EXAMEN SISTEMAS OPERATIVOS AVANZADOS

VIERNES 16 DICIEMBRE 2016

NOMBRE Y FIRMA:

P1:	/0.5	P3:	/1.5	LAB3,P1:	/1.5	LAB3,P2:	/1.5
P2	/0.5	P4:	/1.5	LAB4,P1:	/1.5	LAB4,P2:	/1.5

PROBLEMA 1 (0.5 PUNTO)

Varios *threads* llaman periódicamente la siguiente rutina, para estar seguros de que la "pipe" ha sido abierta. Después de esta rutina, podríamos seguir hacia delante llamando a la función *write()* (por ejemplo). Supón que hay una variable entera global llamada "*pipe*", que toma valor -1 usando la pipe está cerrada. Supón que disponemos de un *lock* global llamado "*lock*" usado para sincronización. El código en cuestión es:

```
void MakeSurePipeIsOpen() {
  mutex_lock(&lock);
  if (pipe == -1)
    pipe = open(''/tmp/fifo'', O_WRONLY);
  mutex_unlock(&lock);
}
```

Sin embargo, se decide rescribir ese código a:

```
void MakeSurePipeIsOpen() {
  if (pipe == -1) {
      mutex_lock(&lock);
   if (pipe == -1)
     pipe = open(''/tmp/fifo'', O_WRONLY);
   mutex_unlock(&lock);
  }
}
```

¿Sigue funcionando correctamente este código? Si es así, qué ventajas tiene sobre la primera implementación. En caso contrario. ¿Por qué no funciona?

PROBLEMA 2 (0.5 PUNTO)

¿Es el siguiente código una solución válida para el problema de los filósofos? Si no lo es propón una válida.

```
adquire(int i) {
   if (i < 4){
    sem_wait(palillo[i]);
    sem_wait(palillo[i+1]);
   } else {
    sem_wait(palillo[0]);
    sem_wait(palillo[4]);
   }
 }
```

PROBLEMA 3 (1.5 PUNTOS)

El siguiente código corresponde a la cabecera de un "objeto" C tipo vector de enteros (similar a la clase Vec de java), con solo un método de suma.

```
#define VECTOR_SIZE (100)
typedef struct __vector {
    pthread_mutex_t lock;
    int values[VECTOR_SIZE];
} vector_t;

void vector_add(vector_t *v_dst, vector_t *v_src)
```

Las implementaciones *thread-safe* de la rutina de suma alternativas son las siguientes. Desafortunadamente ninguna de ellas funciona bien. Una no tiene solución y las otras dos sí (modificando el orden de las líneas que ya existen). Indicar las correcciones de las que son resolubles a la derecha (0.5 puntos por rutina corregida o identificada).

```
void vector_add(vector_t *v_dst, vector_t *v_src) {
    Pthread_mutex_lock(&v_dst->lock);
    Pthread_mutex_lock(&v_src->lock);
    int i;
    for (i = 0; i < VECTOR_SIZE; i++) {
        v_dst->values[i] = v_dst->values[i] + v_src->values[i];
    }
    Pthread_mutex_unlock(&v_dst->lock);
    Pthread_mutex_unlock(&v_src->lock);
}
```

```
void vector_add(vector_t *v_dst, vector_t *v_src) {
    if (v_dst <= v_src) {
        Pthread_mutex_lock(&v_dst->lock);
        Pthread_mutex_lock(&v_src->lock);
    } else if (v_dst > v_src) {
        Pthread_mutex_lock(&v_src->lock);
        Pthread_mutex_lock(&v_dst->lock);
    }
    int i;
    for (i = 0; i < VECTOR_SIZE; i++) {
        v_dst->values[i] = v_dst->values[i] + v_src->values[i];
    }
    Pthread_mutex_unlock(&v_src->lock);
    Pthread_mutex_unlock(&v_dst->lock);
}
```

```
pthread_mutex_t global = PTHREAD_MUTEX_INITIALIZER;

void vector_add(vector_t *v_dst, vector_t *v_src) {
    Pthread_mutex_lock(&global);
    Pthread_mutex_lock(&v_dst->lock);
    Pthread_mutex_lock(&v_src->lock);
    int i;
    for (i = 0; i < VECTOR_SIZE; i++) {
        v_dst->values[i] = v_dst->values[i] + v_src->values[i];
    }
    Pthread_mutex_unlock(&v_dst->lock);
    Pthread_mutex_unlock(&v_src->lock);
    Pthread_mutex_unlock(&global);
}
```

PROBLEMA 4 (1.5 PUNTOS)

Supón la siguiente implementación del problema consumidor/productor con buffer limitado

```
int buf [max];

void *consumer (void *arg) {
  while(1) {
    Pthread_mutex_lock(&mutex);      //c1
    while (count == 0)               //c2
      Phread_cond_wait(&fill,&mutex);//c3
    int tmp = get();                 //c4
    Pthread_cond_signal(&empty);     //c5
    Pthread_mutex_unlock(&empty);    //c6
    print("%d\n", tmp);
  }
}
```

```
void *producer (void *arg) {
  for (int i = 0; i<loops; i++ ) {
    Pthread_mutex_lock(&mutex);      //p1
    while (count == 0)               //p2
      Phread_cond_wait(&empty,&mutex);//p3
    put(i);                          //p4
    Pthread_cond_signal(&fill);      //p5
    Pthread_mutex_unlock(&empty);    //p6
  }
}
```

Supón que cada *thread* solo puede ser bloqueado en el *lock* o la variable condicional (es decir, no hay interrupciones del timer). El sistema tiene una sola CPU. Presentar la traza de ejecución de las líneas presentadas en los siguientes escenarios:

a) Hay 1 Productor (P), 1 Consumidor (C) y max=1. El productor se ejecuta antes. Parar cuando el consumidor haya procesado una entrada.
   P:
   C:

b) Hay 1 Productor (P), 1 Consumidor (C) y max=3. El productor se ejecuta antes. Parar cuando el consumidor haya procesado una entrada.
   P:
   C:

c) Hay 1 Productor (P), 1 Consumidor (C) y max=1. El consumidor se ejecuta antes. Parar cuando el consumidor haya procesado una entrada.
   P:
   C:

d) Hay 1 Productor (P), 2 Consumidores (C1, C2) y max=1. C1 se ejecuta antes, después C2 y después P3. Parar cuando P haya procesado una salida.
   P:
   C1:
   C2:

e) Ahora cambia los "*while*" por "*if*". Cuál es la traza cuando Hay 1 Productor (P), 2 Consumidores (C1, C2) y max=indederminado. Esto va dar problemas!
   P:
   C1:
   C2:

f) Ahora, usando de nuevo "*while*", supón que se está usando una sola variable condicional. Cuál es la traza cuando Hay 1 Productor (P), 2 Consumidores (C1, C2) y max=indederminado. Esto va dar problemas!
   P:
   C1:
   C2:

PREVIO

Descargar el repositorio de trabajo desde: https://gitlab.com/AOSUC/examen2/

Dentro hay dos directorios llamados “Lab3” y “Lab4”. Cada uno de ellos contiene una implementación parcial (y defectuosa) de las prácticas 3 y 4. Se proveen de dos programas de usuario (que forman parte de los *tests* empleados) llamados P1 y P2. Ninguno de los dos logra pasar. Indicar debajo donde están los problemas en cada uno de ellos (indicando fichero, función y la línea/s incorrecta y la/s corregida/s)

LAB 3 P1 (1.5 PUNTO)

LAB 3 P2 (1.5 PUNTO)

LAB 4 P1 (1.5 PUNTO)

LAB 4 P2 (1.5 PUNTO)