CMPT 300: Operating Systems I

Assignment 1

Due June 21, 2018

POLICIES:

1. **Coverage**
    Chapters 1-6
2. **Grade**
    10 points, 100% counted into the final grade
3. **Individual or Group**
    Individual based, but group discussion is allowed and encouraged
4. **Academic Honesty**
    Violation of academic honesty may result in a penalty more severe than zero credit for an assignment, a test, and/or an exam.
5. **Submission**
    Electronic copy via CourSys
6. **Late Submission**
    2-point deduction for late submission within one week;
    5-point deduction for late submission over one week;
    Deduction ceases upon zero.

QUESTIONS:

1. **1 point**
    What is an operating system?
    What is the difference among an operating system, kernel, system programs, and application programs?
    [Grading Rubric: Both questions should be correctly answered to get 1 point. Otherwise, get 0.]

2. **2 points**
    Using the program shown below, explain what the output will be at the lines X and Y.
    ```
    #include <sys/types.h>
    #include <stdio.h>
    #include <unistd.h>
    ```

```
#define SIZE 5

int nums[SIZE] = {0,1,2,3,4};

int main()
{
    int i
    pid_t pid;

    pid = fork();

    if(pid == 0)
    {
        for(i=0; i<SIZE; i++)
        {
            nums[i] *= -i;
            printf("CHILD: %d", nums[i]); /* LINE X */
        }
    }
    else if(pid > 0)
    {
        wait(NULL);
        for(i=0; i<SIZE; i++)
            printf("PARENT: %d", nums[i]); /* LINE Y*/
    }

    return 0;
}
```

[Grading Rubric: Correct output of X gets 1 point; correct output of Y gets 1 point.]

3. **2 points: Amdahl's Law**

A common transformation required in graphics engines is square root. Implementations of floating-point (FP) square root vary significantly in performance, especially among processor designed for graphics. Suppose FP square root (FPSQR) is responsible for 20% of the execution time of a critical graphics benchmark. One proposal is to enhance the FPSQR hardware and speed up this operation by a factor of 10. The other alternative is just to try to make all FP instructions in the graphics processor run faster by a factor of 1.6; FP instructions are responsible for a total of 50% of the execution time for the application. The design team believes that they can make all FP instructions run 1.6 times faster with the same effort as required for the fast square root. Calculate these two design alternatives and decide which one is better.

[Grading Rubric: Correct calculation of one speed up gets 1 point; correct

calculation of both directly specifies the better one.]

4. **1 point**

   Including the initial parent process, how many processes are created by the program shown in the following code segment and why?

   ```
   #include <stdio.h>
   #include <unistd.h>

   int main()
   {
       /* fork a child process */
       fork();

       /* fork another child process */
       fork();

       /* fork another child process */
       fork();

       /* and fork another */
       fork();
   }
   ```

   [Grading Rubric: Only providing the number of created processes gets zero point. Reasoning about the number is required.]

5. **2 points**

   Consider the following set of processes, with the length of the CPU burst given in milliseconds:

Process	Burst Time	Priority
P_1	2	2
P_2	1	1
P_3	8	4
P_4	4	2
P_5	5	3

   The processes are assumed to have arrived in the order P_1, P_2, P_3, P_4, P_5, all at time 0.

   Draw four Gantt charts (i.e., the execution sequence chart) that illustrate the execution of these processes using the following scheduling algorithms:
   FCFS, SFJ, nonpreemptive priority (a larger priority number implies a higher priority), and RR (quantum =2).
   For each scheduling algorithm, **calculate** the waiting time of each process.
   [Grading Rubric: For each scheduling algorithm, correct only if both the

6. **2 points**

What are the three requirements for a solution to the critical-section problem?
Consider the following solution to the dining-philosophers' problem.

```
// Global variables.  Shared among threads.
int state[5];      // Initially state[i]==THINKING for all i.
semaphore mutex;   // Initially set to 1.
semaphore s[5];    // Initially s[i] is set to 0 for all i.


void philosopher(int i) {           int left(int i) {
  while(TRUE){                      // Philosopher to the left of i.
    think();                        // % is the mod operator.
    take_forks(i);                    return (i + 4) % 5;
    eat();                          }
    put_forks(i);
  }                                 int right(int i) {
}                                   // Philosopher to the right of i.
                                      return (i + 1) % 5;
void take_forks(int i) {            }
  P(mutex);
  state[i] = HUNGRY;                void test(int i) {
  test(i);                           if (state[i] == HUNGRY &&
  V(mutex);                                state[left(i)] != EATING &&
  P(s[i]);                                 state[right(i)] != EATING) {
}                                          state[i] = EATING;
                                           V(s[i]);
void put_forks(int i) {               }
  P(mutex);                         }
  state[i] = THINKING;
  test(left(i));
  test(right(i));
  V(mutex);
}
```

This solution has the following characteristics:
   a) The five philosophers are numbered 0 to 4;
   b) Each philosopher is represented by a thread that executes the function philosopher(i), where i is the number of that philosopher.
   c) A philosopher can be in one of three predefined states: HUNGRY (waiting for a fork), EATING (has 2 forks and is eating), or THINKING.
   d) The solution uses a shared array state and a semaphore mutex to ensure mutual exclusion in accessing this array.
   e) The solution also uses an array of semaphores s.
   f) There are predefined functions for thinking and eating that can take any amount of time to complete.

Describe a concrete scenario in which starvation occurs to this solution.