Today's Lecture

1. High-level C program structure
2. C preprocessor (`cpp`)
3. System Calls vs. Library Calls
4. Process, Files, File Systems
   (a) Process related system calls
   (b) File IO
   (c) File system operations
5. Anatomy of a `makefile`

```
/* typical.c */
#include <stdio.h>
⋮                                   preprocessor
#define N_COLS 80                   directives
⋮

/* some types */
typedef unsigned char byte;
⋮
/* some globals */                  declarations
int cursor_x = 0;
int cursor_y = 0;
⋮

/* code starts here */
void main(int argc, char *argv[])
{                                   code
/* body of main goes here */
}

/* end of typical.c */
```

Note function prototypes; compile with `gcc -ansi typical.c`

Preprocessor cpp

Handles directives such as:

- ```
  #define WIDTH 10
  #define HEIGHT 10
  #define VOLUME (HEIGHT * WIDTH)
  ```

- ```
  #define IS_ODD(x) ((x) & 0x1)
  ```

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

  (angle brackets indicate `/usr/include/`)

- ```
  #include "myheader.h"
  ```

  (`cpp` option `-I` can be used to search other paths)

- ```
  #if VOLUME > 100
     /* code for large volume */
  #else
     /* code for small volume */
  #endif
  ```

- ```
  #ifdef USE_SHIFT
  #   define TIMES_TWO(x) ((x) << 1)
  #else
  #   define TIMES_TWO(x) ((x) * 2)
  #endif
  ```

- ```
  #ifndef                        /* if not defined */
  ```
  ```

Preprocessor cpp (cont.)

- `cpp` is first pass of `cc` (and `gcc`) compiler

- `cpp` can be invoked separately

```
/lib/cpp typical.c >typical.cpp.c
```

this is useful for:

  – debugging macros

```
#define TIMES_TWO(x) (x >> 1)
⋮
printf("%d", TIMES_TWO(y + 1));
```

  – preprocessing other input, e.g.,

```
/lib/cpp -P mypage.html >mypage.cpp.html
```

A cpp "Trick"

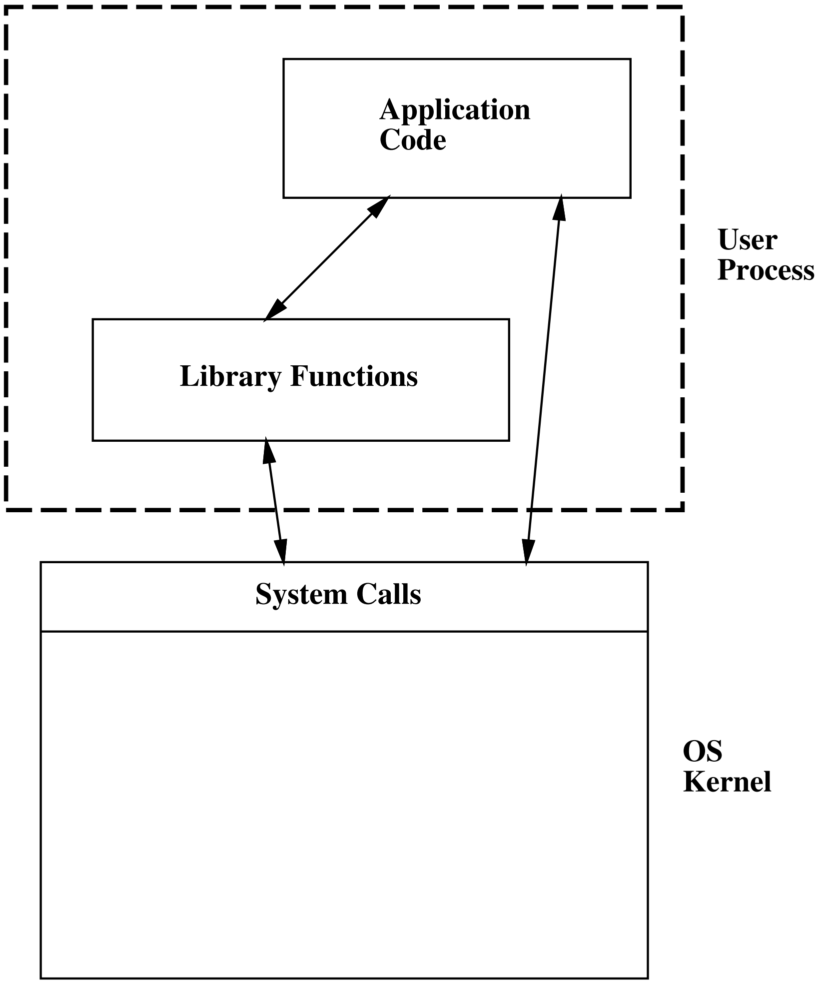
Multiple `#include`s of the same `.h` file can lead to problems. Ways to avoid this:

1. Don't `#include` files from other `#include`d files

2. At the top of a `.h` file do:

```
/* typical.h */
#ifndef __TYPICAL__
#define __TYPICAL__
⋮
/* contents of typical.h */
⋮
#endif
```

System Calls versus Library Calls



Processes

- Roughly, Program = Process
- In Unix, every command you type spawns a process. The process completes (and goes away) when the command completes.
- Command `ps` (with lots of command-line options) shows processes in the system.
- The OS divides access to the CPU among active processes (this is called *time-slicing*).
- In MS-DOS and MacOS, there is in essence only one process. That's why you can't actively run more than one program at a time with those operating systems.
- Windows 95/NT have processes similar to Unix processes.

Processes (cont.)

- Processes may communicate with one another (e.g. via pipes).

- Processes have IDs. In Unix, there is a C **`typedef`**, accessible by,

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

  for process IDs called **`pid_t`**.

- Processes have other attributes: current working directory, environment, resource limits, *etc.*

- Most attributes are inherited from the process that created it.

Process Syscalls: getpid, getppid

From man pages:

```
GETPID(2V)                SYSTEM CALLS                GETPID(2V)

NAME
     getpid, getppid - get process identification

SYNOPSIS
     int getpid()
     int getppid()

SYSTEM V SYNOPSIS
     #include <sys/types.h>

     pid_t getpid()
     pid_t getppid()

DESCRIPTION
     getpid() returns the process  ID  of  the  current  process.
     Most  often  it is used to generate uniquely-named temporary
     files.

     getppid() returns the  process  ID  of  the  parent  of  the
     current process.
```

Example Program: getpid, getppid


```
/* getpid-getppid.c */
#include <sys/types.h>

void main(int argc, char *argv[])
{
  printf("pid is %d and parent pid is %d\n", getpid(), getppid());
}

/* end of getpid-getppid.c */
```

```
$ gcc -ansi -o getpid-getppid getpid-getppid.c
$ getpid-getppid
pid is 22024 and parent pid is 2106
$ getpid-getppid
pid is 22028 and parent pid is 2106
$
```

What assumption did I make in the code?

Process Syscalls: `fork`

`fork` creates a new (child) process. From the `man` pages:

```
FORK(2V)                 SYSTEM CALLS                 FORK(2V)

NAME
     fork - create a new process

SYNOPSIS
     int fork()

SYSTEM V SYNOPSIS
     pid_t fork()

DESCRIPTION
     fork() creates a new process.  The new process  (child  pro-
     cess) is an exact copy of the calling process except for the
     following:

     +  The child process has a unique process ID. The child pro-
        cess ID also does not match any active process group ID.

     +  The child process has a different parent process ID  (the
        process ID of the parent process).

⋮

RETURN VALUES
     On success, fork()  returns  0  to  the  child  process  and
     returns  the  process  ID of the child process to the parent
     process.  On failure, fork() returns -1 to the  parent  pro-
     cess, sets errno to indicate the error, and no child process
     is created.
```

Example Program: fork


```
/* fork.c */
#include <sys/types.h>

int global_variable = 0;

void main(int argc, char *argv[])
{
  int local_variable = 0;
  pid_t pid;

  printf("before fork\n");
  if ((pid = fork()) < 0) {
    printf("error on fork\n");
    exit(1);
  } else if (pid == 0) {  /* child does this */
    global_variable++;
    local_variable++;
  } else {                /* parent does this */
  }
  printf("pid = %d, global_variable = %d, local_variable = %d\n",
         getpid(), global_variable, local_variable);
  exit(0);
}
/* end of fork.c */
```

```
$ gcc -ansi -o fork fork.c
$ fork
before fork
pid = 29720, global_variable = 1, local_variable = 1
pid = 29719, global_variable = 0, local_variable = 0
$
```

Files

- *Files* store data (on disk, tape, memory, *etc.*)

- Files are organized into *file systems* (e.g., directories). We will discuss file-system system calls a few slides in the future.

- File system calls handle storing data in (`write`), and reading data from (`read`), files. The also do bookkeeping (`open`, `close`) and allow random access (`lseek`).

- Windows and MacOS provide system calls with equivalent functionality.

- Unix treats all files as containing 8-bit bytes (characters).

- Windows handles text files differently from binary files (it looks for newline characters). This is a pain when converting software to/from Unix. MacOS?

File Syscall: `open`

Provides a "handle" for a file. This handle is called a *file descriptor* and is used to name the file in subsequent operations.

Paraphrasing the **man** pages:

```
NAME
     open - open or create a file for reading or writing

SYSTEM V SYNOPSIS
     #include <sys/types.h>
     #include <sys/stat.h>
     #include <fcntl.h>

     int open(path, flags[ , mode ] )
     char *path;
     int flags;
     mode_t mode;

DESCRIPTION
     path points to the pathname of a  file.   open()  opens  the
     named  file  for reading and/or writing, as specified by the
     flags argument, and returns a descriptor for that file.

     flags values  are  constructed  by  ORing flags from the following
     list (one and only one of the first three flags  below  must
     be used):

     O_RDONLY       Open for reading only.

     O_WRONLY       Open for writing only.

     O_RDWR         Open for reading and writing.

     O_APPEND       If set, the seek pointer will be set  to  the
                    end of the file prior to each write.

     O_CREAT        If the file exists, this flag has no  effect.
                    Otherwise, the file is created, and the owner
                    ID of the file is set to the  effective  user
                    ID  of the process.
RETURN VALUES
     open() returns a non-negative file  descriptor  on  success.
     On  failure,  it  returns  -1 and sets errno to indicate the
     error.
```

Note: mode parameter is used in conjunction with `O_CREAT` (see `man 2 stat`)

File Syscall: `write`

Once you have opened a file for writing (`O_WRONLY` or `O_RDWR`), you may store data in it using **`write`**.

Paraphrasing the **`man`** pages:

```
WRITE(2V)                 SYSTEM CALLS                 WRITE(2V)

NAME
     write - write output

SYSTEM V SYNOPSIS
     int write(fd, buf, nbyte)
     int fd;
     char *buf;

DESCRIPTION
     write() attempts to write nbyte bytes of data to the  object
     referenced  by  the descriptor fd from the buffer pointed to
     by buf.
     If nbyte  is zero, write() takes no action and returns 0.

RETURN VALUES
     write() returns the  number  of  bytes  actually written  on
     success.   On  failure,  write returns -1 and sets errno to
     indicate the error.
```

File Syscall: `read`

Once you have opened a file for reading (`O_RONLY` or `O_RDWR`), you may retrieve data from it using **`read`**.

Paraphrasing the **`man`** pages:

```
READ(2V)                  SYSTEM CALLS                  READ(2V)

NAME
     read - read input

SYNOPSIS
     int read(fd, buf, nbyte)
     int fd;
     char *buf;
     int nbyte;

DESCRIPTION
     read() attempts to read nbyte bytes of data from the object
     referenced  by  the descriptor fd into the buffer pointed to
     by buf.

     If nbyte is zero, read() takes  no  action  and  returns  0.
     Upon successful completion, read() returns the number of bytes
     actually read and placed in the buffer.

     If nbyte is not zero and read() returns 0, then EOF has been
     reached.

RETURN VALUES
     read() returns the number of bytes actually  read on success.
     On failure,  read returns -1 and sets errno to indicate the error.
```

File Syscall: `lseek`

You can "move" to an arbitrary position in a file (even to positions where data hasn't been written!) with **`lseek`**. Positions are numbered with the first byte (character) occurring at position zero.

Paraphrasing the **`man`** pages:

```
LSEEK(2V)                 SYSTEM CALLS                 LSEEK(2V)

NAME
     lseek - move read/write pointer

SYNOPSIS
     #include <sys/types.h>
     #include <unistd.h>

     off_t lseek(fd, offset, whence)
     int fd;
     off_t offset;
     int whence;

DESCRIPTION
     lseek() sets the seek pointer associated with the open  file
     or  device referred to by the descriptor fd according to the
     value supplied for whence.  whence must be one of  the  fol-
     lowing constants defined in <unistd.h>:

               SEEK_SET
               SEEK_CUR
               SEEK_END

     If whence is SEEK_SET, the seek pointer  is  set  to  offset
     bytes.   If  whence  is SEEK_CUR, the seek pointer is set to
     its current location plus offset.  If  whence  is  SEEK_END,
     the seek pointer is set to the size of the file plus offset.

RETURN VALUES
     On success, lseek() returns the  seek  pointer  location  as
     measured  in  bytes  from  the  beginning  of  the file.  On
     failure, it returns -1 and sets errno to indicate the error.
```

File Syscall: `close`

When you're through reading, writing, and seeking a file, `close` it.

Paraphrasing the `man` pages:

```
CLOSE(2V)                 SYSTEM CALLS                 CLOSE(2V)

NAME
     close - delete a descriptor

SYNOPSIS
     int close (fd)
     int fd;

DESCRIPTION
     close() deletes a descriptor  from  the  per-process  object
     reference  table.  If fd is the last reference to the under-
     lying object, then the  object  will  be  deactivated.

RETURN VALUES
     close() returns:

     0    on success.

     -1   on failure and sets errno to indicate the error.
```

Note: if you exit a program (with `exit`), any open files will be closed for you. However, it is good style to explicitly close a file when your through with it.

Putting File Syscalls Together: `mycp.c`

```
/* mycp.c */
/* copies file named by argv[1] to file named by argv[2] */

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

#define BUF_SZB 512

void an_error(char *s)
{
  printf("%s\n", s);
  exit(1);
}

void main(int argc, char *argv[])
{
  int from_fd,  to_fd;
  char buf[BUF_SZB];
  int bytes_read;

  if (argc != 3) {
    an_error("usage: mycp from-file to-file");
  }
  if ((from_fd = open(argv[1], O_RDONLY)) == -1) {
    an_error("couldn't open from-file for reading");
  }
  if ((to_fd = open(argv[2], O_WRONLY | O_CREAT,  S_IRUSR | S_IWUSR)) == -1) {
    an_error("couldn't create to-file for writing");
  }
  while ((bytes_read = read(from_fd, buf, BUF_SZB)) != 0) {
    if (write(to_fd, buf, bytes_read) == -1) {
      an_error("error writing to to-file");
    }
  }
  if (close(from_fd) == -1) {
    an_error("couldn't close from-file");
  }
  if (close(to_fd) == -1) {
    an_error("couldn't close to-file");
  }
  exit(0);
}
/* end of mycp.c */
```

Running `mycp.c`


```
$ gcc -ansi -o mycp mycp.c
$ ls
mycp*   mycp.c
$ cat >file-a
a cat in the hat
$ ls
file-a  mycp*  mycp.c
$ mycp file-a file-b
$ ls
file-a  file-b  mycp*  mycp.c
$ cat file-b
a cat in the hat
$
```

File Systems

- Files are organized into *file systems*. A file system also maintains *attributes* for every file; i.e. information on who owns it, what operations may be performed on it, *etc.*

- *Directories* of files are central to a filesystem. Related syscalls:
  (**`mkdir`**, **`rmdir`**, **`chdir`**, **`opendir`**, **`seekdir`**, **`rewinddir`**, **`closedir`**)

- The **`chmod`** syscall changes a file's mode (i.e. it's access permissions).

- The **`unlink`** syscall removes a file.

- The **`stat`** and **`fstat`** system calls allow you to inspect a files attributes (size, owner, creation time, access time, ...)

- Windows and MacOS provide system calls with equivalent functionality.

File-System Syscalls: `stat`, `fstat`, `lstat`

More about a file (also directory) than you (usually) want to know.

Paraphrasing the `man` pages:

```
NAME
     stat - get file status

SYNOPSIS
     #include <sys/types.h>
     #include <sys/stat.h>

     int stat(path, buf)
     char *path;
     struct stat *buf;

DESCRIPTION
     stat() obtains information about the  file  named  by  path.
     Read,  write  or execute permission of the named file is not
     required, but all directories listed in the path name  lead-
     ing to the file must be searchable.

     buf is a pointer to a stat structure into which  information
     is  placed  concerning  the file.  A stat structure includes
     the following members:

          dev_t     st_dev;     /* device file resides on */
          ino_t     st_ino;     /* the file serial number */
          mode_t    st_mode;    /* file mode */
          nlink_t   st_nlink;   /* number of hard links to the file */
          uid_t     st_uid;     /* user ID of owner */
          gid_t     st_gid;     /* group ID of owner */
          dev_t     st_rdev;    /* the device identifier (special files only)*/
          off_t     st_size;    /* total size of file, in bytes */
          time_t    st_atime;   /* file last access time */
          time_t    st_mtime;   /* file last modify time */
          time_t    st_ctime;   /* file last status change time */
          long      st_blksize; /* preferred blocksize for file system I/O*/
          long      st_blocks;  /* actual number of blocks allocated */

RETURN VALUES
     stat() returns:
     0    on success.
     -1   on failure and set errno to indicate the error.
```

Note: **`fstat`** retrieves this info from a file descriptor (as returned by **`open`**) instead of from a path name.

Automatic project builds: `make`

- A program file A is said to *depend* on a program file B if a change to B can affect A's behavior.

- For example, if you change code in `typical.h`, the behavior of `typical.c` may change; hence, `typical.c` depends on `typical.h`

- The `make` program lets you specify the file dependencies for the files that constitute a program.

- Create a `makefile` (that's the actual name of the file in the program's directory) that contains dependence information.

- The `makefile` also contains rules on how to rebuild pieces of the program that are out of date.

Anatomy of a `makefile`


```
# a makefile for a sample program

# variables
CC = gcc -ansi

# 'all' is the first target
# without an explicit target, make will build the first
# one in the makefile
all: run-me

# target for the executable
run-me: main.o driver.o graphics.o
	$(CC) -o run-me main.o driver.o graphics.o

# explicit rule for making main.o
main.o: main.c macros.h defines.h
	$(CC) -c main.c

# explicit rule for making graphics.o
graphics.o: graphics.c defines.h
	$(CC) -c graphics.c

# driver.o depends only on driver.c
# the necessary rule is built into make

# a target to remove all files that can be rebuilt
# when a program stops working, it's a good idea
# to 'make clean' followed by 'make'
clean:
	rm -rf run-me run-me.debug *.o *.d
```

Note: a rule must start with a tab charater

For the Next Lecture

- Find out about buffered library IO: `fopen`, `fclose`, `feof`, `fread`, `fwrite`, `fseek`, `fprintf`, ...

- What about IO to the console, from the keyboard, or to some other ephemeral file device? `stdin`, `stdout`, & `stderr`

- Find out what shell you're running:

```
cs333@phoenix.Princeton.EDU (35) % chsh
Changing NIS login shell for cs333 on diskfarm.
Old shell: /bin/csh
New shell:
Login shell unchanged.
cs333@phoenix.Princeton.EDU (36) %
```

  (you can also discover this via `finger`)

- Learn about your shell:
  `man -M /usr/princeton/man csh`
  `man -M /usr/princeton/man ksh`
  `man -M /usr/princeton/man tcsh`
  etc.

- Next lecture: shell programming with the `ksh` (see `man -M /usr/princeton/man ksh`)