Operating Systems, System Calls, and Buffered I/O

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
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: ?

<table>
<thead>
<tr>
<th></th>
<th>1981</th>
<th>1999</th>
<th>Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIPS</td>
<td>1</td>
<td>1000</td>
<td>1,000</td>
</tr>
<tr>
<td>S/MIPS</td>
<td>5100K</td>
<td>55</td>
<td>20,000</td>
</tr>
<tr>
<td>DRAM Capacity</td>
<td>128KB</td>
<td>256MB</td>
<td>2,000</td>
</tr>
<tr>
<td>Disk Capacity</td>
<td>10MB</td>
<td>50GB</td>
<td>5,000</td>
</tr>
<tr>
<td>Network B/W</td>
<td>9600b/s</td>
<td>155Mh/s</td>
<td>15,000</td>
</tr>
<tr>
<td>Address Bits</td>
<td>16</td>
<td>64</td>
<td>4</td>
</tr>
<tr>
<td>Users/Machine</td>
<td>10s</td>
<td>&lt;= 1</td>
<td>&lt; 0.1</td>
</tr>
</tbody>
</table>

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

• 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

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
Layers of Abstraction

User process

Kernel

- Disk blocks
- Variable-length segments
- Hierarchical file system

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
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 mode, performs service, switches back to user mode, and gives control back to user

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

Randy Wang
Interrupt-Driven Operation

- 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`

System Call Processing

- In your program:
  
  ```
  mov arg1,%o0
  mov arg2,%o1
  mov arg3,%o2
  call function; nop
  mov %o0, result
  ```

- In user-level library (`libc`)
  
  ```
  function: set id,%g1
  ta 0
  ret1;
  nop
  ```

- In kernel:
  - look at `%g1 to see what interrupt to process`
Interrupt Processing

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

SYNOPSIS

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

    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 . . .

flags examples:
    O_RDONLY
    O_WRONLY
    O_RDWR|O_CREAT

mode is the permissions to use if file must be created

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 SIZE_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 - read from 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

```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;
    else 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 ? \( 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 ~1975, 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 _jobuf
    Determine appropriate “flags” from “rw” parameter
    Call open to get the file descriptor
    Fill in the _jobuf appropriately
}

Stdio library

- fopen, fclose
- feof, ferror, fileno, fstat
  - status inquiries
- fflush
  - make outside world see changes to buffer
- fgetc, fgets, fread
- fprintf, fscanf
- 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 virtualizes machine
  - Provides each process with illusion of having whole machine to itself
- OS provides services
  - Protection
  - Sharing of resources
  - Memory management
  - File systems
- Protection achieved through separate kernel
  - User processes uses system calls to ask kernel to access protected stuff on its behalf
- User level libraries layered on top of kernel interface