Informe escrito Laboratorio #1
Sistemas operativos

Estudiantes:

Jhon Alexander Botero Gómez

Cristian Daniel Muñoz Botero

Profesor:

Danny Alexandro Múnera Ramírez

Departamento de Ingeniería de Sistemas

Facultad de Ingeniería

Universidad de Antioquia

2025

Repositorio del programa

https://github.com/cristianmunoz1/labs-so
En la carpeta lab01-psinfo/version1

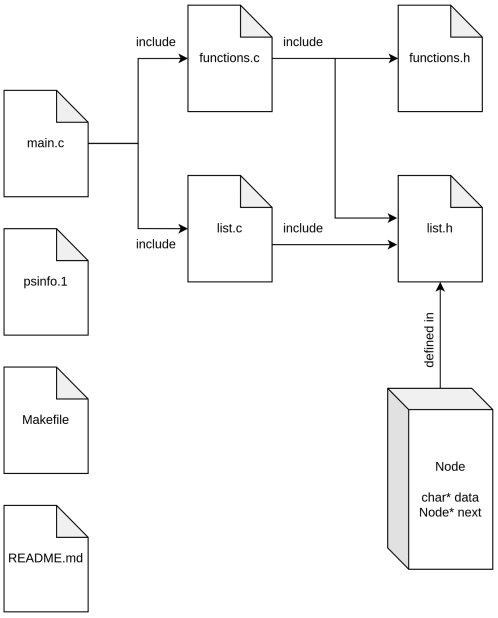
Instalación del programa

Para instalar el programa se deben ejecutar los siguientes comandos:

make
make install-man (opcional)

Este último comando permitirá consultar la man page del programa.

Estructura del programa



El programa psinfo cuenta con los siguientes archivos:

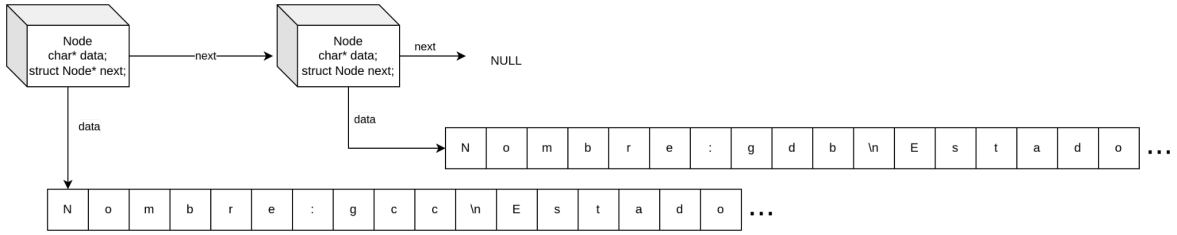
- **main.c**: Clase principal, el punto de entrada de la aplicación
- **list.h**: Declaración de estructura Node. Declaración de funciones necesarias para usar la estructura Node como una lista ligada simple. Borrar al principio. Insertar al final. Crear nodo.
- **list.c**: Implementación de las funciones del archivo list.h
- **functions.h**: Declaración de funciones que serán usadas a lo largo del main para cumplir las diferentes tareas que se presentan en el desarrollo del laboratorio. Serán explicadas con mayor detalle adelante.
- **functions.c**: Implementación de funciones necesarias para el desarrollo de toda la actividad.
- **Makefile**: Archivo con todas las reglas de compilación del proyecto.
- **README.md**: Archivo que contiene información del programa, como instrucciones de instalación, ejemplos, etc.
- **psinfo.1**: Archivo que contiene texto, el cual será usado para construir la manpage del programa con ayuda del Makefile.

Estructura Node

La estructura Node se piensa como una manera de usar listas ligadas en el programa. Agrupa 2 variables. Estas variables son:

char* data;
struct Node next;

“data” es un puntero a caracter. Dicho puntero está presente para guardar toda la información que se requiere de cada proceso. Cada nodo con data apunta a la dirección donde está guardada toda la información de un proceso en particular.



Archivo list.h

Este archivo se usa para definir la estructura Node que se explicó anteriormente, y las funciones necesarias para tratar varios de ellos como una lista ligada.

```c
// Condicional que evita que un archivo header se incluya varias veces al momento de compilar
#ifndef LIST_H
#define LIST_H

// Definición de la estructura del nodo
struct Node {
    char* data;
    struct Node* next;
};

// Declaración de las funciones de la lista
struct Node* createNode(const char* data);
void insertAtEnd(struct Node** head, const char* data);
void deleteFromFirst(struct Node** head);
void print(struct Node* head);

#endif // LIST_H
```

createNode para alojar en memoria una nueva estructura nodo, en caso de ser necesario.
insertAtEnd para cada que se cree un nuevo nodo con información de procesos, se ligue al nodo anterior. Esto sirve para que las estructuras no se pierdan en la memoria, y puedan ser recorridos eficientemente.
deleteFromFirst para que cada que leamos o pasemos la información de un proceso, alojada en un nodo al informe, dicho nodo se elimine, guardando recursos del sistema.
print para imprimir la información de un nodo en su variable data.

Archivo list.c

Este archivo se utiliza para definir todas las funciones que declaramos en el archivo list.h explicado anteriormente. Implementa los métodos que se mencionaron anteriormente. Crea un nodo dinámicamente con malloc en createNode y ayuda a manejar correctamente como cola una lista de nodos.

Archivo functions.h

Este archivo ayuda a definir ciertas funciones que vimos necesarias en la implementación del programa psinfo. Dicha necesidad se presentaba a medida que avanzábamos en el proyecto y por lo tanto, dichas funciones no tienen una finalidad específica entre todas. Es un mix de funciones que permiten al programa ejecutarse correctamente y mantener un método main limpio.

```h
#ifndef FUNCTIONS_H
#define FUNCTIONS_H
#include "list.h"

//Función que escribirá el reporte de los procesos en un archivo
void writeReport(int argc, char* argv[], struct Node* head, int firstProcessPosition);
//Función que encontrará la posición del primer argumento que sea un proceso dentro de
int firstProcessPosition(int argc, char* argv[]);
//Función que indica si un valor es numérico o no
int isNumeric(const char* str);
//Función que guardará toda la información de un proceso dado su process id en la cola
int storeProcessInfo(char pid[], struct Node** queue);
//Función que imprimirá toda la información de los procesos que hay almacenados en una
void printProcessQueue(struct Node* head);
//Función que procesa parcialmente los argumentos que se pasen al comando psinfo, encon
int processPIDs(int argc, char* argv[], struct Node** queue, int *hasR, int *hasL);

#endif // FUNCTIONS_H
```

Las funciones que se implementan en este archivo son:

- printProcessQueue: Esta función recibe el nodo cabeza de una lista ligada de Nodes. Y permite ir por cada uno de los nodos e imprimir la variable data sin borrar los nodos. Recorre la lista sin afectarla imprimiendo en consola la data de cada uno de ellos.
- storeProcessInfo: Esta función recibe el id de un proceso y una cola. Su función principal es guardar la información del proceso con ese pid en el último nodo de la lista ligada que se le pase por parámetro. Consulta la información del proceso y la pone de última en la cola en una estructura Node.
- isNumeric: Esta función sencilla se encarga de validar que un puntero a caracter esté exclusivamente formado por dígitos. Osea, que sea numérico.
- firstProcessPosition: Esta función devuelve un número, el cual es la posición del primer argumento que es un proceso dentro de una lista de argumentos que se pasan al comando. Recibe argc y argv. Esto para que pueda verificar cuantos argumentos se pasaron y cuales son. Analiza cada uno de ellos y devuelve la posición del primero que no sea una flag u otro tipo de argumento.
- writeReport: Esta función recibe argc, argv, el nodo cabeza de una lista ligada previamente cargada con información de procesos y la posición del primer argumento que es un proceso. La función principal es coger la lista ligada, recorrerla e ir escribiendo dentro de un archivo .info la información de cada uno de los procesos para tenerla a la mano si se quiere consultar el reporte.
- processPIDs: Esta función es la más compleja del programa, ya que además de su longitud tiene una estructura con varios condicionales que puede confundir. Esta función recibe argc, argv, el nodo cabeza de una lista ligada en este caso vacía y 2 punteros a entero, hasL y hasR. hasL indica si en los argumentos del comando está la flag -L, lo mismo pasa para hasR. Esta función se encarga de verificar que el

comando se esté escribiendo de manera correcta y advertirle al usuario si está haciendo un uso indebido del comando. Esta función se encarga de varias cosas:

- Verifica que si envía un solo pid, “psinfo 123” se ejecute el programa normalmente.
- Verifica que si se envía “psinfo” sin argumentos el programa no se ejecute y envíe un mensaje al usuario.
- Verifica que si se envía “psinfo -l” sin pids, entonces se envíe una advertencia al usuario para que ingrese al menos un pid.
- Verifica que si -l está en el programa, pero no se encuentra de primer argumento, se envíe un error.
- Verifica que si -r está, dicho -r se encuentre al final del programa. Si se encuentra en una posición diferente, se envía un error.

Archivo functions.c

Este archivo implementa todas las funciones que se definieron en functions.h

Archivo main.c

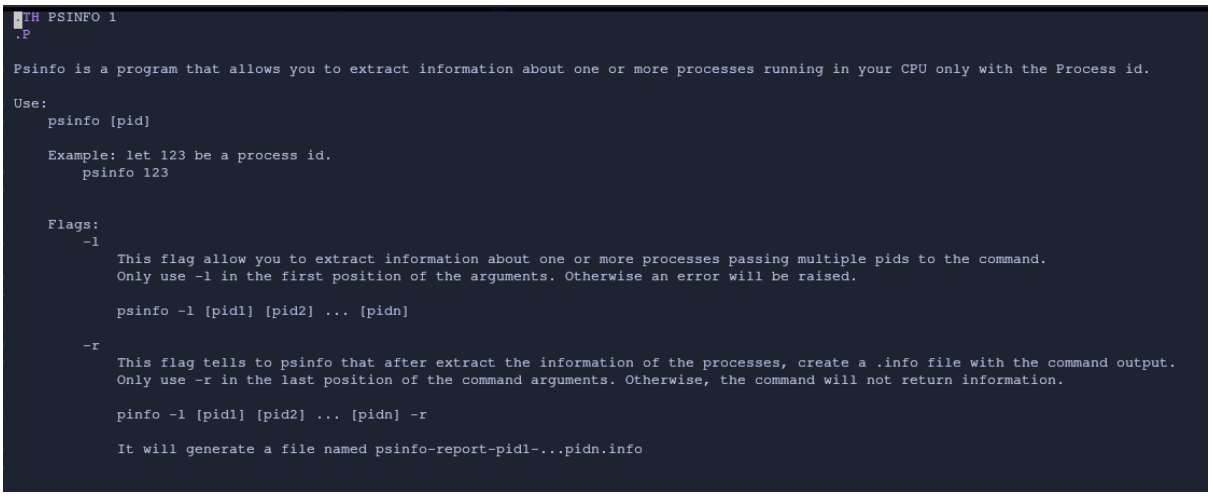
Este archivo es el punto de entrada de la aplicación y principalmente llama a las funciones definidas en functions.c para que el programa siga su rumbo normalmente. También tiene la responsabilidad de inicializar algunas variables como hasR y hasL. Luego llama a procesar los pids y por último, si -r está en los argumentos del comando, entonces llama a la función para generar un reporte.

```c
//Librería que contiene el método para validar si un caracter es dígito o no
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "list.h"
#include "functions.h"

int main(int argc, char* argv[]) {
    int hasL = 0;
    int hasR = 0;
    struct Node* queue = NULL;

    if (processPIDs(argc, argv, &queue, &hasR, &hasL) != 0) {
        return -1;
    }

    if (hasR) {
        int pos = firstProcessPosition(argc, argv);
        writeReport(argc, argv, queue, pos);
    }

    printProcessQueue(queue);
    return 0;
}
```

Archivo psinfo.1

Este archivo contiene simplemente texto, en el cual se recrea una man page del programa. Este archivo es usado para pasarlo a la ruta de las man pages del usuario y que este pueda ejecutar “man psinfo” y vea el contenido de psinfo.1

Esto permitirá a los usuarios tener una información más detallada con ejemplos para usar el comando.



```
.TH PSINFO 1
.P

Psinfo is a program that allows you to extract information about one or more processes running in your CPU only with the Process id.

Use:
    psinfo [pid]

    Example: let 123 be a process id.
        psinfo 123


    Flags:
        -l
            This flag allow you to extract information about one or more processes passing multiple pids to the command.
            Only use -l in the first position of the arguments. Otherwise an error will be raised.

            psinfo -l [pid1] [pid2] ... [pidn]

        -r
            This flag tells to psinfo that after extract the information of the processes, create a .info file with the command output.
            Only use -r in the last position of the command arguments. Otherwise, the command will not return information.

            pinfo -l [pid1] [pid2] ... [pidn] -r

            It will generate a file named psinfo-report-pid1-...pidn.info
```

Archivo Makefile

Este archivo se usa para hacer la compilación del programa más sencilla simplemente ejecuando el comando make. En este archivo definimos algunas reglas de compilado y otras reglas, que se ejecutan con otros comandos, de la siguiente manera:

- make: Instala el programa base. Solo compila los archivos dentro de un ejecutable que puede ser usado para extraer la información de los procesos correctamente.
- make install-man: Este comando permite al usuario usar el archivo psinfo.1 para comprimirlo y copiarlo dentro de una ruta específica de su sistema linux en la cual se guardan las man pages en la sección 1. Esto permite la ejecución posterior de “man psinfo” para ver más información acerca del comando.
- make clean: Este comando permite a los usuarios borrar todos los archivos que genera la compilación e incluso los reportes que genere con las ejecuciones de los comandos. Sin tener que borrar archivo por archivo, solo se ejecuta make clean y la carpeta queda igual que antes de compilar el programa.

Archivo README.md

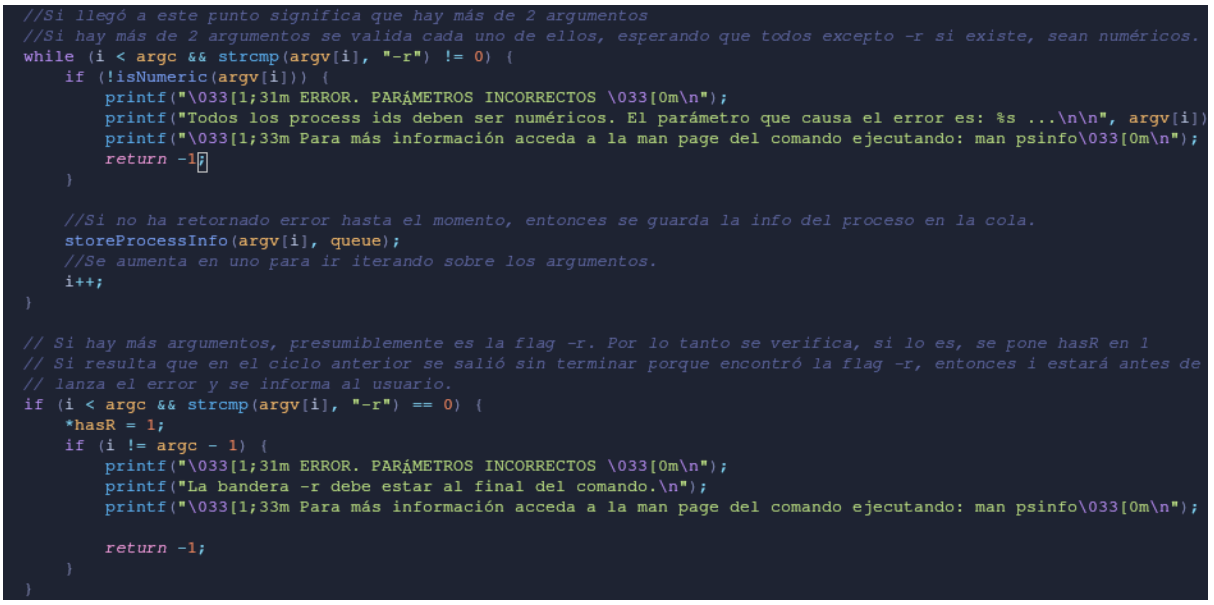
Archivo que se utiliza para visualizar un resúmen del programa, instrucciones de instalación y desinstalación del mismo en GitHub o con un editor de texto.

Funcionamiento principal del programa

Como se puede observar en la función main, hay 4 llamadas a funciones:

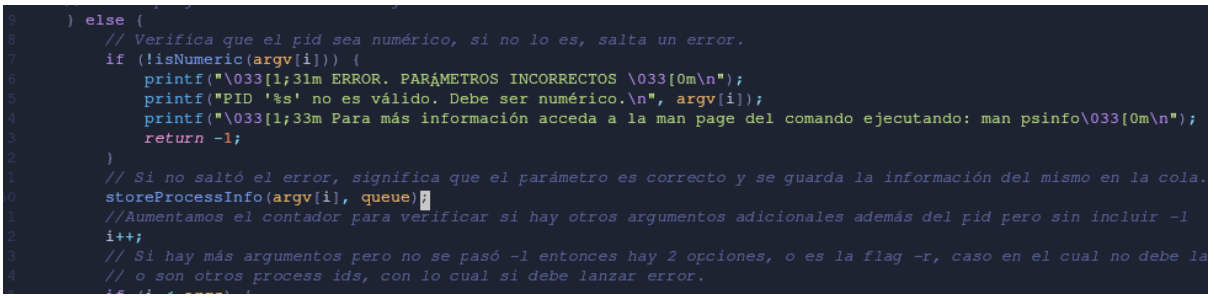
- Se llama a processPIDs para que cargue el valor de hasL y hasR dependiendo de los argumentos que se hayan pasado al comando o devuelva un error si se escribió

de manera incorrecta. Si el usuario ingresa el comando correctamente, entonces processPIDs llama a la función storeProcessInfo para cada uno de los argumentos que corresponden a pids.



```
//Si llegó a este punto significa que hay más de 2 argumentos
//Si hay más de 2 argumentos se valida cada uno de ellos, esperando que todos excepto -r si existe, sean numéricos.
while (i < argc && strcmp(argv[i], "-r") != 0) {
    if (!isNumeric(argv[i])) {
        printf("\033[1;31m ERROR. PARÁMETROS INCORRECTOS \033[0m\n");
        printf("Todos los process ids deben ser numéricos. El parámetro que causa el error es: %s ...\n\n", argv[i]);
        printf("\033[1;33m Para más información acceda a la man page del comando ejecutando: man psinfo\033[0m\n");
        return -1;
    }

    //Si no ha retornado error hasta el momento, entonces se guarda la info del proceso en la cola.
    storeProcessInfo(argv[i], queue);
    //Se aumenta en uno para ir iterando sobre los argumentos.
    i++;
}

// Si hay más argumentos, presumiblemente es la flag -r. Por lo tanto se verifica, si lo es, se pone hasR en 1
// Si resulta que en el ciclo anterior se salió sin terminar porque encontró la flag -r, entonces i estará antes de
// lanza el error y se informa al usuario.
if (i < argc && strcmp(argv[i], "-r") == 0) {
    *hasR = 1;
    if (i != argc - 1) {
        printf("\033[1;31m ERROR. PARÁMETROS INCORRECTOS \033[0m\n");
        printf("La bandera -r debe estar al final del comando.\n");
        printf("\033[1;33m Para más información acceda a la man page del comando ejecutando: man psinfo\033[0m\n");

        return -1;
    }
}
```

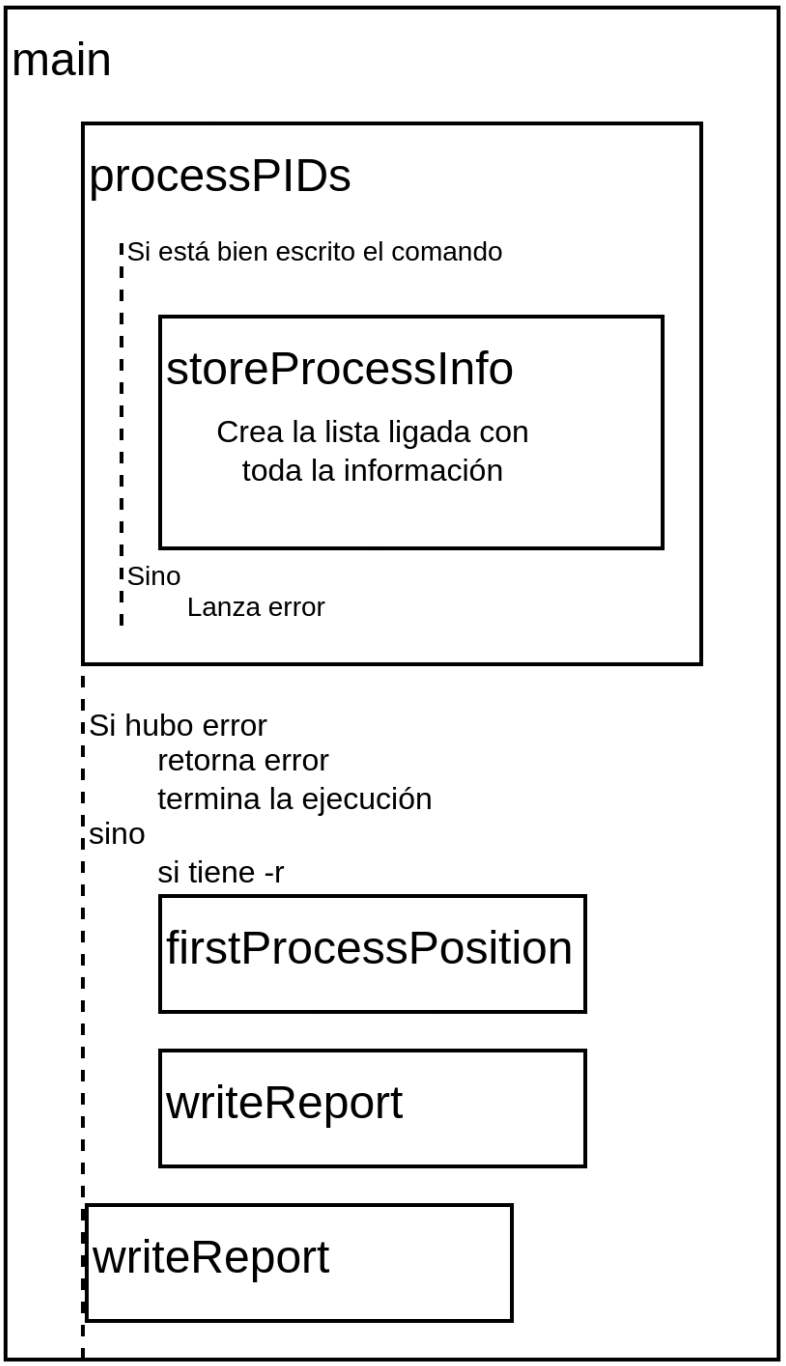
O para un solo argumento pid que se le pase:



```
    } else {
        // Verifica que el pid sea numérico, si no lo es, salta un error.
        if (!isNumeric(argv[i])) {
            printf("\033[1;31m ERROR. PARÁMETROS INCORRECTOS \033[0m\n");
            printf("PID '%s' no es válido. Debe ser numérico.\n", argv[i]);
            printf("\033[1;33m Para más información acceda a la man page del comando ejecutando: man psinfo\033[0m\n");
            return -1;
        }
        // Si no saltó el error, significa que el parámetro es correcto y se guarda la información del mismo en la cola.
        storeProcessInfo(argv[i], queue);
        //Aumentamos el contador para verificar si hay otros argumentos adicionales además del pid pero sin incluir -l
        i++;
        // Si hay más argumentos pero no se pasó -l entonces hay 2 opciones, o es la flag -r, caso en el cual no debe la
        // o son otros process ids, con lo cual si debe lanzar error.
```

Esta función crea la lista ligada con la información de cada proceso en cada nodo.

Si esta función no retorna error, entonces la información está correctamente cargada en memoria y retorna a main.
Ya main con la información cargada, verifica si hay -r en el comando, y si lo hay, calcula la posición del primer argumento que corresponde a un proceso y con esto, llama a la función writeReport para escribir la lista ligada, (toda la información de los procesos cargada en memoria) en un archivo .info.
Por último, se llama a la función printProcessQueue para imprimir la información de los procesos en la terminal.


main
processPIDs
Si está bien escrito el comando
storeProcessInfo
Crea la lista ligada con
toda la información
Sino
Lanza error
Si hubo error
retorna error
termina la ejecución
sino
si tiene -r
firstProcessPosition
writeReport
writeReport

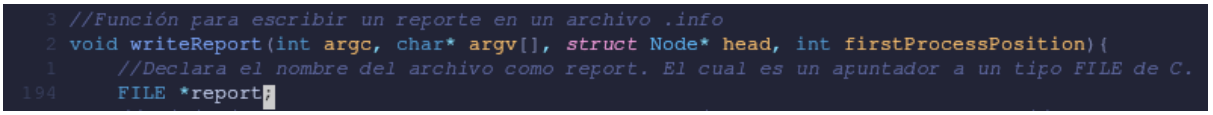
REPORTE DE USO DE IA

Función fprintf

- Se utilizó la IA de Google Gemini para conocer el procedimiento para generar un archivo de texto desde C. Se preguntó en el buscador de Google: “¿Cómo crear un archivo en C?”. Con esto la IA, respondió en el buscador:

```c
#include <stdio.h>

int main() {
    FILE *archivo;
    archivo = fopen("mi_archivo.txt", "w"); // Abre en modo escritura

    if (archivo == NULL) {
        printf("Error al abrir el archivo.\n");
        return 1; // Indica un error
    }

    fprintf(archivo, "Hola, este es el contenido del archivo.\n");
    fprintf(archivo, "Esta es otra linea.\n");

    fclose(archivo);

    printf("Archivo creado y escrito correctamente.\n");

    return 0;
}
```

Acerca de la respuesta, ya sabíamos de la función fopen() pero no sabíamos como usarla en modo escritura. Entonces creamos un puntero a archivo



Lo abrimos con fopen en modo escritura

```
226     report = fopen(nombre_archivo, "w");
```

Y ahora con fprintf ingresar en el archivo los datos

```
229     while (temp != NULL){
1           fprintf(report, "%s\n", temp->data);
2           temp = temp->next;
3       }
```

La función fprintf, Gemini no profundizó acerca de ella, por lo que se decidió buscar en las man pages.
Cuando se consulta la función fprintf en las manpages dice lo siguiente:

The functions in the printf() family produce output according to a format as described below. The functions printf() and vprintf() write output to stdout, the standard output stream; fprintf() and vfprintf() write output to the given output stream; sprintf(), snprintf(), vsprintf(), and vsnprintf() write to the character string str.

Son una familia de funciones, y en específico, fprintf ayuda a escribir una salida a un stream dado. Junto con vprintf. Pero esto abre otra interrogante, ¿qué diferencia hay entre fprintf y vprintf?
En una sección posterior de la man page se puede leer lo siguiente

The functions vprintf(), vfprintf(), vdprintf(), vsprintf(), vsnprintf() are equivalent to the functions printf(), fprintf(), dprintf(), sprintf(), snprintf(), respectively, except that they are called with a va_list instead of a variable number of arguments. These functions do not call the va_end macro. Because they invoke the va_arg macro, the value of ap is undefined after the call. See stdarg(3).

All of these functions write the output under the control of a format string that specifies how subsequent arguments (or arguments accessed via the variable-length argument facilities of stdarg(3)) are converted for output.

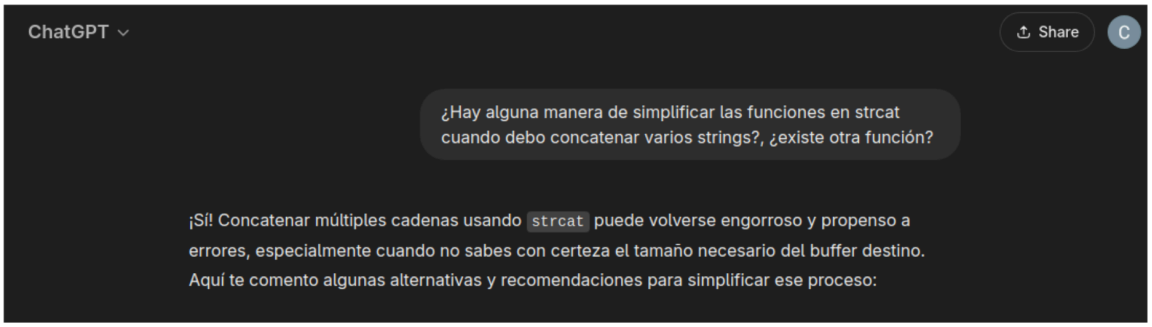
Esto indica que vprintf y las demás son llamadas con una va_list, en vez de una variable con el número de argumentos. Esto es útil cuando se están creando funciones propias como versiones de printf. Por el momento, no tenemos el conocimiento de esto y no es necesario para el desarrollo del laboratorio, y por lo tanto, **fprintf** sigue siendo la más propicia para usar en el código.

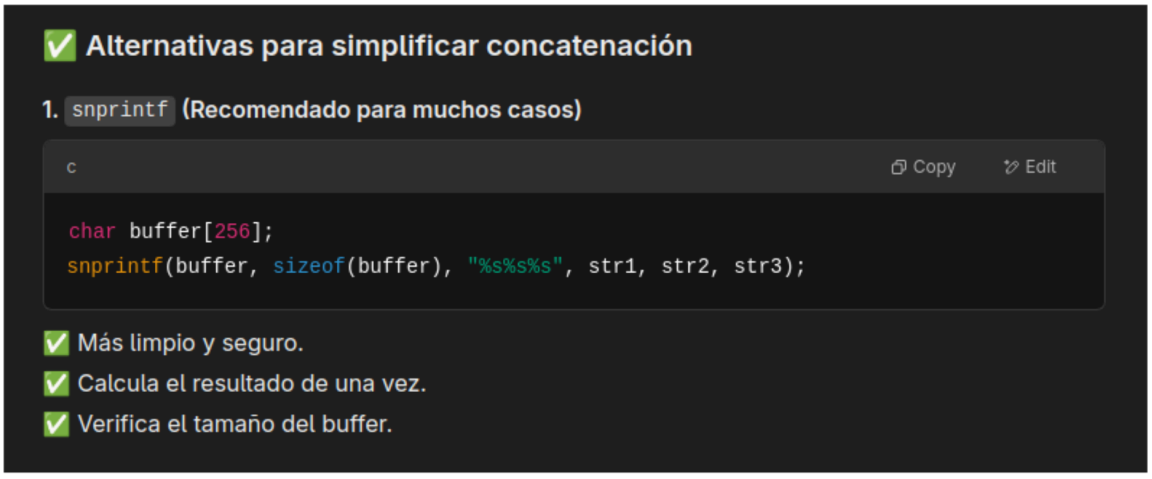
Simplificación de función strcat

En una versión del código teníamos lo siguiente

```c
int storeProcessInfo(char pid[], struct Node** queue){
    // Variable para el inicio del comando
    char inicio_comando[50] = "cat /proc/";
    // Concatenamos el inicio del comando con el primer argumento, que sería el pid
    strcat(inicio_comando, pid);
    // Concatenamos con /status para que el programa imprima la información de estado del proceso
    // El comando queda como: cat /proc/123/status
    // Siendo 123 un ejemplo de pid
    strcat(inicio_comando, "/status");
```

Esto hacía, que para concatenar un string grande, o varios strings, hacía que tuviéramos que llamar la función strcat varias veces. Hacía que en ocasiones se desbordara al concatenar strings y que superara el espacio que se había reservado para el string anteriormente.
Por lo que se preguntó en la herramienta ChatGPT lo siguiente:





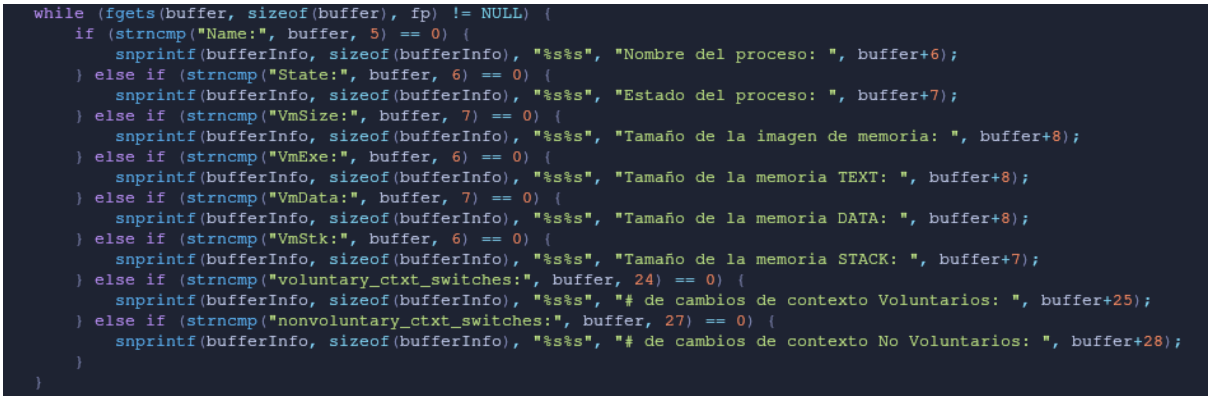
Se puede usar la función snprintf. Esta función según la herramienta recibe varios argumentos: un buffer donde se almacenará el string concatenado, el tamaño del buffer para que la función valide que sí haya espacio en el mismo para el string, el formato y los strings a concatenar como usando la función printf normalmente. Teniendo esto en cuenta, podemos entonces simplificar el pedazo de código donde se llamaba varias veces strcat y ahorrarnos la declaración y definición de una variable que guarde el path inicial. Quedando de la siguiente manera:

```c
    char path[64];
    snprintf(path, sizeof(path), "/proc/%s/status", pid);
```

En la siguiente imágen cambiamos todos los strcat por snprintf

```
    // Fgets ayuda a abrir un pipe del archivo fp y guardar línea por línea en el buffer que defi
    while (fgets(buffer, sizeof(buffer), fp) != NULL) {
        if (strncmp("Name:", buffer, 5) == 0) {
            snprintf(bufferInfo, sizeof(bufferInfo), "%s%s", "Nombre del proceso: ", buffer+6);
        } else if (strncmp("State:", buffer, 6) == 0) {
            snprintf(bufferInfo, sizeof(bufferInfo), "%s%s", "Estado del proceso: ", buffer+7);
        } else if (strncmp("VmSize:", buffer, 7) == 0) {
            strcat(bufferInfo, "Tamaño de la imagen de memoria: ");
            strcat(bufferInfo, buffer + 8);
        } else if (strncmp("VmExe:", buffer, 6) == 0) {
            strcat(bufferInfo, "Tamaño de la memoria TEXT: ");
            strcat(bufferInfo, buffer + 7);
        } else if (strncmp("VmData:", buffer, 7) == 0) {
            strcat(bufferInfo, "Tamaño de la memoria DATA: ");
            strcat(bufferInfo, buffer + 8);
        } else if (strncmp("VmStk:", buffer, 6) == 0) {
            strcat(bufferInfo, "Tamaño de la memoria STACK: ");
            strcat(bufferInfo, buffer + 7);
        } else if (strncmp("voluntary_ctxt_switches:", buffer, 24) == 0) {
            strcat(bufferInfo, "# de cambios de contexto voluntarios: ");
            strcat(bufferInfo, buffer + 25);
        } else if (strncmp("nonvoluntary_ctxt_switches:", buffer, 27) == 0) {
            strcat(bufferInfo, "# de cambios de contexto no voluntarios: ");
            strcat(bufferInfo, buffer + 28);
        }
    }
```

Al terminar, el ciclo quedó de la siguiente manera:



```
while (fgets(buffer, sizeof(buffer), fp) != NULL) {
    if (strncmp("Name:", buffer, 5) == 0) {
        snprintf(bufferInfo, sizeof(bufferInfo), "%s%s", "Nombre del proceso: ", buffer+6);
    } else if (strncmp("State:", buffer, 6) == 0) {
        snprintf(bufferInfo, sizeof(bufferInfo), "%s%s", "Estado del proceso: ", buffer+7);
    } else if (strncmp("VmSize:", buffer, 7) == 0) {
        snprintf(bufferInfo, sizeof(bufferInfo), "%s%s", "Tamaño de la imagen de memoria: ", buffer+8);
    } else if (strncmp("VmExe:", buffer, 6) == 0) {
        snprintf(bufferInfo, sizeof(bufferInfo), "%s%s", "Tamaño de la memoria TEXT: ", buffer+8);
    } else if (strncmp("VmData:", buffer, 7) == 0) {
        snprintf(bufferInfo, sizeof(bufferInfo), "%s%s", "Tamaño de la memoria DATA: ", buffer+8);
    } else if (strncmp("VmStk:", buffer, 6) == 0) {
        snprintf(bufferInfo, sizeof(bufferInfo), "%s%s", "Tamaño de la memoria STACK: ", buffer+7);
    } else if (strncmp("voluntary_ctxt_switches:", buffer, 24) == 0) {
        snprintf(bufferInfo, sizeof(bufferInfo), "%s%s", "# de cambios de contexto Voluntarios: ", buffer+25);
    } else if (strncmp("nonvoluntary_ctxt_switches:", buffer, 27) == 0) {
        snprintf(bufferInfo, sizeof(bufferInfo), "%s%s", "# de cambios de contexto No Voluntarios: ", buffer+28);
    }
}
```

Más corto y conciso.

Pero al ejecutarlo, lo que antes arrojaba la información de los procesos, ahora solo arroja la última línea que son los cambios de contexto no voluntarios.

```
[cristian@my-arch version1]$ make
gcc -c main.c
gcc -c list.c
gcc -c functions.c
gcc -o psinfo main.o list.o functions.o
[cristian@my-arch version1]$ ./psinfo 1

INFORMACIÓN DE LOS PROCESOS:
-----------------------------------------------
# de cambios de contexto No Voluntarios: 1328

-----------------------------------------------
[cristian@my-arch version1]$ htop
[cristian@my-arch version1]$ ./psinfo 1790

INFORMACIÓN DE LOS PROCESOS:
-----------------------------------------------
# de cambios de contexto No Voluntarios: 7

-----------------------------------------------
```

Por lo que iniciamos un proceso de debuggeo para solucionar el error:

```
[cristian@my-arch version1]$ make clean
rm -f *.o psinfo *.info psinfo.1.gz
rm -f /home/cristian/.local/bin/psinfo
[cristian@my-arch version1]$ ls
estudio-C_annotated.pdf  functions.c  functions.h  list.c  list.h  main.c  Makefile  psinfo.1  README.md
[cristian@my-arch version1]$ gcc -g functions.c list.c main.c -o psinfo
[cristian@my-arch version1]$ gdb --args psinfo 1790
GNU gdb (GDB) 16.2
Copyright (C) 2024 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-pc-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from psinfo...
(gdb)
```

```
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-pc-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from psinfo...
(gdb) list 14
9           int hasL = 0;
10          int hasR = 0;
11          struct Node* queue = NULL;
12
13          if (processPIDs(argc, argv, &queue, &hasR, &hasL) != 0) {
14              return -1;
15          }
16
17          if (hasR) {
18              int pos = firstProcessPosition(argc, argv);
(gdb)
```

El error se encuentra dentro de processPIDs en la función storeProcessInfo, por lo que ponemos un breakpoint en 13 para entrar a la función

```
Breakpoint 1 at 0x200e: file main.c, line 13.
(gdb)
```

Corremos el programa

```
(gdb) run
Starting program: /home/cristian/Documents/udea/semestre-8/sistemas-operativos/labs-so/lab01-psinfo/version1/psinfo 1790

This GDB supports auto-downloading debuginfo from the following URLs:
  <https://debuginfod.archlinux.org>
Enable debuginfod for this session? (y or [n]) y
Debuginfod has been enabled.
To make this setting permanent, add 'set debuginfod enabled on' to .gdbinit.
Downloading separate debug info for /lib64/ld-linux-x86-64.so.2
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/usr/lib/libthread_db.so.1".

Breakpoint 1, main (argc=2, argv=0x7fffffffe488) at main.c:13
13          if (processPIDs(argc, argv, &queue, &hasR, &hasL) != 0) {
(gdb)
```

```
(gdb) step
processPIDs (argc=2, argv=0x7fffffffe488, queue=0x7fffffffe350, hasR=0x7fffffffe348, hasL=0x7fffffffe344) at functions.c:79
79          if (argc < 2) {
(gdb) list
74      }
75
76
77      int processPIDs(int argc, char* argv[], struct Node** queue, int *hasR, int *hasL) {
78          //Si hay menos de 2 argumentos, solo se ejecutó el comando y no se pasaron pids ni flags
79          if (argc < 2) {
80              printf("\033[1;33m ------------------- Información de PSINFO --------------------\n\033[0m");
81              printf("Uso: psinfo [ -l pid1 pid2 ... ] [ -r ]\n");
82              printf("\nPara ejemplos más detallados del comando, visita la man page del programa ejecutando \"man psinfo\"\n\n");
83              return -1;
(gdb)
```

```
(gdb) next
86          int i = 1;
(gdb)
90          if (strcmp(argv[i], "-l") == 0) {
(gdb)
135             if (!isNumeric(argv[i])) {
(gdb)
142             storeProcessInfo(argv[i], queue);
(gdb) step
storeProcessInfo (pid=0x7fffffffe823 "1790", queue=0x7fffffffe350) at functions.c:17
17      int storeProcessInfo(char pid[], struct Node** queue){
(gdb)
```

```
(gdb) list
14          return 1;
15      }
16
17      int storeProcessInfo(char pid[], struct Node** queue){
18          char path[64];
19          snprintf(path, sizeof(path), "/proc/%s/status", pid);
20          // Declaramos un buffer para leer cada línea del archivo que retorne la función popen. 128 porque es lo máximo que ocupa una línea.
21          char buffer[128];
22          //Buffer que almacenará toda la información necesaria de un proceso y la pasará a la data de un nuevo nodo
23          char bufferInfo[2048] = "";
(gdb)
```

```
(gdb) next
23          char bufferInfo[2048] = "";
(gdb) next
26          FILE *fp = fopen(path, "r");
(gdb) next
29          if (fp == NULL) {
(gdb) next
35          while (fgets(buffer, sizeof(buffer), fp) != NULL) {
(gdb) next
36              if (strncmp("Name:", buffer, 5) == 0) {
(gdb) next
37                  snprintf(bufferInfo, sizeof(bufferInfo), "%s%s", "Nombre del proceso: ", buffer+6);
(gdb) print bufferInfo
$1 = '\000' <repeats 2047 times>
(gdb) next
35          while (fgets(buffer, sizeof(buffer), fp) != NULL) {
(gdb) print bufferInfo
$2 = "Nombre del proceso: qmgr\n", '\000' <repeats 2022 times>
(gdb)
```

```
(gdb) next
36              if (strncmp("Name:", buffer, 5) == 0) {
(gdb) next
38              } else if (strncmp("State:", buffer, 6) == 0) {
(gdb) next
40              } else if (strncmp("VmSize:", buffer, 7) == 0) {
(gdb) next
42              } else if (strncmp("VmExe:", buffer, 6) == 0) {
(gdb) next
44              } else if (strncmp("VmData:", buffer, 7) == 0) {
(gdb) next
46              } else if (strncmp("VmStk:", buffer, 6) == 0) {
(gdb) next
48              } else if (strncmp("voluntary_ctxt_switches:", buffer, 24) == 0) {
(gdb)
50              } else if (strncmp("nonvoluntary_ctxt_switches:", buffer, 27) == 0) {
(gdb)
35          while (fgets(buffer, sizeof(buffer), fp) != NULL) {
(gdb)
36              if (strncmp("Name:", buffer, 5) == 0) {
(gdb)
38              } else if (strncmp("State:", buffer, 6) == 0) {
(gdb)
39                  snprintf(bufferInfo, sizeof(bufferInfo), "%s%s", "Estado del proceso: ", buffer+7);
(gdb) print bufferInfo
$3 = "Nombre del proceso: qmgr\n", '\000' <repeats 2022 times>
(gdb) next
35          while (fgets(buffer, sizeof(buffer), fp) != NULL) {
(gdb) print bufferInfo
$4 = "Estado del proceso: S (sleeping)\n", '\000' <repeats 2014 times>
(gdb)
```

Por lo que podemos ver que el buffer se está sobreescribiendo. Esto pasa porque snprintf empieza desde buffer y por lo tanto, desde el primer caracter, sobreescribiendo lo que ya hay. Por lo tanto, ya no le debemos pasar bufferInfo sino bufferInfo + strlen(bufferInfo). Esto permite que se empiece a escribir no desde la primera posición del buffer, sino desde la posición donde se encuentra el último string dentro del buffer.

strlen nos trae la longitud de caracteres dentro del buffer.
Ejemplo:
bufferInfo = “”
strlen(bufferInfo) = 0

bufferInfo = “Nombre del proceso: gdmr”
strlen(bufferInfo) = 24

Por lo que a cada bufferInfo le sumamos la longitud del string que está guardado en este momento. Esto hace que el espacio disponible en el buffer se reste en la misma cantidad. Por lo que a sizeof(bufferInfo) también tenemos que restarle strlen(bufferInfo)

```c
// Fgets ayuda a abrir un pipe del archivo fp y guardar línea por línea en el buffer que definimos anteriormente
while (fgets(buffer, sizeof(buffer), fp) != NULL) {
    if (strncmp("Name:", buffer, 5) == 0) {
        snprintf(bufferInfo + strlen(bufferInfo), sizeof(bufferInfo) - strlen(bufferInfo), "%s%s", "Nombre del proceso: ", buffer+6);
    } else if (strncmp("State:", buffer, 6) == 0) {
        snprintf(bufferInfo + strlen(bufferInfo), sizeof(bufferInfo) - strlen(bufferInfo), "%s%s", "Estado del proceso: ", buffer+7);
    } else if (strncmp("VmSize:", buffer, 7) == 0) {
        snprintf(bufferInfo + strlen(bufferInfo), sizeof(bufferInfo) - strlen(bufferInfo), "%s%s", "Tamaño de la imagen de memoria: ", buffer+8);
    } else if (strncmp("VmExe:", buffer, 6) == 0) {
        snprintf(bufferInfo + strlen(bufferInfo), sizeof(bufferInfo) - strlen(bufferInfo), "%s%s", "Tamaño de la memoria TEXT: ", buffer+8);
    } else if (strncmp("VmData:", buffer, 7) == 0) {
        snprintf(bufferInfo + strlen(bufferInfo), sizeof(bufferInfo) - strlen(bufferInfo), "%s%s", "Tamaño de la memoria DATA: ", buffer+8);
    } else if (strncmp("VmStk:", buffer, 6) == 0) {
        snprintf(bufferInfo + strlen(bufferInfo), sizeof(bufferInfo) - strlen(bufferInfo), "%s%s", "Tamaño de la memoria STACK: ", buffer+7);
    } else if (strncmp("voluntary_ctxt_switches:", buffer, 24) == 0) {
        snprintf(bufferInfo + strlen(bufferInfo), sizeof(bufferInfo) - strlen(bufferInfo), "%s%s", "# de cambios de contexto Voluntarios: ", buffer+25);
    } else if (strncmp("nonvoluntary_ctxt_switches:", buffer, 27) == 0) {
        snprintf(bufferInfo + strlen(bufferInfo), sizeof(bufferInfo) - strlen(bufferInfo), "%s%s", "# de cambios de contexto No Voluntarios: ", buffer+28);
    }
}
```

Ahora al ejecutarlo, tira el siguiente resultado

```
[cristian@my-arch version1]$ make
gcc -c main.c
gcc -c list.c
gcc -c functions.c
gcc -o psinfo main.o list.o functions.o
[cristian@my-arch version1]$ ./psinfo 1790

INFORMACIóN DE LOS PROCESOS:
--------------------------------------------------------------------------------
Nombre del proceso: qmgr
Estado del proceso: S (sleeping)
Tamaño de la imagen de memoria:     8912 kB
Tamaño de la memoria DATA:      208 kB
Tamaño de la memoria STACK:      132 kB
Tamaño de la memoria TEXT:      28 kB
# de cambios de contexto Voluntarios: 162
# de cambios de contexto No Voluntarios: 7

--------------------------------------------------------------------------------
```

REFLEXIÓN TÉCNICA

De este error final, queda el aprendizaje de validar para qué sirven las funciones específicamente. Es contraintuitivo para alguien que aprendió con lenguajes que tenían un nivel de abstracción más alto como Java o Python, pero cuando se mira a fondo se puede ver que la diferencia entre algunas funciones y otras es casi obvia.
El uso de una función como snprintf permite al usuario en un buffer escribir un string concatenado de una manera segura validando si el buffer cuenta con suficiente espacio. Esto no significa que si yo hago 2 snprintf con 2 strings diferentes se vayan a concatenar. Para eso está la función strcat. Para concatenar de manera más sencilla, sin tener que validar el espacio que le queda al buffer o desde que posición voy a escribir.

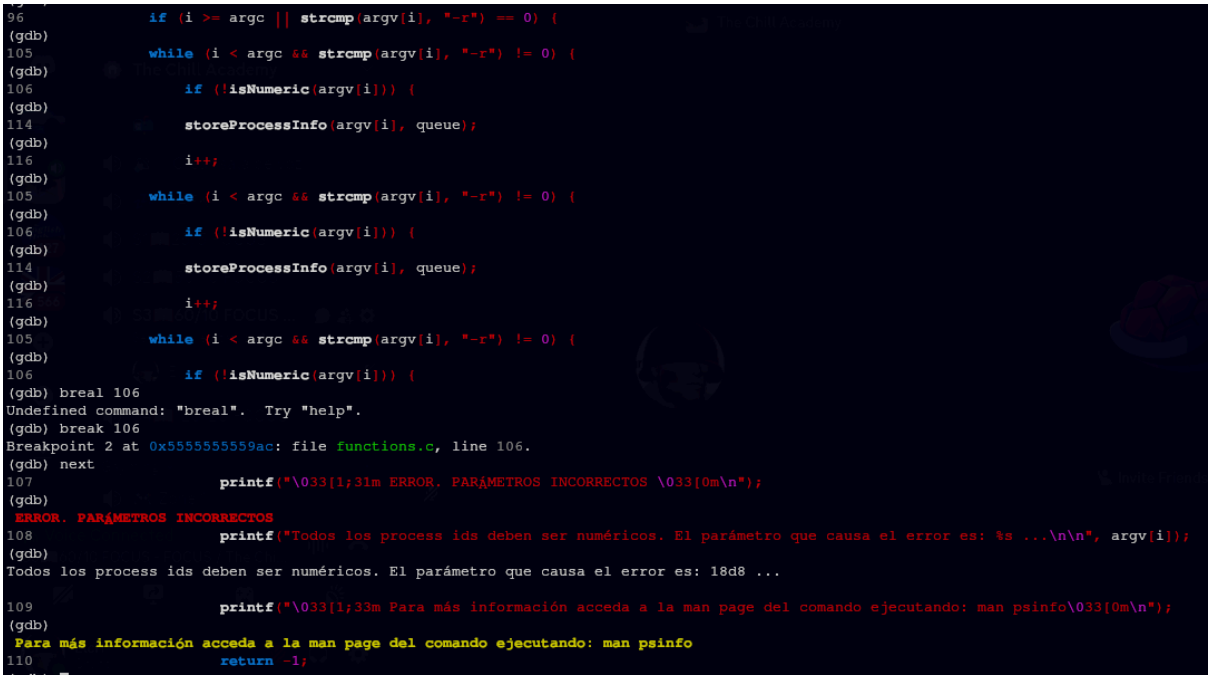
En la documentación en las man pages de strcat, dice

strcat()
This function catenates the string pointed to by src, after the string pointed to by dst **(overwriting its terminating null byte).** The programmer is responsible for allocating a destination buffer large enough, that is, strlen(dst) + strlen(src) + 1.

“(overwriting its terminating null byte).” Esta parte es importante y es lo que le da el sentido a la función. Empieza desde el caracter nulo del string apuntado a escribir el otro string. Cosa que en la documentación de snprintf no aparece. Para solucionar esto, tocó implementar la suma del string que se tiene en este momento para que escribiera desde la última posición y concatenara los datos.

PRUEBAS Y DEBUGGING

- Función isNumeric()
  Enviemos un process id con un caracter que no sea un dígito
  ./psinfo -l 1790 1 18d8



- Función firstProcessPosition()
  psinfo -l 1 -r

```
[cristian@my-arch version1]$ gdb --args psinfo -l 1790 -r
GNU gdb (GDB) 16.2
Copyright (C) 2024 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-pc-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from psinfo...
(gdb) break 17
Breakpoint 1 at 0x2196: file main.c, line 17.
(gdb) run
Starting program: /home/cristian/Documents/udea/semestre-8/sistemas-operativos/labs-so/lab01-psinfo/version1/psinfo -l 1790 -r

This GDB supports auto-downloading debuginfo from the following URLs:
  <https://debuginfod.archlinux.org>
Enable debuginfod for this session? (y or [n]) y
Debuginfod has been enabled.
To make this setting permanent, add 'set debuginfod enabled on' to .gdbinit.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/usr/lib/libthread_db.so.1".

Breakpoint 1, main (argc=4, argv=0x7fffffffe468) at main.c:17
17          if (hasR) {
(gdb) next
18              int pos = firstProcessPosition(argc, argv);
(gdb) print hasR
$1 = 1
(gdb) next
19              writeReport(argc, argv, queue, pos);
(gdb) print pos
$2 = 2
(gdb)
```

psinfo 1790 -r

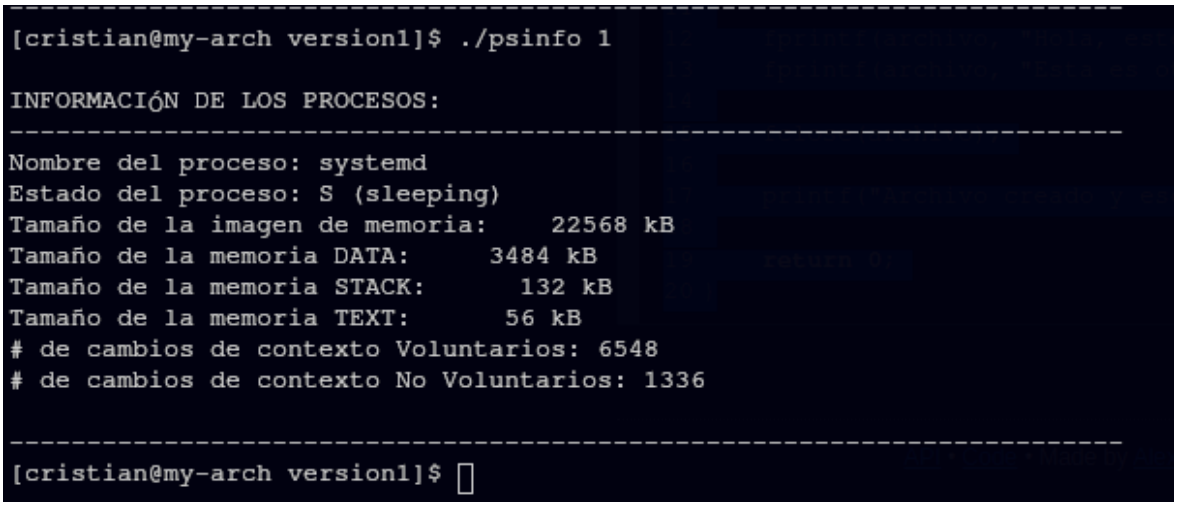
```
[cristian@my-arch version1]$ gdb --args psinfo 1790 -r
GNU gdb (GDB) 16.2
Copyright (C) 2024 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-pc-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from psinfo...
(gdb) break 17
Breakpoint 1 at 0x2196: file main.c, line 17.
(gdb) run
Starting program: /home/cristian/Documents/udea/semestre-8/sistemas-operativos/labs-so/lab01-psinfo/version1/psinfo 1790 -r

This GDB supports auto-downloading debuginfo from the following URLs:
  <https://debuginfod.archlinux.org>
Enable debuginfod for this session? (y or [n]) y
Debuginfod has been enabled.
To make this setting permanent, add 'set debuginfod enabled on' to .gdbinit.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/usr/lib/libthread_db.so.1".

Breakpoint 1, main (argc=3, argv=0x7fffffffe468) at main.c:17
17          if (hasR) {
(gdb) print hasR
$1 = 1
(gdb) next
18              int pos = firstProcessPosition(argc, argv);
(gdb) next
19              writeReport(argc, argv, queue, pos);
(gdb) print pos
$2 = 1
(gdb)
```

Podemos ver que para ambos casos responde bien el programa.

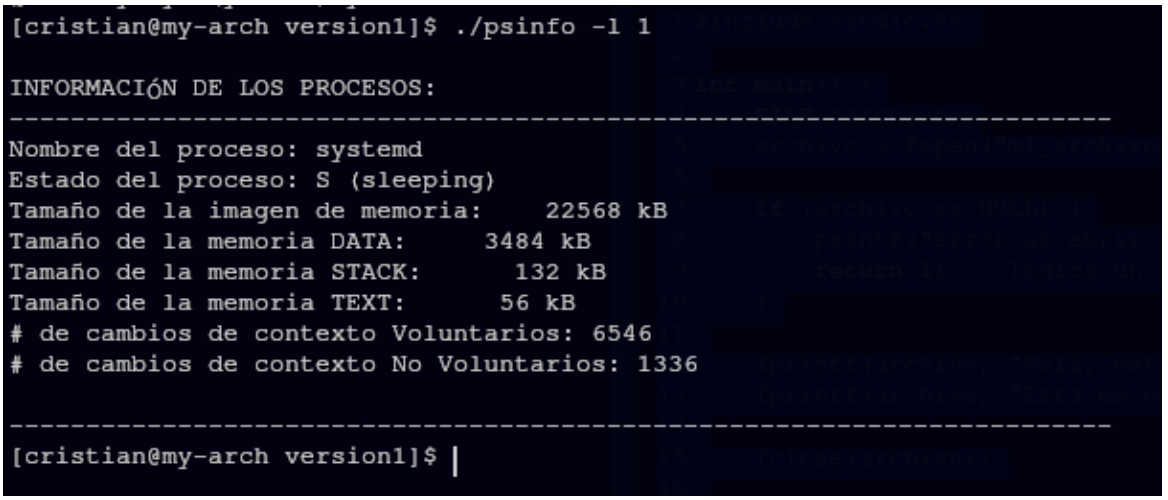
- Función processPIDs.

Para esta función, vamos a correr el programa normal enviando diferentes argumentos.

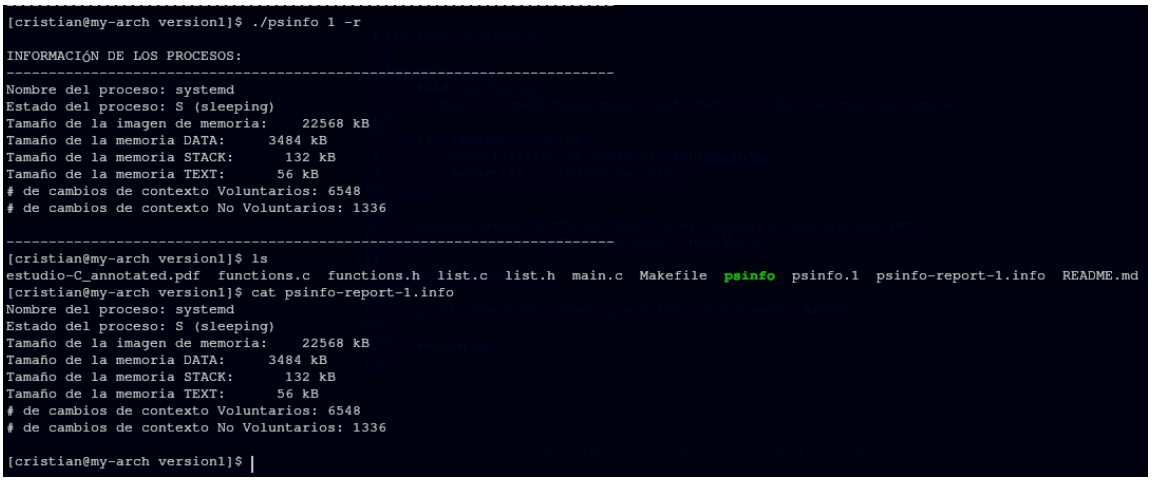
- Un solo pid



- Un solo pid agregando la bandera -l



- Un pid con -r al final



- Un pid con -r al principio

```
[cristian@my-arch version1]$ ./psinfo -r 1
 ERROR. PARÁMETROS INCORRECTOS
PID '-r' no es válido. Debe ser numérico.
 Para más información acceda a la man page del comando ejecutando: man psinfo
[cristian@my-arch version1]$
```

- Varios pids sin la bandera -l

```
[cristian@my-arch version1]$ ./psinfo 1 1790 2
 ERROR. PARÁMETROS INCORRECTOS
Si quiere pasar múltiples process id, debe poner la flag -l. Ejemplo: psinfo -l pid1 pid2 ...

 Para más información acceda a la man page del comando ejecutando: man psinfo
[cristian@my-arch version1]$
```

- Varios pids agregando -l al principio

```
[cristian@my-arch version1]$ ./psinfo -l 1 1790 2

INFORMACIÓN DE LOS PROCESOS:
--------------------------------------------------------------------------------
Nombre del proceso: systemd
Estado del proceso: S (sleeping)
Tamaño de la imagen de memoria:    22568 kB
Tamaño de la memoria DATA:     3484 kB
Tamaño de la memoria STACK:      132 kB
Tamaño de la memoria TEXT:      56 kB
# de cambios de contexto Voluntarios: 6553
# de cambios de contexto No Voluntarios: 1336

--------------------------------------------------------------------------------
Nombre del proceso: qmgr
Estado del proceso: S (sleeping)
Tamaño de la imagen de memoria:     8912 kB
Tamaño de la memoria DATA:      208 kB
Tamaño de la memoria STACK:      132 kB
Tamaño de la memoria TEXT:      28 kB
# de cambios de contexto Voluntarios: 176
# de cambios de contexto No Voluntarios: 7

--------------------------------------------------------------------------------
Nombre del proceso: kthreadd
Estado del proceso: S (sleeping)
# de cambios de contexto Voluntarios: 1081
# de cambios de contexto No Voluntarios: 100

--------------------------------------------------------------------------------
[cristian@my-arch version1]$
```

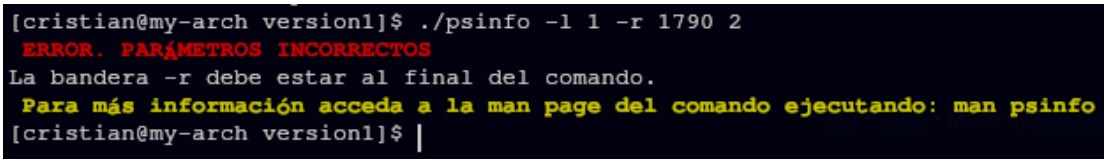
- Varios pids agregando -l en otra posición diferente al primer argumento

```
--------------------------------------------------------------------------------
[cristian@my-arch version1]$ ./psinfo 1 1790 -l 2
 ERROR. PARÁMETROS INCORRECTOS
Si quiere pasar múltiples process id, debe poner la flag -l. Ejemplo: psinfo -l pid1 pid2 ...

 Para más información acceda a la man page del comando ejecutando: man psinfo
[cristian@my-arch version1]$
```

- Varios pids con -l como primer argumento pero -r como último

```
[cristian@my-arch version1]$ ./psinfo -l 1 1790 2 -r

INFORMACIÓN DE LOS PROCESOS:
--------------------------------------------------------------------------------
Nombre del proceso: systemd
Estado del proceso: S (sleeping)
Tamaño de la imagen de memoria:    22568 kB
Tamaño de la memoria DATA:     3484 kB
Tamaño de la memoria STACK:      132 kB
Tamaño de la memoria TEXT:      56 kB
# de cambios de contexto Voluntarios: 6556
# de cambios de contexto No Voluntarios: 1336

--------------------------------------------------------------------------------
Nombre del proceso: qmgr
Estado del proceso: S (sleeping)
Tamaño de la imagen de memoria:     8912 kB
Tamaño de la memoria DATA:      208 kB
Tamaño de la memoria STACK:      132 kB
Tamaño de la memoria TEXT:      28 kB
# de cambios de contexto Voluntarios: 176
# de cambios de contexto No Voluntarios: 7

--------------------------------------------------------------------------------
Nombre del proceso: kthreadd
Estado del proceso: S (sleeping)
# de cambios de contexto Voluntarios: 1081
# de cambios de contexto No Voluntarios: 100

--------------------------------------------------------------------------------
[cristian@my-arch version1]$ ls
estudio-C_annotated.pdf  functions.c  functions.h  list.c  list.h  main.c  Makefile  psinfo  psinfo.1  psinfo-report-1-1790-2.info  psinfo-report-1.info  README.md
[cristian@my-arch version1]$
```

- Varios pids con -l como primer argumento pero -r en otra posición



CONCLUSIONES Y POSIBLES MEJORAS

- El programa funciona correctamente para obtener la información de los pids que se le otorguen siempre y cuando el proceso esté activo.
- El programa cumple con un orden rígido en la manera de escribir el comando, de pasarle los argumentos, que solo permite a -l como primer argumento y -r como último. También hace gestión correcta de los errores cuando se pasan ids inválidos.
- Se logra con éxito cada una de las 4 etapas propuestas en el laboratorio. Se consiguió un buen resultado dejando siempre una versión funcional como se propone en el laboratorio. Se pudo evidenciar la comodidad para el desarrollador el trabajar por tramos, por funciones o por espacios de código específicos, y luego unirlo todo para crear el programa.
- Se consigue un programa modular, que hace uso de múltiples archivos, como functions.c o list.c para ayudar al programa a realizar lo que se pide. Se agregan archivos h para evitar múltiples definiciones de las mismas funciones y por lo tanto, errores al momento de compilar.
- El programa puede mejorar en cuanto a la manera de validar las flags. Que no sea tan rígido.