Exercise 1 (20 points). Autotest.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
C	D	B	A	C	B	C	B	B	C	D	C	D	D	D

Exercise 2 (30 points)

Write a C program that creates the following processes:

$$\begin{tikzcd} & \text{Parent} \arrow[dl] \arrow[dr] & \\ \text{Child1} \arrow[rr, dashed, "\text{pipe}"'] & & \text{Child2} \end{tikzcd}$$

Child 1 will read data form keyboard and will send it to Child2 through a pipe. Child2 displays the received message on the screen.

The process execution will conclude when the Child1 receives the string "exit".

Solution

```
#include <unistd.h>

#include <stdlib.h>

#include <stdio.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <errno.h>

#include <string.h>


#define NUM_HIJOS 2 /* número de hijos a crear. */

void hijo1(int fds[2])

{

      int numbytes;

      char buf[4096];

      close(fds[0]);

      numbytes = read(STDIN_FILENO, buf, sizeof(buf));

      while (numbytes > 0) {
            if (write(fds[1], buf, strlen(buf)) == -1) {
                perror("fallo en write");
                exit(EXIT_FAILURE);
            }
            if (strncmp("fin\n", buf, strlen("fin\n")) == 0)
                break;
            numbytes = read(STDIN_FILENO, buf, sizeof(buf));
        }
        if (numbytes == -1) {
            perror("fallo en read");
            exit(EXIT_FAILURE);
        }
        close(fds[1]);
}
void hijo2(int fds[2])
{
        int numbytes;
        char buf[4096];
        close(fds[1]);
        numbytes = read(fds[0], buf, sizeof(buf));
        while (numbytes > 0) {
            if (strncmp("fin\n", buf, strlen("fin\n")) == 0)
                break;
            if (write(STDOUT_FILENO, buf, strlen(buf)) == -1) {
                perror("fallo en write");
                exit(EXIT_FAILURE);
            }
            numbytes = read(fds[0], buf, sizeof(buf));
        }
        if (numbytes == -1) {
            perror("fallo en read");
            exit(EXIT_FAILURE);
```

```
    }
    close(fds[0]);
}


int main(void)
{
    int ret, i, fds[2];
    if (pipe(fds) == -1) {
        perror("fallo en pipe");
        exit(EXIT_FAILURE);
    }
    for (i=0; i<NUM_HIJOS; i++) {
        ret = fork();
        if (ret == 0) {
            switch(i) {
                case 0:
                    /* tratamiento hijo 1. */
                    hijo1(fds);
                    exit(EXIT_SUCCESS);
                case 1:
                    /* tratamiento hijo 2. */
                    hijo2(fds);
                    exit(EXIT_SUCCESS);
            }
        } else if (ret > 0) {
            /* tratamiento del padre */
        } else if (ret == -1) {
            perror("fallo en fork");
            exit(EXIT_FAILURE);
        }
    }
    // El padre cierra la tubería antes de esperar y salir
```

```
        close(fds[0]);

        close(fds[1]);

        ret = wait(NULL);

        while (ret > 0) {

                ret = wait(NULL);

        }

/* si hay error, ignoramos si no hay más hijos a esperar. */

        if (ret == -1 && errno != ECHILD) {

                perror("fallo en wait");

                exit(EXIT_FAILURE);

        }

}
```

Exercise 3. (30 points)

Given a Producer-Consumer system where the producer generatetes information and increments a message counter. The consumer collects this information and then broadcast it and decrements the message counter. The processes do not interact with each other, except for control purposes. There are no mechanisms to store the produced information, thus it should be ensured strict alternation in the system. That is, when a producer generatetes information, it must wait until it has been collected by the Consumer before producing new information.

The processes are coded according to the following pseudocode:

procedure Producer is
while (true) do
    generatete_and_store;
    increase;
end while;
end procedure;
	procedure Consumer is
	while (true) do
	    get_and_send;
	    decrease;
	end while;
	end procedure;

Increase and decrease operations are atomic and operate on the same variable.

Answer the following questions:

1. Modify the provided code to ensure strict alternation between producer and consumer processes. You can use semaphores and the generic wait and signal instructions.

2. In a second step, you have to modify the solutions to allow the producer to store messages and wait with them until the Consumer collects them. If there are producer and consumer processes waiting for being executed, the system must keep the strict alternation between them. Otherwise, the producer can pass its turn to another Producer process. Write the code implementation using shared integer variables and semaphores.

Solution
SOLUCION:

1.

```
semaphore put :=1;
semaphore get:=0;
signal(put);
```

```
procedure Producer is
   while (true) do
      generate_&_store;
      put.wait;
      increment;
     get.signal;
   end while;
end procedure;
```
	```
	procedure Consumer is
	   while (true) do
	      get_&_send;
	      get.wait;
	      decrement;
	      put.signal;
	   end while;
	end procedure;
	```

2.

```
semaphore: put :=0;
semaphore: get:=0;
semaphore: mutex:=1;
integer: producers:=0;
integer: consumers:=0;
```

```
procedure Producer is
  while (true) do
    generate_&_store;
    mutex.wait;
    producers:=producers + 1;
    if (producers=1 and  consumers=0) then
      put.signal;
    end if
   mutex.signal;
   put.wait;
   increment;
   mutex.wait;
   if (consumers) then
     get.signal;
   else
     if (producers) then
       put.signal;
     else
       get.signal;
     end if;
   end if;
   producers:=producers - 1;
   mutex.signal;
  end while;
end procedure;
```

```
procedure Consumer is
  while (true) do
    get_&_send;
    mutex.wait;
    consumers:=consumers+1;
    mutex.signal;
    get.wait;
    decrement;
    mutex.wait;
    consumers:=consumers-1;
    mutex.signal;
  end while;
end procedure;
```

Exercise 4. (20 points)

Given the following small-size linked file system based on the File-Allocation Table (FAT):

- Block size: 1KB.
- The first 10 blocks (0 through 9) are reserved for filesystem, of which 3 are used to store filesystem information (SuperBlock) and for the filesystem boot (Boot). The remaining 7 blocks are used to store information of directories.

- The information of each directory occupies a block. Each entry has the following fields: filename extension with 64 bytes; metadata (user name, permissions, etc.) with 63 bytes; and starting block with 1 byte. A directory does not contain directories.

- Initially, the file system has two directories named (/documents and / bin) whose content is shown below:

Directory: /documents

Config.cf	Doc.txt							
Metadata	Metadata							
18	11							

Directory: /bin

ls	ps	cd	gcc					
Metadata	Metadata	Metadata	Metadata					
38	25	17	13					

- The FAT table only stores information related to the file data. The value -1 is used to encode the end of the file and 0 to indicate that the block is free. Below, the contents of the **entire** file system FAT table are shown. The upper index is the block number. The free block allocation policy for a new file is to first allocate the free blocks with the lowest identifier.

FAT TABLE

10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
-1	15	0	-1	0	29	0	20	-1	0	33	0	0	37	0

25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
23	0	0	0	10	0	-1	0	-1	0	31	0	35	-1	0

Considering this file system, answer the following questions. Justify your answer.

1. What is the total filesystem size? What is the filesystem effective capacity (the space in bytes that can be used to store file data)?

2. What is the maximum number of directories that the filesystem supports? How many files can be in each directory?

3. What are the files stored in the /bin directory? Briefly describes the functionality of each one of them.

4. Explain what blocks are uses to store information for the file /document/Doc.txt. For the same file: In which block is the file data byte 2050 stored?

5. Show the changes in the file system that would involve creating a new file, named "test.txt", with a size of 3 blocks in the /documents directory.

6. An important factor to improve the filesystem performance is to minimize the number of accesses to the filesystem data structures. This allows to reduce time while locating a particular data block. What are the drawbacks of this type of filesystem when accessing large files randomly (that is, not sequentially)? Do index-based files systems (such as UNIX-based i-nodes) have the same drawbacks?

SOLUTION

1. The FAT table contains the information of the file data blocks. In overall, there are 30 blocks thus the effective capacity is 30KB. There are 10 more blocks used by the filesystem, which represents a total size of 40KB.
2. Given that each directory occupies one block and 3 out of 10 blocks are reserved for the file system, the maximum number of directories will be 7. Each directory has a size of 1KB and each entry occupies (64+63+1=128B). Consequently the maximum number of files (entries in the block) will be 1KB/128B=8 files. .
3. ls shows the directory contents; ps displays the existing running processes; cd allows to change the directory and gcc is the C compiler.
4. According the FAT table, the starting block is the number 11; this one is linked with 15, which is linked with 29 which is linked with 10 (the last block). So, the file occupies the following sequence of 4 blocks 11 -> 15 -> 29 -> 10. The byte 2050 belongs to the third block which is no. 29.

5.

Directory: /documents

Config.cf	Doc.txt	**text.txt**							
Metadata	Metadata	**Metadata**							
18	11	**12**							

TABLA FAT

10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
-1	15	**14**	-1	**16**	29	**-1**	20	-1	0	33	0	0	37	0

25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
23	0	0	0	10	0	-1	0	-1	0	31	0	35	-1	0

6. One of the main problemas with linked files is that for a given Access it is necessary to traverse the block sequence from the starting block. This produces a larger access time when we Access to non-consecutive blocks. The index-based filesystems are more efficient (and fast) given that the linked list doesn't exists.