- Basis for UNIX device-independent I/O.
- Experiment with read and write.
- The code carefully handles errors and interruption by signals.

Device Terminology

peripheral device : A piece of H/W accessed by a computer system

Ex: disks, tapes, CD-ROMs, screens, keyboards, printers, mouse devices and network interfaces.

device drivers : Are the o.s system modules accessed through system calls by user programs, to perform the control and I/O to these devices

- It hides the details of device operation and
- protects the device from unauthorized use.

A single-user machine also needs device drivers.

Some operating systems provide *pseudodevice drivers* to simulate devices such as terminals.

EX: Pseudoterminals, simplify the handling of remote login to computer systems over a network or a modem line.

So it requires learn by system programmer a complex set of system calls.

Device Terminology :UNIX

- All devices are represented by files called *special files* located in the /dev directory
- Provide uniform naming and access to all files (such as disk file and device file)

five functions provided by UNIX —

1. read,
2. write
3. open,
4. close,
5. ioctl

A *regular file* is an ordinary data file on disk.

A *block special file* represents a device with characteristics similar to disk. The data transfer done in blocks or chunks.

A *character special file* represents a device with characteristics similar to a terminal. device appears as a stream of bytes and accessed in sequential order.

Reading and Writing

UNIX provides sequential access to files and other devices

read() - retrieve *nbytes* from the file or device represented by *fildes* into the user variable *buf*

Note : size of buffer is large enough to hold *n* bytes of data.

SYNOPSIS

```
#include <unistd.h>
ssize_t    read(int fildes, void *buf, size_t nbyte);
```

data type

ssize_t	: is a signed integer data type
size_t	: unsigned integer data type for the number of bytes to read

Reading and Writing

returns

success	: number of bytes
unsuccessful	: –1 and sets errno

The read operation commences at the current file offset, and the file offset is incremented by the number of bytes read.

If the current file offset is at or past the end of file, no bytes are read, then read() returns zero.

If *nbyte* is zero, read() may detect errors but if not check for error it returns zero.

Reading and Writing

Errors of read()

regular file

- returns 0 to indicate end-of-file
- may return fewer bytes than requested, for example, it reached end-of-file before completely satisfying the request.

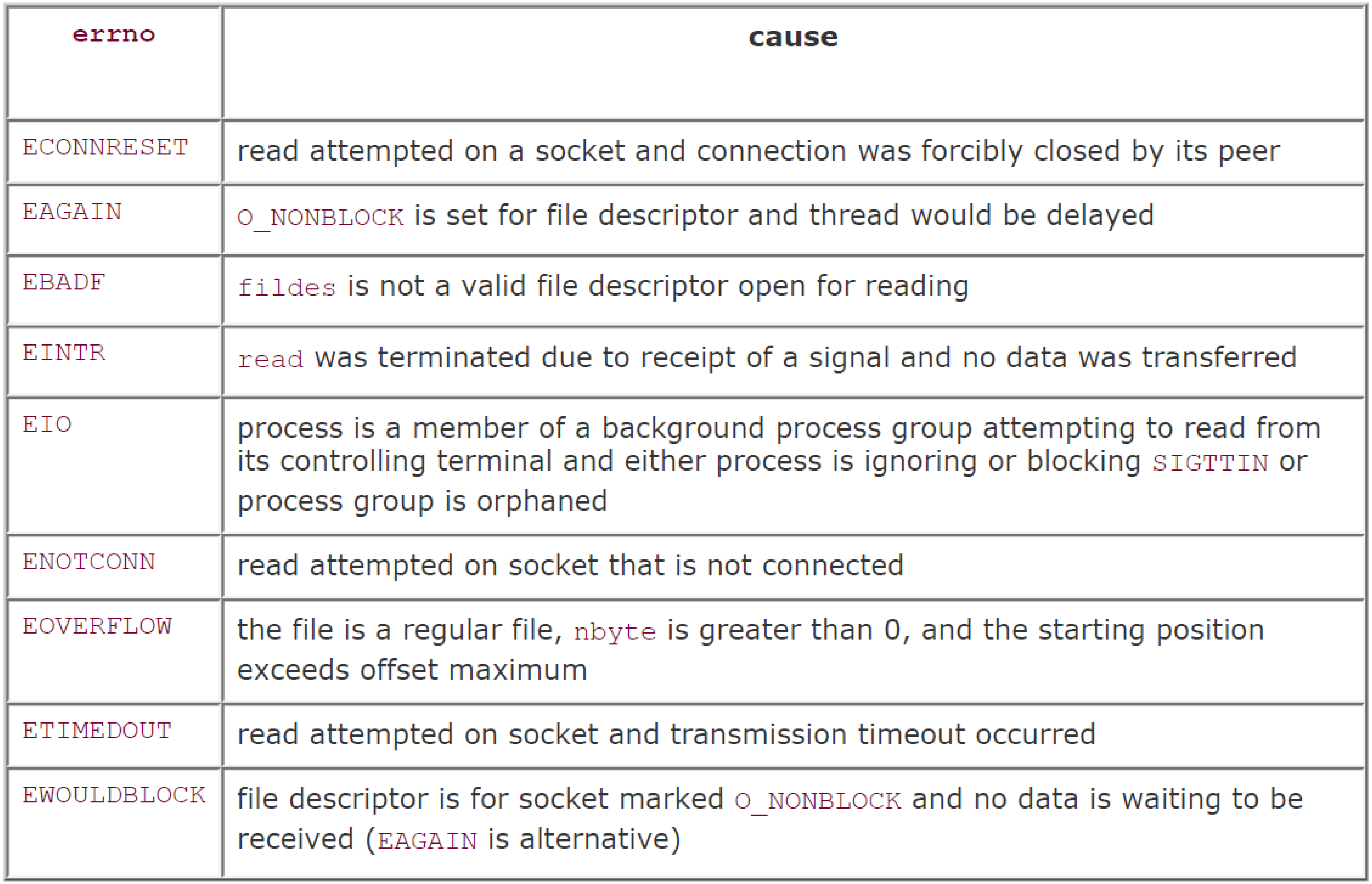
special files corresponding to devices

- return 0 depends on the implementation and the particular device.

reading from a terminal,

- returns 0, when the user enters an end-of-file character (Ctrl-D)

Errors of read()



errno	cause
`ECONNRESET`	read attempted on a socket and connection was forcibly closed by its peer
`EAGAIN`	`O_NONBLOCK` is set for file descriptor and thread would be delayed
`EBADF`	`fildes` is not a valid file descriptor open for reading
`EINTR`	`read` was terminated due to receipt of a signal and no data was transferred
`EIO`	process is a member of a background process group attempting to read from its controlling terminal and either process is ignoring or blocking `SIGTTIN` or process group is orphaned
`ENOTCONN`	read attempted on socket that is not connected
`EOVERFLOW`	the file is a regular file, `nbyte` is greater than 0, and the starting position exceeds offset maximum
`ETIMEDOUT`	read attempted on socket and transmission timeout occurred
`EWOULDBLOCK`	file descriptor is for socket marked `O_NONBLOCK` and no data is waiting to be received (`EAGAIN` is alternative)

Ex 4.1- reads at most 10 bytes into buf from standard input

```
#include<unistd.h>
#include<stdio.h>
int main(void)
{
char buf[10];
ssize_t bytesread;
bytesread = read(STDIN_FILENO, buf, 10);
printf("Number of bytes read = %ld",bytesread);
}
```

O/P

```
1 2 34569m
 Number of bytes read = 10
```

Note: error checking not done.

Reading and Writing

Exercise 4.2 : What happens when the following code executes?

```
#include<unistd.h>
#include<stdio.h>
int main(void)
{
char *buf;
ssize_t bytesread;
bytesread = read(STDIN_FILENO, buf, 10);
printf(“Number of bytes read = %ld",bytesread);
}
```

O/P

```
123456789asd

Number of bytes read = 10
MN$ sd
```

- does not allocate space for buf, result of read is unpredictable
- it will generate a memory access violation

By executing a program from the shell, the program starts with three open streams associated with file descriptors

1. STDIN_FILENO, //standard input ----------- 0
2. STDOUT_FILENO //standard output ----------- 1
3. STDERR_FILENO. ----------- 2

1,2 Usually correspond to keyboard input and screen output.

Programs should use STDERR_FILENO, the standard error device, for error messages and should never close it.

Note: always use symbolic names rather than these numeric values

Program 4.1- Write a user defined *readline()* function that reads bytes at a time, into a buffer of fixed size until a newline character ('\n') or an error occurs.

It handles **end-of-file, limited buffer size and interruption by a signal**

It returns the number of bytes read or –1 if an error occurs.

```
A return value of  0 ------- end-of-file before any characters were read.
                  >0 ------- number of bytes read.
                  –1 ------- set an errno
```

A return value of –1 indicates that errno has been set and one of the following errors occurred.

- An error occurred on read.
- At least one byte was read and an end-of-file occurred before a newline was read.
- *(nbytes-1)* bytes were read and no newline was found.

```
#include <errno.h>
#include <unistd.h>
int readline(int fd, char *buf, int nbytes) {
int numread = 0;
int returnval;
while (numread < nbytes - 1) {
            returnval = read(fd, buf + numread, 1);
            if ((returnval == -1) && (errno == EINTR))
                        continue;
            if ( (returnval == 0) && (numread == 0) )
                        return 0;
            if (returnval == 0)
                        break;
            if (returnval == -1)
                        return -1;
            numread++;
            if (buf[numread-1] == '\n') {
                        buf[numread] = '\0';
            return numread;
            }
}
errno = EINVAL;
return -1;
}
```

```
#include<stdio.h>
int main(void)
{
char buf[10];
ssize_t n;
n = readline(STDIN_FILENO, buf, 10);
printf(" Number of bytes read = %ld\t
%s\n",n,buf);
}
```

Exercise 4.4 - Under what circumstances, readline() function return a buffer with no newline character?

It happens only when the return value is

0 or –1

0 ---- nothing was read.
–1 ----- some type of error

In either case, the buffer may not contain a string.

Writing

write() - attempts to output ***n*** bytes from the user buffer ***buf*** to the file represented by file descriptor ***fildes***.

SYNOPSIS

```
#include <unistd.h>
ssize_t    write(int fildes, const void *buf, size_t nbyte);
```

returns

success	---	number of bytes
unsuccessful	---	–1 and sets errno

data type

ssize_t	: is a signed integer data type

Writing

```
int main(void)
{
char buf[20]="I am a Indian";
ssize_t n;
n = write(STDOUT_FILENO, buf, 10);
}
```

o/p:

```
I am an In
```

errno	cause
ECONNRESET	write attempted on a socket that is not connected
EAGAIN	O_NONBLOCK is set for file descriptor and thread would be delayed
EBADF	fildes is not a valid file descriptor open for writing
EFBIG	attempt to write a file that exceeds implementation-defined maximum; file is a regular file, nbyte is greater than 0, and starting position exceeds offset maximum
EINTR	write was terminated due to receipt of a signal and no data was transferred
EIO	process is a member of a background process group attempting to write to controlling terminal, TOSTOP is set, process is neither blocking nor ignoring SIGTTOU and process group is orphaned
ENOSPC	no free space remaining on device containing the file
EPIPE	attempt to write to a pipe or FIFO not open for reading or that has only one end open (thread may also get SIGPIPE), or write attempted on socket shut down for writing or not connected (if not connected, also generates SIGPIPE signal)
EWOULDBLOCK	file descriptor is for socket marked O_NONBLOCK and write would block (EAGAIN is alternative)

Exercise 4.5 - Find wrong with the following code segment?

```
#define BLKSIZE 1024
char buf[BLKSIZE];
read(STDIN_FILENO, buf, BLKSIZE);
write(STDOUT_FILENO, buf, BLKSIZE);
```

The write function assumes that ---- the read has filled buf with BLKSIZE bytes.

However,

1. read may fail or
2. may not read the full BLKSIZE bytes.

In these two cases, write outputs garbage.

Exercise 4.6 - What can go wrong with the following code segment to read from standard input and write to standard output??

```
#define BLKSIZE 1024
char buf[BLKSIZE];
ssize_t bytesread;
bytesread = read(STDIN_FILENO, buf, BLKSIZE);
if (bytesread > 0)
            write(STDOUT_FILENO, buf, bytesread);
```

either read or write can be interrupted by a signal.

In this case, returns a –1 with errno set to EINTR

Example 4.7 –Program 4.2 -

Write a function that, copies bytes from a file represented by *fromfd* to the file *tofd.*

returns the number of bytes read and does not indicate whether or not an error occurred.

- It copies bytes from the file *fromfd* to the file *tofd*.
- It restarts read and write if either is interrupted by a signal.
- The write statement specifies the buffer by a pointer, bp, rather than by a fixed address such as buf.
- If the previous write operation did not output all of buf, the next write operation must start from the end of the previous output.
- The copyfile function returns the number of bytes read and does not indicate whether or not an error occurred.

```
#include <errno.h>
#include <unistd.h>
#define BLKSIZE 10
int copyfile(int fromfd, int tofd) {
char *bp;
char buf[BLKSIZE];
int bytesread, byteswritten;
int totalbytes = 0;
for ( ; ; ) {/* handle interruption by signal */
   while (((bytesread = read(fromfd, buf, BLKSIZE)) == -1) && (errno == EINTR)) ;
   if (bytesread <= 0) /* real error or end-of-file on fromfd */
               break;
   bp = buf;
   while (bytesread > 0) {/* handle interruption by signal */
      while(((byteswritten = write(tofd, bp, bytesread)) == -1 ) && (errno == EINTR)) ;
      if (byteswritten <= 0) /* real error on tofd */
        break;
      totalbytes += byteswritten;
      bytesread -= byteswritten;
      bp += byteswritten;
   }
   if (byteswritten == -1) /* real error on tofd */
     break;
 }
 return totalbytes;
}
```

simplecopy.c

```
#include <stdio.h>
#include <unistd.h>
int copyfile(int fromfd, int tofd);
int main (void) {
int numbytes;
numbytes = copyfile(STDIN_FILENO, STDOUT_FILENO);
fprintf(stderr, "Number of bytes copied: %d\n", numbytes);
return 0;
}
```

read one line at a time, so I/O is likely be entered and echoed on line boundaries.

The I/O continues until end-of-file character (Ctrl-D)
 or
interrupt the program by entering the interrupt character (Ctrl-C).

Exercise 4.9 - How to use main() of the Example 4.7 to copy the file *myin.dat* to *myout.dat*

Use redirection

Hint : $simplecopy < myin.dat > myout.dat

User defined r_read() and r_write() function

restarts itself if interrupted by a signal

```
ssize_t r_read(int fd, void *buf, size_t size) {
ssize_t retval;
while (retval = read(fd, buf, size), retval == -1 && errno == EINTR) ;
     return retval;
}
```

Program 4.3, 4.4 –
User defined r_read() and r_write() function

restarts itself if interrupted by a signal

```
ssize_t r_write(int fd, void *buf, size_t size) {
char *bufp;
size_t bytestowrite;
ssize_t byteswritten;
size_t totalbytes;
for (bufp = buf, bytestowrite = size, totalbytes = 0;bytestowrite > 0;bufp += byteswritten, bytestowrite -= byteswritten) {
        byteswritten = write(fd, bufp, bytestowrite);
        if ((byteswritten) == -1 && (errno != EINTR))
           return -1;
        if (byteswritten == -1)
            byteswritten = 0;
totalbytes += byteswritten;
 }
return totalbytes;
}
```

Program 4.5 -A function that reads from one file descriptor and writes all the bytes read to another file descriptor using user defined r_read(), r_write().

```
#include <limits.h>
#include "restart.h"
#define BLKSIZE PIPE_BUF
int readwrite(int fromfd, int tofd) {
char buf[BLKSIZE];
int bytesread;
if ((bytesread = r_read(fromfd, buf, BLKSIZE)) == -1)
      return -1;
if (bytesread == 0)
     return 0;
if (r_write(tofd, buf, bytesread) == -1)
     return -1;
return bytesread;
}
```

Program 4.6- Modify the copy file function to copy file using user defined r_read and r_write function.

```
#define BLKSIZE 1024
int copyfile(int fromfd, int tofd) {
char buf[BLKSIZE];
int bytesread, byteswritten;
int totalbytes = 0;
for ( ; ; ) {
   if ((bytesread = r_read(fromfd, buf, BLKSIZE)) <= 0)
       break;
   if ((byteswritten = r_write(tofd, buf, bytesread)) == -1)
       break;
   totalbytes += byteswritten;
}
return totalbytes;
}
```

Program 4.7 - Write function to reads a specific number of bytes from one descriptor to another, and it continue reading until requested number of bytes is read or error occur.

It returns :

- 0 : if an end-of-file occurs before any bytes are read i.e. first call to read
- size : on successful i.e requested number of bytes was successfully read
- –1 : reaches at end of file after some, but not all, of the needed bytes have been read and sets errno to EINVAL.

Example 4.10:

Use the above function to read a pair of integers from an open file descriptor

Program 4.8- Copy/Write fixed number of bytes from one descriptor to another file descriptor using program 4.7

Dr. Mamata Nayak