## 3. Taller

**1.** Escriba en consola “man syscalls” y responda: ¿Qué contiene esta llamada al sistema?

Contiene la lista de los diferentes llamados al sistema que podemos usar para acceder a algunos servicios que solo el sistema tiene acceso cuando está en modo

|kernell. Entre ellas podemos encontrar el fork(), exit(), read() y open() por ejemplo.

**2.** Explique qué hace el siguiente programa:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 | #include <syscall.h> #include <unistd.h> #include <stdio.h> #include <sys/types.h>  int main(void) {  long ID1, ID2;  */\*--------------------------------\*/*  */\* DIRECT SYSTEM CALL \*/*  */\* SYS\_getpid(func no. is 20) \*/*  */\*--------------------------------\*/*  ID1 = syscall(SYS\_getpid);  printf("syscall(SYS\_getpid) = %1d\n", ID1);   */\*-----------------------------------\*/*  */\* "libc" WRAPPED SYSTEM CALL \*/*  */\* SYS\_getpid(func no. is 20) \*/*  */\*-----------------------------------\*/*  ID2 = getpid();  printf("getpid() = %1d\n", ID2);  return 0; } |

Este programa da una muestra del funcionamiento del llamado al sistema “*getpid*”. Este devuelve la identificación del proceso que realizó la llamada. En la línea 12, se usó el llamado de manera indirecta con la función syscal. En Cambio, en la línea 19 se realizó el llamado con la función de la librería estándar getpid

**3.** Ejecute el comando “man execl”. Liste las funciones e indique qué hace cada una de ellas.

La familia de funciones del ejecutor reemplaza la imagen del proceso actual con un nueva imagen de proceso. Las funciones descritas en esta página del manual son front-ends para la función execve (2).  
  
 int execl(const char \*path, const char \*arg, ...);  
 int execlp(const char \*file, const char \*arg, ...);  
 int execle(const char \*path, const char \*arg , ..., char \* const envp[]);  
 int execv(const char \*path, char \*const argv[]);  
 int execvp(const char \*file, char \*const argv[]);

**4.** Compile y ejecute el siguiente programa:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 | #include <stdio.h> #include <stdlib.h> #include <sys/types.h> #include <unistd.h>  int main(void) {  int fd;  pid\_t pid;  int num;  if ((pid = fork()) < 0) {  perror("fork falló");  exit(-1);  } else if (pid == 0) {  for (num=0; num<20; num++) {  printf("hijo: %d\n", num);  sleep(1);  }  } else {  for (num=0; num<20; num+=3) {  printf("padre: %d\n", num);  sleep(1);  }  } } |

**Responda:**

* ¿Qué significa el retorno de la función fork?

Debido a que la a función ”fork()”, crea un proceso hijo, esta puede retornar valores enteros diferentes dependiendo de qué proceso terminó su ejecución. Así, fork() retorna un entero con la identificación del proceso (PID) diferente de cero (0) cuando es el padre quien ha terminado su ejecución. De la misma forma, cuando es el hijo quien ha terminado, esta función retorna cero (0). Por último, fork() retorna un -1 en caso de no poder crear un proceso hijo.

* + ¿Cuál es la salida esperada en pantalla?

padre: 0  
hijo: 0  
padre: 3  
hijo: 1  
padre: 6  
hijo: 2  
padre: 9  
hijo: 3  
padre: 12  
hijo: 4  
padre: 15  
hijo: 5  
padre: 18  
hijo: 6  
hijo: 7 ...

hijo: 8  
hijo: 9  
hijo: 10  
hijo: 11  
hijo: 12  
hijo: 13  
hijo: 14  
hijo: 15  
hijo: 16  
hijo: 17  
hijo: 18  
hijo: 19

Los procesos padre son los primeros en terminar su ejecución pero se imprimen menos ya que estos aumentan la variable num en 3 por cada proceso padre terminado.

* + ¿Cómo es posible que la sentencia printf reporte valores diferentes para la variable num en el hijo y en el padre?

Esto se debe a que en la línea 19, la variable num está siendo aumentada en 3 cuando el proceso que ha terminado es un padre. Esto tampoco afecta el conteo para los hijos ya que cada proceso que crea la función fork() tendrá su propia variable num independiente. Llegado el caso en el que en ambos for, la variable num aumente de manera igual, la sentencia printf reportará valores iguales para la variable num en el hijo y en el padre.

**5.** Modifique los Códigos del 3 al 8 (añadiendo donde sea necesario un llamado a wait) para que los programas se ejecuten como realmente se esperaría.

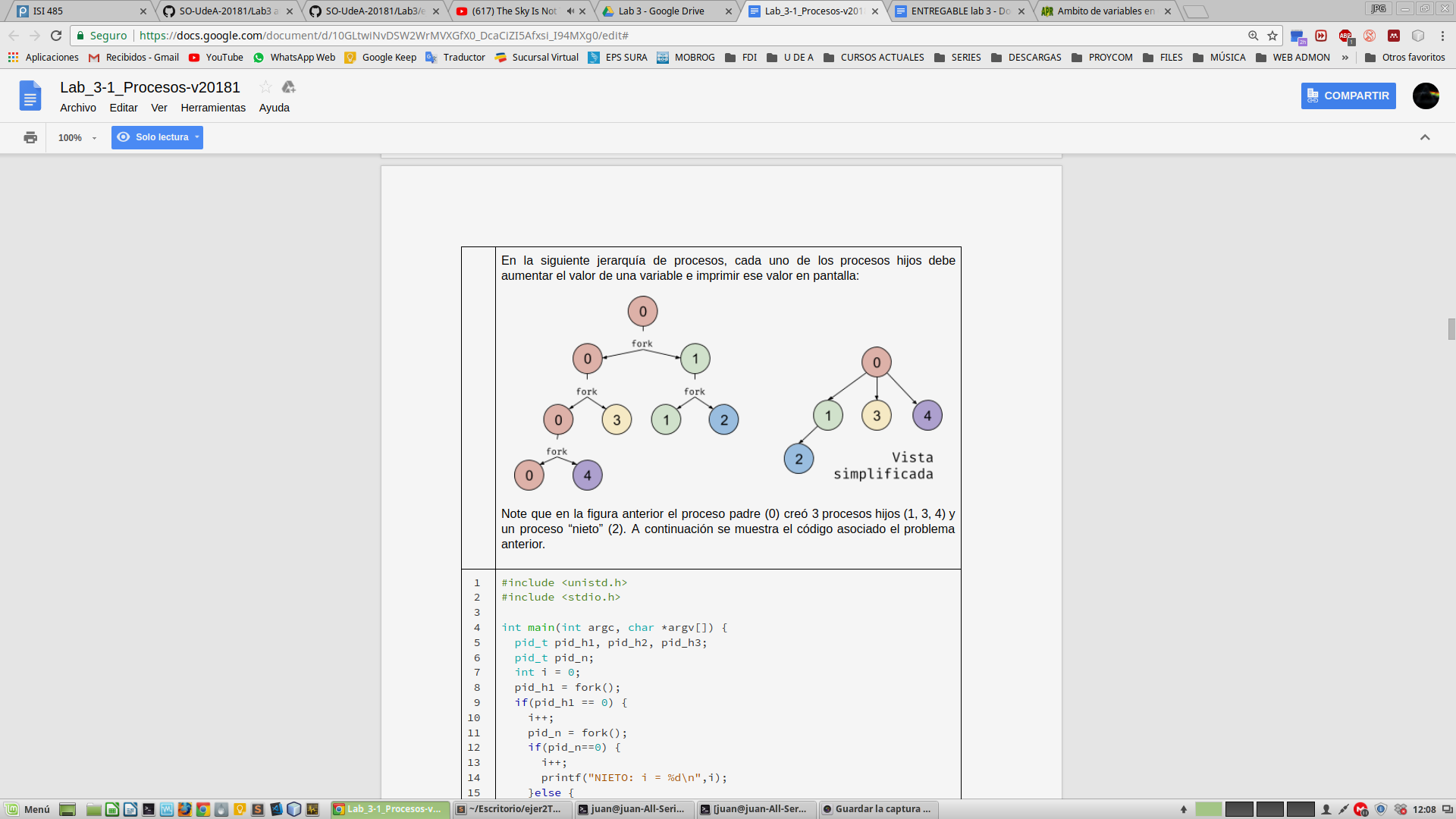
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24  25  26 | #include <stdio.h> #include <stdlib.h> #include <sys/types.h> #include <unistd.h> #include <sys/wait.h>  int main(void) {  pid\_t pid;  int num;  if ((pid = fork()) < 0) {  perror("fork falló");  exit(-1);  } else if (pid == 0) {  for (num=0; num<20; num++) {  printf("hijo: %d\n", num);  sleep(1);  }  } else {  for (num=0; num<20; num+=3) {  wait(&pid);  printf("padre: %d\n", num);  sleep(1);  }  }  wait(&pid); } |

**6.** Dado el siguiente código:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 | #include<stdio.h> #include<unistd.h> main() {  printf("Hola ");  fork();  printf("Mundo");  fork();  printf("!"); } |

**Resuelve:**

* Sin ejecutarlo dibuje la jerarquía de procesos del programa y determine cuál es la posible salida en pantalla.



Salida en pantalla: Hola Mundo!Hola Mundo!Hola Mundo!Hola Mundo!

* Compile y ejecute el programa. ¿Es la salida en consola la que usted esperaba? ¿Cuál puede ser la razón de esto? (ayuda: función fflush: fflush(stdout);)

En efecto, es la salida en pantalla lo que se esperaba ya que el proceso principal imprime : hola mundo! y cada proceso creado es una copia del original, por tanto y como se ve en la jerarquía, se debe mostrar este resultado 4 veces, una vez por el proceso original y las otras 3 veces por los procesos hijos. Para que este resultado solo aparezca una vez, se deben utilizar sentencias de control if o switch cases para filtrar los resultados del fork().

* Modifique el programa de tal manera que se creen exactamente 3 procesos, el padre imprime “Hola”, el hijo imprime “Mundo” y el hijo del hijo imprime “!”, exactamente en ese orden.

#include <unistd.h>  
#include <stdio.h>  
#include <stdlib.h>  
#include <sys/types.h>  
#include <sys/wait.h>  
  
int main(int argc, char \*argv[]) {  
 pid\_t pid\_h;  
 pid\_t pid\_n;  
 pid\_h = fork();  
 if(pid\_h == 0) {  
 pid\_n = fork();  
 if(pid\_n==0) {

printf("!"); //nieto   
 }else printf("Mundo"); //hijo   
 }else printf("Hola "); //padre  
 return 0;  
}

**7.** Dado el siguiente código:

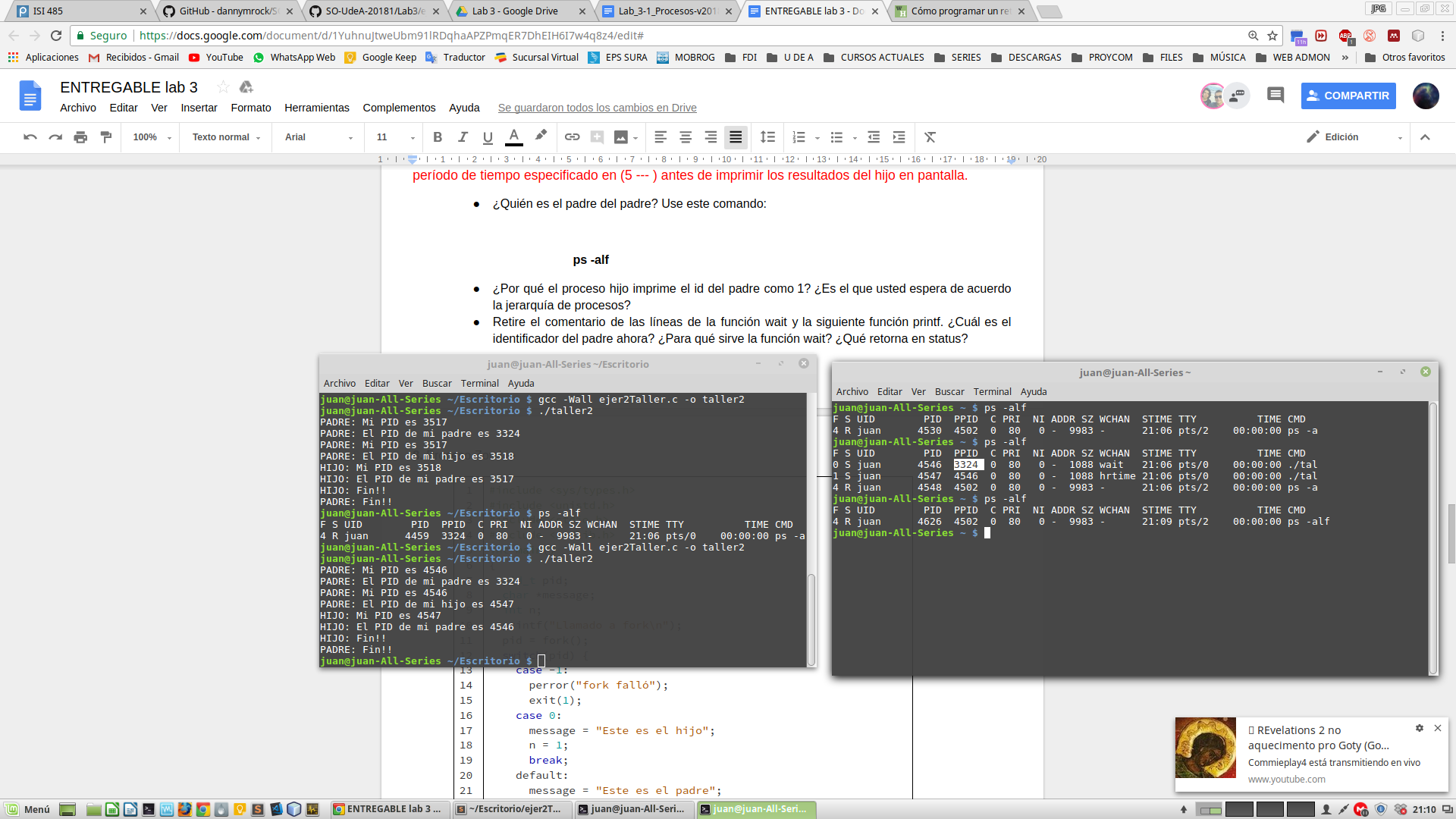
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26  27 | #include <unistd.h> #include <stdio.h> #include <stdlib.h> #include <sys/types.h> #include <sys/wait.h> main() {  pid\_t pid;  int status;  printf("PADRE: Mi PID es %d\n", getpid());  printf("PADRE: El PID de mi padre es %d\n", getppid());  pid = fork();  if(pid == 0){  sleep(5);  printf("HIJO: Mi PID es %d\n", getpid());  printf("HIJO: El PID de mi padre es %d\n", getppid());  printf("HIJO: Fin!!\n");  }  else {  printf("PADRE: Mi PID es %d\n", getpid());  printf("PADRE: El PID de mi hijo es %d\n", pid);  *// wait(&status);*  *// printf("PADRE: Mi hijo ha finalizado con estado %d\n", status);*  printf("PADRE: Fin!!\n");  }  exit(0); } |

**Responda:**

* ¿Cuál es la principal función de sleep en el código anterior?

El sleep() en este código hace que el programa se detenga cuando el proceso es un hijo durante período de tiempo especificado en (5 --- ) antes de imprimir los resultados del hijo en pantalla.

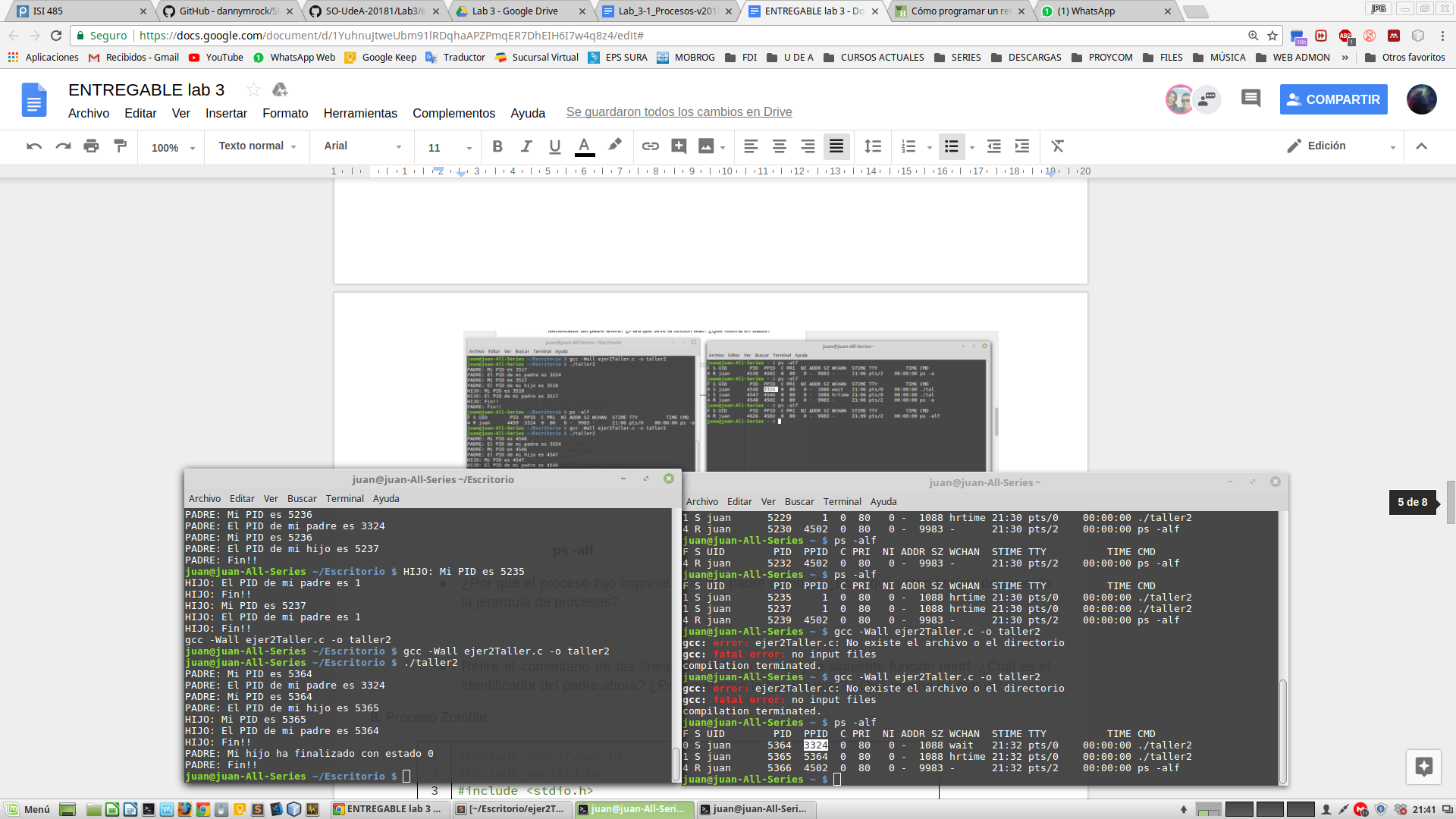
* ¿Quién es el padre del padre? Use este comando:



**ps -alf**

* ¿Por qué el proceso hijo imprime el id del padre como 1? ¿Es el que usted espera de acuerdo la jerarquía de procesos?

* Retire el comentario de las líneas de la función wait y la siguiente función printf. ¿Cuál es el identificador del padre ahora? ¿Para qué sirve la función wait? ¿Qué retorna en status?

1.El identificador ahora no es 1.  
2.La función wait() sirve para que el padre espere a que un hijo termine y así continuar su ejecución.  
3.Retorna un 0  
 

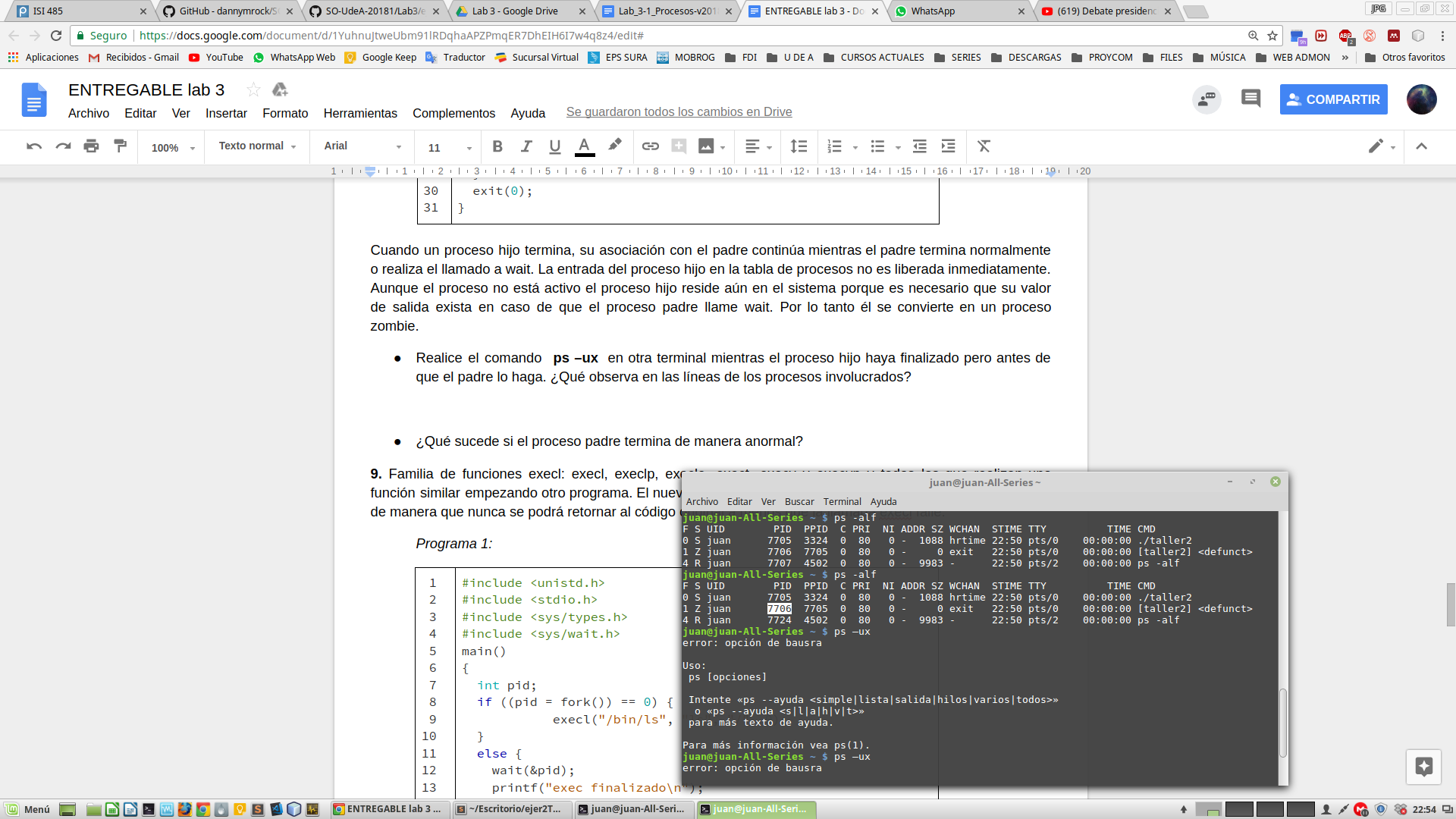
**8.** Proceso Zombie

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 | #include <sys/types.h> #include <unistd.h> #include <stdio.h> #include <stdlib.h> int main() {  pid\_t pid;  char \*message;  int n;  printf("Llamado a fork\n");  pid = fork();  switch(pid) {  case -1:  perror("fork falló");  exit(1);  case 0:  message = "Este es el hijo";  n = 1;  break;  default:  message = "Este es el padre";  n = 30;  break;  }  for(; n > 0; n--) {  printf("n=%d ",n);  puts(message);  sleep(1);  }  exit(0); } |

Cuando un proceso hijo termina, su asociación con el padre continúa mientras el padre termina normalmente o realiza el llamado a wait. La entrada del proceso hijo en la tabla de procesos no es liberada inmediatamente. Aunque el proceso no está activo el proceso hijo reside aún en el sistema porque es necesario que su valor de salida exista en caso de que el proceso padre llame wait. Por lo tanto él se convierte en un proceso zombie.

* Realice el comando **ps –ux** en otra terminal mientras el proceso hijo haya finalizado pero antes de que el padre lo haga. ¿Qué observa en las líneas de los procesos involucrados?

Se observa que efectivamente el estado del proceso hijo está en Z (zombie) mientras que el padre no haya finalizado.

https://github.com/OS-classroom/manejo-de-archivos-en-c-ebedoyaalzate.git

* ¿Qué sucede si el proceso padre termina de manera anormal?

**9.** Familia de funciones execl: execl, execlp, execle, exect, execv y execvp y todas las que realizan una función similar empezando otro programa. El nuevo programa empezado, sobrescribirá el programa existente, de manera que nunca se podrá retornar al código original a menos que la llamada execl falle.

*Programa 1:*

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 | #include <unistd.h> #include <stdio.h> #include <sys/types.h> #include <sys/wait.h> main() {  int pid;  if ((pid = fork()) == 0) {  execl("/bin/ls", "ls", "/", 0);  }  else {  wait(&pid);  printf("exec finalizado\n");  } } |

*Programa 2:*

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 10 | #include <unistd.h> #include <stdio.h> #include <stdlib.h> int main() {  printf("Corriendo ps con execlp\n");  execlp("ps", "ps", "-ax", 0);  printf("Echo.\n");  exit(0); } |

**Resuelve:**

* ¿Qué es lo que hace cada uno de los programas anteriormente mostrados?

El primer programa imprime en pantalla el contenido del directorio raíz del sistema, tanto carpetas como archivos (incluyendo los ocultos).  
El segundo programa muestra una lista de los procesos del sistema, con su id, nombre de consola en la que el usuario inició sesión, estado del proceso, tiempo que este lleva corriendo, y el comando.

**10.** Haga un programa que cree 5 procesos donde el primer proceso es el padre del segundo y el tercero, y el tercer proceso a su vez es padre del cuarto y el quinto:

El programa debe tener la capacidad de:

* Verificar que la creación de proceso con fork haya sido satisfactoria.
* Imprimir para cada proceso su id y el id del padre.
* Imprimir el id del proceso padre del proceso 1.
* A través de la función system imprimir el árbol del proceso y verificar la jerarquía (pstree).

>> Ejer10.c (adjunto)

**11.** Codifique un programa que haga lo siguiente:

* Cree 3 procesos diferentes.
* Cada uno de los procesos hijos, calculará por recursión el factorial de los enteros entre 1 y 10, imprimirá los resultados en pantalla y terminará.
* El mensaje impreso por cada proceso debe ser lo suficientemente claro de modo que sea posible entender cuál es el proceso hijo que está ejecutando la operación factorial.
* Una salida tentativa se muestra a continuación (esto no quiere decir que el orden en que se despliegue sea el mismo):

**HIJO1: fact(1) = 1  
HIJO2: fact(2) = 1  
HIJO2: fact(2) = 2  
HIJO1: fact(2) = 2**

* El proceso padre tiene que esperar a que los hijos terminen.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23  24  25  26  27  28  29  30  31  32  33  34  35  36 | #include <stdio.h>  #include <error.h>  #include <stdlib.h>  #include <fcntl.h>  #include <unistd.h>  #include <sys/wait.h>  int factorial(int num, int i, int j);  int main(){  int i, j, p1, p2;  for(j=1;j<=2;j++){  if(j==1){  if((p1=fork())==0){  i = factorial(10, 10, j);  }  } else{  if((p2=fork())==0 && (p1==fork())!=0){  i = factorial(10, 10, j);  }  }  }  i=i+1;  wait(&p1);  wait(&p2);  }  int factorial(int num, int i, int j){  if (num==0){  return 1;  } else{  num=num\*factorial(num-1, i-1, j);  printf("HIJO %d: fact(%d)=%d\n",j, i, num);  return num;  }  } |

**12.** Realice un programa llamado *ejecutador* que lea de la entrada estándar el nombre de un programa y cree un proceso hijo para ejecutar dicho programa.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23  24  25 | #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <error.h>  #include <unistd.h>  #include <sys/wait.h>  int main(){  char entrada[50];  char programa[52]="./";  int p1;  printf("Ingrese el nombre del programa que desea ejecutar:\n");  scanf("%s", entrada);  strcat(programa, entrada);  if((p1=fork())==0){  system(programa);  } else if((p1=fork())<0){  perror("error al crear el fork()\n");  } else{  wait(&p1);  printf("DONE!\n");  }  sleep(2);  } |

**13.** Dado el siguiente fragmento de código:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 | #include<stdio.h> #include<error.h> #include<stdlib.h> #include<fcntl.h> int main(int argc, char \*argv[]) {  int fd;  int pid;  char ch1, ch2;  fd = open("data.txt", O\_RDWR);  read(fd, &ch1, 1);  printf("En el padre: ch1 = %c\n", ch1);  if ((pid = fork()) < 0) {  perror("fork fallo");  exit(-1); *//Sale con código de error*  } else if (pid == 0) {  read(fd, &ch2, 1);  printf("En el hijo: ch2 = %c\n", ch2);  } else {  read(fd, &ch1, 1);  printf("En el padre: ch1 = %c\n", ch1);  }  return 0; } |

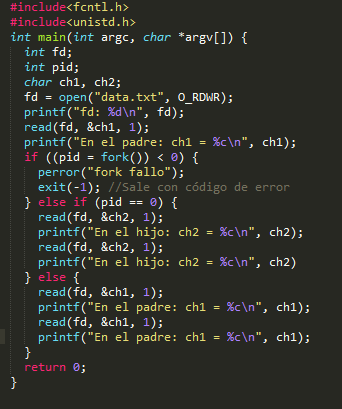
* Cree manualmente el archivo *data.txt* con el siguiente contenido:

**hola**

* Ejecute el programa, capture en pantalla la salida producida. ¿Por qué el programa produce la salida vista? ¿Qué sucede con un padre que abre un archivo, lo hereda?



El hijo sí hereda el archivo abierto por el padre, no muestra la salida “hola” completa porque en el programa se realizan 3 read(), que leen los 3 primeros caracteres en el arreglo, por lo tanto no se genera la salida “a”.



Como se muestra en el programa anterior, al añadir algunas líneas de código, la salida deberá de ser completa:



## 4. Referencias

* Tim Jones, 2010. Kernel command using Linux system calls. <https://www.ibm.com/developerworks/linux/library/l-system-calls/>
* Linux Syscall Reference. https://syscalls.kernelgrok.com