EECE.4810/EECE.5730: Operating Systems

Spring 2019

Exam 1 Solution

1. *(28 **+ 8** points) **Process management***

a. *(8 points) Explain (i) what a zombie process is, and (ii) why all processes are technically zombie processes for at least a brief time. (Hint: think about which of the five process states zombie processes must exist in.)*

Solution:

(i) A zombie process is a process that has finished executing but has not completely terminated, usually because its exit status has not been collected, so its resources (at a minimum, its process control block) have not been deallocated.

(ii) Any process in the terminated state is technically a zombie, since it has finished executing but its resources have not deallocated.

*Questions 1b, 1c, and 1d refer to the two programs **pr1** and **pr2** below. Assume **pr1** always executes first and is invoked as follows:* **`./pr1 4810`**

pr1:

```
int var1;

int main(int argc, char **argv) {
   int var2 = 5730;
   char str[15];
   pid_t pid, pid2;

   var1 = atoi(argv[1]);

   pid = fork();        (1)
   if (pid == 0) {
      printf("P1 child 1: %d %d\n",
               var1, var2);
      var1 = var2;

      pid2 = fork();    (2)
      if (pid2 == 0)
         printf("P1 child 2: %d %d\n",
                  var1, var2);
      else if (pid2 > 0) {
         wait(NULL);
         sprintf(str, "%d %d",
                  var1, var2);
         execlp("./pr2", "pr2",
                  str, NULL);
      }
   }
   else if (pid > 0) {
      var2 = var1;
      wait(NULL);
      printf("P1: %d %d\n",
               var1, var2);
   }
   return 0;
}
```

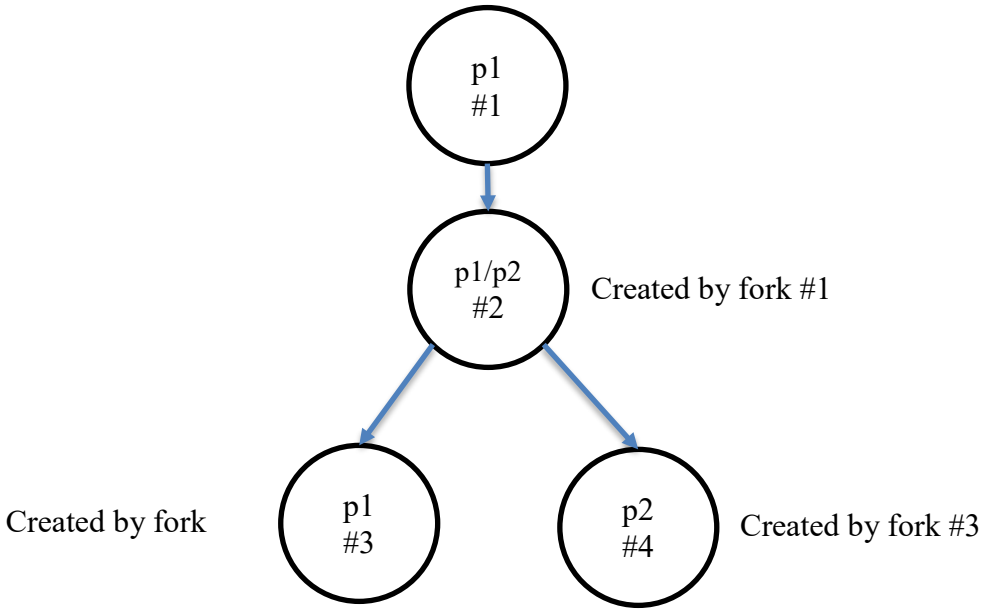
pr2:

```
int main(int argc, char **argv) {
   pid_t pid;
   int var3, var4;

   printf("P2: %s\n", argv[1]);
   sscanf(argv[1], "%d %d",
           &var3, &var4);
   var4 = var4 - 920;

   pid = fork();        (3)
   if (pid > 0) {
      wait(NULL);
      printf("P2 parent: %d %d\n",
               var3, var4);
   }
   else if (pid == 0)
      printf("P2 child: %d %d\n",
               var3, var4);
   return 0;
}
```

*b. (12 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*

Solution: The programs create a total of four processes using the three fork() calls labeled (1), (2), and (3) above. The process tree on the next page describes how these processes are created, and also shows the program each one executes. (Note that the first child process created actually executes both p1 (briefly) and p2.)

1b (continued) Process tree:



c. (12 points) What will these programs print? (If there are multiple possible output sequences, choose <u>one</u> valid sequence and display it.)

<u>Solution:</u> As described in part (d), there is only one valid output sequence, which is as follows:

```
P1 child 1: 4810 5730
P1 child 2: 5730 5730
P2: 5730 5730
P2 child: 5730 4810
P2 parent: 5730 4810
P1: 4810 4810
```

*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: The solution to part (c) is the only possible output because, after each fork() call, the parent process waits for its child before printing anything else.

2. *(22 points)* ***Inter-process communication (IPC)***

a. *(10 points) What benefits does message passing IPC offer over shared memory IPC? Specifically, what does the kernel handle that might make it easier for the programmer to write cooperating processes than it would be using shared memory?*

Solution: The kernel sets up and manages the shared region through which processes communicate (mailbox/message queue), as well as synchronizing accesses to that shared region. All the programmer needs to do is establish a link to the mailbox and utilize the appropriate send/receive primitives.

b. *(12 points) Explain why, in the POSIX shared memory example we discussed in class, (i) the shared region was first established as a memory-mapped file, and (ii) why it is beneficial to map the region into memory, as opposed to simply working with a shared file.*

Solution:

(i) Using a "file" allows each process accessing shared memory to access a named region. It's much easier to write processes that refer to a region by the same name than to establish an absolute address to which they'll all have access.

(ii) Files must be accessed using read/write system calls, which requires each access to invoke the kernel. Memory locations can be accessed using simple load/store instructions, making memory accesses typically simpler and faster than file accesses.

c. *(7 points,* ***EECE.5730 only****) Can you write a set of cooperating processes that model message passing IPC, communicating without invoking the kernel for the actual send/receive operations? If so, explain how, including the differences between modeling indirect and direct communication. If not, explain why not.*

Solution: You could write cooperating processes that use shared memory to model message passing, establishing a queue within that shared region and writing "send" and "receive" functions that write and read "messages" to and from the queue.

If modeling indirect communication, two or more processes would have access to the same shared queue, and the send/receive functions would have to account for the possibility of multiple readers and writers accessing that region concurrently.

If modeling direct communication, each pair of cooperating processes would share one region of memory, with the possibility that a single process might cooperate with multiple other processes and therefore share a separate queue with each process. The send/receive functions would be written to recognize which shared queue needed to be accessed, depending on the name of the other process specified in calls to those functions.

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

a. *(8 points) Is it possible for two concurrent threads to execute without running in parallel? Explain why or why not.*

Solution: Yes. For threads to be concurrent, all of the threads must make progress, but they can do so with only one thread executing at a time and access to the CPU being rotated between the threads in some fashion. The threads don't have to run at exactly the same time—in parallel—to run concurrently.

b. *(8 points) Name the four sections of a process's address space and explain why they can or cannot be shared by multiple threads in the same process.*

Solution:

The four sections are:

- Code section: Must be shared by multiple threads, since they're all part of the same process.
- Stack section: Each thread must have its own private stack, since each thread may call different functions and therefore need different sets of stack frames for those functions.
- Global section: Can be shared, especially if the threads are sharing data.
- Heap: Can be shared, as long as the OS tracks how much data's been allocated on the heap by all the threads and ensures the heap doesn't collide with a thread's stack.

3 (continued)

c. *(10 points) Given the two threads below, is it possible for v1 to be equal to -2 and v2 to be equal to 10 when both threads are done? If so, explain how; if not, explain why not.*

   Assume v1 and v2 are shared variables that are both initialized to 0 before either thread starts. Do not assume each line of code is executed atomically—each instruction executes atomically, but a statement may represent multiple instructions.

Thread 1	*Thread 2*
v1 = v1 + 2	*v1 = v1 - 2*
v2 = v2 + 10	*v2 = v2 – 10*

Solution: That result is possible, and the key is the idea that each statement may represent multiple instructions. Since variables are typically stored in memory (although a smart compiler may allocate some data in registers), each statement effectively breaks down into three instructions:

- Read the variable used on the right hand side of the statement
- Evaluate the right hand side
- Write the result to the variable on the left hand side

So, each thread could be written as a set of 6 instructions, and the following interleaving would produce the desired result. The key is the order in which the values of v1 and v2 are read—both threads must read the variable as its initial value, 0—and written. Evaluating the right hand side of each assignment can happen in any order.

1. T1: read v1 = 0
2. T2: read v1 = 0
3. T1: v1 + 2 = 2
4. T1: write v1 = 2
5. T2: v1 – 2 = -2
6. **T2: write v1 = -2**
7. T1: read v2 = 0
8. T2: read v2 = 0
9. T2: v2 – 10 = -10
10. T2: write v2 = -10
11. T1: v2 + 10 = 10
12. **T1: write v2 = 10**

4. *(20 points)* ***Synchronization***

a. *(10 points) Is it possible to establish a critical section without synchronization primitives such as locks? Explain your answer, including a description of the three properties a critical section must satisfy and how that can or cannot be done without synchronization primitives.*

Solution: Yes—in class, we discussed Dekker's algorithm, a solution to the critical section problem that used no synchronization primitives, and other such solutions exist. These solutions must satisfy:

- Mutual exclusion: if multiple processes attempt to enter their critical section at the same time, only one will be allowed to proceed. Solutions without synchronization primitives typically maintain a variable indicating "whose turn" it is.
- Progress: That same turn variable ensures one of the threads will be chosen when more than one attempt to enter their critical section, thus avoiding a situation where no threads are allowed to proceed.
- Bounded waiting: The turn variable changes each time a thread completes its critical section, ensuring that a different thread will be chosen the next time multiple threads attempt to enter the critical section simultaneously. Each thread will eventually get a turn, so no threads wait indefinitely.

4 (continued)

b. *(10 points) In an in-class example, we briefly discussed the idea of thread joining—in other words, a parent thread waiting for a child thread to finish. This functionality could be implemented using a lock, a condition variable, and a shared variable. The parent thread would set the variable to 0 before starting the child thread; the child would set the variable to 1 when it's finished. The synchronization primitives would protect access to the variable and prevent the parent from wasting cycles on busy waiting until the child finishes.*

   Complete the pseudocode below to show where lock/unlock and condition variable wait/signal operations would be used to force the parent to wait for its child to finish.

Solution: The code written below is pseudo-code—any syntax that makes it clear what operation is being performed is acceptable.

```
/* Global variables */
int done;           // Shared variable
lock L;             // Lock
cond_var CV;        // Condition variable

/* Child thread—complete with lock/CV operations */
void *child(void *arg) {
   ...              // Assume thread does actual work here
   /* COMPLETE SECTION BELOW */
   lock(L);
   done = 1;        // Indicate thread complete
   signal(CV);
   unlock(L);
}

/* Parent thread—complete with lock/CV operations */
int main() {
   ...
   done = 0;        // Clear shared variable
   create(child);   // Create and start child thread
   /* COMPLETE SECTION BELOW */
   lock(L);
   while (done == 0) {    // Wait for shared var to change
      wait(L, CV);
   }
   unlock(L);
}
```