Ph.D. Qualifiers Exam Spring 2021
Operating Systems

Computer Science Department,
New Mexico State University

Qn1.1. There are two different ways to implement semaphore, one is to use busy waiting and the other is to use block and wakeup. What are the differences between the two implementation schemes? In what scenario does the busy-waiting semaphore have a better performance than the block-and-wakeup semaphore?

1. **Implementation with busy waiting: While the semaphore is not available, any process that tries to enter its critical section must loop continuously. The process "spins" while waiting for the semaphore to become available.**
2. **Implementation without busy waiting: When the semaphore is not available, the system places the process invoking the operation on the appropriate waiting queue; the system removes one of processes in the waiting queue and place it in the ready queue when the count of semaphore increases.**
3. **Because busy waiting requires no context switch, and a context switch may take considerable time. Therefore, busy waiting implementation is better when critical section is short and rarely occupied.**

Textbook source: Chapter 5.6 from page 214 to 216

Qn1.2. Consider a system implementing multilevel feedback queue scheduling. What strategy can a computer user employ to maximize the amount of CPU time allocated to the user's process?

The computer user may keep his/her process with highest priority by making it as a IO-bound process. (i.e., each CPU burst generated by the process is less than the time quantum of round-robin in the highest lever.)

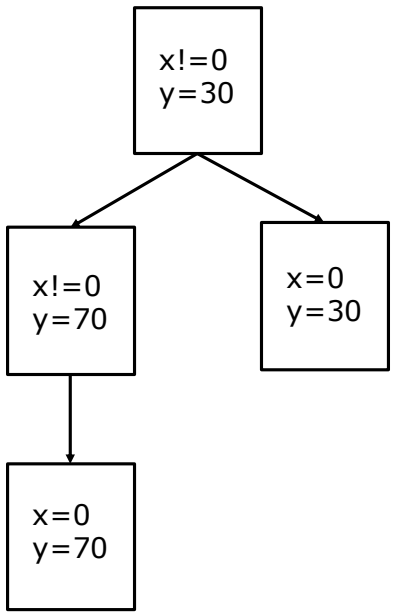
Textbook source: Chapter 6.3.6 from page 275 to 276

Qn2. Assume the following program contains no syntax errors. As it executes it will create one or more processes. Simulate the execution of this program and show how processes are created

```
#include<stdio.h>
main()
{
   int x, y, count;
   count =1;  x=10; y=10;
   while (count < 3)
    {
     if(x != 0) {
                    x = fork(); y = y + 10;}
     else{
                    x  = fork(); y = y+50;}
     printf("y = %d", y);
     count = count + 1;
    }
}
```

1. How many processes are created, including the parent process? Draw a simple tree diagram to show the parent-child hierarchy of the spawned processes. (10 pt)
2. What will this program print on screen when it is executed? (6 pt)

1. 4 processes



2. y=20

y=20

y=30

y=30

y=70

y=70

Textbook source: Chapter 3.3 from page 116 to 119

Qn3. Consider the following program:

3. Suppose that processes P1, P2, and P3 shown below are running concurrently. S1 and S2 are among the statements that P1 will eventually execute, S3 and S4 are among the statements that P2 will eventually execute, and S5 and S6 are among the statements that P3 will eventually execute.

You need to use semaphores to guarantee that 1) Statement S3 will be executed AFTER statement S6 has been executed; 2) Statement S1 will be executed BEFORE statement S4; 3) Statement S5 will be executed BEFORE statement S2.

1). Within the structure of the processes below, show how you would use semaphores to coordinate these three processes. Insert semaphore names and operations. Be sure to show initialize value of semaphores you are using. List semaphores you are using and their initial values here: (10 pt)

...	...	...
Process P1	Process P2	Process P3
...	...	...
S1	S3	S5
...	...	...
S2	S4	S6
...	...	...

2). After the semaphores have been appropriately placed, is it possible to determine which of the 6 statements above will be executed first, and which one will be executed last? If yes, give the statements, if no, give your reason. (6 pt)

1. We need 3 semaphores,

sem_init (x,0)	sem_init (y,0)	sem_init (z,0)
...	...	...
Process P1	Process P2	Process P3
...	wait(y)	...
S1	S3	S5
signal(x)	wait(x)	signal(z)
wait(z)	...	...
S2	S4	S6
...	...	signal(y)

2. No, either S1 or S5 could be executed first; either S2 or S4 could be executed last.

Qn4. Consider a reference string 1,2,3,4,2,5,6,2,3,2,1,6,7; and a system with only 4 frames, pure demand paging, and all frames initially empty.

1. How many page faults would occur with a FIFO replacement scheme? What pages are in the frames when the reference string has completed (5 pt)

   10 page faults, [2 3 1 7]

2. How many page faults would occur with a LRU replacement scheme? What pages are in the frames when the reference string has completed (5 pt)

   9 page faults, [2 1 6 7]

3. Describe one possible implementation scheme for LRU. (4 pt)

   There exist multiple solutions, for example, a link list, or associate a timer with every page.

4. Would increasing the number of frames always decrease the number of page faults for a particular reference string for FIFO? for LRU? Why or why not? Give your explanation. (4 pt)

   FIFO: no. Belady's anomaly indicates that it is possible to have a worse access behavior.

   LRU: yes. Belady's anomaly does not apply to LRU.

Textbook source: Chapter 9.4 from page 414 to 416

Qn5. A computer system has a 32-bit virtual address space with a page size of 4K, and 4 bytes per page table entry.

1. How many pages are in the virtual address space? (5 pt)
   $2^{32}/2^{12} = 2^{20} pages$
2. What is the maximum size of addressable physical memory in this system? (5 pt)
   $2^{32} \times 2^{12} = 2^{44} Bytes$
3. Assume the average process size is 1 GB, would you use a one-level, two-level, or three-level page table? Why? (4 pt)
   Two-level 8|10|12**, so that we can page the page table**
4. Compute the average size of a page table in question 3 above. (4 pt)
   $2^8 \times 4 + 2^8 \times 2^{10} \times 4 \approx 1M$

Textbook source: Chapter 8.6 from page 378 to 380

Qn6. Consider the following program executed by two different processes P1 and P2. x is the shared variable between two processes. Initially it is set to 10.

```
while(1)
   {
    x=x-1;
    x=x+1;
    if (x!=10)
            printf("x is %d,x");
   }
```

Consider that the processes P1 and P2 are executed on a uniprocessor system. Note that the scheduler in a uniprocessor system would implement pseudoparallel execution of these two concurrent processes by interleaving their instructions, without restriction on the order of the interleaving.

1. What is a race condition? Is this situation an example of a race condition? (**4 pts**)
2. If there is a race condition, show a sequence (i.e., trace the sequence of interleavings of statements) such that the statement "x is 10" is printed. (**6 pts**)

   Suggested format: `Pi: <instruction> <relevant new value>`
3. If there is a race condition, show a sequence that leads to the statement "x is 11" be printed. (**6 pts**)

   Hint: Remember that the increment/decrements at the source language level are not done atomically, e.g., the assembly language code below implements the single C increment instruction (x = x + 1).

```
register1 = x                   /* load register1 from memory location x */
register1 = register1 + 1       /* increment R0 */
x = register1                   /* store the incremented value back in x */
```

4. Create a semaphore to protect the shared variable x. Using instructions sem_init(), sem_wait(), and sem_signal(). (**4 pts**)

1. **A race condition is a situation, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place.**

2. **P1: <x=x-1> <x=9>**
   P2: <x=x-1> <x=8>
   P1: <x=x+1> <x=9>
   P1: <if (x!=10)> <x=9, true>
   P2: <x=x+1> <x=10>
   P1: <printf("x is 10")>

3. **P1: <register1=x> <x=10>**
   P1: <register1=register1 -1)> <x=10, register1=9>
   P2: <x=x-1> <x=9>
   P1: <x=register1> <x=9>
   P1: <x=x+1> <x=10>
   P2: <x=x+1> <x=11>
   P2: <if (x!=10)> <x=11, true>
   P2: <printf("x is 11")>

4.
```
sem_init(s,1);
...
sem_wait(s);
<critical section>
sem_signal(s);
...
```

Textbook source: Chapter 5.1 from page 204 to 206