CSE 451: Operating Systems
Spring 2011

Module 1
Overview

John Zahorjan
zahorjan@cs.washington.edu
534 Allen Center

Module Overview

- **Part 1: A history of operating systems**
- **Part 2: An overview of Unix**

What is an Operating System?

- **Answers:**
  - **I don't know.**
  - **Nobody knows.**
  - **(The book knows. Read Chapter 1.)**

Okay. What Are Its Goals?

- **Answers:**
  - **Well, they're programs. They *can do* anything a program can do.**
  - **Did I mention they're programs? Big programs?**
    - **The Linux source you'll be compiling has over 1.7M lines of C code.**

Getting a Grip

- **Operating systems are the result of a 60 year long evolutionary process.**
  - **They were born out of need**
- **We'll follow a bit of their evolution**
- **That should help make clear what some of their functions are, and why**

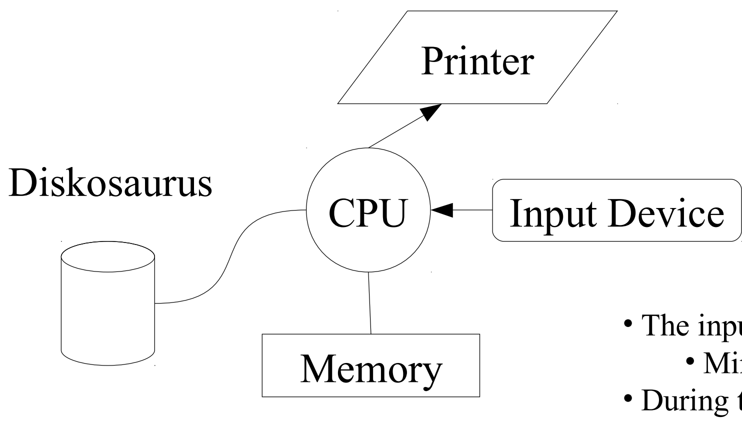
These Slides vs. Chapter 1

- **The text goes describes OS history in much more detail**
  - **It's an interesting read...**
- **These slides will try to give a higher level description, focusing on the impact of this history on today's systems**

In the Beginning...

- **1943:**
  - **T.J. Watson (created IBM):**
    "I think there is a world market for maybe five computers."
- **Fast forward: 1950**
  - **There are maybe 20 computers in the world**
  - **Why do we care?**
    - **They were unbelievably expensive**
    - **Imagine this: machine time is more valuable than person time!**
    - **Ergo: efficient use of the hardware is paramount**
  - **Operating systems are born**
    - **They carry with them the vestiges of these ancient forces**

The Primordial Computer



- The input device is very slow
  - Minutes to read a job
- During those minutes, the mainframe is idle!
- Idea: Let's have a "resident monitor" load the next job into memory while the current job is running

The Resident Monitor Needs Protection

- **This is a good plan, but what happens if the job in execution:**
  - **Goes into an infinite loop?**
  - **Has a bug and corrupts the resident monitor?**

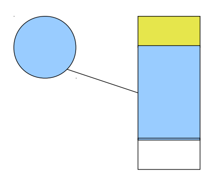
- **We need:**
  - **Interrupt/exception mechanism**
    - **Regain use of CPU, no matter what**
  - **Memory protection**
    - **User program can't overwrite monitor code**
  - **"user mode vs. supervisor mode" ("user/privileged", "user/kernel", "user/root", ...)**

Hey, That Worked!

- **Overlap of job input with job processing resulted in higher CPU utilization (a good thing)**
- **The new bottleneck: the diskosaurus**
  - **Disks were (are) slow**
  - **The CPU was spending a lot of time waiting around for data from the disk**
  - **What to do**
- **Course theme:**
  - **There are a handful of good/great ideas**
  - **(Re)Use them!**

I/O Overlap: Parallelism

- **Add hardware so that disk operates without tying up the CPU**
  - **Disk controller**
- **Hotshot programmers could now write code that:**
  - **Starts an I/O**
  - **Goes off and does some computing**
  - **Checks if the I/O is done at some later time**
  - **I'm going to refer to this kind of overlap, whose goal is to improve the performance of a single "job," as parallelism**

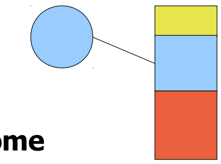


- **Upside**
  - **it helps increase CPU utilization**
- **Downsides**
  - **it's hard to get right (writing a correct program didn't get any easier...)**
  - **the benefits are job specific: is there enough available parallelism?**

I/O Overlap: Concurrency

(An easier way to exploit physical parallelism)

- **Run more than one job at a time**
- **When one starts an I/O, switch CPU to run a different one**

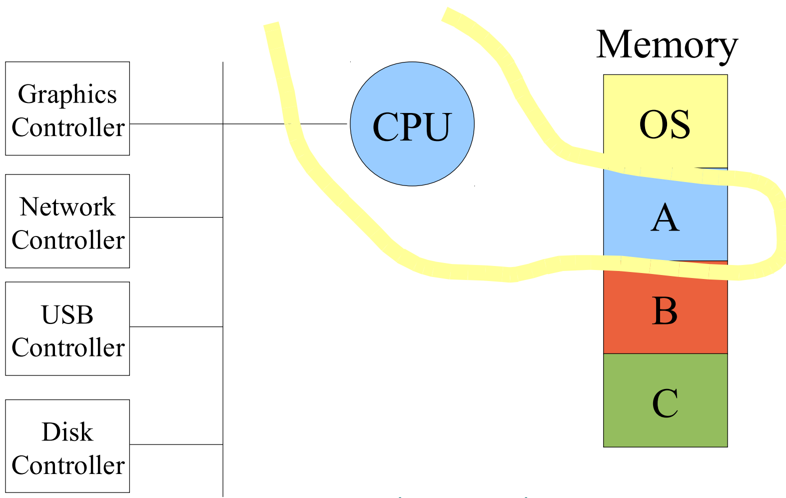


- **Upsides:**
  - **If you have enough jobs in memory, there's always some CPU work to do**
- **Downsides:**
  - **Memory allocation issues**
  - **Protection of one job from another (memory, disk, CPU)**
  - **CPU allocation issues**
  - **(Disk I/O allocation issues)**

Concurrency

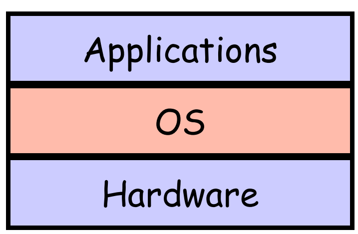
- **The official name for loading more than one job in memory and switching the CPU among them is *multiprogramming***
  - **All modern systems, even on fairly rudimentary devices, are multiprogrammed**
  - **Why?**
- **I'm going to refer to overlapped execution that simplifies programming effort as concurrency**
  - **Concurrent executions involve parallelism**
  - **They *can* have beneficial performance impacts for individual applications**
  - **Most often, though, the biggest win is that the computation is more easily built / managed / understood**
- **How is multiprogramming concurrency, by that definition?**

Multiprogramming



Protection Requirements ⇒
Programs execute directly on the CPU, but cannot touch anything other than their own memory without OS help

The More Customary Drawing



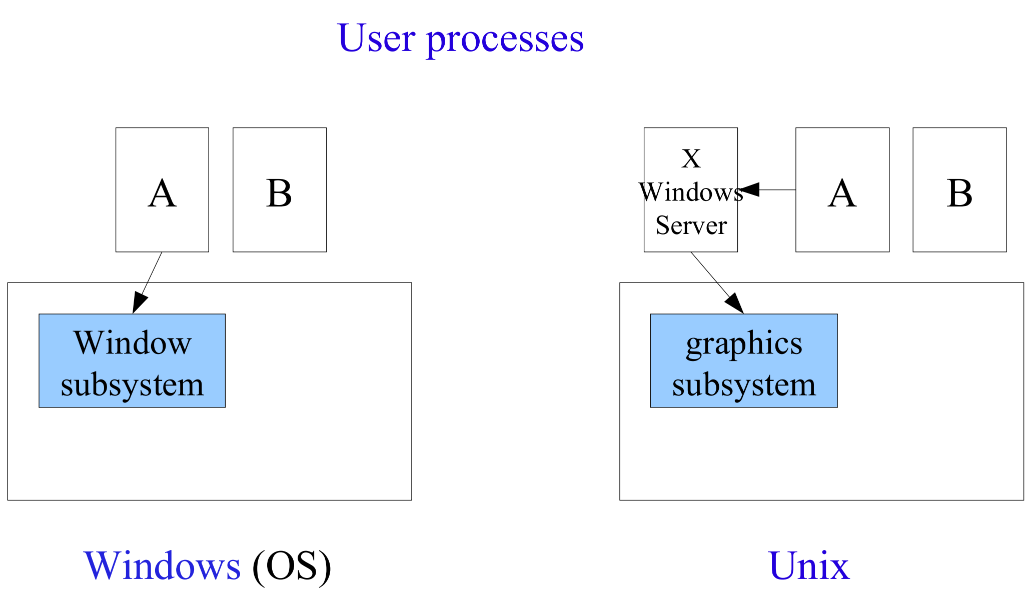
- **This depiction invites you to think of the OS as a library**
  - **It isn't:**
    - **you use the CPU/memory without OS calls**
    - **it intervenes without having been explicitly called**
  - **It is:**
    - **all operations on I/O devices require OS calls (*syscalls)***
- **So long as it is a library as far as I/O devices go, it might as well be a useful one**
  - **Presents nicer abstractions to program to than the raw hardware**

Device Abstractions

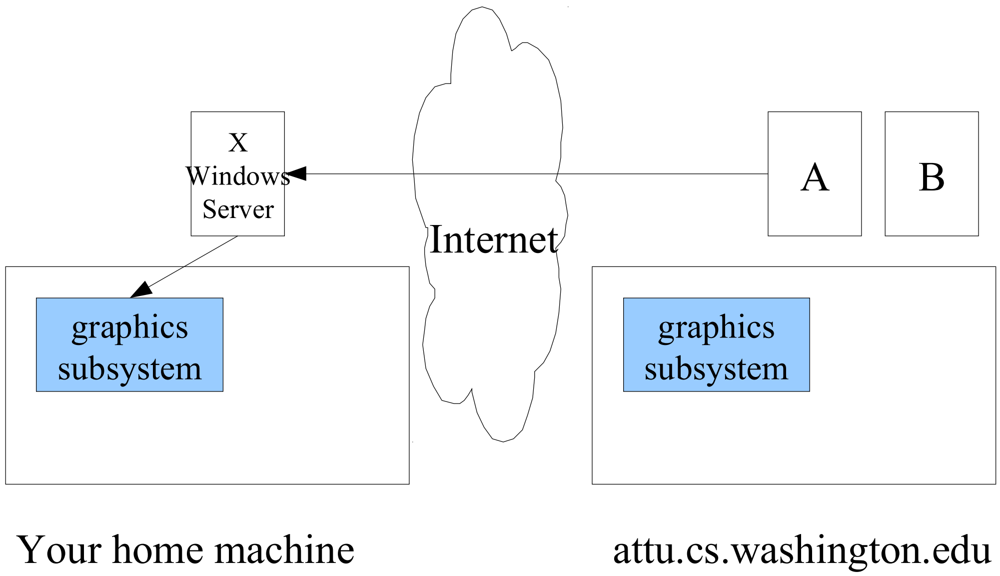
- **Examples:**
  - **Raw disk storage ⇒**
  - **Keyboard/mouse ⇒**
  - **Graphics card ⇒**
  - **Network interface card ⇒**

  - **CPU ⇒ process (/ thread)**
  - **Memory ⇒ virtual address space**
- **Besides protection, allocation, and performance, another role of the OS is programming convenience**

(Back To) What Is An Operating System?



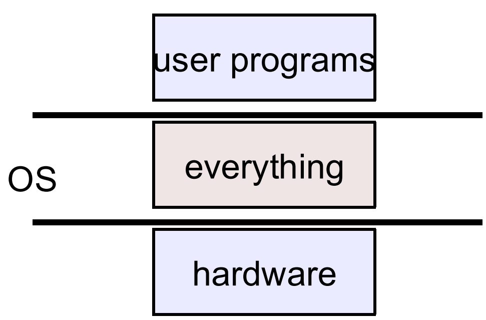
Impact of That Decision



Hey, That Worked! (OS Structure)

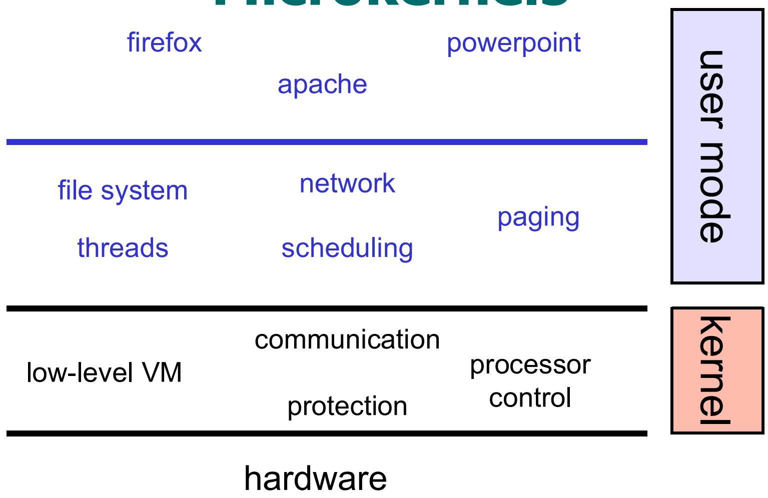
(So let's try using that idea again)

- **OS's evolved as *monolithic* implementations**



- **Pros:**
  - **Fast**
- **Cons:**
  - **Complicated**
  - **Inflexible**

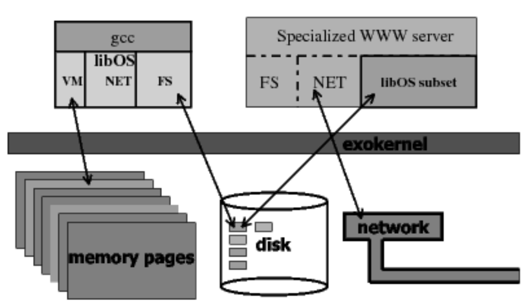
Microkernels



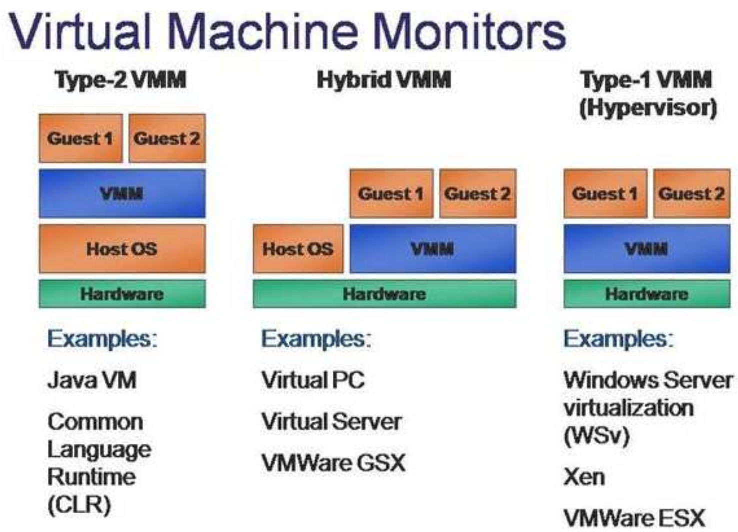
- **Pros:**
  - **Flexible**
  - **Debuggable**
- **Cons:**
  - **Slow**
  - **Can be complicated for applications**

Exokernel ("No Kernel")

- **Export hardware to user level (in a protected way)**



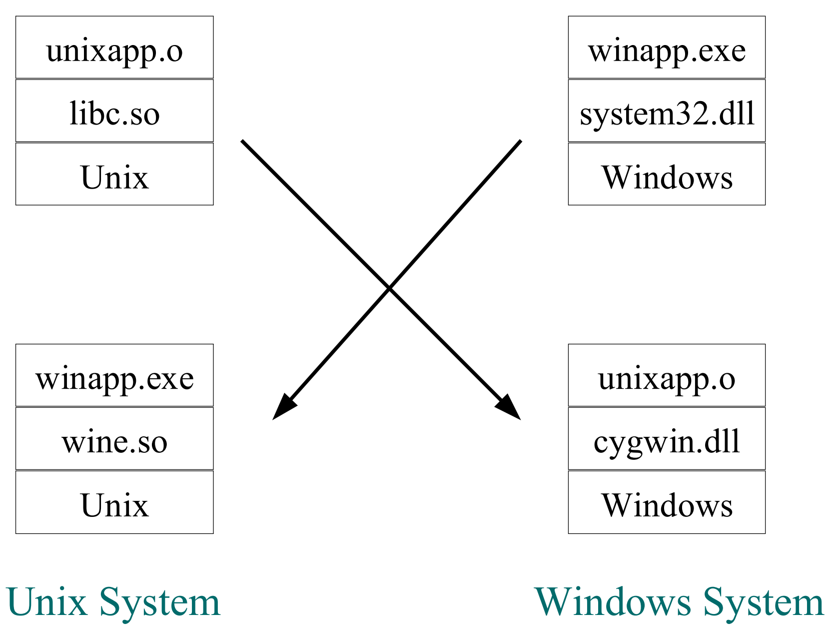
- **Pros:**
  - **Flexible**
  - **Arguably more efficient (than microkernel)**
- **Cons**
  - **Approximately 1.5B existing applications**



http://port25.technet.com/photos/images/images/4155/640x480.aspx

- **Transparently implement "hardware" in software**
- **Voilà, you can boot a "guest OS"**

(Another aside) Cross-system Application Portability



Core OS Functions

- **Programming convenience**
  - **OS provides abstractions / implements objects**
- **Concurrency**
  - **More than one computation is going on at a time**
- **Protection**
  - **Which then requires providing ways around protection**
- **Allocation**
  - **Hardware is shared; no way around that**
- **Performance / Efficiency**
  - **Achieving user specified objectives**

Recap: What is an Operating System?

- **We're still not sure**
- **An operating system (OS) is:**
  - **a software layer to abstract away and manage details of hardware resources**
  - **a set of utilities to simplify application development**
  - **"all the code you didn't write" in order to implement your application**
  - **the code that runs in privileged mode**
  - **The code that enforces allocation *policy***

© 2011 Gribble, Lazowska, Levy, Zahorjan

The major OS issues

- **structure: how is the OS organized?**
- **sharing: how are resources shared across users?**
- **naming: how are resources named (by users or programs)?**
- **security: how is the integrity of the OS and its resources ensured?**
- **protection: how is one user/program protected from another?**
- **performance: how do we make it all go fast?**
- **reliability: what happens if something goes wrong (either with hardware or with a program)?**
- **extensibility: can we add new features?**
- **flexibility: are we in the way of new apps?**
- **communication: how do programs exchange information, including across a network?**

More OS issues…

- **concurrency: how are parallel activities (computation and I/O) created and controlled?**
- **scale: what happens as demands or resources increase?**
- **persistence: how do you make data last longer than program executions?**
- **distribution: how do multiple computers interact with each other?**
- **accounting: how do we keep track of resource usage, and perhaps charge for it?**

There are tradeoffs, not right and wrong.

OS (Unix) Overview

- Processes
- Files
- Directories
- File Representation
- File-oriented System Calls

A Program

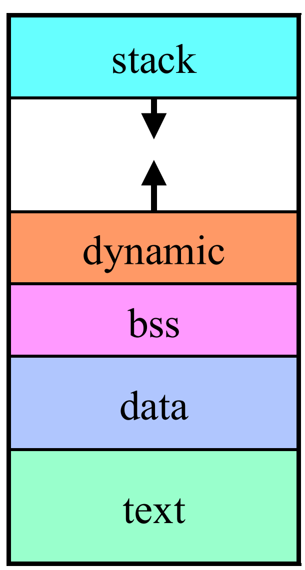
```
const int nprimes = 100;
int prime[nprimes];
int main() {
   int i;
   int current = 2;
   prime[0] = current;
   for (i=1; i<nprimes; i++) {
      int j;
   NewCandidate:
      current++;
      for (j=0; prime[j]*prime[j] <= current; j++) {
         if (current % prime[j] == 0)
            goto NewCandidate;
      }
      prime[i] = current;
   }
   return(0);
}
```

Processes

Fundamental abstraction of program execution

- memory
- processor(s) (core(s))
  - each processor abstraction is a *thread*
- "execution context"

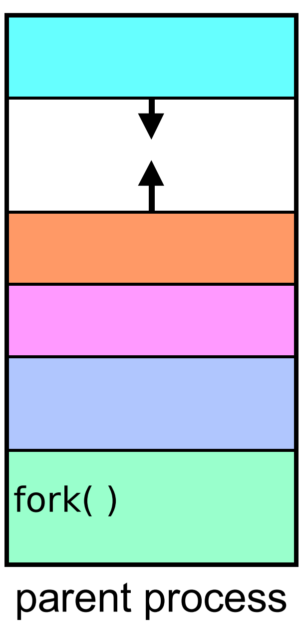
The Unix Address Space



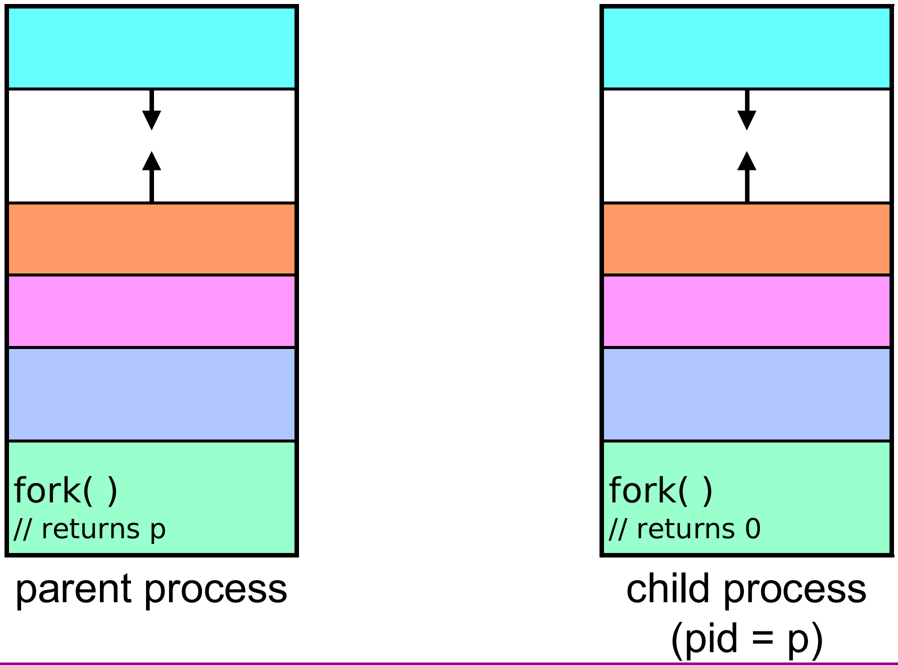
Modified Program

```
const int nprimes = 100;
int *prime;
int main(int argc, char *argv[]) {
   int i;
   int current = 2;
   nprimes = atoi(argv[1]);
   prime = (int *)malloc(nprimes*sizeof(int))
   prime[0] = current;
   for (i=1; i<nprimes; i++) {

      …

   }
   return(0);
}
```

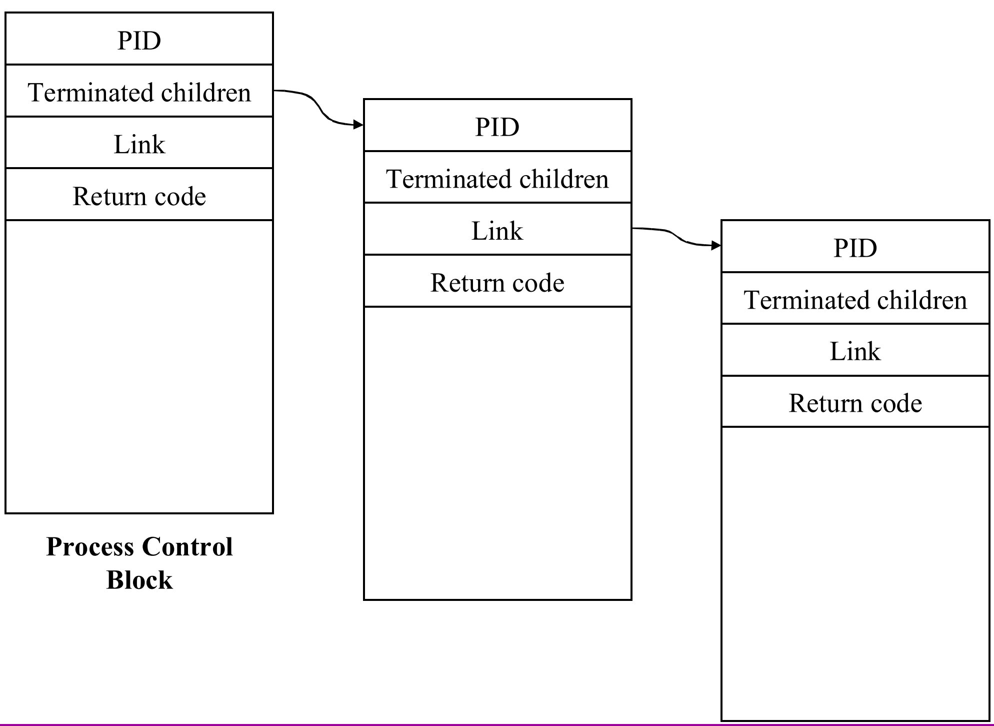

Creating a Process: Before



Creating a Process: After



Process Control Blocks



Fork and Wait

```
short pid;
if ((pid = fork()) == 0) {
   /* some code is here for the child to execute */
   exit(n);
} else {
   int ReturnCode;
   while(pid != wait(&ReturnCode))
      ;
   /* the child has terminated with ReturnCode as its
      return code */
}
```

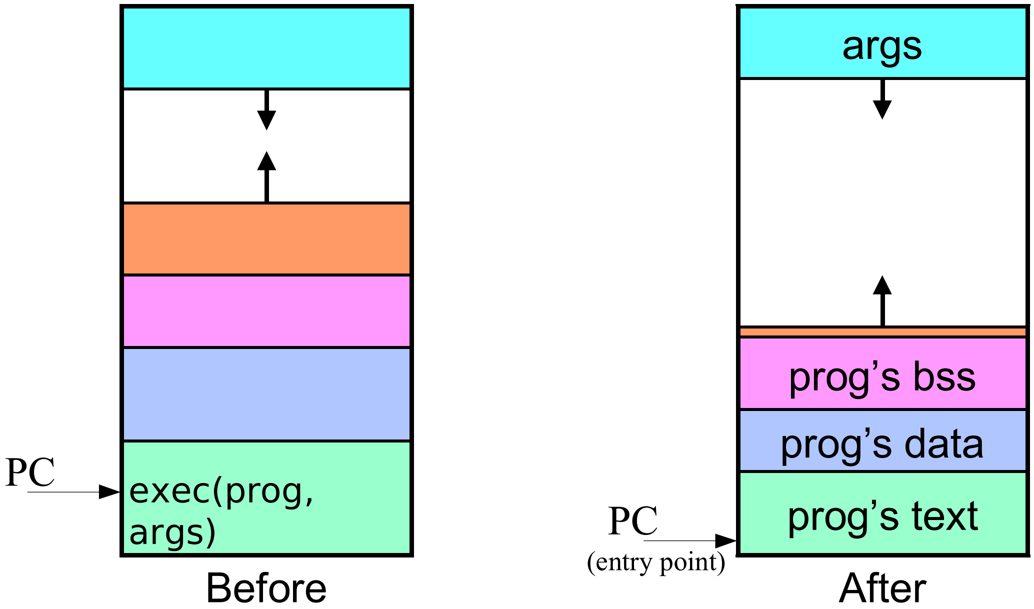

Exec

```
int pid;
if ((pid = fork()) == 0) {
   /* we'll soon discuss what might take place before exec
      is called */
   execl("/home/twd/bin/primes", "primes", "300", 0);
   exit(1);
}

/* parent continues here */

while(pid != wait(0))   /* ignore the return code */
   ;
```

This is the essence of the implementation of a shell:

```
$ /home/twd/bin/primes 300
```


Loading a New Image



System Calls

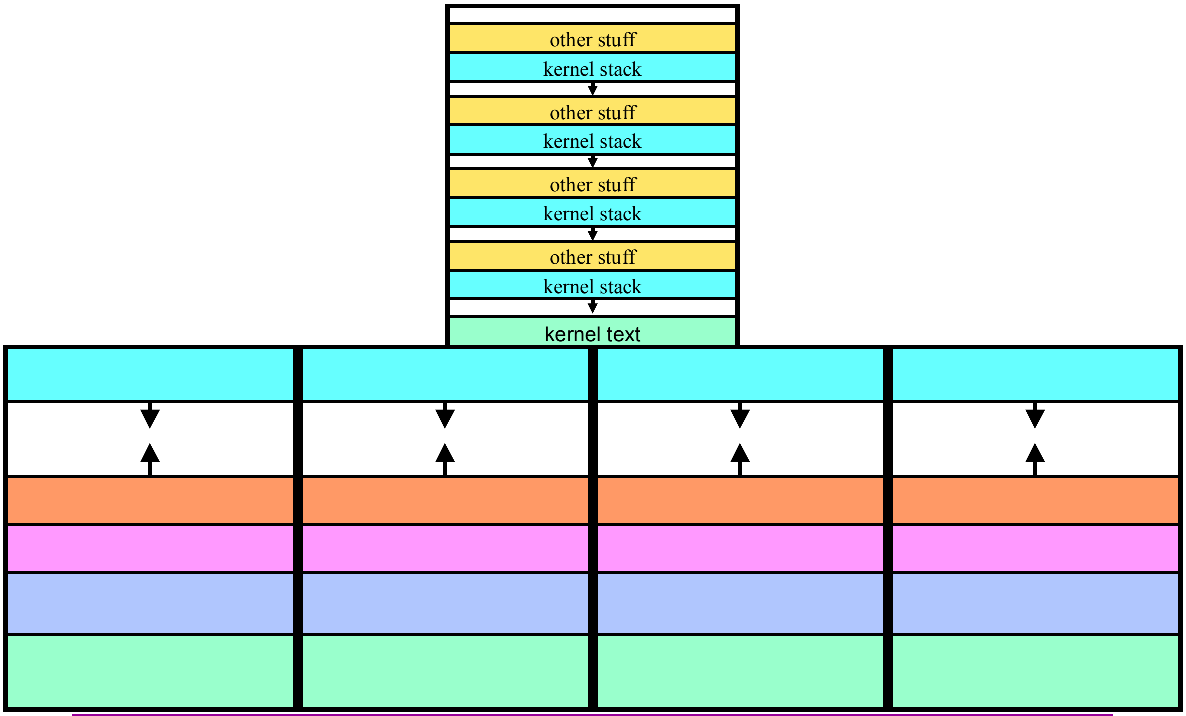
Interface between user and kernel

Typically implemented in two pices:
- a library routines called by user code, and
- a *trap* instruction in the library routine to enter the kernel

Errors indicated by returns of –1; error code is in *errno*

```
if (write(fd, buffer, bufsize) == -1) {
        // error!
        printf("error %d\n", errno);
        // see perror
}
```

Multiple Processes



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*

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/twd/data", O_RDWR);

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

// either way, this works
int bytes = read(file, buffer, sizeof(buffer));
write(device, buffer, bytes);
```


Working Directory

Maintained in kernel *for each process*

paths not starting from "/" start with the working directory

changed by use of the *chdir* system call

displayed (via shell) using "pwd"

- how is this done?

Standard File Descriptors

File number	Name	Use
0	stdin	Input
1	stdout	Normal output
2	stderr	Error output

```
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 …

```
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);
}
```


Human-Readable Output

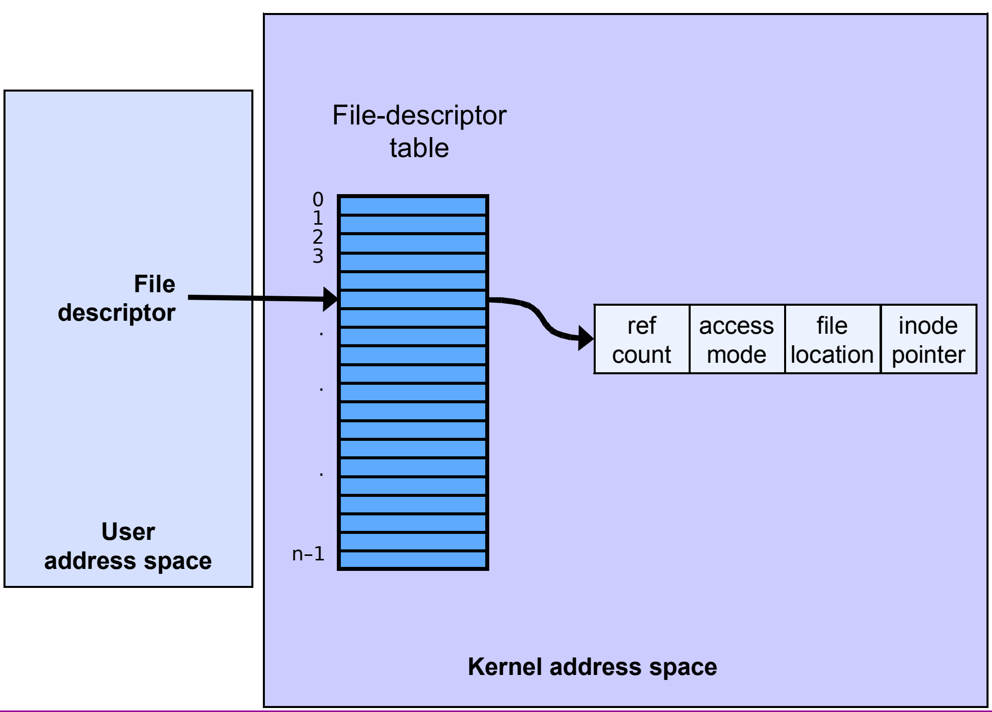
```
int nprimes;
int *prime;
int main(int argc, char *argv[]) {
   …
   for (i=1; i<nprimes; i++) {
      …
   }
   for (i=0; i<nprimes; i++) {
      printf("%d\n", prime[i]);
   }
   return(0);
}
```


Running It

```
if (fork() == 0) {
   /* set up file descriptor 1 in the child process */
   close(1);
   if (open("/home/twd/Output", O_WRONLY) == -1) {
      perror("/home/twd/Output");
      exit(1);
   }
   execl("/home/twd/bin/primes", "primes", "300", 0);
   exit(1);
}

/* parent continues here */

while(pid != wait(0))   /* ignore the return code */
   ;
```

```
$ /home/twd/bin/primes 300 >/home/twd/Output
```


File-Descriptor Table



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);
```

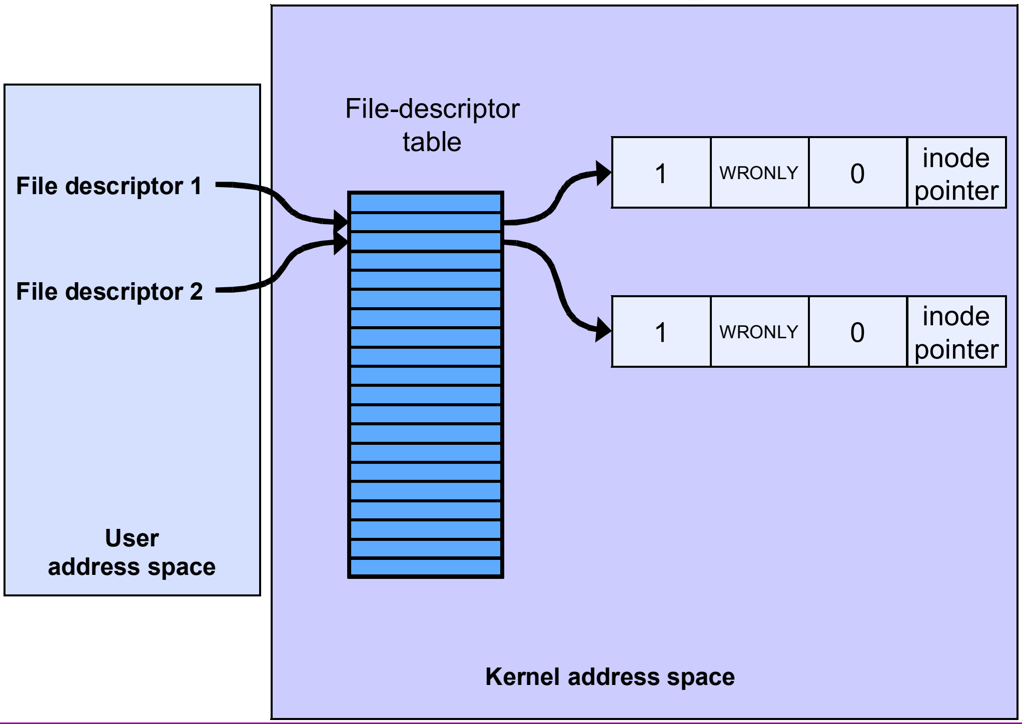
will always associate *file* with file descriptor 0 (assuming that the *open* succeeds)

Redirecting Output … Twice

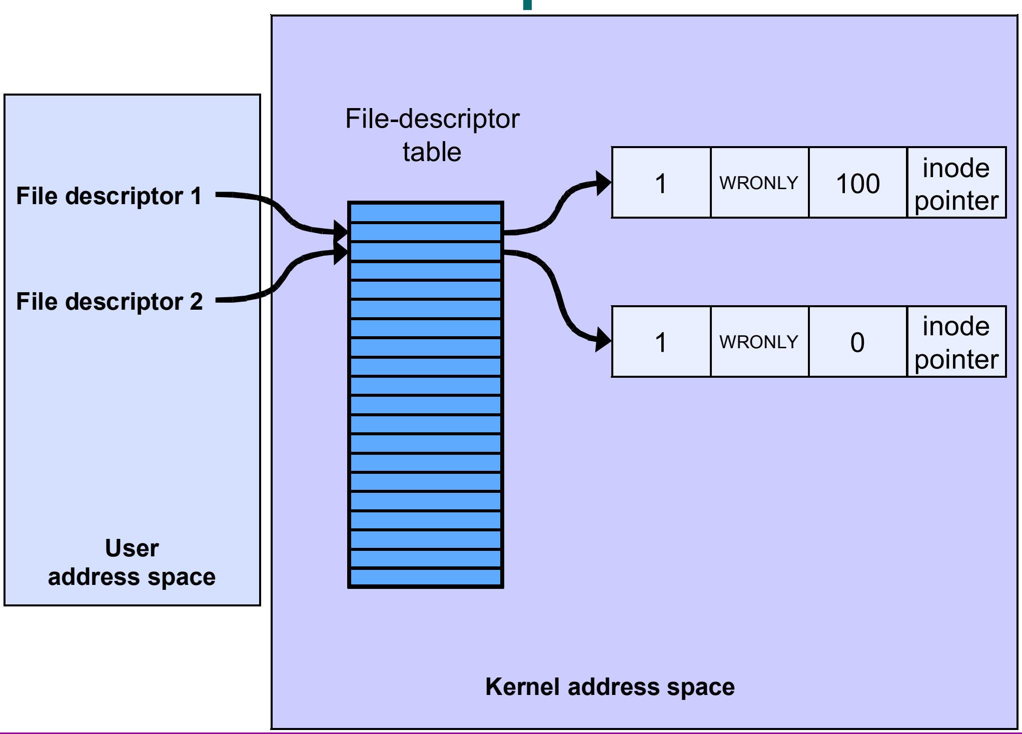
```
if (fork() == 0) {
   /* set up file descriptors 1 and 2 in the child process */
   close(1);
   close(2);
   if (open("/home/twd/Output", O_WRONLY) == -1) {
      exit(1);
   }
   if (open("/home/twd/Output", O_WRONLY) == -1) {
      exit(1);
   }
   execl("/home/twd/bin/program", "program", 0);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

/* parent continues here */
```

```
$ /home/twd/bin/primes 300 >/home/twd/Output 2>/home/twd/Output
```

Redirected Output



Redirected Output After Write

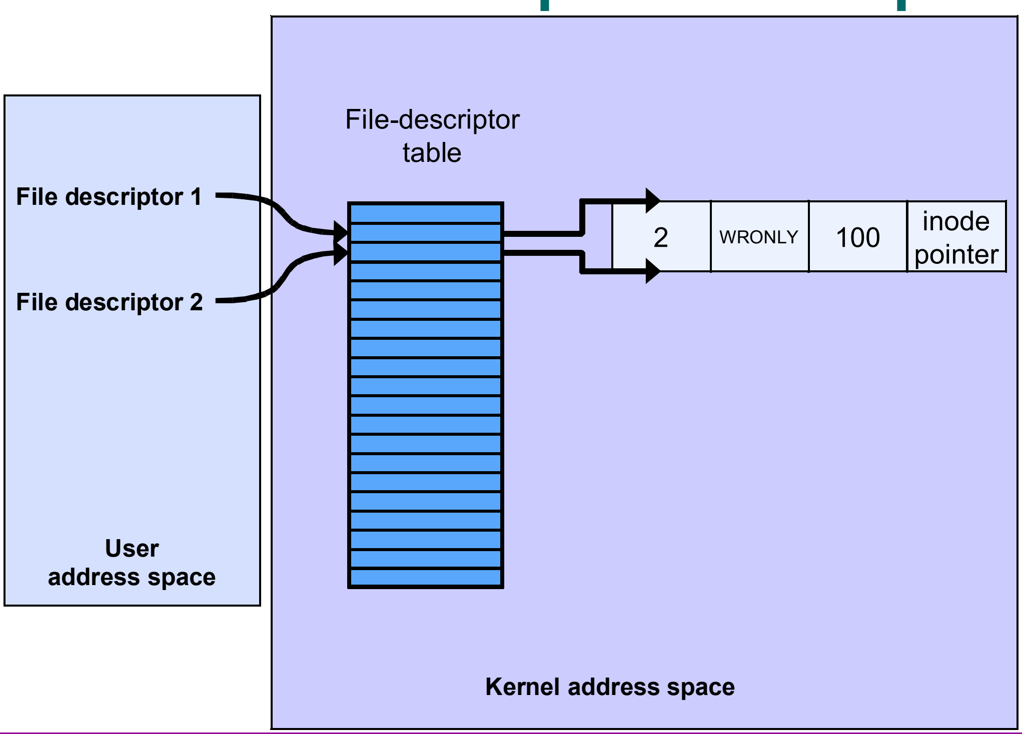


Sharing Context Information

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

```
$ /home/twd/bin/primes 300 >/home/twd/Output 2>&1
```

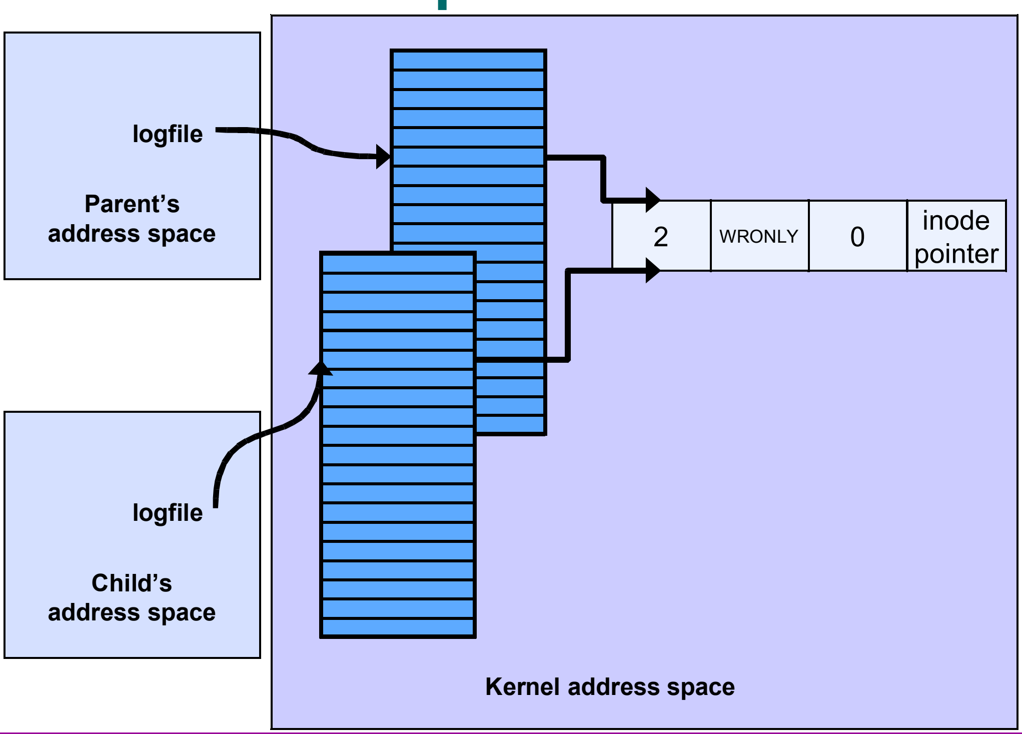

Redirected Output After Dup



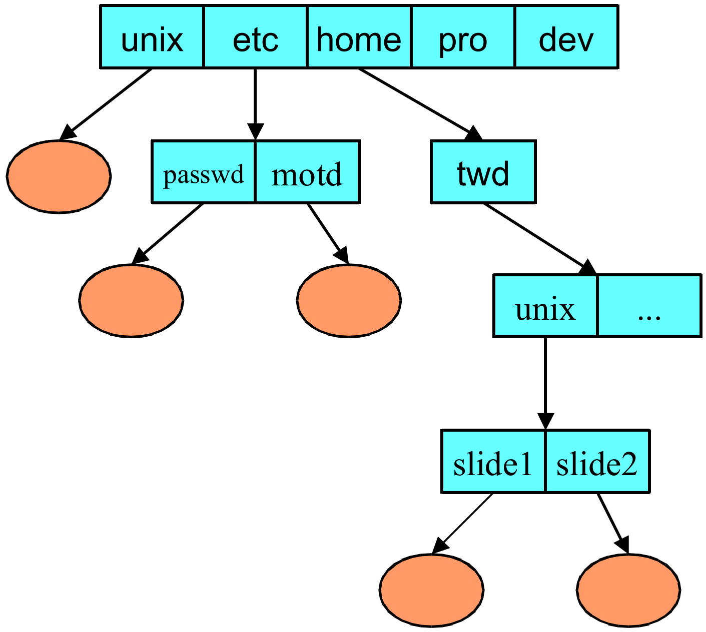
Fork and File Descriptors

```
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));
…
```


File Descriptors After Fork



Directories



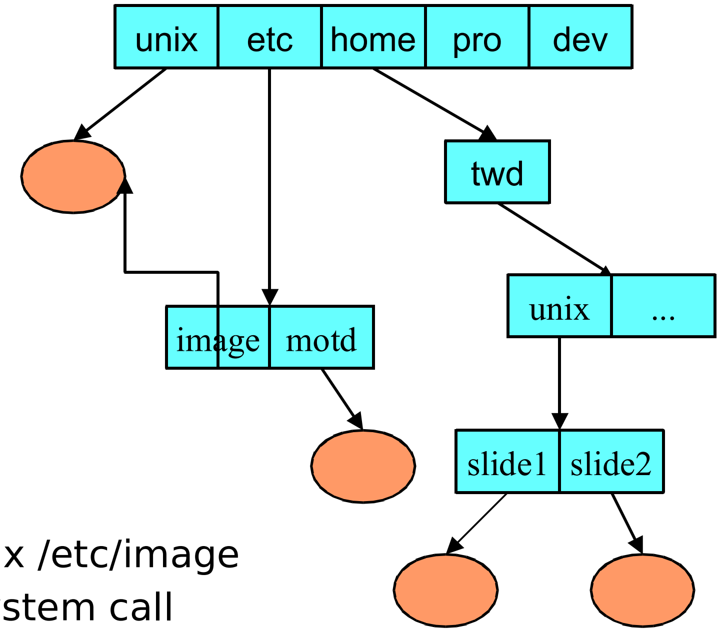
Directory Representation

Component Name	Inode Number

directory entry

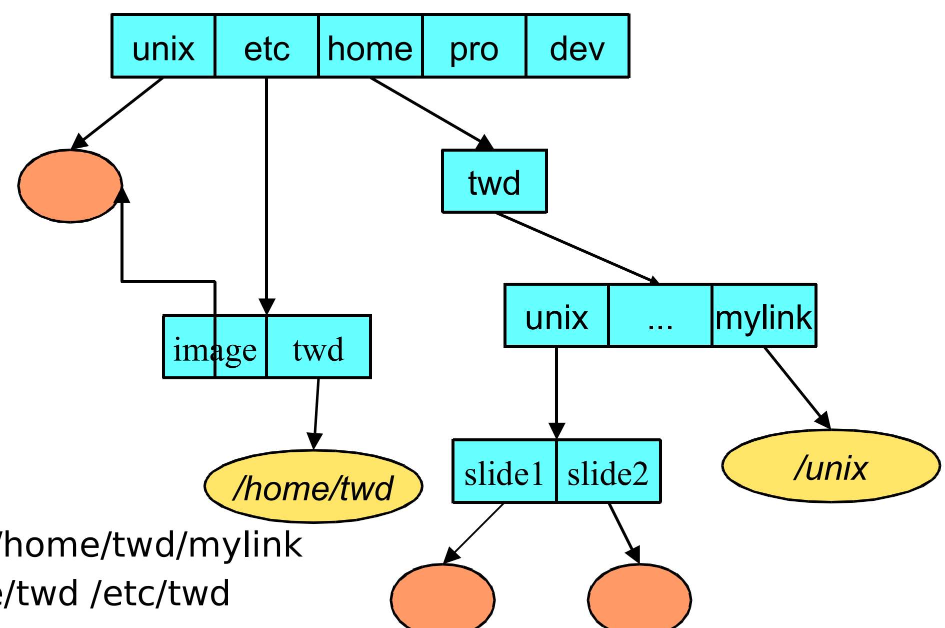
.	1
..	1
unix	117
etc	4
home	18
pro	36
dev	93

Hard Links



% ln /unix /etc/image
# link system call

Soft Links



% ln –s /unix /home/twd/mylink
% ln –s /home/twd /etc/twd
# symlink system call

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 can't be done immediately

File Access Permissions

Who's allowed to do what?

who

- user (owner)
- group
- others (rest of the world)

what

- read
- write
- execute

Permissions Example

% ls -lR

```
.:
total 2
drwxr-x--x   2 tom     adm     1024 Dec 17 13:34 A
drwxr-----   2 tom     adm     1024 Dec 17 13:34 B

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

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


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)

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) shell command

What Else?

Beyond Sixth-Edition Unix (1975)

- multiple threads per process
  - how is the process model affected?
- virtual memory
  - fork?
- interactive, multimedia user interface
  - scheduling?
- networking
- security

Final Note: Performance

- **We've just seen a survey of the functionality offered by an OS**
- **Another side of the coin is the performance of the OS**
  - **if operations are slow, some applications can't run**
- **From repeated experience...**
  - **Build fast and simple at the low levels**
    - **You can layer functionality on top**
    - **You can't "unlayer" overheads for facilities you don't need**

A Sense of Absolute Costs

Obtained using lmbench

Proc	AMD Athlon 64 X2 (2.8GHz, 0.358 nsec. Clock)				
OS	Linux 2.6.31.4				
32-bit int add	2.3	nsec		int parallelism	1.26
32-bit int div	47.6	nsec			
float add	2.5	nsec		float parallelism	2.7
float div	18.1	nsec			
null syscall	220.2	nsec			
stat	1,311.1	nsec			
file open/close	2,727.6	nsec.			
sig hdlr install	326.5	nsec.			
sig hdlr ovrhd	1,602.5	nsec			
protection fault	303.1	nsec			
page fault	1,556.7	nsec.			
ctx switch	2,290.0	nsec.	2 processes writing 0 data bytes		
	5,270.0	nsec.	2 processes writing 64KB data		
	22,470.0	nsec.	16 processes writing 64KB data		
fork	331,500.0	nsec			
fork + exec	342,400.0	nsec.			
fork + sh cmd	1,960,700.0	nsec.			
disk seek	6,000,000.0	nsec.	highly variable		
disk xfer rate	50,000.0	nsec./4KB	highly variable		