Lecture 5:

Operating Systems: The Basics

`www.cl.cam.ac.uk/Teaching/2001/OSFounds/`

Lecture 5: Monday 15th October 2001

Today's Lecture

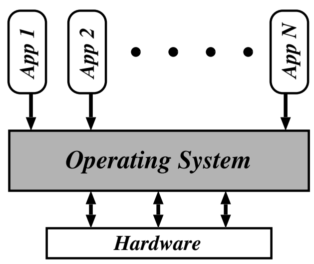
Today we'll cover:

- What is an OS?
  - Abstract view,
  - Historical perspective,
  - Kernel vs Microkernel design,
  - OS functions.
- What hardware support do we need?
  - Dual mode operation,
  - I/O and memory protection,
  - CPU protection.

What is an Operating System?

- A program that controls the execution of all other programs (applications).
- Acts as an intermediary between the user(s) and the computer.
- Objectives:
  - convenience,
  - efficiency,
  - extensibility.
- *Similar to a government. . .* :-)
- Wide variety: Unix (Solaris, Linux, Irix, OSF/1), Windows 9x/2000/XP, MacOS, BeOS, etc.

An Abstract View



- The Operating System (OS):
  - controls all execution.
  - multiplexes resources between applications.
  - abstracts away from complexity.
- Typically also have some *libraries* and some *tools* provided with OS.
- Are these part of the OS? Is IE4 a tool?
  - no-one can agree. . .
- For us, the OS ≈ the **kernel**.

In The Beginning. . .

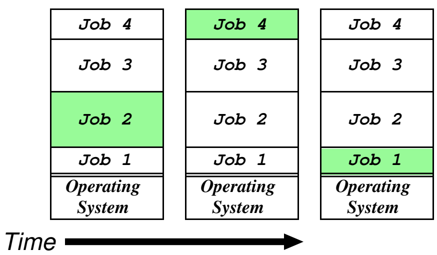
- 1949: First stored-program machine (EDSAC)
- to ∼ 1955: "Open Shop".
  - large machines with vacuum tubes.
  - I/O by paper tape / punch cards.
  - user = programmer = operator.
- To reduce cost, hire an **operator**:
  - programmers write programs and submit tape/cards to operator.
  - operator feeds cards, collects output from printer.
- Management like it.
- Programmers hate it.
- Operators hate it.

⇒ need something better.

Batch Systems

- Introduction of tape drives allow **batching** of jobs:
  - programmers put jobs on cards as before.
  - all cards read onto a tape.
  - operator carries input tape to computer.
  - COMPUTE!
  - results written to output tape.
  - output tape taken to printer.
- Computer now has a **resident monitor**:
  - initially control is in monitor.
  - monitor reads job and transfers control.
  - at end of job, control transfers back to monitor.
- Even better: **spooling systems**.
  - use interrupt driven I/O.
  - use magnetic disk to cache input tape.
  - *fire operator!*
- Monitor now **schedules** jobs. . .

Multi-Programming

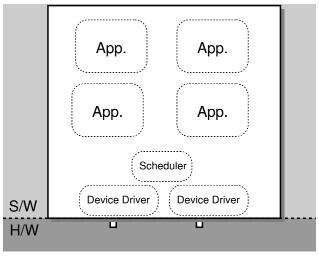


- Use memory to cache jobs from disk ⇒ more than one job active simultaneously.
- Two stage scheduling:
  1. select jobs to load: **job scheduling**.
  2. select resident job to run: **CPU scheduling**.
- Users want more interaction ⇒ **time-sharing**:
- e.g. CTSS, TSO, Unix, VMS, Windows NT. . .

Today and Tomorrow

- Single user systems: cheap and cheerful.
  - personal computers.
  - no other users ⇒ ignore protection.
  - e.g. DOS, Windows, Win 95/98, . . .
- RT Systems: power is nothing without control.
  - hard-real time: nuclear reactor safety monitor.
  - soft-real time: mp3 player.
- Parallel Processing: the need for speed.
  - SMP: 2–8 processors in a box.
  - MIMD: super-computing.
- Distributed computing: global processing?
  - Java: the network is the computer.
  - Clustering: the network is the bus.
  - CORBA: the computer is the network.
  - .NET: the network is an enabling framework. . .

Monolithic Operating Systems



- Oldest kind of OS structure ("modern" examples are DOS, original MacOS)
- **Problem**: applications can e.g.
  - trash OS software.
  - trash another application.
  - hoard CPU time.
  - abuse I/O devices.
  - etc. . .
- No good for fault containment (or multi-user).
- Need a better solution. . .

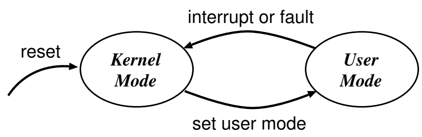
Dual-Mode Operation

- Want to stop buggy (or malicious) program from doing bad things.

$\Rightarrow$ provide **hardware** support to differentiate between (at least) two modes of operation.

1. User Mode : when executing on behalf of a user (i.e. application programs).
2. Kernel Mode : when executing on behalf of the operating system.

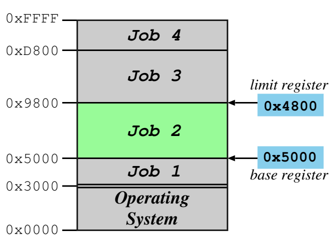
- Hardware contains a **mode-bit**, e.g. 0 means kernel, 1 means user.



- Make certain machine instructions only possible in kernel mode. . .

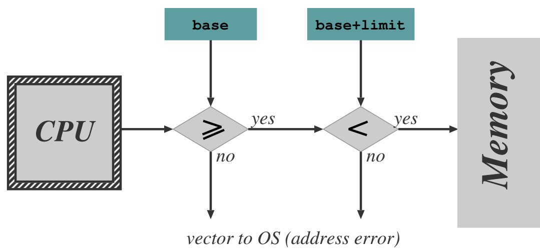
Protecting I/O & Memory

- First try: make I/O instructions **privileged**.
  - applications can't mask interrupts.
  - applications can't control I/O devices.
- But:
  1. Application can rewrite interrupt vectors.
  2. Some devices accessed via *memory*
- Hence need to protect memory also. . .
- e.g. define a **base** and a **limit** for each program.



- Accesses outside allowed range are protected.

Memory Protection Hardware



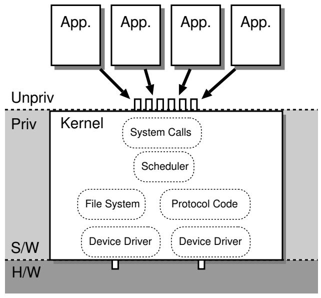
- Hardware checks *every* memory reference.
- Access out of range $\Rightarrow$ vector into operating system (just as for an interrupt).
- Only allow *update* of base and limit registers in kernel mode.
- Typically disable memory protection in kernel mode (although a bad idea).
- In reality, more complex protection h/w used:
  - main schemes are **segmentation** and **paging** (covered later on in course)

Protecting the CPU

- Need to ensure that the OS stays in control.
  - i.e. need to prevent any given application from 'hogging' the CPU the whole time.

  ⇒ use a **timer** device.
- Usually use a **countdown** timer, e.g.
  1. set timer to initial value (e.g. `0xFFFF`).
  2. every *tick* (e.g. $1\mu s$), timer decrements value.
  3. when value hits zero, interrupt.
- (Modern timers have programmable tick rate.)
- Hence OS gets to run periodically and do its stuff.
- Need to ensure only OS can load timer, and that interrupt cannot be masked.
  - use same scheme as for other devices.
  - (viz. privileged instructions, memory protection)
- Same scheme can be used to implement time-sharing (more on this later).

Kernel-Based Operating Systems

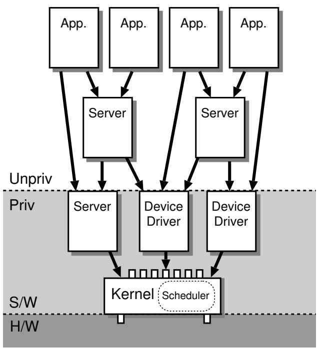


- Applications can't do I/O due to protection

  ⇒ operating system does it on their behalf.
- Need secure way for application to invoke operating system:

  ⇒ require a special (unprivileged) instruction to allow transition from user to kernel mode.
- Generally called a **software interrupt** since operates similarly to (hardware) interrupt. . .
- Set of OS services accessible via software interrupt mechanism called **system calls**.

Microkernel Operating Systems



- Alternative structure:
  - push some OS services into **servers**.
  - servers may be privileged (i.e. operate in kernel mode).
- Increases both *modularity* and *extensibility*.
- Still access kernel via system calls, but need new way to access servers:

  ⇒ interprocess communication (IPC) schemes.

Kernels versus Microkernels

So why isn't everything a microkernel?

- Lots of IPC adds overhead

  ⇒ microkernels usually perform less well.
- Microkernel implementation sometimes tricky: need to worry about synchronisation.
- Microkernels often end up with redundant copies of OS data structures.

Hence today most common operating systems blur the distinction between kernel and microkernel.

- e.g. linux is "kernel", but has kernel modules and certain servers.
- e.g. Windows NT was originally microkernel (3.5), but now (4.0 onwards) pushed lots back into kernel for performance.
- Still not clear what the best OS structure is, or how much it really matters. . .

Operating System Functions

- Regardless of structure, OS needs to **securely multiplex resources**, i.e.
  1. protect applications from each other, yet
  2. share physical resources between them.
- Also usually want to *abstract* away from grungy hardware, i.e. OS provides a **virtual machine**:
  - share CPU (in time) and provide each application with a virtual processor,
  - allocate and protect memory, and provide applications with their own virtual address space,
  - present a set of (relatively) hardware independent virtual devices, and
  - divide up storage space by using filing systems.
- Remainder of this part of the course will look at each of the above areas in turn. . .

Summary

You should now understand:

- What an OS is (abstractly),
- The historical evolution of OS,
- Hardware support needed:
  - Dual mode operation,
  - I/O and memory protection,
  - CPU protection.
- Different approaches to kernel design:
  - Microkernel vs. kernel

Next lecture: Processes

Background Reading: Silberschatz et al.:

- Chapter 1 – History and basics of OS
- Section 2.5 – Hardware protection
- Section 3.5.3 – Microkernel design