UNIVERSIDAD NACIONAL DE SAN AGUSTIN

FACULTAD DE INGENIERÍA DE PRODUCCIÓN Y SERVICIOS

ESCUELA PROFESIONAL DE INGENIERÍA DE SISTEMA

R. JESÚS CÁRDENAS TALAVERA
Magister en Ciencias Informática con mención
En Tecnologías de Información

GUÍA DE LABORATORIO

SISTEMAS OPERATIVOS

SEMESTRE V

COMPETENCIAS

Comprender el funcionamiento interno de los sistemas operativos y el manejo de procesos pesados en el sistema
Utilizar, analizar y explicar con ejemplos el funcionamiento de los procesos del sistema.

Laboratorio

4

Comunicación entre procesos usando Lenguaje C - Linux

I

OBJETIVOS

- El estudiante comprenderá y analizará el funcionamiento interno de los sistemas operativos desde la utilización de comandos hasta la programación de procesos.
- El estudiante deberá de probar, analizar y explicar el comportamiento de los diferentes medios de comunicación que existen entre procesos para un intercambio de datos de forma segura.

II

TEMAS A TRATAR

- Señales
- Pipes de comunicación

III

MARCO TEÓRICO

1. Señales

Las señales son mecanismos de comunicación entre procesos en Linux.

Una señal es un mensaje especial enviado a un proceso. Las señales son asíncronas; cuando un proceso las recibe, debe procesarlas inmediatamente, así no termine la función o línea de código actual.

Por ejemplo, cada vez que hacemos **Ctrl+C** o **Ctrl+D** durante la ejecución de un proceso en el shell estamos invocando a una señal (de terminación).

Existe una docena de señales, cada una con diferente significado. Pueden ser identificadas por su número o por su nombre.

Cuando un proceso recibe una señal, su comportamiento depende de su disposición. Cada señal tiene una disposición por defecto, la cual determina qué pasa si el programa no indica algún comportamiento: la señal puede ser **ignorada** o **enmascarada**. Enmascarar o manejar una señal consiste en especificar un comportamiento explícito mediante alguna función manejadora (*signal handler*).

Un uso de las señales es para mandar algún comando a un proceso. Para este propósito existen dos señales reservadas para el usuario: **SIGUSR1** y **SIGUSR2**.

Para indicar la disposición hacia una señal hacemos uso de la función **sigaction**:

```
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
```

El primer argumento es el número o nombre de la señal. El segundo es un puntero a una estructura **sigaction**, el cual indica la disposición deseada para la señal indicada. El tercer argumento indica la disposición anterior.

El campo más importante de la estructura **sigaction** es **sa_handler**. Puede tener uno de estos tres valores:

- **SIG_DFL**, que especifica la disposición por defecto
- **SIG_IGN**, que especifica que debería ser ignorada
- Un puntero a una función manejadora. La función debe tener un solo argumento, el número de señal y retornar **void**.

Como las señales son asíncronas, el programa principal puede estar en un estado frágil cuando la señal es procesada. Por esto, no se recomiendo manejar operaciones de E/S desde un manejador. Asimismo, un manejador debería realizar el trabajo mínimo requerido para responder, y luego retornar el control al programa. Muchas veces esto consiste en simplemente registrar que se dio la señal. Luego, nuestro programa revisaría periódicamente este registro para actuar apropiadamente.

El mismo hecho de modificar una variable global es peligroso, ya que durante esta modificación otra señal podría intentar modificarla al mismo tiempo, dejando a la variable en un estado corrupto. Una manera de evitar esto es usar el tipo especial **sig_atomic_t** que está diseñado para que Linux siempre realice cualquier asignación en **una sola** operación.

Enviar una señal a un proceso

1.1. Usando el comando kill

Para esto se requiere saber el número de la señal y el PID del proceso al que se desea enviar la señal

```
kill <signal> <PID>
```

```
$ kill -l
```

Nos permite ver las señales con sus respectivos números

```
$pgrep proName
```

Devuelve información del proceso denominado procName

Terminación de procesos

Si queremos terminar a nuestro programa (pid = 1234)podemos usar **kill** sin ningún argumento:

```
$kill 1234
```

Lo cual envía la señal **SIGTERM**. Otra manera de terminarlo es usar la señal **SIGKILL**:

```
$kill -SIGKILL 1234
```

¿Cuál es la diferencia? SIGTERM le pide al proceso que termine; esta petición puede ser ignorada. SIGKILL siempre mata al proceso; es una de las señales no manejables.

1.2. A través de otro proceso

Para enviar señales a un proceso dentro de un programa C se emplea la siguiente llamada:

```
int kill(int pid, int sig);
```

donde:

- El PID (process ID, identificador de proceso) del proceso que tiene que ser "señalizado".
- La señal que le enviaremos al proceso.

kill devuelve 0 si todo ha ido bien, -1 si ha ocurrido algún error, se asignará el valor apropiado a **errno**

Alarmas

Una alarma es una señal que es activada por los temporizadores del sistema a través de SIGALRM

2. Pipes

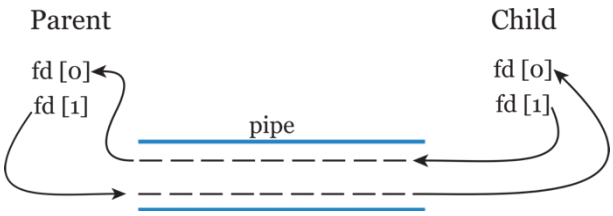
Cualquier proceso en Linux tiene la posibilidad de establecer un medio de comunicación con otro procesos. Estos modelos de comunicación pueden ser por memoria compartida o por envió de mensajes. Un mecanismo de comunicación que el sistema Linux pone a disposición son los ***pipes***.

Los *pipes*, generados mediante la función *pipe()*, abre una "tubería" (o canal) de comunicación y nos devuelve dos descriptores de fichero abiertos, uno por cada extremo de la tubería. Por un extremo de ellos se puede leer datos (con la función r*ead()*) que han sido escritos por el segundo proceso (con el uso de la función *write()*).

La ventaja de este mecanismo, es que si creamos la tubería antes de crear el proceso hijo (antes de llamar a la función *fork()*), el proceso hijo realizará una copia de todo el contenido del proceso padre, incluyendo también una copia de la tubería. Por lo tanto, el proceso padre como el proceso hijo tendrán dos descriptores de la tubería abiertos y utilizables. En caso

de haber ocurrido un error en la invocación de la función *pipe()*, esta retornará el valor de -1.

```
int descriptorTuberia[2];
int pid;

pipe(descriptorTuberia);

pid = fork();
```



No es recomendable que ambos procesos intenten escribir en la misma tubería, al menos salvo que posean mecanismos de sincronización. De ocurrir esta situación, los datos podrían mezclarse y resultarían en datos erróneos. En el caso de la lectura, si el proceso que ha escrito un dato intenta leer la tubería antes que el proceso hijo, recogerá su propio mensaje. Si ambos intentan leer a la vez, recogerán cada uno trozos del mensaje. Por ello, es mejor utilizar la tubería para que un proceso solo envíe datos y sea el otro sea el único encargado de recibirlos. Si queremos una comunicación bidireccional, es mejor abrir dos tuberías (llamar dos veces a la función *pipe()*).

Para evitar equivocaciones, y sobre todo para no mantener recursos ocupados innecesariamente, el proceso padre debe cerrar el lado de la tubería que no vaya a utilizar. El proceso hijo tiene su propia copia de ese lado, así que aunque el proceso padre lo cierre, para él sigue abierto y utilizable. El proceso hijo debe de cerrar el otro extremo.

En el siguiente ejemplo cerraremos el descriptor de lectura en el proceso padre y el de escritura en el proceso hijo. Una vez hecho esto, el padre pude escribir por *descriptorTuberia[1]* y el hijo leer por *descriptorEscritura[0]*, como si se tratara de un fichero normal. Obviamente ambos procesos deben estar de acuerdo en qué estructuras de datos se van a enviar.

```
//Proceso Padre
close(descriptorTuberia[0]);
...
write(descriptorTuberia[1], buffer, length);
...


//Proceso Hijo
close(descriptorTuberia[1]);
...
read(descriptorTuberia[0], buffer, length);
...
```

En el caso del proceso hijo, con la función *read()*, lo dejara bloqueado hasta que el proceso padre envié su dato. Ademas, En caso se este listo a leer los datos con la función *read()*, debemos de asegurar mediante un bucle de lectura completa de todos los datos enviados, ya que podemos obtener una cantidad parcial de la información enviada.

IV

ACTIVIDADES

1. Analice y describa la actividad que realiza el siguiente código. Explique como sucede el proceso de comunicación con el programa en ejecución.

```
//sigusr1.c
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>

sig_atomic_t sigusr1_contador = 0;

void manejador(int nro_senal) {
        ++sigusr1_contador;
        printf("SIGUSR1 se dio %d veces\n", sigusr1_contador);
}

int main() {
        struct sigaction sa;
        memset(&sa, 0, sizeof(sa));
        sa.sa_handler = &manejador;
        sigaction(SIGUSR1, &sa, NULL);

        //por el bucle infinito el programa debe ser
        //abortado mediante SIGKILL o SIGTERM
        while(1);

        return 0;
}
```

   Para esto, desde otro terminal envie a ese proceso la señal SIGUSR1

2. Explique cómo es que sucede el proceso de comunicación con el siguiente programa en ejecución a través de la señal CTRL+C y la señal **alarm**. Verifique los resultados.

```
#include <signal.h>
#include <stdlib.h>
```

```
#include <stdio.h>

int ncortes=0;
int bucle=1;
void alarma();
void cortar();

int main() {
        signal (SIGINT, cortar);
        signal (SIGALRM, alarma);

        printf (“Ejemplo de signal.\n”);
        printf (“Pulsa varias veces CTRL C durante 15 segundo.\n”);
        alarm (15);
        while (bucle);
        signal (SIGINT,SIG_IGN);
        printf (“Has intentado cortar %d veces.\n”,ncortes);
        printf (“Hasta luego…..\n”);
        exit(0);
}

void cortar( ) {
      signal (SIGINT, SIG_IGN);
      printf ("Has pulsado CTRL C \n");
      ncortes++;
      signal (SIGINT, cortar);
}

void alarma ( ) {
      signal (SIGALRM, SIG_IGN);
      bucle = 0;
      printf("!Alarma! \n");
}
```

3. Explique cómo es que sucede el proceso de comunicación entre los procesos que se crean al ejecutar el siguiente programa.

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

void tratasenial (int);

void main(){

      pid_t idProceso;
      idProceso = fork();

      if (idProceso==-1) {
            perror("No se puede lanzar proceso”);
            exit(-1);
```

```
        }

        if (idProceso == 0) {
                signal (SIGUSR1, tratasenial);
                while (1)
                        pause();
        }

        if (idProceso > 0) {
                while (1) {
                        sleep(1);
                        kill (idProceso, SIGUSR1);
                }
        }
}

void tratasenial (int nsenial) {
        printf ("Recibida la señal del Padre\n");
}
```

¿Para qué sirve la función pause()?

4. Analice y describa la actividad que realiza el siguiente código. Explique como sucede el proceso de envió de información del proceso padre al proceso hijo.

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

int main() {
    pid_t idProceso;
    int estadoHijo;
    int descriptorTuberia[2];
    char buffer[100];

    if (pipe (descriptorTuberia) == -1) {
            perror ("No se puede crear Tuberia");
            exit(-1);
    }
    idProceso = fork();

    if (idProceso == -1) {
            perror ("No se puede crear proceso");
            exit(-1);
    }

    if (idProceso == 0) {
```

```
            close (descriptorTuberia[1]);
            read (descriptorTuberia[0], buffer, 9);
            printf ("Hijo : Recibido \"%s\"\n", buffer);
            exit(0);
        }

        if (idProceso > 0) {
            close (descriptorTuberia[0]);
            printf ("Padre : Envio \"Sistemas\"\n");
            strcpy (buffer, "Sistemas");
            write (descriptorTuberia[1], buffer, strlen(buffer)+1);

            wait(&estadoHijo);
            exit(0);
        }
        return(1);
    }
```

V

EJERCICIOS PROPUESTOS

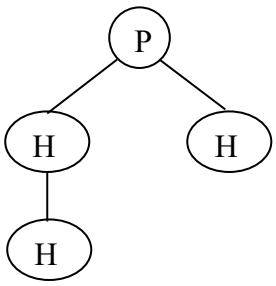
1. Analice y describa la actividad que realiza el siguiente código. Explique como sucede el proceso de envió de información del proceso padre al proceso hijo.

```
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <stdlib.h>
#include <string.h>

#define LEER 0
#define ESCRIBIR 1

char *frase = "Envia esto a traves de un tubo o pipe";
extern int errno;

int main(){
    int fd[2], bytesLeidos;
    char mensaje[100];
    int control;

    // se crea la tuberia
    control = pipe(fd);
    if ( control != 0 ) {
        perror("pipe:");
        exit(errno);
    }

    // se crea el proceso hijo
    control = fork() ;
```

```
        switch(control) {
            case -1 :
                perror("fork:");
                exit(errno);
            case 0 :
                close(fd[LEER]);
                write(fd[ESCRIBIR], frase, strlen(frase) + 1);
                close(fd[ESCRIBIR]);
                exit(0);
            default :
                close(fd[ESCRIBIR]);
                bytesLeidos = read(fd[LEER], mensaje, 100);
                printf("Leidos %d bytes : %s\n", bytesLeidos, mensaje);
                close(fd[LEER]);
        }
        exit(0);
    }
```

2. Elabore un programa propio que emplee la comunicación entre procesos (padre e hijo) utilizando pipes.

3. Escriba un programa que cree la sgte jerarquía de procesos:



P : el proceso padre debe de esperar a que sus hijos terminen para terminarse

H1 : el hijo 1, debe de mostrar su PID y el de su padre, además de enviarle cada 20 segundos una señal SIGUSR1 a H2.

H2 : el hijo 2 se activa cuando recibe la señal de H1, y debe de mostrar su PID como respuesta a la señal SIGUSR1 que recibe, para luego volverse a suspender.

H3 : el hijo 3 ejecuta un script que le de la bienevenida al usuario activo, considerando el horario correspondiente.

4. Investigue como se pueden enviar datos de un proceso padre a un proceso hijo y viceversa.

VI
CUESTIONARIO

1. ¿Existe una capacidad máxima de información que puede ser enviada a través de un *pipe*?

2. ¿Se puede enviar un objeto de una clase a través de un pipe? De ser posible elabore un ejemplo simple.

VII
ENTREGABLES

- Elaborar documento de sus actividades y los ejercicios realizados. Se debe incluir el comando utilizado, la descripción de la actividad que realiza, y una captura de pantalla (puede ser solo de la consola o terminal usado) de los resultados de las pruebas realizadas.

  En este documento incluir la respuesta al cuestionario y conclusiones.

- Subir al aula virtual el documento en formato PDF.
-