

Instituto Politécnico Nacional

Escuela Superior de Cómputo

*Sistemas Operativos*

*“Práctica 3. Administrador de procesos en Linux y Windows (1)”*

**Grupo:** 2CM9

**Integrantes:**

* Martínez Coronel Brayan Yosafat.
* Monteros Cervantes Miguel Angel.
* Ramírez Olvera Guillermo.
* Sánchez Méndez Edmundo Josue.

**Profesor:** Cortés Galicia Jorge



**Práctica 3. Administrador de procesos en Linux y Windows (1) Introducción**

Un proceso podría ser una instancia de un programa en ejecución. A los procesos frecuentemente se les refiere como tareas. El contexto de un programa que está en ejecución es lo que se llama un proceso. Por ejemplo, Linux es un sistema operativo multitarea y multiusuario. Esto quiere decir que múltiples procesos pueden operar simultáneamente sin interferirse unos con los otros. Cada proceso tiene la "ilusión" que es el único proceso en el sistema y que tiene acceso exclusivo a todos los servicios del sistema operativo.

Programas y procesos son entidades distintas, múltiples instancias de un programa pueden ejecutarse simultáneamente. Cada instancia es un proceso separado. Por ejemplo, si usuarios desde equipos diferentes, ejecutan el mismo programa al mismo tiempo, habría tantas instancias del mismo programa, es decir, procesos distintos.

Cada proceso que se inicia es referenciado con un número de identificación único conocido como Process ID PID, que es siempre un entero positivo, al final veremos una curiosidad y diferencia entre Linux y Windows. Prácticamente todo lo que se está ejecutando en el sistema en cualquier momento es un proceso, incluyendo el shell, el ambiente gráfico que puede tener múltiples procesos, etc. La excepción a lo anterior es el kernel en sí, el cual es un conjunto de rutinas que residen en memoria y a los cuales los procesos a través de llamadas al sistema pueden tener acceso.

Ademas, como hemos visto en las lecturas dejados, nos damos cuenta de que es de nuestro interés tener un buen proceso de planificación ya que se pretende usar la CPU el mayor tiempo posible, y esto se debe de hacer con base a los estados de los procesos, en espera, preparado y terminado, mas adelante veremos como en podemos saber el estado de procesos en la computadora (Ubuntu) y de procesos que nosotros creemos y ademas de saber sus estados poder tener control sobre los mismos.

Ademas de ver Planificadores y Multiprogramación de una forma práctica, el primer punto es el programa con 6 procesos con sustitución de código en donde llevaremos una planificación de pasos a ejecutar y el de multiprogramación con 6 procesos con copia exacta de código y veremos un caso en donde nos encontraremos en problemas si no tenemos cuidado en como usaremos los procesos.

Concluyendo esta introducción hay que mencionar que los comandos y funciones que se verán más adelante, también son llamadas del sistema, por lo que, nuestro programa seguirá teniendo las ventajas que en la practica de la anterior unidad se comentaron.

**1. Competencias.**

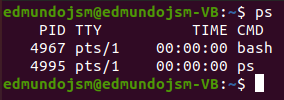
El alumno aprende a familiarizarse con el administrador de procesos del sistema operativo Linux y Windows a través de la creación de nuevos procesos por copia exacta de código y/o por sustitución de código para el desarrollo de aplicaciones concurrentes sencillas.

**2. Desarrollo.**

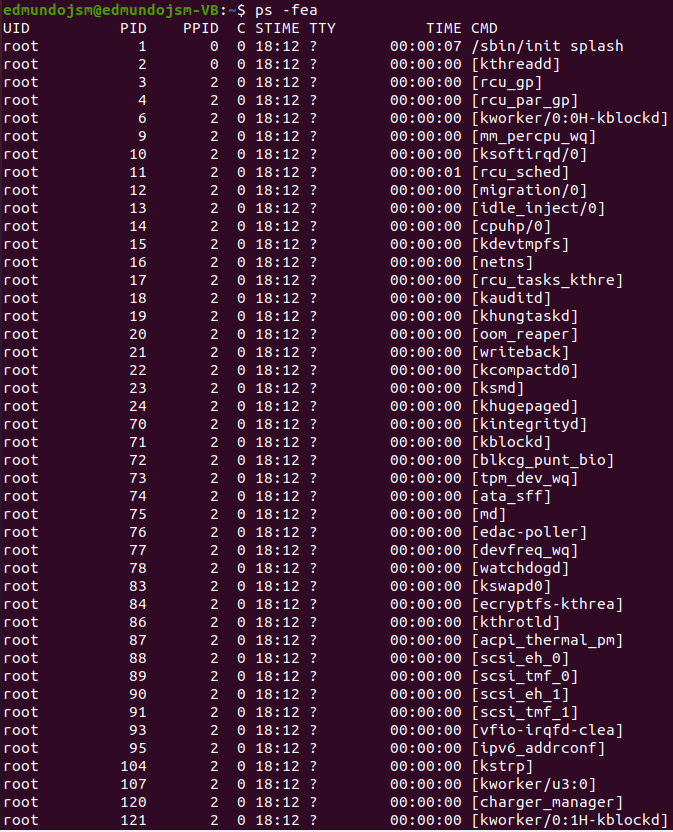
**2.1. Sección Linux**

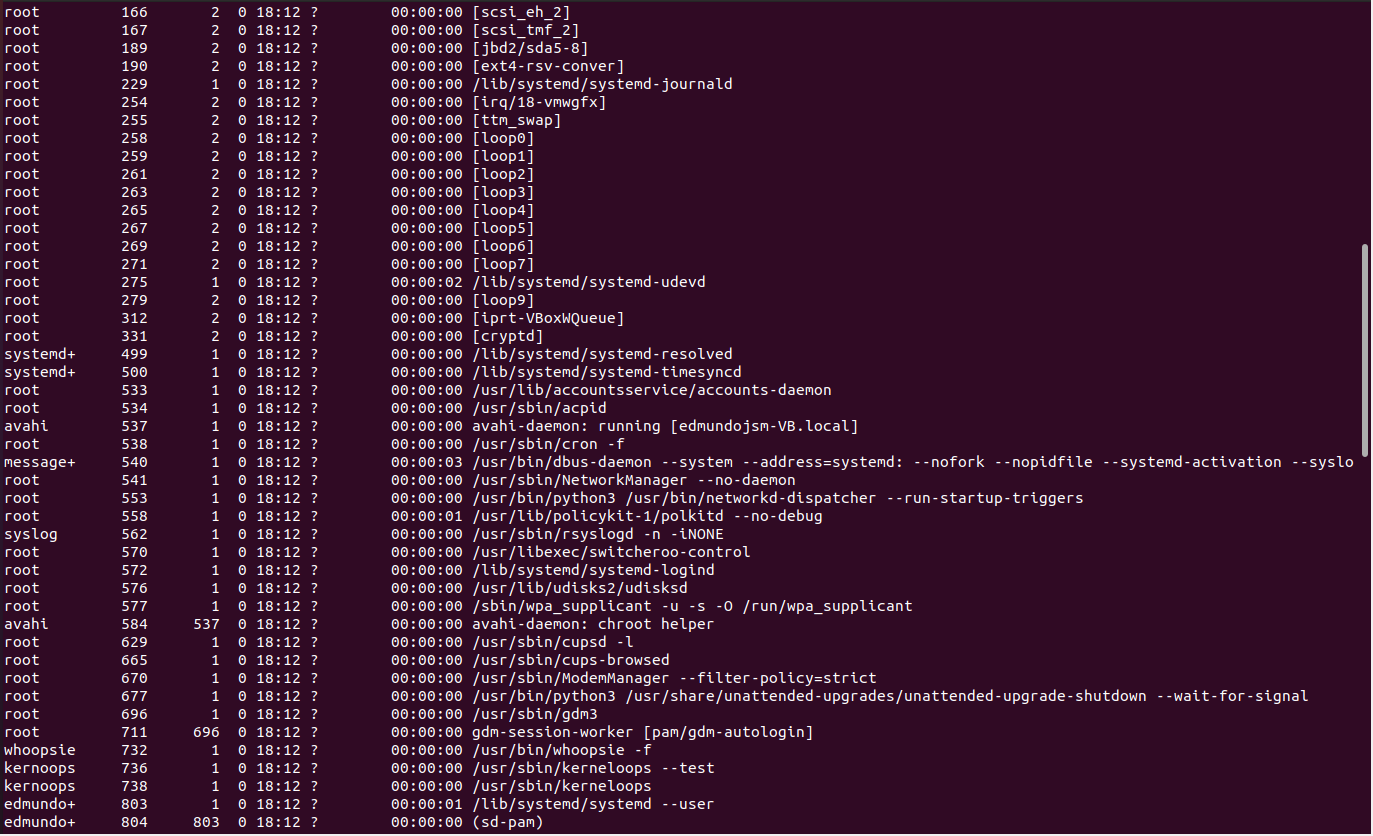
**2.1.1. Ejecución de los comandos *ps* y *ps -fea***

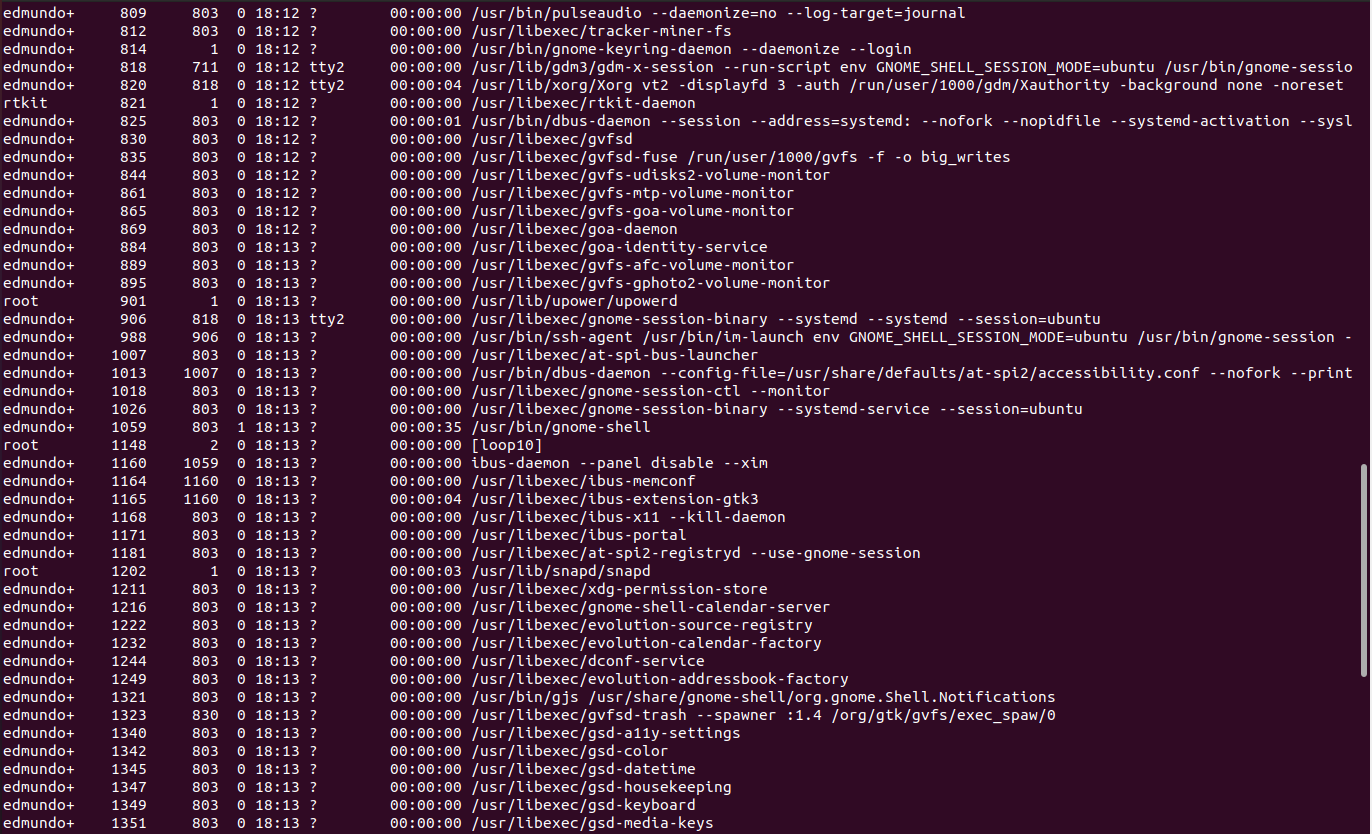
* Comando **ps**. Muestra el número del proceso (**PID**), tipo de terminal (**TTY**), tiempo acumulado de ejecución (**TIME**) y el nombre del comando (**CMD**).

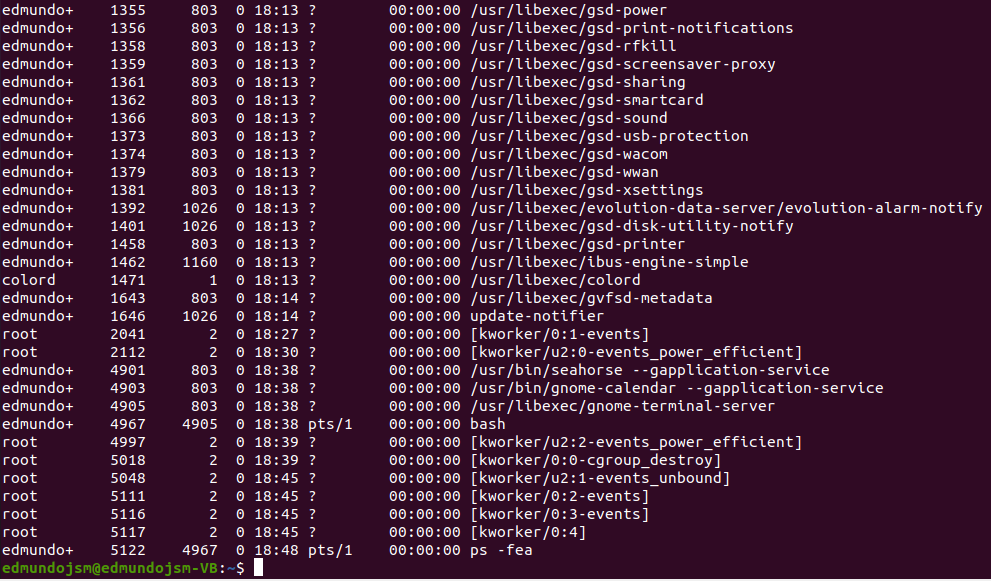


* Comando ps -fea. Muestra el usuario propietario del proceso, la identificación del proceso, el padre de proceso, porcentaje de la CPU utilizado por el proceso, hora de inicio del proceso, terminal asociada al proceso (si no hay terminal aparece entonces un ’?’), tiempo de uso de CPU acumulado por el proceso y el nombre del programa o comandó que inicio el proceso.









**2.1.2. Información de comandos y llamadas al sistema**

Información del comando **ps**:

* **pstree**: Muestra un árbol de procesos.
* **ps -aux** o textttps aux: Lista de los programas que se están ejecutando actualmente con su *PID*.
* **ps -ef l grep nombre-proceso**: Lista los procesos que se están ejecutando con que contengan la cadena nombre-proceso.
* **ps aux**: Lista los procesos con la información de *USER, PID, CPU, MEM, VSZ, RSS, TTY, STAT, START, TIME, COMMAND.*
* **ps ax**: Lista los procesos con la información de *PID, TTY, STAT, TIME, COMMAND.*
* **ps -a**: Lista los procesos de todos los usuarios.
* **ps -u**: Lista información del proceso como por ejemplo el usuario que lo está corriendo, la utilización de CPU y memoria, etc.
* **ps -x**: Lista procesos de todas las terminales y usuarios
* **ps -l**: Muestra información que incluye el *UID*.
* **ps -e**: Visualiza información sobre “todos” los procesos del sistema.
* **ps axjf**: Mostrará un árbol jerárquico con la ruta del programa al que pertenece el proceso.
* **ps aux** | **grep bash**: Realiza filtrado sobre ps para obtener únicamente los procesos pertenecientes a *bash*.

Ahora, veremos los campos de la salida del comando **ps**:

* UID: ID de usuario.
* PID: ID del proceso.
* PPID: ID del proceso padre.
* PGID: ID de grupo de un proceso.
* PRI: Prioridad del proceso.
* NI: valor de bondad, más elevado menor prioridad.
* VSZ: Tamaño de la memoria virtual del proceso en Kb.
* RSS: Tamaño de la memoria física usada en Kb.
* WCHAN. para los procesos que esperan o dormidos, enumera el evento que espera.
* STAT: Estado del proceso:
  + R: Ejecutable.
  + D: Interrumpió.
  + S: Suspendido.
  + s: Es el proceso líder de la sesión.
  + T: Detenido.
  + Z: Zombie.
  + X: Muerto.
  + <: Tiene una prioridad alta que lo normal.
  + N: Tiene una prioridad menor que lo normal.
  + L: Tiene paginas bloqueadas en la memoria (para E/S en tiempo real y personalizadas).
  + l: Es multiproceso.
  + +: Está en el grupo de proceso de primer plano,
* TTY: nombre de la terminal a la que está asociado al proceso.
* TIME: tiempo que lleva en ejecución.

Información del comando **fork()**: Llamada para la creación de un proceso por copia exacta de código. Valores de retorno:

* 0 si es el proceso hijo.
* -1 si no se ha podido crear el proceso hijo
* Mayor a cero si es el proceso padre

Para hacer uso de esta llamada se deben incluir en la cabecera:

* **sys/types.h**
* **unistd.h**

Información del comando **execv (const char\* path, char\* const argv[ ])**: Llamada para la creación de un proceso por sustitución de código. Si la llamada falla se retornara -1. Parámetros:

* **const char\* path**: Ruta del nuevo programa a ejecutar con su ruta.
* **char\* const argv[ ]**: Lista de argumentos disponibles para el nuevo programa. El último de los punteros debe ser NULL. Por convención, este array debe contener al menos un elemento (nombre del programa).

Para hacer uso de esta llamada se deben incluir en la cabecera **unistd.h**

Información del comando **getpid()**: Llamada al sistema para obtener el identificador del proceso quien invoca la llamada. Retorna un entero del tipo **pid\_t**. Para hacer uso de esta llamada se deben incluir en la cabecera:

* **sys/types.h**
* **unistd.h**

Información del comando **getppid()**: Llamada al sistema para obtener el identificador del proceso padre del proceso quien invoca la llamada. Retorna un entero del tipo **pid\_t**. Para hacer uso de esta llamada se deben incluir en la cabecera:

* **sys/types.h**
* **unistd.h**

Información del comando **wait(int \*status)**: Llamada al sistema para hacer esperar un proceso padre a uno de sus procesos hijo. Parámetro:

* int \*status: Esta funcion recibe como argumento un puntero a una variable entera en la que se colocara el estado actual del proceso hijo y retorna el **pid** del proceso hijo que termino. El argumento también puede colocarse **NULL**.

Funciones similares a **execv()**

La familia exec...() es un conjunto de funciones que en esencia realizan la misma actividad ya que solo difieren en la forma de pasar sus argumentos, son utilizadas para poner en ejecución un proceso determinado, la característica es que las instrucciones del proceso que las invoca son sustituidas por las instrucciones del proceso indicado. Si la llamada falla se retornara -1. Las funciones son:

* **int execl (char \*path, char \*arg0, char \*arg1, . . . ,char \*argN, char \*null)**
* **int execle (char \*path, char \*arg0, ... ,char \*argN, char \*null, char \*envp[])**
* **int execlp (char \*file, char \*arg0, char \*arg1, ... ,char \*argN, char \*null)**
* **int execv (char \*path, char \*argv[])**
* **int execve (char \*path, char \*argv[], char \*envp[])**
* **int execvp (char \*file, char \*argv[])**

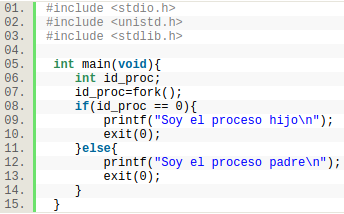
Características de los parámetros:

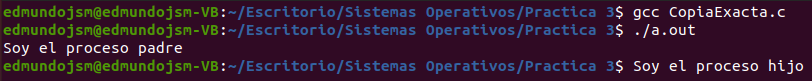
* **path,** **file**: Nombre del nuevo programa a ejecutar con su ruta. Las versiones de ***exec*** que utilizan file en lugar de ***path*** utilizan la variable de entorno **PATH** para localizar el programa a ejecutar, por lo que en esos casos no es necesario especificar la ruta al programa si este se encuentra en alguno de los directorios especificados en **PATH**.
* **arg0**: Primer argumento del programa. Por convención suele asignarse el nombre del programa sin la ruta.
* **arg1, ..., argN, NULL**: Conjunto de parámetros que recibe el programa para su ejecución.
* **argv**: Lista de argumentos disponibles para el nuevo programa. El último de los punteros debe ser NULL. Por convención, este array debe contener al menos un elemento (nombre del programa).
* **envp**: Cadena de caracteres constituyen el entorno de ejecución del nuevo programa.

Para hacer uso de esta llamada se debe incluir la cabecera **unistd.h**

**2.1.3. Ejemplo de creación de procesos por copia exacta de código**

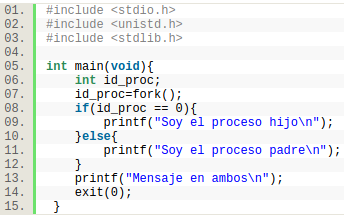
Código(CopiaExacta.c)

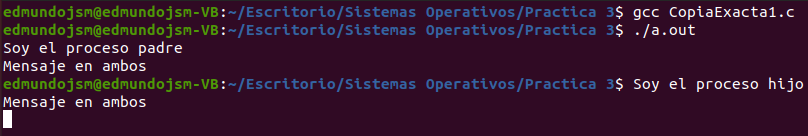




Ejecuta internamente de manera concurrente dos procesos mediante la invocación de la llamada al sistema fork(). El sistema en el que se ejecutó el código está configurado de la manera en que le da prioridad al proceso padre la mayoría de las veces. Por medio de los *if* y el valor que retorna **fork()** podemos ver la salida de los *printf* en pantalla y observar ambos procesos (los cuales finalizan con la llamada al sistema *exit (0)*).

Código(CopiaExacta1.c)



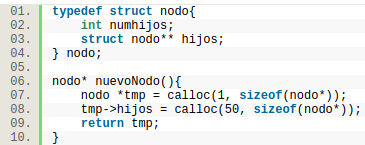


Ejecuta internamente de manera concurrente dos procesos mediante la invocación de la llamada al sistema **fork()**. El sistema en el que se ejecutó el código está configurado de manera que le da prioridad al proceso padre. Por medio de los *if* y el valor que retorna **fork()** podemos ver la salida de los *printf* en pantalla y observar ambos procesos que finalizan con la llamada al sistema *exit(0)* la cual tiene un *printf(”Mensaje en ambos”)* y si se colocara antes el *exit(0)* no lo imprimiría en pantalla.

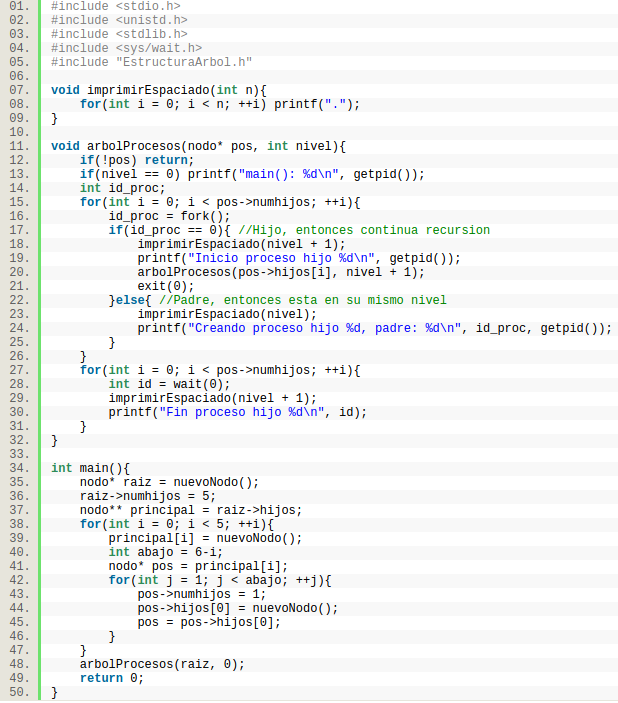
**2.1.4. Creación del árbol de procesos**

Para este punto se decidió crear una estructura ***nodo*** la cual se encuentra en el código “EstructuraArbol.h”, la cual simplemente contiene un arreglo de sus nodos hijos. Después se crea de manera recursiva (en profundidad, usando la técnica de DFS) el árbol de procesos de acuerdo con lo que contenga el nodo raíz.

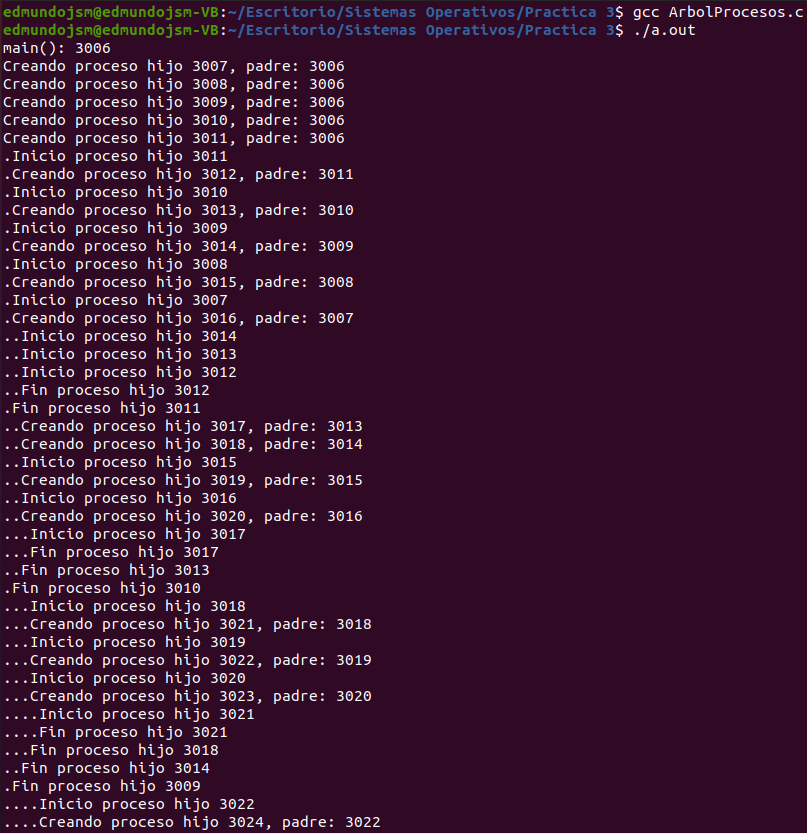
Código(EstructuraArbol.h)



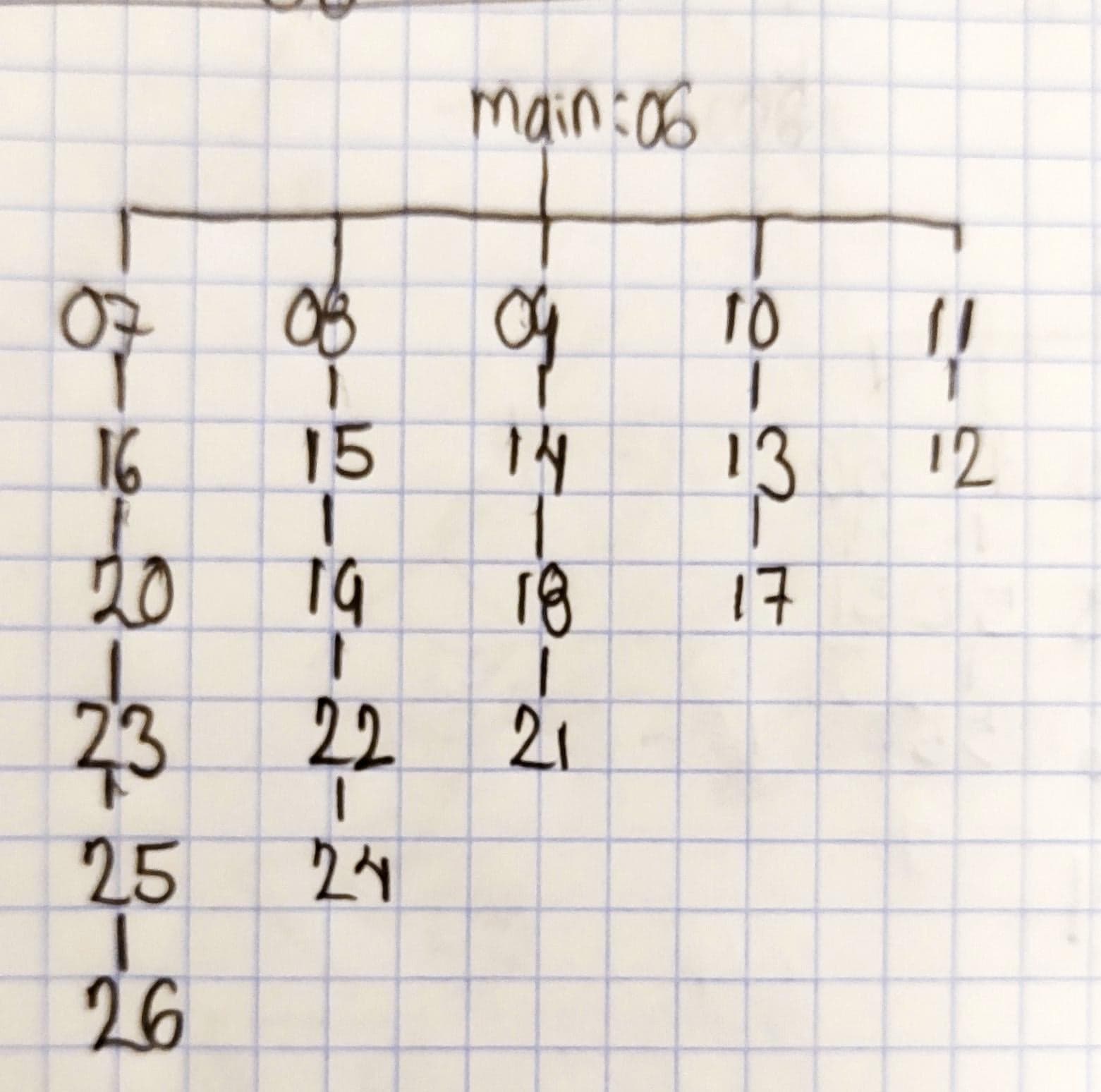
Código(ArbolProcesos.c)



Resultado al momento de ejecutar y compilar el programa:

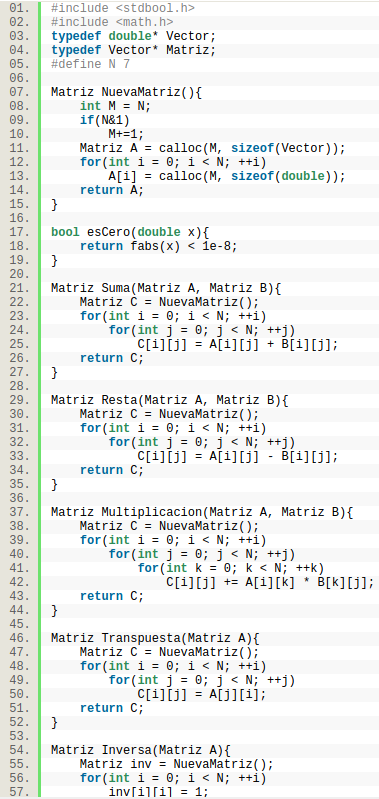
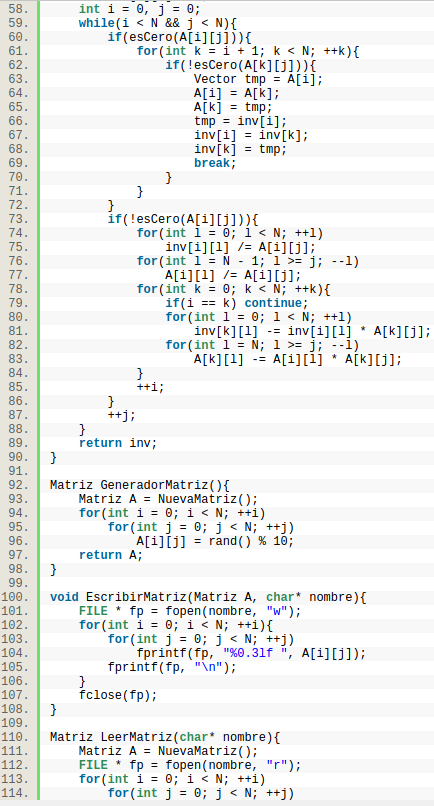
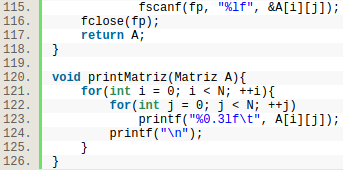
Y de acuerdo con los PID’s impresos, el árbol que se creó realmente fue (solo usamos los dos últimos dígitos del PID, pues todos comienzan con 30):



El cual coincide con el árbol solicitado.**2.1.5. Aplicación con seis procesos por copia exacta de código (Matrices)**

En el siguiente archivo se implementa las operaciones básicas entre dos matrices de un tamaño fijo de 7×7 (Se define una variable N la cual puede ser modificada por otro número entero positivo), así como funciones para importar/exportar desde un archivo, imprimir en pantalla y generar matrices aleatorias; con el fin de reutilizar código y que este se vea más presentable. Las entradas de la matriz son del tipo int (entero).

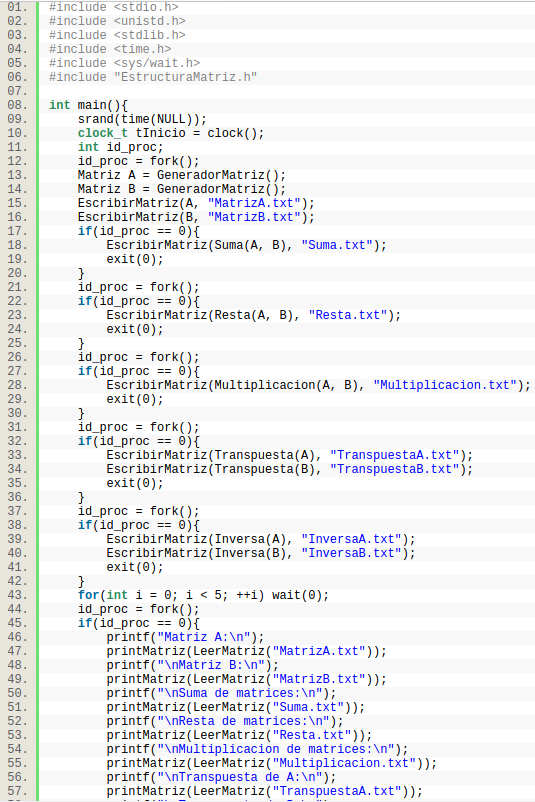
Código(EstructuraMatriz.h)

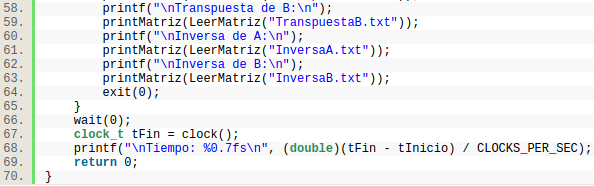


**Parte en donde se hace uso de los 6 procesos**

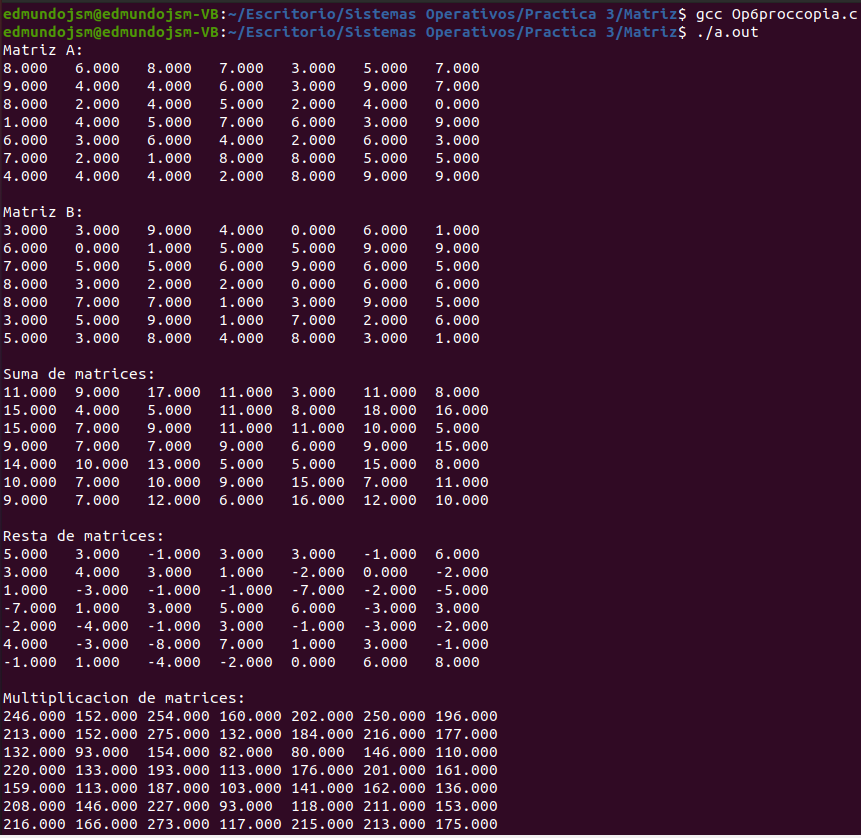
Para la creación de procesos por copia exacta de código, primero crearemos dos matrices A y B con entradas aleatorias enteras entre 0-9, Luego lanzamos cinco procesos con las operaciones requeridas entre las matrices y guardamos el resultado en un archivo. Esperamos a que terminen estos procesos, y luego lanzamos un sexto proceso que muestra los resultados de los archivos. Al final mostramos el tiempo total.

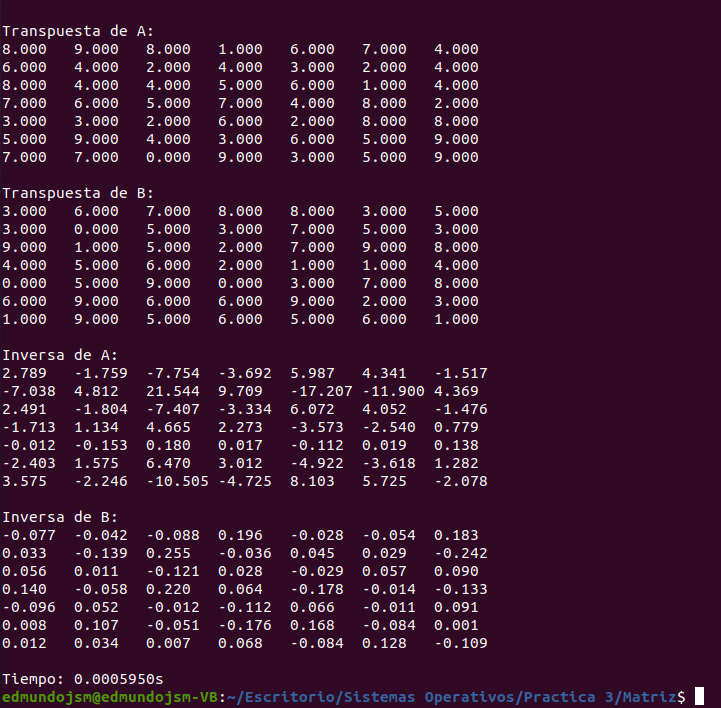
Código(Op6proccopia.c)



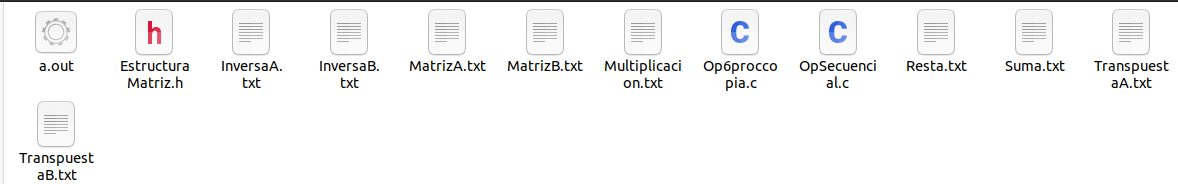


Resultado al momento de ejecutar y compilar el programa, al final del programa se nos muestra el tiempo que tardo en finalizar el programa. Nota: se muestra en segundos.





Archivos generados:



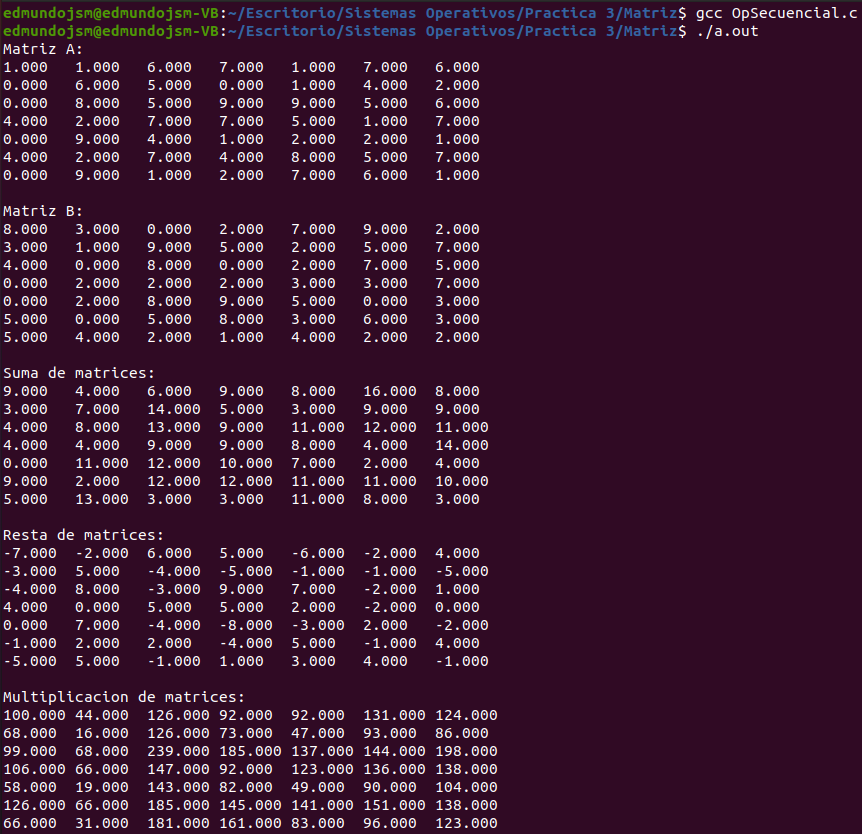
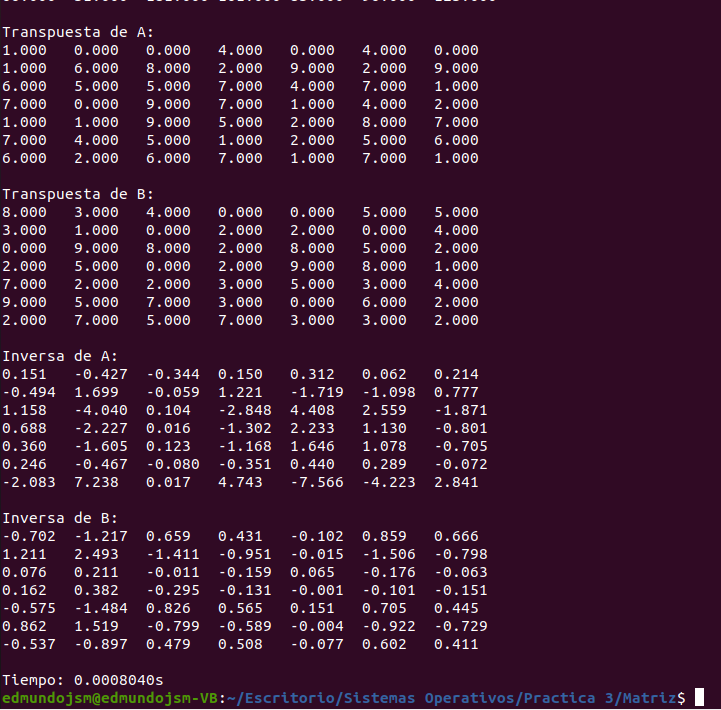
**Parte en donde no se hace uso de los 6 procesos**

Reutilizaremos el código de EstructuraMatriz.h. En este caso simplemente ejecutamos las operaciones de forma secuencial, es decir, pedidas una detrás de otra.

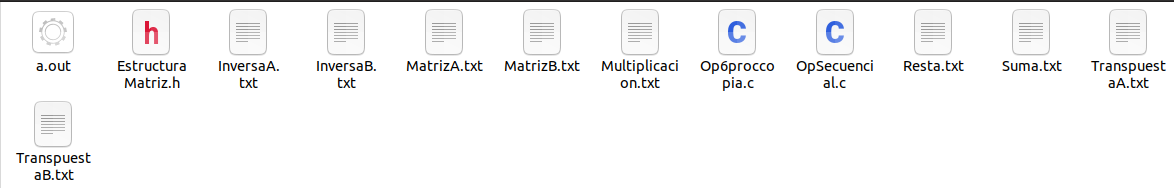
Código(OpSecuencial.c)



Resultado al momento de ejecutar y compilar el programa, al final del programa se nos muestra el tiempo que tardo en finalizar el programa. Nota: se muestra en segundos.

Archivos generados:

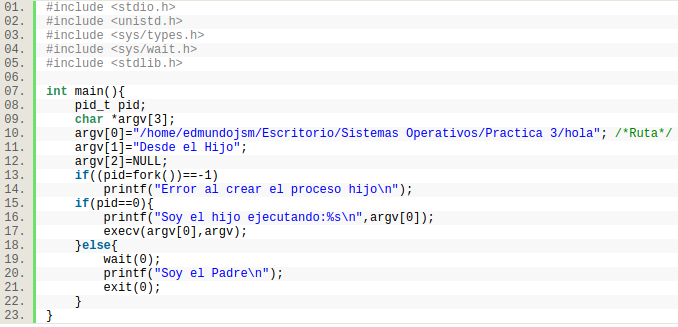


Vemos que la aplicación secuencial tardó un poco más en ejecutarse que la que usa copia exacta de código, con base en el anterior resultado podemos deducir que cada proceso se tardo un poco mas en ser ejecutado en el programa secuencial y podemos concluir que usar copia exacta de código es mejor en cuanto a tiempo de ejecución, es decir, para programas en donde necesitamos bajar el tiempo, usar copia exacta de código es una muy buena opción.

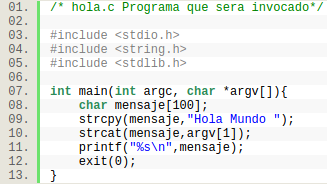
**2.1.6. Ejemplo de creación de procesos por sustitución de código**

Por medio del siguiente código podemos observar el uso de la llamada al sistema **fork()**, **execv(const char\* path, char\* const argv[])** y **wait(int \*status)**. La llamada al sistema **execv(const char\* path, char\* const argv[])** hace la creación de un proceso por sustitución de código, el cual es “*hola.c*”.

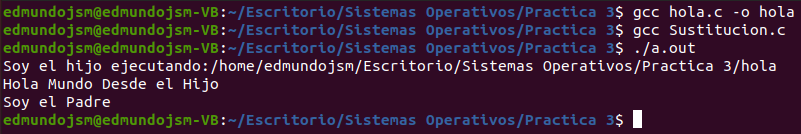
Código(Sustitucion.c)



Código(hola.c)



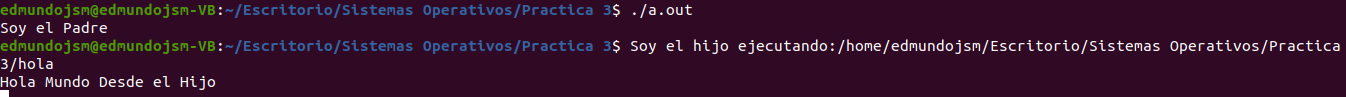
Resultado al momento de ejecutar y compilar el programa:



Con el código realizamos dos procesos con la llamada al sistema **fork()** y podemos observar con la llamada al sistema **wait(int \*status)** como configura al sistema operativo para que se ejecute primero el proceso hijo, mientras que el proceso padre espera a que finalice su proceso hijo.

En el proceso hijo observamos el uso de la llamada al sistema **execv(const char\* path, char\* const argv[])** que hace la creación de un proceso por sustitución de código que imprime el mensaje “Hola Mundo Desde el Hijo”, indicando que es el hijo quien lo ejecuto y terminando el proceso hijo se regresa al proceso padre que imprime el mensaje “Soy el padre”.

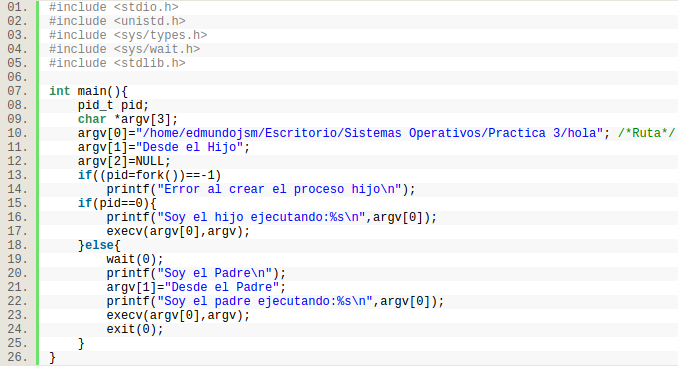
Pero que pasara si quitáramos wait();?



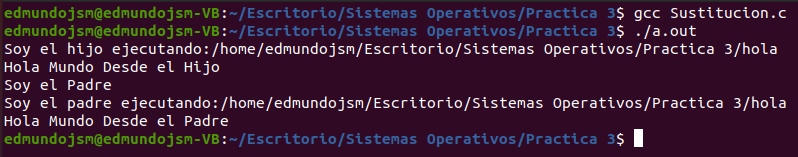
En caso de que no tuviera la llamada al sistema **wait (int \*status)** se ejecutaría primero el proceso padre y después el proceso hijo en la mayoría de los casos.

Por último, y si quisiera que tanto el hijo como padre ejecutaran hola.c?

Sencillo, solo tendríamos que copiar la llamada al sistema **execv(const char\* path, char\* const argv[])** y cambiar el valor de argv[1] por “Desde el Padre”, por lo que el código quedaría:



El resultado sería el siguiente:



**2.1.7. Tres procesos usando sustitución de código**

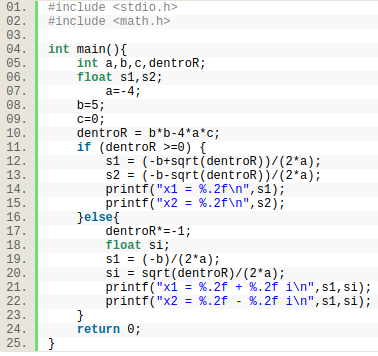
Nuestro código principal ser 3CodigoSustitucion.c el cual en su código fuente hace una sustitución de código con los siguientes 3 programas:

* Resolver ecuación algebraica de segundo grado mediante la fórmula general. (SutituirEc.c)
* Mostrar serie Fibonacci para un número N (SustituirFibo.c)
* Multiplicación de dos matrices de 7x7 de tipo entero(SustituirMat.c)

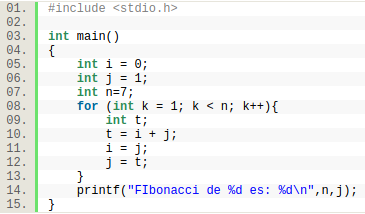
Código(3CodigoSustitucion.c)



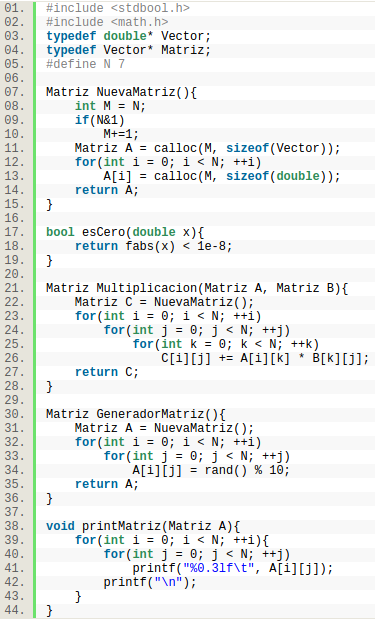
Código(SutituirEc.c) compilado como Sustituir1:

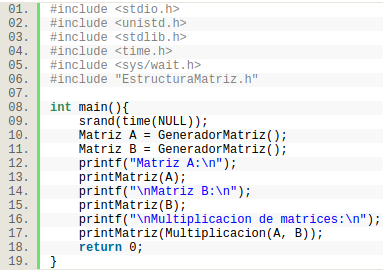


Código(SustituirFibo.c) compilado como Sustituir2:

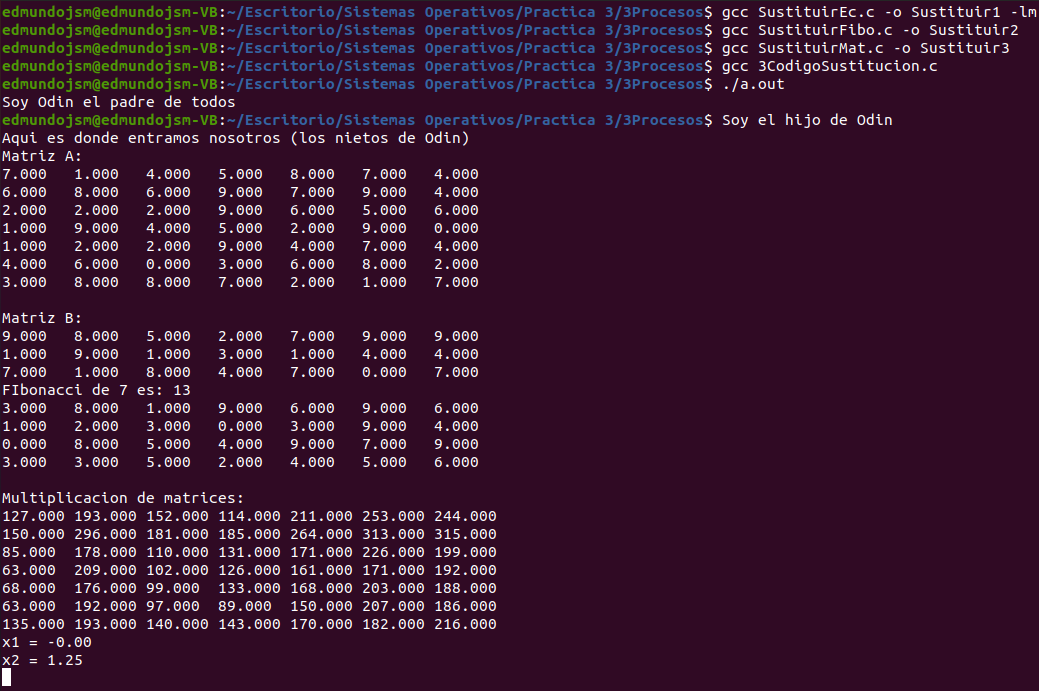


Código(EstucuraMatriz.c) primer imagen y Código(SustituirMat.c) segunda imagen, el ultimo es compilado como Sustituir3:





Compilación y ejecución del código:



¿Es posible un funcionamiento 100 % concurrente de su aplicación? Explique porque sí o no es 100% concurrente su aplicación.

No, no es posible 100 % concurrente, y la razón reside en la creación de los 3 procesos, ya que cuando estos se ejecutan hay un momento en el que la computadora se confunde y ejecuta dos procesos al mismo tiempo como se muestra en las capturas de pantalla y viceversa.

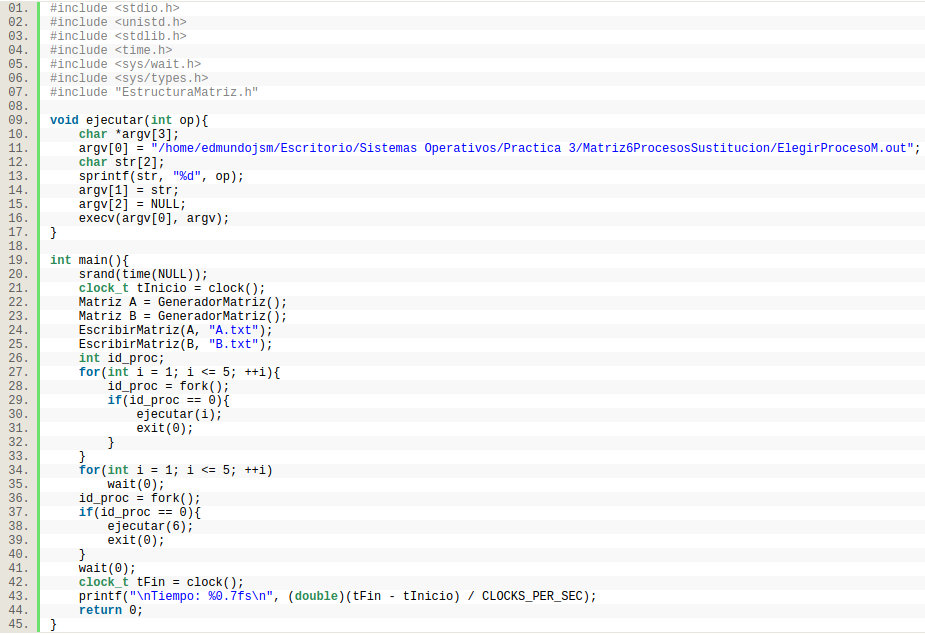
**2.1.7. Programa con 6 procesos por sustitución de código (Matrices)**

Reutilizaremos el código de EstructuraMatriz.h. Haremos lo mismo que en punto 5, pero ahora creando un proceso por sustitución por cada operación y otro para que muestre los resultados. Para lograrlo haremos otro programa llamado ElegirProcesoM.out que recibirá un parámetro del 1 al 6 indicando la acción a realizar. Las matrices ya deben de existir en los archivos A.txt y B.txt, pues este programa las leerá de ahí.

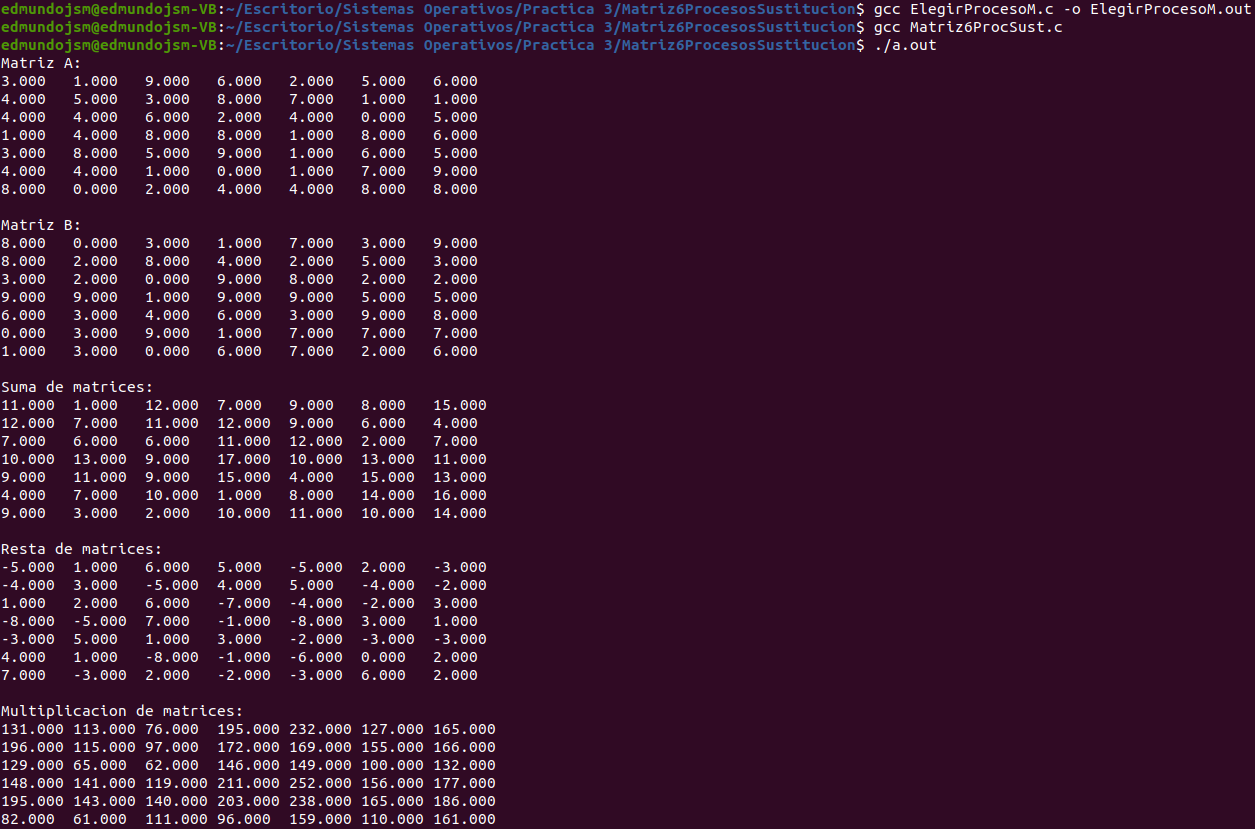
Finalmente, desde el programa principal crearemos 6 procesos por copia exacta y dentro de cada uno crearemos el proceso correspondiente por sustitución.

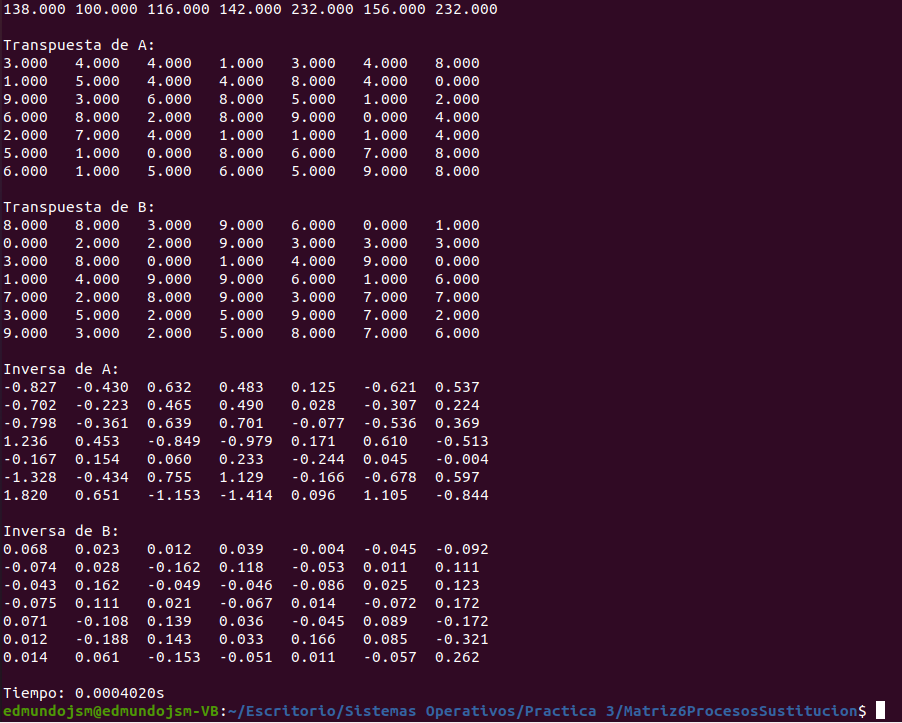
Código(ElegirProcesoM.c) compilado como ElegirProcesoM.out:

Código(Matriz6ProcSust.c)



Compilación y ejecución del código:



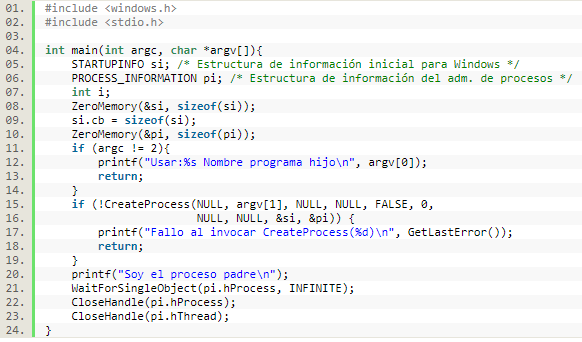


Vemos que el tiempo ahora fue de 0.0004s, superando a la aplicación de copia exacta, lo que quiere decir que lo mas conveniente sería ejecutar una aplicación mediante sustitución de código si es que queremos velocidad.

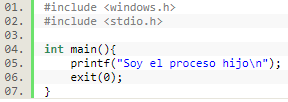
**2.2. Windows**

**2.2.1. Ejemplo de creación de procesos por sustitución de código**

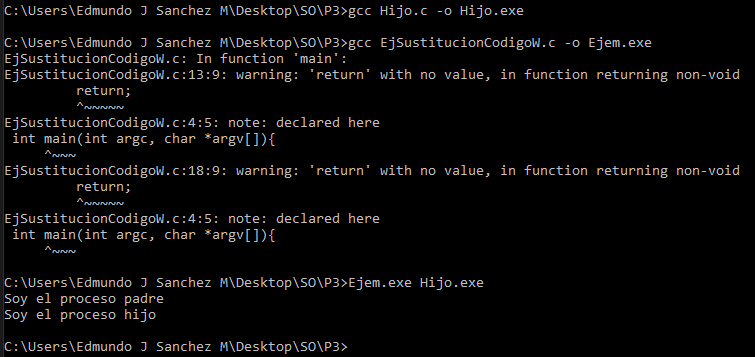
Código(EjSustitucionCodigoW.c)



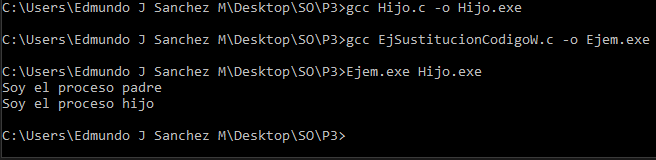
Código(Hijo.c)



Compilación y ejecución del programa EjSustitucionCodigoW pasando como parámetro el ejecutable del programa Hijo. Notar que se nos muestra advertencias de no tener valor de return, pero esto no afecta a la funcionalidad correcta del programa.



Ahora experimentemos con el programa, trataremos de quitarle los warnings que nos aparece dándole un valor los return de 0 y agregando un return 0 al final del código. Ahora veamos la compilación y ejecución del programa con ese cambio.



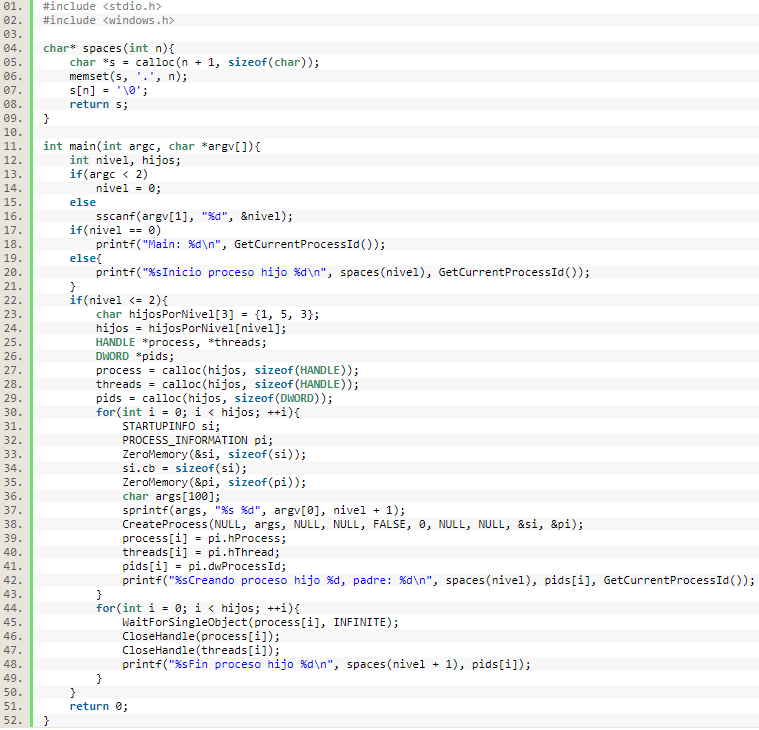
Vemos que no cambia en el funcionamiento, de hecho, logramos quitar las advertencias con lo que se mencionó anteriormente, así concluimos la experimentación de este código.

Ahora comparando con Linux vemos que la diferencia de un programa por sustitución de código en Linux con respecto a Windows se observa que en el programa principal necesita pasar como argumento el programa que ejecutara el proceso secundario (Hijo.exe), el cual a su vez es el argumento de la llamada al sistema. En este caso, si no le pasamos ningún argumento al programa Ejem.exe, nos mostrara un mensaje de la línea 12 en donde argv[0]=”Ejem.exe”.

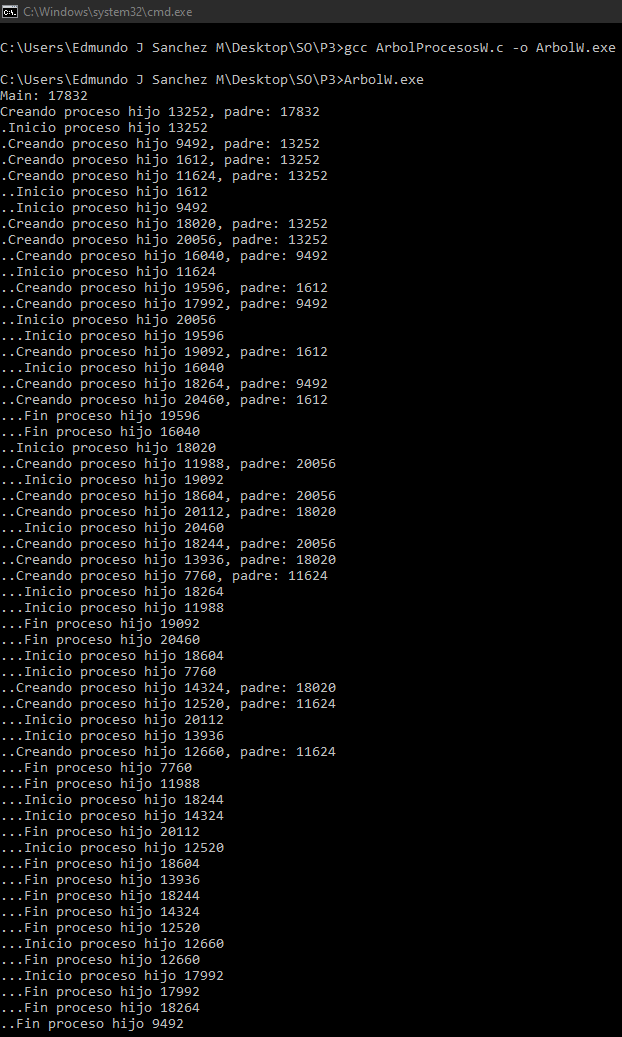
**2.2.2. Creación de árbol de procesos en Windows**

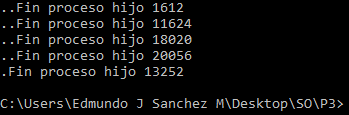
El siguiente programa recibe como argumento el nivel del árbol en el que estamos, para posteriormente revisar cuántos hijos deberá crear el programa, cada uno con el nivel incrementado en uno. También tenemos que guardar la información de cada proceso para esperar a que termine.

Código(ArbolProcesosW.c)

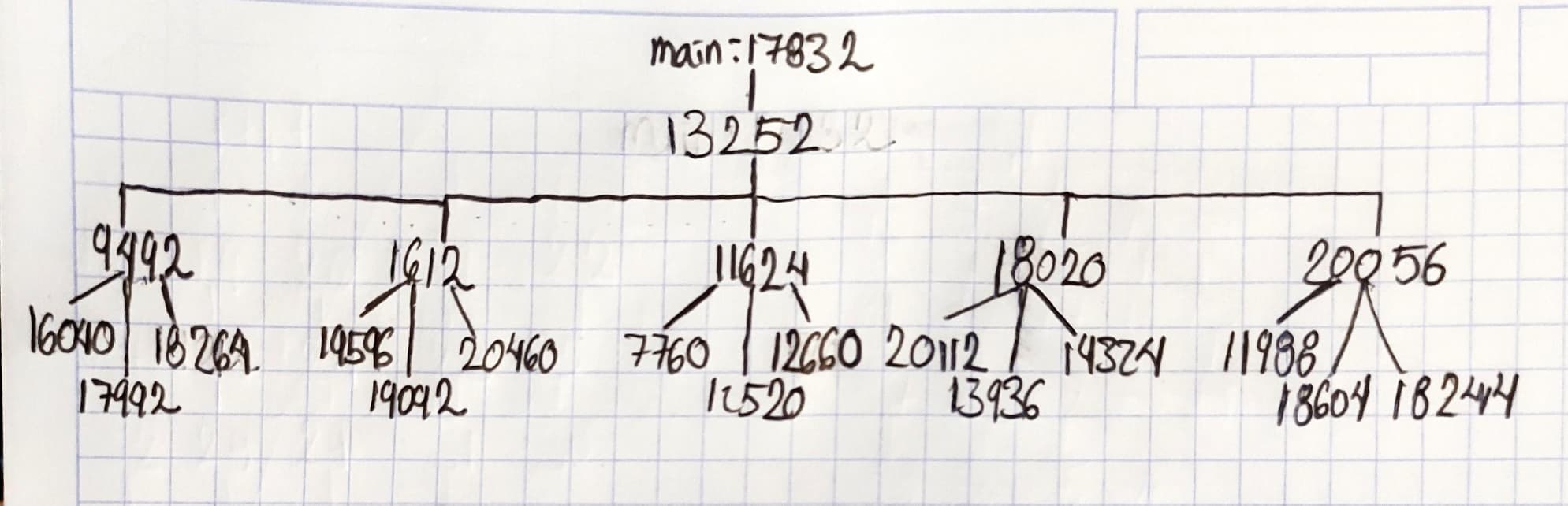


Resultado al momento de ejecutar y compilar el programa:





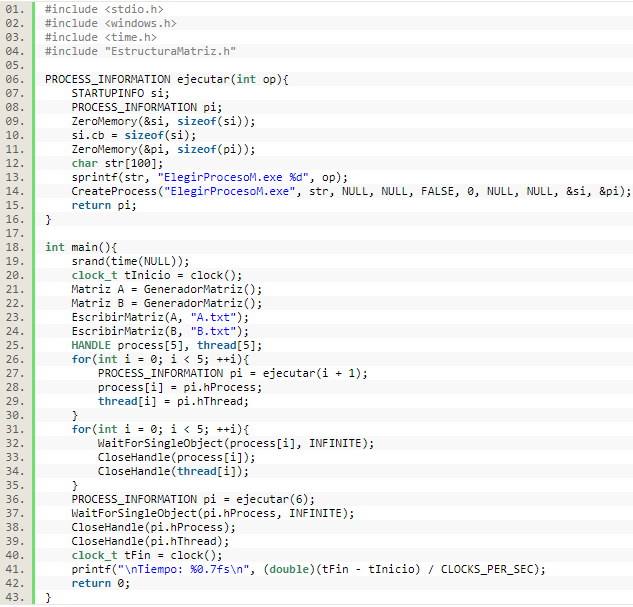
El árbol de procesos creado por el programa de acuerdo con los PID’s de cada uno es el siguiente:



El cual coincide con el árbol solicitado.

**2.2.3. Programa con 6 procesos por sustitución de código (Matrices)**

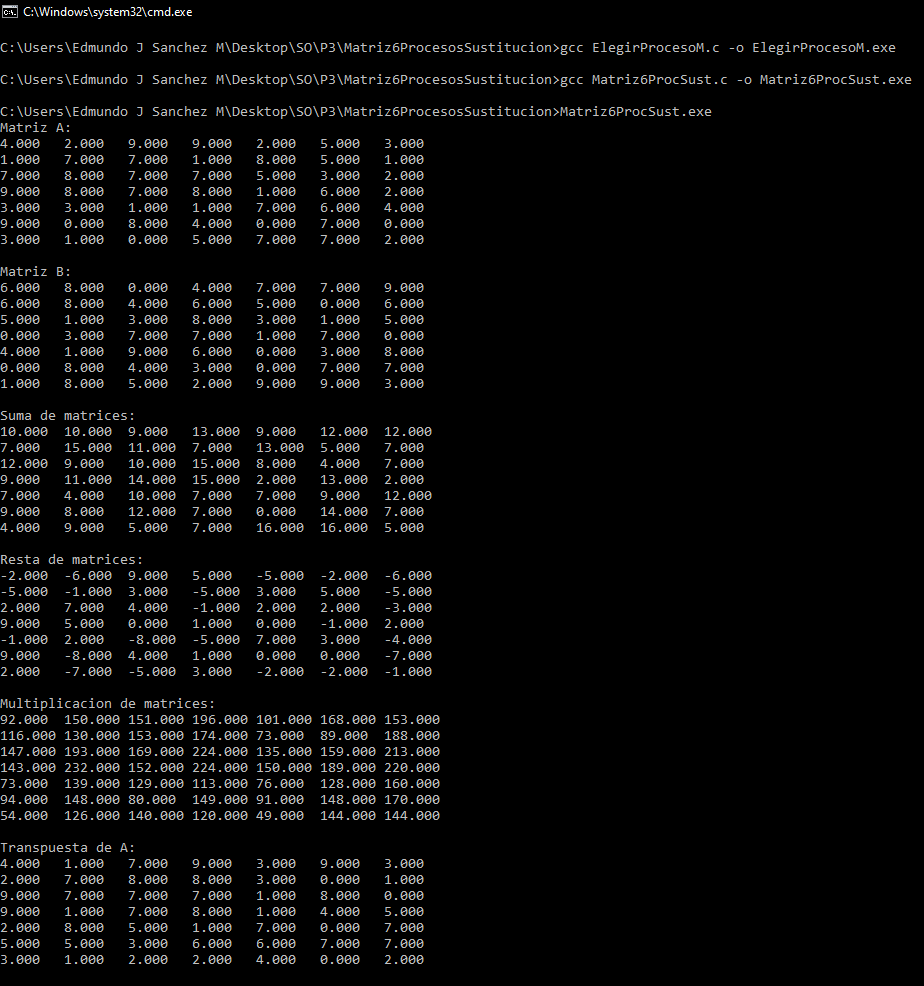
Usaremos de nuevo el código de EstructuraMatriz.h y ElegirProcesoM.c (ahora compilado como ElegirProcesoM.exe) del punto 8 de la sección de Linux, pues la funcionalidad lógica de la aplicación que queremos no cambia, ademas de usar OpSecuencial.c para ejecutar de forma secuencial el programa. Adaptaremos el código de la creación de los procesos en Windows, dado todo lo anterior solo se anexará el código en donde se ejecutarán los 6 procesos y el secuencial.Código(Matriz6ProcSust.c)

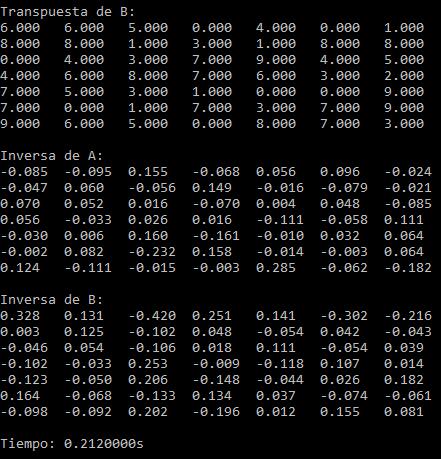


Código(OpSecuencial.c)

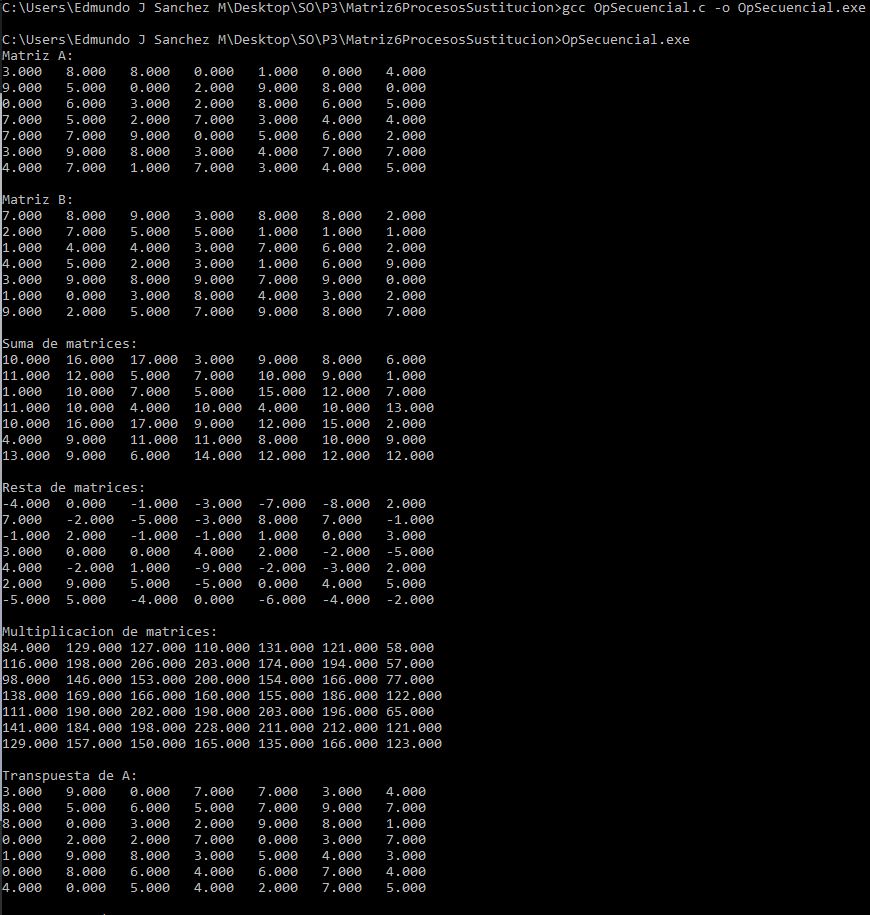


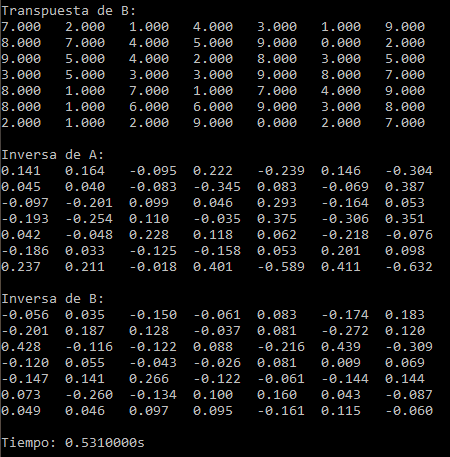
Compilación y ejecución del programa con 6 procesos:





Compilación y ejecución del programa de forma secuencial:





Vemos que el tiempo fue considerablemente mayor que en Linux, esto es debido a que Windows no les da la misma prioridad a los procesos que Linux, y a que su inicialización tarda más, ahora analizando ambos tipos de ejecución, tenemos el mismo resultado que el que obtuvimos de Linux, el cual es que la sustitución de código con procesos es más rápida que el secuencial.

**3. Análisis de la practica**

**3.1. Linux**

Con esta práctica pudimos observar el uso de procesos que se ejecutan de manera concurrente o también se pueden ejecutar un proceso tras otro. Se hizo uso de llamadas el sistema que controlaban los procesos, donde podemos observar las siguientes funciones:

* Crear un nuevo proceso.
* Carga de un proceso dentro de otro.
* Finalizar un proceso.
* Esperar a que termine un proceso.

Se debe tener cuidado como se invoca **fork()** para los procesos ya que se pueden varios procesos del mismo padre o el proceso padre genera un hijo y así sucesivamente los cuales finalizan con la llamada al sistema **exit(0)**. También al hacer uso de la llamada **execv(const char\* path, char\* const argv[])** se debe tener cuidado con la ruta que se manda, ya que se tiene que mandar la del archivo ejecutable y no la del código fuente, en caso de que se mande la del código fuente el valor de retorno de la llamada al sistema **execv** es -1 que significa que fallo la llamada al sistema.

**3.2. Windows**

La creación de procesos en Windows es notablemente más compleja: se necesita un conocimiento mínimo del uso de la función **CreateProcess**, y además los argumentos que se le mandan a la funcion. Esto es muy importante porque cualquier error en esta parte puede provocar un mal funcionamiento de la aplicación.

Un aspecto que hay que tener en cuenta es la similitud entre el primer y segundo argumento de la funcion **CreateProcess**, ya que pueden llegar a causar una confusión y estos dos argumentos, se podría decir, que son los más importantes de todos.

Otro aspecto importante es que por cada proceso que se invoque en el programa a su vez se deben de cerrar con la funcion **CloseHandle(\*especificación de proceso\*).**

**4. Observaciones**

**4.1. Linux**

* Al momento de llamar a **fork()**, se divide (de ahí el nombre de la llamada) en dos a partir de ese punto la ejecución del programa, la primera división es el padre cuyo valor de retorno es cero, y la otra división es el proceso hijo con valor de retorno positivo (el PID del hijo).
* Debido a lo anterior, hay que cuidar mucho en donde colocamos los **exit(0),** dependiendo el comportamiento que deseemos.
* La llamada **wait(status)** pone en espera el proceso actual hasta que alguno de sus hijos termine. Devuelve el PID del proceso que termino y copia en status el valor de retorno de dicho proceso. Si no había procesos hijo, se ignora. Por lo que, por ejemplo, si queremos crear cinco procesos hijo en la memoria, primero llamamos 5 veces a **fork()** y luego 5 veces a **wait()**, y no de forma intercalada.
* En el punto de la creación del árbol de procesos, nunca existió el 100 % del árbol completo en ningún momento, pues algunos procesos padre terminaban su ejecución mientras sus procesos hijo aun seguían. Nos dimos cuenta de esto por el orden de los eventos que se imprimían en las salidas del programa.
* Es un poco más rápido crear aplicaciones concurrentes usando copia exacta que hacerlo secuencialmente, pero todavía es más rápido usar procesos por sustitución, pues no tenemos que copiar todo el código actual, solo el del ejecutable que se necesita.
* Al hacer uso de la llamada **execv** en el parámetro de la ruta se tiene que mandar la ruta el archivo ejecutable, también se debe incluir en el primer elemento del arreglo **argv**.
* Si queremos llamar a otro proceso usando **execv** y que el proceso principal también siga corriendo, hay que hacerlo dentro de un proceso creado por copia exacta, porque de no hacerlo así el proceso original se destruye.
* Con la llamada **fork()** podemos crear a partir de un proceso padre más de un proceso hijo. Y con esto ejecutar concurrentemente varios procesos.
* Al crear procesos de manera concurrente, en algún momento los procesos se llegan a juntar, por lo que el sistema podría llegar a confundirse.
* Los PID’s asignados por Linux tienden a ser consecutivos.

**4.2. Windows**

* En Windows no es posible crear procesos por copia exacta de código.
* Cuando usemos la llamada **CreateProcess**, en el primer argumento usualmente va el nombre del ejecutable, aunque se puede omitir. El segundo argumento es una cadena en donde van todos los argumentos que le vamos a pasar al ejecutable, comenzando justamente con el nombre del ejecutable, separando con espacios los demás argumentos y usando comillas dobles en caso de que un argumento contenga espacios. Es decir, es como unir todos los elementos del arreglo **argv** que usamos en Linux separándolos por espacios.
* Al crear nuevos procesos, estos se crean como hilos, es decir, el programa original seguirá su ejecución normal a diferencia de Linux.
* En Windows también podemos esperar a que un proceso hijo termine usando la llamada **WaitForSingleObject**, pero aquí necesitaremos el ID del proceso, mientras que en Linux no.
* Windows nos obliga también a liberar los recursos del proceso hijo una vez que ya terminó su ejecución.
* La ejecución de procesos concurrentes es más desordenada que en Linux.
* Tanto en Linux como en Windows podemos usar rutas absolutas y relativas para hacer referencia a los ejecutables.
* Los PID’s asignados por Windows parecen ser asignados de forma aleatoria

**5. Conclusiones**

Tanto en Linux como en Windows se cuentan con distintas llamadas al sistema para la creación y manipulación de procesos, ambos sistemas crean, cargan, finalizan y esperan procesos. Estas llamadas al sistema normalmente devuelven cero si se ejecutaron con éxito o -1 en caso de error y su forma de invocación cambia en ambos sistemas operativos, sin embargo, es el mismo funcionamiento igual que en la práctica anterior.

Pudimos observar que las llamadas al sistema **getpid()** y **getppid()** simulan la misma función que el comando “**ps**” o “**ps -fea**”, en este caso son similares a una columna. Y a diferencia de los comandos se pueden ejecutar varias llamadas al sistema en el mismo código, no hay ninguna restricción respecto a sus veces de invocación.

La creación de procesos es notablemente más difícil en el sistema operativo Windows por todas las consideraciones antes descritas.

Hay funciones similares entre los dos sistemas operativos, una de ellas es **wait()** y **getpid()** en el sistema operativo Linux, y **WaitForSingleObject()** y **GetCurrentProcessId()** en el sistema operativo Windows.