

# System Programming

Prof. Chuan-Ju Wang  
Dept. of Computer Science  
University of Taipei

# Unix History and Some Other Unix Basics

# Unix History

- Originally developed in 1969 at Bell Labs by Ken Thompson and Dennis Ritchie.
- ACM Turing award winners for the design of UNIX in 1983.
- C programming language inventor: Dennis Ritchie

# Unix History



Ken (sitting) & Dennis (standing)

# Unix History

- 1973: Rewritten in C.
  - This made it portable and changed the history of OS.
- 1974: Thompson, Joy, Haley and students at Berkeley develop the Berkeley Software Distribution (BSD) of UNIX
- Two main directions emerge: **BSD** and what was to become “**System V**”

Unix history:

[http://www.unix.org/what\\_is\\_unix/history\\_timeline.html](http://www.unix.org/what_is_unix/history_timeline.html)

[http://content.edu.tw/senior/computer/ks\\_ks/comsense/unix.htm](http://content.edu.tw/senior/computer/ks_ks/comsense/unix.htm)

<http://www.levenez.counix/>

# Unix History

- 1984 4.2BSD released (TCP/IP)
- 1986 4.3BSD released (NFS)
- 1991 Linus Torvalds starts working on the Linux kernel
- 1993 Following the settlement of USL vs. BSDi: NetBSD, then FreeBSD are created
- 1994 Single UNIX Specification (<http://is.gd/7fuSxO>) introduced
- 1995 4.4BSD-Lite Release 2 (last CSRG release); OpenBSD forked off NetBSD
- 2000 Darwin created (derived from NeXT, FreeBSD, NetBSD) ...
- ...
- Some of today's main Unix versions:
  - mostly BSD: \*BSD, Linux, Mac OS X
  - mostly SysV: Solaris, HP-UX, IRIX

# Some UNIX versions

- More UNIX (some generic, some trademark, some just unix-like):

1BSD	2BSD	3BSD	4BSD	4.4BSD Lite 1
4.4BSD Lite 2	386 BSD	A/UX	Acorn RISC iX	AIX
AIX PS/2	AIX/370	AIX/6000	AIX/ESA	AIX/RT
AMiX	AOS Lite	AOS Reno	ArchBSD	ASV
Atari Unix	BOS	BRL Unix	BSD Net/1	BSD Net/2
BSD/386	BSD/OS	CB Unix	Chorus	Chorus/MiX
Coherent	CTIX	Darwin	Debian GNU/Hurd	DEC OSF/1 ACP
Digital Unix	DragonFly BSD	Dynix	Dynix/ptx	ekkoBSD
FreeBSD	GNU	GNU-Darwin	HPBSD	HP-UX
HP-UX BLS	IBM AOS	IBM iX/370	Interactive 386/ix	Interactive IS
IRIX	Linux	Lites	LSX	Mac OS X
Mac OS X Server	Mach	MERT	MicroBSD	Mini Unix
Minix	Minix-VMD	MIPS OS	MirBSD	Mk Linux
Monterey	more/BSD	mt Xinu	MVS/ESA OpenEdition	NetBSD
NeXTSTEP	NonStop-UX	Open Desktop	Open UNIX	OpenBSD
OpenServer	OPENSTEP	OS/390 OpenEdition	OS/390 Unix	OSF/1
PC/IX	Plan 9	PWB	PWB/UNIX	QNX
QNX RTOS	QNX/Neutrino	QUNIX	ReliantUnix	Rhapsody
RISC iX	RT	SCO UNIX	SCO UnixWare	SCO Xenix
SCO Xenix System V/386	Security-Enhanced Linux	Sinix	Sinix ReliantUnix	Solaris
SPIX	SunOS	Tru64 Unix	Trusted IRIX/B	Trusted Solaris
Trusted Xenix	TS	UCLA Locus	UCLA Secure Unix	Ultrix
Ultrix 32M	Ultrix-11	Unicos	Unicos/mk	Unicox-max
UNICS	UNIX 32V	UNIX Interactive	UNIX System III	UNIX System IV
UNIX System V	UNIX System V Release 2	UNIX System V Release 3	UNIX System V Release 4	UNIX System V/286
UNIX System V/386	UNIX Time-Sharing System	UnixWare	UNSW	USG
Venix	Wollogong	Xenix OS	Xinu	xMach

# Standards

- ANSI C
  - American National Standards Institute
  - ISO/IEC 9899:1990
    - International Organization for Standardization (ISO)
    - Syntax/Semantics of C, a standard library
- Purpose
  - Provide portability of conforming C programs to a wide variety of OS's.



**Figure 2.1. Headers defined by the ISO C standard**

Header	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9	Description
<assert.h>	•	•	•	•	verify program assertion
<complex.h>	•	•	•		complex arithmetic support
<ctype.h>	•	•	•	•	character types
<errno.h>	•	•	•	•	error codes ( <a href="#">Section 1.7</a> )
<fenv.h>		•	•		floating-point environment
<float.h>	•	•	•	•	floating-point constants
<inttypes.h>	•	•	•	•	integer type format conversion
<iso646.h>	•	•	•	•	alternate relational operator macros
<limits.h>	•	•	•	•	implementation constants ( <a href="#">Section 2.5</a> )
<locale.h>	•	•	•	•	locale categories
<math.h>	•	•	•	•	mathematical constants
<setjmp.h>	•	•	•	•	nonlocal goto ( <a href="#">Section 7.10</a> )
<signal.h>	•	•	•	•	signals ( <a href="#">Chapter 10</a> )
<stdarg.h>	•	•	•	•	variable argument lists
<stdbool.h>	•	•	•	•	boolean type and values
<stddef.h>	•	•	•	•	standard definitions
<stdint.h>	•	•	•		integer types
<stdio.h>	•	•	•	•	standard I/O library ( <a href="#">Chapter 5</a> )
<stdlib.h>	•	•	•	•	utility functions
<string.h>	•	•	•	•	string operations
<tgmath.h>		•			type-generic math macros
<time.h>	•	•	•	•	time and date ( <a href="#">Section 6.10</a> )
<wchar.h>	•	•	•	•	extended multibyte and wide character support
<wctype.h>	•	•	•	•	wide character classification and mapping support

# Standards

- POSIX.1 (Portable Operating System **Interface**)  
developed by IEEE
  - Purpose
    - Define the application programming interface for software compatible with variants of OS's
  - Not restricted for Unix-like systems and **no distinction for system calls and library functions**
  - Originally IEEE Standard 1003.1-1988
  - New: the inclusion of **symbolic links**

**Figure 2.2. Required headers defined by the POSIX standard**

Header	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9	Description
<dirent.h>	•	•	•	•	directory entries ( <a href="#">Section 4.21</a> )
<fcntl.h>	•	•	•	•	file control ( <a href="#">Section 3.14</a> )
<fnmatch.h>	•	•	•	•	filename-matching types
<glob.h>	•	•	•	•	pathname pattern-matching types
<grp.h>	•	•	•	•	group file ( <a href="#">Section 6.4</a> )
<netdb.h>	•	•	•	•	network database operations
<pwd.h>	•	•	•	•	password file ( <a href="#">Section 6.2</a> )
<regex.h>	•	•	•	•	regular expressions
<tar.h>	•	•	•	•	tar archive values
<termios.h>	•	•	•	•	terminal I/O ( <a href="#">Chapter 18</a> )
<unistd.h>	•	•	•	•	symbolic constants
<utime.h>	•	•	•	•	file times ( <a href="#">Section 4.19</a> )
<wordexp.h>	•	•		•	word-expansion types
<arpa/inet.h>	•	•	•	•	Internet definitions ( <a href="#">Chapter 16</a> )
<net/if.h>	•	•	•	•	socket local interfaces ( <a href="#">Chapter 16</a> )
<netinet/in.h>	•	•	•	•	Internet address family ( <a href="#">Section 16.3</a> )
<netinet/tcp.h>	•	•	•	•	Transmission Control Protocol definitions
<sys/mman.h>	•	•	•	•	memory management

Because POSIX.1 includes the ISO C standard library functions, it also requires the headers listed in Figure 2.1.

# Limits

- The implementations define many **magic numbers** and constants.
- Many of these have been **hard coded into programs** or were determined using **ad hoc techniques**.
- Two types of limits are needed:
  - Compile-time limits (e.g., what's the largest value of a short integer?)
    - Can be defined in headers
  - Runtime limits (e.g., how many characters in a filename?)
    - Require the process to call a function to obtain the value of the limit

# Limits

- All the limits defined by **ISO C** are **compile-time limits**. (see `/usr/include/limits.h`)

**Figure 2.6. Sizes or integral values from `<limits.h>`**

Name	Description	Minimum acceptable value	Typical value
CHAR_BIT	bits in a <code>char</code>	8	8
CHAR_MAX	max value of <code>char</code>	(see later)	127
CHAR_MIN	min value of <code>char</code>	(see later)	128
SCHAR_MAX	max value of signed <code>char</code>	127	127
SCHAR_MIN	min value of signed <code>char</code>	127	128
UCHAR_MAX	max value of unsigned <code>char</code>	255	255
INT_MAX	max value of <code>int</code>	32,767	
INT_MIN	min value of <code>int</code>	32,767	
UINT_MAX	max value of unsigned <code>int</code>	65,535	
SHRT_MIN	min value of <code>short</code>	32,767	
SHRT_MAX	max value of <code>short</code>	32,767	
USHRT_MAX	max value of unsigned <code>short</code>	65,535	
LONG_MAX	max value of <code>long</code>	2,147,483,647	
LONG_MIN	min value of <code>long</code>	2,147,483,647	

limits.h = (/usr/include) - VIM

limits.h

```
50
51 /* These assume 8-bit 'char's, 16-bit 'short int's,
52    and 32-bit 'int's and 'long int's. */
53
54 /* Number of bits in a 'char'. */
55 # define CHAR_BIT 8
56
57 /* Minimum and maximum values a 'signed char' can hold. */
58 # define SCHAR_MIN (-128)
59 # define SCHAR_MAX 127
60
61 /* Maximum value an 'unsigned char' can hold. (Minimum is 0.) */
62 # define UCHAR_MAX 255
```



# Primitive System Data Types

**Figure 2.20. Some common primitive system data types**

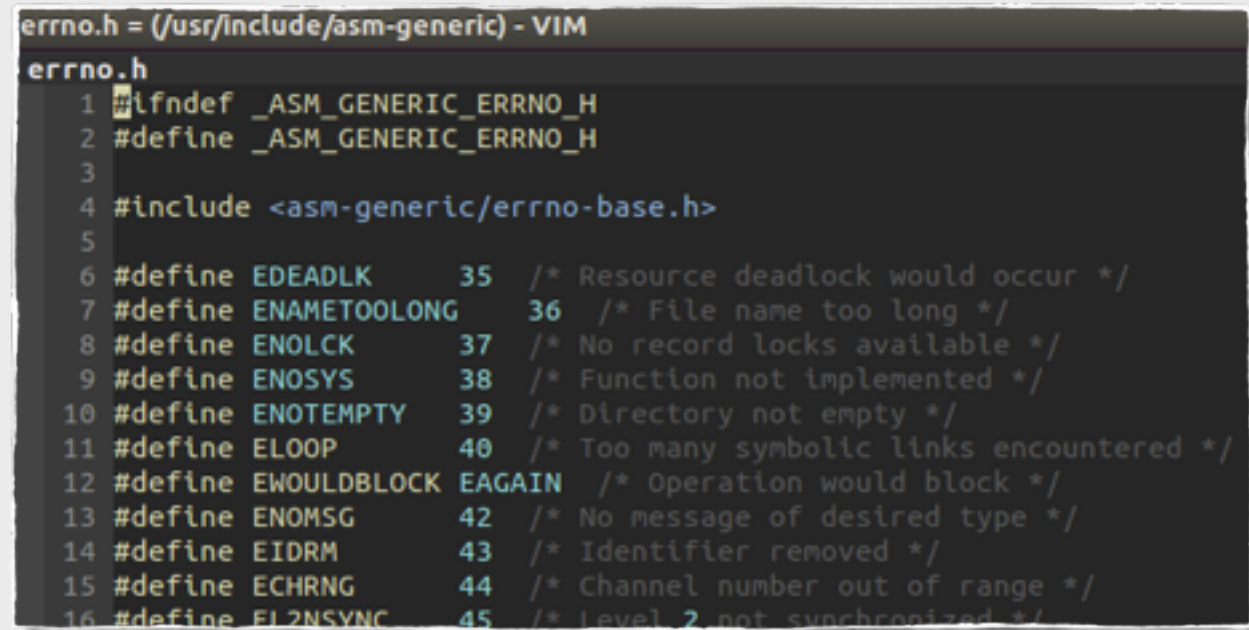
Type	Description
<code>caddr_t</code>	core address ( <a href="#">Section 14.9</a> )
<code>clock_t</code>	counter of clock ticks (process time) ( <a href="#">Section 1.10</a> )
<code>comp_t</code>	compressed clock ticks ( <a href="#">Section 8.14</a> )
<code>dev_t</code>	device numbers (major and minor) ( <a href="#">Section 4.23</a> )
<code>fd_set</code>	file descriptor sets ( <a href="#">Section 14.5.1</a> )
<code>fpos_t</code>	file position ( <a href="#">Section 5.10</a> )
<code>gid_t</code>	numeric group IDs
<code>ino_t</code>	i-node numbers ( <a href="#">Section 4.14</a> )
<code>mode_t</code>	file type, file creation mode ( <a href="#">Section 4.5</a> )
<code>nlink_t</code>	link counts for directory entries ( <a href="#">Section 4.14</a> )
<code>off_t</code>	file sizes and offsets (signed) ( <code>lseek</code> , <a href="#">Section 3.6</a> )
<code>pid_t</code>	process IDs and process group IDs (signed) ( <a href="#">Sections 8.2</a> and <a href="#">9.4</a> )
<code>ptrdiff_t</code>	result of subtracting two pointers (signed)

`/usr/include/i386-linux-gnu/sys/types.h`

`/usr/include/sys/types.h`

# Error Handling

- Error Handling
  - Meaningful return values
  - A value is stored in `errno` by certain library functions when they detect errors.
  - Defined in `/usr/include/errno.h`
  - `<errno.h>*`
    - `/usr/include/asm-generic/errno-base.h`
    - `/usr/include/asm-generic/errno.h`



```
errno.h = (/usr/include/asm-generic) - VIM
errno.h
1 #ifndef _ASM_GENERIC_ERRNO_H
2 #define _ASM_GENERIC_ERRNO_H
3
4 #include <asm-generic/errno-base.h>
5
6 #define EDEADLK      35 /* Resource deadlock would occur */
7 #define ENAMETOOLONG 36 /* File name too long */
8 #define ENOLCK       37 /* No record locks available */
9 #define ENOSYS       38 /* Function not implemented */
10 #define ENOTEMPTY    39 /* Directory not empty */
11 #define ELOOP        40 /* Too many symbolic links encountered */
12 #define EWOULDBLOCK  EAGAIN /* Operation would block */
13 #define ENOMSG       42 /* No message of desired type */
14 #define EIDRM        43 /* Identifier removed */
15 #define ECHRNG       44 /* Channel number out of range */
16 #define EL2NSYNC     45 /* Level 2 not synchronized */
```

\* <http://www.barricane.com/c-error-codes-include-errno>

# Error Handling

- Look up constant error values via two functions:

```
#include <string.h>
```

```
char *strerror(int errnum);
```

This function maps *errnum*, which is typically the **errno** value, into an error message string and returns a pointer to the string.

Returns: pointer to message string

```
#include <stdio.h>
```

```
void perror(const char *msg);
```

The **perror** function produces an error message on the standard error, based on the current value of **errno**, and returns.



# Error Handling

```
#include "apue.h"
#include <errno.h>
```

```
int
main(int argc, char *argv[])
{
    fprintf(stderr, "EACCES: %s\n", strerror(EACCES));
    errno = ENOENT;
    perror(argv[0]);
    exit(0);
}
```

Find **EACCES** & **ENOENT**!

```
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e] cp fig1.8 ./test/fig1-8.c
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e] cd test
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] ls
fig1-8.c  Makefile
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] make fig1-8
gcc -DLINUX -ansi -I/home/jere/SystemProgramming/apue.2e/include -Wall -D_GNU_SOURCE -lapue -o fig1-8
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] ls
fig1-8  fig1-8.c  Makefile
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] ./fig1-8
EACCES: Permission denied
./fig1-8: No such file or directory
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] █
```

# Program Design

- "Consistency underlies all principles of quality."  
Frederick P. Brooks, Jr
- UNIX programs... \*
  - ...are simple
  - ...have a manual page
  - ...follow the element of least surprise
  - ...accept input from stdin
  - ...generate output to stdout
  - ...generate meaningful error messages to stderr

\* Unix philosophy: [https://secure.wikimedia.org/wikipedia/en/wiki/Unix\\_philosophy](https://secure.wikimedia.org/wikipedia/en/wiki/Unix_philosophy)

# Files and Directories

- The UNIX filesystem is a tree structure, with all partitions **mounted under the root (/)**.
- File names may consist of any character except / and NUL as pathnames are a sequence of zero or more filenames separated by /'s.
- Directories are special "files" that contain mappings between inodes and filenames, called directory entries.
- All processes have **a current working directory** from which all relative paths are specified.
- Recall that absolute paths begin with a slash, relative paths do not.

# Unix Time Values

- Calendar time: measured in seconds since the UNIX epoch (Jan 1, 00:00:00, 1970, GMT).
- Stored in a variable of type `time_t` (signed 32-bit integer)



<https://www.xkcd.com/376/>

Year 2038 problem: [https://secure.wikimedia.org/wikipedia/en/wiki/Year\\_2038\\_problem](https://secure.wikimedia.org/wikipedia/en/wiki/Year_2038_problem)

# Unix Time Values

- Process time: central processor resources used by a process.
  - Measured in clock ticks (`clock_t`).
  - Three values:
    - clock time
    - user CPU time
    - system CPU time

```
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] time grep _POSIX_SOURCE /usr/include/**/*.h >/dev/null  
real    0m0.546s  
user    0m0.012s  
sys     0m0.108s  
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] █
```

# Unix Time Values

- Real time
  - **Wall clock time**: time from start to finish of the call
- User time
  - The amount of CPU time spent **in user-mode code** (outside the kernel) within the process
  - A program loops through an array
- System time
  - The amount of CPU time spent **in the kernel**
  - A program executes a **system call** such as `exec` or `fork` within the process

# System Calls and Library Functions

- System calls are **entry points into kernel code** where their functions are implemented.
  - Documented in section **2** of the manual (e.g. `write(2)`).
- Library calls are **transfers to user code** which performs the desired functions.
  - Documented in section **3** of the manual (e.g. `printf(3)`).

# Processes

- Programs **executing in memory** are called processes.
- Programs are brought into memory via one of the six `exec ( 3 )` functions.
- Each process is identified by a guaranteed unique non-negative integer called the processes ID.
- New processes can only be created via the `fork ( 2 )` system call.
- Process control is performed mainly by the `fork ( 2 )`, `exec ( 3 )` and `waitpid ( 2 )` functions.



# Signals

- Signals notify a process that **a condition has occurred**. Signals may be
  - ignored
  - allowed to cause the default action
  - caught and control transferred to a user defined function

File I/O

# Introduction

- Most file I/O on a UNIX system can be performed using only five functions:
  - `open`, `read`, `write`, `lseek`, and `close`.
- The functions described later are often referred to as **unbuffered I/O**.

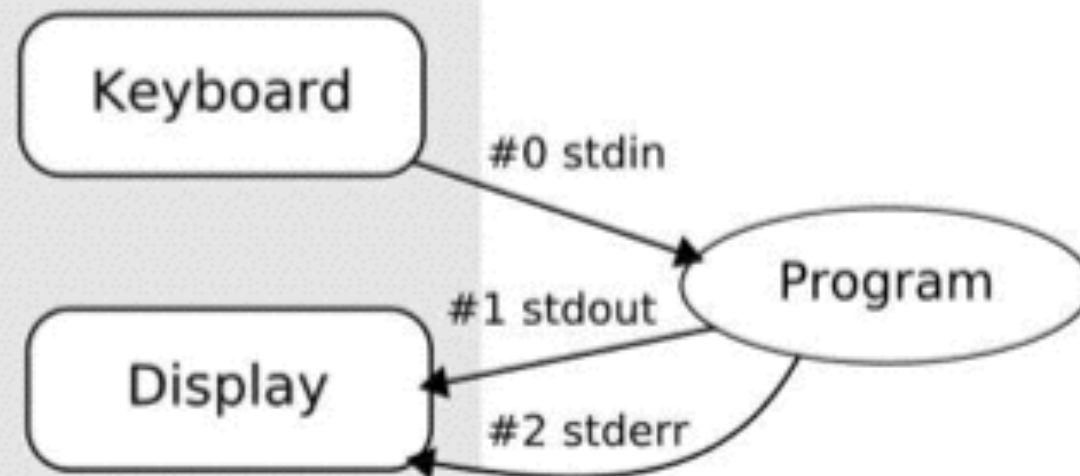
# File Descriptors

- A **file descriptor** (or file handle) is a small, non-negative **integer** which identify a file to kernel.
- Traditionally, stdin, stdout and stderr are 0, 1 and 2 respectively.
- Relying on “magic numbers” is Bad. Use STDIN\_FILENO, STDOUT\_FILENO and STDERR\_FILENO.
- Defined in `<unistd.h>`

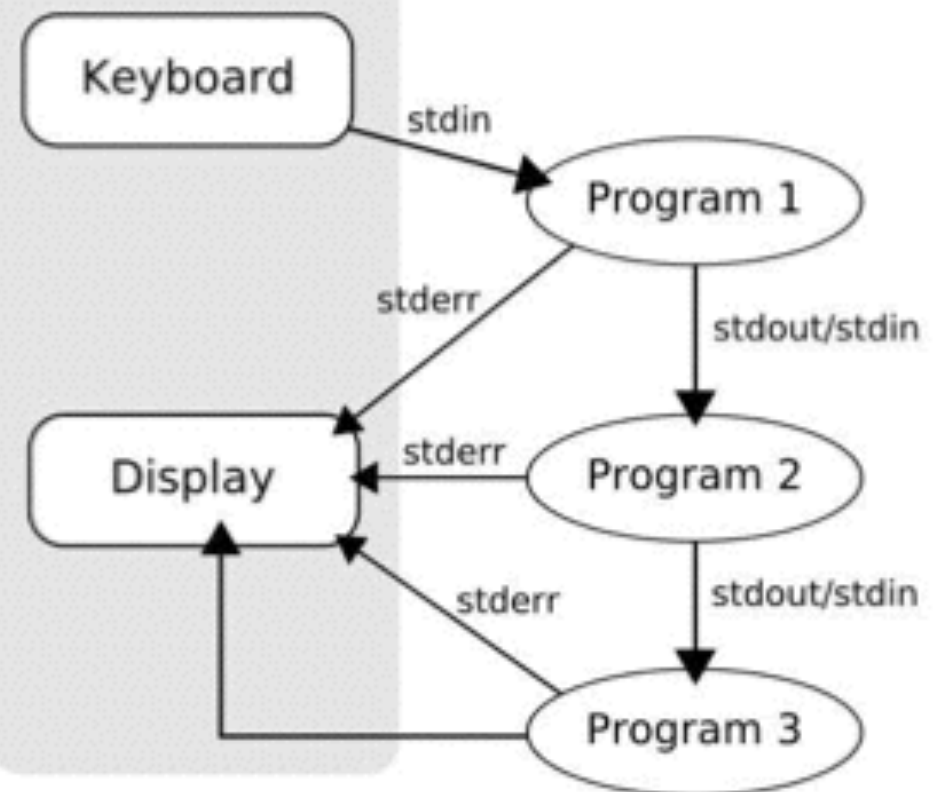
```
unistd.h = (/usr/include) - VIM
unistd.h
207 # include <bits/environments.h>
208 #endif
209
210 /* Standard file descriptors. */
211 #define STDIN_FILENO    0    /* Standard input. */
212 #define STDOUT_FILENO   1    /* Standard output. */
213 #define STDERR_FILENO   2    /* Standard error output. */
214
```

# File Descriptors

## Text terminal



## Text terminal



# Standard I/O

- Basic File I/O: almost all UNIX file I/O can be performed using these five functions:
  - `open(2); close(2); lseek(2); read(2); write(2)`
- Processes may want to share resources.
  - This requires us to look at:
    - **atomicity** of these operations
    - **file sharing**
    - manipulation of file descriptors

# open ( 2 ) Function

- A file is opened or created by calling the **open** function.

the ISO C way to specify that the number and types of the remaining arguments may vary.

```
#include <fcntl.h>

int open(const char *pathname, int oflag, ... /*
    mode_t mode */ );
```

The third argument is used only when a new file is being created (described in Chapter 4.5 and related to the file permission).

Returns: file descriptor if OK, -1 on error

The file descriptor returned by open is guaranteed to be **the lowest-numbered unused descriptor**.

- *oflag* must be one (and only one) of:
  - **O\_RDONLY** – Open for reading only
  - **O\_WRONLY** – Open for writing only
  - **O\_RDWR** – Open for reading and writing

```
#include <fcntl.h>
```

```
int open(const char *pathname, int oflag, ... /*  
→ mode_t mode */ );
```

Returns: file descriptor if OK, 1 on error

- and may be **OR**'d with any of these:
  - **O\_APPEND** – Append to end of file for each write
  - **O\_CREAT** – Create the file if it doesn't exist. Requires mode argument.
  - **O\_EXCL** – Generate error if **O\_CREAT** and file already exists. (atomic)
  - **O\_TRUNC** – If file exists and successfully open in **O\_WRONLY** or **O\_RDWR**, make length = 0
  - **O\_NOCTTY** – If pathname refers to a terminal device, do not allocate the device as a controlling terminal
  - **O\_NONBLOCK** – If pathname refers to a FIFO, block special, or char special, set nonblocking mode (open and I/O)
  - **O\_SYNC** – Each write waits for physical I/O to complete



# creat ( 2 ) Function

- A new file can also be created by calling the `creat` function.

```
#include <fcntl.h>
```

```
int creat(const char *pathname, mode_t mode);
```

Returns: file descriptor opened for write-only if OK,  
-1 on error

Note that this function is equivalent to

```
open (pathname, O_WRONLY | O_CREAT | O_TRUNC, mode);
```

One deficiency with `creat` is that the file is **opened only for writing**.

# close ( 2 ) Function

- An open file is closed by calling the `close` function.

```
#include <unistd.h>

int close(int filedes);
```

Returns: 0 if OK, -1 on error

- Closing a file descriptor releases any record locks on that file (more on that in future lectures).
- File descriptors not explicitly closed **are closed by the kernel when the process terminates.**

# lseek ( 2 ) Function

- Every open file has an associated **current file offset**.
- A non-negative integer



```
#include <unistd.h>
```

```
off_t lseek(int filedes, off_t offset, int whence);
```

Returns: new file offset if OK, -1 on error

# lseek ( 2 ) Function

```
#include <unistd.h>
```

```
off_t lseek(int filedes, off_t offset, int whence);
```

Returns: new file offset if OK, -1 on error

- The value of *whence* determines how offset is used:
  - `SEEK_SET` bytes from the beginning of the file
  - `SEEK_CUR` bytes from the current file position
  - `SEEK_END` bytes from the end of the file
- `lseek` only records the current file offset within the kernel, it does not cause any I/O to take place.
  - This offset is then used by the next read or write operation.

# lseek ( 2 ) Function

- “Weird” things you can do using lseek ( 2 ):
  - seek to a negative offset
    - The /dev/kmem device on FreeBSD for the Intel x86 processor supports negative offsets.
  - seek 0 bytes from the current position

```
off_t    currpos;  
currpos = lseek(fd, 0, SEEK_CUR);
```

- Determine the current offset
  - Determine if a file is capable of seeking
- seek past the end of the file
  - Creating a hole in a file

# lseek ( 2 ) Function

**Figure 3.1. Test whether standard input is capable of seeking**

```
#include "apue.h"

int
main(void)
{
    if (lseek(STDIN_FILENO, 0, SEEK_CUR) == -1)
        printf("cannot seek\n");
    else
        printf("seek OK\n");
    exit(0);
}
```

```
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] make fig3-1
gcc -DLINUX -ansi -I/home/jere/SystemProgramming/apue.2e/include -Wall -D_GNU_SOURCE
-lapue -o fig3-1
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] ./fig3-1 < /etc/motd
seek OK
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] cat < /etc/motd | ./fig3-1
cannot seek
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] █
```

why?

Check the man page



**Figure 3.2. Create a file with a hole in it**

```
#include "apue.h"
#include <fcntl.h>

char    buf1[] = "abcdefghij";
char    buf2[] = "ABCDEFGHIJ";

int
main(void)
{
    int    fd;

    if ((fd = creat("file.hole", FILE_MODE)) < 0)
        err_sys("creat error");

    if (write(fd, buf1, 10) != 10)
        err_sys("buf1 write error");
    /* offset now = 10 */

    if (lseek(fd, 16384, SEEK_SET) == -1)
        err_sys("lseek error");
    /* offset now = 16384 */

    if (write(fd, buf2, 10) != 10)
        err_sys("buf2 write error");
    /* offset now = 16394 */

    exit(0);
}
```

- Use the `od ( 1 )` command to look at the contents of the file.
- The `-c` flag tells it to **print the contents as characters**.
- The unwritten bytes in the middle are read back as **zero**.
- The seven-digit number at the beginning of each line is the byte offset **in octal**.

```
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e] cp fig3.2 ./test/fig3-2.c
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e] cd test
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] ls
fig1-8  fig1-8.c  fig3-1  fig3-1.c  fig3-2.c  Makefile
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] make fig3-2
gcc -DLINUX -ansi -I/home/jere/SystemProgramming/apue.2e/include -Wall -D_GN
-lapue -o fig3-2
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] ls
fig1-8  fig1-8.c  fig3-1  fig3-1.c  fig3-2  fig3-2.c  Makefile
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] ./fig3-2
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] ls
fig1-8  fig1-8.c  fig3-1  fig3-1.c  fig3-2  fig3-2.c  file.hole  Makefile
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] ls -l file.hole
-rw-r--r-- 1 jere jere 16394 2012-01-26 18:09 file.hole
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] od -c file.hole
0000000  a  b  c  d  e  f  g  h  i  j  \0  \0  \0  \0  \0  \0
0000020  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
*
0040000  A  B  C  D  E  F  G  H  I  J
0040012
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test]
```

# read ( 2 ) Function

- Data is read from an open file with the `read` function.

Generic pointer: A pointer to `void` can store an address to any data type.

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

Returns: number of bytes read, 0 if end of file, ~~-1~~ on error

`read` begins reading at the current offset, and increments the offset by the number of bytes actually read.



# read ( 2 ) Function

- There can be several cases where read returns **less than** the number of bytes requested:
  - 1) **EOF reached** before requested number of bytes have been read
    - If 30 bytes remain until the end of file and we try to read 100 bytes, read returns 30. The next time we call read, it will return 0 (end of file).
  - 2) Reading **from a terminal device**, one "line" read at a time
  - 3) Reading **from a network**, buffering can cause delays in arrival of data
  - 4) **Record-oriented devices** (magtape) may return data one record at a time

# write(2) Function

- Data is written to an open file with the `write` function.

```
#include <unistd.h>
```

```
ssize_t write(int filedes, const void *buf, size_t  
→ nbytes);
```

Returns: number of bytes written if OK, -1 on error

- `write` returns *nbytes* or an error has occurred (disk full, file size limit exceeded).
- For regular files, `write` begins writing at the current offset (unless `O_APPEND` has been specified, in which case the offset is first set to the end of the file.)
- After the write, the offset is adjusted by the number of bytes actually written.

# I/O Efficiency

- Assumes that `stdin` and `stdout` have been set up appropriately
- Relies on the fact that when a process terminates, the kernel will close all open files
- Works for “text” and “binary” files since there is no such distinction in the UNIX kernel
- How do we know the optimal `BUFSIZE`?

**Figure 3.4. Copy standard input to standard output**

```
#include "apue.h"

#define BUFSIZE 4096

int
main(void)
{
    int    n;
    char   buf[BUFSIZE];

    while ((n = read(STDIN_FILENO, buf, BUFSIZE)) > 0)
        if (write(STDOUT_FILENO, buf, n) != n)
            err_sys("write error");

    if (n < 0)
        err_sys("read error");

    exit(0);
}
```

BUFFSIZE	User CPU (seconds)	System CPU (seconds)	Clock time (seconds)	Number of loops
1	20.03	117.50	138.73	516,581,760
2	9.69	58.76	68.60	258,290,880
4	4.60	36.47	41.27	129,145,440
8	2.47	15.44	18.38	64,572,720
16	1.07	7.93	9.38	32,286,360
32	0.56	4.51	8.82	16,143,180
64	0.34	2.72	8.66	8,071,590
128	0.34	1.84	8.69	4,035,795
256	0.15	1.30	8.69	2,017,898
512	0.09	0.95	8.63	1,008,949
1,024	0.02	0.78	8.58	504,475
2,048	0.04	0.66	8.68	252,238
4,096	0.03	0.58	8.62	126,119
8,192	0.00	0.54	8.52	63,060
16,384	0.01	0.56	8.69	31,530
32,768	0.00	0.56	8.51	15,765
65,536	0.01	0.56	9.12	7,883
131,072	0.00	0.58	9.08	3,942
262,144	0.00	0.60	8.70	1,971
524,288	0.01	0.58	8.58	986

```
mycat.c
#include "apue.h"
//#define...BUFFSIZE...4096
#ifdef BUFFSIZE
#define BUFFSIZE...4096
#endif
```

Figure 3.6 Timing results for reading with different buffer sizes on Linux

```
mycat-test.sh Figure 3.5 in edition 3
for n in seq 10; do
    dd if=/dev/urandom of=file-$n count=20480
done
i=1
for n in 1 512 1024 2048 4096 8192 16384 32768 65536 131072; do
    gcc -DLINUX -DBUFFSIZE=$n -ansi \
        -I/home/jere/SystemProgramming/apue.3e/include -Wall -D_GNU_SOURCE \
        -L/home/jere/SystemProgramming/apue.3e/lib mycast.c -lapue -o \
        mycat
    time ./mycat < file-$i > file-$i.copy
    i=$((i + 1))
done
```