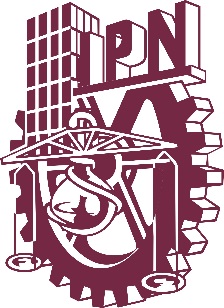
[](https://pbs.twimg.com/profile_images/1423089146/escom.png)**Instituto Politécnico Nacional**

Escuela Superior de Cómputo

Estrada Estrada Jesse Yael

Maya Rocha Luis Emmanuel

Mendoza Ramírez Álvaro

Silva Rojas Karla Lizbeth

**Práctica 7:**

Mecanismos de sincronización de proceso en Linux y Windows (Semáforos)

Prof.: Cortés Galicia Jorge

Sistemas Operativos

2CM10

Fecha de entrega: 5 de junio de 2018

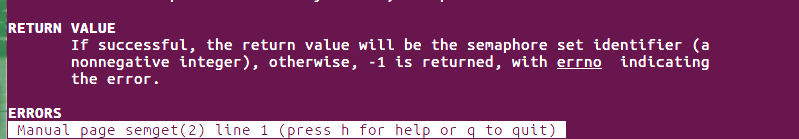
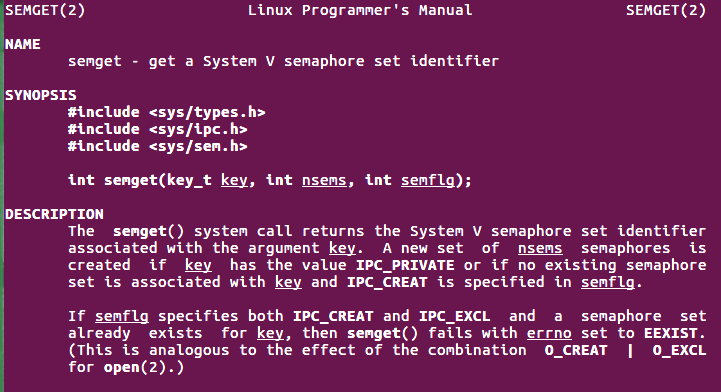
**Práctica 7:** Mecanismos de sincronización de proceso en Linux y Windows (semáforos).

**Competencia:** El alumno comprende el funcionamiento de los mecanismos de sincronización entre procesos utilizando los semáforos como árbitro de acceso para el desarrollo de aplicaciones cooperativas en el sistema operativo Linux como Windows.

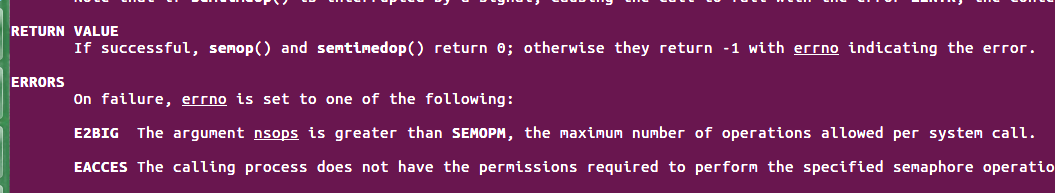
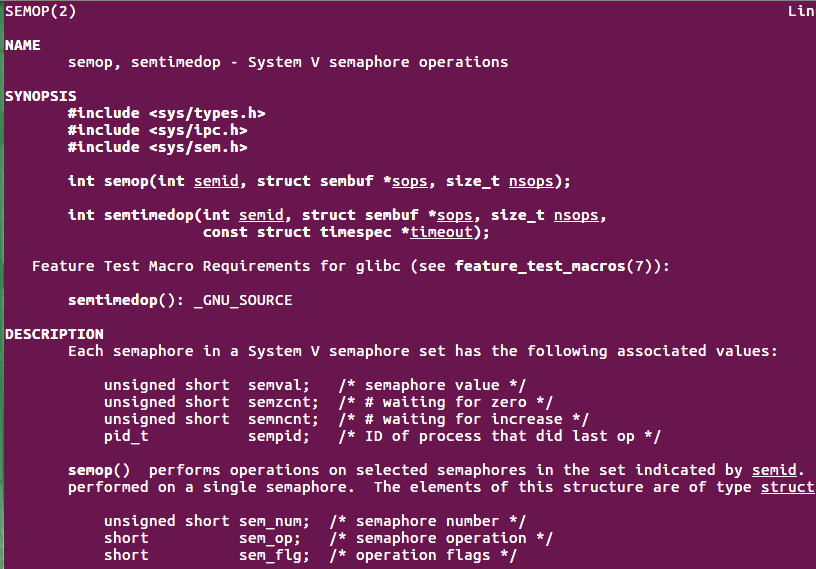
**Desarrollo.**

1. A través de la ayuda en línea que proporciona Linux, investigue el funcionamiento de las funciones **semget(), semop().** Explique los argumentos, retorno de las funciones y las estructuras y uniones relacionadas con dichas funciones.

* **Semget():**



* **Semop():**



1. Capture, compile y ejecute el siguiente programa. Observe su funcionamiento y explique.

#include<stdio.h>

#include<sys/ipc.h>

#include<sys/sem.h>

#include<sys/types.h>

int main(void)

{

int i, j;

int pid;

int semid;

key\_t llave = 1234;

int semban = IPC\_CREAT|0666;

int nsems = 1;

int nsops;

struct sembuf \*sops = (struct sembuf \*)malloc(2\*izeof(struct sembuf));

printf("Iniciando semaforo...\n");

if((semid = semget(llave, nsems, semban))==-1)

{

perror("semget:error al iniciar semaforo");

exit(1);

}

else

printf("Semaforo iniciado...\n");

if((pid = fork())<0)

{

perror("fork: error al crear proceso\n");

exit(1);

}

if(pid == 0)

{

i = 0;

while(i<3)

{

nsops = 2;

sops[0].sem\_num = 0;

sops[0].sem\_op = 0;

sops[0].sem\_flg = SEM\_UNDO;

sops[1].sem\_num = 0;

sops[1].sem\_op = 1;

sops[1].sem\_flg = SEM\_UNDO|IPC\_NOWAIT;

printf("semop: hijo llamando a semop(%d,&sops,%d)con:", semid, nsops);

for(j = 0; j<nsops; j++)

{

printf("\n\tsops[%d].sem\_num = %d,",j,sops[j].sem\_num);

printf("sem\_op = %d,",sops[j].sem\_op);

printf("sem\_flg = %#o\n", sops[j].sem\_flg);

}

if((j = semop(semid, sops, nsops)) == -1)

{

perror("semop:error en operación del semáforo\n");

}

else{

printf("\tsemop: regreso de semop()%d\n",j);

printf("\n\n Proceso hijo toma el control del semaforo: %d/3 veces\n", i+1);

sleep(5);

nsops = 1;

sops[0].sem\_num = 0;

sops[0].sem\_op = -1;

sops[0].sem\_flg = SEM\_UNDO|IPC\_NOWAIT;

if((j = semop(semid,sops,nsops)) == -1)

{

perror("semop: error en operacion del semaforo\n");

}

else

printf("Proceso hijo regresa el control del semaforo:%d/3 veces\n",i+1);

sleep(5);

}

++i;

}

}

else{

i = 0;

while (i<3){

nsops = 2;

sops[0].sem\_num = 0;

sops[0].sem\_op = 0;

sops[0].sem\_flg =SEM\_UNDO;

sops[1].sem\_num = 0;

sops[1].sem\_op = 1;

sops[1].sem\_flg = SEM\_UNDO|IPC\_NOWAIT;

printf("\nsemop: Padre llamando semop(%d, &sops, %d)con:", semid, nsops);

for(j = 0; j<nsops; j++)

{

printf("\n\tsops[%d].sem\_num = %d,",j,sops[j].sem\_num);

printf("sem\_op = %d,", sops[j].sem\_op);

printf("sem\_flg =%#o\n", sops[j].sem\_flg);

}

if((j = semop(semid, sops, nsops))==-1)

{

perror("semop: error en operacion del semaforo\n");

}

else{

printf("semop: regreso de semop()%d\n", j);

printf("Proceso padre toma el control del semaforo: %d/3 veces\n",i+1);

sleep(5);

nsops =1;

sops[0].sem\_num = 0;

sops[0].sem\_op = -1;

sops[0].sem\_flg = SEM\_UNDO|IPC\_NOWAIT;

if((j = semop(semid,sops, nsops)) == -1){

perror("semop: error en semop()\n");

}

else

printf("Proceso padre regresa el control del semaforo: %d/3 veces\n", i+1);

sleep(5);

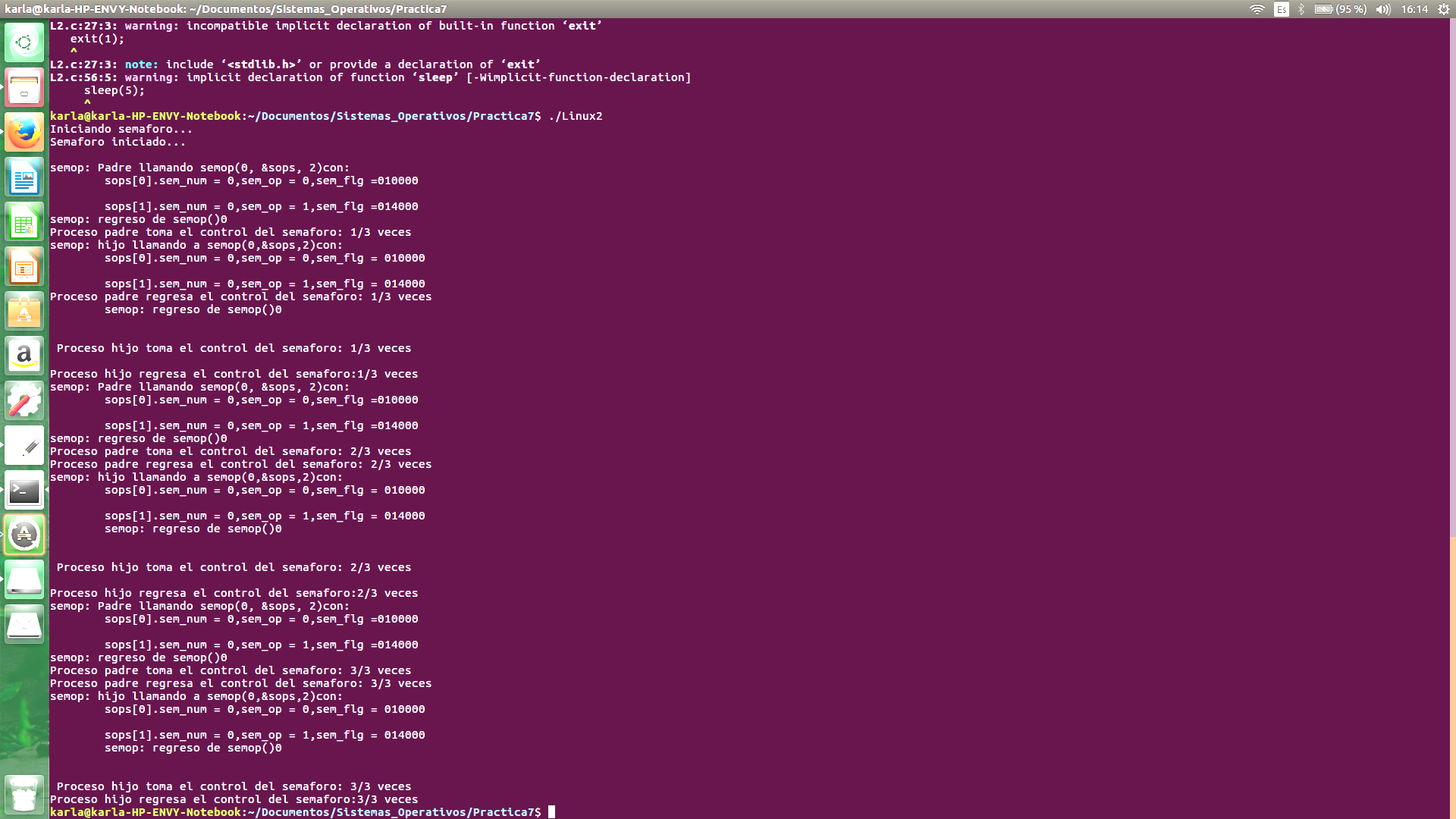
}

++i;

}

}

}



1. Capture, compile y ejecute los siguientes programas. Observe su funcionamiento.

**Programa padre**

#include<windows.h> //Programa Padre

#include<stdio.h>

int main(int argc, char \*argv[])

{

STARTUPINFO si; //Estructura de información inicial para Windows

PROCESS\_INFORMATION pi; //Estrucrura de información del admin de procesos

HANDLE hSemaforo;

int i =1;

ZeroMemory(&si, sizeof(si));

si.cb = sizeof(si);

ZeroMemory(&pi, sizeof(pi));

if(argc!=2)

{

printf("Usar: %s Programa\_hijo\n", argv[0]);

return 0;

}

//Creación del semáforo

if((hSemaforo = CreateSemaphore(NULL, 1, 1, "Semaforo")) == NULL)

{

printf("Falla al invocar CreateSemaphore: %d\n", GetLastError());

return -1;

}

// Creación proceso hijo

if(!CreateProcess(NULL, argv[1], NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi))

{

printf("Falla al invocar CreateProcess: %d\n",GetLastError());

return -1;

}

while(i < 4)

{

//Prueba del semaforo

WaitForSingleObject(hSemaforo, INFINITE);

//Sección crítica

printf("Soy el padre entando %i de 3 veces al semaforo\n", i);

sleep(5000);

//Liberación del semáforo

if(!ReleaseSemaphore(hSemaforo, 1 , NULL))

{

printf("Falla al invocar ReleaseSemaphore: %d\n", GetLastError());

}

printf("Soy el padre liberando %i de 3 veces al semaforo\n", i);

sleep(5000);

i++;

}

//Terminación controlada del poceso hilo asociado de ejecución

CloseHandle(pi.hProcess);

CloseHandle(pi.hThread);

}

**Programa hijo**

#include<windows.h> //Programa hijo

#include<stdio.h>

int main()

{

HANDLE hSemaforo;

int i =1;

//Apertura del semaforo

if((hSemaforo = OpenSemaphore(SEMAPHORE\_ALL\_ACCESS, FALSE, "Semaforo")) == NULL)

{

printf("Falla al invocar OpenSemaphore: %d\n", GetLastError());

return -1;

}

while(i < 4)

{

//Prueba del semaforo

WaitForSingleObject(hSemaforo, INFINITE);

// Sección crítica

printf("Soy el hijo entrando %i de 3 veces al semaforo\n", i);

sleep(5000);

//Liberación del semaforo

if((!ReleaseSemaphore(hSemaforo,1,NULL)))

{

printf("Falla al invocar ReleaseSemaphore: %d\n", GetLastError());

sleep(5000);

i++;

}

}

}

1. Programe la misma aplicación del punto 7 de la práctica 6 (tanto para Linux como para Windows), utilizando como máximo tres regiones de memoria compartida de 400 bytes cada una para almacenar todas las matrices requeridas por la aplicación. Utilice como mecanismo de sincronización los semáforos revisados en esta práctica, tanto para la escritura y como para la lectura de las memorias compartidas. Úselos en los lugares donde haya necesidad de sincronizar el acceso a memoria compartida.

* **Linux**

**Código Fuente.**

**Funcionamiento.**

* **Windows**

**Código Fuente.**

**“Principal.c”**

#include <stdio.h>

#include <windows.h>

#define TAM\_MEM 400

#define TAM 10

#include"math.h"

#define SWAP(a, b) {temp = (a);(a) = (b);(b) = temp;}

#include"time.h"

HANDLE crearArchivo(char\* direccion){

HANDLE hFile;

hFile=CreateFile(direccion,GENERIC\_READ|GENERIC\_WRITE,FILE\_SHARE\_READ,NULL,CREATE\_NEW,FILE\_ATTRIBUTE\_NORMAL,NULL);

return hFile;

}

BOOL write(HANDLE hFile,char\* texto){

BOOL bFile;

DWORD numberofbytes=(DWORD)strlen(texto);

DWORD numberwritten=0;

bFile= WriteFile(hFile,texto,numberofbytes,&numberwritten,NULL);

return bFile;

}

void stchcat(char \*cadena, char chr)

{

size\_t longitud = strlen(cadena);

\*(cadena + longitud) = chr;

\*(cadena + longitud + 1) = '\0';

}

void ImpMatriz(float\*\* a){

int i,j;

for(i=0;i<TAM;i++){

for(j=0;j<TAM;j++){

printf("%.1f|\t",a[i][j]);

}

printf("\n");

}

}

float\*\* generar(){

int i;

float\*\* a=(float\*\*)malloc(sizeof(float\*)\*TAM);

for(i=0;i<TAM;i++){

a[i]=(float\*)malloc(sizeof(float)\*TAM);

}

return a;

}

float\*\* toPointer(float a[TAM][TAM]){

float\*\* x=generar();

int i,j;

for(i=0;i<TAM;i++){

for(j=0;j<TAM;j++){

x[i][j]=a[i][j];

}

}

return x;

}

double

Determinante (float m1[TAM][TAM],int n)

{

double s = 1, det = 0;

int i, j, k, m, x;

HANDLE hFile;

float m2[TAM][TAM];

if (n == 1)

{

return m1[0][0];

}

else

{

for (k = 0; k < n; k++)

{

m = 0;

x = 0;

for (i = 0; i < n; i++)

{

for (j = 0; j < n; j++)

{

m2[i][j] = 0;

if (i != 0 && j != k)

{

m2[m][x] = m1[i][j];

if (x < (n - 2))

{

x++;

}

else

{

x = 0;

m++;

}

}

}

}

det = det + s \* (m1[0][k] \* Determinante(m2, n-1));

s \*= -1;

}

}

return det;

}

void matrizCofactores (float m1[TAM][TAM], int n, double determinante)

{

char resultado[700];

char buffer[8]={};

int q, m, i, j, k, l;

HANDLE hFile;

float m2[TAM][TAM], matrizFactores[TAM][TAM], aux[TAM][TAM], inversa[TAM][TAM];

for (l = 0; l < n; l++)

{

for (k = 0; k < n; k++)

{

m = 0; q = 0;

for (i = 0; i < n; i++)

{

for (j = 0; j < n; j++)

{

if ( i != l && j != k)

{

m2[m][q] = m1[i][j];

if (q < (n-2))

{

q++;

}

else

{

q = 0; m++;

}

}

}

}

matrizFactores[l][k] = (pow(-1, l + k)\* Determinante (m2, n-1));

}

}

for (i = 0; i < TAM; i++)

{

for (j = 0; j < TAM; j++)

{

aux[i][j] = matrizFactores[j][i];

}

}

for (i = 0; i < TAM; i++)

{

for (j = 0; j < TAM; j++)

{

inversa[i][j] = aux[i][j] / determinante;

}

}

strcpy(resultado,"\r\nMATRIZ INVERSA A\r\n ");

for (i = 0; i < TAM; i++)

{

for (j = 0; j < TAM; j++)

{

sprintf(buffer,"%.3lf\t",inversa[i][j]);

strcat(resultado,buffer);

memset(buffer,0,8);

}

strcat(resultado,"\r\n");

}

printf("%s\n",resultado);

DWORD noReaded=0;

hFile=crearArchivo("C:\\Users\\Principal\\Documents\\4to Semestre\\Sistemas Operativos\\Practica7\\inversa1.txt");

BOOL bFile=write(hFile,resultado);

}

void

matrizCofactores2 (float m1[TAM][TAM], int n, double determinante)

{

char resultado[700];

char buffer[8]={};

HANDLE hFile;

int q, m, i, j, k, l;

float m2[TAM][TAM], matrizFactores[TAM][TAM], aux[TAM][TAM], inversa[TAM][TAM];

for (l = 0; l < n; l++)

{

for (k = 0; k < n; k++)

{

m = 0; q = 0;

for (i = 0; i < n; i++)

{

for (j = 0; j < n; j++)

{

if ( i != l && j != k)

{

m2[m][q] = m1[i][j];

if (q < (n-2))

{

q++;

}

else

{

q = 0; m++;

}

}

}

}

matrizFactores[l][k] = (pow(-1, l + k)\* Determinante (m2, n-1));

}

}

for (i = 0; i < TAM; i++)

{

for (j = 0; j < TAM; j++)

{

aux[i][j] = matrizFactores[j][i];

}

}

for (i = 0; i < TAM; i++)

{

for (j = 0; j < TAM; j++)

{

inversa[i][j] = aux[i][j] / determinante;

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*IMPRESION DE LA INVERSA\*\*\*\*\*\*\*\*\*\*\*\*/

strcpy(resultado,"\r\nMATRIZ INVERSA B \r\n");

for (i = 0; i < TAM; i++)

{

for (j = 0; j < TAM; j++)

{

sprintf(buffer,"%.3lf\t",inversa[i][j]);

strcat(resultado,buffer);

memset(buffer,0,8);

}

strcat(resultado,"\r\n");

}

printf("%s\n",resultado);

hFile=crearArchivo("C:\\Users\\Principal\\Documents\\4to Semestre\\Sistemas Operativos\\Practica7\\inversa2.txt");

BOOL bFile=write(hFile,resultado);

}

int main(int argc,char \*argv[]){

STARTUPINFO si;

PROCESS\_INFORMATION pi;

ZeroMemory(&si, sizeof(si));

si.cb = sizeof(si);

ZeroMemory(&pi, sizeof(pi));

float A[10][10], B[10][10], b[10][10], c[10][10], d[10][10], mandada1[10][10], mandada2[10][10];

char \*PH = "PadreHijo";

char \*HP = "HijoPadre";

char \*NP = "NietoPadre";

srand(time(NULL));

HANDLE hArchMapeoPH, hArchMapeoHP, hArchMapeoNP,hSemaforo;

int i,j,k,\*aPH, \*aHP, \*aNP,\*shmPH, \*shmHP, \*shmNP;

int ct=0;

if((hSemaforo=CreateSemaphore(NULL,1,1,"Semaforo"))==NULL){

printf("Falla al invocar CreateSemaphore:%d\n",GetLastError());

return -1;

}

puts("Se creo el semaforo");

if (!CreateProcess(NULL, argv[1], NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi))

{

printf("Fallo al invocar CreateProcess(%d)\n", GetLastError());

return;

}

puts("Se creo el proceso hijo");

if ((hArchMapeoPH = CreateFileMapping(INVALID\_HANDLE\_VALUE, NULL, PAGE\_READWRITE, 0, TAM\_MEM, PH)) == NULL)

{

printf("No se mapeo la memoria compartida: (%i)\n", GetLastError());

exit(-1);

}

puts("Se creo el mapeo padre");

if ((shmPH = (int \*)MapViewOfFile(hArchMapeoPH, FILE\_MAP\_ALL\_ACCESS, 0, 0, TAM\_MEM)) == NULL)

{

printf("No se creo la memoria compartida: (%i)\n", GetLastError());

CloseHandle(hArchMapeoPH);

exit(-1);

}

aPH = shmPH;

for (i = 0; i < 10; i++)

{

for (j = 0; j < 10; j++)

{

\*aPH = rand() % 15;

mandada1[i][j] = \*aPH;

\*aPH++;

}

}

for (i = 0; i < 10; i++)

{

for (j = 0; j < 10; j++)

{

\*aPH = rand() % 15;

mandada2[i][j] = \*aPH;

\*aPH++;

}

}

\*aPH = 101;

while (\*shmPH != -1){

puts("Esperando Padre->Hijo");

sleep(1);

}

WaitForSingleObject(hSemaforo,INFINITE);

/\*printf("2 MATRICES. PADRE -> HIJO. PADRE.\nMatriz 1.\n");

for (i = 0; i < 10; i++)

{

for (j = 0; j < 10; j++)

{

printf("%d\t", mandada1[i][j]);

}

printf("\n");

}

printf("Matriz 2\n");

for (i = 0; i < 10; i++)

{

for (j = 0; j < 10; j++)

{

printf("%d\t", mandada2[i][j]);

}

printf("\n");

}\*/

//WaitForSingleObject(hSemaforo,INFINITE);

UnmapViewOfFile(shmPH);

CloseHandle(hArchMapeoPH);

if((hSemaforo=OpenSemaphore(SEMAPHORE\_ALL\_ACCESS,TRUE,"Semaforo"))==NULL){

printf("Falla al invocar OpenSemaphore nieto:%d\n",GetLastError());

}

puts("Recibiendo del hijo");

if ((hArchMapeoHP = OpenFileMapping(FILE\_MAP\_ALL\_ACCESS, TRUE, HP)) == NULL)

{

printf("No se ario archsadfadsfdfdfdivo de mapeo de la memoria: (%i)\n", GetLastError());

exit(-1);

}

if ((shmHP = (int \*)MapViewOfFile(hArchMapeoHP, FILE\_MAP\_ALL\_ACCESS, 0, 0, TAM\_MEM)) == NULL)

{

printf("No se accedio a la memoria compartida: (%i)\n", GetLastError());

CloseHandle(hArchMapeoHP);

exit(-1);

}

aHP = shmHP;

for (i = 0; i < 10; i++)

{

for (j = 0; j < 10; j++)

{

A[i][j] = \*aHP;

\*aHP++;

}

}

\*shmHP = -1;

/\*printf("PRODUCTO. HIJO -> PADRE. PADRE.\n");

for (i = 0; i < 10; i++)

{

for (j = 0; j < 10; j++)

{

printf("%d\t", A[i][j]);

}

printf("\n");

}

printf("\n");\*/

UnmapViewOfFile(shmHP);

CloseHandle(hArchMapeoHP);

if(!ReleaseSemaphore(hSemaforo,1,NULL)){

printf("Falla al invocar ReleaseSemaphore: %d\n",GetLastError());

}

//RECIBE MATRIZ DEL NIETO

if((hSemaforo=OpenSemaphore(SEMAPHORE\_ALL\_ACCESS,TRUE,"Semaforo"))==NULL){

printf("Falla al invocar OpenSemaphore nieto:%d\n",GetLastError());

}

if ((hArchMapeoNP = OpenFileMapping(FILE\_MAP\_ALL\_ACCESS, FALSE, NP)) == NULL)

{

printf("No se ario PADRE archivo de mapeo de la memoria: (%i)\n", GetLastError());

exit(-1);

}

if ((shmNP = (int \*)MapViewOfFile(hArchMapeoNP, FILE\_MAP\_ALL\_ACCESS, 0, 0, TAM\_MEM)) == NULL)

{

printf("No se accedio a la memoria compartida: (%i)\n", GetLastError());

CloseHandle(hArchMapeoNP);

exit(-1);

}

aNP = shmNP;

for (i = 0; i < 10; i++)

{

for (j = 0; j < 10; j++)

{

B[i][j] = \*aNP++;

}

}

\*shmNP = -1;

/\*printf("SUMA. NIETO -> PADRE. PADRE.\n");

for (i = 0; i < 10; i++)

{

for (j = 0; j < 10; j++)

{

printf("%d\t", B[i][j]);

}

printf("\n");

}

printf("\n");\*/

UnmapViewOfFile(shmNP);

CloseHandle(hArchMapeoNP);

if(Determinante(A,10)!=0){

matrizCofactores(A,10,Determinante(A,10));

}

else{

printf("La matriz producto no tiene inversa");

}

if(Determinante(B,10)!=0){

matrizCofactores2(B,10,Determinante(B,10));

}

else{

printf("La matriz suma no tiene inversa");

}

ReleaseSemaphore(hSemaforo,1,NULL));

//printf("Falla al invocar ReleaseSemaphore: %d\n",GetLastError());

printf("El padre libero el semaforo\n");

Sleep(3000);

CloseHandle(pi.hProcess);

CloseHandle(pi.hThread);

return 0;

}

“Secundario.c”

#include <stdio.h>

#include <stdlib.h>

#include <windows.h>

#include <time.h>

#define TAM\_MEM 400

int main(int argc, char \*argv[])

{

STARTUPINFO siH;

PROCESS\_INFORMATION piH;

ZeroMemory(&siH, sizeof(siH));

siH.cb = sizeof(siH);

ZeroMemory(&piH, sizeof(piH));

HANDLE hSemaforo;

int A[10][10], B[10][10], b[10][10], c[10][10], d[10][10], mandada1[10][10], mandada2[10][10], producto[10][10];

int aux, suma;

char \*PH = "PadreHijo"; //padre hijo

char \*HP = "HijoPadre"; //hijo padre

char \*HN = "HijoNieto"; //hijo nieto

HANDLE hArchMapeoPH, hArchMapeoHP, hArchMapeoHN;

int i, j, k, shmid,ct=0;

int \*aPH, \*aHP, \*aHN;

srand(time(NULL));

int \*shmPH, \*shmHP, \*shmHN;

if((hSemaforo=OpenSemaphore(SEMAPHORE\_ALL\_ACCESS,TRUE,"Semaforo"))==NULL){

printf("Falla al invocar OpenSemaphore:%d\n",GetLastError());

return -1;

}

if (!CreateProcess(NULL,"prueba3", NULL, NULL, FALSE, 0, NULL, NULL, &siH, &piH))

{

printf("Fallo al invocar CreateProcess(%d)\n", GetLastError());

return;

}

puts("Se creo el proceso hijo(en el hijo)\n");

//MANDA MATRIZ A NIETO

if ((hArchMapeoHN = CreateFileMapping(INVALID\_HANDLE\_VALUE, NULL, PAGE\_READWRITE, 0, TAM\_MEM, HN)) == NULL)

{

printf("No se mapeo la memoria compartida: (%i)\n", GetLastError());

exit(-1);

}

puts("Se creo el mapeo hijo");

if ((shmHN = (int \*)MapViewOfFile(hArchMapeoHN, FILE\_MAP\_ALL\_ACCESS, 0, 0, TAM\_MEM)) == NULL)

{

printf("No se creo la memoria compartida: (%i)\n", GetLastError());

CloseHandle(hArchMapeoHN);

exit(-1);

}

puts("Se creo la memoria com hijo\n");

aHN = shmHN;

for (i = 0; i < 10; i++)

{

for (j = 0; j < 10; j++)

{

\*aHN = rand() % 15 \* 7;

mandada1[i][j] = \*aHN;

\*aHN++;

}

}

for (i = 0; i < 10; i++)

{

for (j = 0; j < 10; j++)

{

\*aHN = rand() % 15 \* 8;

mandada2[i][j] = \*aHN;

\*aHN++;

}

}

\*aHN = 101;

while (\*shmHN != -1){

printf("Esperando Hijo->Nieto");

sleep(1);}

WaitForSingleObject(hSemaforo,INFINITE);

printf("2 MATRICES. HIJO -> NIETO. HIJO.\nMatriz 1.\n");

for (i = 0; i < 10; i++)

{

for (j = 0; j < 10; j++)

{

printf("%d\t", mandada1[i][j]);

}

printf("\n");

}

printf("Matriz 2.\n");

for (i = 0; i < 10; i++)

{

for (j = 0; j < 10; j++)

{

printf("%d\t", mandada2[i][j]);

}

printf("\n");

}

UnmapViewOfFile(shmHN);

CloseHandle(hArchMapeoHN);

if(!ReleaseSemaphore(hSemaforo,1,NULL)){

printf("Falla al invocar ReleaseSemaphore: %d\n",GetLastError());

}

if((hSemaforo=OpenSemaphore(SEMAPHORE\_ALL\_ACCESS,TRUE,"Semaforo"))==NULL){

printf("Falla al invocar OpenSemaphore:%d\n",GetLastError());

return -1;

}

//RECIBE MATRIZ DEL PADRE

if ((hArchMapeoPH = OpenFileMapping(FILE\_MAP\_ALL\_ACCESS, FALSE, PH)) == NULL)

{

printf("No se ario archsadfadsfdfdfdivo de mapeo de la memoria: (%i)\n", GetLastError());

exit(-1);

}

if ((shmPH = (int \*)MapViewOfFile(hArchMapeoPH, FILE\_MAP\_ALL\_ACCESS, 0, 0, TAM\_MEM)) == NULL)

{

printf("No se accedio a la memoria compartida: (%i)\n", GetLastError());

CloseHandle(hArchMapeoPH);

exit(-1);

}

aPH = shmPH;

for (i = 0; i < 10; i++)

{

for (j = 0; j < 10; j++)

{

A[i][j] = \*aPH;

aPH++;

}

}

for (i = 0; i < 10; i++)

{

for (j = 0; j < 10; j++)

{

B[i][j] = \*aPH;

aPH++;

}

}

\*shmPH = -1;

printf("2 MATRICES. PADRE -> HIJO. HIJO.\nMatriz 1\n");

for (i = 0; i < 10; i++)

{

for (j = 0; j < 10; j++)

{

printf("%d\t", A[i][j]);

}

printf("\n");

}

printf("\nMatriz 2\n");

for (i = 0; i < 10; i++)

{

for (j = 0; j < 10; j++)

{

printf("%d\t", B[i][j]);

}

printf("\n");

}

printf("\n");

UnmapViewOfFile(shmPH);

CloseHandle(hArchMapeoPH);

//HACE EL PRODUCTO

for (i = 0; i<10; i++)

{

for (j = 0; j<10; j++)

{

aux = 0;

suma = 0;

while (aux<10)

{

suma += A[i][aux] \* B[aux][j];

aux++;

}

producto[i][j] = suma;

}

}

//MANDA MATRIZ AL PADRE

if ((hArchMapeoHP = CreateFileMapping(INVALID\_HANDLE\_VALUE, NULL, PAGE\_READWRITE, 0, TAM\_MEM, HP)) == NULL)

{

printf("No se mapeo la memoria compartida: (%i)\n", GetLastError());

exit(-1);

}

if ((shmHP = (int \*)MapViewOfFile(hArchMapeoHP, FILE\_MAP\_ALL\_ACCESS, 0, 0, TAM\_MEM)) == NULL)

{

printf("No se creo la memoria compartida: (%i)\n", GetLastError());

CloseHandle(hArchMapeoHP);

exit(-1);

}

aHP = shmHP;

for (i = 0; i < 10; i++)

{

for (j = 0; j < 10; j++)

{

\*aHP = producto[i][j];

\*aHP++;

}

}

printf("PRODUCTO. HIJO -> PADRE. HIJO.\n");

for (i = 0; i < 10; i++)

{

for (j = 0; j < 10; j++)

{

printf("%d\t", producto[i][j]);

}

printf("\n");

}

printf("\n");

\*aHP = 101;

while (\*shmHP != -1){puts("Esperando Hijo->Padre");

sleep(1);}

WaitForSingleObject(hSemaforo,INFINITE);

if(!ReleaseSemaphore(hSemaforo,1,NULL)){

printf("Falla al invocar ReleaseSemaphore:%d\n",GetLastError());

}

printf("El hijo libera al semaforo\n");

UnmapViewOfFile(shmHP);

CloseHandle(hArchMapeoHP);

CloseHandle(piH.hProcess);

CloseHandle(piH.hThread);

exit(0);//break;

}

“Prueba3.c”

#include <stdio.h>

#include <stdlib.h>

#include <windows.h>

#include <time.h>

#define TAM\_MEM 400

int main(int argc, char \*argv[])

{

int A[10][10], B[10][10], b[10][10], suma[10][10], d[10][10], mandada1[10][10], mandada2[10][10];

char \*NP = "NietoPadre";

char \*HN = "HijoNieto";

HANDLE hArchMapeoNP, hArchMapeoHN,hSemaforo;

int i, j, k, shmid,ct=0;

int \*aHN, \*aNP;

srand(time(NULL));

int \*shmNP, \*shmHN;

if((hSemaforo=OpenSemaphore(SEMAPHORE\_ALL\_ACCESS,TRUE,"Semaforo"))==NULL){

printf("Falla al invocar OpenSemaphore nieto:%d\n",GetLastError());

}

//RECIBE MATRIZ DEL HIJO

if ((hArchMapeoHN = OpenFileMapping(FILE\_MAP\_ALL\_ACCESS, TRUE, HN)) == NULL)

{

printf("No se abrio en el nieto archivo de mapeo de la memoria: (%i)\n", GetLastError());

exit(-1);

}

if ((shmHN = (int \*)MapViewOfFile(hArchMapeoHN, FILE\_MAP\_ALL\_ACCESS, 0, 0, TAM\_MEM)) == NULL)

{

printf("No se accedio en el nieto a la memoria compartida: (%i)\n", GetLastError());

CloseHandle(hArchMapeoHN);

exit(-1);

}

aHN = shmHN;

for (i = 0; i < 10; i++)

{

for (j = 0; j < 10; j++)

{

A[i][j] = \*aHN;

aHN++;

}

}

for (i = 0; i < 10; i++)

{

for (j = 0; j < 10; j++)

{

B[i][j] = \*aHN;

aHN++;

}

}

\*shmHN = -1;

printf("2 MATRICES. HIJO -> NIETO. NIETO.\nMatriz 1\n");

for (i = 0; i < 10; i++)

{

for (j = 0; j < 10; j++)

{

printf("%d\t", A[i][j]);

}

printf("\n");

}

printf("\nMatriz 2\n");

for (i = 0; i < 10; i++)

{

for (j = 0; j < 10; j++)

{

printf("%d\t", B[i][j]);

}

printf("\n");

}

printf("\n");

UnmapViewOfFile(shmHN);

CloseHandle(hArchMapeoHN);

//HACE LA SUMA

for (i = 0; i<10; i++)

{

for (j = 0; j<10; j++)

{

suma[i][j] = A[i][j] + B[i][j];

}

}

//MANDA MATRIZ AL PADRE

if ((hArchMapeoNP = CreateFileMapping(INVALID\_HANDLE\_VALUE, NULL, PAGE\_READWRITE, 0, TAM\_MEM, NP)) == NULL)

{

printf("No se mapeo la memoria compartida en el nieto: (%i)\n", GetLastError());

exit(-1);

}

if ((shmNP = (int \*)MapViewOfFile(hArchMapeoNP, FILE\_MAP\_ALL\_ACCESS, 0, 0, TAM\_MEM)) == NULL)

{

printf("No se creo la memoria compartida en el nieto: (%i)\n", GetLastError());

CloseHandle(hArchMapeoNP);

exit(-1);

}

aNP = shmNP;

for (i = 0; i < 10; i++)

{

for (j = 0; j < 10; j++)

{

\*aNP = suma[i][j];

\*aNP++;

}

}

printf("SUMA. NIETO -> PADRE. NIETO.\n");

for (i = 0; i < 10; i++)

{

for (j = 0; j < 10; j++)

{

printf("%d\t", suma[i][j]);

}

printf("\n");

}

printf("\n");

\*aNP = 101;

while (\*shmNP != -1){

puts("Esperando Nieto->Padre");

sleep(1);}

WaitForSingleObject(hSemaforo,INFINITE);

UnmapViewOfFile(shmNP);

CloseHandle(hArchMapeoNP);

if(!ReleaseSemaphore(hSemaforo,1,NULL)){

printf("Falla al invocar ReleaseSemaphore:%d\n",GetLastError());

}

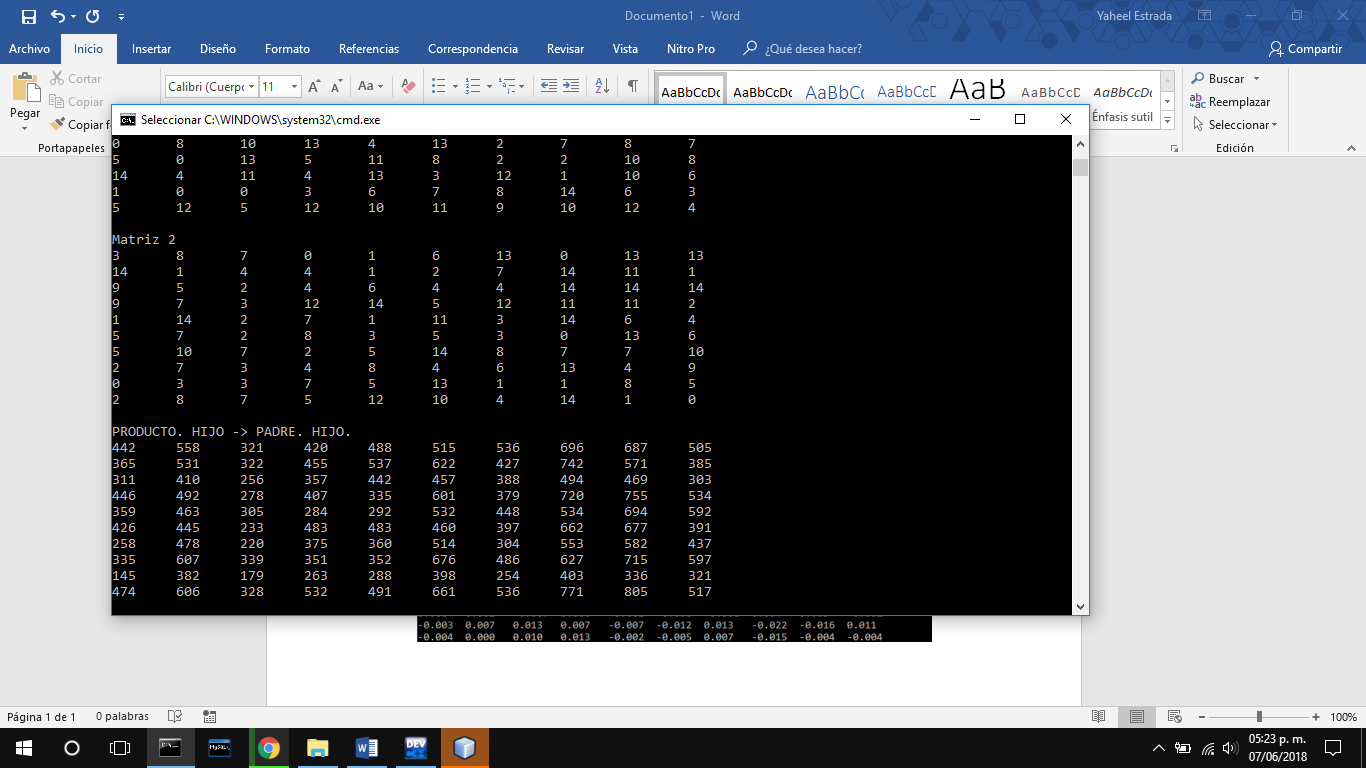
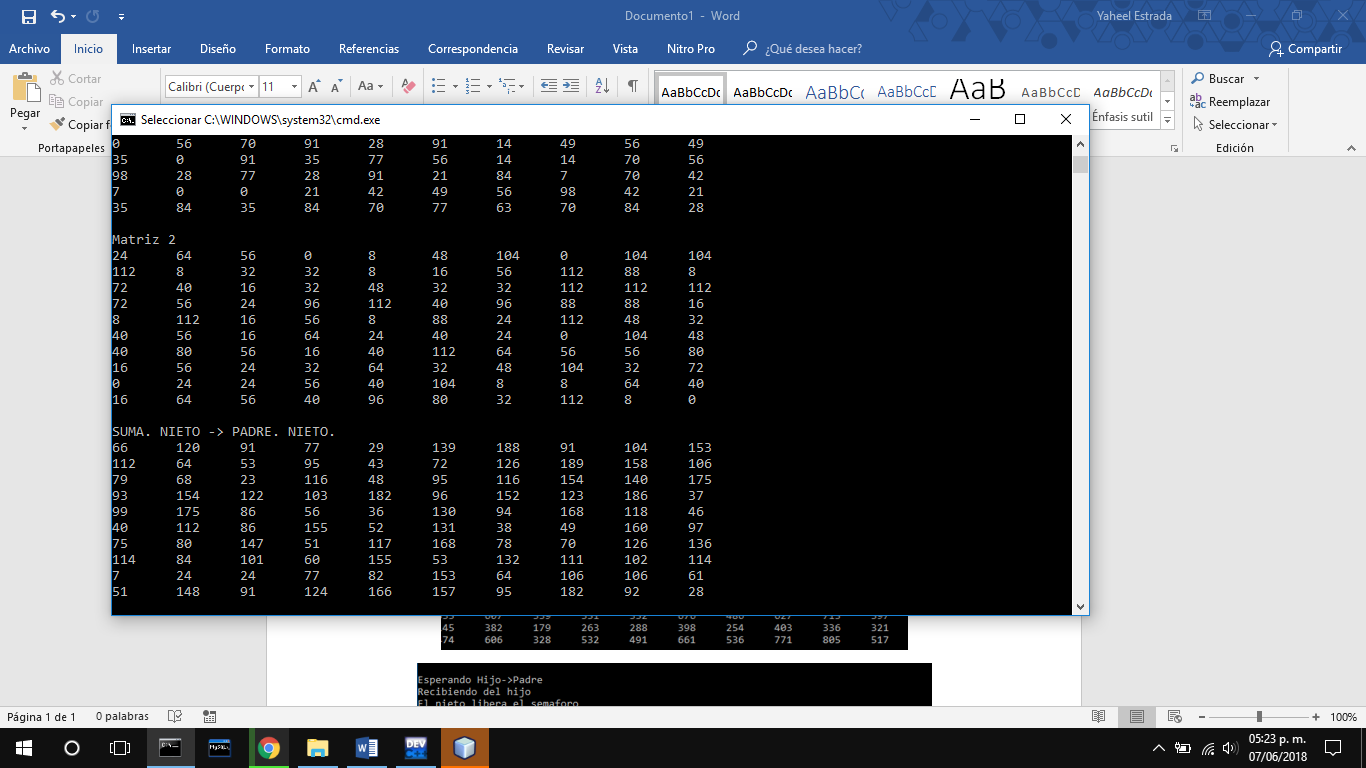
printf("El nieto libera el semaforo\n");

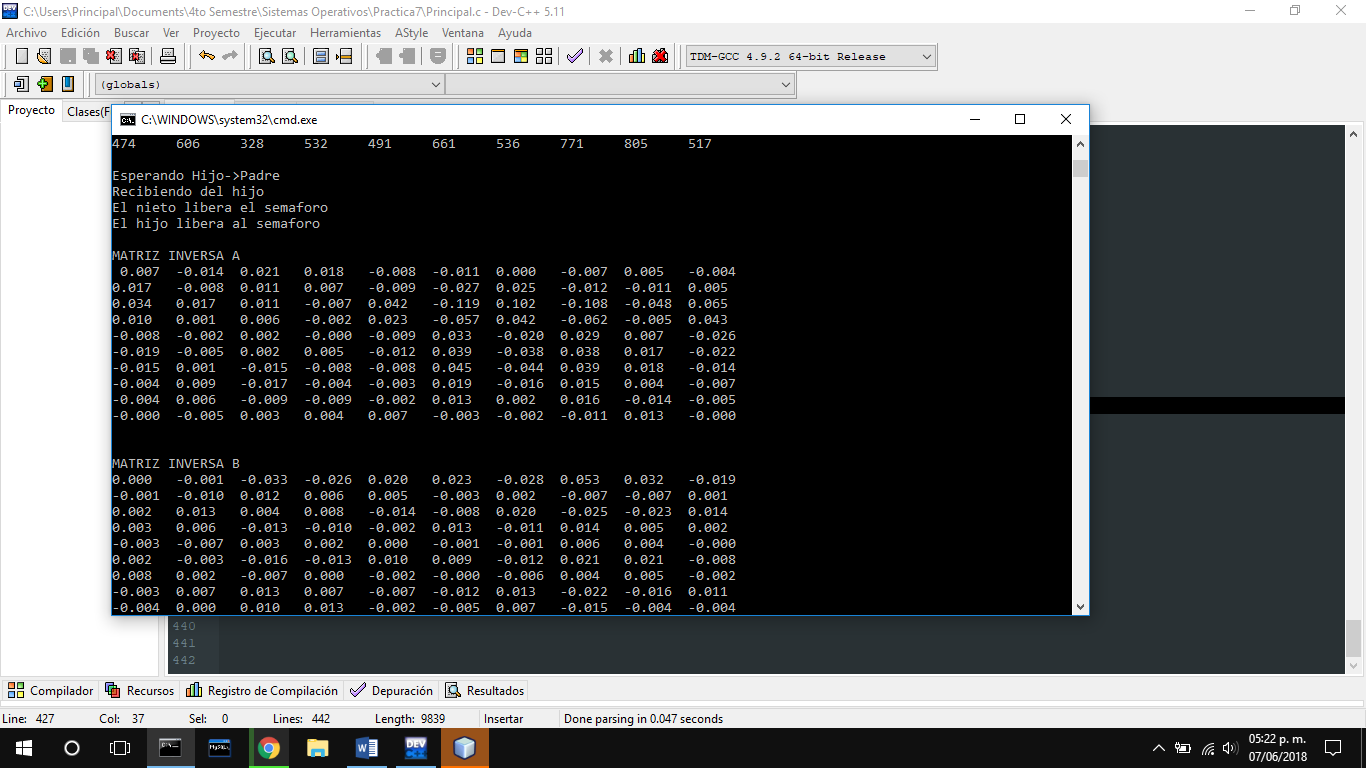
Sleep(3000);

exit(0);

}

**Funcionamiento.**





**Observaciones.**

Los semáforos son más prácticos de utilizar que otros mecanismos vistos anteriormente como las tuberías o los procesos tradicionales; sin embargo son un poco más complejos al momento de progrmarlos.

Como los hilos y las tuberías, los semáforos cuentan con sus propias funciones tanto en Linux como en Windows, las cuales permiten realizar operaciones que van desde abrir hasta bloquear o inicializar.

**Análisis Crítico.**

Con la elaboración de esta práctica pudimos entender la utilidad de los semáforos para controlar la inter comunicación de procesos evitando que los procesos interfieran unos con otros. Asimismo, pudimos observar la complejidad que implica combinar algunas de las técnicas de inter comunicación como lo es la memoria compartida con los semáforos.

En esta práctica notamos que la programación es más sencilla en el sistema operativo de Windows, ya que se utilizan menos funciones que las que ocupamos en el sistema operativo de Linux.

**Conclusiones.**

Los semáforos son mecanismos de inter comunicación con los que no se mueven los datos, esto es posible debido a que solo se puede consultar y modificar su valor al tener un carácter únicamente informativo.

De acuerdo a lo visto en la práctica, un semáforo puede verse como una especie de variable positiva o nula sobre la cual se realizan solo dos operaciones: wait, que permite adquirir el semáforo o bloquearlo al decrementar su valor cuando éste vale más que 0; y signal, que viene siendo la operación contraria, es decir, libera o inicializa al semáforo.

Estas operaciones son procedimientos que se implementan como acciones indivisibles. En sistemas con un único procesador bastará simplemente con inhibir las interrupciones durante la ejecución de las operaciones del semáforo.