1. **Explain the dup() System call with example.**

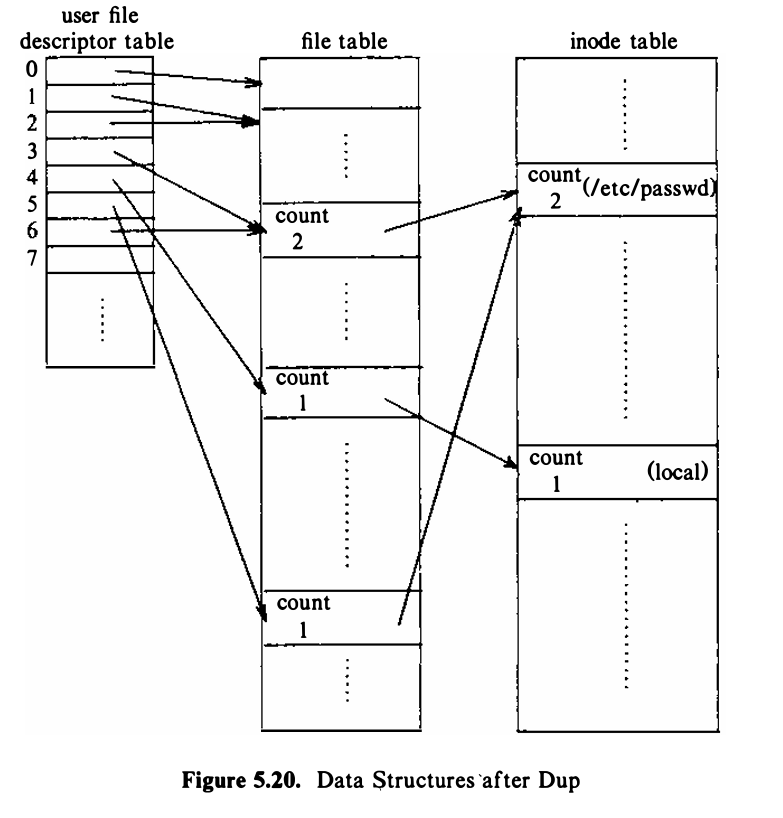
* The dup system call copies a file descriptor into the first free slot of the user file descriptor table, returning the new file descriptor to the user. It works for all file types. The syntax of the system call is

newfd - dup(fd);

where fd is the file descriptor being duped and newfd is the new file descriptor that references the file.

* Because dup duplicates the file descriptor, it increments the count of the corresponding file table entry, which now has one more file descriptor entry that points to it.
* For example, examination of the data structures described in following figure indicates that the process did the following sequence of system calls:

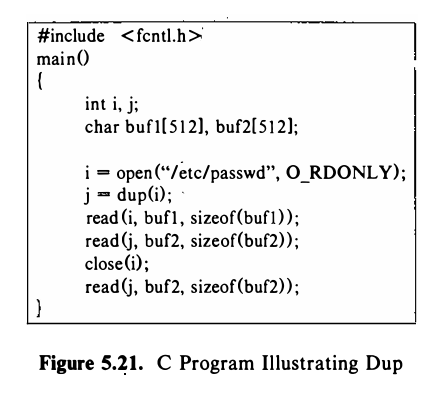
1. It opened the file "/etc/passwd" (file descriptor 3),
2. then opened the file "local" (file descriptor 4),
3. opened the file "/etc/passwd" again (file descriptor 5),
4. and finally, duped file descriptor 3, returning file descriptor 6.



* Dup is perhaps an inelegant system call, because it assumes that the user knows that the system will return the lowest-numbered free entry in the user file descriptor table.
* However, it serves an important purpose in building sophisticated programs from simpler, building-block programs, as exemplified in the construction of shell pipelines.

**Example:**

Consider the program below. The variable i contains the file descriptor that the system returns as a result of opening the file "etc/passwd," and the variable j contains the file descriptor that the system returns as a result of duping the file descriptor i.



In the u area of the process, the two user file descriptor entries represented by the user variables i and j point to one file table entry and therefore use the same file offset.

The first two reads in the process thus read the data in sequence, and the two buffers, buf1 and buf2, do not contain the same data.

A process can close either file descriptor if it wants, but I/O continues normally on the other file descriptor, as illustrated in the example. In particular, a process can, close its standard output file descriptor (file descriptor 1), dup another file descriptor so that it becomes file descriptor 1, then treat the file as its standard output.

1. **Explain difference between named pipe and unnamed pipe.**

|  |  |  |
| --- | --- | --- |
| Naming | Named pipes have a name in the filesystem and are created using the mkfifo() system call or the mkfifo command. | Unnamed pipes are created using the pipe () system call and do not have a name associated with them. |
| Persistence | Named pipes persist beyond the lifespan of the processes that created them. They exist as special files in the file system until explicitly removed. | Unnamed pipes exist only as long as the processes that created them are alive. They are typically used for communication between parent and child processes or between related processes. |
| Accessibility | Named pipes can be accessed by any process with appropriate permissions, even unrelated processes. | Unnamed pipes are typically accessible only by the processes that created them or by processes created subsequently via fork. |
| Creation | Named pipes can be created at any time by any process with appropriate permissions using mkfifo() or mkfifo command. | Unnamed pipes are created implicitly by the operating system when a process calls the pipe() system call. |
| Filesystem Representation | Named pipes are represented as special files in the filesystem, similar to regular files but with a different file type (p). | Unnamed pipes do not have a representation in the filesystem; they exist only within the kernel's memory. |
| Access Control | Named pipes support file permissions and access control mechanisms, allowing for secure communication between processes. | Unnamed pipes do not have explicit access control mechanisms. Access is limited to the processes involved in the pipe. |
| Inter-process Communication | Named pipes allow communication between unrelated processes that have no prior relationship or shared ancestry. | Unnamed pipes are typically used for communication between related processes, such as parent-child relationships. |
| Process Independence | Named pipes provide a way for independent processes to communicate, enabling loose coupling between them. | Unnamed pipes are typically used for tightly coupled communication between cooperating processes. |
| Error Handling | Named pipes can handle errors related to pipe creation, opening, and communication through standard file I/O error mechanisms. | Unnamed pipes can handle errors through standard system call error codes and return values. |
| Persistence of Data | Data written to a named pipe remains available until it is read or the pipe is explicitly removed. | Data written to an unnamed pipe is typically consumed immediately by the reading process, and once read, it is no longer available. |
| Usage | Named pipes are often used for communication between processes running at different times or by different users. | Unnamed pipes are commonly used for communication between processes created in a pipeline or for simple IPC between related processes. |
| Communication Direction | Named pipes can facilitate bidirectional communication between processes, allowing both reading and writing from each end. | Unnamed pipes are typically unidirectional, supporting data flow in one direction only (from the writer to the reader). |
| Cleanup | Named pipes persist until explicitly removed by a process or by an administrator. | Unnamed pipes are automatically closed and deallocated by the operating system when the processes that created them terminate. |
| File Descriptor Handling | Named pipes are managed using file descriptors similar to regular files, allowing for familiar I/O operations like open(), read(), write(), and close(). | Unnamed pipes are created implicitly by the pipe() system call, returning two file descriptors representing the read and write ends of the pipe, respectively. |

1. **Explain the read() System call.**

The syntax of the read system call is

**number = read (fd, buffer, count);**

where fd is the descriptor returned by open.

buffer is the address of the data structure where the data will be read.

count is the number of bytes to be read. And it returns how many bytes were successfully read.

**The algorithm is given below:**

/\* Algorithm: read

\* Input: user file descriptor

\* address of buffer in user process

\* number of bytes to be read

\* Output: count of bytes copied into user space

\*/

{

get file table entry from user file descriptor table;

check file accessibility;

set parameters in u-area for user address, byte count, I/O to user;

get inode from file table;

lock inode;

set byte offset in u-area from file table offset;

while (count not satisfied)

{

convert file offset to disk block (algorithm bmap);

calculate offset into block, number of bytes to read;

if (number of bytes to read is 0)

break; // trying to read End Of File (EOF)

read block (algorithm: bread or breada whichever applicable);

copy data from system buffer to user address;

update u-area fields for file byte offset, read count, address to write into user space;

release buffer; // locked in bread

}

unlock inode;

update file table offset for next read;

return (total number of bytes read);

}

**Figure: Algorithm for Reading a File**

After getting the file table entry from user file descriptor table, the kernel sets some parameters in the u-area and eliminates the need to pass them as function parameters. The parameters in the u-area:

* **mode:** indicates read or write
* **count:** count of bytes to read or write
* **offset:** byte offset in file
* **address**: target address to copy data in user or kernel memory
* **flag:** indicates if address is in user or kernel memory

If a process reads two blocks sequentially, the kernel assumes that all subsequent reads will be sequential until proven otherwise. During each iteration through the loop, the kernel saves the next logical block number in the in-core inode and during the next iteration, compares the current logical block number to the value previously saved. If they are equal, the kernel calculates the physical block number for read-ahead and saves its value in the u-area for use in the breada algorithm. Of course, if a process does not read to the end of the block, the kernel does not invoke read-ahead for the next block.

As we had seen previously, it is possible to have some block numbers in an inode or in an indirect block to have the value 0. In such cases, the kernel allocates an arbitrary buffer and clears its contents to 0 and copying it to user address space.

The kernel always unlocks inode at the end of a system call. Inode is not locked across the system calls. Otherwise, one malicious/erroneous user can block all the other users from accessing a file.

1. **Change Directory and Change Root**

When process 0 is created, it sets its current directory as root. It gets the root inode (iget), saves it in its u-area as the current directory and releases the inode. When a new process is created with fork, it inherits the current directory from the parent process in its u-area and the inode reference count is incremented.

The algorithm chdir (Figure 5.14) changes the current directory of a process.

The syntax for the chdir systein call is

chdir(pathname);

where pathname is the directory that becomes the new current directory of the process. The kernel parses the name of the target directory using algorithm namei and checks that the target file is a directory and that the process owner has access permission to the directory. It releases the lock to the new inode but keeps the

inode allocated and its reference count incremented, releases the inode of the old current directory (algorithm iput) stored in the u area, and stores the new inode ir the u area. After a process changes its current directory, algorithm namei uses the inode for the start directory to search for all path names that do not begin from root. After execution of the chdir s.ystem call, the inode reference count of the new directory is at least one, and the inode reference count of the previous current directory may be 0. In this respect, chdir is similar to the open system call because both system calls access a file and leave its inode allocated. The inode allocated during the chdir system call is released only when the process executes another chdir call or when it exits.

The algorithm is given below:

/\* Algorithm: chdir

\* Input: new directory name

\* Output: none

\*/

{

get inode for new directory name (algorithm: namei);

if (inode not that of directory or if access not permitted)

{

release inode (algorithm: iput);

return (error);

}

unlock inode;

release "old" current directory inode (algorithm: iput);

place new inode in current directory slot of u-area;

}

A process usually uses the global file system root for all path names starting with "/". The kernel contains a global variable that points to the inode of the global root, allocated by iget when the system is booted. Processes can change their notion of the file system root via the chroot system call. This is useful if a user

wants to simulate the usual file system hierarchy and run processes there. Its syntax is

chroot(pathname);

where pathname is the directory that the kernel subsequently treats as the process's root directory. When executing the chroot system call, the kernel follows the same algorithm as for changing the current directory. It stores the new root inode in the process u area, unlocking the inode on completion of the system call. However, since the default root for the kernel is stored in a global variable, it does not release the inode of the old root automatically, but only if it or an ancestor process had executed the chroot system call. The new inode is now the logical root of the file system for the process (and all its children), meaning that all path name searches in algorithm namei that start from root ("/") start from this inode, and that all attempts to use " .. " over the root will leave the working directory of the process in the new root. A process bestows new child processes with its changed root, just as it bestows them with its current directory.

**In a way, chdir is similar to open, as both of them leave the inode allocated. In the case of chdir, the inode will be release only when the process calls chdir again with different pathname, or when the process exits.**

**Processes usually use the global root "/" as for pathnames starting with "/". The kernel has a global variable that points to the inode of the global root. Processes can change their current root via chroot system call. This is useful for simulations.**

**chroot (pathname);**

**The algorithm is similar to chdir. But if the old root was the global root, it is not released. When the current root is changed, searches for pathnames starting from "/" will start from the new root. And all of the child process will inherit the current root.**

1. **Explain algorithm creat() for creating a new file.**

The creat system call creates a new file in the file system.

fd = creat (pathname, modes);

If pathname of an existing file is passed to creat, it will truncate the file and set its size to 0 (if permissions allow, otherwise the system call will fail).

The algorithm is given below:

/\* Algorithm: creat

\* Input: file name

\* permission settings

\* Output: file descriptor

\*/

{

get inode for file name (algorithm: namei);

if (file already exists)

{

if (not permitted access)

{

release inode (algorithm: iput);

return (error);

}

}

else

{

assign free inode from file system (algorithm: ialloc);

create new directory entry in the parent directory: include new file name and newly assigned inode number;

}

allocate file table entry for inode, initialize count;

if (file existed at time of creat)

free all file blocks (algorithm: free);

unlock (inode);

return (user file descriptor);

}

After the steps in the algorithm, the creat system calls follows the steps in open to create entries in the 3 tables.

The algorithm also remembers the inode of the directory being searched, in the u-area and keeps the inode locked as the directory will become the parent directory of the file.

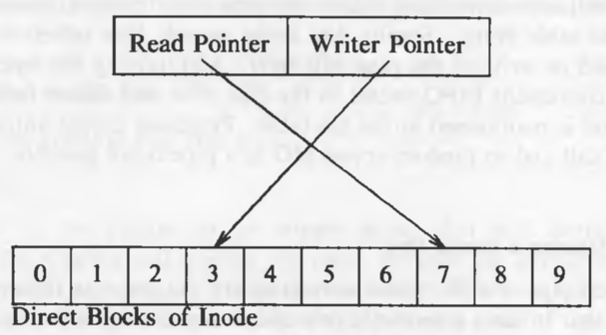
The kernel write the newly allocated inode to disk before it writes the directory with the new name to disk. If the system crashes between the write operations, an inode which is not referenced by any path will just lie there, but the system will functional normally. However, if a directory entry references a corrupt inode, the system will crash.

If the pathname refers to an old file, kernel gets the inode of that file, truncates its contents (if write permission is given, otherwise error), but it does not modify the permissions and owner of the file, it ignores the permission modes given in the system call parameter. The kernel does not check if the parent directory allows write permission, because it is not going to write in the parent directory.

**8. Reading and Writing Pipes**

A pipe should be viewed as if processes write on one end of the pipe and read from the other end. The number of processes reading from a pipe do not necessarily equal the number of processes writing the pipe; when the number is not equal, they must coordinate use of the pipe with other mechanisms.

The storage mechanism for a regular file and a pipe is similar. The kernel uses the data blocks from the pipe device. But there is a difference, the pipe uses only direct blocks of the inode for efficiency. But this places a limit on the size of the pipe at a time. The kernel manipulates the direct blocks of the inode as a circular queue. It maintains read and write pointers internally to preserve the FIFO order, as shown in the figure:



We will study 4 cases:

1. Writing a pipe that has room for the data being written: The sum of bytes being written and the number of bytes that are already there in the pipe is less than the capacity of the pipe. The kernel follows the same algorithm as of normal files, except that it increments the size of the pipe after every write. For normal files, the kernel increments the size only if the data is written beyond the maximum byte offset. If the next byte offset is going to require an indirect block, the kernel just brings the byte offset value in the u-area to 0, but it never overwrites the data in the pipe. It sets the offset to 0 because it has already determined that the the data will not overflow the capacity. When the writer process has written all its data into the pipe, the kernel updates the pipe's write pointer (stored in the inode) so that next writes begins from that location. The kernel then awakens all other processes waiting for to read data from the pipe.
2. Reading from a pipe which has enough data to satisfy the read: Before reading, kernel checks if the pipe is not empty. The reading begins from the read offset (stored in the inode). With every block of read, the kernel decrements the size of the pipe according to the number of bytes it read, and adjusts the u-area offset value to wrap around, if needed. On completion of the call, kernel awakens all sleeping writer processes and saves the current read offset in the inode.
3. Reading from a file that does not contain enough data to satisfy the read: If a process tries to read more data than is in the pipe, the read will complete successfully after returning all data currently in the pipe. If there is no data in the pipe, the reading process sleeps unless no delay option is given.
4. Writing to a pipe which does not have enough space: If a process tries to write data that does not fit into the pipe, the process goes to sleep waiting for space to become available. But the case when data to be written is greater than the capacity of the pipe, is different. In that case, the kernel writes writes as much data as possible and puts the process to sleep until more room becomes available. Because of the race to write the data, the data in a pipe is not guaranteed to be contiguous.

The offsets had to be stored in the inode so that processes could share the read and write offsets. For every open call, a process gets a new entry in the file table, that is why it is not possible to share the offset in the file table.

**9. Open**

Open is the first step to access data in a file. The syntax for the open system call is

fd = open (pathname, flags, mode);

where flags indicate the type of open (reading or writing) and mode gives the permissions if the file is being created. It returns an integer called the user file descriptor. Other file operations use the file descriptor returned by open.

The algorithm is given below:

/\* Algorithm: open

\* Input: file name

\* type of open

\* file permissions (for creation type of open)

\* Output: file descriptor

\*/

{

convert file name to inode (Algorithm: namei);

if (file does not exist or access is not permitted)

return (error);

allocate file table entry for inode, initialize count, offset;

allocate user file descriptor entry, set pointer to file table entry;

if (type of open specifies truncate file)

free all blocks (algorithm: free);

unlock (inode);

return (user file descriptor);

}

After getting the in-core inode of the file to be opened, the kernel allocates an entry in the file table and sets the offset of the file, the offset tells the kernel from where to read or write to the file. In the case of read and write modes, the offset is set to 0, but for write-append mode, the offset is set to the size of the file. Then the kernel allocates an entry in the user file descriptor table which is local to a process and accessible through the u-area. The user file descriptor table entry has a pointer to its file table entry and the file table entry has a pointer to the in-core inode table entry. The file descriptor returned to the user is nothing but the index of the entry in the user file descriptor table.

**11. Draw and explain the file system data structure for each statement ………**

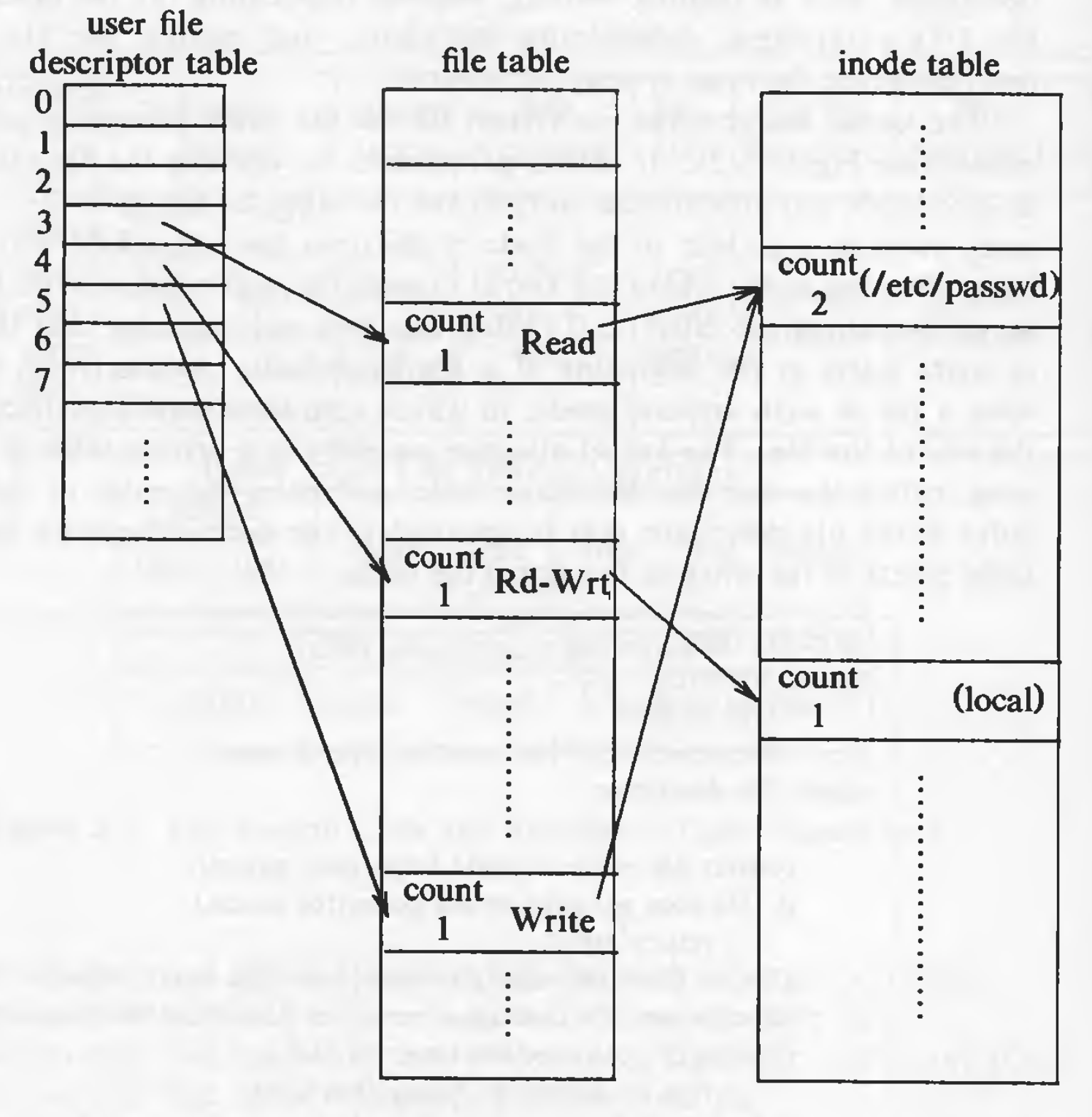
If a process executes following code:

fd1 = open ("/etc/passwd", O\_RDONLY);

fd2 = open ("local", O\_RDWR);

fd3 = open ("/etc/passwd", O\_WRONLY);

In this case, the state of the in-core inode table, file table, and user file descriptor table will be like this:



**Fig. Data structures after open**

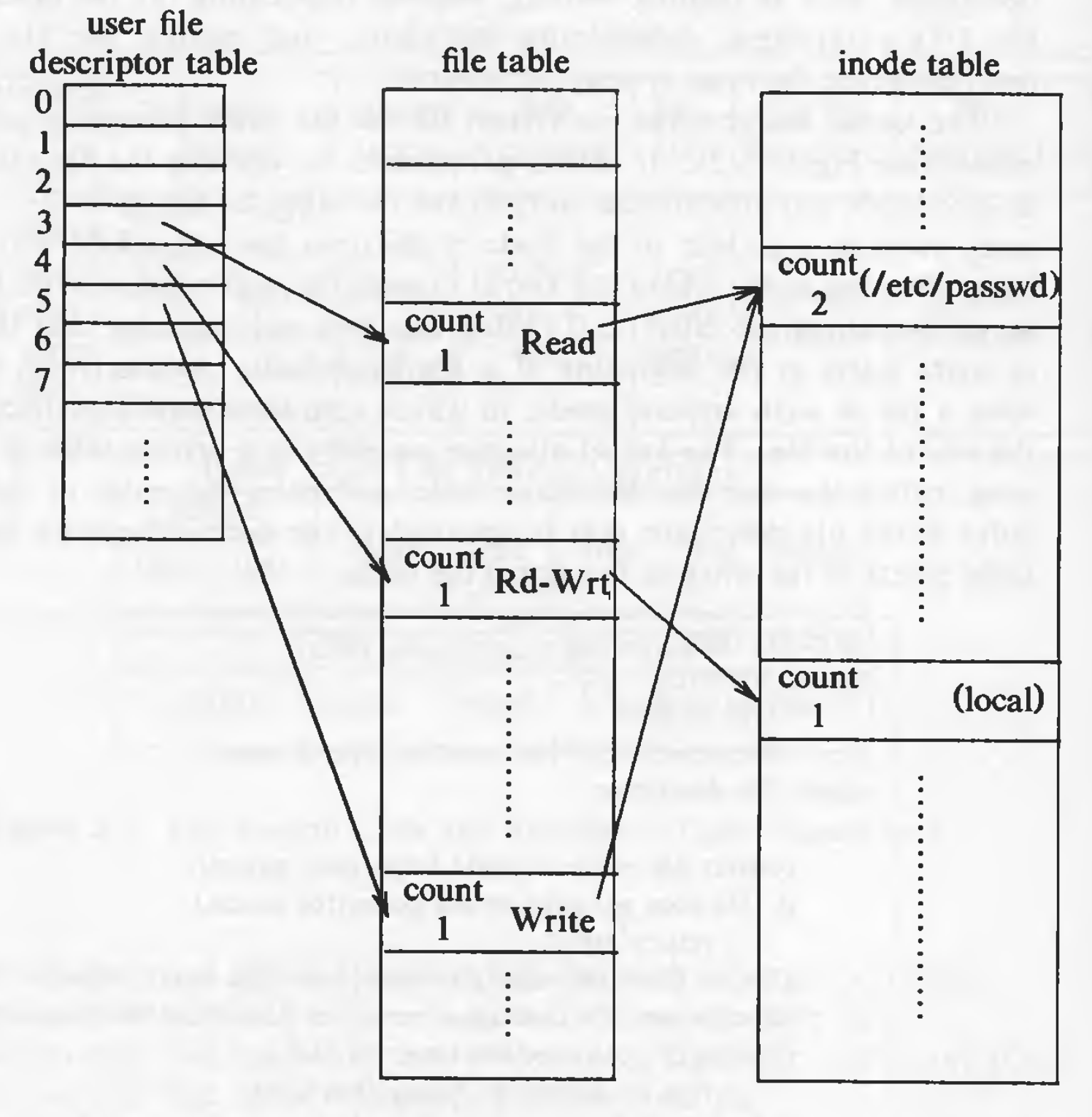
Entries in the user file descriptor table point to unique entries in the file table even though "/etc/passwd" is opened twice. This is needed because the modes of open could be different and even if they are not, the offsets need to be maintained separately. Both the file table entries point to the same in-core inode table entry.

If another process (say process B) executes the following code, in addition to process A executing the above code:

fd1 = open ("/etc/passwd", O\_RDONLY);

fd2 = open ("private", O\_RDONLY);

The state of the tables will be like this:



**Fig. Data structures after two processes open files**

The file table entries created by different process for the same file point to the same in-core inode table entry. The offsets could have been stored in the user file descriptor table entries as well, eliminating the need of the file table. But the additional indirection to the file table enables sharing of files (using dup and fork system calls) as we will see later.

The first 3 entries (0, 1, 2) in the user file descriptor table point to standard input (stdin), standard output (stdout), and standard error (stderr). But it is just a convention and there is nothing special about these entries. stdin is used to read input (usually, keyboard), stdout and stderr are used for printing the output and the errors respectively (usually, monitor for both).

If the super block Free Inode list is empty and remembered Inode is 470. Explain the steps to fill the superblock free Inode list.

Let us assume Disk block contains 1024 bytes and there are 10 direct blocks, 1 single indirect block, 1, double indirect block, 1 triple indirect block. Find the maximum size of the file of a file's table of content. Write your own assumptions if any.