CIS 415: Operating Systems

Prof. Allen D. Malony

Midterm – November 3, 2015

NAME: ______________________________

Section	**Total Points**	**Points Scored**
1. Processes and Threads	5 + 5 + 5 + 5 + 5 + 5	/ 30
2. Scheduling	15 + 15	/ 30
3. Concurrency and Synchronization	5 + 5 + 5 + 5 + 20	/ 40
4. Deadlocks	5 + 5 + 15 + 15	/ 40
Total	60 + 80 = 140	

Comments:

1. Take a deep breath.
2. Write your name on the front page now and initial all other pages.
3. There are 4 sections. Do all sections. Each section is designed to take 20 minutes, more or less.
4. Concept questions in total are worth 60 points. Concept questions are intended to be short answer. If you are spending more than 5 minutes on ANY concept questions, you are writing too much! Move on!
5. If you have a question, raise your hand and we will come to you, if we can. Otherwise, we will acknowledge you and you can come to us.

Exams should be challenging learning experiences. Enjoy it!!!

1 Processes and Threads (30)

There are only concept questions in this section.

1.1 Concepts (30)

What do the terms *multiprogramming*, *multiprocessing*, and *multithreading* mean? Is it possible to do multiprogramming if the computer system only has 1 processor?

Multiprogramming is the ability of the OS to have multiple processes executing at the same time. Multiprocessing is the execution of multiple processes on multiple processors. Multithreading is the abiity for a process to have multiple threads of execution active and for the OS or user thread library to schedule them. Of course it is possible to do multiprogramming on a computer system with only 1 processor.

What is the purpose of system calls and how do they work? Give two examples of common system calls.

System call instructions enable user programs to invoke operations in the kernel. System call instructions result in an exception, which causes the CPU to vector into the OS. The OS saves the process state, detects that it is a system call exception, identifies the system call number from the argument to the instruction, and invokes the system call routine. When the routine finishes, the OS restores the process state and returns from the system call. Common examples are: read(), write(), fork(), open(), ...

What is a process control block and what is its purpose?

A PCB holds information about a process (called the *process context*) necessary to execute the process. The OS uses the PCB of a process store the process context when it is context switched off the CPU, and to restore the process context when it is context switched back onto the CPU.

What is the process address space? What is in it and where are things located? Draw a picture to show this.

The process address space is the logical space that is addressable by a process. The space is partitioned into areas for specific purposes, so as to separate the addresses that a process would use for the particular areas from each other. The areas are: stack, heap, data segment (BSS, global/static), text segment (executable code).

What is the difference between the 1:1 and M:1 threading models? Why might we prefer a 1:1 threading model?

These models define the number of user-space threads per kernel thread. 1:1 has one kernel thread for each user-space thread. M:1 has all the user-space threads on one kernel thread. The 1:1 model permits a user-space thread to block while allowing other user-space threads to run since each corresponds to a schedulable kernel thread.

What is a process? What is thread? How do they differ? What parts of the process address space are shared when using threads versus forking processes?

A process is a unit of work in a system. It is a program in execution. A processes has its own memory space. A process acts as a container for threads (1 or more), which share the process memory space, all of the file descriptors, and other process-level resources and attributes. The threads are the units of execution within the process, they posess a register set, stack, program counter, and scheduling attributes - per thread. Global variables, registers, page table, and so on.

2 Scheduling (30)

There are no concept questions in this section.

2.1 Just Take it Easy, We Will Get to You Eventually (15)

Per Brinch Hansen was a famous computer scientist working in the field of concurrent programming and operating systems. He developed an interesting scheduling strategy called *Highest-Response-ratio-Next (HRN)* that corrects some of the weaknesses in the shortest-job-first (SJF) strategy. HRN is a nonpreemptive scheduling discipline in which the priority of each job is a function not only of the job's service (run) time, but also of the amount of time the job has been waiting for service. Priorities in HRN are calculated dynamically according to the formula:

$$priority \ = \ \frac{timewaiting \ + \ servicetime}{servicetime}$$

What is a possible weakness of the SJF strategy that HRN is trying to correct?

The possible weakness of the SJF strategy is that there is excessive bias against longer running jobs and the favoritism toward short new jobs.

Is indefinite postponement a problem with SJF? How does HRN prevent it?

Certainly, indefinite postponement is a problem with the SJF algorithm because long jobs in the system may be continually postponed by shorter jobs arriving to the system. If the shorter jobs arrive before long jobs make it to the head of the job scheduling queue, the shorter jobs will take priority. This could happen indefinitely, especially when the system is being heavily used.

HRN prevents indefinite postponement by linearly increasing the priority of jobs the longer they stay in the sytem. For long jobs, those susceptible to indefinite postponement in SJF, the priority will be increase to point that it is greater than shorter jobs. Note, the shorter jobs still have preference because their priority increases at a faster rate than long jobs, but the *timewaiting* parameter limits the preference in cases of long waiting times.

How does HRN decrease the favoritism shown by other strategies to short new jobs?

HRN decreases favoritism to short new jobs by forcing all new jobs to start with a priority of 1, due to the fact that $timewaiting = 0$ as a new job enters the system. The priority of a new job is the lowest of all jobs that have been waiting. Thus, if a scheduling decision was made at the time of a new job, the choice would go to a waiting job, but this is true whether the new job is short or long. If a scheduling decision was made shortly after a new short job arrive, preference may very well be give to the short job, but it depends on the length of time spent waiting by other longer jobs. Favoritism to short new jobs is decreased by the influence of the *timewaiting* parameter on priority. At some point, long jobs waiting in the system will have priorities so high that the time a short job must stay in the system to have a higher priority will be longer than the average job scheduling interval. In this case, long jobs will begin to be scheduled, causing a decrease in preference to short new jobs.

Suppose two jobs have been waiting for about the same time. Are their priorities about the same? Explain your answer.

Clearly, two jobs that have been waiting for about the same time, say w, can have very different priorities, depending on their service times. The shorter job ($servicetime = t_s$)will have a higher priority than the longer job ($servicetime = t_l$) since $1 + (w/t_s) \ > \ 1 + (w/t_l)$, and, in fact, its priority will increase at a faster rate, $1/t_s$ vs. $1/t_l$.

2.2 Scheduler Cage Match (15)

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. For Round Robin, assume that the system is multiprogrammed, and that each job gets its fair share of the CPU. For the others, assume that only one job at a time runs, until if finishes. All jobs are completely CPU bound. Provide your answer in minutes.

Round Robin
$0 \leq t \leq 10$: each job gets 1/5 of CPU, at $t = 10$ C finishes
$10 < t \leq 18$: each job gets 1/4 of CPU, at $t = 18$ D finishes
$18 < t \leq 24$: each job gets 1/3 of CPU, at $t = 24$ B finishes
$24 < t \leq 28$: each job gets 1/2 of CPU, at $t = 28$ E finishes
$28 < t \leq 30$: each job gets 1/1 of CPU, at $t = 30$ A finishes
mean process turnaround time: 22 minutes

Priority Scheduling
finishing times: 6 (B), 14 (E), 24 (A), 26 (C), 30 (D)
mean process turnaround time: 18.8 minutes

FCFS (assumed arrival order 10, 6, 2, 4, 8)
finishing times: 10 (A), 16 (B), 18 (C), 22 (D), 30 (E)
mean process turnaround time: 19.2 minutes

SJF
finishing times: 2 (C), 6 (C), 12 (B), 20 (E), 30 (A)
mean process turnaround time: 14 minutes

3 Concurrency and Synchronization (40)

3.1 Concepts (20)

What does mutual exclusion mean?

Mutual exclusion is a condition on access to a resource that could be used by several processes. The condition states that only one process can access (hold) the resource at any time.

What is a critical section? List the requirements for a solution to the critical section problem.

A critical section is a piece of code that should only be execute by 1 process (thread) at a time, usually because it accesses a shared resource (data structure or device) that must not be concurrently accessed by more than one thread of execution.
The requirements for a solution to the critical section problem are:

- *mutual exclusion* – only one process can be executing in its critical section
- *progress* – the decision as to which process is allowed to enter its critical section next is made in a finite time
- *bounded waiting* – a process will be allowed (guaranteed) access to the critical section after a finite waiting time.

Why is atomic execution important for a synchronization construct to have?

Atomic execution means that the operations inside the synchronization construct will proceed without interference from other processes or threads. For the user of the synchronization construct, it allows them to reason about synchronization solutions at a higher level, without concern for atomicity of the construct's implementation.

The Facebook "CPU spinning" problem that Prof. Malony presented in class is an interesting case of a process wanting to be responsive while avoiding excessive use of the CPU. One idea that might help is to implement a blocking mutex, but with the following twist. Suppose that the *acquire()* routine accepts a boolean variable *block* which will cause blocking to occur if *block = 1* and the mutex is not available. However, if *block = 0* and the mutex is not available, no blocking will occur, and the *acquire()* will return a 0 to indicate that the mutex was not available. Otherwise, *acquire()* will return a 1. How might you use this "maybe" blocking mutex to address the dual objectives of responsiveness and efficiency?

Basically, this "maybe" blocking mutex gives you the decision to block or not. You can use it to address the dual objectives of responsiveness and efficiency by "spinning" for a few attempts (with *block = 0* passed in to *acquire()*) and then deciding to block (with *block = 1* passed in to *acquire()*). The idea would then be to put the call to *acquire()* in a "spin" loop. The number of iterations determines how long you decide to spin. However, you need to make sure to exit the loop if the return value ever equals 1. Also, if the return value never equals 1 during the loop, you need to call *acquire()* with *block = 1* after the loop to block. Otherwise, continue.

3.2 Starving Philosophers (20)

Once there were five philosophers whose life consisted of thinking and eating. They would sit around around a round table on which was set a large bowl of spaghetti, five plates (one for each philosopher) and five forks (one between each plate). A philosopher sitting at the table wishing to eat must use the two forks together on either side of the plate to take and eat some spaghetti.

Andrew Tanenbaum, a famous computer scientist working in the area of operating systems, presented the following solution to the dining philosophers problem is one of his operating systems books.

```
#define N           5           /* number of philosophers */
#define LEFT        (i-1) mod N  /* number of i's left neighbor */
#define RIGHT       (i+1) mod N  /* number of i's right neighbor */
#define THINKING    0           /* philosopher is thinking */
#define HUNGRY      1           /* philosopher wants forks to eat */
#define EATING      2           /* philosopher is eating */

typedef int semaphore;          /* semaphores are a special type of int */

int state[N];                   /* array to keep philosopher's state */
semaphore mutex = 1;            /* mutual exclusion semaphore */
semaphore s[N];                 /* semaphore per philosopher, initially 0 */

philosopher(i) {
  int i;                        /* philosopher number, 0 to N-1 */
  while (TRUE) {                /* repeat forever */
    think();                    /* philosopher is thinking */
    take_forks(i);              /* acquire two forks or block */
    eat();                      /* yum-yum, spaghetti */
    put_forks(i);               /* put both forks back on table */
  }
}

take_forks(i) {
  int i:                        /* philosopher number, 0 to N-1 */
  wait(mutex);                  /* enter critical region */
  state[i] = HUNGRY;            /* record that philosopher is hungry */
  test(i);                      /* try to acquire 2 forks */
  signal(mutex);                /* exit critical region */
  wait(s[i]);                   /* block if forks were not acquired */
}

put_forks(i) {
  int i:                        /* philosopher number, 0 to N-1 */
  wait(mutex);                  /* enter critical region */
  state[i] = THINKING;          /* philosopher has finished eating */
  test(LEFT);                   /* see if left neighbor can now eat */
  test(RIGHT);                  /* see if right neighbor can now eat */
  signal(mutex);                /* exit critical region */
}

test(i) {
  int i:                        /* philosopher number, 0 to N-1 */
  if ((state[i] == HUNGRY) &&
      (state[LEFT] != EATING) &&
      (state[RIGHT] != EATING)) {
    state[i] = EATING;
    signal(s[i]);
  }
}
```

a. Describe in words how this solution works.

In the proposed solution, an array, *state*, is used to keep track of whether a philosopher is eating, thinking, or hungry (trying to acquire forks). A philosopher may move only into the eating state if neither neighbor is eating. Philosopher i's neighbors are defined by the macros *LEFT* and *RIGHT*. The program uses an array of semaphores, one per philosopher, so hungry philosophers can block if the needed forks are busy; a hungry philosopher is then released by either his/her left or right neighbor. A single *mutex* semaphore is used for enforcing a critical region while the left and right neighbors' state is being tested. Note that each process runs the procedure $philosopher()$ as its main code, but the other procedures, $take_forks()$, $put_forks()$, and $test()$ are ordinary procedures and not separate processes.

b. Tanenbaum states: "The solution (above) is correct and also allows the maximum parallelism for an arbitrary number of philosophers." Although this solution does prevent deadlocks, it is not entirely correct because starvation is possible. Demonstrate this by counterexample.
Hint: *Consider the case of five philosophers. Assume that these are gluttonous philosophers – they spend almost no time thinking. As soon as a philosopher has finished one round of eating, he/she is almost immediately hungry. Then consider a configuration in which two of the philosophers are currently eating and the other three are blocked and hungry.*

The solution does prevent deadlocks, but starvation is possible. The first thing to note is that testing is always done under a mutual exclusion lock. So there is no problem of race conditions to modify shared variables. For starvation to occur though, there has to be at least one philosopher who cannot acquire forks. Consider a starving philosopher i and a state where upon entry to $take_forks(i)$ and after acquiring the $mutex$ lock there, $state[LEFT] == EATING$ or $state[RIGHT] == EATING$. In this case, philosopher i will call $test(i)$, find out the above condition is true, and will block when calling $wait(s[i])$. Now, the only way for philosopher i to get released from its blocked, waiting state on $s[i]$ is for philosopher $LEFT$ or philosopher $RIGHT$ to encounter the condition $((state[i] == HUNGRY)\&\&(state[LEFT]! = EATING)\&\&(state[RIGHT]! = EATING))$ when calling $test(RIGHT)$ (for philosopher $LEFT$) or $test(LEFT)$ (for philosopher $RIGHT$) from within the $put_forks()$ routine. Note, at the time of either one of these calls, we know that the state of philosopher i is $HUNGRY$ (afterall, he/she is waiting) and that the state of the calling philosopher ($LEFT$ or $RIGHT$) is $THINKING$. At this time, what do we know about the state of the other (non-calling) philosopher on the opposite side of philosopher i of the caller? Answer: nothing! If that philosopher is $EATING$, then philosopher i will not be signalled by the caller.

For starvation to be possible, we only need to imagine one sequence of thinking, fork-taking, and eating where a philosopher cannot acquire forks and subsequently dies. If we start with the above scenario (e.g., $state[i] = HUNGRY$, $state[LEFT] = EATING$, and $state[RIGHT] = THINKING$) and add the supposition that whenever philosopher $RIGHT$ ($LEFT$) exits $put_forks()$ then philosopher $LEFT$ ($RIGHT$) enters $take_forks()$ and exits (i.e., can acquire forks) before philosopher $RIGHT$ ($LEFT$) reenters $take_forks()$ (i.e., philosophers $RIGHT$ and $LEFT$ alternate eating), then it will *never* be the case that philosopher i's left and right neighbor will see the other neighbor of philosopher i not eating. This supposed scenario starves philosopher i.

4 Deadlocks (40)

4.1 Concepts (10)

What are the necessary conditions for a deadlock to occur?

- mutual exclusion
- hold and wait
- no preemption
- circular wait

What is the difference between deadlock prevention and deadlock avoidance?

Deadlock prevention techniques do not allow (prevent) deadlocks from occurring by ensuring that at least one of the necessary conditions cannot hold.

Deadlock avoidance techniques use information about the state of resource assignments and resource requests to control resource allocation, thereby avoiding possible future deadlocks.

4.2 Robin Hood Resource Allocation (15)

One day, while reading about the infamous Robin Hood and his Merry Humans, you suddenly got an idea for a new resource allocation policy. After changing into your green tights, you define the rules for the policy. You assume that requests and releases for resources are allowed at any time. If a request for resources cannot be satisfied because the resources are not available, then any processes that are blocked, waiting for resources, are checked. If they have the desired resources, then these resources are taken away from this "rich" process and are given to the "poor" requesting process. The vector of resources for which the waiting process is waiting is increased to include the resources that were taken away.

Ok, good enough. Now you make up an example. Consider a system with three resource types and the vector *Available* initialized to (4,2,2). If process P_0 asks for (2,2,1), it gets them. If P_1 asks for (1,0,1), it gets them. Then, if P_0 asks for (0,0,1), it is blocked (resource not available). If P_2 now asks for (2,0,0), it gets the available one (1,0,0) and one that was allocated to P_0 (since P_0 is blocked). P_0's *Allocation* vector goes down to (1,2,1), and its *Need* vector goes up to (1,0,1).

Can deadlock occur? If so, given an example. If not, which necessary condition cannot occur?

Deadlock will not happen because the *circular wait* condition can not occur. Consider the scenario where 2 of the processes are blocked. Because a process can satisfy a request either from the available resources or from blocked processes, the last unblocked process will always be able to meet its needs.

Can indefinite blocking occur?

Yes, it is possible that indefinite blocking can occur because there is nothing that guarantees that a process will not always encounter the unfortunate continuing circumstance where none of the other processes are blocked, but there are not enough available resources to satisfy its request.

4.3 "A ship in port is safe, but that's not what ships are built for." Grace Hopper (15)

You have been hired by the U.S. Navy to create a system to allocate different types of ships to a group of Navy Admirals who are trying to apply a new "efficiency" protocol of only using what ships are needed at any time in their war games. You created a system and now you are testing it out. Consider the following snapshot of the system. There are no current outstanding unsatisfied requests for ships (resources).

AVAILABLE			
R1	R2	R3	R4
2	1	0	0

	ALLOCATION				MAXIMUM				NEED			
PROCESS	R1	R2	R3	R4	R1	R2	R3	R4	R1	R2	R3	R4
P1	0	0	1	2	0	0	1	2				
P2	2	0	0	0	2	7	5	0				
P3	0	0	3	4	6	6	5	6				
P4	2	3	5	4	4	3	5	6				
P5	0	3	3	2	0	6	5	2				

a. Compute what each Admiral (process) still might request and display in the columns labeled "NEED."

AVAILABLE			
R1	R2	R3	R4
2	1	0	0

	ALLOCATION				MAXIMUM				NEED			
PROCESS	R1	R2	R3	R4	R1	R2	R3	R4	R1	R2	R3	R4
P1	0	0	1	2	0	0	1	2	0	0	0	0
P2	2	0	0	0	2	7	5	0	0	7	5	0
P3	0	0	3	4	6	6	5	6	6	6	2	2
P4	2	3	5	4	4	3	5	6	2	0	0	2
P5	0	3	3	2	0	6	5	2	0	3	2	0

b. Is this system currently in a safe or unsafe state? Why?

The system is in a safe state because we can come up with an order of process execution such that all of the processes can run to completion with the available resources. This order is as follows, with available resources after the indicated process finishes shown in parentheses: P1 (2,1,1,2), P4 (4,4,6,6), P5 (4,7,9,8), P2 (6,7,9,8), P3 (6,7,12,12).

c. Is this system currently deadlocked? Why or why not?

No, the system is not currently deadlocked because it is in a safe state. Deadlock is not possible if the current resource allocation results in a safe state. (Note, the question is somewhat weird because we do not have information about what resources are currently being requested and waited upon.)

d. Which processes, if any, are or may become deadlocked?

None of the processes are "deadlocked" at this time (again, we do not have any request information to detect deadlock), but certainly some may need to wait for additional available resources to proceed in the future. It is easy to see how the system might move into an unsafe state and, under assumption of resource requests, to deadlock. Granting of just one of R1 to P3 would cause the system to move into an unsafe state because we could not find a finishing sequence. It is also clear in this state that at least one of the needs of any one of the processes is greater than or equal to the available resources (even if P1 completes and returns all resources), and there exists at least two processes needing any particular resource. This state is shown below (assuming P1 has finished):

```
AVAILABLE
R1 R2 R3 R4
 1  1  1  2

         ALLOCATION   MAXIMUM       NEED
PROCESS  R1 R2 R3 R4  R1 R2 R3 R4   R1  R2  R3  R4
   P2     2  0  0  0   2  7  5  0    0   7   5   0

   P3     1  0  3  4   6  6  5  6    5   6   2   2

   P4     2  3  5  4   4  3  5  6    2   0   0   2

   P5     0  3  3  2   0  6  5  2    0   3   2   0
```

An easy way to kick the system into deadlock is to have each process request its need.

If a request from Admiral P3 arrives for (0,1,0,0), can that request be safely granted immediately? In what state (deadlocked, safe, unsafe) would immediately granting that whole request leave the system? Which Admirals (processes), if any, are or may become deadlocked if this whole request is granted immediately?

e. Given the (0,1,0,0) request of P3 and conducting the safety analysis, we see that the request could not be granted safely because no finishing sequence can be found. If granted, the resulting state of the system would be unsafe, but we cannot considered it deadlocked (as explained before).