

1. Entender el funcionamiento de los "Makefile".

- Un archivo Makefile es utilizado por la herramienta make, para la generación/compilación de código. Este lee las instrucciones descritas en el archivo para poder generar el programa o cualquier otra acción del fichero makefile. Con este objetivo, los Makefile especifican qué es lo que hay que hacer por medio de ciertas reglas: Objetivo (define el programa que queremos crear), Dependencias (nombre de otro objetivo que debe hacerse antes de ejecutar el principal, o ficheros de los cuales depende el objetivo) y Comandos (lo que se debe ejecutar para construir el objetivo). Para este laboratorio no se utilizó ningún archivo Makefile, los comandos de compilación se encuentran especificados en un archivo txt llamado "Compilación y ejecución".

2. Correr el ejemplo provisto "PiPorSeries.c", anotar los resultados obtenidos explicando por qué no funciona.

- El programa PiPorSeries tiene un problema de "cooperación", ya que luego de culminar su ejecución, este no almacena los datos proporcionados por todos sus procesos. En este código, al invocar al proceso fork() se crea un proceso hijo sumamente similar al padre (sin embargo, su id es distinto) para que este realice el calculo de pi con los términos que le corresponden (se le da un inicio y un fin). Se debe tener en cuenta que el vector donde debe almacenar el calculo también "se le proporciona". El proceso SÍ realiza la operación que se espera y guarda el calculo en la posición del vector que le corresponde. Ahora, ese es el trabajo de UN proceso, pero no es solo él, son DIEZ. Así que básicamente se le está proporcionando un vector para guardar los datos a CADA UNO de los procesos que se están manejando, y estos simultaneamente deben utilizarlo. Esto no sucede, el vector que se les proporciona no es de uso "común", es un vector que cada uno utiliza internamente, es como una variable privada. Cada proceso posee su inicio, su fin y su vector donde almacenar el resultado. Por esta razón, cuando el proceso "muere" (exit) se lleva consigo a su vector con la suma almacenada. Para evitar esto, lo correcto sería devolver el resultado obtenido y así el proceso "master" almacena cada resultado, de cada proceso, en el vector "global". Pero hay un problema, el proceso no puede retornar el resultado, ya que no habría forma de eliminarlo luego, y seguiría repitiendose una y otra vez. Para evitar todo este problema, se necesita una comunicación entre los procesos, algo como un Semáforo. PD: el código de PiPorSeries está modificado (intentando comprobar la solución dicha anteriormente, el retorno), por esta razón el programa se repite varias veces hasta finalizar. Este documento también se encuentra en la carpeta Pi.

- Ejecución con intercalación de procesos, al no eliminarse.

```
jennifer@jennifer-VirtualBox: ~/Escritorio/ProgramacionParalela_B67751_Villalobos/Labora...
Archivo Editar Ver Buscar Terminal Ayuda
Valor calculado de Pi es 3.14159 con 1000000 terminos
Valor calculado de Pi es 3.14158 con 1000000 terminos
Valor calculado de Pi es 8.36111e-06 con 1000000 terminos
Valor calculado de Pi es 8.12302e-06 con 1000000 terminos
Valor calculado de Pi es 3.14159 con 1000000 terminos
Valor calculado de Pi es 3.14159 con 1000000 terminos
Valor calculado de Pi es 3.14159 con 1000000 terminos
Valor calculado de Pi es 3.14158 con 1000000 terminos
Valor calculado de Pi es 3.14158 con 1000000 terminos
Valor calculado de Pi es 7.69444e-06 con 1000000 terminos
Creating process 16307: starting value 800000, finish at 900000
Creating process 16365: starting value 900000, finish at 1000000
Valor calculado de Pi es 7.58333e-06 con 1000000 terminos
Valor calculado de Pi es 7.40476e-06 con 1000000 terminos
Valor calculado de Pi es 6.2619e-06 con 1000000 terminos
Valor calculado de Pi es 6.15079e-06 con 1000000 terminos
Valor calculado de Pi es 3.14158 con 1000000 terminos
Valor calculado de Pi es 3.14159 con 1000000 terminos
Valor calculado de Pi es 3.14158 con 1000000 terminos
Valor calculado de Pi es 3.14159 con 1000000 terminos
Valor calculado de Pi es 3.14159 con 1000000 terminos
Valor calculado de Pi es 3.14159 con 1000000 terminos
Valor calculado de Pi es 3.14158 con 1000000 terminos
Valor calculado de Pi es 3.14159 con 1000000 terminos
Valor calculado de Pi es 3.14159 con 1000000 terminos
Valor calculado de Pi es 3.14159 con 1000000 terminos
Valor calculado de Pi es 3.14159 con 1000000 terminos
Valor calculado de Pi es 3.14158 con 1000000 terminos
Valor calculado de Pi es 3.14158 con 1000000 terminos
Valor calculado de Pi es 3.14159 con 1000000 terminos
```

(a) Repetición del programa por procesos sin finalizar

3. Semáforos: construir la implantación de la clase. Completar la clase Semaforo para poder sincronizar procesos de una misma computadora.

- Se utilizó la interfaz Semaforo.h (provista por el profesor) para generar la estructura de la clase Semaforo. Dentro del constructor se define el ID del semáforo creado, utilizando el comando semget. En este, se usa el carné como key, se establece que será 1 semáforo creado y se define que se quiere crear el mismo con IPC_CREAT. Además, se adhiere una verificación para asegurar si el semáforo fue creado correctamente, en caso contrario, se avisará de un error (esto con perror). Por último se utiliza el comando semctl para asignarle un valor al semáforo creado (aquí entra en juego el valor adquirido por la unión) por medio de SETVAL. Al declarar el destructor se utiliza semctl nuevamente, esta vez con la instrucción IPCRMID para eliminar el semáforo creado (según su id).

```

19
20     Semaforo::Semaforo(int valor)
21     {
22         this->id = semget(0xB67751, 1, IPC_CREAT|0600);
23
24         //Para saber si falla
25         if (id == -1)
26         {
27             perror("Semaforo::Semaforo");
28             exit(1);
29         }
30
31         u.val = valor; //Parametro entrante (0)
32         int aux = semctl(id, 0, SETVAL, u);
33     }
34
35     Semaforo::~Semaforo()
36     {
37         semctl(id, 0, IPC_RMID);
38     }

```

(b) Constructor y Destructor

•Al inicio se realizaron las declaraciones de union y struct dentro de los métodos del programa, sin embargo, luego se mejoró al comprender que ambas son parte de un todo (se utilizan en diferentes métodos en toda la clase). El struct se declaró para luego otorgarle las indicaciones necesarias según fuera el caso en los métodos clave del Semáforo: Wait (P) y Signal (V).

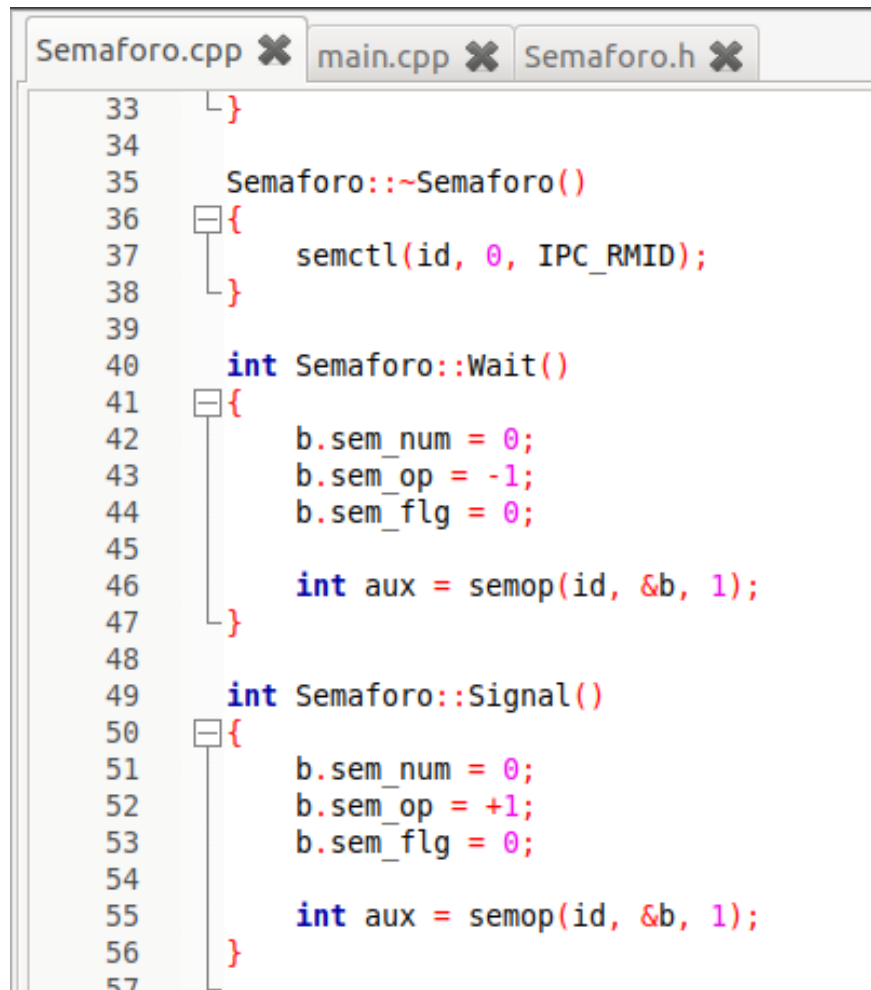
```

7
8     union uni
9     {
10         int val = 0; //Valor para el comando SETVAL
11         struct semid_ds *buf; //Para los comandos IPC SET
12         unsigned short *array; //Para los comandos GETALL y SETALL
13         struct seminfo *__buf; //Para el comando IPC_INFO
14     };
15
16     //Inicializacion de la union y la estructura
17     uni u;
18     struct sembuf b; //Utilizado para el Wait y Signal
19

```

(c) Declaración de la unión y el struct

•El struct se necesitaba para poder implementar la función semop. La diferencia entre ambos es que al asignarle un valor de operador al Wait, este debe ser -1, para así decrementar en uno el valor del semáforo (si este es menor o igual a 0 el proceso espera). Por otro lado, en el método Signal se establece el operador con un +1, para incrementar el valor del semáforo y así "despertar" al primer proceso que exista en la cola.



```
33     }
34
35     Semaforo::~Semaforo()
36     {
37         semctl(id, 0, IPC_RMID);
38     }
39
40     int Semaforo::Wait()
41     {
42         b.sem_num = 0;
43         b.sem_op = -1;
44         b.sem_flg = 0;
45
46         int aux = semop(id, &b, 1);
47     }
48
49     int Semaforo::Signal()
50     {
51         b.sem_num = 0;
52         b.sem_op = +1;
53         b.sem_flg = 0;
54
55         int aux = semop(id, &b, 1);
56     }
57 }
```

(d) Métodos Wait y Signal

- Compilación y resultados de ejecución

```
jennifer@jennifer-VirtualBox:~/Escritorio/ProgramacionParalela_B67751_Villalobo
s/Laboratorios/Semana1/Semaforo$ g++ -c -g Semaforo.cpp
jennifer@jennifer-VirtualBox:~/Escritorio/ProgramacionParalela_B67751_Villalobo
s/Laboratorios/Semana1/Semaforo$ g++ -c -g main.cpp
jennifer@jennifer-VirtualBox:~/Escritorio/ProgramacionParalela_B67751_Villalobo
s/Laboratorios/Semana1/Semaforo$ g++ -g main.o Semaforo.o -o ejecutable
jennifer@jennifer-VirtualBox:~/Escritorio/ProgramacionParalela_B67751_Villalobo
```

(e) Compilar en terminal

```
jennifer@jennifer-VirtualBox:~/Escritorio/ProgramacionParalela_B67751_Villalobo
s/Laboratorios/Semana1/Semaforo$ ./ejecutable
Esperando para activar el semaforo 0
Esperando para activar el semaforo 1
Esperando para activar el semaforo 2
Esperando para activar el semaforo 3
Esperando para activar el semaforo 4
Esperando para activar el semaforo 5
Esperando para activar el semaforo 6
Esperando para activar el semaforo 7
Esperando para activar el semaforo 8
Esperando para activar el semaforo 9
Esperando que el semaforo se active ...
█
```

(f) Esperando lectura

```
jennifer@jennifer-VirtualBox:~/Escritorio/ProgramacionParalela_B67751_Villalobo
s/Laboratorios/Semana1/Semaforo$ g++ -c -g Semaforo.cpp
jennifer@jennifer-VirtualBox:~/Escritorio/ProgramacionParalela_B67751_Villalobo
s/Laboratorios/Semana1/Semaforo$ g++ -c -g main.cpp
jennifer@jennifer-VirtualBox:~/Escritorio/ProgramacionParalela_B67751_Villalobo
s/Laboratorios/Semana1/Semaforo$ g++ -g main.o Semaforo.o -o ejecutable
jennifer@jennifer-VirtualBox:~/Escritorio/ProgramacionParalela_B67751_Villalobo
s/Laboratorios/Semana1/Semaforo$ ./ejecutable
Esperando para activar el semaforo 0
Esperando para activar el semaforo 1
Esperando para activar el semaforo 2
Esperando para activar el semaforo 3
Esperando para activar el semaforo 4
Esperando para activar el semaforo 5
Esperando para activar el semaforo 6
Esperando para activar el semaforo 7
Esperando para activar el semaforo 8
Esperando para activar el semaforo 9
Esperando que el semaforo se active ...
2
Semaforo activado
jennifer@jennifer-VirtualBox:~/Escritorio/ProgramacionParalela_B67751_Villalobo
s/Laboratorios/Semana1/Semaforo$ █
```

(g) Fin de ejecución

Figure 1: Demostración de compilación y ejecución.

: