UNIVERSIDAD DE ANTIOQUIA
Facultad de Ingeniería

Laboratorio 4b - Comunicación y Sincronización Hilos

Objetivos

- Identificar una condición de competencia y los mecanismos existentes para evitarla.
- Uso de mutex y semáforos.

1. Comunicación entre hilos a través de memoria compartida.

En el desarrollo de aplicaciones cooperativas usando técnicas de programación multihilo, se hace necesario utilizar herramientas con las cuales se facilite una comunicación efectiva entre los diferentes entes de procesamiento existentes en una máquina. El sistema operativo posee un diverso conjunto de opciones de comunicación que incluye las tuberías, los sockets, el paso de mensajes y la memoria compartida. La técnica de comunicación a través de memoria compartida es una de las más usadas debido a su facilidad en la implementación y la simpleza en su funcionamiento. Cuando estamos hablando estrictamente de hilos, ellos por definición comparten una serie de recursos del proceso, entre los que se encuentran los espacios de memoria Heap y Global que actúan como una región de memoria compartida que permite establecer una comunicación.

Cuando se tiene un espacio de memoria se hace necesario sincronizar el acceso a este espacio, pues se pueden presentar **condiciones de competencia (race conditions)** que alteran el buen funcionamiento de la aplicación (para profundizar en este concepto remítase al material de clase o al capítulo 2 del libro de Tanenbaum[1]). La siguiente figura muestra un caso típico de una condición de competencia:

[1] Tanenbaum, A. Modern Operating Systems. Prentice Hall. 2008.

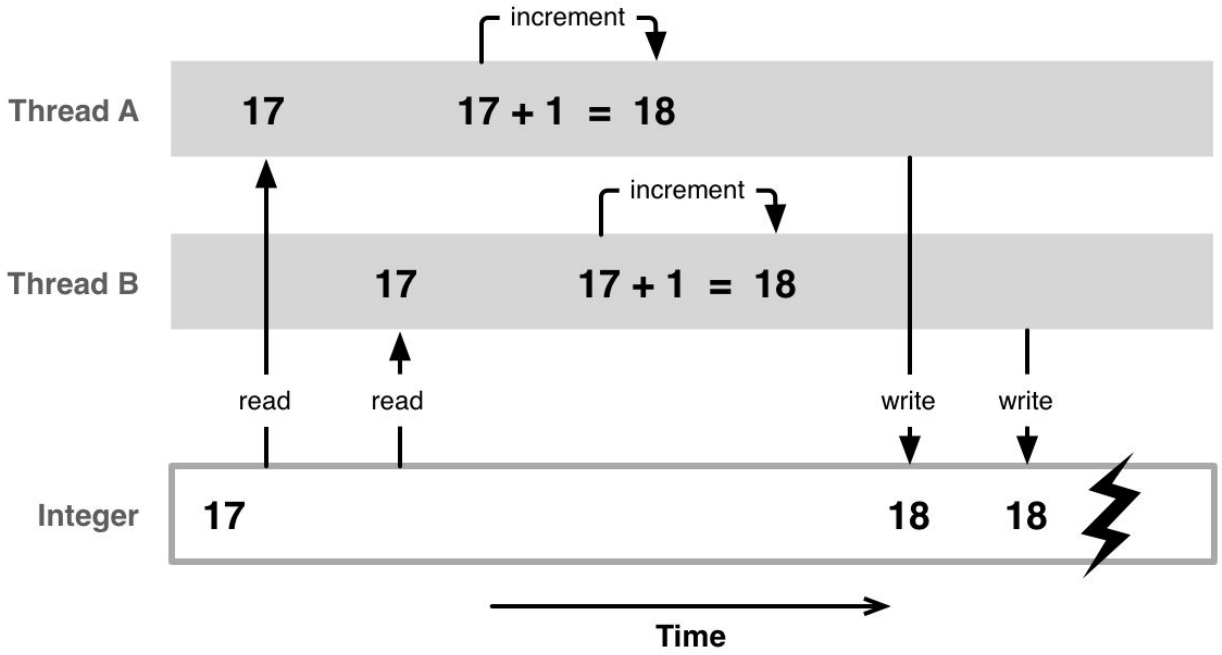

Figura 1. Ejemplo de una condición de competencia[2].

Como se podrá notar, el recurso compartido al cual accede cada hilo es la variable contador; sin embargo, como el acceso a este recurso no es sincronizado pues el acceso al contador por parte del segundo hilo no se da después de que el primer hilo actualiza (escribe) su valor en memoria si no antes. De este modo, el valor del contador definitivo no será 19 como se esperaría tras los dos incrementos (uno por cada hilo) sino 18. En otras palabras tal y como lo describen las palomas, se da una competencia de los dos hilos por esta variable:

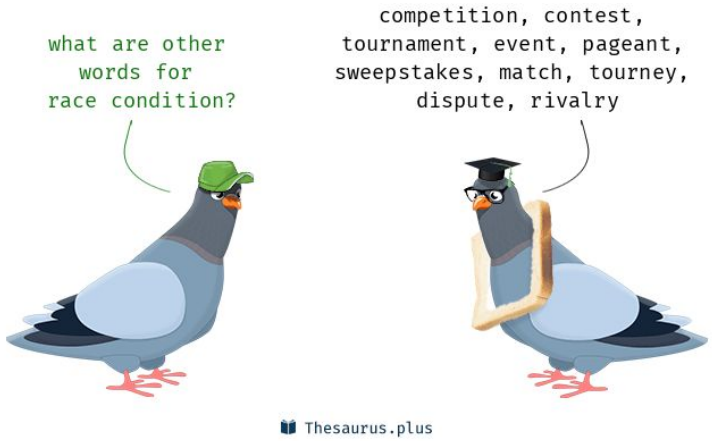

Figura 2. Explicación de una condición de competencia[3].

[2] Figura tomada del siguiente link: http://bigdata-guide.blogspot.com.co/2014/01/what-is-race-condition.html
[3] Figura tomada del siguiente link: https://thesaurus.plus/thesaurus/race_condition

2. Condición de competencia (race condition)

En el siguiente ejemplo se tienen múltiples hilos que acceden concurrentemente a una posición de memoria compartida. Se espera entonces que se configure una condición de competencia. Este es el código:

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

#define NUMTHREADS 8
#define MAXCNT 100000

/* Global variables - shared between threads */
double counter = 0;

/* Declaring functions*/
void* counting(void *);

int main(void) {
  pthread_t tid[NUMTHREADS];
  int i=0;
  for( i=0; i<NUMTHREADS; i++){
     pthread_create (&tid[i], NULL, &counting, NULL);
  }
  for( i=0; i< NUMTHREADS; i++){
     pthread_join(tid[i], NULL);
  }
  printf("\nCounter must be in: %d\n", MAXCNT*NUMTHREADS);
  printf("\nCounter value is: %.0f\n\n", counter);
  return 0;

}
/* Function Thread*/
void* counting(void * unused) {
  int i=0;
  for(i=0; i<MAXCNT; i++)
     counter++;
  return NULL;
}
```

Preguntas:

- Ejecute este código en varias oportunidades, verifique que se presente una condición de competencia. ¿Cómo se presenta en pantalla esta condición de competencia? ¿Cuál es el motivo del problema?

- ¿Cuál es la región crítica para este programa? ¿Cuáles son las posiciones de memoria compartida que generan el problema?

3. Mutex

La solución para asegurar el buen funcionamiento del ejercicio anterior es permitir que sólo uno de los hilos se encuentre en la región crítica (parte asociada al recurso compartido). La siguiente figura ilustra como se haria esto:

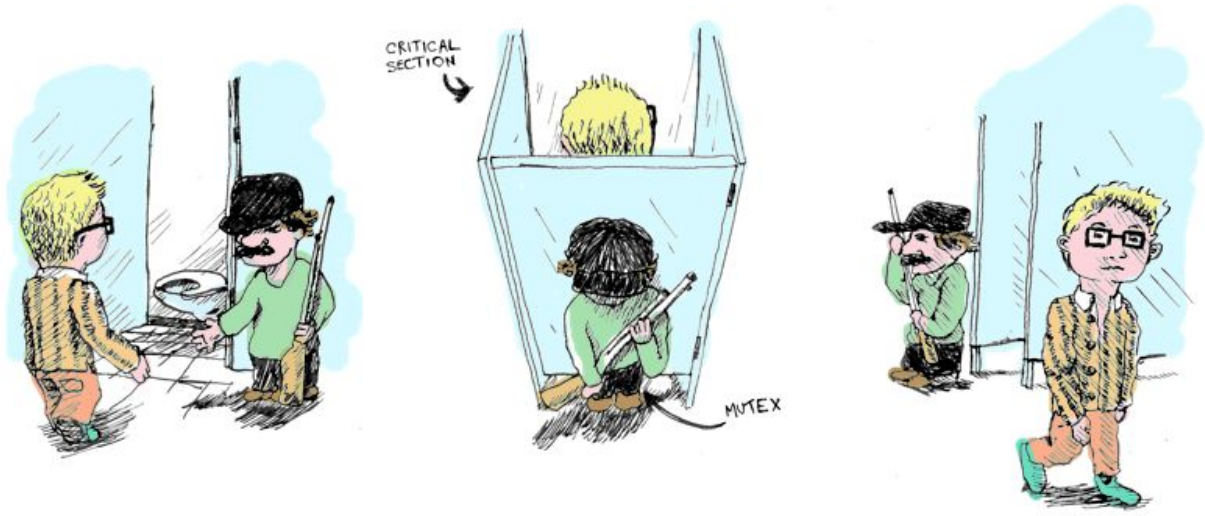

Figura 4. Analogía entre una sección crítica y un mutex[4].

En la librería pthread el mecanismo se conoce como un **mutex**. Para crear un mutex, se debe crear la variable **`pthread_mutex_t`** y luego llamar la función **`pthread_mutex_init`** para inicializarlo. La parte relacionada con los hilos no cambia, se usan de la misma manera las funciones típicas (pthread_start(), pthread_join(), etc); lo único que cambia es que en la tarea (función) ejecutada por cada hilo se deberá asegurar la región crítica empleando mutex, eso es todo. Para asegurar la sección crítica se emplea la función **`pthread_mutex_lock`**, la cual una vez invocada protegerá la **`región crítica`** evitando que otros hilos accedan a esta. Una vez que el código dentro de la región crítica se ha ejecutado, se procede a invocar la función **`pthread_mutex_unlock`** para liberar el mutex de modo que los demás hilos puedan competir nuevamente por el acceso a la región crítica. La siguiente figura muestra el efecto de asegurar y liberar el mutex (en este caso la variable **`lock`**) para controlar el acceso a la región crítica:

[4] Figura tomada del siguiente link:
http://www.rudyhuyn.com/blog/2015/12/31/synchroniser-ses-agents-avec-lapplication/mutex/

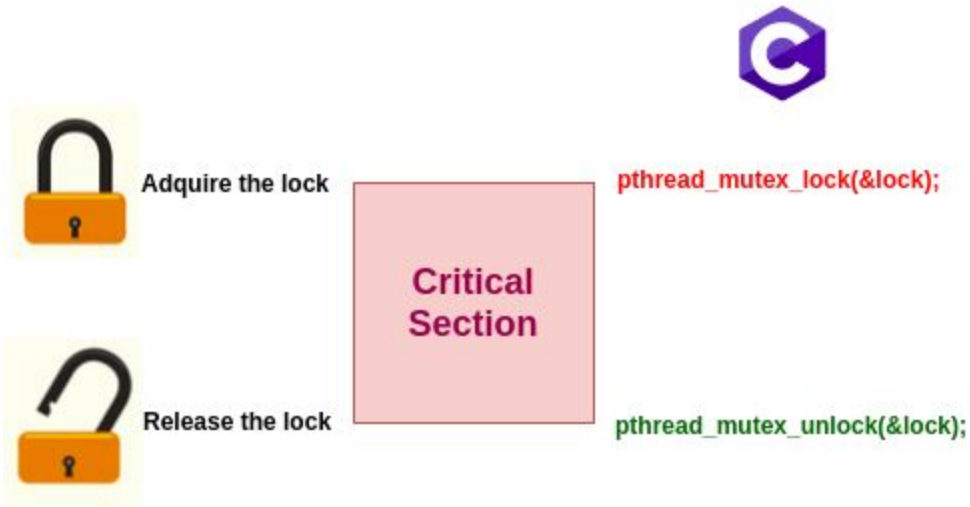

Figura 5. Control de acceso a la sección crítica mediante mutex.

A continuación se presenta un ejemplo del uso de un mutex para la sincronización del hilo anterior.

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

#define NUMTHREADS 8
#define MAXCNT 100000

/* Global variables - shared between threads */
double counter = 0;
pthread_mutex_t lock;

/* Declaring functions*/
void* counting(void *);

int main(void) {
  pthread_t tid[NUMTHREADS];
  int i=0;
  /* mutex init*/
  if (pthread_mutex_init(&lock, NULL) != 0)
  {
   printf("\n mutex init failed\n");
   return 1;
  }
  for( i=0; i<NUMTHREADS; i++){
     pthread_create (&tid[i], NULL, &counting, NULL);
  }
  for( i=0; i< NUMTHREADS; i++){
     pthread_join(tid[i], NULL);
  }
```

```c
  /* mutex destroy*/
  pthread_mutex_destroy(&lock);
  printf("\nCounter must be in: %d\n", MAXCNT*NUMTHREADS);
  printf("\nCounter value is: %.0f\n\n", counter);
  return 0;
}

/* Function Thread*/
void* counting(void * unused) {
  int i=0;
  for(i=0; i<MAXCNT; i++){
     pthread_mutex_lock(&lock);
     counter++; // Región crítica
     pthread_mutex_unlock(&lock);
  }
  return NULL;
}
```

Preguntas:

- ¿Cuál es la diferencia entre el código del ejerció anterior y el presente?
- ¿Según el tema de la clase cuál es principio de funcionamiento de este código?
- Ejecute en varias ocasiones este código. ¿Se presenta alguna condición de competencia?

4. Semáforos

Para el uso de los semáforos se requiere incluir la librería **`semaphore.h`**. Un semáforo es representado por la variable **`sem_t`**. A continuación se listan operaciones que se pueden hacer con los semáforos y la forma de inicializarlos y eliminarlos dependiendo del tipo de semáforo**.** Para más información sobre el concepto de semaforos puede consultar el enlace [Semaphores are Surprisingly Versatile](#).

4.1 Operaciones con semáforos

Función	`sem_wait`
Uso	`#include <semaphore.h>` `int sem_wait(sem_t *sem);`
Descripción	La función sem_wait disminuye el contador del semáforo, si el valor del contador es 0 antes de disminuirlo, se realiza un bloqueo hasta que el contador aumente.
Datos de retorno	• 0, si el contador fue disminuido. • -1, en caso de error.

Función	`sem_trywait`
Uso	`#include <semaphore.h>` `int sem_trywait(sem_t *sem);`
Descripción	La función sem_trywait disminuye el contador del semáforo, si el valor del contador es 0 antes de disminuirlo, se retorna un -1 y el proceso continúa con su ejecución.
Datos de retorno	• 0, si el contador fue disminuido. • -1, en caso de error.

Función	`sem_post`
Uso	`#include <semaphore.h>` `int sem_post(sem_t *sem);`
Descripción	La función sem_post incrementa el contador del semáforo.
Datos de retorno	• 0, si el contador fue incrementado. • -1, en caso de error.

Función	`sem_getvalue`
Uso	`#include <semaphore.h>` `int sem_getvalue(sem_t *sem, int *val);`
Descripción	La función sem_getvalue copia el valor contador del semáforo en la dirección de memoria de la variable val.
Datos de retorno	• 0, si valor es copiado con éxito. • -1, en caso de error.

Función	`sem_init`
Uso	`#include <semaphore.h>` `int sem_init(sem_t *sem, int pshared, unsigned int val);`
Descripción	Inicia el semáforo con el valor del contador igual a val (val >= 0), si `pshared` es igual a 0, solo los hijos creados por el mismo programa pueden acceder al semáforo, en caso contrario cualquier otro programa puede acceder al mismo.

Datos de retorno	• 0, si el semáforo fue inicializado. • -1, en caso de error.

Función	sem_destroy
Uso	`#include <semaphore.h>` `int sem_destroy(sem_t *sem);`
Descripción	Libera el semáforo, si algún otro proceso está esperando el semáforo, esta función retorna un mensaje de error.
Datos de retorno	• 0, si el semáforo fue liberado. • -1, en caso de error.

Mejorar lo que hay empleando este enlace https://www.geeksforgeeks.org/use-posix-semaphores-c/

4.2. Ejemplo del uso de semáforos con hilos

```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#define NUMTHREADS 8
#define MAXCNT 100000

/* Global variables - shared between threads */
double counter = 0;
sem_t sem;

/* Declaring functions*/
void* counting(void *);

int main(void) {
  pthread_t tid[NUMTHREADS];
  int i=0;

  /* Semaphore init*/
  sem_init(&sem,0,1);

  for( i=0; i<NUMTHREADS; i++){
     pthread_create(&tid[i], NULL, &counting, NULL);
  }

  for( i=0; i< NUMTHREADS; i++){
     pthread_join(tid[i], NULL);
  }
  /* Semaphore destroy*/
  sem_destroy(&sem);
  printf("\nCounter must be in: %d\n", MAXCNT*NUMTHREADS);
  printf("\nCounter value is: %.0f\n\n", counter);
```

```
  return 0;
}

/* Function Thread*/
void* counting(void * unused) {
  int i=0;
  for(i=0; i<MAXCNT; i++){
     sem_wait(&sem);
     counter++;
     sem_post(&sem);
  }
  return NULL;
}
```

Preguntas:

- Investigue el funcionamiento y los parámetros de las funciones **sem_init**, **sem_wait**, **sem_post** y **sem_destroy**.
- ¿Cuál es la diferencia del presente ejemplo con el anterior?

El siguiente ejemplo asegura que la tarea s1 se ejecute antes que s2.

```
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#include<semaphore.h>
#define NUMTHREADS 3

sem_t synch;

void *s1(void *arg);
void *s2(void *arg);
void *s3(void *arg);

int main(){
  int i;
  pthread_t tid[NUMTHREADS];

  sem_init(&synch,0,0);

  pthread_create(&tid[0],NULL,&s3,NULL);
  pthread_create(&tid[1],NULL,&s2,NULL);
  pthread_create(&tid[2],NULL,&s1,NULL);

  for( i=0; i< NUMTHREADS; i++){
     pthread_join(tid[i], NULL);
  }

  sem_destroy(&synch);
  printf("\nDone !!\n");

  return 0;
}

void *s1(void *arg){
  printf("\nS1 Executing...\n");
counter++; // Región crítica
```

Actividad:

- Usando semáforos como estrategia de sincronización, modifique el programa anterior con el fin de que siempre se ejecute la tarea **s2** antes que la tarea **s3**. Las tareas se deben ejecutar en el siguiente orden: s1, s2, s3.

5. Ejercicios Propuestos

1. Realice la implementación del problema del barbero dormilón en usando solo semáforos.
2. Genere un deadlock o interbloqueo en el problema del productor consumidor con los semáforos.
3. **Medida de Dispersión**: el profesor de un curso desea un programa en lenguaje C que calcule la desviación estándar (símbolo σ o s) de las notas obtenidas por sus estudiantes en el curso.

$$\sigma = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \overline{x})^2} \qquad \overline{x} = \frac{1}{N}\sum_{i=1}^{N} x_i.$$

Requisitos:

- El número de notas es variable (se debe usar memoria dinámica).
- El programa se debe ejecutar así:

```
$ ./nombre_ejecutable fichero_notas.csv
```

- El programa debe utilizar **2** hilos, uno que calcule el promedio y otro que calcule la desviación estándar.
- Plantee una estrategia usando semáforos y/o mutex para asegurar primero se calcule el promedio antes de iniciar a calcular la desviación estándar. La creación de los hilos se debe realizar desde el main, todos deben de crearse sin ninguna restricción.

Referencias

[1] Tanenbaum, A. Modern Operating Systems. Prentice Hall. 2008.

[2] Marshall, D. Programming in C: UNIX system call and subroutines using C. 1999. Available Online: http://www.cs.cf.ac.uk/Dave/C/node27.html. Last visited: 22/09/11.

[3] Sharing Memory Between Processes
http://menehune.opt.wfu.edu/Kokua/More_SGI/007-2478-008/sgi_html/ch03.html. Last visited: 22/09/11.

[4] Semáforos POSIX. Available Online:
http://isa.umh.es/asignaturas/sitr/TraspSITR_POSIX3_Semaforos.pdf

[5] https://www.cs.helsinki.fi/u/kerola/rio/pdf/lu06_v.pdf

[6] http://www.csl.mtu.edu/cs4411.choi/www/Resource/Semaphore.pdf

[7]
https://www.cs.princeton.edu/courses/archive/fall11/cos318/lectures/L8_SemaphoreMonitor_v2.pdf

[8] http://greenteapress.com/wp/semaphores/

[9] http://www.cs.cmu.edu/~tcortina/15110f11/Unit10PtC.pdf

http://www.csc.villanova.edu/~mdamian/threads/posixsem.html

https://www.geeksforgeeks.org/use-posix-semaphores-c/

http://www.cse.psu.edu/~deh25/cmpsc473/programs/semaphore_sample.c

https://www.geeksforgeeks.org/semaphores-operating-system/

https://www.geeksforgeeks.org/mutex-vs-semaphore/
https://www.geeksforgeeks.org/dining-philosophers-solution-using-monitors/

https://www.geeksforgeeks.org/operating-system-process-management-deadlock-introduction/
https://www.geeksforgeeks.org/g-fact-70/

https://www.geeksforgeeks.org/process-synchronization-set-1/

https://www.geeksforgeeks.org/mutex-vs-semaphore/

http://iit.qau.edu.pk/books/OS_8th_Edition.pdf

https://github.com/hailinzeng/Programming-POSIX-Threads

https://github.com/snikulov/prog_posix_threads
https://github.com/yesoun/Message-Passing-Threads
https://github.com/MariaBamundo/Learning-C