Questions

Q1 - Unix Processes

Examine the following code. Assume that all system calls complete successfully.

int sum = 0;

int main() {

fork();

fork();

fork();

int status;

waitpid(-1, &status, WUNTRACED);

sum++;

printf("%d \n", sum);

}

1. How many processes are created in the above program?

Eight processes are created in the above program (one original and seven children).

2. What is the output from the program?

1

1

1

1

1

1

1

1

3. Draw the tree of processes with edges between parents and children.

P0 (parent)

P1

P2 P3

P4 P5 P6 P7

Q2 - Context Switching

1. Explain what happens on a context switch.

The currently running process is removed from the CPU, either voluntarily or forcibly, and allows another process to run. The content of the CPU registers that belong to this process are saved to its process control block. Now the operating system will perform whatever tasks are necessary for the process that was just removed. Once completed, the process is put back into a ready queue and operating system calls the scheduler. For the next process to run, the register values of that process are copied to the physical registers of the CPU, and the process begins running. This allows multiple processes to share a single CPU, which allows for multitasking.

2. Name three events that can cause a context switch.

Multitasking (the CPU switches to a new process), a kernel switch (the operating system switches between user and kernel mode), and an interrupt (such as a read or write call) are all events which can cause a context switch.

Q3 - Process Life

1. Draw the process state diagram.

Just created/admitted interrupt exit (error or completion)

Ready

Scheduler dispatch

I/O or event complete I/O or event wait

2. For each edge in the answer to (1), give an example of an action that

would force that transition of states.

Just created/admitted: a newly created process is loaded into a ready queue, awaiting its turn on the CPU.

Scheduler dispatch: the scheduler has chosen the specific program and loaded it into the CPU for execution

I/O or event wait: A system call, such as in the form of an I/O request (e.g., a read/write task), has occurred and the currently running process is taken off the CPU until the event completes.

I/O or event complete: the event that initially called the process off the CPU has completed and the process is moved to the ready queue.

Interrupt: The process is interrupted, such as by the timer, and the process is removed from the CPU and added back to the ready queue. In the case of scheduler subroutine, the process will simply wait until it is called back to the CPU.

Exit: The process has either finished as intended, or has encountered an error and crashed. Either way, it is removed from the CPU and not added back to any ready queue.

Q4 - Scheduling

1. Describe the role of the process scheduler in a modern operating system.

The scheduler is a subroutine of the operating system, and it moves processes in and out of the CPU from the ready queue(s). The scheduler loads the program counter of the selected process from that process’s PCB into the CPU to allow it to be run. To select a process, one of many different scheduling algorithms will be run (examples are FIFO, round-robin, multilevel feedback queue, and a Completely Fair Scheduler).

Q5 - Threading

1. Explain the difference(s) between a process and a thread.

A process is a program in execution, which is a (previously written) program that has been loaded into memory and is ready to be executed by the CPU. The execution of a process could be one of several states: ready/new, running, and blocked. Each process has its own space in memory and cannot share memory with another process.

By contrast, a thread is a portion of a process, of which multiple can be created. Threads under a single process segment do share memory. Because of this, switching between threads is much quicker than switching between processes.

2. Explain the difference(s) between user-level threads and kernel-level threads.

User-level threads are implemented at the application level and are implemented by the user. The operating system does not (usually) recognize these threads. These are usually easier to implement, and the language takes care of the context switches. By contrast, a kernel-level thread is created by the operating system and are executed independently of other ongoing processes. Kernel-level threads are more difficult to create, maintain and manage as they require support from the hardware to implement.

3. Given a CPU-bound multi-threaded program running on a 8-core (or 8-

processor) system, give the expected speedup according to Amdahl’s Law

if the parallelizable fraction of the program is 75%, for each of the following

cases:

(a) If the system uses kernel-level threads.



The theoretical speedup would be no greater than 2.909

(b) If the system uses user-level threads. Most user-level threads cannot take advantage of multiple cores. The operating system will only acknowledge the user process itself and will not recognize the process-created threads. As such, there is no expected speedup by adding additional cores/processors.