Process Exercises

Exercise 1.

Given the following program, describe what the program prints and explain the reason:

```
int main () {

  int a=3,pid;

  printf ("Hola\n");

  pid=fork();

  printf ("Adios\n");

}
```

SOLUTION

Solución: forksimple-1.c

Imprime:

```
Hola

Adios

Adios
```

The reason is that after the fork there are 2 processes (father and son) and, after the fork both do the same. Thus, both print “Adios”.

Exercise 2.

Write a C program that creates two processes: parent and child. Both will print I am the father or the son, as the case may be, and at the end they will print Adios and his pid and his father's pid. Important: the father should get some sleep to give the son time to finish and see the pid well. What happens if it doesn't go to sleep?

SOLUTION

Solución: forksimple-2.c

If the sleep () is not put and the parent ends before the child it is seen that the new parent of the child is process 1 (init). That is because, when the father dies, init adopts the son so that there is no zombie.

Exercise 3.

Explain what is the ouput to the screen when the following program is executed? What is the reason?

```
main () {
  int a=3,pid;

  printf ("Hola\n");
  pid=fork();
  if (pid==0){ //hijo
    a++;
    printf ("hijo: %d\n",a);
  }
  else {
    sleep(1);
    printf ("padre: %d\n",a);
  }

}
```

SOLUTION

Solución: forksimple-3.c

Output:
Hola
hijo: 4
padre: 3

The reason is that from fork () each process has its own copy of all variables (memory space has been copied from parent to child). From there, the operations that a process does on a variable will not be seen by the other process.

Exercise 4.

Explain the program of the next example ¿How many times is written each word?

```
#include <stdio.h>
main (){
  printf ("Hola\n");
  fork ();
  printf ("Uno\n");
  fork ();
  printf ("Dos pid=%d\n",getpid());
}
```

¿In which order could they appear?

SOLUTION

Solución: forksimple-4.c

An example of execution:

```
Hola
Uno
Dos pid=4530
Uno
Dos pid=4532
Dos pid=4533
Dos pid=4534
```

The reason is that every time a fork () is made, two processes appear. Therefore after the first fork () there are 2 processes, which in turn fork (), resulting in 4 processes (22).

The order may vary depending on the order of execution of the processes in each case, which depends on the system scheduler. So another example output shows:

```
Hola
Uno
Uno
Dos pid=4628
Dos pid=4630
Dos pid=4631
Dos pid=4629
```

Exercise 5.

When you execute the program of the next example:

```
main () {
int i,pid;

for (i=1; i<=2; i++){
  pid = fork();
  if ( pid==0) {
    printf ("Soy el hijo %d\n", i);
    exit(0);
  }
}
}
```

¿How many times would you see the message "Soy el hijo n" ? Explain your answer..

SOLUCIÓN

Solución: forksimple-5.c

The output is:

Soy el hijo 1
Soy el hijo 2

The reason is that although 2 fork () are made, in each case only one child process is executed. And it doesn't fork () because it prints a message and ends with exit ().

Exercise 6.

¿How many processes are created by the next program?

```
void fhijo(int i){
    printf ("Hijo %d, pid =%d, ppid%d\n",i,
getpid(),getppid());
}
```

```
int main (){
int i,pid;

for (i=0; i<4; i++){
  pid=fork();
  if (pid==0){ //Hijo
    fhijo(i);
  }
  else
    printf ("El padre ha creado %d hijos \n",i);
} // fin for

} // fin main
```

¿Modify it to créate exactly 4?

SOLUTION

Solución: forksimple-6.c

This program creates 16 children, as all children do fork() in the for iterations.

To create only 4 children, we have to modify it as following:

```
void fhijo(int i){
    printf ("Hijo %d, pid =%d, ppid%d\n",i,
getpid(),getppid());
   exit (0);
}
```

Exercise 7.

Write a C program that creates a child program that executes a command received as an argument. The father ends up without doing more.

SOLUTION

Solución: exec-comando-1.c

```
int main(int argc, char *argv[])
{
 int i,pid;

 if (argc < 2){
   printf("Usage: exec-comando <comando>\n");
   exit(-1);
 }

 pid=fork();
 if(pid==0){ //crear hijo
        printf ("Hijo creado, va a ejecutar el comando\n");
        execvp (argv[1], &argv[1]);
        printf ("ERROR, aqui solo se llega si ha fallado el exec\n");
  }
  wait (NULL);
  printf ("FIN  del padre\n");
 } //cierre del main
```

Exercise 8.

You should:

a) Write a C program, called "child.c" that generates an executable "child". This program must print on the screen a number that it receives as an argument and its pid, and fall asleep for 2 seconds.

b) Write a program "parent.c" that executes 3 processes "child" of the above and wait for them to finish.

SOLUTION

Solución: hijo.c. padre.c

```
/// HIJO
int main(int argc,char *argv[])
{
 int pid;
 int cont=0;

   pid=getpid();
   printf("Soy hijo %s con Pid: %d  \n",argv[1],pid);
   sleep(2);
}

/// PADRE
#define NUMHIJOS 3
int main()
{
 int i,pid;
 char num[10];

  for(i=1;i<=NUMHIJOS;i++) {
      sprintf(num,"%d",i);
      pid=fork();
      if(pid==0) // hijo
        execlp("./hijo","hijo",num,NULL);
  }
  for(i=1;i<=NUMHIJOS;i++)
    wait (NULL);

 } //cierre del main
```

Exercise 9.

Given the next program `crear_procesos1`:

```
#include <stdio.h>
main (int argc, char **argv)
{
    int num_proc, i, pid;

    num_proc = atoi (argv[1]);
    for (i=o; i<num_proc; i++) {
        pid = fork();
        if (pid == 0) {
            write (1, "Soy un proceso", 14); }
    }
```

```
}
```

An the next one `crear_procesos2`:

```
#include <stdio.h>
main (int argc, char **argv)
{
    int num_proc, i, pid, status;

    num_proc = atoi (argv[1]);
    for (i=o; i<num_proc; i++) {
        pid = fork();
        if (pid == 0) {
            write (1, "Soy un proceso", 14); }
        else
            while(pid != wait (&status));
             //bucle esperando la finalización del hijo creado
    }
}
```

a) Draw a schematic showing the process hierarchy if the user executes the create_processes1 3 and create_process2 3 commands successively.

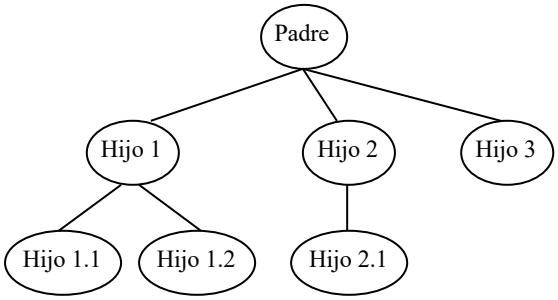
b) Can zombie processes remain?

c) What would happen if the user executed create_processes1 50? Why? Modify the program so that this does not happen.

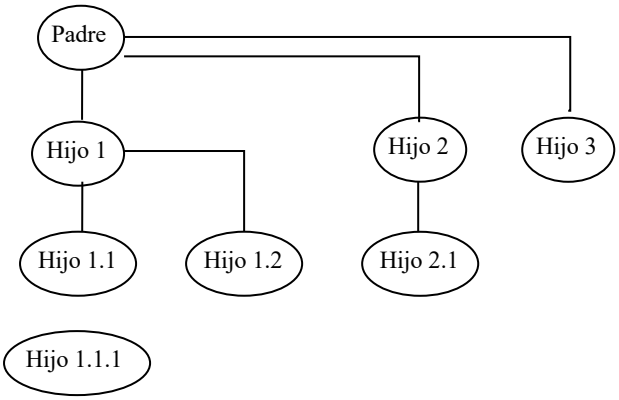
SOLUTION

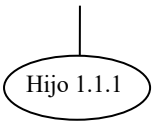
a)

create_processes 1 3



create_processes 2 3





The number of processes that are created is the same. The difference is that the call to "create_processes2" guarantees that the process creation order is like this:

Father - Son1 - Son1.1 - Son1.1.1 - Son1.2 - Son2 - Son2.1 - Son3

This implies that, for example, when Son1.2 is created, it is because both son1.1 and son1.1.1 have already finished.

In the case of "create_processes1" we cannot assure anything about the order of creation of the processes.

b)

Considering the definition of a zombie process: it is the process whose execution has ended and its parent process (without having died yet) did not wait () waiting for the child.

According to this definition, if "create_processes1" is executed, there may be zombie processes, since each parent process does not wait () waiting for the completion of the children it creates. It cannot be guaranteed that there will always be zombie processes, since if the created children die after the parent who created them, they are not considered zombie processes.

With the execution of "create_processes2" it is certain that there will be no zombie processes, since when a process creates a child it always waits for it to finish with the call to wait ().

c)

What would happen if create_processes1 50 were executed is that the system should create 250 processes. Such a number of processes does not fit in the process table of the operating system, so the system would hang waiting for entries to be released to continue execution. Since there would be no entries for anyone, it would not be possible to run maintenance operations, which require new processes. It would therefore be necessary to reboot the system.

To prevent this from happening, it is not worth making the parents wait for the child processes. Why? Because the resulting hierarchy of processes would be completely different and the behavior of the process would be very different. In this case, only one child process would be created and nothing else because the parent would wait for that child to finish.

Una solución sencilla es hacer un cálculo previo del número de procesos a crear y ver si sobrepasan el máximo permitido. En este caso se devuelve un error y ni siquiera se empieza el bucle de creación de procesos.

A simple solution is to make a preliminary calculation of the number of processes to create and see if they exceed the maximum allowed. In this case an error is returned and the process creation loop is not even started.

```
num_proc = atoi (argv[1]);
if  ((2**num_proc) > MAX_PROC){
    printf ("No se puede crear tantos procesos \n");
    exit (-1);
    }
```

This solution allows you to avoid filling the process table by cutting the depth of the created process hierarchy.

Exercise 10.

Given the following program, answer the questions below:

```
#include <stdio.h>
#include <stdlib.h>
main() {
   int pid,i, m=10;
   int tiempoinicial, tiempoactual;
   tiempoinicial = time(NULL);//time devuelve el tiempo actual en segundos
   tiempoactual = time(NULL) - tiempoinicial;
   printf("%d:Inicio del programa \n",tiempoactual );
   for(i=0; i<3; i++) {
       pid=fork();
       sleep(1);
       switch(pid) {
       case -1:
               perror("Error en la creación de procesos");
               exit(-1);
           case 0:
               m++;
               tiempoactual = time(NULL) - tiempoinicial;
               printf("%d:Hijo %d m=%d\n",tiempoactual, i, m);
               sleep(2);
               exit(0);
           default:
               tiempoactual = time(NULL) - tiempoinicial;
               printf("%d:Creado el proceso %d\n", tiempoactual, i);
               if( i%2 == 0 ) {
                   wait(NULL); //wait espera que finalice un hijo cualquiera
```

```
                tiempoactual = time(NULL) - tiempoinicial;
                printf("%d:Finalizó un proceso, valor de m=%d\n",
                        tiempoactual,m);
            } //fin if
        } //fin switch
    } //fin for
    wait(NULL);
    tiempoactual = time(NULL) - tiempoinicial;
    printf("%d:Finalizó un proceso, valor de m=%d",tiempoactual, m);
} //fin main
```

a) Write the messages that are written on the screen and at what time, assuming that the 'Program start' message appears at time 0.

b) How many 'm' variables are created in memory?

SOLUTION

a) *0:Inicio del programa*
1:Hijo 0 m=11
1:Creado el proceso 0
3:Finalizó un proceso, valor de m=10
4:Hijo 1 m=11
4:Creado el proceso 1
5:Hijo 2 m=11
5:Creado el proceso 2
6:Finalizó un proceso, valor de m=10
7:Finalizó un proceso, valor de m=10

b) 4 variables m. (One for the father and one per children).

Exercise 11.

Write a program to create 5 children.

SOLUTION

Solución: crear5hijos.c

```
/*Este programa crea 5 hijos*/

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

```
int main (){
  int i,pid;
  for (i=1; i<=5; i++){
    pid=fork ();
    if ( pid==0){
       printf ("Hijo %d ( pid=%d)\n",i,getpid());
       exit (0); //return(0);
    }
    else;
     // El padre no tiene nada mas que hacer
  }
  /* El padre finaliza sin esperar a los hijos que se quedan sin padre (zombies)
}
```

Exercise 12.

Write a program to create 5 children and 3 grandchildren.

SOLUTION

Solución: crear5hijos3nietos.c

```
// Este programa crea 5 hijos y 3 nietos

#include <stdlib.h>
#include <stdio.h>
int main (){
  int i,j,pid;
  for (i=1; i<=5; i++){
    pid=fork ();  //Creo los hijos
    if ( pid==0){
       printf ("Hijo %d ( pid=%d)\n",i,getpid());
       for (j=1; j<=3; j++){
         pid=fork ();  //Creo los nietos
         if (pid==0){
           printf (" Nieto %d ( pid=%d) del hijo %d (pid=%d)\n",
              j,getpid(),i,getppid());
      exit (0);  //fin de los nietos
         }
       }
       for (j=1; j<=3; j++)  //Los hijos esperan el fin de sus hijos(los nietos)
         wait (NULL);
       exit (0); //return(0);  //fin de los hijos
    } // fin del if del hijo
  } //fin for de creación de 5 hijos
  for (i=1; i<=5; i++)  //El padre espera el fin de sus hijos
     wait (NULL);
}
```

Exercise 13.

Write a program with a parent and child that does the following:

• The child asks the user for a number on the keyboard. When you have read the number it ends.

• The father writes on the screen a sequence of numbers increasing from the last number entered by keyboard (the number must be between 0 and 255). You will be writing the sequence until the child reads another number.

SOLUCIÓN

```
/* En este programa el hijo pide un número y el padre escribe una secuencia
creciente desde el último número introducido (el número tiene que estar entre 0 y
255) hasta que el padre vuelva a leer otro número.
 * Es una forma de ver como se pueden hacer 2 acciones en paralelo y con
comunicación entre los 2 procesos que las llevan a cabo, uno pidiendo números y
otro escribiendo la secuencia que empieza en el número introducido. */

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

/* El hijo pide un número y se lo pasa al padre cuando se muere*/
hijo(){
int numleido=1;
       printf("Introduce un numero: (de 0 a 255)");
       scanf("%d", &numleido);
       exit(numleido);
}

main()
{
int pid,estado,hijomuerto;
int numescrito=1;
   //El padre crea un hijo para que lea un número mientras el está escribiendo la
secuencia
   pid=fork();
   if(pid==0)
       hijo();
   while(1) {
       printf("%d\n",numescrito);
       sleep(1);
       numescrito++;
       //Compruebo si el hijo se ha muerto y por tanto ha leído un número que tengo
que utilizar en la nueva secuencia
       hijomuerto=waitpid(pid,&estado,WNOHANG);
       //if(hijomuerto!=0 && hijomuerto!=-1)
```

```
        // Si el hijo ha muerto creo otro para que lea otro número
        if(hijomuerto==pid ) {
            numescrito=WEXITSTATUS(estado);
            pid=fork();
            if(pid==0)
                hijo();
        }
    }
}
```

Exercise 14.

You want to develop a high performance web server. For its development there is already a library whose header file is the following:

```
#ifndef WEBUTIL_H

#define WEBUTIL_H

struct peticion_web {

  /* Datos de la petición */

};

typedef struct peticion_web peticion_web_t;


void recibir_peticion(peticion_web_t * pet);

void enviar_fichero(peticion_web_t *);


#endif
```

The function receive_request () is a blocking function that returns control to the caller when a request is received, the data of which it leaves in the structure pointed to by pet.

The send_file () function determines the response to be sent to the client and performs the sending. This function is also blocking.

Process Exercises

Design and implement a process-based solution in which each time a request is received, a process will be created to process that request. In this way, the main process will always be attending to the arrival of new requests, except for the time of creating new auxiliary processes.

Solution

```
#include <sys/wait.h>

#include <stdlib.h>

#include "webutil.h"

int main() {

  int status;

  peticion_web_t p;

  for (;;) {

    recibir_peticion(&p);

    while (waitpid(-1, &status, WNOHANG) > 0) {}

    switch (fork()) {

      case -1: perror("No puedo procesar peticion"); break;

      case 0:

        enviar_fichero(&p);

        exit(0);

      default: /* padre */

        break;

    }

  }

  return 0;

}
```

Exercise 15.

Given an array of 4 elements, make a program that creates 1 child and 1 grandchild:

• The grandson process will do the sum: Sum1 = array [0] + array [1]

• The child will do the following operation: Subtraction2 = sum1 - array [2]

• The parent will do the following sum: Total = Sum2 * array [3]

• The grandson must pass on to the son the sum obtained and the son must pass on the father the sum obtained by him. The father will show it on the screen.

Design a program that creates the 3 processes and performs the operations in the proper order.

SOLUCIÓN

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

main (){
pid_t pid1, pid2;
int array[]={1,8,3,10};
int Suma1,Resta2,Total,status;

pid1 = fork();
switch (pid1) {
   case 0 : //hijo
       pid2 = fork();
       switch (pid2) {
           case 0: // nieto
               Suma1 = array[0]+array[1];
   printf ("suma1=%d\n",Suma1);
               exit(Suma1);
           default: //hijo
               while (pid2 !=wait (&status));
               Resta2 = WEXITSTATUS(status)-array[2];
   printf ("resta2=%d, estado=%d\n",Resta2, WEXITSTATUS(status));
               exit(Resta2);
           }
   default: //Proceso 3
       while (pid1 !=wait (&status));
       Total = array[3]* WEXITSTATUS(status);
       printf ("Total=%d,estado=%d\n", Total, WEXITSTATUS(status));
       exit(0);
   }
```

```
}
```

Exercise 16:

You want to implement a process that reads a number from the keyboard and starts typing a sequence of consecutive numbers from the entered number. Simultaneously, you must continue reading from the keyboard and the moment another entry occurs, the sequence of numbers that appear on the screen will change. An example of execution would be the following:

a) Execution in the terminal. Explanation

$> miprograma	Número introducido por teclado
27	Salida proporcionada por el programa
28	“
29	“
30	“
31	“
32	Número introducido por teclado
5	Salida proporcionada por el programa
6	“
7	“
8	Número introducido por teclado
3	Salida proporcionada por el programa
4	“
5	Número introducido por teclado
	Salida del programa. Fin del programa Espera nueva entrada

Since the reading of the keyboard and writing on the screen cannot be simultaneously in the same process, it is decided to make a program with a parent process that is continuously reading from the keyboard, if the input is an integer it launches a child process whose mission consists of write consecutive numbers starting from the one entered. When the user enters another number, the parent kills the process that was writing and launches a new child with the same task. The program ends when the user enters a 0, in this case the parent kills the running process and ends the program by displaying the literal "End of program" on the screen.

SOLUTION

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

main ( )
{
      int  contador;
      pid_t pid;

      scanf ("%d",&contador);
      while (contador)
      {
            pid = fork();
            switch (pid)
            {
                  case 0:     //estoy en el hijo
                        while (1)   printf("%d",++contador );
                  case -1:
                        perror("error al ejecutar el fork")
                  default:
                        scanf("%d", &contador);
                        kill(pid,9);
            }
      }
      printf("Fin del programa");
}
```

NOTA: To see correctly the execution of the child you have to include a sleep.

Exercise 17:

Given the followiing code, answer the questions:

```
void main()
{
      pid_t  pid;
      int i = 0;
      while(i < 3)
      {
            pid = fork();
            switch(pid)
            {
                  case -1:
                        perror("Error al crear el proceso hijo");
                        exit(-1);
                  case 0:
                        printf("Hola, soy el proceso %d \n", getpid());
                        if((i % 2) == 0){
                              i++;
                        }
                        else {
                              i--;
                        }
                        break;
                  default:
                        i++;
                        break;
            }
      }
}
```

a) Show the hierarchy of processes that are created with the execution of this code.

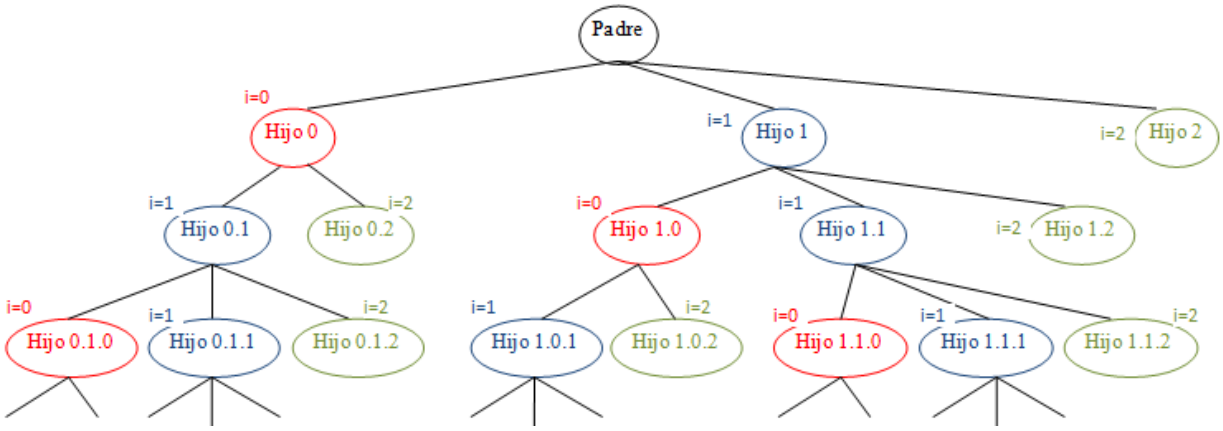
b) Indicate if there is any problem related to the execution of this code, with regard to the number of processes created.

c) Indicate what a child process would have to do to send the SIGKILL signal to its parent process.

SOLUCIÓN

Process Exercises

a)



- Child processes created initially with i = 0 (marked in red), will in turn create two new children (with i = 1, i = 2). This is because initially when created it had i = 0, but increased (i ++) before creating any children.

- Child processes created initially with i = 1 (marked in blue), will in turn create three new children (with i = 0, i = 1, i = 2). This is because initially when created it had i = 1, but decrement (i--) before creating any children.

- Child processes created initially with i = 2 (marked in green) will not create any children. This is because initially when created it had i = 2, but increased (i ++) before creating any children. In this way, by incrementing and having i = 3, it can no longer create more children due to the while condition.

b)

According to the process tree, there are branches in which children will be continuously being generated. As the number of processes that can be launched in the operating system has a limit (marked by the size set for the table that holds information of the running processes), there will come a time when the creation of children with fork () error and no more can be created.

c)

The way to send signals is through the kill system call. In which you have to specify the signal to send and the pid of the process to whom to send it. In our case, to send the signal from a child to the parent process, the way to obtain that pid is with the getppid system call.

```
……

case 0:

        printf("Hola, soy el proceso %d \n", getpid());
```

```
if((i % 2) == 0){
        i++;
}
else {
        i--;
}
break;
kill ( getppid(), SIGKILL );
```

……

Exercise 18.

a) Write a program where a parent process creates a child and waits for it. The child prints "I am the child", its pid, and exits. Analyse the original program: the parent process uses fork(), wait() and exit() system calls to create a child process. It prints ("Parent process - pid). Understand each one of the system calls.

b) Modify the original program allowing the child process to replace its text section (code) by pwd. Hint: use **execlp** function**.** This function has to use the pwd command located in /bin/pwd. The parent process has to wait using the wait() call. Additional information about **execlp** can be obtained in:
Link1 execlp
Link2 execlp

c) Block the child execution during 10 seconds by means of the sleep command (the child process executes this commands **before** execlp). Use the **pstree –p** command to analyse the process tree. How parent and child processes are displayed during the 10-second sleep?

d) Starting from the previous version, remove the wait call of the parent process (in this way, the parent exists immediately). What happens with the child process? Use the pstree –p command.

e) Modify the previous version in the following way: the child immediately finalizes by an exit() call (remove the sleep) and the parent executes wait() call after 10 seconds (use sleep() call). What happens with the child process? Use ps –aux command to analyse its state.

SOLUTION

a)

```
include <sys/types.h>
```

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

int main(){
        pid_t pidHijo;
        int status;

        pidHijo=fork();
        switch (pidHijo){
                case -1: printf ("Error fork()\n");
                         break;
                case 0:  /* Child process */
                         printf("I am the child - pid=%d\n", getpid());
                         exit(0);
                default: /* Parent process */
                         printf("Parent process - pid=%d\n", getpid());
                         wait(&status);
                         if (status==0){
                                 printf("Child exits OK\n");
                         }else{
                                 printf("Child exits KO\n");
                         }
        }
        return 0;
}
```

b)

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

int main(){
        pid_t pidHijo;
        int status;

        pidHijo=fork();
        switch (pidHijo){
                case -1: printf ("Error fork()\n");
                         break;
                case 0:  /* Child process */
                         printf("Child process - pid=%d\n", getpid());
                         execlp("/usr/bin/pstree", "pstree", NULL);
                         exit(0);
```

```
		default: /* Parent process */
			printf("Parent process - pid=%d\n", getpid());
			wait(&status);
			if (status==0){
				printf("Child exits OK\n");
			}else{
				printf("Child exits KO\n");
			}
	}
	return 0;
}
```

c)

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

int main(){
	pid_t pidHijo;
	int status;

	pidHijo=fork();
	switch (pidHijo){
		case -1: printf ("Error fork()\n");
			 break;
		case 0:  /* Child process */
			printf("Child process - pid=%d\n", getpid());
			sleep(10);
			execlp("/usr/bin/pstree", "pstree", NULL);
			exit(0);
		default: /* Parent process */
			printf("Parent process - pid=%d\n", getpid());
			wait(&status);
			if (status==0){
				printf("Child exits OK\n");
			}else{
				printf("Child exits KO\n");
			}
	}
	return 0;
}
```

d)

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

int main(){
        pid_t pidHijo;
        int status;

        pidHijo=fork();
        switch (pidHijo){
                case -1: printf ("Error fork()\n");
                         break;
                case 0:  /* proceso hijo */
                         printf("Proceso hijo - pid=%d\n", getpid());
                         sleep(10);
                         break;
                default: /* proceso padre */
                         printf("Proceso padre - pid=%d\n", getpid());
                         //wait(&status);
                         exit(0);
                         if (status==0){
                                 printf("Hijo termino OK\n");
                         }else{
                                 printf("Hijo termino KO\n");
                         }
        }
        return 0;
}
```

e)

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

int main(){
        pid_t pidHijo;
        int status;

        pidHijo=fork();
        switch (pidHijo){
                case -1: printf ("Error fork()\n");
                         break;
                case 0:  /* proceso hijo */
                         execlp("/bin/ls", "ls", "-la", "-s", NULL);
                         exit(-1);
```

```
                default: /* proceso padre */
                        printf("Proceso padre - pid=%d\n", getpid());
                        sleep(10);
                        wait(&status);
        }
        return 0;
}
```

Exercise 19.

Implement a program with the following functionalities:

- Create a program called **child** that received as input argument an integer number. This program shows 4 times this value on the screen.
- Create another program called **parent** that executes the child program passing as parameters the sum of both the parent and child IDs. In this way, the child shows four times this value. Hint: use the getpid() and execlp() calls.

SOLUTION
a)

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


#define         CONT    4
int main(int argc, char** argv){
        int num;
        int cont=0;

        if (argc!=2)
        {
                printf("Error: uso <%s> <NUM>\%d", argv[0]);
                exit(-1);
        }

        num=atoi(argv[1]);
        while(cont<CONT){
                printf("%d\n",num);
                cont++;
        }
}
```

b)

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

int main(){
        pid_t pidHijo;
        int miPid;
        char cadena1[8];
        int status;

        pidHijo=fork();
        switch (pidHijo){
                case -1: printf ("Error fork()\n");
                         break;
                case 0:  /* child */
                         miPid=getpid();
                         printf("Child - pid=%d\n", getpid());
                         sprintf(cadena1, "%d", miPid);
                         execlp("./hijo", "hijo", cadena1, NULL);
                         exit(-1);
                default: /* parent */
                         printf("Parent - pid=%d\n", getpid());
                         wait(&status);
        }
        return 0;
}
```

Exercise 20

Write a program that creates 10 processes. Each child process must execute an infinite loop, sleeping 60 seconds inside the loop each round. The parent has to print the children pids. Then it must wait for all of them and then exit.

SOLUTION

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


#define NUM_PROCESSES       10              /* 10 processes */
```

```
int main(){
        pid_t PROCESSES[NUM_PROCESSES];
        int i=0;
        char car;

        for(i=0;i<NUM_PROCESSES;i++){
                PROCESSES[i]=fork();
                if (PROCESSES[i]==0){   /* child process */
                        while(1);       /* busy waiting */
                }
                else
                        printf("Process %d-ID %d\n",i,PROCESSES[i]);
        }

        printf("Press any key to exit...\n");
    scanf("%c",&car);
        for (i=0;i<NUM_PROCESSES;i++)
                kill(PROCESSES[i], SIGINT);

        return (0);
}
```

Exercise 21.

Write a program that create a vector of pointers to buffers with dimension N (defined in the program). Then, it must allocate a 1 Kbyte buffer to every vector element, fill them with 'a', print the size allocated or an error, and to finish free all memory allocated to buffers.

SOLUTION

```
#define MAX_SIZE        1024            /* 1 KB */
#define N               10              /* Number of times */

int main(){
        char *buffer[N];
        int i=0;
        char car;

        for(i=0;i<N;i++){
                buffer[i]=(char*) malloc(MAX_SIZE);
                memset(buffer[i],'a',MAX_SIZE);
                if (buffer[i] == NULL){
                        printf("Error with malloc...\n");
                        exit(-1);
                }else{
```

```
                printf("%d - Block of %d bytes allocated\n", i, MAX_SIZE);
            }
            sleep(10);
        }

        printf("Press any key to exit...\n");
        scanf("%c",&car);

        for(i=0;i<N;i++){
            free(buffer[i]);
        }
        exit(0);
}
```

Exercise 22.

Given the program:

```
int main (int argc, char **argv) {

    int contador, i, pid;



    contador = atoi(argv[1]);

    i = 0;

    while (i < contador) {

        pid = fork();

        i++;

    }
```

a) Explain what it does and draw a diagram with the process hierarchy that originates when the command loop_processes 4 is executed.

b) Explain what a zombie process is and when it occurs.

c) Are zombie processes spawned when running the above command? Explain your answer.

d) What would happen if loop_processes 100 were executed? Explain the answer and modify the program so that the existing problem is solved.

e) Modify the previous program so that all the generated processes can share the file named pepe.txt. Explain your solution.

Exercise 23.

Implement a program that reads a number from the keyboard and displays a sequence of consecutive numbers starting from this number. At the same time, it has to be waiting for further keyboard inputs. As soon as a new value is provided, a new sequence of number will be displayed starting from the latest input value. With the 0 value, the program exists.

Both processes have to run concurrently, so the program design has to include the following features:

- The parent process has to be reading input values from the keyboard. For each new value it creates a child process that displays the sequence on the screen.
- When a new number is provided, the parent process kills the current child process and executes a new one that displays the new input value.
- Then the user enters 0, the parent process kills its child and terminates with exit() command.

Hint: use the kill system call.

SOLUTION

To be solved by students