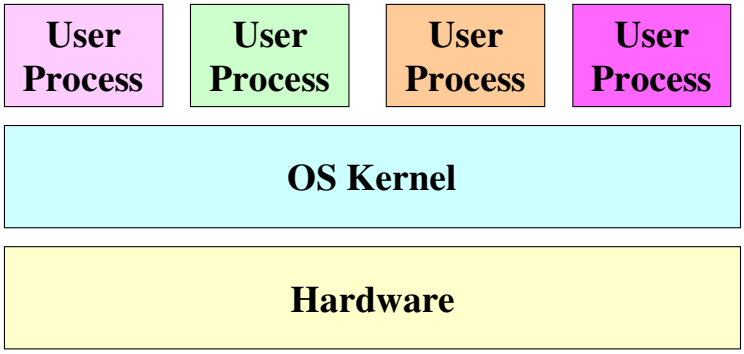
# Operating Systems

CS 217

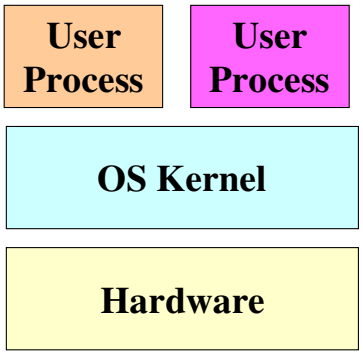
## Operating System (OS)

- Provides each process with a virtual machine
  - Promises each program the illusion of having whole machine to itself



## Operating System

- Coordinates access to physical resources
  - CPU, memory, disk, i/o devices, etc.
- Provides services
  - Protection
  - Scheduling
  - Memory management
  - File systems
  - Synchronization
  - etc.




## OS as Government

- Makes lives easy
  - Promises everyone whole machine (dedicated CPU, infinite memory, …)
  - Provides standardized services (standard libraries, window systems, …)
- Makes lives fair
  - Arbitrates competing resource demands
- Makes lives safe
  - Prevent accidental or malicious damage by one program to another

Randy Wang

## OS History

- Development of OS paradigms:
  - Phase 0: User at console
  - Phase 1: Batch processing
  - Phase 2: Interactive time-sharing
  - Phase 3: Personal computing
  - Phase 4: ?

Randy Wang

| | 1981 | 1999 | Factor |
|---|---|---|---|
| MIPS | 1 | 1000 | 1,000 |
| $/MIPS | $100K | $5 | 20,000 |
| DRAM Capacity | 128KB | 256MB | 2,000 |
| Disk Capacity | 10MB | 50GB | 5,000 |
| Network B/W | 9600b/s | 155Mb/s | 15,000 |
| Address Bits | 16 | 64 | 4 |
| Users/Machine | 10s | <= 1 | < 0.1 |

Computing price/performance affects OS paradigm

## Phase 0: User at Console

- How things work
  - One program running at a time
  - No OS, just a sign-up sheet for reservations
  - Each user has complete control of machine
- 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

Randy Wang

## Phase 1: Batch Processing

- How things work
  - Sort jobs and batch those with similar needs to reduce unnecessary setup time
  - Resident monitor provides "automatic job sequencing": it interprets "control cards" to automatically run a bunch of programs without human intervention
- Advantage
  - Good utilization of machine
- Disadvantagess
  - Loss of interactivity (unsolvable)
  - One job can screw up other jobs, need protection (solvable)

Randy Wang

**Good for expensive hardware and cheap humans**

## Phase 2: Interactive Time-Sharing

- How things work
  - Multiple users per single machine
  - OS with multiprogramming and memory protection
- Advantages:
  - Interactivity
  - Sharing of resources
- Disadvantages:
  - Does not always provide reasonable response time

Randy Wang

**Good for cheap hardware and expensive humans**

## Phase 3: Personal Computing

Randy Wang

- How things work
  - One machine per person
  - OS with multiprogramming and memory protection
- Advantages:
  - Interactivity
  - Good response times
- Disadvantages:
  - Sharing is harder

**Good for very cheap hardware and expensive humans**

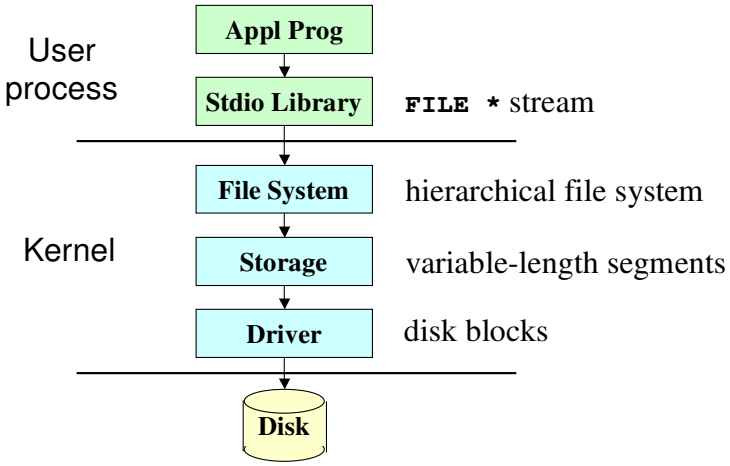
## Phase 4: What Next?

Randy Wang

- How will things work?
  - Many machines per person?
  - Ubiquitous computing?
- What type of OS?

**Good for very, very cheap hardware and expensive humans**

## Layers of Abstraction

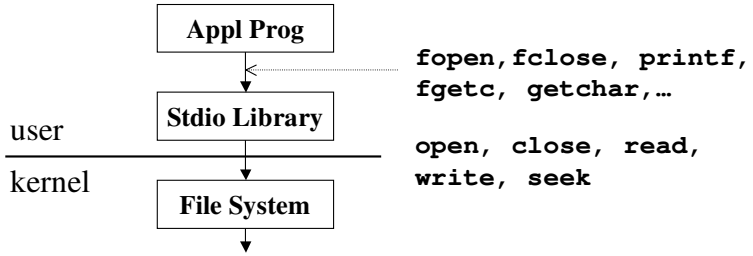



## System Calls

- Processor modes
  - user mode: can execute normal instructions and access only user memory
  - supervisor mode: can also execute 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
  - OS gains control through trap, switches to supervisor model, performs service, switches back to user mode, and gives control back to user

## System Calls

- Method by which user processes invoke kernel services: “protected” procedure call




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

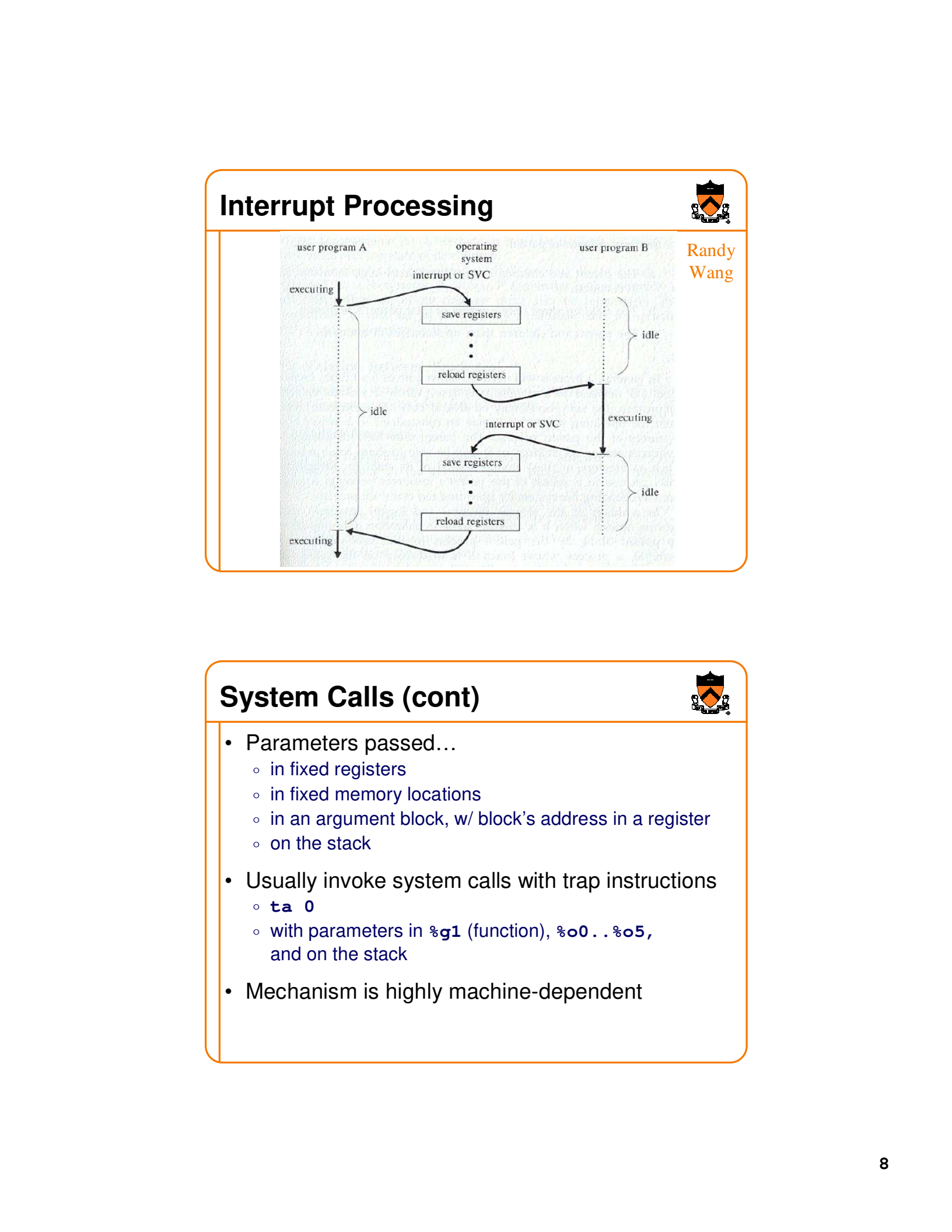
## Interrupt-Driven Operation

- Everything OS does is interrupt-driven
  - System calls use traps to interrupt

Randy Wang

- An interrupt stops the execution dead in its tracks, control is transferred to the OS
  - Saves the current execution context in memory (PC, registers, etc.)
  - Figures out what caused the interrupt
  - Executes a piece of code (interrupt handler)
  - Re-loads execution context when done, and resumes execution

## Interrupt Processing



Randy Wang




## System Calls (cont)



- Parameters passed…
  - in fixed registers
  - in fixed memory locations
  - in an argument block, w/ block's address in a register
  - on the stack
- Usually invoke system calls with trap instructions
  - `ta 0`
  - with parameters in `%g1` (function), `%o0..%o5,` and on the stack
- Mechanism is highly machine-dependent

## Read System Call

- Read call

```
nread = read(fd, buffer, n);
```

- Reads **n** bytes from **fd** into **buffer**
  - returns number of bytes read, or –1 if there's an error
- In the caller

```
mov fd,%o0
mov buffer,%o1
mov n,%o2
call read; nop
mov %o0,nread
```

## Read System Call (cont)

- User-side implementation (**libc**)

```
read: set 3,%g1
      ta 0
      bcc L1; nop
      set _errno,%g1
      st %o0,[%g1]
      set -1,%o0
 L1:  retl; nop
```

- Kernel-side implementation
  - sets the C bit if an error occurred
  - stores an error code in %o0 (see /usr/include/sys/errno.h)

## Write System Call

```
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)) > 0)
         p += n;
      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;
   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;
   n = 0;
   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;
```

## Summary

- OS virtualizes machine
  - Provides each process with illusion of having whole machine to itself
- OS provides services
  - Protection
  - Sharing of resources
  - Memory management
  - File systems
  - etc.
- Protection achieved through separate kernel
  - User processes uses system calls to ask kernel to access protected stuff on its behalf