Notes on Processes (2/17)

Definitions

- The difference between a *program* and a *process* is that a *program* is a passive entity that sits on the hard disk, while a *process* is an actively-running entity that exists in memory.
- Threads are also called lightweight processes. So far, everything up to this point has been single-threaded programs.
- For Multi-Tasking operating systems, we can have multiple processes in memory at one time, but it is not a guarantee that all process are running at the same time.

Parallelism

- With no parallelism, one process runs to completion before another process takes over. While not an issue for a series of trivial programs, any programs that runs for a significant amount of time will prevent short, simple programs from running.
- In pseudo-parallelism, processes are given a slice of time to run before getting swapped back into memory.
- With any sort of parallelism, the overhead costs will start to have an impact on how quickly a series of processes will finish.
- The CPU should, at all times, be busy with something to do.

Process States

- A process can enter a *blocked* state when it is dependent on I/O or when it runs a wait function.
- When a process's time slice is up, it goes back into the *ready* state.
- A process can repeatedly run and enter the ready queue without losing any progress towards completion.
- Only when a process completes entirely will it be properly terminated.

Process Control Blocks

- Each process has a block of memory containing information about the process.
- PCBs are kept in a linked-list structure
- For scheduling, many processes are associated with a priority, so they will be placed in the most appropriate place in the ready queue.

Scheduling

- As mentioned before, the goal is to keep the CPU busy at all times, and scheduling algorithms optimize for this purpose.
- In non-preemptive scheduling, jobs are placed on a queue and run to completion.
    - First-in, first-out queues are very inefficient.

    - Shortest job first is a better pick, but requires some heuristics and has problems of its own.
- Preemptive scheduling is a combination of first-in, first-out queue with time-slices on the CPU.
- *Context switching* is when the OS switches between processes (which causes overhead issues)
- Having multiple processes waiting is insignificantly detrimental to CPU performance.

Process Creation

- Processes are given priority by the program who called it.
- All processes are child processes excluding *init*.
- The *fork* function is used to create processes (parent/child). This is *not* an expensive operation
- When a process is created, the parents and child processes are byte-for-byte identical. However, it's common for the child to completely ditch the old code to run new code.
- ALWAYS be prepared to handle the case when a process fails
- The parent process waits for the child process to finish if it has no extra work to do (which puts it in a *blocked* state).
- When we call *fork()*, both the parent and child processes begin, so we must check if the return from *fork*, which is the process ID, belongs to the parent or child and respond accordingly. (Reference the code in the notes on how to do this).
- We do not control if the parent or child processes run first.
- Nothing is shared between the parent and child (as the clone is a deep copy)
- If we create multiple children, we can either wait until all three are completed. However, if we're only waiting for a specific child to finish first, make the parent process do something in the meantime.
- Absolutely do not use WNOHANG in CS180.

The *exec* Function

- Exec is actually a collection of processes around the *execve* function.
- *execl* is a variatic function.
    - Because of this, you absolutely must remember to pass NULL as the last argument
    - The first argument is the new program you wish to run. Will overwrite current process code in memory with the new program
    - The second argument is nowadays typically ignored. It was a way to tell a program which functionality of the program to execute. For instance, do we want the process to run its convert functionality, or its compress functionality, etc.

    - Because we are running a new program in this process, the child's return value is that of the program that it is running.
    - *execl* is great if we know at compile-time how many arguments to pass.

Process termination

- When a process terminates, all of its resources are given back to the operating system.
- Processes can either exit normally or through error-handling (which is voluntarily) or though a fatal error or being sent a killswitch (which is involuntarily)
- If you do not call *exit*, it will be called for you. You should still call it properly though.