Objetivos:
-Uso del “man”
-Gestionar los procesos usando comandos del “bash”
-Controlar el envío de señales a los procesos usando comandos del “bash”
-Gestionar los procesos usando las llamadas a sistema
-Controlar el envío de señales a los procesos usando las llamadas a sistema

Comandos bash:

1.-Crea un programa en c con un bucle infinito, llamado infinito.c.
a)Compila y ejecuta dicho programa (el intérprete de comandos queda bloqueado a la espera.)
gcc infinito.c
./a.out
b)Para (STOP) el proceso
Ctrl + Z
c)Lleva el proceso al segundo plano.
bg
d)Identifica el proceso con jobs
jobs
e)Lleva el proceso al primer plano.
fg
f)Finaliza dicho proceso desde la misma terminal
kill 1234

2.- Visualiza las cabeceras del comando <ps -aux> .¿Qué información se muestra? Explica el significado de cada columna.(utiliza man ps)
ps -aux

```
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
kepa     10872  0.2  0.0  19788  5228 pts/0    Ss   11:41   0:00 bash
kepa     10883  0.0  0.0  21664  3784 pts/0    R+   11:41   0:00 ps -aux
```

USER: nombre de usuario

PID: número identificador de proceso

%CPU: uso de CPU en %

%MEM: uso de Memoria en %

VSZ: tamaño virtual del proceso en kilobytes. Incluye lo que ocupa tanto en RAM como en swap, tb se incluyen las librerías compartidas.

RSS: muestra la cantidad de memoria RAM que el proceso está utilizando realmente en kilobytes.
TTY: terminal asociada al proceso
STAT: Estado
START: tiempo inicio
TIME: tiempo CPU
3.-Muestra únicamente los procesos enviados por el usuario “lsi”.
ps -u lsi

4.-Realiza el ejercicio 3 pero esta vez usando el comando “tee”, para reenviar la salida simultáneamente a la pantalla y al fichero “salida.txt”.

```
ps -u lsi | tee salida.txt
```

5. Uso de nice y renice:

5a) Ejecuta el programa con bucle infinito, dándole una prioridad menor (cualquier valor entre 1 y 19).

```
renice 19 19394
```

5b) Desde otra terminal identifica su PID y su prioridad

```
ps -l
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 R  1000   19394   18386 99  99  19 -   637 -      pts/0    00:07:31 a.out
```

5c) Cambia la prioridad de dicho proceso a un valor negativo (-1 a -20)

```
sudo renice -20 19394
```

5d) Identifica su nueva prioridad

```
ps -l 19394
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 R  1000   19394   18386 99  60 -20 -   637 -      pts/0    00:09:59 a.out
```

5f) Envíale una señal SIGKILL al proceso.

```
kill -9 19394
```

6.-Realizar un script “backup.sh” que haga el backup de la carpeta “/home/lsi/prueba” en “/tmp/home_lsi.tar.gz”.Crea la carpeta prueba e introduce 2 ficheros en ella.

¿Cómo harías para ejecutarlo?

```
mkdir prueba
touch prueba/p1 prueba/p2
nano backup.sh
        tar cvzf /tmp/home_lsi.tar.gz prueba
        Ctrl O + Enter + Ctrl X
chmod u+x backup.sh
```

a) dentro de 5 minutos

Para instalarlo:

```
sudo apt install at
sudo systemctl status atd
```

Comando:

```
at now +5 minutes -f backup.sh
```

b) todos los días a las 4 pm

```
crontab -e
        0 16 * * * /home/$USER/backup.sh
        Ctrl O + Enter + Ctrl X
```

c) los 5 primeros días del mes, a las 9 am

```
crontab -e
        0 9 1-5 * * /home/$USER/backup.sh
        Ctrl O + Enter + Ctrl X
```

d) cada hora todos viernes de Agosto

```
crontab -e
        0 * * 8 5 /home/$USER/backup.sh
        Ctrl O + Enter + Ctrl X
```

e) una vez cada mes usando “anacron”

```
sudo cp backup.sh /etc/cron.monthly
```

APIs linux: Gestión de Procesos

1. Ejecuta el siguiente programa:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int main(int argc,char *argv[]){
 int rc=0,estado;
 if (argc!=1)
 {
    printf("uso: %s\n",argv[0]);
    rc=1;
    exit(rc);
 }
 else
 {
    if (fork())
    {
     wait(&estado);
     printf("PADRE: pid=%8d ppid=%8d user-id=%8d  \n",getpid(), getppid(), getuid());
     printf("EL ESTADO DEVUELTO POR EL HIJO: %d\n", estado);
     exit(rc);
    }
    else {
      printf("HIJO: pid=%8d \n", getpid());}
      exit(rc);
    }
}
}
```

Y responde a las siguientes preguntas:

1.1.¿Qué se visualiza en pantalla?

Si has puesto más de un argumento, aparecería el uso del comando.

Si no; si es el hijo, imprimiría la id del hijo. Si es el padre, entraría en espera hasta que acabe el hijo, entonces imprimiría los ids del padre y finalmente imprimiría el estado del hijo.

1.2. ¿Qué se ejecuta antes el padre o el hijo? ¿Por qué?

El hijo, porque el padre entra en el wait y espera a que llegue la señal de que ha terminado el hijo.

1.3. En la instrucción, wait(&estado), ¿Qué es lo que se recoge en la variable estado?

El código de terminación del hijo.

1.4. ¿Qué valor envía "exit" en caso de producirse un error? ¿Y en el caso de que no se produzca ningún error?

En caso de error 1. En caso de no error 0.

2. Explica qué es lo que hace este programa:

// **trapper.c**

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
void trapper(int);
int main(int argc, char *argv[])
{
    int i;
```

```
        for(i=1;i<=64;i++)
                {signal(i, trapper);}
        printf("Identificador del proceso: %d\n", getpid() );
        pause();
        printf("Continuando el main. Agur.\n");
        return 0;
    }

    void trapper(int sig)
    {
        printf("Señal que he recogido: %d\n", sig);
    }
```

2.1. Compilarlo y ejecutarlo. Apunta el PID que se muestra por pantalla.
7827
2.2. Desde otra terminal, envíale la señal SIGUSR1. ¿Cuál es el comando que has utilizado?
Kill -10 7827
2.3. ¿Qué es lo que pasa si un proceso recibe una señal que no trata?
La acción por defecto.
2.4. Cambia el programa anterior (trapper.c), para que sólo trate las señales del 1 al 9.
Habría que hacer un for de 1 a 9.
2.5. Ejecútalo y mándale la señal SIGKILL. ¿Cuál es el comando que has utilizado?
kill -9 8246
2.6. ¿Qué ha pasado al enviarle la señal? ¿Por qué?
Ha terminado, porque no la hemos tratado.

3. **Escribe un programa que cuando reciba la señal SIGUSR1, escriba "Hola" por la pantalla.**

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
void trapper(int);
int main(int argc, char *argv[])
{
    int i;
    for(i=1;i<=64;i++)
            {signal(i, trapper);}
    printf("Identificador del proceso: %d\n", getpid() );
    pause();
    printf("Continuando el main. Agur.\n");
    return 0;
}

void trapper(int sig)
{
    printf("Hola");
}
```

kill -10 8354

3.1. Cambia el programa para que haga lo mismo al recibir las señales SIGUSR2, SIGCONT, SIGSTOP y SIGKILL. ¿Es posible hacerlo? ¿Cuál es la conclusión?

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
void trapper(int);
int main(int argc, char *argv[])
{;
    signal(12,trapper)
    signal(18,trapper);
    signal(19,trapper);
    signal(9,trapper);
    printf("Identificador del proceso: %d\n", getpid() );
    pause();
    printf("Continuando el main. Agur.\n");
    return 0;
}

void trapper(int sig)
{
    printf("Hola”);
}
```

No, el sistema no lo permite, porque ni SIGKILL ni SIGSTOP son manipulables y evitables.

4. Compila el programa killer.c

// killer.c

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

int main(int argc, char *argv[])
{
    pid_t pid;
    int sig;
    if(argc==3)
    {
            pid=(pid_t)atoi(argv[1]);
            sig=atoi(argv[2]);

            kill(pid, sig);
    } else {
            printf("Uso correcto:\n %s pid signal\n",  argv[0]);
            return -1;
    }
    return 0;
}
```

4.1. ¿Qué hace la función atoi?

Conviertes una cadena en un entero, un int.

4.2. En Linux, ¿a qué equivale el tipo de dato pid_t?

Equivale a un int.

4.3. Pon en marcha el trapper. Recoge su PID y realiza un ./killer PID 9. ¿Qué ha pasado y por qué?

Termina, porque la señal 9 equivale a SIGKILL

5. Copia el programa alarm.c y ejecútalo.

```
#include <signal.h>
        #include <unistd.h>
        void trapper(int);
        int main(int argc, char *argv[])
        {
                int i;
                signal(14, trapper);
                printf("Identificador proceso: %d\n", getpid() );
                alarm(5);
                pause();
                alarm(3);
                pause();
                for(;;)
                {
                        alarm(1);
                        pause();
                }
                return 0;
        }
        void trapper(int sig)
        {
                printf("RIIIIIIIIING!\n");
        }
```

5.1. Explica qué hacen las siguientes instrucciones:

```
signal(14, trapper);
for(;;)
{
alarm(1);
pause();
}
```

Cada segundo lanza un SIGALARM y llama a trapper, que imprime RING!

5.2. ¿ Qué hace el programa?

Imprime el identificador del proceso, tras 5 segundos lanza un SIGALARM y llama a trapper, tras 3 segundos, lo mismo y luego cada segundo llama a trapper.

6. Compila, ejecuta y explica lo que hace el programa *signalfork.c*.

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

```
void trapper(int sig)
{
  printf("SIGUSR1\n");
}

int main(int argc, char *argv[])
{
pid_t padre, hijo;
padre = getpid();
signal( SIGUSR1, trapper );
if ( (hijo=fork()) == 0 )
{ /* hijo */
        sleep(1);
        kill(padre, SIGUSR1);
        sleep(1);
        kill(padre, SIGUSR1);
        sleep(1);
        kill(padre, SIGUSR1);
        sleep(1);
        kill(padre, SIGKILL);
        exit(0);
}
else
{ /* padre */
        for (;;);
}

return 0;
}
```

Obtiene la id del proceso y lo guarda en padre, ejecuta signal y si registra un SIGUSR1, llama a trapper. Al hacer fork() si es el hijo repite 3 veces esperar 1 segundo y lanza un SIGUSR1 al padre, el cual ejecuta trapper e imprime SIGUSR1 en pantalla, por último, lo termina con un SIGKILL. Si es el padre, entra en un bucle infinito.

6.1. **Hemos visto qué es lo que pasa con las señales que le envía el hijo al padre, pero si el padre le envía al hijo SIGUSR1 ¿qué pasa? ¿Las funciones asignadas mediante signal, se heredan al hacer fork?**

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
void trapper(int sig)
{
        printf("SIGUSR1\n");
}

int main(int argc, char *argv[])
{
        pid_t padre, hijo;
        padre = getpid();
        signal( SIGUSR1, trapper );
```

```
        if ( (hijo=fork()) != 0 )
        { /* hijo */
                sleep(1);
                kill(hijo, SIGUSR1);
                sleep(1);
                kill(hijo, SIGUSR1);
                sleep(1);
                kill(hijo, SIGUSR1);
                sleep(1);
                kill(hijo, SIGKILL);
                exit(0);
        }
        else
        { /* padre */
                for (;;);
        }

        return 0;
}
```

El código funciona igual, porque todas las señales que registra el padre las hereda el hijo.

6.2. Cambia las 2 primeras líneas del hijo así:

```
/*hijo */
sleep(10);
```

Cambia las primeras líneas del padre así:

```
 /* padre */
 sleep(2);
kill(hijo, SIGUSR2);
for (;;);
```

Compilarlo y ejecutarlo. Parece que se bloquea. ¿Por qué?

El padre ejecuta el kill(hijo,SIGUSR2) antes que el hijo despierte, como no se trata hace la acción por defecto, esto es terminar el proceso. Y luego el padre se queda en un bucle infinito.

7. Realiza un programa en C que crea una carpeta llamada “lana” en $HOME utilizando el resto de funciones exec.

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
char *args[]={"mkdir","lana",NULL};
if (execve("/bin/mkdir",args,NULL)==-1){
  perror("execve");
  exit(EXIT_FAILURE);
```

```
}
puts("No debería llegar aquí");
exit(EXIT_SUCCESS);
}
```

Para los otros:
execl(“/bin/mkdir”,args[0],args[1],NULL)
execle(“/bin/mkdir”,args[0],args[1],NULL,NULL)
execlp(args[0],args[0],args[1],NULL)
execv(“/bin/mkdir”,args)
execvp(args[0],args)

9. **Explica lo que hace la siguiente llamada “./padre 60 hijo” dado los siguientes programas:**

reloj.c

```
#include <stdlib.h> /*atoi and exit*/
#include <stdio.h> /*printf*/
#include <unistd.h> /*alarm*/
#include <signal.h> /*signal and SIGALRM*/
void xtimer(){
printf("time expired.\n");
}
int main(int argc, char *argv[]){
unsigned int sec;
sec=atoi(argv[1]);
printf("time is %u\n",sec);
signal(SIGALRM,xtimer);
alarm(sec);
pause();
return 0;
}
```

hijo.c

```
#include <unistd.h> /*sleep*/
int main(){
sleep(6);
return 0;
}
```

padre.c

```
#include <sys/types.h> /*kill y wait*/
#include <sys/wait.h> /*wait*/
```

```
#include <signal.h>/*kill*/
#include <stdlib.h>/*exit*/
#include <stdio.h>/*printf*/
#include <unistd.h>/*fork and exec...*/
#include <time.h> /*time*/
int main(int argc, char *argv[]){
int idHijo, idReloj, id, t1, status1, status2;
id = getpid();
printf("Proceso padre: %d\n", id);
idReloj = fork();
if(idReloj == 0) { /* hijo Reloj */
execl("reloj", "reloj", argv[1], NULL);
}
idHijo = fork()
if((idHijo == 0) {
execv(argv[2], &argv[2]);
}
printf("Proceso hijo Perezoso: %d\n", idHijo);
printf("Proceso hijo Reloj: %d\n", idReloj);
t1 = time(0);
if((id = wait(&status1)) == idHijo) {
kill(idReloj, SIGKILL);
//Si el hijo ha cambiado de estado, antes de la ejecución del wait
//entonces la llamada a wait retornará inmediatamente con su estado
wait(&status2);
} else {
kill(idHijo, SIGKILL);
wait(&status2);
status1 = 1;}
t1=time(0)-t1;
printf("Tiempo del proceso hijo : %d\n", t1);
exit(status1);
}
```

Proceso del padre: Imprime el id del proceso padre. Crea un hijo (Reloj) con el fork, guarda su id en idReloj, crea otro hijo y guarda su id en idHijo. Imprime los ids de ambos hijos. Guarda el tiempo actual en t1. Con el wait bloquea al proceso padre hasta que termine uno de sus hijos:

- Si el hijo que termina es el idHijo, mandaría una señal SIGKILL al Reloj para finalizar el proceso. Imprimiría el tiempo transcurrido y terminaría el proceso padre de manera correcta.

- Si el hijo que termina es el idReloj mandaría una señal SIGKILL al Hijo para finalizar el proceso. Luego con el wait se devolverá automáticamente -1, ya que no se existen procesos hijos. Imprimirá el tiempo y finalizará el proceso padre con un error, ya que se ha sobrescrito status1 a 1.

Proceso hijo Reloj: Ejecuta el ejecutable reloj a el cual se le pasa como parámetro 60. Primero convierte los caracteres 60 a int. Imprime que el tiempo son 60 segundos y ejecuta un alarma después de 60 segundos, mientras se queda en pausa. Pasados los 60 segundos, lo recoge con el signal y hace una llamada a xtimer() el cual imprime que el tiempo ha expirado y termina.

Proceso hijo Hijo: Ejecuta el ejecutable hijo, pasa como parámetro un puntero al vector argv desde la posición 3 (Sin tener en cuenta que se empieza en 0). En hijo, duerme durante 6 segundos y termina.