CS 318/418/618, Fall 2017

Homework #2

1. Explain the tradeoffs between using multiple processes and using multiple threads

   Anwser: Consider a multiprocess 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 multithreaded 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 multithreaded 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.

   Anwser: 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 tradeoffs between preemptive scheduling and non-preemptive scheduling

   Anwser: 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). Nonpreemptive scheduling will not interrupt the thread. The OS will wait until the thread blocks or yields.

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

   Anwser: (1) User-level threads are unknown by the kernel, whereas the kernel is aware of kernel threads. (2) On systems using either M:1 or M:N mapping, user threads are scheduled by the thread library and the kernel schedules kernel threads. (3) Kernel threads need not be associated with a process whereas every user thread belongs to a process. Kernel threads are generally more expensive to maintain than user threads as they must be represented with a kernel data structure.

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

```c
#include <signal.h>
#include <sys/wait.h>
int main()
{
    for (;;) {
        if (! fork() ) { exit(0); }
        sleep(1);
    }
    return 0;
}
```

   Anwser: 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). Corrected way is to call wait each time.

```c
#include <signal.h>
#include <sys/wait.h>
int main()
{
    int status;
    for (;;) {
        if (! fork() ) { exit(0); }
        sleep(1);
        wait(&status);
    }
```

```
    return 0;
}
```

The detailed explanation can be found on the man page for wait:

> In the case of a terminated child, performing a wait allows the system to release the resources associated with the child; if a wait is not performed, then the terminated child remains in a "zombie" state.
>
> The kernel maintains a minimal set of information about the zombie process (PID, termination status, resource usage information) in order to allow the parent to later perform a wait to obtain information about the child. As long as a zombie is not removed from the system via a wait, it will consume a slot in the kernel process table, and if this table fills, it will not be possible to create further processes.

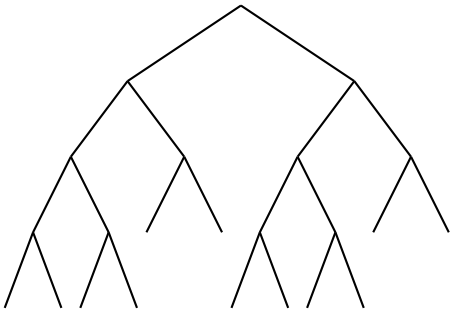
6. Consider the following program:

```
#include <stdlib.h>
int main ()
{
    fork ();
    if (fork ()) {
        fork ();
    }
    fork ();
    return 0;
}
```

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.

Anwser: 12 processes were created in total. The tree diagram:



7. 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, char * y) // All done atomically
{    char temp = *x;
     *x = *y;
     *y = temp
}

struct lock {
  char held = 'n';
}

void acquire (struct lock * l) {
  char newVal = 'y';
  do {
    swap(&l->held, &newVal);
  } while (newVal == 'y');
}

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

8. 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.

Anwser:

```
struct semaphore {
  int value;
  int waiting = 0;
  b_semaphore queue = {0}; // initialized to 0
  b_semaphore mutex = {1}; // initialized to 1
}

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

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

```
    b_signal(s->mutex);
  }
}
```

9. [Anderson] Given the following mix of tasks, task lengths, and arrival times, compute the completion and response time for each task, along with the average response time for the FIFO, RR, and SJF algorithms. Assume a time slice of 10 milliseconds and that all times are in milliseconds.

Task	Length	Arrival Time	Completion Time	Response Time
0	85	0		
1	30	10		
2	35	15		
3	20	80		
4	50	85		
		Average:		

Anwser: The constants in this question are a bit ambiguous for round robin: if a time slice completes at the same time that a job arrives, which should go first? Lets assume the arriving job is lower priority than tasks that are already waiting, but ahead of the task completing its quantum.

FIFO

Task	Length	Arrival Time	Completion Time	Response Time
0	85	0	85	85
1	30	10	115	85
2	35	15	150	115
3	20	80	170	90
4	50	85	220	135
		Average:		101

With round robin, the task schedule is: 0, 1, 0, 2, 1, 0, 2, 1, 0, 2, 1 (done at time 80), 0, 2, 4, 5, 0, 2 (done at time 135), 3 (done at time 145), 4, 0, 4, 0, 4, 0, 4 (done at 215), 0 (done at 220). This is slightly better than FIFO, but considerably less than shortest job first.

RR				
Task	**Length**	**Arrival Time**	**Completion Time**	**Response Time**
0	85	0	220	135
1	30	10	80	50
2	35	15	135	120
3	20	80	145	65
4	50	85	215	130
		Average:		100

With shortest job first, the task schedule is: 0 (until time 10), 1 (until it is done at time 40), 2 (until it is done at time 75), 0 (until time 80), 3 (until it is done at time 100), 4 (until it is done at time 150), and then 0 until is done at time 220.

SJF				
Task	**Length**	**Arrival Time**	**Completion Time**	**Response Time**
0	85	0	220	135
1	30	10	40	30
2	35	15	75	60
3	20	80	100	20
4	50	85	150	65
		Average:		62

10. [Anderson] Are there non-trivial workloads for which Multi-level Feedback Queue is an optimal policy? Why or why not? (A trivial workload is one with only one or a few tasks or tasks that last a single instruction.)

    Anwser: Yes. Suppose a long compute-intensive task arrives first, computes for a while, thereby dropping in priority to the lowest level possible. Suppose the next task is compute-intensive, but would complete just before reaching the lowest priority. It runs until it drops in priority, but still has substantial computing left to do. Then a sequence of very short job start and finish, one after another. The set of tasks is executed in shortest job first order.