UNIVERSIDAD AUTÓNOMA DE CHIHUAHUA

PROCESOS DEL KERNEL EN LINUX

NOMBRE DE LA MATERIA
Sistemas Operativos

Unidad de Aprendizaje
Parcial 2

GRUPO: 5HW1

NOMBRE DEL ALUMNO:
Juan Carlos Solis Montoya 367560
Mara Ximena Marquez Salcido 367668
Jose Fernando Martinez Garcia 355597
Arath Eduardo Saenz Castelo 367951

NOMBRE DEL MAESTRO:
Iván Chavero

Chihuahua, Chih.
FECHA DE ENTREGA LÍMITE: 24/10/2024

Código:

```
#include <stdio.h>

#include <stdlib.h>

#include <dirent.h> //Permite manipular directorios, para leer el contenido de /proc

#include <ctype.h> //Contiene funciones para comprobar y convertir caracteres como isdigit

#include <string.h>

// Funciones

//Se encarga de leer el archivo cmdline de un proceso y devuelve si es proceso de kernel o de usuario

char fetch_process(char *filename, char *procname);



//Aquí se obtiene el nombre del proceso kernel, leyendo /proc/[PID]/comm, PID se escarpia refiriendo al ID del proceso

void fetch_kernel_process_name(char *filename, char *procname);

int main(int argc, char *argv[]) {

    // Inicialización de variables

    struct dirent *entry;

//DIR es una estructura que se utiliza para representar un directorio abierto. En este caso, pDir se usa para almacenar el resultado de abrir el directorio /proc y luego recorrer su contenido.

    DIR *pDir;

    char proc_dir[] = "/proc";

    char proc_id_dir[1024];

    char cmdline[1024];

    char isKernel;
```

```
// Abre el directorio /proc
pDir = opendir(proc_dir);
if(pDir == NULL) {
    printf("No se puede abrir el directorio '%s'\n", proc_dir);
    return 1;
}
// Recorre cada entrada en el directorio /proc
while((entry = readdir(pDir)) != NULL) {
// Solo procesa directorios cuyo primer carácter es un dígito
    if (!isdigit(entry->d_name[0]))
        continue;

    // Se obtiene la ruta al archivo cmdline del proceso
    sprintf(proc_id_dir, "/proc/%s/cmdline", entry->d_name);
    // Función que obtiene el nombre del proceso y determina si es de usuario o de kernel
    isKernel = fetch_process(proc_id_dir, cmdline);

    // Si es un proceso de kernel, obtiene su nombre del archivo 'comm'
    if (isKernel == 'K') {
        sprintf(proc_id_dir, "/proc/%s/comm", entry->d_name);
        fetch_kernel_process_name(proc_id_dir, cmdline);
        printf("[K] %s:%s", entry->d_name, cmdline);
    } else {
        // Imprime el nombre del proceso de usuario en caso de no ser kernel
        printf("[U] %s:%s\n", entry->d_name, cmdline);
```

```
        }
    }
    closedir(pDir);
    printf("\n[U]ser process | [K]ernel process\n");
    return 0;
}
char fetch_process(char *filename, char *procname) {
    // Función para obtener el nombre del proceso
    FILE *file_pointer; //Maneja el puntero al archivo
    char *line = NULL; //Almacena la línea leída
    size_t len = 0; //Tamaño del buffer
    procname[0] = '\0'; //Vacía el buffer de procname

    // Abre el archivo cmdline del proceso
    if (filename)
        file_pointer = fopen(filename, "r");

    // Verifica si el archivo se abrió correctamente
    if (file_pointer == NULL) {
        printf("Error while opening file '%s'\nExit code 2\n", filename);
        exit(2);
    }

// Lee el contenido del archivo línea por línea, getline asigna de forma dinámica la
memoria necesaria para almacenar la línea leída del archivo
```

```
    while (getline(&line, &len, file_pointer) != -1) {
        // Concatena el contenido a procname
        sprintf(procname, "%s%s", procname, line);
    }

    fclose(file_pointer);
    free(line);

    // Si procname está vacío, el proceso es del kernel; de lo contrario, es un proceso de usuario
    if (procname[0] == '\0')
        return 'K';
    else
        return 'U';
}

void fetch_kernel_process_name(char *filename, char *procname) {
    // Función para obtener el nombre del proceso de kernel desde /proc/[pid]/comm
    FILE *file_pointer;
    char *line = NULL;
    size_t len = 0;
    procname[0] = '\0';

    // Abre el archivo comm del proceso
    if (filename)
```

```
        file_pointer = fopen(filename, "r");


    // Verifica si el archivo se abrió correctamente
    if (file_pointer == NULL) {
        printf("Error while opening file '%s'\nExit code 2\n", filename);
        exit(2);
    }


    // Lee el contenido del archivo línea por línea
    if (getline(&line, &len, file_pointer) != -1) {
        // Copia el contenido a procname y lo guarda en el buffer
        sprintf(procname, "%s", line);
    }
    fclose(file_pointer);
    free(line);
}
```

Estructura del Código

1. **Inclusión de Librerías**:
    - Se incluyen librerías estándar de C que permiten trabajar con archivos, directorios y funciones de manipulación de cadenas.
2. **Prototipo de Función**:

    - char fetch_process(char *filename, char *procname);: Declara una función que se encargará de leer el nombre del proceso a partir del archivo `cmdline`.
    - char fetch_kernel_process_name(char *filename, char *procname);: Se encarga de obtener un nombre de proceso kernel
3. **Función `main`**:
    - **Variables**: Se declaran variables para manejar directorios, almacenar rutas y nombres de procesos.
    - **Abrir el Directorio `/proc`**: Intenta abrir el directorio `/proc`. Si no se puede abrir, imprime un error y termina el programa.
    - **Leer Entradas**: Itera mediante un while sobre cada entrada en el directorio:
        - Solo procesa entradas que son nombres numéricos (ID de procesos).
        - Construye la ruta al archivo `cmdline` del proceso.
        - Llama a `fetch_process` para obtener el nombre del proceso y determinar si es un proceso de usuario [U] o del kernel [K].
        - Imprime el resultado.
4. **Función `fetch_process`**:
    - **Abrir el Archivo**: Intenta abrir el archivo `cmdline`. Si no puede, considera que el proceso podría ser del kernel y devuelve 'K'.
    - **Leer el Contenido**: Utiliza `getline` para leer el contenido del archivo. `getline` maneja automáticamente la memoria, lo que simplifica el manejo de cadenas.
    - **Clasificación**: Si el contenido está vacío, devuelve 'K'; de lo contrario, devuelve 'U'.
5. **Función fetch kernel process_name**:
    - Abrir Archivo comm: Abre el archivo comm para un proceso del kernel y obtiene su nombre.
    - Lectura del Nombre: Usa getline() para leer el contenido del archivo y almacenarlo en procname.

Mejoras Sugeridas

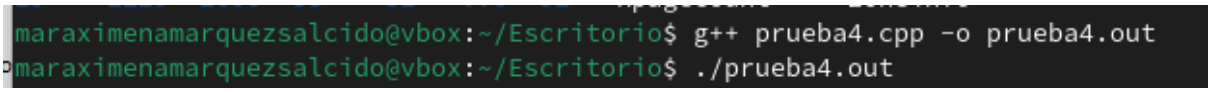
1. **Manejo de Errores**:
    - En lugar de terminar el programa abruptamente con `exit`, se puede retornar un código de error y manejarlo adecuadamente.
2. **Manejo de Memoria**:
    - `getline` asigna memoria para la línea leída. No es necesario liberar `line` manualmente, ya que el uso de `getline` la maneja.
3. **Uso de `snprintf`**:
    - Se utiliza `snprintf` en lugar de `sprintf` para evitar problemas de desbordamiento de búfer. `snprintf` limita el número de caracteres escritos, lo que mejora la seguridad del código.
4. **Tamaño de Buffer**:
    - Aumenté el tamaño del buffer para `cmdline` a 4096 bytes, ya que el límite de longitud de la línea de comando puede ser mayor en sistemas Linux.

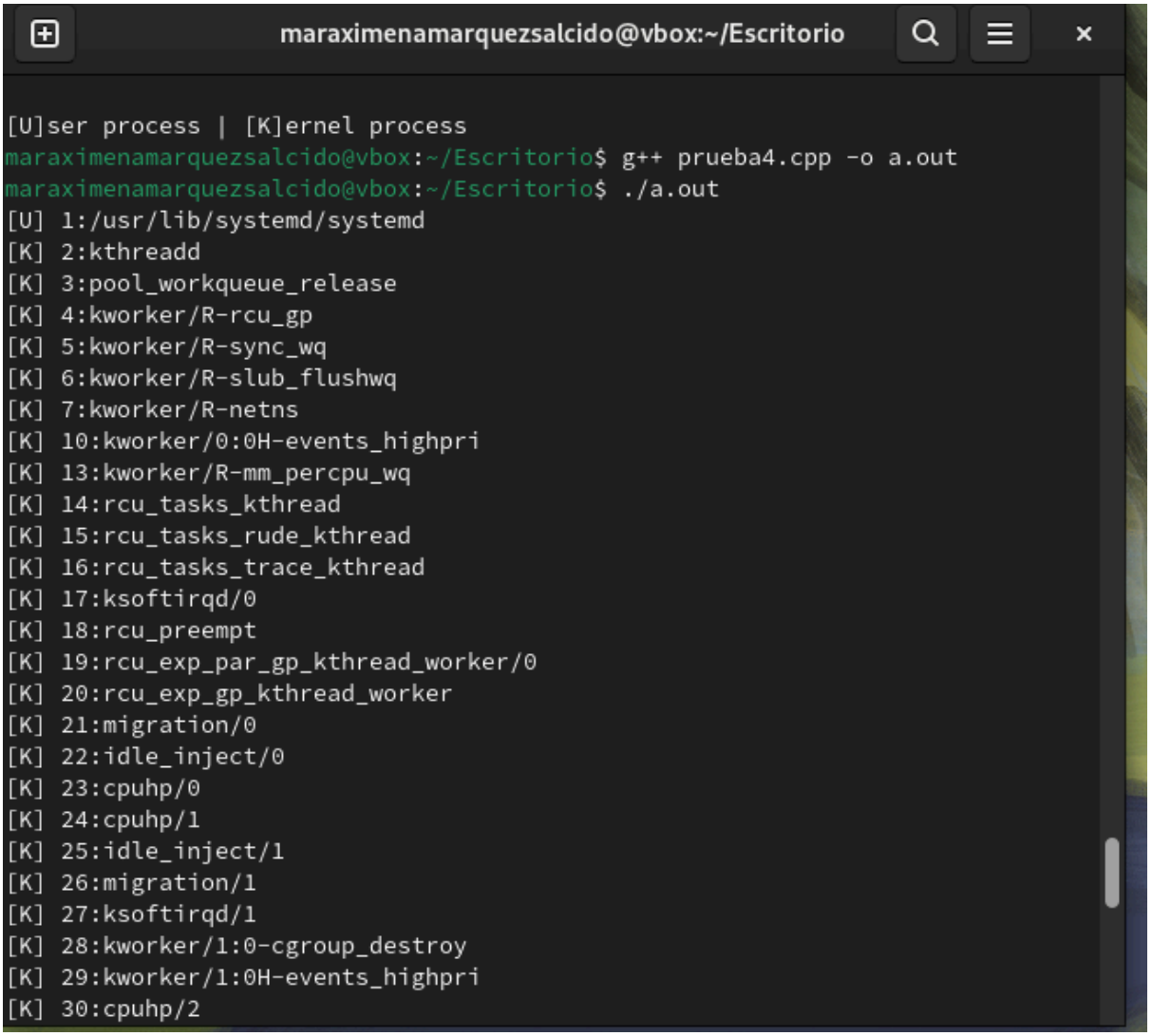
5. **Verificación de Contenido**:
    - Para verificar si el nombre del proceso está vacío, se utiliza `procname[0] == '\0'`, lo que es más claro.

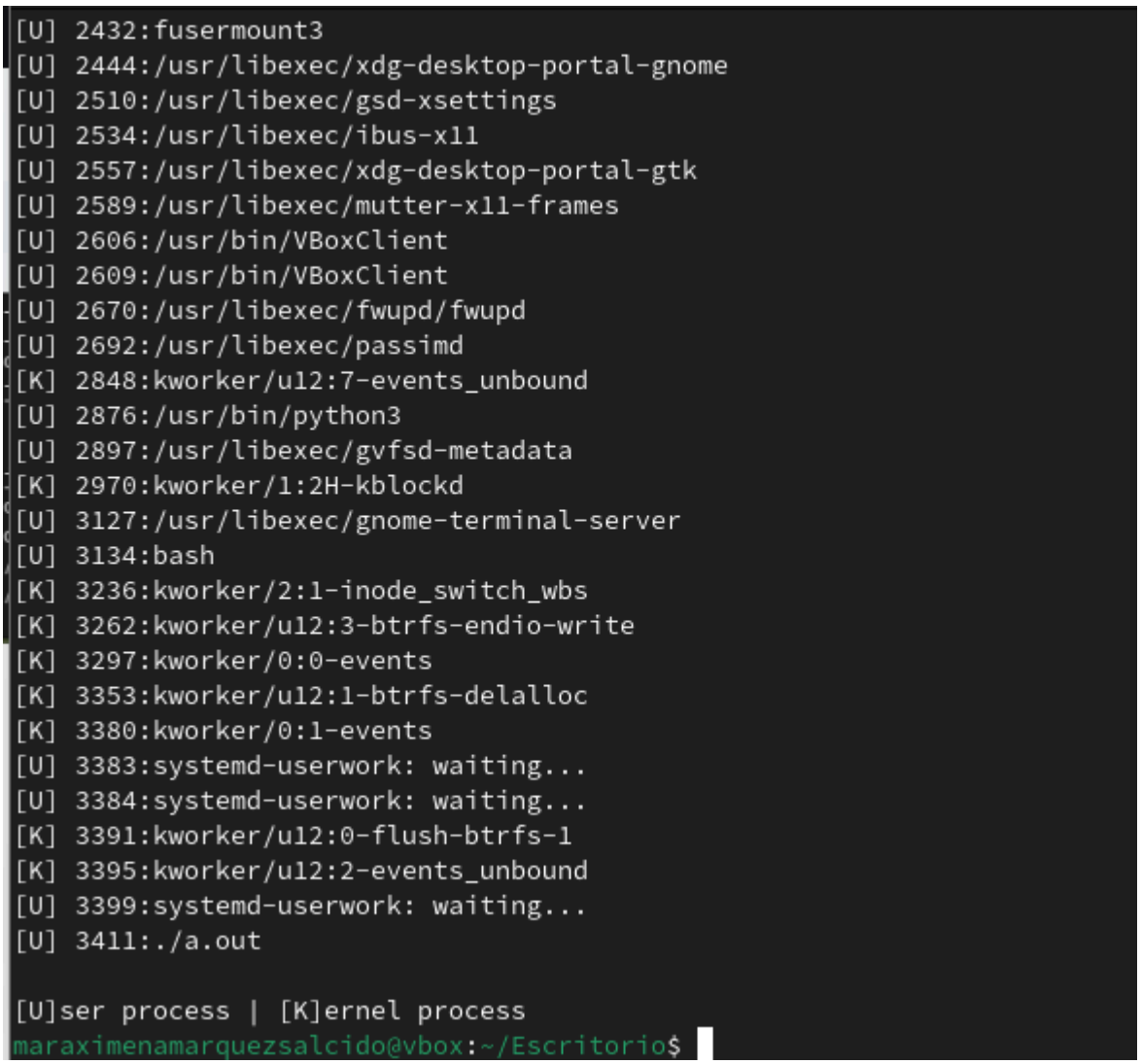
Resultado

El código, al ejecutarse, lista todos los procesos en el sistema, mostrando si son procesos hacen que el código sea más robusto, seguro y eficiente.de usuario o del kernel, junto con sus IDs y líneas de comando. Las mejoras propuestas

primero se compila y ejecuta







Conclusión:

Este programa nos permite ejecutar de manera eficiente un lector de código de los procesos en linux en el directorio /proc y clasificarlos ya sea un proceso de kernel o de usuario. Utilizamos el manejo de archivos, directorios y separamos sus procesos basados en su forma escrita.

El manejo de directorios y archivos se realiza de manera eficaz y manual, garantizando que solo se usen aquellos que representen un ID de procesos válidos (numéricos).El código utiliza buenas prácticas para el manejo de errores que puede tener el código, como la verificación de apertura de archivos y directorios, aunque en algunos puntos puede ser incluso mejor.

Mejoras de implementación, el uso seguro de buffers es esencial para evitar vulnerabilidades que se puedan usar en contra ya que se ejecuta en software a nivel kernel Eficiencia, se aumenta el tamaño del buffer cmdline lo que nos permite manejar líneas de comando más largas garantizando que tenga un uso funcional en diferentes entornos linux con configuraciones personalizadas