Proyecto: Arbol de Procesos

Yefer Rodrigo Miguel Alvarado Tzul

Carné: 201731163

Universidad San Carlos de Guatemala

Centro Universitario de Occidente (CUNOC)

División de Ciencias de la Ingeniería

Sistemas Operativos 1

Ing. Juan Francisco Rojas Santizo

28 de marzo de 2021

Introducción

Linux es un sistema operativo que tiene la característica de poder administrar los procesos que se estén ejecutando dentro del mismo, los procesos se identifican con números enteros estos son llamados o denominados con pid, los procesos son ejecutados pero cuando se solicita la creación de uno nuevo este se vuelve padre, y los procesos creados a partir de este padre son llamados procesos hijos.

Marco Teórico

QT Creator

Qt Creator es un IDE multiplataforma programado en C++, JavaScript y QML creado por Trolltech el cual es parte de SDK para el desarrollo de aplicaciones con Interfaces Gráficas de Usuario (GUI por sus siglas en inglés) con las bibliotecas Qt, Los sistemas operativos que soporta en forma oficial son:

GNU/Linux 2.6.x, para versiones de 32 y 64 bits con Qt 4.x instalado. Además hay una versión para Linux con gcc 3.3.

Mac OS X 10.4 o superior, requiriendo Qt 4.x

Windows XP y superiores, requiriendo el compilador MinGW y Qt 4.4.3 para MinGW.

C/C++

C++ es un lenguaje de programación diseñado en 1979 por Bjarne Stroustrup. La intención de su creación fue extender al lenguaje de programación C mecanismos que permiten la manipulación de objetos. En ese sentido, desde el punto de vista de los lenguajes orientados a objetos, C++ es un lenguaje híbrido.

Posteriormente se añadieron facilidades de programación genérica, que se sumaron a los paradigmas de programación estructurada y programación orientada a objetos. Por esto se suele decir que el C++ es un lenguaje de programación multiparadigma.

Actualmente existe un estándar, denominado ISO C++, al que se han adherido la mayoría de los fabricantes de compiladores más modernos. Existen también algunos intérpretes, tales como ROOT.

El nombre "C++" fue propuesto por Rick Mascitti en el año 1983, cuando el lenguaje fue utilizado por primera vez fuera de un laboratorio científico. Antes se había usado el nombre "C con clases". En C++, la expresión "C++" significa "incremento de C" y se refiere a que C++ es una extensión de C.

CMake

CMake es una herramienta multiplataforma de generación o automatización de código. El nombre es una abreviatura para "cross platform make" (make multiplataforma); más allá del uso de "make" en el nombre, CMake es una suite separada y de más alto nivel que el sistema make común de Unix, siendo similar a las autotools.

CMake es un software libre y de código abierto multiplataforma para gestionar la automatización de la construcción del software utilizando un método independiente del compilador. Soporta jerarquías de directorios y aplicaciones que dependen de múltiples bibliotecas. Se utiliza en conjunción con entornos de construcción nativos como Make, Qt Creator, Ninja, Xcode de Apple, y Microsoft Visual Studio. Tiene dependencias mínimas, requiriendo sólo un compilador C++ en su propio sistema de construcción.

qmake

qmake es un programa que automatiza la creación de ficheros Makefiles. Los ficheros Makefiles son usados por el programa Make para compilar el código de manera automática. Make hace mucho más cómoda la compilación de proyectos relativamente medianos, pero a medida que la complejidad del proyecto aumenta los ficheros Makefile pueden llegar a crecer y complicarse demasiado. Y aquí es dónde entra en juego qmake, automatizando el proceso a un nivel superior.

qmake crea Makefiles que se adaptan a la plataforma deseada por lo que soporta diferentes sistemas operativos, entre los que están: Linux, Apple Mac OS X, Symbian, Android y Microsoft Windows.

qmake fue creado por Trolltech (ahora propiedad de Digia), y forma parte del framework de Qt. Aunque posee características adicionales que facilitan el desarrollo con Qt, automatiza la creación de los códigos moc (meta object compiler) y rcc (resource compiler), puede usarse para desarrollar cualquier proyecto software.

El comando ps

El comando ps sirve para mostrar información sobre procesos, sus sintaxis (POSIX) es la siguiente:

ps [-aA] [-G grouplist] [-o format] ... [-p proclist] [-t termlist] [-U userlist]

Muchas de las implementaciones que hacen los distintos vendedores del comando ps no cumplen con el estándar POSIX, un ejemplo claro es Sun Solaris (SO del servidor de prácticas de la escuela) que emplea la opción -e en lugar de -A para mostrar la información de todos los procesos. La versión larga del comando ps de Solaris muestra mucha información interesante sobre los procesos asociados a un terminal (la figura 1 muestra algunos).

Campo	Significado
UID	ID del usuario propietario del proceso
PID	ID del proceso
PPID	ID del padre del proceso
C	Utilización del preprocesador de C para la administración de procesos
STIME	Hora de comienzo
TTY	terminal de control
TIME	Tiempo acumulado de CPU
CMD	Nombre del comando

Ejemplo 1 El siguiente comando muestra, en formato largo, todos los procesos cuyo usuario propietario es i5599:

murillo:/export/home/cursos/so> ps -fu i5590

```
murillo:/export/home/cursos/so> ps -fu i5590
   UID   PID  PPID  C    STIME TTY      TIME CMD
 i5590 12246 12195  0 20:53:16 pts/15   0:02 clips
 i5590 12164 12150  0 20:30:56 pts/15   0:00 -ksh
 i5590 12194 12164  0 20:35:23 pts/15   0:00 -ksh
 i5590 12175 12152  0 20:31:12 pts/14   0:00 -ksh
 i5590 12195 12194  0 20:35:23 pts/15   0:00 /export/home/cursos/CCIA/bin/tcsh
 i5590 12176 12175  0 20:31:12 pts/14   0:00 /export/home/cursos/CCIA/bin/tcsh
 i5590 12205 12150  0 20:37:29 pts/16   0:00 -ksh
 i5590 12152 12150  0 20:30:39 pts/14   0:00 -ksh
 i5590 12192 12176  0 20:34:47 pts/14   0:01 /opt/sfw/bin/emacs oo.clp
```

Este comando es prácticamente equivalente a ps –ef | grep i5590.

Identificadores de usuarios y procesos

Las funciones C que se utilizan para obtener el identificador de proceso (PID) o el identificador de usuario (UID) son getpid, getppid y getuid:

#include <sys/types.h>

#include <unistd.h>

pid_t getpid(void);

pid_t getppid(void);

uid_t getuid(void);

- pid t es un entero largo con el ID del proceso llamante (getpid) o del padre del proceso llamante (getppid)
- uid t es un entero con el ID del usuario propietario del proceso llamante.
- En caso se error se devuelve -1.

Ejemplo 2 El siguiente programa imprime el identificadores del proceso llamante, del proceso padre y del pro-pietario:

#include <stdio.h>

#include <sys/types.h>

```
#include <unistd.h>
int main(void)
{
    printf("ID de proceso: %ld\n", (long)getpid());
    printf("ID de proceso padre: %ld\n", (long)getppid());
    printf("ID de usuario propietario: %ld\n", (long)getuid());
    return 0;
}
```

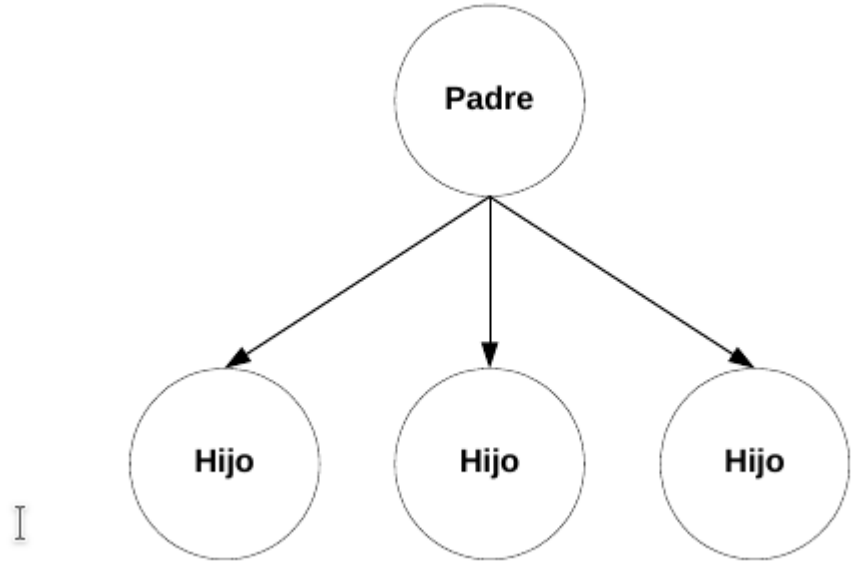
La llamada fork

En UNIX los procesos se crean a través haciendo una llamada fork al sistema. El nuevo proceso que se crea recibe una copia del espacio de direcciones del padre. Los dos procesos continúan su ejecución en la instrucción siguiente al fork:

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

- La creación de dos procesos totalmente idénticos no suele ser útil. Así, el valor devuelto por fork permite diferenciar el proceso padre del hijo, ya que fork devuelve 0 al hijo y el ID del hijo al padre.
- En caso de error devuelve -1.

La llamada fork crea procesos nuevos haciendo una copia de la imagen en la memoria del padre. El hijo hereda 1 la mayor parte de los atributos del padre, incluyendo el entorno y los privilegios. El hijo también hereda alguno de los recursos del padre, tales como los archivos y dispositivos abiertos. Las implicaciones de la herencia son mucho más complicadas de lo que en principio pueda parecer.




```
#include<stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main (void)
{
    int i;
    int padre;
    padre = 1;
    for (i=0; i < 3; i++)
    {
        if (padre == 1)
        {
            if (fork() == 0) /* Proceso hijo */
            {
                fprintf(stdout, "Éste es el proceso hijo con padre %ld\n",
                (long)getppid());
                padre = 0;
```

```
            }
            else /* Proceso padre */
            {
                fprintf(stdout, "Éste es el proceso padre con ID %ld\n",
                (long)getpid());
                padre = 1;
            }
        }
        return 0;
    }
```

Herencia de descriptores

Cuando fork crea un proceso hijo, éste hereda la mayor parte del entorno y contexto del padre, que incluye el estado de las señales, los parámetros de la planificación de procesos y la tabla de descriptores de archivo.

Hay que tener cuidado ya que las implicaciones de la herencia de los descriptores de archivos no siempre resultan obvias, ya que el proceso padre e hijo comparten el mismo desplazamiento de archivo para los archivos que fueron abiertos por el padre antes del fork.

Las llamadas wait y exit

Si queremos que el proceso padre espere hasta que la ejecución del hijo finalice tenemos que utilizar la llamada a wait o a waitpid junto con la llamada exit. La declaración de exit es la siguiente:

```
#include <stdlib.h>

void exit (int status);
```

- exit termina la ejecución de un proceso y le devuelve el valor de status al sistema. Este valor se puede ver mediante la variable de entorno ?. El return efectuado desde la función principal (main) de un programa C tiene el mismo efecto que la llamada a exit.

  Además de esto, la llamada a exit tiene también las siguientes consecuencias:

  - Si el proceso padre del que ejecuta la llamada a exit está ejecutando una llamada a wait, se le notifica la terminación de su proceso hijo y se le envian los 8 bits menos significativos de status. Con esta información el padre puede saber en qué condiciones ha terminado el proceso hijo.
  - Si el proceso padre no está ejecutando una llamada a wait, el proceso hijo se transforma en un proceso zombi.

  Las declaraciones de wait y waitpid son las siguientes:

  ```
  #include <sys/types.h>
  #include <sys/wait.h>
  pid_t wait(int *stat_loc);
  pid_t waitpid(pid_t pid, int *stat_loc, int options);
  ```

  - wait detiene al proceso llamante hasta que un hijo de éste termine o se detenga, o hasta que el proceso llamante reciba una señal.
  - En el caso de que cuando se solicita la espera el hijo ya ha terminado (o simplemente no existe ningún proceso hijo) la llamada wait devuelve el control de inmediato.
  - Cuando un proceso hijo termina, la llamada wait devuelve el ID de dicho hijo al padre. En caso contrario devuelve -1.

- stat loc es un puntero a entero que en caso de valer distinto de NULL la llamada wait devuelve el estado devuelto por el hijo (exit o return). En el estándar POSIX se definen las siguiente macros para analizar el estado devuelto por el proceso hijo:
    - WIFEXITED, terminación normal.
    - WIFSIGNALED, terminación por señal no atrapada.
    - WTERMSIG, terminación por señal SIGTERM.
    - WIFSTOPPED, el proceso está parado.
    - WSTOPSIG, terminación por señal SIGSTOP (que no se puede atrapar ni ignorar).

Ejemplo 4 En el siguiente programa el proceso padre espera a que termine la ejecución del hijo imprimiendo el código de terminación del mismo. El proceso hijo estará inactivo durante dos segundos y devolverá 0 si el ID del padre es mayor que su ID, en caso contrario, devuelve distinto de 0:

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
int main (void) {
5
pid_tint
int
childpid;
status=0;
result;
```

```
if ((childpid = fork()) == -1){
perror("Error en llamada a fork\n");
exit(1);
}
else if (childpid == 0)
{
result = getpid() < getppid() ;
fprintf(stdout, "Soy el proceso hijo (%ld) y
voy a devolver a mi padre (%ld)
el valor %d después
de esperar 2 segundos\n",
(long)getpid(),
(long)getppid(),
result);
sleep(2);
exit (result);
}
else
{
while( childpid != wait(&status));
fprintf(stdout, "Soy el proceso padre (%ld)
y mi hijo (%ld) me ha devuelto %d\n",
(long)getpid(),
(long) childpid,
status);
}
return (0);}
```

La llamada exec

La llamada fork crea una copia del proceso llamante. Muchas veces es necesario que el hijo ejecute un código totalmente distinto al del padre. En este caso, la familia de llamadas exec nos permitirá sustituir el proceso llamante por uno nuevo.

```
#include <unistd.h>
 int execl  (const char *path, const  char  *arg0,  ...,
             const char *argn, char * /*NULL*/);
 int execv  (const char *path, char *const argv[]);
 int execle (const char *path, const char  *arg0,  ...,
             const char *argn, char * /*NULL*/, char *const envp[]);
 int execve (const char *path, char *const argv[], char *const envp[]);
 int execlp (const char *file, const char  *arg0,  ...,
             const char *argn, char * /*NULL*/);
 int execvp (const char *file, char *const argv[]);
```

- Las seis variaciones de la llamada exec se distinguen por la forma en que son pasados los argumentos de la línea de comando y el entorno, y por si es necesario proporcionar la ruta de acceso y el nombre del archivo ejecutable.
- Las llamadas execl (execl, execle y execlp) pasan la lista de argumentos de la línea de comando como una lista y son útiles sólo si se conoce a priori el número de éstos.
- Las llamadas execv (execvl, execvp y execve) pasan la lista de argumentos en una cadena (un array de punteros a char) y son útiles cuando no se sabe el número de argumentos en tiempo de compilación.
- path es la ruta (completa o relativa al directorio de trabajo) de acceso al archivo ejecutable.
- arg es el argumento i–ésimo (sólo en llamadas execl).
- argv es una cadena con todos los argumentos (sólo en llamadas execv).
- envp es una cadena con el entorno que se le quiere pasar al nuevo proceso.

La terminación de procesos

Cuando termina un proceso (normal o anormalmente), el SO recupera los recursos asignados al proceso terminado, actualiza las estadísticas apropiadas y notifica a los demás procesos la terminación.

Las actividades realizadas durante la terminación de un proceso incluyen la cancelación de temporizadores y señales pendientes, la liberación de los recursos de memoria virtual asi como la de otros recursos del sistema ocupados por el proceso.

Cuando un proceso termina, sus hijos huérfanos son adoptados por el proceso init, cuyo ID el 1. Si un proceso padre no espera a que sus hijos terminen la ejecución, entonces éstos se convierten en procesos zombies y tiene que ser el proceso init el que los libere (lo hace de manera periódica).

Las llamadas a exit y exit 3 se utilizan para terminar de forma normal un proceso. La función exit lo que hace es llamar a los manejadores de terminación del usuario (si existen) y después llamar a exit.

pthread.lib: La librería de hilos POSIX de Linux

Para escribir programas multihilo en C podemos utilizar la librería pthread que implementa el standard POSIX (Portable Operating System Interface). Para ello en nuestro programa debemos incluir algunas cabeceras y a la hora de compilar es necesario linkar el programa con la librería. Veamos con un ejemplo lo que estamos diciendo. Hagamos un programa que escriba “Hola Mundo” utilizando para ello dos hilos distintos, uno para la palabra Hola y otro para la palabra Mundo. Además, hagamos que cada palabra se escriba muy lentamente para ver bien el efecto.

Como se puede ver en el listado 2, hemos escrito una función para cada tarea que queremos realizar en un hilo. El estándar POSIX impone que toda función que vaya a ejecutarse en un hilo debe tener un parámetro de entrada de tipo puntero a void (void *) y tiene que devolver también un puntero a void.

Las funciones imprimen un mensaje cada una, pero lo hacen letra a letra. Además, tras cada impresión se bloquean durante un tiempo usando la función usleep(ms), la cual detiene el hilo que la ejecuta durante más microsegundos.

En el programa principal tenemos dos variables de tipo pthread t que van a almacenar el identificador de cada uno de los dos hilos que vamos a crear. El identificador de un hilo es necesario guardarlo ya que, una vez que un hilo comienza a funcionar, la única forma de controlarlo es a través de su identificador.

El tipo pthread t está definido en la cabecera pthread.h, por eso es necesario incluirla al principio del programa.

El lanzamiento o creación de los hilos se realiza con la función pthread create que también está definida en pthread.h. Esta función crea un hilo, inicia la ejecución de la función que se le pasa como tercer argumento dentro de dicho hilo y guarda el identificador del hilo en la variable que se le pasa como primer argumento. Por lo tanto, el primer argumento será la dirección (&) de la variable de tipo pthread t que queramos que guarde el identificador del hilo creado. El tercer argumento será el nombre de la función que queramos que se ejecute dentro de dicho hilo.

El segundo parámetro se puede utilizar para especificar atributos del hilo, pero si usamos NULL el hilo se crea con los parámetros normales. Si deseas más información sobre los

atributos de los hilos pregunta al profesor. Más adelante veremos la utilidad del cuarto parámetro.

Esperando a la terminación de un hilo

Al ejecutar el programa anterior, es bastante improbable que se llegue a ver nada en la consola. La razón es que los hilos que se crean en el programa principal terminan automáticamente cuando el programa principal termina. Si en el programa principal simplemente lanzamos los dos hilos y después escribimos “Fin”, lo más probable es que los hilos no lleguen a ejecutarse completamente o incluso que no lleguen ni a terminar de arrancar antes de que el programa principal escriba y termine.

Evidentemente es necesario un mecanismo de sincronización que permita esperar a la terminación de un hilo. Este mecanismo es el que implementa la función pthread_join que también está definida en pthread h. Para comprobarlo añadamos las siguientes lineas al programa anterior justo antes de la sentencia para escribir “Fin” en el programa principal.

Pasando parámetros a los hilos

Las funciones que se ejecutan en los hilos tienen acceso a las variables globales declaradas antes que la propia función. Pero si queremos pasar un parámetro concreto a la función que se ejecuta en un hilo dicho parámetro debe ser de tipo puntero a void ya que es así como tiene que estar declarada la función.

Afortunadamente, en C una variable de tipo puntero a void puede guardar punteros a cualquier tipo y, utilizando el “casting”, podemos hacer que el compilador lo acepte sin dar error. Por lo tanto, para pasar un argumento de cualquier tipo a una función ejecutada en un hilo, hay que pasarle un puntero a dicha variable enmascarado como puntero a void. Dentro de la función, hay que hacer la operación contraria, es decir, como la función lo que recibe es

un puntero a void, habrá que convertirlo a un puntero al tipo que queramos (el mismo que el pasado) para poder utilizarlo correctamente.

Ahora bien, dado que la función no se ejecuta directamente sino a través de la llamada a pthread_create, el argumento se pasa a la función a través del cuarto parámetro de pthread_create. Como ejemplo, veamos como hacer el programa anterior pero usando una sóla función que se lanza en dos hilos distintos parametrizada con la cadena a mostrar.

Liberación de recursos

Los recursos asignados por el sistema operativo a un hilo son liberados cuando el hilo termina. La terminación del hilo se produce cuando la función que está ejecutando termina, cuando ejecuta un return o cuando ejecuta la función pthread exit. Si la función no ejecuta ni return ni pthread exit, se ejecuta automáticamente un return 0. Teniendo en cuenta que las funciones que se ejecutan en hilos deben devolver un puntero a void, en este caso, el valor devuelto es un puntero con valor NULL.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main() {
    struct timeval t0, t1;
    int i = 0;
    int id = -1;
    gettimeofday(&t0, NULL);
    for ( i = 0 ; i < 100 ; i++ ) {
        id = fork();
        if (id == 0) return 0;
    }
    if (id != 0) {
        gettimeofday(&t1, NULL);
        unsigned int ut1 = t1.tv_sec*1000000+t1.tv_usec;
        unsigned int ut0 = t0.tv_sec*1000000+t0.tv_usec;
        printf("%f\n", (ut1-ut0)/100.0); /* Tiempo medio en microsegundos */
    }
}
```

DESCRIPCIÓN DEL PROBLEMA

Se necesita de una aplicación que pueda crear y modificar árboles, mediante instrucciones, donde las hojas, ramas y los tallos puedan trabajar como un proceso independiente donde cambiaran de color constantemente.

El tiempo de espera para realizar un cambio de color es de 1s, cada hoja, rama y tallo seran procesos en paralelo, donde se pueden crear de uno a diez tallos, cada uno de estos permitira la creacion de cinco ramas como maximo y un minimo de una, y cada rama debera de tener como minimo una hoja y maximo diez, ya que estos son procesos trabajando paralelamente deberan de funcionar hasta que sea eliminado, y para observar el funcionamiento deberan de ir cambiando de color, a continuacion se describen los colores:

Tallo: Gris, Negro

Rama: multicolor

Hojas: Verde, Cafe

Los comandos a utilizar son los siguientes:

(P, a, b, c) Instrucción de crear o reestructurar ramas.

(P, a, b) Instrucción de eliminar ramas y no las hojas.

(P, a, 0) La planta se seca, únicamente quedará el tallo.

(P, a) Intrusión de crear una planta.

Donde:

a = Número de planta o tallo

b = Número rama

c = Número de hoja (Se le asigna a todas las ramas)

Comando para mostrar una planta:

(M, a) Muestra la planta o tallo “a” en pantalla, queda y las modificaciones realizadas en otras plantas se verán reflejadas hasta que se muestran.

Comando para mostrar la estructura de los procesos mediante un archivo de texto:

(I, a) Imprime la planta o tallo “a” en un archivo de texto con el número de procesos y el árbol de procesos actual (pstree).

REQUERIMIENTOS TÉCNICOS

- Requerimientos Mínimos de Hardware
- Procesador: Core i5
- Memoria RAM: 4 Gigabytes (GB)
- Almacenamiento: 50 Gigabytes (GB)
- Requerimientos Mínimos de Software
- Sistema Operativo: GNU / Linux (Distribución derivada de Debian)
- Ubuntu 20.04
- QT Creator
- Compilador de C/C++
- CMake
- QMake

Para poder ejecutar el programa localice el archivo “Arbol”, una vez localizado desde la terminal escriba el siguiente comando “./Arbol”.

Conclusión

Cuando se genera un proceso hijo, este es una copia pero este puede llegar a tener atributos distintos que los de su padre.

Es posible hacer que dos procesos hijos actúen de manera distinta, esto se puede realizar a través de la instancia exect.

Los procesos que se deseen crear pueden ser omitidos haciendo uso de bandera padre, de esta manera es posible generar procesos n y no sólo binario.

Un proceso hijo puede permanecer aún si el padre fue eliminado, ya que este proceso toma la función de padre, y queda anclado al systemctl como padre principal.

E-grafía

1. fork() in C - GeeksforGeeks. (2021). Retrieved 27 March 2021, from https://www.geeksforgeeks.org/fork-system-call/
2. linux c pthread multiproceso - programador clic. (2021). Retrieved 27 March 2021, from https://programmerclick.com/article/35581399098/
3. MULTIPROCESO, MULTITAREA Y MULTIUSUARIO. (2021). Retrieved 27 March 2021, from https://sisoperativoutp2587.wordpress.com/2016/08/12/multiproceso-multitarea-y-multiusuario/
4. pipe() System call - GeeksforGeeks. (2021). Retrieved 27 March 2021, from https://www.geeksforgeeks.org/pipe-system-call/
5. Procesos e hilos en C de Unix/Linux. (2021). Retrieved 27 March 2021, from http://www.chuidiang.org/clinux/procesos/procesoshilos.php