EECE.4810/EECE.5730: Operating Systems

Spring 2017

Midterm Exam Solution

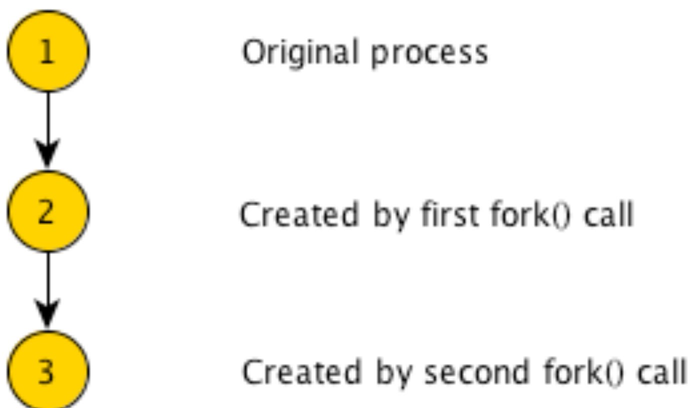
*1. (19 + **6** points)* ***<u>Process management</u>***
Parts (a) and (b) of this problem refer to the following program:

```
int main() {
   pid_t pid1, pid2;

   pid1 = fork();
   if (pid1 == 0) {
      pid2 = fork();
      if (pid2 > 0) {
         wait(NULL);
         printf("1\n");
      }
      else
         execlp("/bin/ls", "ls", NULL);
   }
   else if (pid1 > 0) {
      printf("2\n");
   }
   return 0;
}
```

a. (9 points) How many unique processes does this program create, including the initial process? Draw a process tree to support your answer.

<u>Solution:</u> This program creates 3 processes, one for each of the calls to `fork()`.



1 (continued)

b. (5 points) Will any of the created processes become zombie processes after their termination? If so, explain which one(s) and why; if not, explain why not.

<u>Solution:</u> The original process does not call `wait()`, so the second process (the one created by the first `fork()` call) will become a zombie process if it terminates before its parent.

c. (5 points) How does the operating system differentiate between processes in the ready and waiting states?

<u>Solution:</u> Processes in the ready state wait in the ready queue, which holds a list of all processes waiting for access to the CPU. Processes in the waiting state are kept in individual device queues for the I/O devices for which they are waiting. The state is tracked in the process control block (PCB) for each process.

*d. (6 points, **<u>EECE.5730 only</u>**) Suppose a single parent process creates five different child processes. When each child terminates, the parent process should handle each one in a different, specific way. Describe how the parent process can determine which child terminates so it can handle each one appropriately.*

<u>Solution:</u> Within the parent process, the `fork()` system call will return the PID of each created child, while the `wait()` system call will return the PID of the most recently terminated child process. After each `fork()` call, the parent process must store the PID of the newly created child (in an array, for example). After each `wait()` call, the parent should compare the returned PID against the list of stored PIDs to determine which child terminated.

2. (10 points) ***Inter-process communication***

a. (6 points) Describe how the two major classes of inter-process communication use memory. In your answer, describe (i) what types of data or data structures are used to exchange information, and (ii) in which address space these data are stored.

Solution: Processes using shared memory IPC communicate through (i) general shared variables in memory. Both communicating processes have access to the same data, which are stored in (ii) the address space of one of the processes. Typically, one process establishes the shared region in its own address space and then gives the other process access to that region.

Processes using message-passing IPC communicate through (i) communication links usually set up as message queues. The links are created in (ii) the kernel's address space.

b. (4 points) Explain why indirect communication is often favored over direct communication for message passing inter-process communication.

Solution: Indirect communication tends to perform better because it decouples the send and receive actions. When using direct communication, the sending process must wait for the receiving process to receive its message. When using indirect communication, the sending process must wait for the mailbox (communication link) to receive the message, thus typically allowing the sending process to return to other work sooner.

3. *(11 + **6** points)* ***Multithreading***

a. *(6 points) Explain why each thread in a multithreaded process needs its own (i) program counter, (ii) register copies, and (iii) stack.*

Solution:

(i) Each thread is a separate sequence of instructions. So, each thread executes instructions at different addresses tracked in that thread's program counter.

(ii) Each thread uses the same register names but is operating on different values, thus requiring each thread to keep its own copies of those values.

(iii) Again, each thread is a separate sequence of instructions. While multiple threads may call the same functions, those function calls—and therefore their stack frames—should operate on different data, thus requiring each thread to have its own stack.

b. *(5 points) Given the two threads below, is it possible for x to be equal to 4 and y to be equal to 2 when both threads are done? If so, explain how; if not, explain why not.*

Thread 1	*Thread 2*
x = 4	*y = 5*
x = y + 2	*y = x - 2*

Solution: The result x = 4 and y = 2 is possible, given the following interleaving of statements from the two threads and assuming (as the problem should have said) that x and y are shared between the two threads:

1. Thread 2: y = 5 -or- Thread 1: x = 4
2. Thread 1: x = 4 -or- Thread 2: y = 5
3. Thread 2: y = x – 2 = 4 – 2 = 2
4. Thread 1: x = y + 2 = 2 + 2 = 4

c. *(6 points, **EECE.5730 only**) Explain how (i) two threads in the same process could easily share data, and (ii) how each of those two threads can protect its data from the other thread. Your answer should not require any special mechanisms for sharing—placing the data appropriately should be sufficient.*

Solution: Multiple threads in the same process share parts of the address space (code, global data, heap) while maintaining their own copies of other parts (stack). Therefore, (i) shared data should be placed in the global data section (unless they're dynamically allocated in the heap), while (ii) protected data should be allocated inside functions and therefore in each thread's stack.

4. (21 points) ***Synchronization***

a. (5 points) Explain what busy waiting is and how it is typically avoided when processes need to wait for access to a synchronization primitive (lock, condition variable, etc.)

Solution: When threads or processes busy wait, they repeatedly check on the status of whatever they're waiting for. In the context of synchronization primitives, busy waiting means repeatedly attempting to acquire that primitive until finally being successful.

To avoid this performance-intensive process, synchronization primitives often have queues associated with them. A thread or process that cannot acquire the primitive adds itself to the queue and is woken up when the primitive becomes available.

4 (continued)

b. *(8 points) The example code below shows a pair of processes (one reader, one writer) that use semaphores to allow (i) multiple reader processes to run simultaneously, while also ensuring (ii) the writer process can only enter its critical section if no readers are executing. Assuming the two semaphores* `rwSem` *and* `rcSem` *are initialized to 1, and* `rdCnt` *is initialized to 0, explain how the two semaphores enable conditions (i) and (ii) above.*

Reader process

```
do {
   down(rcSem);
   rdCnt++;
   if (rdCnt == 1)
      down(rwSem);
   up(rcSem);

   /* read shared data */

   down(rcSem);
   rdCnt--;

   if (rdCnt == 0)
      up(rwSem);
   up(rcSem);
} while (true);
```

Writer process

```
do {
   down(rwSem);

   /* write shared data */

   up(rwSem);

} while (true);
```

Solution: Recall that calling `down()` on a semaphore with value < 1 causes the calling process to block, thus implying that a process can lock access to a critical section when it acquires a semaphore with value 1 and decrements it to 0. Another process requesting that same semaphore cannot proceed until the original process calls `up()` to increment the semaphore. Therefore:

(i) (Relevant code is in red) The semaphore `rcSem` controls access to the critical sections of the reader process. Those critical sections only involve access to the shared `rdCnt` variable, which tracks the number of readers and is necessary to determine when the writer process can access its critical section. However, since each reader process increments `rcSem` prior to actually reading data, multiple processes can be in the reading section simultaneously.

(ii) (Relevant code is in blue) The semaphore `rwSem` controls access to the critical section of the writer process—the editing of data, during which no other process should access the shared data. Locking and unlocking access in the writer process is relatively straightforward. In the reader process, the value of `rdCnt` determines when the writer is locked out of or allowed access to its critical section. The writer only needs to be locked out when the first reader enters—not for every reader—so `rwSem` is decremented only if `rdCnt` is 1. Likewise, the writer will be given access once all readers are done, so `rwSem` is incremented only if `rdCnt` is 0.

4 (continued)

c. *(8 points) Consider a multithreaded bank software package in which the following function is used to transfer money. Assume that the program contains a global array of locks,* `locks[]`*, with one entry per account number, such that* `locks[i]` *is the lock for account number* `i`*.*

```
void transfer_money(int src_acct, int dest_acct, int amt) {
     locks[src_acct].lock();    // Lock account sending money
     locks[dest_acct].lock();   // Lock account receiving money
     <transfer money>
     locks[dest_acct].unlock();
     locks[src_acct].unlock();
}
```

Explain how this function can deadlock if called in multiple threads, and rewrite (in pseudo-code or actual code) the function to remove the deadlock condition.

Solution: Deadlock can occur if two threads access the same accounts, with each account used as the source in one thread and the destination in the other—for example, if thread 1 calls `transfer_money(1, 2, 1000)` and thread 2 calls `transfer_money(2, 1, 500)`. If each thread obtains its source account lock before the other obtains its destination account lock, deadlock will occur, since neither thread releases any locks until after it has both. In the specific example above, that could happen with the following statement interleaving:

```
T1: locks[1].lock();     // Thread 1 acquires locks[1]
T2: locks[2].lock();     // Thread 2 acquires locks[2]
T1: locks[2].lock();     // Thread 1 blocks
T2: locks[1].lock();     // Thread 2 blocks
```

A valid solution should resolve one of the two preventable conditions: hold and wait or circular wait. Resolving hold and wait means the thread should release all locks if it can't acquire both, as shown in the pseudo-code below that replaces the two `lock()` calls in the original function:

```
while (both locks not acquired) {
  locks[src_acct].lock();                 // Lock src account
  if (locks[dest_acct].lock() fails)      // Try to lock dest
    locks[src_acct].unlock();             // Release src on
}                                         //  failure
```

To resolve circular waiting, impose an order on the locks so that all threads acquire locks in the same order. The solution below always acquires the lowest-numbered lock first:

```
if (src_acct < dest_acct) {
  locks[src_acct].lock();
  locks[dest_acct].lock();
}
else {
  locks[dest_acct].lock();
  locks[src_acct].lock();
}
```

5. *(20 points)* ***Scheduling***

Consider the following set of processes, with the length of the CPU burst time given in milliseconds. Processes may begin executing 1 ms after they arrive (i.e., a process arriving at time 5 could start executing at time 6). Any process arriving at time 0 is in the ready queue when the scheduler makes a decision about the first process to run.

Process	Burst	Priority	Arrival time
P1	10	1	0
P2	3	4	0
P3	7	2	2
P4	1	2	4
P5	5	3	6

Determine the turnaround time for each process using each of the following scheduling algorithms: (i) first-come, first-serve (FCFS), (ii) shortest job first (SJF), (iii) shortest time to completion first (STCF), (iv) round-robin (RR) with time quantum = 1 ms, and (v) a non-preemptive priority scheme in which lower numbers indicate higher priority.

To break ties between processes (same burst time/priority), use first-come, first-serve ordering.

Solution: This problem is very similar to HW 3 question 1a, with a couple of major differences:

- The homework did not include STCF, which is a preemptive scheduling metric. When each new job arrives, it will preempt the currently running job if
- Varying arrival times means:
  - Turnaround time is not necessarily the same as end time—remember, turnaround time is defined as the time from arrival to completion, so the turnaround time for each process listed here is (end time) – (arrival time).
  - All processes may not be available for scheduling when you need to choose the next process. For example, SJF starts with P2 because it's the shortest process available at time 1.

Accordingly, for each scheduling metric, the solution below shows three columns: start time (St), end time (End), and turnaround time (TT). As with the example in class, a process with a burst time of 1 starts and ends in the same cycle. You may have used slightly different notation.

	(i) FCFS			(ii) SJF			(iii) STCF			(iv) RR			(v) Priority		
Proc	**St**	**End**	**TT**	**St**	**End**	**TT**	**St**	**End**	**TT**	**St**	**End**	**TT**	**St**	**End**	**TT**
P1	1	10	**10**	17	26	**26**	17	26	**26**	1	26	**26**	1	10	**10**
P2	11	13	**13**	1	3	**3**	1	3	**3**	2	10	**10**	24	26	**26**
P3	14	20	**18**	4	10	**8**	4	11	**9**	3	23	**21**	11	17	**15**
P4	21	21	**17**	11	11	**7**	5	5	**1**	7	7	**3**	18	18	**14**
P5	22	26	**20**	12	16	**10**	12	16	**10**	8	21	**15**	19	23	**17**

Detailed schedules for the two preemptive schemes (STCF and round robin) are shown on the next page. (See the note about a possible different result for round robin scheduling.)

5 (continued)

Detailed schedule for STCF, with the time in that process alone in parentheses (for example, "P1 (3-5)" would indicate a 3 ms block in which P1 ran for its 3rd, 4th, and 5th milliseconds):

Time	1 3	4	5	6 11	12 16	17 26
Process (time)	P2 (1-3)	P3 (1)	P4 (1)	P3 (2-7)	P5 (1-5)	P1 (1-10)

Detailed schedule for round robin, with the final cycle for each process shown in bold:

Time	1	2	3	4	5	6	**7**	8	9	**10**	11	12	13	14	15	16
Process	P1	P2	P3	P1	P2	P3	**P4**	P5	P1	**P2**	P3	P5	P1	P3	P5	P1

Time	17	18	19	20	**21**	22	**23**	24	25	**26**
Process	P3	P5	P1	P3	**P5**	P1	**P3**	P1	P1	**P1**

Note: When a new process arrives, it is added to the end of the scheduling queue. A question arose during the exam as to how that queue is handled in round robin scheduling. The above solution uses what I believe is the simplest method—order the queue based on arrival order, so the processes are simply added from P1 to P5. However, if you consider the "end" of the queue to be the spot before the current task (assuming the queue is implemented as a circular linked list), then the queue grows as follows:

- Initially: P1 → P2
- Time 2 (P3 arrives): P2 → P1 Add P3; queue = P2 → P1 → P3
- Time 4 (P4 arrives): P3 → P2 → P1 Add P4; queue = P3 → P2 → P1 → P4
- Time 6 (P5 arrives): P1 → P4 → P3 → P2 Add P5; queue = P1 → P4 → P3 → P2 → P5

Managing the queue in that manner would lead to the following schedule:

Time	1	2	3	4	5	6	**7**	8	**9**	10	11	12	13	14	15	16
Process	P1	P2	P1	P3	P2	P1	**P4**	P3	**P2**	P5	P1	P3	P5	P1	P3	P5

Time	17	18	19	20	21	**22**	23	**24**	25	**26**
Process	P1	P3	P5	P1	P3	**P5**	P1	**P3**	P1	**P1**

6. *(19 points)* ***<u>Memory management</u>***
a. *(4 points) Describe one benefit of base and bounds memory management over segmentation, and one benefit of segmentation over base and bounds.*

<u>Solution:</u> Base and bounds is the simplest form of address translation, requiring no additional information beyond hardware registers to store the base and bounds of the current address space.

Segmentation allows for much greater flexibility than base and bounds, as segmentation allows a process to keep only part of its address space in physical memory, thus reducing fragmentation and supporting virtual memory.

(Note: other answers may apply.)

b. *(5 points) Say the currently running process has 16 active pages, P0-P15, all of which have their reference bits set to 1. If the operating system uses the clock algorithm for page replacement, the pages are ordered numerically around the "clock" (P0 is first, P1 is second, etc.), and the "clock hand" currently points to P6, which page will be replaced if a frame is needed to bring in a new page?* ***<u>Explain your answer for full credit.</u>***

<u>Solution:</u> In the clock algorithm, when a replacement is necessary, pages are checked in "clockwise order" until a page with reference bit equal to 0 is found. The reference bit for each page is cleared as it is visited. In this case, P6 will be checked first, then P7, and so on. After P15 is checked, P0 will be checked.

In all cases, the reference bits are found to be 1 and then cleared. So, the first page that will have its reference bit equal to 0 will be the first one checked for a second time: P6. Therefore, **<u>P6 will be replaced.</u>**

6 (continued)

c. (10 points) A portion of the currently running process's page table is shown below:

Virtual page #	Valid bit	Reference bit	Dirty bit	Frame #
7	1	1	1	3
8	0	0	0	--
9	1	0	1	4
10	0	0	0	--
11	1	1	0	0
12	1	1	0	1

Assume the system uses 20-bit addresses and 16 KB pages. The process accesses four addresses: 0x25FEE, 0x2B149, 0x30ABC, and 0x3170F.

Determine (i) which address would cause a page to be evicted if there were no free physical frames, (ii) which one would mark a previously clean page as modified, if the access were a write, and (iii) which one accesses a page that has not been referenced for many cycles. ***For full credit, show all work.***

Solution: First, note that 16 KB = 2^{14} byte pages imply a 14-bit offset. Therefore, in each virtual address, the upper 6 bits hold the page number, and the lower 14 bits hold the page offset.

Each virtual address is translated below, with the page number underlined:

- 0x25FEE = 0010 0101 1111 1110 1110 → page number 9
- 0x2B149 = 0010 1011 0001 0100 1001 → page number 10
- 0x30ABC = 0011 0000 1010 1011 1100 → page number 12
- 0x3170F = 0011 0001 0111 0000 1111 → page number 12
  (I didn't intend to duplicate a page number, but that's what happens when you write exam questions after midnight)

Now, to answer the questions above:

(i) For an access to cause an eviction, it must be to a page not currently in physical memory, as shown by the valid bit being 0. From the group above, that would be the access to page 10, or address **0x2B149.**

(ii) For an access to mark a previously clean page as modified, the dirty bit for that page must be 0. From the group above, either access to page 12 would qualify: address **0x30ABC** or **0x3170F**. Note that page 10 does not count because it is not a valid access.

(iii) Pages that have not been referenced for many cycles have their reference bits set to 0. From the group above, the access to page 9 qualifies—address **0x25FEE**. Again, the access to page 10 does not count because it is not a valid access.

6 (continued)

d. *(8 points, **<u>EECE.5730 only</u>**) Consider a system with 48-bit virtual addresses and 32-bit physical addresses. The system uses 4 KB pages. Assume each page table entry requires 4 bytes to store. Also, assume the currently running process is using only 16 KB of physical memory—the lowest-addressed 8 KB of its virtual address space, and the highest-addressed 8 KB of its virtual address space.*

If this process uses a two-level page table, and the first- and second-level page tables each use the same number of entries, how much space will that page table require given all of the information above? ***<u>Show all of your work for full credit.</u>***

<u>Solution:</u> The solution can be found as follows:

- If the system uses 4 KB = 2^{12} byte pages, 12 bits of the virtual address make up the page offset, and the remaining 36 bits choose a page table entry.
- Those 36 bits are split—the highest bits index into the first-level page table, while the next bits index into the second-level page table. Since the page tables at each level have the same number of entries, these bits are split evenly—18 bits to index into each.
- Using 18 bits to index into each page table implies each table has 2^{18} entries.
- If each page table entry requires 4 = 2^2 bytes, then each table requires $2^{18} * 2^2 = 2^{20}$ bytes, or 1 MB.
- Finally, the current process uses the highest- and lowest-addressed 8 KB in its virtual address space. The implication is that those two address ranges will not be covered in the same second-level page table—the process will have two second-level page tables active.
- Therefore, the process has three page tables active—the first-level page table and two second-level page tables—and the total amount of space required is **<u>3 * 2^{20} bytes, or 3 MB.</u>**