CMPSCI 677: Operating Systems

Spring 2001

Solution to Written Homework 1

1. True or false questions. Justify your answer in one or two sentences.

   (a) On a uniprocessor, a CPU can execute one process while another process is performing a context switch.
   Ans: False. While performing context switch, system is saving (or loading) the state of the process including the content in CPU registers. It is impossible to execute another process.

   (b) It is desirable for a CPU scheduler to give priority to I/O-bound threads over CPU-bound threads.
   Ans: True. I/O-bound threads normally have short CPU-burst time and require short response time. Therefore it is desirable to have a higher priority for I/O-bound thread than for CPU-bound thread.

   (c) "Hold and Wait", one of the conditions required for deadlock to occur, means that the process is holding the CPU while waiting for a resource.
   Ans: False. "Hold and Wait" means the process is holding some resources(other than CPU) while waiting for more resources.

   (d) Thrashing occurs when there is not enough physical memory allocated to a process to keep the pages the process is actively using in memory.
   Ans: True. If the process does not have all its actively using pages in physical memory, it will very quickly page fault and must replace some page. However, if all pages are in active use, it will have to replace a page that will be needed again right away. Consequently, high paging activity and low CPU utilization (trashing) will occur.

   (e) Contiguous allocation is a reasonable way to store files on write-once disks, such as traditional CD-ROMs.
   Ans: True. Since it is write-once, we don't have to worry about dynamic storage allocation or disk fragments. Thus contiguous allocation is best and easiest way to store files.

   (f) Ethernet is commonly used to connect machines on a wide area network.
   Ans: False. Ethernet is used to connect machines on a Local Area Network.

   (g) All interactive time-shared systems are also multiprogrammed systems.
   Ans: True. In interactive time-shared system, when one process waits for I/O, CPU is switched to another process.

   (h) Traps are a common mechanism used by the OS to implement *all* of the following: system call, page fault, and illegal memory access.
   Ans: True. A trap is a software-generated interrupt. It is often used for request of operating system service from user program(system call) or errors(division by zero, page fault, illegal memory access, etc).

(i) Only a parent process can kill a child process.
Ans: False. A process can kill any other process if its process id is known, and the user has the privilidge to do so.

(j) A synchronization problem that requires counting semaphores can never be implemented using locks.
Ans: False. Counting semaphores can be implemented by (shared) variable counters and locks.

(k) Overlays allow contiguous memory allocation techniques to support process sizes that are larger than the size of physical memory.
Ans: True. As long as the memory needed at the same time does not exceed the size of physical memory, using overlay techniques can help.

(l) Compaction algorithms are required in a memory system that uses pure segmentation.
Ans: True. Otherwise there could be significant waste of external fragmentation.

(m) Thrashing is less likely if you use a global page replacement scheme.
Ans: False. On the contrary, use local page replacement may limit the effects of Thrashing.

(n) Direct memory access (DMA) transfers increase contention on the system bus.
Ans: False. Use DMA will not increase the total amount of traffic need to be transfered. It actually reduce the amount of bus traffic since with a single DMA request to the controller a large amount of data can be transferred. But in interrupt I/O, address must be specified for every word access.

(o) Since a kernel thread is a thread that the kernel knows about, kernel threads have faster context switches than user-level threads.
Ans: False. User level threads are managed through user-level libraries and doesn't need to trap to the kernal during a context switch. Therefore they have faster context switches.

2. Write short answers

(a) Since the shortest job first scheduling algorithm has provably optimal average waiting times, would you use it to schedule processes in a conventional operating system. Why or why not?
Ans: I would not use it to schedule processes in a conventional operating system. Although SJF is provably optimal in minimizing average waiting time, there is no way to know the length of CPU burst time for each of the processes. Another problem is long jobs may starve.

(b) What is LRU page replacement algorithm?
Ans: LRU stands for Least Recently Used algorithm. The idea is to replace the page that has not been used for the longest period of time. LRU replacement associates with each page the time of that page's last use. When a page must be

replaced, LRU chooses the page that has not been used for the longest period of time.

(c) Give a 1-2 sentence definition of a system call.
Ans: System call is the method used by a process to request action by the operating system such as process control, file or device manipulation and communication.

(d) Explain how direct memory access (DMA) works and list some of its advantages.
Ans: To initiate a DMA transfer, the host writes a DMA command block into memory. This block contains pointers to the source and destination of a transfer, a counter of the number of bytes to be tranferred. The CPU writes the address of this command block to the DMA controller, then goes on with other work. The DMA controller then proceeds to operate the memory bus directly, placing addresses on the bus to perform transfers without the help of the main CPU. Only one interrupt is generated at the end of transfer. Comparing to the scheme with no DMA support where an interrupt is generated for each word(or other memory access unit) transfered, DMA greatly reduces the total number of interupts needed for memory-device transfer. This could save lots of unnecessary operations (for example interrupt service routine has to save the contents of some CPU registers) so that CPU is available for other useful works. Thus using DMA is more efficient, especially for high speed device.

(e) What is a process control block (PCB) and what information does it store?
Ans: Process control block is used to represent a process in the operating system. It contains the information associated with each specific process, which includes: process state (new, ready, running, waiting or halted), program counter (PC), CPU registers, CPU schedulign information (such as process priority, pointers to scheduling queues, ...), memory-management information (such as page tables), accounting information(such as amount of CPU time used), I/O status information (List of open files, etc).

(f) Why does a pure paging scheme not suffer from external fragmentation?
Ans: In pure paging scheme, physical memory is broken into fixed-sized blocks(frames). Logical memory is also broken into broken into blocks of the same size(pages).When a process is to be execute, its pages are loaded into avaiable memory on the frame basis. Thus there won't be any external fragmentations. However, internal fragmentations exist.

(g) Consider an OS that improves disk performance using disk read-ahead. For what kinds of file access patterns is disk read-ahead useful? Why is prefetching using disk read-ahead easier than prefetching virtual pages into memory?
Ans: Disk read-ahead is useful for sequential disk access pattern. The assumption is that reading block n makes it more likely that block $n+1$ will be needed soon after. Thus, loading block $n+1$ into memory (or disk cache) in advance will reduce the access latency when block $n+1$ is actually requested. Since disk access is more likely to be sequential, the assumption holds very well. However, in the case of virtual page, predicting which page to prefetch might not be an easy task, and

even we can predict it fairly accurately, the page address needs to be translated into disk block address before prefetching. Thus, prefetching virtual pages is more difficult.

3. Consider a precedence graph in which program segment S_3 must execute only after S_1 and S_2, and S_4 and S_5 must execute only after S_3.

   Assume that each of the S_i's is executed in a separate process P_i. You may assume that the processes are unrelated (i.e., the processes need not be created), and that each process P_i has exactly one computation step S_i. Give the pseudo-code for the individual processes using each of the following: (a) Semaphores (b) Locks (c) UNIX fork() and waitpid() system calls. Your C program syntax need not be correct. But the use of the fork() and waitpid() system calls must be correct. You should permit the maximum amount of concurrency possible. Also, discuss the appropriateness of monitors to achieve this synchronization.
   Ans:
   (a)
   var sem_1, sem_2, sem_3: semaphore;
   sem_1, sem_2, sem_3 is initailized to 0.

P_1	P_2	P_3	P_4	P_5
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
		wait(sem_1);		
		wait(sem_2);	wait(sem_3);	wait(sem_3);
S_1;	S_2;	S_3;	S_4;	S_5;
signal(sem_1)	signal(sem_2)	signal(sem_3)		
		signal(sem_3)		
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

   Here wait is the P operation and signal is the V operation.
   (b)
   var $lock_1$, $lock_2$, $lock_3$, $lock_4$: lock;
   $lock_1$, $lock_2$, $lock_3$, $lock_4$ is initailized to BUSY.

P_1	P_2	P_3	P_4	P_5
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
		acquire($lock_1$);		
		acquire($lock_2$);	acquire($lock_3$);	acquire($lock_4$);
S_1;	S_2;	S_3;	S_4;	S_5;
release($lock_1$)	release($lock_2$)	release($lock_3$)		
		release($lock_4$)		
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

   (c)

   main()

```
{
  pid_t pid1,pid2,pid3,pid4,pid5;
  if ((pid1=fork())==0)
  {
    S1;
    exit(0);
  }
  if ((pid2=fork())==0)
  {
    S2;
    exit(0);
  }
  if ((pid3=fork())==0)
  {
    waitpid(pid1,NULL,0);
    waitpid(pid2,NULL,0);
    S3;
    exit(0);
  }
  if ((pid4=fork())==0)
  {
    waitpid(pid3,NULL,0);
    S4;
    exit(0);
  }
  if ((pid5=fork())==0)
  {
    waitpid(pid3,NULL,0);
    S5;
    exit(0);
  }
}
```

Use monitor with the support of condition variables, we can achieve this synchronization. However, S_1, ...,S_5 is only be excuted once and in a fixed order(condition construct is needed to ensure the order). Using monitor is not necessary indeed.

4. The local laundromat has just entered the computer age. As each customer enters, (s)he puts coins into slots at one of two stations and selects the number of washes she will need. The stations are connected to a central computer that automatically assigns available machines and outputs tokens that also identify the machines to be used. The customer places her laundry into a machine and inserts the appropriate token into the machine. When a machine is done with a wash, it informs the computer that it is

available again. The computer maintains a boolean array `available` to represent if a machine is available, and a semaphore `nfree` that indicates how many machines are available. The code to allocate and release machines is as follows.

All the elements of the `available` array are initialized to true, and `nfree` is initialized to NMACHINES.

```
semaphore nfree;
boolean available[NMACHINES];

allocate()
{
  P(nfree);
  for (int i=0; i< NMACHINES; i++)
    if (available[i] == TRUE) {
      available[i] = FALSE;
      return i;
    }
}

release(int machine)
{
  available[machine] = TRUE;
  V(nfree);
}
```

Explain how the program works. Does it work the way one would expect it to? If not, how can you modify it to work correctly?

Ans: The program works by first initializing a counting semaphore to NMACHINES. Each invokation of allocate() will first check the semaphore to see if there are free machines available, if yes it decrements the semaphore and return a free machine to the caller. Calling release() will return a free machine back to the pool of available machines, then it increments the semaphore to notify this event.

The program will NOT work as expected. This is because array *available* is shared among all the customers, customers may write to the *available* by calling allocate() or release(), so access to *available* must be protected to avoid data inconsistency. For example, when multiple customers simutaneously enter allocate(), since their execution orders may be arbitrarily intermingled, the allocate() function may return the same machine to more than one customer.

To remedy this problem, we need to introduce a mutex (lock or binary semaphore) variable to protect the access of the array *available*, as shown below:

```
semaphore nfree = NMACHINES;
semaphore mutex = 1;
boolean available[NMACHINES];

allocate()
{
  P(nfree);
  P(mutex);                // change
  for (int i=0; i< NMACHINES; i++)
    if (available[i] == TRUE) {
      available[i] = FALSE;
      V(mutex);            // change
      return i;
    }
  V(mutex);                // change

}

release(int machine)
{
  P(mutex);                // change
  available[machine] = TRUE;
  V(mutex);                // change
  V(nfree);
}
```

5. Consider a disk that employs the shortest seek time first (SSTF) scheduling algorithm. Assume that the disk has 100 tracks that are numbered from 1 to 100. Further assume that the current disk head position is track number 52.

   (a) In what order does the disk service requests for blocks that are stored on the following tracks: 67 40 19 82? Assume tht all four requests arrive simultaneously.
   Ans: According to shortest seek time first scheduling, the disk should serve the requests in the order of 40 19 67 82.

   (b) What is the total number of tracks across which the disk seeks to satisfy these requests?
   Ans: The total number of tracks across which the disk seeks is (52-40)+(40-19)+(67-19)+(82-67)=96.

   (c) What order are the requests serviced assuming the SCAN (elevator) disk scheduling policy? Assume that the disk head is currently moving towards track 100 and that the disk head always seeks to the highest and lowest numbered tracks (i.e., tracks 100 and 1) before reversing direction.

Ans: If SCAN disk scheduling policy is used, the disk will serve the requests in the order of 67 82 40 19.

(d) What is the total number of tracks across which the disk seeks in case of SCAN?
Ans: In the case of SCAN, the total number of tracks across which the disk seeks is (67-52)+(82-67)+(100-82)+(100-40)+(40-19)=129.

(e) Explain why SSTF scheduling tends to favor middle tracks over the innermost and outermost tracks of a disk.
Ans: In SSTF scheduling, if the disk (read/write) head is even likely to be above any track of the disk, then the expected number of tracks that the head travels over to complete a request of middle tracks is less than that of innermost or outmost tracks. For example, to satisfy a request on the track 50, the expect number of tracks across which the disk seeks is $\sum_{i=1}^{100} |50-i|/100 = 25$; while for request on track 100, the expect number of tracks is $\sum_{i=1}^{100} |100-i|/100 = 49.5$.

6.
- In standard uniprocessor `Unix`, explain why the following command will not produce a useful result.
`sort < foobar > foobar`
- What (erroneous) result will it always produce ?
Ans: This command always erases the content in "foobar" and results in an empty file "foobar". The reason for this could be that before the process is excuted, it first opens the file handle of "foobar" both for read and for write. When "foobar" is opened for write, it is truncated to zero length. Notice that the input and output use the same file handle. When the "sort" process starts to be excuted, the redirected standand input is merely an empty file. Thus nothing will be generated by "sort", which results an empty output "foobar".
- Why, and when, would the following command *sometimes* produce a useful result?
`sort < foobar | ( cat > foobar )`
where the paranthesis is `sh`-ell notation to execute the contained command in a sub-shell.
Ans: This command sometimes produces an empty file and sometimes gives a correct result. The reason lies in the precedures of how this command is executed. When shell reads this command, it creates a pipe with a pair of file handles: p-in and p-out. Then it creates two processes in parallel, one of which excutes the task "`sort < foobar`" and puts the output in p-in while the other reads input from p-out and performs "`cat > foobar`". However, the sequence of these two processes being executed depends on the scheduling for that moment. If "`cat > foobar`" gets the chance to be executed first, it will truncate "foobar" to zero length and produce a wrong result. Or if "`sort < foobar`" is executed first, it will put the output into pipe and when "`cat > foobar`" is executed, it reads the correct input from the pipe and output to file "foobar".