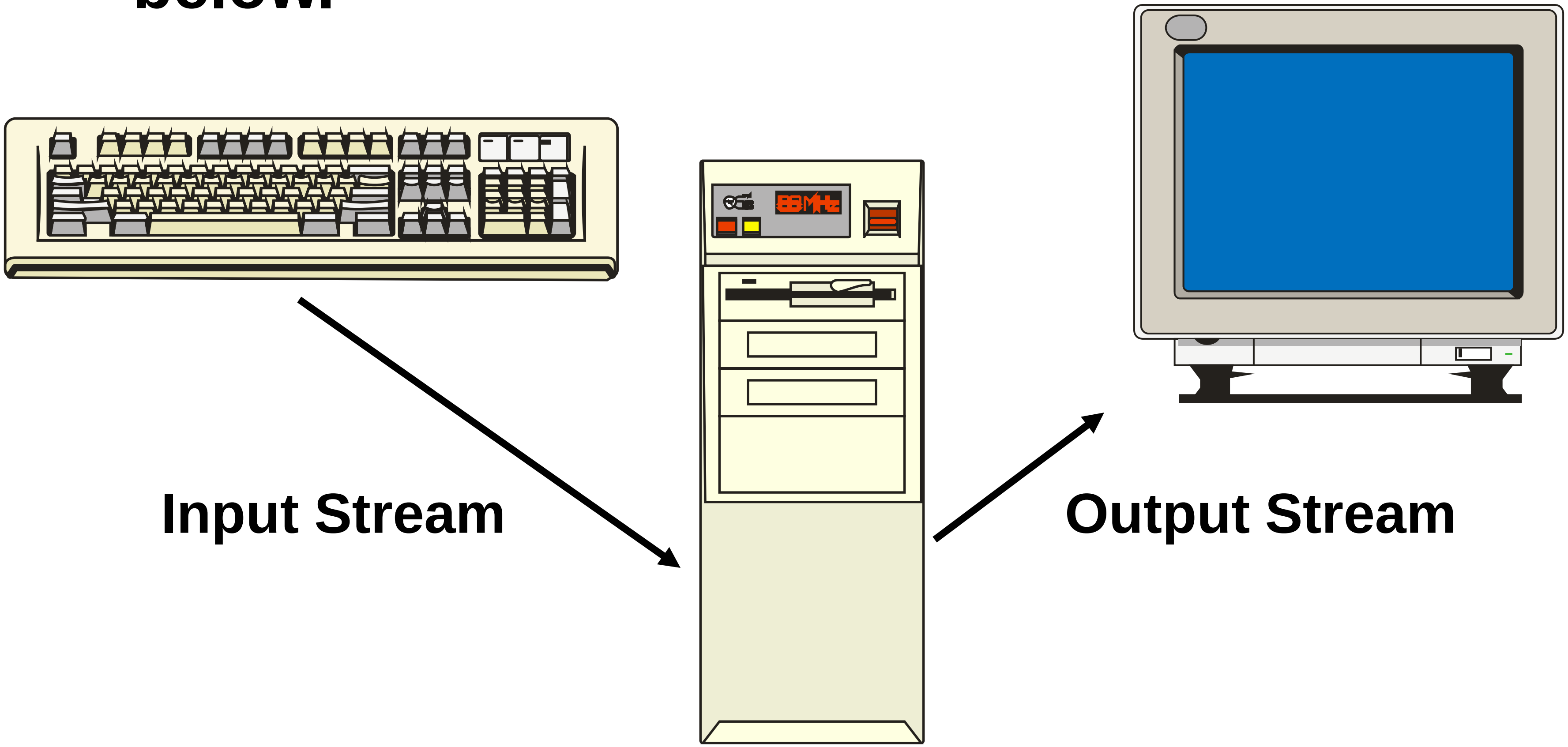
Chapter 5

Input/Output

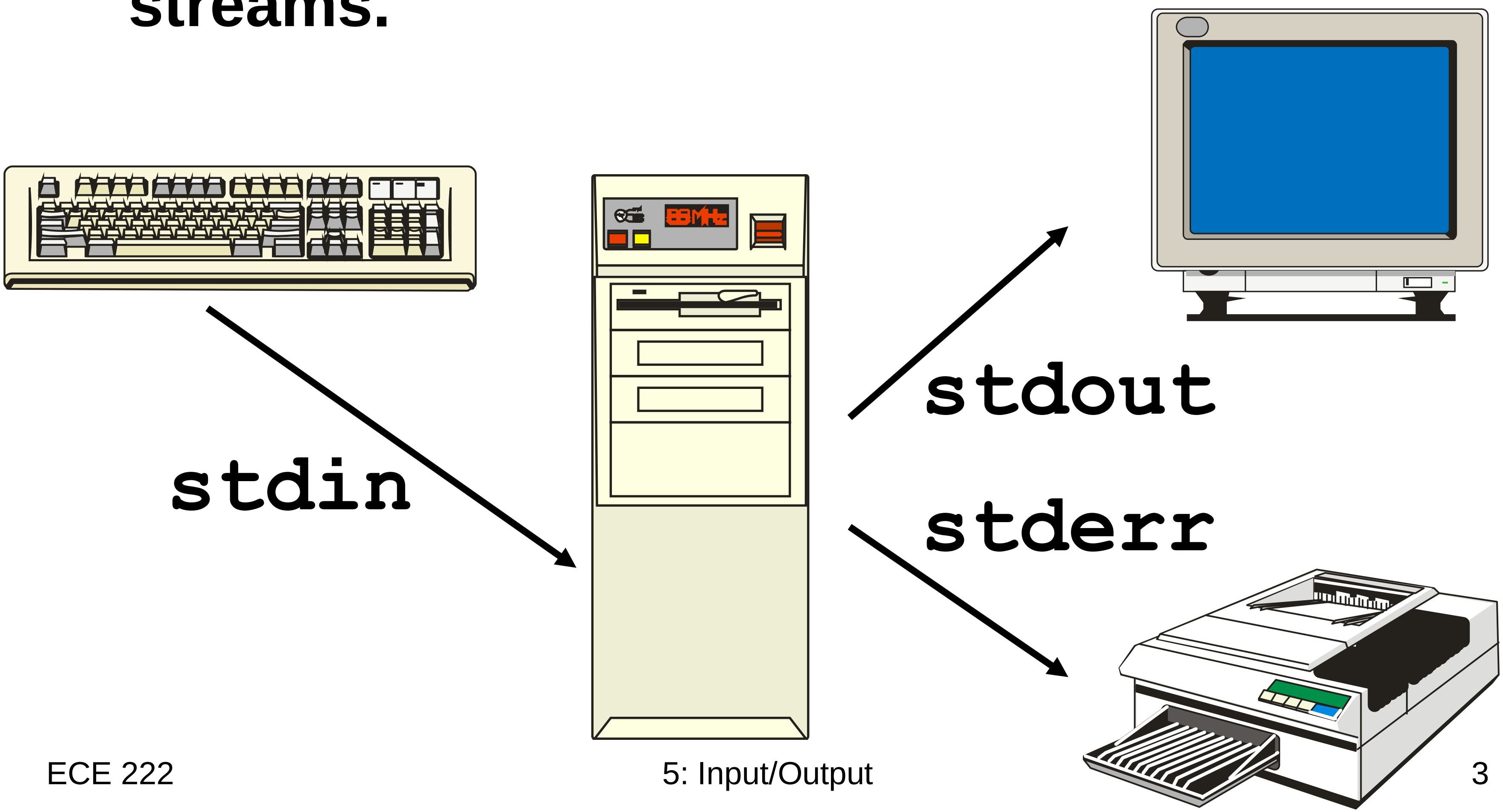
5.1 Streams

Streams are simply channels where data flows. For example, consider the two streams below.



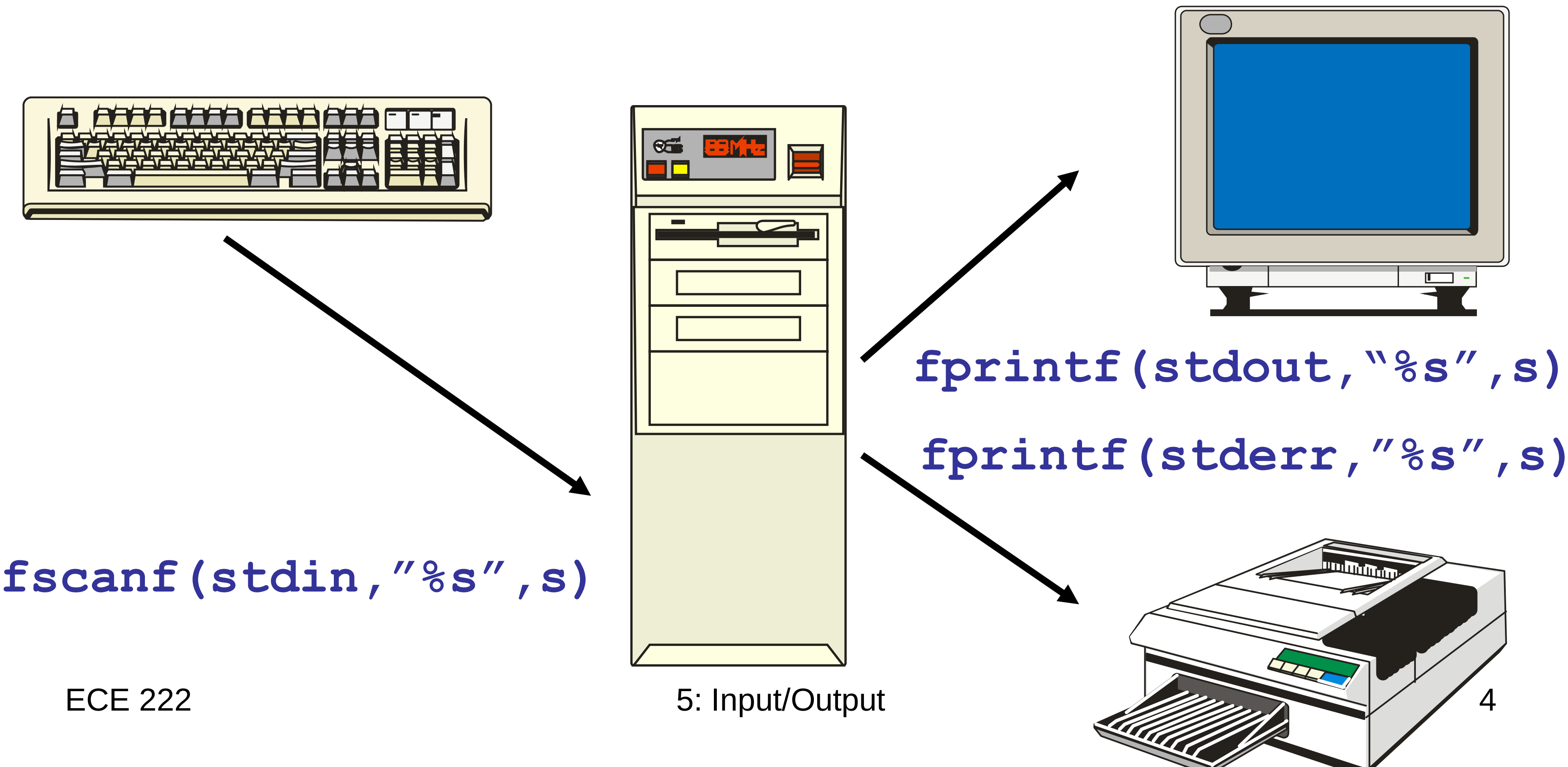
Default Streams

Every time a process is started, the O/S creates and recognizes three default (standard) streams.



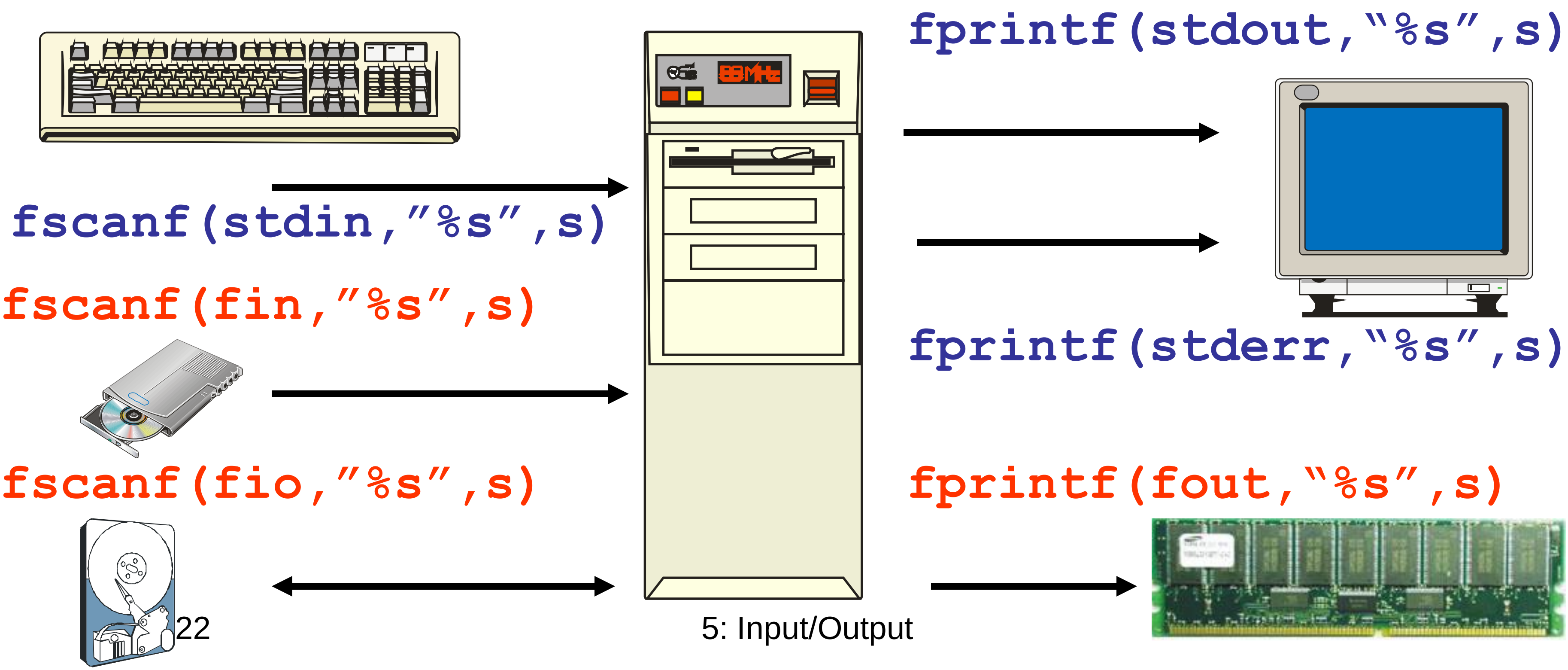
`scanf()` and `printf()`

The functions `scanf()` and `printf()` are actually special versions of `fscanf()` and `fprintf()` which operate on file streams.



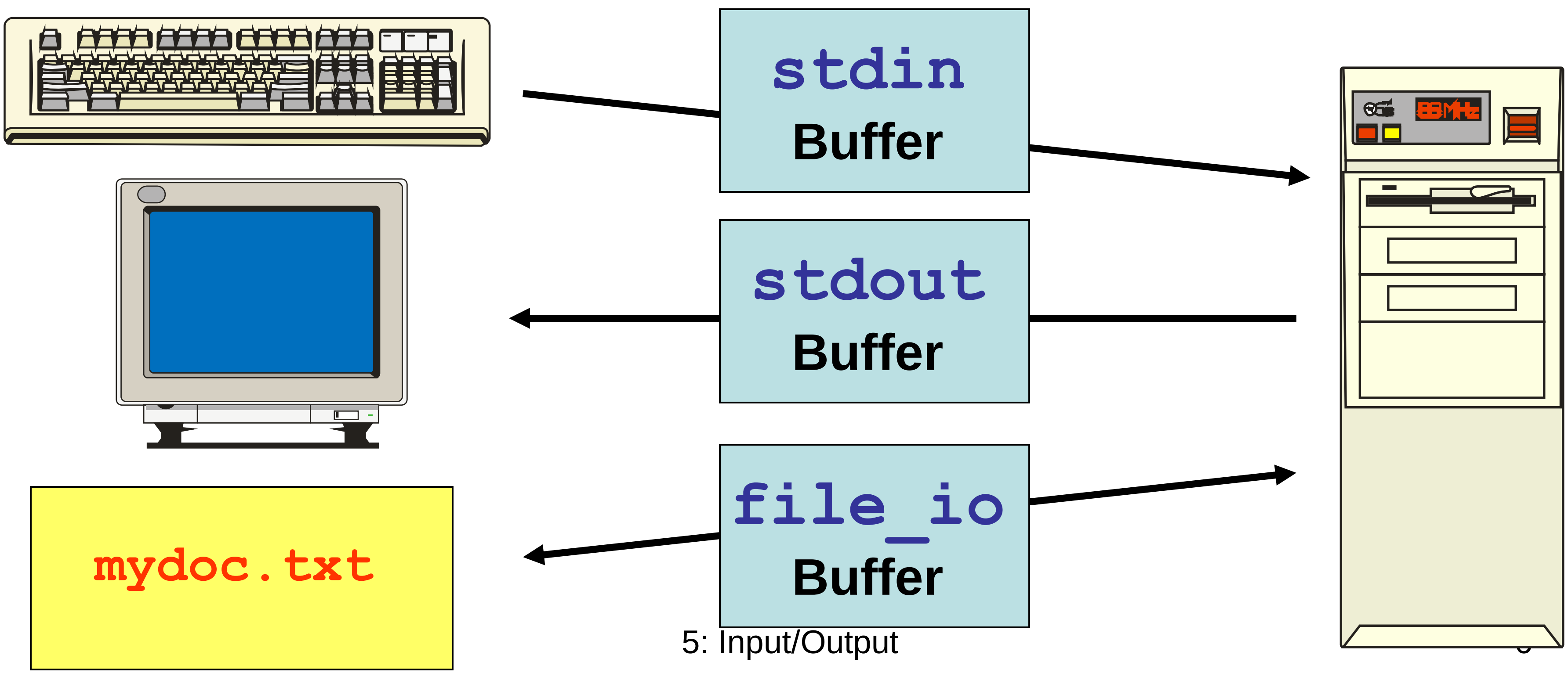
5.2 Buffers

The standard streams `stdout`, `stdin` and `stderr` are really just addresses of buffers which are read from and written to just like a file pointer.



Need for Buffers

Buffers are storage locations that act as “middle-men” between two communication devices that have different rates of flow.



Flushing Buffers

When a buffer actually "passes on" the data it contains to the other device it is called *flushing* the buffer. Buffers can be flushed at different instances.

Block Buffering – The buffer is flushed when it receives a certain amount of data (i.e. 1K, 64 K, etc...)

Line Buffering – The buffer is flushed when it receives a carriage return. (Typically ASCII streams.)

Unbuffered – The buffer doesn't act as a buffer and is flushed every byte.

Buffering Example

```
int i;

for (i=0; i<5; i++)
{ printf("i = %d ", i);
  sleep(1);
}

printf("\nPress Enter to Continue...");
getchar();
printf("\n");

for (i=0; i<5; i++)
{ printf("i = %d \n", i);
  sleep(1);
}
```

buffers.c

Notice no end-of-line character

Output is flushed

sleep(x): OS suspends program for x seconds

fflush()

The `fflush()` function can be used to force the operating system to empty ("flush") an ~~input~~ or output buffer.

buffers.c

C standard does not define behavior when **fflush(stdin)**

Windows OS: discards input in buffer

Linux: does not since the correct method is to use **fgets** and **sscanf**

```
printf("\nEnter a String -> ");
fscanf(stdin, "%s", s);
printf("%s\n", s);
```

buffers2.c

```
printf("\nEnter a String -> ");
fscanf(stdin, "%s", s);
printf("%s\n", s);
```

`fflush()` Example

Flushing a buffer can be very important when using `printf`s to analyze the operation of a program:

```
int main(void)
{ int i;
  int *ptri = (int *)100;

  for (i=0; i<5; i++) {
    printf("i = %d ", i);
    fflush(stdout);
  }
  *ptri = 3;
}
```

segfault.c
segfault2.c

Notice no end-of-line character

A warning about `fflush()`

Note that calling `fflush()` forces a context switch to the operating system. Therefore, it can slow your program down considerably.

`fflush.c`

```
fout = fopen("Buffer.out", "w");

ftime(&time0);
printf("Time0 = %d : %d \n", time0.time, time0.millitm);

for (i=0; i<10000; i++)
{ fprintf(fout, "i = %d\n", i); fflush(fout);
}

ftime(&time1);
printf("Time1 = %d:%d\n", time1.time, time1.millitm);
printf("Difference = %d\n\n", (time1.time - time0.time)
     * 1000 + time1.millitm - time0.millitm);
fclose(fout);
```

5.3 Pipes

Recall that the O/S always opens `stdin`, `stdout`, and `stderr` for each process.

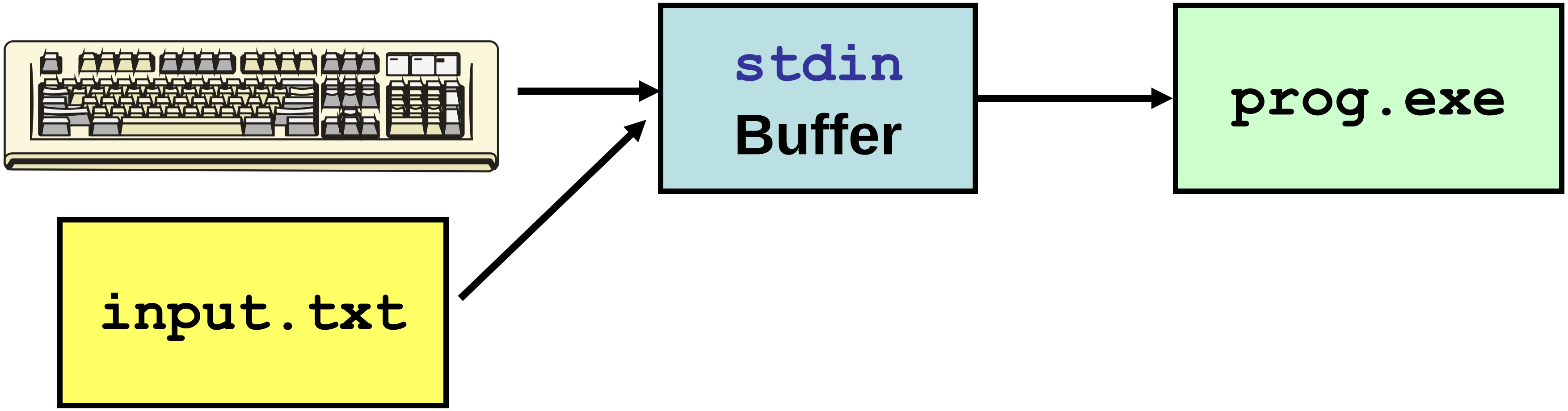
Shells provide three different operators which allows the user to re-direct input or output to different "devices."

- `<` `stdin` **comes from the file given.**
- `>` `stdout` **goes to the file given.**
- `>>` `stdout` **appends the file given.**
- `|` `stdin` **for the second program comes from the** `stdout` **of the first program.**

Pipe Example 1

The following command tells `prog.exe` to get its standard input from `input.txt`.

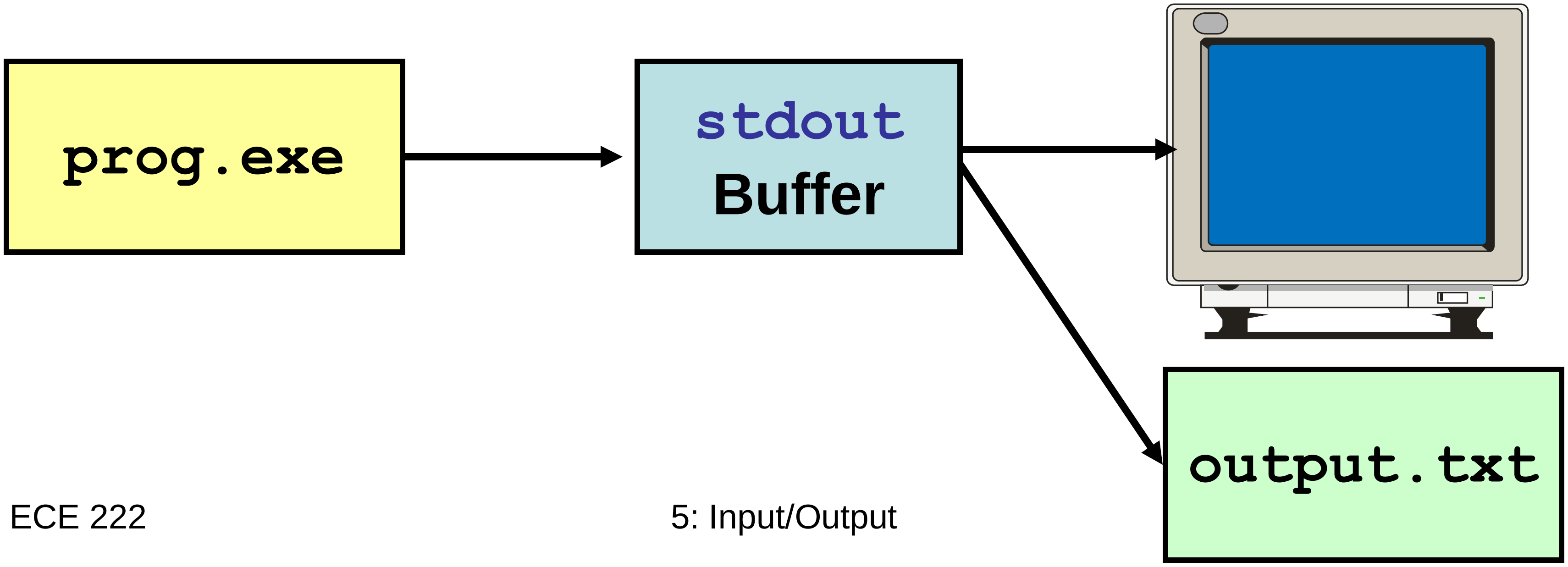
```
$ prog.exe < input.txt
```



Pipe Example 2

The following command tells `prog.exe` to send its standard output to `output.txt`.

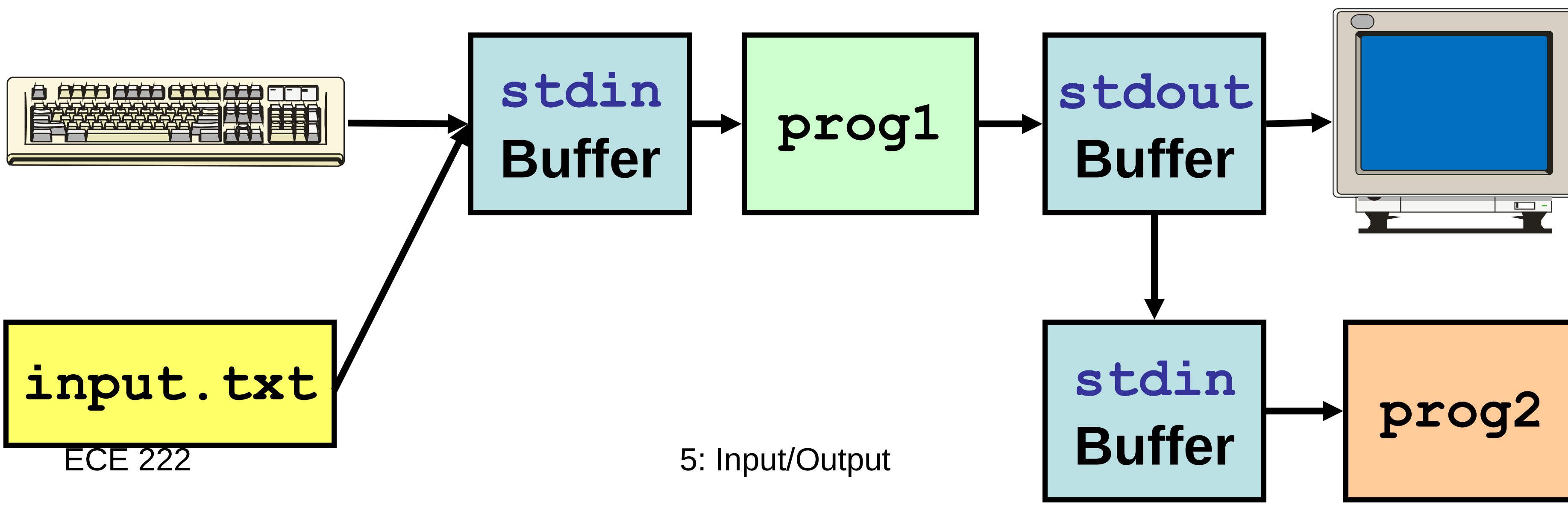
```
$ prog.exe > output.txt
```



Pipe Example 3

The following command tells `prog1` to get its standard input from `input.txt` and send its `stdout` to the `stdin` of `prog2`.

```
$ prog1 < input.txt | prog2
```



More Example Piping

prog_o.c

```
$ prog_o > out.txt
```

prog_i.c

```
$ prog_o | prog_i
```

prog_io.c

```
$ prog_io < in.txt > out.txt
```

Another Pipe Example

chain1.c

```
while (!feof(stdin)) {
      fgets(s, sizeof(s), stdin);
       found = sscanf(s, “%d”, &i);
       if (found) {
          f = (float)i * M_PI / 180;
          fprintf(stdout, "%f\n", sin(f));
       }
}
```

chain2.c

```
while (!feof(stdin)) {
     found = fscanf(stdin, "%f", &f);
      if (found) {
          f = asin(f);
          f = f*180/M_PI;
          fprintf(stdout, "f = %f\n", f);
       }
}
```

```
$ chain1 < in2.txt | chain2 > out.txt
```

Standard Pipes Programs

Linux systems usually come with a useful set of programs built for pipelining.

`grep` ***Searches for a given string*.**

`sort` ***Sorts lines of a text file*.**

`wc` ***Count lines, words and bytes*.**

`more` ***Pauses Lengthy Output*.**

`diff` ***Compares two Files*.**

Linux piping examples

```
cat output.txt | grep 8
prog_o | grep 1

cat output.txt | sort -r
prog_o | sort -r

prog_o | wc

prog_o | more

diff > difference.txt
```

5.4 Files

There are basically two ways to read data from a file in C program.

The first way is to use C standard library functions which are linked at compile time.

```
FILE *fpt;
char c;

fpt = fopen("testme","r");
fread(&c, 1, 1, fpt);
printf("%c\n", c);

fclose(fpt);
```

Reading from a File

The second way is to use O/S kernel system calls which are built into the kernel.

```
int fd;
char x;
fd = open(“testme”,”r”);
read(fd,&x,1);
printf(“%c\n”,x);
close(fd);
```

The first version actually ends up using these functions, but it provides a standardized call and implements *buffering*.

I/O Buffers

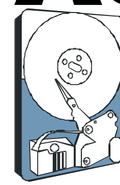
Buffering speeds up computer I/O transfers by holding data to be read or written in temporary storage—the *buffer—so* that large amounts of data can be transferred as a block, saving instruction cycles.

Consider reading a single byte from a hard disk. Using buffering, the system assumes that additional bytes near the one just read are likely to be read in the future. Therefore, the system transfers an entire block of bytes, surrounding the one requested, to a buffer.

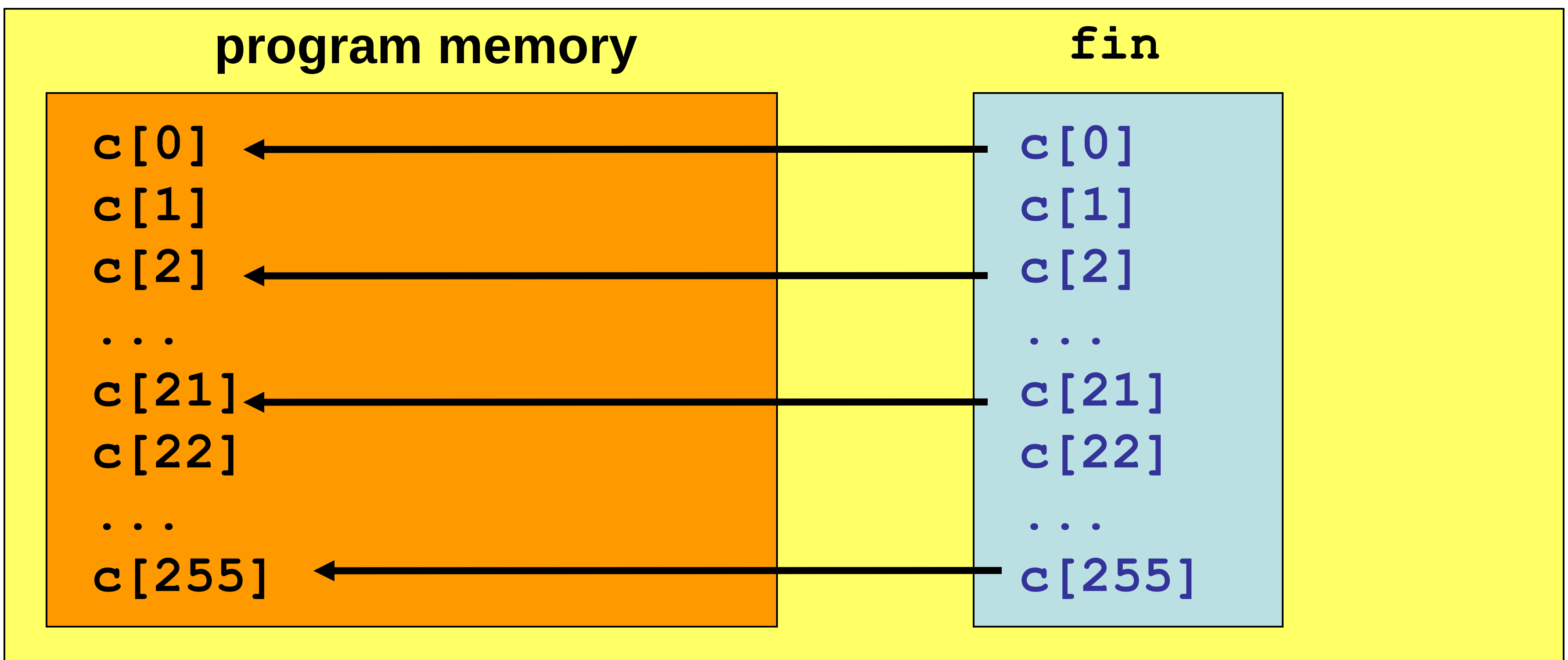
If any additional bytes are requested from nearby the original request, they are now available from the buffer.

A Data Buffer in Action

```
fscanf(fin,"%c",&c[21])
fscanf(fin,"%c",&c[2])
fscanf(fin,"%c",&c[255])
fscanf(fin,"%c",&c[0])
```



RAM (Main Memory)



Buffers are also used for Writes

```
printf("%s", "Go")
printf("%s", " Tigers!")
printf("%s", "\nBeat the
   Chickens!\n")
```

stdout

Go Tigers!

Beat the Chickens!

File I/O Functions

`fopen(const char *filename, const char *mode)`

Opens a file named by `filename` and returns a pointer to a stream (buffer).

`fread(void *ptr, size_t size, size_t n, FILE *stream)`

Reads `n` items of data each of length `size` bytes from the given input stream into a block pointed to by `ptr`. Returns blocks read.

`fwrite(const void *ptr, size_t size, size_t n, FILE *stream)`

Appends `n` items of data beginning at `ptr` each of length `size` bytes to the given output file, `stream`. Returns blocks written.

`fseek(FILE *stream, long offset, int where)`

Sets the file pointer associated with `stream` to a new position that is `offset` bytes from the file location given by `where`.

`ftell(FILE *stream)`

Returns the current file pointer for `stream`.

File I/O Functions

`fclose(FILE *stream)`

Closes the file designated by the pointer `stream`.

`rewind(FILE *stream)`

Sets the file pointer to the beginning of the file pointed to by stream `stream`.

`fprintf(FILE *str, const char *format[,arg, ...])`

Accepts a series of arguments which it applies to the format specifiers contained in the format string pointed to by `format` and outputs the formatted data to the file stream `str`.

`fscanf(FILE *str, const char *format[,addr, ...])`

Scans a input fields of a stream `str` one character at a time, then formats each according to a format specifier `format`. It stores the formatted input at an address passed to it as argument(s) `addr` following `format`.

Differences between `fread/fwrite` vs. `fscanf/fprintf`

The function calls `fscanf()` and `fprintf()` use `fread()` and `fwrite()` to work. They allow the programmer to read or write various sizes of data in one statement.

They also will ignore leading whitespace—spaces, tabs, carriage returns, linefeeds—when processing the buffer.

Generally `fscanf()` and `fprintf()` should be used only with "*text*" files and `fread()` and `fwrite()` are to be used with "*binary*" files.

“Read” vs. “Write”

The function `fopen()` can open files in various *modes*.

First, a file can be opened as read only, or modifiable.

- **`r` – Open an existing file to read from.**
- **`w` – Create a file to be written to. If the file already exists, it will be overwritten.**
- **`a` – Open a file to be appended to and create it if it does not exist.**
- **`r+` – Open an existing file for update (reading and writing).**
- **`w+` – Create a new file for update. If the file already exists, it will be overwritten.**
- **`a+` – Open a file to be read from, but only written to the end.**

Note: When a file is opened for update, both input and output can be done.

- Output, however, cannot be directly followed by input without a using `fseek()` or `rewind()` .
- Likewise, input cannot be directly followed by output without using `fseek()`, `rewind()`, or an input that encounters end-of-file

“Text” vs. “Binary”

- The function **`fopen()`** also allows two different types of files, text or binary.
- Linux/Unix: no difference between “text” and “binary” modes. All lines end with **`\n`**
- Windows:
  - text mode: translate carriage-return/linefeeds
    : (**`\r\n`**) translated to (**`\n`**)
    : (**`\n`**) translated to (**`\r\n`**)
  - binary: no translations occur

`BinVsText.c`

“Text” vs. “Binary”

What is the difference between writing a number to a “binary” file or a “text” file?

Actually, all files are binary files, of course, since everything in a computer is binary.

The difference is that in text files, all binary data represents printable *ASCII* characters, punctuation, or carriage returns and linefeeds.

How, then, would we write the number `100` to both file types, and how would it look in the file?

```
char c = 100;                // 100 = 0x64
fwrite(&c, 1, 1, fout);
fprintf(fout, "%d", c);
```

What would be written to each file?

What if we changed: `int i = 100; fwrite(&i, 4, 1, fout);` ***?***

File I/O Examples

fileio0.c

```
char filename[] = "file.txt";

int main(void)
{   FILE *f;
    int i, j, n;

    f = fopen(filename, "w");

    n = fwrite(&i, 4, 1, f);
    if (n != 1)
    { printf("Error writing to file.\n");
      fclose(f);  return (1);
    }

    fclose(f);

    f = fopen(filename, "r");

    n = fread(&j, 4, 1, f);
    if (n != 1)
    { printf("Error reading from file.\n");
      fclose(f); return(1);
    }
```

Cannot read from a file opened for writing

File I/O Examples

```
char filename[] = "outfile.txt";
char data[] = {71, 97, 109, 101, 99, 111, 99, 107,
               115, 32, 83, 116, 105, 110, 107, 33};

int main(void)
{   FILE *fout;
    char c;
    int i, n;

    fout = fopen(filename, "w");
    n = ftell(fout);  /* returns current position of stream */
    printf("Current File position = %d\n", n);

    /* Write some numbers to a file */
    fwrite("Go", 1, 2, fout);
    c = 0x20;
    fwrite(&c, 1, 1, fout);
    fwrite("Tigers!", 1, 7, fout);
    fwrite("\n", 1, 1, fout);
    fwrite(data, 1, 16, fout);
    fwrite("\n", 1, 1, fout);
...
```

fileio1.c

Can write block of data

File I/O Examples

fileio2.c

```
char filename[] = "infile.txt";

int main(void)
{   FILE *fin, *fout;
    char c;
    int i, n, length;

    if ((fout = fopen(filename, "w")) == NULL)
    {   printf("Error opening \"%s\"\n", filename);
        return (1);
    }
    for (i=0; i<40; i++)
    { n = ('0' + i);    fwrite(&n, 1, 1, fout);
    }
    for (i=0; i<10; i++)
    { n = i;            fwrite(&n, 4, 1, fout);
    }
    for (i=0; i<10; i++)
    { n = rand();       fwrite(&n, 4, 1, fout);
    }
    fclose(fout);
...
```

Make sure the size is correct

File I/O Examples

fileio3.c

```
char filename[] = "fileIO3.txt";
char str[256];

int main(void)
{   int i;
    float x;
    FILE *f;

    if ((f = fopen(filename, "wt")) == NULL)
    {   printf("Error opening \"%s\"\n", filename);
        return (1);
    }

    for (i=0; i<=90; i+=10)
    {   fprintf(f, "i = %i sin(%d) = %f\n",
                                     i, i, sin(i*3.1415926/180));
    }
    fclose(f);
```

File Attributes

```
harlanr@ubuntu:~/Ubuntu One/ece222/notes/ch5$ ls -l
total 608
drwxrwxr-x 2 harlanr harlanr  4096 Oct 24 14:51 backup
-rw-rw-r-- 1 harlanr harlanr   158 Oct 21 13:00 BinVsText.bin
-rw-rw-r-- 1 harlanr harlanr   745 Oct 24 14:48 binvstext.c
-rwxrwxr-x 1 harlanr harlanr  9720 Oct 24 17:10 binvstext.out
-rw-rw-r-- 1 harlanr harlanr   158 Oct 21 13:00 BinVsText.txt
-rw-rw-r-- 1 harlanr harlanr 88890 Oct 27 18:52 Buffer.out
-rw-rw-r-- 1 harlanr harlanr  1139 Oct 29 17:53 filestat.c
-rwxrwxr-x 1 harlanr harlanr  9274 Oct 29 16:57 filestat.out
-rwxr-xr-x 1 harlanr harlanr  1637 Oct 24 14:48 fixfiles.pl
-rwxrw-rw- 1 harlanr harlanr   589 Oct 29 17:57 ls1.c
-rwxrwxr-x 1 harlanr harlanr  8789 Oct 29 17:57 ls1.out
-rw-rw-r-- 1 harlanr harlanr   171 Oct 24 14:56 Makefile
```

Size in bytes

Date and time last modified

Owner, group name

Permissions: read, write, execute

File Attributes

File type: - file, directory, link

```
chmod u-x ls1.c
```

```
drwxrwxr-x 2 harlanr harlanr  4096 Oct 24 14:51 backup
-rw-rw-r-- 1 harlanr harlanr   745 Oct 24 14:48 binvstext.c
-rwxrwxr-x 1 harlanr harlanr  9720 Oct 24 17:10 binvstext.out
-rwxr-xr-x 1 harlanr harlanr  1637 Oct 24 14:48 fixfiles.pl
-rwxrw-rw- 1 harlanr harlanr   589 Oct 29 17:57 ls1.c
```

Permissions: r, w, x
- Permission denied
u: user
g: group
o: all others

who

permission

```
chmod [ugo][+-][rwx] name
```

what: + add - remove

File Statistics

filestat.c

```
#include <sys/stat.h>

int main(int argc, char *argv[])
{ struct stat f_stat;       /* returned info about file */
  int i,n;

  if (argc != 2)
  { printf("Usage: filestat filename\n");
    exit(1);
  }

  if (stat(argv[1], &f_stat) == -1)
  { printf("Error %s\n", argv[1]);
    exit(1);
  }

  printf("mode:    %o\n", f_stat.st_mode);
  printf("links:   %d\n", f_stat.st_nlink);
  printf("user:    %d\n", f_stat.st_uid);
  printf("group:   %d\n", f_stat.st_gid);
  printf("size:    %d\n", f_stat.st_size);
  printf("modtime: %d\n", f_stat.st_mtime);
…
```

Directory Functions

`opendir(char *dirname)`

Opens a directory stream given by `dir_name` for reading.

`readdir(DIR *ptr_dir)`

Reads the current entry from a directory stream pointed to by `ptr_dir`.

`closedir(DIR *ptr_dir)`

Closes a directory stream pointed to by `ptr_dir`.

`stat(const char *file_path, struct stat *stat_buf)`

Stores information about the file `file_path` in the status buffer `stat_buf`.

`ls1.c`

5.5 Devices

Devices are peripherals connected to a computer.

As we saw earlier, as far as input or output to and from these devices are concerned, devices are treated as if they were files.

Where are the associated files for these devices kept? `/dev`

Example Device Program

The "file" for the mouse is usually located in a file called `/dev/psaux`.

```
int main(void)
{
  FILE *fpt;
  int c;
  int buf[30];

  fpt = fopen("/dev/psaux","r");
  if (fpt == NULL)
  { printf("Cannot open input file \"%s\"\n", "/dev/psaux");
    exit(0);
  }
  while (1)
  { c = fread(buf,4,1,fpt);
    printf("Read %d bytes:  %d\n",c,buf[0]);
  }
  fclose(fpt);
  exit(0);
}
```

devmouse1.c

Opening a Device File

The bytes shown in `devmouse1.c` are not meant to be understood by us. They are meant to be handled by the O/S using a “device driver.”

devmouse1.c

```
...
fpt = fopen("/dev/psaux","r");
...
```

O/S Kernel

```
_fopen();
```

Mouse Device Driver

```
int mouse_open()
{
}

int mouse_read()
{
}
```

Opening a Device File

How does the O/S know which driver to open when we call `fopen()`?

`ls -all /dev/psaux`

Instead of a file size, we get a major and minor number associated with the file.

`devmouse1.c`

```
...
fpt = fopen("/dev/psaux","r");
...
```

O/S Kernel

```
_fopen(10,1);
```

Device Files

Where is the code for each device driver located?

When the kernel of the operating system is compiled, the driver code can be linked statically or dynamically.

Device Files

There is not always a one-to-one correspondence between hardware devices and files.

```
ls -all /dev/hda*
```

Notice that a single hard drive can have multiple partitions each associated with a different file.

Terminals

A terminal is a “virtual” device that corresponds to a keyboard-console connection.

```
ls -l /dev/pts
```

To print the name of the terminal connected to standard input use `tty`

Terminal Example

Once we know what terminals are open, we can communicate with them.

```
int main(int argc, char *argv[])
{ FILE *fpt;
  int Terminal;
  char strDev[100];

  if (argc != 2)
  { printf("Usage: devterm n, where n is term. number.\n");
    return -1;
  }
  else Terminal = atoi(argv[1]);

  sprintf(strDev, "/dev/pts/%d", Terminal);

  if ((fpt = fopen(strDev,"w")) == NULL)
  { printf("Unable to open %s\n", strDev);  exit(0);
  }
  fprintf(fpt,"Hello!  Scientia Est Potentia!!!\n");

  fclose(fpt);
}
```

devterm.c

devterm3.c

I/O Settings

Since managing terminal settings is so important, the O/S gives you a program called `stty` which allows the user to set tty terminal parameters.

```
stty erase 1
stty sane
```

```
echostate.c
```

Console Settings

Now consider the following program (compare terminal to xterm):

`ScrDimension.c`

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

int main(int argc, char *argv[])
{ struct winsize WinBuf;

  if (ioctl(0,TIOCGWINSZ,&WinBuf) != 1)
  { printf("Screen Size:\t%d rows x %d cols\n",
                       WinBuf.ws_row, WinBuf.ws_col);
    printf("\t\t%d pixels wide x %d pixels tall\n",
                       WinBuf.ws_xpixel, WinBuf.ws_ypixel);
  }
}
```