

TRABAJO 2 (IC2)

Ayoze Ruano Alcántara, Enrique Reina Hernández y Fabio Nesta Arteaga Cabrera

Vamos a empezar a desarrollar el trabajo según los requisitos iniciales (en un principio), publicados en el campus:

Nota: GrupoRAR significa Grupo Reina, Arteaga, Ruano (Nuestros apellidos).

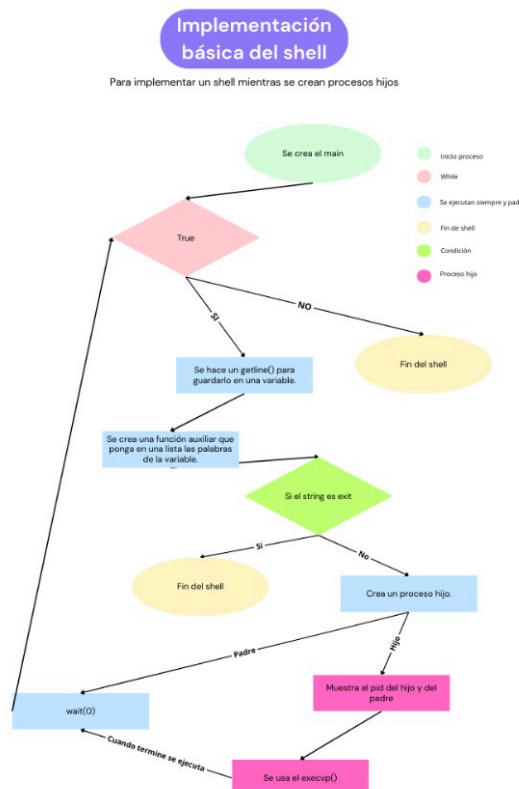
6.1 Implementación básica del shell

Para implementar este *shell*, cada vez que el usuario escriba una línea, tendrás que crear un proceso hijo que ejecute la orden deseada y con los argumentos que haya escrito el usuario. Esto lo harás combinando las llamadas al sistema *fork()* y *exec()*. A su vez, el propio *shell* tendrá que esperar a la terminación de la orden con la operación *wait()*. Tras ello el *shell* volverá a leer una nueva línea escrita por el usuario. (Nota: esta es la técnica que realmente utilizan los shells de Unix para ejecutar las órdenes).

Además, cada vez que cree un proceso hijo, nos imprimirá por pantalla el pid del proceso hijo y del padre, antes del usar *exec()*.

Si el usuario escribe la palabra «exit», el *shell* terminará (reconocerá esa palabra como una orden especial para finalizar).

IMPORTANTE: para lanzar el proceso hijo, **NO** se puede utilizar la función *system()*. Hay que lanzarlo necesariamente con alguna variante de *exec()*. La función *system()* es una operación de alto nivel que justamente realiza lo que estamos pidiendo que tú implementes por ti misma/o.



(DIAGRAMA DE FLUJO INICIAL) (Arriba)

Primero buscaremos cómo utilizar el `getline()` (función mencionada en clase), para más adelante implementar una función llamada **splitter**, que separe las instrucciones dadas en la string resultante de `getline()` en un vector. Esto es necesario ya que la función `execvp()` requiere un vector para funcionar.

<https://www.youtube.com/watch?v=Lx5r2XQvpVA>

Este vídeo lo vimos para tener una noción básica del comportamiento de `getline()`.

Creamos estas instrucciones en el `main()` para ver si el `getline()` está bien definido:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>

void splitter(char str[]){

}

void main(){
    pid_t pid;

    while (1){
        char *line;
        size_t len = 20;
        ssize_t read;
        printf("Escriba una función: ");
        getline(&line, &len, stdin);
        printf("Escribiste: %s", line);
    }
}
```

```
[estudiante_ic2@ic2-centos-vm trabajo2]$ gcc trabajo2.c -o p
[estudiante_ic2@ic2-centos-vm trabajo2]$ ./p
Escriba una función: Hola
Escribiste: Hola
Escriba una función: Hola buenos días
Escribiste: Hola buenos días
Escriba una función: ^C
[estudiante_ic2@ic2-centos-vm trabajo2]$
```

Como se puede observar se escribe en terminal la frase y se devuelve en el print siguiente.

Como hay un `while` solo de momento, se va a estar repitiendo todo el rato.

También en C no existe `True` ni `False` lo que hay son `1` y `0` (respectivamente) por eso cambié el `True` por un `1`.

Ahora formamos la función `splitter()` la cual se va a encargar de dividir el `getline` en un vector.

```

char splitter(char *line){
    char *palabra = NULL;
    int index = 0; //Para palabras
    char *result;
    int i; //Para caracteres
    for (i=0; line[i]; i++){
        if(line[i] != " "){
            palabra = line[i];
        }
        else if(line[i]==" "){
            line[i] = "\\0";
            result[index++] = palabra;
            palabra = NULL;
        }
    }
    return result;
}

```

Ayoze se fijó en que la función estaba declarada (antes) como tipo void por lo que no devolvía nada.

Cambié el `size_t = 20` por un `size_t = 0` porque así se completa el tamaño del `getline()`.

Luego creamos el proceso hijo para hacer que devuelva el pid del padre y del hijo para que seguido a eso utilicemos el `execvp()` para poder realizar la instrucción.

```

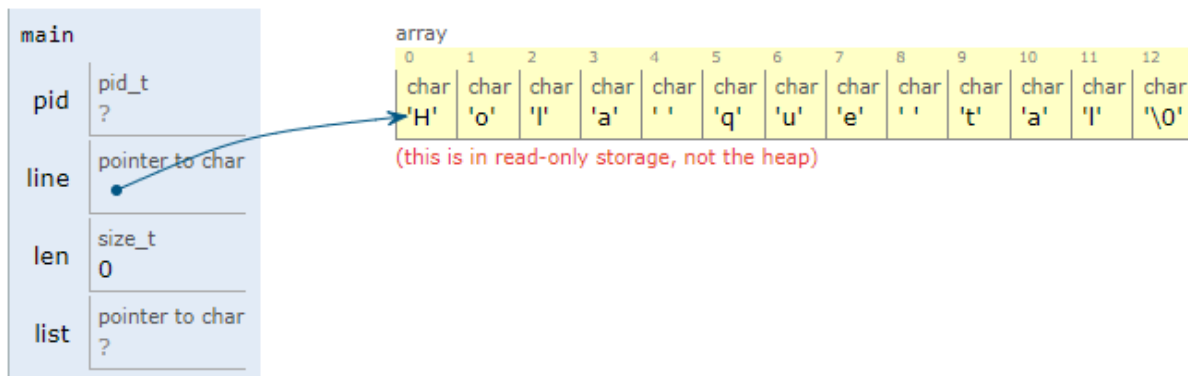
char main(){
    pid_t pid;

    while (1){
        char *line;
        size_t len = 0;
        printf("Escriba una función: ");
        getline(&line, &len, stdin);
        printf("Escribiste: %s", line);
        char *list;
        list = splitter(line);
        if (line == "exit"){
            break;
        }
        else{
            pid = fork();
            if (pid == 0){
                printf("Pid del proceso padre: %d, pid del proceso hijo: %d\n", getppid(), getpid());
                execvp(list[0], 0);
            }
            else{
                wait(0);
            }
        }
    }
}

```

```
[estudiante_ic2@ic2-centos-vm trabajo2]$ ./p
Escriba una función: ls - ln
Escribiste: ls - ln
Segmentation fault (`core' generado)
[estudiante_ic2@ic2-centos-vm trabajo2]$
```

Después de ejecutar el código, el `execvp` nos devuelve un “Segmentation fault(‘core’ generado)”. Por lo que decidimos preguntar a python tutor para darnos cuenta que en `list` no se termina de guardar nada:



Es una foto de python tutor cuando termina por completo la función `splitter`. Por eso vamos a cambiar el funcionamiento de dicha función.

Gracias a este video:

<https://www.youtube.com/watch?v=jHdbmHCVQ2c>

Este es el resultado:

```
char splitter(char *line){
    char *token;
    char *result;
    token = strtok(&line, " ");
    if (token == NULL){
        printf("Se ha producido un error");
        exit(-1);
    }
    while(token != NULL){
        token = strtok(NULL, " ");
        if(token != NULL){
            result += token;
        }
    }
    return result;
}
```

```
[estudiante_ic2@ic2-centos-vm trabajo2]$ ./p
Escriba una función: ls -ln
Escribiste: ls -ln
Pid del proceso padre: 2973, pid del proceso hijo: 2974
Escriba una función: █
```

Luego comprobamos si el exit funcionaba pero nos dimos cuenta que no hacía la condición:

```
[estudiante_ic2@ic2-centos-vm trabajo2]$ ./p
Escriba una función: exit
Escribiste: exit
Pid del proceso padre: 3332, pid del proceso hijo: 3333
Escriba una función: exit
Escribiste: exit
Pid del proceso padre: 3332, pid del proceso hijo: 3343
```

Para ello en el condicional utilizamos el código strcmp para comparar las dos strings y si el resultado es 0 es que son iguales, por eso:

```
-----
list = splitter(line);
if (strcmp(line, "exit\n")==0){
    break;
}
```

```
[estudiante_ic2@ic2-centos-vm trabajo2]$ ./p
Escriba una función: exit
Escribiste: exit
[estudiante_ic2@ic2-centos-vm trabajo2]$ █
```

Esta es una versión temprana del código desarrollada por Enrique y Fabio. Si bien es temprana, ya tiene un buen cuerpo y un concepto sólido. A partir de aquí, Ayoze (junto con ellos dos), modificaría el cuerpo del código agregando algunos retoques a las funciones, así como pulir el output para una mejor lectura.

```
// ESTA ES LA VERSION TEMPRANA DEL SPLITTER DE ENRIQUE Y FABIO
/*
char *splitter(char *line){
    char *token[50]; //un valor grande por si hay alguna función larga
    int i = 0;
    token[i] = strtok(line, " ");
    if (token == NULL){
        printf("Se ha producido un error");
        exit(-1);
    }
    while(token[i] != NULL){
        i++;
        token[i] = strtok(NULL, " ");
    }
    return token;
}
*/
```

En esta versión lo que se planteaba era que token sea un puntero a vector, para que sea más fácil de controlar ya que podemos usar los índices del vector. Pero el problema es que de esa forma daba un Segmentation fault igualmente y también que en vez de colocar la primera palabra, lo que hace es devolver toda la frase.

```
[estudiante_ic2@ic2-centos-vm trabajo2]$ ./p
Escriba una función: Hola buenos dias
Escribiste: 'Hola buenos dias
'

Hola buenos dias
buenos
dias

Segmentation fault (`core' generado)
[estudiante_ic2@ic2-centos-vm trabajo2]$
```

Algo que hicimos para probar fue ponerlo dentro del main para ver si así mejoraba el código.

```

int i = 0;
token[i] = strtok(buffer, s);
if (token == NULL){
    printf("Se ha producido un error");
    exit(-1);
}
while(token[i] != NULL){
    i++;
    token[i] = strtok(NULL, " ");
    printf("%s\n", token[i]);
}

int tam = sizeof(token) / sizeof(char*);
for (i=0;i<tam;i++){
    printf("%s, ", token[i]);
}

if (strcmp(result[0], "exit")==0){
    break;
}
else{
    pid = fork();
    if (pid == 0){

```

```

[estudiante_ic2@ic2-centos-vm trabajo2]$ ./p
Escribe órdenes: Hola buenos días
17 caracteres han sido leídos.
Instrucciones a ejecutar: Hola buenos días
buenos
días

```

El problema que hubo es que no imprimía la primera palabra. Decidimos empezar un nuevo splitter de cero por si acaso rehaciendo el código nos damos cuenta de algo que falta.

Nos apoyamos en estos recursos para entender mejor el getline y repasar conceptos:

<https://c-for-dummies.com/blog/?p=1112>

http://maxus.fis.usal.es/FICHAS_C.WEB/07xx_PAGS/0702.html

```

#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <string.h>

void main(){
    pid_t pid;
    char *buffer;
    size_t bufsize = 32;
    size_t characters;

    while (1){

        buffer = (char *)malloc(bufsize * sizeof(char));

        printf("Escribe órdenes: ");
        characters = getline(&buffer,&bufsize,&stdin);
        printf("%zu caracteres han sido leídos.\n",characters);
        //printf("You typed: '%s'\n",buffer);

        printf("Instrucciones a ejecutar: ");
        printf("%s", buffer);
    }
}

```

En esta parte inicial del código, declaramos un pid, así como las variables que le van a hacer falta al getline.

Vamos a ejecutar el código por separado para ver su funcionamiento.

```

[estudiante_ic2@ic2-centos-vm trabajo2]$ gcc prueba.c -o test
[estudiante_ic2@ic2-centos-vm trabajo2]$ ./test
Escribe órdenes: ls -l
6 caracteres han sido leídos.
Instrucciones a ejecutar: ls -l
[estudiante_ic2@ic2-centos-vm trabajo2]$ █

```

Básicamente le introducimos por teclado una serie de caracteres, y tras *Instrucciones a ejecutar* se printea la **string resultado**.


```

////////////////////////////////////////////////////
/*
int i; // para recorrer la cadena
int counter = 0; // para navegar el vector aux
int ndef = 0; // para navegar el vector result

char aux = (char)malloc(20*sizeof(char)); // crea un vector con espacio para (20) chars
char *result[10];

for (i=0;i<characters;i++){
    if (buffer[i] != ' ' || buffer[i] != "\n"){
        *(aux+counter) = buffer[i];
        counter++;
    }
    else{
        result[ndef] = aux;

        counter = 0;
        free(aux);
        aux = (char*)malloc(20*sizeof(char)); // crea un vector con espacio para (counter) chars
        ndef++;
    }
}

int tam = sizeof(result) / sizeof(char*);
for (i=0;i<tam;i++){
    printf("%s\n", result[i]);
}

// el problema que nos daba esta version es que al hacer free, nos cargabamos el puntero.

*/

```

Esto es un intento de Ayoze (algo enrevesado) de un splitter, pero no funcionaba porque básicamente al utilizar memoria dinámica y trabajar con punteros, al hacer free para liberar aux, se eliminaba el contenido al que estaba apuntando result. Entonces, luego al printear, no salía nada. Se trata de una versión descartada.

Posteriormente, Enrique nos habló de ciertas funciones de la librería string.h, y en conjunto desarrollamos este splitter.

Nos basamos en este ejemplo:

https://www.tutorialspoint.com/c_standard_library/c_function_strtok.htm

También nos apoyamos en este vídeo:

https://www.youtube.com/watch?v=sEx8EdTO8R8&ab_channel=DuarteCorporationTutorial
[es](#)

```

// segmento splitter
char *result[10];

const char s[3] = " \n";
char *token;

/* primer elemento */
token = strtok(buffer, s);

/* recopila los elementos restantes */
int i = 0;
int a = 0;
while( token != NULL ) {
    result[a] = token;
    a++;

    token = strtok(NULL, s);
}

result[a] = NULL;
//a++;

```

Aquí estamos creando un puntero a un vector de chars, que sirve para almacenar varias strings como si fuese un array (explicado en el vídeo).

```

// ESTE ES UN SEGMENTO PARA LEER EL VECTOR (HECHO POR AYOZE)
// int tam = sizeof(hola) / sizeof(char*);
printf("-----\n");
printf("Contenido del vector resultado: ");
for (i=0;i<a;i++){
    if (i==a-1){
        printf("(%s)\n", result[i]);
    }
    else{
        printf("(%s),", result[i]);
    }
}

```

Esta porción de código sirve para comprobar que el vector está almacenando correctamente los datos de la string resultado.

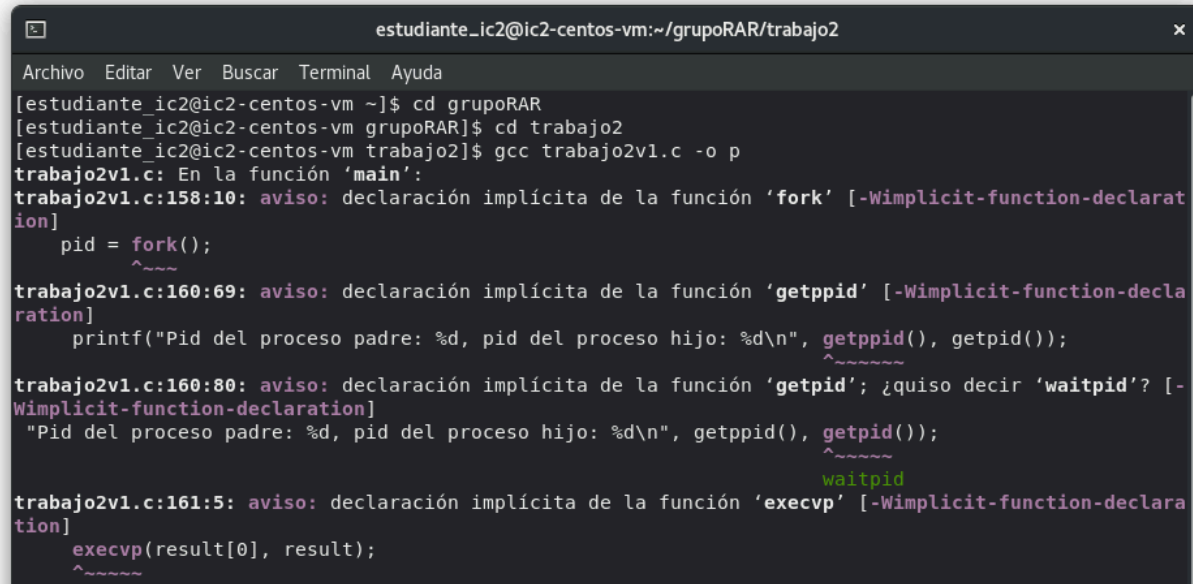
```

    if (strcmp(result[0], "exit")==0){
        break;
    }
    else{
        pid = fork();
        if (pid == 0){
            printf("Pid del proceso padre: %d, pid del proceso hijo: %d\n", getppid(), getpid());
            execvp(result[0], result);
        }
        else{
            wait(0);
        }
    }
}

```

Aquí va a examinar el primer elemento del vector resultado. En caso de que sea exit, se corta el while y acaba el programa. En cualquier otro caso, se crea un proceso hijo; se imprime el PID del padre y del hijo; y se le pide al hijo que ejecute la orden contenida en el vector resultado (mediante `execvp()`).

Vamos a comprobar el output.



```

trabajo2v1.c
~/grupoRAR/trabajo2

estudiante_ic2@ic2-centos-vm: ~/grupoRAR/trabajo2
Archivo Editar Ver Buscar Terminal Ayuda
[estudiante_ic2@ic2-centos-vm ~]$ cd grupoRAR
[estudiante_ic2@ic2-centos-vm grupoRAR]$ cd trabajo2
[estudiante_ic2@ic2-centos-vm trabajo2]$ gcc trabajo2v1.c -o p
trabajo2v1.c: En la función 'main':
trabajo2v1.c:158:10: aviso: declaración implícita de la función 'fork' [-Wimplicit-function-declaration]
    pid = fork();
           ^~~~
trabajo2v1.c:160:69: aviso: declaración implícita de la función 'getppid' [-Wimplicit-function-declaration]
    printf("Pid del proceso padre: %d, pid del proceso hijo: %d\n", getppid(), getpid());
                                                                    ^~~~~~
trabajo2v1.c:160:80: aviso: declaración implícita de la función 'getpid'; ¿quiso decir 'waitpid'? [-Wimplicit-function-declaration]
    "Pid del proceso padre: %d, pid del proceso hijo: %d\n", getppid(), getpid());
                                                                    ^~~~~~
                                                                    waitpid
trabajo2v1.c:161:5: aviso: declaración implícita de la función 'execvp' [-Wimplicit-function-declaration]
    execvp(result[0], result);
    ^~~~~~

```

```
[estudiante_ic2@ic2-centos-vm trabajo2]$ ./p
Escribe órdenes: ls -l
6 caracteres han sido leídos.
Instrucciones a ejecutar: ls -l
Contenido del vector resultado: (ls),(-l)
Pid del proceso padre: 3092, pid del proceso hijo: 3093
total 56
-rwxrwxr-x. 1 estudiante_ic2 estudiante_ic2 13256 abr 20 16:03 p
-rw-r--r--. 1 estudiante_ic2 estudiante_ic2 485 abr 20 14:20 prueba.c
-rwxrwxr-x. 1 estudiante_ic2 estudiante_ic2 13296 abr 19 19:41 result
-rwxrwxr-x. 1 estudiante_ic2 estudiante_ic2 12896 abr 20 14:27 test
-rwxrwx---. 1 estudiante_ic2 estudiante_ic2 3290 abr 20 15:54 trabajo2v1.c
Escribe órdenes: ls -l
6 caracteres han sido leídos.
Instrucciones a ejecutar: ls -l
Contenido del vector resultado: (ls),(-l)
Pid del proceso padre: 3092, pid del proceso hijo: 3094
total 56
-rwxrwxr-x. 1 estudiante_ic2 estudiante_ic2 13256 abr 20 16:03 p
-rw-r--r--. 1 estudiante_ic2 estudiante_ic2 485 abr 20 14:20 prueba.c
-rwxrwxr-x. 1 estudiante_ic2 estudiante_ic2 13296 abr 19 19:41 result
-rwxrwxr-x. 1 estudiante_ic2 estudiante_ic2 12896 abr 20 14:27 test
-rwxrwx---. 1 estudiante_ic2 estudiante_ic2 3290 abr 20 15:54 trabajo2v1.c
```

Este output parece prometedor, pero vamos a intentar mejorar la lectura del output, que ahora mismo parece algo confusa.

```
buffer = (char *)malloc(bufsize * sizeof(char));

printf("-----\n");
printf("Escribe órdenes: ");
characters = getline(&buffer,&bufsize,stdin);
printf("%zu caracteres han sido leídos.\n",characters);
//printf("You typed: '%s'\n",buffer);

//printf("Instrucciones a ejecutar: ");
//printf("%s", buffer);

// ESTE ES UN SEGMENTO PARA LEER EL VECTOR (HECHO POR AYOZE)
// int tam = sizeof(hola) / sizeof(char*);
printf("-----\n");
printf("Contenido del vector resultado: ");
for (i=0;i<a;i++){
    if (i==a-1){
        printf("(%s)\n", result[i]);
    }
    else{
        printf("(%s),", result[i]);
    }
}
```

```

if (strcmp(result[0], "exit")==0){
    printf("-----\n");
    printf("Ejecución terminada. (Se ha introducido exit)\n");
    break;
}
else{
    pid = fork();
    if (pid == 0){
        printf("-----\n");
        printf("Pid del proceso padre: %d, pid del proceso hijo: %d\n", getppid(), getpid());
        printf("-----\n");
        execvp(result[0], result);
    }
    else{
        wait(0);
    }
}
}

```

```

[estudiante_ic2@ic2-centos-vm trabajo2]$ ./result
-----
Escribe órdenes: ps
3 caracteres han sido leídos.
-----
Contenido del vector resultado: (ps)
-----
Pid del proceso padre: 2976, pid del proceso hijo: 2977
-----
  PID TTY          TIME CMD
 2875 pts/0    00:00:00 bash
 2976 pts/0    00:00:00 result
 2977 pts/0    00:00:00 ps
-----
Escribe órdenes: ls -l
6 caracteres han sido leídos.
-----
Contenido del vector resultado: (ls),(-l)
-----
Pid del proceso padre: 2976, pid del proceso hijo: 2978
-----
total 40
-rwxrwxr-x. 1 estudiante_ic2 estudiante_ic2    0 abr 20 16:30 p
-rw-r--r--. 1 estudiante_ic2 estudiante_ic2  485 abr 20 14:20 prueba.c
-rwxrwxr-x. 1 estudiante_ic2 estudiante_ic2 13312 abr 20 20:11 result
-rwxrwxr-x. 1 estudiante_ic2 estudiante_ic2 12896 abr 20 14:27 test
-rwxrwx---. 1 estudiante_ic2 estudiante_ic2  3677 abr 20 16:30 trabajo2v1.c
-----
Escribe órdenes: exit
5 caracteres han sido leídos.
-----
Contenido del vector resultado: (exit)
-----
Ejecución terminada. (Se ha introducido exit)
[estudiante_ic2@ic2-centos-vm trabajo2]$

```

Con esto, el shell inicial ya estaría terminado.

6.2 Requisitos opcionales

Calcular el tiempo que tarda en ejecutarse el proceso hijo:

Para esto usamos la función times y getrusage.

Después de informarme en internet, vi que getrusage no era necesario, ya que se usa para obtener la cantidad de recursos que usa, no el tiempo.

Usando times, conseguí el tiempo que tarda la CPU en el usuario con tms_cutime y en el sistema con tms_cstime, y los sumé.

Le añadí un struct para tener el time_start y el time_end del usuario y del sistema, y al restar al final con el inicio, conseguir el tiempo de ejecución.

```
char main(){
    pid_t pid;
    char *buffer;
    size_t bufsize = 32;
    size_t characters;
    struct tms time_start, time_end;

    while (1){
```

El start lo puse antes del fork(), y el end, debajo del wait(0), para asegurarnos de tener el inicio y el final del proceso hijo:

```
    else{
        times(&time_start);
        pid = fork();
        if (pid == 0){
            printf("Pid del proceso padre: %d, pid del proceso hijo: %d\n",
id());
            execvp(result[0], result);
        }
        else{
            wait(0);
            times(&time_end);
```

Y aquí saqué el tiempo total:

```
double tiempo_usuario = time_end.tms_cutime - time_start.tms_cutime;
double tiempo_sistema = time_end.tms_cstime - time_start.tms_cstime;
double tiempo_total = tiempo_usuario + tiempo_sistema;

    printf("Tiempo de uso de la CPU del proceso hijo: %f", tiempo_total);
}
```

Al compilar me daba error:

```
[estudiante_ic2@ic2-centos-vm trabajo2]$ gcc trabajo2.c -o trabajo2
trabajo2.c: En la función 'splitter':
trabajo2.c:20:9: aviso: se devuelve 'char **' desde una función con tipo de devolución 'char *' incompatible [-Wincompatible-pointer-types]
    return token;
    ^~~~~
trabajo2.c:20:9: aviso: la función devuelve la dirección de una variable local [-Wreturn-local-addr]
trabajo2.c: En la función 'main':
trabajo2.c:172:10: aviso: declaración implícita de la función 'fork' [-Wimplicit-function-declaration]
    pid = fork();
           ^~~~
trabajo2.c:174:69: aviso: declaración implícita de la función 'getppid' [-Wimplicit-function-declaration]
    printf("Pid del proceso padre: %d, pid del proceso hijo: %d\n", getppid(), getpid());
                                                                    ^~~~~~
trabajo2.c:174:80: aviso: declaración implícita de la función 'getpid'; ¿quiso decir 'waitpid'? [-Wimplicit-function-declaration]
    printf("Pid del proceso padre: %d, pid del proceso hijo: %d\n", getppid(), getpid());
                                                                    ^~~~~~
                                                                    waitpid
trabajo2.c:175:5: aviso: declaración implícita de la función 'execvp' [-Wimplicit-function-declaration]
    execvp(result[0], result);
    ^~~~~
trabajo2.c:179:12: error: 'times_end' no se declaró aquí (primer uso en esta función); ¿quiso decir 'time_end'?
    times(&times_end);
           ^~~~~~
           time_end
trabajo2.c:179:12: nota: cada identificador sin declarar se reporta sólo una vez para cada función en el que aparece
[estudiante_ic2@ic2-centos-vm trabajo2]$
```

Y después de un rato me di cuenta de que era porque tenía "times_end" en vez de "time_end":

Al ejecutar, me dio esto:

```
[estudiante_ic2@ic2-centos-vm trabajo2]$ ./trabajo2
Escribe órdenes: ls -l
6 caracteres han sido leídos.
Instrucciones a ejecutar: ls -l
Contenido del vector resultado: (ls),(-l)
Pid del proceso padre: 3805, pid del proceso hijo: 3806
total 24
-rwxrwxr-x. 1 estudiante_ic2 estudiante_ic2 17440 abr 23 10:09 trabajo2
-rwxrwx---. 1 estudiante_ic2 estudiante_ic2 3673 abr 23 10:08 trabajo2.c
Tiempo de uso de la CPU del proceso hijo: 0.000000Escribe órdenes:
```

Y me di cuenta de que era porque los tenía en double en vez de long, quería mostrar los tiempos de una mejor manera así que cambié el tipo de la variable.

Times usa un contador global de ticks, por lo que restando, tendríamos el tiempo que dura la ejecución de un proceso en la CPU, pero me da lo mismo el inicio y el final, y después de

buscar información al no encontrar la solución, leí en varias páginas que el proceso al ser tan sencillo puede que, aunque tenga una espera, realmente se ejecute en menos de un tic, por lo que nos daría 0 al restar los ticks del final (end_clock) y los del inicio (start_clock):

```
[estudiante_ic2@ic2-centos-vm trabajo2]$ gcc trabajo2v1.c -o trabajo2v1
[estudiante_ic2@ic2-centos-vm trabajo2]$ ./trabajo2v1
-----
Escribe órdenes: ls
3 caracteres han sido leídos.
-----
Contenido del vector resultado: (ls)
Start_clock = 430067460
-----
Pid del proceso padre: 12504, pid del proceso hijo: 12505
-----
trabajo2 trabajo2parch.c trabajo2v1 trabajo2v1.c
End_clock = 430067460
Tiempo de uso de la CPU del proceso hijo: 0
-----
Escribe órdenes: █
```

```
start_clock = times(&time_start);
printf("Start_clock = %d\n", start_clock);
pid = fork();
if (pid == 0){
    printf("-----\n");
    printf("Pid del proceso padre: %d, pid del proceso hijo: %d\n", getppid(), getpid());
    printf("-----\n");
    execvp(result[0], result);
}
else{
    wait(0);
    end_clock = times(&time_end);
    printf("End_clock = %d\n", end_clock);

    long tiempo_usuario = (time_end.tms_cutime - time_start.tms_cutime);
    long tiempo_sistema = (time_end.tms_cstime - time_start.tms_cstime);
    long tiempo_total = tiempo_usuario + tiempo_sistema;

    printf("Tiempo de uso de la CPU del proceso hijo: %ld\n", tiempo_total);
}
```

Tras unos días, probamos a introducir un gedit. Este fue el output.


```
prueba.c
~/grupoRAR/trabajo2
prueba.c

estudiante_ic2@ic2-centos-vm:~/grupoRAR/trabajo2

Archivo Editar Ver Buscar Terminal Ayuda
[estudiante_ic2@ic2-centos-vm ~]$ cd grupoRAR
[estudiante_ic2@ic2-centos-vm grupoRAR]$ cd trabajo2
[estudiante_ic2@ic2-centos-vm trabajo2]$ gcc trabajo2v1.c -o result
[estudiante_ic2@ic2-centos-vm trabajo2]$ ./result
-----
Escribe órdenes: gedit prueba.c
15 caracteres han sido leídos.
-----
Contenido del vector resultado: (gedit),(prueba.c)
-----
Pid del proceso padre: 3226, pid del proceso hijo: 3227
-----
Tiempo de uso de la CPU del proceso hijo: 2
```

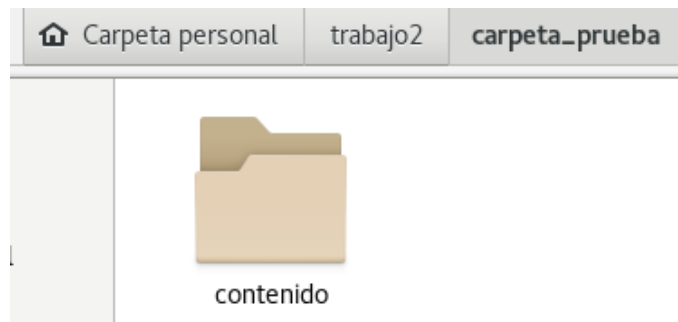
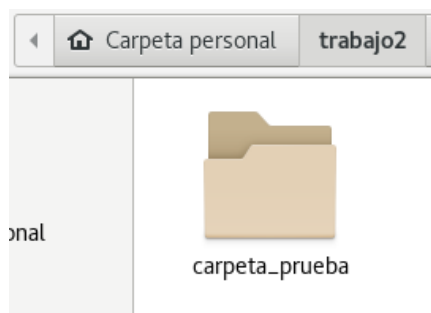
Como se ve, en ese caso se muestra que los ticks de reloj equivalen a 2. Por lo que al parecer estaría bien.

Cambio de directorio:

Ahora haremos que reconozca la instrucción “cd” y, usando `chdir()`, cambie de directorio: Busqué como tenemos que pasarle los parámetros a la función `chdir()` en internet, y al ver que era simplemente pasándole el parámetro, lo hice como cuando añadimos el caso de “exit”, pero como esta función cuenta con argumentos, puse algunos condicionales para tratar los casos que no le pasemos argumentos, o el caso que no exista un directorio con el nombre pasado como parámetro:

```
else if (strcmp(result[0], "cd") == 0){
    if (result[1] != NULL){ //Comprueba que tiene un argumento
        if (chdir(result[1]) != 0){ //si no existe el directorio devuelve error
            perror("Error de cd");
        }
    }
    else{
        printf("Uso: cd <directorio>\n");
    }
}
```

```
[estudiante_ic2@ic2-centos-vm trabajo2]$ ./trabajo2v1
-----
Escribe órdenes: cd carpeta_prueba
18 caracteres han sido leídos.
-----
Contenido del vector resultado: (cd),(carpeta_prueba)
-----
Escribe órdenes: ls
3 caracteres han sido leídos.
-----
Contenido del vector resultado: (ls)
-----
Pid del proceso padre: 13387, pid del proceso hijo: 13398
-----
contenido
Tiempo de uso de la CPU del proceso hijo: 0
-----
Escribe órdenes: █
```



```

if (strcmp(result[0], "cd")==0){
    if (result[1] != NULL){ //Comprueba que tiene un argumento
        if (chdir(result[1]) != 0){ //si no existe el directorio devuelve error
            perror("Error de cd");
        }
    }
    else{
        printf("Uso: cd <directorio>\n");
    }
}

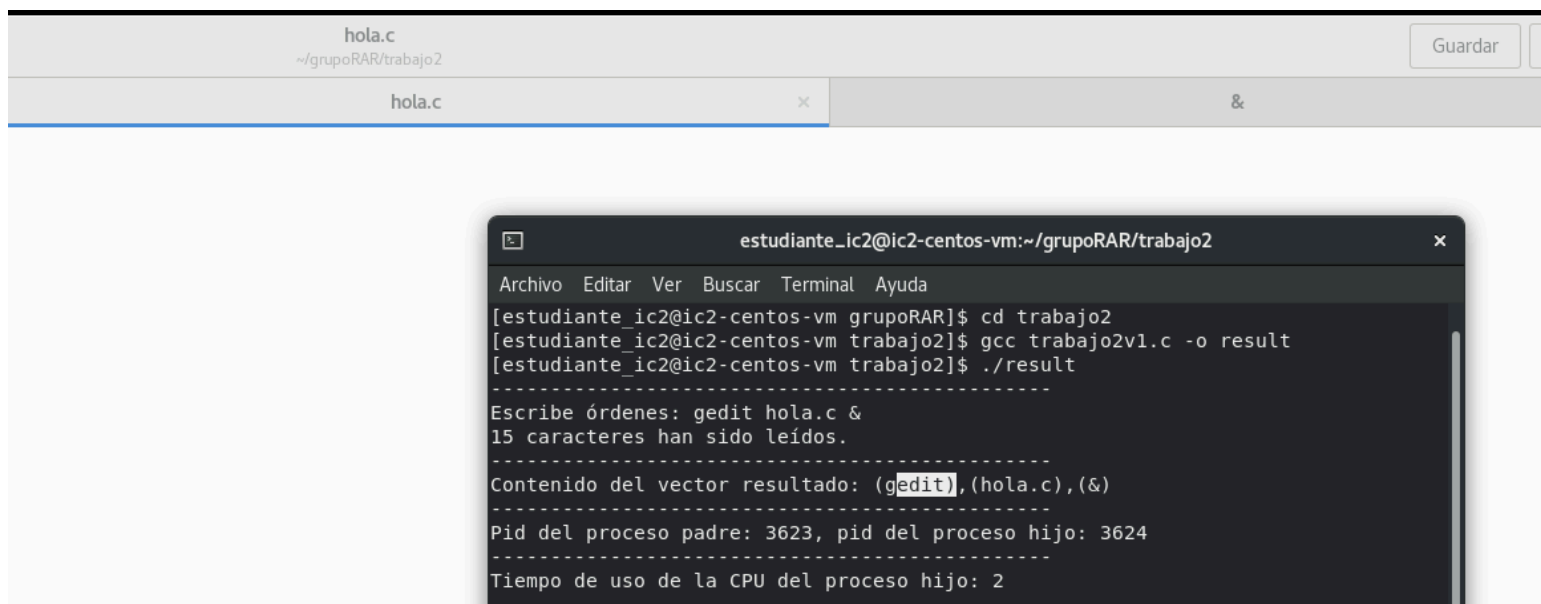
if (strcmp(result[0], "exit")==0){
    printf("-----\n");
    printf("Ejecución terminada. (Se ha introducido exit)\n");
    break;
}

/*if (strcmp(result[0], "cd")==0){
    if (result[1] != NULL){ //Comprueba que tiene un argumento
        if (chdir(result[1]) != 0){ //si no existe el directorio devuelve error
            perror("Error de cd");
        }
    }
}

```

Cambiamos esta parte del código para hacer que el cd dé su tiempo de CPU.

Ejecución asíncrona:



Pensamos que funcionaría de una manera similar al shell de Unix, pero aquí se ve como no detecta el & como ejecución asíncrona.

```
trabajo2v1.c
~/grupoRAR/trabajo2

result[a] = NULL;
//a++;

// ESTE ES UN SEGMENTO PARA LEER EL VECTOR (HECHO POR AYOZE)
// int tam = sizeof(hola) / sizeof(char*);
printf("-----\n");
printf("Contenido del vector resultado: ");
int amper;
for (i=0;i<a;i++){
    if (result[i] == "&"){
        amper = i;
    }
    if (i==a-1){
        printf("(%s)\n", result[i]);
    }
    else{
        printf("(%s),", result[i]);
    }
}
}
```

Aquí modificamos el código para que detecte que se le está pasando un ampersand.

```
*trabajo2v1.c
~/grupoRAR/trabajo2

int ampersand = 0;
else if (amper != NULL){
    result[amper] = NULL;
    ampersand = 1;
}

else{
    start_clock = times(&time_start);
    //printf("Start_clock = %d\n", start_clock);
    pid = fork();
    if (pid == 0){
        printf("-----\n");
        printf("Pid del proceso padre: %d, pid del proceso hijo: %d\n", getppid(), getpid());
        printf("-----\n");
        execvp(result[0], result);
    }
    else{
        if (ampersand){
            ampersand = 0;
            goto inicio;
        }
        wait(0);
        end_clock = times(&time_end);
        //printf("End_clock = %d\n", end_clock);
    }
}
```

Aquí, si ha detectado que hay un ampersand, va a fijar la variable ampersand como True, y a borrar el ampersand del vector. Luego, si en lo que se ejecuta el hijo ampersand está en True, va a la etiqueta inicio para continuar asíncronamente y pedir nuevas órdenes.

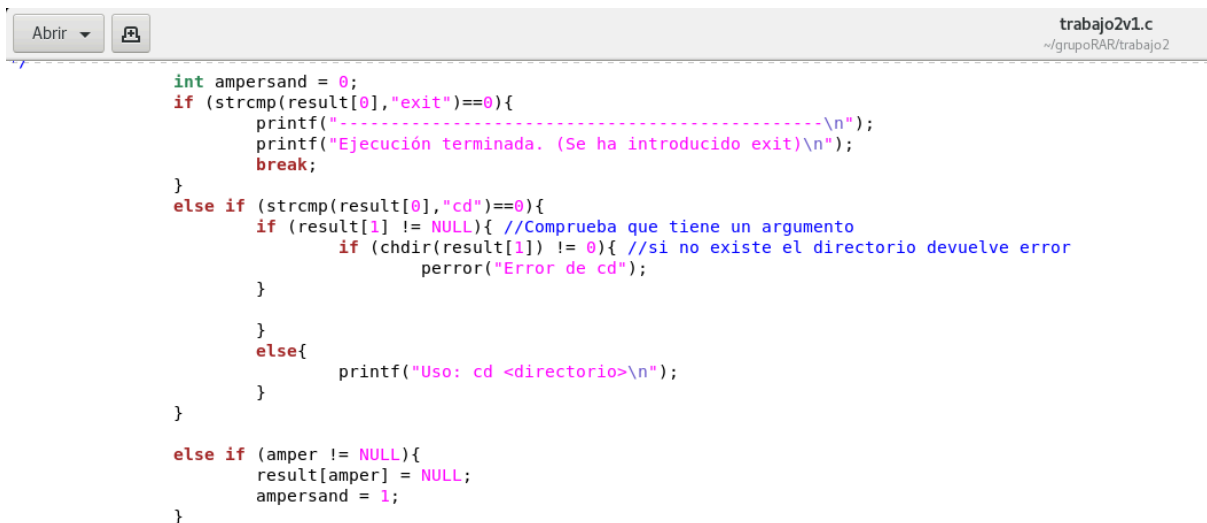
inicio:

```
buffer = (char *)malloc(bufsize * sizeof(char));

printf("-----\n");
printf("Escribe órdenes: ");
characters = getline(&buffer,&bufsize,stdin);
printf("%zu caracteres han sido leídos.\n",characters);
//printf("You typed: '%s'\n",buffer);

//printf("Instrucciones a ejecutar: ");
//printf("%s", buffer);
```

```
[estudiante_ic2@ic2-centos-vm trabajo2]$ sleep 2
[estudiante_ic2@ic2-centos-vm trabajo2]$ gcc trabajo2v1.c -o result
trabajo2v1.c: En la función 'main':
trabajo2v1.c:184:3: error: 'else' sin un 'if' previo
    else if (amper != NULL){
    ^~~~
trabajo2v1.c:184:18: aviso: comparación entre puntero y entero
    else if (amper != NULL){
                  ^~
```



```
trabajo2v1.c
~/grupoRAR/trabajo2

int amperand = 0;
if (strcmp(result[0],"exit")==0){
    printf("-----\n");
    printf("Ejecución terminada. (Se ha introducido exit)\n");
    break;
}
else if (strcmp(result[0],"cd")==0){
    if (result[1] != NULL){ //Comprueba que tiene un argumento
        if (chdir(result[1]) != 0){ //si no existe el directorio devuelve error
            perror("Error de cd");
        }
    }
    else{
        printf("Uso: cd <directorio>\n");
    }
}

else if (amper != NULL){
    result[amper] = NULL;
    amperand = 1;
}
```

No nos dio de esta manera, no nos quita el ampersand del vector.

Probamos luego de esta forma:

```
trabajo2v1.c
~/grupoRAR/trabajo2

//a++;

// ESTE ES UN SEGMENTO PARA LEER EL VECTOR (HECHO POR AYOZE)
// int tam = sizeof(hola) / sizeof(char*);
printf("-----\n");
printf("Contenido del vector resultado: ");
int amper = 0;
for (i=0;i<a;i++){
    if (strcmp(result[i], "&")==0){
        result[i] = NULL;
        //printf("he entrado\n");
        amper = 1;
    }
    if (i==a-1){
        printf("(%s)\n", result[i]);
    }
    else{
        printf("(%s),", result[i]);
    }
}
}
```

```
trabajo2v1.c
~/grupoRAR/trabajo2

else{
    start_clock = times(&time_start);
    //printf("Start_clock = %d\n", start_clock);
    pid = fork();
    if (pid == 0){
        printf("-----\n");
        printf("Pid del proceso padre: %d, pid del proceso hijo: %d\n", getpid(), getpid());
        printf("-----\n");
        execvp(result[0], result);
    }
    else{
        if (amper){
            goto inicio;
        }
        wait(0);
        end_clock = times(&time_end);
        //printf("End_clock = %d\n", end_clock);
    }
}
```

Además creamos este programa auxiliar, del que sabemos el tiempo que tarda en ejecutarse.

```
prueba.c
~/grupoRAR/trabajo2

#include <stdio.h>
#include <unistd.h>

void main () {
    sleep(10);
    printf("Ha terminado prueba.c");
    printf("\n");
}
```

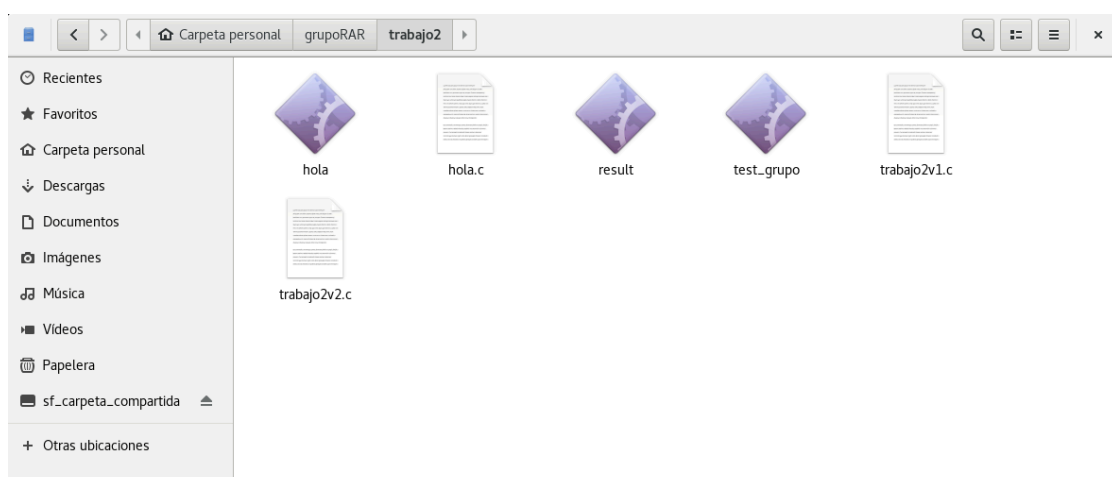
```

[estudiante_ic2@ic2-centos-vm trabajo2]$ ./result
-----
Escribe órdenes: ./hola
7 caracteres han sido leídos.
-----
Contenido del vector resultado: (./hola)
-----
Pid del proceso padre: 5172, pid del proceso hijo: 5174
-----
Ha terminado prueba.c
Tiempo de uso de la CPU del proceso hijo: 0
-----
Escribe órdenes: ./hola &
9 caracteres han sido leídos.
-----
Contenido del vector resultado: (./hola),((null))
-----
Escribe órdenes: -----
Pid del proceso padre: 5172, pid del proceso hijo: 5175
-----
ls
3 caracteres han sido leídos.
-----
Contenido del vector resultado: (ls)
-----
Pid del proceso padre: 5172, pid del proceso hijo: 5176
-----
hola old prueba.c result trabajo2v1.c
Tiempo de uso de la CPU del proceso hijo: 0
-----
Escribe órdenes: Ha terminado prueba.c

```

El ./hola tarda 10 segundos en printear su frase. Entonces, lo ejecutamos con el ampersand, y nos dejó escribir y ejecutar ls en medio de su ejecución. Realmente se acerca al objetivo, pero el output es inconsistente.

Ejecución del 1er entregable (asíncronamente):



En la foto anterior, mostramos cómo el ejecutable de nuestra versión sin structs de nuestro 1er trabajo se encuentra en la misma carpeta a la que pertenece este trabajo nº2. Utilizamos la versión sin structs porque es la única que nos funcionaba.

```
estudiante_ic2@ic2-centos-vm:~/grupoRAR/trabajo2
Archivo Editar Ver Buscar Terminal Ayuda

[estudiante_ic2@ic2-centos-vm trabajo2]$ ./result
-----
Escribe órdenes: ./test_grupo 6
15 caracteres han sido leídos.
-----
Contenido del vector resultado: (./test_grupo),((null))
-----
Escribe órdenes: -----
Pid del proceso padre: 4707, pid del proceso hijo: 4708
-----
Introduzca el tamaño de la lista a crear: ls
3 caracteres han sido leídos.
-----
Contenido del vector resultado: (ls)
-----
Pid del proceso padre: 4707, pid del proceso hijo: 4709
-----
hola hola.c result test_grupo trabajo2v1.c trabajo2v2.c
Tiempo de uso de la CPU del proceso hijo: 0
-----
Escribe órdenes: 10
(-1),(-1),(-1),(-1),(-1),(-1),(-1),(-1),(-1),(-1)
(Se intenta desmatricular cuando no hay nadie matriculado)
No hay alumnos matriculados en el grupo

(Se procede a hacer una matricula multiple)
(38808158),(17598594),(51489657),(-1),(-1),(-1),(-1),(-1),(-1)

(Se procede a hacer otra matricula multiple)
(38808158),(17598594),(51489657),(99808158),(99598594),(99489657),(99857465),(35248697),(-1),(-1)

(Se mete un matriculado nuevo(solo uno))
Se ha matriculado el alumno con DNI 59317174 en el asiento/posición 8
(38808158),(17598594),(51489657),(99808158),(99598594),(99489657),(99857465),(35248697),(59317174),(-1)

(Se intenta matricular a alguien ya matriculado)
El alumno con DNI 59317174 ya está matriculado
(38808158),(17598594),(51489657),(99808158),(99598594),(99489657),(99857465),(35248697),(59317174),(-1)

(Se desmatricula el primero)
El alumno con DNI 38808158 y posición 0 ha sido desmatriculado del grupo
(-1),(17598594),(51489657),(99808158),(99598594),(99489657),(99857465),(35248697),(59317174),(-1)

(Se intenta desmatricular a alguien no matriculado)
No se ha encontrado el DNI solicitado en la lista
(-1),(17598594),(51489657),(99808158),(99598594),(99489657),(99857465),(35248697),(59317174),(-1)
```

```
estudiante_ic2@ic2-centos-vm:~/grupoRAR/trabajo2
Archivo Editar Ver Buscar Terminal Ayuda

(Se procede a hacer una matricula multiple, que sobrepase el límite)
No hay plazas libres para todas las personas
(-1),(17598594),(51489657),(99808158),(99598594),(99489657),(99857465),(35248697),(59317174),(-1)

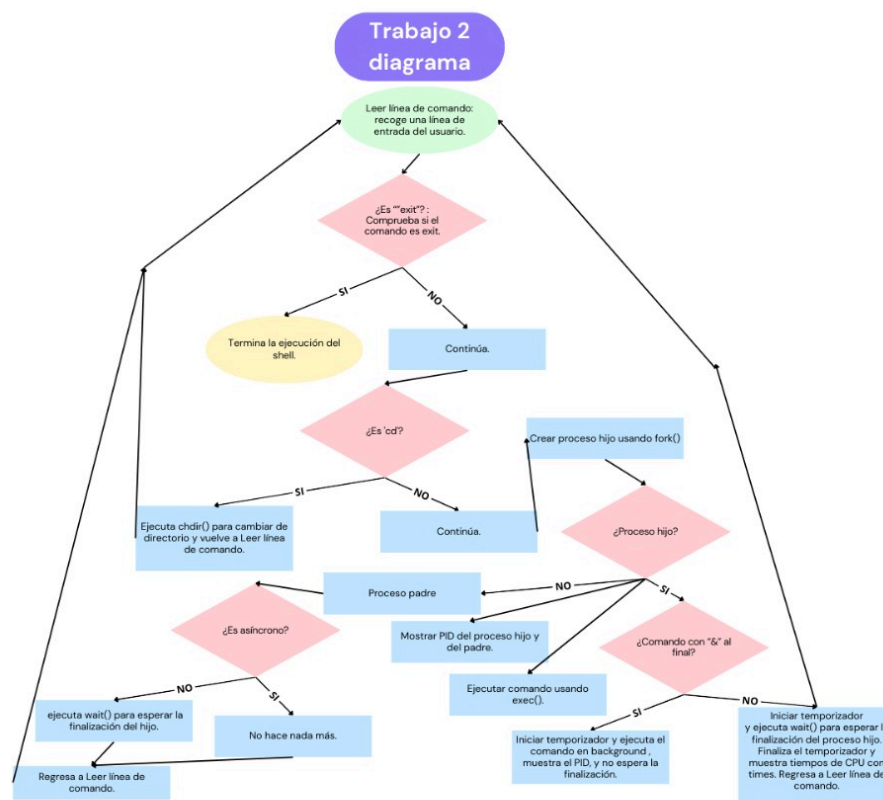
El asiento 11 no se encuentra en el rango
El asiento 28 no se encuentra en el rango
La persona con DNI: 17598594 esta sentado en el asiento: 1

(0),(0),(18653200),(0),(99598594),(99489657),(99857465),(35248697),(59317174),(-1)

Ahora creamos otro grupo: ls
3 caracteres han sido leídos.
-----
Contenido del vector resultado: (ls)
-----
Pid del proceso padre: 4707, pid del proceso hijo: 4710
-----
hola hola.c result test_grupo trabajo2v1.c trabajo2v2.c
Tiempo de uso de la CPU del proceso hijo: 0
-----
Escribe órdenes: 10
(-1),(-1),(-1),(-1),(-1),(-1),(-1),(-1),(-1),(-1)
ps
3 caracteres han sido leídos.
-----
Contenido del vector resultado: (ps)
Tiempo de uso de la CPU del proceso hijo: 0
-----
Escribe órdenes: Pid del proceso padre: 4707, pid del proceso hijo: 4719
-----
      PID TTY          TIME CMD
  4278 pts/0    00:00:00 bash
  4707 pts/0    00:00:00 result
  4719 pts/0    00:00:00 ps
ps
3 caracteres han sido leídos.
-----
```

Primero, printea todo lo que debería printear de los dos procesos y separa cuando ocurre el primer input (espera órdenes en vez de un tamaño de vector). Luego, ejecuta la orden, se printea *Escribe órdenes*, pero lo que espera es un tamaño de vector, porque estaba esperando la lectura en teclado mientras se imprimía lo anterior. Se inutea un tamaño, e imprime todo lo del trabajo 1, pero lo que espera es otra orden, en vez de otro tamaño. Al final, seguido de la ejecución de una orden nueva, espera que se introduzca otro tamaño de vector, como se pidió anteriormente. A partir de ahí, el programa sigue su función habitual, pero con un output algo lioso. La manera en la que se ejecuta el programa del trabajo 1 junto a otras órdenes de forma asíncrona, se puede asociar a una FIFO (primero espera orden, luego número, después orden, más tarde número y finalmente orden).

Diagrama de flujo final:



Distribución del trabajo:

- **Enrique:** Desarrolló la estructura del main, llevó a cabo intentos del splitter, implementó las funciones de string.h en el código + ayuda en los otros apartados + dos últimos apartados (junto con Ayoze).

- **Fabio:** Ayudó a Enrique en todo lo anterior + hizo los dos primeros apartados opcionales (tiempos de cpu, cambio de directorio) + ayudó en el desarrollo en general + primer diagrama de flujo + diagrama flujo final.

- **Ayoze:** Creó la fase final del splitter, perfiló el algoritmo del getline(), creó la función leervector() para debuggear, realizó la versión final del output + dos últimos apartados (junto con Enrique) + gran parte de la realización de la memoria.