



## **Sistemas Operativos**

### ***Tp 1er Parcial***

**- Gestión de procesos**

**Profesor:** Leandro Robles

**Comisión:** (757) 1

**Apellido y nombre:** Viltez, Hernan

**DNI:** 30893000

## Ejercicio 1

Observa el siguiente código y escribe la jerarquía de procesos resultante.

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

int main (int argc, char *argv[]) {
    int num; pidmore t pid;

    for (num= 0; num< 3; num++) {
        pid= fork();

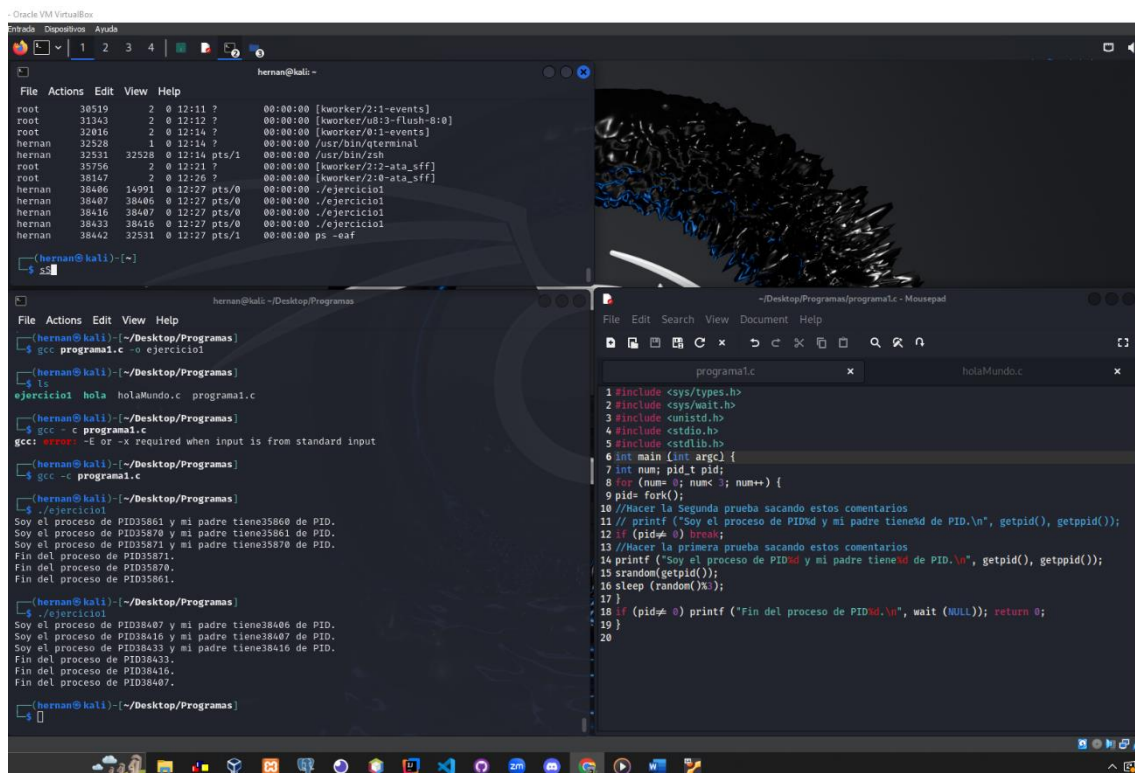
        //Hacer la Segunda prueba sacando estos comentarios
        // printf ("Soy el proceso de PID%d y mi padre tiene%d de PID.\n", getpid(), getppid());

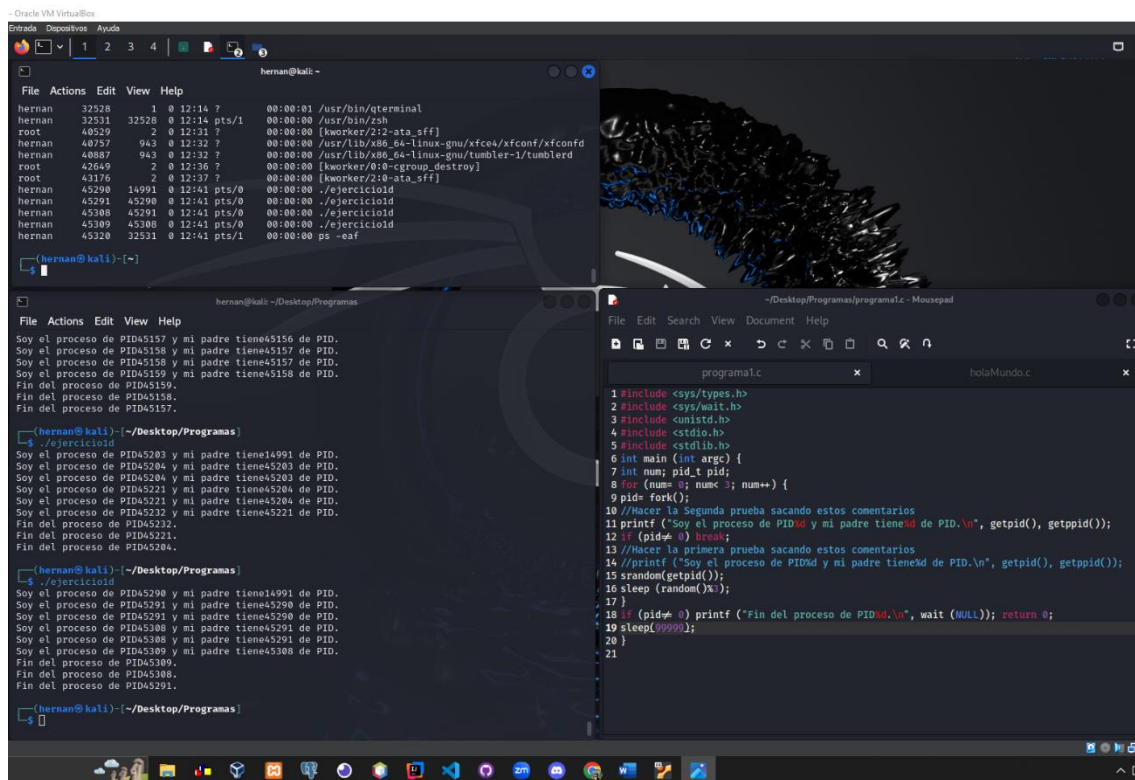
        if (pid!= 0) break;

        //Hacer la primera prueba sacando estos comentarios
        //printf ("Soy el proceso de PID%d y mi padre tiene%d de PID.\n", getpid(), getppid());

        srandom(getpid());
        sleep (random()%3);
    }

    if (pid!= 0) printf ("Fin del proceso de PID%d.\n", wait (NULL)); return 0;
}
```





Contesta las siguientes preguntas:

¿Por qué aparecen mensajes repetidos? Presta atención al orden de terminación de los procesos, ¿qué observas? ¿por qué?.

### Respuesta:

Los mensajes repetidos ocurren porque cada proceso, incluido el proceso padre, está ejecutando su propia copia del código debido al bucle `for()`. Sin embargo, el orden de terminación de los procesos no está garantizado debido al uso de la función `random()`, que genera un tiempo de espera aleatorio para cada proceso (`sleep(random()%3)`). Esto genera una cierta aleatoriedad en el orden en el que terminan los procesos.

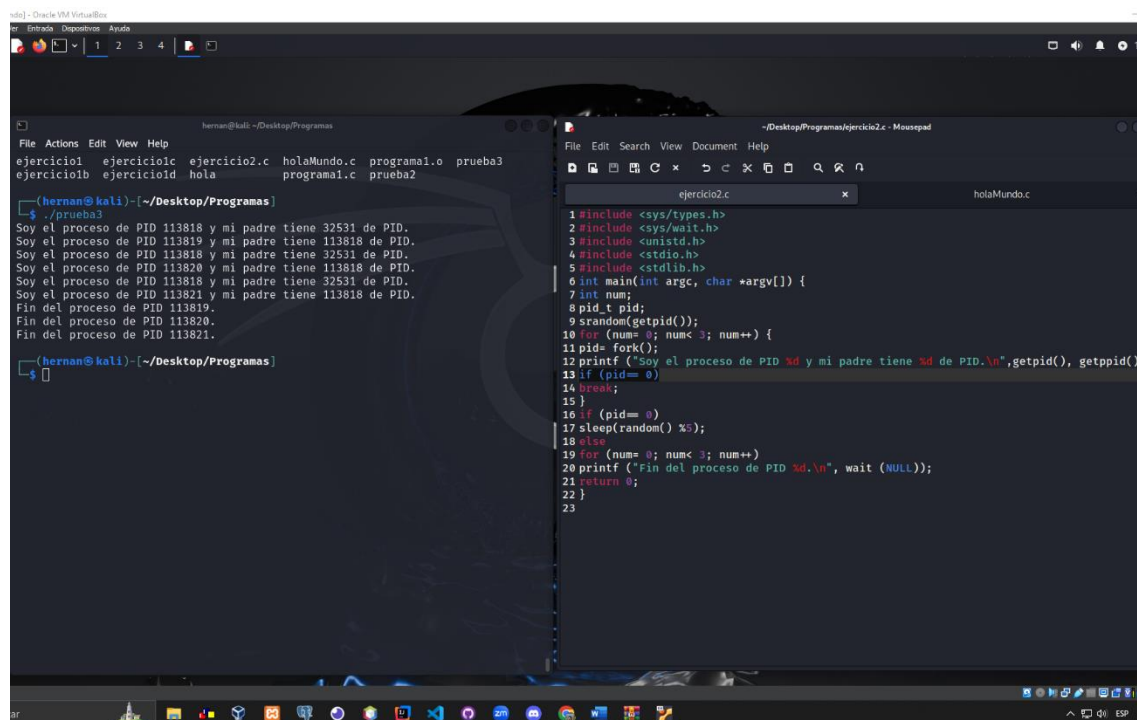
En el escenario 1 los procesos hijos no se reproducen debido al **break**;

En cambio en el escenario 2 el primer hijo le quedan 2 **fork()** que crean sus propios `print()`.

Ahora compila y ejecuta el código para comprobarlo. Presta atención al orden de terminación de los procesos, ¿qué observas? ¿por qué?

El **pid==0** (el proceso hijo) rompe el bucle con el **break**, por eso el padre no genera más hijos, pero vuelve a iniciar siempre con el primer proceso hijo, que crea otro hijo.

El orden de finalización de los procesos hijos no es necesariamente el mismo en el que fueron creados. Esto se debe a la función **sleep(random() %5)**, que introduce un tiempo de espera aleatorio en cada hijo. Esto significa que algunos hijos pueden dormir más tiempo que otros, lo que provoca que los mensajes de finalización aparezcan en un orden no secuencial.



The screenshot shows a Kali Linux desktop environment with two windows. The left window is a terminal with the following output:

```
hernan@kali: ~/Desktop/Programas
$ ./prueba3
Soy el proceso de PID 113818 y mi padre tiene 32531 de PID.
Soy el proceso de PID 113819 y mi padre tiene 113818 de PID.
Soy el proceso de PID 113818 y mi padre tiene 32531 de PID.
Soy el proceso de PID 113820 y mi padre tiene 113818 de PID.
Soy el proceso de PID 113818 y mi padre tiene 32531 de PID.
Soy el proceso de PID 113821 y mi padre tiene 113818 de PID.
Fin del proceso de PID 113819.
Fin del proceso de PID 113820.
Fin del proceso de PID 113821.
$
```

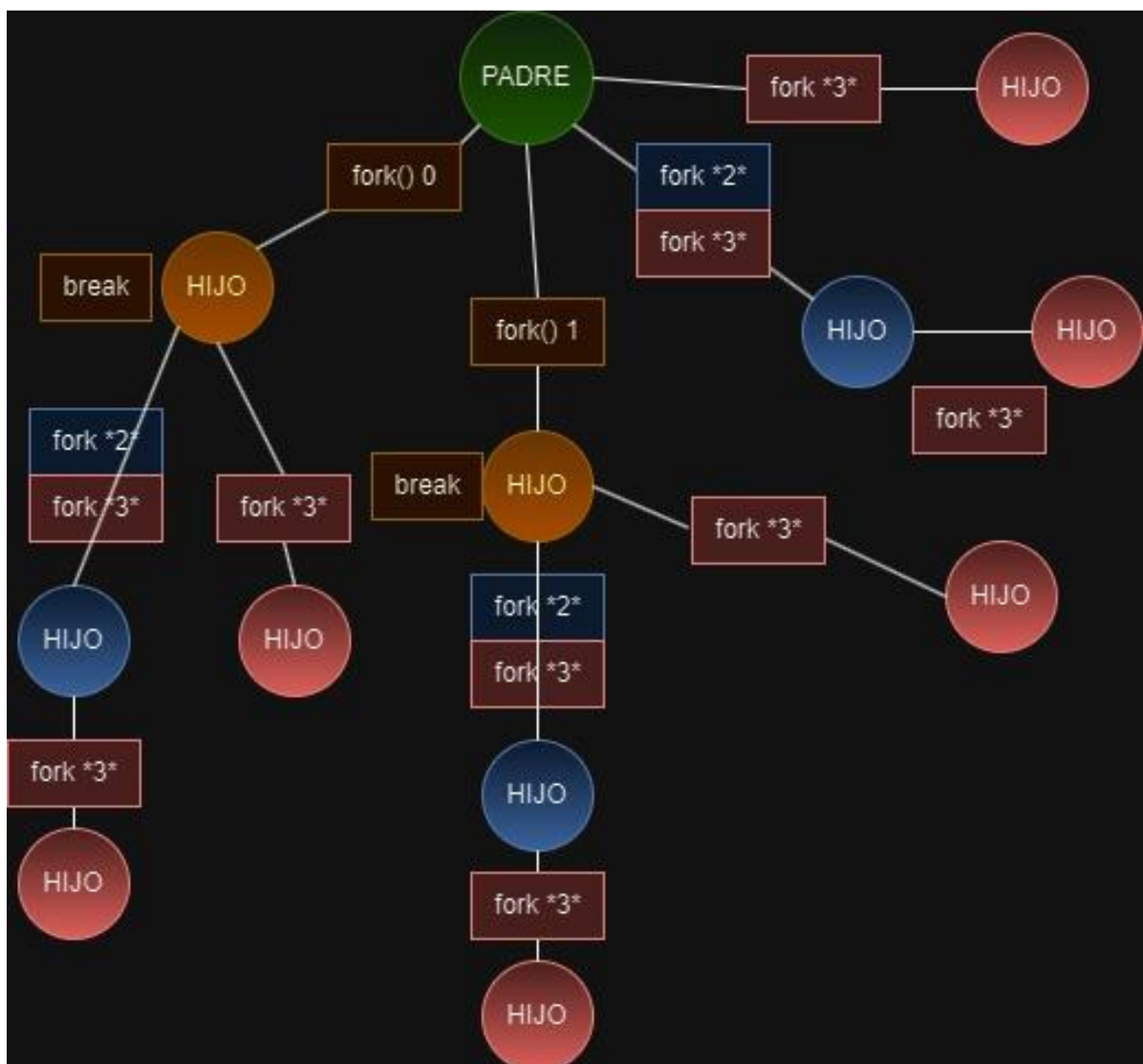
The right window is a code editor showing the source code for **ejercicio2.c**:

```
1 #include <sys/types.h>
2 #include <sys/wait.h>
3 #include <unistd.h>
4 #include <stdio.h>
5 #include <stdlib.h>
6 int main(int argc, char *argv[]) {
7     int num;
8     pid_t pid;
9     srand(getpid());
10    for (num= 0; num< 3; num++) {
11        pid= fork();
12        printf ("Soy el proceso de PID %d y mi padre tiene %d de PID.\n",getpid(), getppid())
13        if (pid== 0)
14            break;
15    }
16    if (pid== 0)
17        sleep(random() %5);
18    else
19        for (num= 0; num< 3; num++)
20            printf ("Fin del proceso de PID %d.\n", wait (NULL));
21    return 0;
22 }
23
```

### Ejercicio 3

Dibuja la estructura del árbol de procesos que obtendríamos al ejecutar el siguiente fragmento de código:

```
for (num= 0; num< 2; num++) {  
    nuevo= fork(); /* 1 */  
    if (nuevo== 0)  
        break;  
}  
nuevo= fork(); /* 2 */  
nuevo= fork(); /* 3 */  
printf("Soy el proceso %d y mi padre es %d\n", getpid(), getppid());
```

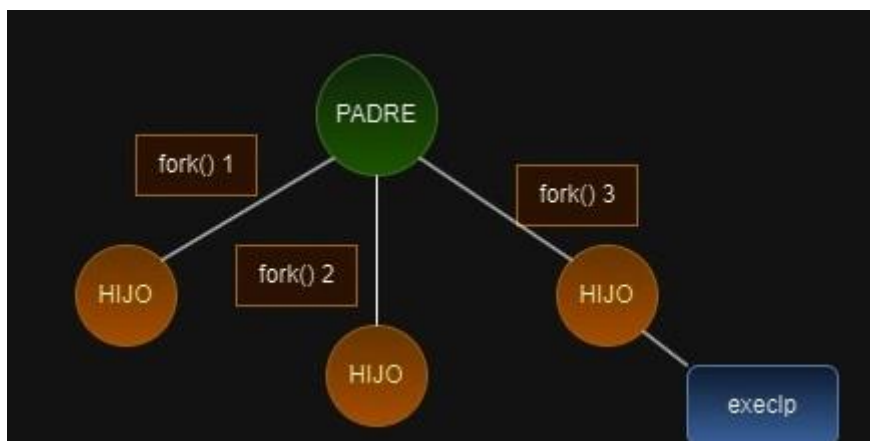


#### Ejercicio 4

Considerando el siguiente fragmento de código:

```
for (num= 1; num<= n; num++){  
    nuevo= fork();  
    if ((num== n) && (nuevo== 0))  
        execlp ("ls", "ls", "-l", NULL);  
}
```

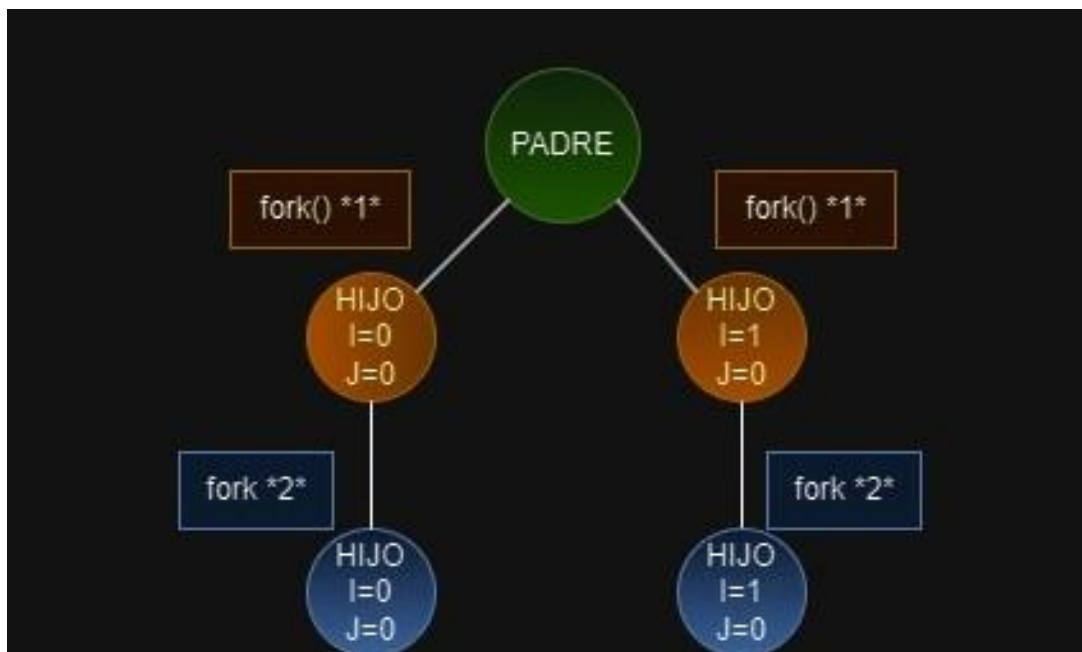
- a) Dibuja la jerarquía de procesos generada cuando se ejecuta y n es 3.
- b) Indica en que procesos se ha cambiado la imagen del proceso usando la función execlp.



### Ejercicio 5

Dibuja la jerarquía de procesos que resulta de la ejecución del siguiente código. Indica para cada nuevo proceso el valor de las variables *i* y *j* en el momento de su creación.

```
for (i= 0; i< 2; i++) {  
  pid= getpid();  
  for (j= 0; j< i+2; j++) {  
    nuevo= fork(); /* 1 */  
    if (nuevo!= 0) {  
      nuevo= fork(); /* 2 */  
      break;  
    }  
  }  
  if (pid!= getpid())  
    break;  
}
```



## Ejercicio 6

Estudia el siguiente código y escribe la jerarquía de procesos resultante. Después, compila y ejecuta el código para comprobarlo (debes añadir llamadas al sistema getpid, getppid y wait para conseguirlo).

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#define L1 2
#define L2 3
int main (int argc, char *argv[]) {
    int cont1, cont2;
    pid_t pid;
    for (cont2= 0; cont2< L2; cont2++)
    {
        for (cont1= 0; cont1< L1; cont1++)
        {
            pid= fork();
            if (pid== 0)
                break;
        }
    }
    if (pid!= 0) break;
}

/*Para Simplificar agregar al final
printf ("Soy el proceso de PID %d y mi padre tiene %d de PID.\n",
getpid(), getppid());
if (pid!= 0)
for (cont1= 0; cont1< L1; cont1++)
printf ("Fin del proceso de PID %d.\n", wait (NULL));
return 0;
• }
```



Padre:

- hijo

-nieto

-bisnieto

-bisibisnieto

-bisbisnieto

-bisnieto

-bisbisnieto

-bisbisnieto

-nieto

-bisnieto

-bisbisnieto

-bisbisnieto

-bisnieto

-bisbisnieto

-bisbisnieto

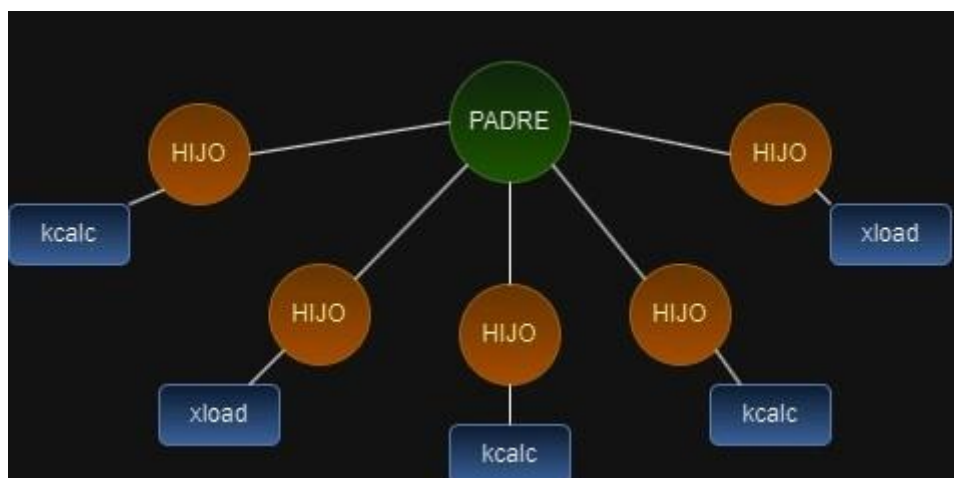
The screenshot displays a Kali Linux desktop environment. On the left, a terminal window titled 'herman@kali: ~' shows the output of the 'ps' command, listing processes like 'systemd-logind', 'systemd-udevd', 'udisksd', 'upowerd', and 'xcapd'. Below this, another terminal window titled 'herman@kali: ~/Desktop/Programas' shows the execution of a script named 'ejercicio6.c'. The script's output consists of multiple lines indicating the creation and termination of child processes, such as 'Soy el proceso de PID 107095 y mi padre tiene 2059 de PID.' and 'Fin del proceso de PID 107101.' On the right, a code editor window titled '~ / Desktop / Programas / ejercicio6.c - Mousepad' displays the source code of the script. The code includes standard C headers, defines two levels of nesting (L1 and L2), and uses 'fork()' to create child processes, printing their PIDs and their parents' PIDs before waiting for them to complete.

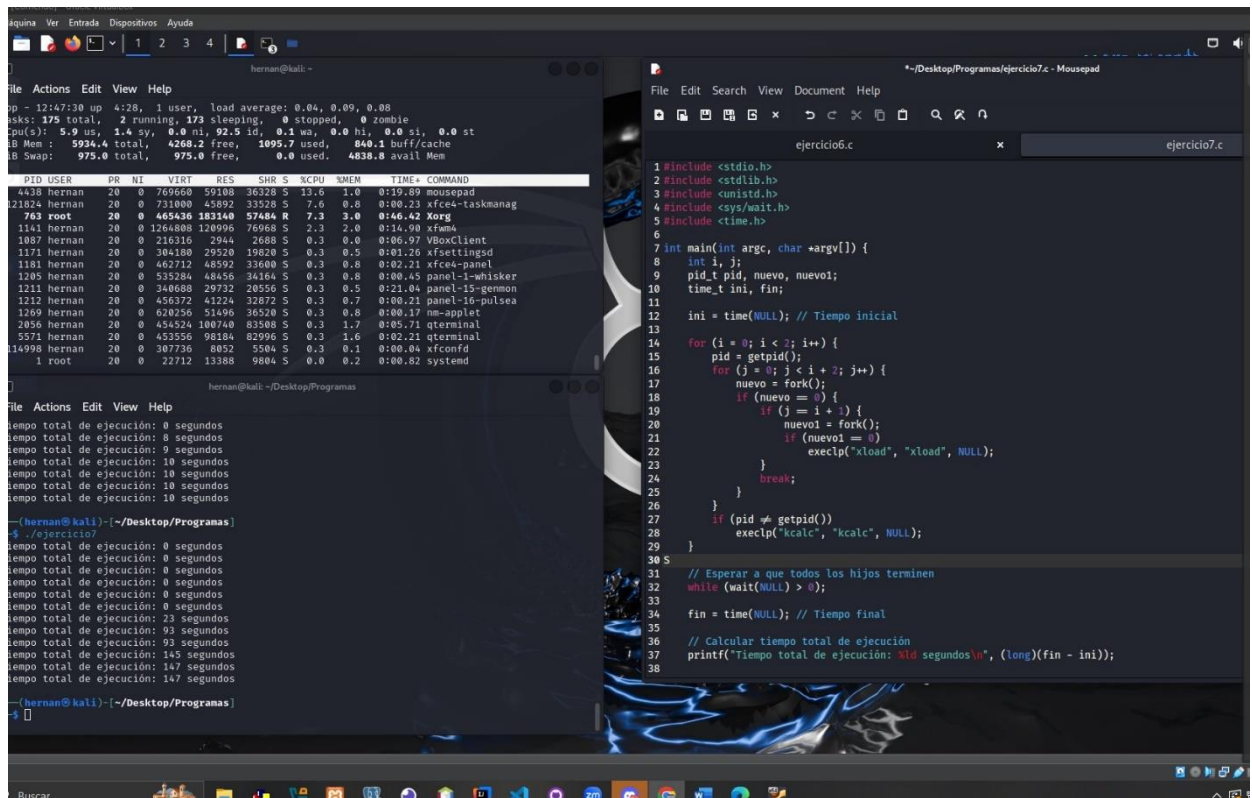
```
1 #include <sys/types.h>
2 #include <sys/wait.h>
3 #include <unistd.h>
4 #include <stdio.h>
5 #include <stdlib.h>
6 #define L1 2
7 #define L2 3
8
9 int main (int argc, char *argv[]) {
10     int cont1, cont2;
11     pid_t pid;
12     for (cont2= 0; cont2< L2; cont2++)
13     {
14         for (cont1= 0; cont1< L1; cont1++)
15         {
16             pid= fork();
17             if (pid== 0)
18                 break;
19         }
20         if (pid!= 0) break;
21     }
22     printf ("Soy el proceso de PID %d y mi padre tiene %d de PID.\n",getpid(), getppid());
23     if (pid!= 0)
24         for (cont1= 0; cont1< L1; cont1++)
25             printf ("Fin del proceso de PID %d.\n", wait (NULL));
26     return 0;
27 }
28
```

## Ejercicio 7

Dibuja la jerarquía de procesos que resulta de la ejecución del siguiente código. Introduce las llamadas al sistema wait para que una vez generado el árbol de procesos los hijos sean esperados por sus respectivos padres. Además, haz que se informe de los tiempos de ejecución de las aplicaciones xload y kcalc que se generen así como del tiempo total de ejecución. Para calcular el tiempo transcurrido, puedes utilizar la función time() de la librería estándar time.h. La llamada time(NULL) devuelve los segundos transcurridos desde las 00:00:00 del 1/1/1970 hasta el instante de la llamada.

```
int main (int argc, char *argv[]) {
    int i, j;
    pid_t pid, nuevo, nuevo1;
    time_t ini, fin;
    for (i= 0; i< 2; i++){
        pid= getpid();
        for (j= 0; j< i+2; j++){
            nuevo= fork();
            if(nuevo== 0){
                break;
            }
            nuevo1= fork();
            if(nuevo1== 0)
                execlp ("xload", "xload", NULL);
        }
        if (pid!= getpid())
            execlp ("kcalc", "kcalc", NULL);
    }
    return 0;
}
```

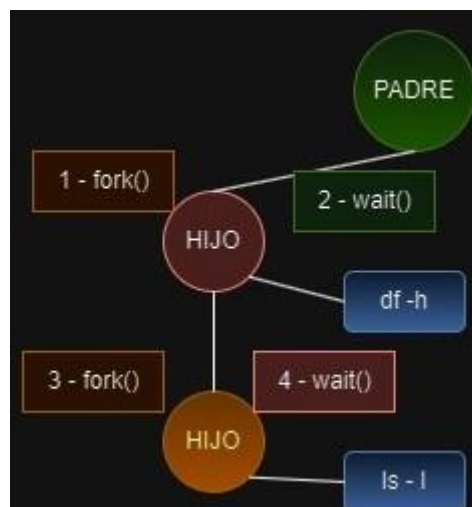




## Ejercicio 8

Dado el siguiente código. Grafique el diagrama de procesos y mencione que realiza cada uno.

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char *argv[]) {
    int i, j;
    pid_t pid;
    pid=fork();
    if(pid!=0)
    {
        printf("SOY EL PROCESO PADRE:%d (PROCESOS) Y MI PADRE ES:%d\n",getpid(),getppid());
        wait(NULL);
        //execlp ("ps", "ps","-eaf", NULL);
    }
    else
    {
        pid=fork();
        if(pid!=0)
        {
            printf("SOY EL PROCESO HIJO: %d (ESPACIO EN DISCO) Y MI PADRE ES:%d\n",getpid(),getppid());
            execlp ("df","df","-h", NULL);
            wait(NULL);
        }
        else
        {
            printf("SOY EL PROCESO HIJO:%d (LS) Y MI PADRE ES:%d\n",getpid(),getppid());
            execlp("ls","ls","-l",NULL);
            exit(0);
        }
    }
    return 0;
}
```



## Ejercicio 9

Dado el siguiente código. Grafique el diagrama de procesos.  
Que diferencia encuentra con el diagrama del ejercicio 8.

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

int main (int argc, char *argv[]) {
    int i, j;
    pid_t pid;
    pid=fork();
    if(pid==0)
    {
        printf("SOY EL PROCESO HIJO:%d (PROCESOS) Y MI PADRE ES:%d\n",getpid(),getppid());
        execlp ("ps", "ps", NULL);
        exit(0);
    }
    else
    {
        pid=fork();
        if(pid==0)
        {
            printf("SOY EL PROCESO HIJO: %d (ESPACIO EN DISCO) Y MI PADRE ES:%d\n",getpid(),getppid());
            execlp ("df", "df", "-h", NULL);
            exit(0);
        }
        else
        {
            printf("SOY EL PROCESO PADRE:%d (LS) Y MI PADRE ES:%d\n",getpid(),getppid());
            execlp("ls", "ls", "-l", NULL);
            wait(NULL);
        }
    }
    wait(NULL);
    return 0;
}
```



La diferencia que encuentro entre los ejercicios 8 y 9 es que el ej.8 se forman en cascada, y el ej.9 tipo árbol.