Need to add in w4_2

What is OS: a program that acts as an intermediary between a “user” of a computer and the computer hardware

Kernel: library of procedures shared by all user programs, but kernel is protected:

User code cant access internal kernel data structures directly

User code can invoke the kernel only at well-defined entry points (system calls)

Process: an executing program: container for computing resources (abstraction)

Thread: an executing stream of instructions normally within a process

Synchronization: solution to problems of concurrency (none know who should go first, or all are blocking the next from moving so none can move.)

File System/Directories: abstraction to make data storing and sharing easier

Process communication: connected by a pipe channel, communicate to decompose complex applications, web browser/server

Structure of a C program:

Structs, arrays, typedefs

One function must be called main: int main(int argc, char *argv[])

No string type in C, close: char *string, char [] string = “abc”, char [length] string

Type conversions: int x,y; x = atoi(argv[1]); //string to int y = x+10;

Program Scoping

```
Global: static int foo; //allocated and avail only to the file containing this declaration
        int bar; //allocated, global and exportable to any module
        extern int baz; //allocated elsewhere, allocation(int baz) must linked in eventually
```

Libraries and include files

Usually functions themselves are in standard libraries, if NOT must use -l<lib-name> when compiling

Compiling

```
gcc -c foo foo.c --- or mult modules: gcc -c foo1.c, gcc -c foo2.c, gcc -o foo foo1.o foo2.o
(OR) gcc -o foo foo1.c foo2.c // all at once //add -v for more data at compile time
```

Error handling

```
#include <unistd.h>
//-1 returned if failure, sets errno(extern int)
int close(int fildes);
if (close (fildes) == -1)
        perror(“close failed …”); //uses errno
```

0:

NULL = 0 = zero pointer = logical not

#define NULL 0

0 is a safe default for int flag in system call

Pointers = memory address

```
int x; int *y; y = &x; *y = 10;
```

Memory Allocation:

```
void *ptr1; my_t *ptr2; ptr1 = malloc(5); ptr2 = (my_t *) malloc(sizeof(my_t));
free(ptr1); free(prt2);
```

VOID *

Void pointers can be caster to any pointer type and vice-versa

```
void *ptr; char *aptr;  ptr=(void *) aptr;  aptr=(char *) ptr;
```

Buffer overflow:

Writing more to an array/pointer than there is space

This will write over local variables on the stack & possibly return address of call

Seg Fault:

Program attempts to access memory outside its boundaries

Debugging:

```
#ifdef DEBUG
        printf(stderr, “A=%d\n”, A);
#endif
gcc -o foo foo.c -DDEBUG
```

Processes

Root is called init, started by OS

Child is created by parent

fork():

#include <sys/types.h> #include <unistd.h> pid_t fork(); pid_t is a process id (#)

Creates a clone of the parent process, same code, same files, same register vals, diff IDs, signals, CPU time measures

```
#include <unistd.h>
int pid;
int status = 0;
pid = fork ();
if (pid > 0) {
printf(“Parent: child has pid =%d”, pid);
pid = waitpid(pid, &status, 0);
} else if (pid== 0) {
printf(“child here”);
exit(status);
} else {
perror(“fork problem”);
exit (-1); }
```

Waiting:

#include <sys/wait.h>

Pid_t wait(int *stat_loc); //any child Pid_t waitpid(pid_t pid, int *stat_loc, int options); //spec chld

Stat_loc = why did child exit, wait(NULL); or int stat_loc; wait(&stat_loc);

exec*:

After the fork then you can call exec() to change the code that it runs, overrides child’s mem except ids)pids)

```
#include <unistd.h>
int pid;
```

```
pid = fork ();
if (pid> 0) {
pid= wait (&status);
} else if (
pid== 0) { // child does not return if successful!
execl("/bin/prog", "prog", (char*)0);
perror("exec failed");
exit(-1);
} else {
perror("fork problem");
exit (-1);}
```

How can fail: bad arguments, bad executable name, permissions on executable say can't run.

Write fail: Invalid fd and invalid permissions

Identities:

- fork() returns pid of the child to the parent
- getpid() returns the pid of the calling process
- getppid() returns the parent's pid
- getuid() returns user id of the user that started the process (eg "jon => 89392)

High-level

- Hides "device" or low-level I/O, low-level abstraction: source/sink for data
- More than raw bytes
- Features: Stream abstraction, Formatted I/O, String-based I/O, Line-oriented buffering

stdio.h

- Abstraction, buffering, lots of functions,
- User-space library: the memory area where application software and some drivers execute.

FILE object: (defined in stdio.h)

- Delivers an ordered sequence of bytes, <stdio.h>, built on top of low level fd
- 3 default streams: stdin, stdout, stderr
- Points to an actual file, the ptr keeps track of the current file offset for read and write
- Mode: read, write, append
- Write buffer
    - Internal character buffer of size BUFSIZE
    - Writes are done to in memory buffer
        - When buffer is full, write out buffer to the file
        - Or if line-buffered stream (stdout) when '\n' is written
- Read buffering
    - Reads are done from in memory buffer
    - When buffer is empty, we read a chunk of BUFSIZE
- Buffering FFLUSH
    - Force the buffer to spill into the OS
    - int fflush(FILE *stream); fprintf(F, "blah"); fflush(F); , stderr() always causes flush stdout with '\n' as well

Open/close file:

- FILE *fopen(const char *filename, const char *mode); mode = "r","w","a"

```
int fclose(FILE *stream); //0=success EOF else -1
```

Character-based I/O

```
    #include <stdio.h>
    //all following calls return EOF on fail
    int getc(FILE *F);
int putc(int char, FILE *F);
int ungetc(int char, FILE *F); goes back 1 char
```

String-based I/O

```
char*fgets(char *buf, int nsize, FILE *inf); //reads nsize-1 characters or up to a
newline '\n' or EOF into buf (caller allocates this) appends \0 at end of buf
//will include '\n' in the string (so will stdin)
int fputs(const char *buf, FILE *outf); //outputs buf to file-- better be \0 terminated!
                                        //returns last char written or EOF if an error occurs
strcpy, strncpy, strlen, strdup
```

I/O for data types

```
        int fprintf(FILE *outf, "words and %d", x);
        int sprintf(char *stringToPutStuffIn, "%d", x);
int x=4; char str[100]; FILE *f; F=fopen("myfile", "w"); fprinf(F, "%d", x);
sprintf(str, %d %d %s", 12, x, "hello"); =>"12 4 hello"

int i1, i2; float f1t;   char str1[10], str2[10];   char *str1, *str2;
sscanf("FileReadingFrom.txt", "%d %d %f %s %s", &i1, &i2, &f1t, str1, str2);

sscanf(FILE *in, const char *fmt, ptr to allocated argos);
```

Binary I/O

Takes up less space and can read/write with fewer bytes

```
        Typedef struct S{
            Int ss; //8 digits
            int phone; //10 digits
        } info_t;
        info_t mine = {12345678, 23232332323);
        fprintf(F, "%d %d", mine.ss, mine.phone);
        fwrite((void *)&mine, sizeof(info_t), 1, F)
        fread((void *)&mine, sizeof(info_t), 1, F);
```

Random I/O

Advance the offset pointer without reading/writing

```
        #include <stdio.h>
        int fseek(FILE *F, long offsetInBytes, int whence);
whence = SEEK_SET,SEEK_CUR, SEEK_END
long ftell(FILE *F); //get current file offset
```

Low-level:

No notion of files, no formatted I/O, just bytes, more control (read 1 char, dont buffer)

Opening and closing "file"

```
        #include <sys/types.h> <stat.h> //depends on gcc version <unistd.h> <font1.h>
```

int creat(char *pname, mode_t permissions); //create a file if not there
int open(char *fileName, int flags, mode_t permissions); // open file, returns a fd or fail(-1 and set errno)
int close(int filedes);

Flags:

O_RDONLY, O_WRONLY, O_RDWR, O_APPEND, O_CREAT
O_TRUNC O_WRONLY => truncates length to 0
O_SYNC: immediately flush writes to disk
O_NONBLOCK: non-blocking I/O

File Descriptors

Special: 0=STDIN_FILENO: stdin,
1=STDOUT_FILENO: stdout,
2=STDERR_FILENO: stderr

//All caps one are for low level lowercase are high level FILE* (from <stdio.h>)

Get fd from FILE *: int fileno(FILE *stream) //returns fd
Other way around: FILE *fdopen(int fd, const char *mode);

Read:

n = read(int fd, bufferToReadTo, #bytes) read data from a file into a buffer

```
ex)     char buf1[12], buf[12];  int fd, n1, n2, n3;  buf1[11]='\0';  buf2[11]='\0'
fd =open("foo", O_RDONLY);    //foo = "Hello John Hello Rich Hello Jay"
                    n1 = read(fd, buf1, 11); //buf1 = "Hello John" n1 = 11
                    n2 = read(fd, buf2, 11); //buf2 = "Hello Rich" n1 = 11
                    n3 = read(fd, buf1, 11); //buf1 = "Hello Jay" n3 = 9
```

Write

ssize_t write(int fd, const char *buffer, size_t nbytes) write data from buffer to file
//returns number of bytes actually written if <n usually problem, -1=error

```
ex)       #define PERM 0644 (user/owner can r/w, 4,4 group, others can r)
          char header1 [512] = "aaa", header2[512]="bbb";  int fd;  ssize_t w1, w2;
          fd = open("newfile", O_WRONLY | O_CREAT, PERM); // | means 'or'
w1= write(fd, header1, 512);   w2= write(fd, header2, 512);
```

//will overwrite any data in file if it exists (unless O_APPEND)

***if two processes open the same file and one writes to it, the reader will see the most recent write

File copy ex)

```
          void copyfile(const char *name1, const char *name2) {
int infile, outfile;
ssize_t nread;
char buffer [BUFSIZE];
infile = open (name1, O_RDONLY);
outfile = open (name2, O_WRONLY| O_TRUNC|O_CREAT, PERM);
while (nread = read (infile, buffer, BUFSIZE)) > 0)
write (outfile, buffer,nread);
close (infile);
close (outfile);
```

}

Low level Unix devices:

All devices integrated into the Unix file system: Block devices (io is fixed size blocks: disk), Character devices (io is byte streams: terminal, modems), Pseudo devices (not physical device: ssh/telnet window)

-Neat aspect of Unix devices: all io sources/sinks are treated as files, all devices mapped to fds, can be named like files “/dev/..”, can read, write to them

Terminal control -- Important to many systems programs and applications

-Terminal is a special file, terminal fd needed in order to control it: “/dev/tty”
pseudo-terminal: “/dev/pts/#”
=Suppress character echoing from the keyboard => e.g. passwords
=Disable ^C (interrupt) => e.gunsafe to stop a program that is writing files
=Process input immediately => e.g. auto command completion
=Other options: Change characters for: interrupt, erase, EOF, eol characters
Change screen dimensions
Address special locations on the screen
Fonts

```
#include <termios.h>
char termbuf [L_ctermid];
ctermid(termbuf);// controlling terminal name, “/dev/tty”
fd= open (termbuf, O_RDONLY);
```

```
#include <termios.h>
int tcgetattr(int fildes, struct termios*termios_p);
int tcsetattr(int fildes, intoptional_actions, const struct termios*termios_p);
```

Optional_actions: when changes take affect, TCSAFLUSH

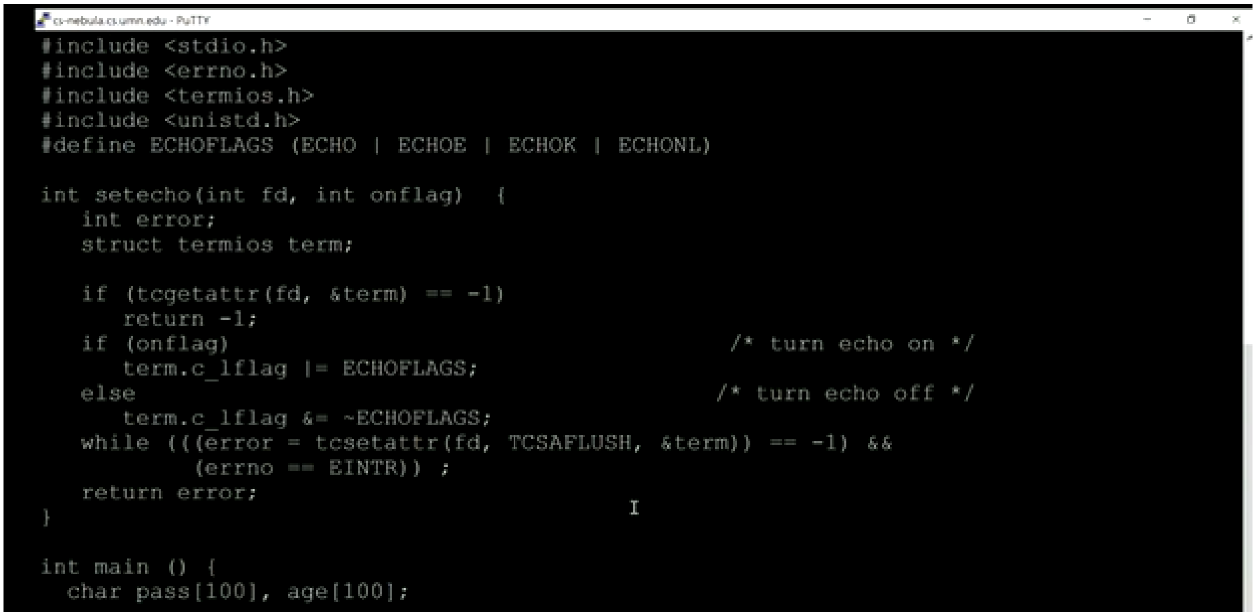
Step 1: get attributes, 2: modify them 3: set them

```
struct termios:
    tcflag_t c_iflag; // input (e.g. CR)
    tcflag_t c_oflag; // output (newline=CR)
    tcflag_t c_cflag; // control h/w
    tcflag_t c_lflag;   // local/editing (e.g. ECHO)
    cc_t c_cc[NCCS]; // control chars (e.g. intris ^g)
```

Terminal is canonical by default (line oriented input- erase and backspace have meaning, waits for input to be entered before reading)

What if instead you only want to wait a little bit, or want to auto fill in after 2 characters etc

Disable echoing:

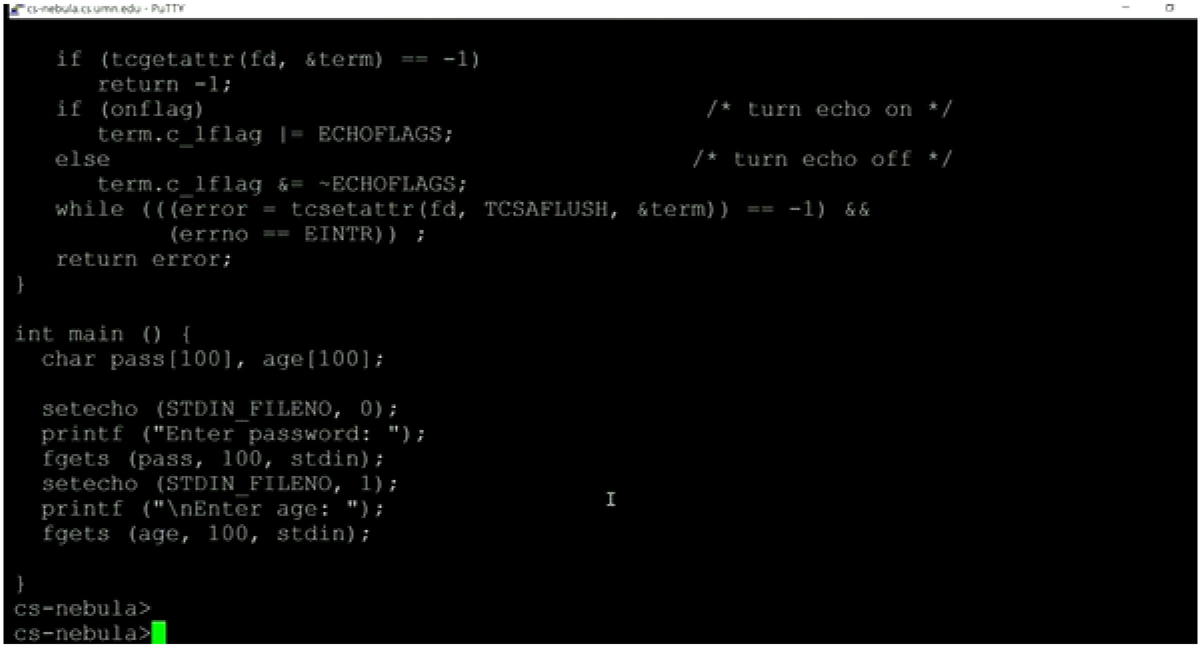


```
#include <stdio.h>
#include <errno.h>
#include <termios.h>
#include <unistd.h>
#define ECHOFLAGS (ECHO | ECHOE | ECHOK | ECHONL)

int setecho(int fd, int onflag)  {
   int error;
   struct termios term;

   if (tcgetattr(fd, &term) == -1)
      return -1;
   if (onflag)                                  /* turn echo on */
      term.c_lflag |= ECHOFLAGS;
   else                                        /* turn echo off */
      term.c_lflag &= ~ECHOFLAGS;
   while (((error = tcsetattr(fd, TCSAFLUSH, &term)) == -1) &&
           (errno == EINTR)) ;
   return error;
}

int main () {
  char pass[100], age[100];
```



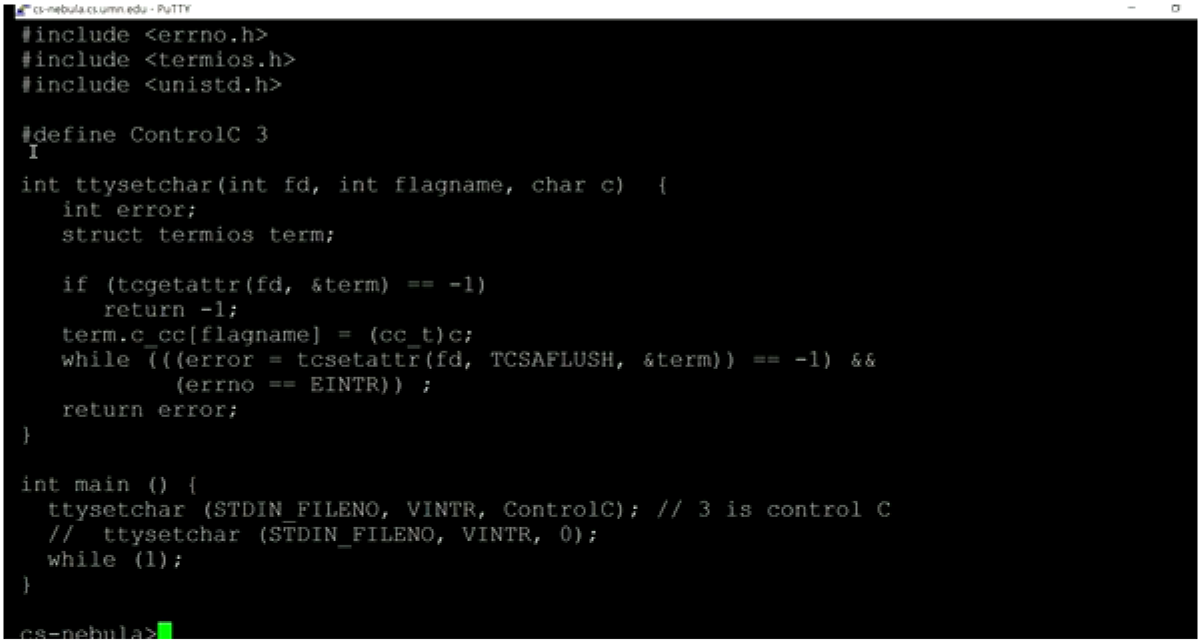
```
   if (tcgetattr(fd, &term) == -1)
      return -1;
   if (onflag)                                  /* turn echo on */
      term.c_lflag |= ECHOFLAGS;
   else                                        /* turn echo off */
      term.c_lflag &= ~ECHOFLAGS;
   while (((error = tcsetattr(fd, TCSAFLUSH, &term)) == -1) &&
           (errno == EINTR)) ;
   return error;
}

int main () {
  char pass[100], age[100];

  setecho (STDIN_FILENO, 0);
  printf ("Enter password: ");
  fgets (pass, 100, stdin);
  setecho (STDIN_FILENO, 1);
  printf ("\nEnter age: ");
  fgets (age, 100, stdin);

}
cs-nebula>
cs-nebula>
```

Disable control c



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

#define ControlC 3

int ttysetchar(int fd, int flagname, char c)  {
   int error;
   struct termios term;

   if (tcgetattr(fd, &term) == -1)
      return -1;
   term.c_cc[flagname] = (cc_t)c;
   while (((error = tcsetattr(fd, TCSAFLUSH, &term)) == -1) &&
           (errno == EINTR)) ;
   return error;
}

int main () {
  ttysetchar (STDIN_FILENO, VINTR, ControlC); // 3 is control C
  //  ttysetchar (STDIN_FILENO, VINTR, 0);
  while (1);
}

cs-nebula>
```

OS Buffers writes

Use O_SYNC flag on open in write mode or at some point: int fsync(int fd);

File management system calls:

Fd = open(file, how) open a file for reading, writing or both

#include <studio.h> FILE *f; F = fopen(/user/barre684/file.cc”. “r”);

S = close(fd) close an open file

position = lseek(fd, offset, wherePrevWas) move the file pointer

s = stat(name, &buf) Get a file’s status information

When path is a symbolic link, the lstat function returns information about the link whereas the stat function returns information about the file referred to by the link. (same way to call lstat as stat)

Redirection:

In terminal: cat foo.bar > baz.out (Writes the contents of foo.bar to baz.out, overwrites the prev contents of baz.out, instead of terminal)

Do same thing w dup2()

fd = open(“baz.out”, O_CREAT | O_WRONLY, …);
//NEED TO SAVE PTR TO 1 BEFORE DUP
dup2(fd, 1); //close stdout and 1 now refers to fd
close(fd); //not needed but good style
write(1, …); //writes to stdout now go to baz.out

Applications that benefit from terminal control:

Disable ^C (interrupt), suppress character echoing from the keyboard, and process input immediately.

Input Redirection:

In terminal: wc -c < mydata.txt

Do same thing w dup2():

```
fd = open(“mydata.txt”, O_RDONLY, …);
//NEED TO SAVE PTR TO 1 BEFORE DUP
dup2(fd, 0); //close stdin and 0 now refers to fd
close(fd); //not needed but good style
read(0, …); //reads from standard in are now directed to mydata.txt
```

Duplicating File Descriptors:

```
#include <uistd.h>
int dup2 (int fd1, int fd2);
```

//closes fd2 if open->frees up fd2, makes fd2 now point to what fd1 pts to

File as abstraction:

Container for related info, named, associated attributes

Destroy hard and soft links: unlink();

Hard Links: files with mult names:

Each name is an alias

```
#include <unistd.h>
int link(const char *original_path, const char * new_path) //new path can’t be file already
link(“foo”, “bar”); //bar refers to file foo
unlink(“bar”); //remove name bar
//if file is opened by someone it will not actually be deleted until all fds to it are closed
//hard links cannot be made to directories or to files in other file systems
```

Hard links all have the same inode

Symbolic Links:

Allows a file/dir name to “point to” another file/dir name

```
int symlink(const char *realname, const char *symname);
        symlink(“/usr/jon/tmp1”, “/usr/bill/tmp2”);
```

An inode is created for symname

Tricky part:

If you remove the linked name the symbolic link goes away but the file doesn’t, but if you remove the file that the symbolic link is linked to this will get rid of the file but the link will remain and cause errors when you try to access it.

Access to metadata

```
#include <sys/stat.h>
int fstat(int filedes, struct stat *buf); and int stat(const char *pathname, struct stat *buf);
```

Structure contains file info:

```
off_t st_size; // file size
nlink_t st_nlink //links
mode_t st_mode;  //type+permission
time_t st_mtime;   //last modification time
```

fcntl can be used to se ot get lower-level attrs

```
struct_t st;   stat(“foo”, &st);
```

Macros:

```
int S_ISDIR(st.st_mode);
```

int S_ISREG(st.st_mode);
int S_ISSOCK(st.st_mode);

Unix inode:

Add stuff from notes

Filesystem:

Directory is a file as well

Has an inode

File contents: list of file_name, inode pairs

Filesystem: files, directories, free disk sectors, root dir

On-disk organization

Inode for root dir of filesystem “/” stored in well-known sector on the disk

inode for disk sector free-list also stored in a well-known sector on the disk

Stored in the superblock

Unix file types/modes: indicated by the first character in ls -l

- regular file, d directory, c character special file b block special file p pipe s socket l symbolic link

```
ls -l ex) drwx-xr-x 3  jon fac 4066 Nov 2 09:14 st
   u^  g^  ^o d=file type, 3=#hard links, 4066= allocation size
```

File Permissions:

r read, w write, x execute

Subjects: u user/owner, g group, o others --user may belong to any number of groups

chmod 0077 st.txt (command line)

int chmod(char*path, mode_t mode);

IDS:

Real user-id: user that initiated a process, not executable owner

Effective user-id: user that system associates with tht process for purposes of protection

Usually same as real user-id

Sometimes want E user-Id to be file owner not user: chmod u+s my_file
chmod g+s *has privilege of group

Masks:

ADD

mmap()

#include <sys.mman.h>

void *mmap(void * start (0), size_t length (how much to map), int prot (access type), int flags, int fd (the file), off_t offset (starting where)

Returns mem address of file data, can use normal mem operations to read/write

Filedes = open(...)

Address = mmap(0, len, MAP_PRIVATE, filedes, some_offset)

Directory

Abstraction: container for related files & other dir, name, location, contents, attributes

Directory Permissions: read: class of users can list ‘ls’ contents of dir
Write: users can create or remove files in dir
Execute: users can ‘cd’ into dir and open and execute files in dir

Create:

```
#include <sys/stat.h>
    int mkdir(const char *pathname, mode_t mode); //also puts 2 links (. and .. in dir)
        mkdir(“tmp/dir1”, 0777);
```

Remove:

```
    int rmdir(const char *pathname); //directory must be empty
```

From shell: rmdir csci4061

Open dir and look at contents:

```
    #include <dirent.h>
    DIR *opendir(const char *dirname);
    struct dirent *readdir (DIR *dirptr);  //returns each directory, NULL at the end
    int closedir(DIR *dirptr);

DIR *dp;
    dp = opendir(“/tmp/dir1”);

    struct dirent{
        ino_t d_ino;
char d_name[NAMESIZE];
}
```

Readdir ex)

```
    int my_ls(const char *name){
        struct dirtent *d;
        DIR *d;
        if ((dp = opendir(name)) == NULL){ return -1}
        while (d= readdir(dp)){ printf(“%s\n”, d->d_name); }
        closedir(dp);
        return 1; }
```

Path names:

Home directory: (~)

Current working directory (cwd) pwd

```
int chdir(const char *path);
char *getcwd(char *name, size_t size);
```

Pipes:

Most basic form of IPC (inter process communication)

```
ps -u weiss039 | grep tch
```

FIFO communication: first in first out

Bi-directional

Communication between a parent and a child or related processes bc need to share fds

```
#include <unistd.h>
int pipe(int ends[2]); //returns -1 on fail
```

ends = ends[0]:read end (receive) ends[1]: write end (send)

```
#include <unistd.h>  #include <stdio.h>  #define MSGSIZE 16
char *msg1 = “hello, world #1”;
```

```
    char *msg2 = “hello, world #2”;
    void main(){
        char inbuf [MSGSIZE];
        int ends[2];
        if (pipe(ends) == -1){
            perror(“pipe error”);
            exit(1);
        }
        //write (send) down pipe
        write(ends[1], msg1, MSGSIZE);
        write(ends[1], msg2, MSGSIZE);
        //read (recieve) from pipe
        read(ends[0], inbuf, MSGSIZE);
        fprintf(stderr, “%s\n”, inbuf);
        read(ends[0], inbuf, MSGSIZE);
        fprintf(stderr, “%s\n”, inbuf);
    }  //output is hello, world #1 \n hello, world #2 \n
```

Read may not get everything but it “usually does” up to max (MSGSIZE and pipe contents)

----blocks if pipe is empty

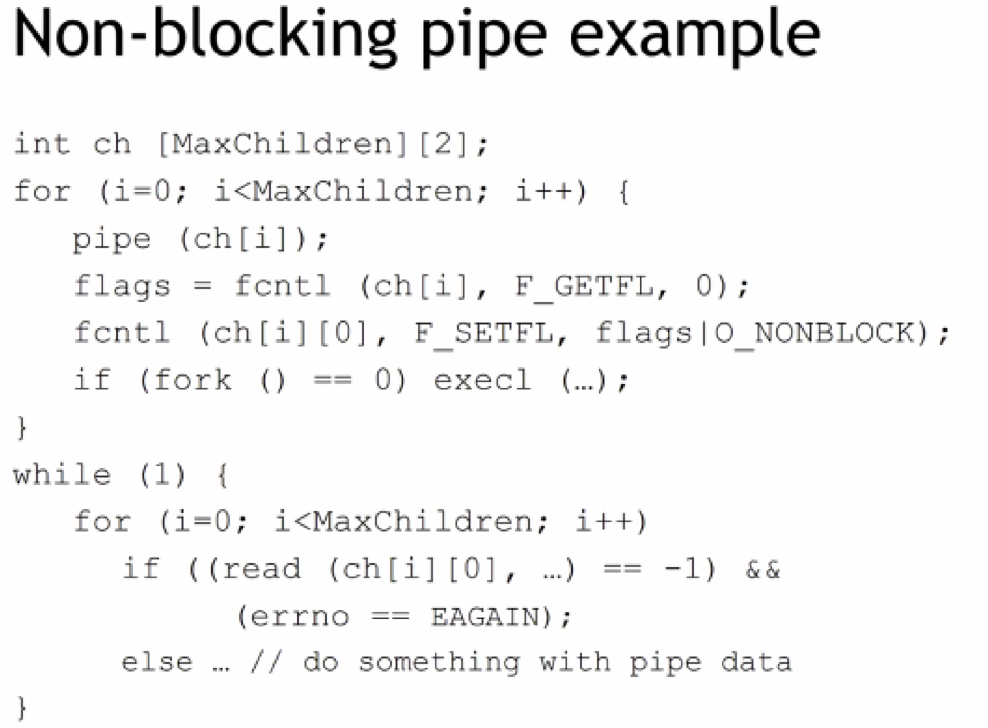
Pipes have finite size (4k/8k) write blocks if not enough space

```
pid= fork ();
if (pid== 0) {
// child sends into pipe
write (ends[1], msg1, MSGSIZE);
write (ends[1], msg2, MSGSIZE);
}
else if (pid> 0) {
// parent receives from pipe
read (ends[0],inbuf, MSGSIZE);
fprintf(stderr, “%s\n”,inbuf);
read (ends[0],inbuf, MSGSIZE);
fprintf(stderr, “%s\n”,inbuf);
wait (NULL);
}
```

Better:

```
if (pid== 0) {// child sends into pipe
close (ends[0]);
write (ends[1], msg1, MSGSIZE);
write (ends[1], msg2, MSGSIZE);
}
else if (pid> 0) { // parent receives from pipe
close (ends[1]);
read (ends[0],inbuf, MSGSIZE);
fprintf(stderr, “%s\n”,inbuf);
```

read (ends[0],inbuf, MSGSIZE);
fprintf(stderr, “%s\n”,inbuf);
wait (NULL);
}
Good bc it is a near infinite stream of data from producer to consumer, consumer(reader) needs to keep up with the producer (writer) and vice versa
If write-end is closed: (no processes have pipe open for write) and the pipe is empty:
    read returns 0
If read-end is closed:
    Write will kill the process: “broken pipe”
If you need two way communication will need a pair of pipes
Non-blocking pipes:



Non-blocking pipe example

```
int ch [MaxChildren][2];
for (i=0; i<MaxChildren; i++) {
   pipe (ch[i]);
   flags = fcntl (ch[i], F_GETFL, 0);
   fcntl (ch[i][0], F_SETFL, flags|O_NONBLOCK);
   if (fork () == 0) execl (…);
}
while (1) {
   for (i=0; i<MaxChildren; i++)
      if ((read (ch[i][0], …) == -1) &&
            (errno == EAGAIN);
      else … // do something with pipe data
}
```

    #include <fcntl.h>
int fcntl(int fd,int cmd, ...);
int ends[2], flags,nread;
pipe (ends);
flags =fcntl(fd, F_GETFL, 0);
fcntl(ends[0], F_SETFL, flags | O_NONBLOCK);
...
nread= read (ends[0],buf, size);
// if nothing to read, returns -1, errno set to EAGAIN
Sending Discrete “Data”
    Typedef struct{
int x;
Int y;

```
char str[20];
} message_t;
message_t m1, m2;
int ends[2]; // send m1 into the pipe
write (ends[1], &m1, sizeof(message_t));  // pull data into m2 from the pipe
read (ends[0], &m2, sizeof(message_t));
```