UNIVERSIDAD
DE ANTIOQUIA

1 8 0 3

Comando Linux:psinfo

Brandon Duque Garcia
Jonathan Andrés Granda

Laboratorio | Sistemas Operativos

Etapa #1

Reciba un solo identificador de proceso (PID) como argumento en línea de comandos.

Implementación basica. Listar proceso por PID

Información para cada PID

Tabla Información de procesos. Datos recopilados en el momento que se consulta el estado del proceso

Mensaje	Variable
Nombre del proceso	Name
Estado del proceso	State
Tamaño total de la imagen de memoria	[Valor]
Tamaño de la sección de memoria **TEXT**	VmExe
Tamaño de la sección de memoria **DATA**	VmData
Tamaño de la sección de memoria **STACK**	VmStk
Número de cambios de contexto	
Voluntarios	voluntary_ctxt_switches
No voluntarios	nonvoluntary_ctxt_switches

Salida esperada

$ ./psinfo 10898

Nombre del proceso: gedit
Estado: S (sleeping)
Tamaño total de la imagen de memoria: 715000 KB
Tamaño de la memoria TEXT: 10000 KB
Tamaño de la memoria DATA: 24200 KB
Tamaño de la memoria STACK: 21000 KB
Número de cambios de contexto (voluntarios - no voluntarios):
17536 - 189

Etapa #1 - Implementación

Reciba un solo identificador de proceso (PID) como argumento en línea de comandos.

Implementación basica. Listar proceso por PID

Anotaciones

En esta primera etapa, se implementa una secuencia de pasos importantes enumerados de la siguiente manera:

- Definir una función para abrir los directorios, recibiendo los parámetros importantes de la instrucción (ruta, id)
- Luego definir una secuencia de textos para almacenar las variables de cada directorio
- Construir la ruta del archivo
- Procesar una expresión regular para validar el proceso que estamos recibiendo (confirmar que solo tengamos números)
- Luego se debe verificar si estamos trabajando con directorios o no dados los IDs del proceso.
- Explorar y abrir directorios en la carpeta `/proc/[ID]/status`
- Luego mostrar la información del proceso

Construcción de la funcionalidad

```
// Función para abrir el directorio
int openDirectory(const char *ruta, const char *id)
{
    struct dirent *entry; // Estructura dirent leer las entradas del directorio
    char path[50];        // Almacenar ruta del archivo
    // Construir ruta del archivo
    snprintf(path, sizeof(path), "%s/%s/status", ruta, id);

    char info[2048];        // información del archivo
    regex_t regex;              // Expresión regular
    int reti;                 // Retorno al compilar la expresión regular
    DIR *dir = opendir(ruta); // Puntero hacia un directorio abierto
    int contador = 0;         // Contadora
    FILE *fptr;               // puntero de tipo FILE
    char content[2048];       // Contenido del directorio
    bool found = false;       // bandera
...
```

Anotaciones

`int openDirectory(const char *ruta, const char *id)`:

Define una función llamada **`openDirectory`** que toma dos argumentos:

- **`const char *ruta`**: Un puntero a una cadena constante que representa la ruta del directorio base (en este caso, se espera que sea `/proc`).
- **`const char *id`**: Un puntero a una cadena constante que representa el ID del proceso que se va a buscar.

- **`struct dirent *entry;`**: Declaramos un puntero `entry` a una estructura de tipo `dirent`. Esta estructura se utiliza para almacenar información sobre cada entrada (archivo o directorio) encontrada dentro de un directorio.

Anotaciones

Variables mencionadas

`char info[2048];`: Se tiene un array de caracteres llamado `info` de tamaño 2048. Se utilizará para almacenar la información que se leerá del archivo `status` del proceso.

`regex_t regex;`: Declara una variable `regex` de tipo `regex_t`

`int reti;`: Declara una variable entera `reti`. Se utilizará para almacenar el valor de retorno de las funciones relacionadas con expresiones regulares

`DIR *dir = opendir(ruta);`: Llama a la función `opendir` para abrir el directorio especificado por `ruta` (`/proc`).

`FILE *fptr;`: Declara un puntero `fptr` de tipo `FILE`. Se utilizará para referenciar el archivo `status` del proceso una vez que se abra.

Anotaciones

Variables mencionadas

`char content[2048];`: Se declara un array de caracteres llamado content de tamaño 2048. Se utilizará para leer líneas individuales del archivo `status`.

`bool found = false;`: Declara una variable booleana `found` y la inicializa en `false`. Se utilizará como una bandera para indicar si se ha encontrado el directorio del proceso con el ID especificado.

Construcción de la funcionalidad

```
snprintf(path, sizeof(path), "%s/%s/status", ruta, id);

...
```

Esta línea en particular resulta importante dado que se formatea la cadena y almacenamos en path.

`%s`: Se reemplaza con el valor de `ruta` (`/proc`).

`%s`: Se reemplaza con el valor de `id` (el ID del proceso).

Entonces vamos a tener cadenas de tipo: **`/{proc}/{id}/status`**

Construcción de la funcionalidad

```
// Compilar la expresión regular

    reti = regcomp(&regex, "^[0-9]+$", REG_EXTENDED);
        // Verficar si existe reti
    if (reti)
    {
        printf("Error al compilar la expresión regular.\n");
        exit(1);
    }
...
```

En esta porción de código:

`reti = regcomp(&regex, "^[0-9]+$", REG_EXTENDED);`: Llama a la función `regcomp` para compilar la expresión regular `"^[0-9]+$"`.

`if (reti)`: Verifica si el valor de `reti` es distinto de cero, lo que indica un error en la compilación de la expresión regular (Si la compilación es exitosa devuelve 0).

Construcción de la funcionalidad

```
// Abrir directorio
    if (dir == NULL) // No existe
    {
        perror("No se puede abrir el directorio");
        return -1;
    }
```

`if (dir == NULL)`: Verifica si el puntero `dir` es `NULL`, lo que significa que la llamada a `opendir` falló.

`perror("No se puede abrir el directorio");`: Llama a la función `perror`, que imprime un mensaje de error descriptivo en la consola basado en el valor de la variable global `errno`, que se establece cuando una llamada al sistema falla.

```c
while ((entry = readdir(dir)) != NULL) // iteraciones de las entradas del directorio
    {
        // Excluimos los directorios '.' y '..'
        if (entry->d_name[0] != '.')
        {
            if (entry->d_type == DT_DIR)
            { // verificamos si es un directorio
                reti = regexec(&regex, entry->d_name, 0, NULL, 0);
                if (!reti)
                {
                    if (strcmp(entry->d_name, id) == 0)
                    {
                        fptr = fopen(path, "r");
                        while (fgets(content, 2048, fptr))
                        {
                            Información del proceso
                        }
                        found = true;
                        break;
                    }
                }
                else if (reti == REG_NOMATCH) {
                    printf("La carpeta no pertenece a un proceso.\n");
                }
                else {
                    char error_buf[100];
                    regerror(reti, &regex, error_buf, sizeof(error_buf));
                    printf("Error al hacer la comparación: %s\n", error_buf);
                }
            }
        }
    }
```

`while ((entry = readdir(dir)) != NULL)`: Inicia un bucle `while` que continúa mientras la función `readdir(dir)` devuelva un puntero no `NULL`

La función `readdir(dir)` Lee las siguientes entradas al directorio, Cuando las haya leído todas devuelve `NULL`

`if (entry->d_name[0] != '.')`: Se debe verificar si el primer carácter del nombre de la entrada del directorio (`entry->d_name`) no es un punto (`.`). Lo que hace es ignorar los directorios especiales "." (el directorio actual) y ".." (el directorio padre).

UNIVERSIDAD DE ANTIOQUIA
1803

while ((entry = readdir(dir)) != NULL): Inicia un bucle while que continúa mientras la función readdir(dir) devuelva un puntero no NULL

La función readdir(dir) Lee las siguientes entradas al directorio, Cuando las haya leído todas devuelve NULL

DIR *dir = opendir(ruta)
readdir(dir)

Ese es el esquema para explorar los directorios presentes en /proc

```c
while ((entry = readdir(dir)) != NULL) // iteraciones de las entradas del directorio
    {
        // Excluimos los directorios '.' y '..'
        if (entry->d_name[0] != '.')
        {
            if (entry->d_type == DT_DIR)
            { // verificamos si es un directorio
                reti = regexec(&regex, entry->d_name, 0, NULL, 0);
                if (!reti)
                {
                    if (strcmp(entry->d_name, id) == 0)
                    {
                        fptr = fopen(path, "r");
                        while (fgets(content, 2048, fptr))
                        {
                            Información del proceso
                        }
                        found = true;
                        break;
                    }
                }
                else if (reti == REG_NOMATCH) {
                    printf("La carpeta no pertenece a un proceso.\n");
                }
                else {
                    char error_buf[100];
                    regerror(reti, &regex, error_buf, sizeof(error_buf));
                    printf("Error al hacer la comparación: %s\n", error_buf);
                }
            }
        }
    }
```

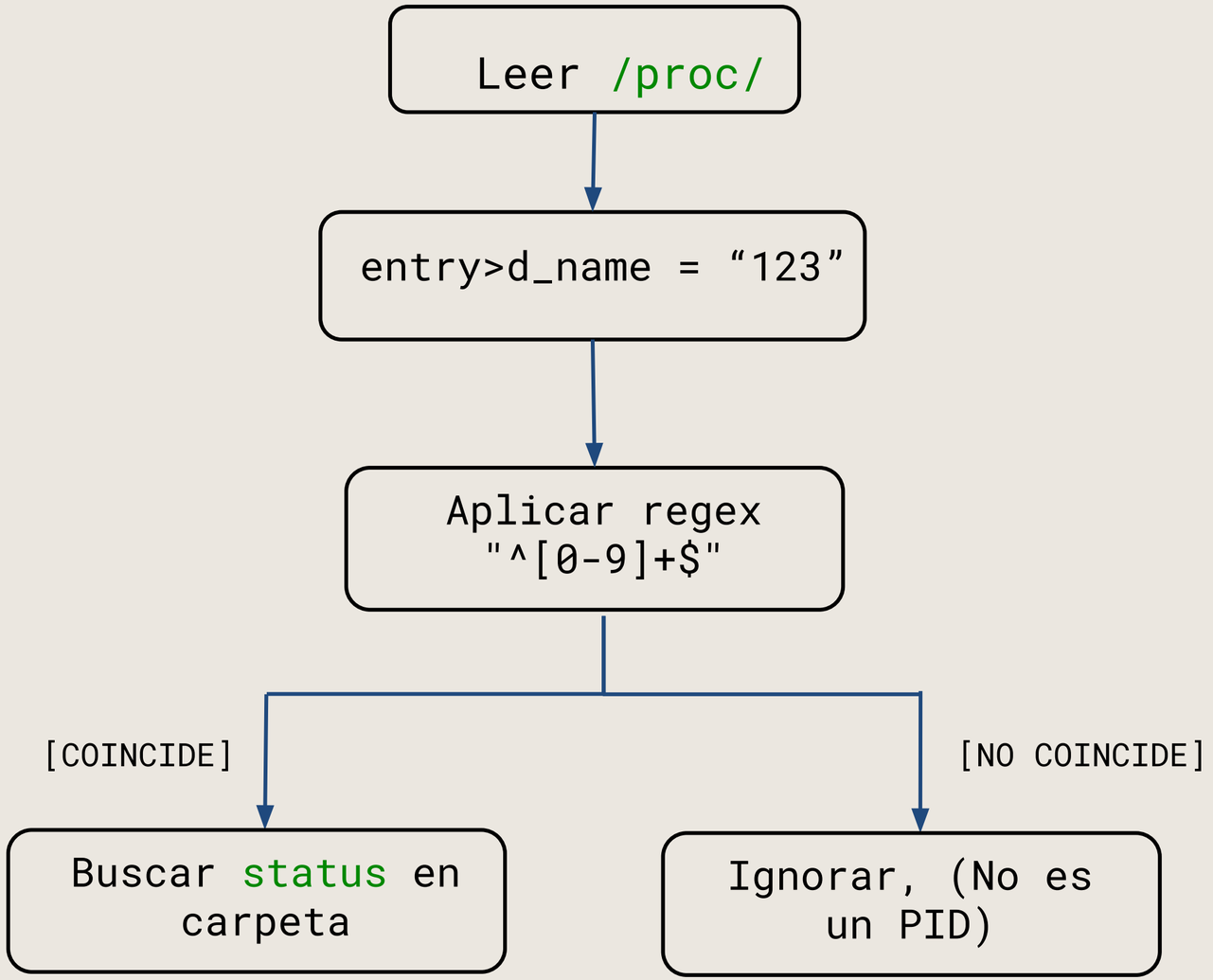
`if (entry->d_type == DT_DIR)`: Verifica si el tipo de la entrada (`entry->d_type`) es `DT_DIR`, lo que indica que es un directorio. Dentro de `/proc`, los directorios que corresponden a procesos tienen números en sus nombres (los IDs de los procesos).

`reti = regexec(&regex, entry->d_name, 0, NULL, 0);`:Ahora usamos a la función `regexec` para intentar hacer coincidir la expresión regular compilada con el nombre de la entrada del directorio

`reti = REG_NOMATCH`
En este caso no coincide y va a haber otro valor distinto a cero si ocurre un error, es decir, no tenemos un número.

UNIVERSIDAD DE ANTIOQUIA
1803

El esquema para la expresión regular es el siguiente



`regcomp()` → compila la expresión regular

`regexec()` → ejecuta la expresión regular

`regfree()` → libera la memoria usada por la regex

`regerror()` → muestra mensajes de error si algo falla

UNIVERSIDAD DE ANTIOQUIA
1803

```c
while ((entry = readdir(dir)) != NULL) // iteraciones de las entradas del directorio
    {
        // Excluimos los directorios '.' y '..'
        if (entry->d_name[0] != '.')
        {
            if (entry->d_type == DT_DIR)
            { // verificamos si es un directorio
                reti = regexec(&regex, entry->d_name, 0, NULL, 0);
                if (!reti)
                {
                    if (strcmp(entry->d_name, id) == 0)
                    {
                        fptr = fopen(path, "r");
                        while (fgets(content, 2048, fptr))
                        {
                            Información del proceso
                        }
                        found = true;
                        break;
                    }
                }
                else if (reti == REG_NOMATCH) {
                    printf("La carpeta no pertenece a un proceso.\n");
                }
                else {
                    char error_buf[100];
                    regerror(reti, &regex, error_buf, sizeof(error_buf));
                    printf("Error al hacer la comparación: %s\n", error_buf);
                }
            }
        }
    }
```

`if (strcmp(entry->d_name, id) == 0)`: Comparamos el nombre del directorio actual (`entry->d_name`) con el ID del proceso que se está buscando (`id`). Si las cadenas son iguales (es decir, se encontró el directorio del proceso con el ID especificado), el resultado de `strcmp` será `0`.

`fptr = fopen(path, "r");`: Se abre el archivo `status` del proceso encontrado en modo lectura (`"r"`). La ruta a este archivo se construyó previamente en la variable `path`, El puntero se asigna al archivo abierto `fptr`.

UNIVERSIDAD DE ANTIOQUIA
1803

```c
while ((entry = readdir(dir)) != NULL) // iteraciones de las entradas del directorio
    {
        // Excluimos los directorios '.' y '..'
        if (entry->d_name[0] != '.')
        {
            if (entry->d_type == DT_DIR)
            { // verificamos si es un directorio
                reti = regexec(&regex, entry->d_name, 0, NULL, 0);
                if (!reti)
                {
                    if (strcmp(entry->d_name, id) == 0)
                    {
                        fptr = fopen(path, "r");
                        while (fgets(content, 2048, fptr))
                        {
                            Información del proceso
                        }
                        found = true;
                        break;
                    }
                }
                else if (reti == REG_NOMATCH) {
                    printf("La carpeta no pertenece a un proceso.\n");
                }
                else {
                    char error_buf[100];
                    regerror(reti, &regex, error_buf, sizeof(error_buf));
                    printf("Error al hacer la comparación: %s\n", error_buf);
                }
            }
        }
    }
```

`while (fgets(content, 2048, fptr))`: Inicia un bucle `while` que lee líneas del archivo `status` hasta que se alcanza el final del archivo o ocurre un error.

Este último bloque de código se ejecuta cuando la función `regexec` (que intenta hacer coincidir la expresión regular con el nombre del directorio) devuelve un código de error distinto de `REG_NOMATCH`.

UNIVERSIDAD DE ANTIOQUIA
1803

Información del proceso

`strncmp(, content, ...)`: Compara los primeros `n` caracteres de dos cadenas. Si coinciden, devuelve 0..

```
if (strncmp("Name:", content, 5) == 0) {
        strcat(info, "Nombre del proceso: ");
        strcat(info, content + 6);
}
else if (strncmp("State:", content, 6) == 0) {
        strcat(info, "Estado del proceso: ");
        strcat(info, content + 7);
}
else if (strncmp("VmSize:", content, 7) == 0) {
        strcat(info, "Tamaño de la imagen de memoria: ");
        strcat(info, content + 8);
}
else if (strncmp("VmExe:", content, 6) == 0) {
        strcat(info, "Tamaño de la memoria TEXT: ");
        strcat(info, content + 7);
}
else if (strncmp("VmData:", content, 7) == 0) {
        strcat(info, "Tamaño de la memoria DATA: ");
        strcat(info, content + 8);
}
else if (strncmp("VmStk:", content, 6) == 0) {
        strcat(info, "Tamaño de la memoria STACK: ");
        strcat(info, content + 7);
}
else if (strncmp("voluntary_ctxt_switches:", content, 24) == 0) {
        strcat(info, "# de cambios de contexto voluntarios: ");
        strcat(info, content + 25);
}
else if (strncmp("nonvoluntary_ctxt_switches:", content, 27) == 0) {
        strcat(info, "# de cambios de contexto no voluntarios: ");
        strcat(info, content + 28);
}
```

```
if (found == true)
    {
        printf("- INFORMACIÓN DEL PROCESO - \n\n %s", info);
    }
    else
    {
        printf("The ID was not found, the process does not exist.");
    }
    // Cerramos el archivo
    fclose(fptr);
    // Liberar la expresión regular compilada
    regfree(&regex);
    // Cerrar el directorio
    closedir(dir);
    return contador;
```

`found = true;`: Establece la bandera `found` en `true` para indicar que se ha encontrado y procesado la información del proceso con el ID especificado.

Implementaciones en el método principal

Ahora se muestra como funciona la implementación de la función en el método principal, aca se maqueta la lógica para la implementación de la funcionalidad de la etapa #2 pero no se tiene ninguna funcionalidad.
Por lo que simplemente válida el número de argumentos y llama al método en el caso de que hay 1 solo argumento, cuando se ejecuta `$ ./psinfo [PID]`

Método principal

```
int main(int argc, char *argv[])
{
    char path[] = "/proc/";
    // Se verifica si el segundo argumento es -l en caso de tener más de 2 argumentos
    if (argc > 2)
    {
        // Si el segundo argumento es -l se procede a leer los argumentos en lista o solo se busca un solo ID
        if (strcmp(argv[1], lista) == 0)
        {
            //Fase 2
        }
        else
        { // No se encuentra el argumento -l
            printf("psinfo: usage error: Syntax error, -l is missing\n\n");
        }
    }
    else if (argc == 1)
    { // Si se ejecuta el comando ./psinfo solo sin argumentos
        printf("psinfo: usage error: Proccess ID required\n\n");
        printf("Options\n\n./psinfo ID ------------------------------> Returns the process info\n./psinfo -l ID1 ID2 ID3
ID4 ... IDn --------> Returns a report of processes info\n");
    }

    else
    { // Si tiene exactamente 2 argumentos se crea la ruta de la carpeta del proceso a buscar
        strcat(path, argv[1]);
        openDirectory("/proc", argv[1], NULL);
    }
    return 0;
}
```

Establece que deben haber 2 argumentos, para este caso el primer argumento es el nombre del programa, y el segundo [PID].

Etapa #2

Agregue la opción -l (lista múltiple). Si el usuario ingresa varios PIDs

Implementación siguiente. Listar varios procesos

Salida esperada

```
-- Información recolectada!!!
Pid: 10898
Nombre del proceso: gedit
...
Pid: 1342
Nombre del proceso: chrome
...
Pid: 2341
Nombre del proceso: bash
...
```

Anotaciones

En esta segunda etapa, se hace necesario la implementación de una estructura, para el almacenamiento de más procesos en el programa

```
typedef struct
{
    char pid[16];
    char info[2048]
} ProcInfo;
```

Esta estructura tiene dos miembros:

- `pid`: Un array de caracteres de tamaño 16 para almacenar el **Process ID** (PID) como una cadena.
- `info`: Un array de caracteres de tamaño 2048 para almacenar la **información detallada** del proceso.

Anotaciones

La función en la etapa #1 ahora sufre una modificación y va a recibir 3 argumentos

```
int openDirectory(const char *ruta, const char *id, char *infoMem)
```

Propósito: Este parámetro permite a la función `openDirectory` **escribir la información del proceso directamente en un puntero** proporcionado por la función que la llama (`main` en este caso). Esto es crucial para la nueva funcionalidad de la lista de procesos.

Consideraciones

Dentro de `openDirectory`, ahora se verifica si `infoMem` es `NULL`

```
if (infoMem != NULL)
{
    strcpy(infoMem, info);
}
else
{
    printf("- INFORMACIÓN DEL PROCESO - \n\n %s", info);
}
// ... dentro del bloque else (found == false) ...
if (infoMem != NULL)
{
    sprintf(infoMem, "Pid: %s\n The ID was not found, the process does not exist. \n", id);
}
else
{
    printf("The ID was not found, the process does not exist.");
}
```

Si `infoMem` es `NULL`: Significa que la función fue llamada para procesar un solo PID (sin la opción `-l`). En este caso, la información se imprime directamente en la consola como en la etapa #!.

La línea `strcpy(infoMem, info);` toma la cadena completa que contiene la información del proceso en la variable info de la **estructura y** la duplica en la región de memoria a la que apunta `infoMem` que es el argumento que quiero almacenar

```
int main(int argc, char *argv[])
{
    char lista[] = "-l";
    bool itslist = false;
    char path[] = "/proc/";
    ProcInfo *procInfos = NULL;
    int numProcesos = 0;
    // Se verifica si el segundo argumento es -l en caso de tener más de 2 argumentos
    if (argc > 2)
    {
        // Si el segundo argumento es -l se procede a leer los argumentos en lista o solo se busca un solo ID
        if (strcmp(argv[1], lista) == 0)
        {
            itslist = true;
            numProcesos = argc - 2;
            procInfos = (ProcInfo *)malloc(numProcesos * sizeof(ProcInfo));
            if (procInfos == NULL)
            {
                printf("Error: No se pudo asignar memoria.\n");
                return 1;
            }
            // Almacenar información de cada proceso en las estructuras
            for (int i = 0; i < numProcesos; i++)
            {
                strcpy(procInfos[i].pid, argv[i + 2]);
                openDirectory("/proc", argv[i + 2], "", procInfos[i].info);
            }
            ...
```

De manera inicial se tienen variables locales.
Se inicia validando el número de argumentos, para posteriormente validar si el segundo es `-l`

Se **asigna memoria dinámicamente** para un array de estructuras `ProcInfo` utilizando `malloc`. El tamaño de la memoria asignada es suficiente para almacenar la información de todos los procesos proporcionados en la lista.

Se itera a través de los PIDs proporcionados después de `-l` (desde `argv[2]` en adelante).

```
            ...
            // Mostrar info
            printf("-- Información recolectada!!! \n\n");
            for (int i = 0; i < numProcesos; i++)
            {
                printf("%s\n ", procInfos[i].info);
            }
            //liberar struct
            free(procInfos);
        }
        else
        { // No se encuentra el argumento -l
            printf("psinfo: usage error: Syntax error, -l is missing\n\n");
        }
    }
    else if (argc == 1)
    { // Si se ejecuta el comando ./psinfo solo sin argumentos
        printf("psinfo: usage error: Proccess ID required\n\n");
        printf("Options\n\n./psinfo ID ------------------------------> Returns the process
info\n./psinfo -l ID1 ID2 ID3 ID4 ... IDn --------> Returns a report of processes info\n");
    }
    else
    { // Si tiene exactamente 2 argumentos se crea la ruta de la carpeta del proceso a
buscar
        strcat(path, argv[1]);
        openDirectory("/proc", argv[1], NULL);
    }
    return 0;
}
```

Se itera a través del array `procInfos` y se **imprime la información de cada proceso** almacenada en el campo `info` de cada estructura.
Finalmente, se **libera la memoria dinámica** asignada para `procInfos` utilizando `free(procInfos)`

Si el segundo argumento no es `-l` cuando hay más de dos argumentos, se muestra un mensaje de error de sintaxis.

Si el número de argumentos es exactamente 2 (el nombre del programa y un PID), se sigue el comportamiento anterior, llamando a `openDirectory` con el tercer argumento como `NULL` para que la información se imprima directamente en la consola.

Etapa #3

Implemente la opción -r (reporte a archivo).

Implementación final. Generar un reporte

Salida esperada

1. Toma varios PIDs.
2. Lee la información de cada proceso.
3. Genera un archivo de salida llamado:
   psinfo-report-[pid1]-[pid2]-...[pidN].info
4. Que contenga la misma información.

Ejemplo:

```
$ psinfo -r 10898 1342
Archivo de salida generado:
psinfo-report-10898-1342.info
```

Anotaciones

En esta última etapa se agregan nuevas variables, pero la implementación lógica funciona de manera similar, añadiendo la función de que en este caso debemos generar el archivo *.info

```
char reporte[] = "-r";
FILE *archivoReporte;
char nombre[100] = "psinfo-report";
char guion[100] = "-";
```

- Variables relacionadas en la función openDirectory.

Finalidad de las nuevas variables

`char reporte[] = "-r";`: Esta se usa para declarar un array de caracteres `reporte` con la cadena `"-r"`. Esta cadena se utilizará para comparar con el segundo argumento de la línea de comandos para activar la funcionalidad de generar un archivo de reportar(como se hace con la lista -l).

`FILE *archivoReporte;`: Se usa un puntero a un tipo `FILE` llamado `archivoReporte`. Este puntero se utilizará para el archivo que se creará para guardar el reporte de los procesos.

`char nombre[100] = "psinfo-report";`: Inicializamos un array de caracteres `nombre` con la cadena `"psinfo-report"`. Esta cadena se utilizará como base para el nombre del archivo de reporte.

`char guion[100] = "-";`: Por último un array de caracteres `guion` con la cadena `"-"`. Este guion se utilizará para separar el nombre base del reporte con los PIDs de los procesos incluidos en el reporte.

Implementación Main

```
else if (strcmp(argv[1], reporte) == 0)
{
    numProcesos = argc - 2;
    procInfos = (ProcInfo *)malloc(numProcesos * sizeof(ProcInfo));
    // Almacenar información de cada proceso en las estructuras
    for (int i = 0; i < numProcesos; i++)
    {
        strcpy(procInfos[i].pid, argv[i + 2]);
        strcat(nombre, guion);
        strcat(nombre, argv[i + 2]);
        openDirectory("/proc", argv[i + 2], procInfos[i].info);
    }
    strcat(nombre, ".info");
    archivoReporte = fopen(nombre, "w");

    for (int i = 0; i < numProcesos; i++)
    {
       fputs(procInfos[i].info, archivoReporte);
       fputs("\n", archivoReporte);
    }
    fclose(archivoReporte);
    free(procInfos);
    printf("Archivo de salida generado: %s", nombre);
}
```

Se inicia validando el número de argumentos, para posteriormente validar si el segundo es `-r`

Copiamos el PID a la estructura `procInfos[i].pid`. Luego usamos `OpenDirectory` para la información de cada PID copiado
Se intenta abrir un nuevo archivo con el nombre generado en modo escritura ("w"). Si la apertura es exitosa, el puntero al archivo se guarda en `archivoReporte`.

Etapa #4

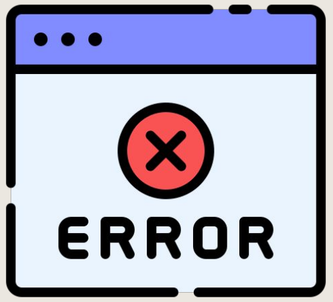
Gestione los usos incorrectos

Implementación final. Revisar errores en las ejecuciones

Puntos de gestión de errores

La gestión de errores en el laboratorio psinfo se contempló en varios puntos clave para asegurar que el programa se comporte de manera predecible y proporcione información útil al usuario. Los puntos donde se gestionan los errores son los siguientes:

1. Cuando se compila la expresión Regular
2. En el momento de Apertura del Directorio
3. Cuando se abre el archivo del estado de proceso
4. Al momento de ejecutar la expresión regular
5. Asignando la memoria dinamica
6. [PIDs] no encontrados
7. Manejo de un número incorrecto de argumentos



Excepción #1 - Expresión regular

Compilación de la expresión regular

Después de intentar compilar la expresión regular, el valor de retorno `reti` se verifica.
Para saber si es distinta de cero e indicar el error(una expresión no válida en la regex)

```
reti = regcomp(&regex, "^[0-9]+$", REG_EXTENDED);
if (reti)
{
    printf("Error al compilar la expresión regular.\n");
    exit(1);
}else{
//manejo de error con REG_NOMATCH
}
```

Excepción #2 - Directorios nulos

Apertura del directorio /proc

Para este caso, queremos acceder al directorio `opendir` usando la función `opendir(ruta)`.
Está la posibilidad de que `opendir` retorna NULL, por muchas razones.
Por lo que se usa un mensaje de error en consola `perror`

```
DIR *dir = opendir(ruta);
if (dir == NULL)
{
    perror("No se puede abrir el directorio");
    return -1;
}
```

Excepción #3 - Archivo de estado del proceso

Estado de proceso con `fopen`

Dentro del ciclo que busca el directorio del proceso en cuestión, una vez se coincide el directorio con el `[pid]` se abre el archivo.

Está la posibilidad de que no se puede abrir, Por lo que se se establece la bandera `found` en `false` y se sale del ciclo while de lectura del directorio. Esto asegura que se informe que no se pudo acceder a la información del proceso.

```
fptr = fopen(path, "r");
if (fptr == NULL)
{
    strcpy(info, "No se pudo abrir el archivo de estado de proceso");
    found = false;
    break;
}
```

Excepción #4 - Ejecución de la REGEX

Coincidencia de REGEX con [Pid]

Después de intentar hacer coincidir la expresión regular con el nombre del directorio usando `regexec`, se verifica el valor de retorno (`reti`).

Si `reti` es diferente de `0` (coincidencia exitosa) y diferente de `REG_NOMATCH` (no hubo coincidencia), indica que ocurrió un error durante la ejecución de la expresión regular.

```
reti = regexec(&regex, entry->d_name, 0,
NULL, 0);
if (reti != 0 && reti != REG_NOMATCH)
{
    char error_buf[100];
    regerror(reti, &regex, error_buf,
sizeof(error_buf));
    printf("Error al hacer la comparación: %s\n",
error_buf);
}
```

Excepción #5 - Asignación de memoria

Utilizar `-l` y `-r` para multiples `[Pids]`

Se verifica si el puntero devuelto por `malloc` es `NULL`. Si lo es, significa que no se pudo asignar la memoria solicitada (por ejemplo, por falta de memoria disponible).

```
reti = regexec(&regex, entry->d_name, 0, NULL, 0);
if (reti != 0 && reti != REG_NOMATCH)
{
    char error_buf[100];
    regerror(reti, &regex, error_buf, sizeof(error_buf));
    printf("Error al hacer la comparación: %s\n",
error_buf);
}
```

Excepción #6 - "ID no encontrado":

Utilizar `-l` y `-r` para multiples `[Pids]`

Si `found` es `false`, significa que no se encontró ningún directorio con el ID proporcionado.
Este caso es importante, ya que podemos directamente a la consola o copiándolo al buffer `infoMem` si se está procesando una lista de PIDs(para expresar alguno).

```
if (found == true)
{
    // ... (imprimir información o copiar a buffer) ...
}
else
{
    if (infoMem != NULL)
    {
        sprintf(infoMem, "Pid: %s\n The ID was not found, the process does not exist. \n", id);
    }
    else
    {
        printf("The ID was not found, the process does not exist.");
    }
}
```

Excepción #7 - "Número incorrecto de argumentos":

Gestionar posibilidades en los argumentos

Caso de ningún argumento (solo el nombre del programa): Si `argc` es igual a 1, significa que el usuario ejecutó el programa sin proporcionar ningún PID.

Caso de sintaxis incorrecta con múltiples argumentos: Si `argc` es mayor que 2 (lo que sugiere que se intentó usar la opción de lista o reporte), pero el segundo argumento no es ni `"-l"` ni `"-r"`.

```
if (argc == 1)
{ // Si se ejecuta el comando ./psinfo solo sin argumentos
    printf("psinfo: usage error: Proccess ID required\n\n");
    printf("Options\n\n./psinfo ID ----------------------------> Returns
the process info\n./psinfo -l ID1 ID2 ID3 ID4 ... IDn -------->
Returns a report of processes info\n");
}
else if (argc > 2 && strcmp(argv[1], lista) != 0 && strcmp(argv[1],
reporte) != 0)
{ // Si hay más de 2 argumentos y el segundo no es -l ni -r
    printf("psinfo: usage error: Syntax error, -l is missing\n\n");
}
```

Excepción #7 - "Número incorrecto de argumentos":

Gestionar posibilidades en los argumentos

Caso de ningún argumento (solo el nombre del programa): Si `argc` es igual a 1, significa que el usuario ejecutó el programa sin proporcionar ningún PID.

Caso de sintaxis incorrecta con múltiples argumentos: Si `argc` es mayor que 2 (lo que sugiere que se intentó usar la opción de lista o reporte), pero el segundo argumento no es ni `"-l"` ni `"-r"`.

```
if (argc == 1)
{ // Si se ejecuta el comando ./psinfo solo sin argumentos
    printf("psinfo: usage error: Proccess ID required\n\n");
    printf("Options\n\n./psinfo ID ----------------------------> Returns
the process info\n./psinfo -l ID1 ID2 ID3 ID4 ... IDn -------->
Returns a report of processes info\n");
}
else if (argc > 2 && strcmp(argv[1], lista) != 0 && strcmp(argv[1],
reporte) != 0)
{ // Si hay más de 2 argumentos y el segundo no es -l ni -r
    printf("psinfo: usage error: Syntax error, -l is missing\n\n");
}
```

Uso reflexivo de herramientas generativa

Gestione los usos incorrectos

Revisiones de LLM. Código proveniente de IA

ChatGPT

Se usó la herramienta de **chatgpt** para el uso de las expresiones regulares, el prompt que se usó para la consulta fue la siguiente:

“Como usar expresiones regulares en el lenguaje de programación c”

Dicha consulta retorna los siguientes pasos:

1. **Incluir la biblioteca:** `#include <regex.h>`
2. **Declarar un objeto regex_t para guardar la expresión compilada:** `regex_t regex;`
3. **Compilar la expresión regular:** `int reti = regcomp(&regex, "^[0-9]+$", REG_EXTENDED);`
4. **Ejecutar la expresión regular sobre una cadena:** `reti = regexec(&regex, "12345", 0, NULL, 0);`
5. **Liberar memoria usada por la expresión:** `regfree(&regex);`

La bandera `REG_EXTENDED` permite usar expresiones regulares extendidas (como +, |, etc.).

Donde cambiamos “12345” por el nombre de cada directorio, facilitando así la búsqueda de todos los procesos.

ChatGPT

Se usó la herramienta de **chatgpt** para trabajar con las estructuras y las cadenas el prompt que se usó para la consulta fue la siguiente:

“Necesito guardar la información de un mensaje (String) en una estructura”

Dicha consulta retorna el siguiente ejemplo:

1. **Salida codigo**

```c
#define MAX_MENSAJE 256
typedef struct {
    char contenido[MAX_MENSAJE];
} Mensaje;
int main() {
    Mensaje m;
    strcpy(m.contenido, "Hola, este es un mensaje de ejemplo.");

    printf("Mensaje: %s\n", m.contenido);

    return 0;
}
```

ChatGPT

Se usó la herramienta de **chatgpt** para el uso de conocimiento de archivos, el prompt que se usó para la consulta fue la siguiente:

“Necesito escribir archivos en C”

Dicha consulta retorna el siguiente ejemplo:

```c
#include <stdio.h>

int main() {
    FILE *archivo;

    archivo = fopen("archivo.txt", "w"); // modo escritura
    if (archivo == NULL) {
        printf("No se pudo abrir el archivo para escritura.\n");
        return 1;
    }

    fprintf(archivo, "Hola mundo desde C.\n");
    fprintf(archivo, "Otra línea de texto.\n");

    fclose(archivo);
    return 0;
}
```

Pruebas y Debugging

Documentar casos de prueba

Funcionalidades. Capturas del programa funcionando

CP01 Mostrar información [Pid]

Este primero, es el caso exitoso, en el cual le paso un Pid de un proceso existente y con los argumentos correctos.

De manera inicial se muestra cuando se hace el proceso de compilación del proyecto

```
┌──(jonas㉿JonathanG)-[~/Documents/lab-01-SO]
└─$ make
gcc -c psinfo.c
cc    -c -o funciones.o funciones.c
gcc -o psinfo psinfo.o funciones.o

┌──(jonas㉿JonathanG)-[~/Documents/lab-01-SO]
└─$ ./psinfo 1905
- INFORMACIÓN DEL PROCESO -

 Pid: 1905
Nombre del proceso: bash
Estado del proceso: S (sleeping)
Tamaño de la imagen de memoria:     7544 kB
Tamaño de la memoria DATA:      828 kB
Tamaño de la memoria STACK:      136 kB
Tamaño de la memoria TEXT:      804 kB
# de cambios de contexto voluntarios: 102
# de cambios de contexto no voluntarios: 0
```

CP02 Error cuando no existe [Pid]

Este segundo caso de prueba, es el caso en el cual hay un fallo, y es que el usuario trata de buscar el [Pid] de un proceso no existente

```
┌──(jonas㉿JonathanG)-[~/Documents/lab-01-SO]
└─$ ./psinfo 3
The ID was not found, the process does not exist.
```

CP03 Error cuando no hay argumento

Este caso es cuando el programa solo recibe un argumento.

```
┌──(jonas㉿JonathanG)-[~/Documents/lab-01-SO]
└─$ ./psinfo
psinfo: usage error: Process ID required

Options

./psinfo ID ------------------------------> Returns the process info
./psinfo -I ID1 ID2 ID3 ID4 ... IDn --------> Returns a report of processes info
```

CP04 Mostrar varios [Pid]

Este caso exitoso de la segunda etapa, en la cual usamos el argumento `-l`
Con esta opción podemos ver la información de múltiples procesos

```
┌──(jonas㉿JonathanG)-[~/Documents/lab-01-SO]
└─$ ./psinfo -l 10 353
-- Información recolectada!!!

Pid: 10
Nombre del proceso: Relay(11)
Estado del proceso: S (sleeping)
Tamaño de la imagen de memoria:     2616 kB
Tamaño de la memoria DATA:      324 kB
Tamaño de la memoria STACK:      132 kB
Tamaño de la memoria TEXT:     1484 kB
# de cambios de contexto voluntarios: 48
# de cambios de contexto no voluntarios: 0

 Pid: 353
Nombre del proceso: dbus-daemon
Estado del proceso: S (sleeping)
Tamaño de la imagen de memoria:     6628 kB
Tamaño de la memoria DATA:     1452 kB
Tamaño de la memoria STACK:      132 kB
Tamaño de la memoria TEXT:      144 kB
# de cambios de contexto voluntarios: 2497
# de cambios de contexto no voluntarios: 9
```

CP05 Ausencia de [Pid] en una lista

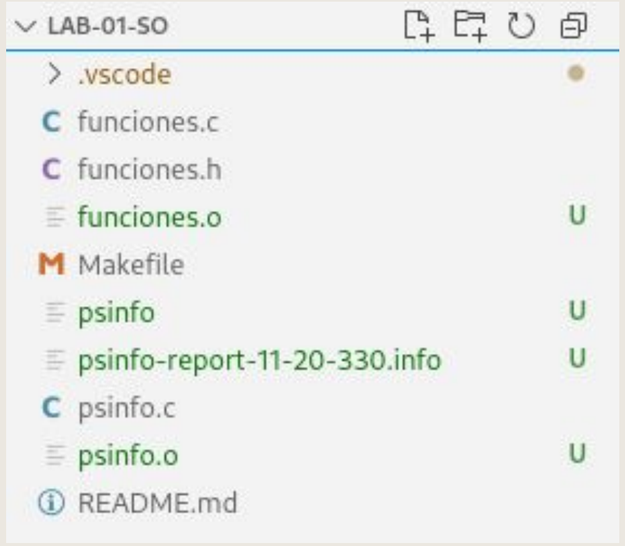
Ahora podemos contemplar la posibilidad de que pasemos una lista de [pid] con el argumento -l, pero puede que alguno de esos no sea el id a un proceso

En este caso el [pid=21] no existía dentro de la carpeta /proc, por lo que arroja el mensaje y pudo mostrar el siguiente pid

```
┌──(jonasⓀJonathanG)-[~/Documents/lab-01-SO]
└─$ ./psinfo -l 11 21 330
-- Información recolectada!!!

Pid: 11
Nombre del proceso: bash
Estado del proceso: S (sleeping)
Tamaño de la imagen de memoria:     7412 kB
Tamaño de la memoria DATA:      700 kB
Tamaño de la memoria STACK:      132 kB
Tamaño de la memoria TEXT:      804 kB
# de cambios de contexto voluntarios: 118
# de cambios de contexto no voluntarios: 0

Pid: 21
The ID was not found, the process does not exist.

Pid: 330
Nombre del proceso: tigervncserver
Estado del proceso: S (sleeping)
Tamaño de la imagen de memoria:    19260 kB
Tamaño de la memoria DATA:     8736 kB
Tamaño de la memoria STACK:      132 kB
Tamaño de la memoria TEXT:     1720 kB
# de cambios de contexto voluntarios: 8
# de cambios de contexto no voluntarios: 0
```

CP06 Reporte de información de procesos

La etapa de la funcionalidad `-r` nos menciona la posibilidad de poder generar un reporte simplemente dando la lista de `pids` necesarios en dicho reporte




```
┌──(jonas㉿JonathanG)-[~/Documents/lab-01-SO]
└─$ ./psinfo -r 11 20 330
```

Archivo de salida generado: psinfo-report-11-20-330.info

psinfo-report-11-20-330.info

```
Pid: 11
Nombre del proceso: bash
Estado del proceso: S (sleeping)
Tamaño de la imagen de memoria:     7412 kB
Tamaño de la memoria DATA:      700 kB
Tamaño de la memoria STACK:      132 kB
Tamaño de la memoria TEXT:      804 kB
# de cambios de contexto voluntarios: 118
# de cambios de contexto no voluntarios: 0

Pid: 20
Nombre del proceso: pulseaudio.exe
Estado del proceso: S (sleeping)
Tamaño de la imagen de memoria:     2476 kB
Tamaño de la memoria DATA:      192 kB
Tamaño de la memoria STACK:      132 kB
Tamaño de la memoria TEXT:     1484 kB
# de cambios de contexto voluntarios: 15
# de cambios de contexto no voluntarios: 0

Pid: 330
Nombre del proceso: tigervncserver
Estado del proceso: S (sleeping)
Tamaño de la imagen de memoria:    19260 kB
Tamaño de la memoria DATA:     8736 kB
Tamaño de la memoria STACK:      132 kB
Tamaño de la memoria TEXT:     1720 kB
# de cambios de contexto voluntarios: 12
# de cambios de contexto no voluntarios: 0
```

CP07 Ausencia de proceso en un reporte

Se tiene exactamente el mismo funcionamiento de la opción -l, queremos que si no existe algún proceso, se guarde el mensaje de error en la estructura y se sigue iterando sobre los demás [pids]

```
┌──(jonas㉿JonathanG)-[~/Documents/lab-01-SO]
└─$ ./psinfo -r 11 21 330
Archivo de salida generado: psinfo-report-11-21-330.info
```

psinfo-report-11-21-330.info

Pid: 11
Nombre del proceso: bash
Estado del proceso: S (sleeping)
Tamaño de la imagen de memoria: 7412 kB
Tamaño de la memoria DATA: 700 kB
Tamaño de la memoria STACK: 132 kB
Tamaño de la memoria TEXT: 804 kB
de cambios de contexto voluntarios: 118
de cambios de contexto no voluntarios: 0

Pid: 21
The ID was not found, the process does not exist.

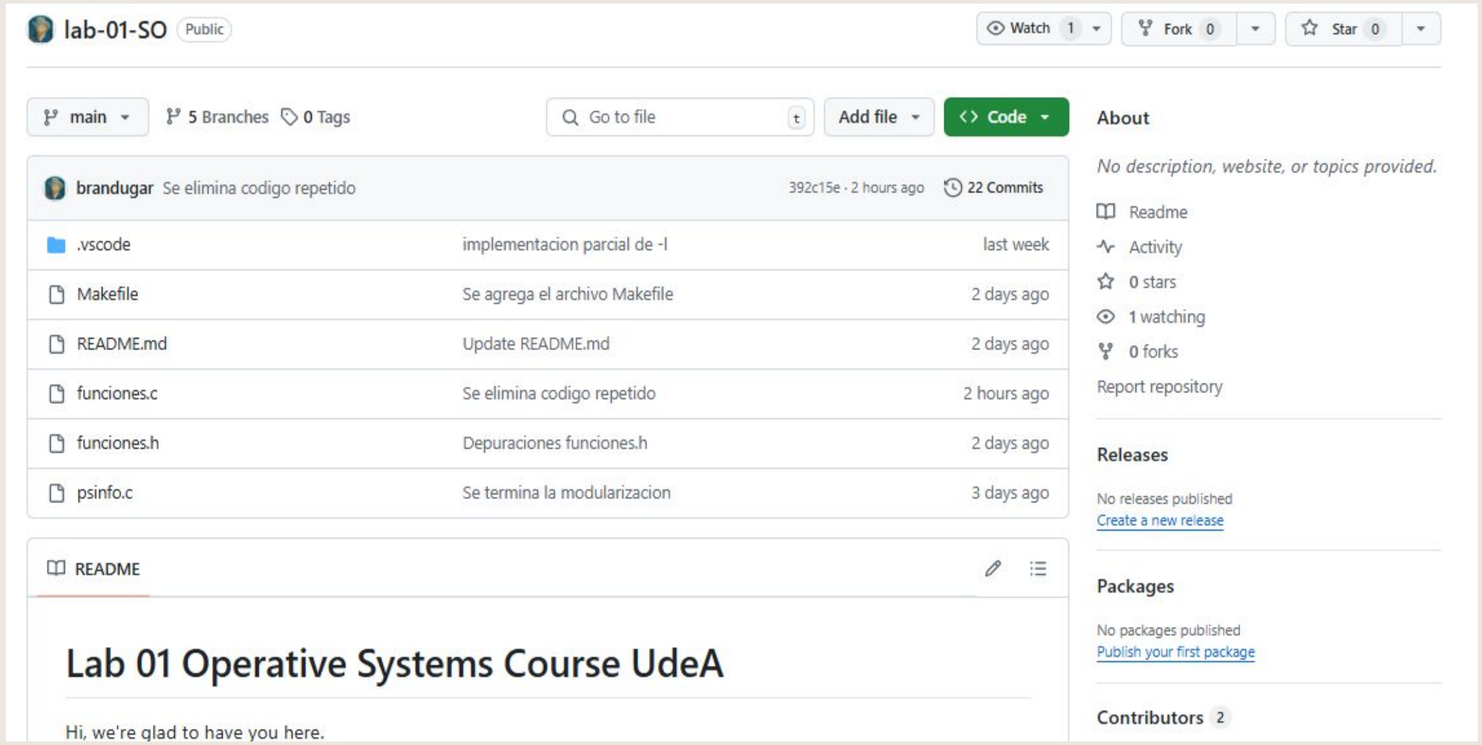
Pid: 330
Nombre del proceso: tigervncserver
Estado del proceso: S (sleeping)
Tamaño de la imagen de memoria: 19260 kB
Tamaño de la memoria DATA: 8736 kB
Tamaño de la memoria STACK: 132 kB
Tamaño de la memoria TEXT: 1720 kB
de cambios de contexto voluntarios: 8
de cambios de contexto no voluntarios: 0

Repositorio de Github

Documentar casos de prueba

Versionamiento del código.
Repositorio donde se llevó el proyecto

Link del repositorio



Ramas del repositorio

`main` → Rama principal del repositorio, donde juntamos los PRs

`fase2` → Se implementan los ajustes necesarios para usar `-l`

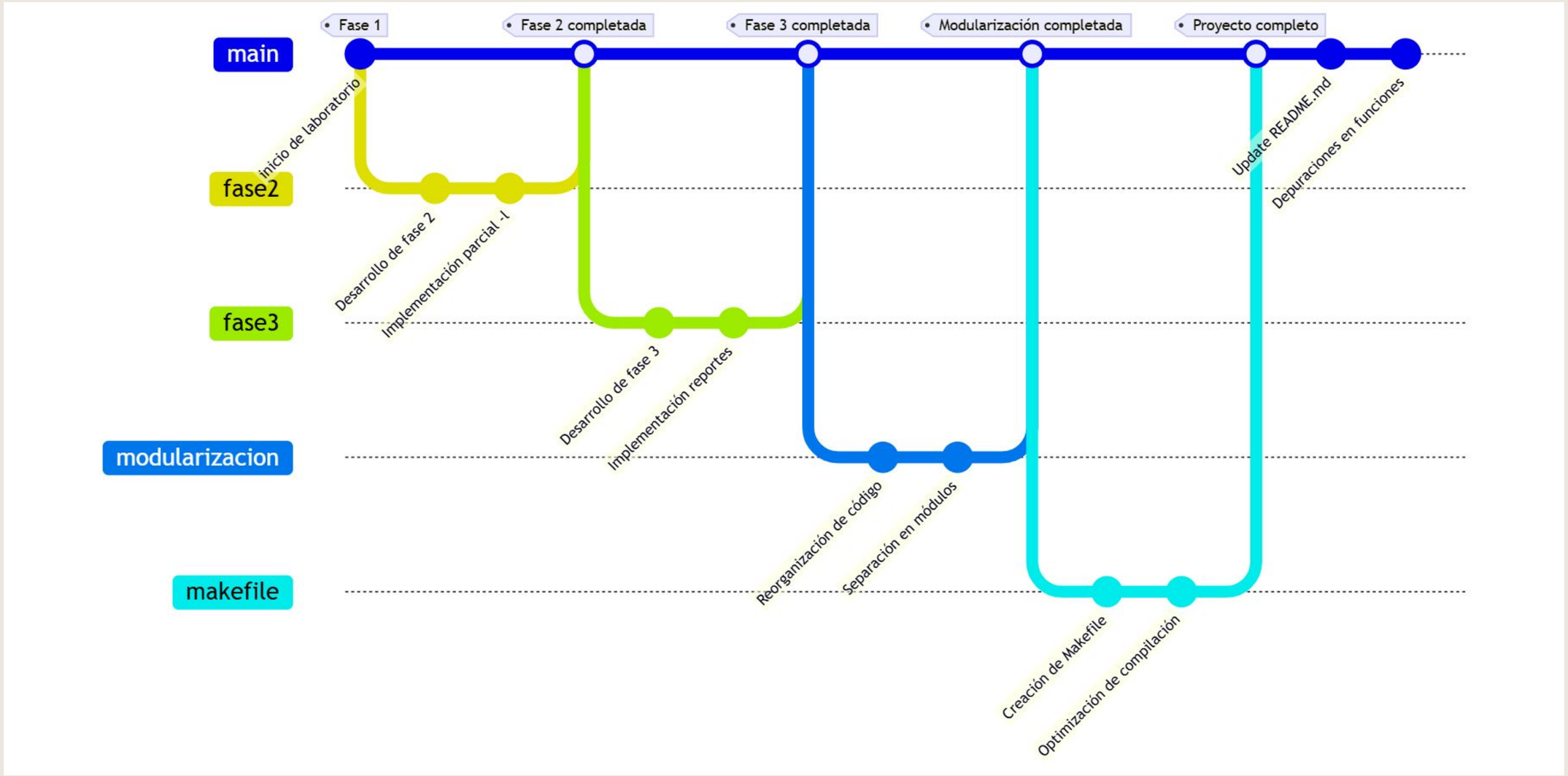
`fase3` → Se implementa la opción para la creación de reportes

`modularización` → Se tiene para separar funcionalidades y código

`makefile` → Se usa para implementar el archivo makefile del proyecto

UNIVERSIDAD DE ANTIOQUIA
1803

Estructura general del versionamiento del proyecto



Conclusiones y aspectos a mejorar

Aspectos y conclusiones personales sobre el laboratorio

Conclusiones y mejoras.

Es importante destacar que el laboratorio trajo consigo conocimientos en el lenguaje C muy importantes, se considera que el aspecto de las funcionalidades para procesar Strings, y todo la familia de funciones fue vital en el proyecto.

```
#include <string.h>
```

```
strcat()        strcpy()        strcmp()
```

También lo de archivos resulta muy productivo para el aprendizaje, e importante para la práctica.

```
#Manejar archivos de tipo FILE y los modos de manejo
```

Implementar funciones de este tipo resultan muy interesantes y productivas para implementar, nutre bastante el conocimiento acerca del funcionamiento de Linux

Se considera importante aspectos a mejorar, como la estructura de impresión del proceso, abstraer o almacenar la información del proceso, no simplemente una cadena extensa de caracteres

También tratar de tomar alternativas más simples a la hora de procesar el [pid], diferentes a usar expresiones regulares

Por último creemos que es importante separar cada etapa como estructuras, una estructura para un proceso individual, y luego otra mayor para almacenar la lista de estas estructuras

Anexos

Integrante #1: Brandon Duque Garcia
brandon.duque@udea.edu.co

Integrante #2: Jonathan Andrés Granda Orrego
jonathan.granda@udea.edu.co

Repositorio de GitHub
https://github.com/brandugar/lab-01-SO

Carpeta Drive

https://drive.google.com/drive/u/1/folders/1_IfZMBbNNLyaulsk3sgdC06WFwGQroaW