1.3 A Simple OS

- OS Structure
- Processes, Address Spaces, & Threads
- Managing Processes
- Loading Program Into Processes
- *Files*
  - (first 13 slides overlap with "A Simple OS" slides)

Files

- Our primes program wasn't too interesting
  - it has no output!
  - cannot even verify that it's doing the right thing
  - other program cannot use its result
  - how does a process write to someplace outside the process?
- The notion of a *file* is our Unix system's sole abstraction for this concept of "someplace outside the process"
  - modern Unix systems have additional abstractions
- Files
  - abstraction of persistent data storage
  - means for fetching and storing data outside a process
    - including disks, another process, keyboard, display, etc.
    - need to *name* these different places
      - hierarchical naming structure
    - part of a process's *extended address space*

Naming Files

- Directory system
  - shared by all processes running on a computer
    - although each process can have a different view
    - Unix provides a means to restrict a process to a subtree
      - by redefining what "root" means for the process
  - name space is outside the processes
    - a user process provides the name of a file to the OS
    - the OS returns a *handle* to be used to access the file
      - after it has verified that the process is allowed *access* along the *entire path*, starting from root
    - user process uses the handle to read/write the file
      - avoid access checks
- Using a handle to refer to an object managed by the kernel is an important concept
  - handles are essentially an *extension* to the process's *address space*
    - can even survive execs!

The File Abstraction

- A file is a simple array of bytes
- Files are made larger by writing beyond their current end
- Files are named by paths in a naming tree
- System calls on files are *synchronous*
- File API
  - `open()`, `read()`, `write()`, `close()`
  - e.g., `cat`

File Handles (File Descriptors)

```
int fd;
char buffer[1024];
int count;
if ((fd = open("/home/bc/file", O_RDWR) == -1) {
  // the file couldn't be opened
  perror("/home/bc/file");
  exit(1);
}
if ((count = read(fd, buffer, 1024)) == -1) {
  // the read failed
  perror("read");
  exit(1);
}
// buffer now contains count bytes read from the file
```

- what is O_RDWR?
- what does `perror()` do?
- *cursor* position in an opened file depends on what functions/system calls you use
  - what about C++?

Standard File Descriptors

- Standard File Descriptors
  - 0 is stdin (by default, the keyboard)
  - 1 is stdout (by default, the display)
  - 2 is stderr (by default, the display)

```
main() {
  char buf[BUFSIZE];
  int n;
  const char *note = "Write failed\n";

  while ((n = read(0, buf, sizeof(buf))) > 0)
    if (write(1, buf, n) != n) {
      (void)write(2, note, strlen(note));
      exit(EXIT_FAILURE);
    }
  return(EXIT_SUCCESS);
}
```

Back to Primes

Have our primes program write out the solution, i.e., the `primes[]` array

```
int nprimes;
int *prime;
int main(int argc, char *argv[]) {
  ...
  for (i=1; i<nprimes; i++) {
    ...
  }
  if (write(1, prime, nprimes*sizeof(int)) == -1) {
    perror("primes output");
    exit(1);
  }
  return(0);
}
```

- the output is not readable by human

Human-Readable Output

```
int nprimes;
int *prime;
int main(int argc, char *argv[]) {
  ...
  for (i=1; i<nprimes; i++) {
    ...
  }
  for (i=0; i<nprimes; i++) {
    printf("%d\n", prime[i]);
  }
  return(0);
}
```

Allocation of File Descriptors

Whenever a process requests a new file descriptor, the lowest numbered file descriptor not already associated with an open file is selected; thus

```
#include <fcntl.h>
#include <unistd.h>
...
close(0);
fd = open("file", O_RDONLY);
```

- will always associate "file" with file descriptor 0 (assuming that the open succeeds)

Running It

```
if (fork() == 0) {
  /* set up file descriptor 1 in the child process */
  close(1);
  if (open("/home/bc/Output", O_WRONLY) == -1) {
    perror("/home/bc/Output");
    exit(1);
  }
  execl("/home/bc/bin/primes", "primes", "300", 0);
  exit(1);
}
/* parent continues here */
while(pid != wait(0)) /* ignore the return code */
  ;
```

- `close(1)` removes file descriptor 1 from *extended address space*
- file descriptors are allocated *lowest first* on `open()`
- *extended address space* survives *execs*
- new code is same as running

```
% primes 300 > /home/bc/Output
```

I/O Redirection

```
% primes 300 > /home/bc/Output
```

The ">" parameter in a shell command that instructs the command shell to *redirect* the output to the given file

- If ">" weren't there, the output would go to the display

Can also redirect input

```
% cat < /home/bc/Output
```

- when the "`cat`" program reads from file descriptor 0, it would get the data byes from the file "/home/bc/Output"

File-Descriptor Table

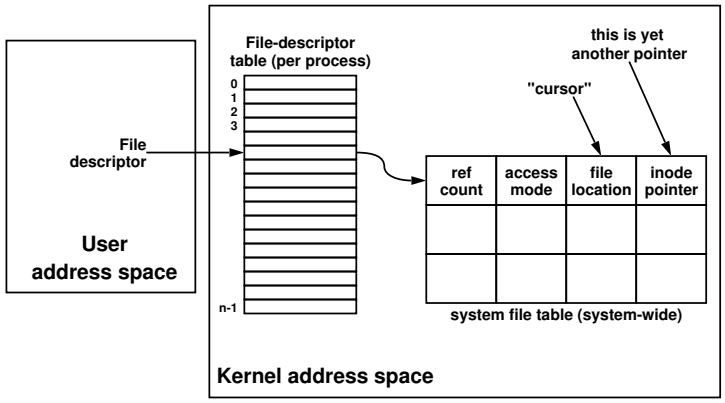
A file descriptor refers not just to a file

- it also refers to the *process's* current *context* for that file
  - includes how the file is to be accesses (how `open()` was invoked)
  - *cursor* position

Context information must be maintained by the OS and not directly by the user program

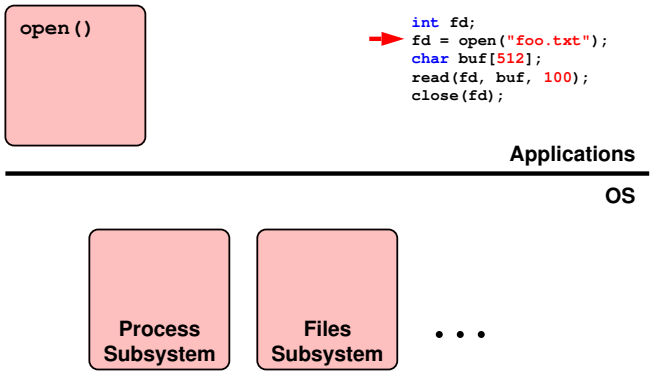
- let's say a user program opened a file with O_RDONLY
- later on it calls `write()` using the opened file descriptor
- how does the OS knows that it doesn't have write access?
  - stores O_RDONLY in context
- if the user program can manipulate the context, it can change O_RDONLY to O_RDWR
- therefore, user program must not have access to context!
  - all it can see is the handle
  - the handle is an *index* into an array maintained for the process in kernel's address space

File-Descriptor Table

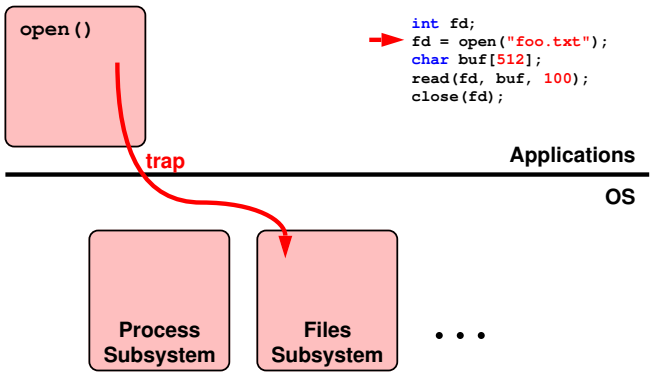


- context is not stored directly into the file-descriptor table
  - one-level of *indirection*

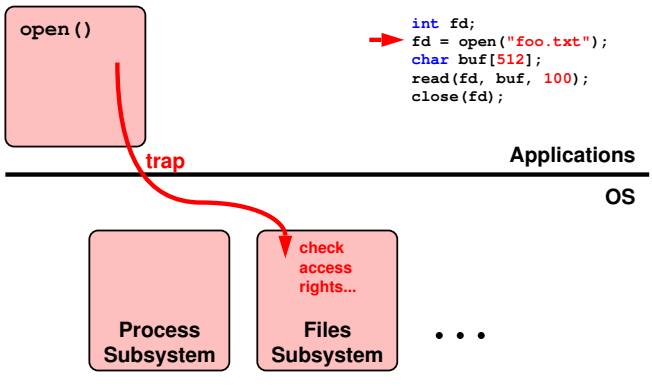
Put It All Together



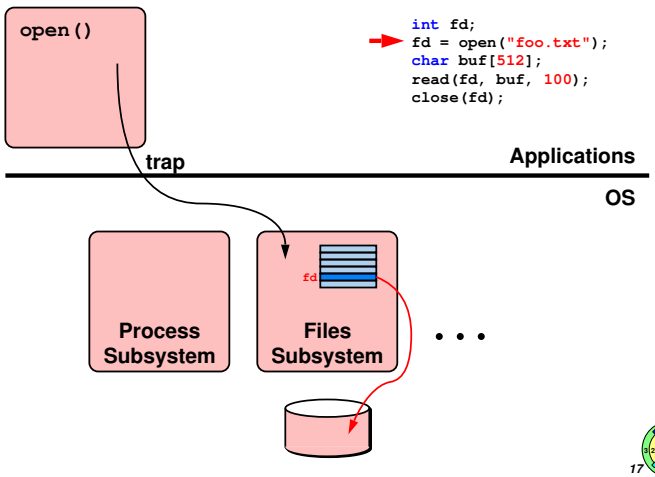
Put It All Together



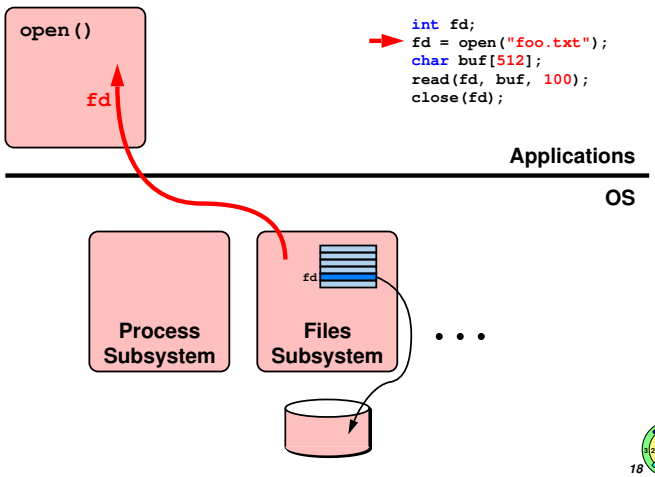
Put It All Together



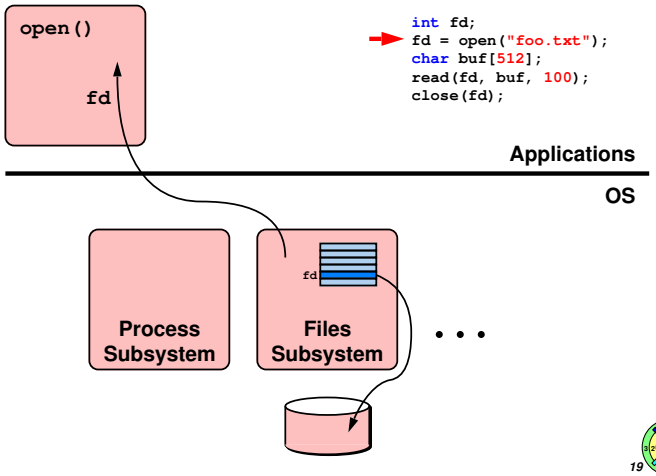
Put It All Together



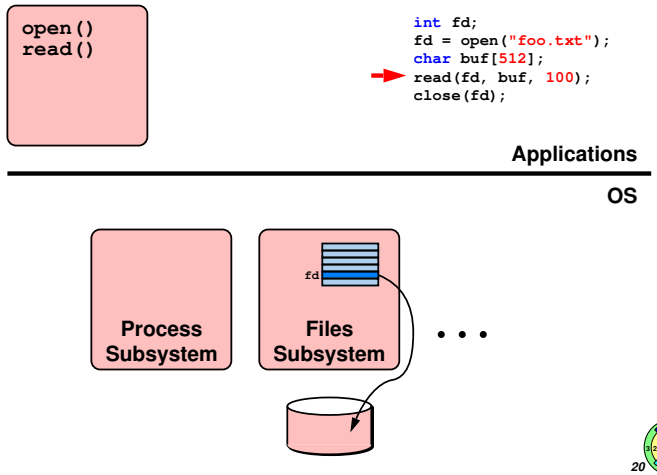
Put It All Together



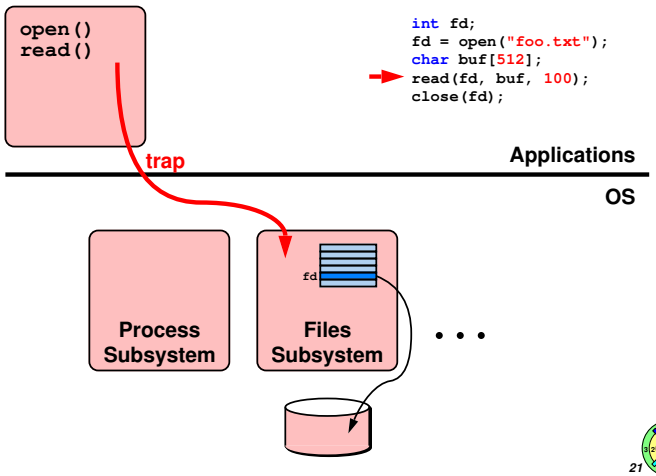
Put It All Together



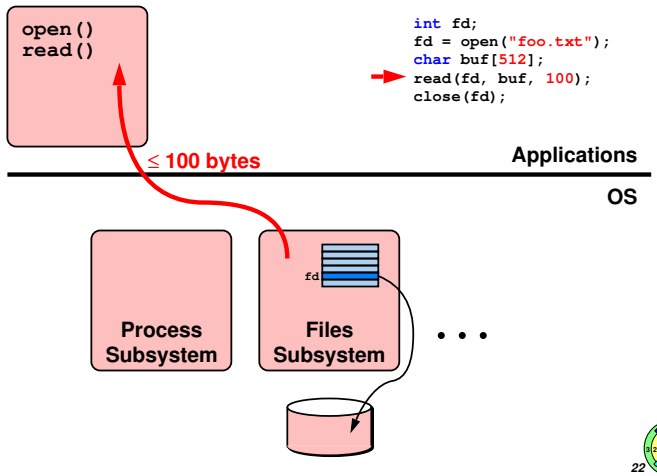
Put It All Together



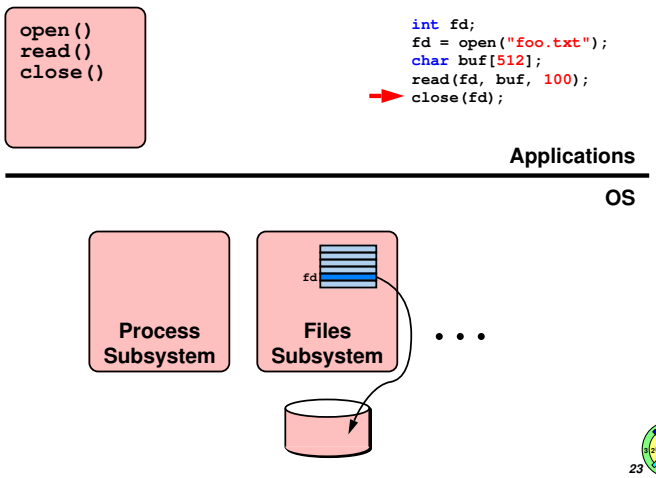
Put It All Together



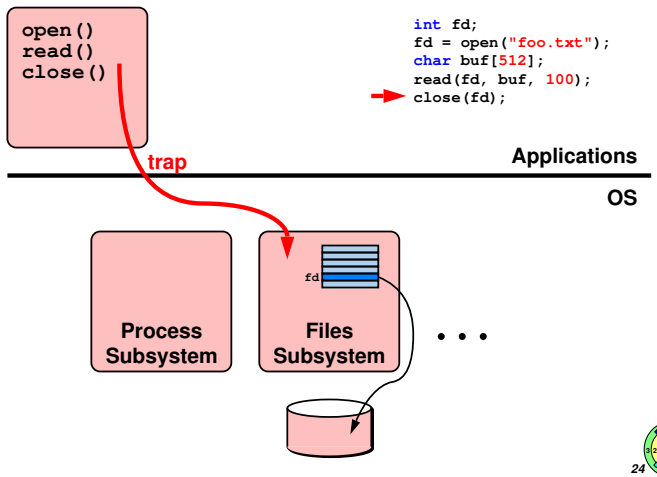
Put It All Together



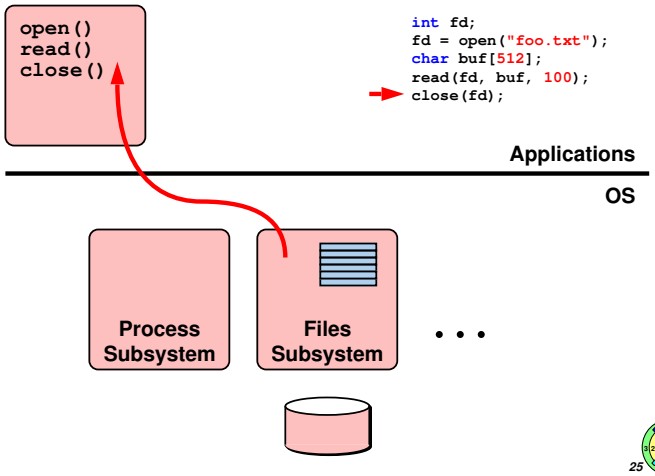
Put It All Together



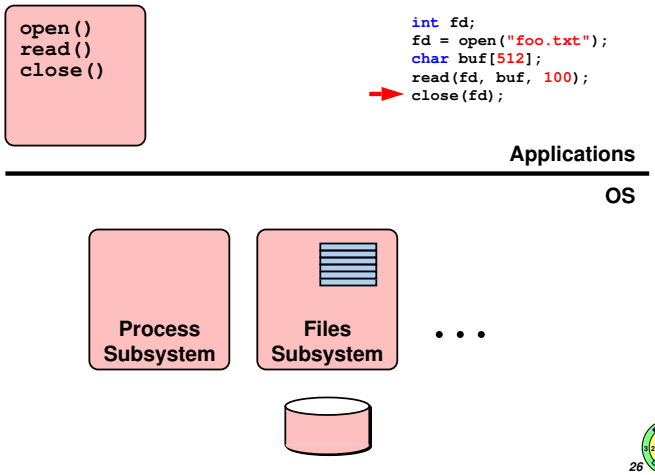
Put It All Together



Put It All Together



Put It All Together



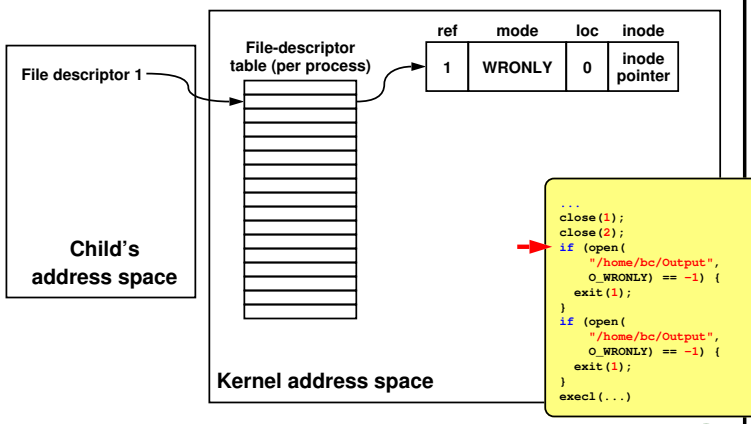
Redirecting Output ... Twice

- Every calls to `open()` creates a new entry in the system file table

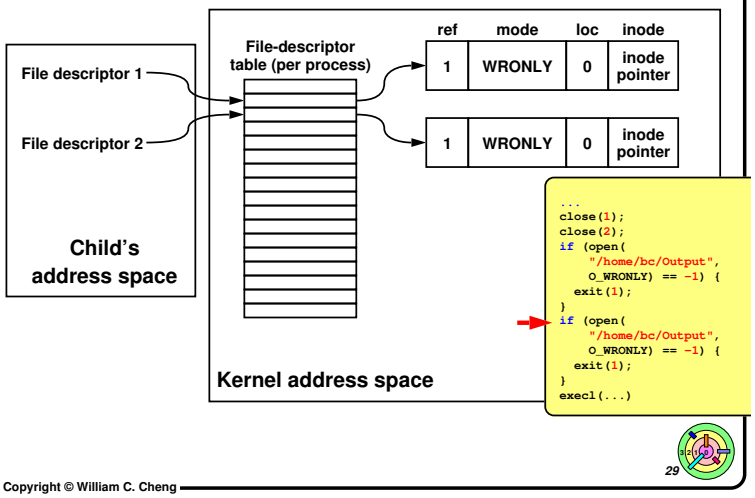
```
if (fork() == 0) {
  /* set up file descriptors 1 and 2 in the child
     process */
  close(1);
  close(2);
  if (open("/home/bc/Output", O_WRONLY) == -1) {
    exit(1);
  }
  if (open("/home/bc/Output", O_WRONLY) == -1) {
    exit(1);
  }
  execl("/home/bc/bin/program", "program", 0);
  exit(1);
}
/* parent continues here */
```

- `stdout` and `stderr` both go into the same file
  - would it cause any problem?

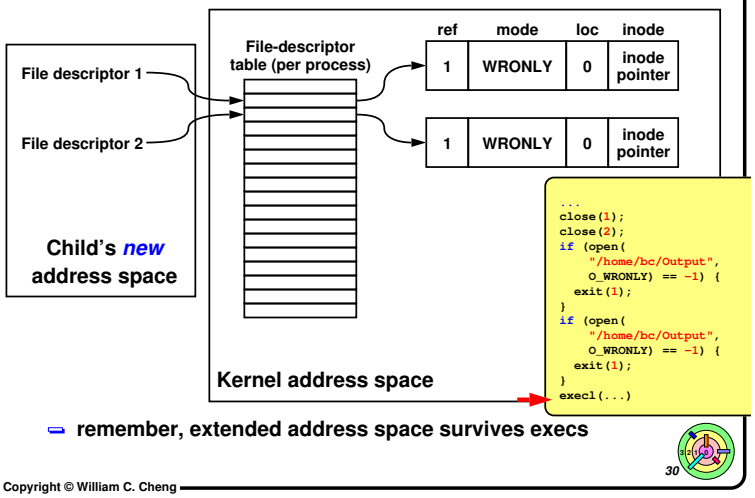
Redirected Output



Redirected Output

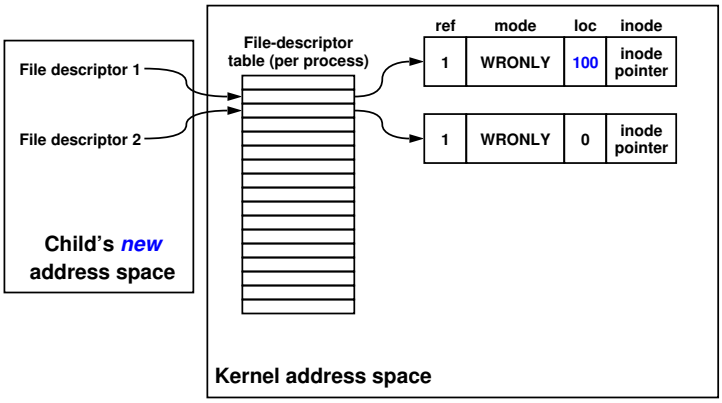


Redirected Output



- remember, extended address space survives execs

Redirected Output After Writing 100 Bytes



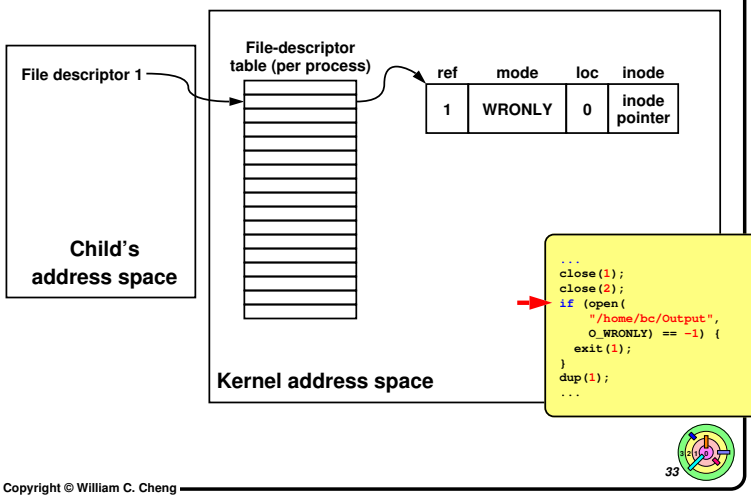
- `write()` to fd=2 will wipe out data in the first 100 bytes
  - that may not be the intent

Sharing Context Information

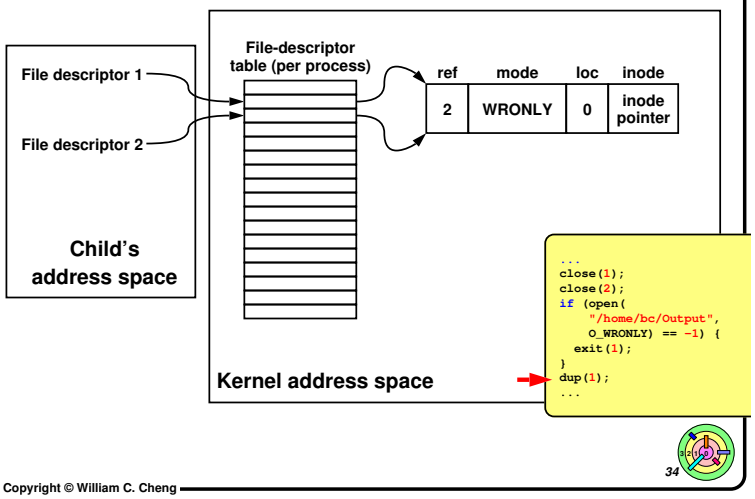
```
if (fork() == 0) {
  /* set up file descriptors 1 and 2 in the child
     process */
  close(1);
  close(2);
  if (open("/home/bc/Output", O_WRONLY) == -1) {
    exit(1);
  }
  dup(1);
  execl("/home/bc/bin/program", "program", 0);
  exit(1);
}
/* parent continues here */
```

- use the `dup()` system call to *share* context information
  - if that's what you want

Redirected Output After Dup



Redirected Output After Dup



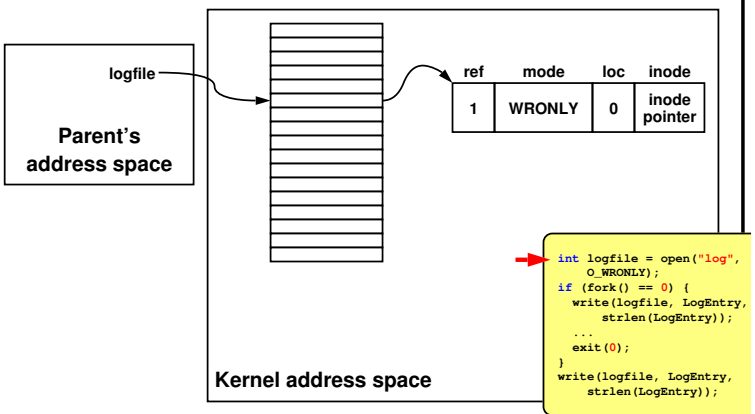
Fork and File Descriptors

- When `fork()` is called, the child process gets a *copy* of the parent's file descriptor table

```
int logfile = open("log", O_WRONLY);
if (fork() == 0) {
  /* child process computes something, then does: */
  write(logfile, LogEntry, strlen(LogEntry));
  ...
  exit(0);
}
/* parent process computes something, then does: */
write(logfile, LogEntry, strlen(LogEntry));
...
```

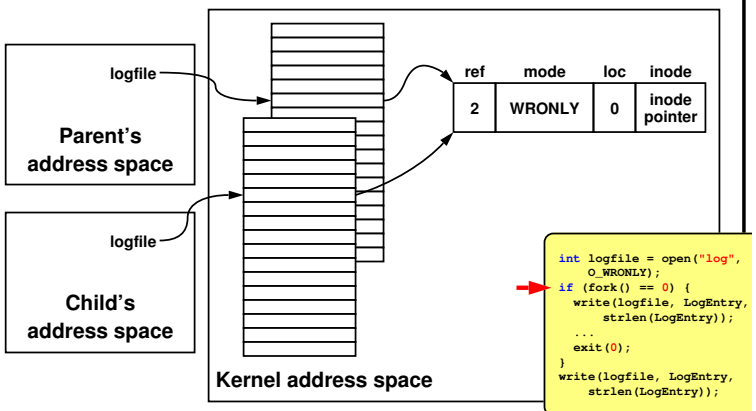
- remember, extended address space survives execs
  - also `fork()`

File Descriptors After Fork



- parent and child processes get *separate* file descriptor table but *share* extended address space

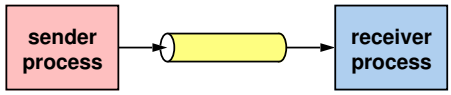
File Descriptors After Fork



- parent and child processes get *separate* file descriptor table but *share* extended address space

Pipes

- A pipe is a means for one process to send data to another directly, as if it were writing to a file

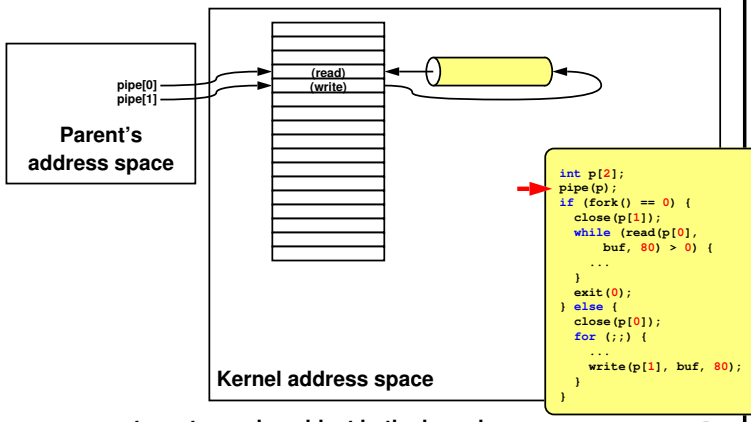


  - the sending process behaves as if it has a file descriptor to a file that has been opened for writing
  - the receiving process behaves as if it has a file descriptor to a file that has been opened for reading
- The `pipe()` system call creates a pipe object in the kernel and returns (via an output parameter) the two file descriptors that refer to the pipe
  - one, set for write-only, refers to the input side
  - the other, set for read-only, refers to the output side
  - a pipe has no name, cannot be passed to another process

Pipes

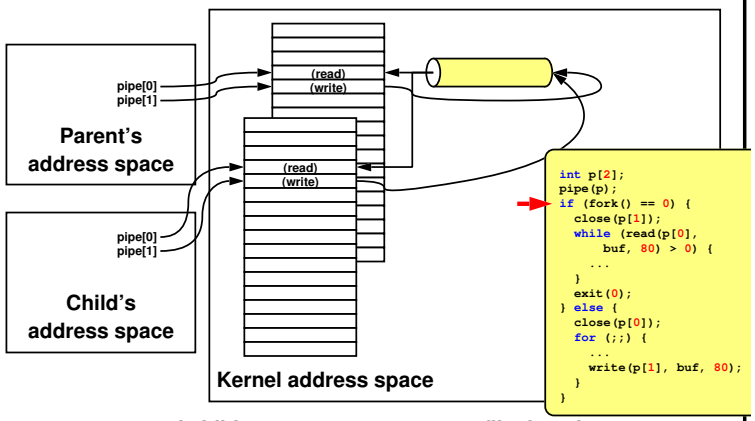
```
int p[2]; // array to hold pipe's file descriptors
pipe(p);  // creates a pipe, assume no errors
  // p[0] refers to the read/output end of the pipe
  // p[1] refers to the write/input end of the pipe
if (fork() == 0) {
  char buf[80];
  close(p[1]); // not needed by the child
  while (read(p[0], buf, 80) > 0) {
    // use data obtained from parent
    ...
  }
  exit(0); // child done
} else {
  char buf[80];
  close(p[0]); // not needed by the parent
  for (;;) {
    // prepare data for child
    ...
    write(p[1], buf, 80);
  }
}
```


Pipes



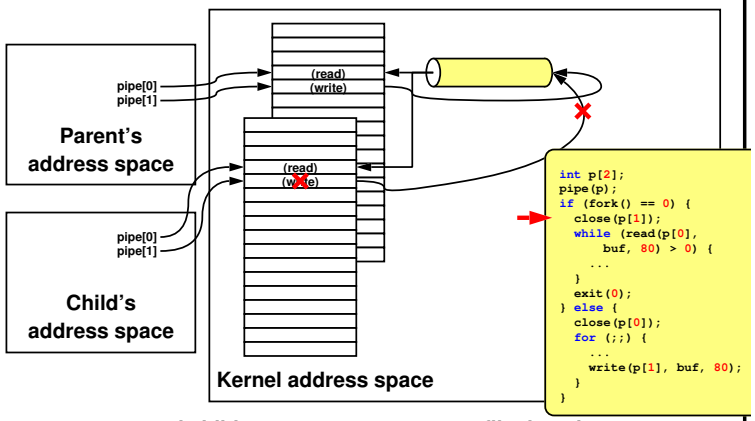
- parent creates a pipe object in the kernel

Pipes



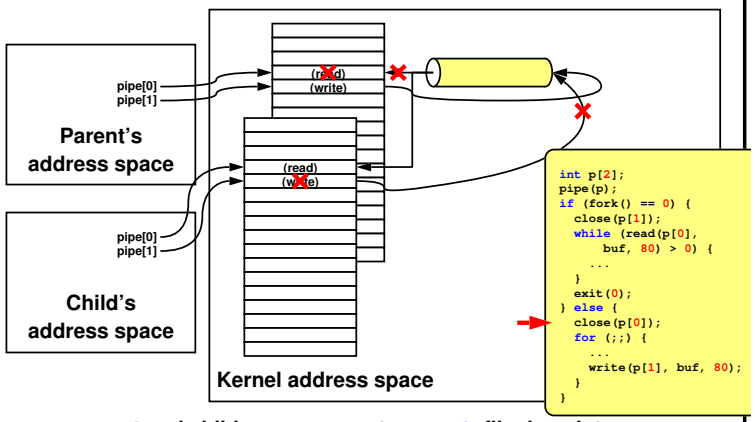
- parent and child processes get *separate* file descriptor table but *share* extended address space

Pipes



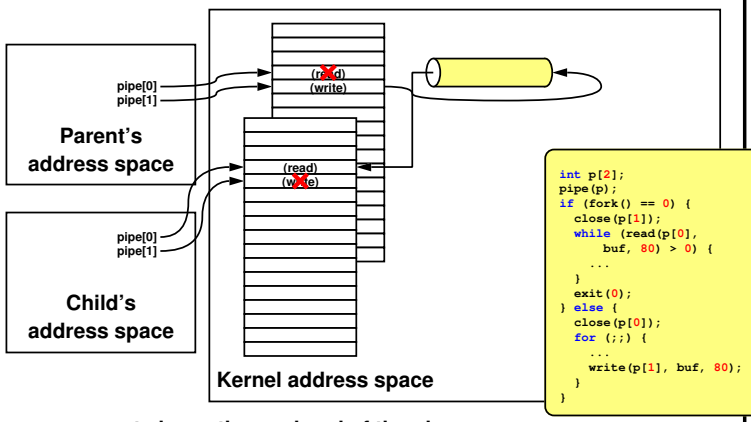
- parent and child processes get *separate* file descriptor table but *share* extended address space

Pipes



- parent and child processes get *separate* file descriptor table but *share* extended address space

Pipes



- parent closes the read-end of the pipe
  child closes the write-end of the pipe

Command Shell

- Now you know enough to write a command shell
  - execute a command
  - redirect I/O
  - pipe the output of one program to another

    ```
    cat f0 | ./warmup1 sort
    ```

    - the shell needs to create a pipe
    - create two child processes
    - in the first child
      - have `stdout` go to the write-end of the pipe
      - close the read-end of the pipe
      - exec "`cat f0`"
    - in the 2nd child
      - have `stdin` come from the read-end of the pipe
      - close the write-end of the pipe
      - exec "`./warmup1 sort`"
  - run a program in the background

    ```
    primes 1000000 > primes.out &
    ```


Random Access

```
fd = open("textfile", O_RDONLY);
// go to last char in file
fptr = lseek(fd, (off_t)(-1), SEEK_END);
while (fptr != -1) {
  read(fd, buf, 1);
  write(1, buf, 1);
  fptr = lseek(fd, (off_t)(-2), SEEK_CUR);
}
```

- "man lseek" gives

  ```
  off_t lseek(int fd, off_t offset, int whence);
  ```
- `whence` can be SEEK_SET, SEEK_CUR, SEEK_END
- if succeeds, returns cursor position (always measured from the beginning of the file)
  - otherwise, returns (-1)
  - `errno` is set to indicate the error
- `read(fd,buf,1)` advances the cursor position by 1, so we need to move the cursor position back 2 positions

More On Naming

- (Almost) everything has a path name
  - files
  - directories
  - devices (known as *special files*)
  - keyboards
  - displays
  - disks
  - etc.
- Uniformity

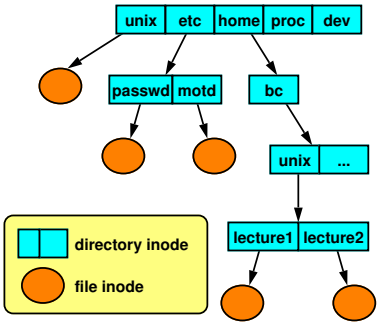
```
// opening a normal file
int file = open("/home/bc/data", O_RDWR);

// opening a device (one's terminal or window)
int device = open("/dev/tty", O_RDWR);

int bytes = read(file, buffer, sizeof(buffer));
write(device, buffer, bytes);
```


Directories

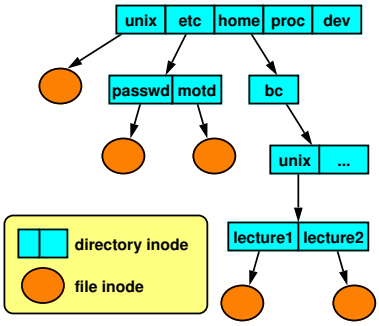
- A *directory* is a *file*
  - interprets differently by the OS as containing *references* to other files/directories
  - a file is represented as an *index node* (or *inode*) in the file system



- A *directory* maps a *file name* to an *inode number*
  - maps a string to an integer
  - done inside Virtual File System in `weenix`
- An *inode* maps an *inode number* to *sectors on disk*
  - done inside Actual File System in `weenix`

Directory Representation

- A root directory entry example
  - parent inode number = its own inode number



Component Name	Inode number
.	1
..	1
unix	117
etc	4
home	18
proc	36
dev	93

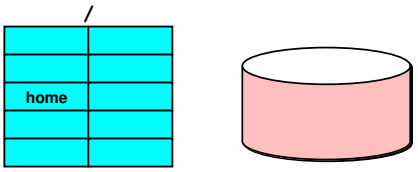
directory entry

this is what a S5FS directory looks like in weenix

- Tree structured hierarchy

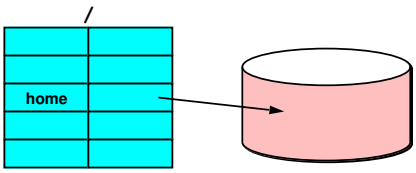
Look Up Inode Number Of A Path

- Ex: how do figure out the inode number of "`/home/bc/foo.c`"?



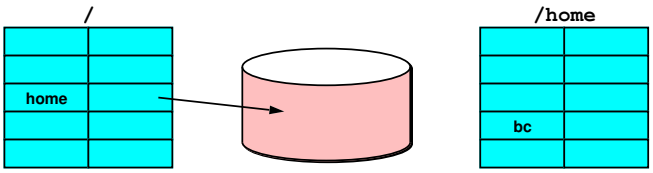
Look Up Inode Number Of A Path

- Ex: how do figure out the inode number of "`/home/bc/foo.c`"?



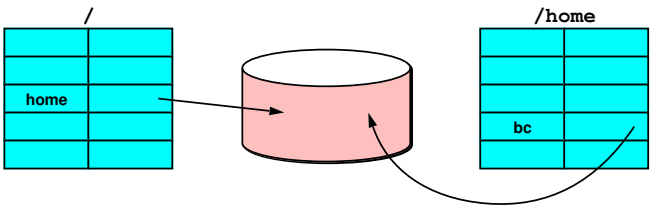
Look Up Inode Number Of A Path

- Ex: how do figure out the inode number of "`/home/bc/foo.c`"?



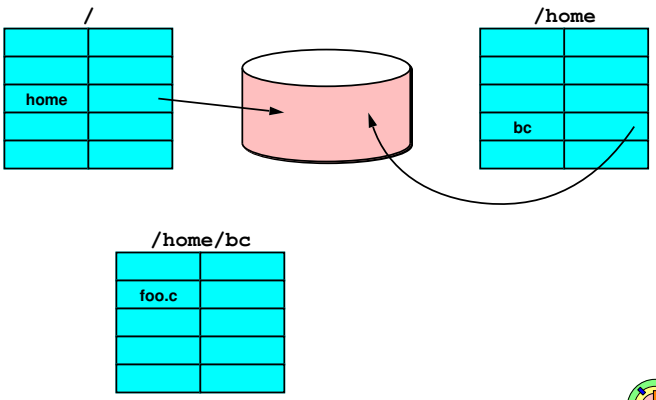
Look Up Inode Number Of A Path

- Ex: how do figure out the inode number of "`/home/bc/foo.c`"?



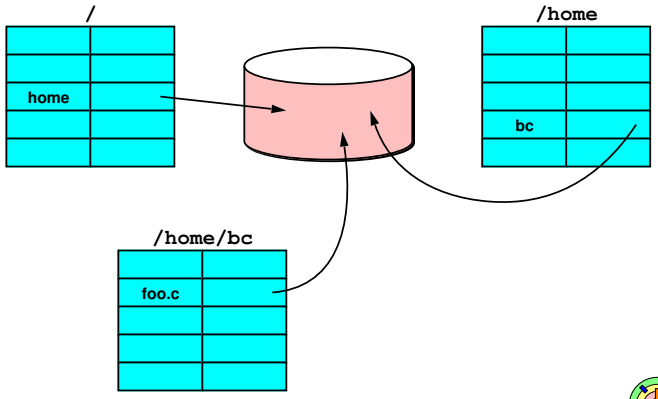
Look Up Inode Number Of A Path

- Ex: how do figure out the inode number of "`/home/bc/foo.c`"?



Look Up Inode Number Of A Path

- Ex: how do figure out the inode number of "`/home/bc/foo.c`"?

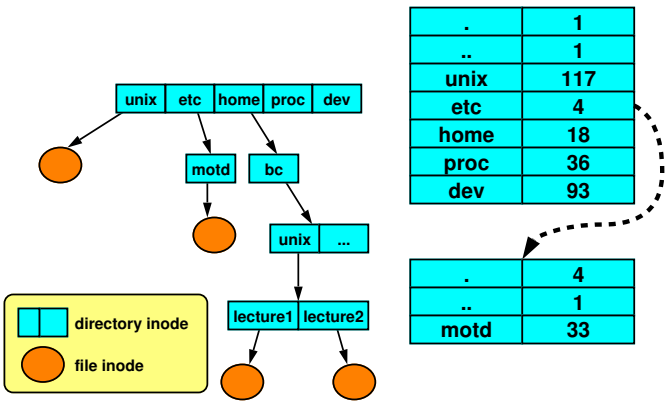


Directory Hierarchy

- Unix and many other OSes allow limited deviation from trees
  - *hard links*
    - reference to a *file* (not a directory) in one directory that also appears in another
    - using the `link()` system call or the "ln" shell command
  - *soft links* or *symbolic links*
    - a special kind of *file* containing the *name* of another file or directory
    - using the `symlink()` system call or the "ln -s" shell command
- Why hard link cannot be used on a directory?
  - to avoid cycles
  - Unix directory hierarchy can be viewed as a *directed acyclic graph* (DAG)
    - can be traversed efficiently

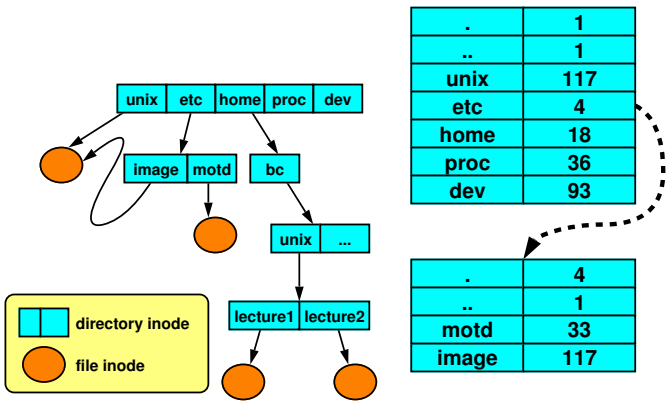
Hard Links

```
% ln /unix /etc/image
```



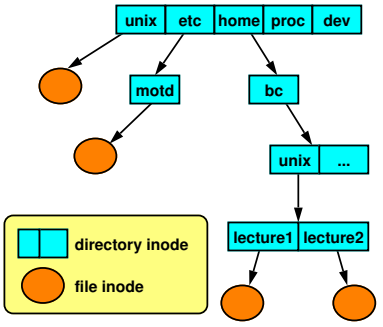
Hard Links

```
% ln /unix /etc/image
```



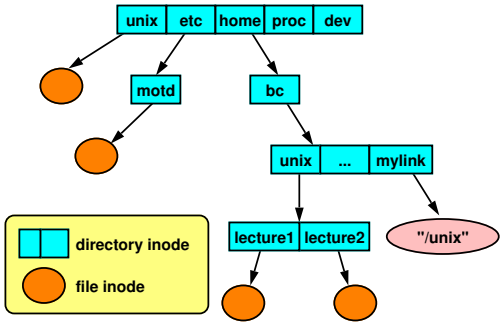
Soft Links

```
% ln -s /unix /home/bc/mylink
```



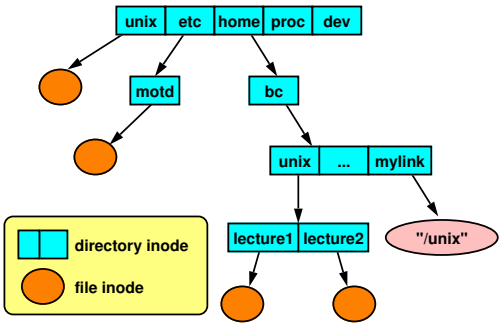
Soft Links

```
% ln -s /unix /home/bc/mylink
```



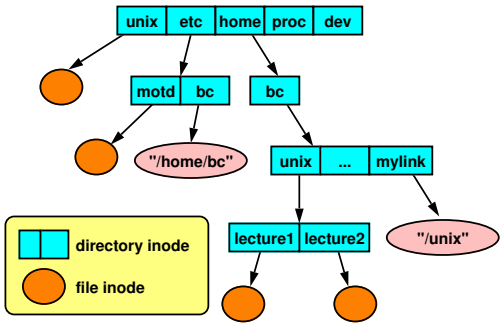
Soft Links

```
% ln -s /unix /home/bc/mylink
% ln -s /home/bc /etc/bc
```



Soft Links

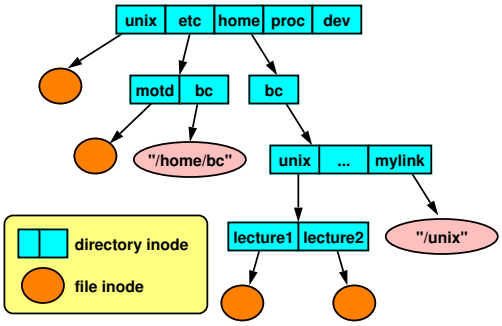
```
% ln -s /unix /home/bc/mylink
% ln -s /home/bc /etc/bc
```



Soft Links

`% ls -l /etc/bc/unix/lecture1`

- same as "`ls -l /home/bc/unix/lecture1`", or is it?
  - yes for the "`root`" account, may be no for the "`bc`" account
    - see "access protection"



Working Directory

- Maintained *in kernel for each process*
  - paths not starting from "`/`" start with the working directory
  - get by using the `getcwd()` system call
  - set by using the `chdir()` system call
  - displayed (via shell) using "`pwd`"

Access Protection

- OS needs to make sure that only authorized processes are allowed access to system resources
  - various ways to provide this
- Unix (and many other systems, such as Windows) associates with files some indication of which *security principals* are allowed access
  - along with what sort of access is allowed
- A *security principal* is normally a user or group of users
  - a "user" can be an identity used by processes performing system functions
  - each running process can have several security principals associated with it
    - all processes have a user identification and a set of group identifications
    - for Sixth-Edition Unix, only one user ID and one group ID

Access Protection

- Each file has associated with it a set of access permissions
  - there are 3 classes of security principals:
    - *user:* owner of the file
    - *group:* group owner of the file
    - *others:* everyone else
  - for each of the 3 classes of principals, specify what sorts of operations on the file are allowed
  - the operations are grouped into 3 classes:
    - *read:* can read a file or directory
    - *write:* can write a file or directory
    - *execute:* one must have *execute permission* for a *directory* in order to *follow a path through it*
- *Rules for checking permissions*
  1) determines the *smallest class of principals the requester belongs to* (user being smallest and others being largest)
  2) then it checks for appropriate permissions with that class

Permissions Example

```
% ls -lR
.:
total 2
drwxr-x--x  2 bill   adm    1024 Dec 17 13:34 A
drwxr-----  2 bill   adm    1024 Dec 17 13:34 B

./A:
total 1
-rw-rw-rw-  1 bill   adm     593 Dec 17 13:34 x

./B:
total 2
-r--rw-rw-  1 bill   adm     446 Dec 17 13:34 x
-rw----rw-  1 trina  adm     446 Dec 17 13:45 y
```

- Suppose that `bill` and `trina` are members of the `adm` group and `andy` is not
  1) Q: May `andy` list the contents of directory `A`?

Permissions Example

```
% ls -lR
.:
total 2
drwxr-x--x  2 bill   adm    1024 Dec 17 13:34 A
drwxr-----  2 bill   adm    1024 Dec 17 13:34 B

./A:
total 1
-rw-rw-rw-  1 bill   adm     593 Dec 17 13:34 x

./B:
total 2
-r--rw-rw-  1 bill   adm     446 Dec 17 13:34 x
-rw----rw-  1 trina  adm     446 Dec 17 13:45 y
```

- Suppose that `bill` and `trina` are members of the `adm` group and `andy` is not
  1) Q: May `andy` list the contents of directory `A`?
     A: No

Permissions Example

```
% ls -lR
.:
total 2
drwxr-x--x  2 bill   adm    1024 Dec 17 13:34 A
drwxr-----  2 bill   adm    1024 Dec 17 13:34 B

./A:
total 1
-rw-rw-rw-  1 bill   adm     593 Dec 17 13:34 x

./B:
total 2
-r--rw-rw-  1 bill   adm     446 Dec 17 13:34 x
-rw----rw-  1 trina  adm     446 Dec 17 13:45 y
```

- Suppose that `bill` and `trina` are members of the `adm` group and `andy` is not
  2) Q: May `andy` read `A/x`?

Permissions Example

```
% ls -lR
.:
total 2
drwxr-x--x  2 bill   adm    1024 Dec 17 13:34 A
drwxr-----  2 bill   adm    1024 Dec 17 13:34 B

./A:
total 1
-rw-rw-rw-  1 bill   adm     593 Dec 17 13:34 x

./B:
total 2
-r--rw-rw-  1 bill   adm     446 Dec 17 13:34 x
-rw----rw-  1 trina  adm     446 Dec 17 13:45 y
```

- Suppose that `bill` and `trina` are members of the `adm` group and `andy` is not
  2) Q: May `andy` read `A/x`?
     A: Yes

Permissions Example

```
% ls -lR
.:
total 2
drwxr-x--x  2 bill   adm    1024 Dec 17 13:34 A
drwxr-----  2 bill   adm    1024 Dec 17 13:34 B

./A:
total 1
-rw-rw-rw-  1 bill   adm     593 Dec 17 13:34 x

./B:
total 2
-r--rw-rw-  1 bill   adm     446 Dec 17 13:34 x
-rw----rw-  1 trina  adm     446 Dec 17 13:45 y
```

- Suppose that `bill` and `trina` are members of the `adm` group and `andy` is not
  3) Q: May `trina` list the contents of directory `B`?

Permissions Example

```
% ls -lR
.:
total 2
drwxr-x--x  2 bill   adm    1024 Dec 17 13:34 A
drwxr-----  2 bill   adm    1024 Dec 17 13:34 B

./A:
total 1
-rw-rw-rw-  1 bill   adm     593 Dec 17 13:34 x

./B:
total 2
-r--rw-rw-  1 bill   adm     446 Dec 17 13:34 x
-rw----rw-  1 trina  adm     446 Dec 17 13:45 y
```

- Suppose that `bill` and `trina` are members of the `adm` group and `andy` is not
  3) Q: May `trina` list the contents of directory `B`?
     A: Yes

Permissions Example

```
% ls -lR
.:
total 2
drwxr-x--x  2 bill   adm    1024 Dec 17 13:34 A
drwxr-----  2 bill   adm    1024 Dec 17 13:34 B

./A:
total 1
-rw-rw-rw-  1 bill   adm     593 Dec 17 13:34 x

./B:
total 2
-r--rw-rw-  1 bill   adm     446 Dec 17 13:34 x
-rw----rw-  1 trina  adm     446 Dec 17 13:45 y
```

- Suppose that `bill` and `trina` are members of the `adm` group and `andy` is not
  4) Q: May `trina` modify `B/y`?

Permissions Example

```
% ls -lR
.:
total 2
drwxr-x--x  2 bill   adm    1024 Dec 17 13:34 A
drwxr-----  2 bill   adm    1024 Dec 17 13:34 B

./A:
total 1
-rw-rw-rw-  1 bill   adm     593 Dec 17 13:34 x

./B:
total 2
-r--rw-rw-  1 bill   adm     446 Dec 17 13:34 x
-rw----rw-  1 trina  adm     446 Dec 17 13:45 y
```

- Suppose that `bill` and `trina` are members of the `adm` group and `andy` is not
  4) Q: May `trina` modify `B/y`?
     A: No

Permissions Example

```
% ls -lR
.:
total 2
drwxr-x--x  2 bill   adm    1024 Dec 17 13:34 A
drwxr-----  2 bill   adm    1024 Dec 17 13:34 B

./A:
total 1
-rw-rw-rw-  1 bill   adm     593 Dec 17 13:34 x

./B:
total 2
-r--rw-rw-  1 bill   adm     446 Dec 17 13:34 x
-rw----rw-  1 trina  adm     446 Dec 17 13:45 y
```

- Suppose that `bill` and `trina` are members of the `adm` group and `andy` is not
  5) Q: May `bill` modify `B/x`?

Permissions Example

```
% ls -lR
.:
total 2
drwxr-x--x  2 bill   adm    1024 Dec 17 13:34 A
drwxr-----  2 bill   adm    1024 Dec 17 13:34 B

./A:
total 1
-rw-rw-rw-  1 bill   adm     593 Dec 17 13:34 x

./B:
total 2
-r--rw-rw-  1 bill   adm     446 Dec 17 13:34 x
-rw----rw-  1 trina  adm     446 Dec 17 13:45 y
```

- Suppose that `bill` and `trina` are members of the `adm` group and `andy` is not
  5) Q: May `bill` modify `B/x`?
     A: No

Permissions Example

```
% ls -lR
.:
total 2
drwxr-x--x  2 bill   adm    1024 Dec 17 13:34 A
drwxr-----  2 bill   adm    1024 Dec 17 13:34 B

./A:
total 1
-rw-rw-rw-  1 bill   adm     593 Dec 17 13:34 x

./B:
total 2
-r--rw-rw-  1 bill   adm     446 Dec 17 13:34 x
-rw----rw-  1 trina  adm     446 Dec 17 13:45 y
```

- Suppose that `bill` and `trina` are members of the `adm` group and `andy` is not
  6) Q: May `bill` read `B/y`?

Permissions Example

```
% ls -lR
.:
total 2
drwxr-x--x  2 bill   adm    1024 Dec 17 13:34 A
drwxr-----  2 bill   adm    1024 Dec 17 13:34 B

./A:
total 1
-rw-rw-rw-  1 bill   adm     593 Dec 17 13:34 x

./B:
total 2
-r--rw-rw-  1 bill   adm     446 Dec 17 13:34 x
-rw----rw-  1 trina  adm     446 Dec 17 13:45 y
```

- Suppose that `bill` and `trina` are members of the `adm` group and `andy` is not
  6) Q: May `bill` read `B/y`?
     A: No

Open

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *path, int options [, mode_t mode])
```

- `options`
  - O_RDONLY open for reading only
  - O_WRONLY open for writing only
  - O_RDWR open for reading and writing
  - O_APPEND set the file offset to end of file prior to each write
  - O_CREAT if the file does not exist, then create it, setting its mode to mode adjusted by umask
  - O_EXCL: if O_EXCL and O_CREAT are set, then open fails if the file exists
  - O_TRUNC delete any previous contents of the file
  - O_NONBLOCK don't wait if I/O cannot be done immediately

Setting File Permissions

```
#include <sys/types.h>
#include <sys/stat.h>
int chmod(const char *path, mode_t mode)
```

- sets the file permissions of the given file to those specified in `mode`
- only the owner of a file and the superuser may change its permissions
- nine combinable possibilities for mode (read/write/execute for user, group, and others)
- `S_IRUSR` (0400), `S_IWUSR` (0200), `S_IXUSR` (0100)
- `S_IRGRP` (040), `S_IWGRP` (020), `S_IXGRP` (010)
- `S_IROTH` (04), `S_IWOTH` (02), `S_IXOTH` (01)
  - note: numeric prefix of 0 means the number is in octal format

Creating a File

- Use either `open` or `creat`
  - `open(const char *pathname, int flags, mode_t mode)`
    - flags must include O_CREAT
  - `creat(const char *pathname, mode_t mode)`
  - `open` is preferred
- The `mode` parameter helps specify the permissions of the newly created file
  - permissions = `mode & ~umask`

Umask

- Standard programs create files with "maximum needed permissions" as *mode*
  - compilers: `0777`
  - editors: `0666`
- Per-process parameter, *umask*, used to *turn off* undesired permission bits
  - e.g., turn off all permissions for others, write permission for group: set umask to `027`
  - compilers: permissions = `0777 & ~(027) = 0750`
  - editors: permissions = `0666 & ~(027) = 0640`
  - set with `umask()` system call or (usually) `umask` shell command

1.4 Beyond A Simple OS

- Extensions
- New Functionality

What Else?

- Beyond Sixth-Edition Unix (1975)
  - multiple threads per process
    - how is the process model affected?
  - virtual memory
    - in Sixth-Edition Unix, all currently running process had to fit into the computer's memory at once, along with the OS
    - virtual memory separates the address space from physical resources
  - name everything using directory-system path names
    - e.g., `/proc`
  - security
    - Unix solution is pretty elegant
    - new types of requirement such as permissions to add new software, perform backups, etc.

What Else?

- New functionalities
  - networking
    - much is beyond the scope of this class
  - interactive, multimedia user interface
    - make sure interactive user receives excellent response
  - software complexity
    - in Sixth-Edition Unix, to add a new device you need to:
      - write a "device driver" to handle the device
      - modify OS source code by adding references to the driver to a few tables
      - recompile the OS and reboot your computer
    - plug-and-play is desirable
      - need to support dynamic linking of modules into a running system
    - microkernel
      - how much of the OS functionality can be moved out of the kernel?