Fundamentos de los Sistemas Operativos

Departamento de Informática de Sistemas y Computadoras (DISCA)

Universitat Politècnica de València

fSO

Part 3. File system management Exercises

1 Questions about file implementation

1. A file requires a total of N data blocks. Assuming that its metadata are allocated in main memory calculate the number of disk accesses needed to access its last data block (N block), in a system with the following disk allocation policies:

a) Contiguous allocation
b) Linked list allocation
c) Indexed allocation

Explain your answer.

Solution

- A single access, since the position of the N-th block of the file in the disk can easily computed (just adding N-1 to the number of the first block of the file) and then it can be accessed directly.
- We need (at least) N accesses. The initial N-1 to scroll through the linked list until the requested position and another one to access the N-th block.
- As the index block is supposed to be already loaded in main memory and it contains all the block numbers of the file, only a single access on the requested block is required.

2. Suppose a file that contains a movie. Which block allocation system will be more efficient to view it? Explain your answer

Solution

Among the studied systems the best suited one is contiguous allocation, compared to linked or indexed. In this case given viewing the movie implies a sequential access to the file from the first byte to the last (it is assumed that you will not jump forward or backward), the fact that the bytes are in contiguous data blocks would minimize the disk for arm movements for reading, delivering the máximum posible performance.

3. What are the main advantages and disadvantages of contiguous allocation in a file system. It is not required to explain them, just enunciate them:

Solution

<u>Advantages</u>:

- It improves the performance of sequential access, allowing readings in advance and minimizing the movement of the disk heads (if there is no access to other files in the meantime)
- It eases the calculation of positions in direct access.
- It eases the maintenance of the list of blocks allocated to the file (it is enough to save the location of the first block and the number of blocks)

<u>Disadvantages:</u>

- It may avoid or limit the growth of files. This can be solved through compaction/defragmentation strategies (involving relocation of blocks), but this technique takes a long time to be applied on a large disk. It could also be resolved partially with overallocation of space but it would generate a high internal fragmentation
- It can produce external fragmentation

2 Questions about pipes and redirection

4. Enunciate the sequence of steps required to establish the communication between two UNIX processes, parent and child, in a way that: everything that the parent writes on its standard output it is read by the child from its standard input. Put in each step the system calls and C instructions required.

Solution:

The sequence of steps is:

1. Creating a pipe with:

```
int fd[2], ret;
pipe(fd);
```

2. Creating the child process:

```
ret = fork();
```

3. Assign the standard input of the child process to the pipe reading descriptor and close both pipe file descriptors in the child process:

```
if (ret == 0){
   dup2(fd[0], STDIN_FILENO);
   close(fd[0]);
   close(fd[1]);
}
```

4. Assign the standard output of the parent process to the pipe writing descriptor and close both pipe file descriptors in the parent process:

```
if (ret > 0){
   dup2(fd[1], STDOUT_FILENO);
   close(fd[0]);
   close(fd[1]);
}
```

5. Given the following code, indicated in the file descriptor table of the parent and child processes when execution reaches the line marked as B) on the parent and the line designated as A) on the child.

```
int fd[2], fd1, fde;
fde = open("error.txt", NEWFILE, MODE644);
dup2(fde, STDERR_FILENO);
close (fde);
pipe(fd);

if (fork() == 0) {
  /* CHILD CODE */
  dup2 (fd[0], STDIN_FILENO);
  close (fd[0]);
  close (fd[1]);
  fd1 = open("output.txt", NEWFILE, MODE644));
  dup2(fd1, STDOUT_FILENO);
  close (fd1);
  // ---> A) CHILD FILE DESCRIPTOR TABLE STATE
  ...
                                    (continues in the next page)
```

```
} else {
  /* PARENT CODE */
  dup2 (fd[1], STDOUT_FILENO);
  close (fd[0]);
  close (fd[1]);
  // ---> B) PARENT FILE DESCRIPTOR TABLE STATE
  ...
}
```

Assume that the initial state of the file descriptor table in the parent process is the following:

0	STDIN
1	STDOUT
2	STDERR
3	--
4	--

Solution:

	PARENT		CHILD
0	STDIN	0	Pipe_reading_descriptor
1	Pipe_writing_descriptor	1	output.txt
2	STDERR	2	STDERR
3	--	3	--
4	--	4	--

6. Given the following code that generates at least three process P1, P2, and P3:

```
// Code to be analyzed
...
pipe(fd);
pipe(fd2);
if(fork() != 0){
  /***Proceso P1 ***/
  dup2(fd[1],STDOUT_FILENO);
  close(fd[0]);
  close(fd[1]);
  dup2(fd2[0],STDIN_FILENO);
  close(fd2[0]);
  close(fd2[1]);

}else{
  /***Proceso P2 ***/
  dup2(fd[0],STDIN_FILENO);
  close(fd[0]);
  close(fd[1]);
```

```
  pipe(fd3);
  if(fork() != 0){
    close(fd2[0]);
    close(fd2[1]);
    dup2(fd3[1],STDOUT_FILENO);
    close(fd3[0]);
    close(fd3[1]);
  }else{
     /***Proceso P3 ***/
     dup2(fd3[0],STDIN_FILENO);
     close(fd3[0]);
     close(fd3[1]);
     dup2(fd2[1],STDOUT_FILENO);
     close(fd2[0]);
     close(fd2[1]);
  }
}
...
```

Describe the content of the file descriptor tables of processes P1, P2 y P3 after the execution of the former code. Explain what is the parentship relation between them and the communication scheme implemented.

Solution

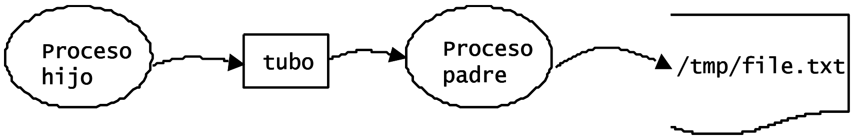
The execution of the proposed code creates three pipes, that we call T1 (the pipe with descriptor vector fd), T2 (the pipe with descriptors vector fd2) and T3 (the pipe with descriptors vector fd3). It also clear that the code will create a total of 3 processes.

The file descriptor tables will be:

```
        P1            P2            P3
0:      T2(reading)   T1(reading)   T3(reading)
1:      T1(writing)   T3(writing)   T2(writing)
2:      inherited     inherited     inherited
```

Where «inherited» indicates that such descriptor has not been "redirected" and that it continues to be associated to the file that was in the parent process of P1.
The relationship between the three processes is: P1 is the parent of P2 and P2 is the parent of P3.
The resulting communication scheme is a kind of "ring" between P1, P2 and P3, as it can be deduced from the previous file descriptor tables. Thus, T1 bridges the standard output of P1 to the standard input of P2, T3 passes the standard output of P2 to the standard input of P3 and T2 passes the standard output of P3 to the standard input of P1.

7. It is intended to implement a communication scheme between two processes: parent process and child process, like the one shown in the following figure:



The “tubo” device is a pipe and the parent process redirects is output to the file /tmp/file.txt.
The following code is proposed:

```
   ...
   pipe(fd);
   if(fork() == 0){
     dup2(fd[1],STDOUT_FILENO);
     close(fd[0]); close(fd[1]);
     /* Remaining child code */
   } else {
      dup2(fd[0],STDIN_FILENO);
      close(fd[0]); close(fd[1]);
      fd_open=open("/tmp/file.txt",O_WRONLY|O_TRUNC|O_CREAT,0600);
      /* Remaining parent code */
   }
   ...
```

Assuming that the rest of the missing code does not alter processes I/O redirection, indicate if the code is correct or not. In case of not being correct write the changes that are required to correct it.

Solution
The code inside else (parent code) should be the following:

```
} else {
   dup2(fd[0],STDIN_FILENO);
   fd_open=open("/tmp/file.txt",O_WRONLY|O_TRUNC|O_CREAT,0600);
   dup2(fd_open,STDOUT_FILENO);
   close(fd[0]); close(fd[1]);
   close(fd_open);
   /* Reamining parent code */
 }
```

3 Questions about directories and protection

8. Indicate if the following statements about UNIX systems are true or false. Explain your answer:

- **a)** For every directory it is true that the number of links is always greater than 1
- **b)** Directory entries occupy a dedicated area at the beginning of the disk.

Solution

- **a)** TRUE. Every directory has at least two links: the one in the parent directory and the reference to itself. The only directory that doesn't have a parent directory is the root directory, but in this case the second entry (reference to the parentr) points to itself too, so it also complies with the statement.
- **b)** FALSE. All directories entries appear in the data blocks belonging to files of type directory. Like the other data areas, they occupy the blocks that are selected when the directory grows. There is NO dedicated area, and the information is spread among different blocks in the data area.

9. Given the following list of a directory in a POSIX system:

```
drwxr-xr-x     2    sterrasa   fso   4096        may   8     2002       .
drwxr-xr-x     11   sterrasa   fso   4096        mar   21    14:39      ..
-rwsrw-r-x     1    root       fso   1139706     abr   9     2002       new
-rw-rw-r--     1    sterrasa   fso   634310      abr   9     2002       file1
-rw-rw-r--     1    sterrasa   fso   104157      abr   9     2002       file3
```

Where the "new" program copies the file passed as the first argument into the file passed as the second argument. Explain if the following commands may be completed successfully or not assuming that they are executed in the directory listed above:

- **a)** "new file1 file2", with initial attributes eUID=jose y eGID=fso in process "new"
- **b)** "new file1 file3", with initial attributes eUID=juan y eGID=grupo3 in process "new"

Solution:

- **a)** As "jose" is not the superuser, neither matches the owner of file "new", but "fso" is the owner group of the file, we will have to use the group permission group. In it we see that for this process it will be impossible to run the "new" program, so the command will not be successful.
- **b)** In this second case, neither the eUID is the owner of the file nor the eGID is the owner group of the file. Therefore, the triplet that applies will be the third (corresponding to "others"). In it, we see that the execution is possible. As this program has the SETUID bit active, running it we will adopt as eUID value of the "root". By adopting this identity, all accesses that require the process will be allowed, regardless of the permissions set on the files accessed, so this second command will succeed.

10. Explain if a process can change its effective UID and GID during its execution. If it is impossible, indicate why. If possible, describe how it could do it.

Solution:

Yes, it can be done. There are two ways. The first one can only be used by processes with effective UID equal to 0 (root). To do the eUID ore GID change, simply use setegid() and seteuid() calls. They would fail if the process is not running previously with identity (eUID = 0).

The second option is to start a program that has the SETUID bit or the SETGID bit active in its permission word. To do the eUID ore GID change, the process must have execution rights on the corresponding program and its effective UID or GID would become the owner or group of the file, depending on the bit that was active.

11. Given the following listing of a directory content on a UNIX system:

```
-rwsr--r-x 1 calif group1 1014  May  17 11:10 miprog
-r---w---- 1 calif group 14487 Jun   25 09:11 datos
-rw----r-- 1 calif group1 2099  Jun   25 08:49 punt2
```

And users "ramon" (belonging to the group "group1"), "marta" (belonging to the group “group3”), and "John" (belonging to the group "group2"). Explain, between the three mentioned users, who can use the program "miprog", and with it, process the information in files "data" and "punt2" (only reading reading the contents of both files is required).

Solution
Group “group1” can not execute “miprog”.
Groups 2 y 3 can execute “miprog” and also process “miprog” will be able to read files “datos” and “punt2”.

12. The following code, included in the executable file “miprog”, is designed to open a file which name is passed as an argument.

```
int main( int argc, char *argv[] ) {
   int fd;
   char path_command[80], *fichero;
   sprintf(path_command, "%s", argv[1]);
   fd = open( path_command, O_RDONLY );
   if (fd == -1) {
      fprintf(stdout,"error open to read %s", path_command);
   } else {
      fprintf(stdout,"successful open to read %s\n", path_command);
   }
   fd = open( path_command, O_WRONLY );
   if (fd == -1) {
      fprintf(stdout,"error open to write %s", path_command);
   } else {
      fprintf(stdout,"successful open to write %s\n", path_command);
   }
}
```

Considering the following file listing:

```
-rwsr-xr-x 1 xedu gedu 1014  May 17 11:10 miprog
-rw-r--r-- 1 xedu gedu 14487 Jun 25 09:11 datos
-r--rw-rw- 1 xedu gedu 2099  Jun 25 08:49 punt2
```

And given the following users: “xedu” that belongs to group "gedu" and "xedu2" that belongs to group "others". Specify the messages that will appear on the screen after users “xedu” and “xedu2” execute the following commands:

```
$ ./miprog datos
$ ./miprog punt2
```

Solution
User `xedu` executes “`./miprog datos`”:
sucessful open to read datos, sucessful open to write datos
User `xedu` executes “`./miprog punt2`”:
sucessful open to read punt2, error open to write punt2
User `xedu2` executes “`./miprog datos`”:
sucessful open to read datos, succesful open to write datos
User `xedu2` executes “`./miprog punt2`”:
sucessful open to read punt2, error open to write punt2

4 Questions about Minix

13. A minix file system has been created on device /dev/imagen. The device capacity is 15360 KBytes and it has been formatted to contain up to 17984 i-nodes. Determine the structure of the device, indicating the parts that make it up, as well as the number of blocks in each of the areas.

NOTE. Minix standard sizes are the following: 32 byte i-nodes (with 7 direct pointers, 1 indirect pointer and 1 double indirect pointer), 16 byte directory entries, 1 area = 1 block = 1024 bytes, 16 bit pointers to zone.

Solution

- **Nº of zones in the device** = 15360 Kbytes = 15360 blocks of 1Kbyte = 15360 zones
- **i-node bit map** = 1bit per i-node → 17984 bits → 17984/(1024*8) = 2,195 → 3 blocks
- **Zone bit map** = 1 bit per zone → 15360 bits → 15360/(1024*8) = 1,875 → 2 blocks
- **i-node area**: one i-node takes 32bytes → 32bytes*17984 = 575488bytes = 562 zones.

Nº of blocks in the header = 1 boot block + 1 super block + 3 blocks i-node bit map + 2 blocks zone bit map + 562 blocks i-node area = 569 blocks

Nº of blocks in the data area = 15360 – 569 = 14791 blocks (or zones)

1	1	3	2	562	14791
Boot block	Superblock	i-node bit map	Zone bit map	i-node area	Data area
0	1	2	5	7	569

14. A Minix file system contains the following directory tree, where b, d, e, h, i, j are regular files and the remaining are directories:

- /a/b
- /a/c/d /a/c/e /a/f/g/h
- /a/i /a/j /a/k

Obtain the content of all the directories and compute their sizes.
NOTE. Assume i-node allocation by alphabetical order

Solution:

Root directory : 48 bytes

1	.
1	..
2	a

Directory a : 128 bytes

2	.
1	..
3	b
4	c
7	f
10	i
11	j
12	k

Directory c: 64 bytes

4	.
2	..
5	d
6	e

Directory f: 48 bytes

7	.
2	..
8	g

Directory g: 48 bytes

8	.
7	..
9	h

Directory k: 32 bytes

12	.
2	..

15. In an standard MINIX file system (zones of 1 KB, i-nodes of 32 bytes with 7 direct pointers, 1 single indirect and 1 double indirect, 16 byte directory entries, pointer to zone of 16-bits) you intend to copy a 230 KB file with the command "cp fich1.txt fichero2.txt", answer to the following questions:

a) Determine the number of new zones that Minix will have marked as occupied as a result of the execution of the former copy command and the type of information that will contain these new zones.

b) List the sequence of system calls required to complete the copy command, providing a simple algorithm that describes what actually this command does.

Solution

a) If the file has 230 KB, you need 230 data areas. Indexed allocation is used in MINIX and pointers to zone are 2 bytes, so 512 zone pointers fit inside a zone of pointers. As we only have 7 direct pointers in the i-node, we need to use a zone of pointers (its location is stored in the simple indirect pointer inside the i-node) to store the allocation of zones from 8 to 230. For this reason we have a total of 231 zones (230 for data and 1 for pointers).

b) The copy command would perform the following calls:

- Create a destination file using open() o creat().
- Open the file en write only mode with open().
- Start a loop until reaching the end of the source file, doing the following operations:
  - Read a fragment of the source file with read().
  - Write the read fragment in the destination file with write().
- Close both files with close().

16. It has been created a Minix file system with 8100 blocks in total with 10016 i-nodes. Obtain the structure of the file system, indicating the parts that make it up, as well as the number of blocks in every area.
NOTE. Minix standard sizes are as follows: 32 byte i-nodes (7 direct pointers, 1 indirect and 1 double indirect), 16 byte directory entries, 1 area = 1 block = 1024 bytes, 16 bit pointers to zone

Solution:

1	1	2	1	313	7782
Boot block	Superblock	i-node bit map	Zone bit map	i-nodes area	Data area
0	1	2	4	5	318

17. In a 128Mbytes device a MINIX file system with standard sizes is created with 3500 i-nodes. Compute the free space that remains for data just after creating the file system.

Solution:

The header will require:

- 1 boot block
- 1 super block
- 1 block for the i-node bit map (because 3500 < 8192).
- 16 blocks for the zone bit map (128*1024/8192 = 128/8 = 16)
- 110 blocks for the i-node area (3500*32/1024 = 109,3 --> 110)

And this gives a total of 129 blocks for the header. Thus, the number of blocks for data will be equal to 128 * 1024 - 129 = 130943, although the first block will be occupied by the data in the root directory ("." And "..") directory entries) that is created when the MINIX file system is created.

18. In a Minix file system initially empty and mounted on /minix, the following commands are executed:

```
$ echo "hello FSO" > /minix/file1
$ cat /minix/file1 | grep h | wc -l > /minix/lines
$ mkdir /minix/maria
$ ls /proc
$ ps -la
$ mkdir /minix/bin
$ ln /minix/file1 /minix/maria/file2
$ ln -s /minix/file1 /minix/maria/file3
$ ln -s /minix/lines /minix/maria/file4
```

Considering Minix standard sizes (16 bit pointers to zone, 16 byte directory entries, 32 byte i-nodes, 1 block = 1 zone = 1Kbyte), obtain:

a) Total number of busy i-nodes and to what files they correspond
b) Size in bytes of directory /minix and the value of its attribute "number of links"

Solution

a) 7 i-nodes busy that correspond to:
/minix
/minix/file1 y /minix/maria/file2 (the same i-node for both)
/minix/lines
/minix/maria
/minix/bin
/minix/maria/file3
/minix/maria/file4

b) Size of root directory: 6 directory entries * 16 bytes = 96 bytes
Number of links = 4, that correspond to:
/minix/.
/minix/..
/minix/maria/..
/minix/bin/..

19. Explain in detail how many i-nodes would be busy, and what would be the value of the attribute 'number of links' of every one of them, in a UNIX file system that contains only the following files:

a) /f1 (2 Mbyte regular file)
b) /f2 (symbolic link to f1)
c) /f3 (hard link to /f1)
d) /dir1 (empty directory)

Solution

There are only four i-nodes busy that correspond to: the root directory, f1, f2 and dir1. (f3 is a physical link to f1 and therefore have the same i-node). The value of the attribute "number of links" for every i-node is:

a) Root directory: 3 links ("." and ".." inside the root directory itself and ".." inside the directory "/dir1")
b) f1: 2 links (f3 is the other link to f1)
c) f2: 1 link
d) dir1: 2 links ("dir1" in the root directory and "." inside the dir1 directory itself)