산업과 예술의 만남
홍익대학교

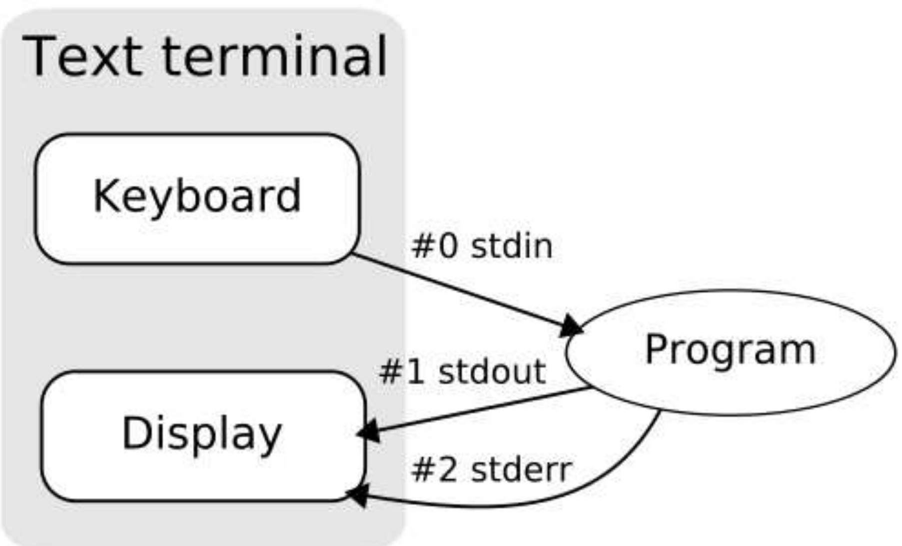
File I/O, File Sharing

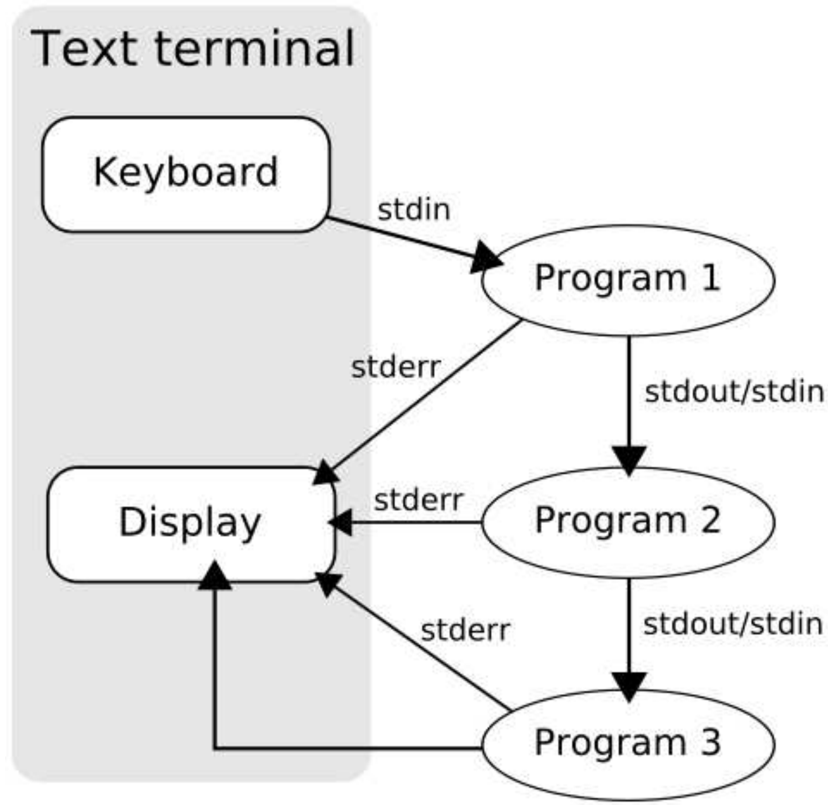
Hongik University
Eunsung Jung

Disclaimer: The slides are borrowed from many sources!

File Descriptors

- A file descriptor (or file handle) is a small, non-negative integer which identifies a file to the kernel.
- Traditionally, stdin, stdout and stderr are 0, 1 and 2 respectively.





File Descriptors

- Relying on “magic numbers” is BadTM. Use STDIN FILENO, STDOUT FILENO and STDERR FILENO.

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

creat(2)

```
#include <fcntl.h>

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

Returns: file descriptor if OK, -1 on error

- This interface is made obsolete by open(2).
- What does 2 mean?
  - man 2 system calls

Section	Description
1	General commands
2	System calls
3	Library functions, covering in particular the C standard library
4	Special files (usually devices, those found in /dev) and drivers
5	File formats and conventions
6	Games and screensavers
7	Miscellanea
8	System administration commands and daemons

open(2)

```
#include <fcntl.h>

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

Returns: file descriptor if OK, -1 on error

- 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
- 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

open(2) variants

```
#include <fcntl.h>

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

Returns: file descriptor if OK, -1 on error

- openat(2) is used to handle relative pathnames from different working directories in an atomic fashion.
- On some platforms additional oflags may be supported:
  - O_EXEC – Open for execute only
  - O_SEARCH – Open for search only (applies to directories)
  - O_DIRECTORY – If path resolves to a non-directory file, fail and set errno to ENOTDIR.
  - O_DSYNC – Wait for physical I/O for data, except file attributes
  - O_RSYNC – Block read operations on any pending writes.
  - O_PATH – Obtain a file descriptor purely for fd-level operations. (Linux >2.6.36 only)

openat(2)

- POSIX (https://is.gd/3hZ4EZ) says:
- *The purpose of the openat() function is to enable opening files in directories other than the current working directory without exposure to race conditions. Any part of the path of a file could be changed in parallel to a call to open(), resulting in unspecified behavior. By opening a file descriptor for the target directory and using the openat() function it can be guaranteed that the opened file is located relative to the desired directory. Some implementations use the openat() function for other purposes as well.*

close(2)

```
#include <unistd.h>

int close(int fd);
```

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.
- To avoid leaking file descriptors, always close(2) them within the same scope

read(2)

```
#include <unistd.h>

ssize_t read(int filedes, void *buff, size_t nbytes );
```

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

- There can be several cases where read returns less than the number of bytes requested. For example:
  - EOF reached before requested number of bytes have been read
  - Reading from a terminal device, one ”line” read at a time
  - Reading from a network, buffering can cause delays in arrival of data
  - Record-oriented devices (magtape) may return data one record at a time
  - Interruption by a signal
- read begins reading at the current offset, and increments the offset by the number of bytes actually read.

write(2)

```
#include <unistd.h>

ssize_t write(int filedes, void *buff, size_t nbytes );
```

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

- write returns nbytes or an error has occurred
- 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

write(2)

- Some manual pages note:
- If the real user is not the super-user, then write() clears the set-user-id bit on a file. This prevents penetration of system security by a user who “captures” a writable set-user-id file owned by the super-user.
  - When an executable file has been given the setuid attribute, normal users on the system who have permission to execute this file gain the privileges of the user who owns the file (commonly root) within the created process.
  - E.g. passwd, df
    - which passwd
    - ls –l /bin/passwd
    - To add setuid bit, chmod u+x file
- Think of specific examples for this behaviour. Write a program that attempts to exploit a scenario where write(2) does not clear the setuid bit, then verify that your evil plan will be foiled.

lseek(2)

```
#include <sys/types.h>
#include <fcntl.h>

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

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

lseek(2)

```
#include <sys/types.h>
#include <fcntl.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
- "Weird" things you can do using lseek(2):
  - seek to a negative offset
  - seek 0 bytes from the current position
  - seek past the end of the file
    - creating a hole in a file and is allowed.

Figure 3.1

```
#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);
}
```

Figure 3.1 Test whether standard input is capable of seeking

```
$ ./a.out < /etc/passwd
seek OK
$ cat < /etc/passwd | ./a.out
cannot seek
$ ./a.out < /var/spool/cron/FIFO
cannot seek
```

Figure 3.2

The program shown in Figure 3.2 creates 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 */
```

```
$ ./a.out
$ ls -l file.hole                        check its size
-rw-r--r--  1 sar       16394 Nov 25 01:01 file.hole
$ od -c file.hole                        let's look at the actual contents
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
```

I/O Efficiency

```
#include "apue.h"

#define BUFFSIZE    4096

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

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

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

    exit(0);
}
```

Figure 3.5 Copy standard input to standard output

I/O Efficiency

- assumes that stdin and stdout have been set up appropriately
- works for “text” and “binary” files since there is no such distinction in the UNIX kernel
- how do we know the optimal BUFFSIZE?

I/O Efficiency

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

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

- Read-ahead function in file systems.
  - the elapsed time for buffer sizes as small as 32 bytes is as good as the elapsed time for larger buffer sizes.