COMP 346 – Fall 2020
Assignment 2

Due date and time: Friday October 30, 2020 by midnight

Name: Hualin Bai

ID: 40053833

Written Questions (50 marks):

Question # 1

What are the main differences between the user and kernel threads models? Which one of these models is likely to trash the system if used without any constraints?

Question #1

User threads are management done by user-level threads library, such as windows library.

Kernel threads are supported by the kernel, and threads are acting on kernel (an operating system core) managed by the OS (operating system).

Kernel threads models are more likely to trash the system without any constraints, since the kernel can modify the system directly.

Question # 2

Why threads are referred to as "light-weight" processes? What resources are used when a thread is created? How do they differ from those used when a process is created?

Question #2

Threads are referred to as "light-weight" processes because they execute in light environment, and use fewer resources, and share the address space.

When a thread is created, two types of resources are used. The first type is the shared elements with the controlling process such as code, data, and file. The second type is unshared elements including stacks, data, and descriptors.

The difference between thread and process is that process creation is heavy-weight, which means time-consuming and resource-intensive, since it requires creating new memory space and inter-process communication is costly. However, threads can simply code and increase efficiency.

Question # 3

Does shared memory provide faster or slower interactions between user processes? Under what conditions is shared memory not suitable at all for inter-process communications?

Question #3

Shared memory provides faster interactions between user processes. Because the communication is under the control of the users processes not the operating system, it uses less system call in shared memory.

Shared memory is not suitable at all for inter-process communications under these conditions:

(1) Deadlock: two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.

(2) Starvation: a process may never be removed from the semaphore queue in which it is suspended.

This means that unsynchronized access to shared memory is not suitable at all for inter-process communications, since we need to control mutual exclusion.

Question # 4

a) Consider three concurrent processes A, B, and C, synchronized by three semaphores: *mutex*, *goB*, and *goC*, which are initialized to 1, 0 and 0 respectively:

```
 Process  A                     Process  B                   Process  C
-------------                  -----------                  -------------
wait  (mutex)                 wait  (mutex)                wait  (mutex)
...                                 ...                              ...
signal  (goB)                 wait  (goB)                  wait  (goC)
...                                 signal  (goC)           ...
signal  (mutex)                ...                          signal  (mutex)
                                signal  (mutex)
```

Does there exist an execution scenario in which: (i) All three processes block permanently? (ii) Precisely two processes block permanently? (iii) No process blocks permanently? Justify your answers.

Question #4 (a)

(1) Execute Process B, then wait(goB), then execute either Process C or A will cause them block.

If execute Process C, wait(mutex) is blocked since process B has already performed wait(mutex).

If execute Process A, wait(mutex) is also blocked for same reason.

Thus, start with process B will cause all three processes block permanently.

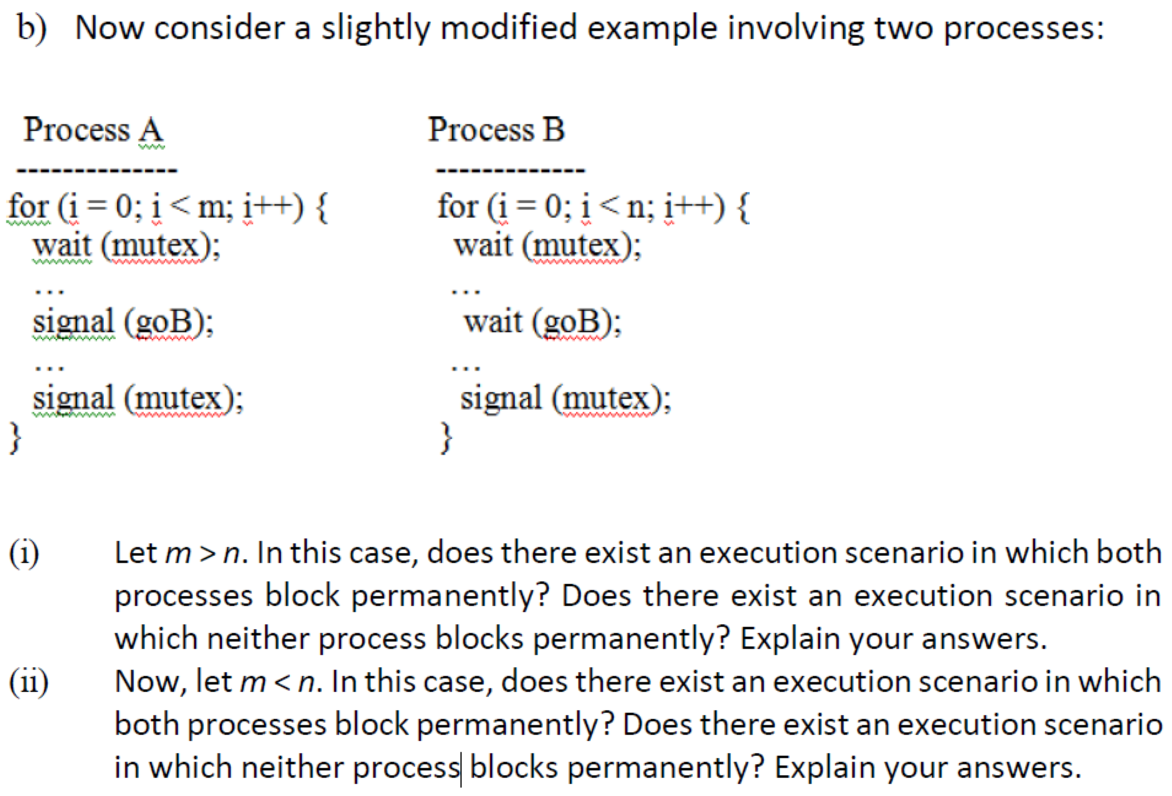
(2) Execute Process A, signal(goB) will make goB to change 1.

Then execute Process C, the PC will be blocked at wait(goC). Then execute Process B, it will be blocked at wait(mutex).

Thus, Process A -> C -> B will cause Process B and C to be blocked permanently.

(3) Execute Process A, then Process B, then Process C.

Process A will makes goB =1, then process B will make goC=1, then process C will execute without blocked. No process blocks permanently.



b) Now consider a slightly modified example involving two processes:

```
 Process A                      Process B
--------------                 -------------
for (i = 0; i < m; i++) {      for (i = 0; i < n; i++) {
  wait (mutex);                  wait (mutex);
  ...                            ...
  signal (goB);                   wait (goB);
  ...                            ...
  signal (mutex);                 signal (mutex);
}                              }
```

(i) Let $m > n$. In this case, does there exist an execution scenario in which both processes block permanently? Does there exist an execution scenario in which neither process blocks permanently? Explain your answers.

(ii) Now, let $m < n$. In this case, does there exist an execution scenario in which both processes block permanently? Does there exist an execution scenario in which neither process blocks permanently? Explain your answers.

Question #4 (b)

(1) Yes, if execute Process B before process A, PB will be blocked at wait(goB), and PA will be blocked at wait(mutex). Thus, Process B -> A will make both processes blocked permanently.

Yes, if execute all steps of Process A, then executes all steps of Process B and go on go forth.

Since m>n, the wait(goB) will be signaled all the way. Thus, in this case neither process will be blocked permanently.

(2) Yes, similar as (1). If execute Process B before Process A, both processes will be blocked permanently.

No, since m<n, there does not exist a scenario in which neither process blocks permanently.

Because after Process A finished its loop, the value of goB will be always 1 in Process B, and wait(goB) will not be signaled.

Question # 5

In a swapping/relocation system, the values assigned to the <base, limit> register pair prevent one user process from writing into the address space of another user process. However, these assignment operations are themselves privileged instructions that can only be executed in kernel mode.

Is it conceivable that some operating-system processes might have the entire main memory as their address space? If this is possible, is it necessarily a bad thing? Explain.

Question #5

Yes, it is possible. However, it is necessarily a bad thing, since the operating system will take up the whole memory space which will cause a low degree of multiprogramming and possibly moving the content from main memory to secondary memory.

Question # 6

Sometimes it is necessary to synchronize two or more processes so that all process must finish their first phase before any of them is allowed to start its second phase.

For two processes, we might write:

```
semaphore s1 = 0, s2 = 0;

process P1 {            process P2 {
<phase I>                <phase I>
V (s1)                   V (s2)
P (s2)                   P (s1)
<phase II>              <phase II>
         }                       }
```

a) Give a solution to the problem for three processes P1, P2, and P3.
b) Give the solution if the following rule is added: after all processes finish their first phase, phase I, they must execute phase II in order of their number; that is P1, then P2 and finally P3.

Question #6

(a)

Semaphore s1=0, s2=0, s3=0.

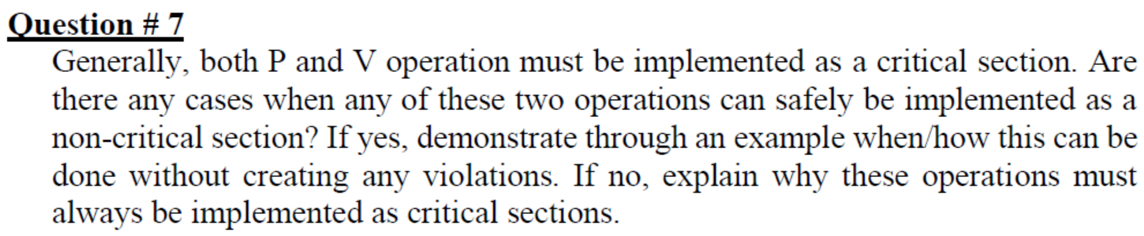
```
 Process P1 {          process P2 {               process P3 {
<phase I>                 <phase I>                   <phase I>
V(s1)                      V(s2)                       V(s2)
P(s2)                      P(s1)                       P(s3)
P(s2)                      P(s1)
V(s1)                      V(s3)
<phase ii>                <phase ii>                  <phase ii>
        }                          }                           }
```

(b)

Semaphore s1=0, s2=0, s3=0.

```
Process P1 {           process P2 {               process P3 {
<phase I>                 <phase I>                   <phase I>
V(s1)                      V(s2)                      V(s2)
P(s2)                      P(s1)                      P(s3)
P(s2)                      p(s1)

<phase ii>                <phase ii>                  <phase ii>
V(s1)                      V(s3)
        }                          }                           }
```

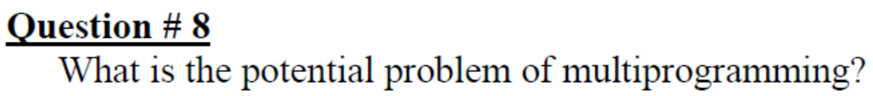


Question # 7

Generally, both P and V operation must be implemented as a critical section. Are there any cases when any of these two operations can safely be implemented as a non-critical section? If yes, demonstrate through an example when/how this can be done without creating any violations. If no, explain why these operations must always be implemented as critical sections.

Question #7

No. P and V operation must be implemented as a non-critical section, since they are operating the shared data. However, if these operations are capsulized into hardware, it is possible to see P and V as a non-critical section.



Question # 8

What is the potential problem of multiprogramming?

Question #8

Multiprogramming is a rudimentary form of parallel processing in which several programs are run at the same time on a uniprocessor. In multiprogramming, several programs may have the access to each other's resources, which results in data inconsistency and security issues.
There exist two kinds of potential security problems:
(1) Copying or Stealing one's programs or data.
(2) Using system resources (CPU, memory, disk space, etc.) with improper accounting.