Lecture 3. File I/O (continued)

I/O Efficiency

Both the `read()` and the `write()` functions take a buffer size as an argument (`size_t` *`nbytes`*), meaning the number of bytes to read and write, respectively.

Consider the following program, source code mycopy.c:

```
#include <unistd.h> /* for read() and write() */
#include <stdio.h> /* for printf() */

#define BUFFSIZE 8192

int main(int argc, char *argv[])
{
     int n;
     char buf[BUFFSIZE];

     char prompt[31]="Enter a string, or ^C to exit: ";
     write(STDOUT_FILENO, prompt, 31);

     while ( (n=read(STDIN_FILENO, buf, BUFFSIZE))>0 )
     {
          if (write(STDOUT_FILENO, buf, n)!=n)
               printf("Write error\n");
     }

     if (n<0)
          printf("Read error\n");
return 0;
}
```

This program writes a prompt to standard output, asking the user to type a string, it keeps reading strings as the user keeps typing, outputs each one at the terminal using the `write()` function, and exists on a `^C` control character.

Observations:

- the file descriptors for the files to read from, respectively write to, were not obtained through an `open()` system call. They were rather obtained as the standard input/output descriptors from the shell, so the user can use the redirect operators to capture input from/output into a file;

- the file descriptor values were obtained from the following entries of `<unistd.h>` (`/usr/include/unistd.h`):

```
#define   STDIN_FILENO   0 /* standard input file descriptor */
#define   STDOUT_FILENO  1 /* standard output file descriptor */
#define   STDERR_FILENO  2 /* standard error file descriptor */
```

- the program doesn't use `close()` to close either of the input or the output files.

The BUFFSIZE value (8192 bytes) was chosen arbitrarily. The following table gives the program times for different values of BUFFSIZE, when reading input from a 1,468,802-byte file.

BUFFSIZE	real	usr	sys
1	0m2.683s	0m0.311s	0m2.363s
2	0m1.333s	0m0.154s	0m1.171s
4	0m0.688s	0m0.077s	0m0.589s
8	0m0.373s	0m0.040s	0m0.302s
16	0m0.227	0m0.021s	0m0.172s

To avoid polluting the terminal with such big output, the standard output could be redirected to /dev/null. The large file was created with a `write()` system call appending strings in a loop, source code mylargefile.c:

```
#include <unistd.h> /* for read(), write() */
#include <fcntl.h> /* for open() */
#include <string.h> /* for strcat() */
#include <stdio.h>/* for printf() */

#define BUFFSIZE 4096
#define FILESIZE 1474560

int main(int argc, char *argv[])
{
     char file_chunk[BUFFSIZE+1];
     int  fd=open("/Users/awise/Stevens/Lecture3/large_file.txt",
O_CREAT|O_RDWR|O_APPEND, S_IRUSR|S_IWUSR);

     file_chunk[0]='\0';
     for (int i=0; i<BUFFSIZE; i++)
     {
          strcat(file_chunk, "a");
     }
     printf("%s\n", file_chunk);
     for (int j=0; j<FILESIZE/BUFFSIZE; j++)
          ssize_t num_bytes=write(fd, file_chunk, BUFFSIZE);

return 0;
}
```

In this program, the string `file_chunk` is created in size BUFFSIZE+1 (the +1 to accommodate the start null character `'\0'`). The large file is created using the C function `strcat()`, called on `file_chunk` in a loop FILESIZE/BUFFSIZE times. Note the options to `open()`, O_APPEND to allow additions to the file of `file_chunk` strings, and S_IRUSR|S_IWUSR to set the access modes equivalent to `chmod u+rw`, needed to read the file by the following program.

The size of the resulting file can be verified with the `ls -l` command:

```
$ ls -l large_file.txt
-rw-rw-rw-  1 awise  staff  1474560 Feb  4 16:47 large_file.txt
```

The following program reads the file in a while loop, BUFFSIZE bytes at a time. By giving BUFFSIZE different values, we can monitor the efficiency of the `read()` system call, noticing that, indeed, the larger the BUFFSIZE block, the smaller program execution times are obtained, source code mybuffers.c:

```
#include <unistd.h> /* for read(), write() */
#include <fcntl.h> /* for open() */
#include <stdio.h>

#define BUFFSIZE 16
#define FILESIZE 1474560

int main(int argc, char *argv[])
{
     int  fd=open("/Users/awise/Stevens/Lecture3/large_file.txt",
O_RDWR);
     off_t offset=lseek(fd, 0, SEEK_SET);
     char buff[BUFFSIZE];
     ssize_t bytes;
     bytes=read(fd, buff, BUFFSIZE);
     //printf("There were %zd bytes read\n", bytes);
     while ((bytes=read(fd, buff, BUFFSIZE))>0)
     {
          //printf("There were %zd bytes read\n", bytes);
          if ((write(STDOUT_FILENO, buff, bytes))!=bytes)
               write(STDOUT_FILENO, "Write error\n", 12);
     }
     if (bytes<0)
          write(STDOUT_FILENO, "Read error\n", 11);
return 0;
}
```

The table on page 2 summarizes the output of the `time` command for the execution of this program:

```
$ time ./mybuffers
```

The output of the time command has three fields:

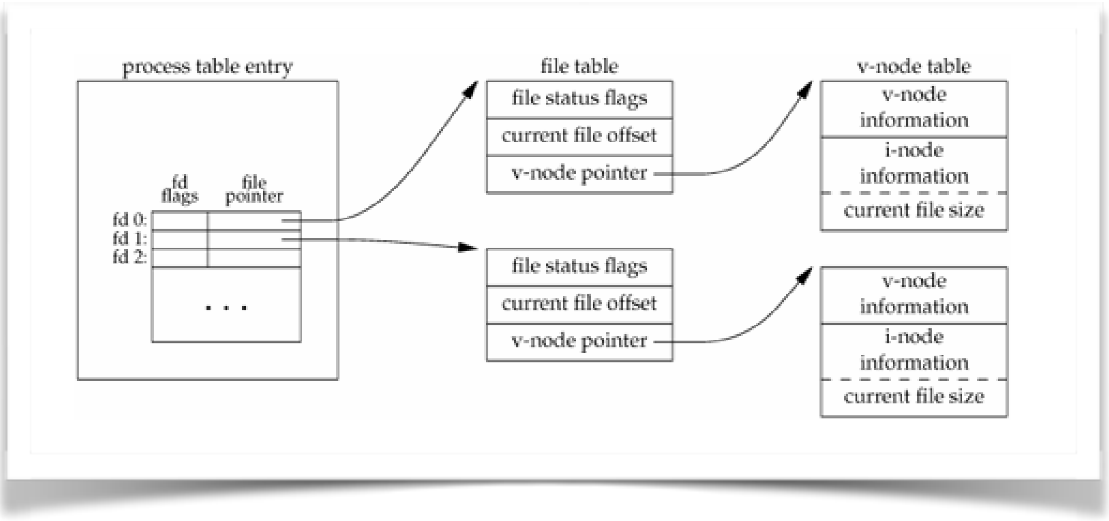
- `sys` — time spent by the program in system calls on behalf of your program;
- `usr` — time spent by the program in user program function calls/instructions (e.g. looping through an array, executing the user part of a function)
- `real` — larger than the sum of the two `real>user+sys`, due to wait times and I/O

File Sharing

UNIX supports file sharing across processes. To study the `dup()` system call, which duplicates a file for use by multiple processes, we will start by describing the structures used by the kernel for all I/O:

1. Processes are catalogued in a **process table**. Each entry in the process table (one per process) has a table of open file descriptors. Each file descriptor has:
   (a) file descriptor flags
   (b) a pointer to a file table entry
2. Files are catalogued in a **file table**. Each entry in the file table has:
   (a) file status flags (read, write, append etc.)
   (b) the current file offset
   (c) a pointer to the v-node table entry for the file
3. Files (devices) have a **v-node structure**. This contains information about the type of file and pointers to functions that operate on the file:
   - **i-node** information
   - **current file size**

The following figure shows a visual arrangement of the three tables (process, file, v-node) for a process using 2 files:



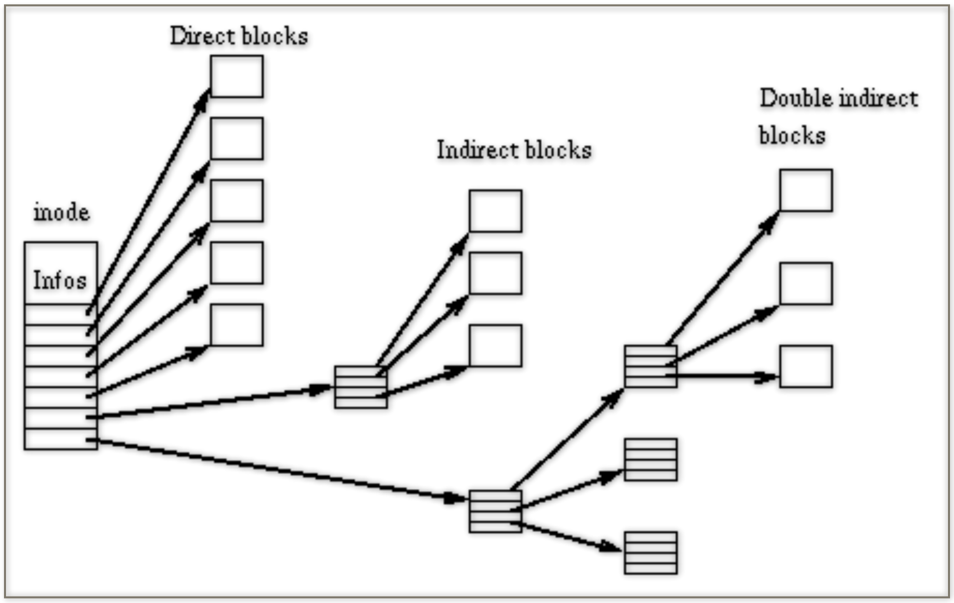
The i-node structure contains the following entries:

- the **file size** in bytes
- **device ID** (the device containing the file)
- the **user ID** of the file's owner
- the **group ID** of the file
- the **file mode** (file type and the access modes for user, group, others
- **system** and **user flags** to further protect the file (limit its use and modification)
- **timestamps** telling when the i-node itself was last modified (`ctime` *`inode change time`*), the file content last modified (`mtime,` *`modification time`*), and last accessed (`atime,` *`access time`*)
- a **link count** telling how many hard links point to the i-node
- **pointers to the disk blocks** that store the file's contents (see i-node pointer structure below)

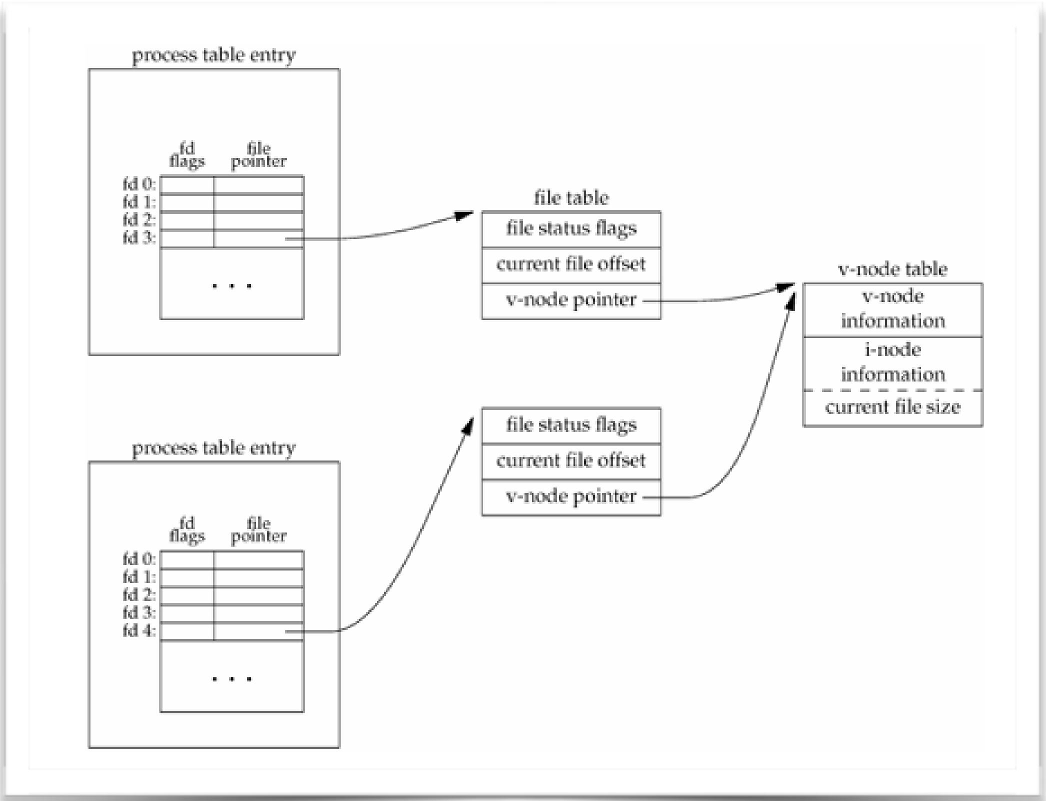
There are 15 pointers in the **i-node pointer structure**:

- 12 direct pointers — pointing directly to disk blocks. If the block size is 512 bytes, the max file size that could be created would be $12 \cdot 512\text{B} = 6{,}144\text{B}$.
- 1 singly indirect pointer — pointing to a block of pointers, each of which pointing to a disk block. This means that, if a pointer is 4 bytes, on a 512-byte block there can be 128 pointers, addressing a total of $128 \cdot 512\text{B} = 65{,}536\text{B}$.

- 1 doubly indirect pointer — pointing to a block of pointers, each of which pointing to another block of pointers, each of which pointing to a disk block. You get the idea.
- 1 triply indirect pointer — wouldn't it be fun to calculate the max file size for that?



The following figure shows the visual arrangement for 2 processes sharing the same file:



Having these tables in mind, the following changes can occur during a `write()` or an `lseek()` operation:

- the current file offset in the file table is incremented by the # of bytes written. If this exceeds the current file size, the current file size in the i-node table is set to the current offset (past the old EOF), *thus increasing the file size*;
- if the option O_APPEND was set, an "append" flag is set in the file status flags (from i-node), and the current file offset in the file table is set to the current file size from the i-node table (meaning, at the current EOF). Each `write()` appends contents past the old EOF, *thus increasing the file size*;
- the `lseek()` function only modifies the current file offset, with *no increase in the file size*;
- if `lseek()` is called with the SEEK_END option, the current file offset in the file table is set to the current file size (=EOF) from the i-node table, with *no increase in the file size*.

Since file descriptors are unique per process, but not unique per system, the following situations can occur:

- the same file descriptor can be used by different processes to refer to two different files in the file table;
- two different file descriptors from two different processes can point to the same entry in the file table.

To understand what happens when multiple processes access the same file, we must understand first the concept of atomic operations.

Atomic Operations

Example 1: Let's assume two processes, A and B, are using the same file. Each will maintain its own version of the file table. So, for instance, assume process A opened the file without the O_APPEND option and performed an `lseek()` that changed the offset to 1500 (say, EOF). Then assume that the kernel switches to process B, performing an `lseek()` to set the offset to the EOF, available to process B in its own file table. Then B could perform a `write()`, which will occur at the current offset, say 1500, but increasing the file by 100 bytes, to 1600. If the kernel switches back to process A, as per its own table, the current offset is still at 1500, which will cause A to overwrite the data of B.

This situation arises, potentially, when two different instances of the same program write the same log file.

The cause of this overwrite, in this case, is the fact that the logical operation of “position to the EOF and write” requires two separate system calls, `lseek()` and `write()`. The solution would be to have the `lseek()` and `write()` as an atomic operation. This is achieved by setting the O_APPEND flag, which causes `lseek()` to automatically go to EOF before each `write()`.

Example 2: Another example of an atomic operation would be the logical operation of “check if a file exists, and, if it doesn’t, create it”. This could be mistakenly attempted with a conditional call to `open()` (which returns an `fd=-1` if the file didn’t exist) followed by a call to `creat()`, thus allowing the `creat()` sys call to successfully create and return a valid `fd` if `open()` returned an `fd` of `-1`. The following (wrong) code snippet attempts to make the call to `creat()` conditional on the `-1` return value from `open()`:

```
if ((fd=open(pathname, O_WRONLY)<0)
{
     if (errno==ENOENT)
     {
          if ((fd=creat(pathname, mode)<0)
               printf(“creat error\n”);
     }
     else
          printf(“open error\n”);
}
```

However, if process B calls `creat()` and `write()` between the `open()` and `creat()`of process A, then the call to `creat()` by process A will erase the data written by process B. Therefore, `open()` and `creat()` must be coded as an atomic operation. To do this successfully, the call to `open()` must be made with the O_CREAT and O_EXCL options, which would make `open()` fail if the file already exists *without the separate calls* to `open()` and `creat()`.

dup() and dup2()

These functions duplicate a file descriptor:

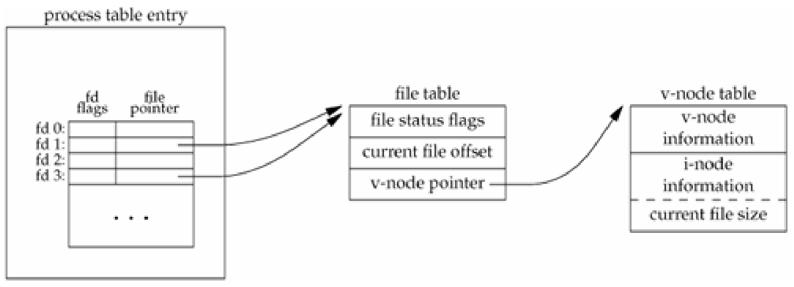
```
#include <unistd.h>

int dup(int fd);
int dup2(int fd, int fd2);
```

Both return new fd if OK, -1 if error.

The *`fd`* returned is the lowest available file descriptor for `dup()`, and the file descriptor specified by the second argument, for `dup2()`. If *`fd2`* is open, it is closed first. If *`fd=fd2`*, then `dup2()` returns *`fd`* without closing it.

What actually happens is that two file descriptors will point at the same file table:



The following program duplicates the file descriptor returned after the call to `open()` on file `large_file.txt`, source code mydup.c:

```
#include <unistd.h> /* for dup() */
#include <fcntl.h> /* for open() */
#include <sys/types.h> /* for O_RDONLY */
#include <stdio.h> /* for printf() */

int main(int argc, char *argv[])
{
     int fd=open("/Users/awise/Stevens/Lecture3/large_file.txt",
O_RDONLY);
     int new_fd=dup(fd);
     int set_new_fd=dup2(fd, 44);

     char buff[4];
     printf("Original file descriptor fd=%d\n", fd);
     printf("Duplicated file descriptor new_fd=%d\n", new_fd);
     printf("Set duplicated file descriptor set_new_fd=%d\n",
set_new_fd);
return 0;
}
```

The output of this program is:

```
Original file descriptor fd=3
Duplicated file descriptor new_fd=4
Set duplicated file descriptor set_new_fd=44
```

`sync()`, `fsync()`, `fdatasync()`

UNIX has a **buffer cache** in the kernel through which most disk I/O passes. When we write data into a file, it is first copied by the kernel into the buffer cache and queued for writing to disk at a later time. This is called **delayed write**, and is not the same thing as the "fully buffered" buffering type provided by standard I/O. The delayed-write blocks are written to disk, usually when the buffer cache needs to be cleared for another I/O operation (write back). The functions `sync()`, `fsync()`, and `fdatasync()` ensure consistency between buffer cache and disk.

```
#include <unistd.h>

int fsync(int fd);
int fdatasync(int fd);
                                        Both return 0 if OK, -1 if error.
void sync(void);
```

`sync()` is called periodically (every 30s) from a system daemon. It queues all the modified block buffers for writing, and returns. `fsync()` refers to a single file, specified by *`fd`*, and waits for the actual write to occur. It is used to make sure an application has written its data blocks onto disk. `fdatasync()` also applies to a single file, but only refers to the data portions of the file.

The `fcntl()` System Call

```
#include <fcntl.h>

int fcntl(int fd, int cmd, …, /* int arg */);
                         Returns: depends on cmd if OK, -1 otherwise.
```

`fcntl()` serves five different purposes:

1. Duplicate an existing file descriptor:
   `cmd``=F_DUPFD or F_DUPFD_CLOEXEC`
2. Get/set file descriptor flags (in process table):
   `cmd``=F_GETFD or F_SETFD`
3. Get/set file status flags (in file table):
   `cmd``=F_GETFL or F_SETFL`
4. Get/set asynchronous I/O ownership:
   `cmd``=F_GETOWN or F_SETOWN`
5. Get/set record locks:

`cmd=F_GETLK, F_SETLK, or F_SETLKW`

The `cmd` options are:

F_DUPFD	Duplicate file descriptor `fd`, the lowest # available >= `arg`. The effect is the same as after `dup()` (different entries in process table, same entry in file table). Its FD_CLOEXEC file descriptor flag is cleared (`fd` is open across an `exec()`).
F_DUPFD_CLOEXEC	Duplicate `fd` and set the FD_CLOEXEC file descriptor flag (from process table).
F_GETFD	Returns the file descriptor flags (proc table) for `fd`.
F_SETFD	Set the file descriptor flags for `fd`. The set value is `arg`.
F_GETFL	Returns the status flags for `fd` (file table). They are the same we described for `open()`:

File Status Flag	Description
O_RDONLY	open for read only
O_WRONLY	open for write only
O_RDWR	open for read/write
O_EXEC	open for execute only
O_SEARCH	open dir for search only
O_APPEND	append on each write
O_NONBLOCK	non-blocking mode
O_SYNC	wait for writes to complete (data and attributes)
O_DSYNC	wait for writes to complete (data only)
O_RSYNC	sync reads and writes
O_FSYNC	wait for writes to complete (BSD and OS X only)
O_ASYNC	asynchronous I/O (BSD and OS X only)

Because the first 5 access-mode flags are not separate bits, the O_ACCMODE mask is used to select the desired one through a bitwise AND operation.

F_SETFL	Set file status flags (file table) to value specified by `arg`.
F_GETOWN	Get the PID and process GID currently receiving the SIGIO and SIGURG signals (to be described later).
F_SETOWN	Set the PID and process GID to receive the SIGIO and SIGURG signals to modulus of `arg` (`arg` can be negative).

The following program lists the file status flags (we call the first 5 *access modes*):

```
#include <fcntl.h>
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h> /* for atoi() */

int main(int argc, char *argv[])
{
     int val;

     if (argc!=2)
          printf("Usage ./mystatusflags.c <file_descriptor>");

     if ( (val=fcntl(atoi(argv[1]), F_GETFL, 0))<0)
          printf("Invalid fcntl() return value val=%d\n", val);

     switch (val&O_ACCMODE)
     {
          case O_RDONLY:
               printf("Access mode=\"read only\"\n");
               break;
          case O_WRONLY:
               printf("Access mode=\"write only\"\n");
               break;
          case O_RDWR:
               printf("Access mode=\"read/write only\"\n");
               break;
          default:
               printf("Unknown access mode\n");
     }

     if (val==O_APPEND)
          printf("Status flag=\"append\"\n");
     if (val==O_NONBLOCK)
          printf("Status flag=\"non-blocking\"\n");
     if (val==O_SYNC)
          printf("Status flag=\"synchronous write\"\n");

return 0;
}
```