

Universidad de San Carlos de Guatemala.

Facultad de ingeniería.

Nombre: JONATAN LEONEL GARCIA ARANA

Carné: 202000424

LABORATORIO SISTEMAS OPERATIVOS II

Sección: A

PRACTICA 1

# MANUAL TECNICO

## **Introducción**

El presente documento describe la practica destinada a comprender el funcionamiento de las llamadas al sistema en el contexto del sistema operativo Linux MINT. Se abordarán temas relacionados con la gestión de procesos, así como la capacidad de interceptar estas llamadas para monitorear y registrar su actividad.

El objetivo principal es desarrollar un programa en lenguaje C que actúe como proceso padre y cree dos procesos hijos. Estos procesos hijos realizarán operaciones de manejo de archivos sobre un archivo específico, mientras que el proceso padre interceptará y registrará las llamadas al sistema realizadas por los hijos.

## Requerimientos

Requerimientos para Linux Mint:

- Sistema operativo: Linux Mint (versión específica, por ejemplo, Linux Mint 20.3 "Una")
- Procesador: Procesador compatible con arquitectura de 32 bits (x86) o 64 bits (x64)
- Memoria RAM: Se recomienda al menos 1 GB de RAM para compilaciones simples y desarrollo básico en C. Para proyectos más complejos, se recomiendan 2 GB o más.
- Espacio en disco: Se recomienda un mínimo de 16 GB de espacio disponible en disco para la instalación del sistema operativo y las herramientas de desarrollo.
- Herramientas de desarrollo: Compilador de C compatible con Linux Mint (se puede instalar GCC, por ejemplo)

Instalación y Configuración de Herramientas de Desarrollo:

1. Compilador GCC: El compilador GCC generalmente viene preinstalado en la mayoría de las distribuciones de Linux, incluyendo Linux Mint. Si no está instalado, puede instalarlo a través del gestor de paquetes de su distribución con el siguiente comando:  
**sudo apt-get install build-essential**
2. Editor de texto o IDE: Puede utilizar cualquier editor de texto de su preferencia, como Vim, Emacs, o instalar un IDE como NetBeans, Code::Blocks, o Visual Studio Code

## INICIANDO EJECUCION DEL PROGRAMA

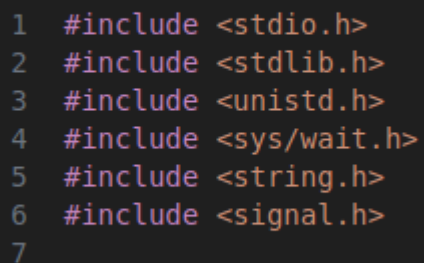
Para esta práctica estaremos utilizando 3 archivos los cuales son los siguientes:  
Padre.C: se encargará de la creación de los procesos hijos y de la llamada a systemtrap

Hijo.C: tendrá la lógica necesaria para simular los procesos para cuando el padre cree los hijos, se mande a llamar este archivo

Trace.stp: archivo de systemtrap, en el cual estaremos monitoreando todas las llamadas a los procesos hijos (Read, Write, Seek)

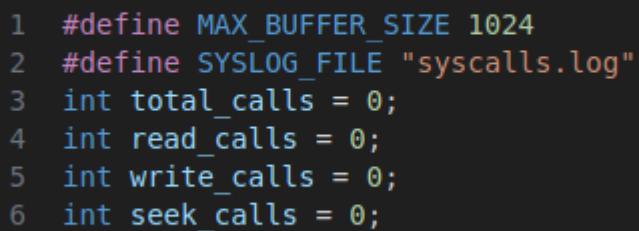
### Archivo Padre:

librerías utilizadas:



```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <sys/wait.h>
5 #include <string.h>
6 #include <signal.h>
7
```

Variables:



```
1 #define MAX_BUFFER_SIZE 1024
2 #define SYSLOG_FILE "syscalls.log"
3 int total_calls = 0;
4 int read_calls = 0;
5 int write_calls = 0;
6 int seek_calls = 0;
```

Inicio del Código: en esta parte crearemos unos contadores, los cuales nos ayudaran a llevar el control de los PID, y declaramos el método de signal. Este método es el que obtiene la llamada cuando se hace ctrl + C, mas adelante explicaremos la función que manda a llamar.

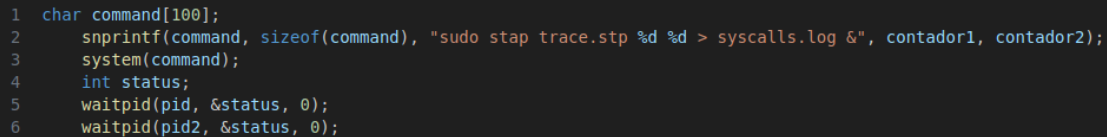
```
1  int main() {
2      int contador1 = 0;
3      int contador2 = 0;
4      signal(SIGINT, sigint_handler);
5      pid_t pid2 = 0;
6      FILE *fp;
7      pid_t pid = fork();
```

Luego declaramos el primero fork()); que es el primer hijo que estaremos utilizando en el programa

```
1  if (pid == -1) {
2      perror("Error al hacer fork");
3      exit(EXIT_FAILURE);
4  } else if (pid > 0) {
5      // Estamos en el proceso padre
6      //printf("Proceso hpadre creado con PID: %d\n", getpid());
7      contador1 = pid; // Guardamos el PID del primer hijo
8
9      pid_t pid2 = fork(); // Creamos el segundo proceso hijo
10     if (pid2 == -1) {
11         perror("Error al hacer fork");
12         exit(EXIT_FAILURE);
13     } else if (pid2 > 0) {
14         // Estamos en el proceso padre
15         contador2 = pid2; // Guardamos el PID del segundo hijo
16     } else {
17         // Estamos en el proceso hijo 2
18         //printf("Proceso hijo 2 creado con PID: %d\n", getpid());
19         char *args[] = {"0", NULL};
20         execv("/home/oem/Desktop/S02_202000424/Practica 1/hijo.bin", args);
21         perror("Error en execv");
22         exit(EXIT_FAILURE);
23     }
24 } else {
25     // Estamos en el proceso hijo 1
26     //printf("Proceso hijo 1 creado con PID: %d\n", getpid());
27     char *args[] = {"0", NULL};
28     execv("/home/oem/Desktop/S02_202000424/Practica 1/hijo.bin", args);
29     perror("Error en execv");
30     exit(EXIT_FAILURE);
31 }
32
```

En esta parte del código, tendremos la lógica para ejecutar bien nuestros procesos hijos, entra a un if el cual verifica si es -1, esto para encontrar algún error si es que lo hay, luego sabemos que si el pid que viene es mayor a 0 pues es el proceso padre, entonces cuando entra a este else if, creamos el segundo proceso hijo y siguiendo la misma lógica para el primer hijo pues verificamos si el pid2 que viene es padre o hijo y si no es mayor a 0 es el proceso hijo, entonces en el else del proceso hijo, mandamos a ejecutar el archivo hijo.bin, esto lo hacemos con `execv`. Ambos hijos llaman al mismo archivo, ya que estaremos simulando el mismo proceso.

Afuera de todo este código tenemos la llamada a `systemtrap`



```
1 char command[100];
2 snprintf(command, sizeof(command), "sudo stap trace.stp %d %d > syscalls.log &", contador1, contador2);
3 system(command);
4 int status;
5 waitpid(pid, &status, 0);
6 waitpid(pid2, &status, 0);
```

En esta parte tenemos un char de 100 caracteres, el cual utilizaremos para correr el comando, luego utilizamos un `snprintf`, que nos ayuda a hacer comando, que sería `sudo stap trace.stp`, el cual va redirigir toda su salida al archivo `syscalls.log`, este archivo `syscalls.log` va recibir dos parámetros, el cual es el pid del hijo uno en este caso `contador1` y el pid del hijo 2 en este caso `contador2`, con el `system(command)` mandamos a ejecutar el `systemtrap` y luego con un `waitpid` esperamos a que terminen los procesos hijo.

### **Método hijo que escribe en el archivo:**

En esta parte, tenemos un método llamado `hijo_escribir`, el cual recibe el file, que sería el open del archivo `practica1.txt`, se crea un char de 8 caracteres y con un `for`, el cual tiene un `random`, ingresa a escribir números o a escribir números, luego con un `write` escribir en el archivo

```

1
2 void hijo_escribir(int file_descriptor) {
3     char line[9];
4     for (int i = 0; i < 8; ++i) {
5         if (rand() % 2 == 0) {
6             // Genera un número aleatorio
7             line[i] = '0' + rand() % 10;
8         } else {
9             // Genera una letra aleatoria
10            line[i] = 'a' + rand() % 26;
11        }
12    }
13    line[8] = '\0';
14    write(file_descriptor, line, strlen(line));
15 }

```

### Método que lee el archivo:

en esta parte tenemos un char el cual nos ayuda a leer las 8 posiciones del archivo

```

1 void hijo_leer(int file_descriptor) {
2     char buffer[9];
3     read(file_descriptor, buffer, 8);
4     buffer[8] = '\0';
5 }

```

### Método que redirecciona a la primera posición:

Con un lseek podemos redireccionar a cualquier lugar del archivo el apuntador, en este caso lo redireccionamos a la posición 0

```

1 void hijo_seek(int file_descriptor) {
2
3     lseek(file_descriptor, 0, SEEK_SET);
4 }

```

### Método que se ejecuta cuando se manda a llamar ctrl + C:

En esta parte del código tendremos la lógica para obtener el conteo de los procesos que se hicieron, para esto utilizamos un fopen, el cual abre el archivo syscalls.logs, y recorremos el archivo con un while, en el cual si encuentra la palabra read, write o seek, puede suma la variable a donde ingresaría el if, luego con una serie de pasos simples, obtenemos el conteo total y ya imprimos en consola nuestros números de procesos.

```
1 void sigint_handler(int signum) {
2
3     char buffer[MAX_BUFFER_SIZE];
4     FILE *fp_calls = fopen(SYSLOG_FILE, "r");
5     if (fp_calls == NULL) {
6         perror("Error al abrir el archivo de llamadas");
7         exit(EXIT_FAILURE);
8     }
9
10    while (fgets(buffer, sizeof(buffer), fp_calls) != NULL) {
11        if (strstr(buffer, "read") != NULL) {
12            read_calls++;
13        } else if (strstr(buffer, "write") != NULL) {
14            write_calls++;
15        } else if (strstr(buffer, "seek") != NULL) {
16            seek_calls++;
17        }
18    }
19
20    // Cerrar el archivo después de leerlo
21    fclose(fp_calls);
22
23    // Calcular el total de llamadas
24    total_calls = read_calls + write_calls + seek_calls;
25
26    // Imprimir resultados
27    printf("Número total de llamadas al sistema: %d\n", total_calls);
28    printf("Número de llamadas al sistema por tipo:\n");
29    printf("Read: %d\n", read_calls);
30    printf("Write: %d\n", write_calls);
31    printf("Seek: %d\n", seek_calls);
32    // Terminar la ejecución del programa
33    exit(EXIT_SUCCESS);
34 }
```



## Funcion main del archivo Hijo

en esta función, creamos el archivo practica1.txt en el cual se hace los procesos de escribir, leer y redireccionar el puntero, entonces con un while hacemos que se repita el código, luego con un random de 1 a 3, ponemos un sleep para que espere un tiempo aleatorio y con un switch y un random, elije que función manda a llamar, que en este caso seria hijo\_escribir, hijo\_leer e hijo\_seek, esto se repetiría hasta que se termine el proceso hijo

```
1  int main() {
2      int file_descriptor = open("practica1.txt", O_RDWR | O_CREAT | O_TRUNC, 0644);
3      if (file_descriptor == -1) {
4          perror("Error al abrir el archivo");
5          return 1;
6      }
7      while (1) {
8          int rand_call2 = rand() % 3;
9          sleep(rand() % 3 + 1); //
10         switch (rand_call2) {
11             case 0:
12                 hijo_escribir(file_descriptor);
13                 break;
14             case 1:
15                 hijo_leer(file_descriptor);
16                 break;
17             case 2:
18                 hijo_seek(file_descriptor);
19                 break;
20         }
21     }
22     // Cierra el archivo
23     close(file_descriptor);
24     return 0;
25 }
```

## Métodos en el archivo de systemtrap:

Systemtrap es una herramienta que nos ayuda a llevar el control de procesos en Linux, entonces con systemtrap podemos crear un script el cual nos ayudara a saber que procesos se hacen con el respectivo id

### Método que controla los read:

Systemtrap nos da la opción de controlar las llamadas que se hacen, entonces con un probe syscall.read, podemos obtener todos los read, pero para buscar los read que de los pid que queremos podemos poner un if, el cual con los parámetros que mandamos podemos buscar si el pid pertenece al del hijo, luego ya solo imprimos lo que nos pedía la práctica, que era la hora y el número de pid de tal proceso

```
1 probe syscall.read {
2   if (pid() == $1 || pid() == $2) {
3     system("sleep 0.1");
4     system(sprintf("printf \"Proceso %d: %s (\n" && date +"%%Y-%%m-%%d %%H:%%M:%%S\" && printf \"\\n\\n\"", pid(), "read"))
5     system("sleep 0.1");
6   }
}
```

### Método que controla los write:

De igual manera que el método de read, lo único que cambia con este método es el syscall.read, en vez de ser read, pues utilizaríamos la palabra write, la cual buscaría todos los procesos write, que concuerden con el pid de los procesos hijos

```
1 probe syscall.write {
2   if (pid() == $1 || pid() == $2) {
3     system("sleep 0.1");
4     system(sprintf("printf \"Proceso %d: %s (\n" && date +"%%Y-%%m-%%d %%H:%%M:%%S\" && printf \"\\n\\n\"", pid(), "write"))
5     system("sleep 0.1");
6   }
7 }
8
```

### Método que controla los seek:

Como con los métodos anteriores, lo único que cambieremos será el nombre del método, que en este caso seria syscall.lseek, el cual leerá todos los seek que hagamos cuando concuerde los pid de los procesos hijos

```
1
2 probe syscall.lseek {
3   if (pid() == $1 || pid() == $2) {
4     system("sleep 0.1");
5     system(sprintf("printf \"Proceso %d: %s (\n" && date +"%%Y-%%m-%%d %%H:%%M:%%S\" && printf \"\\n\\n\"", pid(), "seek"))
6     system("sleep 0.1");
7   }
8 }
9
10
```