CS 126 Lecture S2: Operating Systems

Outline

- **Introductions**
- History
- General mechanisms
- Process management
- Memory management
- File systems
- Conclusions

Why Learn About OS

- Be an **informed citizen** in the age of hype, controversies, and lawyer talks
- Learn something about a big part of your **daily computing** life
- Gain an appreciation of "the big picture"
  - In terms of the crucial role of **technology advance**, and
  - In terms of **synthesis** of many areas of computer science: hardware, algorithms, language, and ...
- Gain some insight into how to put together arguably one of the most **challenging softwares**

OS as Government

- Everyone learns to hate it, but you will miss it dearly if it's not there
- Makes lives **easy**: **virtualizing resources**: promises everyone illusions of
  - separate dedicated **CPUs** (using a single CPU)
  - unlimited amount of **memory** (using limited physical memory)
  - directories and files (using **disk** blocks)
- Makes lives **easy**: providing standard **services**:
  - development environment
  - standard libraries
  - window systems
- Makes lives **fair**: arbitrate competing resource demands
- Makes lives **safer**: prevent accidental or malicious damage/intrusion
- A good way of understanding OS is to look at the history of where they come from... (We keep going back to the future!)

Phase 0: User at Console

- ***How things work***
  - One TOY machine for CS126, what do we do?
  - No OS, just a sign-up sheet for reservations!
  - Each user has complete control of machine
  - Soon added device libraries, compilers, assemblers for convenience
- ***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

Outline

- ~~Introductions~~
- History
- General mechanisms
- Process management
- Memory management
- File systems
- Conclusions

Phase 2: Interactive Time-Sharing

(Cheap Hardware, Expensive Humans)

- ***How things work***
  - Multiple cheap terminals for multiple users per single machine
  - OS keeps multiple programs active at the same time and switches among them rapidly to provide the illusion of one machine per user
- ***Advantage***: interactivity, sharing (collaboration)
- ***Problems***
  - Must provide reasonable response time (hard sometimes)
  - Must provide human friendly interfaces: command shell, hierarchical name structure for file systems, etc. (solvable)
  - Higher degree of multiprogramming places heavier demand on protection mechanism (solvable but hard)

Phase 1: Batch Processing

(Expensive Hardware, Cheap Humans)

- ***How things work***
  - Sort jobs and batch those with similar needs to reduce unnecessary setup time
  - A 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, (jargon: high throughput: jobs per second)
- ***Problems***
  - Loss of interactivity (unsolvable)
  - One job can screw up other jobs, need protection (solvable)

Phase 3: Personal Computing

(Very Cheap Hardware, Very Expensive Humans)

- ***How things work***
  - One machine per person, now several machines per person
  - Initially, OS goes back to “square 1” (like those of Phase 0)
  - Later added back multiprogramming and memory protection
- ***Advantages***
  - Better response time
  - Protection becomes a little easier
- ***Problems***
  - How do you share information? (sill not solved)
- ***What’s next? Networked ubiquitous computing?***
  - Much of what we will talk about is motivated by the Phase 0-3 historical developments.
  - Is the next phase fundamentally different? What kind of OS do we need then?

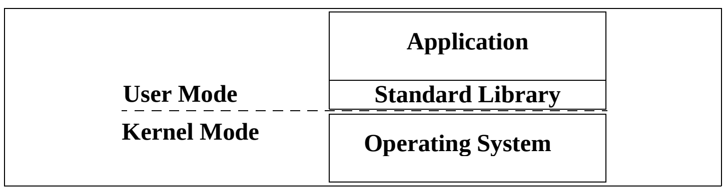
Technology Advances Determine OS

	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

Outline

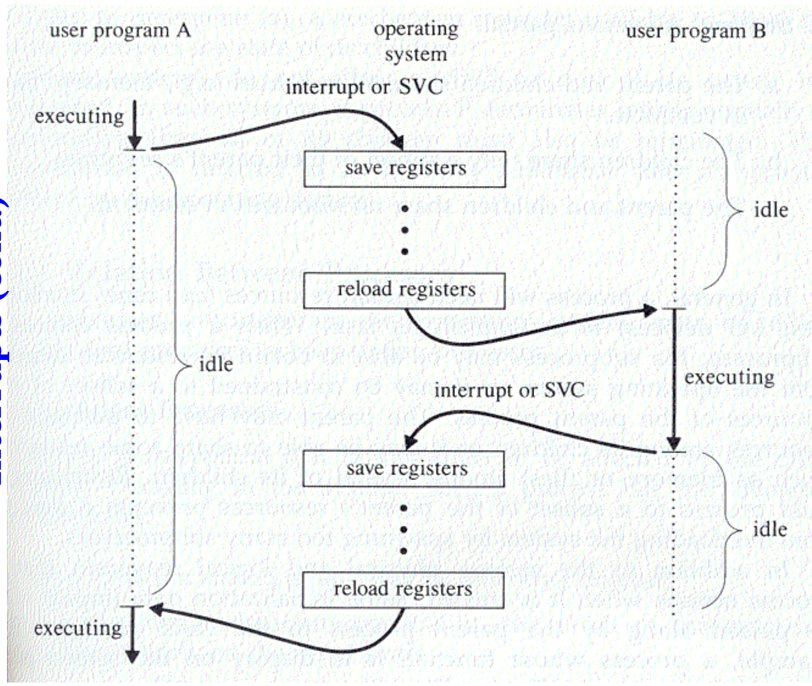
- ~~Introductions~~
- ~~History~~
- <u>**General mechanisms**</u>
- Process management
- Memory management
- File systems
- Conclusions

Dual-Mode Operation



- The machine has two modes of operation: <u>user mode</u> and <u>kernel mode</u> (also called monitor mode, supervisor mode, system mode, privileged mode)
- Divide all instructions into two categories: unprivileged and <u>privileged instructions</u>
- Users can’t execute privileged instructions
- Users must ask the OS to do it on its behalf: <u>system calls</u>
- The OS gains control upon a system call, switches to kernel mode, performs service, switches back to user mode, and gives control back to user

Interrupts (cont.)



Interrupt-Driven Operation

- Everything the OS does is <u>interrupt-driven</u>
- An interrupt stops the execution dead in its track, control is transferred to the OS
- The OS saves the current <u>execution context</u> in memory. These include the PC, the registers, and other stuff (later)
- The OS figures out what caused the interrupt
- Executes a piece of code (<u>interrupt handler</u>) to handle this particular type of interrupt
- Loads some execution context (possibly the one saved before the interrupt, or possibly some other saved one) and resumes execution

Close Interaction Between Architecture and OS

- The TOY architecture, as it is, is not sufficient to support even a minimum OS
- Dual-mode operation and interrupts are a good example of how architects and OS writers must work together to build a working "system"
- We will see more examples of this dialogue

Interrupt-Driven Operation (cont.)

- Everything the OS does is <u>interrupt-driven</u>
- System call: when user asks service from OS
- When a device needs attention
- (Periodic) timer interrupts
- Program errors or "abnormal conditions", such as illegal instructions or attempts of referencing illegal memory addresses
- More examples which we will see later...

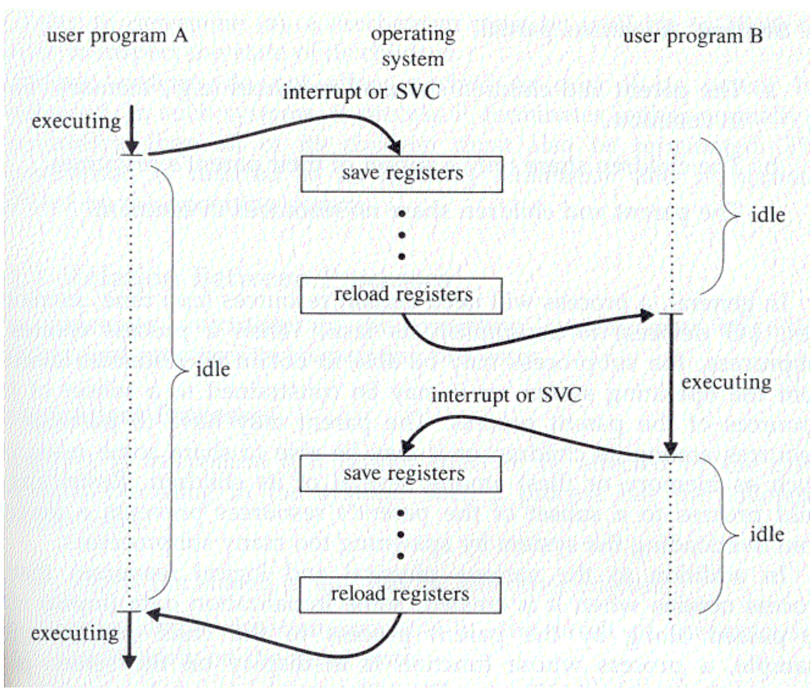
Outline

- ~~Introductions~~
- ~~History~~
- ~~General mechanisms~~
- **Process management**
  - A process is a running program
  - There are many of them
  - How do we create the illusion that each has its own CPU?
- Memory management
- File systems
- Conclusions

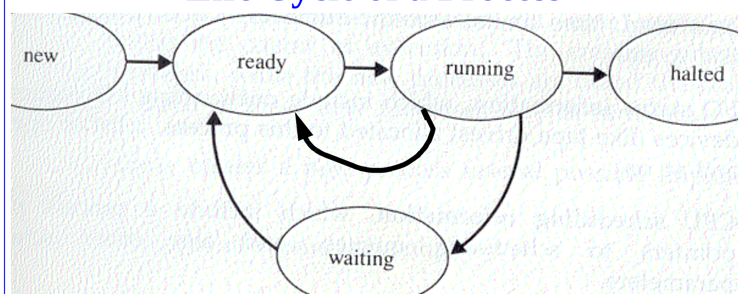
What Next?

- **What next**?
  - Process management: a virtual **CPU** for every user, and indeed, every program
  - Memory management: infinite and safe **memory** for every program
  - File system: make files and directories out of **disk** blocks
- **What features** are we shooting for for each of these?
  - Higher level (nicer) abstractions
  - Fairness
  - Protection
  - Sharing
- **What** common **strategies** do we employ?
  - Chop up resources into small pieces and allocate them at this **fine-grain** level: time quantum, memory pages, disk blocks
  - Introduce levels of **indirection**: users use logical names which are translated into physical names
  - Use past **history** to predict future behavior for optimizations

Context Switches

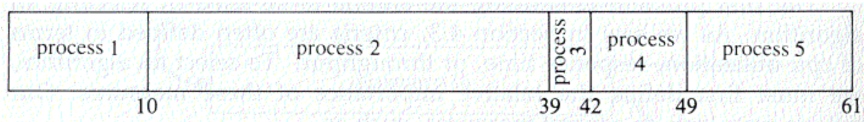


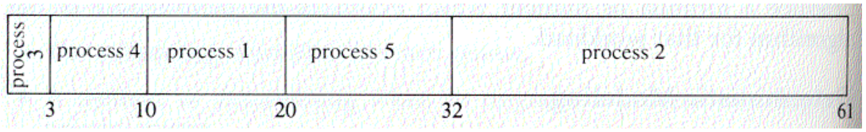
Life Cycle of a Process



- Running: instructions are being executed
- Waiting: the process is waiting for some event to occur (such as an I/O completion)
- Ready: the process is waiting to be assigned to a processor

First-Come-First-Serve vs. Shortest-Job-First





- Sum of running time of all processes are the same for two strategies
- FCFS
  - Average wait time of processes: (0+10+39+42+49)/5 = 28
  - What's wrong: short processes getting stuck behind long ones
- SJF
  - Average wait time of processes: (0+3+10+20+32)/5 = 13
  - Provably optimal!
  - Problem: we can't predict how long a job will take
- What happens when you run an infinite loop?

Process Scheduling

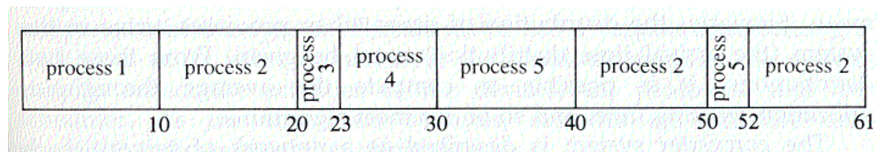
- We have a whole bunch of processes that are ready to run
- Which one do we run next?
- The answer depends on what you're trying to optimize for
- In the following discussion, suppose
  - We are interested in minimizing **average wait time** of each,
  - and we have the following processes

Process	Burst Time
1	10
2	29
3	3
4	7
5	12

Outline

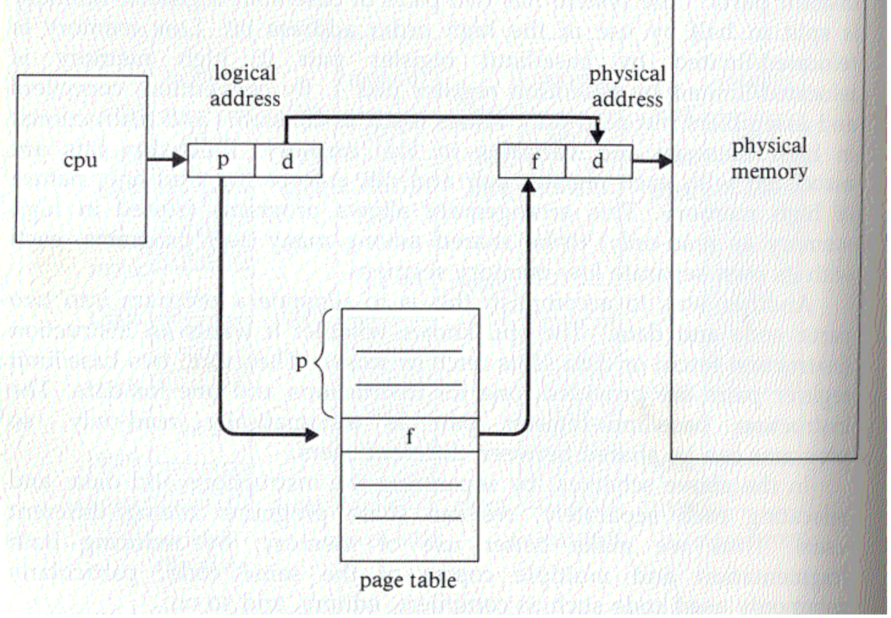
- ~~Introductions~~
- ~~History~~
- ~~General mechanisms~~
- ~~Process management~~
- **Memory management**
- File systems
- Conclusions

Round-Robin Scheduling



- Divide up time into quantums (10 in this case)
- Timer set to interrupt at the end of each quantum
- Two things can happen during a quantum
  - The process finishes before the timer goes off, OS picks someone else
  - The process doesn't finish by the end of the quantum, OS suspends this process and pick someone else
- Average wait time of processes in this case: (0+32+20+23+40)/5 = 23, this is in between FCFS and SJF
- Infinite loops are not a problem!
- Quantum length is an important consideration for performance

Virtual vs. Physical Memory Addresses



TOY Memory Problems

- Problem 1:
  - Can't run two instances of the same program simultaneously!
  - Why? Consider the instruction: `mem[0x30]<-r1`
  - Two people modify the same memory location at the same time
- Problem 2:
  - How do you make sure other people don't accidentally or maliciously change or snoop your memory?
- Problem 3:
  - Can't access more than 256 words of memory
- There are many hacks around these and many other memory management problems, but it turns out that virtual memory provides a common elegant solution to all of them

Paging

- Basic idea: allowing remapping of memory at **word** granularity is too much trouble
- So only remap at **page** granularity:
  - Divide up memory into blocks that are called pages
  - Each virtual page can be placed in any physical memory frame
  - Each translation involves two steps:
    - + Decide which physical frame holds the logical page
    - + Decide where the address is inside the page (the offset)
    - + The physical address is formed by gluing together the physical page number and the offset

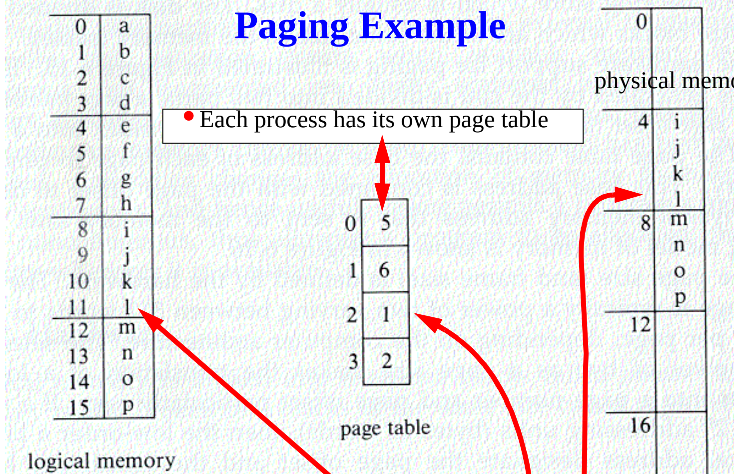
Basic Idea Behind Virtual Memory

- Basic idea
  - Programs don't (and can't) name **physical memory addresses**.
  - Instead, they use **virtual addresses**: each process has its own memory
  - Each virtual address must be translated to physical address before the memory operation can be carried out
- Why does this fix our problems? Consider `mem[0x30]<-r1`
  - We can run two instances of the same program, because 0x30 is only a logical name that can be translated to different physical locations, and each process has its own trans. table
  - One person can't hurt another because he can't see or use other people's page table (he can't touch others' 0x30)
  - We can run program that uses more physical memory than we have because we can name a huge amount of virtual memory, not all of which fit in physical memory (can name 0xF9AB)

Paging to Disk

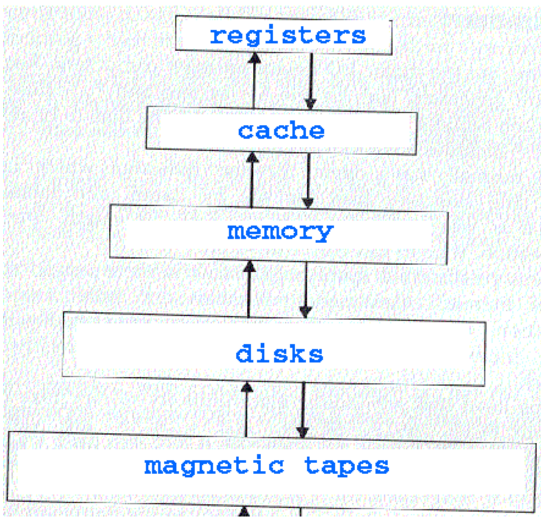
- If we can’t fit all the virtual memory in physical memory, we need to temporarily stash some pages on disk
- To optimize performance, we need to decide which ones to toss out and which ones to keep, this is called <u>page replacement</u>
- The provably optimal strategy:
  - Replace the page which will not be needed for the longest period in the future
  - Problem: requires prediction of future, which is impossible
- Many heuristics used in real life
  - One of the most popular ones is <u>LRU: least recently used</u>

Paging Example



- 4-byte pages
- Consider the virtual address $11_{10}=1011_2$
- Chop it into two parts
  - Virtual page number $2_{10}=10_2$
  - Offset within page $3_{10}=11_2$
- Look up the page table and find that virtual page 2 is stored at physical page 1
- The physical address is $7_{10}=0111_2$

Storage Hierarchies

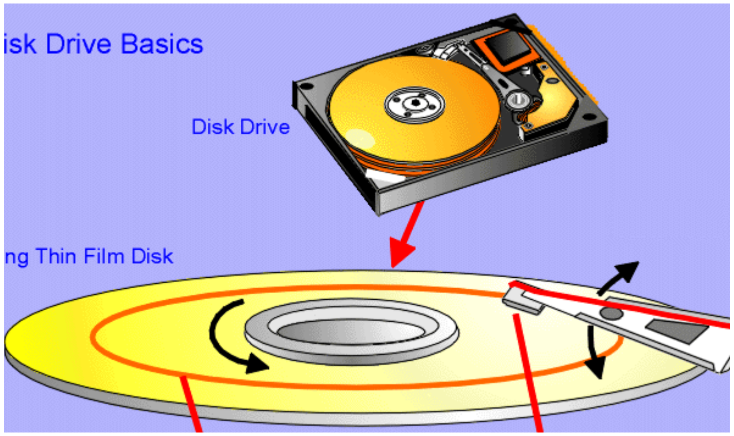


- Each lower level is
  - slower,
  - bigger,
  - farther away, and
  - cheaper
- Who manages what
  - registers: compiler
  - cache: hardware
  - memory: OS
  - disk: OS
- The performance of lower level is becoming increasingly important

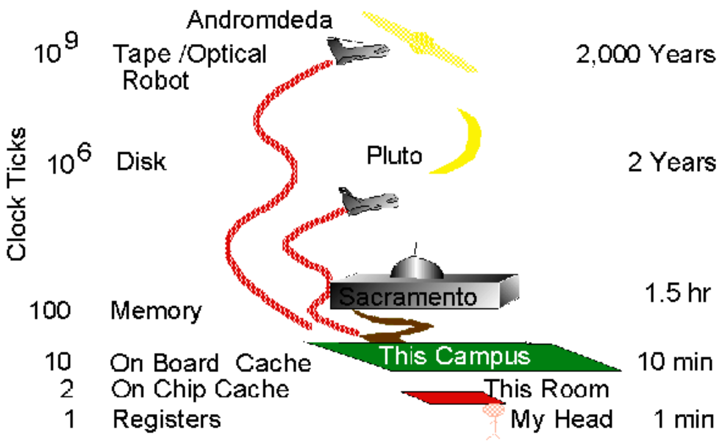
Outline

- ~~Introductions~~
- ~~History~~
- ~~General mechanisms~~
- ~~Process management~~
- ~~Memory management~~
- **<u>File systems</u>**
- Conclusions

Have You Ever Opened Up a Disk Drive?



Storage Hierarchy Latency (by Jim Gray)

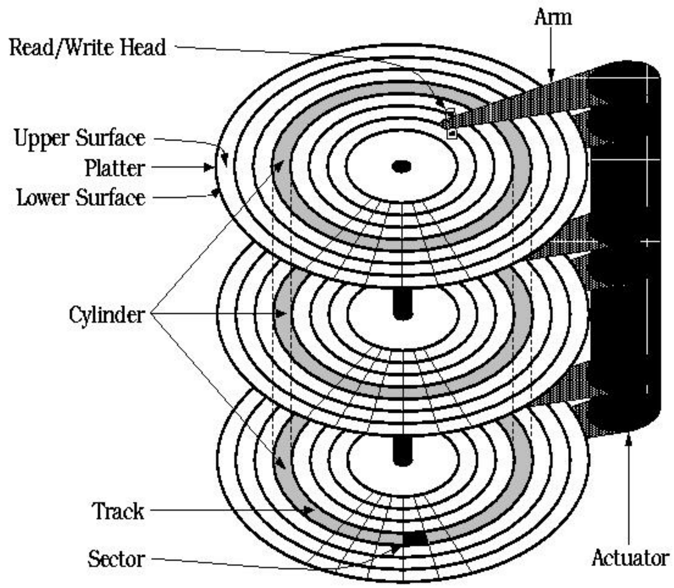


- And the "universe" is expanding -- farther things are getting farther faster!

Levels of Abstractions

- Inside the disk: things are complicated
- Abstraction exported by the disk to the operating system: an array of blocks, which are called <u>sectors</u>, 512 bytes each
- The abstraction exported by the operating system to the user: directories and files
- In reality, the abstraction isn't quite as clean: problem: disks have non-uniform access time and we need to worry about where things sit

Have You Ever Opened Up a Disk Drive? (cont.)



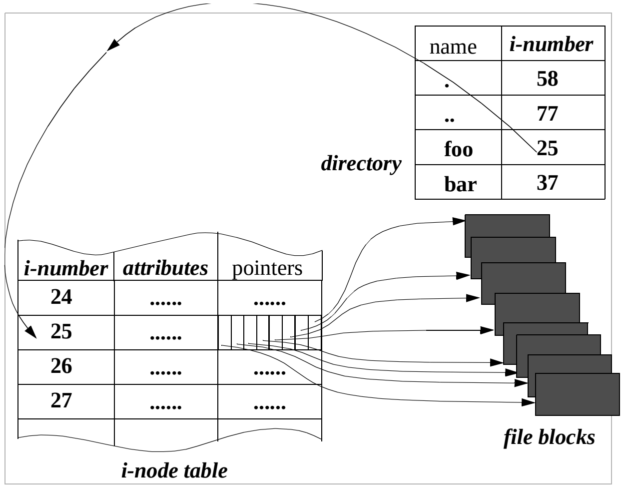
State-of-art (1999):

- Rotation speed: 10,000 RPM
- Capacity: 50 GB
- Bandwidth: ~20MB/s
- Average latency: ~10ms
- Improvement: both capacity and bandwidth are increasing at the rate of about 50% per year!

Outline

- ~~Introductions~~
- ~~History~~
- ~~General mechanisms~~
- ~~Process management~~
- ~~Memory management~~
- ~~File systems~~
- **Conclusions**

Unix File System Internals



Challenge to OS Designers: Distributed Systems

Some **example** problems for each of the areas we looked at

- CPU scheduling: it can be proven that optimal scheduling for multiple CPUs is NP-complete!
- Memory management: how to form a giant global memory to cache, for example, web pages?
- File system: how to gain access to your files anywhere any time?
- How to provide security and reliability for all these resources?

Common Strategies

- Chop up resources into small pieces and allocate them at this **fine-grain** level: time quantum, memory pages, disk sectors
- Introduce levels of **indirection**: users use logical names which are translated into physical names: virtual memory addresses, file system directory names, inode numbers, ...
- Use past **history** to predict future behavior for optimizations: CPU scheduling, memory replacement, and disk block allocation

A More Fundamental Question: Do We Need to Reexamine How We Make OSes

- Much of everything in OS we looked at is inherited from the **historical** development of **multiprogramming**
- Some predicted that the PC revolution would kill OSes, didn't happen, we ended up "going back to the future"
- Is the next wave fundamentally different?
- Or are we doomed to "going back to the future" again?

What Does Java Have To Do with All This?

From NY Times article, May 25, 1998

- "necessary to fundamentally blunt Java momentum" in order "to protect our core asset, Windows" - Paul Maritz, a Microsoft group vice president
- "Strategic Objective: kill cross-platform Java by growing the polluted Java market." - internal Microsoft planning document

- **JVM** provides far more than simple portablity
- It manages resources, provides security, and provides sharing
- So it's in effect an **OS**!
- Intriguing: fundamentally different way of providing protection: at language level
  - Java: **s/w** based protection based on **type safety of objects**
  - Virtual memory: **h/w** protection based on **pages** of memory
  - Can you tell which is better??

Meta-Advice: Stay Broad

- The developments in OS are a perfect example of why you want to stay broad, as this class is
- Why don't you just teach me programming?
  - Robot programmers never get to define the future
  - Robot programmers die along with obsolete systems
- Today there is a shortage of 25-year old engineers, and a surplus of 45-year-old ones. Why? How do you make sure that you don't become a surplus when you're 45?