CSE 120: Homework #2

Spring 2009

Out: 4/15 In: 4/27 (beginning of lecture)

For the homework questions below, if you believe that you cannot answer a question without making some assumptions, state those assumptions in your answer.

1. Explain the trade-offs between using multiple processes and using multiple threads

   A: Consider a multi-process web server application (like from lecture). Every request creates a new process. This means a new PCB, stack, page table, etc. Then the OS must schedule this new process before it can service the request. This involves a context switch. No contrast that with the same web server written as a multi-threaded application. Now just a TCB needs to be created, a portion of the existing process' stack is used as the thread's stack, no new page table. Further, the process has control over which thread to run, and can switch to the most appropriate without a context switch. However, the main drawback to the multi-threaded version is that the OS doesn't know about all these threads and will not give the web server more CPU time.

2. Does a multi-threading solution always improve performance? Please explain your answer and give reasons.

   A: Not always. Because the OS doesn't have visibility into the threads, it cannot make informed scheduling decisions. It may waste time context switching to a process where all the threads are blocked. Alternatively, it will not provide fair CPU time to all the threads of a process because it doesn't know they exist.

3. Explain the trade-offs between preemptive scheduling and non-preemptive scheduling

   A: Preemptive scheduling will interrupt a thread that has not blocked or yielded after a certain amount of time has passed (amount of time depends on the CPU scheduling algorithm). Non-preemptive scheduling will not interrupt the thread. The OS will wait until the thread blocks or yields.

4. What would be a possible problem if you executed the following program (and intend for it to run forever)? How can you solve it?

   A: The code below will run out of child process resources because it does not reap child processes after they terminate (doesn't clean up after them).

```
#include <signal.h>
#include <sys/wait.h>

main {
    int status; // solution
```

```
    for ( ; ; ) {
        if (! fork() ) { exit(0); }
        sleep(1);
        wait(&status); // solution
    }
}
```

Consider the following program:

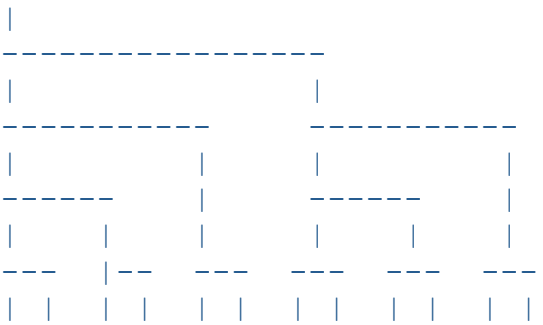
```
#include <stdlib.h>

int main (int argc, char *arg[])
{
    fork ();
    if (fork ()) {
        fork ();
    }
    fork ();
}
```

Draw a tree diagram showing the hierarchy of processes created when the program executes. How many total processes are created (including the first process running the program)?

Hint: You can always add debugging code, compile it, and run the program to experiment with what happens.

A: 12



6. Show how a lock, and acquire( ) and release( ) functions can be implemented using atomic SWAP instruction. The following is the definition of swap instruction:

```
void Swap (char* x,* y) \\ All done atomically
{    char temp = *x;
     *x = *y;
     *y = temp
}

struct lock {
  char held = 'n';
}

void acquire (struct lock *) {
  char newVal = 'y';
```

```
  do {
    Swap(&lock->held, &newVal);
  } while (newVal != 'n');
}

void release (struct lock *) {
  char newVal = 'n';
  Swap(&lock->held, &newVal);
}
```

Suppose you have an operating system that has only binary semaphores. You wish to use counting semaphores. Show how you can implement counting semaphores using binary semaphores.

Hints: You will need two binary semaphores to implement one counting semaphore. There is no need to use a queue — the queuing on the binary semaphores is all you'll need. You should not use busy waiting. The wait() operation for the counting semaphore will first wait on one of the two binary semaphores, and then on the other. The wait on the first semaphore implements the queueing on the counting semaphore and the wait on the second semaphore is for mutual exclusion.

```
struct semaphore {
  int value;
  int waiting = 0;
  b_semaphore mutex = {0}; // initialized to 0 – 0 threads pass
  b_semaphore queue = {1}; // initialized to 1 – 1 thread passes
}

void wait(struct semaphore *s) {
  b_wait(s->mutex);
  if (s->value == 0) {
    s->waiting++;
    b_notify(s->mutex);
    b_wait(s->queue);
  } else {
    s->value--;
    b_notify(s->mutex);
  }
}

void notify(struct semaphore *s) {
  b_wait(s->mutex);
  if (s->waiting > 0) {
    s->waiting--;
    b_notify(s->queue);
    b_notify(s->mutex);
  } else {
    s->value++;
    b_notify(s->mutex);
  }
}
```

yyzhou@cs.ucsd.edu