Sheets for MIEIC's SOPE

based on teaching material supplied by A. Tanenbaum for book: Modern Operating Systems, ed...

Revisiting Topics

All chapters

1. Introduction: Operating system structure
2. Processes: Process scheduling
3. Coordination: Deadlocks (cont.)
4. Memory Management: Paging (cont.)
5. Input/Output: Data storage (cont.)
6. File Systems: Accessing a File System

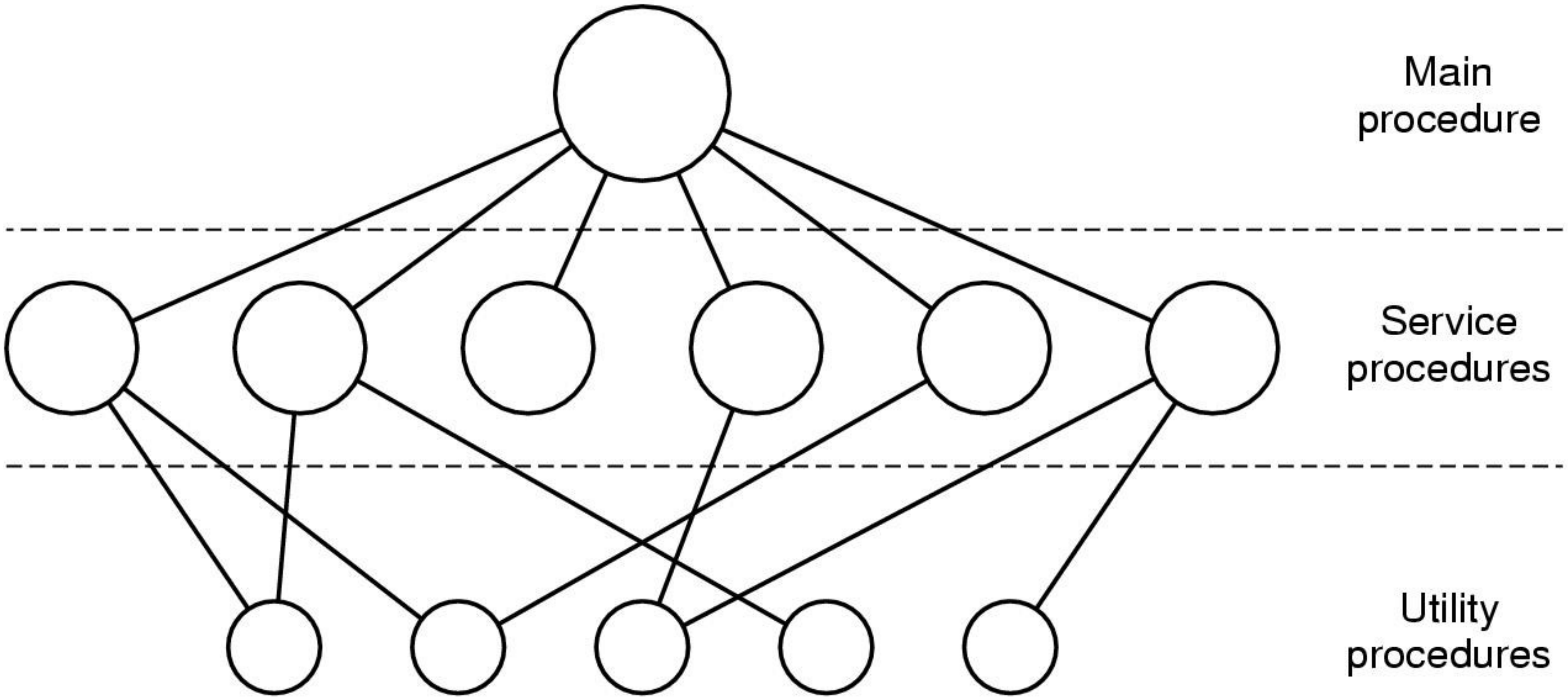
Chapter 1- Introduction

Operating system structure

The Operating System Zoo

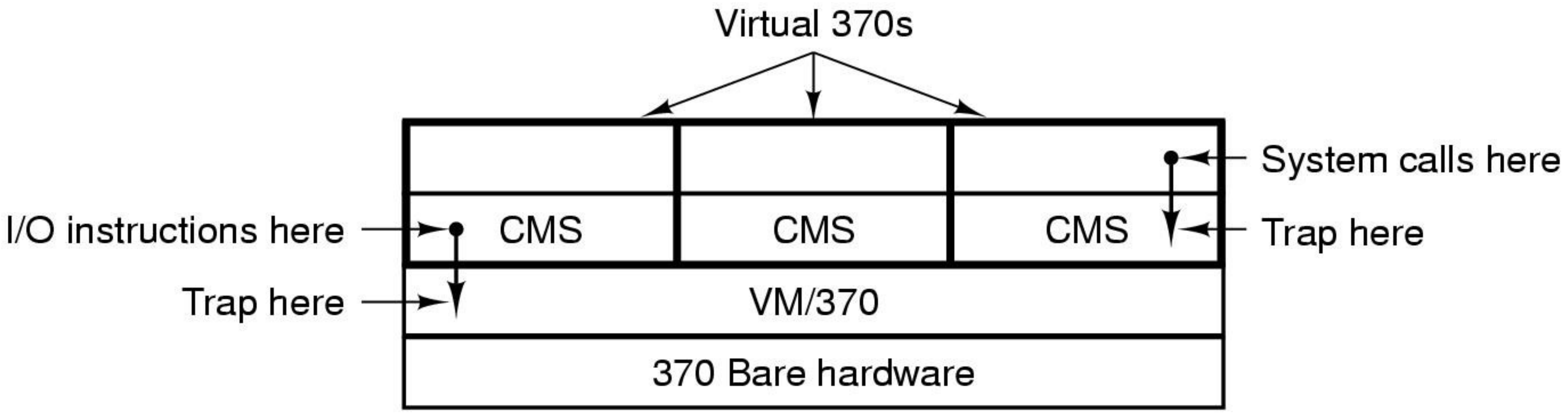
- Mainframe operating systems
- Server operating systems
- Multiprocessor operating systems
- Personal computer operating systems
- Real-time operating systems
- Embedded operating systems
- Smart card operating systems
- …

Operating System Structure (1)



Simple structuring model for a monolithic system

Operating System Structure (2)

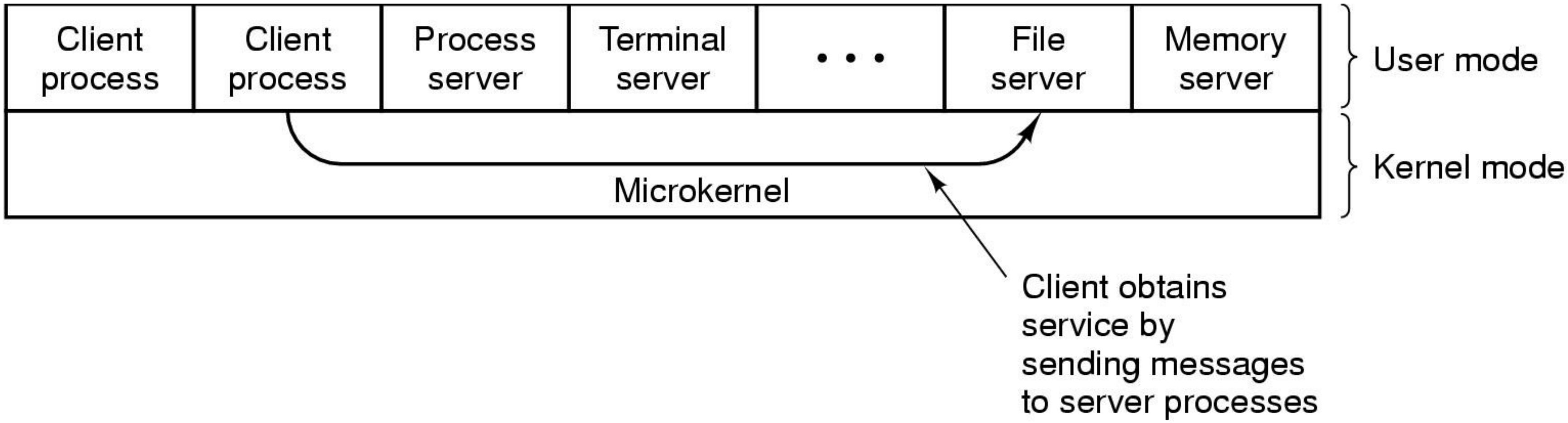


Structure of IBM's VM/370 with CMS*: virtual machines emulating the bare hardware

* Conversational Monitor System

«... the OS has thus far served as the master illusionist, tricking unsuspecting applications into thinking they have their own private CPU and a large virtual memory, while secretly switching between applications and sharing memory as well. Now, we have to do it again, but this time underneath the OS, who is used to being in charge.»
Appendix B. Virtual Machine Monitors, OSTEP, Arpaci-Dusseau

Operating System Structure (3)



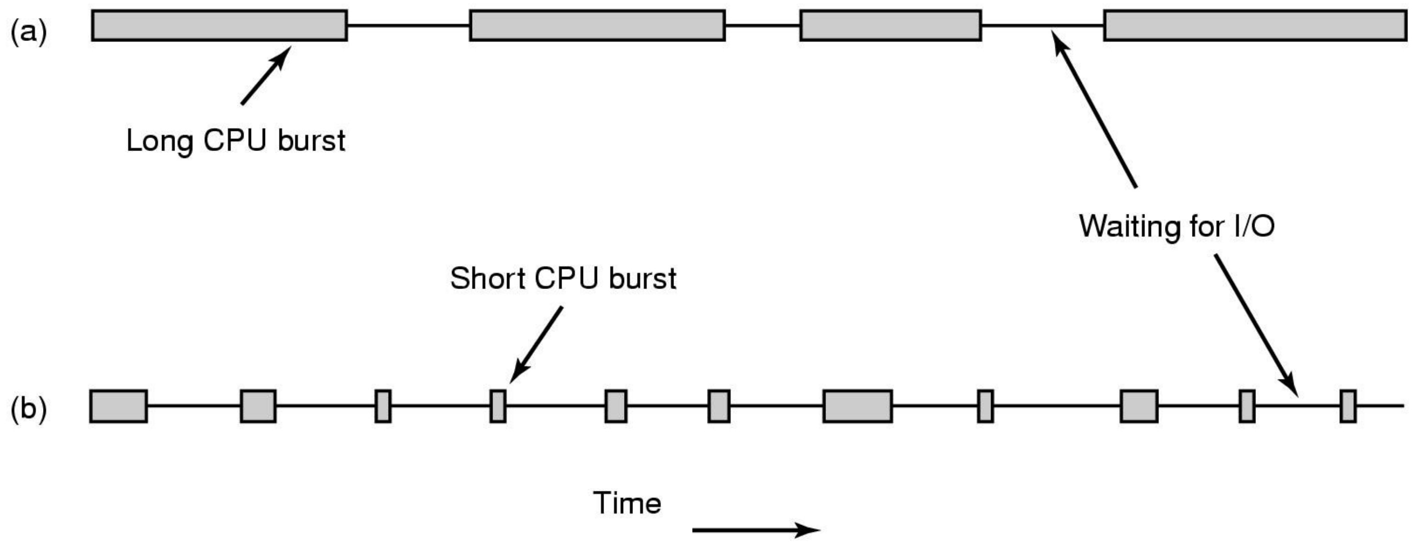
The client-server model and the micro kernel.

Chapter 2- Processes

Scheduling

Scheduling: Motivation (1)

In many cases CPU is underloaded

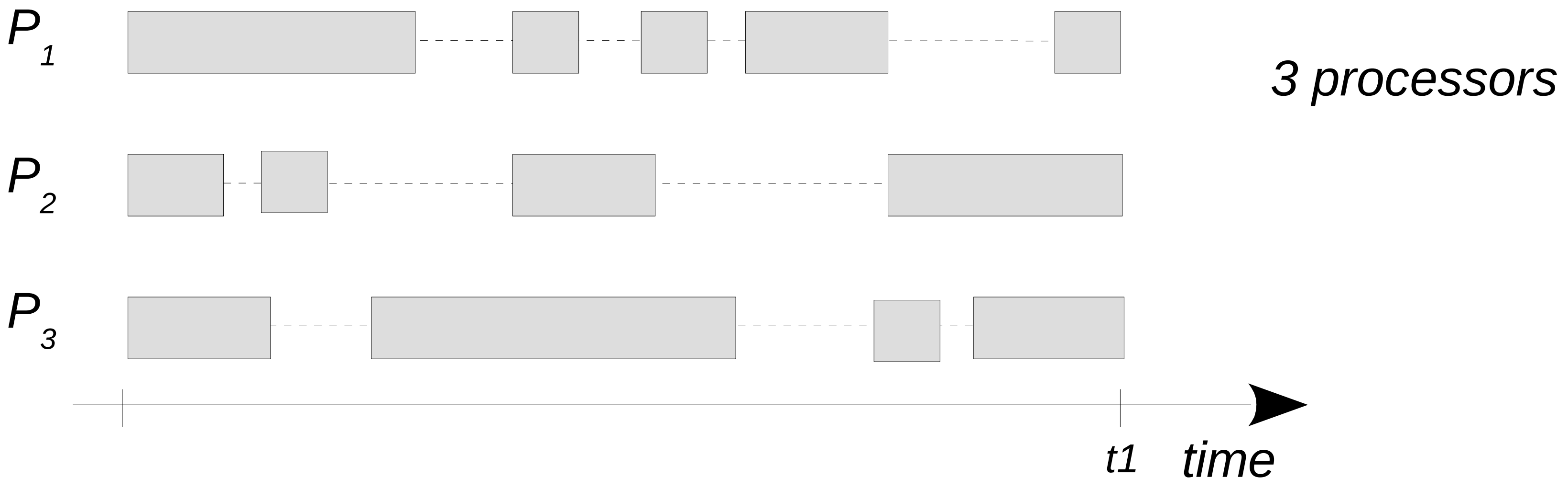


Bursts of CPU usage alternate with periods of I/O wait

- a CPU-bound process
- an I/O bound process

Scheduling: Motivation (2)

Advantage of multiprogramming even with a single processor...



For the example pictured, show that with a specific scheduling in a single processor, the total processing time is less than double the time with 3 processors.

Scheduling: Algorithm's goals (3)

All systems

Fairness - giving each process a fair share of the CPU
Policy enforcement - seeing that stated policy is carried out
Balance - keeping all parts of the system busy

Batch systems

Throughput - maximize jobs per hour
Turnaround time - minimize time between submission and termination
CPU utilization - keep the CPU busy all the time

Interactive systems

Response time - respond to requests quickly
Proportionality - meet users' expectations

Real-time systems

Meeting deadlines - avoid losing data
Predictability - avoid quality degradation in multimedia systems

Scheduling: Operation issues (4)

Preemption?

- yes: kernel rules!
- alternative is possible, but...
  - user programs should voluntarily yield the processor!

Interruptions

- kernel can run scheduler
- Clock: 100 Hz (10 ms period)

Quantum of CPU time

- 100 ms is typical

Process (or context) switch

- cost!

Getting info from kernel:

ps

GNU's time

getconf

syscall sysconf()

syscall getrusage()

syscall getrlimit()

...

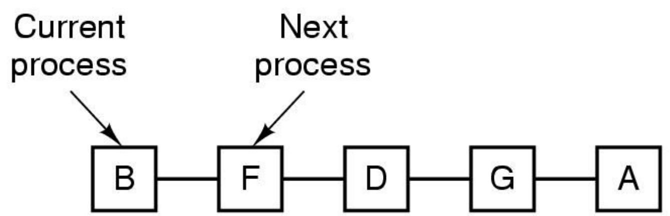
Scheduling: Algorithms/Policies (5-1)

First-come First-Served

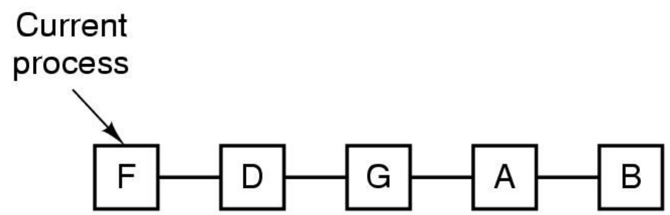
- priority is time of arrival!
- good with:
  - non-preemptive
  - batch systems

Round Robin

- equal priority
- good with:
  - preemptive
  - interactive/batch systems



a) list of runnable processes

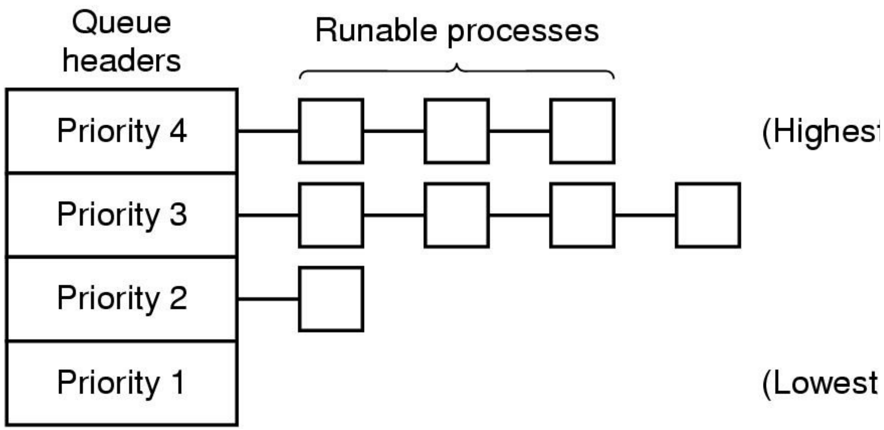


b) list after B uses up its quantum

Scheduling: Algorithms (5-2)

Priority classes (or multiple queues)

- first priority queue runs first
- good with
  - preemptive
  - interactive/batch systems
- risk of starvation!
- variant:
  - different *quantum* for different queues



Linux 2.6...Scheduling (5-3)

Completely Fair Scheduler (CFS)...

TaskClass	Preemption	Algorithm	Niceness	Quantum (CPU time slice)
system FIFO	N	-	-	∞
system	Y	round-robin	adjustable	adjustable
user	Y	"completely fair"	[-20.. 0 ..19]	dynamic: [`min_granularity`.. *virtual runtime* ..`sched_latency`]

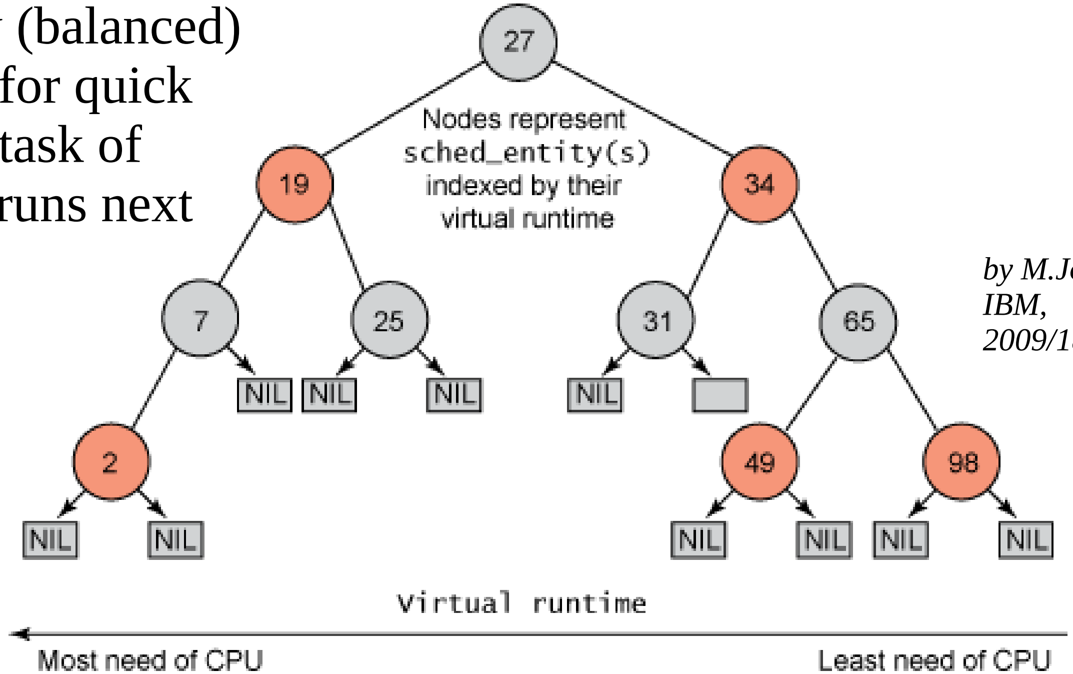
Note: the quantum top level varies with the number of runnable tasks – the more there are, the smaller the level is

```
$ cat /proc/sys/kernel/sched_min_granularity_ns
2250000
$ cat /proc/sys/kernel/sched_latency_ns
18000000
$ cat /proc/sys/kernel/sched_rr_timeslice_ms
100
```

Linux 2.6...Scheduling (5-4)

Completely Fair Scheduler (CFS)...

- user task (process/thread) scheduling:
  - member of class
  - can be grouped (e.g. threads of same process)
- virtual runtime
  - function (real CPU runtime, niceness)
  - represented by (balanced) red-black tree for quick manipulation, task of leftmost node runs next



by M.Jones, IBM, 2009/18

Chapter 3- Coordination

Deadlocks (cont.)

Deadlock: dealing with it

(*from Coordination chapter*)

Strategies:

- just ignore the problem altogether
  – ok, if it is rare or does not "hurt" much when happens
- detection and recovery
  – monitor system and terminate (some of) deadlocked processes
- prevention
  – negate one of the four necessary conditions
- avoidance
  – do careful resource allocation

Deadlock (1): *Ostrich* strategy

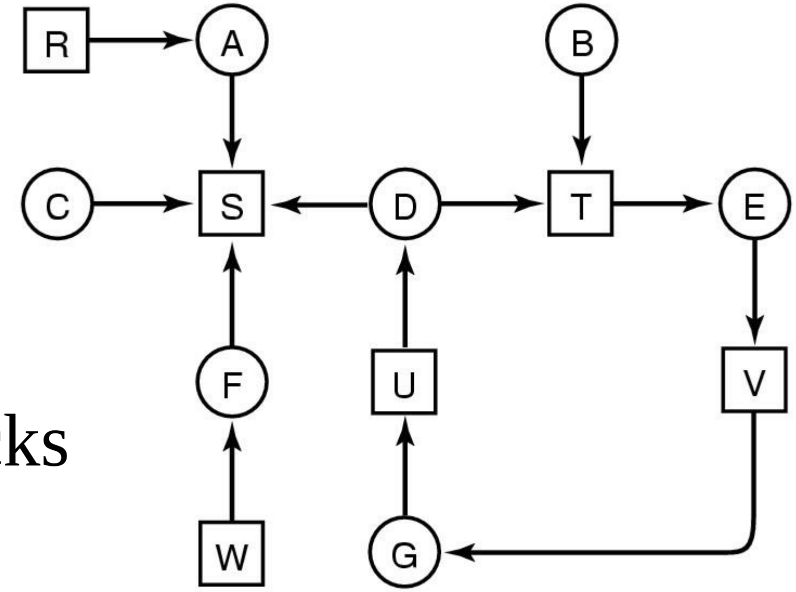
Strategy: pretend there is no problem

- Reasonable if
  - deadlocks occur very rarely
  - cost of prevention is high
- It is a trade off between
  - convenience
  - correctness
- UNIX and MsWindows take this approach

Deadlock (2): *detection* strategy

Strategy: detection and recovery!

- use a Monitor System



- then, try to remedy deadlocks
  - preempt
  - rollback
  - terminate

Deadlock (3): *prevention* strategy

Strategy: prevention!

- negate one of the four necessary conditions

Condition	Approach
Mutual exclusion	"Spool" everything
Hold and wait	Request all resources initially
No preemption	Take resources away
Circular wait	Order resources numerically

thus, eliminating concurrent access to resource

Summary of approaches to deadlock prevention

Deadlock (4): *avoidance* strategy

Strategy: avoidance!

- allocate resources carefully
- example: Banker's algorithm
  - 22 resources are requested; 10 are available

there is no way a process can finish normally (and so liberate resources needed for the others)!

a) safe state

	Has	Max
A	0	6
B	0	5
C	0	4
D	0	7

Free: 10

b) safe state

	Has	Max
A	1	6
B	1	5
C	2	4
D	4	7

Free: 2

c) unsafe state

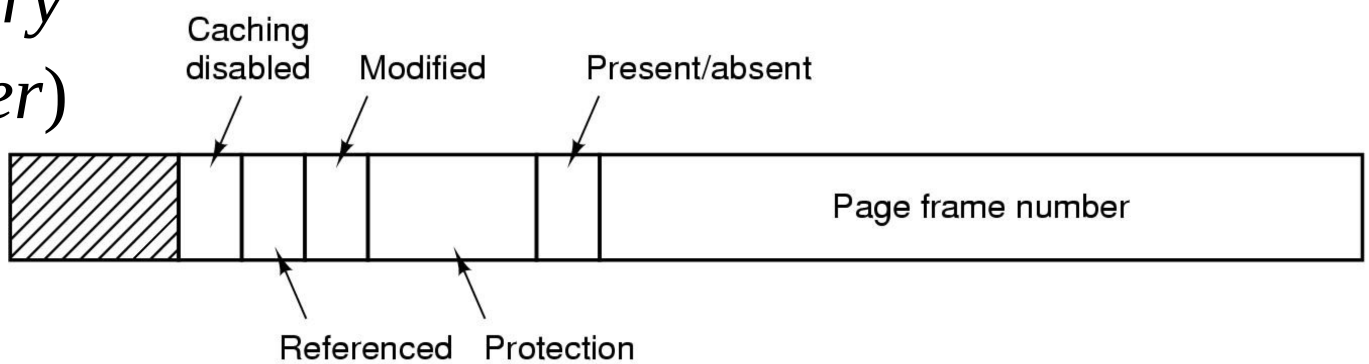
	Has	Max
A	1	6
B	2	5
C	2	4
D	4	7

Free: 1

Paging (cont.)

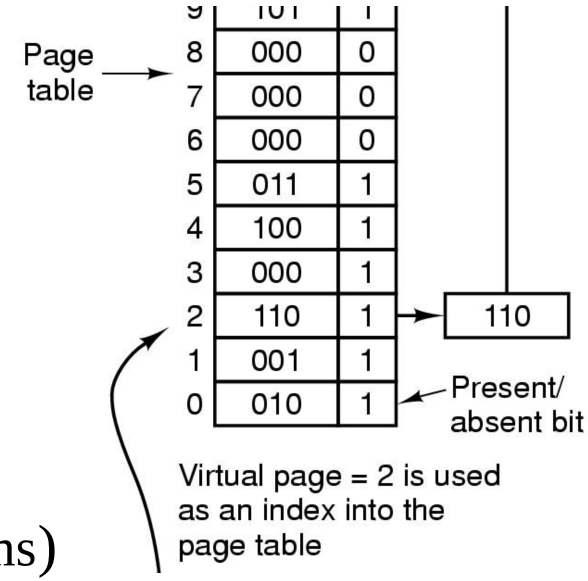
(from Memory chapter)

Paging: page table entry



Typical page table entry:

- Page frame number: *n* bits
  (e.g. *n* = 18 w/ 1GiB of physical memory)
- Present/absent : 1b (page hit or page fault?)
- Protection: 3b (RWX)
- Modified: 1b (dirty bit)
- Referenced: 1b (for page replacement algorithms)
- Caching disabled: 1b (for I/O memory mapped devices)



Disk Page addresses are not here:
(other Oper. Syst. tables...)

Why 18 bits, when 30 bits are necessary for addressing all memory?...

Paging: Address Translation Cache

Translation Lookaside Buffer (TLB)

All memory references have to look up Page Table...

- TLB speeds this by caching some table page info
  - inside hardware (e.g. MMU)
  - associative memory, searched in parallel
  - 64 entries is typical

Because, usually, code has "localities": spatial (e.g. array), temporal (e.g. cycles). Program's "working set"!...

Not valid when: system boots, context changes...

Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

Paging: management strategies

Page management strategies

- Page loading:
  - on demand --> when they are needed
  - pre-fetching ("read ahead") --> "working set"!...
- Page eviction:
  - swap (or page) daemon monitors n. of free page frames (physical memory pages)
    - High and Low watermarks
    - some pages are kept free, ready to be used (already written to disk, if W pages)!
  - pages modified (written) are kept as long as possible
    - "working set"!...

Chapter 5- Input/Output

Data storage (cont.)

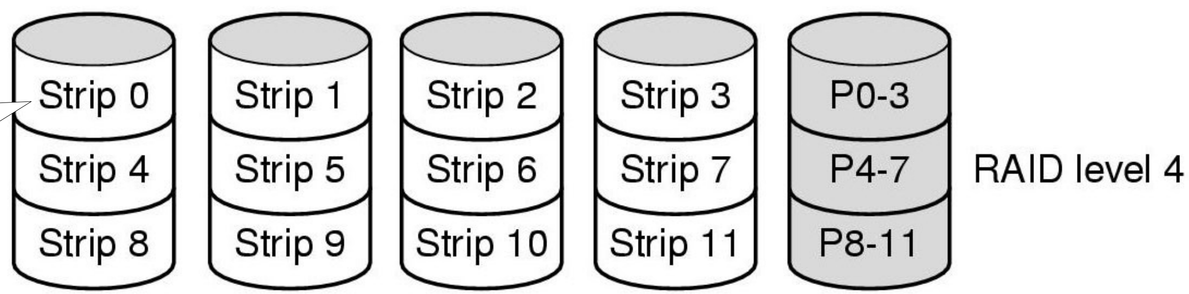
Data storage: RAID

Inexpensive, originally!...

Redundant Data Array of Independent Disks (RAID)

- Because (correct) data is important:
  - redundancy
- Because fast access to data is important:
  - parallelism
- RAID has several possible configurations ("levels")
  - each has different pros and cons: (capacity/cost, performance, reliability)

strip: group of consecutive sectors



Data storage: remote

Network-attached storage (NAS)

- file-level data storage
  - as opposed to block-level storage e.g. disk, RAID
  - different file systems: Unix's, MsWindows'...
- local area network service for heterogeneous computers
- internally, can use RAID

Cloud storage

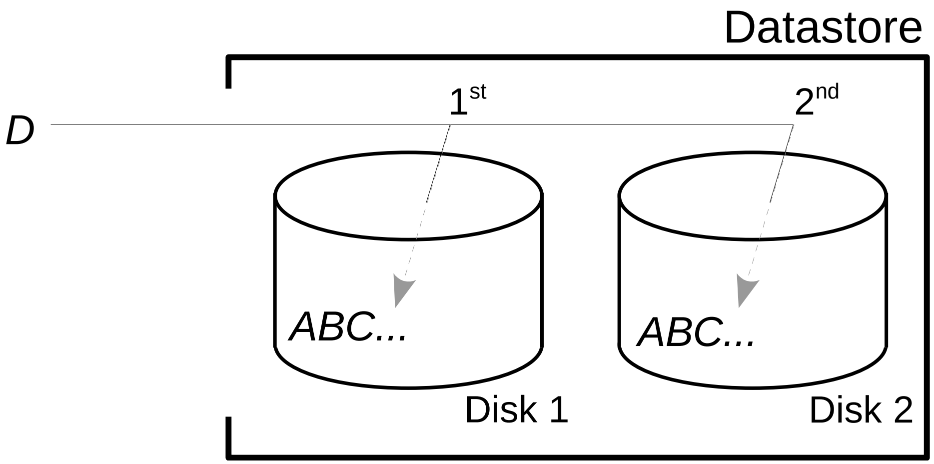
- really remote storage (network access!)
- pros:
  - high availability (through redundancy & data distribution)
  - ease of administration
- cons:
  - external (foreign) dependency
  - security concerns

Data storage: Stable Storage (1)

When data should never be lost or corrupted...

- *no* real solution; real *next best* solution: **stable storage**
  - data is either correctly written to disk or is not written, and existing data remains intact
  - correction of data is detected by disk's internal *Error Detection Codes (ECC)*

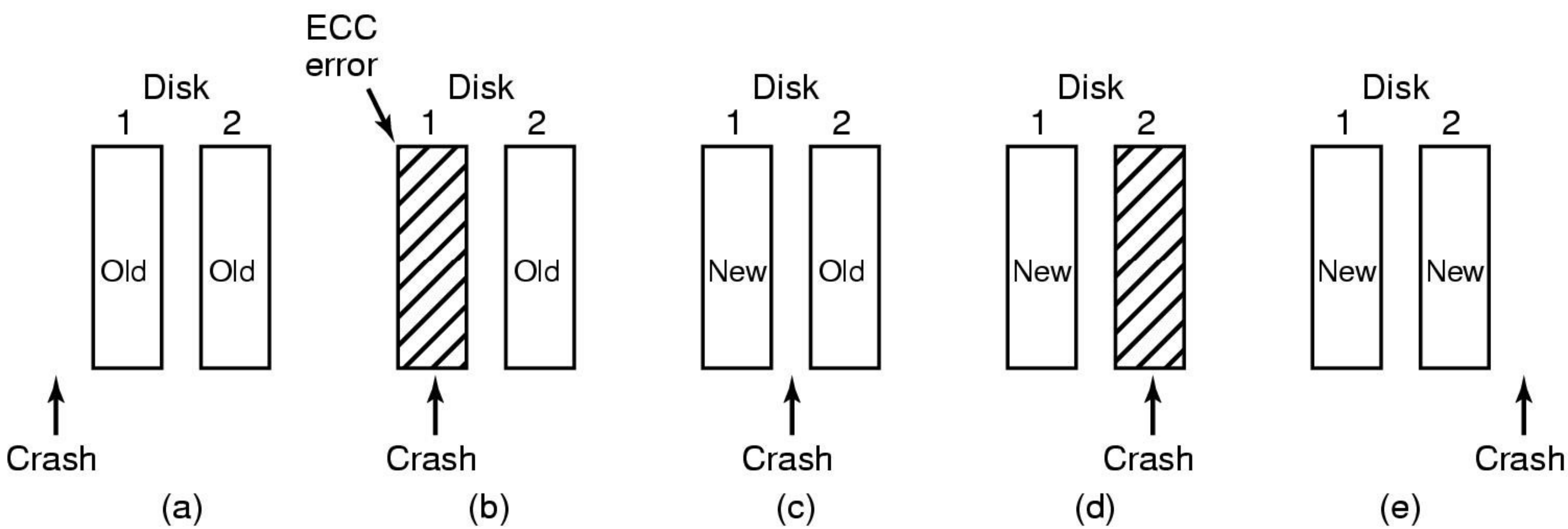
ECC, because sometimes *Correction* is also possible



Data storage: Stable Storage (2)

What if computer crashes while writing?

- show that the promise of *Stable Storage* is kept in every of the situations pictured
  - e.g.: for (a), crash is before write, so new data (in memory!) is lost but preexisting (old) data is kept correct



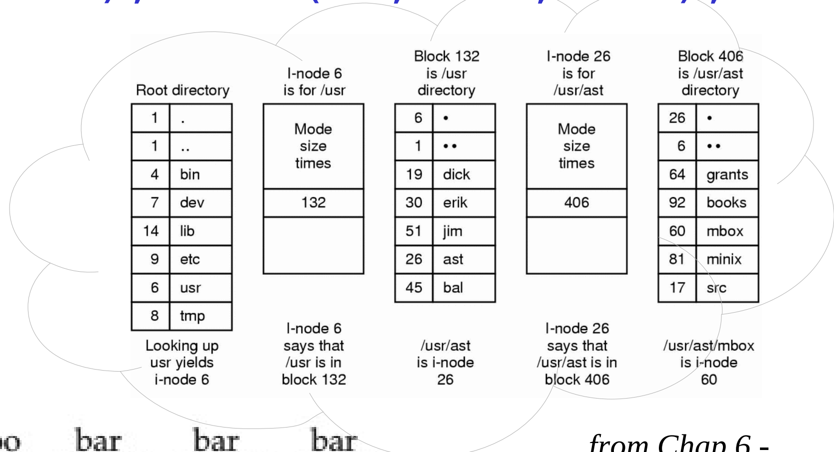
Chapter 6- File Systems

Accessing a File System

Accessing a file system: read

```
fd = open("/foo/bar", O_RDONLY); // 12 kiB (3 blocks' file)
read(fd,buff,4096); read(fd,buff,4096); read(fd,buff,4096);
```

- find `bar`'s inode!
- start with root inode ("well known")
- why write in `bar` inode? (attrs...)



from Chap.6 - File Systems

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
open(bar)			read							
						read				
				read						
							read			
					read					
read()					read					
								read		
					write					
read()					read					
									read	
					write					
read()					read					
										read
					write					

Figure 40.3: **File Read Timeline (Time Increasing Downward)**

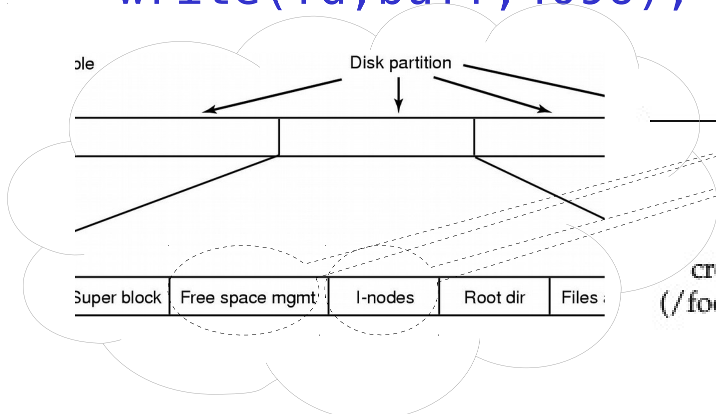
in Arpaci-Dusseau's OSTEP

fd is allocated

file offset in open file table is updated

Accessing a file system: write

```
fd = open("/foo/bar", O_WRONLY|O_CREATE, S_IRUSR|S_IWUSR);
write(fd,buff,4096); write(fd,buff,4096); write(fd,buff,4096);
```



from Chap.6 - File Systems

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
create (/foo/bar)			read							
						read				
				read						
							read			
		read write								
							write			
					read write					
				write						
write()					read					
	read write									
								write		
					write					
write()					read					
	read write									
									write	
					write					
write()					read					
	read write									
										write
					write					

in Arpaci-Dusseau's OSTEP

Figure 40.4: **File Creation Timeline (Time Increasing Downward)**

Accessing a file system: lessening the effort

Accessing the file system means a huge management effort. How to lessen it?

- caching
  - mainly for reading
- buffering
  - mainly for writing

Both techniques are very useful

- as long as they can by superseded sometimes
  - e.g. fsync()

Operating Systems revisitation...

could be continued...

...but will not!

:-)