PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ
FACULTAD DE CIENCIAS E INGENIERÍA

SISTEMAS OPERATIVOS

2da práctica (tipo a)
(Segundo semestre de 2012)

Horario 0781: prof. V. Khlebnikov

Duración: 1 h. 50 min.
Nota: No se puede usar ningún material de consulta.
La presentación, la ortografía y la gramática influirán en la calificación.
Puntaje total: 20 puntos

Pregunta 1 **(4 puntos – 20 min.)** Suppose processes A and B that share the queue both try to add new nodes 5 and 6 respectively to the queue at about the same time.

Shared queue:

```
head -> 1 -> 2 -> 3 -> 4 -> null
                       ^
tail ------------------|
```

to add a new node x to the queue:

`old_tail` is a local variable

```
link(node x) := null             /* null out x's next pointer */
old_tail := tail                 /* follow tail to node 4 */
tail := address x                /* change tail from 4 to x */
link(old_tail) := address x      /* change 4 from -> null to -> x */
```

What is the result of a race condition? Describe the way to obtain this result.

Pregunta 2 **(4 puntos – 20 min.)** Barrier. A limitation of the rendezvous solution is that it does not work with more than two threads. Puzzle: Generalize the rendezvous solution. Every thread should run the following code:

```
Barrier code:
        rendezvous
        critical point
```

The synchronization requirement is that no thread executes `critical point` until after all threads have executed `rendezvous`. You can assume that there are *n* threads and that this value is stored in a variable, `n` , that is accessible from all threads. When the first $n - 1$ threads arrive they should block until the *n*th thread arrives, at which point all the threads may proceed.

```
Barrier hint:
        n = the number of threads
        count = 0
        mutex = Semaphore(1)
        barrier = Semaphore(0)
```

`count` keeps track of how many threads have arrived. `mutex` provides exclusive access to `count` so that threads can increment it safely. `barrier` is locked (zero or negative) until all threads arrive; then it should be unlocked (1 or more).

```
Barrier non-solution:
        rendezvous

        mutex.wait()
            count = count + 1
        mutex.signal()

        if count == n: barrier.signal()

        barrier.wait()

        critical point
```

Since `count` is protected by a `mutex`, it counts the number of threads that pass. The first $n - 1$ threads wait when they get to the barrier, which is initially locked. When the *n*th thread arrives, it unlocks the barrier.
What is wrong with this solution? The problem is a deadlock. As an example, imagine that $n = 5$ and that 4 threads are waiting at the barrier. The value of the semaphore is the number of threads in queue, negated, which is -4. When the 5th thread signals the barrier, one of the waiting threads is allowed to proceed, and the semaphore is incremented to -3. But then no one signals the semaphore again and none of the other threads can pass the barrier. This is a deadlock.

Puzzle: Does this code always create a deadlock? Can you find an execution path through this code that does *not* cause a deadlock?

Pregunta 3 (7 puntos – 35 min.) An incorrect solution to the infinite-buffer producer/consumer problem using binary semaphores:

```
/* program producerconsumer */
int n;
binary_semaphore s = 1, delay = 0;

void producer()
{
    while (true)
    {
        produce();
        semWaitB(s);
        append();
        n++;
        if (n==1) semSignalB(delay);
        semSignalB(s);
    }
}

void consumer()
{
    semWaitB(delay);
    while (true)
    {
        semWaitB(s);
        take();
        n--;
        semSignalB(s);
        consume();
        if (n==0) semWaitB(delay);
    }
}

void main() { n = 0; parbegin(producer,consumer); }
```

There is a flaw in this program. When the consumer has exhausted the buffer, …
… it means that the consumer has consumed an item from the buffer that does not exist:

	Producer	Consumer	s	n	delay
1			1	0	0
2	semWaitB(s)		0	0	0
3	n++		0	1	0
4	if (n == 1) semSignalB(delay)		0	1	1
5	semSignalB(s)		1	1	1
6		semWaitB(delay)	1	1	0
7		semWaitB(s)	0	1	0
8		n--	0	0	0
9		semSignalB(s)	1	0	0
10	semWaitB(s)		0	0	0
11	n++		0	1	0
12	if (n == 1) semSignalB(delay)		0	1	1
13	semSignalB(s)		1	1	1
14		if (n == 0) semWaitB(delay)	1	1	1
15		?	?	?	?
16		?	?	?	?
17		?	?	?	?
18		?	?	?	?
19		?	?	?	?
20		?	?	?	?
21		?	?	?	?

Complete the table.

Pregunta 4 (3 puntos – 15 min.) Implementando semáforos usando monitores (conociendo las definiciones de ambas herramientas de sincronización), ¿de qué consistiría el monitor y cómo funcionaría?

Pregunta 5 (2 puntos – 10 min.) Implementando semáforos usando paso de mensajes (conociendo las definiciones de ambas herramientas de sincronización), ¿cómo se construiría el semáforo y cómo funcionaría?

La práctica ha sido preparada por VK.

Profesor del curso: (0781) V. Khlebnikov

Pando, 26 de septiembre de 2012