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

```
User Process  User Process  User Process  User Process

OS Kernel

Hardware
```
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: ?

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
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

- Disadvantages
  - Loss of interactivity (unsolvable)
  - One job can screw up other jobs, need protection (solvable)

Good for expensive hardware and cheap humans

Randy Wang

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

Good for cheap hardware and expensive humans

Randy Wang
Phase 3: Personal Computing

- 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

Randy Wang

Phase 4: What Next?

- How will things work?
  - Many machines per person?
  - Ubiquitous computing?

- What type of OS?

Good for very, very cheap hardware and expensive humans

Randy Wang
Layers of Abstraction

User process

- Appl Prog
- Stdio Library
- File System
- Storage
- Driver
- Disk

Kernel

- hierarchical file system
- variable-length segments
- disk blocks

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

  ![Diagram]

  - User
  - Stdio Library
  - File System
  - Kernel
  - Appl Prog

  - System calls: `fopen, fclose, printf, fgetc, getchar,…`
  - System calls: `open, close, read, write, seek`

- 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

- 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

- 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

System Calls (cont)
**Read System Call**

- **Read call**
  
  \[
  \text{nread} = \text{read}(\text{fd}, \text{buffer}, n);
  \]

- **Reads n bytes from fd into buffer**
  
  \(\circ\) returns number of bytes read, or \(-1\) if there’s an error

- **In the caller**
  
  \[
  \begin{align*}
  \text{mov} & \text{ fd, }%0 \\
  \text{mov} & \text{ buffer, }%1 \\
  \text{mov} & \text{ n, }%2 \\
  \text{call} & \text{ read; nop} \\
  \text{mov} & \%0,\text{nread}
  \end{align*}
  \]

**Read System Call (cont)**

- **User-side implementation (libc)**
  
  \[
  \begin{align*}
  \text{read: set} & \ 3, %1 \\
  \text{ta} & \ 0 \\
  \text{bcc} & \ L1; \ \text{nop} \\
  \text{set} & \ _\text{errno}, %1 \\
  \text{st} & \ %0, [%1] \\
  \text{set} & \ -1, %0 \\
  L1: \ \text{retl}; \ \text{nop}
  \end{align*}
  \]

- **Kernel-side implementation**
  
  \(\circ\) sets the C bit if an error occurred
  
  \(\circ\) stores an error code in \%0
  
  (see /usr/include/sys/errno.h)
Write System Call

```c
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

```c
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

```c
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

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