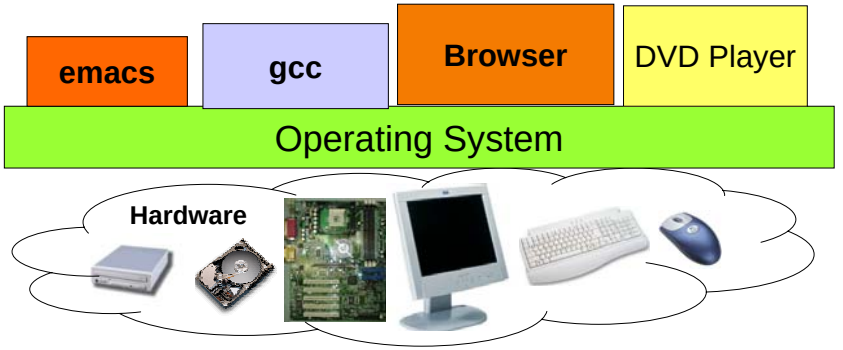
Operating Systems, System Calls, and Buffered I/O

CS 217

What Is Operating System?



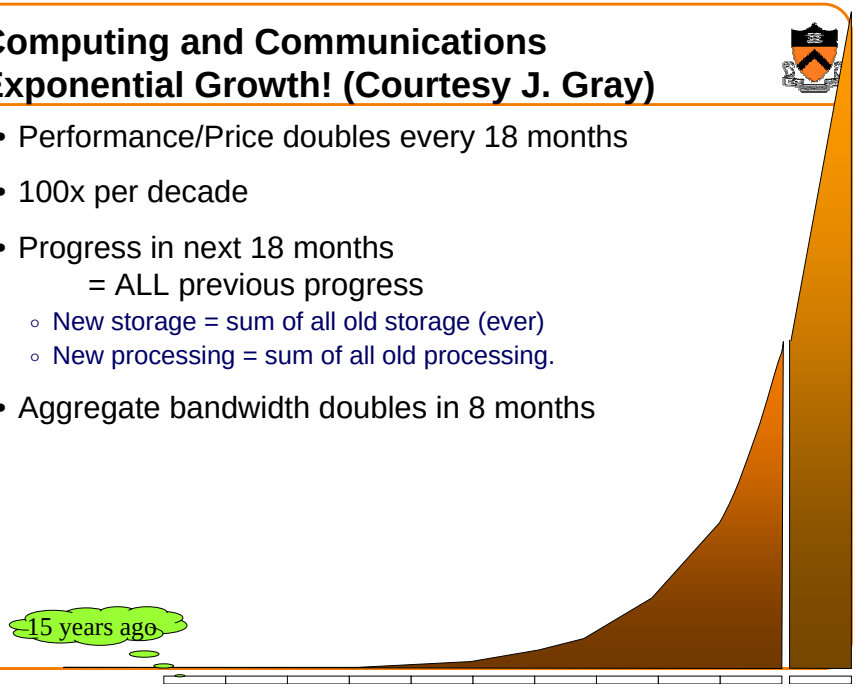
- Abstraction of hardware
- Virtualization
- Protection and security

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

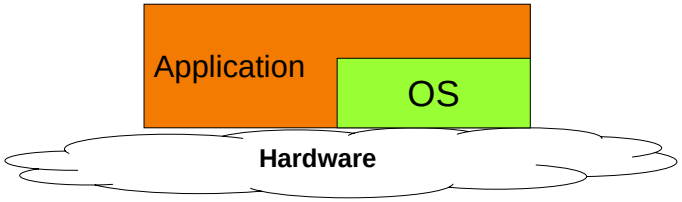
Computing and Communications Exponential Growth! (Courtesy J. Gray)

- 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



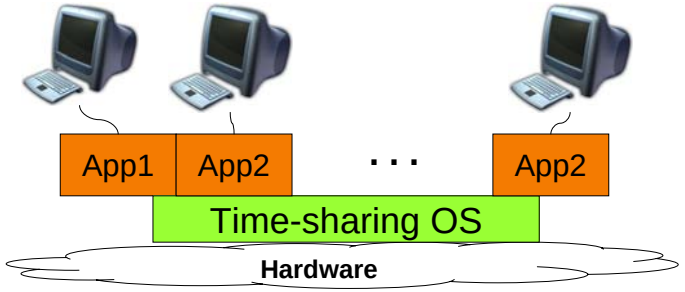
Phase 1: Hardware Expensive, Human Cheap

- User at console, OS as subroutine library
- Batch monitor (no protection): load, run, print
- Development
  - Data channels, interrupts; overlap I/O and CPU
  - DMA
  - Memory protection: keep bugs to individual programs
  - Multics: designed in 1963 and run in 1969
- Assumption: No bad people. No bad programs. Minimum interactions



Phase 2: Hardware Cheap, Human Expensive

- Use cheap terminals to share a computer
- Time-sharing OS
- Unix enters the mainstream
- Problems: thrashing as the number of users increases



Phase 3: HW Cheaper, Human More Expensive

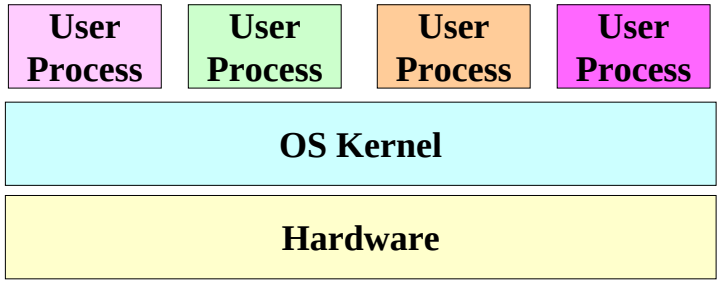
- Personal computer
  - Altos OS, Ethernet, Bitmap display, laser printer
  - Pop-menu window interface, email, publishing SW, spreadsheet, FTP, Telnet
  - Eventually >100M unites per year
- PC operating system
  - Memory protection
  - Multiprogramming
  - Networking

Phase 4: > 1 Machines per User

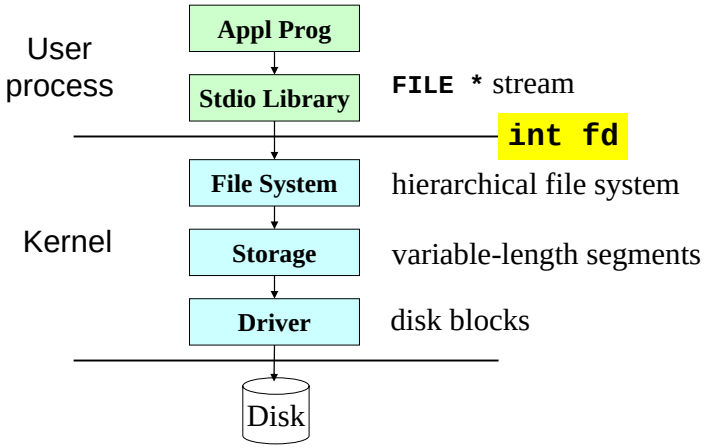
- Parallel and distributed systems
  - Parallel machine
  - Clusters
  - Network is the computer
- Pervasive computers
  - Wearable computers
  - Computers everywhere
- OS are general and specialized

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 its “own” machine (COS318)
- Protection & security: make the machine safe (COS318)

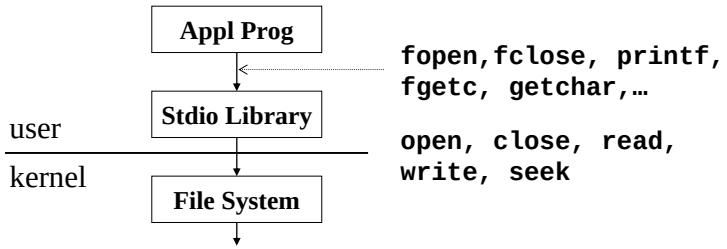


Layers of Abstraction



System Calls

- Kernel provided system services: “protected” procedure call



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

System Call Mechanism

- Processor modes
  - User mode: can execute normal instructions and access only user memory
  - Supervisor mode: can execute normal instructions, privileged instructions and access all of memory (e.g., devices)
- System calls
  - User cannot execute privileged instructions
  - Users must ask OS to execute them - system calls
  - System calls are often implemented using traps (**`int`**)
  - OS gains control through trap, switches to supervisor model, 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>

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

mode is the permissions to use if file must be created

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