EECE.4810/EECE.5730: Operating Systems

Spring 2017

Homework 1 Solution

Due **3:15 PM**, **Wednesday, 2/1/17**

1. *(10 points)*

a. *(3 points) Briefly describe the characteristics of zombie and orphan processes.*

Solution: A zombie process is a process that has terminated, but its parent has not yet read its exit status (using `wait()`), so the zombie process still maintains some minimal state in the system's job queue.

An orphan process is an active process for which the parent has terminated without invoking `wait()`, meaning the orphan process may remain active even after the parent has finished.

b. *(3 points) When a process becomes a zombie, what information about that process is still maintained in the operating system? Why?*

Solution: When a process becomes a zombie, its process ID, exit status, and entry in the job queue (or process table) remain, so that a parent process can check the exit status to ensure the process terminated without errors.

c. *(4 points) Are there conditions under which you would want a process to be orphaned? If so, give an example of a good situation in which a process should be orphaned.*

Solution: A process should be orphaned if it is a long-running task that does not require further attention from the user, or if it is an indefinitely running service, such as a daemon system process intended to run in the background without user control.

2. *(15 points) List the five states in which a process can exist, describe why a process would transition into that state, and describe why a process might transition out of that state.*

Solution: The five states are:

- *New*: Processes enter this state when they are first started, and transition out of this state as soon as they have been appropriately set up.
- *Running:* A process transitions into this state every time it is moved to the CPU to execute instructions, and transitions out of this state every time it leaves the CPU, either to wait for another device, to wait for an event, or because its CPU time has expired.
- *Waiting:* A process enters this state when it is waiting for some event—such as an interrupt, the termination of a child process, or access to an I/O device. It transitions out of the waiting state as soon as the desired event occurs.
- *Ready:* A process enters this state when it is waiting for access to the CPU to execute instructions. It leaves this state when it transitions to the running state and is actually able to execute code.
- *Terminated:* A process enters this state once it has exited; it leaves this state once its resources (memory, files, process control block) have been deallocated and its exit status has been checked by its parent process.

3. *(10 points) For the program below, assume the parent process has process ID (PID) 4810, and the child process has PID 5730. If the `getpid()` function returns the PID of the currently executing process, what will the program print?*

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

   pid = fork();
   if (pid == 0) {
      pid1 = getpid();
      printf("child: pid = %d\n", pid);
      printf("child: pid1 = %d\n", pid1);
   }
   else if (pid > 0) {  // Parent process
      pid1 = getpid();
      printf("parent: pid = %d\n", pid);
      printf("parent: pid1 = %d\n", pid1);
      wait(NULL);
   }
   return 0;
}
```

Solution: Recall that the `fork()` function copies the currently running process, making that process a parent process and the copy a child process. `fork()` returns 0 to the child process and the PID of the child (5730, in this case) to the parent.

We can't determine the exact order in which the parent and child print their information because the `wait()` system call in the parent process isn't called until after the two `printf()` statements. However, we can determine what each process alone prints:

Parent prints:
```
parent: pid = 5730
parent: pid1 = 4810
```

Child prints:
```
child: pid = 0
child: pid1 = 5730
```

4. *(15 points) Briefly describe each of the components of a process control block and explain why the operating system needs to track each of these pieces of information.*

Solution: A process control block (PCB) contains the following information:

- Process ID: used to identify this process
- Process state: used to determine in which queue the process currently resides and to where it can transition
- Memory limits: lowest and highest accessible memory addresses for this process, used to ensure code/data accesses are valid
- Scheduling information: used to determine how/when a process will be scheduled for execution when it is ready
- I/O status info: tracks devices accessible to process
- Open files: allows process to determine what files it can actually access

In addition, the PCB contains copies of the following information, which is updated on every context switch but not kept up to date while the process is executing. Saving these data allow a process to pick up where it left off the next time it is context switched back onto the CPU:

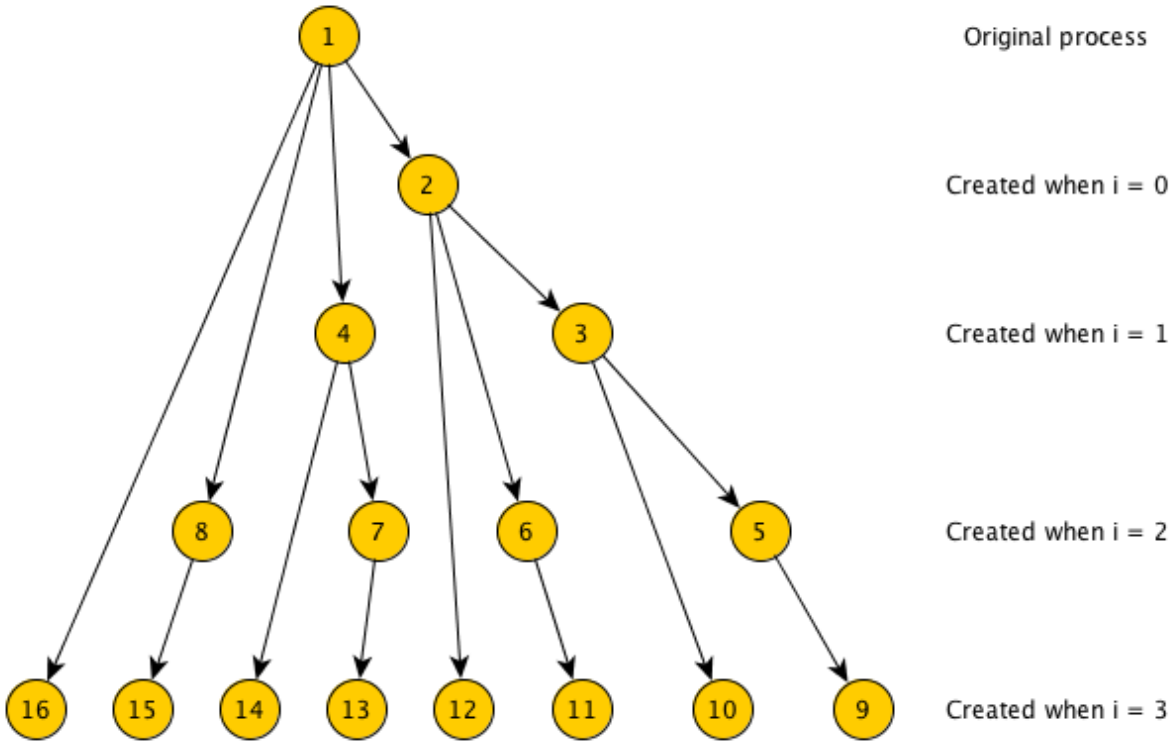
- Program counter: address of the next instruction to be executed
- Register copies: copies of the general-purpose and special-purpose register values used by the process, which (along with the program counter) represent the state of the process.

5. *(15 points, EECE.5730 only) Including the initial parent process, how many separate processes are created by the program shown below? Draw a process tree showing all of the processes created to help explain your answer.*

```
int main() {
   for (int i = 0; i < 4; i++)
      fork();

   return 0;
}
```

Solution: The program creates a total of 16 process, since each fork() call creates a second copy of the currently running process. The number of processes therefore double for each loop iteration (1→2, 2→4, 4→8, 8→16). The process tree below illustrates this process:



6. (10 points, EECE.5730 only) What will the program below print?

```
int nums[5] = {0,1,2,3,4};

int main() {
   int i;
   pid_t pid;

   pid = fork();

   if (pid == 0) {
      for (i = 0; i < 5; i++) {
         nums[i] *= -i;
         printf("CHILD: %d\n", nums[i]);
      }
   }
   else if (pid > 0) {
      wait(NULL);
      for (i = 0; i < 5; i++)
         printf("PARENT: %d\n", nums[i]);
   }
}
```

Solution: Remember that a `fork()` system call creates two copies of the existing process, including its address space. That means that each process gets its own copies of all data declared in the process, even if those variables are declared globally. The parent process therefore never changes its `nums[]` array—those values only change in the child process.

Since the parent waits for the child before printing its array contents, the program prints:

```
CHILD: 0
CHILD: -1
CHILD: -4
CHILD: -9
CHILD: -16
PARENT: 0
PARENT: 1
PARENT: 2
PARENT: 3
PARENT: 4
```