**Universidad de Antioquia**

**Facultad de Ingeniería**

**Laboratorio de Sistemas Operativos**

**Comunicación y Sincronización Hilos/Procesos**

**Objetivos**

* Identificar una codición de carrera y los mecanismos existentes para evitarla.
* Uso de mutex y semáforos
* Explorar el uso de la técnica de memoria compartida para comunicación entre procesos e hilos.

**Contenido**

[1 Comunicación entre Procesos/Hilos a través de memoria compartida.](#_gjdgxs)

[1.1 Comunicación entre Hilos](#_30j0zll)

[1.1.1 Condición de carrera](#_1fob9te)

1.1.2 Mutex

1.1.3 Semáforos

[1.2 Creación y uso de memoria compartida en procesos](#_3znysh7)

2 Ejercicios Propuestos

[3 Referencias](#_3rdcrjn)

1. **Comunicación entre Procesos/Hilos a través de memoria compartida.**

En el desarrollo de aplicaciones cooperativas usando técnicas de multiprogramación y multihilo, se hace necesario poseer herramientas con las cuales se facilite una comunicación efectiva entre los diferentes entes de procesamiento existentes en una máquina (hilos o procesos). 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. Ésta última técnica de comunicación a través de memoria compartida es una de las más usadas actualmente 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 la comunicación ente hilos.

Para los proceso la situación cambia, ya que éstos no poseen a priori, ningún espacio de memoria compartido, por lo que se hace necesaria la intervención del sistema operativo para asignar un espacio de memoria que sea visible por los procesos que se desean comunicar.

En ambos casos tras la definición de un espacio de memoria se hace necesario sincronizar el acceso a la misma, pues se puede presentar una condición de carrera que altera 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]](#footnote-0)]).

* 1. **Comunicación entre Hilos**

En la clase de sistemas operativos se abordó el uso de semáforos como mecanismo de sincronización entre procesos e hilos. Debido a que el concepto de semáforo se aplica muy similar en procesos e hilos, se ha decidido mostrar el funcionamiento solo para estos últimos y se deja al estudiante la tarea de verificar el funcionamiento en procesos. A continuación realizaremos algunos ejemplos acerca del uso de esta técnica tan popular.

* + 1. **Condición de carrera**

En el siguiente ejemplo se tiene dos hilos que acceden a posiciones de memoria compartida. Se espera entonces que se configure una condición de carrera. Este es el código:

|  |
| --- |
| #include <stdio.h>  #include <pthread.h>  #define NUMTHREADS 200  #define MAXCNT 1000  /\* 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;  } |

* Ejecute este código en varias oportunidades, verifique que se presente una condición de carrera. ¿Cómo se presenta en pantalla esta condición de carrera? ¿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?
  + 1. **Mutex**

La solución para asegurar el buen funcionamiento del ejercicio anterior es permitir que solo uno de los hilos se encuentre en la región crítica al mismo tiempo. 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. A continuación se presenta un ejemplo del uso de un mutex para la sincronización del hilo anterior.

|  |
| --- |
| #include <stdio.h>  #include <pthread.h>  #define NUMTHREADS 200  #define MAXCNT 1000  /\* 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);  }  /\* 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;    pthread\_mutex\_lock(&lock);    for(i=0; i<MAXCNT; i++)  counter++;    pthread\_mutex\_unlock(&lock);    return NULL;  } |

* ¿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 carrera?
  + 1. **Semáforos**

Para el uso de los semáforos se requiere incluir la librería **semaphore.h**. Un semáforo es representado por la variable **sem\_t**, se debe inicializar usando la función *sem\_init* y en caso de ya no requerirlo más en el programa se debe destruir con la función *sem\_destroy*. Para hacer un *down* sobre el semáforo se usa la función sem\_wait y para hacer un *up* se usa la función sem\_post. A continuación un ejemplo sobre el uso de semáforo.

|  |
| --- |
| #include <stdio.h>  #include <pthread.h>  #include <semaphore.h>  #define NUMTHREADS 200  #define MAXCNT 1000  /\* 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;    sem\_wait(&sem);    for(i=0; i<MAXCNT; i++)  counter++;    sem\_post(&sem);    return NULL;  } |

* 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");  sem\_post(&synch);  return 0;  }  void \*s2(void \*arg){    //printf("\nS2 Waiting...\n");  sem\_wait(&synch);  printf("\nS2 Executing...\n");  return 0;  }  void \*s3(void \*arg){    printf("\nS3 Executing...\n");  return 0;  } |

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

A continuación se presenta un ejemplo muy simple del uso de memoria compartida entre procesos. Se tienen dos procesos denominados el servidor y el cliente, el servidor crea una región de memoria compartida y pone una información allí, luego el cliente se adhiere a esa región lee los datos y señaliza de manera simple al servidor para que finalice [[[2]](#footnote-1)].

Este es el código fuente del servidor:

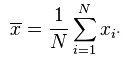
|  |
| --- |
| #include <sys/types.h>  #include <sys/ipc.h>  #include <sys/shm.h>  #include <stdio.h>  #define SHMSZ 27  int main(){  char c;  int shmid;  key\_t key;  char \*shm, \*s;  /\*Nombre del segmento de memoria compartida = "1234".\*/  key = 1234;  /\* Se crea el segmento de memoria\*/  if ((shmid = shmget(key, SHMSZ, IPC\_CREAT | 0666)) < 0) {  perror("shmget");  exit(1);  }  /\* El programa se adhiere (attach) al segmento ya creado \*/  if ((shm = shmat(shmid, NULL, 0)) == (char \*) -1) {  perror("shmat");  exit(1);  }  /\* Se ponen algunos datos en el segmento para que el proceso  cliente los lea \*/  s = shm;  for (c = 'a'; c <= 'z'; c++)  \*s++ = c;  \*s = NULL;  /\* Por último, se espera a que el proceso cliente cambie el primer caracter de la memoria compartida a '\*' indicando que ya leyó la información \*/  while (\*shm != '\*')  sleep(1);    return(0);  } |

Este es el código fuente del cliente:

|  |
| --- |
| #include <sys/types.h>  #include <sys/ipc.h>  #include <sys/shm.h>  #include <stdio.h>  #define SHMSZ 27  main() {  int shmid;  key\_t key;  char \*shm, \*s;  /\*Se requiere el segmento llamado "1234" creado por el servidor \*/  key = 1234;  /\* Ubica el segmento \*/  if ((shmid = shmget(key, SHMSZ, 0666)) < 0) {  perror("shmget");  exit(1);  }  /\* Se adhiere al segmento para poder hacer uso de él \*/  if ((shm = shmat(shmid, NULL, 0)) == (char \*) -1) {  perror("shmat");  exit(1);  }  /\* Lee lo que el servidor puso en la memoria \*/  for (s = shm; \*s != NULL; s++)  putchar(\*s);  putchar('\n');  /\* Finalmente, cambia el calor del primer carácter indicando que  ha leído el segmento \*/  \*shm = '\*';  exit(0);  } |

* Consulte el uso de las funciones **shmget**, **shmat**, **shmdt** y **shmctl**. ¿Para qué sirven estas funciones? ¿Qué argumentos reciben y para qué sirven estos argumentos? ¿Cuál es la forma tradicional de usar estas funciones?
* Analice el código anterior y verifique el uso y los parámetros pasados a las funciones mencionadas anteriormente.
* Ejecute ambos programas, primero el servidor en una terminal y luego el cliente en otra terminal. ¿cómo es el funcionamiento del programa? ¿Qué sucede si ejecuta los programas en un orden diferente (cuál es la salida en pantalla)?
* El flujo normal para el uso de memoria compartida entre procesos es solicitar la memoria compartida (shmget) , adherirse a ella(shmat), usarla, luego des-adherirse (shmdt) y por último liberarla (shmctl). ¿Todos los procesos involucrados en la comunicación debe realizar este proceso? ¿Qué sucede si un proceso no realiza se des-adhiere antes de salir o si el proceso principal no libera la memoria antes de salir?

1. **Ejercicios Propuestos**
2. Realice la implementación del problema del barbero dormilón en usando solo semáforos.
3. Realice la implementación del problema del productor consumidor usando solo semáforos.
4. Genere un deadlock o interbloqueo en el problema del productor consumidor con los semáforos.
5. **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.

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.

1. [] Tanenbaum, A. Modern Operating Systems. Prentice Hall. 2008. [↑](#footnote-ref-0)
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: 23/05/16. [↑](#footnote-ref-1)