CMPSCI 377: Operating Systems

Solution to homework 1: Processes and Threads

1. (10 pts) What are differentiates between a program, an executable, and a process?

   Solution: A program is a collection of source files in some high level language that you write to do some function, for example, C++ files that implement sorting lists. An executable is the file that the compiler creates from these source files containing machine instructs that can execute on the CPU. A process is the active execution of the executable on the CPU and in the memory. It includes the memory management information, the current PC, SP, HP, registers, etc.

2. (10 pts) What happens on a context switch? Should context switches happen frequently or infrequently? Explain your answer.

   Solution:

   On a context switch, the OS must save the current execution context (the PC, SP, registers, memory mapping information, etc.) in the OS's PCB for this process, and changes the program's status to "ready", "waiting", or "terminated" as appropriate. The OS then selects a process to execute from the ready queue, changes this process's status to "running", and loads the process's execution context from the OS's PCB for this process onto the hardware.

   A context switch should happen frequently enough so that all jobs in the system make process, e.g., short jobs should not have to wait for lots of long jobs to finish before they run. Context switches should happen infrequently enough that their overhead does not take up a significant amount of the total system CPU time. Usually, we want to keep context switch time around 1 to 2% of total system time. The length of a time slice to force a context switch should thus be tuned to this percentage.

3. (20 pts) Using the *fork()*, *waitpid()*, and *kill()* system calls, write a program in which a parent creates two children. The parent then waits for the first child to complete, and kills the second when the first completes. After that, the parent exits.

   Solution:

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
#include <stdio.h>
#include <signal.h>

main()
{
  pid_t pid1, pid2;

  printf("PARENT: forking child1\n");
```

```
  if ((pid1 = fork()) == 0) {
    // Child process
    printf("CHILD1: running\n");
    sleep(5);
    printf("CHILD1: exiting\n");
    exit(0);
  }
  else {
    printf("PARENT: forking child2\n");
    if ((pid2 = fork()) == 0) {
      // Child process
      printf("CHILD2: running\n");
      while (1);
      printf("CHILD2: exiting\n");
      exit(0);
    }
    else {
      printf("PARENT: waiting for child1 to finish\n");
      waitpid(pid1, 0, 0);
      printf("PARENT: Child1 dead\n"); fflush(stdout);
      if (!kill(pid2, SIGHUP)) {
        printf("PARENT: Killed child2\n");
      }
    }
  }
}


// Output
//
// > run
// PARENT: forking child1
// PARENT: forking child2
// CHILD1: running
// PARENT: waiting for child1 to finish
// CHILD2: running
// CHILD1: exiting
// PARENT: Child1 dead
// PARENT: Killed child2
```

4. (10pts) What are the differences between user-level threads and kernel threads? Under what circumstances is one type better than the other?

   Solution: User-level threads are created, destroyed and scheduled by a user-level library. The operating systems is unaware of the presence of user-level threads. Kernel threads (also called lightweight processes) are threads that the operating system kernel is aware of.

   User-level threads are faster than kernel threads since no system calls are needed to create such threads, and the OS is not involved in their scheduling (no context switch overhead is incurred when switching from one thread to another). Since the user-level threads library can implement their own scheduling policy, these thread are suitable for problems that require a specific scheduling technique.

Since the OS is unaware of user-level threads, a drawback is that the OS can make poor scheduling decisions (e.g., by scheduling a process in which all its threads are waiting) and affect throughput. Also, the process gets the same fraction of the CPU regardless of the number of user-level threads in its address space. Kernel threads do not suffer from these drawbacks—the OS explicitly schedules such threads and a process get to run more often if it has more kernel threads.