INSTITUTO POLITÉCNICO NACIONAL.
ESCUELA SUPERIOR DE CÓMPUTO.

SISTEMAS OPERATIVOS.

PRÁCTICA 3

CREACIÓN DE PROCESOS.

Integrantes del equipo:

- Chavarría Vázquez Luis Enrique.
- Juárez Espinoza Ulises.
- Machorro Vences Ricardo Alberto.
- Pastrana Torres Víctor Norberto.

Índice de contenido.

Índice de figuras

Glosario de términos.

Procesos.
No es más que un programa o comando en ejecución. Ahora vale la pena ahondar en algunas características.

- Un proceso consta de código, datos y pila.
- Los procesos existen en una jerarquía de árbol (varios Hijos, un sólo padre).
- El sistema asigna un identificador de proceso (PID) único al iniciar el proceso.
- El planificador de tareas asigna un tiempo compartido para el proceso según su prioridad (sólo root puede cambiar prioridades).

Ejecución en 1er plano.
Proceso iniciado por el usuario o interactivo.

Ejecución en 2o plano.
Proceso no interactivo que no necesita ser iniciado por el usuario.

Fork.
Fork crea un proceso hijo que difiere de su proceso padre sólo en su PID y PPID, y en el hecho de que el uso de recursos esté asignado a 0. Los candados de fichero (file locks) y las señales pendientes no se heredan.

Exec.
Exec abrirá, cerrará y podrá o no copiar los descriptores de archivo según lo especificado por cualquier redirecciones como parte del comando.

Proceso zombi.
Proceso parado que queda en la tabla de procesos hasta que termine su padre. Este hecho se produce cuando el proceso padre no recoge el código de salida del proceso hijo.

Proceso huérfano.
Proceso en ejecución cuyo padre ha finalizado.

Contenido (Investigación)

Consideraciones.

Un proceso es un programa en ejecución que puede ejecutarse concurrentemente con otros procesos. El sistema operativo Unix es un sistema multitarea y multiusuario que permite a los usuarios crear varios procesos simultáneos, los cuales pueden sincronizarse, comunicarse y compartir información. [1]

La única forma que tiene el sistema para identificar un proceso es mediante un identificador de proceso (PID), que es único en el sistema. Asociado con cada proceso se encuentra el contexto del proceso.

Contexto de proceso.

Está integrado por el contenido de su espacio de direccionamiento, el contenido dentro de los registros hardware y las estructuras de datos del kernel asociadas con el proceso. Basicamente, el contexto de un proceso es la unión del contexto a nivel de usuario, el de registro y el de sistema. [2]

Contexto de usuarios.

A nivel de los usuarios, este consta del código, los datos y la pila, así como de la memoria compartida que pueda tener dentro de sí. [2]

Contexto de registro.

En el contexto de registro tenemos el contador de programa, el registro de estado del procesador, los punteros de la pila y los registros de propósito general integrados en el sistema. [2]

Contexto bajo el nivel del sistema.

El contexto a nivel de sistema consta de la entrada en la tabla de procesos, la pila del kernel, etc. [2]

Creación de procesos.

Para crear un nuevo proceso en el S.O. Unix se utiliza la llamada al sistema fork. Esta llamada hace que el proceso que la ejecuta se divida en dos procesos.

Al proceso que ejecuta fork se le conoce como proceso padre (parent process) y al nuevo proceso creado se le llama proceso hijo (child process).

Tras ejecutarse esta llamada al sistema, los dos procesos tendrán copias idénticas de su contexto a nivel de usuario. La única diferencia será que el valor entero que devuelve fork para el proceso padre es el PID del proceso hijo, mientras que para el proceso hijo es 0.

Tipos de procesos. [3]

Procesos en primer plano.

Estos se inicializan y controlan a través de una sesión de terminal. En otras palabras, tiene que haber un usuario conectado al sistema para iniciar dichos procesos.

Procesos en segundo plano.

Son procesos que no están conectados a una terminal; no esperan ninguna entrada del usuario.

Los procesos Daemons. [2]

Son tipos especiales de procesos en segundo plano que comienzan al inicio del sistema y continúan ejecutándose como un servicio, cabe aclarar que ellos no mueren. Se inician como tareas del sistema (se ejecutan como servicios), de forma espontánea. Sin embargo, un usuario puede controlarlos mediante el proceso de inicio ya generado.

Estado de los procesos. [2]

Ejecución.

Aquí se está ejecutando (es el proceso actual en el sistema) o está listo para ejecutarse (está esperando ser asignado a una de las CPU).

Esperando.

Nos remonta a cuando un proceso está esperando que ocurra un evento o un recurso del sistema.

Detenido.

Es cuando un proceso se ha detenido, normalmente al recibir una señal. Por ejemplo, un proceso que se está depurando.

Zombie.

Es cuando un proceso está muerto, se ha detenido, pero todavía tiene una entrada en la tabla de procesos.

¿Qué hace fork?

El fork () creará un nuevo proceso hijo. El proceso hijo creado es un proceso idéntico al padre, excepto que tiene un nuevo ID de proceso del sistema. El proceso se copia en la memoria de su proceso padre, luego el kernel asigna una nueva estructura de proceso. El valor de retorno de la función es el que discrimina los dos hilos de ejecución. La función fork devuelve un 0 en el proceso del hijo, mientras que el PID del proceso hijo se devuelve en el proceso del padre. [4]

Creación de procesos con fork().

La llamada al sistema fork () se usa para crear procesos. No toma argumentos y devuelve un ID de proceso. El propósito de fork () es crear un nuevo proceso, que se convierte en el proceso hijo de la persona que llama. Después de que se crea un nuevo proceso hijo, ambos procesos ejecutarán la siguiente instrucción después de la llamada al sistema fork(). [5]

- Si fork () devuelve un valor negativo, la creación de un proceso hijo no tuvo éxito.
- fork () devuelve un cero al proceso hijo recién creado.
- fork () devuelve un valor positivo, el ID de proceso del proceso hijo, al padre.

A continuación, presentamos un ejemplo básico, en donde se muestran los puntos anteriores. Nota: la explicación detallada comienza posterior a esta imagen (ilustración 1).

```c
#include  <stdio.h>
#include  <string.h>
#include  <sys/types.h>

#define   MAX_COUNT  200
#define   BUF_SIZE   100

void  main(void)
{
    pid_t  pid;
    int    i;
    char   buf[BUF_SIZE];

    fork();
    pid = getpid();
    for (i = 1; i ≤ MAX_COUNT; i++) {
        sprintf(buf, "Linea del pid %d, valor = %d\n", pid, i);
        write(1, buf, strlen(buf));
    }
}
```

Ilustración 1 Código de ejemplo para fork()

Para mostrarlo de manera más gráfica tenemos el siguiente diagrama (ilustración 2) en donde hacemos uso de la llamada al sistema fork(). [6]

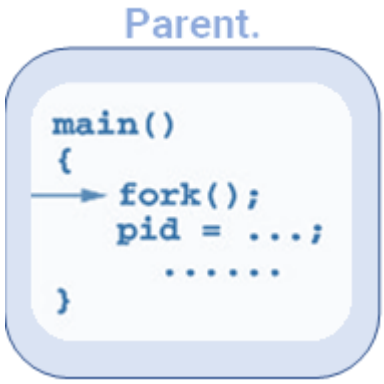

Ilustración 2 diagrama ejemplo para explicar el código superior.

Si la llamada a fork () se ejecuta correctamente, se harán dos copias idénticas de los espacios de direcciones, una para el padre y la otra para el hijo; pero ademas ambos procesos comenzarán su ejecución en la siguiente instrucción que sigue a la llamada a fork () . En este caso, ambos procesos comenzarán su ejecución en la declaración de asignación como se muestra a continuación en la siguiente imagen (ilustración 3).

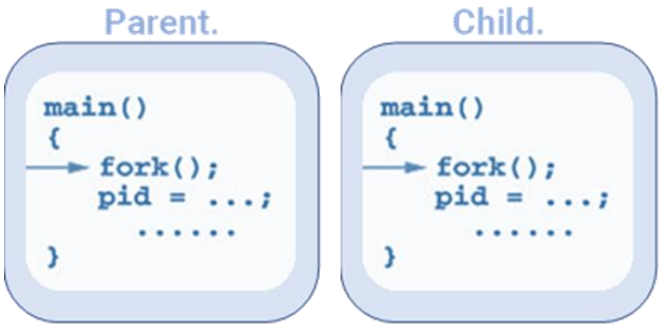

Ilustración 3 Diagrama ejemplo para explicar el código superior.

Ambos procesos comienzan su ejecución justo después de que el sistema llame a fork () . Dado que ambos procesos tienen espacios de direcciones idénticos pero separados, esas variables inicializadas antes de la llamada a fork () tienen los mismos valores en ambos espacios de direcciones.

Debido a que cada proceso tiene su propio espacio de direcciones, cualquier modificación será independiente de las demás. Si el padre cambia el valor de su variable, la modificación solo afectará a la variable en el espacio de direcciones del proceso padre. Otros espacios de direcciones creados por las llamadas a fork () no se verán afectados, aunque tengan nombres de variables totalmente idénticas.

Códigos y ventajas de ejecución.

Programa31.c

Realizar un programa que cree diez procesos hijos del mismo padre y cada uno muestre el mensaje “Hola soy el proceso con pid XXXX y mi padre es XXXX” y el conteo del uno al diez. Al final el padre espera a los hijos y termina.

Código programa31:

A continuación, podemos apreciar una ilustración (ilustración 4) absolutamente todo el código desplegado y en la parte inferior a la imagen usted puede encontrar la explicación del mismo de manera totalmente detallada.

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

int main(){

  for(int i=0;i<10;i++){
    pid_t pid =fork();
    if(pid==-1){
      perror("Error al crear el proceso");
      exit(-1);
    }
    if(pid==0){
      printf("Proceso %d -Hola soy el proceso con pid %d y mi padre es %d \n",(i+1),getpid(),getppid());
      sleep(5);
  exit(0);
      }
      wait(NULL);
  }

printf("Soy el proceso padre y tengo el pid %d \n",getpid());

  return 0;
}
```

Ilustración 4 Código de la práctica 31.c

Explicación código:

En el cómo se puede ver se emplea un ciclo for que crea procesos hijos que muestra el número del proceso creado en el ciclo for, su identificador de proceso y el identificador de su proceso padre, mientras el ciclo for espera a que los procesos terminen de crearse para hacer el incremento de la variable i por medio de función wait, para que al final pueda imprimir el identificador del proceso final. Todo esto se puede ver en la ilustración 5 donde todos los procesos numerados tienen el mismo padre, siendo esto el del programa principal.

Ejecución:

Cómo podemos apreciar en la imagen de abajo (ilustración 5), directamente en nuestra terminal del sistema operativo Ubuntu tenemos la ejecución de nuestro programa de manera exitosa, lo primero que hacemos es compilar nuestro programa y posteriormente ejecutar el código ya compilado, podemos ver como en efecto se han creado diez procesos, los cuales nos retornan no solamente un mensaje, sino que también nos devuelve los datos de vital importancia como el pid que cada uno de los procesos tiene y finalmente podemos ver cuál es el padre de dicho proceso, en este caso todos los procesos tienen un padre común por lo que todo los números coinciden.

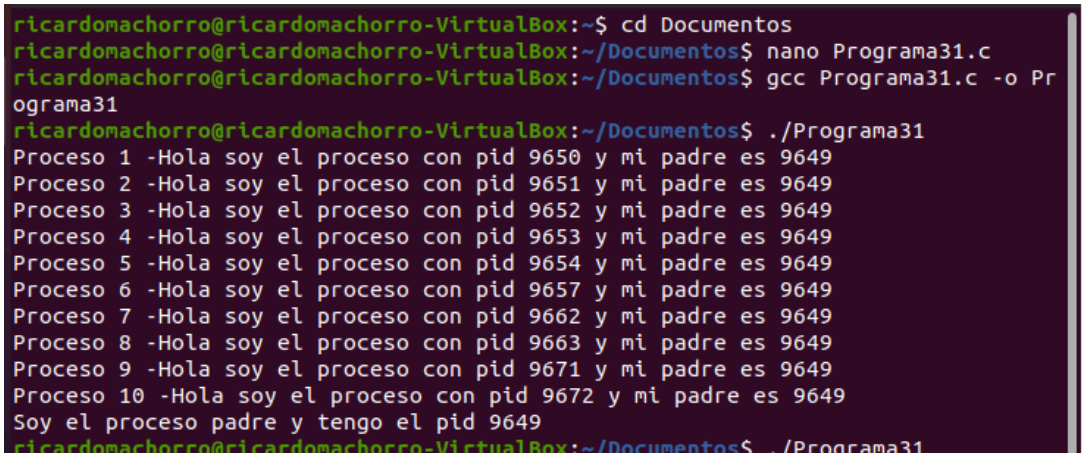

Ilustración 5 Ejecución del Programa31.c

Programa32.c

Realice un programa que cree un proceso hijo a partir de un proceso padre, el hijo creado a su vez creará tres procesos hijos más. A su vez cada uno de los tres procesos creará dos procesos más. Cada uno de los procesos creados imprimirá en pantalla el PID de su padre y su propio PID.

Código del programa32.

En esta sección ahora procedemos a, revisar el código de este segundo programa de la práctica número tres; es esencial recalcar que la explicación de todo el código se encuentre la parte de debajo de forma totalmente detallada, pero por cuestiones de practicidad se ha incluido el código en una imagen de forma totalmente completa en esta primera parte, como se puede apreciar en la ilustración número 6.

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

int main(){
   printf("Soy el proceso padre con un solo hijo, mi pid es:%d y mi ppid:%d \n",getpid(),getppid());
   pid_t pid=fork();
   if(pid==-1){
     perror("Error al crear el proceso con 3 hijos \n");
     exit(-1);
   }
   if(pid==0){
    printf("Soy el proceso padre con tres hijos, mi pid es:%d y mi ppdid:%d \n",getpid(),getppid());
      for(int i=0;i<3;i++){
            pid_t pidNivel2=fork();
           if(pidNivel2==-1){
               perror("Error al crear un proceso \n");
               exit(-1);
            }
            if(pidNivel2==0){
              printf("Soy el proceso %d  padre con dos hijos, mi pid %d y mi ppdid es:%d \n",(i+1),getpid(),getppid());
              for(int j=0;j<2;j++){
                  pid_t pidNivel3=fork();
                  if(pidNivel3==-1){
                    perror("Error al crear un proceso \n");
                    exit(-1);
                    }
                   if(pidNivel3==0){
                    printf("Soy el proceso %d con 0 hijos, mi pid %d y mi ppid es:%d \n",(j+1),getpid(),getppid());
                     sleep(1);
                     exit(0);
                     }
                  wait(NULL);
                }
               // printf("Soy el proceso %d  padre con dos hijos, mi pid %d y mi ppid:%d \n",(i+1),getpid(),getppid());
               sleep(1);
               exit(0);
             }
            wait(NULL);
         }
     //  printf("Soy el proceso padre con tres hijos, mi pid es:%d y mi ppdid:%d \n",getpid(),getppid());
        sleep(1);
        exit(0);
   }
    wait(NULL);
  //   printf("Soy el proceso padre con un solo hijo, mi pid es:%d y mi ppid:%d \n ",getpid(),getppid());

   return 0;
}
```

Ilustración 6 Código del programa 32.c

Explicación código:

Como se puede ver se emplea una creación anidada de diferente procesos hijos controlados por varios ciclos que mandan como mensaje el identificador el número de hijos que tienen, el número del mismo proceso al que pertenece (esto es si su padre tiene más de un hijo), su identificador de proceso y el identificador de su padre. El resultado se ve en la Ilustración 7 donde comienza el padre con un solo hijo enviando el mensaje, luego el padre con tres hijos, para luego cada uno de los tres hijos mostrar su menaje y el de sus dos hijos respectivamente en orden.

Ejecución:

Ahora bien, mostramos una imagen (ilustración 7) de la captura de la terminal en el proceso de ejecución del programa segundo de esta práctica número tres, ya una vez compilado se ejecuta y nos desplegará en pantalla el proceso que solamente tiene un hijo y nos devuelve en esta primera línea el pid y ppid, también podemos ver cómo podemos identificar el proceso padre que ya tiene tres hijos, ya posteriormente vemos que nos hace mención del proceso uno en el proceso dos con cada una de sus variantes y va imprimiendo los procesos en el momento en que tiene cada uno de ellos en lo hijos o cuando tienen dos hijos siempre estan incluyendo su identificador en cada una de las impresiones.

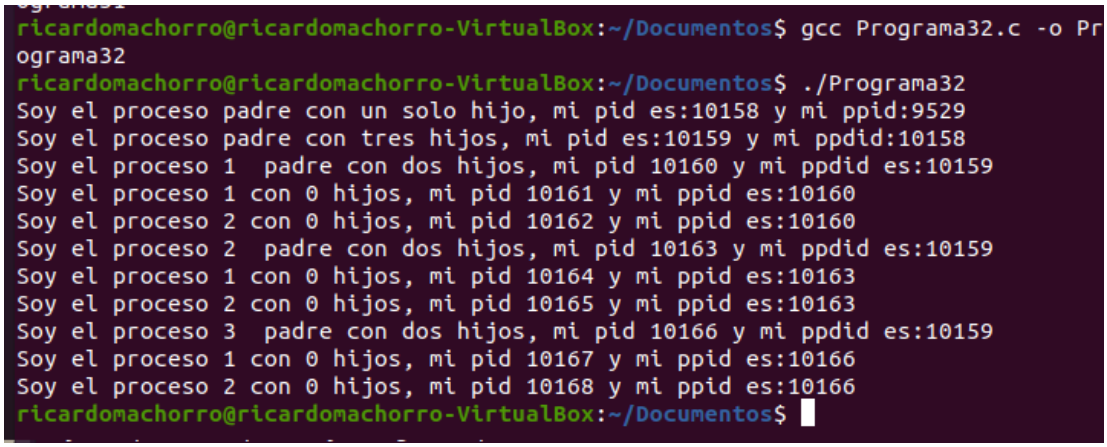

Ilustración 7 Ejecución Progrma32.c

Programa33.c

Realice un programa que cree cinco procesos. Cada uno de estos procesos escribirá en un archivo sólo una palabra de la siguiente oración: “Hola esta es mi práctica uno”. El proceso padre se encargará de escribir la última palabra (uno).

Código:

Del mismo modo que en los otros programas aquí también tenemos la imagen con todo el código, del mismo modo tenemos la explicación del código en la parte inferior junto con la muestra de cómo se ejecutó dicho programa y con los resultados que estamos esperando. Cabe aclarar que en la ilustración 8 tenemos el código desplegado.

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

int main(){

   const char *oracion[5]={"Hola ","esta ","es ","mi ","practica "};
   for(int i=0; i< 5;i++){
      pid_t pid=fork();
      if(pid==-1){
        perror("error creando proceso");
      }
      if(pid==0){
         FILE *fp;
          fp=fopen("Practica33.txt","a+");
          fputs(oracion[i],fp);
          fclose(fp);
//          sleep(1);
         exit(0);
      }
     wait(NULL);
   }

    FILE *fp=fopen("Practica33.txt","a+");
    fputs("uno",fp);
   fclose(fp);
  return 0;
}
```

Ilustración 8 Código del programa 33.c

Explicación código:

En el código solo se crea un array de longitud cinco con las primeras cinco palabras de la oración “Esta es mi practica uno” y luego se hace un ciclo for que crea cinco proceso , esperando que cada uno le agregue a un archivo existente o no llamado practica 33.txt una de la letra del array declarado con ayuda del índice del ciclo for. Para que al final agregue el proceso principal la palabra “uno”. Esto se puede ver donde en la Ilustración 9 antes no había tal documento, luego al ejecutar el código como se muestra en la Ilustración 10, se crea el documento con la frase completa como se ve en la Ilustración 11.

Ejecución:

Antes de ejecutarse.

Se mencionó anteriormente en la parte superior, que no es posible ver todavía los archivos y en la siguiente imagen (ilustración 9) como dicha afirmación se cumple de forma cabal.

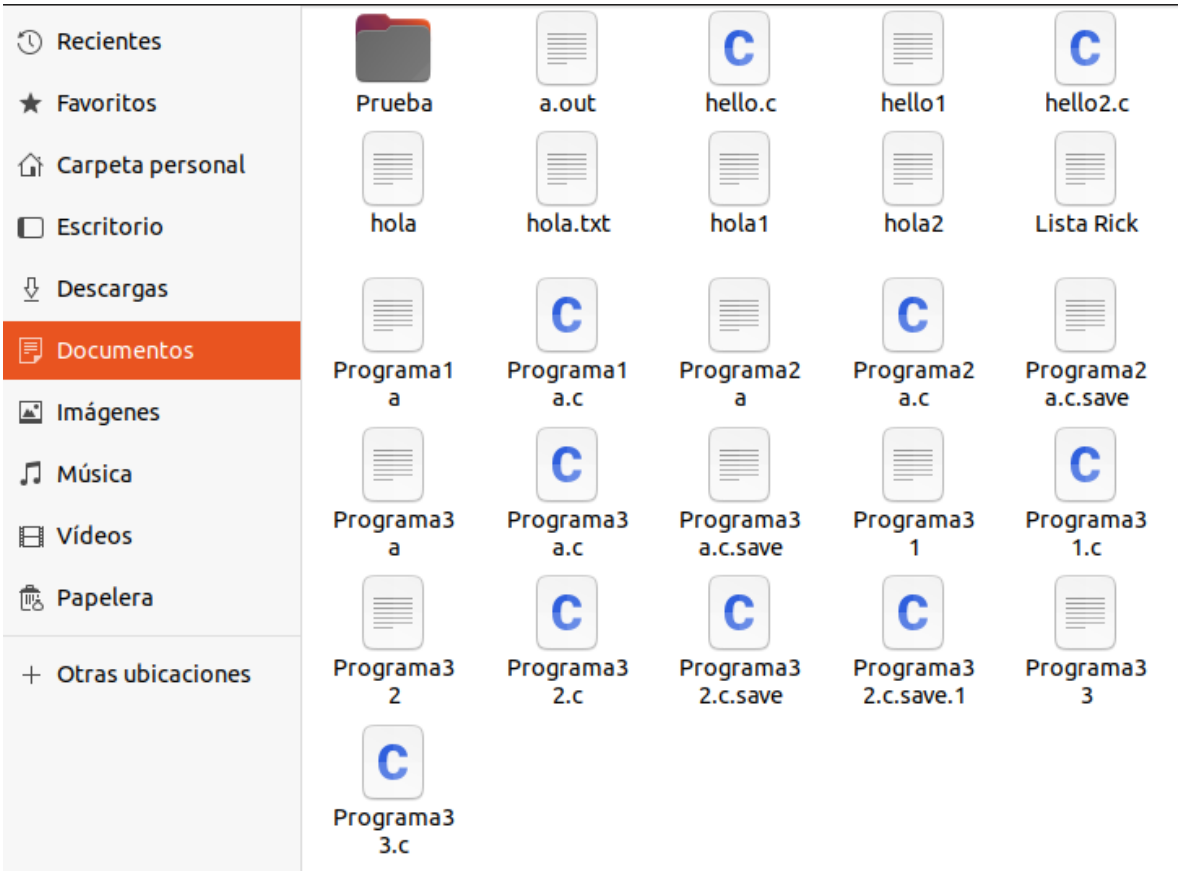

Ilustración 9 Documentos antes de la ejecución Programa33.c

Compilación y ejecución.

Ahora bien, procedemos compilar el programa dentro de nuestra terminal de Ubuntu y simplemente ejecutamos el programa; ahora bien, es recomendable ir directamente a dónde estan nuestros archivos guardados en la ruta que definimos para poder comprobar que en efecto el mensaje en el archivo de texto se encuentre creado de manera correcta, esto se aprecia de forma clara en la ilustración 10 y la 11.

```
ricardomachorro@ricardomachorro-VirtualBox:~/Documentos$ gcc Programa33.c -o Pr
ograma33
ricardomachorro@ricardomachorro-VirtualBox:~/Documentos$ ./Programa33
ricardomachorro@ricardomachorro-VirtualBox:~/Documentos$
```

Ilustración 10 Ejecución Programa33.c

Después de la ejecución.

Si bien, ya hemos mencionado como es que se ejecuta el programa y que eso que debía dar de resultado, pero para corroborarlo hemos ido directamente al directorio con la aplicación de gestor de carpetas y con el editor de texto o hemos abierto el archivo de texto generado con el mensaje que se esperaban un principio, lo cual se aprecia en la ilustración 11 de forma precisa.

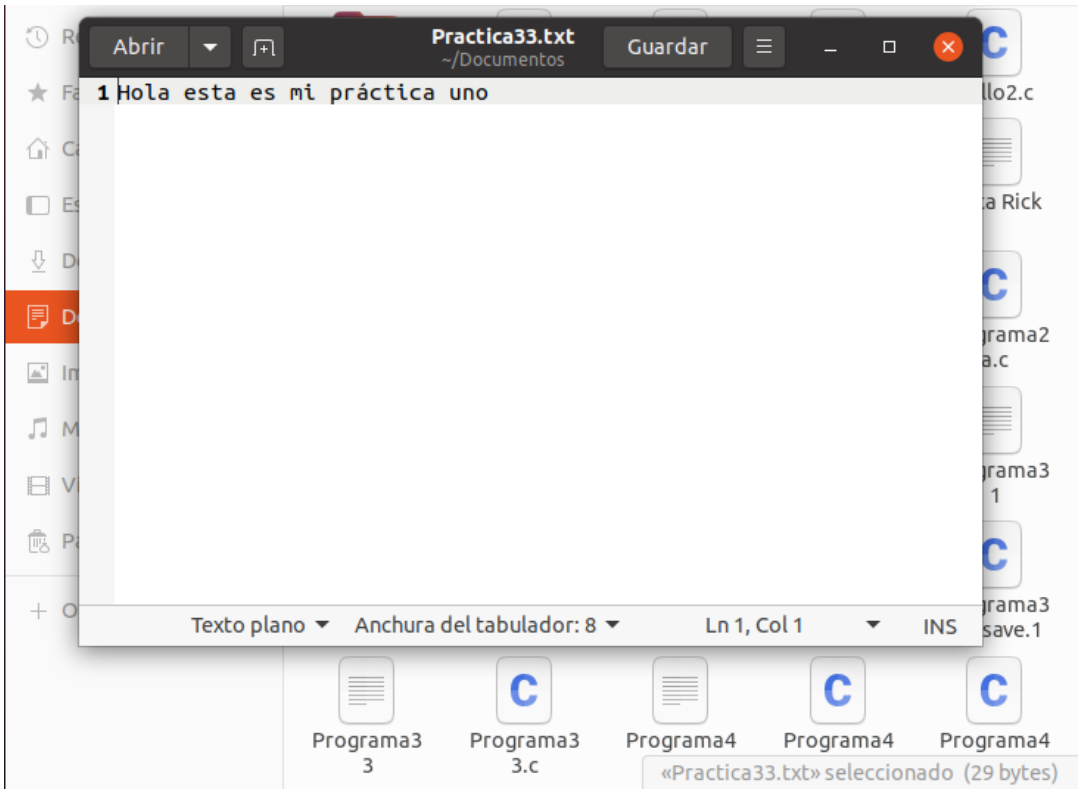

Ilustración 11 Documentos después de la ejecución del Programa33.c

Conclusiones.

Chavarría Vázquez Luis Enrique.

Durante el desarrollo de esta primera práctica de la unidad dos, desde mi perspectiva he logrado aprender cómo directamente en el código poder crear procesos para el cumplimiento de diversos menesteres, a decir verdad el tema para mí y para mis compañeros en el equipo parecía un poco intimidante en un comienzo, pero conforme intercambiamos ideas y también otros comentaban mucho más sobre la información referente a los procesos, nos dimos cuenta de que en realidad no era tan complejo y muchas veces las dificultades al igual que muchos de los conflictos provienen de fallos en la codificación, lo cual muchas veces genera los problemas en el código pero se puede ver tremendamente aminorado sí se entiende de manera concreta como es que los procesos funcionan y cómo es que estos se crean, por lo cual queda demostrado que vale la pena remontarse a la teoría primero antes de comenzar a codificar los distintos problemas.

Uno de los aportes más valiosos de esta práctica es que, en lo personal la parte de ver la forma en que los procesos mantienen las interacciones con el sistema operativo es bastante de genial porque da la impresión de que uno está directamente interactuando con el sistema operativo de primera mano, mientras estábamos codificando nos percatamos de esto o al tratar de gestionar de manera óptima el orden de los procesos y como los mismos procesos con los estamos trabajando como lo fue en el caso del segundo código creaban otros hijos a partir de un hijo ya concebido por un proceso padre codificado desde un comienzo. A decir verdad, la práctica me gustó mucho la sección de programa número treinta y tres, porque directamente estuvimos escribiendo en archivos, pero no solo eso sino que también estuvimos haciendo que de manera directa los procesos interactúan en distintas partes del resultado final. A mí me resulto muy llamativo porque me hace pensar en las múltiples posibilidades que esto tendría ya que en algún programa con muchísima complejidad podríamos optimizar nuestro código a manera de que todo este dividido de tal forma que múltiples procesos puedan ejecutar múltiples tareas dependiendo de la necesidad de nuestros clientes, lo cual puede resumirse en un posible ahorro de recursos e inclusive una mayor modularidad y nuestros programas y en las soluciones que demos a los diversos problemas que se nos presenten.

Siempre he dicho, que parte de disfrutar del proceso es aprender a disfrutar de los fallos que muchas veces se presentan, refiriéndome a este caso como equipo nos estuvimos apoyando bastante para poder solucionar dudas y hacer aportaciones en el código que estábamos desarrollando, con lo que toda la retroalimentación que recibimos de manera mutua nos permitió poder llegar a soluciones óptimas además de poder adquirir aprendizaje directamente de la experiencia, lo que en mi humilde opinión es mucho más valioso que simplemente la experiencia teórica, que aunque es de suma relevancia para poder desarrollar los problemas pienso que necesita complementarse con la parte práctica; con esto en mente y habiendo mencionado la importancia de ellos, me quedo con una satisfacción tremenda porque conseguir de forma certera y minimalista entender el funcionamiento de los procesos con un acercamiento diferente.

Juárez Espinoza Ulises.

Esta practica me ayudo a comprender como funciona la creación de procesos en Linux, que a primera instancia suena complicado, sin embargo, el sistema operativo lo hace todo el tiempo y con ayuda de lenguajes de programación flexibles como C, nosotros también podemos hacerlo con un código nada complejo y de manera muy rápida. A nivel teórico aprendí muchas cosas, como la jerarquía de procesos, específicamente la jerarquía de árbol que es como se maneja este sistema operativo, un padre varios hijos, además como el propio sistema asigna por defecto un identificador tanto a los procesos padre como hijo. Comprendí la importancia de finalizar correctamente los procesos que se crean par no tener un proceso zombie, a su vez identifiqué el estado en el que se encuentran los procesos en determinado momento. Entendí que los procesos tienen cierto tiempo de vida llegando incluso a morir, dejando asi procesos huérfanos momentáneamente.

A nivel de programación me sorprendió la facilidad con la que se crean procesos hijos, con el simple uso de la llamada al sistema fork, y que el proceso hijo creado sea una copia exacta del padre a excepción del PID y la memoria que ocupa, esta es de las cosas que más llama la atención ya que se presta a un gran sin fin de aplicaciones como por ejemplo: facilitar una copia de las variables del proceso padre y de los descriptores de fichero y lo mas importante a mi parece a que a pesar de ser una copia exacta son independientes, es decir si yo hago un cambio en una, no afectara a la otra esto le da mucha flexibilidad a los procesos.
Mencionaba que me sorprendió la facilidad de crear procesos, porque como se pudo ver en un programa, con un simple ciclo for o while y utilizando la llamada al sistema fork() se puede crear n cantidad de procesos hijos, y a su vez con la misma llamada al sistema fork() crear hijos de estos hijos, que heredan las variables del proceso padre de todos.

Al final fue interesante ver como se pueden relacionar estos procesos, mediante un archivo de texto, se observo que tanto procesos hijos como padre tuvieron acceso a el archivo y le añadieron texto para finalmente formar una oración producto de todos.

Quede muy satisfecho con las libertades que brinda el sistema operativo aunque también puede ser contraproducente para usuarios no tan experimentados, ya que podemos realizar cambios que no queremos, también que satisfecho con las bondades que brinda c para trabajar con procesos en este sistema operativo, desconocía eso y sin duda le da mucho valor al lenguaje en el manejo del sistema operativo, aunque no debería nada nuevo ni raro porque como se sabe UNIX fue rescrito a partir del lenguaje C, es aquí donde se comprende la facilidad con la que se comunican el lenguaje y el SO.

Machorro Vences Ricardo Alberto.

En esta práctica aprendí que a pesar de que al principio la creación de procesos suene algo difícil no lo es ya que para el lenguaje C se facilitan mucho al ser solo controlados con una simple función y variable, cuyas instrucciones pueden ser contenidas dentro de una sencilla secuencia if , donde además se puede considerar todo lo que está allí como un programa por separado, dejando libre las acciones que uno instruye sin tener verdaderas consecuencias que afecten a los demás, aunque se tomar en cuenta como instruir tan al proceso como el hijo cunado salir (morir) o esperar a uno respectivamente.

Algo que si tengo que admitir es que tuve cierto problema en ver el orden en que los procesos se creaban en ciclos como los son el while y el for porque en mis primeros intentos estos no iban en orden, siendo esto posible porque los trataba de crear fuera de estos, dejando al propio del sistema operativo ver cunado crea estos. Otro problema que también tuve fue ver en como otros procesos hijos crearían a otros hijos, pero no fue propiamente porque no tuviera idea de cómo hacerlo sino porque en lo largo y entre mezclado de los ciclos que creaban los procesos no sabía cuándo empezaba uno y terminaba otro, haciendo que algunos resultados no sucedieran como yo planeaba por que las variables no coincidían, pero afortunadamente por no ser practicas con acciones tan largas y que de hecho piden el identificador de los procesos fue fácil ver donde estaba el error.

En resumen, se puede decir que esta práctica me sirvió como una introducción de cómo se crean los procesos en lenguaje C, algunos de los problemas inesperados que se pueden encontrar como los que mencione anteriormente, detalles en los que uno de se tiene que fijar cuando trabajo como puede ser los tiempos de espera y de salida tanto para procesos padre como hijo respectivamente y algunos de los usos que estos pueden tener, pero en general lo yo pienso que esta práctica me ayudo a ver qué es lo que implica verdaderamente trabajar con procesos.

Pastrana Torres Victor Norberto.

De primera instancia este tema me pareció complicado porque en los cursos previos en los que he usado el lenguaje C, nunca antes había utilizado procesos, de hecho, ni siquiera sabía que existían, pero ahora, gracias al desarrollo de la practica he comprendido como administra una computadora las diferentes tareas que desarrolla y puedo decir que me parece sorprendente como es que esto se le pudo haber ocurrido a un persona o a un conjunto de personas.

Durante la realización de la practica nos enfrentamos a diferentes retos, uno de los que más me cuesta en lo personal, es acostumbrarme al nuevo entorno de trabajo, Linux, admito que es un gran sistema que tiene muchas ventajas ante el popular Windows, pero me sigue constando trabajo utilizar y sacarle provecho a la terminal, que es una herramienta muy útil en cualquier distribución Linux.

En lo que concierne a los temas aplicados a la práctica, de manera teórica los comprendí correctamente, aunque llevarlos a la ejecución fue distinto. El principal inconveniente que tuvimos fue que en nuestros primeros intentos desarrollando los programas, nosotros esperábamos que los procesos creados debían salir de manera consecutiva, pero gracias a las clases posteriores de dudas, entendimos que la forma en la que se ejecuten los procesos no es algo que este bajo control del programador, esto es meramente una tarea del sistema operativo y nosotros únicamente tenemos la posibilidad de creación de más procesos.

Esta práctica me gusto en lo personal porque tuve una directa interacción con el sistema operativo, literalmente yo le decía que hacer al sistema y podía ver con exactitud lo que le pedí, crear un proceso, crear procesos hijos, matar esos procesos o darle una función específica a cada uno, eso es algo que no muchas personas pueden hacer, en Windows por ejemplo creo que no es posible decirle al sistema que haga tal cosa, tan solo muchas de las variables que se usaron para el desarrollo de la practica únicamente están disponibles en el sistema Linux. Complementando esto mencionare que en la unidad de aprendizaje de Análisis de algoritmos también estoy utilizado una distribución de Linux porque solo en ese sistema es posible medir los tiempos de ejecución de un proceso. Puedo concluir que este tipo de actividades forzosamente necesitan de un entorno Linux para poder ejecutarse.

Bibliografía

[1] F. J. S. Castaño., «MEMORIA Gestión de procesos en los sistemas operativos.,» UOC, p. 72.

[2] H. J. O. Pabón, «Sistemas Operativos Modernos.,» Medellín, Sello, 2005.

[3] T. A. S., «Sistemas Operativos Modernos.,» 2da edición ed., México, Prentice Hall, 2003.

[4] UBUNTU, «UBUNTU manuals,» Ubuntu, 2020. [En línea]. Available: http://manpages.ubuntu.com/manpages/trusty/man2/fork.2.html. [Último acceso: 2 Octubre 2020].

[5] HIMANSHU ARORA, «THE GEEK STUFF,» THE GEEK STUFF, 2020. [En línea]. Available: https://www.thegeekstuff.com/2012/05/c-fork-function/#:~:text=The%20fork()%20function%20is,process%20becomes%20the%20child%20process.. [Último acceso: Octubre 2020].

[6] Air netlines, «linuxhint,» linuxhint, 2020. [En línea]. Available: https://linuxhint.com/c_fork_system_call/. [Último acceso: Octubre 2020].