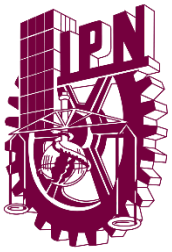


Instituto Politécnico Nacional
Escuela Superior de Cómputo
"ESCOM"

Unidad de Aprendizaje:
Cómputo Paralelo

Práctica No. 3
Programas de cómputo paralelo usando memoria compartida

Integrantes:
Sánchez De Los Ríos Flavio Josué - 6BV1
Tinoco Videgaray Sergio Ernesto - 6BV1
Ibarra González Emilio Francisco- 6BV1

Maestro:
Jiménez Benítez José Alfredo

Fecha de Entrega: 30/04/23

Índice

Resumen

En la presente práctica se desarrolló una pequeña investigación sobre la programación y el cómputo paralelo con el objetivo de elaborar un programa capaz de descomponer una tarea en múltiples procesos en el lenguaje C en el sistema operativo Ubuntu. En específico el programa toma una matriz de tres filas y realiza la suma de cada elemento de estas en tres procesos diferentes, para posteriormente guardar el resultado de estos en un archivo. Para esto se hace uso de la memoria compartida con el fin de agilizar el acceso a la memoria

Introducción

La memoria compartida es un recurso crucial en la programación de sistemas y aplicaciones que necesitan compartir datos entre procesos. En un sistema operativo, cada proceso tiene su espacio de memoria, y la comunicación entre procesos puede ser difícil y costosa debido a la necesidad de copiar datos entre los diferentes espacios de memoria.La memoria compartida es una región de memoria que puede ser accedida y modificada por múltiples procesos. Los procesos pueden compartir datos almacenados en la memoria compartida sin tener que copiarlos a sus propios espacios de memoria. Esto significa que la comunicación entre procesos se puede lograr de manera más eficiente y con menor uso de recursos.

En C, la biblioteca estándar ofrece una interfaz para la creación y administración de memoria compartida a través de la función shmget(). Esta función crea un nuevo segmento de memoria compartida, o recupera uno existente, y devuelve un identificador de memoria compartida que se puede utilizar para acceder al segmento de memoria compartida.

Una vez que se ha creado un segmento de memoria compartida, los procesos pueden acceder a él utilizando la función shmat(), que adjunta el segmento de memoria compartida al espacio de memoria del proceso. Después de adjuntar el segmento de memoria compartida, el proceso puede leer y escribir datos en la memoria compartida.

La memoria compartida es especialmente útil en situaciones donde se requiere que varios procesos trabajen juntos en la misma tarea o compartan datos en tiempo real. Por ejemplo, en un sistema de control de procesos industriales, varios procesos pueden acceder a un segmento de memoria compartida que contiene datos en tiempo real sobre el estado del proceso, como la temperatura y la presión.

La memoria compartida es un recurso crucial en la programación de sistemas y aplicaciones que necesitan compartir datos entre procesos. Permite la comunicación eficiente entre procesos y puede ser utilizado en una variedad de situaciones donde se requiere el intercambio de datos en tiempo real o la colaboración entre múltiples procesos.

Marco Teórico

En informática, un proceso es la instancia de un programa informático que está siendo ejecutado por uno o varios subprocesos. Hay muchos modelos de procesos diferentes, algunos de los cuales son livianos, pero casi todos los procesos (incluso las máquinas virtuales completas) están arraigados en un proceso del sistema operativo (SO) que comprende el código del programa, los recursos del sistema asignados, los permisos de acceso físico y lógico, y estructuras de datos para iniciar, controlar y coordinar la actividad de ejecución. Según el sistema operativo, un proceso puede estar compuesto por varios subprocesos de ejecución que ejecutan instrucciones al mismo tiempo.[1]

Mientras que un programa de computadora es una colección pasiva de instrucciones típicamente almacenadas en un archivo en el disco, un proceso es la ejecución de esas instrucciones después de ser cargadas desde el disco a la memoria[2]. Varios procesos pueden estar asociados con el mismo programa; por ejemplo, abrir varias instancias del mismo programa a menudo da como resultado la ejecución de más de un proceso.

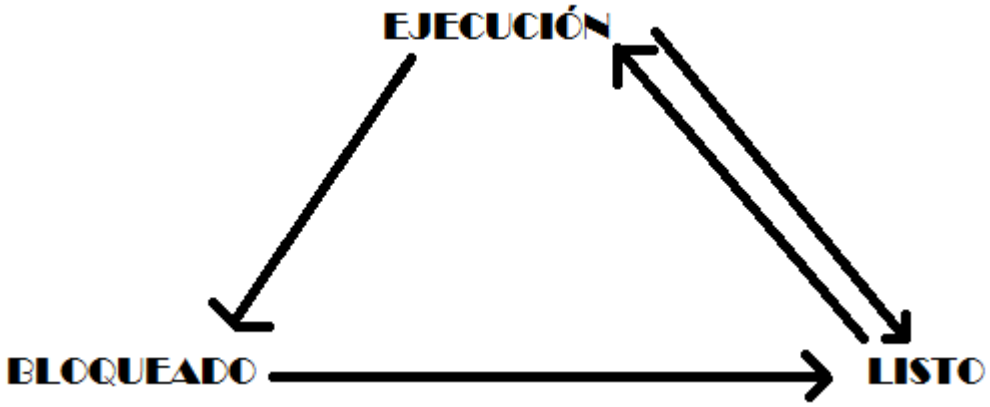
La multitarea es un método que permite que varios procesos compartan procesadores (CPU) y otros recursos del sistema. Cada CPU (núcleo) ejecuta un solo proceso a la vez. Sin embargo, la multitarea permite que cada procesador cambie entre las tareas que se están ejecutando sin tener que esperar a que finalice cada tarea. Dependiendo de la implementación del sistema operativo, los cambios se pueden realizar cuando las tareas se inician y esperan la finalización de las operaciones de entrada/salida, cuando una tarea cede voluntariamente la CPU, en las interrupciones de hardware y cuando el programador del sistema operativo decide que un proceso ha expirado.

Estados del proceso [2]

Un proceso puede estar en la CPU, es decir, estará en ejecución.

Puede ser que esté cargado en la RAM, preparado para que el SO le de paso a la CPU. Estará listo.

O puede ser que esté esperando a utilizar los recursos que otro proceso ha cogido y hasta que no finalice el sistema operativo no le va a dejar continuar. Entonces estará bloqueado.



La contienda es como se llama a la pugna que hacen los procesos listos para conquistar la CPU.

Hay distintos modos de realizar la asignación de la contienda. Entre otros:

è Round-Robin. Método circular

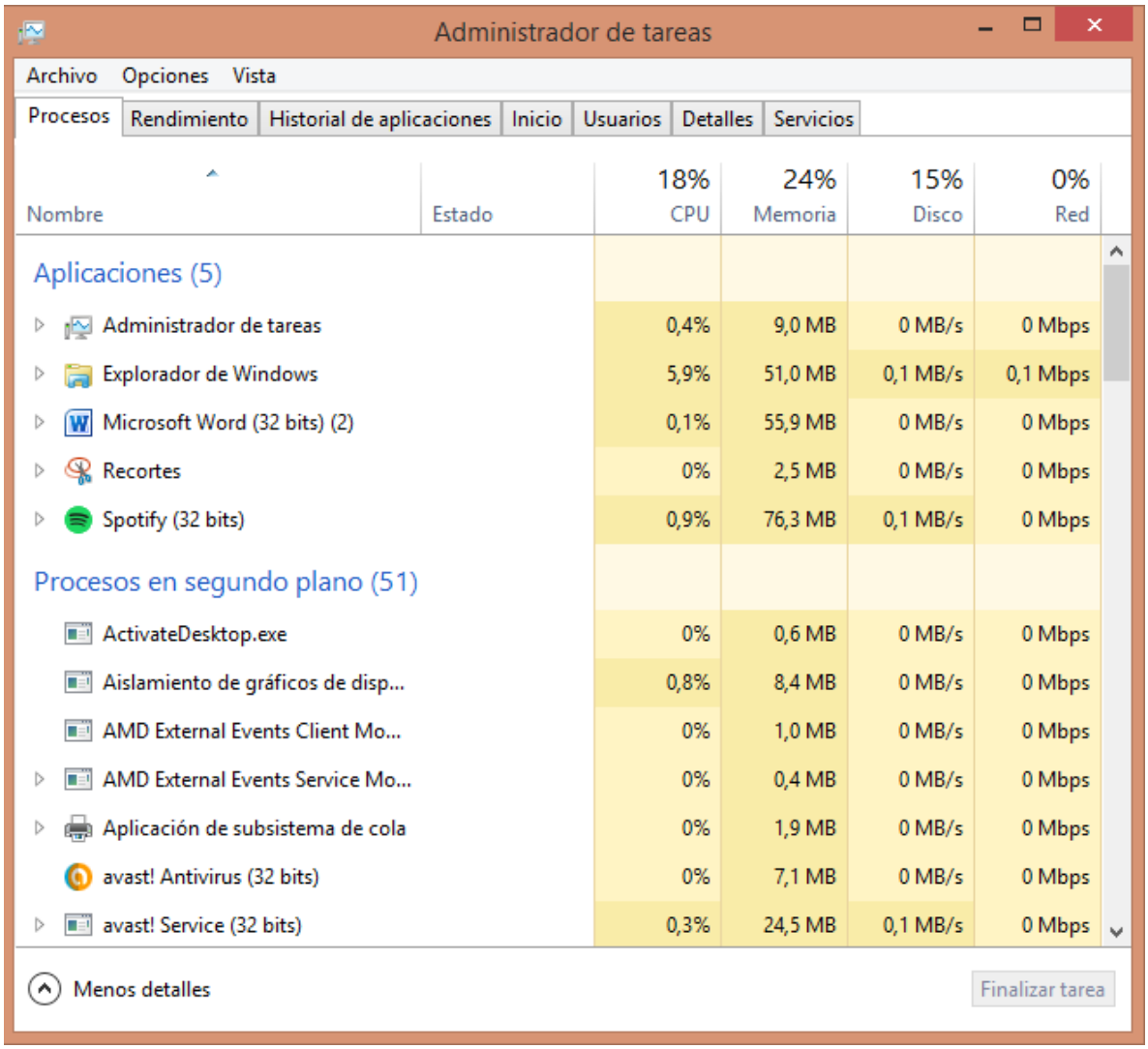
è Por prioridades. Es decir, el antivirus tendrá más prioridad que la calculadora, por ejemplo. O el Word al estar tecleando conseguirá más prioridad de la ventana activa para no dar sensación de lentitud o ir a saltitos… Pero esto tiene varios problemas, como la inacción. Es decir, si mientras P1 (proceso 1) está en ejecución por ganar a la P2 pero entra P3 que tiene mayor prioridad, saldrá P1 y ganará la contienda P3.

è Por tiempos. Mayor prioridad el que menor tiempo estimado tiene para terminar.

Eso sí, al programar no podremos controlar los métodos de ordenación de la contienda. No podemos garantizar que los procesos vayan a coger la CPU en el momento que vayamos a querer.

Procesos en los SSOO [3]

Para ver los procesos en Windows, iremos al Administrador de Tareas à Procesos



Para verlos desde la línea de comandos, en CMD escribimos tasklist. Además al hacerlo por línea de comandos veremos el PID, el identificador del proceso.

```
C:\Users\usuario>tasklist

Nombre de imagen              PID Nombre de sesión Núm. de ses Uso de memor
========================= ======== ================ =========== ============
System Idle Process             0 Services                   0         24 KB
System                          4 Services                   0     15.672 KB
smss.exe                      348 Services                   0      1.180 KB
csrss.exe                     520 Services                   0      4.588 KB
wininit.exe                   576 Services                   0      4.644 KB
csrss.exe                     596 Console                    1     21.852 KB
winlogon.exe                  652 Console                    1      7.796 KB
services.exe                  692 Services                   0      9.344 KB
lsass.exe                     700 Services                   0     20.148 KB
lsm.exe                       708 Services                   0      4.412 KB
svchost.exe                   808 Services                   0     10.344 KB
svchost.exe                   888 Services                   0     10.492 KB
MsMpEng.exe                   956 Services                   0    147.136 KB
svchost.exe                   368 Services                   0     20.384 KB
svchost.exe                   552 Services                   0    190.376 KB
svchost.exe                   680 Services                   0     19.708 KB
svchost.exe                   824 Services                   0     46.276 KB
svchost.exe                  1104 Services                   0      5.988 KB
svchost.exe                  1240 Services                   0     37.272 KB
spoolsv.exe                  1368 Services                   0     11.700 KB
```

En Ubuntu podemos verlo en la terminal con el comando ps.

```
inazio@inazio-VirtualBox:~$ ps
  PID TTY          TIME CMD
 1611 pts/1    00:00:00 bash
 1678 pts/1    00:00:00 ps
inazio@inazio-VirtualBox:~$
```

Si hacemos ps –f también aparecerá el PPID, el identificador del proceso padre.

```
inazio@inazio-VirtualBox:~$ ps -f
UID        PID  PPID  C STIME TTY          TIME CMD
inazio    1611  1604  0 00:31 pts/1    00:00:00 bash
inazio    2032  1611  0 00:35 pts/1    00:00:00 ps -f
```

Con ps –AF aparecen todos los procesos lanzados en el sistema.

```
inazio@inazio-VirtualBox:~$ ps -AF
UID        PID  PPID  C    SZ   RSS PSR STIME TTY          TIME CMD
root         1     0  0   885  1980   0 00:30 ?        00:00:00 /sbin/init
root         2     0  0     0     0   0 00:30 ?        00:00:00 [kthreadd]
root         3     2  0     0     0   0 00:30 ?        00:00:00 [ksoftirqd/0]
root         5     2  0     0     0   0 00:30 ?        00:00:00 [kworker/0:0H]
root         7     2  0     0     0   0 00:30 ?        00:00:00 [migration/0]
root         8     2  0     0     0   0 00:30 ?        00:00:00 [rcu_bh]
root         9     2  0     0     0   0 00:30 ?        00:00:00 [rcu_sched]
root        10     2  0     0     0   0 00:30 ?        00:00:00 [watchdog/0]
root        11     2  0     0     0   0 00:30 ?        00:00:00 [khelper]
root        12     2  0     0     0   0 00:30 ?        00:00:00 [kdevtmpfs]
root        13     2  0     0     0   0 00:30 ?        00:00:00 [netns]
root        14     2  0     0     0   0 00:30 ?        00:00:00 [writeback]
root        15     2  0     0     0   0 00:30 ?        00:00:00 [kintegrityd]
root        16     2  0     0     0   0 00:30 ?        00:00:00 [bioset]
root        17     2  0     0     0   0 00:30 ?        00:00:00 [crypto]
root        18     2  0     0     0   0 00:30 ?        00:00:00 [kworker/u3:0]
root        19     2  0     0     0   0 00:30 ?        00:00:00 [kblockd]
root        20     2  0     0     0   0 00:30 ?        00:00:00 [ata_sff]
root        21     2  0     0     0   0 00:30 ?        00:00:00 [khubd]
root        22     2  0     0     0   0 00:30 ?        00:00:00 [md]
root        23     2  0     0     0   0 00:30 ?        00:00:00 [devfreq_wq]
root        24     2  0     0     0   0 00:30 ?        00:00:00 [kworker/0:1]
root        26     2  0     0     0   0 00:30 ?        00:00:00 [khungtaskd]
```

PID y procesos en C [3]

Vamos a realizar las primeras prácticas con los procesos, los PID y los PPID en el lenguaje C. Para ello hemos instalado un entorno VitaLinux que trae ya incorporado el compilador gcc, pero podéis utilizar cualquier herramienta de vuestra elección que os permita trabajar programando en C.

EXECL

Execl sirve para ejecutar comandos del sistema y cualquier programa. Los argumentos son (const char *fichero, cons char *arg0, …, char *argN, (char*)NULL).

Devolverá -1 si hay condición de error.

Veamos un código de ejemplo

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

void main(){
     printf("Los archivos del directorio son: \n");
     execl("/bin/ls", "ls", "-l", (char *)NULL);
     printf("ERROR!!!");
}
```

Aquí estaremos haciendo un ls –l para ver todos los archivos del directorio desde donde ejecutemos este programa.

System

A diferencia de execl, system ejecutará sólo comandos del sistema, como podemos ver abajo. Vamos a ver el siguiente programa para hacernos una idea.

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

void main(){
     system("ls -l > ficSalida");
     printf("FIN");
}
```

Lo que hace el código es guardar el resultado de un ls –l de la carpeta actual del PATH a un fichero salida.

PID

En la arquitectura cliente – servidor tenemos dentro de un servidor web un proceso que está escuchando, en listen. Le hacen una petición y este proceso crea un hijo que sirva la página web al cliente. Una vez finalizada la tarea, el hijo sale de la memoria al finalizar su tarea. Durante todo la ejecución, el proceso padre se ha mantenido activo permaneciendo a la escucha de otras peticiones.

Vamos a ver un programa que nos muestre los identificadores del proceso actual y de su padre.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void main(){
     pid_t id_pactual, id_padre;

     id_pactual = getpid();
     id_padre = getppid();

     printf("PID actual: %d \n", id_pactual);
     printf("PID padre: %d \n", id_padre);
}
```

Analicemos este código.

- La línea de declaración *pid_t* nos indica el tipo de variable que será, una que nos sirva para almacenar el número de proceso que tenemos.
- La función *getpid()* devuelve el id del proceso actual
- Y la función *getppid()* devuelve el id del padre del proceso actual.

Ahora ya sabemos ver los id de los procesos, pero… ¿cómo se crean los procesos hijos?

Para eso tenemos la instrucción *fork()*, que es la encargada de crear un proceso hijo. Es decir, una copia exacta del proceso padre, pero a partir de ahora independientes.

Estos dos procesos, junto con todos los del sistema, entrarán en la contienda por la puja de la CPU independientemente. Y por tanto el control de tiempo también será independiente.

Para poder diferenciar uno de otro lo conseguimos con el valor devuelto por *fork()*. Con un ejemplo se verá más claro.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void main(){

     pid_t pid, hijo_pid;

     pid = fork(); // Aquí crea el proceso hijo

     if (pid == -1){
          printf("Ha habido un error");
          exit(-1);
     }
     if(pid == 0){
          // Nos encontramos dentro del proceso hijo
          printf("soy el proceso hijo \n\t Mi PID es %d. El PID de mi padre es:
%d. \n", getpid(), getppid());
     }
     else{
          hijo_pid = wait(NULL);
          printf("Soy el proceso padre: \n\t Mi PID es %d. El PID de mi padre es:
%d. \n\t Mi hijo %d terminó. \n", getpid(), getppid(), pid);
     }
}
```

Hay varios puntos que comentar. El primero es que como ya hemos dicho, los procesos pese a estar en un mismo programa, tanto el hijo como el padre son independientes en el momento de ejecutar *fork()*.

Si nos devuelve un valor igual a 0, sabemos que estamos tratando con el hijo, mientras que cualquier otro valor que no indique una condición de error nos hará saber que está funcionando el padre.

Dentro del padre tenemos la línea

hijo_pid = wait(NULL);

Con ésta línea indicaremos que va a esperar a la finalización del proceso hijo, y la variable pid guardará el PID del padre.

¿Ha quedado claro? Atrevámonos con un ejercicio sencillo.

Ejercicio. tomarun proceso, crear una variable y guarda un valor. 7, por ejemplo. El proceso crea un hijo que le suma 5, y el padre le resta 5. Visualiza por pantalla el resultado de ambas operaciones así como el PID y el PPID de ambos procesos.

Solución.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void main(){
     pid_t pid, hijo;
     int n = 7;

     pid = fork();

     if (pid == -1){
          printf("Error \n");
          exit(-1);
     }
     if (pid == 0){
          n = n + 5;
          printf("Soy el hijo. Valor de n = %d.\n Proceso %d, padre %d \n\n", n,
getpid(), getppid());
     }
     else{
          n = n - 5;
          printf("Soy el padre. Valor de n = %i.\n Proceso %d, padre %d \n\n", n,
getpid(), getppid());
     }

}
```

¿Qué es la memoria compartida? [4]

Cuando se crea un proceso, se reserva una zona de memoria con acceso exclusivo para dicho proceso, de tal manera que si otro proceso quiere acceder a esta zona de memoria, generalmente dará un error. El error suele aparecer con cierta frecuencia cuando no se gestionan adecuadamente los punteros en C y es entonces cuando ocurre el famoso “segmentation fault”.

Sin embargo hay una manera de indicarle al sistema operativo, que esa zona de memoria sí va a poder ser accedida por varios procesos. Para ello tenemos que utilizar una función especial, no nos valen las funciones de reserva de memoria dinámica habituales (malloc y calloc).

Desarrollo

Primeramente se van a generar los archivos para cada uno de los procesos (A, B, C y el principal).

Para ello se va a abrir una ventana de la terminal dentro de la carpeta de la práctica (figura 5.1).

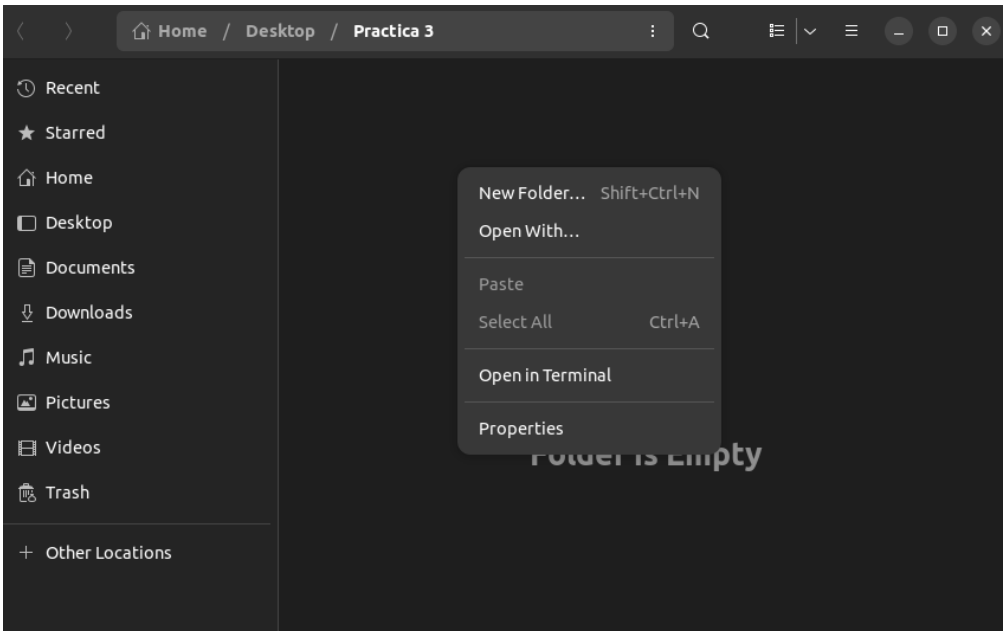


Figura 5.1 Vista de la carpeta Practica 3.

Dentro de la terminal se van a crear los archivos correspondientes para cada uno de los procesos que van a realizar la suma de los elementos de cada fila a través del comando touch (figuras 5.2).

```
:~/Desktop/Practica 3$ touch Main.c
:~/Desktop/Practica 3$ touch ProcesoA.c
:~/Desktop/Practica 3$ touch ProcesoB.c
:~/Desktop/Practica 3$ touch ProcesoC.c
```

Figura 5.2 Creación de los archivos para cada uno de los procesos.

Una vez ejecutados los comandos se verifica que se hayan creado todos los archivos (figura 5.3).

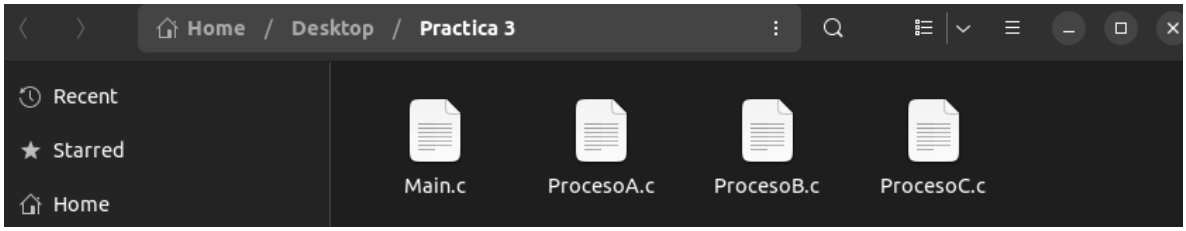


Figura 5.3 Ventana de la carpeta Practica 3 con los archivos creados.

Una vez creados los archivos, se van a programar cada uno de los procesos correspondientes.

- Proceso Principal.

Para el proceso principal (llamado archivo "Main"), se van a declarar las variables que nos van a permitir gestionar los espacios de memoria para la matriz que se va a almacenar dentro de la memoria compartida, así como las variables auxiliares para la creación de los espacios de memoria (figura 5.4).

```
int shmid0, shmid1, i,j;
int *resultados, *datos_matriz;
int **app_mat;
int matriz[3][9]={ {1, 2, 3, 4, 5, 6, 7, 8, 9},
                   {1, 3, 5, 7, 9, 11, 13, 15, 17},
                   {2, 4, 6, 8, 10, 12, 14, 16, 18 }};
key_t llave0, llave1;
pid_t pid;
```

Figura 5.4 Código del archivo Main donde se declaran las variables y se define la matriz de números que se va a almacenar en la memoria compartida.

Posteriormente se van a inicializar las llaves que van a definir cada región de la memoria compartida a través de la función “ftok” [5].

Una vez definidas las llaves se va a definir el espacio de la memoria compartida a manera de arreglos unidimensionales (vectores) por medio de la función “shmget” la cual va a retornar los ID’s correspondientes a cada segmento de la memoria compartida [6].

Nótese que el primer segmento de memoria compartida corresponde con la matriz numérica y en este caso se va a definir por medio de un vector de 3x9 (osease 27 elementos). Mientras que el segundo segmento de la memoria compartida se va a definir como un vector de únicamente 3 elementos para cada resultado obtenido por los procesos A, B y C.

Finalmente y haciendo uso de la función “shmat” se va a asociar cada segmento de la memoria compartida con una variable de tipo apuntador para poder hacer referencia a ese espacio de memoria (figura 5.5) [7].

```
llave0=ftok("Archivo1",'k');
llave1=ftok("Archivo2",'k');

shmid0=shmget(llave0,3*9*sizeof(datos_matriz[0]),IPC_CREAT|0777);
shmid1=shmget(llave1,3*sizeof(int),IPC_CREAT|0777);

datos_matriz=(int*)shmat(shmid0,0,0);
resultados=(int *)shmat(shmid1,0,0);
```

Figura 5.5 Código del programa Main donde se definen los segmentos de memoria compartida.

En el paso previo se definió el espacio de memoria para la matriz a manera de vector, en este caso se va a mapear dicho espacio de memoria compartida a un apuntador doble que se va a comportar como una matriz. Para ello, primeramente se va a reservar el espacio de memoria correspondiente con las 3 filas para dicho apuntador doble y posterior a eso, dentro de un bucle for se va a asignar el espacio en memoria de cada fila correspondiente a cada elemento múltiplo de 9, puesto que la matriz que se va a utilizar consta de 9 columnas.

De tal forma que se obtendrá una matriz de 3 filas por 9 columnas (figura 5.6).

```
app_mat=(int**)malloc(3*sizeof(app_mat[0]));

for(i=0;i<3;i++)
  {
     app_mat[i]=datos_matriz+i*9;
  }
```

Figura 5.6 Mapeo del vector de memoria compartida a un apuntador doble.

Posteriormente se va a asignar cada elemento de la matriz numérica a la memoria compartida a través del apuntador doble que se definió previamente (figura 5.7).

```
for(i=0;i<3;i++)
  {
     for(j=0;j<9;j++)
    {
        app_mat[i][j]=matriz[i][j];
    }
  }
```

Figura 5.7 Asignación de la matriz numérica a la memoria compartida.

Una vez definida la matriz de números dentro del segmento de memoria compartida, el programa va a estar a la espera de los resultados por los otros procesos, para ello se va a definir un bucle while que va a evaluar el valor almacenado dentro del arreglo de resultados. Una vez que los resultados de todos los procesos se hayan registrado en el arreglo, el programa va a salir del bucle while, para finalmente imprimir los resultados almacenados en el arreglo (figura 5.8).

```
while(resultados[0]==1 || resultados[1]==2 || resultados[2]==3);

//Evalua si ya se registraron todos los resultados de las sumas
if(resultados[0]!=1 && resultados[1]!=2 && resultados[2]!=3)
{
 for(i=0;i<3;i++)
  {
     printf("Suma Fila %d: %d\n",i+1,resultados[i]); //Imprime los resultados de cada fila
  }
}
```

Figura 5.8 Código del programa Main donde se espera a que los otros procesos registren su resultado dentro del arreglo correspondiente.

Finalmente, por medio de la función “shmdt” se van a disociar los segmentos de memoria compartida de los apuntadores [8] y posteriormente se van a eliminar dichos segmentos de memoria compartida a través de la función “shmctl” [9] como se muestra en la figura 5.9.

```
shmdt(&app_mat);
shmdt(&resultados);
shmctl(shmid0,IPC_RMID,0);
shmctl(shmid1,IPC_RMID,0);
return 0;
```

Figura 5.9 Destrucción de los segmentos de memoria existentes.

- Proceso A.

Para el proceso A se van a declarar e inicializar todas las variables que se declararon previamente en el proceso Main (figura 5.10).

```
int shmid0, shmid1, i,j,suma=0;
int *resultados, *datos_matriz;
int **app_mat;
key_t llave0, llave1;
pid_t pid;

llave0=ftok("Archivo1",'k');
llave1=ftok("Archivo2",'k');

shmid0=shmget(llave0,3*9*sizeof(datos_matriz[0]),IPC_CREAT|0777);
shmid1=shmget(llave1,3*sizeof(int),IPC_CREAT|0777);

datos_matriz=(int*)shmat(shmid0,0,0);
resultados=(int *)shmat(shmid1,0,0);

//Reservar memoria
app_mat=(int**)malloc(3*sizeof(app_mat[0]));

for(i=0;i<3;i++)
  {
     app_mat[i]=datos_matriz+i*9;
  }
```

Figura 5.10 Código del proceso A con la declaración de las variables.

Para este caso se va a realizar la suma de la primera fila por lo que se recorre la matriz almacenada en el segmento de la memoria compartida indicado por el índice 0 para la fila 1. En este caso se va a parar por 1 segundo la ejecución del programa para mantener cierta “sincronicidad” con los demás procesos (figura 5.11).

Figura 5.11 Código del proceso A con la suma de los elementos de la fila 1 de la matriz.

Se imprime el valor resultante de la suma y se guarda en la primera posición del arreglo de resultados (figura 5.12).

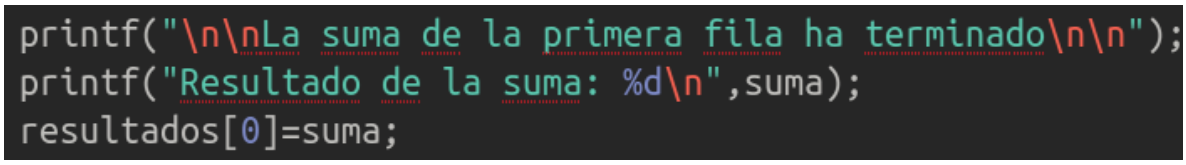


Figura 5.12 Código del proceso A con la impresión del resultado y la asignación del mismo al arreglo de resultados.

Finalmente se disocian los elementos del espacio de memoria compartida creado para este proceso. (figura 5.13)

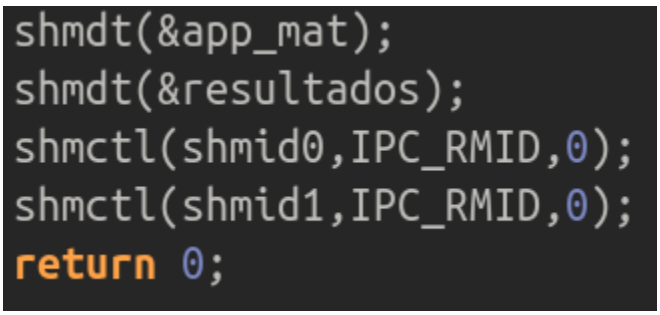


Figura 5.13 Código del proceso A con la disociación de los apuntadores al espacio de memoria compartida.

- Proceso B.

De manera similar se van a declarar e inicializar todas las variables que se declararon previamente en el proceso Main para el proceso B (figura 5.14).

```c
int shmid0, shmid1, i,j,suma=0;
int *resultados, *datos_matriz;
int **app_mat;
key_t llave0, llave1;
pid_t pid;

llave0=ftok("Archivo1",'k');
llave1=ftok("Archivo2",'k');

shmid0=shmget(llave0,3*9*sizeof(datos_matriz[0]),IPC_CREAT|0777);
shmid1=shmget(llave1,3*sizeof(int),IPC_CREAT|0777);

datos_matriz=(int*)shmat(shmid0,0,0);
resultados=(int *)shmat(shmid1,0,0);

//Reservar memoria
app_mat=(int**)malloc(3*sizeof(app_mat[0]));

for(i=0;i<3;i++)
  {
     app_mat[i]=datos_matriz+i*9;
  }
```

Figura 5.14 Código del proceso B con la declaración de las variables.

Para este caso se va a realizar la suma de la segunda fila de la matriz por lo que se recorre dicha matriz almacenada en el segmento de la memoria compartida indicado por el índice 1 para la fila 2. De igual forma se va a parar por 1 segundo la ejecución del programa para mantener cierta "sincronicidad" con los demás procesos (figura 5.15).

```c
for(j=0;j<9;j++)
{
    suma+=app_mat[1][j];
    sleep(1);
}
```

Figura 5.15 Código del proceso B con la suma de los elementos de la fila 2 de la matriz.

Se imprime el valor resultante de la suma y se guarda en la segunda posición del arreglo de resultados (figura 5.16).

```
printf("\n\nLa suma de la segunda fila ha terminado\n\n");
printf("Resultado de la suma: %d\n",suma);
resultados[1]=suma;
```

Figura 5.16 Código del proceso B con la impresión del resultado y la asignación del mismo al arreglo de resultados.

Finalmente se disocian los elementos del espacio de memoria compartida creados para este proceso. (figura 5.17)

```
shmdt(&app_mat);
shmdt(&resultados);
shmctl(shmid0,IPC_RMID,0);
shmctl(shmid1,IPC_RMID,0);
return 0;
```

Figura 5.17 Código de la disociación de los apuntadores al espacio de memoria compartida.

- Proceso C.

Por último y de manera similar se van a declarar e inicializar todas las variables que se declararon previamente en el proceso Main para el proceso C (figura 5.18).

```c
int shmid0, shmid1, i,j,suma=0;
int *resultados, *datos_matriz;
int **app_mat;
key_t llave0, llave1;
pid_t pid;

llave0=ftok("Archivo1",'k');
llave1=ftok("Archivo2",'k');

shmid0=shmget(llave0,3*9*sizeof(datos_matriz[0]),IPC_CREAT|0777);
shmid1=shmget(llave1,3*sizeof(int),IPC_CREAT|0777);

datos_matriz=(int*)shmat(shmid0,0,0);
resultados=(int *)shmat(shmid1,0,0);

//Reservar memoria
app_mat=(int**)malloc(3*sizeof(app_mat[0]));

for(i=0;i<3;i++)
  {
     app_mat[i]=datos_matriz+i*9;
  }
```

Figura 5.18 Código del proceso C con la declaración de las variables.

Para este caso se va a realizar la suma de la tercera fila de la matriz por lo que se recorre dicha matriz almacenada en el segmento de la memoria compartida indicado por el índice 2 para la fila 3. De igual forma se va a parar por 1 segundo la ejecución del programa para mantener cierta "sincronicidad" con los demás procesos (figura 5.19).

```c
for(j=0;j<9;j++)
{
   suma+=app_mat[2][j];
   sleep(1);
}
```

Figura 5.19 Código del proceso C con la suma de los elementos de la fila 3 de la matriz.

Se imprime el valor resultante de la suma y se guarda en la tercera posición del arreglo de resultados (figura 5.20).

```
printf("\n\nLa suma de la tercer fila ha terminado\n\n");
printf("Resultado de la suma: %d\n",suma);
resultados[2]=suma;
```

Figura 5.20 Código del proceso C con la impresión del resultado y la asignación del mismo al arreglo de resultados.

Finalmente se disocian los elementos del espacio de memoria compartida creados para este proceso. (figura 5.21)

```
shmdt(&app_mat);
shmdt(&resultados);
shmctl(shmid0,IPC_RMID,0);
shmctl(shmid1,IPC_RMID,0);
return 0;
```

Figura 5.21 Código de la disociación de los apuntadores al espacio de memoria compartida.

- Resultados

Para la ejecución de los procesos se van a abrir 4 terminales (1 para el proceso principal y 3 para los procesos A, B y C) dentro de la carpeta de la práctica 3 como se muestra en la figura 5.22.

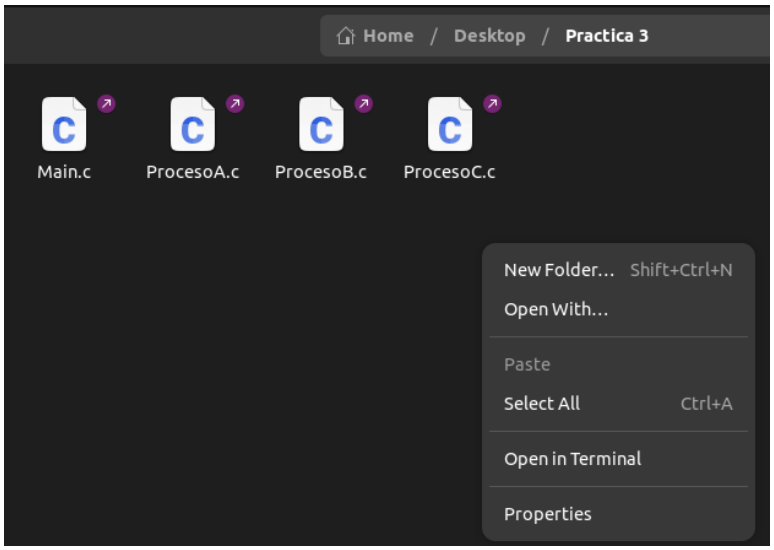


Figura 5.22 Creación de 4 ventanas de la terminal dentro de la carpeta "Practica 3".

Una vez creadas las 4 ventanas de la terminal se va a compilar cada archivo por separado como se muestra en la figura 5.23.

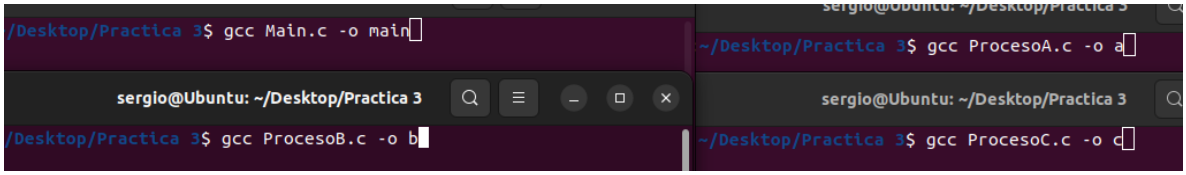


Figura 5.23 Compilación de los programas por separado.

Finalmente se van a ejecutar los programas en cada terminal como se muestra en la figura 5.24.

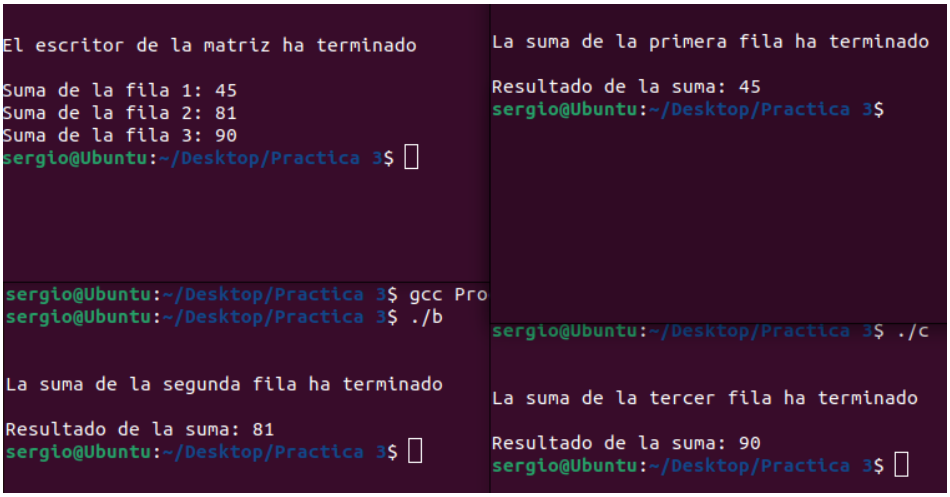


Figura 5.24 Ejecución de los programas de forma simultánea.

Como se puede observar en los resultados cada programa accede a los elementos de la matriz que se está almacenando en la memoria compartida de tal forma que accede únicamente a una fila de dicha matriz y durante la ejecución de cada programa hay un tiempo de 5 segundos aproximadamente, antes de que se observen los resultados esto debido al uso de la función sleep en cada programa, ya que si se retiran dichos sleeps, todos los programas acceden a dicho espacio de memoria compartida al mismo tiempo, causando que dicho segmento de memoria se bloquee impidiendo el acceso a los procesos y por ende, impidiendo que realicen dicha suma, lo que a su vez causa que el proceso principal nunca reciba la información de los resultados, entrando en un bucle infinito como se observa en la figura 5.25.

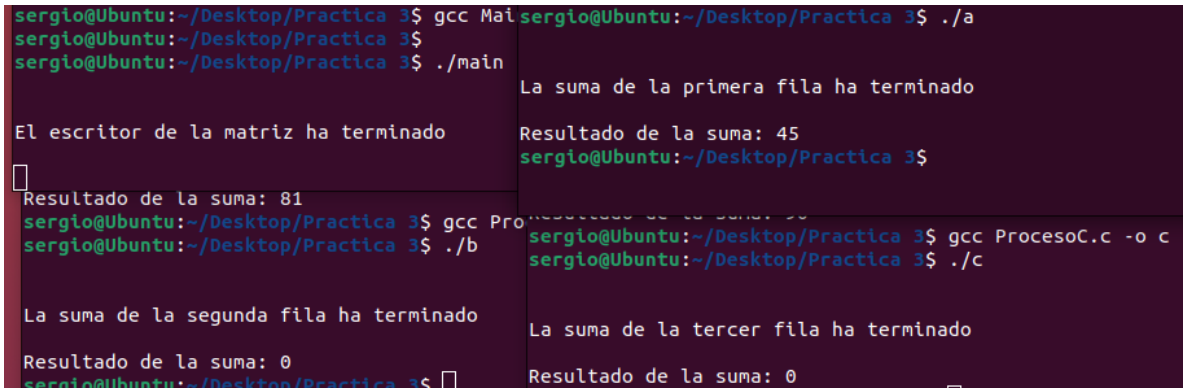


Figura 5.25 Ejecución de los programas de forma simultánea sin el uso de "sleeps".

Conclusiones

Tinoco Videgaray Sergio Ernesto

La memoria compartida es un elemento de gran utilidad cuando se trata de paralelizar tareas o procesos, ya que el uso de memoria compartida nos permite trabajar directamente sobre la memoria principal (en este caso la RAM) de tal forma que los tiempos de acceso se ven reducidos de manera significativa en comparación con los tiempos de acceso a la memoria secundaria (como el disco duro), y es por ello que al utilizar la memoria compartida estamos generando un área de trabajo en la que diferentes procesos puedan acceder de forma simultánea para leer o escribir la información que necesiten sin tener que trabajar con espacios de memoria independientes en cada proceso como en el caso de la memoria distribuida, lo cual resulta más eficiente en términos de espacio. Aunque claro que no es del todo eficiente ya que en este caso estamos trabajando con cierta asincronicidad a la hora de acceder a los elementos de la matriz debido al uso de la función sleep, aunque esto se puede mejorar con el uso de otras herramientas como los semáforos.

Ibarra Gonzalez Emilio Francisco

La memoria compartida es un mecanismo fundamental en el desarrollo de sistemas operativos y aplicaciones de software. Permite que múltiples procesos accedan a un segmento de memoria compartido, lo que facilita la comunicación y la cooperación entre procesos. Es importante en situaciones en las que múltiples procesos necesitan compartir datos entre ellos, pero no es práctico copiar los datos a cada proceso individualmente. En lugar de eso, los procesos pueden acceder a una región de memoria compartida, lo que les permite compartir información en tiempo real sin

necesidad de copiar datos de un proceso a otro. Sin embargo, también presenta ciertos desafíos, como la sincronización y la protección de datos compartidos. Los procesos deben coordinar su acceso a la memoria compartida para evitar conflictos y garantizar la integridad de los datos. Además, se debe utilizar un mecanismo de protección para garantizar que los procesos sólo accedan a las partes de la memoria compartida que les corresponde.

Sánchez De Los Ríos Flavio Josué

En resumen, se desarrolló un programa en el lenguaje C en el sistema operativo Ubuntu que utiliza la memoria compartida para descomponer una tarea en múltiples procesos y realizar la suma de cada elemento de una matriz de tres filas en tres procesos diferentes. La memoria compartida permite una comunicación eficiente entre procesos y se utiliza en situaciones donde se requiere el intercambio de datos en tiempo real o la colaboración entre múltiples procesos. Además, se discutió sobre los procesos en informática, la multitarea y los diferentes estados de un proceso.

Referencias

[1] Rewini H.; Abd-El-Barr, Mostafa (2005). *Advanced Computer Architecture and Parallel Processing*. Wiley-Interscience. pp. 77–80. ISBN 978-0-471-46740-3.

[2] Jeffrey S. Chase; Henry M.; Michael J. Feeley; and Edward D. Lazowska. "Sharing and Protection in a Single Address Space Operating System". doi:10.1145/195792.195795 1993. p. 3

[3]Inazio; Programando a Paitos http://www.programandoapasitos.com/2015/09/servicios-y-procesos-procesos-y-c-i.html#:~:text=Proceso%20%3D%20Programa%20en%20ejecuci%C3%B3n&text=Esto%20se%20conoce%20como%20el,dejar%20espacio%20a%20otro%20proceso

[4]Jorge Duran; Programación paralela en C : Memoria compartida (2015)

https://www.somosbinarios.es/programacion-paralela-en-c-memoria-compartida/

[5] Ubuntu Manpage Repository. (2019). “ftok” Disponible en:

https://manpages.ubuntu.com/manpages/trusty/es/man3/ftok.3.html

[6] Ubuntu Manpage Repository. (2019). “shmget” Disponible en:

https://manpages.ubuntu.com/manpages/bionic/es/man2/shmget.2.html

[7] Kerrisk M. (2022). “shmat” Disponible en:
https://man7.org/linux/man-pages/man3/shmat.3p.html

[8] Kerrisk M. (2022). “shmdt” Disponible en:

https://linux.die.net/man/2/shmdt

[9] Ubuntu Manpage Repository. (2019). “shmctl” Disponible en:

https://manpages.ubuntu.com/manpages/bionic/es/man2/shmctl.2.html