, the intruder does not know V_i, so she cannot extract this information from the message sent by the KDC. Hence the intruder cannot gain any advantage by tampering with the message.

Problem 3: In Step 3, P_i passes the unit $E_{U_j}(P_i, SK_{i,j})$ containing the session key $SK_{i,j}$ to P_j, which is encrypted using P_j's public key, U_j. This is a weakness because an intruder can fabricate such a unit containing a session key of its own choice because it knows P_j's public key. Use of the fabricated unit would permit the intruder to masquerade as P_i while communicating with P_j using the chosen session key.

Problem 4: Tampering can lead to denial of service. For example, consider some key distribution schematic. A process requests the KDC for a key to communicate with some other process P_i. If all request messages sent to the KDC are tampered with to change P_i to some other process id (say P_k), the requester may never be able to obtain a key to communicate with P_i.

Problem 5: A client (C) authenticates a user using his/her password, and finds his/her key V_U by using the function call $f(\textit{password})$. KAS knows V_U from its database. This arrangement avoids passing of the password over the network. When

a user wishes to change her password, C prompts her to provide her old and new passwords. It now obtains a new V_U through the function call $f(new_password)$. It also provides the new password or the new V_U to KAS by encrypting it with the old V_U. This password change protocol assumes that the old password is secure at the time it is changed. If it is not so, the user would have to approach the system administrator for a change of password.

Problem 6: It is important that the lifetime be a 'tight' guess of the actual length of a session, as using a longer lifetime would extend exposure of the session key over an unnecessarily large amount of time. However, difficulties arise if the actual session extends over a longer time period than specified in the lifetime—the user will be forced to initiate another session. Hence lifetimes tend to be longer than necessary. Kerberos provides for lifetimes of up to 10 hours.

Problem 7: Implications and use of the lifetime field may give the impression that replay attacks are possible during the lifetime of a ticket; however, it is not so because an authenticator is enclosed in every message from a client to a server. The authenticator contains the time-stamp of the request, i.e., the time at which the request originated. Kerberos uses loosely synchronized clocks to check whether the request is 'timely'. Hence replay attacks are possible only in very close temporal proximity of the original message. Such attacks can be countered by saving each request for a certain period of time so that duplicate requests can be detected. This arrangement would make replay attacks impossible.

Problem 8:

3.1 C → Server : $T_{<Server_id>}, AU, E_{SK_{U,<Server_id>}}(<service\ request>, n_3)$
3.2 Server → C : $E_{SK_{U,<Server_id>}}(n_3)$

The session key for communication between U and the server is embedded in the ticket for the Server. The ticket itself is encrypted using the private key of the server. Hence extraction of the nonce n_3 from the unit encrypted using the session key requires knowledge of the server's key. Private keys are considered secure hence it is safe to assume that if a process can extract n_3, it must be the server.

Problem 9: P_j can use a challenge–response protocol to authenticate P_i. However, the authentication must itself be carried out using some other key. This dilemma can be resolved by requiring P_j to request a fresh key for a session with P_i. This request generates another session key $SK^*_{i,j}$. This key can be used for the session. (The challenge–response protocol can be used with this key, however such authentication is unnecessary since $SK^*_{i,j}$ is a fresh key hence replay attacks using messages encrypted with it would have to happen "on line", which is unlikely.)