UNIVERSIDAD DE CÓRDOBA
ESCUELA POLITÉCNICA SUPERIOR
DEPARTAMENTO DE INFORMÁTICA Y ANÁLISIS NUMÉRICO

ASIGNATURA *SISTEMAS OPERATIVOS*

2º DE GRADO EN INGENIERÍA INFORMÁTICA

PRÁCTICA 4

Comunicación entre procesos ligeros – Semáforos generales

Profesorado: Juan Carlos Fernández Caballero
Alberto Cano Rojas

Índice de contenido

1 Objetivo de la práctica

La presente práctica persigue familiarizar al alumnado con la comunicación y sincronización entre procesos o hilos para el acceso en exclusión mutua a recursos compartidos. En un primera parte se explicará brevemente teoría sobre semáforos generales, siendo en la segunda parte de la práctica cuando, mediante programación en C, se practicarán los conceptos sobre esta temática, utilizando para ello las rutinas de interfaz del sistema que proporcionan a los programadores la librería *semaphore* basada en el estándar POSIX.

2 Recomendaciones

El lector debe completar las nociones dadas en las siguientes secciones con consultas bibliográficas, tanto en la Web como en la biblioteca de la Universidad, ya que unos de los objetivos de las prácticas es potenciar su capacidad autodidacta y su capacidad de análisis de un problema. Es recomendable que, aparte de los ejercicios prácticos que se proponen, pruebe y modifique otros que se encuentren en la Web (se dispone de una gran cantidad de problemas resueltos en C sobre esta temática), ya que al final de curso deberá acometer un examen práctico en ordenador como parte de la evaluación de la asignatura.

No olvide que debe consultar siempre que lo necesite el estándar POSIX http://pubs.opengroup.org/onlinepubs/9699919799/ y la librería GNU C (*glibc*) en http://www.gnu.org/software/libc/libc.html.

Los conceptos teóricos que se exponen a continuación se aplican tanto a procesos como a hilos, pero la parte práctica de este guión va dirigida a la resolución de problemas de concurrencia mediante hilos. Para obtener soluciones usando procesos el lector debería utilizar las rutinas de interfaz para memoria compartida entre procesos que proporcione el sistema que esté utilizando. POSIX (*POSIX Shared Memory)*[1] proporciona también un estándar para ello que implementan los sistemas basados en UNIX.

3 Conceptos teóricos

3.1 Semáforos

Un semáforo (propuesto por Dijkstra en 1965) es un mecanismo de sincronización que se utiliza generalmente en sistemas monoprocesador o multiprocesador.

Un semáforo es un objeto o estructura con un valor entero al que se le puede asignar un valor inicial no negativo, con una cola de procesos, y al que sólo se puede acceder utilizando dos operaciones atómicas: *wait()* y *signal ()*. Su filosofía es parecida a los mutexs que estudió en la práctica anterior, ya que se basan en que dos o más procesos pueden cooperar por medio de simples señales, tales que un proceso pueda ser obligado a parar en un lugar específico hasta que haya recibido una señal específica de otro.

Hay dos tipos de semáforos: **Semáforos binarios**, que se corresponden con los mutexs POSIX estudiados en la práctica anterior, y los **semáforos generales o con contador**, que son los que estudiaremos aquí.

1 http://en.wikipedia.org/wiki/Shared_memory

De manera abstracta (veremos más adelante la implementación POSIX), las operaciones *wait()* y *signal()* de un semáforo general o con contador (se entiende de aquí en adelante) son las siguientes:

```
wait (s) /* Operación atómica wait() para un semáforo "s" */
{
        s = s –1;
        if (s < 0)
        {
                Poner proceso en cola;
                Bloquear al proceso;
        }
}
```

La operación *wait()* decrementa el valor del semáforo "s". Si el valor pasa a ser negativo, entonces el proceso que está ejecutando *wait()* se bloquea. Cuando al decrementar "s" el valor es cero o positivo, el proceso entra en la sección crítica. El semáforo se suele inicializar a 1, para que el primer proceso que llegue, A, lo decremente a cero y entre en su sección crítica. Si algún otro proceso llega después, B, y A todavía está en la sección crítica, B se bloquea porque al decrementar el contador pasa a ser negativo.

Si el valor inicial del semáforo fuera, por ejemplo, 2, entonces dos procesos podrían ejecutar la llamada *wait()* sin bloquearse y por tanto se permitiría que ambos ejecutaran de forma simultánea dentro de la sección crítica. Algo peligroso si se realizan operaciones de modificación en la sección crítica y no se controlan.

```
signal (s) /* Operación atómica signal() para un semáforo "s" */
{
        s = s + 1;
        if ( s <= 0 )
        {
                Extraer un proceso de la cola bloqueado en la operación wait();
                Poner el proceso listo para ejecutar;
        }
}
```

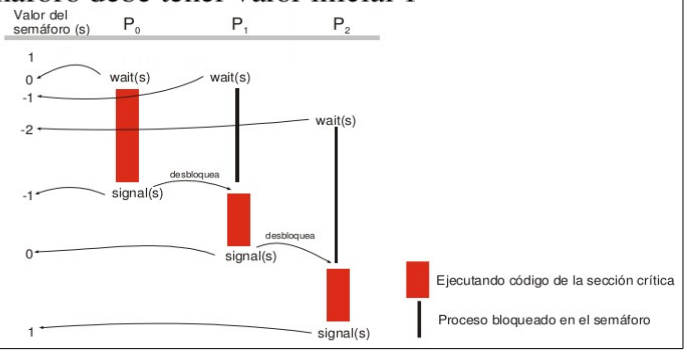
La operación *signal()* incrementa el valor del semáforo. Si el valor es menor o igual que cero, entonces se desbloquea uno de los procesos bloqueados en la operación *wait()*.

El número de procesos, que en un instante determinado se encuentran bloqueados en una operación *wait()*, viene dado por el valor absoluto del semáforo si es negativo. Cuando un proceso ejecuta la operación *signal()*, el valor del semáforo se incrementa, y en el caso de que haya algún proceso bloqueado en una operación *wait()* anterior, se desbloqueará a un solo proceso.

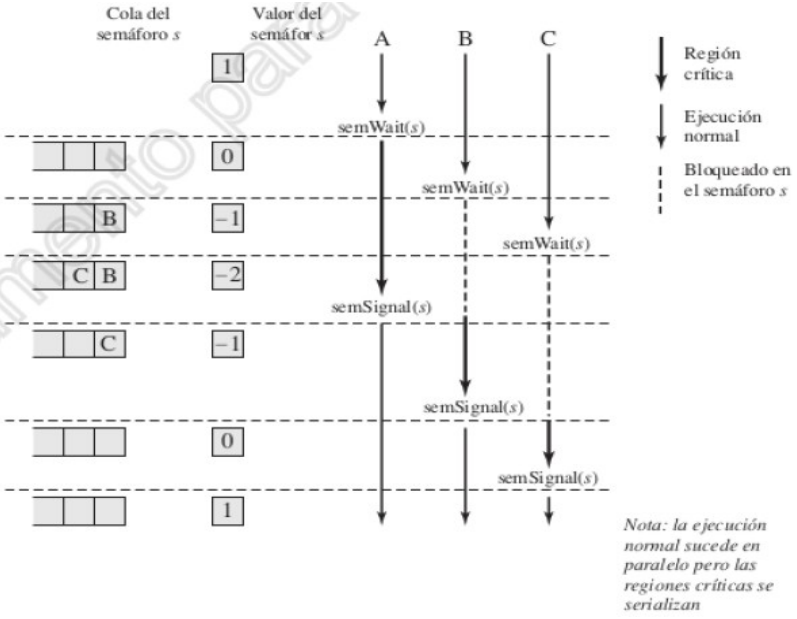
Para resolver el problema de la sección crítica utilizando semáforos debemos proteger el código que constituye la sección crítica de la siguiente forma (ver figura):

```
wait(s); /* entrada en la seccion critica */
< seccion critica >
signal(s);  /* salida de la seccion critica */
```

El semáforo debe tener valor inicial 1



La Figura siguiente figura muestra una posible secuencia de tres procesos usando la disciplina de exclusión mutua explicada anteriormente. En este ejemplo, tres procesos (A, B, C) acceden a un recurso compartido protegido por el semáforo "s". El proceso A ejecuta semWait(), y dado que el semáforo tiene el valor 1 en el momento de la invocación, A puede entrar inmediatamente en su sección crítica y el semáforo toma el valor 0. Mientras A está en su sección crítica, ambos B y C realizan una operación semWait() y son bloqueados, pendientes de la disponibilidad del semáforo. Cuando A salga de su sección crítica y realice semSignal(), B, que fue el primer proceso en la cola, podría entonces entrar en su sección crítica. Cuando B haga otro semSignal() y salga de la sección crítica, entonces C podrá entrar en la misma.



El estándar POSIX especifica una interfaz para los semáforos generales (los binarios son los mutexs). Esta interfaz no es parte de la librería *pthread,* es decir, *pthread* no proporciona funciones para el manejo de semáforos generales. A pesar de ello, la mayoría de los sistemas Unix actuales que implementan hilos también proporcionan semáforos generales a partir de otra librería, concretamente la librería *semaphore*, definida en *"semaphore.h"*. Esta librería se basa en el

estándar POSIX.

Existe otra implementación de semáforos, conocida como "semáforos en *Unix System V*", que es la implementación tradicional de Unix, pero son más complejos de utilizar y no aportan más funcionalidad. Se define en *<sys/sem.h>*. A partir de la versión de kernel 2.6 de los sistemas Linux actuales, los semáforos POSIX se soportan por el sistema operativo y se puede utilizar sin problema *semaphore.h*. Si alguna vez tuviéramos que usar semáforos en máquinas con sistemas operativos muy antiguos, sería más conveniente usar los semáforos *System V*, ya que son más portables a equipos viejos.

3.2 Semáforos generales POSIX

En POSIX un semáforo se identifica mediante una variable del tipo *sem_t*. El estándar POSIX define dos tipos de semáforos:

• **Semáforos sin nombre.** Permiten sincronizar a los procesos ligeros que ejecutan dentro de un mismo proceso, o a los procesos que heredan el semáforo a partir de una llamada *fork()* (padre e hijo). En este segundo caso habría que utilizar técnicas de memoria compartida entre procesos para el acceso a la sección crítica, por lo que en esta práctica nos ceñiremos al uso de hilos.

• **Semáforos con nombre**. En este segundo tipo de semáforos, éstos llevan asociado un nombre que sigue la convención de nombrado que se emplea para archivos. Con este tipo de semáforos se pueden sincronizar procesos sin necesidad de que tengan que heredar el semáforo utilizando la llamada *fork()*, en este caso los procesos tienen que abrir el semáforo con el mismo nombre. El estándar tiene en cuenta la posibilidad de su utilización para la sincronización de procesos (además de hilos), pero esta posibilidad no está soportada en todas las implementaciones y por ello, en esta asignatura, sólo serán utilizados los semáforos sin nombre.

3.2.1 Inicialización de un semáforo (sem_init())

Los semáforos deben inicializarse antes de usarlos. Para ello utilizaremos la siguiente función[2]:

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, int value);
```

- sem: Puntero a parámetro de tipo *sem_t* que identifica el semáforo.
- pshared: Entero que determina si el semáforo sólo puede ser utilizado por hilos del mismo proceso que inicia el semáforo, en cuyo caso pondremos como parámetro el valor 0, o se puede utilizar para sincronizar procesos que lo hereden por medio de una llamada a *fork()* (! =0). En esta práctica no se contempla el último caso.
- value: Entero que representa el valor que se asigna inicialmente al semáforo.

Esta función devuelve 0 en caso de que que pueda inicializar el semáforo o -1 en caso contrario estableciendo el valor del error en *errno (*[EINVAL], [ENOSPC], [EPERM]*)*. Consulte *OpenGroup*.

2 http://pubs.opengroup.org/onlinepubs/9699919799/functions/sem_init.html

3.2.2 Petición de bloqueo o bajada del semáforo (sem_wait())

Para entrar en la sección crítica por parte de un proceso antes se debe consultar el semáforo con *sem_wait()*[3]:

```
#include <semaphore.h>
int sem_wait(sem_t *sem);
```

- sem: Puntero a parámetro de tipo *sem_t* que identifica el semáforo.

Como se comento al principio de la práctica, al realizar esta llamada, el contador del semáforo se decrementa. Si el valor pasa a ser negativo, entonces el proceso que está ejecutando *sem_wait()* se bloquea.

Esta función devuelve 0 en caso de que no haya problemas con la llamada o -1 en caso contrario estableciendo el valor del error en *errno*.

Otra función similar es *sem_trywait()*:

```
#include <semaphore.h>
int sem_trywait(sem_t *sem);
```

La función *sem_trywait()* es igual que *sem_wait()* pero no bloqueante, es decir, si el valor del semáforo al decrementarse llega a negativo el hilo invocador de *sem_trywait()* no se bloquea sino que continua con su ejecución. En este caso la función *sem_trywait()* devuelve el valor -1, estableciendo *errno* el valor EAGAIN. Consulte *OpenGroup*. En caso de que al invocarse el decremento no llegase a negativo la función *sem_trywait()* devuelve 0 y el proceso entra en sección crítica. Por tanto es una función que permite a un proceso seguir con la siguiente sentencia de código despues de *sem_trywait()* en caso de que la sección crítica no hay sido desbloqueada. En función de eso quizás nos interese que nuestro hilo haga otras tareas alternativas.

3.2.3 Petición de desbloqueo o subida del semáforo (sem_post())

Cuando un proceso va a salir de la sección crítica es necesario que envíe una señal indicando que ésta queda libre, y eso lo hace con la función *sem_post()*[4] (equivalente al *semSignal()* teórico comentado al principio de la práctica):

```
#include <semaphore.h>
int sem_post(sem_t *sem);
```

- sem: Puntero a parámetro de tipo *sem_t* que identifica el semáforo.

La operación *sem_post()* incrementa el valor del semáforo. Si el valor es menor o igual que cero, entonces se desbloquea uno de los procesos bloqueados en la operación *sem_wait(),* normalmente se suele emplear para ello un esquema FIFO (First In First Out). Si el valor del semáforo resultante de

3 http://pubs.opengroup.org/onlinepubs/9699919799/functions/sem_wait.html
4 http://pubs.opengroup.org/onlinepubs/9699919799/functions/sem_post.html

esta operación es positivo (>0), entonces es que no hay hilos bloqueados esperando en el semáforo para desbloquearse, por lo que el valor del semáforo es simplemente incrementado. Esto puede suceder por ejemplo cuando tengamos problemas en los que se acepte más de un proceso en sección crítica.

Esta función devuelve 0 en caso de que no haya problemas con la llamada o -1 en caso contrario estableciendo el valor del error en *errno*. Consulte *OpenGroup*.

3.2.4 Examinar el valor de un semáforo (sem_getvalue())

Si necesitamos saber cuál es el valor de un semáforo podemos consultarlo con la siguiente función[5]:

```
#include <semaphore.h>
int sem_getvalue(sem_t *sem, int *sval);
```

- sem: Puntero a parámetro de tipo *sem_t* que identifica el semáforo.
- sval: Puntero a variable de tipo entero en la que se depositará el valor del semáforo.

El uso de *sem_getvalue()* no altera el valor del semáforo. *sval* mantiene el valor que tenía el semáforo en algún momento no especificado durante la llamada, pero no necesariamente el valor en el momento de la devolución. Tenga en cuenta que puede haber más de un hilo intentando acceder al semáforo indicado como primer parámetro, o simplemente, por ejemplo, antes de que *sem_getvalue()* nos devuelva algo un hilo puede haber salido de la sección crítica.

Si el semáforo está bloqueado, *sval* contendrá el valor cero o un valor negativo que indica, en valor absoluto, el número de subprocesos en espera.

Esta función devuelve 0 en caso de que no haya problemas con la llamada o -1 en caso contrario estableciendo el valor del error en *errno*. Consulte *OpenGroup*

3.2.5 Destrucción de un semáforo (sem_destroy())

Para destruir un semáforo previamente creado con *sem_init()* se utiliza la siguiente función[6]:

```
#include <semaphore.h>
int sem_destroy(sem_t *sem);
```

- sem: Puntero al semáforo a destruir para liberar recursos.

Esta función devuelve 0 en caso de que que pueda destruir el semáforo o -1 en caso contrario estableciendo el valor del error en *errno*. Consulte *OpenGroup* para el tratamiento de errores.

5 http://pubs.opengroup.org/onlinepubs/9699919799/functions/sem_getvalue.html
6 http://pubs.opengroup.org/onlinepubs/9699919799/functions/sem_destroy.html

3.2.6 Ejemplos

3.2.6.1 Ejemplo1

El siguiente programa crea dos hilos. Cada hilo hace exactamente lo mismo, incrementar una variable global un numero de veces. El resultado final debe ser el doble del numero de veces que se incrementa la variable, ya que cada hilo la incrementa el mismo número de veces. Si no hubiera semáforo general, el resultado podría ser inconsistente. Pruebe a ejecutarlo con semáforos y quitando los semáforos.

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>

//Programa sin el necesario tratamiento de errores

#define NITER 100 //Cada hilo incrementará 100 veces la variable global.
int count = 0;
sem_t mutex;

void * ThreadAdd(void * a)
{
   int i, tmp;
   for(i = 0; i < NITER; i++)
   {
                    sem_wait(&mutex);
                    tmp = count;      /* copy the global count locally */
                    tmp = tmp+1;      /* increment the local copy */
                    count = tmp;      /* store the local value into the global count */
                    sem_post(&mutex);
   }
}

int main(int argc, char * argv[])
{
          //inicializamos nuestro semáforo a 1
          //y el cero indica que el semáforo solo
          //estara disponible para este main() y sus hilos().
          sem_init(&mutex, 0, 1);
          pthread_t tid1, tid2;

          //Note que podría hacer esto con un bucle for()
          if(pthread_create(&tid1, NULL, ThreadAdd, NULL))
          {
                    printf("\n ERROR creating thread 1");
                    exit(EXIT_FAILURE);
          }
          if(pthread_create(&tid2, NULL, ThreadAdd, NULL))
          {
                    printf("\n ERROR creating thread 2");
                    exit(EXIT_FAILURE);
          }

          //Note que podría hacer esto con un bucle for()
          if(pthread_join(tid1, NULL))   /* wait for the thread 1 to finish */
```

```
        {
                printf("\n ERROR joining thread");
                exit(EXIT_FAILURE);
        }
        if(pthread_join(tid2, NULL))        /* wait for the thread 2 to finish */
        {
                printf("\n ERROR joining thread");
                exit(EXIT_FAILURE);
        }

        if ( (count< (2 * NITER)) ||  (count>(2 * NITER)) )
          printf("\n BOOM! count is [%d], should be %d\n", count, 2*NITER);
        else
          printf("\n OK! count is [%d] \n", count);

        pthread_exit(NULL);
}
```

3.2.6.2 Ejemplo2

Programa de sincronización entre hilos que suma los números impares entre 0 y 20, es decir, los números 1+3+5+7+9+11+13+15+17+19 = 100. El primer hilo comprueba que el número es impar, si lo es deja que el segundo hilo lo sume, sino comprueba el siguiente número. Cópielo y ejecútelo para comprobar su salida. Haga una traza en papel de su funcionamiento teniendo en cuenta varios casos en los que el procesador da rodaja de tiempo al hilo P1 o al hilo P2.

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
//Programa sin el necesario tratamiento de errores

//Semáforo s1 para lo impares y s2 para los pares.
sem_t s1, s2;
//Sección crítica
int num=0, suma=0;
//Prototipos de funciones
void *p1 ();
void *p2 ();

int main()
{
  int error1, error2;
  pthread_t h1, h2;
  sem_init (&s2, 0, 0);  //Semáforo inicializado a 0
  sem_init (&s1, 0, 1); //Semáforo inicializado a 1
  error1 = pthread_create (&h1, NULL, p1, NULL);
  error2 = pthread_create (&h2, NULL, p2, NULL);
  //Complete el control de errores
  // ...
  pthread_join (h1, NULL);
  pthread_join (h2, NULL);
  //Complete el control de errores
  // ...
  printf ("Main() - La suma es %d\n", suma);
```

```
 return 0;
}
 void *p1()
{
        int i;
        for (i=0; i<20; i++)
        {
                sem_wait (&s1); /*Cuando haya que sumar un impar se bloqueará aquí hasta que termine P2.
                                        La primera vez que entra no se bloquea, esta inicializado a 1*/
                if (i%2) /*Si el resultado del modulo es distinto de cero (cierto en C) es que es impar. */
                {
                        num=i;
                        printf ("HILO1: Número impar %d. Desbloqueando a HILO 2\n", i);
                        sem_post (&s2); //Si p2() está esperando en s2, P2 se desbloqueará.
                }
                else  //Si el resultado del modulo es cero el numero es par.
                        sem_post (&s1); /*Incrementa s1, el siguiente numero será impar y
                                                debe entrar en sección critica y no quedarse bloqueado en el wait(). */
        }
        pthread_exit(NULL);
}

void *p2()
{
        int i;
        for (i=0; i<10; i++)
        {
                /*Se queda esperando cuando el número es par y lo tiene que desbloquear P1 cuando encuentre un
                 impar, ya que "num" tendra el valor impar a sumar*/
                sem_wait (&s2);
                printf ("HILO2: Suma actual = %d + %d\n", suma, num);
                suma = suma+num;
                /*Cuando haya sumado un impar desbloqueará a P1() incrementando s1 para
                que busque el siguiente impar. P1 se encontrará esperando a que P2 haga la
                suma antes de poder volver a entrar en la sección crítica*/
                sem_post (&s1);
        }
        pthread_exit(NULL);
}
```

3.2.6.3 Ejemplo3

Problema lectores-escritores donde hay 2 lectores y 2 escritores (prioridad a los lectores). Un buffer con un solo dato que se va incrementando. Compílelo, ejecútelo y observe su salida. Haga varias trazas del comportamiento del programa para poder entender los semáforos.

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

//Programa sin el necesario tratamiento de errores

/* datos a leer y escribir. Podría usar también un while(){} */
#define DATOS_A_ESCRIBIR 10
```

```
/* Recurso. No hay buffer solo se puede escribir en o leer de una variable. Una mejora interesante sería hacer un buffer
donde pudieran escribir y leer los escritores y lectores sin causar incoherencia en los datos*/
int dato = 0;

int n_lectores = 0; /* numero de lectores */
sem_t sem_lec; /* controlar el acceso n_lectores */
sem_t mutex; /* controlar el acceso a dato */

void * Lector(void * arg)
{
	//Complete el manejo de errores
	//....
	int i=0;
	for(i=0; i < DATOS_A_ESCRIBIR; i++ )
	{
		sem_wait(&sem_lec);
		n_lectores = n_lectores + 1;
		if (n_lectores == 1)
			sem_wait(&mutex);
		sem_post(&sem_lec);
		printf("Lector %u y leo %d\n", (unsigned int)pthread_self(), dato); /* leer dato */

		sem_wait(&sem_lec);
		n_lectores = n_lectores - 1;
		if (n_lectores == 0)
		sem_post(&mutex);
		sem_post(&sem_lec);
	}
	pthread_exit(NULL);
}


void * Escritor(void * arg)
{
	//Complete el manejo de errores
	//....
	int i=0;
	for(i=0; i < DATOS_A_ESCRIBIR; i++ )
	{
		sem_wait(&mutex); //No más de un escritor a la vez
		dato = dato + 1; /* modificar el recurso */
		printf("Escritor %u y escribo %d\n", (unsigned int)pthread_self(), dato);
		sem_post(&mutex);
	}
	pthread_exit(NULL);
}

void main(void)
{
	//Complete el manejo de errores en el main()
	//....
	pthread_t th1, th2, th3, th4;
	sem_init(&mutex, 0, 1);
	sem_init(&sem_lec, 0, 1);
	//Mejorable con un for()...
	pthread_create(&th1, NULL, Lector, NULL);
	pthread_create(&th2, NULL, Escritor, NULL);
	pthread_create(&th3, NULL, Lector, NULL);
```

```
        pthread_create(&th4, NULL, Escritor, NULL);
        //Mejorable con un for()...
        pthread_join(th1, NULL);
        pthread_join(th2, NULL);
        pthread_join(th3, NULL);
        pthread_join(th4, NULL);
        /* eliminar todos los semaforos */
        sem_destroy(&mutex);
        sem_destroy(&sem_lec);
}
```

4 Ejercicios prácticos

4.1 Ejercicio1

Resuelva el problema del productor-consumidor con un buffer circular y acotado usando semáforos generales en vez de semáforos binarios y variables de condición. Haga su programa genérico para que se pueda indicar el tamaño del buffer y la cantidad de datos a producir-consumir.

4.2 Ejercicio2

En el departamento de informática de una prestigiosa Universidad se dispone de un único servicio o WC que sirve tanto para hombres como para mujeres, pero por problemas de espacio y de intimidad en el aseo no puede haber hombres y mujeres a la vez y su capacidad es de 3 personas como máximo. Dado esta problemática, el profesorado propone a sus alumnos resolver este conjunto de restricciones mediante un software que utilice semáforos. Como alumno de 2º de Grado en Ingeniería Informática intente resolver este asunto de extrema urgencia.

4.3 Ejercicio3

Otro problema más presentado por Dijkstra es el de los filósofos comensales. Cinco filósofos viven en una casa, donde hay una mesa preparada para ellos. Básicamente, la vida de cada filósofo consiste en pensar y comer, y después de años de haber estado pensando, todos los filósofos están de acuerdo en que la única comida que contribuye a su fuerza mental son los espaguetis. Se presentan las siguientes suposiciones y/o restricciones:

- Debido a su falta de habilidad manual, cada filósofo necesita dos tenedores para comer los espaguetis, cogiendo solo un tenedor a la vez (en cualquier orden, izquierda-derecha o derecha-izquierda).
- La disposición para la comida es simple (ver figura): una mesa redonda en la que está colocado un gran cuenco para servir espaguetis, cinco platos, uno para cada filósofo, y cinco tenedores.
- Un filósofo que quiere comer utiliza,en caso de que estén libres, los dos tenedores situados a cada lado del plato, retira y come algunos espaguetis.

- Si cualquier filósofo toma un tenedor (solo un tenedor a la vez) y el otro está ocupado, se quedará esperando, con el tenedor en la mano, hasta que pueda tomar el otro tenedor, para luego empezar a comer.
- Si un filósofo piensa, no molesta a sus colegas. Cuando un filósofo ha terminado de comer suelta los tenedores y continua pensando.
- Si dos filósofos adyacentes intentan tomar el mismo tenedor a una vez, se produce una condición de carrera: ambos compiten por tomar el mismo tenedor, y uno de ellos se queda sin comer.
- Si todos los filósofos toman el tenedor que está a su derecha al mismo tiempo (o a su izquierda), entonces todos se quedarán esperando eternamente, porque alguien debe liberar el tenedor que les falta. Nadie lo hará porque todos se encuentran en la misma situación (esperando que alguno deje sus tenedores). Se produce un interbloqueo (deadlock o abrazo de la muerte). Entonces los filósofos se morirán de hambre.

El problema: diseñar un ritual (algoritmo) que permita a los filósofos comer. El algoritmo debe satisfacer la exclusión mutua (no puede haber dos filósofos que puedan utilizar el mismo tenedor a la vez) y evitar el interbloqueo e inanición (en este caso, el término inanición tiene un sentido literal, además de algorítmico). A partir del siguiente esquema de pseudocódigo y del que está disponible en Moodle, además de consultar la Web[7] para informarse en profundidad del problema, implemente una solución en C utilizando semáforos que resuelva el problema de sincronización e inanición de la cena de los filósofos.

7 http://es.wikipedia.org/wiki/Problema_de_la_cena_de_los_fil%C3%B3sofos

Pseudocódigo:

```
#define N                5              /* número de filósofos */
#define IZQ          (i - 1) % N        /* número del vecino izq.  de i */
#define DER          (i + 1) % N        /* número del vecino der. de i */
#define PENSANDO     0                  /* el filósofo está pensando */
#define HAMBRIENTO   1                  /* el filósofo está hambriento */
#define COMIENDO     2                  /* el filósofo está comiento */

int estado[N];                          /* vector para llevar el estado de los filósofos */
semáforo exmut = 1;                     /* exclusión mutua para las secciones críticas */
semáforo s[N];                          /* un semáforo por filósofo */
```

```
void filósofo (int i)                   /* i: de qué filósofo se trata (desde 0 hasta N - 1) */
{
while (1) {                             /* se ejecuta eternamente */
    pensar ( );                         /* el filósofo está pensando */
    coger_tenedores (i);                /* obtiene dos tenedores, bloqueándose si no puede */
    comer ( );
    dejar_tenedores (i);                /* deja ambos tenedores en la mesa */
}
}
```

```
void coger_tenedores (int i)        /* i: de qué filósofo se trata (desde 0 hasta N - 1) */
{
wait (&exmut);                      /* entra en la sección crítica */
estado[i] = HAMBRIENTO;             /* registra el hecho de que el filósofo i tiene hambre */
prueba(i);                          /* intenta coger dos tenedores */
signal(&exmut);                     /* sale de la sección crítica */
wait(&s[i]);                        /* se bloque si no consiguió los tenedores */
}
```

```
void dejar_tenedores (int i)        /* i: de qué filósofo se trata (desde 0 hasta N - 1) */
{
wait (&exmut);                      /* entra en la sección crítica */
estado[i] = PENSANDO;               /* registra el hecho de que el filósofo i ha dejado de comer */
prueba(IZQ);                        /* comprueba si el vecino izquierdo puede comer ahora */
prueba(DER);                        /* comprueba si el vecino derecho puede comer ahora */
signal(&exmut);                     /* sale de la sección crítica */
}
```

```
void prueba (int i)                 /* i: de qué filósofo se trata (desde 0 hasta N - 1) */
{
if (estado[i] == HAMBRIENTO && estado[IZQ] != COMIENDO
      && estado[DER] != COMIENDO)
{
    estado[i] = COMIENDO;
    signal(&s[i]);
}
}
```