

Sistemas Operativos 2/2021

Laboratorio 2

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

Este laboratorio tiene como objetivo aplicar los conocimientos de la asignatura sobre creación de procesos y la comunicación entre ellos en sistemas operativos basados en linux. Para lograr dicha creación y comunicación, se espera que los estudiantes apliquen el llamado a sistema `fork()` junto con las funciones `pipe()`, `exec()` y `dup2()`.

II. Objetivos Específicos

1. Validar los parámetros recibidos con `getopt()`.
2. Creación de procesos con `fork()`
3. Comunicación de procesos mediante pipes
4. Uso de funciones `fork()`, `exec()`, `pipe()` y `dup2()`
5. Conocer y practicar uso de `Makefile` para compilación de programas.
6. Simular y apreciar el impacto de partículas en un material.

III. Conceptos

III.A. Funciones `exec`

La familia de funciones `exec()` reemplaza la imagen de proceso actual con una nueva imagen de proceso. A continuación se detallan los parámetros y el uso de dichas funciones:

- `int execl(const char *path, const char *arg, ...);`
 - Se le debe entregar el nombre y ruta del fichero ejecutable
 - Ejemplo:
`execl("/bin/ls", "/bin/ls", "-l", (const char *)NULL);`
- `int execlp(const char *file, const char *arg, ..., (const char *)NULL);`
 - Se le debe entregar el nombre del fichero ejecutable (implementa búsqueda del programa).
 - Ejemplo:
`execlp("ls", "ls", "-l", (const char *)NULL);`

- **int execl(const char *path, const char *arg,..., char * const envp[]);**
 - Se le debe entregar el nombre y ruta del fichero ejecutable, recibe además un arreglo de strings con las variables de entorno
 - Ejemplo:

```
char *env[] = { "PATH=/bin", (const char *)NULL};
execl("ls", "ls", "-l", (const char *)NULL, char *env[]);
```
- **int execlv(const char *path, char *const argv[]);**
 - Se le debe entregar el nombre y ruta del fichero ejecutable, recibe además un arreglo de strings con los argumentos del programa
 - Ejemplo:

```
char *argv[] = { "/bin/ls", "-l", (const char *)NULL};
execlv("ls",argv);
```
- **int execlvp(const char *file, char *const argv[]);**
 - Se le debe entregar el nombre del fichero ejecutable (implementa búsqueda del programa), recibe además un arreglo de strings con los argumentos del programa
 - Ejemplo:

```
char *argv[] = { "ls", "-l", (const char *)NULL};
execlvp("ls",argv);
```
- **int execlvpe(const char *file, char *const argv[], char *const envp[]);**
 - Se le debe entregar el nombre del fichero ejecutable (implementa búsqueda del programa), recibe además un arreglo de strings con los argumentos del programa y un arreglo de strings con las variables de entorno
 - Ejemplo:

```
char *env[] = { "PATH=/bin", (const char *)NULL};
char *argv[] = { "ls", "-l", (const char *)NULL};
execlvpe("ls",argv,env);
```

Tomar en consideración que por convención se utiliza como primer argumento el nombre del archivo ejecutable y además **todos** los arreglos de *string* deben tener un puntero *NULL* al final, esto le indica al sistema operativo que debe dejar de buscar elementos.

Otra cosa importante es que la definición de estas funciones vienen incluidas en la biblioteca **unistd.h**, que algunos compiladores la incluyen por defecto, pero en ciertos sistemas operativos deben ser incluidas explícitamente al comienzo del archivo con la sentencia **#include <unistd.h>**.

III.B. Función fork

Esta función crea un proceso nuevo o “proceso hijo” que es casi exactamente igual que el “proceso padre”. Si **fork()** se ejecuta con éxito devuelve:

- Al padre: el PID del proceso hijo creado.
- Al hijo: el valor 0.

Es decir, la única diferencia entre estos procesos (padre e hijos) es el valor de la variable de identificación PID.

A continuación un ejemplo del uso de **fork()**:

```

1  #include <sys/types.h>
2  #include <unistd.h>
3  #include <stdio.h>
4
5  int main(int argc, char *argv[])
6  {
7      pid_t pid1, pid2;
8      int status1, status2;
9
10     if ( (pid1=fork()) == 0 )
11     { /* hijo */
12         printf("Soy el primer hijo (%d, hijo de %d)\n", getpid(), getppid());
13     }
14     else
15     { /* padre */
16         if ( (pid2=fork()) == 0 )
17         { /* segundo hijo */
18             printf("Soy el segundo hijo (%d, hijo de %d)\n", getpid(), getppid());
19         }
20         else
21         { /* padre */
22             /* Esperamos al primer hijo */
23             waitpid(pid1, &status1, 0);
24             /* Esperamos al segundo hijo */
25             waitpid(pid2, &status2, 0);
26             printf("Soy el padre (%d, hijo de %d)\n", getpid(), getppid());
27         }
28     }
29
30     return 0;
31 }

```

Figure 1. Creando procesos con fork().

III.C. Getopt

La función `getopt()` pertenece a la biblioteca `<unistd.h>` y sirve para analizar argumentos ingresados por línea de comandos, asignando *options* de la forma: "-x", en donde "x" puede ser cualquier letra. A continuación, se detallan los parámetros y el uso de la función:

`int getopt(int argc, char * const argv[], const char *optstring)`

- **argc**: Argumento de la función main de tipo `int`, indica la cantidad de argumentos que se introdujeron por línea de comandos.
- **argv[]**: Arreglo de la función main de tipo `char * const`, almacena los argumentos que se introdujeron por línea de comandos.
- **optstring**: String en el que se indican los "option characters", es decir, las letras que se deben acompañar por signos "-". Si un option requiere un argumento, se debe indicar en optstring seguido de ":", en el caso contrario, se considerará que el option no requiere argumentos y por lo tanto el ingreso de argumentos es opcional.
- **option character**: Si en `argv[]` se incluyó el option "-a" y optstring contiene "a", entonces `getopt` retorna el char "a" y además se setea a la variable `optarg` para que apunte al argumento que acompaña al option character encontrado.

`Getopt` retorna distintos valores, que dependen del análisis de los argumentos ingresados por línea de comandos (contenidos en `argv[]`) y busca los option characters reconocidos en `optstring`. Por ejemplo, retorna -1 cuando se terminan de leer los option characters; y retorna -? cuando se lee un option no reconocido en `optstring`. También este es un valor de retorno cuando un option requiere un argumento, pero en línea de comandos se ingresó sin argumento.

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.

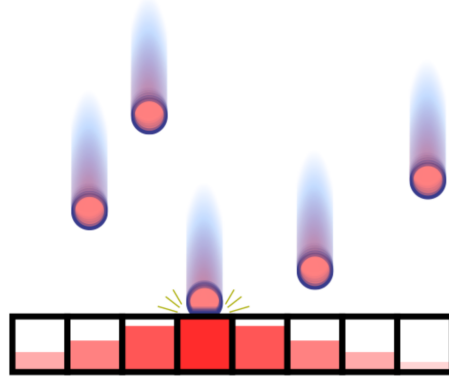


Figure 2. Acumulación de energías por partículas.

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. Procesos

En este laboratorio habrá un proceso principal, el cual creará una cantidad p de procesos hijos (parámetro entregado por consola), estos se dividirán y procesarán el impacto de las partículas. Los procesos deben ser creados obligatoriamente con la función `fork()` y mediante pipes deben recibir los parámetros que se estimen convenientes para que puedan procesar los impactos de partículas, como por ejemplo: el nombre del archivo de entrada, cantidad de partículas a procesar, etc. La cantidad de partículas por procesar para cada proceso hijo está dictada por la siguiente fórmula:

$$n = \frac{cantidad_de_particulas}{cantidad_de_procesos} \quad (2)$$

En caso que la división no resulte exacta, habrá un proceso que trabajará con menos o más partículas de las que le corresponda, por ejemplo, si hay 17 partículas y 4 procesos, cada proceso hijo trabajará con 4 partículas, excepto el último que trabajará con 5. Una vez los procesos hijos reciben los parámetros de entrada por pipes, deben calcular parcialmente la acumulación de energía en el material, y enviar dicho resultado al proceso padre, que recopilará todos los arreglos de energías parciales y los sumará para obtener el resultado final. Recordar que para lograr esta comunicación de procesos se deben aplicar las funciones de pipes, `exec()` y `dup2()`

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 el código fuente y un *Makefile* que entregue un archivo ejecutable llamado lab2. La ejecución del programa tendrá los siguientes parámetros que deben ser procesados por `getopt()`:

- `-N` : número de celdas
- `-p`: número de procesos
- `-c`: cantidad de líneas del archivo de entrada
- `-i filename` : archivo con el bombardeo
- `-o filename` : archivo de salida
- `-D`: bandera que indica si se entregan resultados por `stdout` (opcional).

Por ejemplo:

```
$ ./lab2 -N 6 -p 8 -c 10 -i input.txt -o output.txt -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:

```

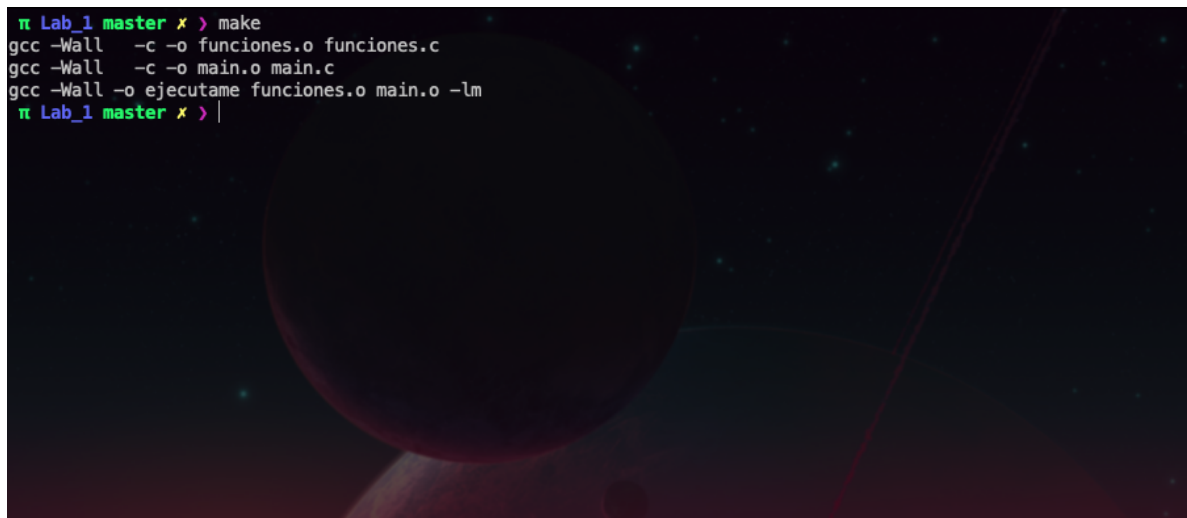
0 8881.3223 |oooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo
1 9608.2910 |oooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo
2 10596.1777|oooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo
3 12098.4746|oooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo
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 `lab2`.



```

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 3. Ejemplo de funcionamiento de un Makefile

- **archivos .c y .h** Se debe tener como mínimo un archivo `.c` principal llamado `lab2.c` con el main del programa y el código del proceso padre, mientras que para los procesos hijos se debe crear un archivo llamado `bomb.c` que será el que procesará las partículas y hará los cálculos parciales. Además, se debe tener mínimo un archivo `.h` que tenga cabeceras de funciones, estructuras o datos globales, y 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.
- 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 5 de Diciembre 2021, hasta las 23:55 hrs.