| **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.- Visualizar los distintos formatos de ps ejecutando la orden con las opciones: -a, -u, -x (¿Qué información muestra cada opción?)

USER PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND
root 1 0.2 0.0 33888 3164 ? Ss 08:07 0:00 /sbin/init
kepa 2607 2.8 2.5 1253276 100000 ? Sl 08:11 0:05 /usr/lib/libreoffice/program/soffice.bin
%CPU
%MEM %RAM utilizado
Resident Set Size:Uso de la RAM (No se incluye lo q ocupa en la swap)+ Pila + Heap
Virtual Memory size:RAM+Swap

ps -a muestra también los procesos procesos iniciados por todos los usuaios
ps -u muestra los procesos del usuario
ps -x muestra los procesos con o sin terminal includos los del sistema.

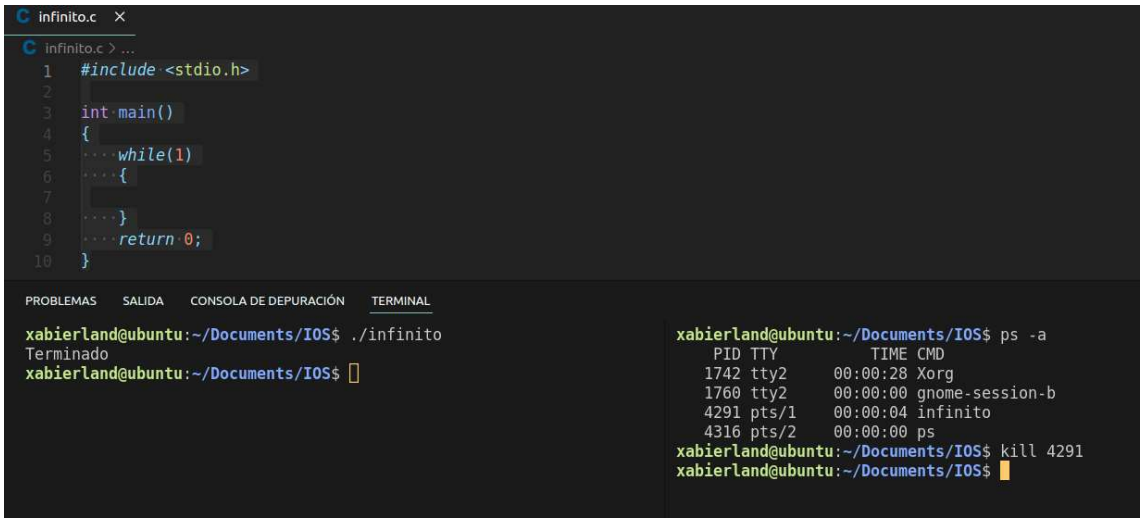
UID: la ID de usuario del propietario de esteproceso.
PID: el ID del proceso del proceso.
PPID: ID del proceso padre del proceso.
C: El número de hijos que tiene el proceso.
VSZ: Tamaño en páginas RAM de la imagen de proceso.
RSS: Tamaño de conjunto residente. Esta es la memoria física no intercambiada utilizada por el proceso.
PSR: El procesador al que está asignado el proceso.
TIEMPO: Hora de inicio. La hora en que comenzó el proceso.
TTY: El nombre de la consola en la que el usuario inició sesión.
HORA: La cantidad de tiempo de procesamiento de la CPU que el proceso ha utilizado.
CMD: el nombre del comando que inició el proceso.

2.-Crea un programa en c con un bucle infinito. Al ejecutarlo, el intérprete de comandos queda bloqueado a la espera. Abre otra terminal y acaba dicho proceso usando el comando “kill”.

```
#include <stdio.h>

int main()
{
   while(1)
   {

   }
   return 0;
}
```




3.-Muestra únicamente los procesos enviados por el usuario “lsi”.
Mi usuario en este caso es xabierland

```
xabierland@ubuntu:~$ ps -u
USER        PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME
COMMAND
xabierl+    1738  0.0  0.1 166800  6684 tty2     Ssl+ 17:05   0:00 /usr/lib/gdm3/gdm-x-
session --run-script env GNOME_SHELL_SESSION_MODE=ubuntu /usr/bin/gnome-
session --systemd --ses
xabierl+    1742  2.0  2.3 343352 94544 tty2     Sl+  17:05   0:29 /usr/lib/xorg/Xorg vt2 -
displayfd 3 -auth /run/user/1000/gdm/Xauthority -background none -noreset -keeptty -
verbose
xabierl+    1760  0.0  0.3 193384 15464 tty2     Sl+  17:05   0:00 /usr/libexec/gnome-
session-binary --systemd --systemd --session=ubuntu
xabierl+    3690  0.0  0.1  14076  5900 pts/1    Ss+  17:22   0:00 /usr/bin/bash
xabierl+    3974  0.0  0.1  13368  4992 pts/2    Ss+  17:23   0:00 /usr/bin/bash
xabierl+    4380  0.2  0.1  13368  4740 pts/0    Ss   17:28   0:00 bash
xabierl+    4387  0.0  0.0  14192  3340 pts/0    R+   17:29   0:00 ps -u
```

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”.

```
xabierland@ubuntu:~$ ps -u | tee salida.txt
USER        PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME
COMMAND
xabierl+    1738  0.0  0.1 166800  6684 tty2     Ssl+ 17:05   0:00 /usr/lib/gdm3/gdm-x-
session --run-script env GNOME_SHELL_SESSION_MODE=ubuntu /usr/bin/gnome-
session --systemd --session=ubuntu
xabierl+    1742  2.0  2.3 343352 94544 tty2     Sl+  17:05   0:30 /usr/lib/xorg/Xorg vt2 -
displayfd 3 -auth /run/user/1000/gdm/Xauthority -background none -noreset -keeptty -
verbose 3
xabierl+    1760  0.0  0.3 193384 15464 tty2     Sl+  17:05   0:00 /usr/libexec/gnome-
session-binary --systemd --systemd --session=ubuntu
xabierl+    3690  0.0  0.1  14076  5900 pts/1    Ss+  17:22   0:00 /usr/bin/bash
xabierl+    3974  0.0  0.1  13368  4992 pts/2    Ss+  17:23   0:00 /usr/bin/bash
xabierl+    4380  0.0  0.1  13368  4844 pts/0    Ss   17:28   0:00 bash
```

```
xabierl+    4391  0.0  0.0  14192  3316 pts/0    R+   17:30   0:00 ps -u
xabierl+    4392  0.0  0.0  11004   580 pts/0    S+   17:30   0:00 tee salida.txt
xabierland@ubuntu:~$ ls
BurpSuiteCommunity  Desktop  Documents  Downloads  Music  Pictures  Public
salida.txt  snap  Templates  Videos
xabierland@ubuntu:~$ cat salida.txt
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME
COMMAND
xabierl+    1738  0.0  0.1 166800  6684 tty2     Ssl+ 17:05   0:00 /usr/lib/gdm3/gdm-x-
session --run-script env GNOME_SHELL_SESSION_MODE=ubuntu /usr/bin/gnome-
session --systemd --session=ubuntu
xabierl+    1742  2.0  2.3 343352 94544 tty2     Sl+  17:05   0:30 /usr/lib/xorg/Xorg vt2 -
displayfd 3 -auth /run/user/1000/gdm/Xauthority -background none -noreset -keeptty -
verbose 3
xabierl+    1760  0.0  0.3 193384 15464 tty2     Sl+  17:05   0:00 /usr/libexec/gnome-
session-binary --systemd --systemd --session=ubuntu
xabierl+    3690  0.0  0.1  14076  5900 pts/1    Ss+  17:22   0:00 /usr/bin/bash
xabierl+    3974  0.0  0.1  13368  4992 pts/2    Ss+  17:23   0:00 /usr/bin/bash
xabierl+    4380  0.0  0.1  13368  4844 pts/0    Ss   17:28   0:00 bash
xabierl+    4391  0.0  0.0  14192  3316 pts/0    R+   17:30   0:00 ps -u
xabierl+    4392  0.0  0.0  11004   580 pts/0    S+   17:30   0:00 tee salida.txt
```

5.-Ejecuta el programa con bucle infinito, dándole una prioridad menor (cualquier valor entre 1 y 19). Desde otra terminal envíale una señal SIGKILL al proceso.

```
xabierland@ubuntu:~/Documents/IOS$
nice -n 1 ./infinito
Terminado (killed)
xabierland@ubuntu:~/Documents/IOS$
```

```
SIGPWR
31) SIGSYS      34) SIGRTMIN    35)
SIGRTMIN+1  36) SIGRTMIN+2  37)
SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5
40) SIGRTMIN+6  41) SIGRTMIN+7
42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10
45) SIGRTMIN+11 46) SIGRTMIN+12
47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15
50) SIGRTMAX-14 51) SIGRTMAX-13
52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10
55) SIGRTMAX-9  56) SIGRTMAX-8
57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5
60) SIGRTMAX-4  61) SIGRTMAX-3
62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
xabierland@ubuntu:~/Documents/IOS$
ps -a
    PID TTY          TIME CMD
   1742 tty2     00:00:32 Xorg
   1760 tty2     00:00:00 gnome-session-b
   4533 pts/1    00:01:00 infinito
```

| | |
|---|---|
| | 4622 pts/2 00:00:00 ps<br>xabierland@ubuntu:~/Documents/IOS$<br>kill -9 4533<br>xabierland@ubuntu:~/Documents/IOS$ |

6.-Realizar un script “backup.sh” que haga el backup de la carpeta /home/lsi en /tmp/home_lsi.tar.gz.
¿Cómo harías para ejecutarlo?
crontab -e

a) dentro de 5 minutos
   at now +5 minutes -f /home/xabierland/bashscript.sh
b) todos los días a las 4 pm
   0 16 * * * /home/xabierland/backup.sh
c) los 5 primeros días del mes, a las 9 am
   0 9 1-5 * * /home/xabierland/backup.sh
d) cada hora todos viernes de Agosto
   * */1 * 8 5 /home/xabierland/backup.sh
e) una vez cada mes usando “anacron”
   sudo cp /home/xabierland/backup.sh /etc/cron.montly

# 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;
 }
 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);
  }
  else {
      printf("HIJO: pid=%8d \n", getpid());}
      exit(rc);
}
```

**Y responde a las siguientes preguntas:**

**1.1.¿Qué se visualiza en pantalla?**

```
xabierland@ubuntu:~/Documents/IOS/LAB6$ ./proc1
HIJO: pid=    5635
PADRE: pid=    5634 ppid=    5478 user-id=    1000
EL ESTADO DEVUELTO POR EL HIJO: 0
```

Primero se crea un hijo con fork y se espera a que este acabe devolviendo en el exit un 0 como estado para indicar que todo esta bien. Después procede a ejecutar el padre que manda

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

Se ejecuta antes el hijo ya que el padre se queda esperando con un wait hasta que el hijo lo devuelve su estado y entonces termina el de ejecutarse.

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

Devuelve el estado del hijo que siempre va a ser 0 porque el exit(rc) el rc este iniciado a 0.

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

0 sin errores

1 con errores

**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.**

**gcc -o trapper trapper.c**

**2.2. Desde otra terminal, envíale la señal SIGUSR1. ¿Cuál es el comando que has utilizado?**

| | |
|---|---|
| xabierland@ubuntu:~/Documents/IOS/LAB6$ ./trapper<br>Identificador del proceso: 6385<br><br>Señal que he recogido: 10<br>Continuando el main. Agur.<br>xabierland@ubuntu:~/Documents/IOS/LAB6$ | xabierland@ubuntu:~/Documents/IOS/LAB6$ kill -10 6385<br>xabierland@ubuntu:~/Documents/IOS/LAB6$ |

**2.3. ¿Qué es lo que pasa si un proceso recibe una señal que no trata?**

Si el programa no la trata hará lo que esa señal tenga que hacer. Aun así hay dos señales que no pueden ser capturadas 9 y 19 | SIGKILL y SIGSTOP

**2.4. Cambia el programa anterior (trapper.c), para que sólo trate las señales del 1 al 9.**

```
3.  #include <signal.h>
4.  #include <stdio.h>
5.  #include <unistd.h>
6.  void trapper(int);
7.
8.  int main(int argc, char *argv[])
9.  {
10.     int i;
11.     for(i=1;i<=9;i++)
12.             {signal(i, trapper);}
13.     printf("Identificador del proceso: %d\n", getpid() );
14.     pause();
15.     printf("Continuando el main. Agur.\n");
16.     return 0;
17. }
18.
19. void trapper(int sig)
20. {
21.     printf("\nSeñal que he recogido: %d\n", sig);
22. }
```

**2.5. Ejecútalo y mándale la señal SIGUSR1. ¿Cuál es el comando que has utilizado?**

**kill -10 7327**

**2.6. ¿Qué ha pasado al enviarle la señal? ¿Por qué?**

**Que se ha ejecutado la función definida en la señal SIGUSR1 que es escribir una señal por defecto, en mi caso 1**

**xabierland@ubuntu:~/Documents/IOS/LAB6$ ./trapper**
**Identificador del proceso: 7585**
**Señal definida por el usuario 1**

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=10;
        if(i=10)
                {signal(i, trapper);}
        printf("Identificador del proceso: %d\n", getpid() );
        pause();
        return 0;
}

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

| | |
|---|---|
| xabierland@ubuntu:~/Documents/IOS/LAB6$ ./hola<br>Identificador del proceso: 8544<br><br>HOLA<br>xabierland@ubuntu:~/Documents/IOS/LAB6$ | xabierland@ubuntu:~/Documents/IOS/LAB6$ kill -10 8544<br>xabierland@ubuntu:~/Documents/IOS/LAB6$ |

**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?**

Que algunas señales se pueden tratar y otras no, por ejemplo SIGSTOP Y SIGKILL no se pueden tratar pero SIGUSR2 y SIGCONT si.

4. **Compila el programa killer.c**

```
// killer.c
#include <sys/types.h>
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.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?**

Convierte un string a integer

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

```
xabierland@ubuntu:~/Documents/IOS/LAB6$ gcc -o killer.i -E killer.c
xabierland@ubuntu:~/Documents/IOS/LAB6$ grep "pid_t" killer.i
typedef int __pid_t;
typedef __pid_t pid_t;
```

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

| | |
|---|---|
| xabierland@ubuntu:~/Documents/IOS/LAB6$ ./trapper<br>Identificador del proceso: 3478<br>Terminado (killed)<br>xabierland@ubuntu:~/Documents/IOS/LAB6$ | xabierland@ubuntu:~/Documents/IOS/LAB6$ ./killer 3478 9<br>xabierland@ubuntu:~/Documents/IOS/LAB6$ |

**5. Copia el programa alarm.c y ejecútalo.**

```
#include <signal.h>
#include <unistd.h>
#include <stdio.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();
}
```

Recibe la señal 14 que es la SIGALARM la cual salta cada vez que se ejecuta la línea alarm(1) y tarda 1 segundo en salta. Todo esto dentro de un bucle infinito.

**5.2. ¿ Qué hace el programa?**

El programa lo que hace es hacer una alarma que escribe RING cuando se recibe un SIGALARM. Primero se recibe una al de 5 segundos de iniciar el programa seguida de una de tres para terminar en un bucle infinito en el que suena una alarma cada segundo.

**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>
#include <stdlib.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;
}
```

El programa crea un hijo mientras el padre se queda en un bucle. Después el hijo manda una señal al padre la cual le hace escribir SIGUSR1 tres veces para después matar el proceso padre.

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?**
Si, el hijo hereda la función

**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?**
Porque no tratamos la señal SIGUSR2 y entra en el for con 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[]={"/bin/mkdir","lana",NULL};
if (execve("/bin/mkdir",args,NULL)==-1)
{perror("execve");
  exit(EXIT_FAILURE);}
puts("No debería llegar aquí");
exit(EXIT_SUCCESS);}
```

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
   char *args[]={"/bin/mkdir","/home/xabierland/lana",NULL};
   if (execve("/bin/mkdir",args,NULL)==-1)
   {
      perror("execve");
      exit(EXIT_FAILURE);
   }
   puts("No debería llegar aquí");
   exit(EXIT_SUCCESS);
}
```

**8. 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);
}
```

Lo primero que hace al llamar la función padre es ejecutar el programa reloj programando con el parámetro 1 que recibe que en este caso es 60, es decir, un minuto y una vez pasado ese tiempo manda una señal de SIGALRM mediante alarm. La cosa es que el hijo duerme durante solo 6 segundos por lo que el hijo acaba antes de llegar la interrupción de la alarma forzando así la salida del programa. Aun así, si en vez de ./padre 60 hijo ponemos ./padre 5 hijo el tiempo de vida del hijo en vez de ser 6 será de 5 segundos