# CS262 Lecture 08
# Chapter 7 File IO

Jyh-Ming Lien
Department of Computer Science

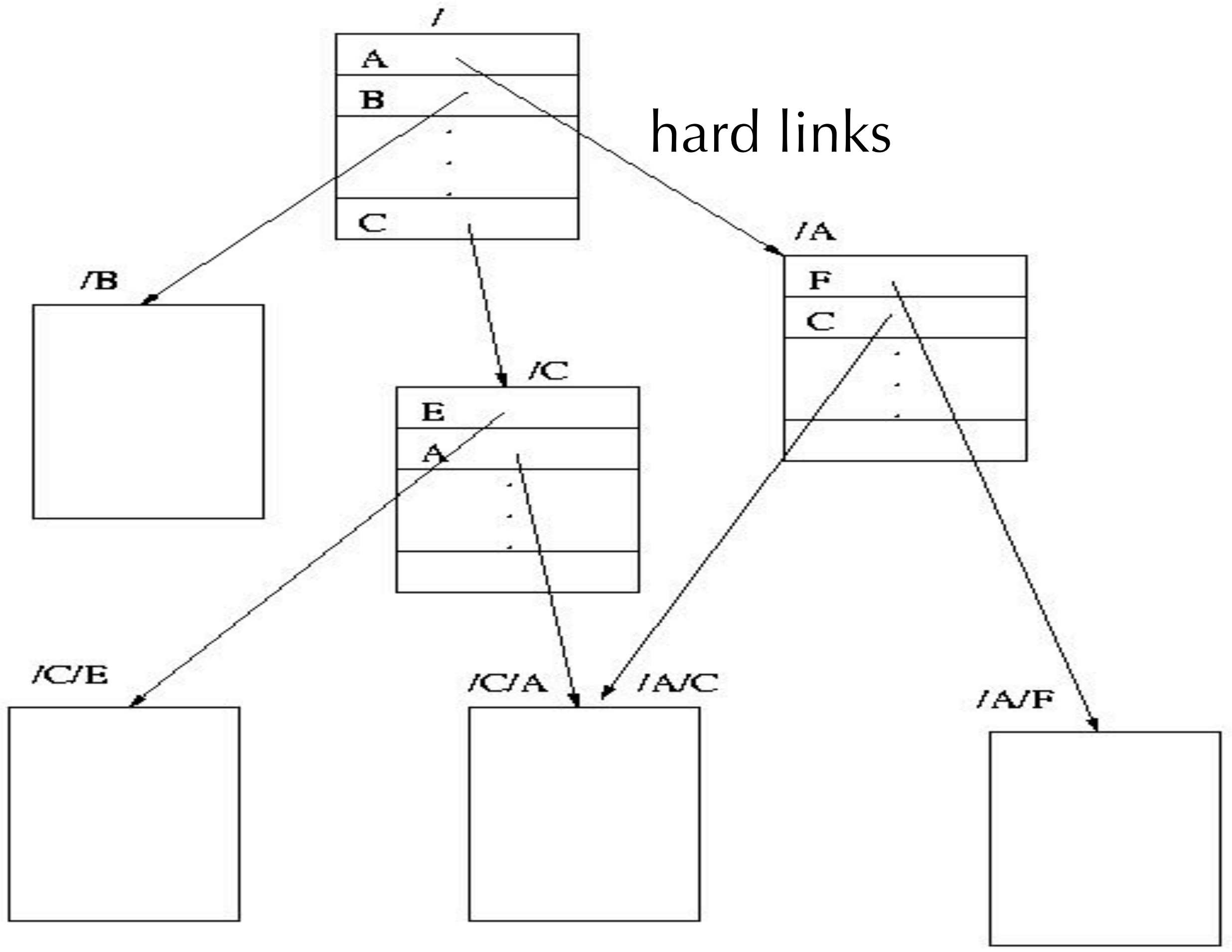
## File system abstraction

- **File**: a sequence of bytes of data
- **Filesystem**: a space in which files can be stored
- **Link**: a named logical connection from a directory to a file
- **Directory**: a special kind of file, that can contain links to other files
- **Filename**: the name of a link
- **Pathname**: a chain of one or more filenames, separated by /'s

## File system abstraction

- ***inode***: a segment of data in a filesystem that describes a file, including how to find the rest of the file in the system
- **File descriptor**: a non-negative integer, with a per-process mapping to an open file description
- **Open file description**: an OS internal data-structure, shareable between processes

# Directories



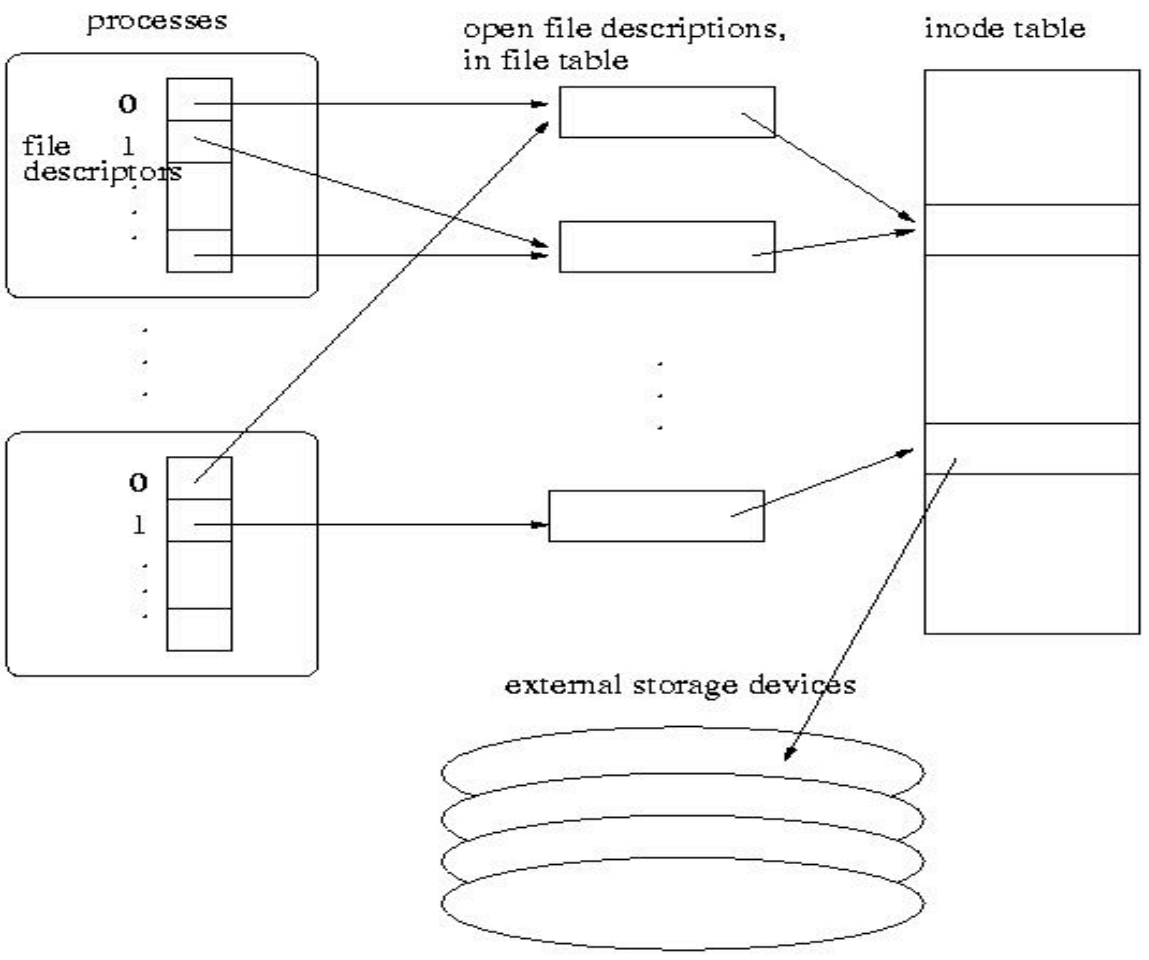
## Directories ... continued

- Names belong to links, not to files
- There may be multiple **hard links** to one file
- *Renaming* only renames one link to that file
- Unix allows both **hard** and **soft** links
- A file will exit even after the last hard link to it has been removed, as long as there are references to it from open file descriptions
  - Soft links do not prevent deletion of the file

## File Descriptors

- Each open file is associated with an *open file description*
  - Each process has a (logical) array of references to open file descriptions
  - Logical indices into this array are *file descriptors*
    - These integer values are used to identify the files for I/O operations
  - The file descriptor 0 is reserved for standard input, the file descriptor 1 for standard output, and the file descriptor 2 for the standard error

## File Descriptors



## File Descriptors

- The POSIX standard defines the following
  - **File descriptor:** A per-process, unique, nonnegative integer used to identify an open file for the purposes of file access
  - **Open file description:** A record of how a process or group of processes are *currently* accessing a file
    - Each file descriptor refers to exactly one open file description, but an open file description may be referred to by more than one file descriptor
    - A file offset, file status, and file access modes are attributes of an open file description
  - **File access modes:** Specification of whether the file can be read and written

## File Descriptors

- **File offset:** The byte position in the file where the next I/O operation *through that open file description* begins
  - Each open file description associated with a regular file, block special file, or directory has a file offset
  - A character special file that does not refer to a terminal device may have a file offset
  - There is no file offset specified for a pipe or FIFO (described later)
- **File status:** Includes the following information
  - append mode or not
  - blocking/nonblocking
  - Etc

## File Descriptors ... continued

- Important points
  - A file descriptor does not describe a file
    - It is just a number that is ephemerally associated with a particular open file description
  - An open file description describes a past "open" operation on a file; its does not describe the file
  - The description of the file is in the *inode*
    - There may be several different open file descriptors (or none) referring at it any given time

## Files in C

- In C, each file is simply a sequential stream of bytes. C imposes no structure on a file.

- A file must first be opened properly before it can be accessed for reading or writing. When a file is opened, a stream is associated with the file.

- Successfully opening a file returns a pointer to (i.e., the address of) a file structure, which contains a file descriptor and a file control block.

## Files in C

- The statement:

```
FILE *fptr1, *fptr2 ;
```

  declares that ***fptr1*** and ***fptr2*** are pointer variables of type FILE. They will be assigned the address of a file descriptor, that is, an area of memory that will be associated with an input or output stream.

- Whenever you are to read from or write to the file, you must first open the file and assign the address of its file descriptor (or structure) to the file pointer variable.

## Opening Files

- The statement:

```
fptr1 = fopen ( "mydata", "r" ) ;
```

  would open the file ***mydata*** for input (reading).

- The statement:

```
fptr2 = fopen ("results", "w" ) ;
```

  would open the file ***results*** for output (writing).

- Once the files are open, they stay open until you close them or end the program (which will close all files.)

## Testing for Successful Open

- If the file was not able to be opened, then the value returned by the ***fopen*** routine is NULL.

- For example, let's assume that the file ***mydata*** does not exist. Then:

```
FILE *fptr1 ;
fptr1 = fopen ( "mydata", "r") ;
if (fptr1 == NULL)
{
        printf ("File 'mydata' did not open.\n") ;
}
```

## Reading From Files

- In the following segment of C language code:

```
int a, b ;
FILE *fptr1, *fptr2 ;
fptr1 = fopen ( "mydata", "r" ) ;
fscanf ( fptr1, "%d%d", &a, &b) ;
```

the ***fscanf*** function would read values from the file "pointed" to by ***fptr1*** and assign those values to ***a*** and ***b***.

## End of File

- The end-of-file indicator informs the program when there are no more data (no more bytes) to be processed.
- There are a number of ways to test for the end-of-file condition. One is to use the ***feof*** function which returns a true or false condition:

```
fscanf (fptr1, "%d", &var) ;
if ( feof (fptr1) )
{
        printf ("End-of-file encountered.\n”);
}
```

## End of File

- There are a number of ways to test for the end-of-file condition. Another way is to use the value returned by the ***fscanf*** function:

```
int istatus ;
istatus = fscanf (fptr1, "%d", &var) ;
if ( istatus == EOF )
{
        printf ("End-of-file encountered.\n”) ;
}
```

## Writing To Files

- Likewise in a similar way, in the following segment of C language code:

```
int a = 5, b = 20 ;
FILE  *fptr2 ;
fptr2 = fopen ( "results", "w" ) ;
fprintf ( fptr2, "%d %d\n", a, b ) ;
```

the fprintf functions would write the values stored in ***a*** and ***b*** to the file "pointed" to by ***fptr2***.

## Closing Files

- The statements:

```
    fclose ( fptr1 ) ;
    fclose ( fptr2 ) ;
```

  will close the files and release the file descriptor space and I/O buffer memory.

## Reading and Writing Files

```
#include <stdio.h>
int main ( )
{
   FILE *outfile, *infile ;
   int b = 5, f ;
   float a = 13.72, c = 6.68, e, g ;

   outfile = fopen ("testdata", "w") ;
   fprintf (outfile, "%6.2f%2d%5.2f", a, b, c) ;
```

## Reading and Writing Files

```
    infile = fopen ("testdata", "r") ;
    fscanf (infile,"%f %d %f", &e, &f, &g) ;
    printf ("%6.2f,%2d,%5.2f\n", e, f, g) ;
    fclose(infile);
    fclose (outfile) ;
}
```

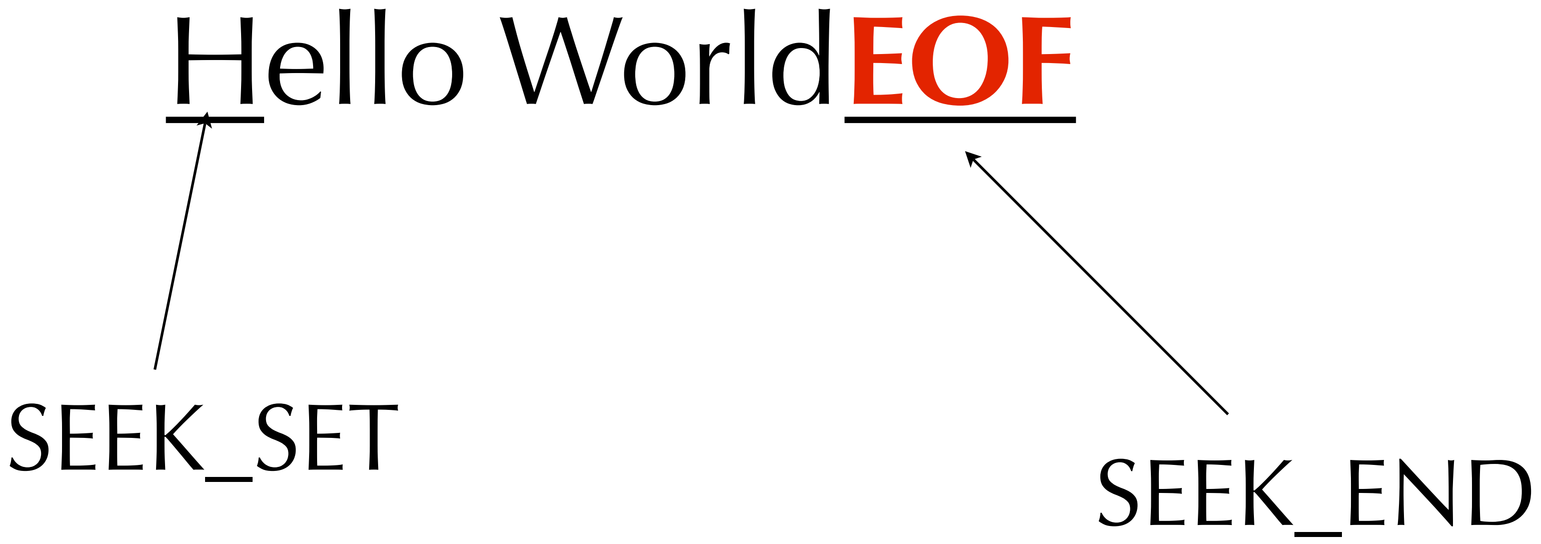
```
12345678901234567890
****************************
 13.72 5 6.68
 13.72, 5, 6.68
```

## position in a file

- change the position in a file
  - fseek(FILE * f, long int offset, int from)
- from: SEEK_SET, SEEK_CUR, or SEEK_END

- rewind(f)
  - move back to the beginning of the file
  - same as fseek(f,0,SEEK_SET)
- ftell (get current position in the file)

## position in a file



## Exception handling: signal

- examples:
  - divided by zero
  - timer
  - killed by other or its own process
  - abort abnormally
  - user press ctl-C (interrupt)
  - segmentation fault, bus error
  - file is ready to read/write
  - ...

## Exception handling: signal

- examples:
  - **SIGFPE**: divided by zero
  - **SIGALRM**: timer
  - **SIGKILL**, **SIGTERM, SIGSTOP, SIGTSTP**: killed by other or its own process
  - **SIGABRT**: abort abnormally
  - **SIGINT**: user press ctl-C (interrupt)
  - **SIGSEGV**, **SIGBUS**: segmentation fault, bus error
  - **SIGIO**: file is ready to read/write
  - ...

## Exception handling: signal

- There are many default signals and 3rd party defined signals
  - Program Error Signals
  - Termination Signals
  - Alarm Signals
  - Asynchronous I/O Signals
  - Job Control Signals
  - Nonstandard Signals

## What can you do?

- Once a signal is received, your program can
  - ignore the signal
  - specify a handler function
  - accept the default action for that kind of signal
  - unless your signal is **SIGKILL** or **SIGSTOP**

## handle a signal

- #include <signal.h>
- void signal(int sig, void (*handler)(int)))
  - handler is a function pointer and provides a call back function
- Examples
  - signal(SIGABRT,SIG_DEF); //handle SIGABRT using default behavior
  - or define a call-back function
    - void handler(int signum)

## handle a signal

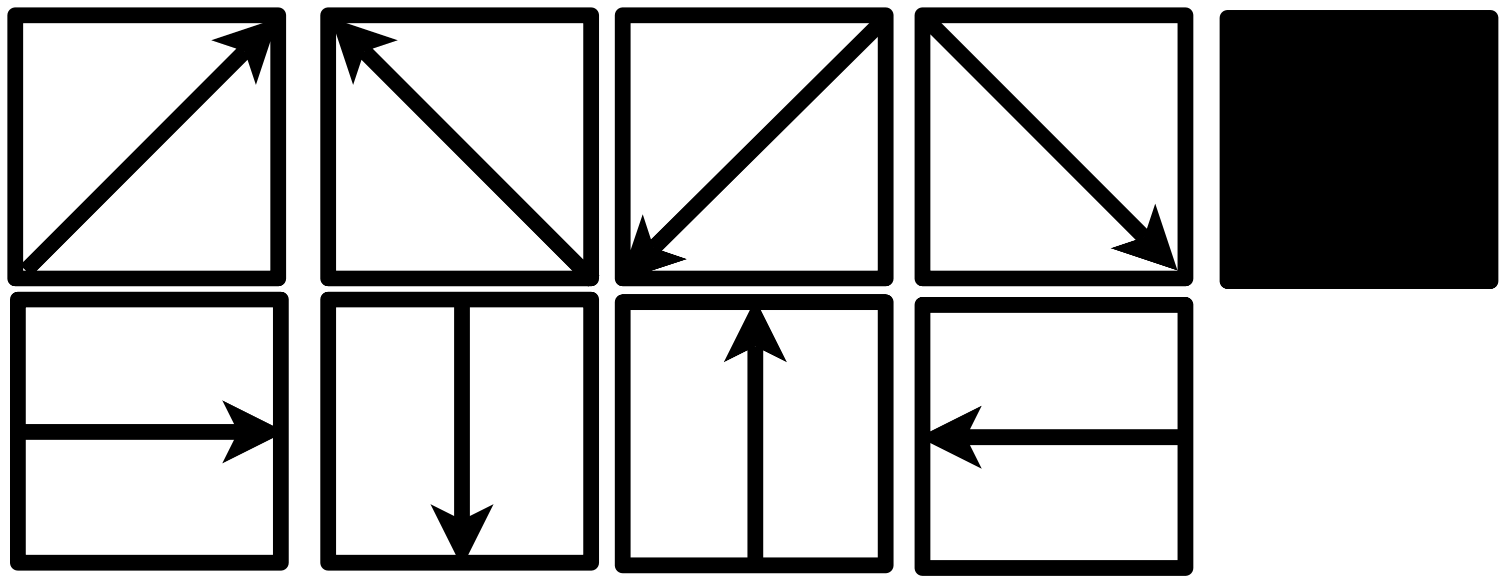
- a call-back function can be used to handle multiple signals
- strsignal(signum) converts the signum (e.g., SIGABRT) to a human readable text
- see pa3/main.c

## create signals

- int raise(int sig)
  - ex: raise(SIGABRT); //very similar to abort(0);

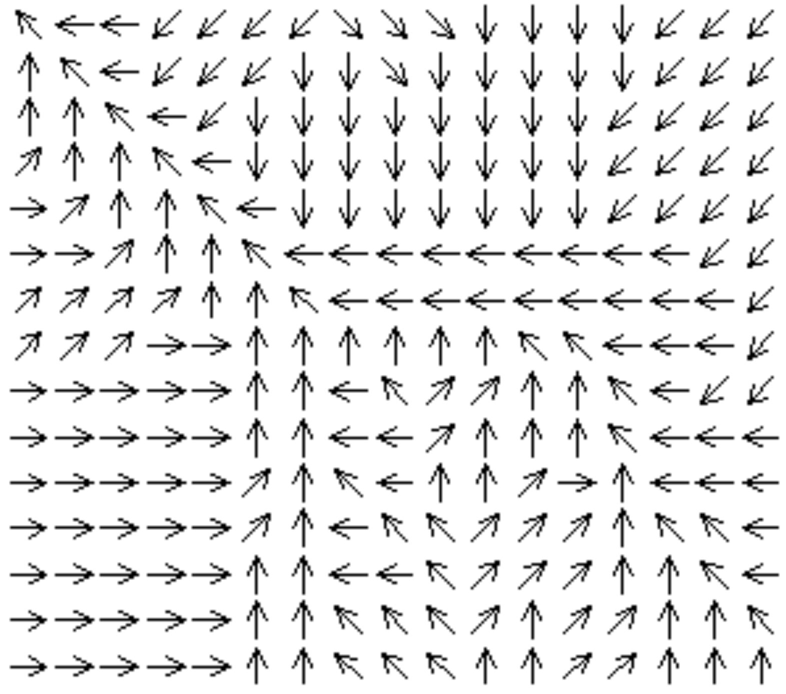
## Final Project Assignment

- using a “display library” using ANSI code
- emphasis: pointers, struct, FILE I/O, signal
- description: A bitmap in text describing an environment (a game level) is given.



## Final Project Assignment

- Each pixel in the bitmap is either a vector or an obstacle
- The environment is a vector field with obstacles



## Final Project Assignment

- Your tasks (100 pts)
  - Read given code
  - Display the vector field and obstacle (40%)
    - the size of the environment is unlimited
  - Given a particle at a given position (x,y) in the environment, trace and display the trajectory of the particle in the vector field. (60%)
  - the particle *p* is dead if
    - p is trapped in a local minimum
    - p hits an obstacle
    - p hits the boundary
    - p hits its own trajectory

## Final Project Assignment

- Your tasks (bonus points)
  - input control of the particle (for interactive mode) 10%
  - design a method to control the particle so that the particle has the longest trail (auto mode) 30%
  - design 1 large interesting environment (to be defined) 10%
- Time: 3 weeks (from the time you have the code)
-