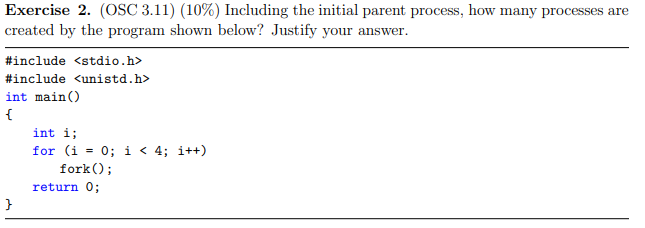




User Level: User level threads are easy to create/manage and don't need to be managed by the CPU scheduler/OS. They have no access to the CPU on their own, and must be paired with a kernel thread if they wish to perform any elevated functions. They are typically better for running user-applications which don't require read/write to the CPU. They exist entirely within the process that spawns them. They are optimal for tasks that involve heavy context switching.

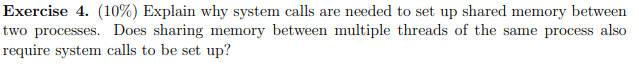
Kernel-Level: Kernel level threads have read/write access to the CPU, require a CPU scheduler to assign, and must be carefully maintained. They are powerful but much slower and scarcer a resource, which makes them ideal mostly for OS related tasks. I/O bound tasks prefer kernel-level threads. Kernel-level threads also require system calls for context switching to occur, which makes switching much slower.



fork(); gets iterated over a total of 4 times, each time doubling the number of processes. Therefore, 2n processes will get created, including the parent. In this case, n = 4, so 24 = 16 processes (including parent) get created by the program shown above.



Threads can run asynchronously, therefore different data may be contained across different threads. Therefore, they require their own register state so that, if the thread gets pre-empted, it can be saved and reloaded independently of other threads.

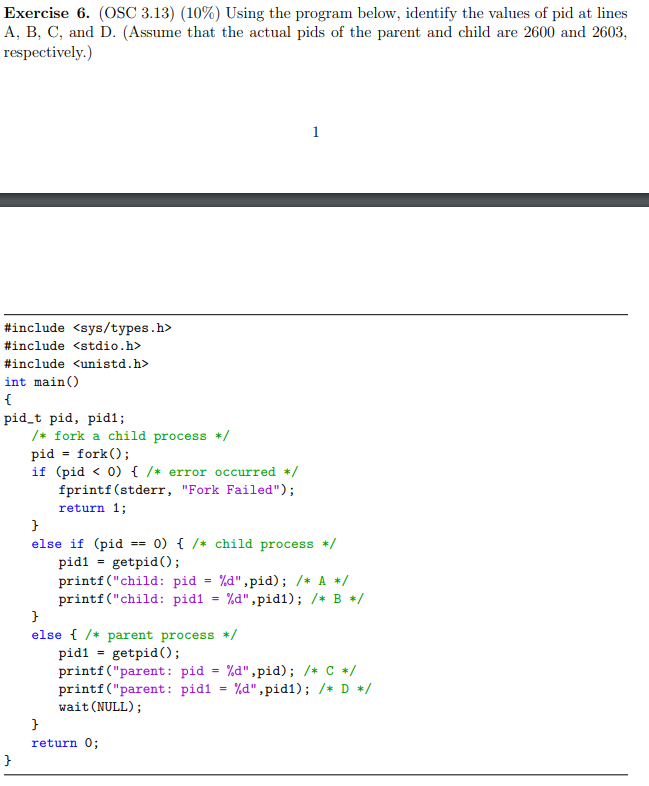


System calls are necessary for any activity that allocates memory for a process, and would be necessary to re-allocate the memory that two existing processes would begin to share, given that each process by definition has its own unique address space. Threads don't need system calls to set up shared memory, as they share address space with other threads within the same process. Two threads belonging to two distinct processes would require system calls to establish shared memory.

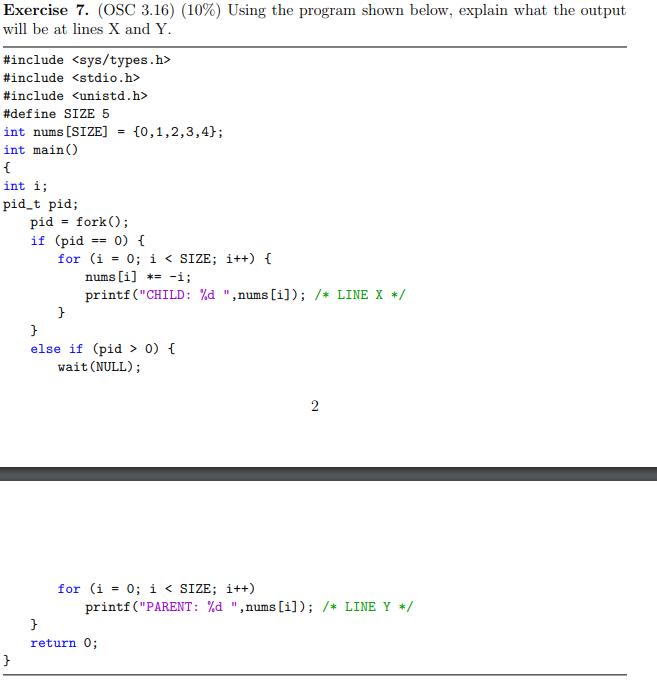


Similarities: Involves switching memory, state, and Program Counter

Differences: Processes also require switching code, data, and files

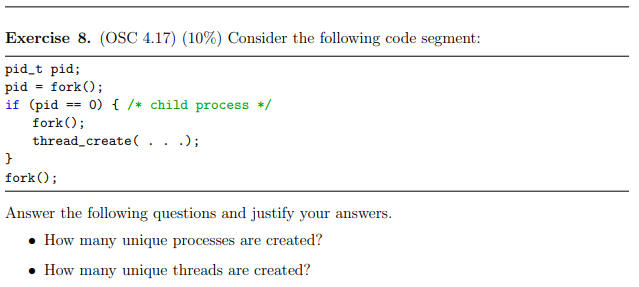


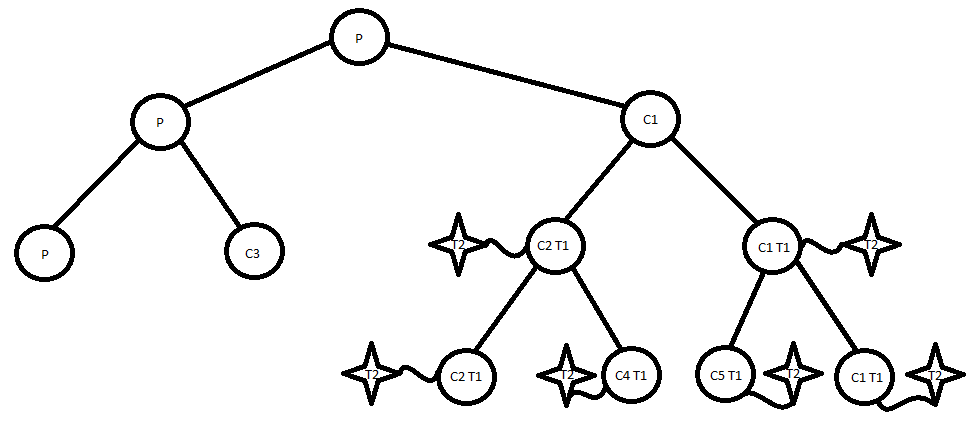
1. child: pid = 0
2. child: pid1 = 2603
3. parent: pid = 2603
4. parent: pid = 2600



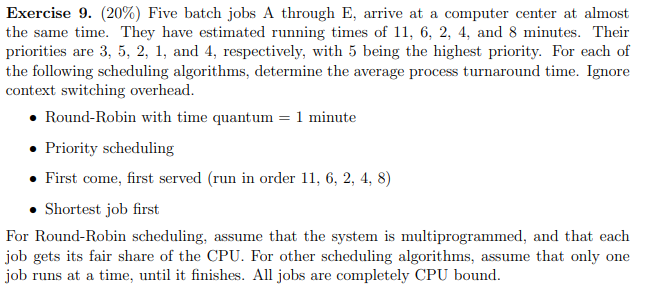
1. CHILD: 0  
   CHILD: -1  
   CHILD: -4  
   CHILD: -9  
   CHILD: -16
2. PARENT: 0  
   PARENT: -1  
   PARENT: -4  
   PARENT: -9  
   PARENT: -16

Parent waits for child, the nums array (as a global) will contain the values post child-modification





* Six processes are created. The threads are passed no function so they don't fork an additional time
* Ten unique threads are created, as each process counts as a thread, and four additional threads get created by children and children of children. (See Diagram Above)



|  |  |  |  |
| --- | --- | --- | --- |
| Job Name | Priority | Arrival Time | Run Time |
| A | 3 | 0 | 11 |
| B | 5 | 0 | 6 |
| C | 2 | 0 | 2 |
| D | 1 | 0 | 4 |
| E | 4 | 0 | 8 |

Round-Robin:

10+18+24+28+31 = 111/5 = 22.2 minutes/job

Priority Scheduling:

4 + 6 + 17 + 25 + 31 = 83/5 = 16.6 minutes/job

FCFS:

11 + 17 + 19 + 23 + 31 = 101/5 = 20.2 minutes/job

SJF:

2 + 6 + 12 + 20 + 31 = 71/5 = 14.2 minutes/job

Survey:

1. 4 Hours
2. 3
3. Trying to solve the questions was great. Trying to format everything on a computer to turn this in electronically was a pain in the butt.