PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ
FACULTAD DE CIENCIAS E INGENIERÍA

SISTEMAS OPERATIVOS
2da práctica (tipo a)
(Segundo semestre de 2016)

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

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 (5 puntos – 25 min.) (*PCDP2E by M. Ben-Ari*) Exercise 5. (Apt and Olderog) Assume that for the function *f*, there is some integer value *i* for which *f*(*i*) = 0. Here are five concurrent algorithms that search for *i* (usando 2 procesos que lo buscan en dos segmentos de valores diferentes). An algorithm is correct if for all scenarios, ***both*** processes terminate after one of them has found the zero. For each algorithm, show that it is correct or find a scenario that is a counterexample.

Algorithm 2.11: Zero A	
boolean found	
p	**q**
integer i ← 0	integer j ← 1
p1: found ← false	q1: found ← false
p2: while not found	q2: while not found
p3: i ← i + 1	q3: j ← j − 1
p4: found ← f(i) = 0	q4: found ← f(j) = 0

Algorithm 2.12: Zero B	
boolean found ← false	
p	**q**
integer i ← 0	integer j ← 1
p1: while not found	q1: while not found
p2: i ← i + 1	q2: j ← j − 1
p3: found ← f(i) = 0	q3: found ← f(j) = 0

Algorithm 2.13: Zero C	
boolean found ← false	
p	**q**
integer i ← 0	integer j ← 1
p1: while not found	q1: while not found
p2: i ← i + 1	q2: j ← j − 1
p3: if f(i) = 0	q3: if f(j) = 0
p4: found ← true	q4: found ← true

In process p of the following two algorithms, `await turn = 1` and `turn` ← 2 are executed as a one atomic statement when the value of `turn` is 1; similarly, `await turn = 2` and `turn` ← 1 are executed atomically in process q.

Algorithm 2.14: Zero D

boolean found ← false
integer turn ← 1

p	q
integer i ← 0	integer j ← 1
p1: while not found	q1: while not found
p2: await turn = 1	q2: await turn = 2
turn ← 2	turn ← 1
p3: i ← i + 1	q3: j ← j − 1
p4: if f(i) = 0	q4: if f(j) = 0
p5: found ← true	q5: found ← true

Algorithm 2.15: Zero E

boolean found ← false
integer turn ← 1

p	q
integer i ← 0	integer j ← 1
p1: while not found	q1: while not found
p2: await turn = 1	q2: await turn = 2
turn ← 2	turn ← 1
p3: i ← i + 1	q3: j ← j − 1
p4: if f(i) = 0	q4: if f(j) = 0
p5: found ← true	q5: found ← true
p6: turn ← 2	q6: turn ← 1

Pregunta 2 (5 puntos – 25 min.) (POSIX Threads, part2 - *Daniel Robbins, IBM developerWorks*) El siguiente programa en lenguaje C, utiliza pthreads para demostrar el trabajo con secciones críticas. Se ha modificado ligeramente para, en lugar de usar *mutexes* de la librería POSIX pthreads, hacerlo mediante un algoritmo *busy waiting* conocido.

```
alulab@linuxmint18 ~/INF239-Pa2-2/ $ cat thread4.c
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

int myglobal;
int turn;
int interested[2];

#define TRUE 1
#define FALSE 0

void enter_region(int thread) { /* main thread 0, thread 1 */
   int other;
   other = 1 - thread;
   interested[thread] = TRUE;
   turn = thread;
   while(turn == thread && interested[other] == TRUE) ;
}
```

```
void leave_region(int thread) {
   interested[thread] = FALSE;
}

void *thread_function(void *arg) {
        int i,j,k,x=0;
        for ( i=0; i<20; i++ ) {
                enter_region(1);
                j=myglobal;
                j=j+1;
                printf(".");
                fflush(stdout);
                /* sleep(1); */

                for(k=0;k<30000;k++) x=x+1;
                myglobal=j;
                leave_region(1);
        }
        return NULL;
}
int main(void) {
        pthread_t mythread;
        int i,k,x=0;
        if ( pthread_create( &mythread, NULL, thread_function, NULL) ) {
                printf("error creating thread.");
                abort();
        }
        for ( i=0; i<20; i++) {
                enter_region(0);
                myglobal=myglobal+1;
                printf("o");
                fflush(stdout);
                leave_region(0);
                /* sleep(1); */
                for(k=0;k<30000;k++) x=x+1;

        }
        if ( pthread_join ( mythread, NULL ) ) {
                printf("error joining thread.");
                abort();
        }
        printf("\nmyglobal equals %d\n",myglobal);
        exit(0);
}
alulab@linuxmint18 ~/INF239-Pa2-2/ $ gcc -pthread -lpthread -o thread4 thread4.c
alulab@linuxmint18 ~/INF239-Pa2-2/ $ time ./thread4
o.o.o.o.o.o.o.o.o.o.o.o.o.o.o.o.o.o.o.o.o.o.
myglobal equals 40

real    0m0.014s
user    0m0.026s
sys     0m0.000s
```

a) (3 puntos – 15 min.) Determine cuáles son las líneas que se consideran la Sección Crítica en cada hilo y discuta si son las necesarias para proteger el/los recursos compartidos o puede mejorarse. Explique cómo se protegen las secciones críticas si los hilos intentan a la vez ingresar a su Sección Crítica.

b) (2 puntos – 10 min.) Una crítica que se hace a la solución anterior, es la **espera ocupada, o *busy-waiting***. Con la librería POSIX pthreads, se puede ejecutar *pthread_yield()* como cuerpo del lazo de espera. El programa *thread5.c* contiene esta modificación. Su ejecución mediante **`time`** y resultado:

```
alulab@linuxmint18 ~/INF239-Pa2-2/ $ time ./thread5
o.o.o.o.o.o.o.o.o.o.o.o.o.o.o.o.o.o.o.o.o.o.
myglobal equals 40

real    0m0.010s
user    0m0.009s
sys     0m0.009s
```

El tiempo ***real***, es el tiempo trascurrido durante toda la ejecución, mientras que el tiempo ***user*** es el utilizado en CPU en modo usuario, y el tiempo ***sys*** el utilizado en CPU en modo kernel (por el sistema operativo, a cuenta del proceso e hilos) (en estos dos últimos casos, se suman tiempos en cada CPU, si hay más de una).

¿Cómo mejoran los tiempos de ejecución por el uso de *pthread_yield()*? Explique y justifique ayudándose de una gráfica en el tiempo (no escalada, pero algo proporcional), que muestre la ejecución de los hilos para thread4 y thread5.

Pregunta 3 (5 puntos – 25 min.) (*Implementing counting semaphores using binary semaphores, A. Udaya Shankar*) Assignment is ←. Equality is "=". `CSem` stands for counting semaphores. `BSem` stands for binary semaphores.

Implementation 1 (incorrect):

```
CSem(K) cs {    // counting sem, init K
    int val ← K;    // value of csem
    BSem wait(0);    // to block on csem
    BSem mutex(1);   // protects val

    Pc(cs) {
        P(mutex);
        val ← val - 1;
        if val < 0 {
            V(mutex);
      1:    P(wait);
        } else
            V(mutex);
    }

    Vc(cs) {
        P(mutex);
        val ← val + 1;
        if val ≤ 0
            V(wait);
        V(mutex);
    }
}
```

Evolution showing error:

```
Initial:
    cs = 0; val = 0; wait = 0; mutex = 1.

Thread t1 attempts Pc(cs):
    t1 at 1; val = -1; wait = 0; mutex = 1.

Thread t2 attempts Pc(cs):
    t1, t2 at 1; val = -2; wait = 0; mutex = 1.

Thread t3 executes Vc(cs) twice:
    t1, t2 at 1; val = 0; wait = 1; mutex = 1.

Thread t2:
    t1 at 1; ...

Thread t1:
    ...
```

a) (3 puntos – 15 min.) Complete la información del desarrollo de los hilos `t2` y `t1`.

b) (2 puntos – 10 min.) Presente el desarrollo del mismo escenario para la siguiente implementación:

Implementation 2:

```
CSem(K) cs {          // counting semaphore initialized to K
    int val ← K;             // the value of csem
    BSem gate(min(1,val));    // 1 if val > 0; 0 if val = 0
    BSem mutex(1);            // protects val

    Pc(cs) {
        P(gate);
    a1: P(mutex);
        val ← val - 1;
        if val > 0
            V(gate);
        V(mutex);
    }

    Vc(cs) {
        P(mutex);
        val ← val + 1;
        if val = 1
            V(gate);
        V(mutex);
    }
}
```

Pregunta 4 (5 puntos – 25 min.) (Modificado de cs.villanova.edu, mdamian – PC.java) Java associates a monitor with each object. The monitor enforces mutual exclusive access to synchronized methods invoked on the associated object. When a thread calls a synchronized method on an object, the JVM checks the monitor for that object:

- If the monitor is unowned, ownership is assigned to the calling thread, which is then allowed to proceed with the method call.
- If the monitor is owned by another thread, the calling thread will be put on hold until the monitor becomes available.

When a thread exits a synchronized method, it releases the monitor, allowing a waiting thread (if any) to proceed with its synchronized method call.

El siguiente código, corresponde a una ligera modificación del código PC.java, como ProducerConsumer.java, modificaciones en los mensajes enviados a salida estándar, y esperas en CPU frente a esperas con *Sleep(1):*

```
public class ProducerConsumer {
    /**
     * @param args the command line arguments
     */
   public static void main(String[] args) {
     Buffer b = new Buffer(4);
     Producer p = new Producer(b);
     Consumer c = new Consumer(b);
     p.start();
     c.start();
    }
}

class Buffer {
   private char [] buffer;
   private int count = 0, in = 0, out = 0;
   Buffer(int size)
   {
      buffer = new char[size];
   }
   public synchronized void Put(char c) {
     while(count == buffer.length) {  }
     buffer[in] = c;
     in = (in + 1) % buffer.length;
     count++;
     System.out.println("PUT " + c + " ...");
     System.out.flush();
     notify();
   }
   public synchronized char Get() {
     while (count == 0)  {  }
     char c = buffer[out];
     out = (out + 1) % buffer.length;
     count--;
     System.out.println("GET " + c + " ...");
     System.out.flush();
     return c;
   }
}

class Producer extends Thread {
   private Buffer buffer;
   Producer(Buffer b) { buffer = b; }
   public void run() {
     for(int i = 0; i < 10; i++) {
         int x = 0;
         char c = (char)('A'+i%26);
         System.out.println("Producing " + c + " ...");
         System.out.flush();
         for(long j=0; j< 10; j++) { x = x+1 ;}
         buffer.Put(c);
     }
   }
}

class Consumer extends Thread {
   private Buffer buffer;
   Consumer(Buffer b) { buffer = b; }
   public void run() {
     char r;
     for(int i = 0; i < 10; i++) {
        int x = 0;
        r = buffer.Get();
```

```
        System.out.println("Consuming " + r + " ...");
        System.out.flush();
        for(long j=0; j< 10; j++) { x = x+1 ;}
      }
    }
}
```

a) (2 puntos – 10 minutos) El programa, como está, llega a un ***deadlock***. Haga una ejecución (alternada o no) de los hilos Producer y Consumer, que muestre la situación descrita, a pesar de la exclusión mutua que brindan los métodos ***syncronized*** como procedimientos o funciones de un ***monitor***.

b) (3 puntos – 15 minutos) Agregue las sentencias necesarias para que NO se produzca el ***deadlock*****,** en base a las condiciones del problema Productor – Consumidor expuesto. Haga una ejecución (alternada o no) de los hilos Producer y Consumer, que muestre que ya no se dá la situación anterior, es decir, no se producirá ***deadlock***.

La práctica ha sido preparada por FS(2,4) y VK(1,3)
con LibreOffice Writer en Linux Mint 18 Sarah.

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

Pando, 23 de septiembre de 2016