PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ
FACULTAD DE CIENCIAS E INGENIERÍA

SISTEMAS OPERATIVOS
2da práctica (tipo a)
(Segundo semestre de 2014)

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.) ¿Cuáles pueden ser los valores finales de la variables *x* después de completar la ejecución concurrente de los procesos *A* y *B* en ambos casos presentados:

Case 1:

```
x = 0

Process A:          Process B:

x = 10*x + 1        x = 10*x + 3
x = 10*x + 2        x = 10*x + 4
```

Case 2:

```
x = 0; sa = Semaphore(0); sb = Semaphore(0)

Process A:          Process B:

x = 10*x + 1        x = 10*x + 3
sa.signal()         sb.signal()
sb.wait()           sa.wait()
x = 10*x + 2        x = 10*x + 4
```

Pregunta 2 (2 puntos – 10 min.) Se distinguen los semáforos fuertes (*strong*) donde los procesos bloqueados en este semáforo forman una cola (FIFO) y los semáforos débiles (*weak*) donde los procesos bloqueados forman un conjunto (sin orden establecido de retiro, y por eso se libera un proceso aleatorio). Los procesos bloqueados en un semáforo débil tienen el peligro de inanición. ¿Existe este peligro para el caso de sincronización de 2 procesos con un semáforo débil? Explíquelo en términos de acciones de la operación *up*() sobre el semáforo. Considere que el valor del semáforo no toma los valores negativos.

Pregunta 3 (7 puntos – 35 min.) (*The Well-Tempered Semaphore: Theme With Variations by Kenneth A. Reek*) **2.1 First Theme: Dijkstra's Semantics.** Dijkstra defined a semaphore as a shared variable that can be manipulated only by the special operations **P** and **V**. He describes these operations as shown in Figure 1.

```
P( s ):
    s = s - 1
    if s < 0 then
        wait on s
    end

V( s ):
    s = s + 1
    if s <= 0 then
        unblock one process waiting on s
    end
```

Figure 1 — Dijkstra's P and V operations

The important property is the manner in which the **V** operation works. If processes are blocked on the semaphore *s*, a **V** causes one of them to be unblocked. The unblocked process has nothing further to check; it can proceed as soon as it gets the CPU.

If a new process arrives and does a **P** before the unblocked process starts running again, … (**(a) 1 punto – ¿qué pasará? ¿Qué procesos tienen la prioridad?**).

I call these semantics … *semaphores* (**(b) 1 punto – ¿*strong* or *weak*?**).

2.2 Variation: POSIX Semantics. POSIX semaphores are a variation on this theme because they have different semantics.

```
sem_wait( s ):
    lock mutex
    while count <= 0 do
        unlock mutex
        sleep
        lock mutex
    done
    count = count - 1
    unlock mutex

sem_post( s ):
    lock mutex
    count = count + 1
    unlock mutex
    wakeup one sleeping process
```

Figure 2 — POSIX semaphores

The POSIX **P** (called **sem_wait**) and **V** (called **sem_post**) are implemented as shown in Figure 2.
If there are sleeping processes **sem_post** awakens one of them. But the awakened process must check the count again – there is nothing to prevent a newly arriving process from doing … (**(c) 2 puntos – ¿qué? Presente para este caso problemático la secuencia de las operaciones de los procesos despertado y nuevo. ¿Qué problema puede surgir?**).

For this reason, the POSIX semantics are labeled as … *semaphores* (**(d) 1 punto – ¿*strong* or *weak*?**).

((e) 2 puntos) Ilustre el comportamiento problemático de los 2 procesos en el siguiente código para las implementaciones de los semáforos según Dijkstra y POSIX:

```
shared semaphore s = 1;

Process A:                  Process B:
    while true do               while true do
        noncritical section         noncritical section
        P( s )                      P( s )
        critical section            critical section
        V( s )                      V( s )
    end                         end
```

Pregunta 4 (4 puntos – 20 min.) (*MOS4E, Chapter 2, Problem 7*) When multiprogramming is used, the CPU utilization can be improved. Crudely put, if the average process computes only 20% of the time it is sitting in memory, then with five processes in memory at once the CPU should be busy all the time. This model is unrealistically optimistic, however, since it tacitly assumes that all five processes will never be waiting for I/O at the same time.

A better model is to look at CPU usage from a probabilistic viewpoint. Suppose that a process spends a fraction p of its time waiting for I/O to complete. With n processes in memory at once, the probability that all n processes are waiting for I/O (in which case the CPU will be idle) is p^n. The CPU utilization is then given by the formula

$$\text{CPU utilization} = 1 - p^n$$

Multiple jobs can run in parallel and finish faster than if they had run sequentially. Suppose that two jobs, each needing 20 minutes of CPU time, start simultaneously. How long will the last one take to complete if they run sequentially? How long if they run in parallel? Assume 50% I/O wait.

Pregunta 5 (3 puntos – 15 min.) Aplicando el algoritmo SRTF, completar la siguiente tabla:

	Arrival time	Burst time	Waiting time	Turnaround time	Average Turnaround time
P1	0	8	?	?	
P2	1	4	?	?	?
P3	2	9	?	?	
P4	3	5	?	?	

La práctica ha sido preparada por VK con LibreOffice Writer en Linux Mint 17.

Profesor del curso: (0781) V. Khlebnikov

Pando, 30 de septiembre de 2014