PRÁCTICA 1

Sistemas Operativos

Roberto Loor.

A8590323

Ejercicio 1. Gestión básica de procesos. (4 puntos)

***a) Realiza un programa llamado malla.c que produzca el siguiente árbol de procesos. El programa recibirá dos argumentos ‘x’ e ‘y’ que representan el número de filas y columnas. Para comprobar la estructura de procesos que realmente estamos creando debemos emplear la llamada al sistema “pstree –c” (2 puntos).***

Para este ejercicio se utilizaron las siguiente librerías para los siguientes propósitos:

**#include <stdio.h>:** Es la librería estándar de E/S. En este programa se utiliza principalmente para imprimir mensajes en pantalla.

**#include <stdlib.h>:** Es la librería estándar de C. En este programa se utiliza principalmente para convertir cadenas a números.

**#include <unistd.h>:** Es la librería que contiene las funciones de llamadas al sistema. En este programa se utiliza principalmente para crear procesos.

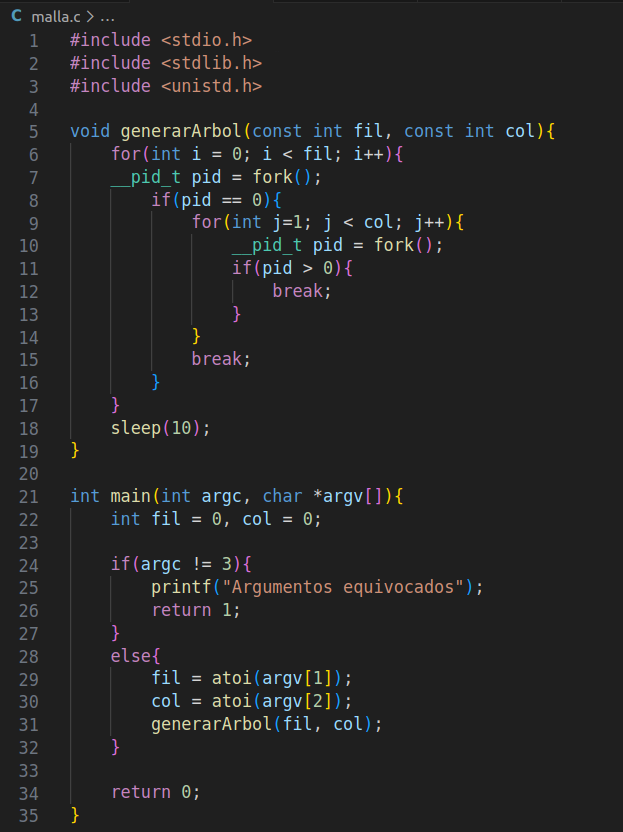
Para la ejecución del programa, se utilizaron los siguientes módulos:

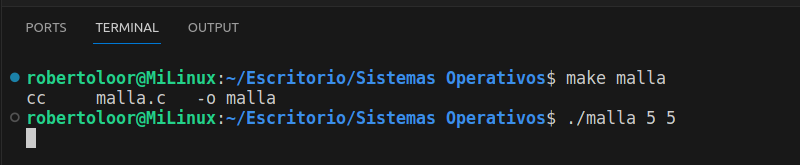
**void generarArbol(const int fil, const int col){:** Lo que hace este módulo es crear un árbol de procesos con una cantidad de filas y columnas dadas. La forma en la que lo hace es creando un proceso padre y luego dentro de este proceso padre se crean procesos hijos usando la función fork() y bucles for.

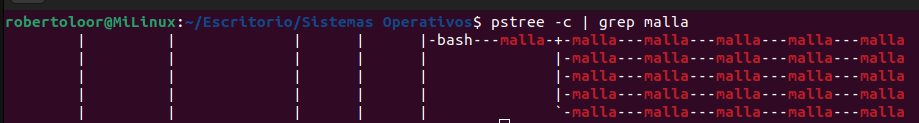
**int main(int argc, char \*argv[]){:** Esta función main recibe dos argumentos que son la cantidad de filas y columnas que tendrá el árbol de procesos, y luego llama a la función generarArbol.

La forma en la que se implementó este programa y su funcionamiento paso a paso es:

1. El programa espera dos argumentos de la línea de comandos: el número de filas (fil) y el número de columnas (col). Si no se proporcionan exactamente dos argumentos, el programa imprime un mensaje de error y termina.
2. Convierte los argumentos de la línea de comandos a enteros con la función atoi y los asigna a fil y col.
3. Llama a la función generarArbol con fil y col como argumentos.
4. Dentro de la función generarArbol, se realiza un bucle for que se repite fil veces. En cada iteración del bucle, se crea un nuevo proceso con la función fork.
5. Si fork devuelve 0, significa que estamos en el proceso hijo. En este caso, se realiza otro bucle for que se repite col - 1 veces. En cada iteración de este bucle, se crea otro nuevo proceso con fork.
6. Si fork devuelve un número mayor que 0, significa que estamos en el proceso padre y que se ha creado un proceso hijo. En este caso, se rompe el bucle for interno y se continúa con la siguiente iteración del bucle for externo.
7. Si fork devuelve 0 en el bucle for interno, significa que estamos en el proceso hijo. En este caso, se rompe el bucle for interno y se vuelve al bucle for externo.
8. Este proceso de creación de procesos se repite hasta que se han creado fil procesos, cada uno de los cuales ha creado col - 1 procesos hijos. Esto genera un árbol de procesos con fil como ramas, cada una de las cuales tiene col como niveles.
9. Después de crear todos los procesos, la función generarArbol pone a dormir a todos los procesos durante 10 segundos con la función sleep. Esto se hace para que tengas tiempo de ver la estructura del árbol de procesos con herramientas como pstree antes de que los procesos terminen.
10. Finalmente, el programa termina con éxito.







**b) Realiza un programa llamado ejec.c que reciba un argumento. El programa tendrá que generar el árbol de procesos que se indica y llevar a cabo la funcionalidad que se describe a continuación. El proceso Z, transcurridos los segundos indicados por el argumento, ejecutará el comando “pstree”. El proceso Z no puede utilizar el comando “sleep”, por lo que el proceso Z debe planificarse una alarma con los segundos indicados por el argumento. Se deberá controlar la correcta destrucción del árbol (los padres no pueden morir antes que los hijos). (2 puntos)**

Para este ejercicio se utilizaron las siguiente librerías para los siguientes propósitos:

**#include <stdio.h>:** Es la librería estándar de E/S. En este programa se utiliza principalmente para imprimir mensajes en pantalla.

**#include <unistd.h>:** Es la librería que contiene las funciones de llamadas al sistema. En este programa se utiliza principalmente para crear procesos.

**#include <stdlib.h>:** Es la librería estándar de C. En este programa se utiliza principalmente para convertir cadenas a números.

**#include <wait.h>:** Es la librería que contiene las funciones para esperar a que un proceso hijo termine. En este programa se utiliza principalmente para esperar a que los procesos hijos terminen.

**#include <signal.h>:** Es la librería que contiene las funciones para el manejo de señales. En este programa se utiliza principalmente para enviar señales a los procesos.

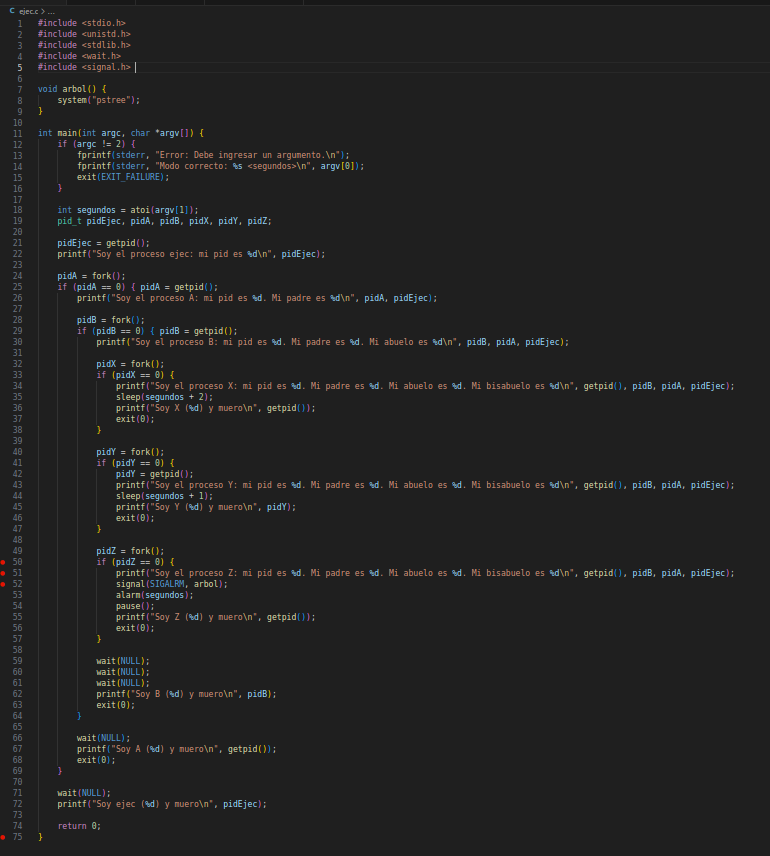
Para la ejecución del programa, se utilizaron los siguientes módulos:

**void arbol ():** Función que imprime el árbol de procesos.

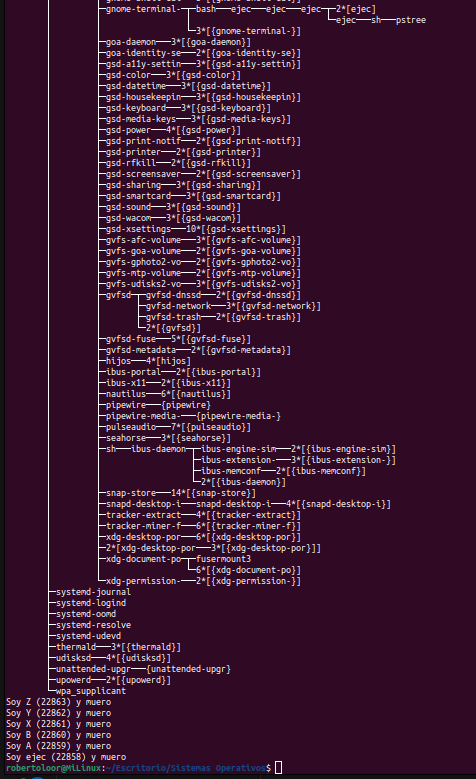
**int main (int agrc, char \* argv[]):** Función principal del programa, que se encarga de recibir los datos como argumentos, crear los procesos y mostrar el árbol de procesos.

La forma en la que se implementó este programa y su funcionamiento paso a paso es:

1. El programa espera un argumento de la línea de comandos: el número de segundos (segundos). Si no se proporciona exactamente un argumento, el programa imprime un mensaje de error y termina.
2. Convierte el argumento de la línea de comandos a un entero con la función atoi y lo asigna a segundos.
3. Obtiene el PID del proceso actual (ejec) con la función getpid y lo imprime.
4. Crea un nuevo proceso (A) con la función fork.
5. Si fork devuelve 0, estamos en el proceso hijo (A). En este caso, obtenemos el PID de A con getpid, imprimimos su PID y el PID de su padre (ejec), y creamos otro nuevo proceso (B) con fork.
6. Si fork devuelve 0 en el proceso B, estamos en el proceso hijo (B). En este caso, obtenemos el PID de B con getpid, imprimimos su PID y los PIDs de su padre (A) y su abuelo (ejec), y creamos tres nuevos procesos (X, Y, Z) con fork.
7. Si fork devuelve 0 en los procesos X, Y, Z, estamos en los procesos hijos. En estos casos, imprimimos sus PIDs y los PIDs de sus padres, abuelos y bisabuelos, y hacemos que cada proceso duerma durante un cierto número de segundos (segundos + 2 para X, segundos + 1 para Y, segundos para Z). Después de despertar, cada proceso imprime un mensaje y termina.
8. En el proceso Z, además de dormir, también configuramos una señal de alarma (SIGALRM) para que llame a la función arbol cuando se dispare. La función arbol ejecuta el comando pstree, que muestra la estructura del árbol de procesos.
9. En los procesos B y A, después de crear los procesos hijos, esperamos a que todos los hijos terminen con la función wait, imprimimos un mensaje y terminamos.
10. En el proceso ejec, después de crear el proceso hijo A, esperamos a que A termine con wait, imprimimos un mensaje y terminamos el programa con éxito.







Ejercicio 2. Comunicación entre procesos: tuberías. (3 puntos)

**Realizar un programa llamado copiar.c que permite copiar archivos. El programa recibirá dos argumentos: archivo\_origen y archivo\_destino, el archivo origen debe existir y el archivo destino se creará. El proceso copiar (proceso padre) generará un proceso hijo y, a partir de ese momento, el proceso padre será el encargado de leer el archivo origen y enviarle la información al proceso hijo que la almacenará en el archivo de destino. La comunicación entre los dos procesos se realizará mediante tuberías (pipe).**

Para este ejercicio se utilizaron las siguiente librerías para los siguientes propósitos:

**#include <stdio.h>:** Es la librería estándar de E/S. En este programa se utiliza principalmente para imprimir mensajes en pantalla.

**#include <stdlib.h>:** Es la librería estándar de C. En este programa se utiliza principalmente para convertir cadenas a números.

**#include <unistd.h>:** Es la librería que contiene las funciones de llamadas al sistema. En este programa se utiliza principalmente para crear procesos.

**#include <fcntl.h>:** Es la librería que contiene las funciones para el manejo de archivos. En este programa se utiliza principalmente para abrir archivos.

**#include <sys/types.h>:** Es la librería que contiene los tipos de datos básicos. En este programa se utiliza principalmente para definir el tipo de datos pid\_t.

**#include <sys/wait.h>:** Es la librería que contiene las funciones para esperar a que un proceso hijo termine. En este programa se utiliza principalmente para esperar a que los procesos hijos terminen.

Para la ejecución del programa, se utilizaron dos módulos:

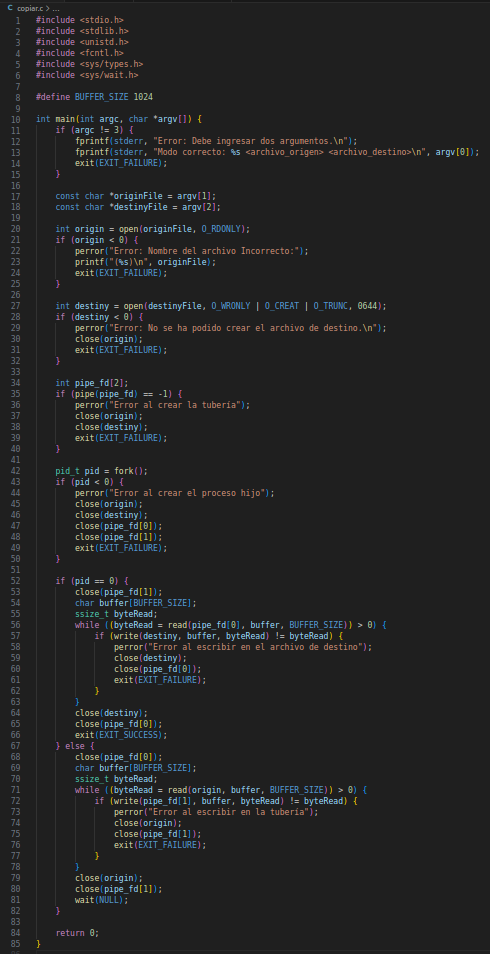
**int main (int argc, char \* argv []):** Función principal del programa. Se encarga de copiar el contenido de un archivo a otro.

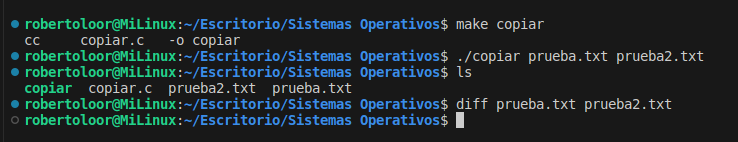
La forma en la que se implementó este programa y su funcionamiento paso a paso es:

1. El programa espera dos argumentos de la línea de comandos: el nombre del archivo de origen y el nombre del archivo de destino. Si no se proporcionan exactamente dos argumentos, el programa imprime un mensaje de error y termina.
2. Abre el archivo de origen para lectura con la función open. Si no puede abrir el archivo, imprime un mensaje de error y termina.
3. Abre el archivo de destino para escritura con la función open. Si el archivo no existe, lo crea. Si el archivo ya existe, borra su contenido. Si no puede abrir o crear el archivo, imprime un mensaje de error, cierra el archivo de origen y termina.
4. Crea una tubería con la función pipe. Si no puede crear la tubería, imprime un mensaje de error, cierra los archivos de origen y destino y termina.
5. Crea un nuevo proceso con la función fork. Si no puede crear el proceso, imprime un mensaje de error, cierra los archivos de origen y destino, cierra la tubería y termina.
6. En el proceso hijo cierra el extremo de escritura de la tubería, después, lee datos del extremo de lectura de la tubería y los escribe en el archivo de destino. Si encuentra un error al leer de la tubería o escribir en el archivo de destino, imprime un mensaje de error, cierra el archivo de destino y el extremo de lectura de la tubería, y termina. Por último, cuando ha terminado de copiar los datos, cierra el archivo de destino y el extremo de lectura de la tubería, y termina con éxito.
7. En el proceso padre cierra el extremo de lectura de la tubería, luego lee datos del archivo de origen y los escribe en el extremo de escritura de la tubería. Si encuentra un error al leer del archivo de origen o escribir en la tubería, imprime un mensaje de error, cierra el archivo de origen y el extremo de escritura de la tubería, y termina. Finalmente, cuando ha terminado de copiar los datos, cierra el archivo de origen y el extremo de escritura de la tubería.

Espera a que el proceso hijo termine con la función wait.

1. Finalmente, el programa termina con éxito.

****

****

Ejercicio 3. Comunicación entre procesos: memoria compartida. (3 puntos)

**Diseña un programa llamado hijos.c que cree un árbol de procesos según la siguiente estructura a partir de dos parámetros: $hijos x y.**

Para este ejercicio se utilizaron las siguiente librerías para los siguientes propósitos:

**#include <stdio.h>:** Es la librería estándar de E/S. En este programa se utiliza principalmente para imprimir mensajes en pantalla.

**#include <unistd.h>:** Es la librería que contiene las funciones de llamadas al sistema. En este programa se utiliza principalmente para crear procesos.

**#include <stdlib.h>:** Es la librería estándar de C. En este programa se utiliza principalmente para convertir cadenas a números.

**#include <wait.h>:** Es la librería que contiene las funciones para esperar a que un proceso hijo termine. En este programa se utiliza principalmente para esperar a que los procesos hijos terminen.

**#include <sys/shm.h>:** Es la librería que contiene las funciones para el manejo de memoria compartida. En este programa se utiliza principalmente para crear un segmento de memoria compartida.

**#include <sys/ipc.h>:** Es la librería que contiene las funciones para el manejo de claves IPC. En este programa se utiliza principalmente para generar una clave IPC.

Para la ejecución del programa, se utilizaron dos módulos:

**int main (int argc, char \* argv []):** La función principal del programa recibe dos argumentos, la altura y ancho de los hijos. Se encarga de crear los procesos hijos y de imprimir los padres de cada proceso hijo.

La forma en la que se implementó este programa y su funcionamiento paso a paso es:

1. El programa espera dos argumentos de la línea de comandos: x e y. Si no se proporcionan exactamente dos argumentos, el programa imprime un mensaje de error y termina.
2. Convierte los argumentos de la línea de comandos a enteros con la función atoi y los asigna a x e y.
3. Crea un segmento de memoria compartida con shmget y lo asigna a pids con shmat. El primer elemento de pids se establece en el PID del proceso actual.
4. Crea x procesos hijos con un bucle for y fork. Si fork devuelve 0, estamos en el proceso hijo. En este caso, asignamos el segmento de memoria compartida a pids y establecemos el siguiente elemento de pids en el PID del proceso hijo.
5. Si fork devuelve un número mayor que 0, estamos en el proceso padre. En este caso, esperamos a que todos los procesos hijos terminen con wait.
6. Si el PID del proceso actual es igual al primer elemento de pids, estamos en el proceso padre original (el "superpadre"). En este caso, imprimimos el PID del superpadre y los PIDs de sus "hijos finales" (los procesos que no tienen hijos), desasociamos el segmento de memoria compartida con shmdt y terminamos el proceso.
7. Si fork devuelve 0 en el paso 4, estamos en un proceso hijo. En este caso, creamos y procesos hijos con un bucle for y fork. Si fork devuelve 0, estamos en el proceso nieto. En este caso, asignamos el segmento de memoria compartida a pids, establecemos el siguiente elemento de pids en el PID del proceso nieto y terminamos el bucle for.
8. Si fork devuelve un número mayor que 0 en el paso 7, estamos en el proceso hijo que ha creado procesos nietos. En este caso, esperamos a que todos los procesos nietos terminen con wait, desasociamos el segmento de memoria compartida con shmdt y terminamos el proceso.
9. Si fork devuelve 0 en el paso 7, estamos en un proceso nieto. En este caso, imprimimos el PID del proceso nieto y los PIDs de sus antecesores, desasociamos el segmento de memoria compartida con shmdt y terminamos el proceso.
10. Finalmente, eliminamos el segmento de memoria compartida con shmctl y terminamos el programa con éxito.

