SFU SIMON FRASER UNIVERSITY
ENGAGING THE WORLD

CMPT 300: Operating Systems I

Assignment 1

Sample Solution

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.]

   An operating system is a program that manages a computer's hardware. It also provides a basis for application programs and acts as an intermediary between the computer user and the computer hardware.

   An operating system includes kernel and system programs.
   Kernel is the part of the operating system that is running at all times on the

computer.
System programs correspond to the other part of the operating system that is not necessarily part of the kernel.
Application programs include all programs not associated with the operation of the system.
That is, the operating system and application programs constitute the computer software.

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.]

Line X: 0, -1, -4, -9, -16

Line Y: 0, 1, 2, 3, 4

[Hints: no required for grading.
Any changes the child makes will occur in its copy of the data and will not be reflected in the parent.]

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.]

$Speedup_{FPSQR} = 1/((1-0.2)+0.2/10) = 1/0.82 = 1.22$

$Speedup_{FP} = 1/((1-0.5)+0.5/1.6) = 1/0.8125 = 1.23$

Improving the performance of the FP operations overall is slightly better because of the higher frequency.

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.]

16.

The purpose of fork() is to create a new process, which becomes the child process of the caller.
After a new child process is created, both processes will execute the next instruction following the fork() system call.

In the program shown above, originally there is one parent process and four fork() system calls are executed.

After the first fork() call, one new child process is created. Including the parent process, now there are two processes.

Both of the two processes then run the second fork() call, each creating a new child process and making the number of processes four.

All these four processes then run the third fork() call, each creating a new child process and making the number of processes eight.

Finally, all these eight processes run the fourth fork() call, each creating a new child process and making the number of processes 16.

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, SJF, 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 Gantt chart and the waiting times are correct. Correct answers for all four scheduling algorithms get 2 points. Correct answers for two or three scheduling algorithms get 1 point. Correct answers for less than two scheduling algorithms get 0 point.]

FCFS

P_1	P_2	P_3	P_4	P_5	
0	2	3	11	15	20

Waiting time of each process:
P_1: 0
P_2: 2
P_3: 3
P_4: 11
P_5: 15

SJF:

P_2	P_1	P_4	P_5	P_3	
0	1	3	7	12	20

Waiting time of each process:
P_1: 1
P_2: 0
P_3: 12
P_4: 3
P_5: 7

Nonpreemptive priority

P_3	P_5	P_1	P_4	P_2	
0	8	13	15	19	20

Waiting time of each process:
P_1: 13
P_2: 19
P_3: 0
P_4: 15
P_5: 8

RR

P_1	P_2	P_3	P_4	P_5	P_3	P_4	P_5	P_3	P_5	P_3

0 2 3 5 7 9 11 13 15 17 18 20

Waiting time of each process:

P_1: 0

P_2: 2

P_3: 12

P_4: 9

P_5: 13

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) {
  while(TRUE){
    think();
    take_forks(i);
    eat();
    put_forks(i);
  }
}

void take_forks(int i) {
  P(mutex);
  state[i] = HUNGRY;
  test(i);
  V(mutex);
  P(s[i]);
}

void put_forks(int i) {
  P(mutex);
  state[i] = THINKING;
  test(left(i));
  test(right(i));
  V(mutex);
}
```

```
int left(int i) {
// Philosopher to the left of i.
// % is the mod operator.
  return (i + 4) % 5;
}

int right(int i) {
// Philosopher to the right of i.
  return (i + 1) % 5;
}

void test(int i) {
  if (state[i] == HUNGRY &&
        state[left(i)] != EATING &&
        state[right(i)] != EATING) {
      state[i] = EATING;
      V(s[i]);
  }
}
```

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.
[Grading Rubric: Correct description of the three requirements gets 1 point. Correct description of a starvation scenario gets 1 point.]

Three requirements: mutual exclusion, progress, and bounded waiting.

Mutual exclusion. If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.

Progress. If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.

Bounded waiting. There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Possible starvation scenario, may not necessarily be the only one:
a. Philosopher 1 starts eating.
b. Philosopher 3 starts eating.
c. Philosopher 2 wants to eat, calls take_forks(2), and gets blocked.
d. Philosopher 1 finishes eating but cannot call V(s[2]) to unblock philosopher 2 because philosopher 3 is still eating.
e. Philosopher 1 starts eating again.
f. Philosopher 3 finishes eating but cannot call V(s[2]) to unblock philosopher 2 because philosopher 1 is still eating.
g. Philosopher 3 starts eating again.
h. Goto step d.

This way, Philosopher 2 never gets a chance to eat.