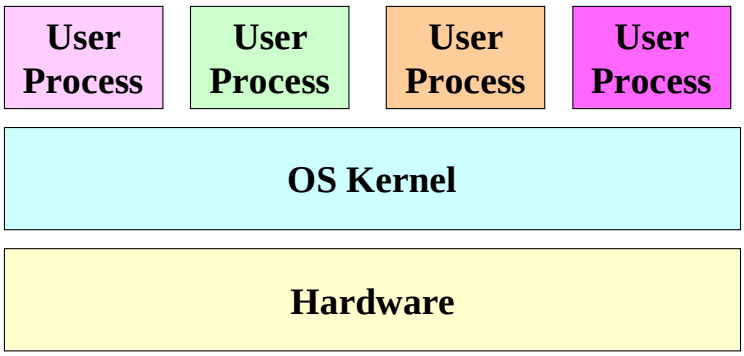
Operating Systems, System Calls, and Buffered I/O

CS 217

1

Operating System (OS)

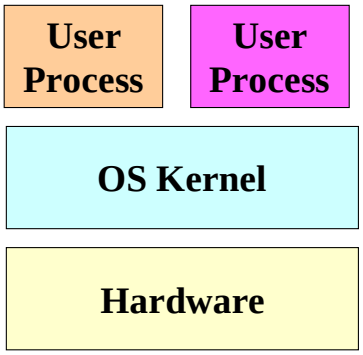
- Provides each process with a virtual machine
  - Promises each program the illusion of having whole machine to itself



2

Operating System

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



3

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

4

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

5

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

6

Phase 1: Batch Processing

- How things work — Randy Wang
  - 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)

Good for expensive hardware and cheap humans

7

Phase 2: Interactive Time-Sharing

- How things work — Randy Wang
  - 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

8

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

Randy Wang

Good for very cheap hardware and expensive humans

9

Phase 4: What Next?

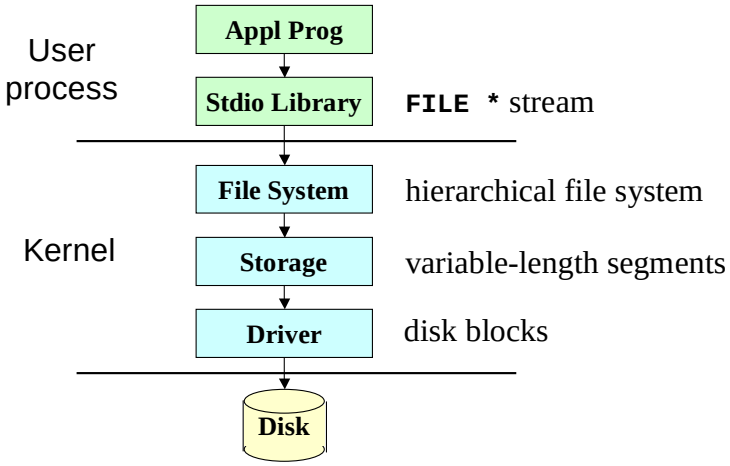
- How will things work?
  - Many machines per person?
  - Ubiquitous computing?
- What type of OS?

Randy Wang

Good for very, very cheap hardware and expensive humans

10

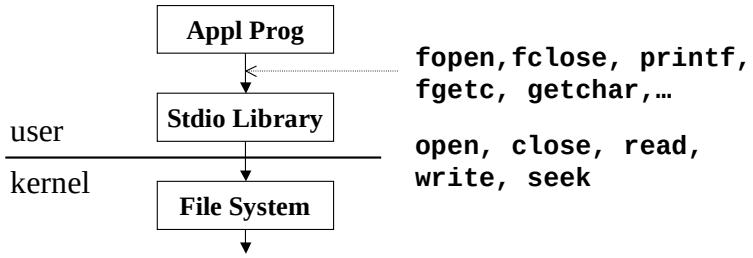
Layers of Abstraction



11

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

12

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

13

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

14

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

15

System Call Processing

- In your program:

```
mov arg1,%o0
mov arg2,%o1
mov arg3,%o2
call function; nop
mov %o0, result
```

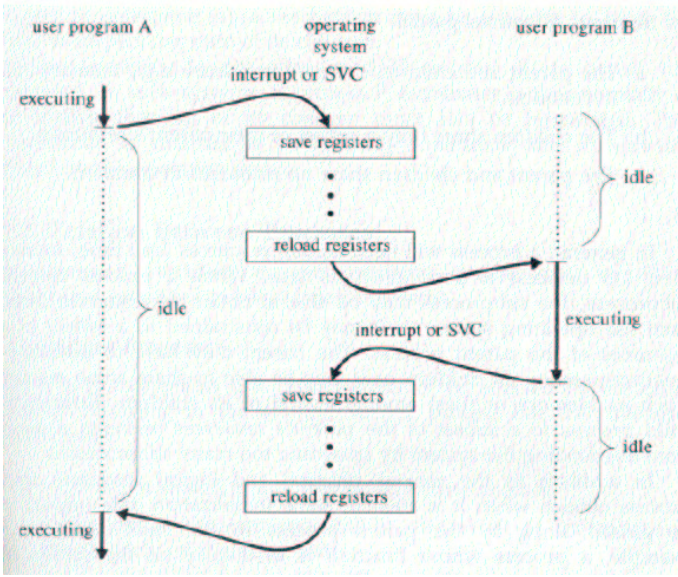
- In user-level library (**`libc`**)

```
function: set id,%g1
          ta 0
          retl;
          nop
```

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

16

Interrupt Processing



Randy Wang

17

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 ...

18

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);
```

flags examples:
O_RDONLY
O_WRITE|O_CREATE

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 . . .

19

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) . . . .

20

`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.

21

`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.

22

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)) > 0)
         p += n;
      else
         perror("safe_write:");
   }
   return nbytes;
}
```

23

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;
}
```

24

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();
}
```

25

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;
```

26

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

27

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

```
}
```

28

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.

29

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

30