Introduction to OSs

- What **is** an Operating System?
- Architectural Support for Operating Systems
- System Calls
- Basic Organization of an Operating System

Introduction to OSs

- What **is** an Operating System?
- Architectural Support for Operating Systems
- System Calls
- Basic Organization of an Operating System

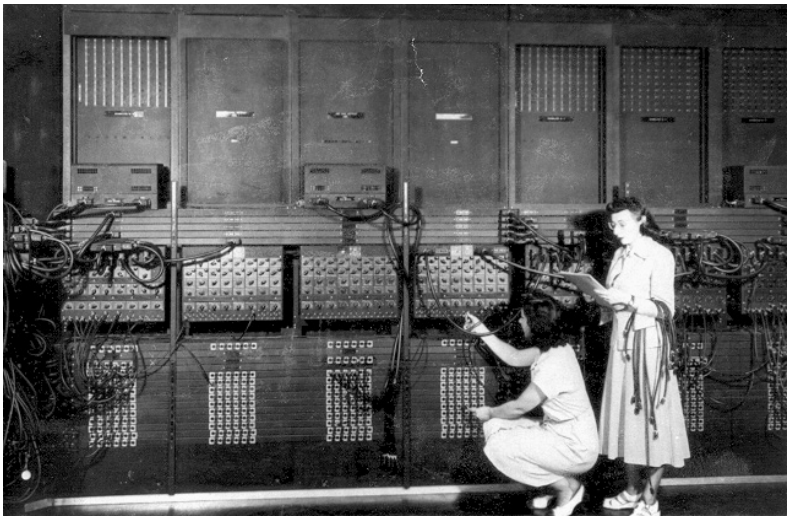
What is an operating system?

- What an operating system is **not**:
  - An o.s. is **not** a language or a compiler
  - An o.s. is **not** a command interpreter / window system
  - An o.s. is **not** a library of commands
  - An o.s. is **not** a set of utilities

A Short Historical Tour

- **First Generation** Computer Systems (1949-1956):
  - Single user: writes program, operates computer through console or card reader / printer
  - Absolute machine language
  - I/O devices
  - Development of libraries; device drivers
  - Compilers, linkers, loaders
  - Relocatable code

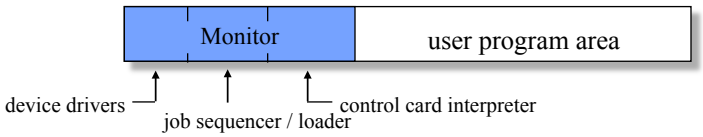
Programming Early Machines



Wiring the ENIAC with a new program
(U.S. Army photo, from archives of the ARL Technical Library)

Second-Generation Computers (1956-1963)

- Problems: scheduling, setup time
- Automation of Load/Translate/Load/Execute
  - Batch systems
  - Monitor programs



  - Job Control Language
  - Advent of operators: computers as input/output box
- Problem: Resource management and I/O still under control of programmer
  - Memory protection
  - Timers
  - Privileged instructions

Example: IBM Punch Card System

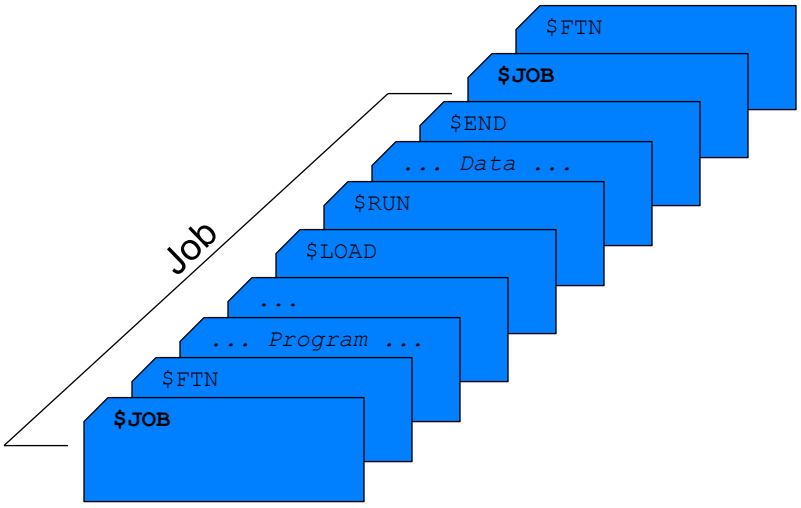
Card Punch

Card Verifier

Card Sorter

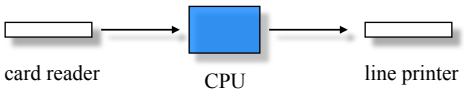
(Computer Museum of America)

Batching Program Execution

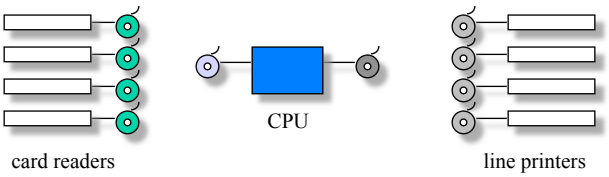


Overlapping CPU and I/O Operations

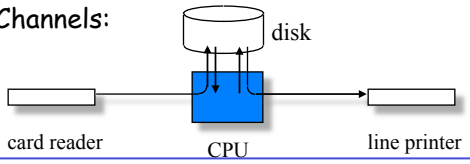
Traditional Batch Operation:



Off-Line Processing:

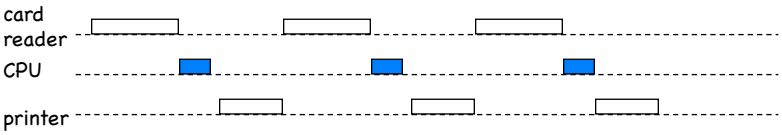


Spooling; I/O Channels:

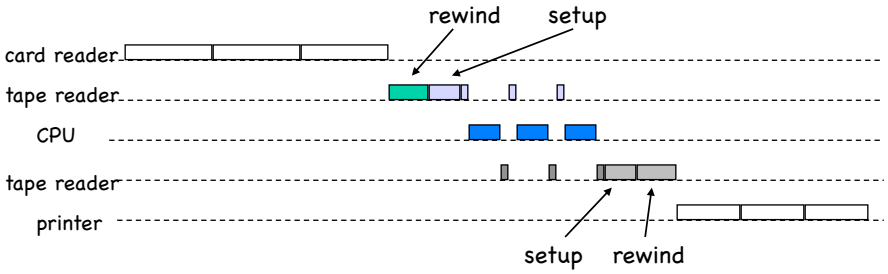


Off-Line vs. Pure Batch

- batch

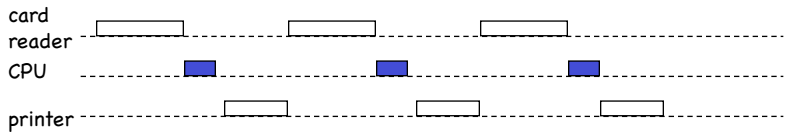


- off-line (single set of card reader/printer)

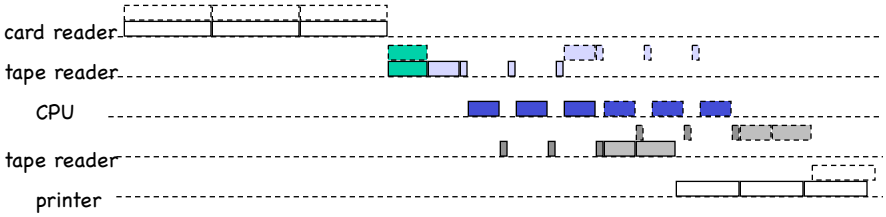


Off-Line vs. Pure Batch (II)

- batch

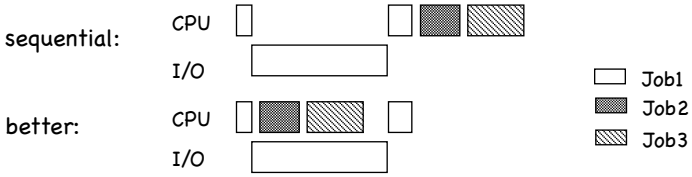


- off-line (multiple of card readers/printers)

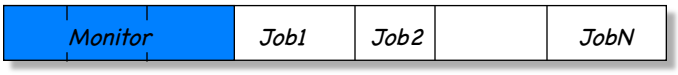


Third-Generation Computer Systems (1964-1975)

- Problem with batching: one-job-at-a-time



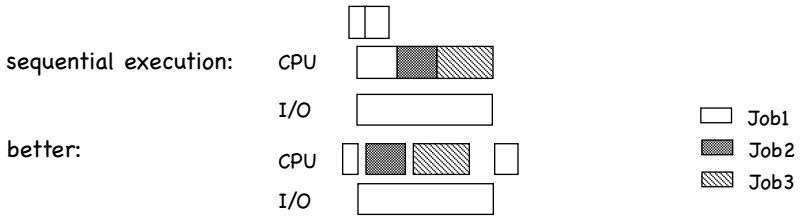
- Solution: **Multiprogramming**
  - Job pools: have several programs ready to execute
  - Keep several programs in memory



- New issues:
  - Job scheduling
  - Memory management
  - Protection

Multiprogramming

- Problem with batching: one-job-at-the-time



Time Sharing (mid 1960s on)

- Remote interactive access to computer: "Computing as Utility"
- OS interleaves execution of multiple user programs with time quantum
  - CTSS (1961): time quantum 0.2 sec
- User returns to own the machine
- New aspects and issues:
  - On-line file systems
  - resource protection
  - virtual memory
  - sophisticated process scheduling
- Advent of systematic techniques for designing and analyzing OSs.

The Recent Past

- Personal computers
  - History repeats itself
- Parallel systems
  - Resource management
  - Fault tolerance
- Real-Time Systems
- Distributed Systems
  - Communication
  - Resource sharing
  - Network operating systems
  - Distributed operating systems
- Secure Systems
- Virtualization
- Data Centers

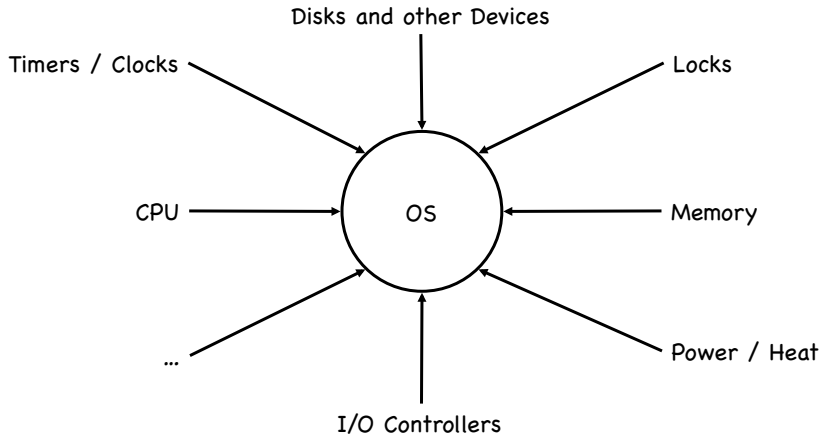
The Future?

- The "Invisible Computer"
- Computing-in-the-ultra-small
- Speed vs. Power vs. Heat
- Breaking up the layered design

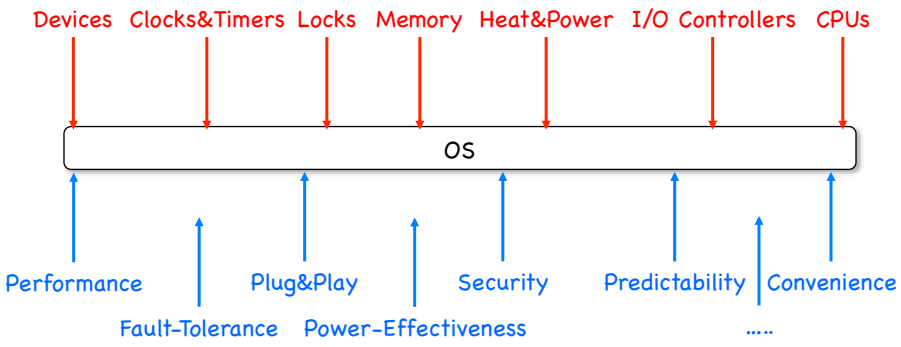
What, then, is an Operating System?

- The OS controls and coordinates the use of system resources.
- Primary goal: Provide a convenient environment for a user to access the available resources (CPU, memory, I/O)
  - Provide appropriate abstractions (files, processes, ...)
  - "virtual machine"
- Secondary goal: Efficient operation of the computer system.
- Resource Management
  - **Transforming:** Create virtual substitutes that are easier to use.
  - **Multiplexing:** Create the illusion of multiple resources from a single resource
  - **Scheduling:** "Who gets the resource when?"

Resources



The OS as Servant to Two Masters



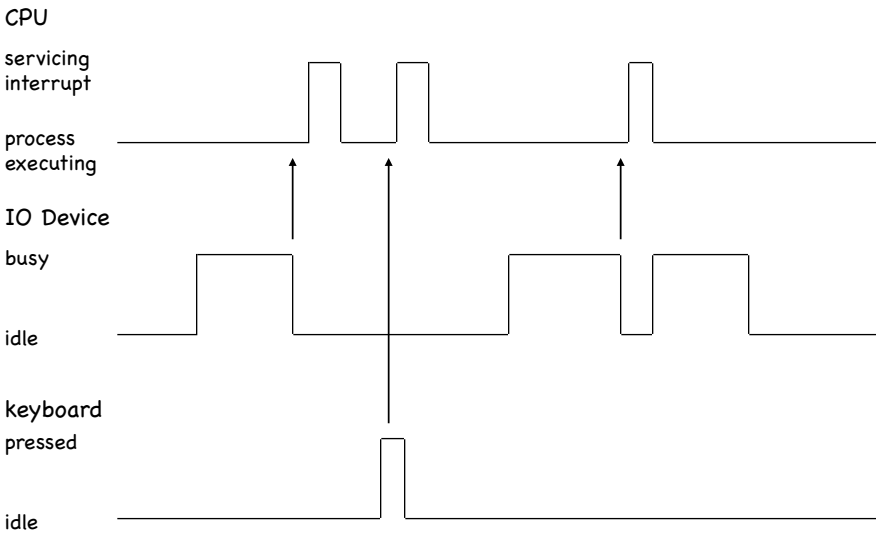
Introduction to OSs

- What is an Operating System?
- Architectural Support for Operating Systems
- System Calls
- Basic Organization of an Operating System

Architectural Support for OS's

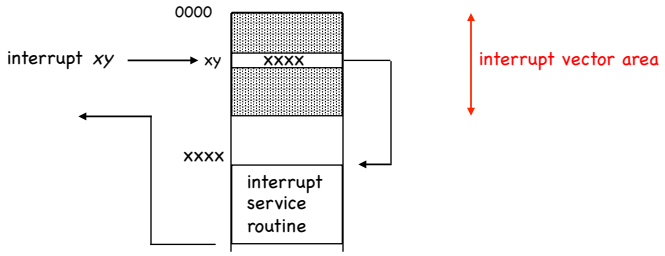
- Dealing with Asynchronous Events: Exceptions, Interrupts
  - Modern OS's are interrupt-driven (some still are not!).
  - Simple interrupt handling vs. exception handling MIPS-style.
- Hardware Protection
  - Privilege Levels (e.g. user/kernel/supervisor, etc.)
  - Priviledged instructions: typically CPU control instructions
  - I/O Protection
  - Memory Protection
- Support for Address Spaces
- Timers

Modern OS's are Interrupt-Driven



Interrupts / Exceptions

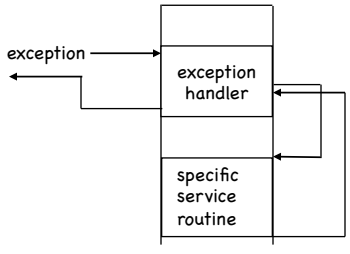
- When an interrupt occurs, CPU stops, saves state, typically changes into supervisor mode, and immediately jumps to predefined location.
- Appropriate interrupt service routine is found through the *interrupt vector*.
- Return-from-interrupt automatically restores state.



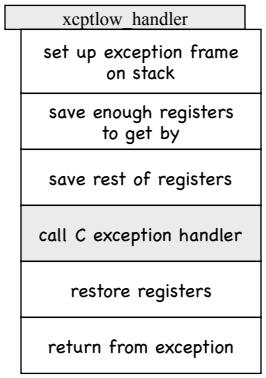
- Interrupts/Exceptions can be invoked by *asynchronous events* (I/O devices, timers, various errors) or can be *software-generated* (system calls).

Exceptions, MIPS-Style

- MIPS CPU deals with exceptions.
  - Interrupts are just a special case of exceptions.
- The MIPS Architecture has no interrupt-vector table!
  - All exceptions trigger a jump to the same location, and de-multiplexing happens in the exception handler, after looking up the reason for the exception in the **CAUSE** register.



MIPS Exception Handler (low-level)

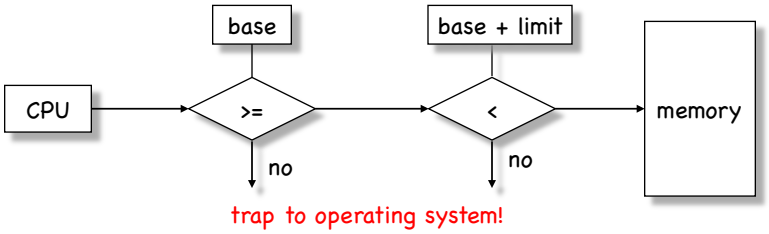


Hardware Protection

- Originally: User owned the machine, no monitor. No protection necessary.
- Resident monitor, resource sharing: One program can adversely affect the execution of others.
- Examples
  - **halt** and other instructions
  - modify data or code in other programs or monitor itself
  - access/modify data on storage devices
  - refuse to relinquish processor
- Benign (bug) vs. malicious (virus)

Hardware Protection (2)

- Dual-mode operation
  - *user mode* vs. *supervisor mode*
  - e.g. **halt** instruction is privileged.
- I/O Protection
  - define all I/O operations to be privileged
- Memory Protection
  - protect interrupt vector, interrupt service routines
  - determine legal address ranges



Timers

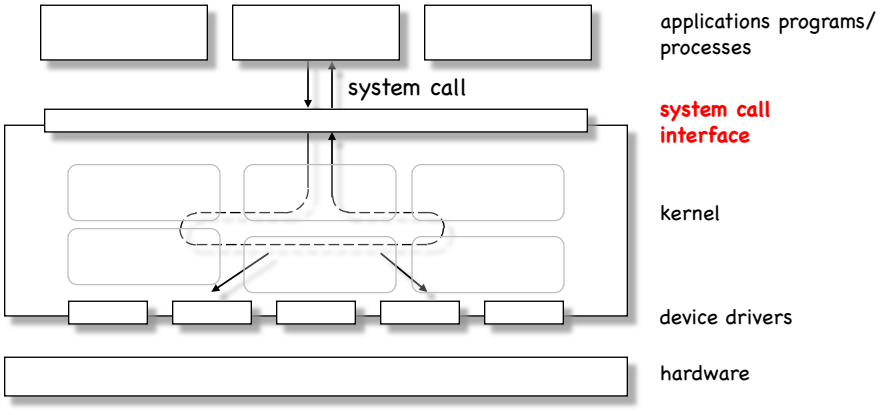
- Timers can be set, and a trap occurs when the timer expires. (And OS acquires control over the CPU.)
- Other uses of timers:
  - time sharing
  - time-of-day

Introduction to OSs

- What **is** an Operating System?
- Architectural Support for Operating Systems
- System Calls
- Basic Organization of an Operating System

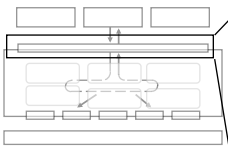
External Structure of an OS

The outsider's view of the OS.



System Calls

Provide the interface between a process and the OS.



Example: vanilla **copy**:

```
int copy(char * fname1, *fname2) {
  FILE *f, *g;
  char c;
  f = fopen(fname1, "r");
  g = fopen(fname2, "w");
  while (read(f, &c, 1) > 0)
      write(g, c, 1);
  fclose(f);
  fclose(g);
}
```

System Call Implementation: Linux on x86

- Example: `_syscall(int, setuid, uid_t, uid)`
- expands to:

```
_setuid:
  subl $4,%exp
  pushl %ebx
  movzwl 12(%esp),%eax
  movl %eax,4(%esp)
  movl $23,%eax   <<<---- System Call number (setuid = 23)
  movl 4(%esp),%ebx
  int $0x80       <<<---- call transfer to kernel entry point _system_call()
  movl %eax,%edx
  testl %edx,%edx
  jge L2
  negl %edx
  movl %edx,_errno
  movl $-1,%eax
  popl %ebx
  addl $4,%esp
retL2:
  movl %edx,%eax
  popl %ebx
  addl $4,%esp
  ret
```

Why Interrupts?

Reason 1: Can load user program into memory without knowing exact address of system procedures

Reason 2: Separation of address space, including stacks: *user stack* and *kernel stack*.

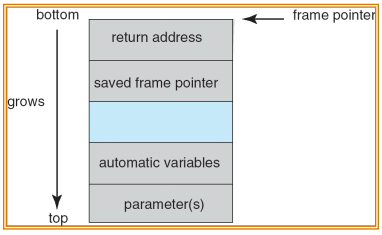
Reason 3: Automatic change to *supervisor mode*.

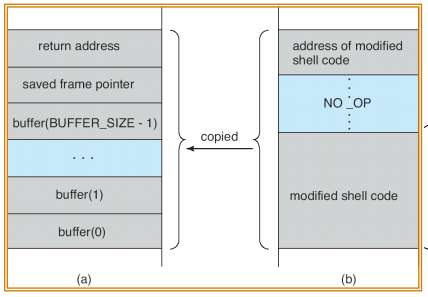
Reason 4: Can control *access* to kernel by masking interrupts.

Reason2: Buffer Overrun Attacks (Silberschatz et al)

[Example and illustrations from Silberschatz et al. "Operating Systems Concepts" Ch. 15]

```
#include <stdio.h>
#define BUFFER SIZE 256
int main(int argc, char *argv[])
    {
    char buffer[BUFFER SIZE];
    if (argc < 2)
          return -1;
    else {
          strcpy(buffer,argv[1]);
          return 0;
    }
}
```



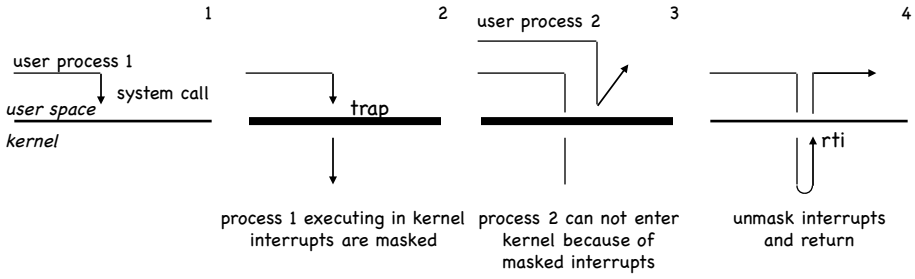



```
#include <stdio.h>
int main(int argc, char *argv[])
{
    execvp(''\bin\sh'',''\bin \sh'', NULL);
    return 0;
}
```

Stack Separation sufficient?

- Buffer overruns in kernel code?
- Device drivers?

Reason 4: Mutual Exclusion in Kernel

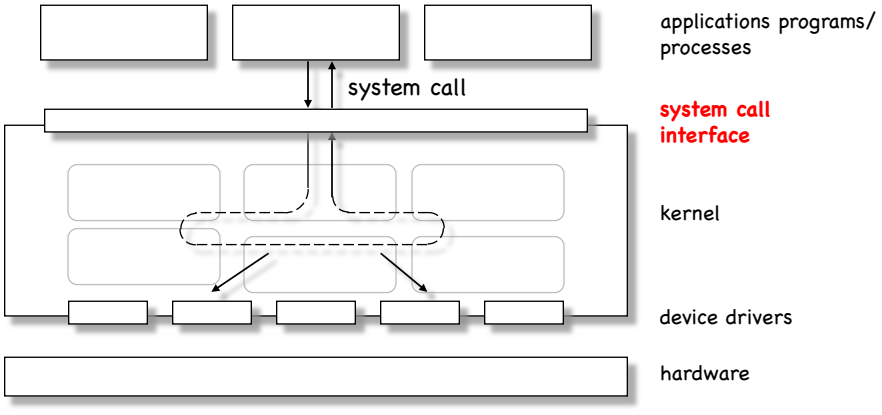


Introduction to OSs

- What **is** an Operating System?
- Architectural Support for Operating Systems
- System Calls
- Basic Organization of an Operating System

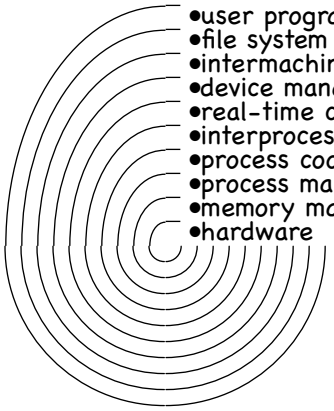
External Structure of an OS

The outsider's view of the OS.



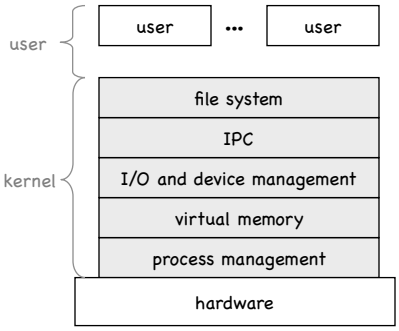
Internal Structure: Layered Services

The insider's view of the OS.
Example: XINU [Comer 1984]



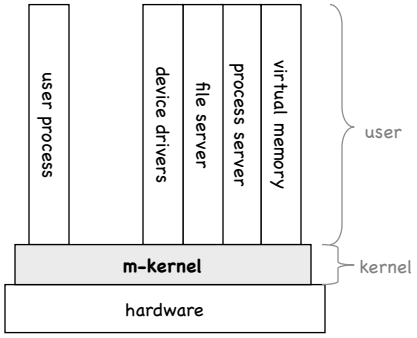
Internal Structure: μ-Kernels

- Layered Kernel



Hierarchical decomposition.
Interaction only between adjacent layers.

- Microkernels



Kernel has only core operating system functions (memory management, IPC, I/O, interrupts)
Other functions run in *server processes* in user space.

Operations in a μ-Kernel

- Non-kernel components of the OS are implemented as server processes.
- Communication between user and servers using messages through kernel.
- "client-server architecture within a single computer"
- Examples: Mach, Windows NT/2000/XP/7/8/10/..., Chorus, L4, ...

Windows System Structure

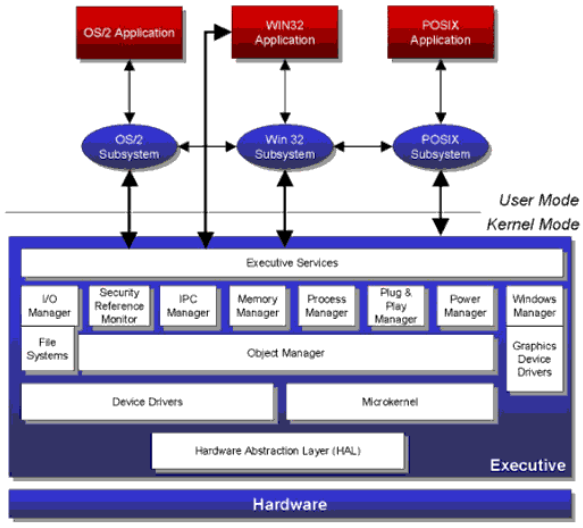
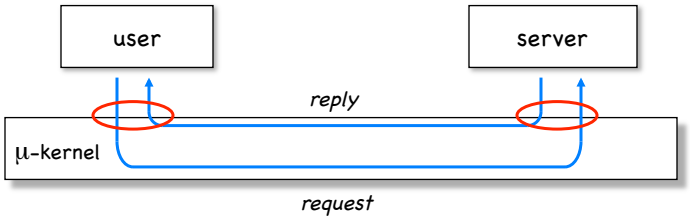


Figure 2.– System architecture

Benefits of μ-Kernels

- Extensibility:
  - New services can be added by adding server processes.
- Flexibility:
  - Services can be customized.
- Portability:
  - Kernel small, with well-defined interface.
- Distributed System Support:
  - Interface between users and services is message-based.

μ-Kernels: Performance is Problem



- Request traverses user/kernel boundary twice, same for reply.
- Solutions:
  - Move critical services back into the kernel ("make kernel bigger")
  - Make kernel "smaller"

Why are OSs so Slow?

(Why Aren't Operating Systems Getting Faster As Fast As Hardware? *John Ousterhout*, 1989)

Hardware	Abbreviation	RISC/CISC	MIPS
MIPS M2000	M2000	RISC	20
DECstation 3100	DS3100	RISC	13
Sun-4/280	Sun4	RISC	9
VAX 8800	8800	CISC	6
Sun-3/75	Sun3	CISC	1.8
Microvax II	MVAX2	CISC	0.9

Table 1: Hardware Platforms

Configuration	Time (microseconds)	MIPS-Relative Speed
M2000 RISC/os 4.0	18	0.54
DS3100 Sprite	26	0.49
DS3100 Ultrix 3.1	25	0.60
8800 Ultrix 3.0	28	1.15
Sun4 SunOS 4.0	32	0.68
Sun4 Sprite	32	0.58
Sun3 Sprite	92	1.0
Sun3 SunOS 3.5	108	1.0
MVAX2 Ultrix 3.0	207	0.9

Table 2: Getpid kernel call time

Why are OSs so Slow? (2)

Hardware	Abbreviation	RISC/CISC	MIPS
MIPS M2000	M2000	RISC	20
DECstation 3100	DS3100	RISC	13
Sun-4/280	Sun4	RISC	9
VAX 8800	8800	CISC	6
Sun-3/75	Sun3	CISC	1.8
Microvax II	MVAX2	CISC	0.9

Table 1: Hardware Platforms

Configuration	Time (ms)	MIPS-Relative Speed
M2000 RISC/os 4.0	0.30	0.71
DS3100 Ultrix 3.1	0.34	0.96
DS3100 Sprite	0.51	0.65
8800 Ultrix 3.0	0.70	1.0
Sun4 SunOS 4.0	1.02	0.47
Sun4 Sprite	1.17	0.41
Sun3 SunOS 3.5	2.36	1.0
Sun3 Sprite	2.41	1.0
MVAX2 Ultrix 3.0	3.66	1.3

Table 3: Cswitch: echo one byte between processes using pipes.