PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ
FACULTAD DE CIENCIAS E INGENIERÍA

SISTEMAS OPERATIVOS
2da práctica (tipo a)
(Segundo semestre de 2020)

Horario 0781: prof. V. Khlebnikov
Horario 0782: prof. F. Solari A.

Duración: 1 h. 50 min.
Nota: **La presentación, la ortografía y la gramática influirán en la calificación.**
Puntaje total: 20 puntos

Pregunta 1 (8 puntos – 30 min.) Su respuesta debe estar en la carpeta **Buzón 1** de la **Práctica 2** en PAIDEIA **antes de las 11:40**. Por cada 3 minutos de retardo son -2 puntos.
El nombre de su archivo debe ser <*su_código_de_8_dígitos*>`_21.txt`. Por ejemplo, `20171903_21.txt`.

a) (4 puntos) (*Busy Waiting with Hardware Instructions help*) Los procesadores Intel, de tipo o arquitectura IA-32 (antigua iAPX86, conocidos también como x86 y x86_64), no tienen exactamente una instrucción TSL, pero si cuentan con una instrucción XCHG que permite intercambiar el contenido de un registro con una posición de memoria de manera indivisible. Entonces, se plantea el siguiente código para procedimientos `void AcquireLock(int *lock)` y `void ReleaseLock(int *lock)`:

```
void AcquireLock(int *lock) {
   asm {
      L1: MOV  eax,#00000001h   ;colocar el valor #00000001h en el registro eax
          XCHG eax,[lock]       ;intercambiar contenido con variable en memoria
          CMP  eax, #0h         ;comparar con valor #0h
          JNE  L1               ;saltar si no fue igual a 0, a etiqueta L1
          RET                   ;retornar
   }
}

void ReleaseLock(int *lock) {
   asm {
         MOV [lock], #0h        ; poner #0h en lock
   }
}
```

a1. (1 punto) Describa, usando algún código, cómo debería ser el protocolo de uso de este par de procedimientos.

a2. (1 punto) ¿Qué ocurre cuando dos procesos invocan de manera **concurrente** estos procedimientos? Muestre si hay alguna secuencia que podría producir ***deadlock*****,** o si no se cumple la exclusión mutua.

a3. (2 puntos) Muestre en una gráfica en el tiempo (Gantt), cómo dos procesos ejecutan el protocolo, considerando una ***sección_crítica()*** algo más larga que el tiempo máximo asignado a un hilo o proceso.

b) (4 puntos) (*Message Passing*)

b1. (1 punto) ¿Por qué el paso de mensajes, como un método de comunicación de procesos, parece a semáforos y no parece a monitores?

b2. (1 punto) Compare la velocidad de operaciones de paso de mensajes con las velocidades de operaciones con semáforo y la entrada en monitor.

b3. (1 punto) En clase fue presentada la solución del problema de productor-consumidor usando *N* mensajes (las diapositivas 53, 54 de la Clase 6), ¿cuáles son las secciones críticas de los procesos `producer` y `consumer`?

b4. (1 punto) ¿En qué consiste la estrategia ***rendezvous*** en el paso de mensajes?

Pregunta 2 (6 puntos – 30 min.) Su respuesta debe estar en la carpeta **Buzón 2** de la **Práctica 2** en PAIDEIA **antes de las 12:20**. Por cada 3 minutos de retardo son -2 puntos.
El nombre de su archivo debe ser <*su_código_de_8_dígitos*>`_22.txt`. Por ejemplo, `20171903_22.txt`.

La siguiente es una solución propuesta para el problema del barbero dormilón:

```
barbero() {
   while(TRUE) {
     wait(sala);
     lock(mutex);
     sillas=sillas-1;
     unlock(mutex);
     signal(afeitando);
     afeitar();
   }
}
```

```
cliente() {
lock(mutex);
if(sillas<N) {
   signal(sala);
   sillas=sillas+1;
   unlock(mutex);
   wait(afeitando);
}
else {
   unlock(mutex);
}
```

Y el programa principal:

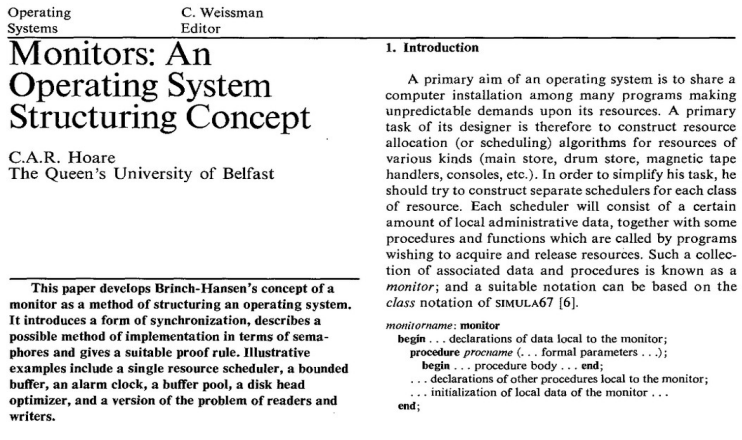
```
main() {
   sillas=0;
   sem_init(afeitando,1);
   sem_init(sala,0);
   mutex_init(mutex,1);
   cobegin {
      barbero(), cliente(), cliente()….
   }
}
```

a) (2 puntos) Describa, sin hacer una traducción literal del código, cómo se sincronizan: barbero con clientes, y clientes entre sí **(1 punto)**; y explique si el conjunto funciona como se espera.

b) (2 puntos) Proponga algún cambio al código propuesto, que cambie la situación de **a)**.

c) (2 puntos) ¿Qué más puede cambiarse o agregarse para asegurar que el cliente no se vaya sin que el barbero termine?

Pregunta 3 (6 puntos – 30 min.) Su respuesta debe estar en la carpeta **Buzón 3** de la **Práctica 2** en PAIDEIA **antes de las 13:00**. Por cada 3 minutos de retardo son -2 puntos.
El nombre de su archivo debe ser <*su_código_de_8_dígitos*>`_23.txt`. Por ejemplo, `20171903_23.txt`.



Operating Systems — C. Weissman, Editor

Monitors: An Operating System Structuring Concept

C.A.R. Hoare
The Queen's University of Belfast

This paper develops Brinch-Hansen's concept of a monitor as a method of structuring an operating system. It introduces a form of synchronization, describes a possible method of implementation in terms of semaphores and gives a suitable proof rule. Illustrative examples include a single resource scheduler, a bounded buffer, an alarm clock, a buffer pool, a disk head optimizer, and a version of the problem of readers and writers.

1. Introduction

A primary aim of an operating system is to share a computer installation among many programs making unpredictable demands upon its resources. A primary task of its designer is therefore to construct resource allocation (or scheduling) algorithms for resources of various kinds (main store, drum store, magnetic tape handlers, consoles, etc.). In order to simplify his task, he should try to construct separate schedulers for each class of resource. Each scheduler will consist of a certain amount of local administrative data, together with some procedures and functions which are called by programs wishing to acquire and release resources. Such a collection of associated data and procedures is known as a *monitor*; and a suitable notation can be based on the *class* notation of SIMULA67 [6].

monitorname: **monitor**
begin . . . declarations of data local to the monitor;
procedure *procname* (. . . formal parameters . . .);
begin . . . procedure body . . . **end**;
. . . declarations of other procedures local to the monitor;
. . . initialization of local data of the monitor . . .
end;

Este artículo fue publicado en *Communications of the ACM, October 1974, Volume 17, Number 10, pp.549-557*.

"… The procedures of a monitor are common to all running programs, in the sense that any program may at any time attempt to call such a procedure. However, it is essential that only one program at a time actually succeed in entering a monitor procedure, and any subsequent call must be held up until the previous call has been completed. Otherwise, if two procedure bodies were in simultaneous execution, the effects on the local variables of the monitor could be chaotic. The procedures local to a monitor should not access any nonlocal variables other than those local to the same monitor, and these variables of the monitor should be inaccessible from outside the monitor. If these restrictions are imposed, it is possible to guarantee against certain of the more obscure forms of time-dependent coding error; and this guarantee could be underwritten by a visual scan of the text of the program, which could readily be automated in a compiler.

Any dynamic resource allocator will sometimes need to delay a program wishing to acquire a resource which is not currently available, and to resume that program after some other program has released the resource required. We therefore need: a "wait" operation, issued from inside a procedure of the monitor, which causes the calling program to be delayed; and a "signal" operation, also issued from inside a procedure of the same monitor, which causes exactly one of

the waiting programs to be resumed immediately. If there are no waiting programs, the signal has no effect. In order to enable other programs to release resources during a wait, a wait operation must relinquish the exclusion which would otherwise prevent entry to the releasing procedure. However, we decree that a signal operation be followed immediately by resumption of a waiting program, without possibility of an intervening procedure call from yet a third program. It is only in this way that a waiting program has an absolute guarantee that it can acquire the resource just released by the signalling program without any danger that a third program will interpose a monitor entry and seize the resource instead.

In many cases, there may be more than one reason for waiting, and these need to be distinguished by both the waiting and the signalling operation. We therefore introduce a new type of "variable" known as a "condition"; and the writer of a monitor should declare a variable of type condition for each reason why a program might have to wait. Then the wait and signal operations should be preceded by the name of the relevant condition variable, separated from it by a dot:

condvariable.wait;
condvariable.signal;

Note that a condition "variable" is neither true nor false; indeed, it does not have any stored value accessible to the program. In practice, a condition variable will be represented by an (initially empty) queue of processes which are currently waiting on the condition; but this queue is invisible both to waiters and signallers. This design of the condition variable has been deliberately kept as primitive and rudimentary as possible, so that it may be implemented efficiently and used flexibly to achieve a wide variety of effects.

…

If more than one program is waiting on a condition, we postulate that the signal operation will reactivate the longest waiting program. This gives a simple neutral queuing discipline which ensures that every waiting program will eventually get its turn.

…

However, in the design of an operating system, there are many cases when such simple scheduling on the basis of first-come-first-served is not adequate. In order to give a closer control over scheduling strategy, we introduce a further feature of a conditional wait, which makes it possible to specify as a parameter of the wait some indication of the priority of the waiting program, e.g.:

condvariable.*wait*(*p*);

When the condition is signalled, it is the program that specified the lowest value of *p* that is resumed.

…

I shall yield to one further temptation, to introduce a Boolean function of conditions:

condvariable.queue

which yields the value true if anyone is waiting on *condvariable* and false otherwise.

…

6. Further Examples

6.2 Disk Head Scheduler

On a moving head disk, the time taken to move the heads increases monotonically with the distance traveled. If several programs wish to move the heads, the average waiting time can be reduced by selecting, first, the program which wishes to move them the shortest distance. But unfortunately this policy is subject to an instability, since a program wishing to access a cylinder at one edge of the disk can be indefinitely overtaken by programs operating at the other edge or the middle.

A solution to this is to minimize the frequency of change of direction of movement of the heads. At any time, the heads are kept moving in a given direction, and they service the program requesting the nearest cylinder in that direction. If there is no such request, the direction changes, and the heads make another sweep across the surface of the disk. This may be called the "elevator" algorithm, since it simulates the behavior of a lift in a multi-storey building.

There are two entries to a disk head scheduler:

(1) *request*(*dest*:*cylinder*);

where

type *cylinder* = 0..*cylmax*;

which is entered by a program just *before* issuing the instruction to move the heads to cylinder *dest*.

(2) *release*;

which is entered by a program when it has made all the transfers it needs on the current cylinder.

The local data of the monitor must include a record of the current headposition, *headpos*, the current direction of *sweep*, and whether the disk is *busy*:

headpos: *cylinder*;
direction: (*up*, *down*);
busy: *Boolean*

We need two conditions, one for requests waiting for an *upsweep* and the other for requests waiting for a *downsweep*:

upsweep, *downsweep*: *condition*

```
dischead: monitor
begin
        headpos: cylinder;
        direction: (up, down);
        busy: Boolean;
        upsweep, downsweep: condition;
   procedure request(dest: cylinder);
      begin if busy then
         {if headpos < dest ∨ headpos = dest & direction = up
            then upsweep.wait(dest)
            else downsweep.wait(cylmax - dest) };
        busy := true; headpos := dest
      end request;
   procedure release;
      begin busy := false;
         if direction = up then
            {if upsweep.queue then upsweep.signal
                                 else {direction := down;
                                       downsweep.signal} }
         else if downsweep.queue then downsweep.signal
                                    else {direction := up;
                                          upsweep.signal}
      end release;
      headpos := 0; direction := up; busy := false
end dischead;
"
```

Considere que 4 procesos (A, B, C y D) hacen las solicitudes al ***mismo tiempo*** pero en este orden:

```
A:      request(200);
B:      request(50);
C:      request(700);
D:      request(100);
```

Claro que solo el primer proceso podrá salir del procedimiento *request* mientras que otros procesos se bloquearán en las variables de condición. Al salir del *request* el primer proceso completará su transferencia de datos al disco y invocará al procedimiento *release*. Esto hará cada proceso que logrará a salir del procedimiento *request*.

Complete la traza de la secuencia de ejecución de todos los procesos considerando que la variables *cylmax* tiene el valor 1000 y manteniendo el siguiente formato de su respuesta:

```
A:      request(200);
                busy ← true; headpos ← 200;
        operación de E/S; CPU está libre
B:      request(50);
                … .wait(…) se bloquea en la cola de …
C:      request(700);
                … .wait(…) se bloquea en la cola de …
D:      request(100);
                … .wait(…) se bloquea en la cola de …
A:      completa la operación de E/S
        release;
                busy ← false; …
…
```

La práctica ha sido preparada por FS (1a,2) y VK (1b,3)
con LibreOffice Writer en Linux Mint 20 “Ulyana”

Profesores del curso: (0781) V. Khlebnikov
(0782) F. Solari A.

Lima, 16 de octubre de 2020