CIS 415 (Fall 2023)

Prof. Allen D. Malony

Midterm – October 31, 2023

NAME: _____

Section	True / False	Multiple Choice	Short Answer	Problems	Total	Score
1. Processes and Threads	12	6	12	—	30	
2. CPU Scheduling	12	6	12	15	45	
3. Concurrency / Synchronization	12	—	12	30	54	
Total	36	12	36	45	129	

Comments:

1. Take a deep breath. You did your best to study and prepare. Congratulations!!! The hard part is over!
2. Write your name on the front page now and initial all other pages.
3. You have 80 minutes to do the midterm. There are 3 sections. Do all 3 sections. Each section is designed to take 25 minutes, more or less.
4. Concept questions (true/false, multiple choice, short answer) are 65% of the total points. Do these first!!! If you are spending more than 5 minutes on ANY concept questions, move on!
5. Problem questions appear in Sections 2 (1 question) and Section 3 (2 questions).
6. If you have a question, raise your hand and we will try 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

1.1 True/False (12)

It is not possible to have concurrent execution unless there are multiple CPUs in a computer system.

_____ TRUE __✓__ FALSE

The size of the process address space is determined by the amount of physical memory available in a computer system.

_____ TRUE __✓__ FALSE

The addresses used in CPU instructions when executing a program are referencing physical memory.

_____ TRUE __✓__ FALSE

A child process must executed exec() immediately after being after being forked by its parent process.

_____ TRUE __✓__ FALSE

1.2 Multiple Choice (6)

Which of the following are shared across threads in a multithreaded process?

_____ stack memory
__✓__ heap memory
_____ register values
__✓__ executable code
__✓__ global variables
__✓__ process address space

Select all valid IPC mechanisms.

_____ shared threads
_____ stacks
__✓__ sockets
__✓__ shared memory segments
__✓__ pipes
_____ PCB

1.3 Short Answer (12)

!!! CHOOSE ONLY 2 OF THE FOLLOWING CONCEPT QUESTIONS TO ANSWER !!!
NOTE: Each question is worth the same. You can choose to answer more than 2 questions, but only 2 will be counted towards your score. Please mark the questions you want graded with *.

Describe 2 forms of interprocess communication (IPC)? Give 2 advantages for each.

Shared memory. Through the use of shared segments or memory-mapped shared regions, processes that are running on the same machine are able to read and write to the same memory areas. This mechanism requires the OS to set up the shared segments or do the memory mapping (using `mmap()`). Once this has happened, read/write

operations to the memory shared between the processes happens at memory speeds. (The `mmap()` interface is more modern and has the nice property of being able to more easily control the access properties of the mapped address region.) Processes sharing the memory are responsible for maintaining its consistency. Pipes are another shared memory mechanism provided by the OS. The pipe buffer resides in the OS memory and `read()/write()` calls are used to access the pipe in a FIFO manner, with consistency maintain by the pipe implementation. Again, data access is fast because read and write operations are to memory.

Message passing. IPC based on message passing utilizes mechanisms in the OS to allow processes to send and receive messages. Either mailboxes or sockets are used. In each case, the messaging interface avoids issues of memory synchronization that are present in shared memory IPC. The IPC implementation handles all of the message passing details and buffer management for the processes. In the case of sockets, internet protocols will allow processes on different computer systems to communicate. Here, sockets use internet-based protocols (e.g., EDP, TCP/IP, RPC) and library to enable distributed processes to communicate.

Why are processes regarded as "heavyweight" and threads as "lightweight" from the perspective of being managed by the OS?

The notion of "weight" here has to do with 2 things: how much information the OS has to manage and the overhead of that management to do certain things (e.g., context switching). A process requires all information necessary for it to be able to execute to be stored in the process control block (PCB). The PCB is allocated in the OS memory. However, a thread of a process only needs incremental information about its state to be stored (in the thread control block (TCB)), specifically its CPU state. A thread shares the rest of the process state (from the PCB). Thus, threads are relatively lighter weight in terms of the TCB size, plus the TCB is stored in user memory. Because of this, the context switching of a process will have higher overhead than the context switching of a thread because the OS has more work to do to manage the PCB information. Furthermore, many more threads can be created than processes because the TCB is smaller and resides in the user memory. In summary, isolation is an important property for processes, but there is a price for process isolation in that it is more heavyweight.

What do the terms *user mode* and *system mode* mean? What are the reasons for being in one mode versus the other?

In general, the terms have to do with a mode of the CPU that allows certain instructions to be executed depending on which mode you are in. In particular, system mode lets the currently running code to execute instructions that will let it do more "privileged" things. Typically, it is only the OS that is allowed to run in system mode. Some of the privileged things have to do with accessing kernel data structures, controlling devices, and context switching processes. In user mode it is important to disallow certain operations that can violate process protection, manipulate shared data outside of the user process, and so on. In this way, the notion of "mode" extends beyond just allowing certain CPU instructions to be executed to include enabling certain (more privileged) software to be run.

What is the purpose of interrupts? Is it possible for a user program to disable interrupts? What is the common mechanism in a machine to get control back to the OS?

The main purpose of interrupts is so that the OS can be made aware of and respond to external events happening in the computer system. Interrups are hardware actions whereby the CPU will start executing OS code (i.e., the interrupt handler) associated with the specific interrupt when that interrupt occurs. In this way, the OS is able to take back control from the currently running process. In particular, a timer interrupt is a key mechanism to prevent a situation whereby no other interrupts occur and an errant process can take over control of the machine (e.g., entering an infinite loop). Interrupts can be disabled, but only by the OS. Otherwise, a user process could prevent the OS from every gaining back control, if all interrupts (especially the timer interrupt) are disabled.

2 Scheduling

2.1 True/False (12)

If there were no interrupts occurring in a computer system, it is possible that a user program could execute an infinite loop and lock out the OS.

__✓__ TRUE _____ FALSE

Assume that the jobs to be scheduled each take finite time. Round-robin scheduling optimizes (minimizes) average completion time (defined as the time from when a process enters the system to when it completes).

_____ TRUE __✓__ FALSE

Interactive processes should always be scheduled before any other processes regardless of priority.

_____ TRUE __✓__ FALSE

Increasing the priority of a process the longer it is in the system helps to address problems of indefinite blocking.

__✓__ TRUE _____ FALSE

2.2 Multiple Choice (6)

Which of the following scheduling disciplines can lead to starving of processes? Assume a system in which processes do not synchronize with each other or otherwise coordinate.

__✓__ strict priority _____ round-robin
__✓__ SJF preemptive _____ FCFS non-preemptive

In the exponential averaging technique for estimating shortest job first (SJF), which values of alpha (a) in the equation:

$$\tau(n+1) \;=\; \alpha * t(n) \;+\; (1-\alpha) * \tau(n)$$

takes history more into account.

__✓__ 1 _____ 0
_____ 0.5 _____ 1/n

Also give Some credit for 0.5 selection.

2.3 Short Answer (12)

!!! ANSWER ALL QUESTIONS !!!

List 3 actions that might cause the OS to make a scheduling decision.

1. Round-robin, quanta-based scheduling depends on timer interrupts.
2. A process makes a system call to perform file I/O which might result in that process blocking until the I/O is completed. In that case, the OS will make a decision to schedule another process.
3. Various actions during a process execution can cause the OS to regain control and decide to schedule another process. A segmentation fault is a good example. The process in this case will halt and the OS must make a scheduling decision.

What is the quantum in Round-Robin scheduling? How does it get implemented? What happens in Round-Robin scheduling as the quantum increases? What happens in Round-Robin scheduling as the quantum decreases?

The quantum represents that amount of time the RR scheduler will allot to a process to run on the CPU before it will preempt it. The preemption mechanism is implemented using a clock interrupt to enable the OS to regain control. Because there is overhead in responding to the clock interrupt, as well as process context switching, the quantum size matters. Increasing the quantum can reduce responsiveness and increase average waiting time. Too big of a quantum results in the RR scheduler approximating FCFS. Decreasing the quantum can make scheduling less efficient because relatively more time will be spent in overhead versus executing the processes.

2.4 Problems (15)

Bursting the CPU Bubble!

Consider the following set of processes, their CPU burst times, their arrival times, and their priorities (where a lower number represents a higher priority).

Process ID	CPU Burst Time	Arrival Time	Priority
P_1	6	0	2
P_2	3	4	3
P_3	4	5	1
P_4	1	7	6

(a) Fill in the Gantt chart (timeline) below for the task schedule that would be generated using a First-Come, First-Serve scheduler. Each cell represents 1 time unit, starting at time 0. Put an 'X' in a time slot when a process is running.

P_i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
P_1	X	X	X	X	X	X													
P_2							X	X	X										
P_3										X	X	X	X						
P_4														X					

(b) Fill in the Gantt chart (timeline) below for the task schedule that would be generated using a preemptive Shortest Job First scheduler. (Each cell represents 1 time unit, starting at time 0.) Put an 'X' in a time slot when a process is running.

P_i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
P_1	X	X	X	X	X	X													
P_2							X		X	X									
P_3											X	X	X	X					
P_4								X											

(c) Fill in the Gantt chart (timeline) below for the task schedule that would be generated using a preemptive priority scheduler. (Each cell represents 1 time unit, starting at time 0.) Put an 'X' in a time slot when a process is running.

P_i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
P_1	X	X	X	X	X					X									
P_2											X	X	X						
P_3						X	X	X	X										
P_4														X					

3 Concurrency and Synchronization

3.1 True/False (12)

The existence of a race condition in the code implies that there is an execution schedule that will result in deadlock.

_____ TRUE __✓__ FALSE

Consider a system that has multiple kernel threads, but only one processor. You are writing an application and decide to use a shared data structure between threads. Knowing that the CPU can only be running 1 kernel thread at a time, you reason that there is no need to protect the shared data structure with synchronization mechanisms like mutexes. Is your reasoning is correct (True) or incorrect (False).

_____ TRUE __✓__ FALSE

Solving the critical section problem is impossible without hardware-based synchronization mechanisms.

_____ TRUE __✓__ FALSE

The Dining Philosophers problem only requires synchronization when the number of philosophers is odd.

_____ TRUE __✓__ FALSE

3.2 Multiple Choice

!!! INTENTIONALLY LEFT BLANK (NO QUESTIONS) !!!

3.3 Short Answer (12)

!!! CHOOSE ONLY 2 OF THE FOLLOWING CONCEPT QUESTIONS TO ANSWER !!!
NOTE: Each question is worth the same. You can choose to answer more than 2 questions, but only 2 will be counted towards your score. Please mark the questions you want graded with *.

The Pthreads library provides the `pthread_mutex_trylock()` function that is equivalent to `pthread_mutex_lock()`, except that if the mutex object referenced by the mutex pointer passed to the function is currently locked (by any thread, including the current thread), the call immediately returns. What reasons to you see for using this function?

The `pthread_mutex_trylock()` is used when a thread does not want to block when accessing the mutex and the mutex is locked. One possible reason why is because the thread could do something else before it absolutely needs to access the mutex. In this case, it trys the mutex and, if the mutex is not available, it does those other things. At some point when it needs the mutex, the process will instead use `pthread_mutex_lock()`.

Why do we care about concurrency in operating systems? Is it advantageous to have logical concurrency even without physical concurrency (e.g., multiple CPUs)? Give one reason why.

The concept of concurrency in computer science is simply when more than one action can take place at the same (logical) time. We think of concurrency in the context multiple processes executing during the same (logical) time.

Because an operating system is managing a complex machine environment, it is advantageous for it to be able to logically separate actions taking place in the machine and allow those actions to be concurrent. By doing so, it can manage resources more efficiently and support logical operating abstractions that hide lower-level complexity. Logical concurrency does not require physical concurrency to have positive advantages. For instance, allowing logical concurrency is necessary for the OS to effectively orchestrate the sharing of machine resources (e.g., CPU, memory) across multiple processes.

Suppose we know that a mutex will be locked by a process for only a short period of time after it is acquired. Does it make sense to use a blocking mutex in this case? Why or why not?

The question here has to do with the strategy for waiting at a mutex: busy waiting or blocking. The general objective is to reduce the amount of time a process waits. There are 2 things to consider: how often the mutex is being accessed by processes and how long a process keeps the mutex locked. If there are many processes accessing the mutex, it is likely that a process will encounter the mutex to be locked. This is also true if the mutex is locked for a long time. Thus, it is not so simple. Knowing that the mutex is locked for a short period of time suggests that busy waiting is a good strategy. This is particularly true when there are multiple processors/cores. However, it the # processes accessing the mutex is large, the mutex could turn into a bottleneck and busy waiting would just consume wasted CPU cycles.

3.4 Problems (30)

Critical Sections – Believe It or Not!

Consider the following proposed solution to the critical section problem for two processes as published by Hyman in the Communications of the ACM in 1966:

```
int blocked[2]={0,0}; /* initially blocked[0] = 0, blocked[1] = 0 */
int turn=0;           /* initially turn=0 */

// process 0
while (1) {
  <code outside critical section>
  blocked[0] = 1;
  while (turn != 0) {
    while (blocked[1] == 1);
      ;
    turn = 0;
  }
  <CRITICAL SECTION>
  blocked[0] = 0;
  <code outside critical section>
}
```

```
// process 1
while (1) {
  <code outside critical section>
  blocked[1] = 1;
  while (turn != 1) {
    while (blocked[0] == 1);
      ;
    turn = 1;
  }
  <CRITICAL SECTION>
  blocked[1] = 0;
  <code outside critical section>
}
```

Was the CACM fooled into publishing an incorrect solution? If yes, find a counter example that demonstrates this.

Answer: Yes. Suppose that both processes have set their `blocked` flag and are executing the next `while` statement. If `turn` is 0, process 0 will be allowed to execute the critical section, but process 1 will execute the next `while` statement where it waits for `blocked[0]` to be set to 0. This will happen when process 0 exits the critical section, but suppose it then immediately wants to enter the critical section again and resets `blocked[0]` to 1. If this happens before process 1 sees that it was set to 0, process 1 will have to wait. If this continues, process 1 will be locked out and bounded waiting is not satisfied.

Barriers are Made to be Broken!

A *barrier* is a synchronization mechanism used to guarantee that all processes reach a common synchronization point in their execution before any process is allow to proceed further. Barriers are often used to satisfy a number of data dependencies simultaneously.

Informally, each process participating in the barrier performs the following general operations:

```
barrier:
  indicate somehow that THIS process has arrived at the barrier
  if (THIS process is the last process to arrive)
    then notify somehow other processes that the barrier
         synchronization has been satisfied and they can proceed
    else wait until notified somehow to continue executing
```

Assume that there are exactly N processes participating in the barrier. A barrier might be implemented using a semaphore by the following pseudo code. Each process would execute this code when they want to enter the barrier.

```
integer   count = 0;
semaphore barrier_sem = 1;  // semaphore is ready

barrier:
  wait(barrier_sem);
  count = count + 1;
  if (count == N)
    then {
      count = 0;
      signal(barrier_sem);
    }
    else {
      signal(barrier_sem);
      while (count > 0) ;
    }
```

Often more than one barrier synchronization is performed between cooperating processes during an execution. In the case where many barrier synchronizations will be performed, it would be good if the barrier implementable is reusable. Is your barrier implementation above reusable (i.e., can it be executed again for the next barrier synchronization)? Why or why not?

Answer: No, the barrier implementation above is not reusable. Consider the last process arriving at the barrier. If it sets `count` to 0 and then re-enters the barrier before the other processes waiting test `count`, all processes end up busy waiting forever.

If it is not reusable, rewrite it to make it reusable. (HINT: To make the barrier reusable, we need to guarantee that all processes are aware that the barrier synchronization has been reached before any processes is allowed to re-enter the barrier. Think about using two another counters, one to count the number entering and another to count the number leaving. The first part of the code is the same, except it uses the entering counter. An additional part is needed after it.)

Answer:

```
integer   enter_count = 0;
integer   leave_count = 0;
semaphore barrier_sem = 0;

barrier:
  wait(barrier_sem);                    // exactly like before ...
  enter_count = enter_count + 1;        //   |
  if (enter_count == N)                 //   |
    then {                              //   |
      enter_count = 0;                  //   |
      signal(barrier_sem);              //   |
    }                                   //   |
    else {                              //   |
      signal(barrier_sem);              //   |
      while (enter_count > 0) ;         //   v
    }                                   // but with enter_count

  wait(barrier_sem);
  leave_count = leave_count + 1;
  signal(barrier_sem);
  if (leave_count == N)
    then leave_count = 0;
    else while (leave_count > 0) ;
```