

# Sistemas Operativos 2/2021

## Laboratorio 3

Profesores:

Cristóbal Acosta (cristobal.acosta@usach.cl)

Fernando Rannou (fernando.rannou@usach.cl)

Ayudantes:

Manuel I. Manríquez (manuel.manriquez.b@usach.cl)

Benjamín Muñoz (benjamin.munoz.t@usach.cl)

Camila Norambuena (camila.norambuena.v@usach.cl)

### I. Objetivos Generales

El objetivo del presente laboratorio es aplicar los conocimientos adquiridos en cátedra sobre concurrencia. Se espera que los estudiantes generen una solución teniendo en cuenta las consideraciones propias del diseño multihebreado y que realicen llamadas al SO Linux de manera exitosa.

### II. Objetivos Específicos

1. Validar los parámetros recibidos con `getopt()`.
2. Implementar medidas de sincronización de hebras.
3. Implementar soluciones de exclusión mutua por software.
4. Practicar técnicas de buena documentación de programas.
5. Practicar uso de `Makefile` para compilación de programas.

### III. Conceptos

#### III.A. Concurrencia y Sincronización

Cuando dos o más tareas (procesos o hebras) comparten algún recurso en forma concurrente o paralela, debe existir un mecanismo que sincronice sus actividades.

De no existir una sincronización, es posible sufrir una corrupción en los recursos compartidos u obtener soluciones incorrectas.

#### III.B. Sección Crítica

Porción de código que se ejecuta de forma concurrente y podría generar conflicto en la consistencia de datos debido al uso de variables globales.

#### III.C. Mutex

Provee exclusión mutua, permitiendo que sólo una hebra a la vez ejecute la sección crítica.

### III.D. Hebras

Los hilos POSIX, usualmente denominados *threads*, son un modelo de ejecución que existe independientemente de un lenguaje, además es un modelo de ejecución en paralelo. Estos permiten que un programa controle múltiples flujos de trabajo que se superponen en el tiempo.

Para poder utilizar hebras, es necesario incluir la librería **pthread.h**. Por otro lado, dentro de la función `main`, se debe instanciar la variable de referencia a las hebras, para esto se utiliza el tipo de dato **pthread\_t** acompañado del nombre de variable.

Luego, el código que ejecutarán las hebras se debe construir en una función de la forma:

```
void * function (void * params)
```

La cual recibe parámetros del tipo `void`, por lo cual es necesario castear el o los parámetros de entrada para así poder utilizarlos sin problemas.

Algunas funciones para manejar las hebras son:

- **pthread\_create:** función que crea una hebra. Recibe como parámetros de entrada:
  - La variable de referencia a la hebra que desea crear.
  - Los atributos de éste, los cuales no es obligación de modificar, por lo que en caso de no querer hacerlo, simplemente se deja `NULL`.
  - El nombre de la función que la hebra ejecutará (la cual debe cumplir con la descripción antes mencionada)
  - Por último, los parámetros de entrada (de la función que se ejecutará) previamente casteados.

Un ejemplo sería:

```
while(i < numeroHebras)
{
    pthread_create(&hilo[i], NULL, escalaGris, (void *) &structHebra[i].id);
    i++;
}
```

- **pthread\_join:** función donde la hebra que la ejecuta, espera por las hebras que se ingresan por parámetro de entrada.

Un ejemplo sería:

```
while(i < numeroHebras)
{
    pthread_join(hilo[i], NULL);
    i++;
}
```

- **pthread\_mutex\_init:** función que inicializa un mutex, pasando por parámetros la referencia al mutex, y los atributos con que se inicializa.

Para inicializar la estructura se utiliza:

```
while(i < numeroHebras)
{
    pthread_mutex_init(&MutexAcumulador, NULL);
    i++;
}
```

Donde **MutexAcumulador**, es una variable (de tipo `pthread_mutex_t`) que representa un mutex, el cual permitirá implementar exclusión mutua a una sección crítica que será ejecutada por varias hebras.

- **pthread\_mutex\_lock:** entrega una solución a la sección crítica. Ésta recibe como parámetros la variable que se desea bloquear para el resto de hebras. Un ejemplo de uso sería:

```
pthread_mutex_lock(&MutexAcumulador);
```

Cabe destacar que la primera hebra en ejecutar **pthread\_mutex\_lock** podrá ingresar a la sección crítica, el resto de hebras quedarán bloqueadas a la espera de que se libere el mutex.

- **pthread\_mutex\_unlock:** permite liberar una sección crítica. Ésta recibe como parámetro de entrada, la variable que se desea desbloquear. Su implementación es la siguiente:

```
pthread_mutex_unlock(&MutexAcumulador);
```

- **pthread\_barrier\_init:** permite asignar recursos a una barrera e inicializar sus atributos. Su implementación es:

```
pthread_barrier_init(&rendezvous, NULL, NTHREADS);
```

Donde `rendezvous` es una variable de tipo `pthread_barrier_t`, `NULL` indica que se utilizan los atributos de barrera predeterminados (se pide usar estos atributos, no debiese ser necesario modificarlos) y `NTHREADS` es la cantidad de hebras que deben esperar en la barrera.

- **pthread\_barrier\_wait:** permite sincronizar los hilos en una barrera especificada. El hilo de llamada bloquea hasta que el número requerido de hilos ha llamado a `pthread_barrier_wait()` especificando la barrera. Un ejemplo de su uso:

```
funcion1();  
pthread_barrier_wait(&rendezvous);  
funcion2();
```

- **pthread\_barrier\_destroy:** cuando ya no se necesita una barrera, se debe destruir. Su implementación es:

```
pthread_barrier_destroy(&rendezvous);
```

## IV. Enunciado

En este laboratorio, se busca simular la energía depositada en un material por partículas de alta energía que lo impactan (ej: partículas en un satélite). El programa a desarrollar calculará y registrará la energía en Joules que cada partícula va depositando en algún punto del material.

Para facilitar el trabajo, se utilizará un modelo de una dimensión (1D). Entonces, se usará un arreglo para llevar registro de las energías acumuladas.

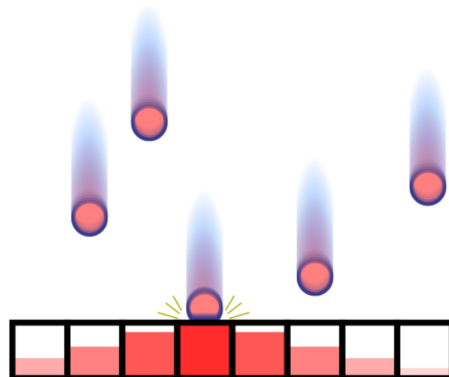


Figure 1. Acumulación de energías por partículas.

Por cada partícula simulada, se entrega su posición de impacto  $j$  y la energía potencial  $E_p$  que trae. La energía depositada en cada celda  $i$  del material es:

$$E_i = E_i + \frac{10^3 * E_p}{N \sqrt{|j - i| + 1}}, \forall i = 0, 1, \dots, N - 1 \quad (1)$$

Para este caso  $N$  representa el número de celdas que dispone el material. Cabe destacar que una partícula puede impactar en una posición mayor a  $N$ , y aunque impacte fuera del material, su energía igual puede afectar en alguna zona del material. En caso de que la energía depositada sea muy pequeña, esta no se registrará, para lo cual se utilizará un umbral que se calcula como  $MIN\_ENERGY = 10^{-3}/N$ , por lo que se depositará energía solo en el caso que  $E_i \geq MIN\_ENERGY$ .

#### IV.A. Hebras

Para este caso, habrá una hebra **productora** que leerá el archivo de manera secuencial y llenará un buffer de largo  $b$  (parámetro ingresado por terminal) con las posiciones de las partículas y sus impactos. Dicho buffer actuará como **recurso compartido** para una cantidad  $h$  de hebras hijas (parámetro también ingresado por consola), las cuales **consumirán** del buffer cuando se encuentre lleno, hasta tener una cantidad de partículas definida por la siguiente fórmula:

$$n = \frac{cantidad\_de\_particulas}{cantidad\_de\_hebras} \quad (2)$$

Una vez las hebras hijas tengan la cantidad de partículas que les corresponde, procesarán los impactos obteniendo un arreglo con resultados parciales de la energía acumulada de sus partículas. Al obtener el resultado parcial, se deben escribir las energías sobre una estructura común, para la cual también debe proveerse exclusión mutua. Finalmente, cuando todas las hebras terminen su trabajo, la hebra madre leerá los resultados de la estructura común y los escribirá en el archivo resultante.

#### IV.B. Archivos de entrada y salida

Un bombardeo de partículas se especifica en un archivo de entrada, donde en cada línea se muestra una partícula, siendo el primer elemento la posición de impacto y el segundo la cantidad de energía con la que impacta. Es importante destacar que en este archivo **NO** se especifica el número de celdas, ya que este parámetro será ingresado por la terminal. El siguiente es un ejemplo de un archivo de bombardeo:

```
4 81
8 10
10 100
7 35
11 12
5 8
```

Para el archivo de salida, después de haber impactado todas las partículas, se debe escribir en la primera línea la posición del material con máxima energía y la cantidad acumulada. En las siguientes líneas, se debe indicar cada posición en orden con su respectiva energía acumulada (se deben escribir todas las celdas del material). Por ejemplo, haciendo uso de la entrada anterior, teniendo un material con 35 celdas, el resultado sería el siguiente, donde se logra ver en la celda 10 la máxima energía y luego la energía acumulada en todas las posiciones del material.

```
10 4732.568359
0 2537.520752
1 2745.225830
2 3027.479004
3 3456.707031
...
```

## V. Parámetros de Entrada

Para este trabajo se debe entregar al menos un archivo llamado `lab3.c` y un *Makefile* que entregue un archivo ejecutable llamado `lab3`. La ejecución del programa tendrá los siguientes parámetros que deben ser procesados por `getopt()`:

- `-N` : número de celdas
- `-c`: cantidad de líneas del archivo de entrada
- `-i filename` : archivo con el bombardeo
- `-o filename` : archivo de salida
- `-b número` : tamaño del buffer de la hebra productora.
- `-h número` : número de hebras.
- `-D`: bandera que indica si se entregan resultados por `stdout` (opcional).

Por ejemplo:

```
$ ./lab3 -N 6 -c 10 -i input.txt -o output.txt -b 64 -h 4 -D
```

En caso que se utilice el flag `-D` para debug, se debe imprimir por `stdout` un gráfico simple y normalizado con las energías. Esto se hará mediante una función externa que debe ser enlazada y declarada en el código:

```
extern void niceprint(int N, float *Energy, float Maximum);
```

Donde los argumentos son el número de celdas y el arreglo con las energías. Para el caso anterior, utilizando un `N=10` y el flag `D` presente se obtiene:

```

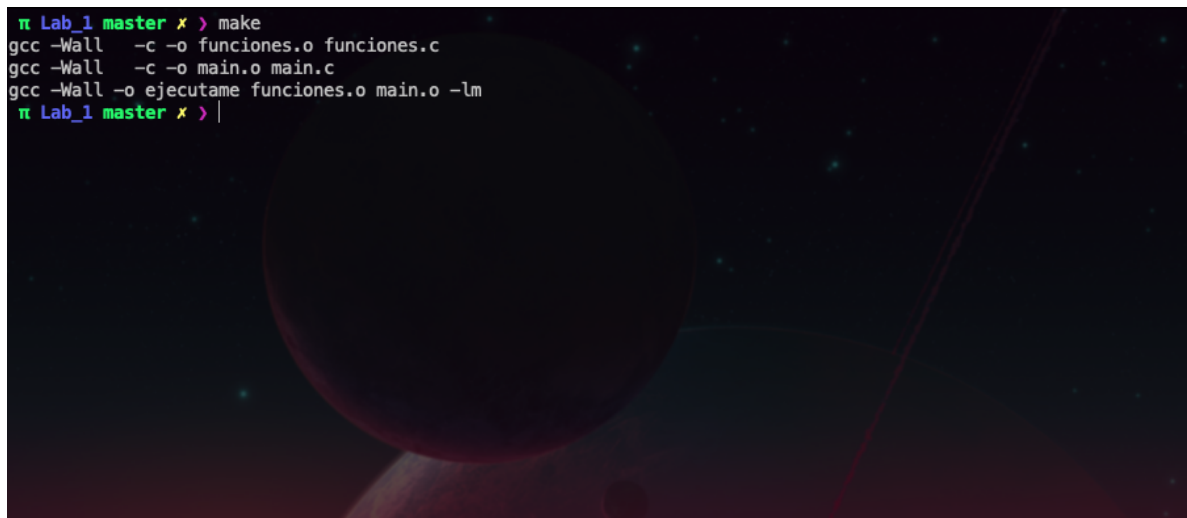
0 8881.3223 |oooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo
1 9608.2910 |oooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo
2 10596.1777|oooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo
3 12098.4746|oooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo
4 15066.8076|oooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo
5 13584.3311|oooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo
6 13256.4805|oooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo
7 14255.6436|oooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo
8 13870.8066|oooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo
9 14156.3018|oooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo

```

## VI. Entregables

El laboratorio es individual o en parejas y se descontará 1 punto por día de atraso. Cabe destacar que no entregar un laboratorio implica la reprobación de la asignatura. Finalmente, debe subir en un archivo comprimido a Uvirtual los siguientes entregables:

- **Makefile:** Archivo para `make` que compila el programa. El makefile debe compilar cada vez que haya un cambio en el código. Si no hay cambios, el comando `make` no debe crear un nuevo objeto. Una vez terminada la compilación debe crear el ejecutable llamado `lab3`.



```

π Lab_1 master x > make
gcc -Wall -c -o funciones.o funciones.c
gcc -Wall -c -o main.o main.c
gcc -Wall -o ejecutame funciones.o main.o -lm
π Lab_1 master x > |

```

Figure 2. Ejemplo de funcionamiento de un Makefile

- **archivos .c y .h** Se debe tener como mínimo un archivo .c principal llamado `lab3.c` con el main del programa. Además, deben existir dos o mas archivos con el proyecto (\*.c, \*.h) . Se debe tener mínimo un archivo .h que tenga cabeceras de funciones, estructuras o datos globales. Se deben comentar todas las funciones de la forma:

```

//Entradas: explicar qué se recibe
//Funcionamiento: explicar qué hace
//Salidas: explicar qué se retorna

```

- Se considerará para evaluación las buenas prácticas de programación, como modularidad y nombres de variables representativos.
- Trabajos con códigos que hayan sido copiados de un trabajo de otro grupo serán calificados con la nota mínima.

El archivo comprimido debe llamarse: RUTESTUDIANTE1\_RUTESTUDIANTE2.zip

Ejemplo: 19689333k\_186593220.zip

**NOTA:** Si los laboratorios son en parejas, pueden ser de distintas secciones. Si es así, por favor avisar a los ayudantes Benjamín Muñoz y Camila Norambuena.

## **VII. Fecha de entrega**

Domingo 16 de Enero 2021, hasta las 23:55 hrs.