System Programming
(ELEC462)

Connection Control

Dukyun Nam
HPC Lab@KNU

Contents

Introduction

- Objectives
  - Similarities and differences between files and devices
    - How to use those ideas to manage connections to devices.
  - Attributes of connections
  - Race conditions and atomic operations
  - Controlling device drivers
  - Streams
- What to learn and play with
  - System Calls: `fcntl, ioctl`
  - Functions: `tcsetattr, tcgetattr`
  - Commands: `stty, write`

Devices Are Just Like Files

- Files:
  - Contain data
  - Have properties
  - Identified by names in directories
    - We can read and write bytes from and into a file
- Devices
  - Microphone, ear phone, speaker, keyboard, USB, …
    - Attached to a computer
  - Indeed, *terminal*, with keyboard and display, behaves like a file
    - Keystrokes that we type can be read by a program as input data
    - Characters that a program writes to the terminal are displayed on the screen

Devices Are Just Like Files (cont.)

- Device I/O is like file I/O
  - Devices are just like files
  - You get data from them and send data to them
- In Linux (Unix)
  - Various kinds of devices, such as sound/ graphic cards, terminals, mice, and disks, are treated as the same sort of object
  - Every device is treated as a “file” with:
    - A filename, an inode number, an owner, a set of permission bits, and a last-modified time

Devices Have Filenames

- Every device attached to a Linux machine is represented by a filename
  - You communicate with the device with

    `read(), write(), open(), close(), lseek()`
- `/dev` directory
  - Files that represent devices are in the `/dev` directory
  - A **device file** is an interface to a *device driver*
  - A **device driver** is a computer program that operates or controls *a particular type of device*

Devices Have Filenames (cont.)

```
$ tty
/dev/pts/2
$ cp /etc/passwd /dev/pts/2
$ ls > /dev/pts/2
```

- Types of devices
    - Character device
        - Device with which the driver communicates by sending and receiving data in unit of character
        - e.g., Terminals, serial ports, parallel ports, sound cards…
    - Block device
        - Device with which the driver communicates by sending and receiving data in unit of block.
        - e.g., HDD, SSD, …

Devices Have Filenames (cont.)

```
dynam@DESKTOP-Q4IJBP7:~/lab6$ ls -li /dev/pts/0
3 crw--w---- 1 dynam tty 136, 0 Oct  4 18:03 /dev/pts/0
```

- `tty`
  - Used to print the file name of the terminal connected to standard input
  - “136”: an identifier representing the device driver (a program controlling a device, of a terminal)
  - “0”: the specific terminal number: type “`ls -li /dev/pts/0`” to see a different minor number
    - inode, mode, link, owner, group, major # (device driver), minor # (device), Last modified, filename
    - `pts`: pseudo-terminal slave (cf. `ptmx`: pseudo-terminal master)
    - Directory(d), Symbolic Link(l), Character Device(c), Block Device(b)
  - * `tty` (TeleTYpewriter): Refers to the original printing terminals manufactured by the Teletype Corporation
    - https://en.wikipedia.org/wiki/Teleprinter

Terminals Can be Treated Exactly Disk Files

- The `write` Command
  - Before instant messaging, chat rooms, or KakaoTalk, Unix users chatted with friends at other terminals with the write command.
  - Do `man 1 write`:

```
$ man 1 write
WRITE(1)              Linux Programmer's Manual              WRITE(1)

NAME
       write - send a message to another user

SYNOPSIS
       write user [ttyname]

DESCRIPTION
       Write allows you to communicate with other users by copy-
       ing lines from your terminal to theirs.
```

  - Sample screen shots

```
dynam@ubuntu:~$ tty
/dev/pts/9
dynam@ubuntu:~$ write dynam /dev/pts/10
Hi
CSE
^Cdynam@ubuntu:~$ |
```

```
dynam@ubuntu:~$ tty
/dev/pts/10
dynam@ubuntu:~$
Message from dynam@ubuntu on pts/9 at 02:08 ...
Hi
CSE
EOF
```

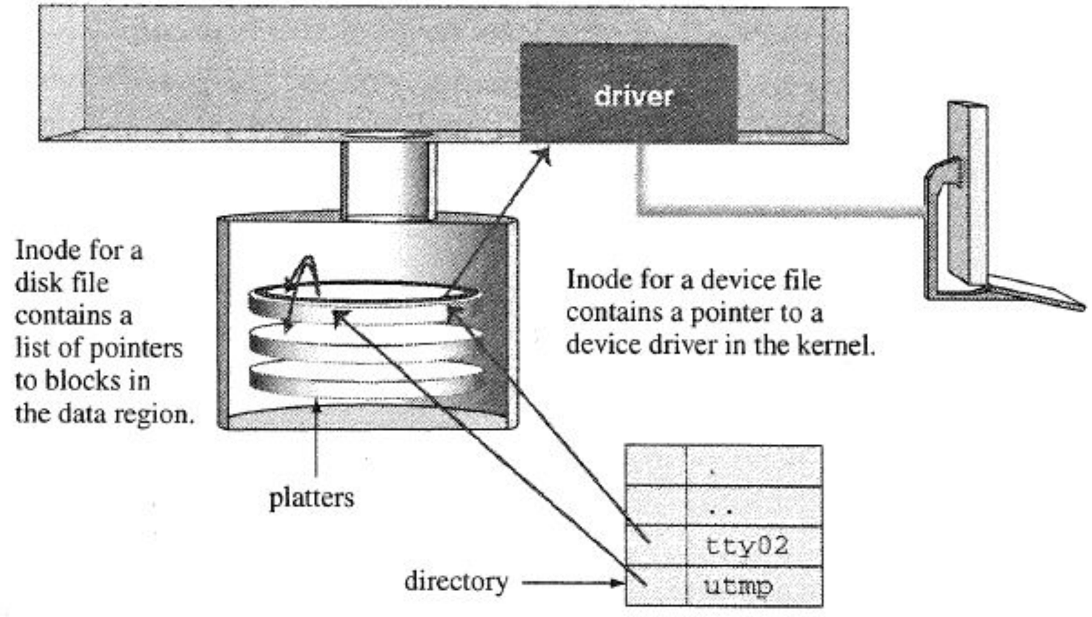
Write the `write` program: `write0.c`

- BUFSIZ: 8192
    - May be subject to change depending on system

```
$ ./write0 /dev/pts/1
Hi
KNU
^C
```

```c
/* write0.c
 *
 *      purpose: send messages to another terminal
 *       method: open the other terminal for output then
 *               copy from stdin to that terminal
 *        shows: a terminal is just a file supporting regular i/o
 *        usage: write0 ttyname
 */

#include        <stdio.h>
#include        <fcntl.h>
#include        <stdlib.h>
#include        <string.h>
#include        <unistd.h>

int main( int ac, char *av[] )
{
        int     fd;
        char    buf[BUFSIZ];

        /* check args */
        if ( ac != 2 ){
                fprintf(stderr,"usage: write0 ttyname\n");
                exit(1);
        }

        /* open devices */
        fd = open( av[1], O_WRONLY );
        if ( fd == -1 ){
                perror(av[1]); exit(1);
        }

        /* loop until EOF on input */
        while( fgets(buf, BUFSIZ, stdin) != NULL )
                if ( write(fd, buf, strlen(buf)) == -1 )
                        break;
        close( fd );

        return 0;
}
```

Device Files and Inodes

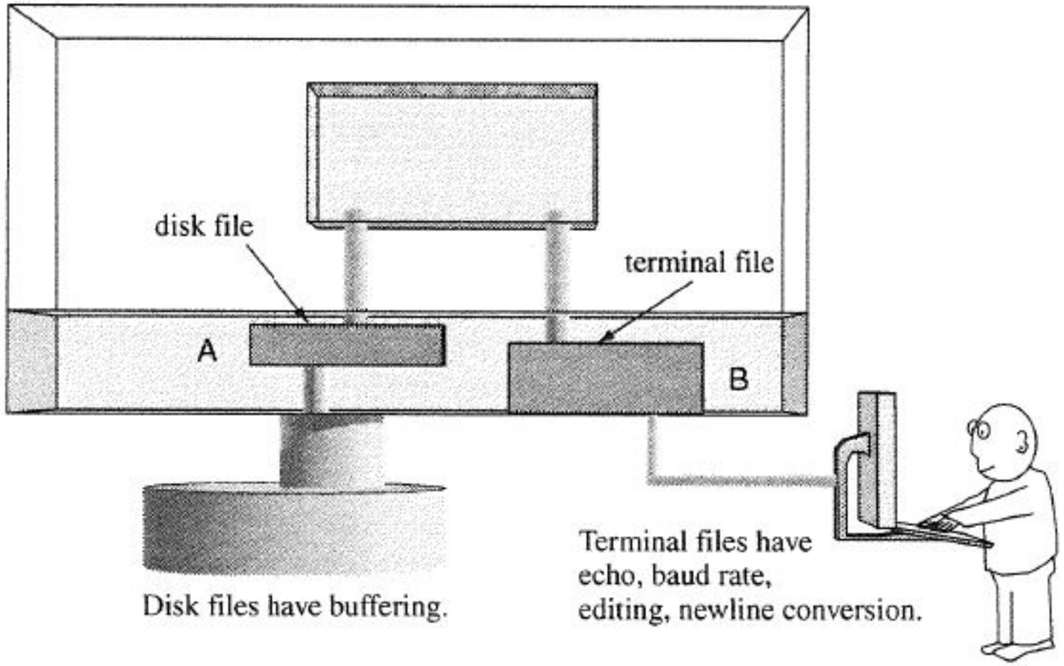


< Inode points to data blocks or to driver code >

- How do these device files work? How does the Unix/Linux file system of inodes and data blocks support this idea of device files?
- Consider how `read` works
  - The kernel finds the inode for the file descriptor
  - The inode tells the kernel the type of the file
  - If the file is a disk file
    - the kernel gets data by consulting the block allocation list
  - If the file is a device file
    - the kernel reads data by calling the read part of the driver code for that device

Device Files and Inodes (cont.)

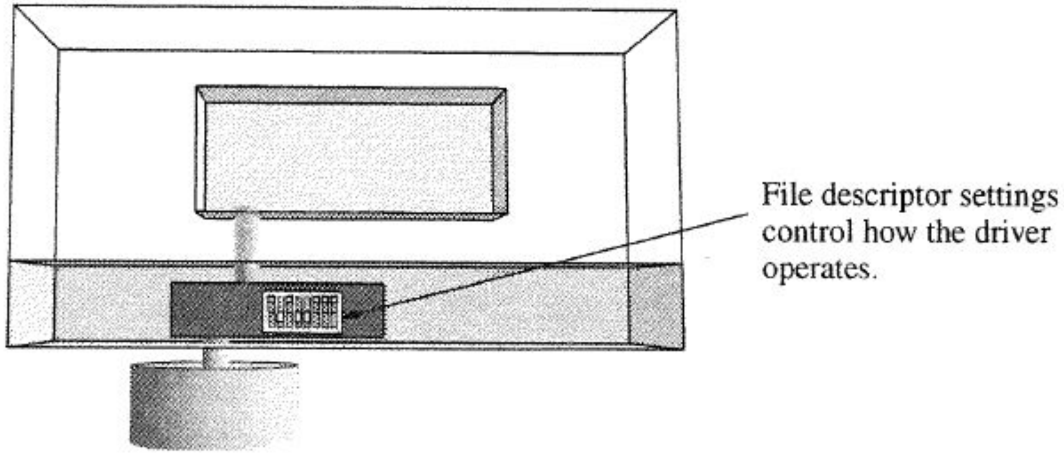
- Disk Files vs Device (here, Terminal) Files
  - Devices have their own attributes
    - e.g., Connection attributes available for the Terminal device
  - We can control the disk and connection attributes using data structures and system calls
  - baud rate: the rate at which information is transferred in a communication channel
    - “1024 baud”: “1K bits per sec."



< A process with two file descriptors >

Attributes of Disk Connections

- The `open` system call
  - Creates a connection between a process and a disk file
  - That connection has several attributes
  - Attribute 1: Buffering
  - Attribute 2: Auto-append mode



< A processing unit in a data stream >

System Call: fcntl

- Control a file descriptor by reading and writing that descriptor (in integer)
- Perform operation cmd on the open file specified by fd

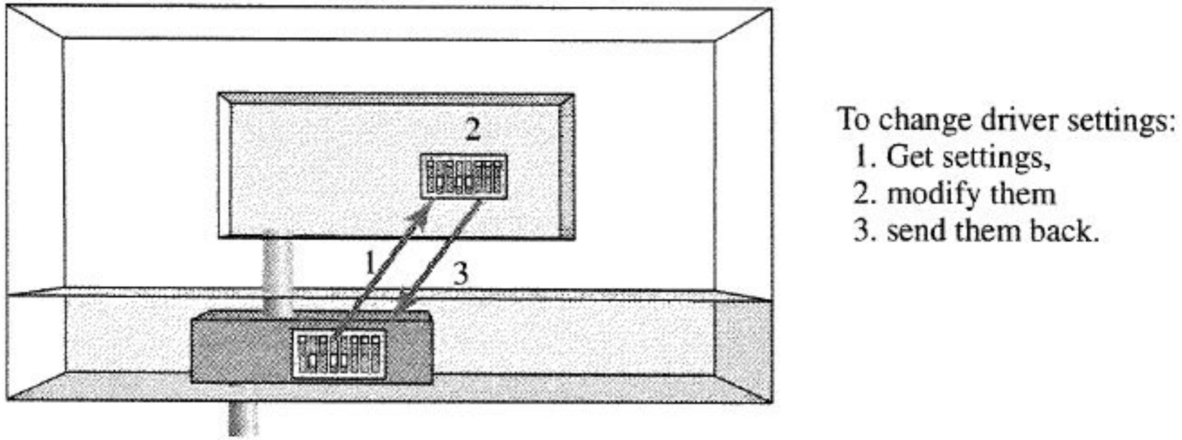
fcntl	
PURPOSE	Control file descriptors
INCLUDE	#include <fcntl.h> #include <unistd.h> #include <sys/types.h>

USAGE	int result = fcntl(int fd, int cmd);
	int result = fcntl(int fd, int cmd, long arg);
	int result = fcntl(int fd, int cmd, struct flock *lockp);

ARGS	fd the file descriptor to control cmd the operation to perform arg arguments to the operation lock lock information
RETURNS	-1 if error other depends on operation

Changing Settings – Disk File

- Example 1) Buffering can be set ON or OFF



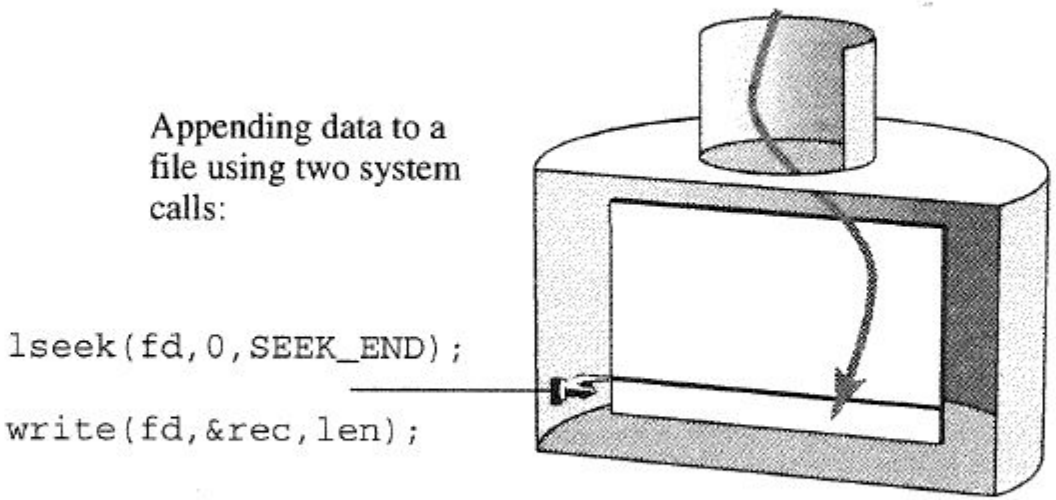
< Modifying the operation of a file descriptor >

- `s`: stores the information of control variables
- `O_SYNC`: set SYNC bit
  - Can be set to turn off kernel buffering
- `write()`s to disk will be synced to disk

```
#include  <fcntl.h>
int s;                                // settings
s = fcntl(fd, F_GETFL);               // get flags
s |= O_SYNC;                          // set SYNC bit
result = fcntl(fd, F_SETFL, s);       // set flags
if ( result == -1 )                   // if error
   perror("setting SYNC");            //    report
```

Changing Settings – Disk File (cont.)

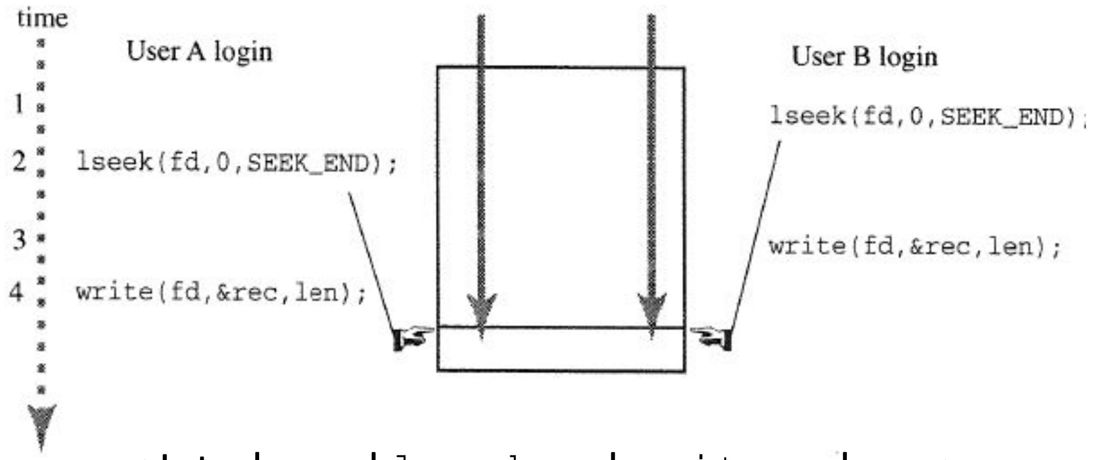
- Example 2) auto-append mode is another `fd` attribute
  - Consider a logfile, like wtmp
  - Each login is recorded by appending a record to the file
  - How does one append to a file?



< Appending with `lseek` and `write` >

Changing Settings – Disk File (cont.)

- What if two people login at the “same” time, how can this *Race Condition* be avoided?



< Interleaved `lseek` and `write` = chaos >

  - Time 1: B’s login process seeks to end of file
  - Time 2: B’s time slice up, A’s login process seeks to find end of file
  - Time 3: A’s time slice is up, B’s login process writes record
  - Time 4: B’s time slice is up, A’s login process writes record

Changing Settings – Disk File (cont.)

- Solution 1: Set `O_APPEND` attribute in `fd`
  - When the `O_APPEND bit` is set for `fd`, each call to `write` automatically, includes an `lseek` to the end of file
    - The kernel combines `lseek` and `write` into an ***atomic operation***
    - The two are joined into one indivisible unit

```c
#include  <fcntl.h>

int s;                              // settings
s = fcntl(fd, F_GETFL);             // get flags
s |= O_APPEND;                      // set APPEND bit
result = fcntl(fd, F_SETFL, s);     // set flags
if ( result == -1 )                 // if error
   perror("setting APPEND");           //    report
else
   write(fd, &rec, 1);              // write record at end
```

Changing Settings – Disk File (cont.)

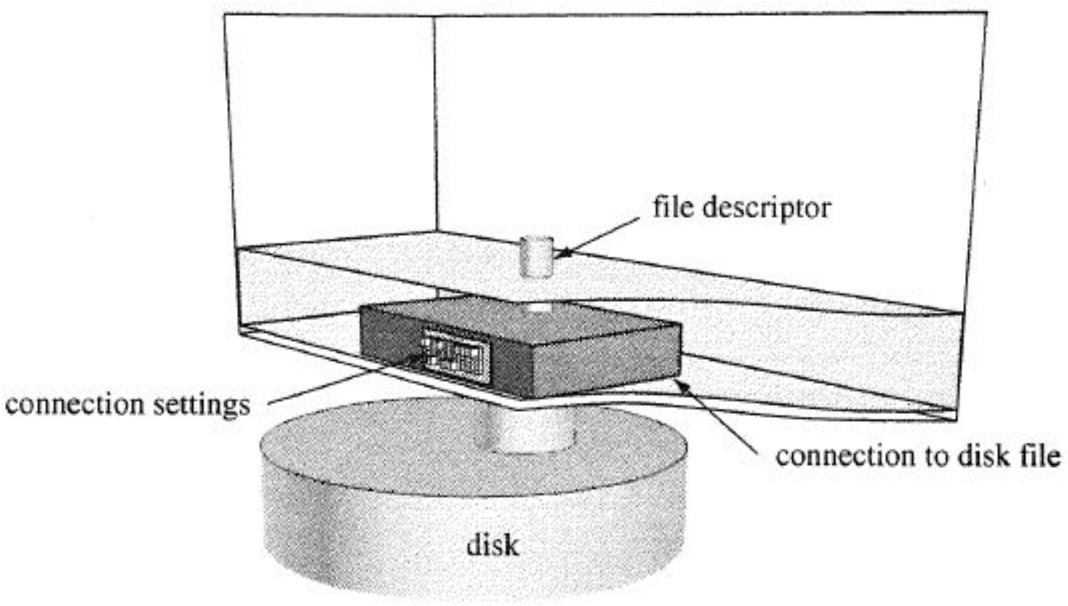
- Solution 2: Include flag in `open()`

```
$ fd = open(WTMP_FILE, O_WRONLY|O_APPEND|O_SYNC);
```

  - Opens the `wtmp` file for writing with `O_APPEND` and `O_SYNC` bits on
    - `O_APPEND`: The kernel combines lseek and write into an atomic operation, which can NOT be divided
    - `O_SYNC`: The kernel that calls to write should return only when the bytes are written to the actual disk (rather than to the kernel buffer)
  - Other flags
    - `O_CREAT`: Create the file if it does not exist.
    - `O_TRUNC`: If the file exists, truncate the file to length zero
    - `O_EXCL`: This flag is intended to prevent two processes from creating the same file. The second call returns -1

Disk Connections

- The kernel transfers data between disks and processes
  - A program can use the `open` and `fcntl` system calls to control the inner workings of these data transfers



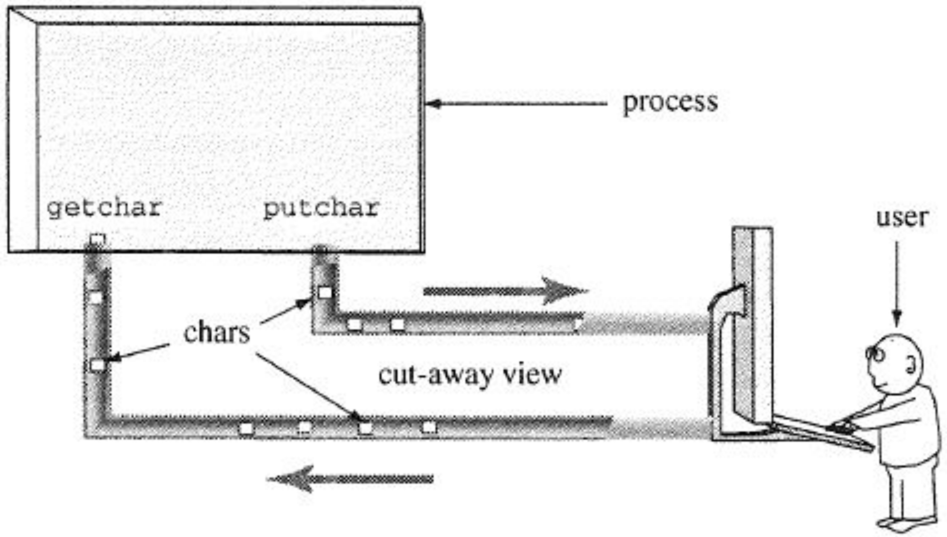
< Connections to files have settings >

Attributes of Terminal Connections

- The `open` system call
  - Creates a connection between a process and a terminal
  - All connections to `ttys` have attributes of regular `fd`'s
  - All connections to `ttys` have additional attributes appropriate for human interface

Terminal I/O Not As Simple As It Appears

- A connection between a terminal and a process looks simple…
  - You can use `getchar` and `putchar` to transfer bytes between device and process



< The illusion of a simple, direct connection >

Example) A Simple Experiment

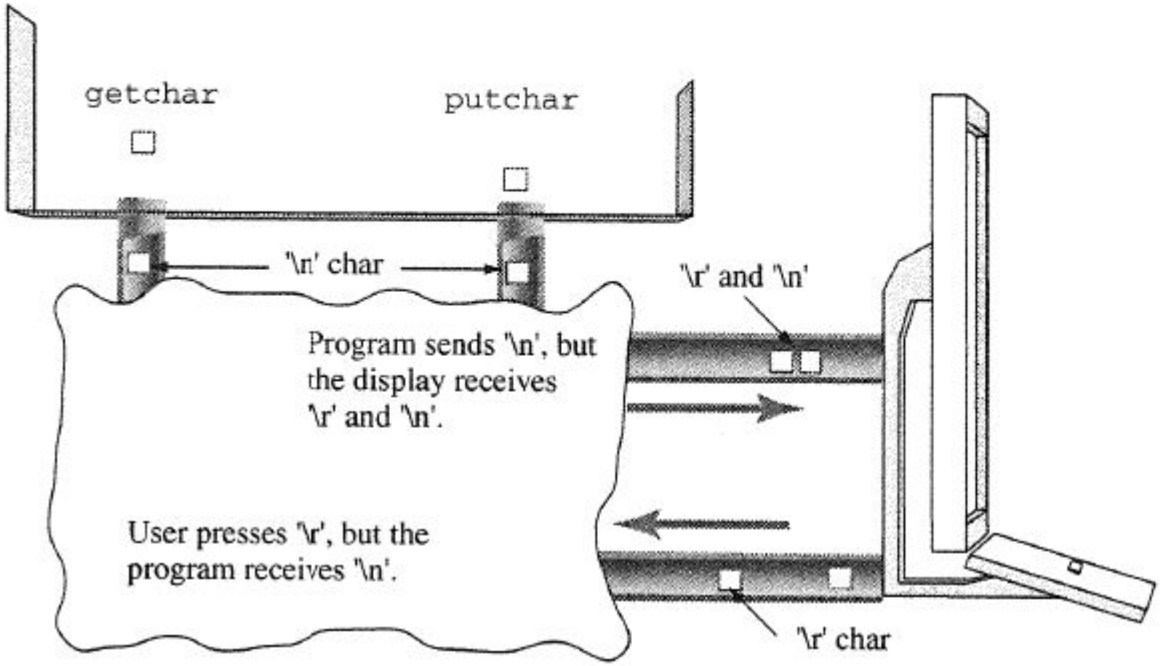
- The process receives no data until user presses Return
- User presses Return (ASCII 13), but processes sees newline (ASCII 10)
- Process sends newline, terminal receives Return-Newline pair

```c
/*  listchars.c
 *      purpose: list individually all the chars seen on input
 *       output: char and ascii code, one pair per line
 *        input: stdin, until the letter Q
 *        notes: usesful to show that buffering/editing exists
 */

#include        <stdio.h>

int main()
{
        int     c, n = 0;

        while( ( c = getchar()) != 'Q' )
                printf("char %3d is %c code %d\n", n++, c, c );

        return 0;
}
```

```
dynam@DESKTOP-Q4IJBP7:~/lab6$ ./listchars
hello
char   0 is h code 104
char   1 is e code 101
char   2 is l code 108
char   3 is l code 108
char   4 is o code 111
char   5 is
 code 10
```

Kernel Processes Terminal Data

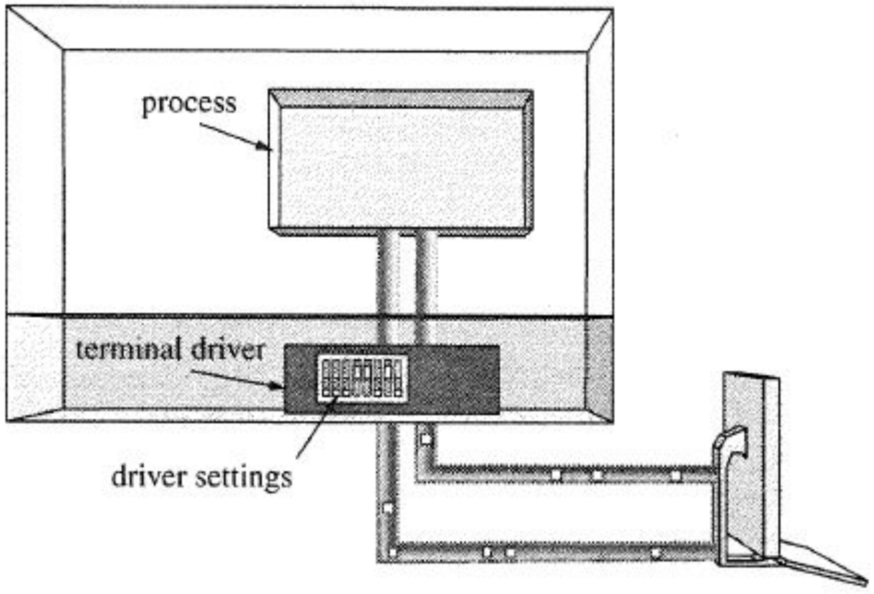
- There MUST be a ***processing layer*** somewhere in the middle of the file descriptor (of the Terminal device)



< Kernel processes terminal data >

The Terminal Driver

- The collection of kernel functions is called the *terminal driver*
  - Process data flowing between a process and the external device
- A connection between a terminal and a process
  - So, we may want to change the driver settings as we wish



< The terminal driver is part of the kernel >

Change Settings – Terminal

- The terminal (`tty`) driver is software in the kernel
  - It stands between the external device and the processes
- The driver contains lots of settings that control its operation
- The stty command allows you to examine and modify those settings

Change Settings – Terminal (cont.)

- The `stty` command
  - Lets users "read and change" settings in the driver for a terminal
    - `$ stty -a`
  - Some variables with values; e.g., rows, columns, …
  - Some variables with boolean; `icrnl, -olcuc`, etc. with on or off
    - `icrnl` stands for *Input: convert Carriage Return to NewLine*
    - `-olcuc` means to disable the action for *Output: convert LowerCase to UpperCase*
- How to change driver settings? Some examples:
  - `stty erase X` # make 'X' the erase key
  - `stty -echo` # type invisibly; turn off keystroke
  - `stty erase @ echo` # change erase char to @ and turn on echo mode

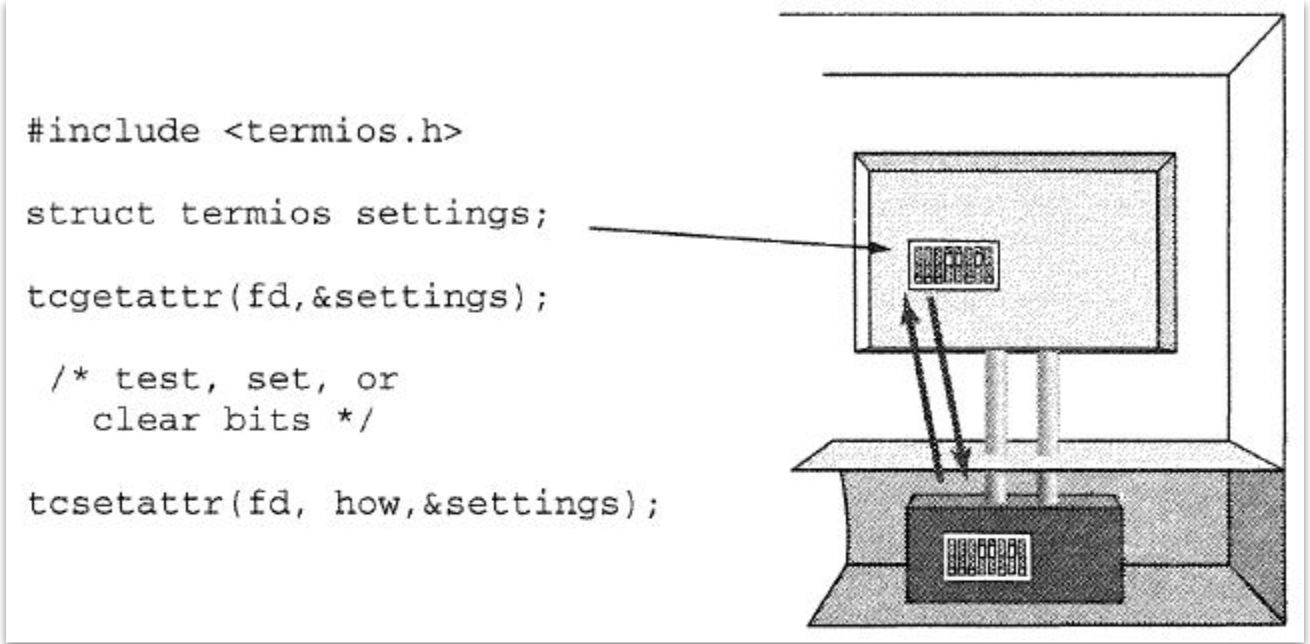
Change Settings – Terminal (cont.)

- The tty driver contains dozens of operations
- These operations are grouped into four categories:
  - 1) Input: what the driver does with chars “coming from” the terminal
    - Includes converting lowercase to uppercase and carriage returns to newlines
  - 2) Output: what the driver does with chars “going to” the terminal
    - Includes replacing tab chars by sequences of spaces, converting newlines to carriage returns, and converting lowercase to uppercase, …
  - 3) Control: how to represent characters: # of bits, parity, stop bits, etc.
    - Include *even parity, odd parity*, etc.
  - 4) Local: what the driver does while characters are inside the driver
    - Includes echoing keystrokes back to the user and buffering input until Return is pressed

Change Settings – Terminal (cont.)

- 3 steps for changing the terminal driver

```c
#include <termios.h>
struct termios attribs;               // struct to hold attributes
tcgetattr(fd, &attribs);              // Step 1: Get "attribs" from driver
settings.c_lflag |= O_ECHO;           // Step 2: Modify "attribs"
                                      //         Turn on ECHO bit in flagset
tcsetattr(fd, TCSANOW, &attribs);     // Step 3: Send "attribs" back to driver
```



< Controlling the terminal driver with `tcgetattr` and `tcsetattr` >

System Call: `tcgetattr()`

- Copies current settings from the terminal driver associated with the open file `fd` into the struct pointed to by info

tcgetattr		
PURPOSE	Read attributes from tty driver	
INCLUDE	#include <termios.h> #include <unistd.h>	
USAGE	int result = tcgetattr(int fd, struct termios *info);	
ARGS	fd info	file descriptor connected to a terminal pointer to a struct termios
RETURNS	-1 0	if error if success

System Call: `tcsetattr()`

- Copies driver settings from the struct pointed to by info to the terminal driver associated to the open file `fd`

tcsetattr		
PURPOSE	Set attributes in tty driver	
INCLUDE	#include <termios.h> #include <unistd.h>	
USAGE	int result = tcsetattr(int fd, int when,struct termios *info);	
ARGS	fd	file descriptor connected to a terminal
	when	when to change the settings
	info	pointer to a struct termios
RETURNS	-1	if error
	0	if success

  - `TCSANOW`: Update driver settings immediately
  - `TCSADRAIN`: Wait until all output already queued in the driver has been transmitted to the terminal. Then update the driver
  - `TCSAFLUSH`: Wait until all output already queued in the driver has been transmitted. Next, discard all queued input data. Then make the changes

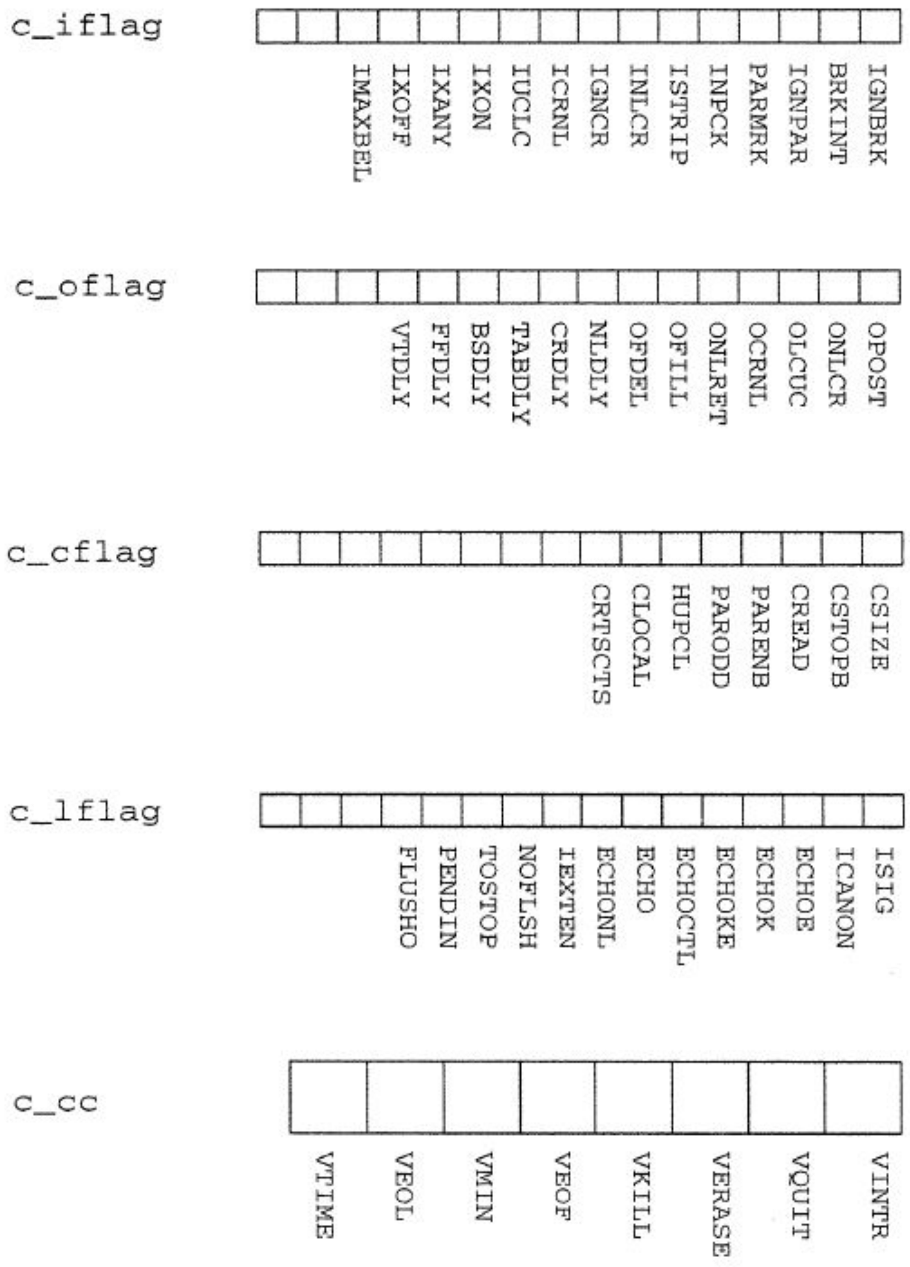
Struct `termios`

- Contains several flagsets and an array of control characters

```
struct termios
  {
    tcflag_t c_iflag;             /* input mode flags */
    tcflag_t c_oflag;             /* output mode flags */
    tcflag_t c_cflag;             /* control mode flags */
    tcflag_t c_lflag;             /* local mode flags */
    cc_t     c_cc[NCCS];          /* control characters */
    speed_t  c_ispeed;            /* input speed */
    speed_t  c_ospeed;            /* output speed */
  };
```

Bits and chars in `termios` members

- The first four members are flagsets
  - When the current attributes are read from the driver into a struct termios, all the values in this struct are available
- `c_cc`: an array of control characters
  - Stores “keystrokes” that perform various control functions



< Bits and chars in termios members >

Example: echostate.c

```c
/* echostate.c
 *   reports current state of echo bit in tty driver for fd 0
 *   shows how to read attributes from driver and test a bit
 */

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

int main()
{
        struct termios info;
        int rv;

        rv = tcgetattr( 0, &info );     /* read values from driver      */

        if ( rv == -1 ){
                perror( "tcgetattr");
                exit(1);
        }
        if ( info.c_lflag & ECHO )
                printf(" echo is on , since its bit is 1\n");
        else
                printf(" echo is OFF, since its bit is 0\n");

        return 0;
}
```

```
$ gcc -o echostate echostate.c
$ ./echostate
 echo is on , since its bit is 1
$ stty -echo


$ stty echo
```

Example: setecho.c – change state

```c
/* setecho.c
 *   usage:  setecho [y|n]
 *   shows:  how to read, change, reset tty attributes
 */

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

#define  oops(s,x) { perror(s); exit(x); }

int main(int ac, char *av[])
{
        struct termios info;

        if ( ac == 1 )
                exit(0);

        if ( tcgetattr(0,&info) == -1 )          /* get attribs   */
                oops("tcgettattr", 1);

        if ( av[1][0] == 'y' )
                info.c_lflag |= ECHO ;          /* turn on bit    */
        else
                info.c_lflag &= ~ECHO ;         /* turn off bit   */

        if ( tcsetattr(0,TCSANOW,&info) == -1 ) /* set attribs    */
               oops("tcsetattr",2);

        return 0;
}
```

```
$ ./echostate; ./setecho n; ./echostate ; stty echo
 echo is on , since its bit is 1
 echo is OFF, since its bit is 0
$ stty -echo; ./echostate; ./setecho y; ./setecho n
 echo is OFF, since its bit is 0
```

- Note that the driver and driver settings are stored in the kernel, NOT in the process.
    - That is, one process can change settings in the driver, and a different process can read or change the settings.

Example: `showtty.c`

- Display many driver attributes
  - For this example, we write part of the stty command
  - This one reads the settings and creates a report by testing bits and printing informative messages

```c
/* showtty.c
 *      displays some current tty settings
 */

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

struct flaginfo {
        int     fl_value;
        char    *fl_name;
};
```

```c
struct flaginfo input_flags[] = {

                IGNBRK  ,       "Ignore break condition",
                BRKINT  ,       "Signal interrupt on break",
                IGNPAR  ,       "Ignore chars with parity errors",
                PARMRK  ,       "Mark parity errors",
                INPCK   ,       "Enable input parity check",
                ISTRIP  ,       "Strip character",
                INLCR   ,       "Map NL to CR on input",
                IGNCR   ,       "Ignore CR",
                ICRNL   ,       "Map CR to NL on input",
                IXON    ,       "Enable start/stop output control",
                /* _IXANY  ,    "enable any char to restart output",    */
                IXOFF   ,       "Enable start/stop input control",
                0       ,       NULL };
```

Example: showtty.c (cont.)

```
struct flaginfo local_flags[] = {
                ISIG    ,       "Enable signals",
                ICANON  ,       "Canonical input (erase and kill)",
                /* _XCASE       ,       "Canonical upper/lower appearance", */
                ECHO    ,       "Enable echo",
                ECHOE   ,       "Echo ERASE as BS-SPACE-BS",
                ECHOK   ,       "Echo KILL by starting new line",
                0       ,       NULL };
```

```
void show_flagset( int thevalue, struct flaginfo thebitnames[] )
/*
 * check each bit pattern and display descriptive title
 */
{
        int     i;

        for ( i=0; thebitnames[i].fl_value ; i++ ) {
                printf( "  %s is ", thebitnames[i].fl_name);
                if ( thevalue & thebitnames[i].fl_value )
                        printf("ON\n");
                else
                        printf("OFF\n");
        }
}
```

```
void show_some_flags( struct termios *ttyp )
/*
 *      show the values of two of the flag sets_: c_iflag and c_lflag
 *      adding c_oflag and c_cflag is pretty routine - just add new
 *      tables above and a bit more code below.
 */
{
        show_flagset( ttyp->c_iflag, input_flags );
        show_flagset( ttyp->c_lflag, local_flags );
}
```

Example: `showtty.c` (cont.)

```c
void showbaud( int thespeed )
/*
 *      prints the speed in english
 */
{
        printf("the baud rate is ");
        switch ( thespeed ){
                case B300:      printf("300\n");        break;
                case B600:      printf("600\n");        break;
                case B1200:     printf("1200\n");       break;
                case B1800:     printf("1800\n");       break;
                case B2400:     printf("2400\n");       break;
                case B4800:     printf("4800\n");       break;
                case B9600:     printf("9600\n");       break;
                default:        printf("Fast\n");       break;
        }
}
```

```c
int main()
{
        struct  termios ttyinfo;        /* this struct holds tty info */

        if ( tcgetattr( 0 , &ttyinfo ) == -1 ){   /* get info */
                perror( "cannot get params about stdin");
                exit(1);
        }
                                                  /* show info */
        showbaud ( cfgetospeed( &ttyinfo ) );     /* get + show baud rate */
        printf("The erase character is ascii %d, Ctrl-%c\n",
                        ttyinfo.c_cc[VERASE], ttyinfo.c_cc[VERASE]-1+'A');
        printf("The line kill character is ascii %d, Ctrl-%c\n",
                        ttyinfo.c_cc[VKILL], ttyinfo.c_cc[VKILL]-1+'A');

        show_some_flags( &ttyinfo );              /* show misc. flags     */

        return 0;
}
```

System call: ioctl

- Provides access to the attributes and operations of the device driver connected to fd

ioctl		
PURPOSE	Control a Device	
INCLUDE	# include <sys/ioctl.h>	
USAGE	int result = ioctl (int fd, int operation [, arg..])	
ARGS	fd	file descriptor connected to device
	operation	operation to perform
	arg...	any args required for the operation
RETURNS	-1	if error
	other	depends on device

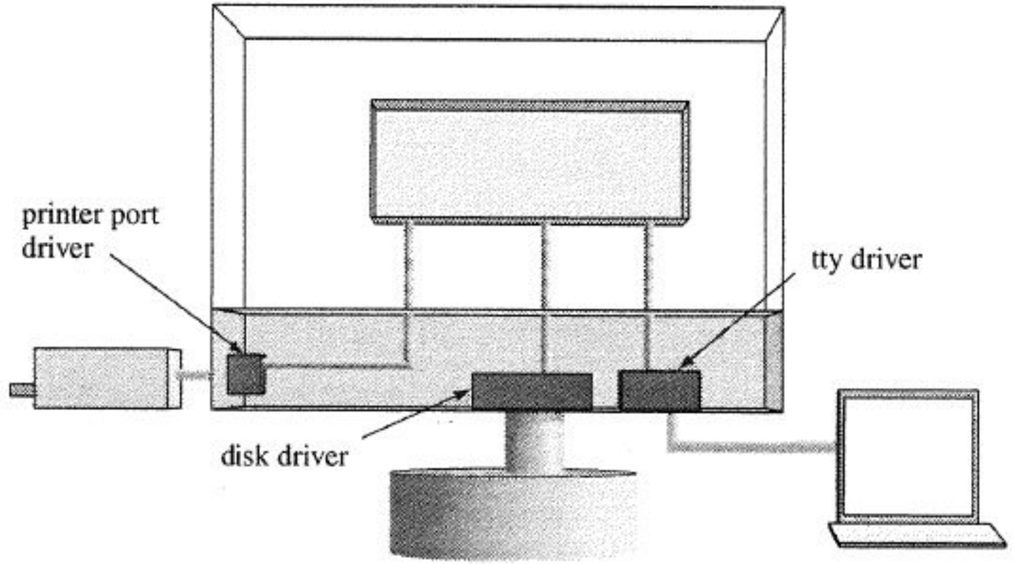
Programming Other Devices: `ioctl`

- Code for showing the settings of a video terminal screen having a size measured in rows and columns or in pixels:

```
#include <stdio.h>
#include <sys/ioctl.h>

void print_screen_dimensions(){
        struct winsize wbuf;

        if(ioctl(0, TIOCGWINSZ, &wbuf) != -1) {
                printf("%d rows x %d cols\n", wbuf.ws_row, wbuf.ws_col);
                printf("%d wide x %d tall\n", wbuf.ws_xpixel, wbuf.ws_ypixel);
        }
}

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

        return 0;
}
```

Summary

- Connections to disk files is different than connection to device files in terms of data processing and transfer
  - Device driver : kernel code to manage connections to a device
  - `fcntl` and `ioctl` can be used to read and change settings in a device driver



< File descriptors, connections, and drivers >

- File descriptors, connections, and drivers
  - Make sure every device attached to a Linux system is controlled by its respective device driver