

- Write a C program that computes the sum of 1,2,3,...,956. Compile and execute the program in Ubuntu. Submit the C program to BB.

GCC

Compilation (1)

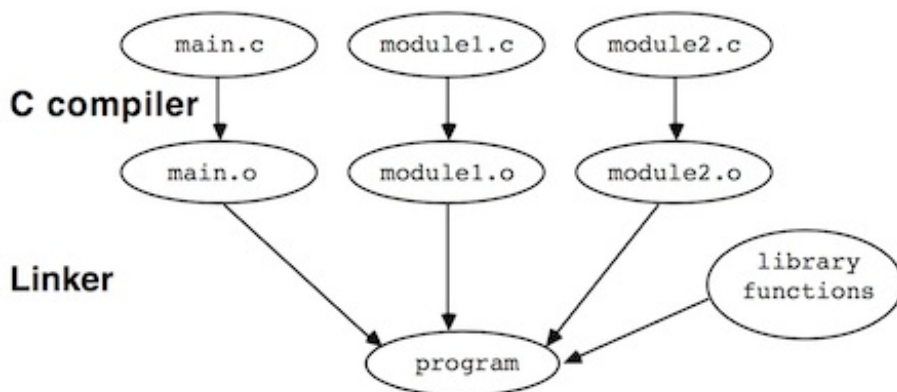
Two steps of compilation

- *compiling*:
 - input: text `.c` file
 - output: relocatable `.o` (object) file
- *linking*:
 - input: multiple relocatable `.o` files
 - output: one executable `.o` file
 - *symbol*: reference to link the declaration in one `.o` file to the definition in another `.o` file

Compilation (2)

```
gcc hello.c -o a.out
gcc -S hello.c -o hello.s #compiler
gcc -c hello.s -o hello.o #assembler
gcc hello.o -o a.out #linker
```

- gcc is a compilation system
 1. **preprocessor**: from source file to source
 2. **compiler**: from source to assembly file
 - *assembly file*
 3. **assembler**: from assembly file to relocatable object file
 4. **linker**: from multiple objects to an executable object



Linker

Compiling multiple C programs

In file1.c:

```
#include <stdio.h>
extern void foo();
int main(){
    printf("main();\n");
    foo();
}
```

In file2.c:

```
#include <stdio.h>
void foo(){
    printf("foo();\n");
}
```

Compiling multiple C programs (2)

```
gcc file1.c file2.c
# try this?
gcc file1.c
gcc file2.c
```

Compiling multiple C programs (3)

```
gcc -c file1.c # compiler & assembler
gcc -c file2.c # compiler & assembler
gcc file1.o file2.o # linker
```

Or

```
gcc -S file1.c # compiler
gcc -c file1.s # assembler
gcc -S file2.c # compiler
gcc -c file2.s # assembler
gcc file1.o file2.o # linker
```

Link library files

```
gcc -S file1.c # compiler
gcc -c file1.s # assembler
gcc file1.o file2.o # linker

mv file2.o ../libfile2.a
gcc file1.o ../libfile2.a # linker
gcc file1.o -L.. file2.o # linker
gcc file1.c -L.. file2.o # linker
```

- Gcc flag: -Ldir -lmylib for library to link

Include header file

In header1.h:

```
extern void foo();
```

In file11.c:

```
#include <stdio.h>
#include "header1.h"
int main(){
    printf("main();\n");
    foo();
}
```

```
gcc file11.c file2.c
```

Include header file (2)

Header file in another directory

```
mv header1.h ..
#will the following work?
```

```
gcc file11.c file2.c
gcc -I .. file11.c file2.c
```

- Gcc flag for searching header file: `-I path`
 - path is where the header file is

Gcc Flags (Summary)

- `-c` for specifying using gcc as a compiler
- `-o` for specifying the name of output file
- `-Ldir -lmylib` for linking a library
 - search library file `dir/libmylib.a` for solving symbols (functions, global variables) when linking
- `-I` for `#include`
 - header file (storing declarations)
- `-Wall, w` for warning
- `-g` for debug (later): `gcc -g file1.c file2.c`
- ref [[link](#)]

Exercise

- Write two C files:
 - `filea.c` defines functions `main()` and `bar()`
 - `fileb.c` defines function `foo()`
 - function `main()` calls `foo()`
 - function `foo()` calls `bar()`
 - Compile your program.
 - Submit the program and commands to BB.

Make and Makefile

Download course repo.

To download course repository, type the following commands

```
sudo apt-get update
sudo apt-get upgrade
sudo apt-get install git
git clone https://github.com/SUCourses/cis300-18spring.git
```

Makefile: Dependency rules

- `make` is a tool for project management in shell
- `Makefile` is the configuration file that tells `make` what to do
- A `Makefile` consists of a series of dependency rules
- Each dependency rule expresses IFTTT logic (if-this-then-that)

```
target: files/objects
(tab)commands
```

There is a **tab** before the commands

Helloworld Makefile

In Makefile (related files are in dir. demos/mar7)

```
all:
    gcc file1.c file2.c
```

To run it, type following command in a terminal

```
make
```

(Try change file.c, and make it again).

Makefile of Multiple Rules

```
c:
    gcc file1.c file2.c

exec: c
    ./a.out

clean:
    rm *.o *.out
```

Note there are empty lines btwn. two rules.

Use Makefile to link (1)

Recall how to run compiler, assembler and linker

```
gcc -c file1.c # compiler & assembler
gcc -c file2.c # compiler & assembler
gcc file1.o file2.o # linker
```

Use Makefile to link (2)

- Makefile helps manage different "targets"
 - A target can be compiling, assembling or linking

```
link: file1.o file2.o
    gcc file1.o file2.o
```

```
file1.o: file1.c
    gcc -c file1.c
```

```
file2.o: file2.c
    gcc -c file2.c
```

```
make
make # see if the list of files compiled changes
```

Use Makefile to link (3)

Use a default rule to compile individual C file

```
link: file1.o file2.o
    @gcc file1.o file2.o
```

```
make
make # see if the list of files compiled changes
```

- @ used to hide the command in printout.

Makefile: Using Variables

```
SRCS = file1.c file2.c
OBJS = $(SRCS:.c=.o)
CFLAGS = -g -Iheaders
#LDFLAGS = -L. -lxxx

link: $(OBJS)
      $(CC) $(LDFLAGS) $(OBJS)
```

Makefile: Using Variables (2)

- A Makefile variable is a text string
- There're standard variables
 - CC is equal to gcc (the compiler)
 - OBJS = \$(SRCS:.c=.o):
 - This incantation says that the object files have the same name as the .c files, but with .o extension
 - LDFLAGS is the linker flag of library search path (-L)
 - CFLAGS is the default compiling flags

Exercise

1. Write a Makefile such that make always clean .o files, recompiles all .c files and executes the new .o file.
2. Write a Makefile such that make link will compile a file.c file against a library file libxxx.a

GDB

References

- "Reviewing gcc, make, gdb, and Linux Editors", [[pdf](#)]
- "Unix Programming Tools", [[link](#)]

A buggy C program

```
#include<stdio.h> //printf
int array_stack[] = {0,1,2};
int main(){
    int sum; // local variable
    for(int i=0; i<=3; i++){
        sum += array_stack[i];
    }
    printf("sum = %d\n", sum);
    return 0;
}
```

Use gdb to find bug

- Installing gdb
 - on MacOS: [youtu.be/Vj33vsrDkE80]
 - on Ubuntu: `sudo apt-get install gdb`
- Compile: `gcc -g`
- Run gdb: `gdb a.out`

Gdb command: control execution

- CPU executes a C program, statement by statement

- Breakpoint is a GDB mechanism to control where CPU should pause execution
 - `break/b` is a GDB command to set breakpoint.
 - `break/b file:n|fn|file:fn`: breakpoint can be file:line number (n), function name or file:function name.
 - `disable/enable/delete` are GDB commands to disable/enable/delete a breakpoint
 - `disable/enable/delete i`: i is the index of breakpoint

Gdb command: control execution (2)

- Stepping is a GDB mechanism to control the CPU execution step by step.
 - `run/r`: start to run the program until next breakpoint or the end of program.
 - `next/n`: run just the next statement (step over a function call)
 - `continue/c`: continue the execution until the next breakpoint

Gdb command: Examine runtime

- Examine runtime data
 - `print v/p` v: print variable v
- Examine code (with `gcc -g`)
 - `list/l`
- Examine execution environment: e.g. stack (later)

Gdb commands

functionality commands

breakpoints `b,disable/enable/delete` breakpoint

stepping `r,s,n,c,finish,return`

examine_data `p/i v,display/undisplay,watch,set v=expr`

examine_code `list`

examine_stack `bt,where,info,up/down,frame`

misc. `editmode vi, b fn if expression, help, disassembler, shell cmd`

Demo

- Debug the following program using gdb

```
#include<stdio.h> //printf
int array_stack[] = {0,1,2};
int main(){
    int sum; // local variable
    for(int i=0; i<=3; i++){
        sum += array_stack[i];
    }
    printf("sum = %d\n", sum);
    return 0;
}
```

Exercise

- Exercise: Debug the following program using gdb, upload the correct program to BB.

```
#include<stdio.h>
int main() {
    int x = 5;
    int y = 3;
    int z = x - y;
```

```

int a = x * y;
int b = a - 7*z;
b--;
int c = z + y;
int d = c / b;
int e = a + 12;
int f = e - b;
printf("%d\n", f);
}

```

Pointer in C

References

- Pointer Basics: <http://cslibrary.stanford.edu/106/>
- Point fun with Binky: <http://cslibrary.stanford.edu/104/>

C Pointer

- A C pointer is a C variable that stores the reference to something.
 - "something", called pointee, is usually another variable.
- In the figure below, a pointer variable named x stores a reference to a "pointee" variable of value 42.



pointer pointee

Pointer Operations

- Definition/initialization: `int *p1 = p2;`
- Assignment: `p1 = p2;`
- Dereference: `*p = 3`
- Get reference of: `&a`
 - get the *address* (memory location) of variable a

```

#include<stdio.h>
int main(){
    int a = 10;
    int * p = & a;
    int b = *p;
    printf("a=%d,b=%d,*p=%d,p=%p\n", a, b, *p, p);
}

```

Binky's code (1)

```

void main() {
    int* x; // Allocate the pointers x and y
    int* y; // (but not the pointees)
}

```

x

y

Allocate pointer

Binky's code (2)

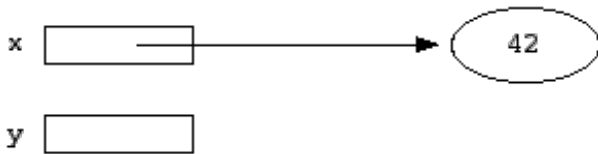
```
void main() {  
    int* x; // Allocate the pointers x and y  
    int* y; // (but not the pointees)  
    x = malloc(sizeof(int)); // Allocate an int pointee,  
                             // and set x to point to it  
}
```



Allocate pointee

Binky's code (3)

```
void main() {  
    int* x; // Allocate the pointers x and y  
    int* y; // (but not the pointees)  
    x = malloc(sizeof(int)); // Allocate an int pointee,  
                             // and set x to point to it  
    *x = 42; // Dereference x to store 42 in its pointee  
}
```



Dereference pointer

Binky's code (4)

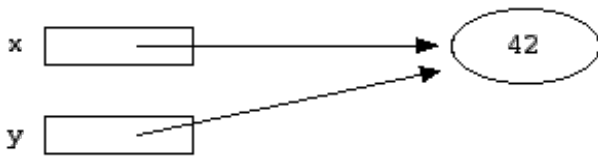
```
void main() {  
    int* x; // Allocate the pointers x and y  
    int* y; // (but not the pointees)  
    x = malloc(sizeof(int)); // Allocate an int pointee,  
                             // and set x to point to it  
    *x = 42; // Dereference x to store 42 in its pointee  
    *y = 13; // CRASH -- y does not have a pointee yet  
}
```



Dereference failure

Binky's code (5)

```
void main() {  
    int* x; // Allocate the pointers x and y  
    int* y; // (but not the pointees)  
    x = malloc(sizeof(int)); // Allocate an int pointee,  
                             // and set x to point to it  
    *x = 42; // Dereference x to store 42 in its pointee  
    *y = 13; // CRASH -- y does not have a pointee yet  
    y = x; // Pointer assignment sets y to point to x's pointee  
}
```

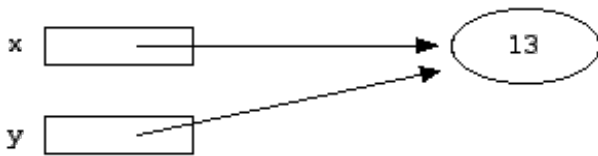


Pointer assignment

Binky's code (6)

```

void main() {
    int* x; // Allocate the pointers x and y
    int* y; // (but not the pointees)
    x = malloc(sizeof(int)); // Allocate an int pointee,
                             // and set x to point to it
    *x = 42; // Dereference x to store 42 in its pointee
    *y = 13; // CRASH -- y does not have a pointee yet
    y = x; // Pointer assignment sets y to point to x's pointee
    *y = 13; // Dereference y to store 13 in its (shared) pointee
}
  
```



Dereference pointer

Life cycle of a c pointer

	pointer	variable	function
declare	extern int * p	extern int x	void foo()
define	int *p;	int x	void foo(){}
initialize	int *p=&a; int*q=malloc(7)	int x=6	
(de)reference	*p=x;x=*p	y=x	foo()
destroy	delete p		

Exercise

- Do the following to complete the code snippet at the bottom. Then compile and execute your program. Submit the completed program to BB.
 - define two pointers p1 and p2, both pointing to variable x.
 - Use p1 to update x's value to 5.
 - Then use p2 to read the value of variable x and printf it on terminal.

```

#include<stdio.h>
int main(){
    int x = 4;
    // To complete the program below:

}
  
```

C Pointer and Data Types

Data type

- C is a typed language
- Data type in C determines:
 - How much space to allocate for storing a variable in memory
 - How to interpret bit-string stored in the memory
 - How to carry out the arithmetic on primitive types

Primitive types

- types: signed, unsigned, long long, float, char

type	signed	unsigned	short	long	long	float	char
sizeof()	4	4	2	8	4	1	

- unsigned: a 32-bit unsigned integer, value from 0 to $2^{32} - 1$.
- signed: a 32-bit signed integer, value from -2^{31} to $2^{31} - 1$.
 - first bit determines whether negative
- Typecasting: convert the type of a variable.
 - `int x = 1; double f = (double) x;`

Demo 1: Type interpretation

```
#include<stdio.h>
int main(){
    unsigned int u = 2147483649;
    int v = (int) u;
    printf("unsigned vs signed: %ud,%d\n",u,v);

    int i=1;
    float f = (float) i;
    printf("float vs int: %f,%d\n",f/3,i/3);
}
```

Demo 2: Data type size

```
#include<stdio.h>
int main(){
    signed int a;
    unsigned int b;
    short c;
    long long d;
    float e;
    char f;
    printf("signed int: %lu\n", sizeof(a));
    printf("unsigned int: %lu\n", sizeof(b));
    printf("short: %lu\n", sizeof(c));
    printf("long long: %lu\n", sizeof(d));
    printf("float: %lu\n", sizeof(e));
    printf("char: %lu\n", sizeof(f));
    return 0;
}
```

Pointer and Array

- An array in C stores a list of elements in adjacent memory locations.
- Use pointer to access array element
 - Pointer type: `char *`, `int *`
 - Pointer arithmetic:
 - `int * p = array; p += 1;`
 - `int pp = array; pp += sizeof(int);`

```
#include<stdio.h>
int main(){
    int a[] = {2,1,0};
    int *b = a; // b points to the first element in a
    unsigned long c = (unsigned long)a;//long
    for (int i=0; i<3; i++){
        printf("%d,%d,%d,%d,%d\n",a[i],*(b+i),*(a+i),b[i],*((int *) (c+i*sizeof(int))));
    }
}
```

Function pointer

- Two classes of pointers
- Data pointer: pointing to variables, arrays
- Code pointer: pointing to functions

```
#include <stdio.h> /* for printf */
// https://en.wikipedia.org/wiki/Function_pointer
double cm_to_inches(double cm) {
    return cm / 2.54;
}
int main(void) {
    double (*func1)(double) = cm_to_inches;
    printf("%f\n", func1(15.0));
    return 0;
}
```

Exercise

1. Write a C program that defines function `void foo(void)` and `int bar(long x)`. Call these two functions through function pointers. Upload your program to BB.
2. Complete the following program that scans the array using index `long_index`. Upload your program to BB.

```
#include<stdio.h>
int main(){
    int a[] = {7,9,6};
    unsigned long long_index = (unsigned long)a;
    for(int i=0; i<3; i++){
        printf("%d,",*(int *) (long_index));
        long_index += XXX;// fill out XXX
    }
}
```

File I/O

References

- "Advance Programming in the Unix Environment" (APUE), Chapter 3.1-3.8 [\[link\]](#)

Introduction

- Five functions: `open`, `read`, `write`, `lseek`, `close`
- They are unbuffered IO in the sense that each call (`read`) invokes a `syscall`.
 - Unbuffered IO functions are not ISO C, but part of POSIX.1.
- Atomic functions over shared resources.

File Descriptors

- All open files are referred to by file descriptors.

- A file descriptor is a non-negative integer.
- FD is returned by `open` or `creat`, and is used as argument to `read` or `write`.
- 0 is FD for `stdin`, 1 is the FD for `stdout`, 2 is FD of `stderr`.

open

Open a file

```
#include <fcntl.h>
int open(const char * pathname, int oflag, ...);
//returns: file descriptor if OK, -1 on error
```

- `oflag` takes one of three mandatory values and OR with optional values.
 - mandatory: `O_RDONLY`, `O_WRONLY`, `O_RDWR`
 - optional:
 - `O_CREAT`: Create a file if it doesn't exist
 - `O_TRUNC`: Truncate a file to zero if it exists and if it is opened for write-only or read-write
- the file descriptor returned is the lowest-numbered unused descriptor.

creat

Create a file

```
#include <fcntl.h>
int creat(const char * pathname, mode_t mode);
//returns: file descriptor opened for write-only if OK, -1 on error
//equiv. to
open(pathname, O_WRONLY | O_CREAT | O_TRUNC, mode);
```

close

Close a file

```
#include <unistd.h>
int close(int fd);
//return: 0 if OK, -1 on error
```

lseek

- Every open file has a "current file offset"
- `Read` and `write` starts at the offset and cause it to increment by the number of bytes read/written.

```
#include <unistd.h>
off_t lseek(int fd, off_t offset, int whence);
//Returns: new file offset if OK, 1 on error
```

- `whence`:
 - `SEEK_SET`: set offset to be `offset` plus the beginning of the file.
 - `SEEK_CUR`: set offset to `offset` plus the current value.
 - `SEEK_END`: set offset to be file size plus `offset`

read and write

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t nbytes);
//Returns: number of bytes read, 0 if end of file, 1 on error
```

- It requests to read `nbytes` bytes from `fd` and stores them in `buf`.
- If the read is successful, it returns the actual number of bytes read.

- If the end of a file is reached, it returns 0

```
#include <unistd.h>
ssize_t write(int fd, const void *buf, size_t nbytes);
//Returns: number of bytes written if OK, 1 on error
```

- It requests to write nbytes bytes to fd from buf.

Seekable Files

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

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

> ./a.out
> ./a.out < file #redirection is seekable
> cat file | ./a.out #pipe file is not seekable
```

Creat file with a hole

- header.h

```
#include<stdio.h>
#include<unistd.h> //lseek, STDIN_FILENO
#include<stdlib.h>
#include <fcntl.h>
#define FILE_MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
void err_sys(const char* x) {
    perror(x);
    exit(1);
}

#include "header.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 */
    //comment out the following two lines, get a file file.nohole
    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);
}

> cat file.hole
> cat file.nohole
> ls -ls file.hole file.nohole
> od -c file.hole
> od -c file.nohole
```

Exercise

- Write a C program that does the same thing to the following shell script. Upload your code to BB.

```
touch file1.txt
```

```
echo "Alice" >> file1.txt
cat file1.txt
```

Hint: to print a char array buf, printf("%s\n", buf);

File I/O (2)

read and write (cont'ed)

- Demo1: Copy text from stdin to stdout
 - header.h

```
#include<stdio.h>
#include<unistd.h> //lseek, STDIN_FILENO
#include<stdlib.h>
#include <fcntl.h>
#define FILE_MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
void err_sys(const char* x) {
    perror(x);
    exit(1);
}
```

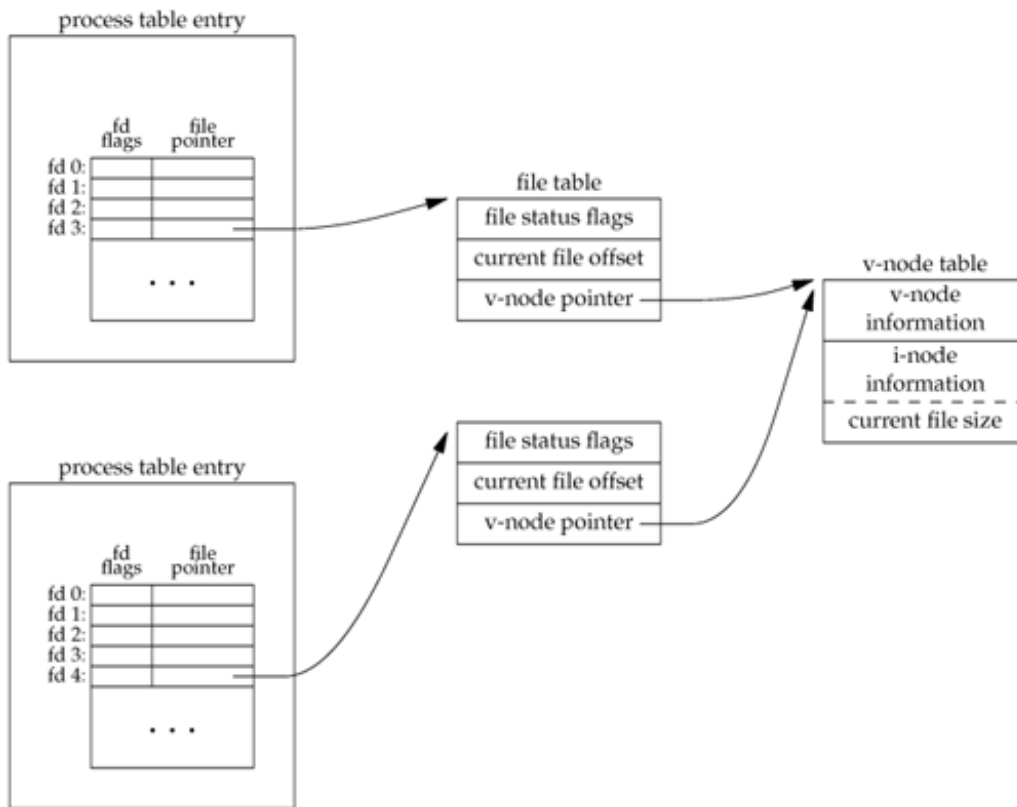
```
#include "header.h"
#define BUFFSIZE 4096
int main(void){
    int n;
    char buf[BUFFSIZE];

    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);
}
```

File Sharing & Atomic Operations

- An open file can be shared among multiple processes
- There are operations that are safe in multi-processing environment (atomic operations)
 - File append
 - pread/pwrite function
 - sync, fsync



File kernel representation and sharing

File Append

- Two ways to append text to a file: `lseek/write` and `O_APPEND`.
 - `lseek/write` is not safe in shared files.
 - `O_APPEND` is safe.

```
if (lseek(fd, 0L, SEEK_END) < 0) /* position to EOF */
    err_sys("lseek error");
if (write(fd, buf, 100) != 100) /* and write */
    err_sys("write error");
```

```
open(pathname, O_WRONLY | ... | O_APPEND, mode);
```

pread/pwrite function

- `pread` and `pwrite` functions: do seek and perform I/O atomically.
 - Calling `pread/pwrite` is equivalent to calling `lseek` followed by a call to `read/write`. There is no way to interrupt `lseek` and `read/write`.

```
#include <unistd.h>
ssize_t pread(int filedes, void *buf, size_t nbytes, off_t offset);
// Returns: number of bytes read, 0 if end of file, 1 on error

ssize_t pwrite(int filedes, const void *buf, size_t nbytes, off_t offset);
//Returns: number of bytes written if OK, 1 on error
```

Sync, fsync

- Traditional UNIX system has a page cache in the kernel that queues data writes before the buffer overflows and it flushes data to disk.
- `sync/fsync` ensures consistency of the file system on disk with the contents of the buffer cache.
- The `sync` function simply queues all the modified block buffers for writing and returns; it does not wait for the disk writes to take place.

- The function `fsync` waits for the disk writes to complete before returning. The intended use of `fsync` is database applications that need be sure that the modified blocks have been written to the disk.

fcntl

- The `fcntl` function reads/changes the properties of an open file.

```
#include <fcntl.h>
int fcntl(int filedes, int cmd, ... /* int arg */);
//Returns: depends on cmd if OK (see following), 1 on error
```

Demo2: Print flags for specified descriptor

- see demos/apr16/demo2.c

```
#include "header.h"
int main(int argc, char *argv[]) {
    int val;
    if (argc != 2)
        err_quit("usage: a.out <descriptor#>");
    if ((val = fcntl(atoi(argv[1]), F_GETFL, 0)) < 0)
        err_sys("fcntl error for fd %d", atoi(argv[1]));
    switch (val & O_ACCMODE) {
    case O_RDONLY:
        printf("read only");
        break;
    case O_WRONLY:
        printf("write only");
        break;
    case O_RDWR:
        printf("read write");
        break;
    default:
        err_dump("unknown access mode");
    }

    if (val & O_APPEND)
        printf(", append");
    putchar('\n');
    exit(0);
}
```

```
./a.out 0 < /dev/tty
./a.out 1 > temp.foo; cat temp.foo
./a.out 2 2>> temp.foo
./a.out 5 5<> temp.foo
```

5<>temp.foo opens file temp.foo for read/write on file descriptor 5

stat

- `stat` shell command
- `stat` returns information about a named file.
- `fstat` returns information about the open file by descriptor.

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

int stat(const char *restrict pathname, struct stat *restrict buf);
int fstat(int fd, struct stat *buf);
//All three return: 0 if OK, 1 on error
```

File types

- Regular file: The most common file type, which contains data of some form.
- Directory file: file that contains the names of other files and pointers to information on these files.

- Block special file: file providing buffered I/O access in fixed-size units to devices such as disk drives.
- Character special file: file providing unbuffered I/O access in variable-sized units to devices.
- FIFO: file used for communication between processes, such as named pipe.
- Socket: file used for network communication between processes.
- Symbolic link: file that points to another file.

Demo3: Print file types

```
int main(int argc, char *argv[]) {
    int i;
    struct stat buf;
    char *ptr;

    for (i = 1; i < argc; i++) {
        printf("%s: ", argv[i]);
        if (lstat(argv[i], &buf) < 0) {
            err_ret("lstat error");
            continue;
        }
        if (S_ISREG(buf.st_mode))
            ptr = "regular";
        else if (S_ISDIR(buf.st_mode))
            ptr = "directory";
        else if (S_ISCHR(buf.st_mode))
            ptr = "character special";
        else if (S_ISBLK(buf.st_mode))
            ptr = "block special";
        else if (S_ISFIFO(buf.st_mode))
            ptr = "fifo";
        else if (S_ISLNK(buf.st_mode))
            ptr = "symbolic link";
        else if (S_ISSOCK(buf.st_mode))
            ptr = "socket";
        else
            ptr = "** unknown mode **";
        printf("%s\n", ptr);
    }
    exit(0);
}
```

```
>./a.out /etc/passwd
/etc/passwd: regular
>./a.out .
.: directory
>./a.out /dev/disk0
/dev/disk0: block special
>./a.out /dev/io8log
/dev/io8log: character special
```

Exercise

- Write a C program to simulate echo command
 - Hint: use read/write file IO functions.