Sistemas Operativos 1/2020
Laboratorio 2

Profesores:
Cristóbal Acosta (cristobal.acosta@usach.cl)
Miguel Cárcamo (miguel.carcamo@usach.cl)
Fernando Rannou (fernando.rannou@usach.cl)
Ayudantes:
Isaac Espinoza (isaac.espinoza@usach.cl)
Marcela Rivera (marcela.rivera.c@usach.cl)

I. Objetivos Generales

Este laboratorio tiene como objetivo aplicar los conceptos de creación de procesos, la comunicación entre estos a través de *pipes* y envío de señales a través del uso de la función exec e, mediante el soporte de un sistema operativo basado en el núcleo Linux y el lenguaje de programación C.

II. Objetivos Específicos

1. Conocer y usar las funcionalidades de `getopt()` como método de recepción de parámetros de entradas.
2. Construir funciones de lectura y escritura de archivos binarios usando `open()`, `read()`, y `write()`.
3. Construir funciones de procesamiento de imágenes.
4. Practicar técnicas de documentación de programas.
5. Conocer y practicar uso de makefile para compilación de programas.
6. Crear procesos a través del uso de `fork()`.
7. Ejecutar programas usando alguna de los llamados de la familia `exec()`
8. Comunicar procesos mediante pipes, usando `pipes`

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** execle(**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**};
    execle(*"ls"*, *"ls"*, *"-l"*,(**const char** *)**NULL**, char *env[]);

- **int** execv(**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**};
    execv(*"ls"*,argv);

- **int** execvp(**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**};
    execvp(*"ls"*,argv);

- **int** execvpe(**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**};
    execvpe(*"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 computador 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():

```c
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    pid_t pid1, pid2;
    int status1, status2;

    if ( (pid1=fork()) == 0 )
    { /* hijo */
        printf("Soy el primer hijo (%d, hijo de %d)\n",  getpid(), getppid());
    }
    else
    { /*  padre */
        if ( (pid2=fork()) == 0 )
        { /* segundo hijo  */
            printf("Soy el segundo hijo (%d, hijo de %d)\n",  getpid(), getppid(
        }
        else
        { /* padre */
/* Esperamos al primer hijo */
            waitpid(pid1, &status1, 0);
/* Esperamos al segundo hijo */
            waitpid(pid2, &status2, 0);
            printf("Soy el padre (%d, hijo de %d)\n", getpid(), getppid());
        }
    }

    return 0;
}
```

Figure 1. Creando procesos con fork().

III.C. Funciones pipe

En sistemas operativos basados en UNIX, las funciones pipe son útiles para comunicar procesos relacionados (padre-hijo). Cabe destacar que la comunicación establecida es de una sola vía, es decir, un proceso escribe mientras que el receptor solo puede leer lo recibido. Si un proceso intenta leer antes de que algo sea escrito en el pipe, este se suspenderá hasta que se escriba en el pipe. Además, al utilizar dup2 se pueden cambiar los file descriptor y redirigir las salidas o entradas estándar de los procesos, pudiendo redirigir un printf() (que normalmente consiste en salida estándar por consola) hacia un pipe o archivo anteriormente definido. Por lo tanto, el uso de dup2 es útil cuando se desea comunicar procesos mediante pipes.

IV. Definición de etapas

La aplicación consiste en un *pipeline* de procesamiento de imágenes astronómicas. Cada imagen pasará por tres etapas de procesamiento tal que al final del *pipeline* se clasifique la imagen como satisfaciendo o no alguna condición a definir. El programa procesará varias imágenes, una a la vez.

Las etapas del *pipeline* son:

1. Lectura de imagen RGB
2. Conversión a imagen en escala de grises
3. Filtro de realce
4. Binarización de imagen
5. Análisis de propiedad
6. Escritura de resultados

IV.A. Lectura de imágenes

Las imágenes estarán almacenadas en binario con formato jpg. Habrá n imágenes y sus nombres tendrán el mismo prefijo seguido de un número correlativo. Es decir, los nombres serán imagen_1, imagen_2, ... , imagen_n.

Para este laboratorio se leerá una nueva imagen cuando la anterior haya finalizado el *pipeline* completo. Es decir, en el *pipeline* sólo habrá una imagen a la vez.

IV.B. Conversión de RGB a escala de grises

En una imagen RGB, cada pixel de la imagen está representado por tres números, que representan el componente de color rojo (Red), el de color verde (Green) y el de color azul (Blue). Para convertir una imagen a escala de grises, se utiliza la siguiente ecuación de luminiscencia:

$$Y = R * 0.3 + G * 0.59 + B * 0.11 \tag{1}$$

donde R, G, y B son los componentes rojos, verdes y azul, respectivamente.

IV.C. Aplicar filtro laplaciano

Este tipo de filtro se basa en un operador derivativo, por lo que acentúa las zonas que tienen gran discontinuidad en la imagen. Su fórmula es:

$$\bigtriangledown^2 f = \frac{\partial^2 f}{\partial^2 x} + \frac{\partial^2 f}{\partial^2 y} \tag{2}$$

Las derivadas segundas pueden aproximarse digitalmente por:

$$\frac{\partial^2 f}{\partial^2 x} \approx f(x+1, y) + f(x-1, y) - 2f(x, y) \tag{3}$$

$$\frac{\partial^2 f}{\partial^2 y} \approx f(x, y+1) + f(x, y-1) - 2f(x, y) \tag{4}$$

La correspondiente aproximación digital del Laplaciano resulta entonces:

$$\bigtriangledown^2 f(x, y) = f(x+1, y) + f(x-1, y) + f(x, y+1) + f(x, y-1) - 4f(x, y) \tag{5}$$

que corresponde a la máscara espacial:

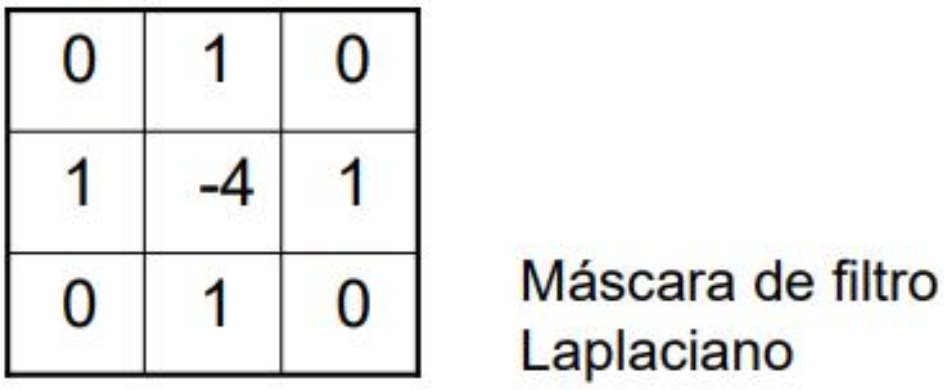

Figure 2. Mascara laplaciana

Lo anterior significa que el pixel [x,y] será igual a la suma total de la multiplicación de cada posición de la mascara con la respectiva posición de la imagen real.

Para los casos de borde puede mantener el valor original del pixel o bien puede rellenar con 0 el pixel faltante para aplicar la mascara normalmente.

Lo anterior se puede representar como la convolución, gráficamente el resultado de aplicar la mascara laplaciana se encuentra en la figura 3.

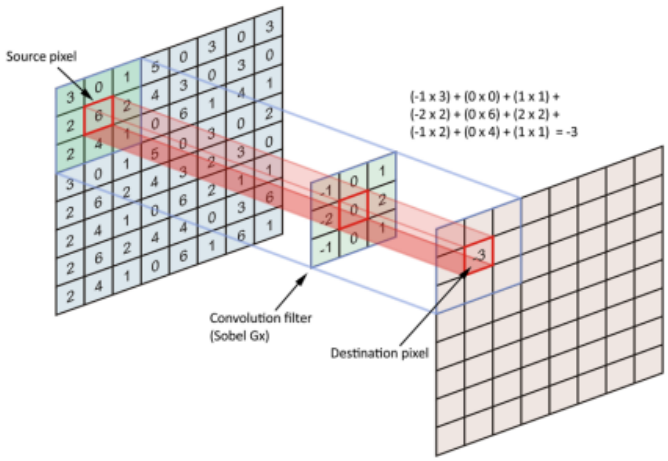

Figura 1: Operación de convolución para filtrado lineal en un punto de una imagen.

Figure 3. Implementando Mascara laplaciana

Para mayor información puede visitar el link https://www.fceia.unr.edu.ar/dip/Filtrado_Espacial.pdf

IV.D. Binarización de una imagen

Para binarizar la imagen basta con definir un umbral, el cual define cuáles píxeles deben ser transformados a blanco y cuáles a negro, de la siguiente manera. Para cada pixel de la imagen, hacer

```
si el pixel > UMBRAL
    pixel = 255
sino
    pixel = 0
```

Al aplicar el algoritmo anterior, obtendremos la imagen binarizada.

IV.E. Clasificación

Se debe crear una función que concluya si la imagen es *nearly black* (casi negra). Esta característica es típica de muchas imágenes astronómicas donde una pequeña fuente puntual de luz está rodeada de vacio u oscuridad. Entonces, si el porcentaje de píxeles negros es mayor o igual a un cierto umbral la imagen es clasificada como *nearly black*.

El programa debe imprimir por pantalla la clasificación final de cada imagen y escribir en disco la imagen binarizada resultante.

V. Uso de conceptos de procesos

Para ocupar funciones propias de procesos en este contexto, cada etapa del *pipeline* debe ejecutarse por un proceso único. Cada proceso se comunica con el siguiente por medio de *pipes*, entregándole su respectiva salida. A continuación se muestra un diagrama explicativo de esta estrategia:

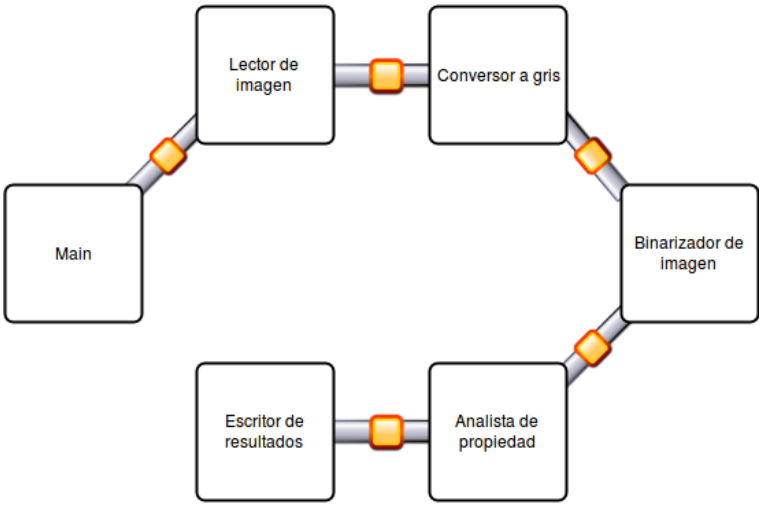

Figure 4. Diagrama de comunicación de procesos.

La creación de procesos debe realizarse sí o sí con uso de *fork()*. Además, es importante destacar que un *pipe* **sólo** comunica procesos que compartan el pipe correspondiente. Por ejemplo, Si un padre crea un pipe y luego crea un hijo, entonces ellos podrán comunicarse. Si un padre crea un pipe y luego dos hijos, ambos hijos podrán comunicarse también entre ellos.

Por lo tanto, mediante el uso de *fork()*, funciones de la familia *exec* y *pipes*, se debe lograr la arquitectura mostrada en la figura 4, respetando el orden de las etapas y el flujo de información entre los procesos, el cual debe ser unidireccional y solo entre etapas "adyacentes". Por ejemplo, el proceso Main sólo debería recibir los argumentos ingresados por consola al ejecutar el proyecto, crear un hijo **Lector de Imagen** y mediante un pipe debe entregarle los datos necesarios para que este hijo pueda completar su etapa de lectura. A su vez, el proceso **Lector de Imagen** debe crear al proceso **Conversor a Gris** y entregarle mediante un pipe su resultado final correspondiente a la lectura de la imagen, y así sucesivamente para cada otra etapa del pipeline. El proceso Main sólo debe terminar una vez que todos los otros procesos hayan acabado.

VI. Otros conceptos

Como ayuda, la manera de leer una imagen es la siguiente:

- Abrir el archivo con el uso de open(). Recuerde que este archivo debe contener la matriz (imagen).
- Leer la imagen con read().

Con los pasos previamente descritos, se puede leer la imagen y manipularla de tal forma que se pueda crear la matriz. Recuerde que la imagen tiene una estructura para poder obtener el valor de los pixeles, por lo que se pide investigar al respecto.

Para el caso de escribir la imagen resultante:

- Se debe escribir con write().
- Finalmente se debe cerrar el archivo con close().

Por último, el programa debe recibir la cantidad de imágenes a leer, el valor del umbral de binarización, el valor del umbral de negrura y una bandera que indica si se desea mostrar o no la conclusión final hecha por la tercera función. Por lo tanto, el formato getopt es:

- -c: cantidad de imágenes
- -u: UMBRAL para binarizar la imagen.
- -n: UMBRAL para clasificación

- -m: NOMBRE del archivo que contiene la máscara a utilizar de la siguiente forma:

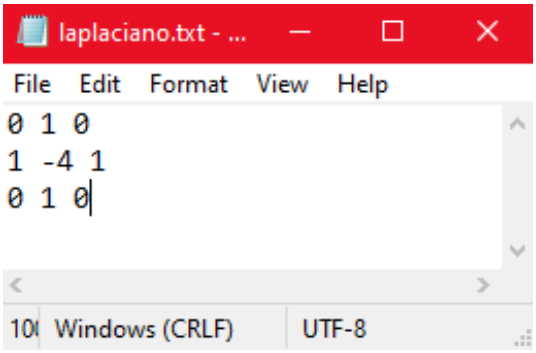

Figure 5. Ejemplo archivo máscara

  Sólo se podrán recibir máscaras de 3x3, con 3 filas y 3 columnas. Las columnas separadas por espacios y las filas separadas por saltos de línea.

- -b: bandera que indica si se deben mostrar los resultados por pantalla, es decir, la conclusión obtenida al leer la imagen binarizada.

Por ejemplo, la salida por pantalla al analizar 3 imágenes sería lo siguiente:

```
$ ./pipeline -c 3 -u 50 -n 50 -m mascara.txt -b
|     image     |     nearly black     |
|---------------|----------------------|
|   imagen_1    |         yes          |
|   imagen_2    |         no           |
|   imagen_3    |         no           |
```

Donde sólo la imagen_1 fue clasificada como *nearly black*.

VII. Entregables

Debe entregarse un archivo comprimido que contenga al menos los siguientes archivos:

1. *Makefile*: archivo make para que compile los programas.
2. Dos o mas archivos con el proyecto (*.c, *.h).
3. Clara documentación, es decir, entregar códigos debidamente comentados, en donde a lo menos se incluya le descripción de cada función, sus parámetros de entrada y salida. Los comentarios deben seguir la siguiente estructura:

```
//Entradas: Explicar entradas, qué representan y su tipo de dato
//Funcionamiento: Explicación breve del funcionamiento
//Salidas: Explicar el tipo de dato de la salida y lo que representa
funcion( ... ){
//comentario especifico 1

//comentario específico 2

//...

}
```

Además, también se evalúan buenas prácticas de programación, respecto al nombramiento de variables con nombres significativos, etc.

El archivo comprimido debe llamarse: RUTESTUDIANTE1_RUTESTUDIANTE2.zip

Ejemplo: 19689333k_186593220.zip

NOTA: los laboratorios son en parejas, las cuales deben ser de la misma sección. De lo contrario no se revisarán laboratorios.

VIII. Fecha de entrega

Domingo 26 de Julio.