Instituto Politécnico Nacional

Escuela Superior de Cómputo

Sistemas Operativos

“Práctica 6. Mecanismos de sincronización de procesos en Linux y Windows (semáforos)”

Grupo: 2CM8

Integrantes:

- Martínez Coronel Brayan Yosafat.
- Monteros Cervantes Miguel Angel.
- Ramírez Olvera Guillermo.
- Sánchez Méndez Edmundo Josue.

Profesor: Cortés Galicia Jorge

ESCOM

Práctica 5. Mecanismos de sincronización de procesos en Linux y Windows (semáforos)

Introducción

Un **semáforo** es una variable especial (o tipo abstracto de datos) que constituye el método clásico para restringir o permitir el acceso a recursos compartidos (por ejemplo, un recurso de almacenamiento del sistema o variables del código fuente) en un entorno de multiprocesamiento (en el que se ejecutarán varios procesos concurrentemente). Fueron inventados por Edsger Dijkstra en 1965 y se usaron por primera vez en el sistema operativo THEOS.

Los semáforos se emplean para permitir el acceso a diferentes partes de programas (llamados **secciones críticas**) donde se manipulan variables o recursos que deben ser accedidos de forma especial. Según el valor con que son inicializados se permiten a más o menos procesos utilizar el recurso de forma simultánea.

Un tipo simple de semáforo es el **binario**, que puede tomar solamente los valores 0 y 1. Se inicializan en 1 y son usados cuando sólo un proceso puede acceder a un recurso a la vez. Son esencialmente lo mismo que los mutex. Cuando el recurso está disponible, un proceso accede y decrementa el valor del semáforo con la operación P. El valor queda entonces en 0, lo que hace que si otro proceso intenta decrementarlo tenga que esperar. Cuando el proceso que decrementó el semáforo realiza una operación *V*, algún proceso que estaba esperando comienza a utilizar el recurso.

Para hacer que dos procesos se ejecuten en una secuencia predeterminada puede usarse un semáforo inicializado en 0. El proceso que debe ejecutar primero en la secuencia realiza la operación *V* sobre el semáforo antes del código que debe ser ejecutado después del otro proceso. Éste ejecuta la operación *P*. Si el segundo proceso en la secuencia es programado para ejecutar antes que el otro, al hacer *P* dormirá hasta que el primer proceso de la secuencia pase por su operación *V*. Este modo de uso se denomina señalación (*signaling*), y se usa para que un proceso o hilo de ejecución le haga saber a otro que algo ha sucedido.

Tipos de semáforos

Dependiendo de la función que cumple el semáforo, vamos a diferenciar los siguientes tipos:

- Semáforo de exclusión mutua, inicialmente su contador vale 1 y permite que haya un único proceso simultáneamente dentro de la sección crítica.
- Semáforo contador, permiten llevar la cuenta del número de unidades de recurso compartido disponible, que va desde 0 hasta N.
- Semáforo de espera, generalmente se emplea para forzar que un proceso pase a estado bloqueado hasta que se cumpla la condición que le permite

ejecutarse. Por lo general, el contador vale 0 inicialmente, no obstante, podría tener un valor distinto de cero.

Ventajas e inconvenientes

La principal ventaja de los semáforos frente a los cerrojos es que permiten sincronizar dos o más procesos de manera que no se desperdician recursos de CPU realizando comprobaciones continuadas de la condición que permite progresar al proceso. Los inconvenientes asociados al uso de semáforos son los siguientes:

- Los programadores tienden a usarlos incorrectamente, de manera que no resuelven de manera adecuada el problema de concurrencia o dan lugar a interbloqueos.
- No hay nada que obligue a los programadores a usarlos.
- Los compiladores no ofrecen ningún mecanismo de comprobación sobre el correcto uso de los semáforos.
- Son independientes del recurso compartido al que se asocian.

Debido a lo anteriormente mencionado se desarrollaron los monitores.

Granularidad

Número de recursos controlados por cada semáforo. Hay de dos tipos:

- Granularidad fina: Un recurso por semáforo. Hay un mayor grado de paralelismo, lo que puede mejorar la rapidez de ejecución de la protección. Aunque a mayor número de semáforos existe una mayor probabilidad de que se dé un interbloqueo entre semáforos, que no sería una mejora de lo que intentamos evitar.
- Granularidad gruesa: Un semáforo para todos los recursos. Caso totalmente inverso al anterior: Ahora al tener un semáforo no se produce interbloqueo entre ellos, aunque los tiempos de ejecución son excesivamente largos.

1. Competencias.

El alumno comprende el funcionamiento de los mecanismos de sincronización entre procesos cooperativos utilizando los semáforos como árbitros de acceso para el desarrollo de aplicaciones cooperativas tanto en el sistema operativo Linux como Windows.

2. Desarrollo

2.1. Sección Linux

2.1.1. Información de las llamadas al sistema

2.1.1.1. semget()

```
01. #include <sys/types.h>
02. #include <sys/ipc.h>
03. #include <sys/sem.h>
04.
05. int semget(key_t key, int nsems, int semflg);
```

La llamada al sistema **semget()** retornará un número no negativo que será el ID del semáforo asociado al argumento ***key*** de la función o -1 si es que existe un error. Argumentos pasados a la función:

- **key_t key**: Es un tipo de identificador de grupos de semáforos. Primero se compara si es que ya existe este valor dentro del kernel para los conjuntos de semáforos. La operación de apertura o acceso depende del contenido de semflg.
- **int nsems**: Especifica el número de semáforos que se crearán en el conjunto.
- **int semflg**: Son tipo de opciones de qué forma se crearán el nuevo grupo de semáforos:
    - **IPC_CREAT**: Crea los semáforos si es que el valor de ***key*** no existe previamente en el kernel, o devuelve el ID de los semáforos con el valor de key.
    - **IPC_EXCL**: Crea los semáforos si es que el valor de ***key*** no existe previamente en el kernel, o devuelve -1 si es que ya existe el valor de ***key*** en el kernel.

2.1.1.2. semop()

```
01. #include <sys/types.h>
02. #include <sys/ipc.h>
03. #include <sys/sem.h>
04.
05. int semop(int semid, struct sembuf *sops, size_t nsops);
```

La llamada al sistema **semop()** retornara 0 en caso de éxito o -1 si llega a ocurrir algún error. Argumentos de la funcion:

- int semid: Es el ID del grupo de semáforos a los que se les aplicara la operación especificada en **sops**.

- struct sembuf *sops: Es donde se especifica la operación de a qué semáforo se le aplicara y las banderas que se ocuparan.

```
struct sembuf {
        short sem_num;
        short sem_op;
        short sem_flg;
};
```

    - short sem_num: El número de semáforo que se manipulara.
    - short sem_op: La operación de semáforo que se ocupara.
    - short sem_flg: Banderas de operaciones.
- size_t nsops: Especifica el número de estructuras sembuf.

2.1.2. Ejemplo de semáforos en Linux

Código(EjemploSemaforo.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.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*sizeof (struct sembuf));
    printf("Iniciando semaforo...\n");
    if ((semid = semget (llave, nsems, semban)) == -1){
        perror ("semget: error al iniciar el semaforo");
        exit (1);
    }
    else
    printf("Semaforo iniciado...\n");
    if (semctl (semid, 0, SETVAL, 0) == -1){
      printf ("Error al crear el semaforo\n");
    }
    else{
    printf ("Semaforo creado...\n");
    }
    if ((pid = fork ()) < 0){
        perror ("fork: error al crear el 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 operacion del semaforo\n");
            }
            else{
                printf ("\tsemop: regreso de semop () %d\n", j);
                printf("\n\nProceso 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 la 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 ("semop: Padre 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 operacion del semaforo\n");
                }
                else{
                    printf ("\tsemop: 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 regreso de semop ()\n");
                    }
                    else
                        printf ("Proceso padre regresa el control del semaforo: %d/3 veces\n", i + 1);
                        sleep (5);
                    }
                    ++i;
            }
        }
}
```

Compilación y ejecución del programa:

```
edmundojsm@edmundojsm-VB:~/Escritorio/Sistemas Operativos/Practica 6$ gcc EjemploSemaforo.c -o EjSemaforo
edmundojsm@edmundojsm-VB:~/Escritorio/Sistemas Operativos/Practica 6$ ./EjSemaforo
Iniciando semaforo...
Semaforo iniciado...
Semaforo creado...
semop: Padre llamando a semop (0, &sops, 2) con:
        sops [0].sem_num = 0,sem_op 0,sem_flg = 010000

        sops [1].sem_num = 0,sem_op 1,sem_flg = 014000
        semop: regreso de semop () 0
Proceso padre toma el control del semaforo: 1/3 veces
semop: Hijo llamando a semop (0, &sops, 2) con:
        sops [0].sem_num = 0,sem_op 0, sem_flg = 010000

        sops [1].sem_num = 0,sem_op 1, sem_flg = 014000
Proceso padre regresa el control del semaforo: 1/3 veces
        semop: regreso de semop () 0


Proceso hijo toma el control del semaforo: 1/3 veces
semop: Padre llamando a semop (0, &sops, 2) con:
        sops [0].sem_num = 0,sem_op 0,sem_flg = 010000

        sops [1].sem_num = 0,sem_op 1,sem_flg = 014000
Proceso hijo regresa el control del semaforo: 1/3 veces
        semop: regreso de semop () 0
Proceso padre toma el control del semaforo: 2/3 veces
Proceso padre regresa el control del semaforo: 2/3 veces
semop: Hijo llamando a semop (0, &sops, 2) con:
        sops [0].sem_num = 0,sem_op 0, sem_flg = 010000

        sops [1].sem_num = 0,sem_op 1, sem_flg = 014000
        semop: regreso de semop () 0


Proceso hijo toma el control del semaforo: 2/3 veces
semop: Padre llamando a semop (0, &sops, 2) con:
        sops [0].sem_num = 0,sem_op 0,sem_flg = 010000

        sops [1].sem_num = 0,sem_op 1,sem_flg = 014000
Proceso hijo regresa el control del semaforo: 2/3 veces
        semop: regreso de semop () 0
Proceso padre toma el control del semaforo: 3/3 veces
Proceso padre regresa el control del semaforo: 3/3 veces
```

```
semop: Hijo llamando a semop (0, &sops, 2) con:
        sops [0].sem_num = 0,sem_op 0, sem_flg = 010000

        sops [1].sem_num = 0,sem_op 1, sem_flg = 014000
        semop: regreso de semop () 0


Proceso hijo toma el control del semaforo: 3/3 veces
edmundojsm@edmundojsm-VB:~/Escritorio/Sistemas Operativos/Practica 6$ Proceso hijo regresa el control del semaforo: 3/3 veces
```

Observamos la manipulación de semáforos con las llamadas al sistema **semget()** para obtener el identificador para el semáforo, la estructura **sembuf** que tendrá la información del semáforo, **semctl()** creará el semáforo y **semop()** para la sincronización del semáforo de encendido (Semáforo en verde) y apagado (Semáforo en rojo). Vemos como por medio de los while que contienen la llamada al sistema **semop()** van permitiendo la sincronización de la información a la cual se tiene acceso una vez que el semáforo está en verde() y se mueve al otro proceso cuando el semáforo se pone en rojo(). Observamos como uno se apaga y el otro se prende y viceversa para dar acceso a la información que esta después del apagado del semáforo.

2.1.3. Programa con semáforos en Linux (Usando Matrices)

Código(SemaforoMatriz.c)

```c
#include <sys/shm.h>
#include <sys/wait.h>
#include <unistd.h>
#include <sys/sem.h>
#include <stdlib.h>
#include <stdio.h>
#include "FuncionesSemaforos.h"

int main(){
    int semaforo [2];
    Matriz A, B, C, D, E, F;
    double *AuxMemoria1, *AuxMemoria2;
    key_t llave1 = 1000;
    key_t llave2 = 2000;

    AuxMemoria1 = IniciarMemoria(llave1);
    AuxMemoria2 = IniciarMemoria(llave2);
    semaforo [0] = IniciarSemaforo();
    semaforo [1] = IniciarSemaforo();

    if(fork() == 0){
        if(fork() == 0){
            D = MemoriatoMatriz(AuxMemoria1);
            E = MemoriatoMatriz(AuxMemoria2);
            F = Suma(D, E);
            EscribirMemoria(AuxMemoria1, F);
            EncenderSemaforo(semaforo [1]);
        }else{
            A = MemoriatoMatriz(AuxMemoria1);
            B = MemoriatoMatriz(AuxMemoria2);
            C = Multiplicacion(A, B);

            D = GeneradorMatriz();
            E = GeneradorMatriz();
            EscribirMemoria(AuxMemoria1, D);
            EscribirMemoria(AuxMemoria2, E);
            ApagarSemaforo(semaforo [1]);
            EscribirMemoria(AuxMemoria2, C);
            EncenderSemaforo(semaforo [0]);
        }
    }else{
        A = GeneradorMatriz();
        B = GeneradorMatriz();

        EscribirMemoria(AuxMemoria1, A);
        EscribirMemoria(AuxMemoria2, B);

        ApagarSemaforo(semaforo [0]);
        printf("\nInversa de la multiplicacion de dos matrices\n");
        C = Inversa(MemoriatoMatriz(AuxMemoria2));
        EscribirMatriz(C, "inversaMultiplicacion.txt");
        printMatriz(C);

        printf("\nInversa de la suma de dos matrices\n");
        F = Inversa(MemoriatoMatriz(AuxMemoria1));
        EscribirMatriz(F, "inversaSuma.txt");
        printMatriz(F);
    }
    exit(0);
}
```

Código(FuncionesSemaforos.h)

```c
#include "EstructuraMatriz.h"
#define TAM_MEM 400

double *IniciarMemoria(key_t llave){
    int shmid;
    double *shm;
    if((shmid = shmget(llave, TAM_MEM, IPC_CREAT|0666)) < 0){
        perror("Error al obtener memoria compartida: shmget");
        exit(-1);
    }
    if((shm = shmat(shmid, NULL, 0)) == (double *)-1){
        perror("Error al enlazar la memoria compartida: shmat");
        exit(-1);
    }
    return shm;
}

void EscribirMemoria(double *shm, Matriz A){
    for(int i = 0; i < N; i++)
        for(int k = 0; k < N; k++)
            *shm++ = A[i][k];
}

Matriz MemoriatoMatriz(double *shm){
    Matriz A = NuevaMatriz();
    for(int i = 0; i < N; i++)
        for(int k = 0; k < N; k++)
            A[i][k] = *shm++;
    return A;
}
int IniciarSemaforo(){
    int semid;
    int nsems = 1;
    key_t llave = ftok("/bin/ls", 1234);
    if((semid = semget(llave, nsems, IPC_CREAT | 0666)) == -1){
        perror("semget: error al iniciar el semaforo");
        exit(1);
    }
    if(semctl(semid, 0, SETVAL, 0) == -1){
        printf("Error al crear el semaforo\n");
    }
    return semid;
}

void EncenderSemaforo(int semid){
    struct sembuf sops;
    sops.sem_num = 0;
    sops.sem_op = 1;
    sops.sem_flg = 0;
    if(semop(semid, &sops, 1) == -1){
        perror("semop: error en operacion del semaforo\n");
    }
}

void ApagarSemaforo(int semid){
    struct sembuf sops;
    sops.sem_num = 0;
    sops.sem_op = -1;
    sops.sem_flg = 0;
    if(semop(semid, &sops, 1) == -1){
        perror ("semop: error en regreso de semop ()\n");
    }
}
```

Código(EstructuraMatriz.h)

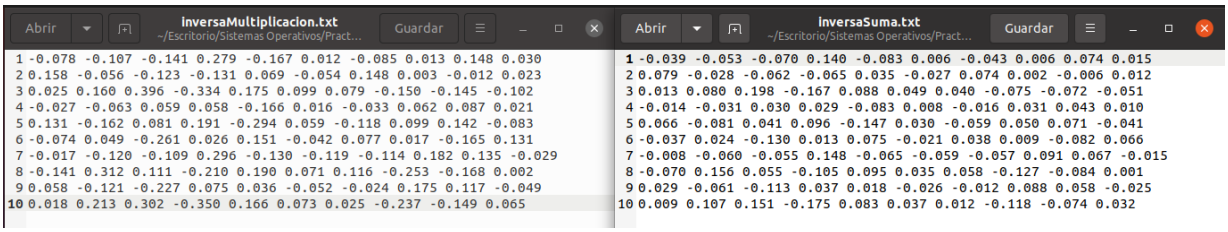
```
#include <stdbool.h>
#include <math.h>
#include <time.h>
typedef double* Vector;
typedef Vector* Matriz;
#define N 10

Matriz NuevaMatriz(){
    int M = N;
    if(N&1)
        M+=1;
    Matriz A = calloc(M, sizeof(Vector));
    for(int i = 0; i < N; ++i)
        A[i] = calloc(M, sizeof(double));
    return A;
}

bool esCero(double x){
    return fabs(x) < 1e-8;
}

Matriz Suma(Matriz A, Matriz B){
    Matriz C = NuevaMatriz();
    for(int i = 0; i < N; ++i)
        for(int j = 0; j < N; ++j)
            C[i][j] = A[i][j] + B[i][j];
    return C;
}


Matriz Multiplicacion(Matriz A, Matriz B){
    Matriz C = NuevaMatriz();
    for(int i = 0; i < N; ++i)
        for(int j = 0; j < N; ++j)
            for(int k = 0; k < N; ++k)
                C[i][j] += A[i][k] * B[k][j];
    return C;
}

Matriz Inversa(Matriz A){
    Matriz inv = NuevaMatriz();
    for(int i = 0; i < N; ++i)
        inv[i][i] = 1;
    int i = 0, j = 0;
    while(i < N && j < N){
        if(esCero(A[i][j])){
            for(int k = i + 1; k < N; ++k){
                if(!esCero(A[k][j])){
                    Vector tmp = A[i];
                    A[i] = A[k];
                    A[k] = tmp;
                    tmp = inv[i];
                    inv[i] = inv[k];
                    inv[k] = tmp;
                    break;
                }
            }
        }
        if(!esCero(A[i][j])){
            for(int l = 0; l < N; ++l)
                inv[i][l] /= A[i][j];
            for(int l = N - 1; l >= j; --l)
                A[i][l] /= A[i][j];
            for(int k = 0; k < N; ++k){
                if(i == k) continue;
                for(int l = 0; l < N; ++l)
                    inv[k][l] -= inv[i][l] * A[k][j];
                for(int l = N; l >= j; --l)
                    A[k][l] -= A[i][l] * A[k][j];
            }
            ++i;
        }
        ++j;
    }
    return inv;
}

Matriz GeneradorMatriz(){
    srand(time(NULL));
    Matriz A = NuevaMatriz();
    for(int i = 0; i < N; ++i)
        for(int j = 0; j < N; ++j)
            A[i][j] = rand() % 10;
    return A;
}

void EscribirMatriz(Matriz A, char* nombre){
    FILE * fp = fopen(nombre, "w");
    for(int i = 0; i < N; ++i){
        for(int j = 0; j < N; ++j)
            fprintf(fp, "%0.3lf ", A[i][j]);
        fprintf(fp, "\n");
    }
    fclose(fp);
}

Matriz LeerMatriz(char* nombre){
    Matriz A = NuevaMatriz();
    FILE * fp = fopen(nombre, "r");
    for(int i = 0; i < N; ++i)
        for(int j = 0; j < N; ++j)
            fscanf(fp, "%lf", &A[i][j]);
    fclose(fp);
    return A;
}

void printMatriz(Matriz A){
    for(int i = 0; i < N; ++i){
        for(int j = 0; j < N; ++j)
            printf("%0.3lf\t", A[i][j]);
        printf("\n");
    }
}
```

Compilación y ejecución del programa:

```
edmundojsm@edmundojsm-VB:~/Escritorio/Sistemas Operativos/Practica 6$ gcc SemaforoMatriz.c -o SemaforoMatriz
edmundojsm@edmundojsm-VB:~/Escritorio/Sistemas Operativos/Practica 6$ ./SemaforoMatriz

Inversa de la multiplicacion de dos matrices
-0.078  -0.107  -0.141  0.279   -0.167  0.012   -0.085  0.013   0.148   0.030
0.158   -0.056  -0.123  -0.131  0.069   -0.054  0.148   0.003   -0.012  0.023
0.025   0.160   0.396   -0.334  0.175   0.099   0.079   -0.150  -0.145  -0.102
-0.027  -0.063  0.059   0.058   -0.166  0.016   -0.033  0.062   0.087   0.021
0.131   -0.162  0.081   0.191   -0.294  0.059   -0.118  0.099   0.142   -0.083
-0.074  0.049   -0.261  0.026   0.151   -0.042  0.077   0.017   -0.165  0.131
-0.017  -0.120  -0.109  0.296   -0.130  -0.119  -0.114  0.182   0.135   -0.029
-0.141  0.312   0.111   -0.210  0.190   0.071   0.116   -0.253  -0.168  0.002
0.058   -0.121  -0.227  0.075   0.036   -0.052  -0.024  0.175   0.117   -0.049
0.018   0.213   0.302   -0.350  0.166   0.073   0.025   -0.237  -0.149  0.065

Inversa de la suma de dos matrices
-0.039  -0.053  -0.070  0.140   -0.083  0.006   -0.043  0.006   0.074   0.015
0.079   -0.028  -0.062  -0.065  0.035   -0.027  0.074   0.002   -0.006  0.012
0.013   0.080   0.198   -0.167  0.088   0.049   0.040   -0.075  -0.072  -0.051
-0.014  -0.031  0.030   0.029   -0.083  0.008   -0.016  0.031   0.043   0.010
0.066   -0.081  0.041   0.096   -0.147  0.030   -0.059  0.050   0.071   -0.041
-0.037  0.024   -0.130  0.013   0.075   -0.021  0.038   0.009   -0.082  0.066
-0.008  -0.060  -0.055  0.148   -0.065  -0.059  -0.057  0.091   0.067   -0.015
-0.070  0.156   0.055   -0.105  0.095   0.035   0.058   -0.127  -0.084  0.001
0.029   -0.061  -0.113  0.037   0.018   -0.026  -0.012  0.088   0.058   -0.025
0.009   0.107   0.151   -0.175  0.083   0.037   0.012   -0.118  -0.074  0.032
edmundojsm@edmundojsm-VB:~/Escritorio/Sistemas Operativos/Practica 6$
```

Archivos generados en la carpeta raíz.

Archivos generados exitosamente con el mismo contenido que el impreso.



Se observa cómo se envía dos matrices entre tres procesos, el primer apaga el primer semáforo el cual protegerá las funciones para obtener la inversa de la multiplicación y suma de las matrices que le enviaran su proceso hijo y nieto pero antes del apagado del semáforo el proceso le manda dos matrices a su proceso hijo a través de una memoria compartida y obtendrá la multiplicación entre ambas matrices y tendrá esta información protegida por el segundo semáforo junto con el encendido del primero semáforo y este segundo semáforo estaría apagado que a su vez mandara el resultado a su proceso padre por memoria compartida una vez que el segundo semáforo este encendido, después este hijo creara un hijo de él y le mandara dos matrices a su hijo a través de memoria compartida de las cuales obtendrá la suma de las dos matrices recibidas y mandara al primer proceso creado el resultado de la suma y encenderá el segundo semáforo. Finalmente, en el primer proceso será encendido el semáforo que contiene después de que el segundo

semáforo se enciende (Ya que este protege el encendido del primer semáforo) y entonces recuperara la información escrita en las memorias compartidas por parte del hijo y del nieto para obtener la matriz inversa de la matriz resultante de la multiplicación y de la suma. Donde escribir a ambas inversas resultantes en un archivo .txt distinto, la matriz inversa de la multiplicación con el nombre ”inversaMultiplicacion.txt” y la matriz inversa de la suma con el nombre ”inversaSuma.txt”. Se realiza la sincronización de los semáforos con las funciones EncenderSemaforo(), ApagarSemaforo() y se crean los semáforos con IniciarSemaforo().

2.2. Sección Windows

2.2.1. Ejemplo de semáforos en Windows

Código(EjSemaforoPadreW.c)

```c
#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; /* Estructura de información del adm. 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 Nombre_programa_hijo\n", argv[0]);
        return;
    }
    // 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 semáforo
        WaitForSingleObject(hSemaforo, INFINITE);

        //Sección crítica
        printf("Soy el padre entrando %i de 3 veces al semaforo\n", i);
        Sleep(5000);

        //Liberación el 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 proceso e hilo asociado de ejecución
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

Código(EjSemaforoHijoW.c)

```c
#include <windows.h> /*Programa hijo*/
#include <stdio.h>

int main(){
    HANDLE hSemaforo;
    int i = 1;

    // Apertura del semáforo
    if((hSemaforo = OpenSemaphore(SEMAPHORE_ALL_ACCESS, FALSE, "Semaforo")) ==NULL)
        printf("Falla al invocar OpenSemaphore: %d\n", GetLastError());
        return -1;
    }

    while (i < 4){
        // Prueba del semáforo
        WaitForSingleObject(hSemaforo, INFINITE);

        //Sección crítica
        printf("Soy el hijo entrando %i de 3 veces al semaforo\n", i);
        Sleep(5000);

        //Liberación el semáforo
        if (!ReleaseSemaphore(hSemaforo, 1, NULL)){
            printf("Falla al invocar ReleaseSemaphore: %d\n", GetLastError());
        }
        printf("Soy el hijo liberando %i de 3 veces al semaforo\n", i);
        Sleep(5000);
        i++;
    }
    return 0;
}
```

Compilación y ejecución del código.

```
C:\Windows\system32\cmd.exe

C:\Users\Edmundo J Sanchez M\Desktop\SO\P7>gcc EjSemaforoPadreW.c -o Padre

C:\Users\Edmundo J Sanchez M\Desktop\SO\P7>gcc EjSemaforoHijoW.c -o Hijo

C:\Users\Edmundo J Sanchez M\Desktop\SO\P7>Padre.exe Hijo.exe
Soy el padre entrando 1 de 3 veces al semaforo
Soy el padre liberando 1 de 3 veces al semaforo
Soy el hijo entrando 1 de 3 veces al semaforo
Soy el hijo liberando 1 de 3 veces al semaforo
Soy el padre entrando 2 de 3 veces al semaforo
Soy el padre liberando 2 de 3 veces al semaforo
Soy el hijo entrando 2 de 3 veces al semaforo
Soy el hijo liberando 2 de 3 veces al semaforo
Soy el padre entrando 3 de 3 veces al semaforo
Soy el hijo entrando 3 de 3 veces al semaforo
Soy el padre liberando 3 de 3 veces al semaforo
Soy el hijo liberando 3 de 3 veces al semaforo

C:\Users\Edmundo J Sanchez M\Desktop\SO\P7>
```

Al igual que el programa ejemplo en Linux observamos la manipulación de semáforos como por medio con las llamadas al sistema para la sincronización del semáforo de encendido (Semáforo en verde) y apagado (Semáforo en rojo). Vemos de los while son los que van permitiendo la sincronización de la información a la cual se tiene acceso una vez que el semáforo está en verde() y se mueve al otro proceso cuando el semáforo se pone en rojo(). Observamos como uno se apaga y el otro se prende y viceversa para dar acceso a la información que esta después del apagado del semáforo.

2.2.2. Programa con semáforos en Windows (Usando Matrices)

Código(SemaforosMatriz.c)

```c
#include <windows.h>
#include <stdio.h>
#include <time.h>
#include "EstructuraMatriz.h"
#define TAM_MEM 400

void escribir(Matriz A, float *p){
    for(int i = 0; i < N; ++i)
        for(int j = 0; j < N; ++j)
            *p++ = A[i][j];
}

void leer(Matriz A, float *p){
    for(int i = 0; i < N; ++i)
        for(int j = 0; j < N; ++j)
            A[i][j] = *p++;
}

float *crearMemoria(char *idMem){
    HANDLE hMem = CreateFileMapping(INVALID_HANDLE_VALUE, NULL, PAGE_READWRITE, 0, TAM_MEM, idMem);
    return MapViewOfFile(hMem, FILE_MAP_ALL_ACCESS, 0, 0, TAM_MEM);
}

float *leerMemoria(char *idMem){
    HANDLE hMem = OpenFileMapping(FILE_MAP_ALL_ACCESS, FALSE, idMem);
    return MapViewOfFile(hMem, FILE_MAP_ALL_ACCESS, 0, 0, TAM_MEM);
}

void proceso(char *name, int nivel){
    STARTUPINFO si;
    PROCESS_INFORMATION pi;
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));
    char args[100];
    sprintf(args, "%s %d", name, nivel);
    CreateProcess(NULL, args, NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi);
}

HANDLE hSem[5];

void go(int i){
    srand(time(NULL));
    if(i == 0){
        float *p = crearMemoria("mem");
        Matriz A = GeneradorMatriz(), B = GeneradorMatriz();
        Matriz AB = NuevaMatriz(), C_D = NuevaMatriz();

```

```
49.          escribir(A, p);
50.          ReleaseSemaphore(hSem[0], 1, NULL);
51.          WaitForSingleObject(hSem[1], INFINITE);
52.
53.          escribir(B, p);
54.          ReleaseSemaphore(hSem[0], 1, NULL);
55.          WaitForSingleObject(hSem[1], INFINITE);
56.
57.          leer(AB, p);
58.          ReleaseSemaphore(hSem[0], 1, NULL);
59.
60.          WaitForSingleObject(hSem[4], INFINITE);
61.          leer(C_D, p);
62.          ReleaseSemaphore(hSem[0], 1, NULL);
63.
64.          Matriz AB_inv = Inversa(AB);
65.          Matriz C_D_inv = Inversa(C_D);
66.          EscribirMatriz(AB_inv, "inversaMultiplicacion.txt");
67.          EscribirMatriz(C_D_inv, "inversaSuma.txt");
68.          printf("Inversa de la multiplicacion de dos matrices:\n");
69.          printMatriz(AB_inv);
70.          printf("Inversa de la suma de dos matrices:\n");
71.          printMatriz(C_D_inv);
72.
73.      }else if(i == 1){
74.          float *p = leerMemoria("mem");
75.          Matriz A = NuevaMatriz(), B = NuevaMatriz();
76.          Matriz C = GeneradorMatriz(), D = GeneradorMatriz();
77.          C = GeneradorMatriz(), D = GeneradorMatriz();
78.
79.          WaitForSingleObject(hSem[0], INFINITE);
80.          leer(A, p);
81.          ReleaseSemaphore(hSem[1], 1, NULL);
82.
83.          WaitForSingleObject(hSem[0], INFINITE);
84.          leer(B, p);
85.
86.          Matriz AB = Multiplicacion(A, B);
87.          escribir(AB, p);
88.          ReleaseSemaphore(hSem[1], 1, NULL);
89.          WaitForSingleObject(hSem[0], INFINITE);
90.
91.          escribir(C, p);
92.          ReleaseSemaphore(hSem[2], 1, NULL);
93.          WaitForSingleObject(hSem[3], INFINITE);
94.
95.          escribir(D, p);
96.          ReleaseSemaphore(hSem[2], 1, NULL);
97.          WaitForSingleObject(hSem[3], INFINITE);
98.
```

```
    }else if(i == 2){
        float *p = leerMemoria("mem");
        Matriz C = NuevaMatriz();
        Matriz D = NuevaMatriz();

        WaitForSingleObject(hSem[2], INFINITE);
        leer(C, p);
        ReleaseSemaphore(hSem[3], 1, NULL);

        WaitForSingleObject(hSem[2], INFINITE);
        leer(D, p);

        Matriz C_D = Suma(C, D);
        escribir(C_D, p);
        ReleaseSemaphore(hSem[4], 1, NULL);
        WaitForSingleObject(hSem[0], INFINITE);
        ReleaseSemaphore(hSem[3], 1, NULL);
    }
}

void crearSems(int nivel){
    for(int i = 0; i < 5; ++i){
        char name[100];
        sprintf(name, "sem%d", i);
        if(nivel == 0) hSem[i] = CreateSemaphore(NULL, 0, 1, name);
        else hSem[i] = OpenSemaphore(SEMAPHORE_ALL_ACCESS, FALSE, name);
    }
}

int main(int argc, char *argv[]){
    srand(time(NULL));
    int nivel = 0; //0-padre, 1-hijo, 2-nieto
    if(argc > 1) sscanf(argv[1], "%d", &nivel);

    crearSems(nivel);
    if(nivel < 2) proceso(argv[0], nivel + 1);
    go(nivel);
    return 0;
}
```

Código(EstructuraMatriz.h)

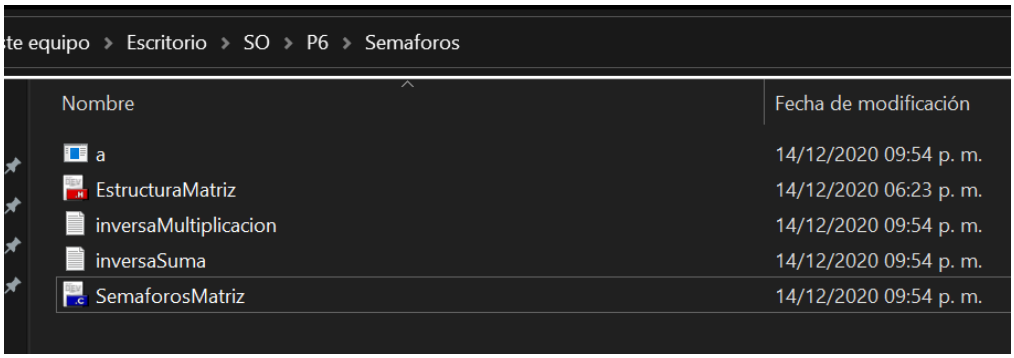
```
#include <stdbool.h>
#include <math.h>
#include <time.h>
typedef double* Vector;
typedef Vector* Matriz;
#define N 10

Matriz NuevaMatriz(){
    int M = N;
    if(N&1)
        M+=1;
    Matriz A = calloc(M, sizeof(Vector));
    for(int i = 0; i < N; ++i)
        A[i] = calloc(M, sizeof(double));
    return A;
}

bool esCero(double x){
    return fabs(x) < 1e-8;
}

Matriz Suma(Matriz A, Matriz B){
    Matriz C = NuevaMatriz();
    for(int i = 0; i < N; ++i)
        for(int j = 0; j < N; ++j)
            C[i][j] = A[i][j] + B[i][j];
    return C;
}


Matriz Multiplicacion(Matriz A, Matriz B){
    Matriz C = NuevaMatriz();
    for(int i = 0; i < N; ++i)
        for(int j = 0; j < N; ++j)
            for(int k = 0; k < N; ++k)
                C[i][j] += A[i][k] * B[k][j];
    return C;
}

Matriz Inversa(Matriz A){
    Matriz inv = NuevaMatriz();
    for(int i = 0; i < N; ++i)
        inv[i][i] = 1;
    int i = 0, j = 0;
    while(i < N && j < N){
        if(esCero(A[i][j])){
            for(int k = i + 1; k < N; ++k){
                if(!esCero(A[k][j])){
                    Vector tmp = A[i];
                    A[i] = A[k];
                    A[k] = tmp;
                    tmp = inv[i];
                    inv[i] = inv[k];
                    inv[k] = tmp;
                    break;
                }
            }
        }
        if(!esCero(A[i][j])){
            for(int l = 0; l < N; ++l)
                inv[i][l] /= A[i][j];
            for(int l = N - 1; l >= j; --l)
                A[i][l] /= A[i][j];
            for(int k = 0; k < N; ++k){
                if(i == k) continue;
                for(int l = 0; l < N; ++l)
                    inv[k][l] -= inv[i][l] * A[k][j];
                for(int l = N; l >= j; --l)
                    A[k][l] -= A[i][l] * A[k][j];
            }
            ++i;
        }
        ++j;
    }
    return inv;
}
```

```
78.  Matriz GeneradorMatriz(){
79.      srand(time(NULL));
80.      Matriz A = NuevaMatriz();
81.      for(int i = 0; i < N; ++i)
82.          for(int j = 0; j < N; ++j)
83.              A[i][j] = rand() % 10;
84.      return A;
85.  }
86.
87.  void EscribirMatriz(Matriz A, char* nombre){
88.      FILE * fp = fopen(nombre, "w");
89.      for(int i = 0; i < N; ++i){
90.          for(int j = 0; j < N; ++j)
91.              fprintf(fp, "%0.3lf ", A[i][j]);
92.          fprintf(fp, "\n");
93.      }
94.      fclose(fp);
95.  }
96.
97.  Matriz LeerMatriz(char* nombre){
98.      Matriz A = NuevaMatriz();
99.      FILE * fp = fopen(nombre, "r");
100.     for(int i = 0; i < N; ++i)
101.         for(int j = 0; j < N; ++j)
102.             fscanf(fp, "%lf", &A[i][j]);
103.     fclose(fp);
104.     return A;
105. }
106.
107. void printMatriz(Matriz A){
108.     for(int i = 0; i < N; ++i){
109.         for(int j = 0; j < N; ++j)
110.             printf("%0.3lf\t", A[i][j]);
111.         printf("\n");
112.     }
113. }
```

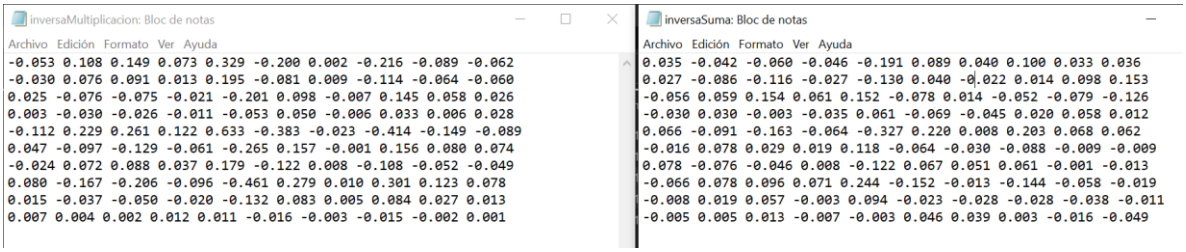
Compilación y ejecución del programa:

```
C:\Windows\system32\cmd.exe

C:\Users\Edmundo J Sanchez M\Desktop\SO\P6\Semaforos>gcc SemaforosMatriz.c

C:\Users\Edmundo J Sanchez M\Desktop\SO\P6\Semaforos>a.exe
Inversa de la multiplicacion de dos matrices:
-0.053  0.108   0.149   0.073   0.329   -0.200  0.002   -0.216  -0.089  -0.062
-0.030  0.076   0.091   0.013   0.195   -0.081  0.009   -0.114  -0.064  -0.060
0.025   -0.076  -0.075  -0.021  -0.201  0.098   -0.007  0.145   0.058   0.026
0.003   -0.030  -0.026  -0.011  -0.053  0.050   -0.006  0.033   0.006   0.028
-0.112  0.229   0.261   0.122   0.633   -0.383  -0.023  -0.414  -0.149  -0.089
0.047   -0.097  -0.129  -0.061  -0.265  0.157   -0.001  0.156   0.080   0.074
-0.024  0.072   0.088   0.037   0.179   -0.122  0.008   -0.108  -0.052  -0.049
0.080   -0.167  -0.206  -0.096  -0.461  0.279   0.010   0.301   0.123   0.078
0.015   -0.037  -0.050  -0.020  -0.132  0.083   0.005   0.084   0.027   0.013
0.007   0.004   0.002   0.012   0.011   -0.016  -0.003  -0.015  -0.002  0.001
Inversa de la suma de dos matrices:
0.035   -0.042  -0.060  -0.046  -0.191  0.089   0.040   0.100   0.033   0.036
0.027   -0.086  -0.116  -0.027  -0.130  0.040   -0.022  0.014   0.098   0.153
-0.056  0.059   0.154   0.061   0.152   -0.078  0.014   -0.052  -0.079  -0.126
-0.030  0.030   -0.003  -0.035  0.061   -0.069  -0.045  0.020   0.058   0.012
0.066   -0.091  -0.163  -0.064  -0.327  0.220   0.008   0.203   0.068   0.062
-0.016  0.078   0.029   0.019   0.118   -0.064  -0.030  -0.088  -0.009  -0.009
0.078   -0.076  -0.046  0.008   -0.122  0.067   0.051   0.061   -0.001  -0.013
-0.066  0.078   0.096   0.071   0.244   -0.152  -0.013  -0.144  -0.058  -0.019
-0.008  0.019   0.057   -0.003  0.094   -0.023  -0.028  -0.028  -0.038  -0.011
-0.005  0.005   0.013   -0.007  -0.003  0.046   0.039   0.003   -0.016  -0.049

C:\Users\Edmundo J Sanchez M\Desktop\SO\P6\Semaforos>
```

Archivos generados en la carpeta raíz.



Archivos generados exitosamente con el mismo contenido que el impreso.



Usaremos cinco semáforos y una sola memoria compartida de 400 bytes (con espacio para 100 flotantes de 4 bytes cada uno, el tamaño exacto para la matriz):

- ***hSem0*** y ***hSem1*** están encargados de sincronizar la comunicación entre el nivel 0 y el 1, en donde se crean A y B y se calcula AB.
- ***hSem2*** y ***hSem3*** están encargados de sincronizar la comunicación entre el nivel 1 y el 2, en donde se crean C y D.
- ***hSem4*** está encargada de sincronizar la comunicación entre el nivel 2 y el 0, en donde se calcula C + D.

En el nivel 0 calculamos (AB) −1 y (C + D) −1 . En este nivel creamos todos los semáforos, y en los niveles 1 y 2 los leemos.

Para demostrar el buen uso de los semáforos, creamos los tres procesos a la vez, es decir, el nivel 0 crea el nivel 1 y el nivel 1 crea el nivel 2 de forma inmediata. Se observa como los semáforos cumplen con su trabajo, y no importa en qué orden ejecutemos los tres niveles.

3. Análisis de la práctica

Un semáforo es una simple estructura que lleva un contador, el cual no puede ser menor a 0 ni mayor al valor máximo. Si al momento de tratar de decrementar el semáforo está en cero, el proceso actual tendrá que esperar hasta que su valor incremente y así poder volverlo a decrementar.

Normalmente el valor máximo indica cuantos recursos tenemos disponibles en total, el valor actual cuantos tenemos disponibles al momento; y las operaciones de incremento/decremento representan la asignación/liberación de un recurso.

3.1 Linux

Permite dar señal de acceso a cierta cantidad de recursos los cuales se encuentran después del semáforo, se debe tener un buen control la señal que si no le damos señal de prendido el programa nunca marcará error y seguirá corriendo los demás procesos hasta terminar por completo.

3.2 Windows

Es esta práctica los tres procesos tenían que usar solo una memoria virtual para lectura y escritura, así que pusimos el valor máximo de todos los semáforos en 1 y su valor inicial en 0. De esta forma, un valor de “1” representaba disponible y “0” ocupado.

4. Observaciones

4.1. Linux

- La estructura sembuf contiene la información del semáforo, como la de encendido y apagado.
- Cada semáforo cuenta con su respectivo identificador. Con la llamada al sistema **semctl()** creamos el semáforo, en caso de no invocarla la llamada al sistema **semop()** funcionara.
- Los semáforos tienen la misma funcionalidad que la llamada al sistema **wait()**, nada más que los semáforos permiten mayor sincronización y no solo esperar a que un proceso termine.
- Una vez que un semáforo se apague o se prende dentro de un proceso no finalizara dicho proceso, sirve como señal para la sincronización entre procesos.
- La llamada al sistema **semop()** manda la señal de encendido y apagado de los semáforos.

4.2. Windows

- La función **WaitForSingleObject** decrementa el valor actual del semáforo (si ya es cero se espera hasta que se incremente para volverlo a decrementar) y **ReleaseSemaphore** incrementa el valor actual.
- A veces, los semáforos no se destruían inmediatamente después de finalizar el proceso, por lo que, si lo volvíamos a ejecutar casi de inmediato, no se creaban con su valor inicial 0 y la aplicación se trababa.

4.3. Generales

- Al utilizar de forma correcta los semáforos, podemos resolver muchos problemas de sincronización.
- Se pueden utilizar más de un semáforo en el mismo código en ambos sistemas operativos para la sincronización de procesos.

5. Conclusiones

Tanto en Windows como en Linux tenemos acceso a distintas llamadas al sistema para la creación y manipulación de semáforos y memoria compartida. Los semáforos para una mejor sincronización de la información y la memoria compartida como canal de comunicación o de transferencia de datos entre los procesos. Pudimos observar la sincronización del comportamiento de la memoria compartida, en ambos sistemas operativos se ve un mejor comportamiento de la memoria compartida que a su vez permitió un menor uso de estas. Las llamadas al sistema de Linux y Windows varían en sus parámetros y forma de invocación.