Alexandria University
Faculty of Engineering
Comp. & Comm. Engineering
CC373: Operating Systems

جامعة الاسكندرية
كلية الهندسة
برنامج هندسة الحاسب والاتصالات
مادة نظم التشغيل

Sheet5
CONCURRENCY: SYNCHRONIZATION

1) What is the producer/consumer problem?
2) What is a monitor?
3) What is the distinction between blocking and nonblocking with respect to messages?
4) What conditions are generally associated with the readers/writers problem?
5) Consider the following definition of semaphores:

```
void semWait(s)
{
    if (s.count > 0) {
      s.count--;
    }
    else {
      place this process in s.queue;
      block;
    }
}
void semSignal (s)
{
    if (there is at least one process blocked on
        semaphore s) {
        remove a process P from s.queue;
        place process P on ready list;
    }
    else
        s.count++;
}
```

Compare this set of definitions with that of Figure 5.6. Note one difference: With the preceding definition, a semaphore can never take on a negative value. Is there any difference in the effect of the two sets of definitions when used in programs? That is, could you substitute one set for the other without altering the meaning of the program?

6) It should be possible to implement general semaphores using binary semaphores. We can use the operations semWaitB and semSignalB and two binary semaphores, delay and mutex. Consider the following:
Initially, s is set to the desired semaphore value. Each semWait operation decrements s , and each semSignal operation increments s . The binary semaphore mutex, which is initialized to 1, assures that there is mutual exclusion for the updating of s . The binary semaphore delay, which is initialized to 0, is used to block processes.
There is a flaw in the preceding program. Demonstrate the flaw and propose a change that will fix it. Hint: Suppose two processes each call semWait(s) when s is initially 0, and after the first has just performed semSignalB(mutex) but not performed semWaitB(delay) , the second call to semWait(s) proceeds to the same point. All that you need to do is move a single line of the program.
7) The following pseudo code is a correct implementation of the producer/consumer problem with a bounded buffer:

```
item[3] buffer; // initially empty
semaphore empty; // initialized to +3
semaphore full; // initialized to 0
binary_semaphore mutex; // initialized to 1
```

```
void producer()
{
    ...
    while (true) {
        item = produce();
p1:     wait(empty);
  /     wait(mutex);
p2:     append(item);
  \     signal(mutex);
p3:     signal(full);
    }
}
```

```
void consumer()
{
    ...
    while (true) {
c1:     wait(full);
  /     wait(mutex);
c2:     item = take();
  \     signal(mutex);
c3:     signal(empty);
        consume(item);
    }
}
```

Labels p1, p2, p3 and c1, c2, c3 refer to the lines of code shown above (p2 and c2 each cover three lines of code). Semaphores empty and full are linear semaphores that can take unbounded negative and positive values. There are multiple producer processes, referred to as Pa, Pb, Pc, etc., and multiple consumer processes, referred to as Ca, Cb, Cc, etc. Each semaphore maintains a FIFO (first-in-first-out) queue of blocked processes.

In the scheduling chart below, each line represents the state of the buffer and semaphores after the scheduled execution has occurred. To simplify, we assume that scheduling is such that processes are never interrupted while executing a given portion of code p1, or p2, …, or c3. Your task is to complete the following chart.

Scheduled Step of Execution	full's State and Queue	Buffer	empty's State and Queue
Initialization	full = 0	OOO	empty = +3
Ca executes c1	full = -1 (Ca)	OOO	empty = +3
Cb executes c1	full = -2 (Ca, Cb)	OOO	empty = +3
Pa executes p1	full = -2 (Ca, Cb)	OOO	empty = +2
Pa executes p2	full = -2 (Ca, Cb)	X OO	empty = +2
Pa executes p3	full = -1 (Cb) Ca	X OO	empty = +2
Ca executes c2	full = -1 (Cb)	OOO	empty = +2
Ca executes c3	full = -1 (Cb)	OOO	empty = +3
Pb executes p1	full =		empty =
Pa executes p1	full =		empty =
Pa executes __	full =		empty =
Pb executes __	full =		empty =
Pb executes __	full =		empty =
Pc executes p1	full =		empty =
Cb executes __	full =		empty =
Pc executes __	full =		empty =
Cb executes __	full =		empty =
Pa executes __	full =		empty =
Pb executes p1-p3	full =		empty =
Pc executes __	full =		empty =
Pa executes p1	full =		empty =
Pd executes p1	full =		empty =
Ca executes c1-c3	full =		empty =
Pa executes __	full =		empty =
Cc executes c1-c2	full =		empty =

Pa executes __	full =		empty =
Cc executes c3	full =		empty =
Pd executes p2-p3	full =		empty =

8) Consider a sharable resource with the following characteristics: (1) As long as there are fewer than three processes using the resource, new processes can start using it right away. (2) Once there are three process using the resource, all three must leave before any new processes can begin using it. We realize that counters are needed to keep track of how many processes are waiting and active, and that these counters are themselves shared resources that must be protected with mutual exclusion. So we might create the following solution:

```
1   semaphore mutex = 1, block = 0;     /* share variables: semaphores, */
2   int active = 0, waiting = 0;                       /* counters, and */
3   boolean must_wait = false;                      /* state information */
4
5   semWait(mutex);                          /* Enter the mutual exclusion */
6   if(must_wait) {                        /* If there are (or were) 3, then */
7      ++waiting;                        /* we must wait, but we must leave */
8      semSignal(mutex);                     /* the mutual exclusion first */
9      semWait(block);              /* Wait for all current users to depart */
10     semWait(mutex);                     /* Reenter the mutual exclusion */
11     --waiting;                          /* and update the waiting count */
12  }
13  ++active;                          /* Update active count, and remember */
14  must_wait = active == 3;                     /* if the count reached 3 */
15  semSignal(mutex);                        /* Leave the mutual exclusion */
16
17  /* critical section */
18
19  semWait(mutex);                                /* Enter mutual exclusion */
20  --active;                                /* and update the active count */
21  if(active == 0) {                                  /* Last one to leave? */
22     int n;
23     if (waiting < 3) n = waiting;
24     else n = 3;                                /* If so, unblock up to 3 */
25     while( n > 0 ) {                                /* waiting processes */
26        semSignal(block);
27        --n;
28     }
29     must_wait = false;               /* All active processes have left */
30     }
31  semSignal(mutex);                        /* Leave the mutual exclusion */
```

The solution appears to do everything right: All accesses to the shared variables are protected by mutual exclusion, processes do not block themselves while in the mutual exclusion, new processes are prevented from using the resource if there are (or were) three active users, and the last process to depart unblocks up to three waiting processes.

a) The program is nevertheless incorrect. Explain why.

b) Suppose we change the if in line 6 to a while. Does this solve any problem in the program? Do any difficulties remain?

9) Explain what is the problem with this implementation of the one-writer many readers problem?

```
int readcount;                      // shared and initialized to 0
Semaphore mutex, wrt;               // shared and initialized to 1;
```

```
// Writer:
semWait(wrt);
/* Writing performed*/
semSignal(wrt);
```

```
// Readers:
semWait(mutex);
readcount++;
if readcount == 1 then semWait(wrt);
semSignal(mutex);
/*reading performed*/
semWait(mutex);
readcount--;
if readcount == 0 then semSign (wrt);
semSignal(mutex);
```

How to submit the homework assignments?

- Solve the sheet individually without looking up the solution on the Internet. The sheet is to practice; it is a learning tool not an exam.
- Assignments are to be **handwritten**.
- Papers are to be scanned (I like camscanner app). Put all images in a pdf file (camscanner does that for you)
- Use MS Teams to submit
  - Your filename should be your user id
