# Tutorial Comunicación y Sincronización Hilos/Procesos

Por: Henry Arcila – Danny Múnera

**Objetivos**

- Explorar el uso de la técnica de memoria compartida para comunicación entre procesos e hilos.
- Identificar una condición de competencia y la forma como esta se puede evitar.

## Contenido



# 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]).

## 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 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.

### 1.1.1 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>
/* Variables globales compartidas entre hilos */
char * ptr;
int c;

static void imprimirAlMismoTiempo(char *);

void* eluno (void * h) {
	imprimirAlMismoTiempo("Soy eluno 11111\n");
};

void* elotro (void * h) {
	imprimirAlMismoTiempo("Soy elotro 22222\n");
};

int main(void) {
	pthread_t thread1_id;
	pthread_t thread2_id;
	pthread_create (&thread1_id, NULL, &eluno, NULL);
	pthread_create (&thread2_id, NULL, &elotro, NULL);
	pthread_join (thread1_id, NULL);
	pthread_join (thread2_id, NULL);
	exit(0);
}

static void imprimirAlMismoTiempo(char * str) {
	/* Se asegura que los caracteres sean enviados a la salida estandar
		lo más rápido que sea posible - make stdout unbuffered. */
	setbuf(stdout,NULL);

	for(ptr = str; c = *ptr++; )
		putc(c, stdout);
}
```

- 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?

### 1.1.2 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.

```
#include <stdio.h>
#include <pthread.h>
/* Variables globales compartidas entre hilos */
char * ptr;
int c;
pthread_mutex_t my_mutex = PTHREAD_MUTEX_INITIALIZER;

static void imprimirAlMismoTiempo(char *);

void* eluno (void * h) {
	pthread_mutex_lock (&my_mutex);
	imprimirAlMismoTiempo("Soy eluno 11111\n");
```

```
	pthread_mutex_unlock (&my_mutex);
};

void* elotro (void * h) {
	pthread_mutex_lock (&my_mutex);
	imprimirAlMismoTiempo("Soy elotro 22222\n");
	pthread_mutex_unlock (&my_mutex);
};

int main(void) {
	pthread_t thread1_id;
	pthread_t thread2_id;
	pthread_create (&thread1_id, NULL, &eluno, NULL);
	pthread_create (&thread2_id, NULL, &elotro, NULL);
	pthread_join (thread1_id, NULL);
	pthread_join (thread2_id, NULL);
	exit(0);
}

static void imprimirAlMismoTiempo(char * str) {
	/* Se asegura que los caracteres sean enviados a la salida estandar
		lo más rápido que sea posible - make stdout unbuffered. */
	setbuf(stdout,NULL);

	for(ptr = str; c = *ptr++; )
		putc(c, stdout);
}
```

- ¿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?

### 1.1.3 Semáforos

Para el uso de los semáforos se requiere incluir la librería semaphore.h. Un semáforo es representado por la variable sem_t, 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>
/* Variables globales compartidas entre hilos */
char * ptr;
int c;
sem_t sem1;

static void imprimirAlMismoTiempo(char *);

void* eluno (void * h) {
	sem_wait (&sem1);
	imprimirAlMismoTiempo("Soy eluno 11111\n");
	sem_post (&sem1);
};

void* elotro (void * h) {
	sem_wait (&sem1);
	imprimirAlMismoTiempo("Soy elotro 22222\n");
	sem_post (&sem1);
};
```

```
int main(void) {
	/* Inicialización del semáforo*/
	sem_init(&sem1,0,1);
	pthread_t thread1_id;
	pthread_t thread2_id;
	pthread_create (&thread1_id, NULL, &eluno, NULL);
	pthread_create (&thread2_id, NULL, &elotro, NULL);
	pthread_join (thread1_id, NULL);
	pthread_join (thread2_id, NULL);

	sem_destroy(&sem1);
	exit(0);
}

static void imprimirAlMismoTiempo(char * str) {
	/* Se asegura que los caracteres sean enviados a la salida estandar
		lo más rápido que sea posible - make stdout unbuffered. */
	setbuf(stdout,NULL);

	for(ptr = str; c = *ptr++; )
		putc(c, stdout);
}
```

- 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?

## 1.2 Creación y uso de memoria compartida en procesos

(Ejercicio adaptado de [2]) 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.

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

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);
   exit(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 caracter 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 proceso 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?

# 2 Ejercicios Propuestos

a) Realice la implementación del problema del barbero dormilón usando solo semáforos.
b) Realice la implementación del problema del productor consumidor usando solo semáforos.
c) Genere un deadlock o interbloqueo en el problema del productor consumidor con los semáforos.

# 3 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.