

Función main

```
int main(){  
    ...  
}
```

→ Código

La función main es el punto de entrada de nuestro programa. Desde aquí se ejecuta todo

Información proporcionada en la clase de laboratorio del 20 de marzo

Ingresar datos por teclado e imprimir en consola

Para ingresar datos por teclado e imprimir en consola podemos usar las funciones scanf y printf, ambas incluidas en la librería stdio.

Para incluir una librería, al inicio del archivo, debemos escribir:
#include <stdio.h>

std -> Standard
io -> Input/Output
.h -> Indica que es un archivo header

Información proporcionada en la clase de laboratorio del 20 de marzo

Archivos header

Los archivos .h son similares a las interfaces en Java. Estos archivos contienen firmas de funciones, firmas de métodos y macros las cuales pueden ser incluidas en otro archivo .c para ser utilizadas. Estas deben tener claramente su implementación en un archivo del mismo nombre con extensión .c

Estos archivos permiten la reusabilidad, organización y correcta compilación de programas.

Para definir archivos header se crea un archivo con la extensión .h (ejemplo: myheader.h) y se escribe dentro de él lo siguiente:

```
#ifndef MYHEADER_H  
#define MYHEADER_H  
  
void sayHello();  
int add(int a, int b);  
double area(double radius);  
int length(char *x);  
  
#endif
```

} Si no está definido, definirlo

→ Evita ser incluido más de 1 vez

Header guard

} Firma de métodos

Luego, debemos darle una implementación a la firma de los métodos que se definieron en el archivo header.
Para eso creamos un archivo con el mismo nombre del header pero con extensión .c

myheader.c

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

#define PI 3.142


void sayHello(){
    printf("Hello World\n");
}

int add(int a, int b){
    int result;
    result = a+b;
    return result;
}

double area(double radius){
    double areaofcircle = PI*pow(radius, 2);
    return areaofcircle;
}


int length(char *x){
    return strlen(x);
}
```

Para usar los métodos y variables definidas, entonces incluimos el archivo header

```
#include <stdio.h>
#include "myheader.h"  Inclusión del header

int main() {

    int p = 10, q = 20;
    double x = 5.25;

    sayHello();  Función incluida.
    printf("sum of %d and %d is %d\n", p, q, add(p,q));
    printf("Radius: %lf Area: %lf", x, area(x));

    return 0;
}
```

Información extraída de https://www.tutorialspoint.com/cprogramming/c_header_files.htm

.....

Parámetros desde consola para un programa en C

En el método main se le pasan 2 argumentos, argc de tipo int, que representa la cantidad de parámetros que se pasen desde consola, contando el mismo script y char* argv[], un arreglo con todos los parámetros que se pasen al comando.

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

}
```

Manejo de Strings

En C no existen los strings, sino que se representan como arreglos de caracteres. Para manejo de cadenas de caracteres, en C se tiene la librería string.

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

Dicha librería permite manejar los arreglos de caracteres de manera sencilla.

Lo que se desea realizar es concatenar un string. Esto para almacenar el comando requerido en un arreglo de caracteres que después podamos usar.

La librería string.c tiene un método para concatenar strings llamado strcat.

Este método recibe 2 argumentos, de tipo arreglo de caracter.

Los une en el orden que se pasen y se deja solo el arreglo que se pasó primero.

Ejemplo:

```
char inicio_comando[50] = "cat /proc/";
```

Ahora, en los argumentos que nos pasará el usuario, también tendremos un arreglo de caracteres llamado pid (process id). Debemos concatenar estos 2 para obtener el comando cat /proc/pid. Siendo pid el process id del proceso que se quiere consultar.

```
strcat(inicio_comando, argv[1]);
```

Después de esto, sabemos que buscamos el archivo status dentro de la carpeta del pid. Podemos entonces concatenar este comando que ya tenemos con "/status"

```
strcat(inicio_comando, "/status");
```

Imprimamos el comando para verificar qué arreglo de caracteres tenemos hasta el momento

```
printf("El comando a ejecutar es %s", inicio_comando)
```

Suponiendo que el programa recibe solamente un argumento, entonces el programa se vería así hasta el momento:

```
1 #include <stdio.h>
2 #include <string.h>
3
4 int main(int argc, char* argv[]){
5     char inicio_comando[50] = "cat /proc/";
6     strcat(inicio_comando, argv[1]);
7     strcat(inicio_comando, "/status");
8     printf("%s\n", inicio_comando);
9 }
```

Ahora, necesitamos ejecutar este comando desde C.

Para eso existe la función system(). Esta pertenece a la librería stdlib y simplemente se le pasa el comando a ejecutar.

Por lo tanto, solo debemos pasarle inicio_comando, y este ejecuta el comando en la terminal.

Ya el comando imprime la información del proceso que se pasó como parámetro en la consola, pero no lo guarda realmente. Para guardarlo, debemos entonces usar una función que nos permita interactuar con la salida del comando específico. Esta función se llama popen(), pertenece a stdlib.

Esto se crea vía pipe, el cual va a presentar línea por línea la salida del comando que ejecutemos con popen. Una línea tiene máximo 128 bytes. Debemos tener entonces una variable auxiliar que permita almacenar línea por línea lo que nos pase el pipe.

Para guardar la salida del comando, debemos abrir un tipo archivo al cual le pasamos lo que retorna popen()

```
// Declaramos un buffer para leer cada línea del archivo que retorne la función popen. 128 porque es lo máximo que ocupa una línea.
char buffer[128];

//Abrimos un flujo de archivo para abrir el pipe en modo lectura en el cual guardaremos la información de la salida del comando.
FILE *fp = popen(inicio_comando, "r");

//Si fp es nulo, osea que no se pudo ejecutar el comando o que el comando no retornó nada
if (fp == NULL) {
    perror("Comando popen falló, intente de nuevo");
    // Retorna 1 para salir
    return 1;
}

// 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) {
    printf("%s", buffer);
}

// Se cierra el pipe para ahorrar recursos.
pclose(fp);
```

Información obtenida de <https://www.baeldung.com/linux/c-code-execute-system-command>

Estudio de la función fgets https://www.tutorialspoint.com/c_standard_library/c_function_fgets.htm

Estudio de popen <https://pubs.opengroup.org/onlinepubs/009696799/functions/popen.html>

Para filtrar solo la información pedida, entonces hacemos condicionales en el while. Estos condicionales deben coincidir con las primeras letras de cada línea, Por ejemplo Sabemos que en el archivo status está el nombre del proceso en una línea que empieza con Name:

Esto nos permite buscar estas similitudes e imprimir solo dichas líneas.

¿Como encontramos estas similitudes?. Podemos buscar similitudes entre strings con la función strncmp de la librería string. A esta función se le pasan 3 parámetros: 2 strings a comparar y el número n de caracteres a comparar, se comparan los primeros n caracteres

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

La salida del comando retorna algo así

```
[cristian@my-arch lab01-psinfo]$ ./a.out 1
Nombre del proceso: systemd
Estado del proceso: S (sleeping)
Tamaño de la imagen de memoria: 22096 kB
Tamaño de la memoria DATA: 3456 kB
Tamaño de la memoria STACK: 132 kB
Tamaño de la memoria TEXT: 56 kB
# de cambios de contexto voluntarios: 5072
# de cambios de contexto no voluntarios: 1169
```

Etapa 2

Ahora, se realizará una validación para incluir el argumento `-l`. Este argumento permitirá entonces agregar varios pids para listar la información de cada uno.

Para listar la información de un proceso en específico, se tomará el código que ya se tiene para la recolección de información del proceso e impresión de la misma para crear una función.

Una función en C tiene la misma estructura que la función `main`, y recibirá entonces el argumento de un pid con el cual consultará la información y la imprimirá

Ahora, antes de imprimir debemos guardar en una estructura de memoria la información de cada uno de los pids que se le ingresen. Esto permitirá tomar un snapshot del estado actual del proceso. La estructura de memoria que se usará será una lista ligada, la cual se manejará como una cola.

Se usó la implementación de `geeks for geeks` simplificándola solo con los métodos necesarios para manejarlo como una cola.

`list.h`

```
// 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
```

Diagrama de la estructura del nodo:

```
graph LR
    Node[struct Node] --> data[Char para almacenar los datos]
    Node --> next[puntero a char  
puntero a next  
otro nodo.]
```

Métodos para implementar.

Por lo tanto, debemos ver las structs en c

En C, las estructuras son tipos de datos definidos por el usuario que pueden ser usados como un grupo de items probablemente de diferente tipo dentro de un solo tipo de dato. Cada item (variable) se llama un miembro de la estructura.

```
struct A {
    int x;
};

// Function to increment values
void increment(struct A a, struct A* b) {
    a.x++;
    b->x++;
}
```

https://www.geeksforgeeks.org/generic-linked-list-in-c-2/?ref=ml_lbp

Implementación en de los métodos;

Retorna estructura nodo

```
struct Node* createNode(const char* data) {  
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));  
    // Copia dinámica de la cadena  
    newNode->data = strdup(data);  
    newNode->next = NULL;  
    return newNode;  
}
```

Puntero

Duplica Strings

¿Diferencia con *data?

duda

Crea un espacio de memoria en el heap del tamaño de una estructura node. Retorna un puntero a este espacio. Se castea a un puntero Node*.

Doble apuntador → diferencia con solo poner head.

```
void insertAtEnd(struct Node** head, const char* data) {  
    struct Node* newNode = createNode(data);  
    // Si no hay head, osea que la cola esté vacía, entonces head ahora debe apuntar a newNode  
    if (*head == NULL) {  
        *head = newNode;  
        return;  
    }  
    // Si la lista no está vacía, ejecuta lo siguiente  
    // Crea un nodo temporal que apunta a head  
    struct Node* temp = *head;  
    while (temp->next != NULL) {  
        temp = temp->next;  
    }  
    temp->next = newNode;  
}
```

Avanzamos hasta el último

Al último le ponemos de siguiente el nuevo nodo.

```
void deleteFromFirst(struct Node** head) {  
    // Si la lista está vacía que lo imprima  
    if (*head == NULL) {  
        printf("La lista enlazada está vacía\n");  
        return;  
    }  
    // Se crea un nodo temporal que apunta a head  
    struct Node* temp = *head;  
    *head = temp->next;  
    // Se libera memoria  
    free(temp);  
}
```

Borra temp

```

void print(struct Node* head) {
    // Se crea un nuevo nodo apuntando a head
    struct Node* temp = head;
    // Mientras este nodo que apuntaba a head sea diferente de nulo
    while (temp != NULL) {
        printf("%s -> ", temp->data);
        temp = temp->next;
    }
    //Imprimimos NULL Para saber que llegamos al final de la lista.
    printf("NULL\n");
}

```

Se incluye el header en el archivo main

La estrategia para guardar toda la info necesaria de un proceso en la lista ligada será la siguiente

Recoger toda la info de cada proceso en un string. (Se usará un buffer de 2048)

Este buffer al final de cada iteración vaciará toda la info que recoja en el miembro data de un nuevo nodo el cual se insertará en la lista ligada.

```

// 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) {
        strcat(bufferInfo, "Nombre del proceso: ");
        strcat(bufferInfo, buffer + 6);
    } else if (strncmp("State:", buffer, 6) == 0) {
        strcat(bufferInfo, "Estado del proceso: ");
        strcat(bufferInfo, 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);
    }
}

insertAtEnd(queue, bufferInfo);
pclose(fp);
printf("-----\n");
return 0;

```

Ahora que estamos almacenando la información, necesitamos una función para imprimir los elementos de una cola.

```

void printProcessList(struct Node* head) {
    //creamos un nodo temporal que apunte a head
    struct Node* temp = head;
    printf("\nINFORMACIÓN DE LOS PROCESOS: \n");

    //mientras no sea null entonces vamos a imprimir la data del nodo temp
    while (temp != NULL) {
        printf("%s\n", temp->data);
        //Ahora temp debe apuntar al siguiente
        temp = temp->next;
    }
}

```

Para concluir la etapa 2, tenemos entonces que verificar que cuando se agregue el parámetro `-l` después de llamar el comando, se imprima la información de varios pids

Creamos la función `processPIDs(int argc, char* argv[], struct Node** queue)`

1. Verificar que ingrese siempre al menos un pid

```
if (argc < 2) {
    printf("Por favor, ingresa al menos un PID.\n");
    return -1;
}
```

2. Verificar que si el segundo parámetro es `-l` y además el número de argumentos es 2 lance un error, ya que esta situación sería algo como `psinfo -l`.

```
if (strcmp(argv[1], "-l") == 0) {
    // Si el nro de argumentos es igual a 2, significa que después de -l no ingresó n
    if (argc == 2) {
        printf("Por favor, ingresa al menos un PID después de -l.\n");
        return -1;
    }
}
```

Si esto no ocurre, osea, que si hay uno o más pids que guarde la información de estos en la lista ligada que se pasó como parámetro.

```
for (int i = 2; i < argc; i++) {
    //Guardamos la información del proceso en la cola queue
    storeProcessInfo(argv[i], queue);
}
```

3. Si no se cumple que el segundo parámetro es `-l` entonces imprime la información del proceso que se pasó y sale.

```
else {
    // Si no se pasa -l pero hay varios pids, mostramos un mensaje
    if (argc > 2) {
        printf("Si desea imprimir la información de varios PIDs, agrega el parámetro\n");
        printf("Ejemplo: psinfo -l 123 456\n");
        return -1;
    }

    // Si no se pasa -l procesamos un solo pid
    storeProcessInfo(argv[1], queue);
}
```

Desde main ahora llamamos este método y luego el de imprimir

```
//inicializamos la cola
struct Node* queue = NULL;

// Procesar los argumentos y almacenar los pids en la cola
if (processPIDs(argc, argv, &queue) != 0) {
    // Si hay algún error al procesar salimos
    return -1;
}

// Imprimir toda la cola después de almacenar la información
printf("\nTodos los procesos almacenados en la cola:\n");
// Imprimimos la lista enlazada
printProcessList(queue);

return 0;
```