INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE COMPUTO

“Practica 6”

Sistemas Operativos

INSTITUTO POLITÉCNICO NACIONAL

ESCOM

Equipo 7

- Héctor Chávez Rodríguez
- María Fernanda Delgado Mendoza
- Jonathan Said Gómez Marbán
- Luis Antonio Ramírez Fárias

- **Grupo:** 4CM1

I.Introducción teórica:

Los semáforos son un mecanismo de sincronización de procesos inventados por Edsger Dijkstra en 1965. Los semáforos permiten al programador **asistir al planificador del sistema operativo en su toma de decisiones** de manera que permiten sincronizar la ejecución de dos o más procesos. A diferencia de los cerrojos, los semáforos nos ofrecen un mecanismo de espera no ocupada.

Los semáforos son un tipo de datos que están compuestos por dos atributos:

- Un contador, que siempre vale >= 0.
- Una cola de procesos inicialmente vacía.

Y disponen de dos operaciones básicas que pasamos a describir en pseudocódigo:

```
down(semáforo s)
{
        si s.contador == 0:
                añade proceso a s.cola_procesos
                proceso a estado bloqueado
        sino:
                s.contador--
}
```

Nótese que siempre que queramos forzar una transición de un proceso ha estado bloqueado, tenemos que hacer que dicho proceso realice una operación *down* sobre un semáforo cuyo contador vale cero.

```
up(semáforo s)
{
        si hay procesos en s.cola_procesos:
                retira proceso de s.cola_procesos
                proceso a estado preparado
        sino:
                s.contador++
}
```

Nótese que una operación *up* sobre un semáforo en el que hay procesos en su cola resulta en que se retire uno de los procesos (el primero de la cola, es decir, el que lleva más tiempo en la cola), realizando éste la transición a estado preparado. Es un error frecuente pensar que una operación *up* resulte en que el proceso retirado de la cola pase a estado activo. Recuerde que las transiciones de estado activo a preparado y viceversa son siempre controladas por el planificador del sistema operativo.

Además, se disponen de un constructor y un destructor que permiten la creación y la liberación de un semáforo.

Ejemplo de uso de semáforos

Suponga el siguiente ejemplo en el que se quieren sincronizar dos hilos de un proceso, de manera que se dé la siguiente secuencia de ejecución: A, B, A, B, ... y así sucesivamente. El siguiente código resolvería el problema:

```
semaforo a = semaforo.create(1);
semaforo b = semaforo.create(0);

/* código del hilo A */
while (1) {
        a.down();
        printf("hilo A\n");
        b.up();
}

/* código del hilo B */
while (1) {
        b.down();
        printf("hilo B\n");
        a.up();
}
```

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 estos inconvenientes, 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.

II. Desarrollo de la practica

Función semget()

int semget (key_t *key*, int *nsems*, int *semflg*);

semget devuelve el identificador del semáforo correspondiente a la clave key. Puede ser un semáforo ya existente, o bien **semget** crea uno nuevo si se da alguno de estos casos:

a) key vale **IPC_PRIVATE**. Este valor especial obliga a **semget** a crear un nuevo y único identificador, nunca devuelto por ulteriores llamadas a **semget** hasta que sea liberado con **semctl**.

b) key no está asociada a ningún semáforo existente, y se cumple que (semflg & **IPC_CREAT**) es cierto.

A un semáforo puede accederse siempre que se tengan los permisos adecuados.

Si se crea un nuevo semáforo, el parámetro nsems indica cuántos semáforos contiene el conjunto creado; los 9 bits inferiores de semflg contienen los permisos estilo UNIX de acceso al semáforo (usuario, grupo, otros).

semflg es una máscara que puede contener **IPC_CREAT**, que ya hemos visto, o **IPC_EXCL**, que hace crear el semáforo, pero fracasando si ya existía.

Ejemplo:

int semid = semget (IPC_PRIVATE, 1, IPC_CREAT | 0744);

Operación semop

int semop (int *semid*, struct sembuf* *sops*, unsigned *nsops*);

Esta función realiza atómicamente un conjunto de operaciones sobre semáforos, pudiendo bloquear al proceso llamador. ***semid*** es el identificador del semáforo y ***sops*** es un apuntador a un vector de operaciones. ***nsops*** indica el número de operaciones solicitadas.

La estructura **sembuf** tiene estos campos:

```
struct sembuf {
 unsigned short sem_num;        // número del semáforo dentro del conjunto
 short          sem_op; // clase de operación
                                // según sea >0, <0 o ==0
 short          sem_flg;        // modificadores de operación
};
```

Cada elemento de ***sops*** es una operación sobre algún semáforo del conjunto de ***semid***. El algoritmo simplificado de la operación realizada es éste (***semval*** es el valor entero contenido en el semáforo donde se aplica la operación).

```
si semop<0
        si semval >= |semop|
                semval -= |semop|
        si semval < |semop|
                si (semflag & IPC_NOWAIT)!=0
                        la función semop() retorna
                si no
                        bloquearse hasta que semval >= |semop|
                        semval -= |semop|
si semop>0
        semval += semop
si semop==0
        si semval = 0
                SKIP
        si semval != 0
                si (semflag & IPC_NOWAIT)!=0
                        la función semop() retorna
                si no
                        bloquearse hasta que semval == 0
```

Resumiendo, un poco, si el campo ***semop*** de una operación es positivo, se incrementa el valor del semáforo. Asimismo, si ***semop*** es negativo, se decrementa el valor del semáforo si el resultado no es negativo. En caso contrario el proceso espera a que se dé esa circunstancia. Es decir, ***semop***==1 produce una operación V y ***semop***==-1, una operación P.

Programa 1

```
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 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 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 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;
           }
        }
}
```

Ejecución

```
Iniciando semaforo...
Semaforo iniciado...

semop: Padre llamando 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 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 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
hector@hector-VirtualBox:~/Documentos/SO$ Proceso hijo regresa el control del semaforo: 3/3 veces
hector@hector-VirtualBox:~/Documentos/SO$
```

Programa 2
Padre

```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 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 semáforo
                 WaitForSingleObject(hSemaforo, INFINITE);

                 //Sección critica
                 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);
}
```

Hijo

```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 critica
             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++;
         }
}
```

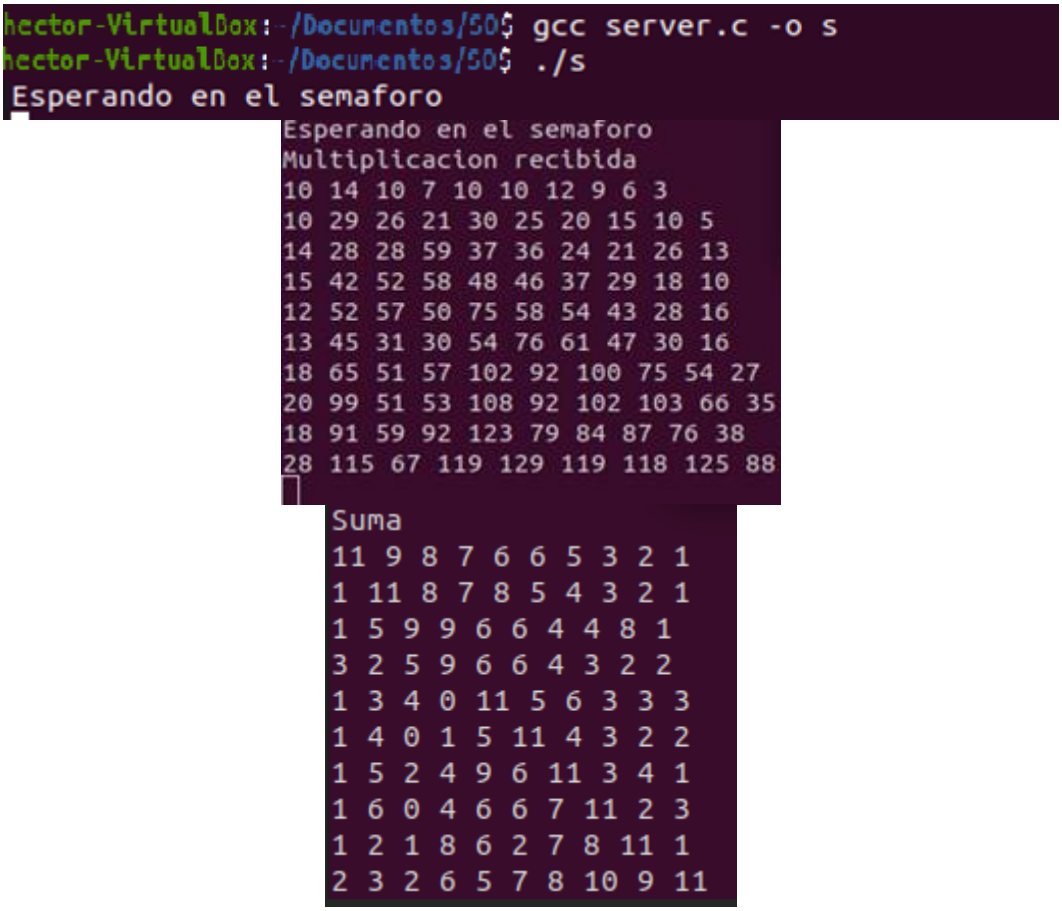
Ejecución

```
C:\Users\Hector\OneDrive\Desktop\Escuela\SO\Practicas\P6>padre hijo
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 padre entrando 3 de 3 veces al semaforo
Soy el hijo liberando 2 de 3 veces al semaforo
Soy el padre liberando 3 de 3 veces al semaforo
Soy el hijo entrando 3 de 3 veces al semaforo

C:\Users\Hector\OneDrive\Desktop\Escuela\SO\Practicas\P6>Soy el hijo liberando 3 de 3 veces al semaforo
```

Programa 3

Linux



```
Inversa Multiplicacion
-0.20 0.00 0.69 -0.40 0.36 -0.29 0.01 0.42 -0.42 -0.09
0.12 0.15 -0.44 0.23 -0.28 0.16 0.01 -0.27 0.25 0.07
0.33 -0.21 -0.42 0.25 -0.12 0.22 -0.06 -0.26 0.28 0.03
-0.15 0.08 0.15 -0.05 0.02 -0.09 0.03 0.09 -0.10 -0.01
-0.34 0.11 0.62 -0.38 0.32 -0.29 0.04 0.38 -0.38 -0.07
-0.09 -0.01 0.17 -0.10 0.09 -0.02 -0.02 0.10 -0.09 -0.03
0.26 0.06 -0.64 0.35 -0.37 0.22 0.06 -0.40 0.35 0.10
-0.42 -0.10 0.91 -0.45 0.48 -0.38 -0.01 0.62 -0.56 -0.14
0.41 -0.24 -0.58 0.38 -0.30 0.33 -0.06 -0.35 0.44 0.01
0.51 0.01 -0.85 0.28 -0.25 0.35 -0.04 -0.63 0.47 0.21

Inversa Suma
0.098 -0.044 -0.056 0.003 0.022 0.011 -0.036 -0.022 0.042 -0.003
0.012 0.139 -0.079 -0.075 -0.049 0.039 -0.006 -0.037 0.042 0.020
0.014 -0.105 0.216 -0.052 0.033 -0.097 0.036 0.078 -0.146 -0.005
-0.031 0.072 -0.087 0.135 -0.067 0.001 -0.006 -0.041 0.056 0.004
-0.020 0.073 -0.128 0.076 0.119 0.025 -0.081 -0.047 0.097 -0.033
-0.006 -0.048 0.059 0.010 -0.022 0.099 -0.001 0.009 -0.038 -0.013
0.009 -0.089 0.086 -0.040 -0.058 -0.082 0.186 0.046 -0.100 0.017
0.006 -0.077 0.085 -0.024 0.029 -0.044 -0.054 0.153 -0.027 -0.031
0.014 0.011 0.014 -0.084 0.007 0.054 -0.027 -0.079 0.091 0.016
-0.017 0.073 -0.091 0.034 0.010 -0.014 -0.021 -0.070 -0.021 0.111
```

InversaMult.txt

1	-0.20	0.00	0.69	-0.40	0.36	-0.29	0.01	0.42	-0.42	-0.09
2	0.12	0.15	-0.44	0.23	-0.28	0.16	0.01	-0.27	0.25	0.07
3	0.33	-0.21	-0.42	0.25	-0.12	0.22	-0.06	-0.26	0.28	0.03
4	-0.15	0.08	0.15	-0.05	0.02	-0.09	0.03	0.09	-0.10	-0.01
5	-0.34	0.11	0.62	-0.38	0.32	-0.29	0.04	0.38	-0.38	-0.07
6	-0.09	-0.01	0.17	-0.10	0.09	-0.02	-0.02	0.10	-0.09	-0.03
7	0.26	0.06	-0.64	0.35	-0.37	0.22	0.06	-0.40	0.35	0.10
8	-0.42	-0.10	0.91	-0.45	0.48	-0.38	-0.01	0.62	-0.56	-0.14
9	0.41	-0.24	-0.58	0.38	-0.30	0.33	-0.06	-0.35	0.44	0.01
10	0.51	0.01	-0.85	0.28	-0.25	0.35	-0.04	-0.63	0.47	0.21

InversaSuma.txt

1	0.10	-0.04	-0.06	0.00	0.02	0.01	-0.04	-0.02	0.04	-0.00
2	0.01	0.14	-0.08	-0.08	-0.05	0.04	-0.01	-0.04	0.04	0.02
3	0.01	-0.10	0.22	-0.05	0.03	-0.10	0.04	0.08	-0.15	-0.00
4	-0.03	0.07	-0.09	0.13	-0.07	0.00	-0.01	-0.04	0.06	0.00
5	-0.02	0.07	-0.13	0.08	0.12	0.02	-0.08	-0.05	0.10	-0.03
6	-0.01	-0.05	0.06	0.01	-0.02	0.10	-0.00	0.01	-0.04	-0.01
7	0.01	-0.09	0.09	-0.04	-0.06	-0.08	0.19	0.05	-0.10	0.02
8	0.01	-0.08	0.08	-0.02	0.03	-0.04	-0.05	0.15	-0.03	-0.03
9	0.01	0.01	0.01	-0.08	0.01	0.05	-0.03	-0.08	0.09	0.02
10	-0.02	0.07	-0.09	0.03	0.01	-0.01	-0.02	-0.07	-0.02	0.11

Conclusiones

La sincronización entre procesos es necesaria para evitar error de temporización debido al acceso concurrente a recursos compartidos, tales como estructuras a datos o dispositivos de E/S, por parte de procesos competidores. La sincronización entre procesos también permite el intercambio de señales de temporización (Parar / Seguir) entre procesos cooperativos con el fin de preservar las relaciones especificadas de precedencia impuesta por el problema que se resuelva.

Los semáforos son un mecanismo de sincronización entre procesos simples pero potente basado en esta filosofía. Los semáforos satisfacen la mayoría de los extensos requisitos que hemos especificados para un buen mecanismo de control de concurrencia incluyendo la no existencia de suposiciones con respecto a las velocidades relativas ni las prioridades de los procesos competidores excepto que posiblemente existan.

De las estrategias para la implementación de semáforos, la implementación con cola facilita la eliminación del problema del aplazamiento indefinido y tiene una mayor eficiencia potencial que la implementación con espera activa.