COS214 Tutorial 1

Roberto Togneri, 2000

SOLUTIONS

1. What are the **main** functions of an operating system?

Answer

Functions of an operating system are:

- Shield the user from the intricate low-level details and provide an easier to program and more powerful interface to the machine or network => *virtual machine.*
- Manage the system resources: protection, scheduling, sharing, file system, accounting, etc. This function is obviously important for multitasking (OS2) and mutiusing/multitasking (UNIX, VMS) operating systems. Even for single-user/single task, resource management is important (e.g. file system allocation and structure).

2. What is meant by the term *virtual machine*? What is the benefit of viewing a system such as an operating system as a set of layers? Why shouldn't an operating system be built as one large program?

Answer

Virtual machine: Operating system provides the user with a higher-level way to control the underlying hardware of the machine.

For example: Using the high-level *read(fp, buf, n)* system call to read a byte from the hard disk which the OS translates to the low-level assembly language sequence to set up and perform the disk transfer

Viewing the operating system as a set of layers has the following advantages:

- Easier to understand OS. Presents an ordered view of an OS.
- Easy to improve OS by modifying a layer. Layers are assumed independent in operation with well-defined interfaces. Layering is an example of software modularity.
- Possibility of standardisation of different layers.
- Different teams can be responsible for software development of different layers.
- Easy to debug and maintain OS. Makes it possible to develop sophisticated systems.

On the other hand a monolithic OS has the following problems:

- Very large intricate source code. A nightmare to program and maintain.
- Not easy to develop OS as a team: can't allocate programming tasks.
- Harder to debug any problems.

3. Explain the terms *time-sharing* and *multi-programming*.

Answer

Mutli-programming: Multiple programs or jobs are kept in memory and executed together. Job execution scheduling will allow the CPU to be used by one job while the other job is waiting for I/O. This permits almost 100% utilisation of CPU and other resources (note that the schedular is an overhead).

Time-sharing: Allows many users to access the computer directly and on-line and at the same time. Concept relies on the fact that users are idle for the majority of the time. Both compute-intensive jobs and interactive service can be provided.

4. What is *spooling* and what are the benefits? Is it useful on a single-user system?

Answer

Spooling: refers to queuing jobs on disk and having the system service each job in turn as resources become available. A good example is print job spooling. Spooling is useful on single-user system as it allows the user to submit a job for printing or processing without having to wait for the current job to finish.

5. Which of the following instructions should be allowed only in kernel mode?

   (a) Disable all interrupts.
   (b) Read the time of day clock.
   (c) Set the time of day clock.
   (d) Change the memory map.

Answer

Choice a), c) and d) should be restricted to kernel mode.

6. Why is the process table needed in a *timesharing* system? Is it also needed in personal computer systems in which only one process exists, that process taking over the entire machine until it is finished?

Answer

The process table is needed to store the state of a process that is currently suspended, either ready or blocked. It is not needed in a single process system because the single process is in control of the system and there is no need to "track" it.

7. We have stressed the need for an operating system to make efficient use of the computing hardware. When is it appropriate for the operating system to forsake this principle and to "waste" resources? Why is such a system not really wasteful (even in a single-user system)?

Answer

Single-user systems should maximise use of the system for the user. A GUI might "waste" CPU cycles but it optimises the user's interaction with the system.

COS214 Tutorial 2

Roberto Togneri, 2000

SOLUTIONS

1. (a) Give a definition of the term *process*. Explain the difference between procedure, program, process, task, and job.
   (b) On all current computers, at least part of the interrupt handlers are written in assembly language. Why?
   (c) For UNIX, give some examples of process manipulation functions.

Answer

(a) Definitions:
Process - instantiation of a program in the computer system; the program when it is running in the system or the activity associated with a program.
Procedure - the part of a program or region of a process which performs a well-defined function.
Program -the source or object code that resides on disk that contains the sequence of instructions to carry out a task.
Task - what you want the computer to do for you, usually when running as a process
Job - set of {programs, processes} that you want executed. Usually describes the programs not the running → submit a job… when it runs it creates one or more processes.

(b) Generally, high level languages do not allow one or other kind of access to the CPU hardware that is required. For instance, an interrupt handler may be required to enable and disable the interrupt servicing a particular device, or to manipulate data within a process' stack area. Also, interrupt service routines must execute as rapidly as possible.

(c) Process manipulation functions:
fork(): spawn a child process.
wait (): wait for the child process.
exit(): terminate a process.
setpriority(): set the scheduling priority of the process.
getpriority(): get the scheduling priority of the process.
signal(): configure the signal handler for a process
kill(): send a signal to a (another) process (e.g. SIGINTR, SIGKILL, etc.)

2. The Running, Blocked and Ready states of a process imply that the process is always in main memory. However, high-level scheduling will include the ability to swap the process's main memory to disk, which is defined as the Suspend state for a process, and reload the process into main memory when it is activated.
   (a) Explain from which state or states a process can become suspended and why?
   (b) Explain the problem in deciding which state a suspended process should go to when that process is activated? What is the UNIX System V solution?

Answer

(a) It makes sense for the high-level scheduler to swap (suspend) processes which are wasting main memory → in the blocked state or waiting too long in the ready queue.

(b) Since a process was in a blocked/ready state when suspended it should go back to the ready state upon wakeup/activate. The UNIX System V solution is to create two suspend states "*Ready to Run, swapped*" and "*Sleep, swapped*"

3. (a) A blocked process still consumes system resources. Is it possible for a program to either accidentally or otherwise continually create processes that immediately block on some event that won't occur and so bring the system down?
   (b) If a parent process dies, what should happen to the child processes? What happens to a parent process when a child dies?

Answer

(a) Yes! By spawning processes and having them block you can fill up the process table making the system unusable → hit the panic button.

(b) If a parent process dies the child should die; however, UNIX does allow child processes to continue by giving them *init* as the parent when the real parent dies. Note: If the child process's parent loops then the child will <*defunct*> when it finishes since it is necessary for the child to return the exit status to somebody. A parent process should acknowledge that a child process has exited (via the appropriate *wait*(.) system call) otherwise the child will <*zombie*>.

4. Five batch jobs *A* through *E*, arrive at a computer center at almost the same time. They have estimated running times of 10, 6, 2, 4, and 8 minutes. Their (externally determined) priorities are 3, 5, 2, 1 and 4, respectively, with 5 being the highest priority. For each of the following scheduling algorithms, determine the mean process turnaround time. Ignore process switching overhead

   (a) Round Robin (RR).
   (b) Priority Scheduling.
   (c) First come, first served (FCFS).
   (d) Shortest job first (SJF).

   For (a), assume that the system is multi-programmed, and that each job gets its fair share of the CPU. For (b) through (d) assume that only one job at a time runs, until it finishes. All jobs are completely CPU bound.

Answer

For Round Robin, during the first 10 minutes each job gets 1/5 of the CPU. At the end of ten minutes, C finishes. During the next 8 minutes, each job gets 1/4 of the CPU, after which time D finishes. Then each of the three remaining jobs gets 1/3 of the CPU for 6 minutes, until B finishes, and so on. The finishing times for the five jobs are 10, 18, 24, 28 and 30, for an average of 22 minutes. For priority scheduling, B is run first. After 6 minutes it is finished. The other jobs finish at 14, 24, 26, and 30, for an average of 20 minutes. If the jobs in the order A through E, they finish at 10, 16, 18, 22, and 30, for an average of 19.2 minutes. Finally, shortest job first yields finishing times of 2, 6, 12, 20 and 30, for an average of 14 minutes.

5. (a) Consider the implementation of all the short-term scheduling algorithms discussed in the lecture.
      (i) Which algorithms are effectively impossible to implement? Carefully explain why?
      (ii) Which algorithms require a timer interrupt for the CPU?
      (iii) Hence explain why RR is the only real scheduling algorithm that can be used in practice.
   (b) Can a single processor system have no processes in the ready queue? Explain what can happen in such a situation.

Answer

(a)

(i) RR and FCFS are easy to implement. However SJF, SRT and HRRN require knowledge of the CPU burst-time (or service time) which is theoretically impossible. Why? Even though the actual assembly instructions can be used to estimate the CPU execution cycle time, it is not possible to identify which instruction will cause a process to block since this depends on the state of the system (e.g. if data is in memory caches or buffers there is no need to block). Furthermore, the proposed aging algorithm is a major overhead and unless it is accurate (which it isn't!) will seriously undermine the effectiveness of SJF, SRT and HRRN.

(ii) The pre-emptive algorithms: SRT and RR, require a timer interrupt facility on the CPU (early CPUs did not have this!) to allow the current process to be interrupted and the OS scheduler( ) to run and either select the next process to run from the ready queue or resume the currently running process (if the quantum has not expired in the case of RR, and if there are no processes in the ready queue with a shorter remaining time as in the case of SRT).

(iii) From (i) SJF, SRT and HRRN need to be ruled out as a practical implementation is not possible. FCFS is simpler than RR but yields very poor and highly variable response and turnaround times. RR is has better response times which can be improved with priority scheduling and the response/turnaround times are not dependent on the arrival order of processes as is the case with FCFS.

(b) Yes it can. Unless the scheduler can cope with the absence of processes the system may crash. A dummy process can be used to avoid the problem if the scheduler can't be modified. Most operating systems don't have this problem as there is always some processes running (*init*, *swapper* for UNIX).

6. (a) Explain why two-level scheduling is commonly used.
   (b) Define the difference between preemptive and nonpreemptive scheduling. State why strict nonpreemptive scheduling is unlikely to be used in a computer center.
   (c) A CPU scheduling algorithm determines an order for the execution of its scheduled processes. Given **n** processes to be scheduled on one processor, how many possible different schedules are there?

Answer

(a) Two-level scheduling is needed when memory is too small to hold all the ready processes. Some set of them is put into memory, and a choice is made from that set. From time to time the set of in-core processes is adjusted. This algorithm is easy to implement and reasonably efficient, certainly a lot better than, say, round robin without regard to whether the process was in memory or not.

(b) *Nonpreemptive:* process uses CPU until terminate or block event, next process on ready queue will then run.
Preemptive: process uses CPU and is subject to a timer interrupt to permit next process in ready queue to run.
Problems with nonpreemptive scheduling? A 1 hour CPU job will freeze the computer for other users/processes for 1 hour, not very friendly.

(c) n!

7. Consider the following measured CPU service times for a process:
   1 1 2 4 5 5 3 2 2 6 7 7 7
   Use the aging algorithm with a = 0.5 to predict the next CPU service time and comment on the accuracy of this method. Assume the initial predicted service time is 1.

Answer

1:	Pred: 1	Meas: 1	Error: 0
2:	Pred: (1+1)/2 = 1	Meas: 1	Error: 0
3:	Pred: (1+1)/2 = 1	Meas: 2	Error: +1
4:	Pred: (1+2)/2 = 1.5	Meas: 4	Error: +2.5
5:	Pred: (1.5+4)/2 = 2.75	Meas: 5	Error: +2.75
6:	Pred: (2.75+5)/2 = 3.88	Meas: 5	Error: +1.12
7:	Pred: (3.88+5)/2 = 4.44	Meas: 3	Error: -1.44
8:	Pred: (4.44+3)/2 = 3.72	Meas: 2	Error: -1.72
9:	Pred: (3.72+2)/2 = 2.86	Meas: 2	Error: -0.86
10:	Pred: (2.86+2)/2 = 2.43	Meas: 6	Error: +3.57
11:	Pred: (2.43+6)/2 = 4.22	Meas: 7	Error: +2.78
12:	Pred: (4.22+7)/2 = 5.61	Meas: 7	Error: +1.39
13:	Pred: (5.61+7)/2 = 6.30	Meas: 7	Error: +0.30

COS214 Tutorial 3

Roberto Togneri, 2000

SOLUTIONS

1. What two advantages do threads have over multiple processes? What major disadvantage do they have? Suggest one application that would benefit from the use of threads, and one that would not.

Answer
Threads are very inexpensive to create and destroy, and they use very little resources while they exist. They do use CPU time for instance, but they don't have totally separate memory spaces. Unfortunately, threads must "trust" each other to not damage shared data (concurrency and protection problem). For instance, one thread could destroy data that all the other threads rely on, while the same could not happen between processes unless they used a system feature to allow them to share data. Any program that may do more than one task at once could benefit from multitasking. For instance, a program that reads input, processes it, and out-puts it could have three threads, one for each task. "Single-minded" processes would not benefit from multiple threads; for instance, a program that displays the time of day.

2. (a) What resources are used when a thread is created? How do they differ from those used when a process is created?
   (b) Describe the actions taken by a kernel to switch context:
   (i) Among threads.
   (ii) Among processes.

Answer
(a) A context must be created, including a register set storage location for storage during context switching, and a local stack to record the procedure call arguments, return values, and return addresses, and thread-local storage. A process creation additionally results in memory being allocated for program instructions (text) and data.
(b)
(i) The thread context must be saved (registers and accounting if appropriate), and another thread's context must be loaded.
(ii) The same as (i), plus the memory context (e.g page tables, MMU registers, etc.) must be stored and that of the next process must be loaded.

3. Consider the following set of processes, with the length of the CPU-burst time given in milliseconds:

Process	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	3
P4	1	4
P5	5	2

Calculate the per process and average turnaround times for each of the following scheduling algorithms:
(a) Shortest-Job-First;
(b) Nonpreemptive priority (smaller number means higher priority);
(c) Round Robin with q=1.
Assume P1 is at the head of the ready queue and P5 is at the tail.

Answer
(a) SJF:

P2	1	
P4	1+1 = 2	
P3	2+2 = 4	Finish Time = (Start-time + Burst-time)
P5	4+5 = 9	Turnaround Time = Finish Time (since Arrival Time = 0)
P1	9+10 = 19	

<u>*Ave:*</u> (1+2+4+9+19)/5=7.0

(b) Nonpreemptive priority (if equal priorities default to FCFS):

P2	1	
P5	1+5 = 6	
P1	6+10 = 16	Finish Time = (Start-time + Burst-time)
P3	16+2 = 18	Turnaround Time = Finish Time (since Arrival Time = 0)
P4	18+1 = 19	

<u>*Ave:*</u> (1+6+16+18+19)/5 = 12.0

(c) RR(q=1)

RR → Process	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	X					X			X		X		X		X	X	X	X	X
2		X																	
3			X				X												
4				X															
5					X			X		X		X		X					
Finish		2		4			3							5					1

<u>*Ave:*</u> (2+4+7+14+19)/5=9.2

4. Consider the following set of processes, with the length of the CPU-burst time given in milliseconds:

Process	Burst Time	Arrival Time
P1	10	3
P2	1	1
P3	2	4
P4	1	3
P5	5	0

Calculate the per process and average turnaround times for each of the following scheduling algorithms:
(a) Shortest-Job-First (SJF);
(b) Shortest Remaining Time (SRT);
(c) Round Robin (RR) with q=3.
Assume each context switch takes 1 unit of time.

Answer
(a) P5 is scheduled first since it is the only process in the ready queue at t=0. When P5 completes the remaining processes have all arrived and are scheduled in the order of SJF: P2, P4, P3, P1 (if equal burst-times default to FCFS):
Finish Time = (Start-time + Context Switch overhead + Burst Time)
Turnaround Time = Finish Time - Arrival Time

(t=0)	P5	0+1+5 = 6	→ 6 - 0 = 6
(t=6)	P2	6+1+1 = 8	→ 8 - 1 = 7
(t=8)	P4	8+1+1 = 10	→ 10 - 3 = 7
(t=10)	P3	10+1 +2 = 13	→ 13 - 4 = 9
(t=13)	P1	13+1+ 10 = 24	→ 24 - 3 = 21

Ave: (6+7+7+9+21)/5=10.0

(b) P5 is scheduled first since it is the only process in the ready queue at t=0. When P2 arrives it has a shorter remaining time and so it pre-empts P5 at t=2 (P5 will then have a burst-time of 4). When P2 finishes at t=4 (an extra unit of time is needed for each context switch!), P3 and P4 have arrived, so P4 is scheduled next, then P3, then P5 and finally P1:
Finish Time = (Start-time + Context Switch overhead + Burst Time)
Turnaround Time = Finish Time - Arrival Time

(t=0)	P5	1+1 = 2	(P5 has not yet finished!)
(t=2)	P2	2+1+1 = 4	→ 4 - 1 = 3
(t=4)	P4	4+1+1 = 6	→ 6 - 3 = 3
(t=6)	P3	6+1+2 = 9	→ 9 - 4 = 5
(t=9)	P5	9+1+4 = 14	→ 14 - 0 = 14
(t=14)	P1	14+1+10 = 25	→ 25 - 3 = 22

Ave: (3+3+5+14+22)/5=9.4

(c)

RR → Process	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	…	26
1							**C**	**X**	**X**	**X**									**C**	**X ..X**	**X**
2					**C**	**X**															
3													**C**	**X**	**X**						
4											**C**	**X**									
5	**C**	**X**	**X**	**X**												**C**	**X**	**X**			
Arrival	2		1,4	3																	
Finish						2						4			3			5			1

Ave: ([6 - 1] + [12 - 3] + [15 - 4] + [18 - 0] + [26 - 3])/5=13.2

5. Suppose that a scheduling algorithm (at the level of short-term CPU scheduling) favors those processes that have used the least processor time in the recent past. Why will this algorithm favor I/O-bound programs and yet not permanently starve CPU-bound programs?

Answer
It will favor the I/O-bound programs because of the relatively short CPU burst requested by them (least processor time); however, the CPU-bound programs will not starve because the I/O-bound programs will relinquish the CPU relatively more often to do their I/O.

6. Consider a variant of the Round Robin (RR) scheduling algorithm where the entries in the ready queue are pointers to the Process Control Block's (PCB).
(a) What would be the effect of putting two pointers to the same process in the ready queue?
(b) What would be the major advantages and disadvantages of this scheme?
(c) How would you modify the basic RR algorithm to achieve the same effect without the duplicate pointers?

Answer
(a) Process appear twice in the ready queue and is scheduled twice as often as other processes.
(b) *Advantage:* implement priorities. *Disadvantage:* overheads in maintaining pointers; same number of context switches.
(c) Adaptive quantums for each process. A higher priority process can use up to 2/3/4/etc. quantums of time over the single quantum for normal processes.

7. Consider a system that collects and processes data from two sensors, A and B. The deadline for collecting data from sensor A must be met every 20 ms, and that for B every 50 ms. It takes 10 ms, including operating system overhead, to process each sample of data from A and 25 ms to process each sample of data from B.
(a) List the execution profile of task A, listing the arrival time, execution time, and completion deadline. Repeat for the execution profile of task B.
(b) Show how an earliest deadline scheduling policy with preemptive scheduling satisfies the real-time requirements for both tasks.

Answer

(a)

For Task A:

Process	Arrival Time	Execution Time	Deadline
A(1)	0	10	20
A(2)	20	10	40
A(3)	40	10	60
A(4)	60	10	80
…	…	10	…

For Task B:

Process	Arrival Time	Execution Time	Deadline
B(1)	0	25	50
B(2)	50	25	100
B(3)	100	25	150
…	…	25	…

(b)

With earliest-deadline scheduling and preemption the scheduling sequence would be:

Start	End	Process	Completion	Deadline	Desc
0	10	A(1)	10	20	A(1) completes
10	20	B(1)	--	--	B(1) pre-empted by A(2) since A(2) has an earlier deadline
20	30	A(2)	30	40	A(2) completes
30	45	B(1)	45	50	B(1) resumes and completes, A(3) arrives at 40
45	55	A(3)	55	60	A(3) completes, B(2) arrives at 50
55	60	B(2)	--	--	B(2) pre-empted by A(3)
…	…	…	…	…	…

COS214 Tutorial 4

Roberto Togneri, 2000

SOLUTIONS

1. Explain how the following algorithms work in allocating memory:
   (a) First fit.
   (b) Best fit.
   (c) Worst fit.

   Given free memory blocks of 100K, 500K, 200K, 300K and 600K (in this order on a linked list), how would each of the above algorithms place requests for 212K, 417K, 112K and 426K (requested in that order)? Which algorithm makes best use of memory?

Answer

Initially:

BLOCK:	a	b	c	d	e
FREE:	100	500	200	300	600

Memory Allocation:

First Fit	Blocks	Free	Best Fit	Blocks	Free	Worst Fit	Blocks	Hole
212	b	288	212	d	88	212	e	388
417	e	183	417	b	83	417	b	83
112	b	176	112	c	88	112	e	276
426	Wait		426	e	174	426	Wait	

Best fit is best!

2. (a) What is the difference between internal and external fragmentation? In these days of large RAM memories is this any longer important?
   (b) What is the difference between physical and virtual addresses?
   (c) Is it reasonable to have the following situations:
      (i) physical address space >> virtual address space
      (ii) virtual address space >> physical address space
      (iii) virtual address space == physical address space
   (d) In the lecture the MMU is shown as translating every virtual address to a physical address. Explain what is required to make this an efficient process given that even a simple instruction will need at least one translation and some may require several.

Answer

(a) internal fragmentation: unused space within allocated segment. Common in systems where allocated space is greater than required.
external fragmentation: unused space between allocated segments.
With large RAM memories the proportion of memory wasted due to fragmentation is usually less. The cost in time and performance of de-fragmenting memory is not usually worth it.

(b) physical address: real RAM locations needed at micro level.
virtual address: pseudo addresses used by processes which must be mapped to physical addresses by MMU.

(c) (i) PA >> VA: Looks silly, but useful for preventing any one process hogging all the memory, that is each process has a VA << PA.
(ii) VA >> PA: Normal use of VA to allow processes to "use" more memory than is physically available, especially separating text/data and stack by placing them at opposite ends of the VA range.
(iii) VA == PA: May eliminate some overheads in the VA to PA mapping or result in fast mapping implementations since the size of f = size of p

(d) Use a *fast associative memory (translation lookaside buffer)*.

3. (a) With the ever increasing number of bits per memory chip (e.g. 1M, 4M, 16M, and 64M), will there still be the need to consider virtual memory techniques?
   (b) The Intel 8086 processor did not support virtual memory. Nevertheless, some companies sold systems that contained an unmodified 8086 CPU and did paging. Make an educated guess as to how they did it.

Answer

(a) Yes probably. So far the demand for more memory (graphics, images, ANN etc) has outstripped technology.

(b) They built an MMU and inserted it between the CPU and the bus. Thus all 8086 physical addresses went into the MMU as virtual addresses. The MMU then mapped them onto physical addresses, which went to the bus.

4. (a) A machine has a CPU with a 32 bit address space and uses 8K pages. The page table is entirely in hardware, with one 32 bit word per entry. When a process starts the page table is copied to the hardware from memory, at one word every 100nsec. If each process runs for 100msec (including time to load the page table), what fraction of the CPU time is devoted to loading the page tables?
   (b) A computer with a 32 bit address space uses a two-level page table. Virtual addresses are split into a 9 bit top-level page table field, an 11 bit second level page table field, and an offset. How large are the pages and how many are there in the virtual address space?

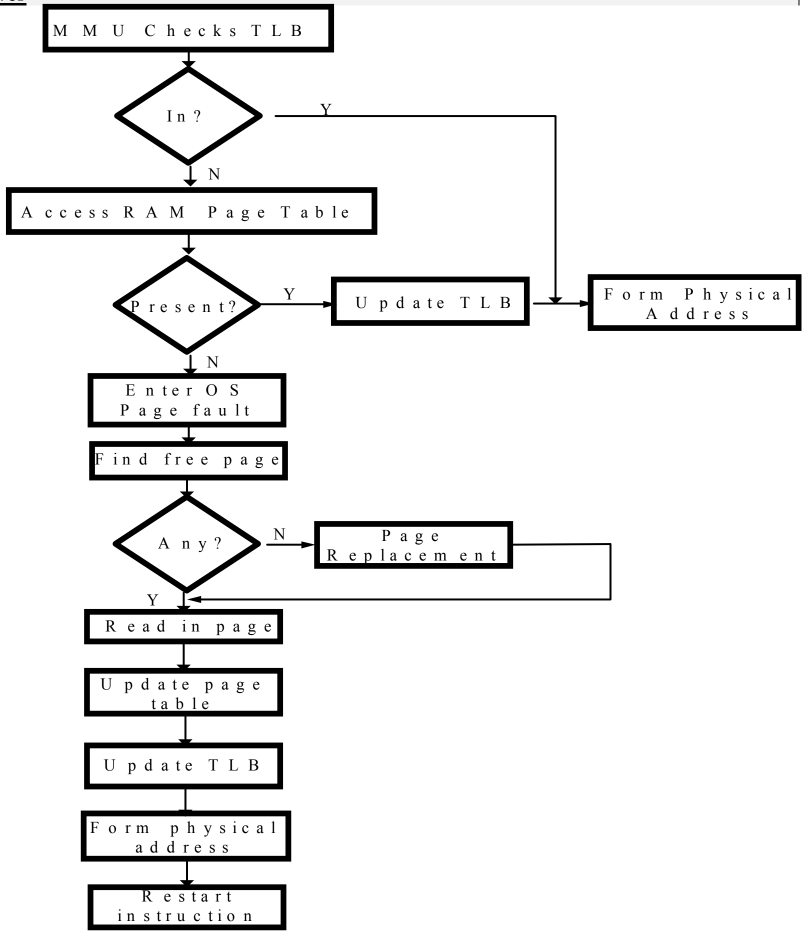
Answer

(a) The page table contains $2^{32}/2^{13}$ entries, which is 524,288. Loading the page table takes 52.4 msec. If a process gets 100msec, this consists of 52.4 msec for loading the page table and 48 msec for running with it. Thus 52 percent of the time is spent loading page tables.

(b) Twenty bits are used for the virtual page numbers, leaving 12 over for the offset. This yields a 4K page. Twenty bits for the virtual page implies 2^{20} pages.

5. Draw up a flow-chart to cover all the situations that arise in the operation of virtual memory paging with a TLB. Ignore the detail of running another waiting process whilst waiting for a missing page to be read in from the swap/paging disk area.

Answer



6. (a) A computer whose processes have 1024 pages in their address spaces keeps its page tables in memory. The overhead required for reading a word from the page table is 500 nsec. To reduce this overhead, the computer has an associative memory, which holds 32 (virtual page, physical page frame) pairs, and can do a look up in 100 nsec. What hit rate is needed to reduce the mean overhead to 200 nsec?

(b) A group of operating system designers for the Frugal Computer Company are thinking about ways of reducing the amount of backing store needed in their new OS. The head guru has just suggested not bothering to save the program text in the swap area at all, but just page it in directly from the binary file whenever it is needed. Are there any problems with this approach?

(c) Consider a paging system with the page table stored in memory. If a memory reference takes 200 nanoseconds, how long does a paged memory reference take? If we add associative registers, and 75 percent of all page-table references are found in the associative registers, what is the effective memory reference time? (Assume that finding a page-table entry in the associative registers takes zero time, if the entry is there.)

Answer

(a) The effective instruction time is $100h + 500(1-h)$, where h is the hit rate. If we equate this formula with 200 and solve for h we find that h must be at least 0.75.

(b) What happens if the executable binary file is modified while the program is running? Using a mixture of pages from the old and new binaries is sure to crash it. While this event is unlikely, it is a calculated risk. Alternatively, the binary could be locked against modification while it was being used as a backing store.

(c) 400 nanoseconds: 200 nanoseconds to access the page table and 200 nanoseconds to access the word in memory. With a TLB the effective access time = 0.75 x (200 nanoseconds) + 0.25 x (400 nanoseconds) = 250 nanoseconds.

7. (a) For a 2-level page table system with pure paging detail the exact software and hardware implementation requirements.

(b) A 32-bit OS uses a 2-level page table scheme with 4K pages. The page table structures are stored in main memory (this is the hint!). Explain why p = 10 (10-bits to index top-level page table) and q =10 (10-bits to index the 2nd level page table) is the only solution of assigning p and q (such that p + q = 20) to minimise internal fragmentation?

(c) The OS in (b) gets a VA = 00D4D578H. Detail which page table structures and entries are accessed (in the event of a TLB miss) and how the PA is formed.

<u>Answer</u>

(a) *Software*: Main memory used to store the top-level and 2nd level page tables. Kernel code to handle memory management.
Hardware: TLB cache, Top-level page table pointer Register

(b) With p = 10 and q = 10 there are 2^{10} = 1024 entries in each page table. The top-level page table will require 32-bit entries to store the pointer to the 2nd level page table. The 2nd level page tables will also have 32-bit entries to conform to the 32-bit word boundary operation of the OS. Hence each page table is 1024 x 4 bytes = 4096 bytes = 4K in size which fits neatly in one page frame without any internal fragmentation. IT Engineers will ensure that is a deliberate design outcome not luck!

(c) VA = 00D4D578
➔ p = 00[11]H, q = [01]4DH, d = 578H
➔ p = 3, q = 333, d = 1400
Entry #3 in the top-level page table is indexed and the pointer to 2nd level page table #3 is retrieved. Entry #333 in 2nd level page table #3 yields the page frame number, f and the PA = [f | 578H], that is offset 1400 in page frame #f.

COS214 Tutorial 5

Roberto Togneri, 2000

SOLUTIONS

1. Discuss two approaches to the replacement of pages in a paged memory system. What information is required to optimise the decisions made in the algorithms you describe? Consider the following sequence of page references: 1 2 3 4 4 2 1 4 1 3 4. Determine how many page faults will occur for each algorithm, assuming there are only 2 physical page frames and that both are initially invalid.

Answer

Page replacement strategies:

- Optimal (need to predict future page references)
- Random
- FIFO (need a queue for selecting the oldest page)
- Least Recently Used / LRU (need timestamp of last reference)
- Second Chance FIFO (need R bit)
- Clock (need R bit)
- Not Recently Used / NRU (need both R and M bits)
- Not Frequently Used / NFU (need R bit and counter)

Optimal

Page requested	1	2	3	4	4	2	1	4	1	3	4
1	1	1*	3*	4	4*	4	4	4*	4	4	4
2		2	2	2*	2	2*	1*	1	1*	3*	3*
Page fault	p	p	p	p			p			p	

6 page faults

FIFO

Page requested	1	2	3	4	4	2	1	4	1	3	4
1	1	1*	3	3*	3*	2	2*	4	4	4*	4*
2		2	2*	4	4	4*	1	1*	1*	3	3
Page fault	p	p	p	p		p	p	p		p	

8 Page faults

LRU

Page requested	1	2	3	4	4	2	1	4	1	3	4
1	1	1*	3	3*	3*	2	2*	4	4*	3	3*
2		2	2*	4	4	4*	1	1*	1	1*	4
Page fault	p	p	p	p		p	p	p		p	p

9 Page faults

2. A segmented memory has paged segments, with each virtual address having a 2-bit segment number, a 2-bit page number and an 11-bit address within the page. The main memory contains 32Kb. Each segment is either read-only (ro), read execute (rx), read-write (rw) or read-write-execute (rwx). The page table and protection is as follows:

Seg 0		**Seg 1**		**Seg 2**		**Seg 3**	
ro		**rx**		**rwx**		**rw**	
Virtual Page	Page Frame	Virtual Page	Page Frame	Virtual Page	Page Frame	Virtual Page	Page Frame
0	9	0	disk	Page table not in memory		0	14
1	3	1	0			1	1
2	disk	2	15			2	6
3	12	3	8			3	disk

(a) How many virtual pages are there in the address space?

(b) How many physical pages can fit in main memory?

(c) For each of the following indicate what physical address is computed from the virtual address, if any. If a segment, page or protection fault occurs specify which one.

No	**Access**	**Seg**	**Page**	**Offset**
1	read	0	1	1
2	read	1	1	10
3	read	3	3	2047
4	write	0	1	4
5	write	2	1	2
6	write	1	0	14
7	jump to	1	3	100
8	read	0	2	50
9	read	2	0	5
10	jump to	3	0	60

Answer

(a) 4 segments with 4 virtual pages $\rightarrow$ 16 virtual pages

(b) 11 bits for offset $\rightarrow$ Page Size = 2^{11}=2048(2K) $\rightarrow$ 32K/2K = 16 physical pages

(c)

No	access	seg	page	offset	frame	phys addr	fault
1	read	0	1	1	3	3x2K+1 = 6145	
2	read	1	1	10	0	0x2K+10 = 10	
3	read	3	3	2047	-		page fault
4	write	0	1	4	-		Protection
5	write	2	1	2	-		Segment
6	write	1	0	14	-		protection
7	jump to	1	3	100	8	8x2K+100=16484	
8	read	0	2	50	-		page fault
9	read	2	0	5	-		segment
10	jump to	3	0	60	-		protection

3. Page replacement algorithms work best with hardware support. For example, many systems provide *Referenced (R)* and *Modified (M)* flags in hardware. Explain how these flags give the name *Second-Chance FIFO*. Next, explain how the flags could be simulated at only a small amount of expense in software.

Answer
In the modified FIFO algorithm the reference (R) bit is used to give a page a "second chance" in the FIFO.

The bits can be simulated as follows. When a process is started up, all of its page table entries are marked as not in memory. As soon as a page is referenced, a page fault will occur. The OS then sets the R bit (in its internal tables), changes the page table entry to point to the correct page, with mode READ ONLY, and restarts the instruction. If the page is subsequently written on, another page fault will occur, allowing the OS to set the M bit and change the page's mode to READ/WRITE.

4. A computer provides each process with 64K of address space divided into pages of 4K. A particular program has a text size of 32,768 bytes, a data size of 16,386 bytes, and a stack size of 15,870 bytes. Will this program fit in the address space? If the page size were 512 bytes, would it fit? Explain! Remember that a page may not contain parts of two different segments.

Answer
The text is 8 pages, the data is 5 pages (16,386 = 16K + 2 bytes → 4 + 1) and the stack requires 4 pages ➔ 17 pages
There are 16 pages in physical memory so the program does not fit.

With 512 byte pages, the text is 64 pages, the data is 33 pages (16,386 = 16K + 2 bytes = 32 + 1) and the stack requires 31 pages ➔ 128 pages
There are 128 pages in physical memory so the program now fits. A smaller page size results in reduced internal fragmentation and allows the program to fit in physical memory

5. Which of the following programming techniques and structures are "good" for a demand-paged environment ? Which are "not good"? Explain your answers.
(a) Stack
(b) Hashed symbol table
(c) Sequential search
(d) Binary search
(e) Pure code
(f) Vector operations
(g) Indirection

Answer
Demand paging is good in techniques or data structures which reference data in the same general vicinity (i.e. hit ratio on the same page is high), e.g. sequential access to an array versus random access.
The following techniques and data structures are good: (a), (c), (e), (f)
but the following are not so good: (b), (d), (g)

6. An operating system supports a paged virtual memory, using a central processor with a cycle time of 1 microsecond. It costs an additional 1 microsecond to access a page other than the current one. Pages have 1000 words, and the paging device is a drum that rotates at 3000 revolutions per minute, and transfers 1 million words per second. The following statistical measurements were obtained from the system:
- 1 percent of all instructions executed accessed a page other than the current page.
- Of the instructions that accessed another page, 80 percent accessed a page already in memory.
- When a new page was required, the replaced page was modified 50 percent of the time.

Calculate the effective instruction time on this system, assuming that the system is running one process only, and that the processor is idle during drum transfers. Assume that the drum rotational delay is half a revolution and seek time is negligible.

Answer
Time to read/write one page from swap = (rotational delay) + (transfer time)
3000 rev/min = 50 rev/sec = 0.02 sec/rev → average rotational delay = 0.01 sec
1000 words and transfer rate of 10^6 words/sec → transfer time = 1 µsec
∴ Time to read/write page = 0.01 sec + 1 µsec = 11,000 µsec

99% of instructions access current page =	0.99 x (1 µsec)
1% access another page, 80% of these in memory =	0.008 x (2 µsec)
other 20%, half don't need replacement =	0.001 x (11,000 µsec)
half do need replacing =	0.001 x (2 x 11,000 µsec)
Effective access time:	**34 µsec**

7. Consider the following page table:

Index	f	P	time of last access
0	--	0	--
1	0	1	100
2	--	0	--
3	2	1	99
4	--	0	--
5	--	0	--
6	1	1	80
7	3	1	115

What is the PA if VA = [p = 2 | d = 233] is referenced at time 120 and pages are 1K? Assume local scope LRU page replacement. Describe any changes made to the page table.

Answer
With p = 2, we get a page fault and need to perform page replacement of the local process. With LRU the candidate page is virtual page 6 / page frame 1. This page is evicted (virtual page 6 has P set to 0) and virtual page 2 now has f = 1 and P =1 and time of last access set to 120. The PA has f = 1 and d = 233 ➔ 1024 + 233 = 1257.

8. A 64-bit CPU has a VA address range of 2^{64} = 4GB x 4GB which is some really, really, really huge number. What minimum level of page tables do you think is reasonable assuming a 16K page? Comment on the performance cost of adopting such a scheme and is this an issue?

Answer

With a 16K page, d = 14 bits and 64 - 14 = 50 bits need to be distributed among the n-level page table indices. Assuming each page table has 64-bit = 8 byte entries and we would like to fit each page table in an integer number of page frames, then the number of bits used to index the page table should be no less than (16K/8) = 2048 entries = 2^{11} → 11 bits. With (50/11) = 4.5, 5-level page tables are implied, but one of these will require an index < 11 bits. Hence a 4-level page table scheme is the most appropriate with VA = [p | q | r | s | d] and p = 12, q = 12, r = 13, s = 13 and d = 14 bits.

Is there a serious performance cost? With 4-level page tables the memory access is now 5x as slow(!). However if a large enough TLB is available the TLB hit ratio can be quite high, say 98%, and this mitigate the costs of accessing the 4-level page table under normal operation.

COS214 Tutorial 6

Roberto Togneri, 2000

SOLUTIONS

1. (a) What is "device independence"?
   (b) In which of the four I/O software layers is each of the following done.
   (i) Computing the track, sector, and head for a disk read.
   (ii) Maintaining a cache of recently used blocks.
   (iii) Writing commands to the device registers.
   (iv) Checking to see if the user has permission to use the device.
   (v) Converting binary integers to ASCII for printing.

Answer

(a) Device independence means that files and devices are accessed the same way, independent of their physical nature. Systems that have one set of calls for writing on a file, but a different set of calls for writing on the console (terminal) do NOT exhibit device independence.

(b) (i) device driver
(ii) device-independent software
(iii) device driver
(iv) device-independent software
(v) user-level software

2. A user process is accessing data from a large file and performs 0.5 seconds of computation for each 128K of data read. The disk rotation speed is 5000 rpm, average seek time is 2 msec, and data transfer rate is 10 MB/sec. Memory to memory data transfer rate is 100 nsec per 4-byte word. Calculate the CPU time used by the user process for every 5 minutes wall-clock time, and the amount of memory used by kernel buffers, assuming:
   (a) no buffering
   (b) single buffer
   (c) double buffer
   Assume the process is the only active one in the system and the file is stored contiguously on the disk.

Answer

(a) The time taken for each I/O request cycle is T + C, where C = 0.5 sec and T is the time to satisfy the I/O request. T_s = 2 msec, T_r = 0.5(60/5000) = 6 msec,
T_d = (128x1024/10x1024x1024) = 12.5 msec, ➔ T = 20.5 msec
∴ T + C = 520.5 msec
→ CPU time used in 5 minutes = C / (T + C) x 300 seconds
→ = (500/520.5)(300) =288.2 seconds = 4.8 minutes.
→ the kernel does not allocate any buffers

(b) With a single buffer, the kernel driver can read the next 128K into the system buffer while the user process is accessing the current 128K of data. At the next I/O request the system buffer is copied to the user buffer:
Each I/O request cycle is max[T,C] + M, where C = 500 msec, T = 20.5 msec
and M is the memory-to-memory copy of 128K of data from system to user buffer,
M = (128K = 32,768 words) x 100 nsec/word = 3.28 msec
∴ max[T,C] + M = C + M = 503.28 msec.
→ CPU time used in 5 minutes = C / (C + M) x 300 seconds
→ = (500/503.28)(300) =298 seconds = 4.97 minutes.
→ the kernel allocates a buffer of 128K

(c) With a double buffer, the kernel driver can read the next 128K into one system buffer while the other system buffer is available immediately to be copied to the user buffer. Each I/O request cycle is max[T,C+M] = C + M = 503.28 msec and the CPU time in 5 minutes is 4.97 minutes. However the kernel needs to allocate two buffers of 128K each, 256K total. ***Think!*** So why use double buffering?

3. A local area network (LAN) is used as follows. The user issues a system call to write to the network. The OS then copies the data to a kernel buffer. Then it copies the data to the network controller board. When all the bytes are safely inside the controller, they are sent over the network at a rate of 10 Mbits/sec. When the last bit arrives, the destination CPU is interrupted, and the kernel copies the new data to a kernel buffer to inspect it. Once it has figured out which user they are for, the kernel copies the data to the user space. If we assume that each interrupt and its associated processing takes 1 msec, that packets are 1024 bytes (ignore the headers), and that copying a byte takes 1 μsec, what is the maximum rate at which one process can pump data to another? Assume that the sender is blocked until the work is finished at the receiving side and an acknowledgment comes back. For simplicity, assume that the time to get the acknowledgment back is so small it can be ignored.

Answer

A packet must be copied four times during this process, which takes 4 x 1024 x 1 μsec = 4.1 msec. There are also two interrupts (system call to send data, and interrupt when data is ready), which account for 2 msec. Finally, the transmission time is (1024x8)/(10x10^6) = 0.82 msec, for a total of 6.92 msec per 1024 bytes. The maximum data rate is thus 147,977 bytes/sec or 1.18 Mbits/sec, or about 12 percent of the nominal 10 Mbits/sec network capacity. (If we include protocol overhead the figures get even worse).

4. (a) A floppy disk has 40 cylinders (an old one anyway, most now have 80). A seek takes 6 msec per cylinder moved. If no attempt is made to put the blocks of a file close to each other, two blocks that are logically consecutive (i.e. follow each other in the file) will be about 13 cylinders apart on average. If, however, the operating system makes an attempt to cluster related blocks, the mean interblock distance can be reduced to 2 cylinders (for example). How long does it take to read a 100 block file in both cases, if the rotational latency is 100 msec and the transfer time is 25 msec per block?
   (b) Why are output files for the printer normally spooled on disk before being printed, instead of being printed directly from the application program?

Answer

(a) The time per block is built up of three components: seek time, rotational latency and transfer time. In all cases the rotational latency plus transfer time is the same, 125 msec. Only the seek time differs. For 13 cylinders it is 78 msec; for 2 cylinders it is 12 msec. Thus for randomly placed files the total is 203 msec, and for clustered files it is 137msec.

(b) If the printer were assigned as soon as the output appeared, a process could tie up the printer by printing a few characters and then going to sleep for a week.

5. Disk requests come in to the disk driver for cylinders 10, 22, 20, 2, 40, 6 and 38, in that order (Initially, the arm is at cylinder 20). A seek takes 6 msec per cylinder moved. How much seek time is needed for
   (a) First-Come, First Served (FCFS).
   (b) Shortest Seek First (SSF).
   (c) SCAN elevator algorithm (initially moving upwards)

Answer

Initially at cylinder 20 and moving up

(a) (10,22,20,2,40,6,38) → 10+12+2+18+38+34+32 = 146 cylinders = 876 msec.

(b) (20,22,10,6,2,38,40) → 0+2+12+4+4+36+2 = 60 cylinders = 360 msec.

(c) (20,22,38,40,10,6,2) → 0+2+16+2+30+4+4 = 58 cylinders = 348 msec.

6. A personal computer salesman visiting a university in Tasmania remarked during a sales pitch that his company had devoted substantial effort to making their version of UNIX very fast. As an example, he noted that their disk driver used the elevator algorithm and also queued multiple requests within a cylinder in sector order. A student, Harriet Hacker, was impressed and bought one. She took it home and wrote a program to randomly read 10,000 blocks spread across the disk. To her amazement, the performance measured was identical to first come first served. Was the salesman lying?

Answer

Not necessarily. A UNIX program that reads individual 10,000 blocks randomly issues the requests one at a time, blocking after each one is issued until after it is completed. Thus the disk driver sees only one request at a time; it has no opportunity to do anything but process them in the order of arrival. Harriet should have started up many processes at the same time to see if the elevator algorithm worked.

COS214 Tutorial 7

Roberto Togneri, 2000

SOLUTIONS

1. (a) The Macintosh OS allows file names of the form “Tutorial Six”, so does MS Windows 95. As the C shell with the UNIX OS uses a space as a command line separator what are the consequences of having a name with an embedded space? Could you have a UNIX filename with an end of line character in it for example?

   (b) A company uses a mixture of UNIX workstations and Pentium PC’s running Windows 95 connected to the same network. For ease of administration user accounts are stored on a UNIX file server and all machines have access to the server across the network. A user, on one of the UNIX workstations, creates a file and names it, *ChocolateChipCookies.doc*. The same user now wishes to edit that file on a Pentium PC using MS Word. Can this be done? If not, can you suggest a cure? Does it make any difference if the Pentium is running Windows 3.1x or Windows 95? What if the file was created on one of the Pentium systems and then accessed via a UNIX workstation?

Answer

(a) In UNIX files with non-printable (^L, ^M, ^V, etc) and non-alphanumeric (‘,!,?,;) can be created by specifying the ASCII code directly in the filename string. (usually via a C system call like create()) or, depending on the shell, specify the names enclosed by ‘ ’ or “ ”. Such files are hard to manipulate and especially remove.

(b) Win 3.1 only supports 8.3 file names. Win 95 will be okay as it support long file names BUT the UNIX server will need to support the format conversion since UNIX and Win 95 support long file names in different ways. The problem of accessing a file created on a Win 95 system from a UNIX workstation is less problematic since UNIX has greater filename flexibility (except for non-printable characters and different case conversion semantics).

2. (a) What is the purpose of a filename extension? How does UNIX know whether a file contains an executable program?

   (b) UNIX has many different file types. What is the purpose of this? Give examples of the different types.

   (c) List the typical attributes a files might have? Can you think of other attributes a file ought to have?

Answer

(a) File name extensions are basically useful to the user. In the case of Windows 3.x and 95 filename extensions are used to associate documents with programs. Under UNIX the magic number may be used to identify executable/text/binary etc file types, although this is not rigorously the case.

(b) Flexibility for process to process and process to peripheral interaction:
- regular files (-)
- directories (d)
- character (serial I/O, raw disk, tape device) (c)
- block (block I/O) (b)
- symbolic links (l)
- sockets (s)
- fifo (p)

(c) Typical and other attributes:
- filename
- read, write, execute access.
- setuid, setgid, for execute ownership
- filesize
- access/create/modify times
- hard links
- type of file
- read-only, hidden, system, archive (MS-DOS)
- icon, working directory, minimised (MS-WINDOWS)

3. (a) If */usr/jim* is the working directory, what is the absolute path name for the file whose relative path name is *../ast/x*?

   (b) What are the advantages and disadvantages of recording the name of the creating program with the file’s attributes (as is done in the Macintosh operating system)?

Answer

(a) */usr/ast/x*

(b) By recording the name of the creating program, the operating system is able to implement features (such as automatic program invocation when the file is accessed) based on this information. It does add overhead in the operating system and require space in the file descriptor, however. Furthermore, the creating program may have been removed, modified or may simply not be the program you want to use on the file.

4. (a) A file can be moved in two different ways:
   (i) Simply “rename” the file by updating the directory information
   (ii) Copying the file to the new location and deleting the original file
   Compare the two approaches. What are the corresponding UNIX commands?

   (b) Some systems automatically open a file when it is referenced (for reading or writing) for the first time, and close the file when the process terminates. Discuss the advantages and disadvantages of this scheme as compared to the more traditional one, where the user has to open and close the file explicitly.

Answer

(a) Renaming the file is faster than copying and deleting data, especially for large files. However it is usually limited to work within same filesystem and cannot be used to copy files across different filesystems or media. Under UNIX the *mv* command implements the rename operation (if on the same filesystem), *cp* implements the file copy and *rm* implements the file remove.

(b) Automatic opening and closing of files relieves the user from the invocation of these functions, and thus makes it more convenient for the user; however, it requires more overhead than the case where explicit opening and closing is required. Furthermore, it removes needed flexibility: What happens if the user needs to open a file in order to get the file handler (to pass to another routine, child process or thread) but does not otherwise to read or write to the file? What happens if the process first referencing the file passes the handler to a thread or child process to continuing processing and then terminates (and automatically closes the file even though the child process or thread is still accessing the data)?

5. (a) An operating system only supports a single directory but allows that directory to have arbitrarily many files with arbitrarily long file names. Can something approximating a hierarchical file system be simulated? How?
   (b) Directories can be implemented as "special files" that can only be accessed in limited ways, or as ordinary data files. What are the advantages and disadvantages of each approach?

Answer

(a) Use filenames such as */usr/ast/file*. While it looks like a hierarchical path name, it is really just a single name containing embedded slashes.

(b) Directories should be special files in that directory operations are different than file operations (e.g. you can't simply delete a directory file without first deleting all the entries), so the operating system should implement directories differently to ordinary files. The only disadvantage is the extra processing and data structures to support an additional file type.

6. (a) The UNIX file system supports hard and symbolic links. Is there an equivalent technique in MS Windows 95? Does this technique behave like a hard link or symbolic link?
   (b) What additional parameters should be passed to the file create function besides the name of the file?
   (c) It has been suggested that the first part of each Unix file be kept in the same disk block as its inode, rather than locating the inodes in the first part of the filesystem. What would the advantages be? And what are the disadvantages?

Answer

(a) Windows 95 supports "shortcuts" which are actually extra files stored on the disk containing information about what they point. Shortcuts act as symbolic links.

(b) create(filename, type of file, access permissions)

(c) *Advantage:* avoid long seeks as data and i-nodes are close; no limitation on the number of files that can exist
Disadvantages: no good for random access of larger files or accessing all of a large file; i-node entries are no longer together on the same area of disk making it longer to search for the location of i-node itself, especially if its placement is random.

COS214 Tutorial 8

Roberto Togneri, 2000

SOLUTIONS

1. (a) Consider a system that supports the strategies of contiguous, linked, and indexed allocation. What criteria should be used in deciding which strategy is best utilized for a particular file?
   (b) Fragmentation on a storage device could be eliminated by re-compaction of the information. Typical disk devices do not have relocation or base registers (such as are used when memory is to be compacted), so how can we relocate files? Give reasons why re-compacting and relocation of files often are avoided.

Answer

(a) *Contiguous* – if file is relatively small or never updated (i.e. write once, read only!).
Linked – if file is large and usually accessed sequentially.
Indexed – if file is large and usually accessed randomly.

(b) Relocation of files on secondary storage involves considerable overhead —data blocks would have to be read into main memory and written back out to their new locations. Relocation registers only make sense for contiguous allocation files, and most file-systems do not use contiguous allocation. Furthermore, many new files will not require contiguous disk space; even sequential access files can be allocated non-contiguous blocks if links between logically sequential blocks are maintained by the disk system. So fragmentation is not that much of a problem, especially with proper read and write caching.

2. Consider a file system on a disk that has both logical and physical block sizes of 512 bytes. Assume that the information about each file is already in memory. For each of the three allocation strategies (contiguous, linked, and indexed), how is the logical-to-physical address mapping accomplished in this system? Assume a pointer size of 1 byte. (For the indexed allocation, also assume that a file is always less than 512 blocks long.).

Answer

Let Z be the logical address of the file.
Contiguous. Divide Z by 512 with X and Y the resulting quotient and remainder respectively. Add X to the first block number of the file, S, to obtain the required physical disk block, S + X. Y is the displacement into that block.
Linked. Divide Z by 511 (byte 0 is the pointer) with X and Y the resulting quotient and remainder respectively. Chase down the linked list, reading X + 1 blocks, the last block read is the required physical disk block. Y + 1 is the displacement into this block (byte 0 is the pointer).
Indexed. Divide Z by 512 with X and Y the resulting quotient and remainder respectively. The required physical block address is contained in location X of the index block. Y is the displacement into the desired physical block.

3. Consider a file currently consisting of 100 blocks. Assume that the file control block (and the index block, in the case of indexed allocation) is already in memory. Describe the type of operations required for contiguous, linked, and indexed (single-level) allocation strategies, if, for one block, the following conditions hold. In the contiguous-allocation case, assume that there is no room to grow in the beginning, but there is room to grow in the end. Assume that the block information to be added is stored in memory.
   (a) The block is added at the beginning.
   (b) The block is added at the end.

Answer

(a) *Contiguous:* The 100 blocks have to be copied to another free partition in order to include the new block at the beginning.
Linked: The head of the linked list is updated to include the new first data block.
Indexed: The index entries have to be shifted down by one entry which may require copying to a new index. The new block address is added in as the first entry.

(b) *Contiguous:* The block is simply appended to the last block.
Linked: The linked list has to be read in its entirety to locate the last block and then the new block is added to the tail of the list.
Indexed: The new block address is simply added to the index table.

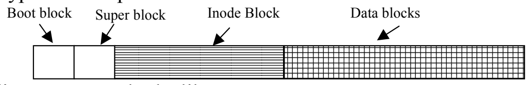
4. Consider a system where free space is kept in a free-space list.
   (a) Suppose that the pointer to the free-space list is lost. Can the system reconstruct the free-space list? Explain your answer.
   (b) Suggest a scheme to ensure that the pointer is never lost as a result of memory failure.

Answer

(i) In order to reconstruct the free list, it would be necessary to perform "garbage collection." This would entail searching the entire directory structure to determine which blocks are already allocated to files. Those remaining unallocated blocks could be relinked as the free-space list.

(ii) The free-space list pointer could be stored on the disk, perhaps in several places.

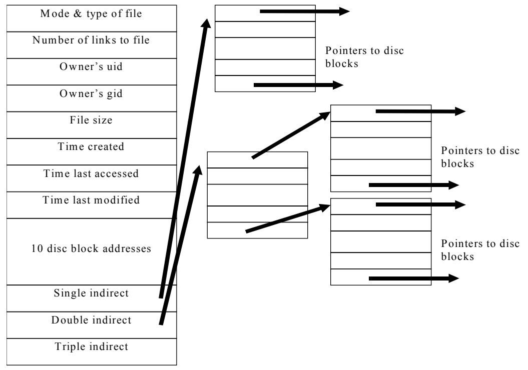
5. Consider a typical Unix partition:



The Unix directory entry looks like:

Inode Number 2 bytes	File name 14 bytes

The other information is kept in the inode entry:



Assume disk blocks are 8K bytes and that disc addresses are 32 bits

(a) How is a file opened? Use as an example the file */usr/home/test.c*.

(b) What size of file can be directly addressed from the information in the inode?

(c) What size of file requires a double indirect block?

(d) What is the largest possible file?

Answer

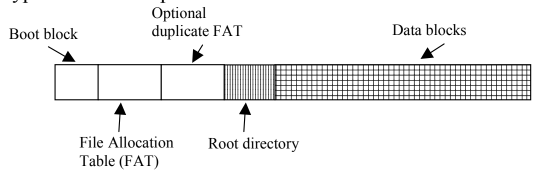
(a) First the root of the file system is located. In the filesystem the first block is reserved as the boot block, the next block is the superblock that describes the filesystem and then there are the blocks for the inodes (inode Block). Inode #1 is reserved for linking bad blocks to and inode #2 is reserved for the root of the file system. The inode #2 is used to locate the root directory data. The root directory is scanned for the "usr" entry which returns the "usr" inode number. The "usr" inode is retrieved from the Inode Block and the inode entry lists the data blocks for the "usr" directory file. These blocks are retrieved to form the "usr" directory. The "usr" directory is then scanned for the "home" entry which returns the "home" inode number. The same process is repeated to retrieve the "home" directory. The "home" directory is scanned for the "test.c" entry which returns the "test.c" inode number. The "test.c" inode is retrieved from the Inode Block and the inode entry is used to retrieve the data blocks for the "test.c" data file.

(b) disk blocks = 8k bytes; disk address = 32 bits
$\rightarrow$10 disk block address X 8k = 80k can be directly accessed.

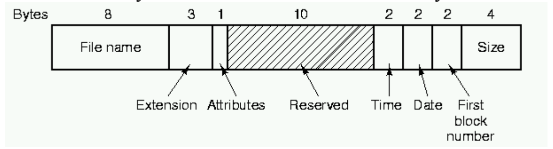
(c) *single indirect*: 1x 8k block of 32 bit addresses
$\rightarrow$ 8x1024 bytes / 4 bytes = 2048 addresses of 8k blocks = 16M
double indirect block: 2048x2048 addresses of 8k blocks = 32G
Can access a file of size no larger than 32G (+16M+80k)

(d) *triple indirect block*: 2048 x 32G = 64 Terabytes (+ 32G+16M+80k) $\rightarrow$ largest file

6. Consider a typical MS-DOS partition:



Both regular files and directory files can be defined and located by the FAT. Each directory contains an entry for each file within that directory:



Clearly show how the FAT is used to retrieve the data from the file *c:\windows\system.ini*.

Answer

The Root directory is scanned for the "windows" directory entry. The "windows" directory entry contains the first block number of the "windows" directory file. The FAT is used to retrieve the "windows" directory data. The "windows" directory is then scanned for the "system.ini" directory entry. The "system.ini" directory entry contains the first block number of the "system.ini" data file. The FAT is then used to retrieve the "system.ini" data.

7. A file system checker has built up its two lists as shown below:

Block #	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
In Use:	1	0	1	0	0	1	1	1	1	0	2	0	0	1	0
Free:	0	0	0	1	1	1	0	0	0	1	0	1	2	0	1

Are there any errors? If so, are they serious? Why?

Answer

Errors:

1. *block #1* $\rightarrow$ not on free list, not used $\rightarrow$just add block to free list, not serious
2. *block #5* $\rightarrow$ on free list, also used $\rightarrow$ remove from free list, not serious
3. *block #10* $\rightarrow$used twice! that means there are two inodes referencing data block. If not a "hard linked" data block, this is serious since two files are sharing the same data and they shouldn't be $\rightarrow$ copy data block so each inode references a different data block, one or both files possibly corrupted and may have to be deleted
4. *block #12* $\rightarrow$ freed twice! this means the free list is corrupted or incorrect and will need to be rebuilt, this should not be serious.

COS214 Tutorial 9

Roberto Togneri, 2000

SOLUTIONS

1. (a) Imagine there's a twin track railway line with a single track section going through a tunnel. How would you design a system that would ensure that only one train at a time enters the tunnel in any one direction? What are the attributes of your solution? For example, does this allow trains from one direction through even it there are no trains from the other direction?

   (b) In a restaurant there is a seating area for patrons with a narrow corridor (with a swing door at each end) for the waiters to gain access to the kitchens. It is so narrow that it will only allow for a waiter to go through it in one direction at a time. The manager installed a solution to the problem of collisions - a switch at each end that put a light on at the other end. If the light was on then you didn't enter the corridor. If the light was off, then you threw the switch, travelled through and then threw the switch at the other end to turn the light off. This worked fine, but yet there were still complaints with occasional accidents. The waiters complained that the system was unfair. Can you explain why there were still accidents and why the waiters complained?

Answer

(a)

Place an *entry detector* and an *exit detector* at each end of the tunnel with one pair for each track (i.e. 4 detectors). First carriage that hits *enter detector* trips a warning light at the other end and increments a train count register. When last carriage of the train passes the *exit detector* the train register count is decremented. If the count is zero then the warning light goes off and any opposite going train is allowed onto the single track. A similar system occurs for the trains going the other way. This solution will provide minimal interference with a series of trains entering the track as the count will keep track of them and wait until they are all through. To avoid collisions and possible starvation a decision program must be run to prevent two trains from entering simultaneously (crash!) and trains entering from the same direction after one another when an opposing train is waiting (opposing train may *starve*).

(b)

Unfair? If there is a queue of waiters waiting to go through then they can keep hitting the light switch and go through thereby preventing any waiters entering from the other end.
Problems? Waiters hit the light simultaneously or so quickly that they don't realise what has happened and enter the corridor at the same time leading to a collision.
Also: Since the light switch is a single action event then only one waiter can be permitted to travel through the corridor at any one time. If there is a queue of waiters wanting to go in the same direction it would make more sense to have them go together (but how would you do this?).

2. (a) What is meant by the term concurrent processing?
   (b) Give a definition of mutual exclusion. Give examples of where mutual exclusion is required for proper operation.
   (c) What is a race condition? What is a critical region?
   (d) What is meant by the term "deadlock"?

Answer

(a) *Concurrent processing*: Processes run in parallel or concurrently.
(b) *Mutual exclusion*: allow only one process to use a shared resource at any one time. Processes must finish using the resource before other processes are allowed access. E.g. printer, system email boxes.
(c) *Race condition*: condition where processes are reading and writing some shared data and the final result depends on who runs precisely when. How about if processes are only reading? One reading the other writing? Both writing?
Critical region: Part of a program where the shared memory is accessed. A shared file can be considered a special case of shared memory.
(d) *Deadlock*: process A enters critical region and sets up file lock/semaphore to prevent other processes from entering. Process A blocks or dies whilst in critical region without releasing the file lock/semaphore. Therefore, other processes will indefinitely block when attempting to enter critical region => deadlock.

3. For the following program fragments *assume that ProcessNumber is a shared integer variable initialised to 1; P1Inside,* P2Inside are shared Boolean variables initialised to false; and *P1WantsToEnter* and *P2WantsToEnter* are shared Boolean variables initialised to false. Further, assume each process is in some repeating loop and so continually, but at possibly differing rates, re-executes the code shown. For each of the three cases explain why they are unsatisfactory as solutions to the mutual exclusion problem.

Process One	Process Two

(a)
while(ProcessNumber==2);
CriticalSection();
ProcessNumber=2;
…
	(a)
	while(ProcessNumber==1);
	CriticalSection();
	ProcessNumber=1;
	…

(b)
while(P2Inside);
P1Inside=TRUE;
CriticalSection();
P1Inside=FALSE;
…
	(b)
	while(P1Inside);
	P2Inside=TRUE;
	CriticalSection();
	P2Inside=FALSE;
	…

(c)
P1WantsToEnter=TRUE;
while(P2WantsToEnter);
CriticalSection();
P1WantsToEnter=FALSE;
…
	(c)
	P2WantsToEnter=TRUE;
	while(P1WantsToEnter);
	CriticalSection();
	P2WantsToEnter=FALSE;
	…

Answer

(a) Processes MUST alternate. Big problem if one process takes considerably longer than the other to return to the critical section.
(b) If both processes execute the initial while loop simultaneously or process one sets P1Inside=TRUE just after process two has tested P1Inside then processes enter the critical region together ➔ race condition.
(c) Same as (b) except that processes will block ➔ deadlock.

4. You are designing a mutual exclusion primitive for a computer that does not have a test and set instruction. Instead it has an instruction that interchanges the contents of a register with a memory location in a single atomic action. Use this instruction to create primitives to enter and leave critical regions.

Answer

```
MOV A,0             ; initialise memory location to 0
XMEM A
…
enter_region:
        MOV A,1
        XMEM A
        CPI 00H
        JNZ enter_region
…
leave_region:
        MOV A,0
        XMEM A
```

5. From the earlier tutorial that gave the problem of the waiters in a restaurant colliding in a one way tunnel between the eating area and the kitchen, can you now think of how to solve this problem using a simple semaphore solution? Describe any limitations of your solution.

Answer

Simple semaphore solution:

```
semaphore corridor=1;

void waiter(void)
{
while(working) {
   process_dining( );
   down(&corridor);
   enter_corridor( );
   up(&corridor);
   process_kitchen( );
   down(&corridor);
   enter_corridor( );
   up(&corridor);
   }
}
```

Limitations? Only one waiter can be in the tunnel at any one time even if all are going in the same direction

6. A road bridge has only a single lane. To the North and South are feeder roads that are two laned. More than one car can be on the bridge if travelling in the same direction. Using semaphores, write the pseudo-code routines *enter_south*( ), *leave_south*( ), *enter_north*( ), *leave_north*( ) that arrange for cars to safely cross the bridge [**Hint:** Readers & Writers semaphore solution]. What are the attributes and problems of your proposed solution?

Answer

```
semaphore mutexS = mutexN = 1;
semaphore bridge = 1;
int south = north = 0;

enter_south( ){                              leave_south( ){
  down(&mutexS);                               down(&mutexS);
  south = south + 1;                           south = south - 1;
  if (south == 1) down (&bridge);              if (south == 0) up (&bridge);
  up(&mutexS);                                 up(&mutexS);
}                                            }

enter_north( ){                              leave_north( ){
  down(&mutexN);                               down(&mutexN);
  north = north + 1;                           north = north - 1;
  if (north == 1) down (&bridge);              if (north == 0) up (&bridge);
  up(&mutexN);                                 up(&mutexN);
}                                            }
```

Features? Allows more than one car in the same direction to travel on bridge, but without controlled alternation (i.e. cars waiting from the opposite side may be starved if there is always at least one car coming from the opposite direction).

COS214 Tutorial 10

Roberto Togneri, 2000

SOLUTIONS

1. The UNIX *ln* function is used to create a link to a file. Show how it can be used as an effective mechanism for mutual exclusive access to a file if there are multiple processes attempting to access a file. Are there are any problems with this solution?

Answer

Create a lock file as follows: *ln -s file file.lock*

(You can also use *touch file.lock*, but the *ln -s* allows the *file.lock* to be symbolically linked with *file*. A symbolic link is preferrable to a hard link for various reasons)

Each processes checks for and then attempts to create the lock file:

```
#!/bin/sh
…
while [ -f file.lock ]; do
   sleep 60
done
ln -s file file.lock
… {critical-section} …
rm file.lock
…
```

Problems?

1. If processes simultaneously execute the *while [-f file.lock]; do ...* test, all processes will be allowed through (race condition)! Fix? Test by attempting to run *ln -s file file.lock* and loop if an error condition that "*file already exists*" is returned.
2. If a process dies while in the critical section the file.lock will not be removed (deadlock).

2. The readers and writers problem can be formulated in several ways with regard to which category of processes can be started and when. Carefully describe three different variations of the problem, each one favoring (or not favoring) some category of processes. For each variation, specify what happens when a reader or a writer becomes ready to access the database, and what happens when a process is finished using the database. Carefully analyse the monitor solution presented in lectures, which variation does it represent?

Answer

Readers have Priority

No writer may start when a reader is active. When a new reader appears, it may start immediately unless a writer is currently active. When a writer finishes, if readers are waiting, they are all started, regardless of the presence of waiting writers. Problem? Writers may starve.

Writers have Priority

No reader may start when a writer is waiting. When the last active reader or writer process finishes, a writer is started, if there is one, otherwise, all the readers (if any) are started. Problem? Readers may starve.

Symmetric Priority

When a reader is active, new readers may start immediately. When a writer finishes, a new writer has priority, if one is waiting. In other words, once we have started reading, we keep reading until there are no readers left. Similarly, once we have started writing, all pending writers are allowed to run. Problem? Readers or writers may starve!

Monitor Solution

Implements a modified writers have priority, where when a writer finishes it allows any readers currently waiting to go through in preference to waiting writers. Problem? No starvation apparent since a current active writer defers to waiting readers and the arriving readers defer to waiting writers.

3. *The Cigarette-Smokers Problem*. Consider a system with three smoker processes and one agent process. Each smoker continuously rolls a cigarette and then smokes it. But to roll and smoke a cigarette, the smoker needs three ingredients: tobacco, paper, and matches. One of the smoker processes has paper, another has tobacco, and the third has matches. The agent has an infinite supply of all three materials. The agent places two of the ingredients on the table. The smoker who has the remaining ingredient then makes and smokes a cigarette, signalling the agent on completion. The agent then puts out another two of the three ingredients, and the cycle repeats. Complete the following program fragment to synchronize the agent and the smokers:

```
semaphore a[3] = 0;          /* a[0] for tobacco, a[1] for paper, a[2] for matches */
semaphore agent = 1;

Agent(void) {
   int   i,j;
   repeat
      i = 3 * drand48( );    /* returns a random integer 0, 1 or 2 for i */
      j = 3 * drand48( );    /* returns a random integer 0, 1 or 2 for j */
     while (i != j) {        /* i and j must be different */
          the rest of it
     }
until false;
}

Smoker(int r) {              /* r indicates which ingredients this smoker has */
   repeat
      the rest of it
   until false;
}
```

Answer

The shared data structures are:

```
semaphore a[2] = 0;      /* a[0] for tobacco, a[1] for paper, a[2] for matches, all initialised to 0 */
semaphore agent = 1;     /* initialised to 1 */
```

The agent process code is as follows:

```
Agent (void) {
   int   i,j,k;
   repeat
         /* i, j  represent the two materials the agent places on the table) */
         i = 3 * drand48( );   j = 3 * drand48( );
         while (i != j) {
               down(&agent);
               k = 3 - (i+j);          /* the smoker with the kth ingredient is identified */
               up(&a[k]);
         }
   until false;
}
```

Each smoker process needs the ingredient represented by the integer *r*.

```
Smoker(int r) {
   repeat
         down(&a[r]);
         smoke( );
         up(&agent);
   until false;
}
```

4. (a) Consider a system consisting of a number of processes attempting to access three identical line printers. Write a monitor that allocates line printers to these processes.

   (b) Parallel execution of list operations (e.g. max( ), min( ), sort( )) is trivially accomplished using a master-slave paradigm. Consider the case of one master and two slave processes. The master takes the list and divides it into two halves and passes one half to slave[0] and the other half to slave[1]. The slave processes then concurrently perform the list operation on their local list (e.g. find the local maximum for that half of the list) and return their result to the master (e.g. the local maximum). The master then merges the result from each slave to derive the global result (e.g. the maximum of each local maximum returned is the global maximum of the whole list). Describe the synchronisation between the master and the two slaves using message passing.

Answer

(a)

```
monitor Printers
  boolean P[0..2];          /* P[i] is true if printer i is busy */
  condition busy;

  procedure acquire (var printer-id: integer);
  begin
    if P[0] = true and P[1] = true and P[2] = true then wait(busy)
    if P[0] = false then printer-id := 0;
    else if P[1] = false then printer-id := 1;
    else printer-id := 2;
     P[printer-id]:= true;
end

  procedure release (printer-id: integer)
  begin
    P[printer-id]:= false;
    signal(busy);
  end

  P[0] := P[1] := P[2] := false;
end monitor
```

(b)

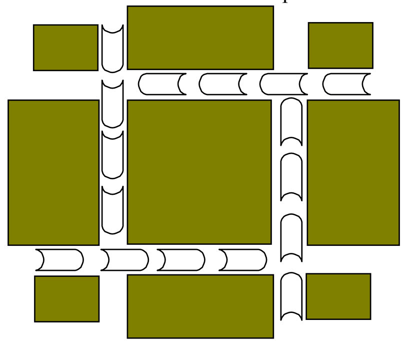
Master process:

```
…
send(slave[0], first half of list);
send(slave[1], second half of list);
for (i=0; i < 2; i++) {
   receive(any_slave, &result);
  /* merge result */
}
/* return global result */
```

Slave process *i* (for *i*=0, 1):

```
…
receive(master, half of the list);
/* derive local result */
send(master, result);
…
```

5. Consider the traffic deadlock depicted below:



(a) Show that the four necessary conditions for deadlock indeed hold in this example.
(b) State a simple rule that will avoid deadlocks in this system.

Answer

(a) Each section of the street is considered a resource.
Mutual-exclusion —only one vehicle allowed through each intersection.
Hold-and-wait — each queue of vehicles is occupying (holding) an intersection and the vehicle at the head of the queue is waiting to cross (acquire) the next intersection.
No-preemption — an intersection that is occupied by a vehicle cannot be taken away from the vehicle unless the car is able to move.
Circular-wait — each queue of vehicles is waiting to access the next intersection and the streets cross one another in a circular route.

(b) Allow a vehicle to cross an intersection only if it is assured that the vehicle will not need to wait whilst in the intersection (i.e. block the intersection). Not surprisingly, blocking an intersection is illegal even if you have the green light!

6. In an electronic funds transfer system, there are hundreds of identical processes that work as follows. Each process reads an input line specifying an amount of money, the account to be credited, and the account to be debited. Then it locks both accounts and transfers the money, releasing the locks when done. With many processes running in parallel, there is a very real danger that having locked account **x** it will be unable to lock **y** because **y** has been locked by a process waiting for **x**. Devise a scheme that avoids deadlocks. Do not release an account record until you have completed the transactions (e.g. don't lock one account and then release it immediately if the other account is found to be locked). Why is this last condition important?

Answer

To avoid a circular wait, use the account numbers as follows: After reading an input line, a process locks the lower-numbered account first, then when it gets the lock (which may entail waiting), it locks the other one. Since no process ever waits for an account lower than what it already has locked, there is never a circular wait, and hence no deadlock.
Releasing the lock on one account before the transaction is finished will lead to a race condition.

COS214 Tutorial 11

Roberto Togneri, 2000

SOLUTIONS

1. (a) A computer science department has a large collection of UNIX machines on its local network. Users on any machine can issue a command of the form:

   machine4 who

   and have the *who* command executed on *machine4*, without having the user log in on the remote machine. This feature is implemented by having the user's kernel send both the command and the *uid* to the remote machine. Currently the department's workstations are on the local network. It is proposed to allow students to also connect their PC to the network. Why is this a dangerous move?

   (b) One of the computer support people proposes to solve the security problem by restricting such commands to a group of trusted hosts. Is this solution fool-proof?

Answer

(a) The student PC can send a message to a departmental workstation asking it to carry out some command on behalf of, say, the super-user (*uid*=0). The machine receiving the message has no way of telling if the command really did originate with the super-user, or with a student.

(b) No! A student can set up a PC to masquerade as one of the trusted hosts! To avoid detection of multiply named hosts, the student can simply unplug the targetted workstation or force it to crash.

2. (a) After getting your degree, you apply for a job as director of a large university computer center that has just put its ancient operating system out to pasture and switched over to UNIX. You get the job. Fifteen minutes after starting work, your assistant bursts into your office screaming: "Some students have discovered the algorithm we use for encrypting passwords and posted it on the bulletin board." What should you do?

   (b) When a file is removed, its blocks are generally put back on the free list, but they are not erased. Do you think it would be a good idea to have the operating system erase each block before releasing it?

Answer

(a) Nothing! The password encryption algorithm is public since you require a secret key to perform the decryption. Furthermore, passwords are encrypted by the login program as soon as they are typed in, and the encrypted password is compared to the entry in the password file. So what you need to know is the actual password.

(b) From a security point of view it would be ideal since confidential information is not left lying around in the filesystem, but from a performance point of view it would generate a large number of disk writes and degrade performance. The compromise solution is for the system to only erase files which have a "security" attribute.

3. The Unix file system has *rwx* permissions for the user (i.e. owner), the group, and others. Write, in pseudo code, a function, *check*( ), that is given the user id (*uid*) and group id (*gid*) of the calling process, an operation (*op* = r, w, or x), the owner and group id of the file (*fuid, fgid*) and the set of 9 bits for the permissions (*perm*). It returns a boolean to indicate whether the operation is legal.

Answer

```
check (int uid, int gid, char op, int fuid, int fgid, int perm)
{
    int op_perm;

    switch(op) {
      case 'r':  op_perm= 4;       /* binary 100 */
        break;
      case 'w':op_perm= 2;         /* binary 010 */
        break;
      case 'x': op_perm= 1;        /* binary 001 */
        break;
      default: error( );
    }

    if (uid==fuid)
        op_perm=op_perm<<6; /* owner so check owner bits */
    else if(gid==fgid)
        op_perm=op_perm<<3; /* same group so check group bits */

    if (perm & op_perm)
          return(TRUE);
    else  return(FALSE);
}
```

4. (a) Explain what the UNIX *rwx* bits mean for:
   (i) a regular file
   (ii) a directory

   (b) Derive the UNIX file type and permissions, uid and gid:
   (i) to allow only the owner read/write access to a file?
   (ii) to allow only a selected group of users to create/delete/list/modify files in a directory?
   (iii) to allow only a selected group of users to list/modify files in a directory, but a single maintainer to create/delete files?

   (c) What can you say about the following UNIX file permissions:

(i)	drwx-wx--x	john	student
(ii)	-rw-rw-rw	root	sys
(iii)	-rwxr-x--x	root	wheel

Answer

(a)(i) *r* = can read/copy the contents of a file
w = can modify the contents of a file
x = can execute the file (i.e. file contains executable object code)

(ii) *r* = can list the contents of the directory, but with errors if the *x* is not set
w = can create/delete the contents of a directory, but only if *x* is also set
x = can access the contents of a directory and perform the *r* or *w* actions

(b) (i) -rw------- owner group
(ii) drwxrwx--- owner group
(iii) drwxr-x--- owner group

(c)(i) owner john can create/delete/list files in a directory, other student users can create/delete but not list the directory contents, all other users can access the contents of the directory but cannot list the contents.
(ii) all users can read/write the file.
(iii) owner root can copy/modify/execute the program file, users in group wheel can copy and execute the file, all other users can only execute the file.

5. Consider a system that supports 5000 users. Suppose that you want to allow 4990 of these users to be able to access one file. How would you specify this protection scheme in UNIX?

Answer
Put the 10 users you want to deny access to in one group (e.g. *noperm*) and set the file *perm* access to, say, 0404 (i.e. -r-----r-- owner *noperm* file)

6. For each of the following protection problems, indicate which mechanisms (ACL, C-lists or UNIX *rwx*) can be used most effectively:
(a) Ken wants his files readable by everyone except his office mate
(b) Mitch and Steve want to share some secret files
(c) Linda wants some of her files to be public.
(d) George wants his files read/writable by a close group of friends, readable by another group of friends, and inaccessible by everybody else.

Answer
(a) ACL attributes for files → (*Ken*, rw-) (*office-mate*, ---) (*, r--)
UNIX perm → -rw----r-- *Ken* noperm files
(*office mate* belongs to noperm group)
C-list not easy without a default attribute of (*, r--) for all domains

(b) ACL attributes for secret files → (*Mitch*, rw-) (*Steve*, rw-)
UNIX perm → -rw-rw---- *Mitch* secret files
(*Mitch* owns files, *Mitch* and *Steve* belong to secret group)
C-list attributes for *Mitch* and *Steve* domains → (files, rw-)

(c) ACL attribute for public files → (*, r--)
UNIX perm → --rw-r--r-- *Linda* student files
C-list not easy without a default attribute of (*, r--) for all domains

(d) ACL attribute for files → (*close_group*, rw-) (*group*, r--) (*,---)
C-list attribute for *close_group* domain → (files, rw-)
group domain → (files, r--)
UNIX perm does not allow this type of access.