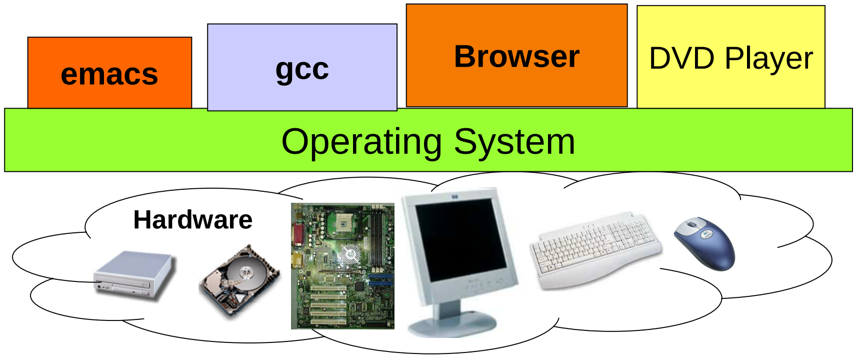
Operating Systems, System Calls, and Buffered I/O

CS 217

What Is Operating System?



emacs
gcc
Browser
DVD Player

Operating System

Hardware

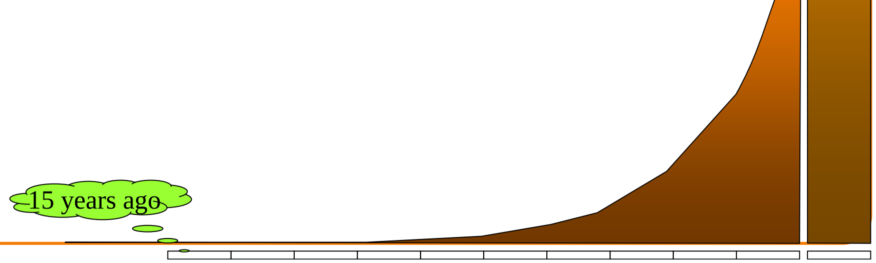
- Abstraction of hardware
- Virtualization
- Protection and security

Academic Computers in 1983 and 2003

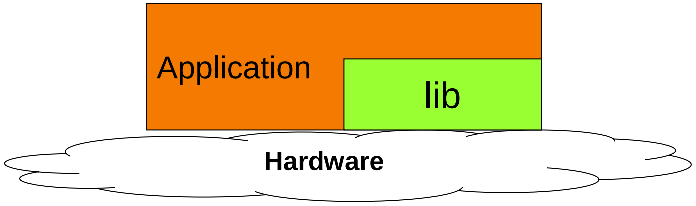
	1883	2003	Ratio
CPU clock	3Mhz	3Ghz	1:1000
$/machine	$80k	$800	100:1
DRAM	256k	256M	1:1000
Disk	20MB	200GB	1:10,000
Network BW	10Mbits/sec	1GBits/sec	1:100
Address bits	16-32	32-64	1:2
Users/machine	10s	1 (or < 1)	> 10:1
$/Performance	$80k	< $800/1000	100,000+:1

Computing and Communications Exponential Growth! (Courtesy J. Gray)

- Performance/Price doubles every 18 months
- 100x per decade
- Progress in next 18 months
  = ALL previous progress
  - New storage = sum of all old storage (ever)
  - New processing = sum of all old processing.
- Aggregate bandwidth doubles in 8 months



Phase 0: User at Console



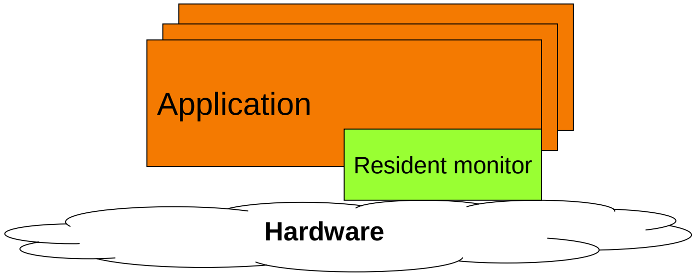
Phase 0: User at Console

- ***How things work***
  - One TOY machine for CS126, what do we do?
  - No OS, just <u>a sign-up sheet</u> for reservations!
  - Each user has complete control of machine
  - Soon added <u>device libraries, compilers, assemblers</u> for convenience
- ***Advantages***
  - Interactive!
  - No one can hurt anyone else
- ***Disadvantages***
  - Reservations not accurate, leads to inefficiency
  - Loading/unloading tapes and cards takes forever and leaves the machine idle

---

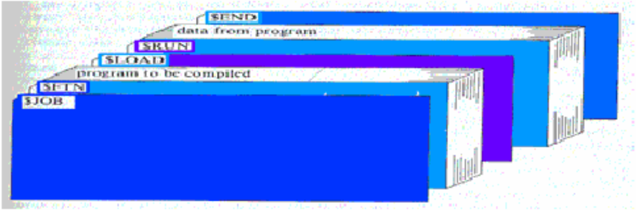
Phase 1: Batch Processing

(Expensive Hardware, Cheap Humans)



Phase 1: Batch Processing

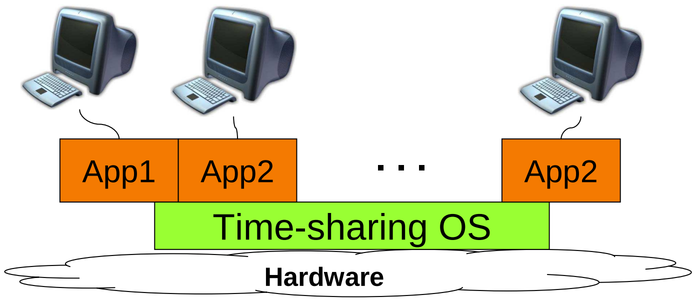
(Expensive Hardware, Cheap Humans)



- ***How things work***
  - Sort jobs and batch those with similar needs to reduce unnecessary setup time
  - <u>A resident monitor</u> provides "automatic job sequencing": it interprets "control cards" to automatically run a bunch of programs without human intervention
- ***Advantage***
  - Good utilization of machine, (jargon: high <u>throughput</u>: jobs per second)
- ***Problems***
  - Loss of interactivity (unsolvable)
  - One job can screw up other jobs, need protection (solvable)

Phase 2: Interactive Time-Sharing

(Cheap Hardware, Expensive Humans)



Phase 2: Interactive Time-Sharing

(Cheap Hardware, Expensive Humans)

- ***How things work***
  - Multiple cheap terminals for multiple users per single machine
  - OS keeps multiple programs active at the same time and switches among them rapidly to provide the illusion of one machine per user
- ***Advantage***: interactivity, sharing (collaboration)
- ***Problems***
  - Must provide reasonable response time (hard sometimes)
  - Must provide human friendly interfaces: command shell, hierarchical name structure for file systems, etc. (solvable)
  - Higher degree of multiprogramming places heavier demand on protection mechanism (solvable but hard)

Phase 3: Personal Computing

(Very Cheap Hardware, Very Expensive Humans)

Phase 3: Personal Computing

(Very Cheap Hardware, Very Expensive Humans)

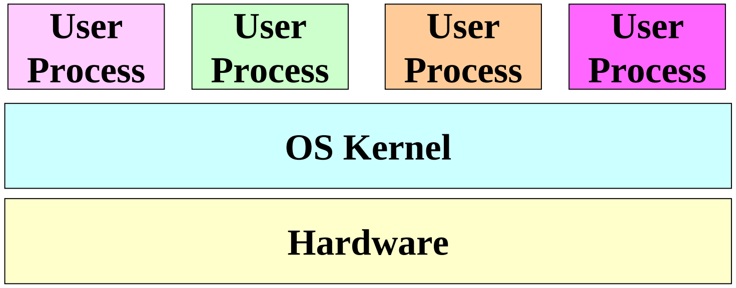
- ***How things work***
  - One machine per person, now several machines per person
  - Initially, OS goes back to “square 1” (like those of Phase 0)
  - Later added back multiprogramming and memory protection
- ***Advantages***
  - Better response time
  - Protection becomes a little easier
- ***Problems***
  - How do you share information? (sill not solved)
    - networking
    - user interfaces

Phase 4: > 1 Machines per User

- Parallel and distributed systems
  - Parallel machine
  - Clusters
  - Network is the computer
- Pervasive computers
  - Wearable computers
  - Computers everywhere
- OS are general and specialized

A Typical Operating System

- Abstraction: Layered services to access hardware
  - We learn how to use the services here
  - COS318 will teach how to implement
- Virtualization: Each user with his “own” machine
- Protection & security: safe from yourself and others

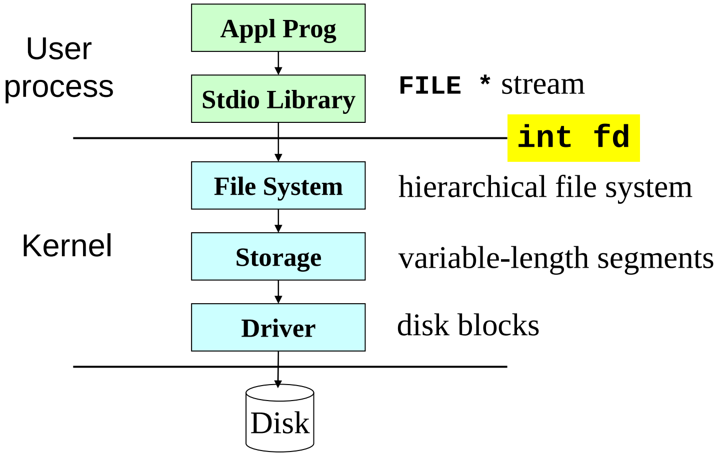


User Process | **User Process** | **User Process** | **User Process**

OS Kernel

Hardware

Layers of Abstraction



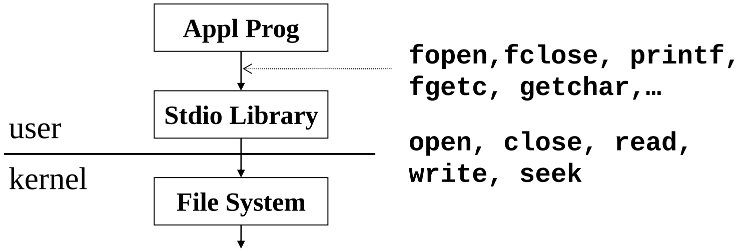
User process
- **Appl Prog**
- **Stdio Library** — `FILE *` stream
- `int fd`

Kernel
- **File System** — hierarchical file system
- **Storage** — variable-length segments
- **Driver** — disk blocks

Disk

System Calls

- Kernel provided system services: “protected” procedure call



user
- **Appl Prog**
- **Stdio Library**

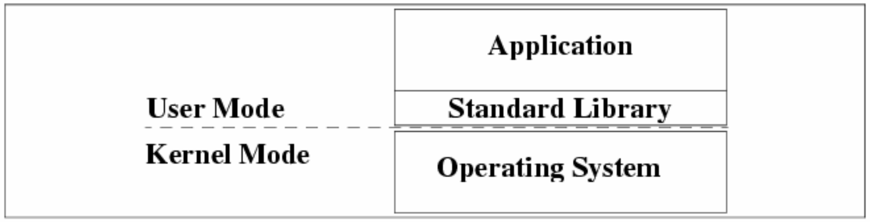
kernel
- **File System**

`fopen,fclose, printf, fgetc, getchar,…`

`open, close, read, write, seek`

- Unix has ~150 system calls; see
  - man 2 intro
  - /usr/include/syscall.h

Dual-Mode Operation



- The machine has two modes of operation: user mode and kernel mode (also called monitor mode, supervisor mode, system mode, privileged mode)
- Divide all instructions into two categories: unprivileged and privileged instructions
- Users can't execute privileged instructions ◦(`int`)
- Users must ask the OS to do it on its behalf: system calls
- The OS gains control upon a system call, switches to kernel mode, performs service, switches back to user mode, and gives control back to user ◦(`iret`)

System-call interface = ADTs

ADT
  operations

- File input/output
  - open, close, read, write, dup
- Process control
  - fork, exit, wait, kill, exec, ...
- Interprocess communication
  - pipe, socket ...

---

open system call

NAME

open - open and possibly create a file or device

`flags` `examples:`
`O_RDONLY`
`O_WRITE|O_CREATE`

SYNOPSIS

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

mode is the permissions to use if file must be created

int open(const char *pathname, int flags, mode_t mode);

DESCRIPTION

The open() system call is used to convert a pathname into a file descriptor (a small, non-negative integer for use in subsequent I/O as with read, write, etc.). When the call is successful, the file descriptor returned will be . . .

`close` system call

NAME

`close` - close a file descriptor

SYNOPSIS

int close(int fd);

DESCRIPTION

`close` closes a file descriptor, so that it no longer refers to any file and may be reused. Any locks held on the file it was associated with, and owned by the process, are removed (regardless of the file descriptor that was used to obtain the lock) . . . .

`read` System Call

NAME

`read` - read from a file descriptor

SYNOPSIS

int read(int fd, void *buf, int count);

DESCRIPTION

`read()` attempts to read up to **count** bytes from file descriptor **fd** into the buffer starting at **buf**.

If count is zero, read() returns zero and has no other results. If count is greater than SSIZE_MAX, the result is unspecified.

RETURN VALUE

On success, the number of bytes read is returned (zero indicates end of file), and the file position is advanced by this number. It is not an error if this number is smaller than the number of bytes requested . . . . On error, -1 is returned, and **`errno`** is set appropriately.

`write` System Call

NAME

`write` – write to a file descriptor

SYNOPSIS

int write(int fd, void *buf, int count);

DESCRIPTION

`write` writes up to **count** bytes to the file referenced by the file descriptor **fd** from the buffer starting at **buf**.

RETURN VALUE

On success, the number of bytes written is returned (zero indicates nothing was written). It is not an error if this number is smaller than the number of bytes requested . . . . On error, -1 is returned, and **`errno`** is set appropriately.

Making Sure It All Gets Written

```
int safe_write(int fd, char *buf, int nbytes)
{
   int n;
   char *p = buf;
   char *q = buf + nbytes;
   while (p < q) {
      if ((n = write(fd, p, (q-p)*sizeof(char))) > 0)
         p += n/sizeof(char);
      else
         perror(“safe_write:”);
   }
   return nbytes;
}
```

Buffered I/O

- Single-character I/O is usually too slow

```
int getchar(void) {
   char c;
   if (read(0, &c, 1) == 1)
      return c;
   else return EOF;
}
```

Buffered I/O (cont)

- Solution: read a chunk and dole out as needed

```
int getchar(void) {
   static char buf[1024];
   static char *p;
   static int n = 0;

   if (n--) return *p++;

   n = read(0, buf, sizeof(buf));
   if (n <= 0) return EOF;
   p = buf;
   return getchar();
}
```

Standard I/O Library

```
#define getc(p) (--(p)->_cnt >= 0 ? \
   (int)(*(unsigned char *)(p)->_ptr++) : \
   _filbuf(p))

typedef struct _iobuf {
   int _cnt;    /* num chars left in buffer */
   char *_ptr;  /* ptr to next char in buffer */
   char *_base; /* beginning of buffer */
   int _bufsize;/* size of buffer */
   short _flag; /* open mode flags, etc. */
   char _file;  /* associated file descriptor */
} FILE;

extern FILE *stdin, *stdout, *stderr;
```

Why Is “getc” A Macro?

```
#define getc(p) (--(p)->_cnt >= 0 ? \
   (int)(*(unsigned char *)(p)->_ptr++) : \
   _filbuf(p))

#define getchar() getc(stdin)
```

- Invented in 1970s, when
  - Computers had slow function-call instructions
  - Compilers couldn’t inline-expand very well
- It’s not 1975 any more
  - Moral: don’t invent new macros, use functions

fopen

```
FILE *fopen(char *name, char *rw) {
  Use malloc to create a struct _iobuf
  Determine appropriate “flags” from “rw” parameter
  Call open to get the file descriptor
  Fill in the _iobuf appropriately
}
```

Stdio library

- fopen, fclose
- feof, ferror, fileno, fstat
  - status inquiries
- fflush
  - make outside world see changes to buffer
- fgetc, fgets, fread
- fputc fputs, fwrite
- printf, fprintf
- scanf, fscanf
- fseek
- *and more ...*

This (large) library interface is not the operating-system interface; much more room for flexibility.

This ADT is implemented in terms of the lower-level “file-descriptor” ADT.

Summary

- OS is the software between hardware and applications
  - Abstraction: provide services to access the hardware
  - Virtualization: Provides each process with its own machine
  - Protection & security: make the environment safe
- System calls
  - ADT for the user applications
  - Standard I/O example
  - User-level libraries layered on top of system calls