EECE.4810/EECE.5730: Operating Systems

Spring 2018

Exam 1 Solution

1. *(28 <u>+ 8</u> points)* ***Process management***

a. *(8 points) Explain (i) what an orphan process is, and (ii) why UNIX systems reassign processes to a new "parent" rather than simply terminating a process as soon as it's orphaned.*

Solution: (i) An orphan process is one for which the parent terminate first without collecting the child's exit status, making the child an orphan.

(ii) An orphaned process might still have useful work to do, but something must collect its exit status once it does terminate to ensure the orphan completed without errors.

Questions 1b, 1c, and 1d refer to programs ***pr1*** *and* ***pr2*** *below. Assume* ***pr1*** *always executes first.*

pr1:

```
int v1 = 10;

int main() {
   int v2 = 20;
   char str[10];
   pid_t pid;

   pid = fork(); (1)
   if (pid == 0) {
      v1 = 30;
      v2 = 40;
   }

   pid = fork(); (2)
   if (pid == 0) {
      sprintf(str, "%d", v1 + v2);
      execlp("./pr2", "pr2", str,
               NULL);
   }
   else if (pid > 0) {
      wait(NULL);
      printf("P1: %d %d\n", v1, v2);
   }
   return 0;
}
```

pr2:

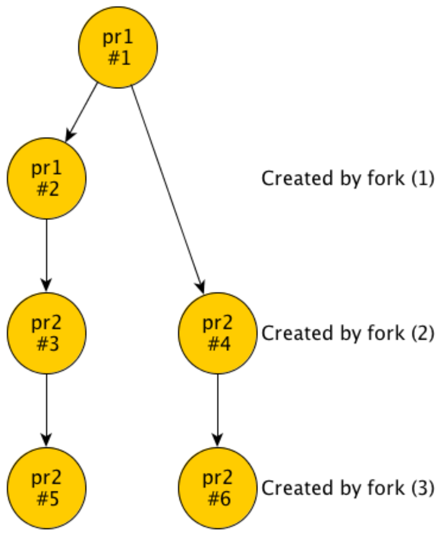
```
int main(int argc, char **argv) {
   pid_t pid;

   pid = fork(); (3)

   if (pid == 0)
      printf("C2: %s\n",
               argv[1]);
   else if (pid > 0) {
      wait(NULL);
      printf("P2: %s\n",
               argv[1]);
   }
   return 0;
}
```

b. *(10 points) How many unique processes do the programs* ***pr1*** *and* ***pr2*** *create when executed, including the initial process? Draw a process tree to support your answer.*

<u>Solution:</u> The programs create a total of six processes using the three fork() calls labeled (1), (2), and (3) above. The process tree below describes how these processes are created, and also shows the program each one executes:



1 (continued)

c. *(10 points) What will these programs print? (If there are multiple possible output sequences, choose one valid sequence and display it.)*

Solution: There are a few possible sequences, as described in the answer to part d. One such sequence is shown below. (Process numbers refer to the process tree in the answer to 1b.)

Note that, on a fork() call, the address space of the parent process is completely copied, meaning that changes to variable values in a child process will not be seen by the parent, and vice versa. In particular, after the first fork() call (labeled (1)), the child sets v1 = 30 and v2 = 40. Half of the processes—the first child (#2) and all processes that follow from it (#3 and #5)—see these values for v1 and v2, while the other half (#1, #4, and #6) see v1 = 10 and v2 = 20.

One possible output sequence is therefore:

```
C2: 70
C2: 30
P2: 70
P2: 30
P1: 30 40
P1: 10 20
```

d. *(8 points,* ***EECE.5730 only****) Are there multiple possible output sequences for this program? (In other words, could the same output statements print in a different order each time you run the program?) If so, explain why; if not, explain why your answer to part (c) is the only possible output.*

Solution: Multiple possible output sequences do exist because program *pr1* only calls `wait()` once. There are some constrained orderings (process numbers refer to the process tree in the answer to 1b):

- After the fork() labeled (3), the parent process in program *pr2* waits for its child, so process #3 will wait for #5, and #4 will wait for #6.
  - Since #4 waits for #6, we know their outputs will print in the order:
    - `C2: 30       (output of #6)`
    - `P2: 30       (output of #4)`
- We also know that #2 will wait for #3, since #2 (which runs *pr1*) has only one child.
  - Based on the statement above, we therefore know the statements using v1 = 30 and v2 = 40 will execute in the order:
    - `C2: 70       (output of #5)`
    - `P2: 70       (output of #3)`
    - `P1: 30 40    (output of #2)`
- What we don't know is which of the two children of #1 (#2 or #4) will finish first. Once either one finishes, #1 is free to print **`P1: 10 20`** and then exit.

2. *(20 points)* ***Inter-process communication (IPC)***

a. *(10 points) Name the two operations that message passing IPC systems must support, and explain how these operations function differently in direct communication than they do in indirect communication.*

Solution: IPC systems must support send and receive operations. In direct communication, processes communicate with one another in a shared link between the pair of processes, so each send or receive message must identify the process receiving the message. In indirect communication, processes communicate through shared mailboxes or ports, so each send or receive message must identify the mailbox being used.

b. *(10 points) Describe the steps necessary for two processes to establish and share a region of memory in a shared memory IPC system.*

Solution: The general steps are:

(1) One of the processes creates the shared region by requesting the desired amount of space and getting a handle through which it can be accessed. In our in class example using POSIX shared memory, the shared region is created as a memory-mapped file, with both processes initially accessing the region by name.

(2) Each process sharing the region must explicitly map the region into its address space.

3. *(26 points)* ***Multithreading***

a. *(8 points) What data or information can threads in the same process share that separate, independent processes do not share?*

Solution: Threads in the same process share the same code section (using separate PCs to access different parts of it as needed), global variables, and heap.

b. *(8 points) You are designing a program to run on a system with hardware multithreading support. What characteristics determine whether part (or all) of the program should be written to run using multiple threads?*

Solution: Parts of the program that must be serialized (i.e., performed in a specific order) should not be multithreaded, as those operations cannot be executed concurrently. Code that can be executed concurrently can be written to run using multiple threads.

3 (continued)

c. *(10 points) Given the two threads below, is it possible for x to be equal to 11 and y to be equal to 15 when both threads are done? If so, explain how; if not, explain why not.*

Assume x and y are shared variables, and each line of code (but not necessarily each whole thread) is executed atomically.

Thread 1	*Thread 2*
x = 10	*y = 15*
x = y - 3	*y = x + 4*

Solution: This question turned out to be very vague because I forgot to specify initial values for x and y. Any reasonable explanation therefore got most, if not all, of the available credit.

If you assume y initially = 14, then the following valid interleaving would set x = 11 and y = 15:

1. T1: x = 10
2. T1: x = y – 3 = 14 – 3 = 11
3. T2: y = 15
4. T2: y = x + 4 = 11 + 4 = 15

If y has any other initial value, then it's impossible for x = 11 and y = 15 when both threads are done without an invalid interleaving that reorders statements from Thread 2. Remember, while statements from different threads may be interleaved in several different ways if no ordering is imposed, threads from a single thread always execute in order. The impossible interleaving that gives the desired result is:

1. T1: x = 10
2. T2: y = x + 4 = 14 *(out of order—2nd line of Thread 2 executed first)*
3. T1: x = y – 3 = 14 – 3 = 11
4. T2: y = 15 *(out of order—1st line of Thread 2 executed second)*

4. *(26 points)* ***Synchronization***

a. *(8 points) Is a critical section required to always run to completion without any interruption from other threads or processes? Explain why or why not.*

Solution: No, a critical section is not always required to run to completion without interruption if those interruptions come from independent threads or processes. Remember that a critical section must be atomic only with respect to selected other pieces of code—typically code that shares data with the thread executing the critical section.

b. *(8 points) Programs that combine condition variables with locks for synchronization typically waste fewer processor cycles than programs that only use locks. Explain why.*

Solution: Without condition variables, your program may have to busy wait on a particular condition. For example, a dequeue() function called on an empty queue will have to wait for data to be added to the queue. Condition variables have associated waiting lists that are used to store all threads waiting on that variable, thus removing the need for busy waiting.

4 (continued)

c. *(10 points) Processes that share memory while executing concurrently are cooperating threads requiring protected access to shared data. The pseudocode below outlines the behavior of concurrent producer/consumer processes. We ran similar programs when discussing IPC, but the producer ran first; then, the consumer ran separately.*

In this example, both processes concurrently access a shared array used as a circular queue, `buffer[BUFFER_SIZE]`, and two integers, `in` and `out`. `in` is the position to store a newly produced item, while `out` is the position from which the next item will be consumed. The buffer is empty if the two positions are equal, and full if `in` is one spot behind `out`.

Determine which line(s) in each program represent a critical section, and show where you would place lock()/unlock() operations to ensure the atomicity of each critical section.

Solution: This problem is a bit too vague and therefore difficult to answer, so I graded it fairly leniently. I should have explicitly specifed the number of producer/consumer processes, as locks alone only work if there is exactly 1 of each process.

1 producer, 1 consumer: These processes share all of the variables described above—the shared array `buffer[]` as well as the two positions `in` and `out`. Accesses that modify any of these variables should be protected with locks, so lock() and unlock() operations should be placed as shown below to establish critical sections. (The code below assumes the lock is named `L`.)

Note: If you place the lock before the inner while loop in each case (the loops that check for an empty or full queue), the processes will deadlock if either loop condition is ever true. For example, if the queue becomes full and the producer enters the loop that waits for a spot to open, the consumer can't consume any data, because the producer has locked access to the buffer.

>1 producer and/or >1 consumer: Now the problem can't safely be solved with locks alone. Calling lock() before the inner while loop still potentially causes deadlock. However, placing lock()/unlock() calls as shown potentially creates a condition where multiple producers or consumers exit their busy-waiting loop at (effectively) the same time, despite only one spot opening in the queue (for a producer to use) or one item being placed in the queue (for a consumer to read).

Producer

```
item next_prod;
while (true) {
   // store new data in next_prod

   while (((in + 1) % BUFFER_SIZE)
                == out)
      ; /* do nothing */

   L.lock();
   buffer[in] = next_prod;
   in = (in + 1) % BUFFER_SIZE;
   L.unlock();
}
```

Consumer

```
item next_cons;
while (true) {

  while (in == out)
      ; /* do nothing */

  L.lock();
  next_cons = buffer[out];
  out = (out + 1) % BUFFER_SIZE;
  L.unlock();

  // do something with next_cons
}
```

4 (continued)

d. *(7 points, **<u>EECE.5730 only</u>**) We covered the concept of hand-over-hand locking in thread-safe queues that maintain one lock per node. Explain this concept and describe what potential problems it avoids.*

<u>Solution:</u> Hand-over-hand locking is the concept that, in a pointer-based data structure, a thread traversing the structure should lock the node it's moving to next before unlocking the node it's currently accessing. For example, in a linked list with nodes A → B → C in that order, a thread moving from node A to node B would lock node B before unlocking A.

Hand-over-hand locking ensures that a thread that is "behind" another thread in its traversal will not be able to pass that thread, modify the data structure, and affect the correctness of the first thread's operation. For example, given the queue above, if threads do not use hand-over-hand locking, you could see the following interleaving of operations:

1. Thread 1 (in node A): find A→next = B
2. Thread 1 (in node A): unlock A
3. Thread 2: lock A
4. Thread 2 (in node A): find A→next = B
5. Thread 2: unlock A
6. Thread 2: lock B
7. Thread 2: remove B from list

In this case, Thread 1 can no longer make progress, because the address it had for the node after A (the address of B) is no longer valid.