

Sistemas Operativos 1/2021

Laboratorio 2

Profesores:

Cristóbal Acosta (cristobal.acosta@usach.cl)

Fernando Rannou (fernando.rannou@usach.cl)

Ayudantes:

Cristóbal Fernández (cristobal.fernandez@usach.cl)

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

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

I. Objetivos Generales

El objetivo de este laboratorio es aplicar los conocimientos adquiridos en cátedra sobre llamadas al sistema operativo Linux, de tal forma que se puedan crear múltiples procesos conectados entre sí. Se espera que los estudiantes puedan aplicar de manera exitosa llamadas a funciones como `fork()`, `pipe()`, `exec()` y `dup2()` aplicadas a un pipeline de procesamiento de imágenes.

II. Objetivos Específicos

1. Conocer y usar la llamada de `fork()` como método de creación de procesos.
2. Modificar la imagen de un proceso usando algún miembro de la familia `exec()`.
3. Comunicar distintos procesos usando llamadas al sistema operativo como `pipe()` y `dup2()`.
4. Practicar técnicas de buena documentación de programas.
5. Practicar uso de `Makefile` para compilación de programas.
6. Practicar distintas técnicas de procesamiento de imágenes.

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 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 * constargv[], const char *optstring)`

- **argc**: Argumento de main de tipo `int`, indica la cantidad de argumentos que se introdujeron por línea de comandos.
- **argv[]**: Arreglo de 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.

III.D. Funciones Pipe

En sistemas operativos basados en UNIX se utiliza la función pipe para comunicar dos procesos relacionados (padre-hijo) a través de descriptores de archivo. Los pipes se comunican de forma unidireccional, es decir, para dos procesos solo uno escribe y el otro solamente puede leer lo escrito. Por otro lado, se puede utilizar la

funcion `dup2()` para cambiar los File Descriptor estándar de los procesos, pudiendo redirigir un `printf` u otra operación de escritura estándar hacia un pipe, y lo mismo para una operación de lectura, por lo que dicha función es útil al momento de comunicar procesos.

III.E. Input/Output y procuración de memoria

El sistema operativo provee a los procesos de varios servicios para acceder a diversos recursos del sistema. Por ejemplo, la procuración de memoria es un servicio que el SO presta a los procesos. Los procesos en sí no procuran memoria, sino le solicitan al SO tal servicio.

En este lab usaremos dos familias de llamados al sistema: los servicios de lectura y escritura en un archivo, y los servicios de procuración de memoria dinámica.

I/O :

Los llamados al sistema que usaremos en este lab para leer y escribir imágenes en archivo son

1. `int open(const char *pathname, int flags)`
2. `ssize_t read(int fd, void *buf, size_t count)`
3. `ssize_t write(int fd, const void *buf, size_t count)`
4. `int close(int fd)`

`open()` se usa para abrir (y crear) un archivo llamado `pathname`. La variable `flags` indica el modo en que se abrirá el archivo.

`open()` retorna un entero (`int`) que es el descriptor del archivo abierto. En caso que hubo algún error, `open()` retorna -1. Note que todos los llamados retornan un número. Es responsabilidad del programador recibir esos números y tomar las acciones apropiadas en caso de error. Por ejemplo,

```
int f1;

if (( f1 = open("/home/estudiante1/lab1/image1.raw", O_RDONLY)) == -1) {
    printf("Error al abrir archivo\n");
    exit(-1);
}
```

El descriptor de archivos (variable entera) representa al archivo en el sistema operativo, y es el argumento que se debe usar en `read()` y `write()`. Luego, asumiendo que `A` es un puntero y que tiene memoria procurada, suficiente para almacenar `N` bytes, usamos

```
int nbytes;
if ((nbytes = read(f1, A, N)) != N) {
    printf("Error al leer archivo\n");
    exit(-1);
}
```

Nuevamente, chequeamos el retorno del llamado al sistema para verificar si hubo o no algún error.

Note que para `read()`, no es importante el tipo de datos que está leyendo. No se especifica si se leen enteros, caracteres, flotantes, etc. Simplemente se dice cuantos bytes se debe leer del archivo. Estos bytes son almacenados en la memoria apuntada por `A`, el cual necesariamente tiene un tipo de datos. Luego, los datos que están en el archivo deben estar justamente en el formato binario o de representación interna del tipo de dato del puntero que apunta a la memoria donde serán leídos. Por ejemplo, si se desea leer `N` enteros en el arreglo `A[]`, éste debe ser declarado como `int`, y se debe especificar el número total de bytes que esos `N` enteros representan:

```
int A[N];
int nbytes;
if ((nbytes = read(f1, A, N*sizeof(int))) != N) {
    printf("Error al leer archivo\n");
}
```

```

        exit(-1);

    }

```

Pero si queremos leer N caracteres (`char`),

```

char A[N];
int nbytes;
if ((nbytes = read(f1, A, N*sizeof(char))) != N) {
    printf("Error al leer archivo\n");
    exit(-1);
}

```

En el primer caso, de enteros, si se abre el archivo de entrada, no se verían los números como se ven cuando éstos están en formato ASCII, pues están almacenados en formato de representación binaria de `int`. Esta misma lógica se aplica a `write()`.

Al finalizar el uso de un archivo, siempre ciérralo con `close()`.

MANEJO DE MEMORIA DINÁMICA :

El programa de manejo de imágenes debe estar preparado para trabajar con imágenes de distinto tamaño. Por lo tanto, para procurar y manejar el espacio de memoria necesitado se usa los llamado al sistema:

1. `void *malloc(size_t size)`
2. `void free(void *ptr)`

`malloc()` procura `size` bytes y retorna un puntero (tipo `void`) que apunta a dicha memoria. Si hubo algún error, retorna `NULL`. Entonces, al igual que los llamados de I/O, `malloc()` no se preocupa del tipo de dato que será almacenado en dicha memoria. Estos se pueden usar apropiadamente especificando conversiones de tipo al puntero. Por ejemplo, si se desea procurar memoria para N numeros flotantes de precisión simple, usamos

```

float *p;
if ((p = (float *) malloc(N*sizeof(float))) == NULL) {
    printf("No se pudo procurar memoria\n");
    exit(-1);
}

```

Siempre libere la memoria con `free()`, cuando ya no la necesite.

IV. Enunciado

El trabajo consiste en un pipeline de procesamiento de imágenes de 4 etapas: lectura de imagen, zoom-in, suavizado y rotación. Es decir, la imagen de entrada debe pasar por cada una de las etapas entregando una imagen resultante para la etapa siguiente. Las etapas a realizar son:

IV.A. Lectura de Imagen

Las imágenes a procesar estarán en formato binario(float). En esta etapa, se debe realizar la lectura de archivo que se utilizará en Zoom-in.

IV.B. Zoom in

Consiste en una técnica de replicación de pixeles para toda imagen de $M \times N$, donde cada pixel es replicado según un factor de escala r que será pasado como parámetro de entrada, lo cual da origen a una imagen resultante de dimensiones $(M*r) \times (N*r)$.

Por ejemplo, se tiene una imagen A de 2×2 y un factor de replicación de 2, por lo que la imagen resultante tendrá dimensiones resultantes de 4×4 y quedará similar al siguiente ejemplo:

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \quad \text{Zoom-in}(A) = \begin{bmatrix} a & a & b & b \\ a & a & b & b \\ c & c & d & d \\ c & c & d & d \end{bmatrix}$$

Para una imagen de 2x2 con factor de replicación 3 se tiene:

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \quad \text{Zoom-in}(A) = \begin{bmatrix} a & a & a & b & b & b \\ a & a & a & b & b & b \\ a & a & a & b & b & b \\ c & c & c & d & d & d \\ c & c & c & d & d & d \\ c & c & c & d & d & d \end{bmatrix}$$

IV.C. Filtro Suavizado

Técnica que sirve para suavizar los bordes de una imagen, reducir el ruido o disminuir los cambios de intensidad en la imagen. En este caso se hará uso del filtro media, en el cual se obtiene el pixel de salida obteniendo la media aritmética de su vecindario.

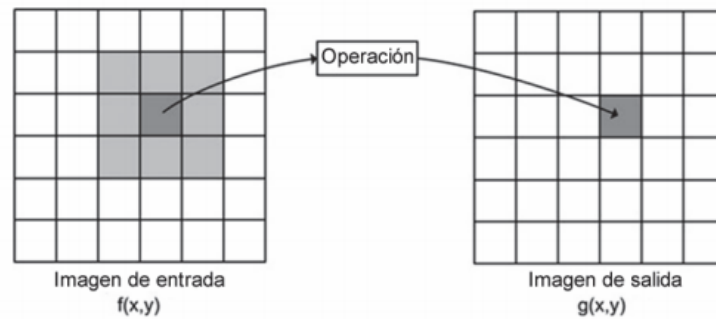


Figure 2. Ejemplo de obtención de pixel de salida

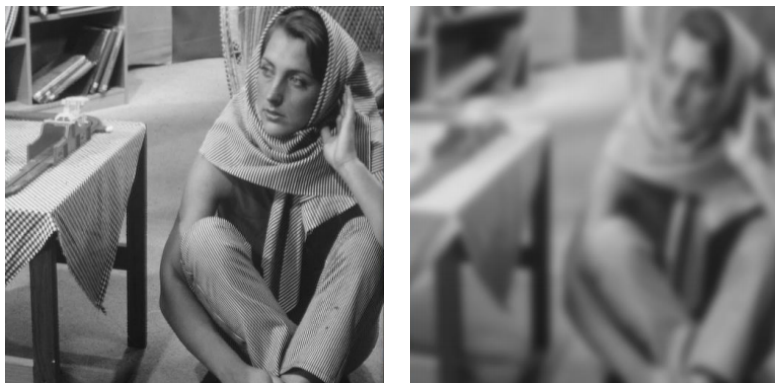


Figure 3. Ejemplo de imagen pasada por filtro de suavizado

IV.D. Rotación de Imagen

La imagen deberá ser rotada en sentido horario dependiendo de los grados entregados mediante los parámetros de ejecución. Los grados pueden ser cualquier número múltiplo de 90, es decir: 0, 90, 180, 270, etc. Dicho parámetro será entregado con el flag -g con getopt.



Figure 4. Ejemplo de imagen rotada 270° en sentido horario

V. Procesos

Para este laboratorio el pipeline se desarrollará con procesos que deben ser creados sí o sí con la función *fork()*. En este caso, cada etapa será un proceso, y estos se comunicarán mediante pipes, como se ilustra en la siguiente figura:

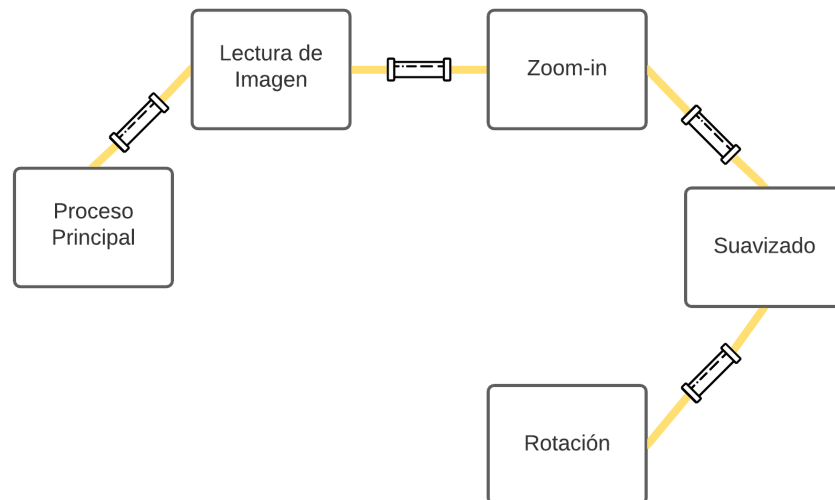


Figure 5. Arquitectura de Pipeline de procesos

Por lo tanto, mediante las funciones *fork()*, *pipes* y las funciones *exec()* se debe armar la arquitectura señalada, y los pipes solo comunicarán procesos adyacentes de forma unidireccional. Por ejemplo, la etapa de **Zoom-in** recibirá la imagen leída y los parámetros necesarios, después de realizar su etapa creará otro proceso para la etapa de **Suavizado**, entregándole mediante pipes la imagen con Zoom-in y el resto de argumentos, y así hasta la etapa de **Rotación**, donde se escribe la imagen final después de aplicar todas las etapas. El proceso principal debe terminar solo cuando todos los procesos del pipeline hayan terminado, para esto se debe hacer uso de la función *wait()*.

Finalmente, debe considerarse que cada etapa del Pipeline es un programa distinto, que posee su propio main, entradas y salidas por defecto(stdin y stdout), por lo que se podrían ejecutar por separado de forma independiente y deben ser considerados al momento de armar su Makefile (recordar que las funciones *exec* reciben archivos ejecutables), pero cada proceso necesita saber los parámetros, dimensiones de la imagen y el resultado de su etapa previa, por lo que debe recibir dichos argumentos desde el proceso anterior, para lo que deben hacer uso de la función *dup2()* al momento de definir las entradas y salidas de cada proceso.

VI. Parámetros de Entrada

Para este trabajo se debe entregar al menos un archivo llamado `lab2.c` y un *Makefile* que entregue archivos ejecutables para `lab2` y cada etapa del pipeline. Cada imagen estará en formato raw (`float`). La ejecución del programa tendrá los siguientes parámetros que deben ser procesados por `getopt()`:

- `-I filename` : especifica el nombre de la imagen de entrada
- `-O filename` : especifica el nombre de la imagen final resultante del pipeline.
- `-M número` : especifica el número de filas de la imagen
- `-N número` : especifica el número de columnas de la imagen
- `-r factor` : factor de replicación para Zoom-in
- `-g factor` : especifica en cuantos grados rotará la imagen: 0, 90, 180 o 270.
- `-b`: bandera que indica si se entregan resultados por consola. En caso de que se ingrese este flag deberá mostrar: etapas por las que va pasando la imagen, dimensiones de la imagen antes y después de aplicar zoom-in, ejecución finalizada.


Por ejemplo:

```
./lab2 -I imagen1.raw -O imagen1F.raw -M 512 -N 512 -r 2 -g 270 -b
```

VII. 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 **formato zip** 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. Para este caso debe compilar también los archivos `.c` correspondientes a cada etapa del Pipeline.



```
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 6. 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. Además, debe existir un archivo `.c` para cada etapa del pipeline, los cuales tienen su propio `main`. 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, éstas puede ser de distintas secciones, pero se debe avisar previamente vía correo a los ayudantes (Cristóbal Fernández y Benjamín Muñoz).

VIII. Visualización de las imágenes

Tanto las imágenes de entrada como de salida estarán en formato binario (`float`). Por lo tanto, no pueden ser visualizadas directamente por los software de imágenes. Luego, usaremos Matlab (disponible gratuitamente a todos los estudiantes de la Universidad) para visualizar imágenes. El siguiente script `verimagenraw()` realiza el trabajo.

```
function a = verimanageraw('filename', M, N)
    f = fopen(filename, 'r');
    a = fread(f, 'float');
    a = reshape(s, N, M); % si, es N, M, no M, N
    a = a'; % al trasponer queda de MxN
    figure; imagesc(a); axis('square');axis('off');
    colormap(gray)
```

Aunque el script abre una ventana y muestra la imagen, también retorna el arreglo (la imagen) como una matriz, la cual usted podría seguir usando. Por ejemplo, para visualizar con una colormap distinto:

```
a = verimagenraw('imagen_1.raw', 512, 512);
figure; imagesc(a); axis('square');axis('off');
colormap(hot)
```

IX. Fecha de entrega

Domingo 13 de Junio 2021, hasta las 23:55 hrs.