Universidad de Antioquia
Facultad de Ingeniería
Sistemas Operativos y Laboratorio

Laboratorio 4 - Comunicación y Sincronización Hilos/Procesos

Objetivos

- Identificar una condición de competencia 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.

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. La 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 entre hilos.

Para los procesos 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 competencia 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]).

1.1 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 de manera 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.

1.1.1 Condición de competencia

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

```
#include <stdio.h>
#include <pthread.h>
#define NUMTHREADS 200
#define MAXCNT 10000

/* Global variables - shared between threads */
double counter = 0;
/* Declaring functions*/
void* counting(void *);

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

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

```
  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 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?

1.1.2 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.

```c
#include <stdio.h>
#include <pthread.h>
#define NUMTHREADS 200
#define MAXCNT 10000

/* 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;

  for(i=0; i<MAXCNT; i++){
     pthread_mutex_lock(&lock);
     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 competencia?

1.1.3 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.

1.1.3.1 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.

1.1.3.2 Operaciones para semáforos sin nombre

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.

1.1.3.3 Operaciones para semáforos con nombre

Función	`sem_open`
Uso	`#include <semaphore.h>` `sem_t *sem_open(const char *name, int oflag, mode_t mode, unsigned int value);`
Descripción	Retorna el apuntador al semáforo con nombre `name`, si `oflag` es 0, solo se intenta acceder a un semáforo previamente creado, si `oflag` es `O_CREAT`, se crea el semáforo en caso de no existir y se accede al semáforo, si oflag es `O_CREAT | O_EXECL`, crea el semáforo en caso de no existir o retorna un -1 si existe. el `mode` y el `value`, especifican los permisos con los que se crea el semáforo y el valor inicial

	respectivamente.
Datos de retorno	• Dirección, si se pudo acceder o inicializar el semáforo. • -1, en caso de error.

Función	`sem_close`
Uso	`#include <semaphore.h>` `int sem_close(sem_t *sem);`
Descripción	Libera los recursos que el sistema le asigna al proceso del semáforo. No siempre elimina el semáforo, solo lo hace inaccesible desde el proceso.
Datos de retorno	• 0, si se liberó el semáforo. • -1, en caso de error.

Función	`sem_unlink`
Uso	`#include <semaphore.h>` `int sem_unlink(sem_t *sem);`
Descripción	Elimina el semáforo con nombre, si otros procesos están usando el semáforo, se pospone la eliminación.
Datos de retorno	• 0, si se eliminó el semáforo. • -1, en caso de error.

1.1.3.4 Ejemplo sobre el uso de semáforo con hilos

```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#define NUMTHREADS 200
#define MAXCNT 10000

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

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

```c
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;
}
```

- 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 **s2** antes que la tarea **s3**. Las tareas se deben ejecutar en el siguiente orden: s1, s2, s3.

1.2 Comunicación entre procesos

Como es bien sabido los procesos no comparten memoria. Si se desea diseñar una aplicación colaborativa entre procesos es necesario usar un mecanismo que permita la comunicación entre estos, ya sea memoria compartida o paso de mensajes. A continuación se presenta un ejemplo muy simple del uso de memoria compartida entre procesos.

1.2.1 Memoria compartida en 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].

Este es el código fuente del servidor:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define SHMSZ 27

int main(){
```

[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.

```
  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
```

```c
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 el proceso de des-adherirse antes de salir o si el proceso principal no libera la memoria antes de salir?

1.3 Creación y uso de semáforos con procesos

Una vez visto el uso de semáforos en hilos, procedemos a ver el uso de ellos en procesos, primero que todo vale la pena mencionar que hay dos tipos diferentes de semáforos: semáforos con nombre y sin nombre, cada uno tiene un uso específico como se verá a continuación.

1.3.1 Semáforos con nombre

Estos semáforos son útiles para comunicar procesos que no tienen una relación directa, es decir, cuando los procesos son ejecutados de forma independiente, y para mantener la comunicación entre ellos es necesario declarar un nombre en el semáforo para que este pueda ser accedido por cualquier proceso. A continuación se presenta el código del problema del Productor/Consumidor.

Este es el código fuente del Productor

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <semaphore.h>
#include <sched.h>
#include <sys/stat.h>
#include <fcntl.h>

#define BUFFER_SIZE 10    // tamaño del buffer
#define CICLOS 10 // numero de ciclos de ejecución

int buffer[BUFFER_SIZE];
sem_t *sem_cont, *sem_free; // declaramos un puntero para el
identificador de los semaforos

int main (int argc, char *argv[]) {
 int i;
 pid_t hijo;
 int val;
 int entrada, salida;    // índices a las zonas de insercion y
```

```
extraccion del buffer

 entrada = salida = 0; // inicializacion

 printf("Creando semaforos .....\n");

 /* comprueba si ya existe el semaforo del contador de
productos sino lo crea inicializado(0)*/
 if((sem_cont = sem_open("/sem_cont", O_CREAT, 0644, 0)) ==
(sem_t *)-1) {
   perror("Error creando semaforo 1");
 }

 /* comprueba si ya existe el semaforo del espacio libre y sino
lo crea inicializado (BUFFER_SIZE)*/
 if((sem_free = sem_open("/sem_free", O_CREAT, 0644,
BUFFER_SIZE)) == (sem_t *)-1) {
   perror("Error creando semaforo 2");
 }

 printf("Creando proceso hijo .....\n");

 hijo = fork() ;

 if (hijo == -1) {
   printf("error creando proceso hijo\n");

   sem_unlink("/sem_cont");
   sem_unlink("/sem_free");

   exit(0);
 } else if (hijo == 0) {
   /*estamos en el padre-> productor */
   printf("Soy el padre (productor) con PID:%d\n", getpid());
   sleep(1);

   for (i = 0; i <= CICLOS; i++) {
     sem_wait(sem_free); /* espera que haya espacio en el
buffer y decrementa */
     buffer[entrada] = i; // produce un elemento
     entrada = (entrada + 1) % BUFFER_SIZE; // buffer circular
     sem_post(sem_cont); /* incrementa el contador del semáforo
*/
     sem_getvalue(sem_cont, &val); // valor del contador del
```

```
semáforo
     printf("Soy el Productor: entrada=%d, Estado %d
productos\n", entrada, val);
     sleep(1);
   }

   /* libero semáforos */
   sem_close(sem_cont);
   sem_close(sem_free);

   exit(0);
 }

 printf("Soy el padre espero que termine el hijo .....\n");

 wait(0); /* Esperar que acabe el hijo */

 printf("Soy el padre destruyo los semaforos y
termino.....\n");

 sem_unlink("/sem_cont");
 sem_unlink("/sem_free");

 exit(0);
}
```

Este es el código fuente del Consumidor

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <semaphore.h>
#include <sched.h>
#include <sys/stat.h>
#include <fcntl.h>

#define BUFFER_SIZE 10    // tamaño del buffer
#define CICLOS 10 // numero de ciclos de ejecución

int buffer[BUFFER_SIZE];
sem_t *sem_cont, *sem_free; // declaramos un puntero para el
identificador de los semaforos

int main (int argc, char *argv[]) {
 int i;
```

```
 pid_t hijo;
 int val;
 int entrada, salida;    // índices a las zonas de insercion y
extraccion del buffer

 entrada = salida = 0; // inicializacion

 printf("Creando semaforos .....\n");

 /* comprueba si ya existe el semaforo del contador de
productos sino lo crea inicializado(0)*/
 if((sem_cont = sem_open("/sem_cont", O_CREAT, 0644, 0)) ==
(sem_t *)-1) {
   perror("Error creando semaforo 1");
 }

 /* comprueba si ya existe el semaforo del espacio libre y sino
lo crea inicializado (BUFFER_SIZE)*/
 if((sem_free = sem_open("/sem_free", O_CREAT, 0644,
BUFFER_SIZE)) == (sem_t *)-1) {
   perror("Error creando semaforo 2");
 }

 printf("Creando proceso hijo .....\n");

 hijo = fork();

 if (hijo == -1) {
   printf("error creando proceso hijo\n");

   sem_close(sem_cont);
   sem_close(sem_free);

   exit(0);
 } else if (hijo == 0) {
   printf("Soy el hijo (consumidor) con PID:%d\n", getpid());

   sleep(1);

   for (i = 0; i <= CICLOS; i++) {
     sem_wait(sem_cont); /* espera que haya datos en el buffer
(contador>0) y decrementa */
     buffer[salida] = 0; // consume un elemento
     salida= (salida+1) % BUFFER_SIZE; // buffer circular
```

```
      sem_post(sem_free); /* incrementa el contador de espacio
*/
      sem_getvalue(sem_cont, &val); // valor del contador del
semáforo
      printf("Soy el Consumidor: salida=%d, Estado %d
productos\n", salida, val);
      sleep(2);
    }

    exit(0);
  }

  wait(0);

  /* libero semáforos */
  sem_close(sem_cont);
  sem_close(sem_free);

  printf("Soy el hijo y termino.....\n");

  exit(0);
}
```

1.3.2 Semáforos sin nombre

Estos semáforos son usados cuando se necesita comunicarse entre procesos creados por el mismo padre, por lo tanto no se requiere el uso de un nombre para ser accedido a ellos.

Ejemplo

```
#include <stdio.h>
#include <stdlib.h>
#include <semaphore.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <sys/mman.h>

#define NUMHIJOS 3
```

```c
sem_t *sem1;

void s1();
void s2();
void s3();

int main(){
  int i;
  pid_t pid[NUMHIJOS];
  int val= 0;
  int    pid_hijo =0;

  sem1 = mmap(0, sizeof(sem_t), PROT_READ|PROT_WRITE,
MAP_SHARED|MAP_ANONYMOUS, -1, 0);

  sem_init(sem1,1,0); //direccion de la variable, 0->hilos y
diferente de 0->procesos, valor de inicializacion del semaforo.

  for (i = 0; i < NUMHIJOS; i++) {
     pid[i] = fork();//Crea los procesos y los guarda en un
arreglo

     if (pid[i] == -1) {
        /* Verificacion de Error */
        printf("No fue posible crear un hijo\n");
        return -1;
     }

     if (pid[i] == 0) {
        //printf("\nSoy un proceso hijo con PID: %d!\n",
getpid());
          switch (i){
           case 0:
              s3();
           break;

           case 1:
              s2();
           break;

           case 2:
              s1();
           break;
          }
```

```
        printf("El proceso hijo con PID %d ha terminado!\n",
getpid());
        exit(0);
     }
  }

  for (i = 0; i < NUMHIJOS; i++) {
     wait(&val);
  }

  sem_destroy(sem1);

  printf("\nEl padre termino !!\n"); //mensaje al final
  return 0;
}

void s1(){
  printf("\nS1 Executing...\n");
  sem_post(sem1);
}

void s2(){
  sem_wait(sem1);//ejecuta primero el S1
  printf("\nS2 Executing...\n");
}

void s3(){
  printf("\nS3 Executing...\n");
}
```

2. 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