



CSCI-GA.2250-001

# Operating Systems

## Networking

Hubertus Franke  
[frankeh@cs.nyu.edu](mailto:frankeh@cs.nyu.edu)



# Why we need networking ?

- Everything is now connected !!!
- Everything is an online service now
  - (whatsapp, facebook, Netflix, ...)
- IoT (internet of things) exploding
- Folks are constantly online



# What is a “internet”?

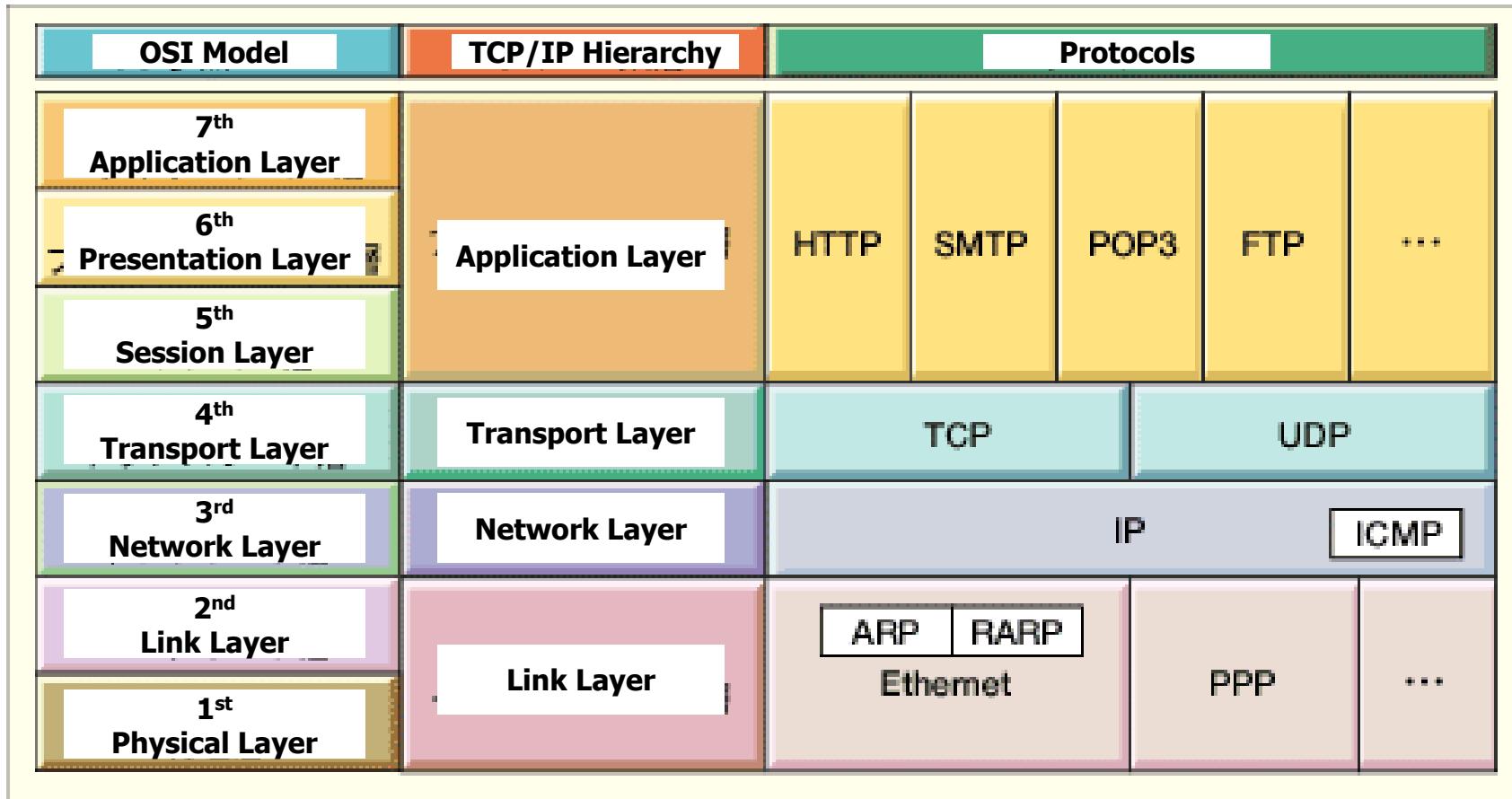
- A set of *interconnected networks*
- **The Internet** is the most famous example
- Physical networks can be completely different technologies:
  - Ethernet, ATM, modem, ...
- *Routers* (nodes) are devices on multiple networks that pass traffic between them
- Individual networks pass traffic from one router or endpoint to another
- Each endpoint has a unique MAC address (**Media Access Control / 48-bits**)
- But, its more about the services that are consumed over the network than the technology building the network
- TCP/IP is what links them together.

# TCP/IP protocol family

- TCP/IP hides the details as much as possible
- IP : Internet Protocol
  - UDP : User Datagram Protocol
    - RTP, traceroute
  - TCP : Transmission Control Protocol
    - HTTP, FTP, ssh



# OSI and Protocol Stack



Link Layer : includes device driver and network interface card

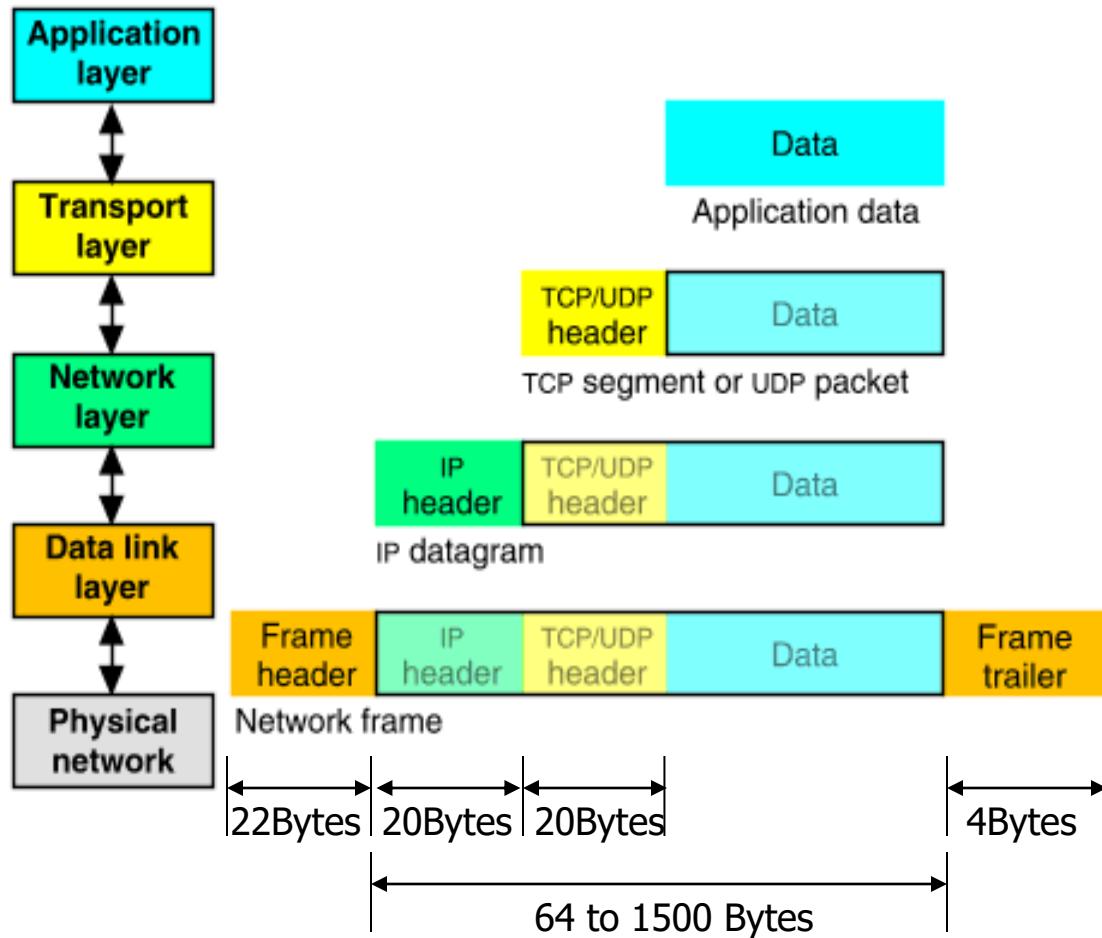
Network Layer : handles the movement of packets, i.e. Routing

Transport Layer : provides a reliable flow of data between two hosts

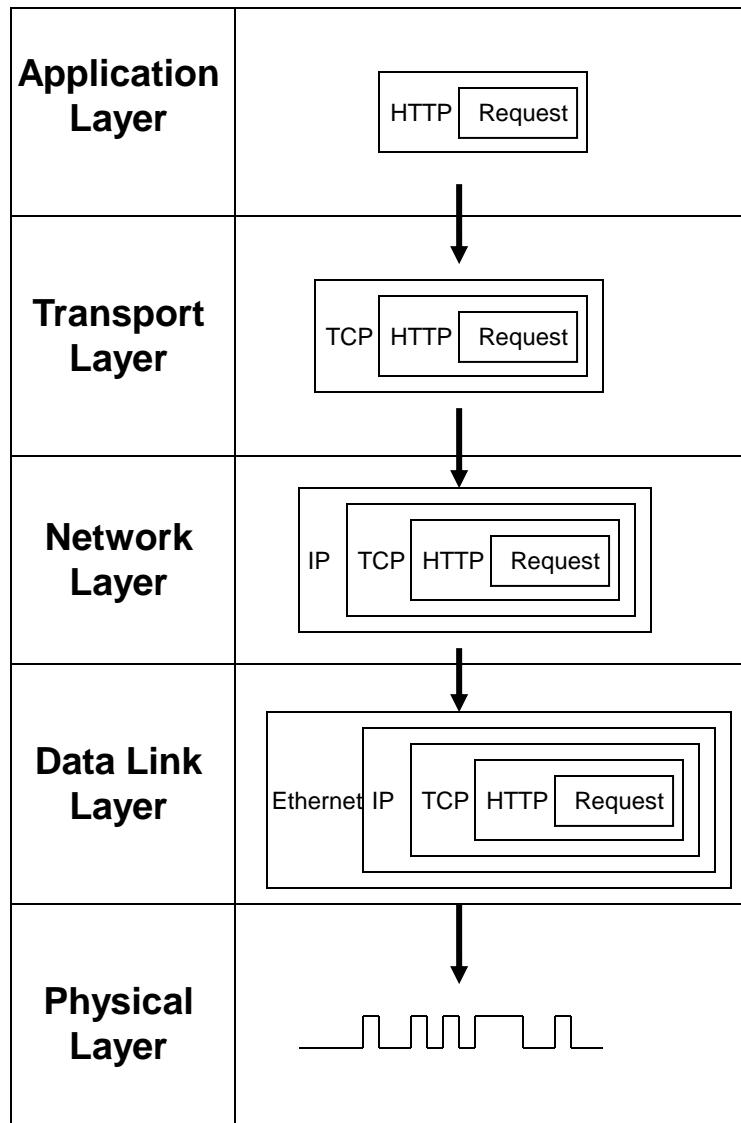
Application Layer : handles the details of the particular application

# Packet Encapsulation

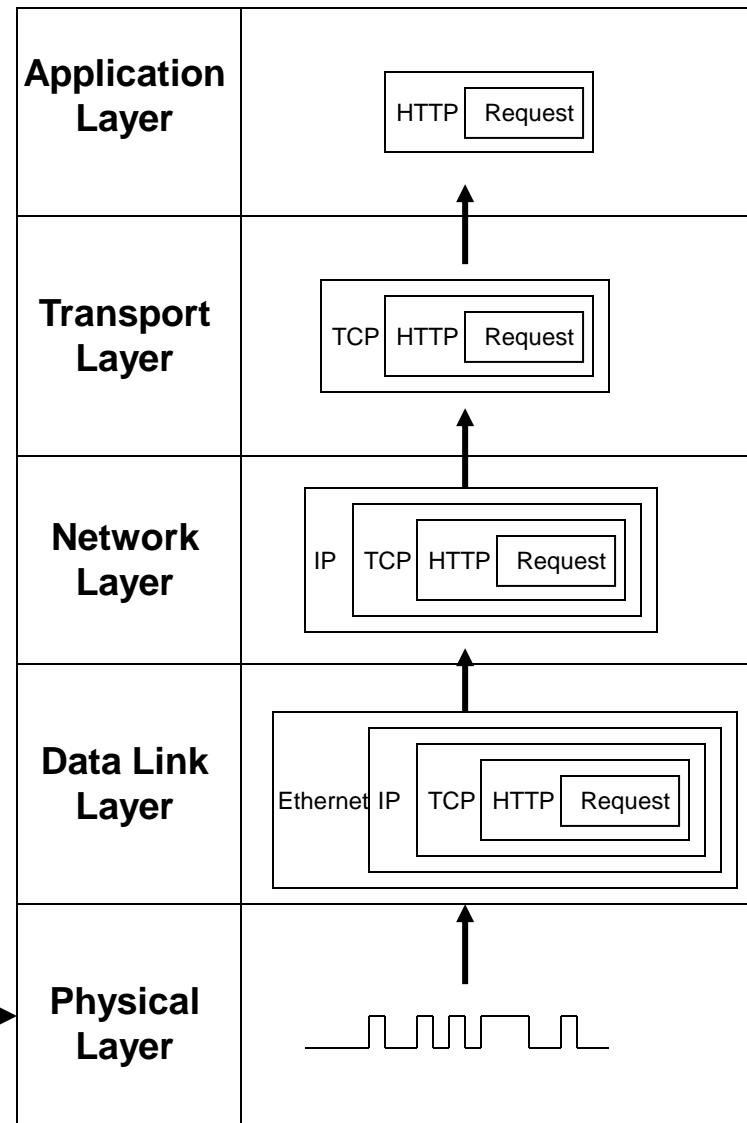
- The data is sent down the protocol stack
- Each layer adds to the data by prepending headers



# Sender



# Receiver



# IP

- Responsible for end to end transmission between nodes/machines
- Sends data in individual packets
- Maximum size of packet is determined by the networks and devices
  - Fragmented if too large
  - MTU: maximum transmission unit
- Unreliable
  - Packets might be lost, corrupted, duplicated, delivered out of order

# IP addresses

- 4 bytes
  - e.g. 163.1.125.98
  - Each device normally gets one (or more)
  - In theory there are about **4 billion** available
- You can have **private** networks so there can be replication.
  - 10.0.0.0 ... 10.255.255.255      ( $256^3 = 16M$  devices)
  - 172.16.0.0 ... 172.31.255.255      ( $16 * 256^2 = 1M$  devices)
  - 192.168.0.0 ... 192.168.255.255    ( $256^2 = 65K$  devices )
- To get out of a private network you need access to an IP outside those ranges or bridge via NAT (Network address translation) or a gateway

# Allocation of IP addresses

- Controlled centrally by ICANN
  - Fairly strict rules on further delegation to avoid wastage
    - Have to demonstrate actual need for them
- Organizations that got in early have bigger allocations than they really need

# IPv6

- Created due to the IPv4 limitations ( $2^{32}$ )  
→ now 128 bit addresses
  - Make it feasible to be very wasteful with address allocations and assign each device its unique IPv6 id.
- Many other new features
  - Built-in autoconfiguration, security options, ...
- Slowly entering into production use yet:

Google's statistics show  
IPv6 usage at ~22%  
and US at 32%  
times

- Time comparison for a simple “curl command” (2016)

## NEW YORK

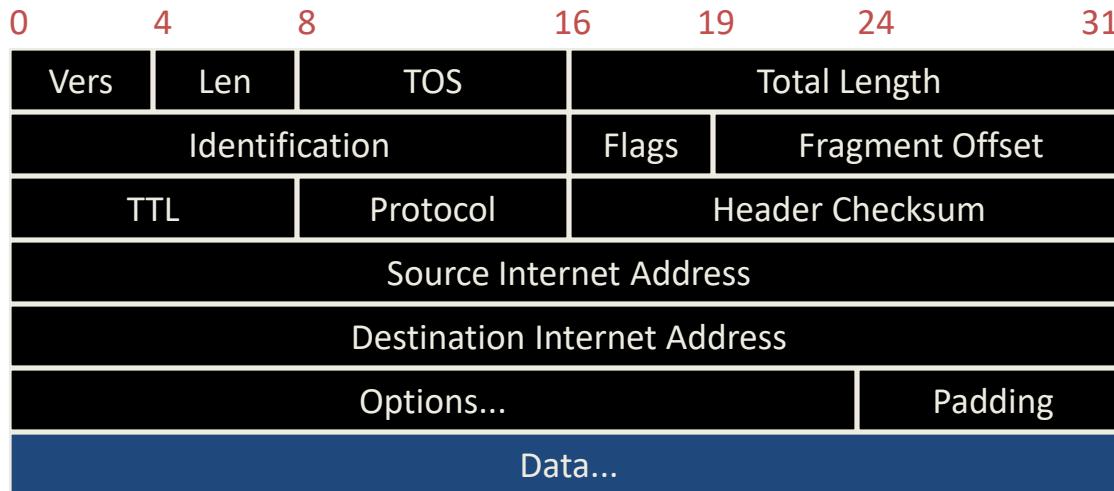
IPv4 x IPv6 Connection/Total Time Comparison

DOMAIN	CONNECT TIME		TOTAL TIME	
	IPv4	IPv6	IPv4	IPv6
GOOGLE	.031 sec	.030 sec	.061 sec	.055 sec
FACEBOOK	.062 sec	.055 sec	.100 sec	.085 sec
YOUTUBE	.030 sec	.003 sec	.063 sec	.056 sec
WIKIPEDIA	.036 sec	.035 sec	.043 sec	.042 sec
NETFLIX	.036 sec	.068 sec	.050 sec	.085 sec
LINKEDIN	.035 sec	.037 sec	.042 sec	.044 sec
PANDORA	.102 sec	.092 sec	.176 sec	.158 sec
CLOUDFLARE	.029 sec	.030 sec	.035 sec	.033 sec
SUCURI	.035 sec	.035 sec	.041 sec	.042 sec

# IP packets

- Source and destination addresses
- Protocol number
  - 1 = ICMP, 6 = TCP, 17 = UDP
- Various options
  - e.g. to control fragmentation
- Time to live (TTL)
  - Prevent routing loops

# IP Datagram

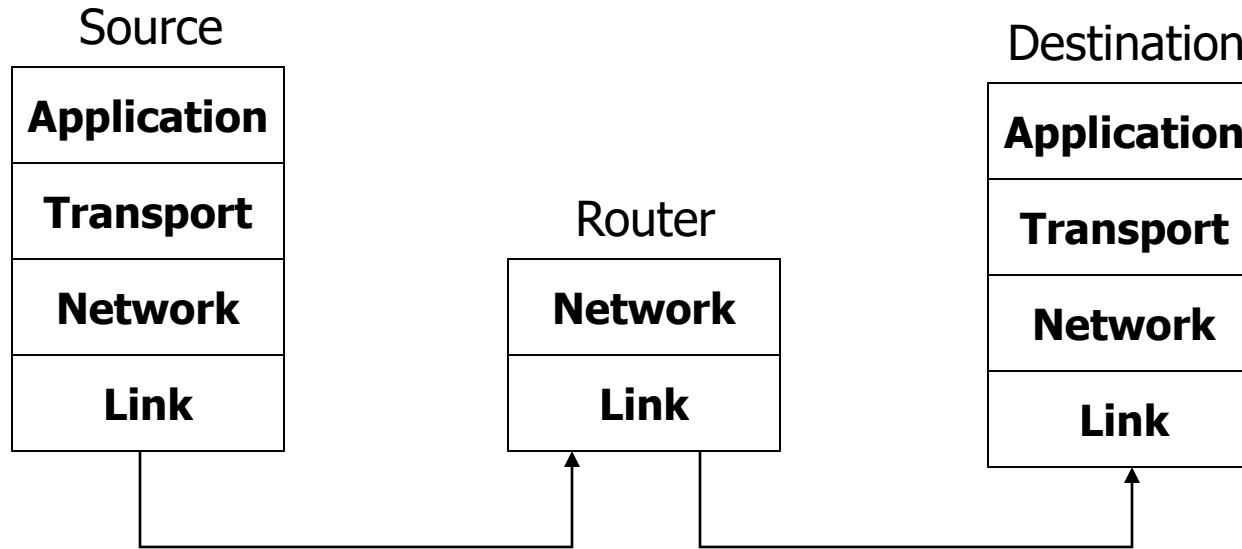


Field	Purpose
Vers	IP version number
Len	Length of IP header (4 octet units)
TOS	Type of Service
T. Length	Length of entire datagram (octets)
Ident.	IP datagram ID (for frag/reassembly)
Flags	Don't/More fragments
Frag Off	Fragment Offset

Field	Purpose
TTL	Time To Live - Max # of hops
Protocol	Higher level protocol (1=ICMP, 6=TCP, 17=UDP)
Checksum	Checksum for the IP header
Source IA	Originator's Internet Address
Dest. IA	Final Destination Internet Address
Options	Source route, time stamp, etc.
Data...	Higher level protocol data

We only looked at the IP addresses, TTL and protocol #

# IP Routing



- **Routing Table**

- Destination IP address

- IP address of a next-hop router

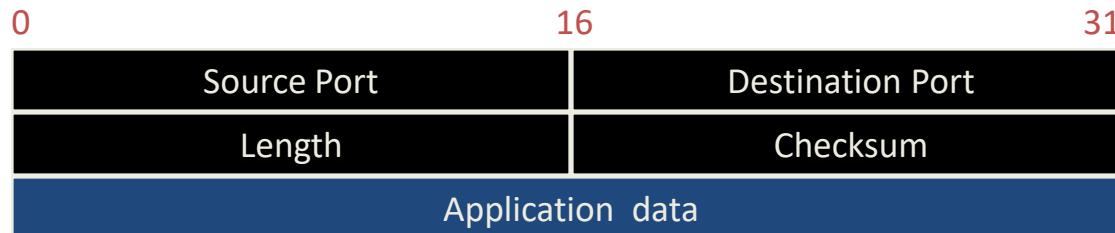
- Flags

- Network interface specification

# UDP (User Datagram Protocol)

- Thin layer on top of IP
- Adds packet length + checksum
  - Guard against corrupted packets
- Also source and destination *ports*
  - Ports are used to associate a packet with a specific application at each end
- Still unreliable:
  - Duplication, loss, out-of-orderness possible
- Allows multiplexing over IP

# UDP datagram



<u>Field</u>	<u>Purpose</u>
Source Port	16-bit port number identifying originating application
Destination Port	16-bit port number identifying destination application
Length	Length of UDP datagram (UDP header + data)
Checksum	Checksum of IP pseudo header, UDP header, and data

# Typical applications of UDP

- Where packet loss etc is better handled by the application than the network stack
- Where the overhead of setting up a connection isn't wanted
- VOIP
- NFS – Network File System
- Most games

# TCP

- Reliable, *full-duplex, connection-oriented, stream* delivery
  - Interface presented to the application doesn't require data in individual packets
  - Data is guaranteed to arrive, and in the correct order without duplications
    - Or the connection will be dropped
  - Imposes significant overheads
- Allows multiplexing of IP

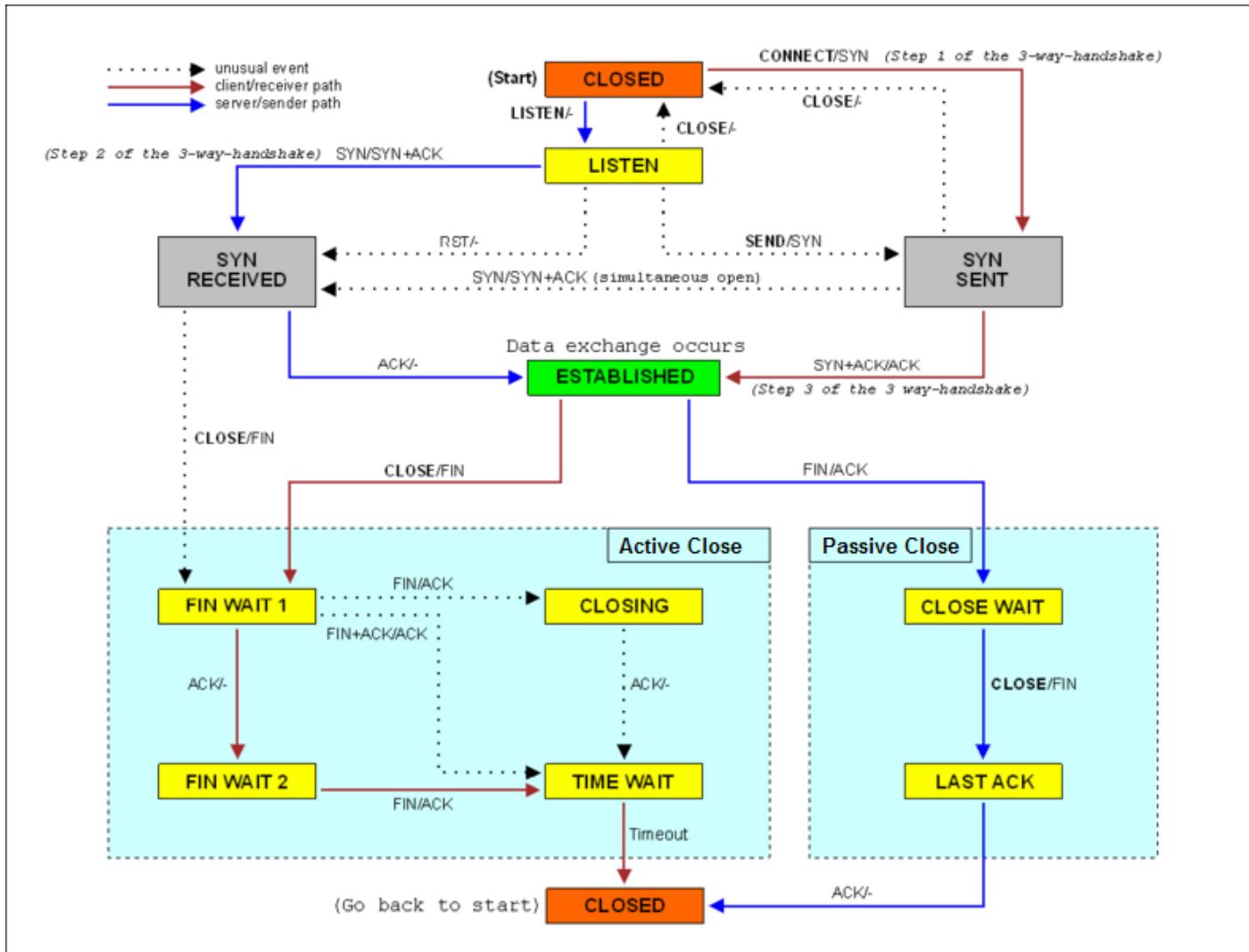
# Applications of TCP

- Most things!
  - HTTP, FTP, ...
- Saves the application a lot of work, so used unless there's a good reason not to

# TCP implementation

- Connections are established using a *three-way handshake*
- Data is divided up into packets by the operating system
- Packets are numbered, and received packets are acknowledged
- Connections are explicitly closed
  - (or may abnormally terminate)

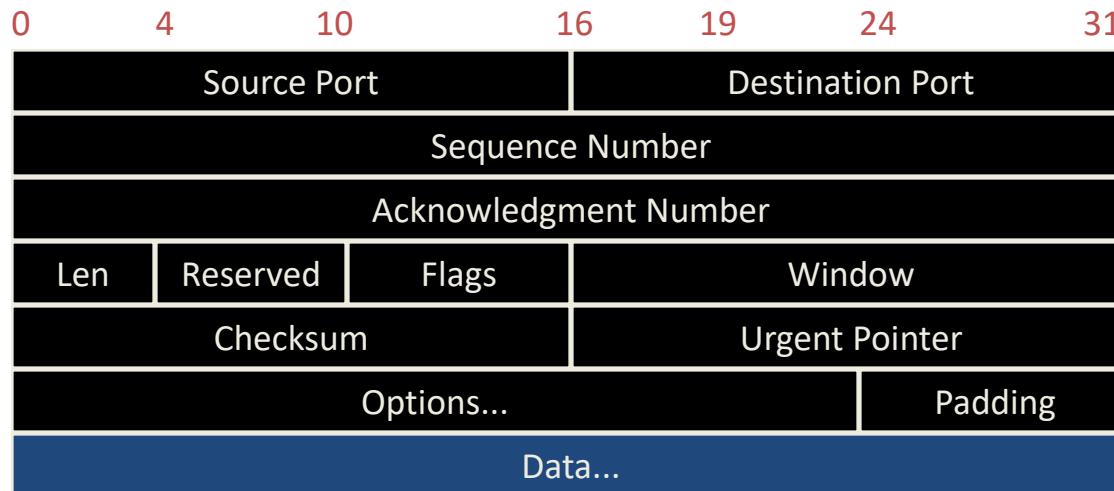
# TCP connection state diagram



# TCP Packets

- Source + destination ports
- Sequence number (used to order packets)
- Acknowledgement number (used to verify packets are received)

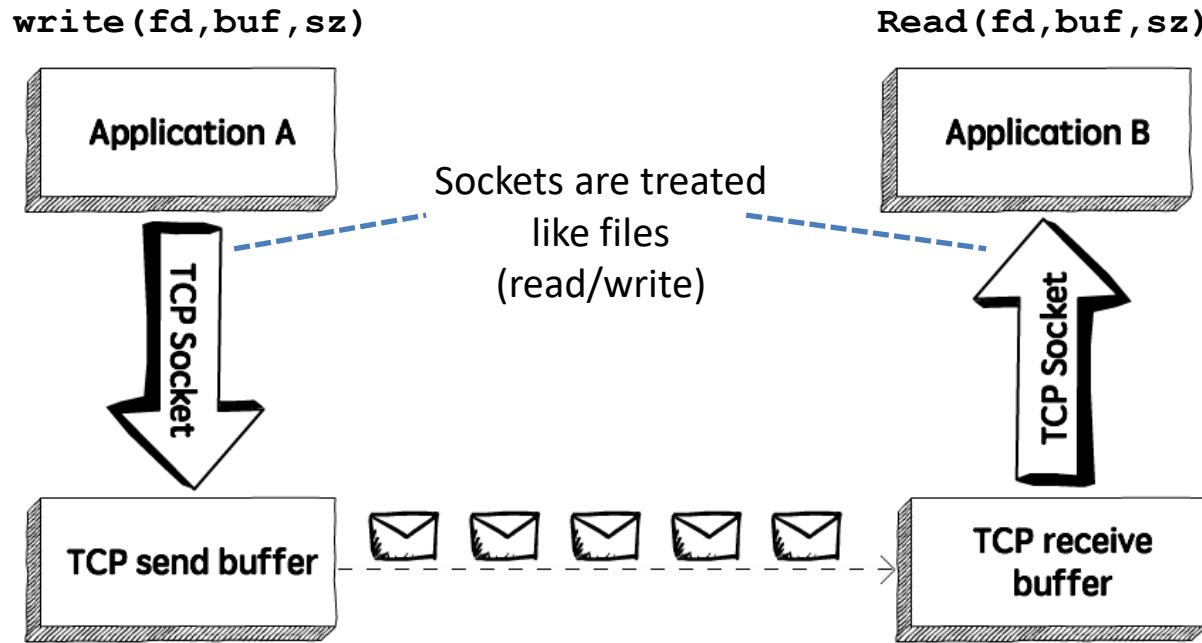
# TCP Segment



Field	Purpose
Source Port	Identifies originating application
Destination Port	Identifies destination application
Sequence Number	Sequence number of first octet in the segment
Acknowledgment #	Sequence number of the next expected octet (if ACK flag set)
Len	Length of TCP header in 4 octet units
Flags	TCP flags: SYN, FIN, RST, PSH, ACK, URG
Window	Number of octets from ACK that sender will accept
Checksum	Checksum of IP pseudo-header + TCP header + data
Urgent Pointer	Pointer to end of "urgent data"
Options	Special TCP options such as MSS and Window Scale

You just need to know port numbers, seq and ack are added

# Basics of TCP/IP



- How do we make sure data is not lost and delivered in order ?
- We want to utilize available bandwidth (changing over time due to other users)
- How much can we send? If application doesn't consume we must buffer ( limited )
- How to throttle traffic on ingestion, often your CPU (app) can produce more network data (e.g. 200Gbps) than your network card can handle (10Gbps)

# Example

- `int socket(int domain, int type, int protocol);`  
domain: AF\_UNIX, AF\_INET, AF\_INET6, ..  
type: SOCK\_STREAM, SOCK\_DGRAM, ...  
creates a socket object
- `int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);`  
Binds a socket object to an IP(..) address

```
struct sockaddr {  
    sa_family_t sa_family;  
    char      sa_data[14];  
}  
  
myname.sin_family = AF_INET;  
myname.sin_addr.s_addr = inet_addr("129.6.25.3");  
/* specific interface */  
myname.sin_port = htons(1024);
```

# Example

- `int listen(int sockfd, int backlog);`  
Tells OS to prepare for incoming requests at that socket address (`SOCK_STREAM`) and buffer up to `<backlog>` outstanding requests.
- `int accept(int sockfd,  
              struct sockaddr *restrict addr,  
              socklen_t *restrict addrlen);`  
accepts connection (after `socket/bind/listen`) and returns new socket with `addr/addrlen` filled with connection peer information. `Sockfd` can continue to accept other connection requests.

# Application Interface

## CLIENT

```
int sd = socket();
```

```
connect(sd);  
  
read(sd, ...);  
write(sd, ...);  
close(sd);
```

## SERVER

```
int sd = socket();
```

```
bind(sd);
```

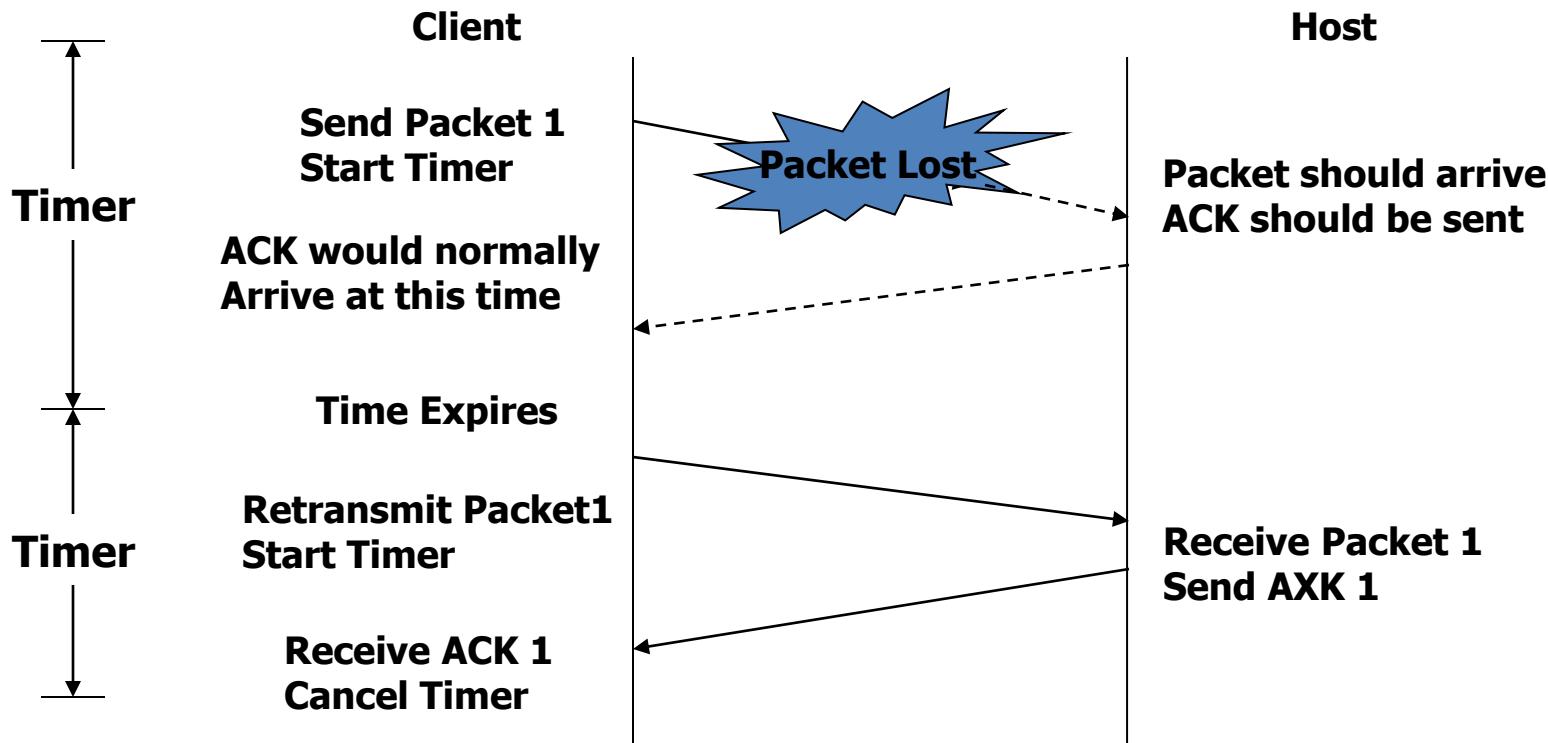
```
listen(sd, ...);
```

```
while (cond) {  
    sd2 = accept(sd, ...);
```

```
        write(sd2, ...);  
        read(sd2, ...);  
        close(sd2);  
    }  
close(sd);
```



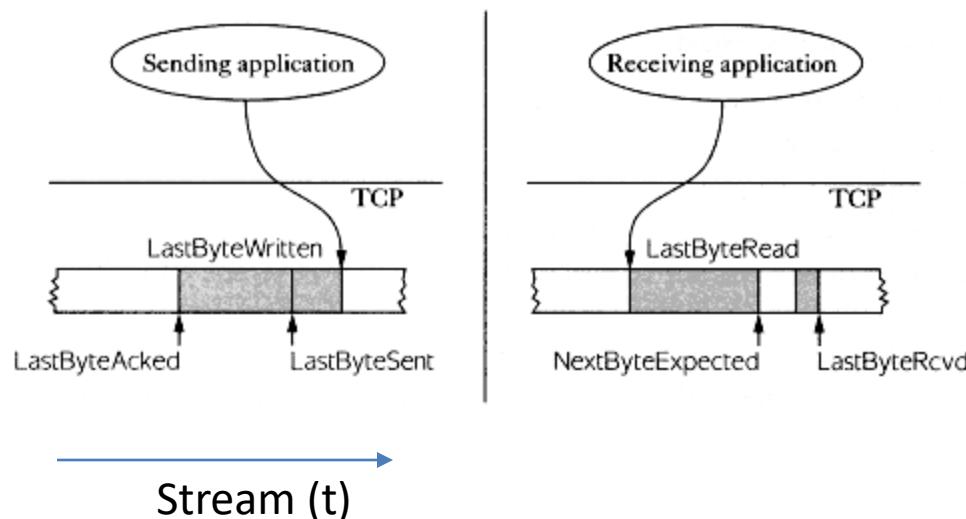
# TCP : Data transfer



- Several packets can be transmitted, while waiting for acks; this is not serialized
- Up the TCP/IP subsystem to keep track of outstanding packets

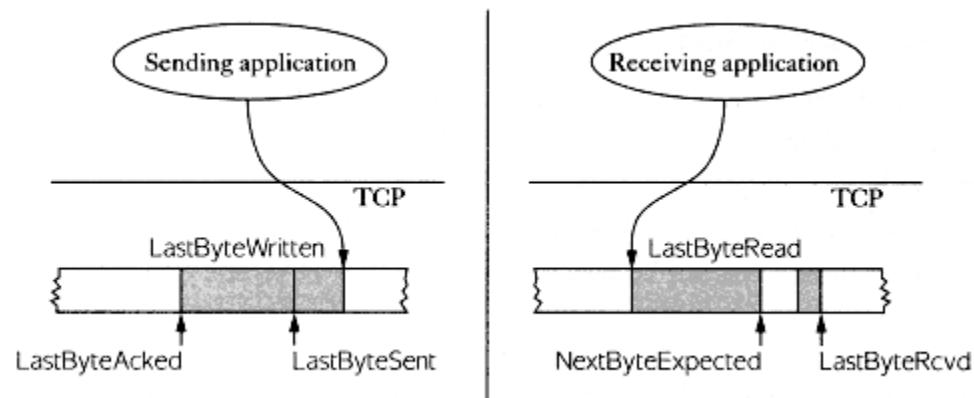
# TCP/IP Details

- TCP/IP uses a sliding window to buffer and control data flow
- The sliding window serves several purposes:
  - (1) it guarantees the reliable delivery of data
  - (2) it ensures that the data is delivered in order,
  - (3) it enforces flow control between the sender and the receiver.



# Flow Control

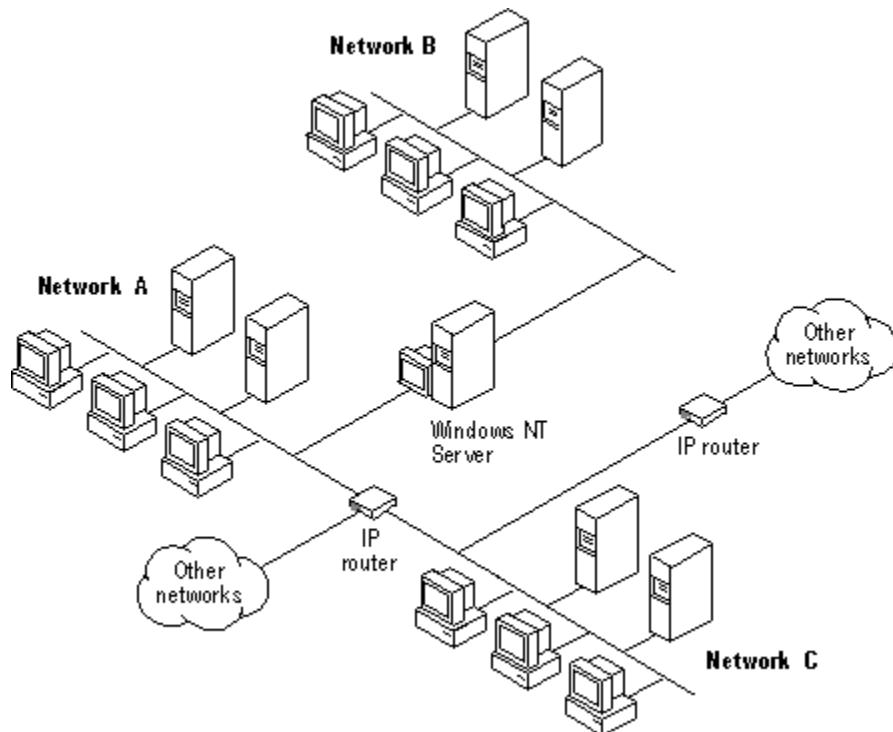
- Max Send and Receive Buffer sizes
- In order delivery to the consumer
- Acknowledgement of reception
- Retransmit when ack is not received in RTT (RoundTripTime) setting
- Only when last byte is ack'd can the window move



# Congestion Control

- Window size synonymous with current available bandwidth:
  - No error → we could push more data → increase the window
  - Error → we are pushing too much → decrease the window
- Slow Start
  - Start with 1 congestion window and then doubling it
- When ack timeout occurs reduce the window size
- Fast Retransmit
  - When out of order packet is received immediately ACK
- Fast Recovery
- Many different Flow Control protocols

# Routing



# Routing

- How does a device know where to send a packet?
  - All devices need to know what IP addresses are on directly attached networks
  - If the destination is on a local network, send it directly there
- How do we discover these IP addresses
  - DNS & ARP

# Routing (cont)

- If the destination address isn't local
  - Most non-router devices just send everything to a single local router (aka gateway)
  - Routers need to know which network corresponds to each possible IP address

```
[frankeh@access2 ~]$ route
Kernel IP routing table
Destination     Gateway         Genmask        Flags Metric Ref    Use Iface
default         nyu-vl424-gw.ne 0.0.0.0      UG    100    0        0 eth0
128.122.49.0   0.0.0.0        255.255.255.0 U     100    0        0 eth0
128.122.49.0   0.0.0.0        255.255.255.0 U     100    0        0 eth0
```

```
[frankeh@access2 ~]$ traceroute 8.8.8.8
traceroute to 8.8.8.8 (8.8.8.8), 30 hops max, 60 byte packets
1 wwhgwa-vl424.net.nyu.edu (128.122.49.2)  0.356 ms  0.368 ms  0.364 ms
2 10.254.4.20 (10.254.4.20)  0.330 ms  0.342 ms  0.307 ms
3 128.122.1.36 (128.122.1.36)  0.382 ms  0.371 ms  0.371 ms
4 ngfw-palo-vl1500.net.nyu.edu (192.168.184.228)  0.806 ms  0.759 ms  0.733 ms
5 nyugwa-outside-ngfw-vl3080.net.nyu.edu (128.122.254.114)  1.023 ms  1.055 ms  1.021 ms
6 dmzgwb-ptp-nyugwa-vl3082.net.nyu.edu (128.122.254.111)  1.513 ms  1.300 ms  1.440 ms
7 128.122.254.70 (128.122.254.70)  0.952 ms  1.043 ms  0.995 ms
8 ix-xe-7-3-2-0.tcore2.nw8-new-york.as6453.net (64.86.62.13)  1.351 ms  1.318 ms  1.265 ms
9 if-ae-0-2.tcore1.nw8-new-york.as6453.net (209.58.75.217)  1.898 ms  1.976 ms  1.863 ms
10 if-ae-3-2.tcore1.n0v-new-york.as6453.net (216.6.90.72)  3.668 ms  3.677 ms  2.304 ms
11 72.14.223.124 (72.14.223.124)  2.133 ms  2.068 ms  2.042 ms
12 108.170.248.33 (108.170.248.33)  3.020 ms  3.206 ms  3.131 ms
13 209.85.245.99 (209.85.245.99)  2.064 ms  108.170.235.183 (108.170.235.183)  2.303 ms  209.85.243.193 (209.85.243.193)  2.060 ms
14 google-public-dns-a.google.com (8.8.8.8)  2.076 ms  2.051 ms  2.283 ms
```

# DNS: (domain name service)

## Name to IP lookup

```
frankeh@NYU2: nslookup access.cims.nyu.edu  
Server: 127.0.1.1  
Address: 127.0.1.1#53
```

Non-authoritative answer:

```
Name: access.cims.nyu.edu  
Address: 128.122.49.15  
Name: access.cims.nyu.edu  
Address: 128.122.49.16
```

```
frankeh@NYU2: nslookup www.cnn.com  
Server: 127.0.1.1  
Address: 127.0.1.1#53
```

Non-authoritative answer:

```
www.cnn.com canonical name = turner-tls.map.fastly.net.  
Name: turner-tls.map.fastly.net  
Address: 151.101.209.67
```

Name lookups can change  
Reflecting that multiple systems  
serve the same service/content

```
frankeh@linax1[~]$ nslookup www.cnn.com  
Server: 128.122.253.79  
Address: 128.122.253.79#53
```

```
Non-authoritative answer:  
www.cnn.com canonical name = turner-tls.map.fastly.net.  
Name: turner-tls.map.fastly.net  
Address: 151.101.117.67
```

# ARP ( Address Resolution Protocol)

- Protocol to discover Link layer address (e.g. MAC address)
- who has IP <128.122.49.137> ?
  - broadcasts address on local network
  - owning node responds with MAC
  - receiving node caches the MAC in ARP cache

```
frankeh@linax1[~]$ arp -n
Address          HWtype  HWaddress          Flags Mask   Iface
128.122.49.137  ether    52:54:00:c2:72:79  C      eth1
128.122.49.112  ether    52:54:00:67:66:e2  C      eth1
128.122.49.99   ether    00:14:4f:2b:0f:ae  C      eth1
128.122.49.10   ether    24:6e:96:19:24:28  C      eth1
128.122.49.3    ether    88:75:56:3c:a6:c0  C      eth1
128.122.49.75   ether    52:54:00:d5:b0:2e  C      eth1
128.122.49.95   ether    00:21:28:a3:10:43  C      eth1
```

```
frankeh@linax1[~]$ arp
Address          HWtype  HWaddress          Flags Mask   Iface
VM-PUBPC11.CIMS.NYU.EDU  ether    52:54:00:c2:72:79  C      eth1
PROXY1.CIMS.NYU.EDU       ether    52:54:00:67:66:e2  C      eth1
MX.CIMS.NYU.EDU          ether    00:14:4f:2b:0f:ae  C      eth1
FS-U2.CIMS.NYU.EDU        ether    24:6e:96:19:24:28  C      eth1
WSSGW-VL424.NET.NYU.EDU   ether    88:75:56:3c:a6:c0  C      eth1
WEBMAIL.CIMS.NYU.EDU      ether    52:54:00:d5:b0:2e  C      eth1
MAILFS.CIMS.NYU.EDU       ether    00:21:28:a3:10:43  C      eth1
```

# Network setup

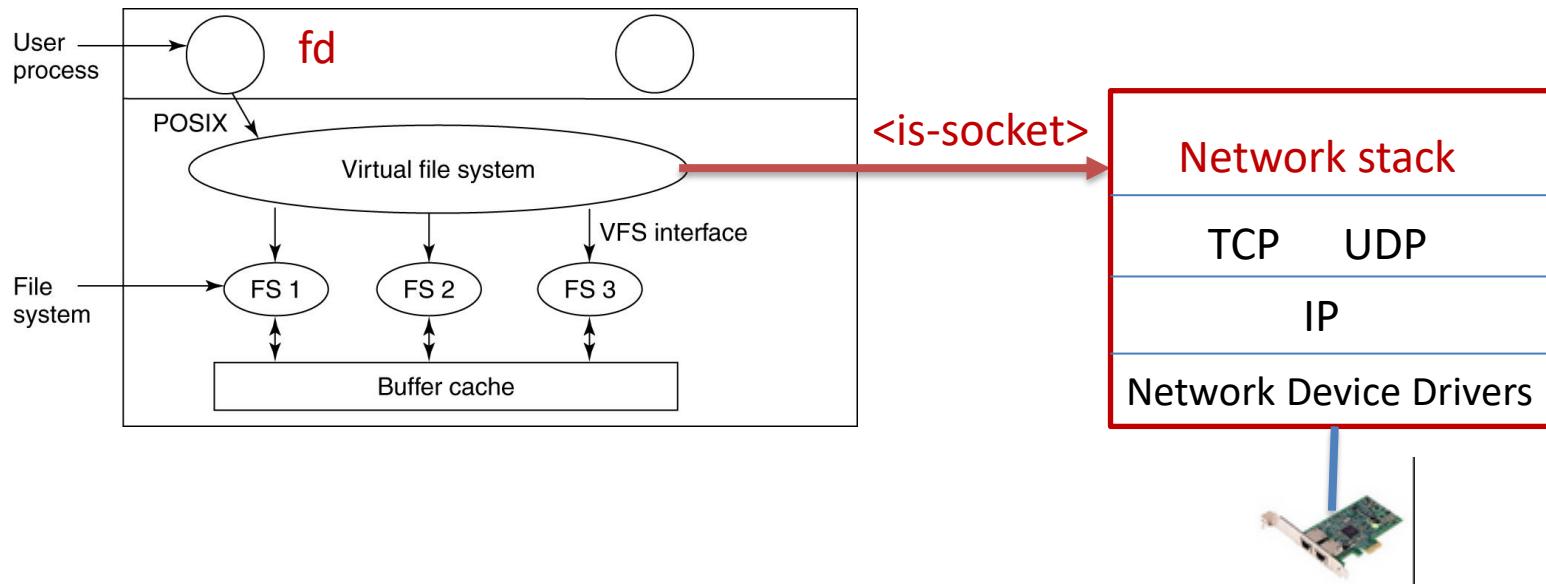
- Static
  - information is stored in filesystem
  - /etc/network/interfaces
- Dynamic
  - information is retrieved from DHCP (dynamic host configuration protocol)
  - DHCP server provides config
  - /etc/network/interfaces

```
auto eth0
iface eth0 inet static
address 192.168.0.42
network 192.168.0.0
netmask 255.255.255.0
broadcast 192.168.0.255
gateway 192.168.0.1
```

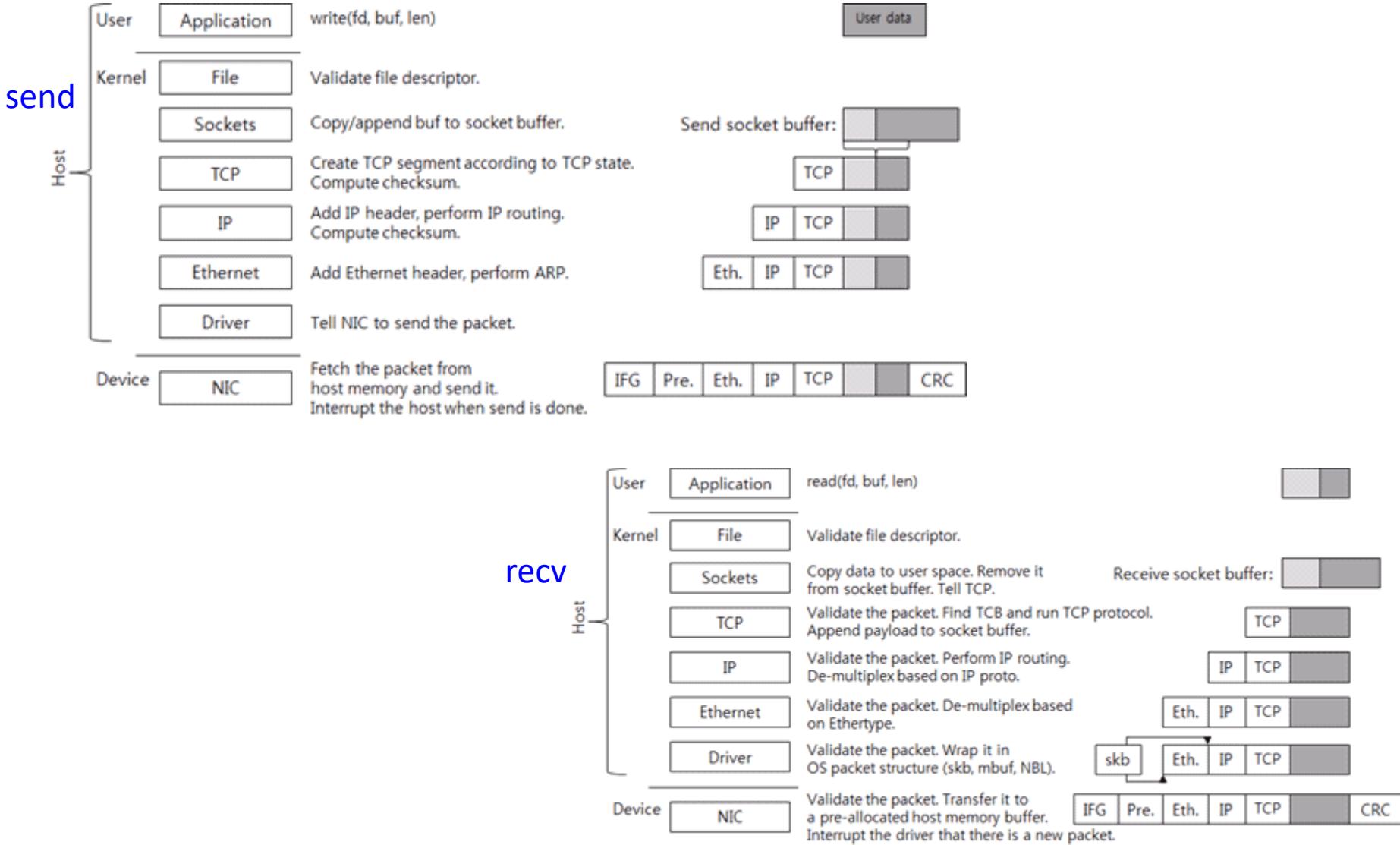
```
auto eth0
iface eth0 inet dhcp
```

# Kernel Interface

- Unix → everything is a file descriptor
- Sockets are special file handles
- Diverted at high level into network stack



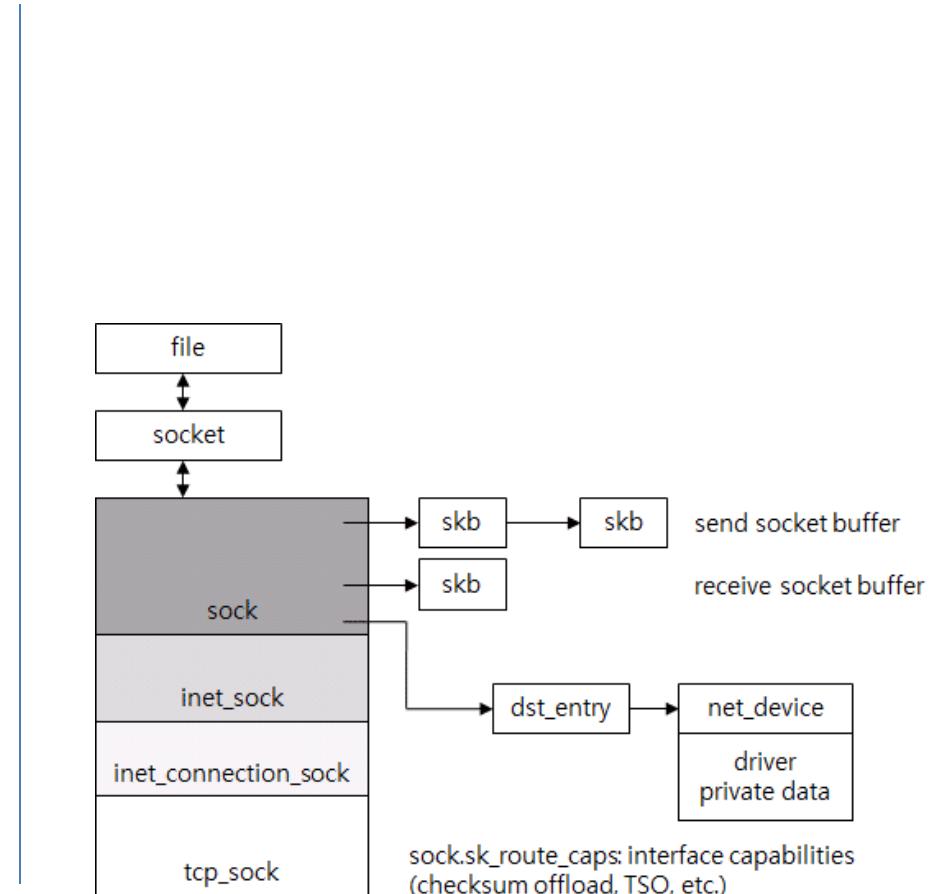
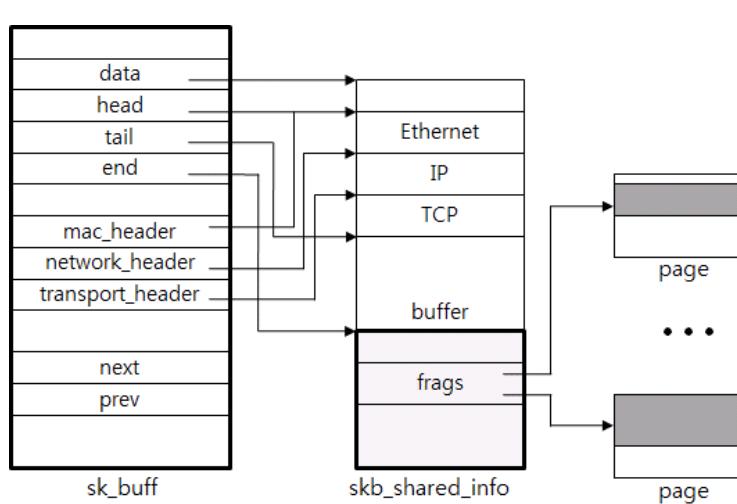
# Putting it all together



(source: <https://www.cubrid.org/blog/understanding-tcp-ip-network-stack> )

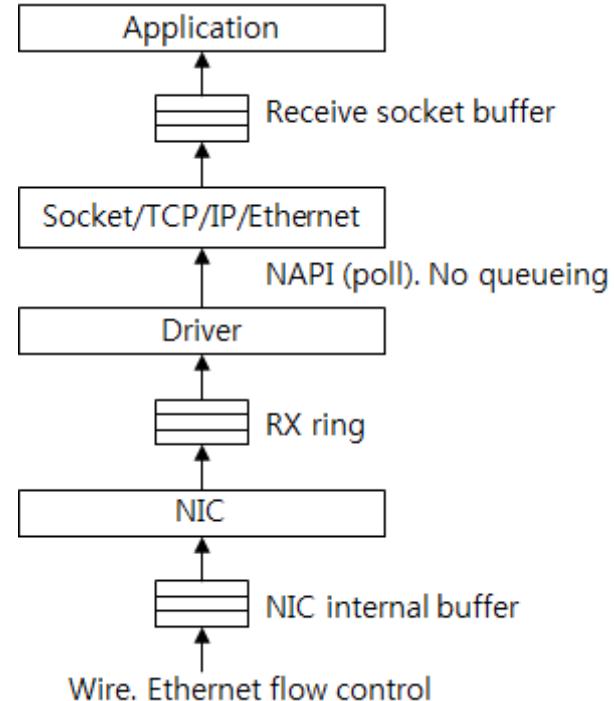
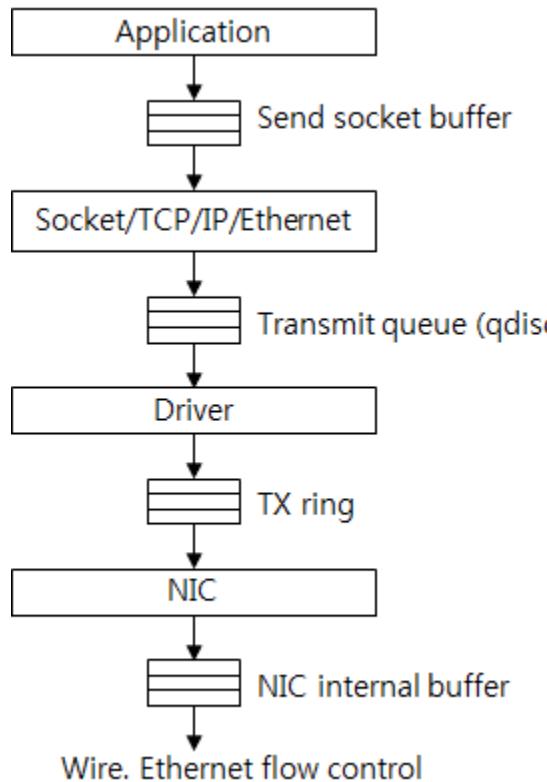
# SKB (Socket Kernel Buffer)

Generic object for storing socket data.



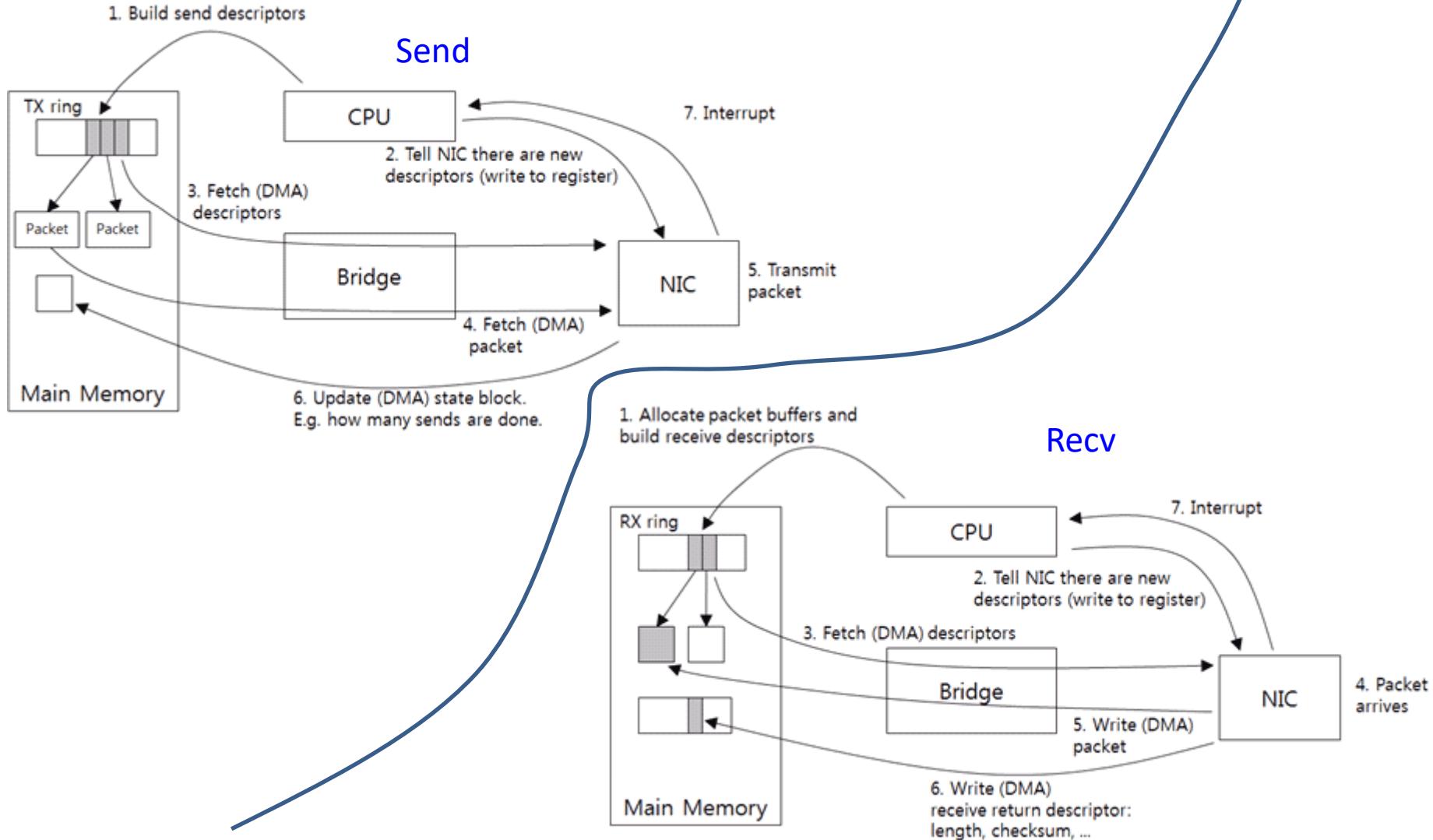
(source: <https://www.cubrid.org/blog/understanding-tcp-ip-network-stack> )

# Down and Up flow of Data



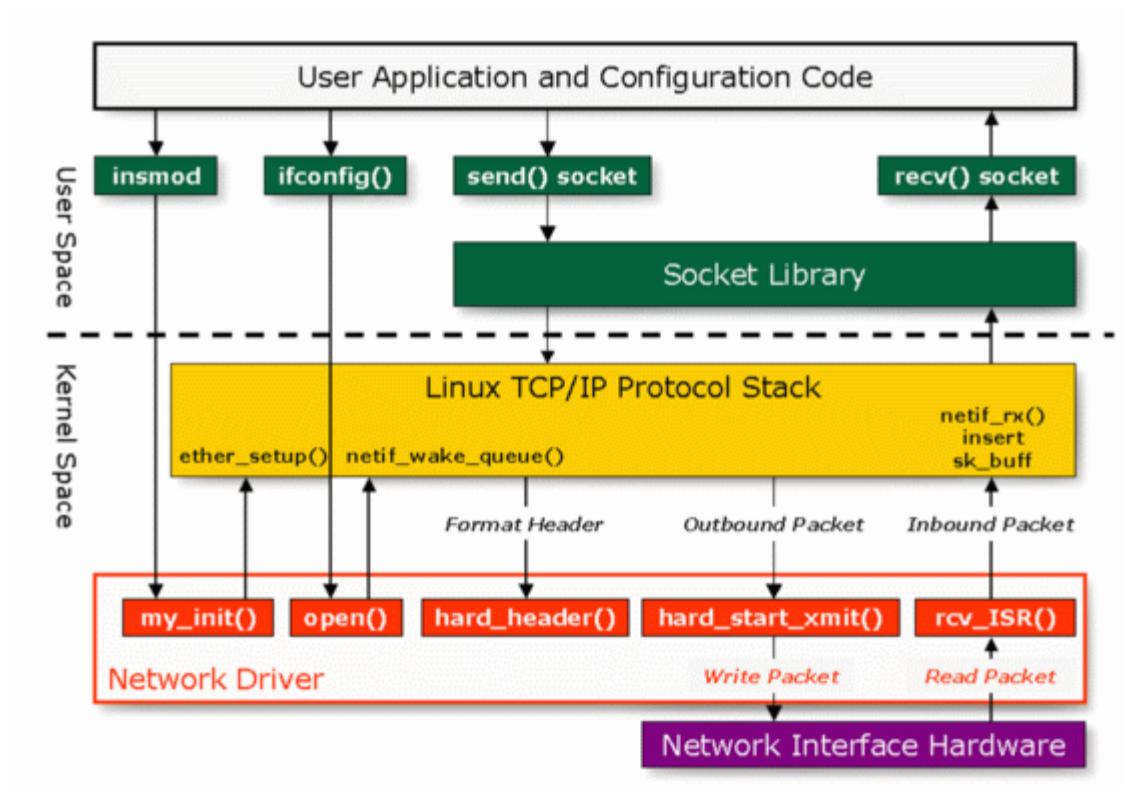
(source: <https://www.cubrid.org/blog/understanding-tcp-ip-network-stack> )

# Down and Up flow of Data



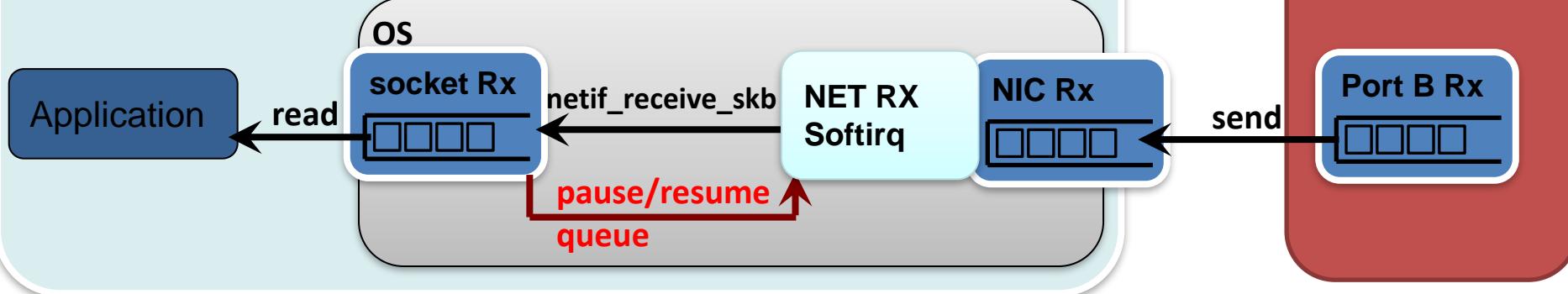
(source: <https://www.cubrid.org/blog/understanding-tcp-ip-network-stack> )

# HighLevel View: Network Stack

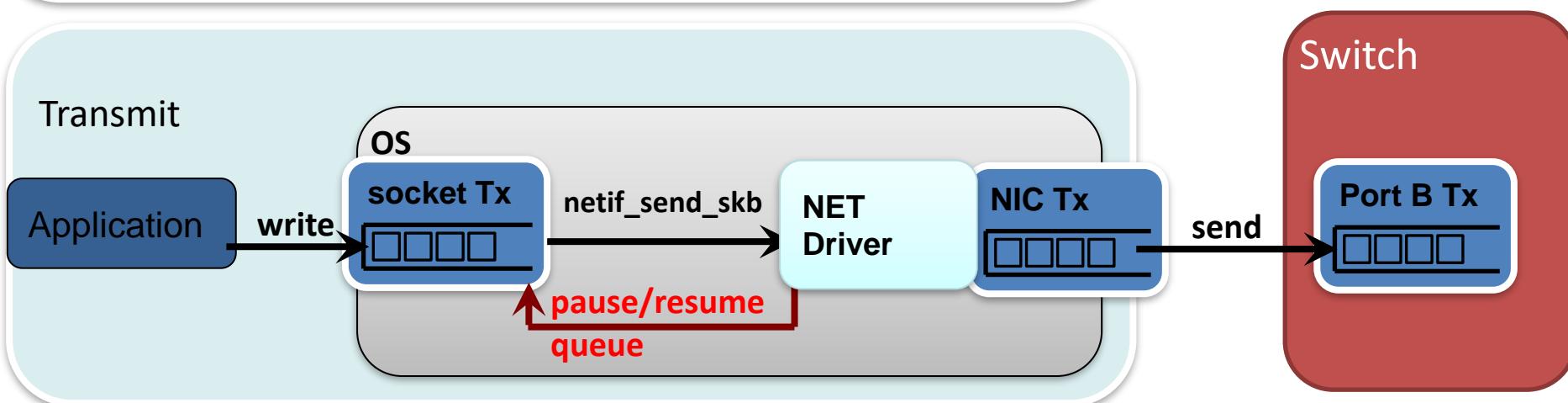


# Device Driver Details

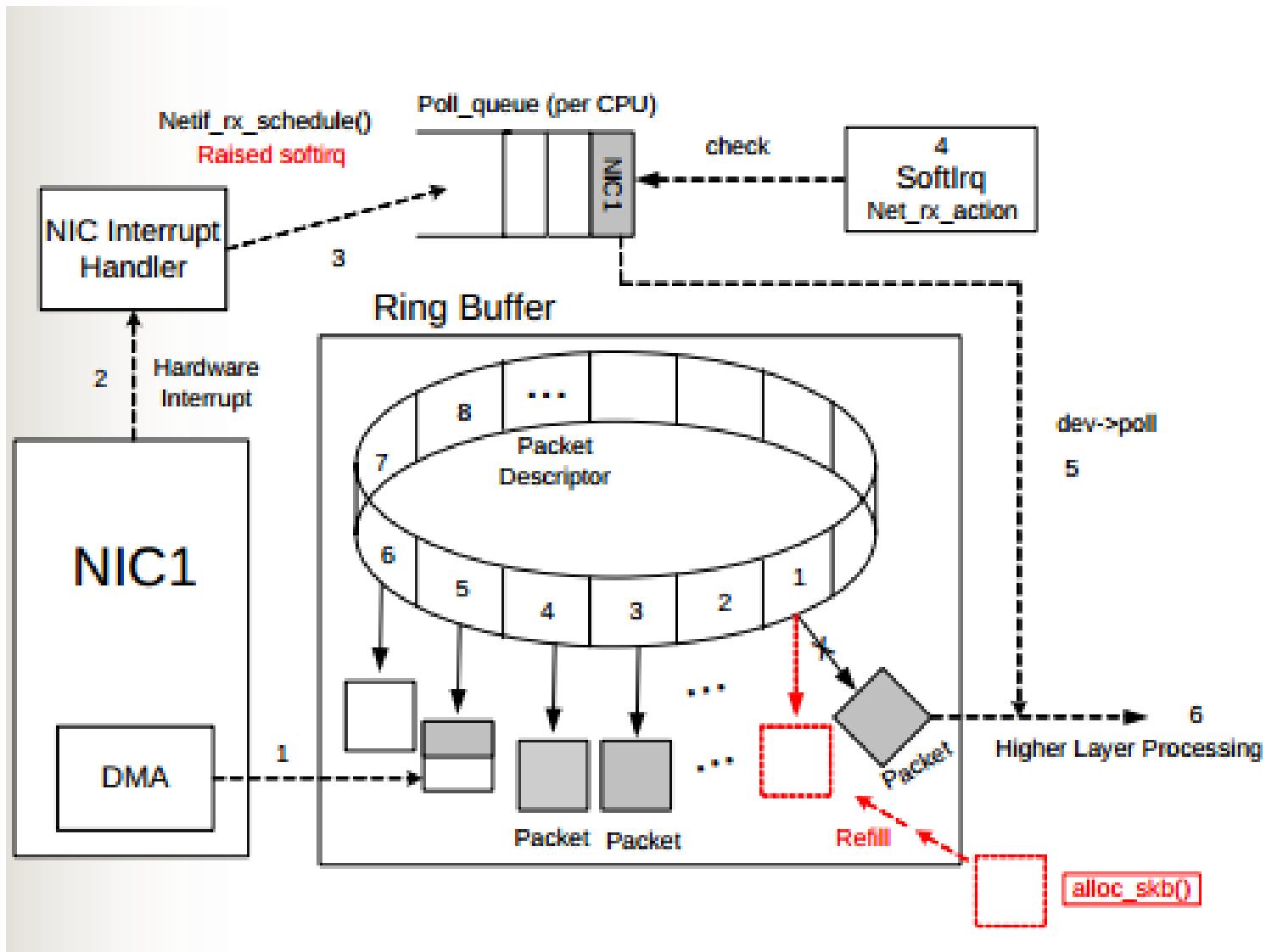
Receive



Transmit



# Linux Tx/Rx Ring handling



# Network Security

- Utilizes keys, encryption, and encapsulation
- Keys are exchanged ( e.g. RSA)

The encryption key is public and it is different from the decryption key which is kept secret.

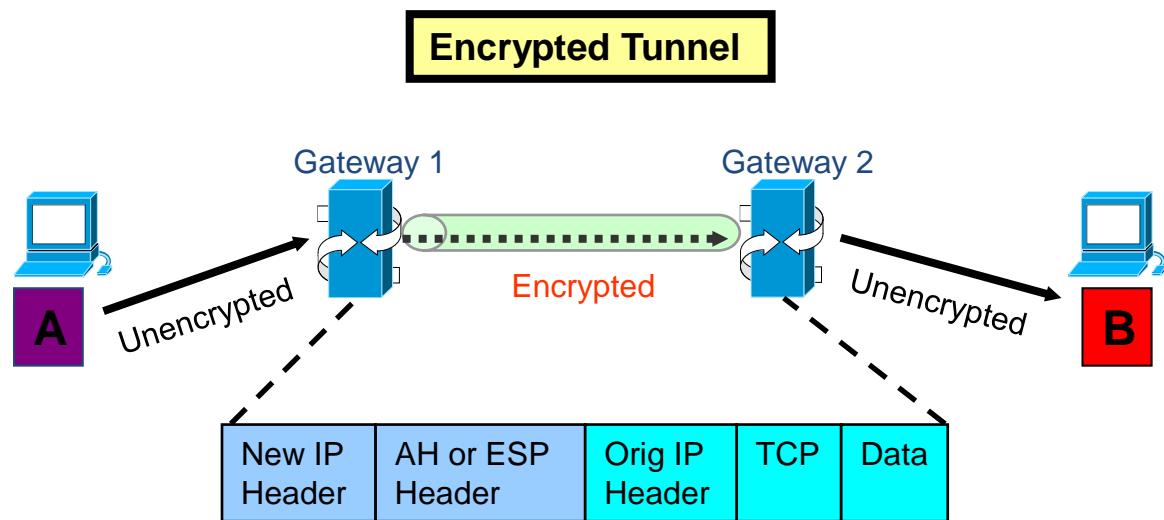
- Can be done at
  - application level ( https , ssh, sftp, ..)  
using  
a) SSL (secure socket layer) now deprecated or  
b) TLS (transport layer security)
  - kernel level (Ipsec)

# TLS setup/handshake

- Client:
  - connects to a TLS-enabled server requesting a secure connection and the client presents a list of supported ciphers and hash
    - Cipher: Diffie Hellman , Eliptic Curve, ...
    - Hash Function: MD5, SHA-1, SHA-2, ...
- Server:
  - selects a cipher and hash function it supports and notifies the client of the decision.
  - provides identification in the form of a digital certificate.  
The certificate contains the server name, the trusted certificate authority (CA) that vouches for the authenticity of the certificate, and the server's public encryption key.
- Client:
  - confirms the validity of the certificate.
  - Initiate Session Key (symmetric) using Diffie Hellman key exchange
  - Session Key used for all communication on secure connection

# Modes

- Transport mode: host → host
- Tunnel mode: host->gateway or gateway->gateway



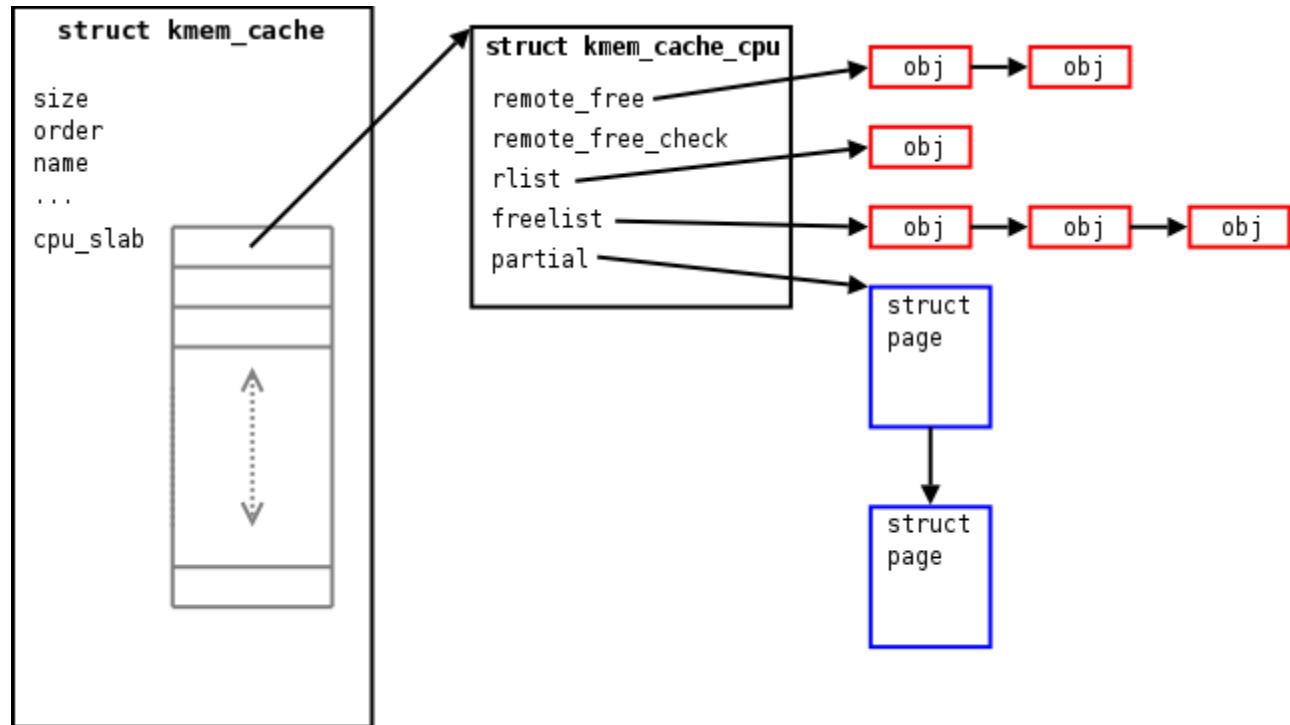
Authentication header (AH)  
Encapsulating security payload (ESP)

# Some other useful general "stuff" (not just network)

- Slab-Cache
  - The primary motivation for slab allocation:
    - initialization and destruction of kernel data objects can actually outweigh the cost of allocating memory for them.
    - As object creation and deletion are widely employed by the kernel, mitigating overhead costs of initialization can result in significant performance gains.
    - “object caching” was therefore introduced in order to avoid the invocation of functions used to initialize object state.
  - Group same dynamically allocated objects under one “allocator” object
  - E.g. `skb_buff_alloc()`

# Some useful general “stuff”

- General implementation



- Other features:
  - Coloring → Cache utilization

# Linux Example of slab caches

- ">#> slabtop

OBJS	ACTIVE	USE	OBJ	SIZE	SLABS	OBJ/SLAB	CACHE	SIZE	NAME
48720	48701	99%	0.13K	1624	30	6496K	dentry		
41408	41407	99%	0.06K	647	64	2588K	buffer_head		
33969	33968	99%	0.61K	2613	13	20904K	ext4_inode_cache		
13754	13608	98%	0.09K	299	46	1196K	vm_area_struct		
10240	10092	98%	0.01K	20	512	80K	kmalloc-8		
9180	9176	99%	0.05K	108	85	432K	sysfs_dir_cache		
6656	6440	96%	0.12K	208	32	832K	kmalloc-128		
4608	4547	98%	0.02K	18	256	72K	anon_vma		
3712	3458	93%	0.03K	29	128	116K	kmalloc-32		
3584	3103	86%	0.02K	14	256	56K	kmalloc-16		
3471	3415	98%	0.29K	267	13	1068K	radix_tree_node		
2893	2889	99%	0.35K	263	11	1052K	inode_cache		
2240	2158	96%	0.06K	35	64	140K	kmalloc-64		
1580	1570	99%	0.38K	158	10	632K	proc_inode_cache		
1530	1517	99%	0.04K	15	102	60K	Acpi-Operand		
968	937	96%	0.50K	121	8	484K	kmalloc-512		
966	958	99%	0.09K	23	42	92K	kmalloc-96		
861	791	91%	0.19K	41	21	164K	kmalloc-192		
782	778	99%	0.45K	46	17	368K	shmem_inode_cache		
666	653	98%	0.44K	74	9	296K	mm_struct		
624	621	99%	2.00K	78	8	1248K	kmalloc-2048		



CSCI-GA.2250-001

# Operating Systems

## Security

### (small digest)

Hubertus Franke  
[frankeh@cs.nyu.edu](mailto:frankeh@cs.nyu.edu)



# The Security Environment

## Threats

<b>Goal</b>	<b>Threat</b>
Confidentiality	Exposure of data
Integrity	Tampering with data
Availability	Denial of service

Figure 9-1. Security goals and threats.

# Can We Build Secure Systems?

Two questions concerning security:

1. Is it possible to build a secure computer system?
2. If so, why is it not done?

# Trusted Computing Base

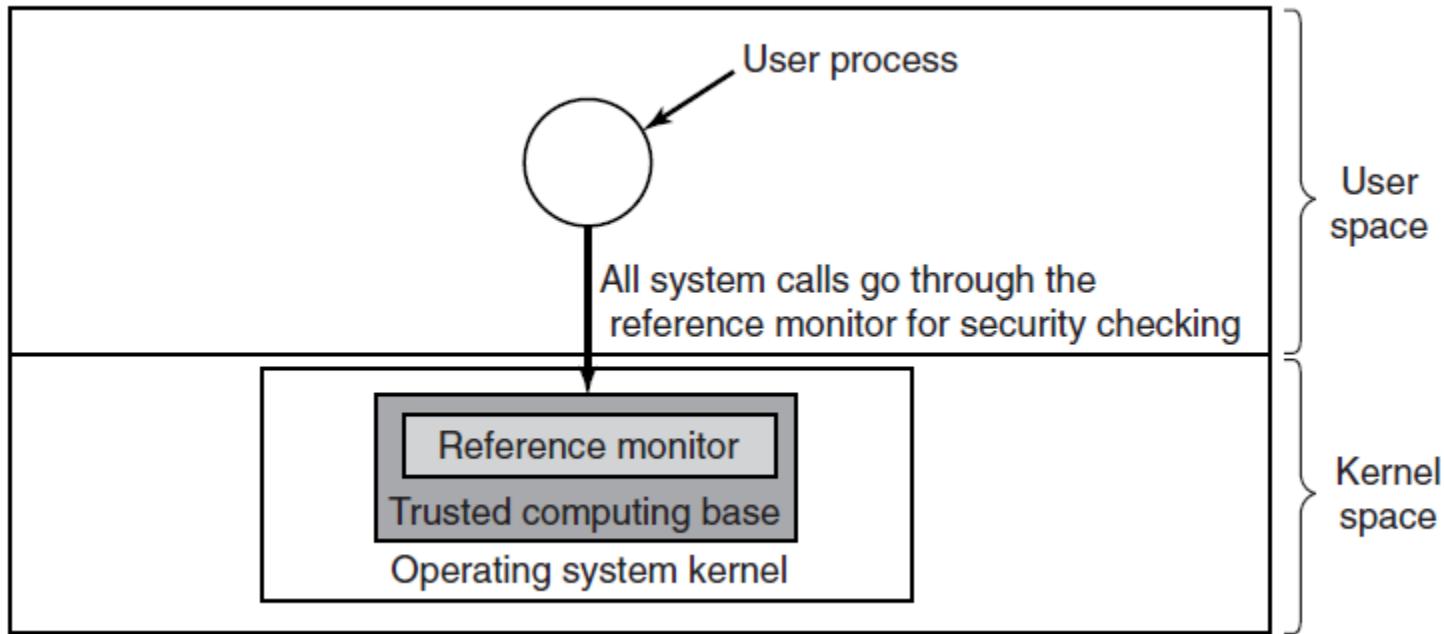


Figure 9-2. A reference monitor.

# Protection Domains (1)

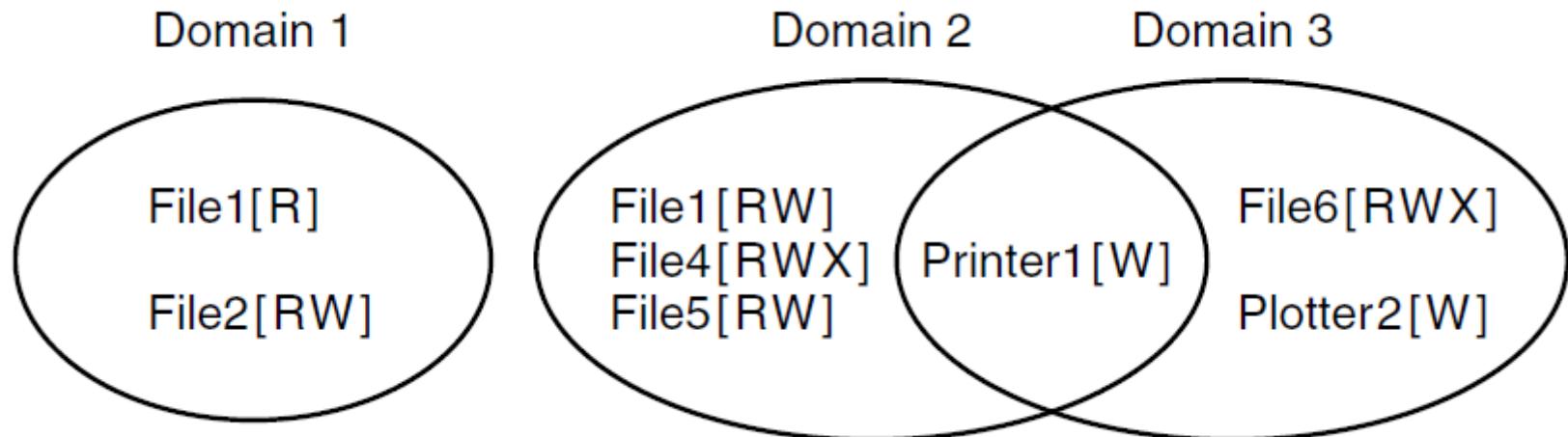


Figure 9-3. Three protection domains.

# Protection Domains (2)

		Object							
		File1	File2	File3	File4	File5	File6	Printer1	Plotter2
Domain	1	Read	Read Write						
	2			Read	Read Write Execute	Read Write		Write	
	3						Read Write Execute	Write	Write

Figure 9-4. A protection matrix.

# Access Control Lists (1)

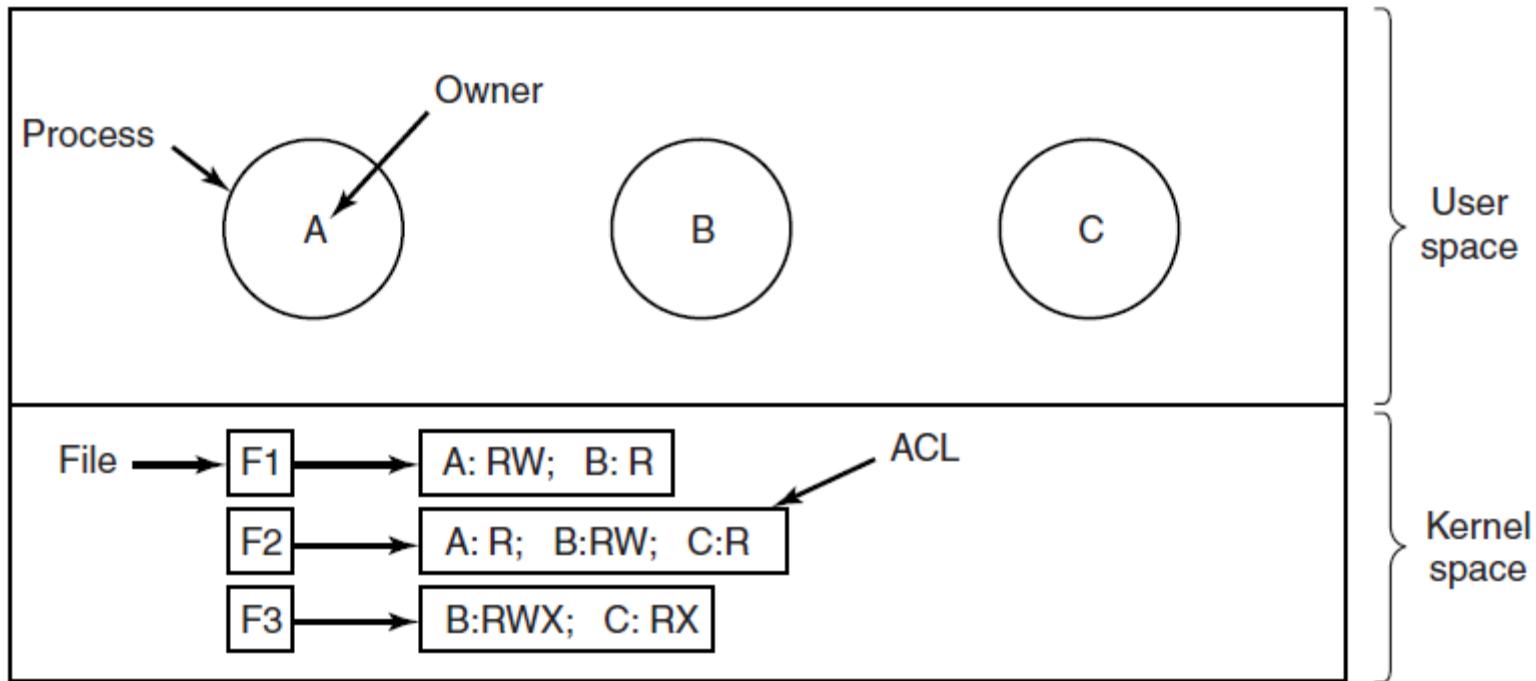


Figure 9-6. Use of access control lists to manage file access.

# Access Control Lists (2)

<b>File</b>	<b>Access control list</b>
Password	tana, sysadm: RW
Pigeon_data	bill, pigfan: RW; tana, pigfan: RW; ...

Figure 9-7. Two access control lists.

# Capabilities (1)

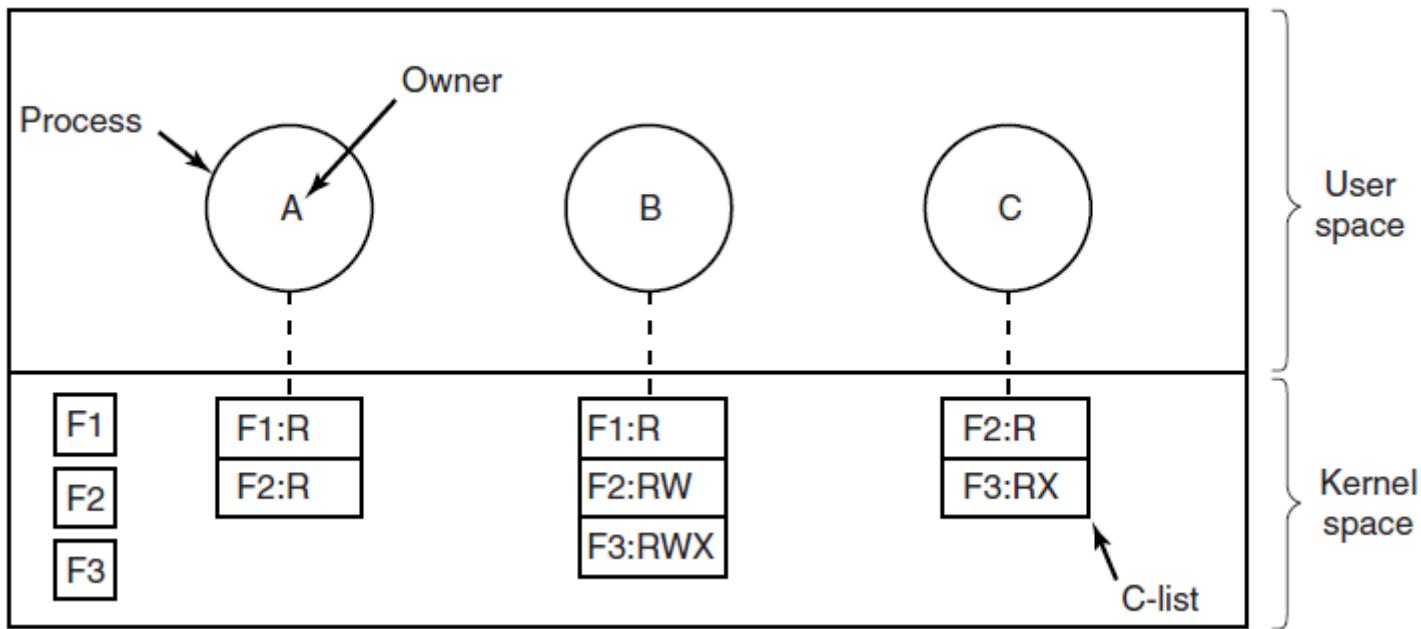


Figure 9-8. When capabilities are used, each process has a capability list.

# Capabilities (2)

Server	Object	Rights	$f(\text{Objects}, \text{Rights}, \text{Check})$
--------	--------	--------	--------------------------------------------------

Figure 9-9. A cryptographically protected capability.

# Covert Channels (1)

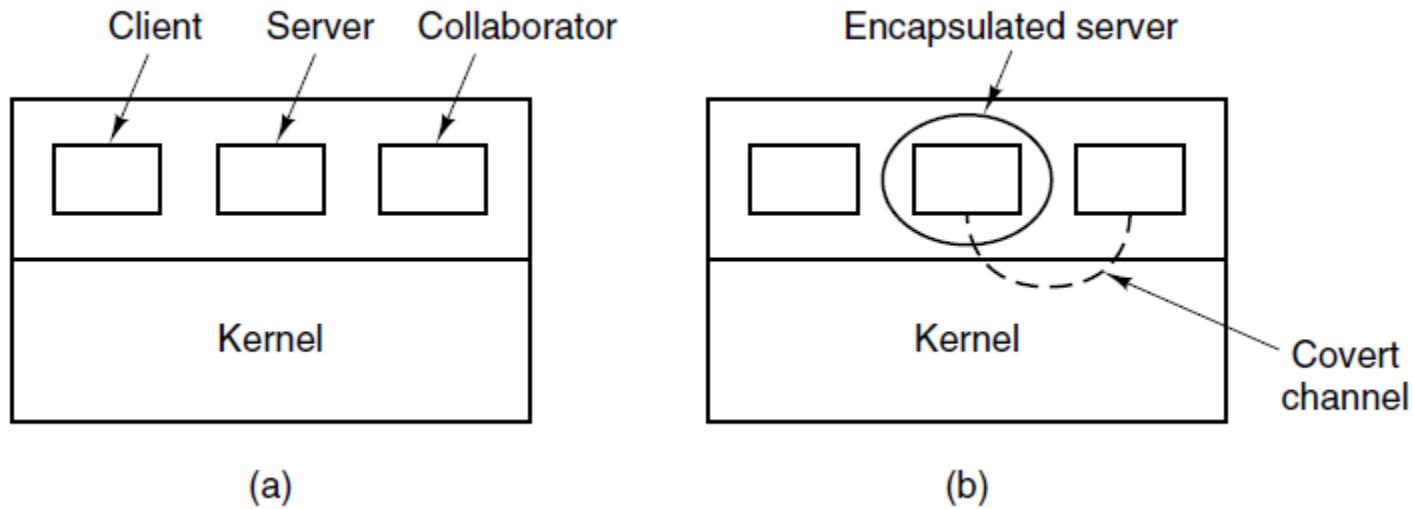


Figure 9-12. (a) The client, server, and collaborator processes. (b) The encapsulated server can still leak to the collaborator via covert channels.

# Covert Channels (2)

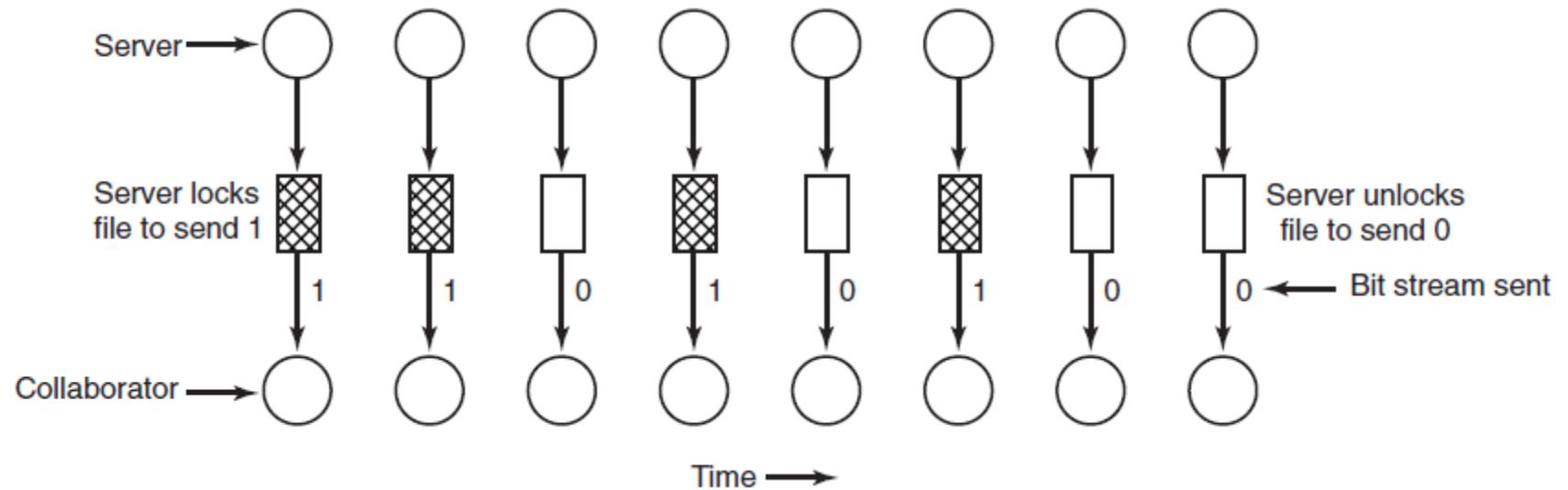
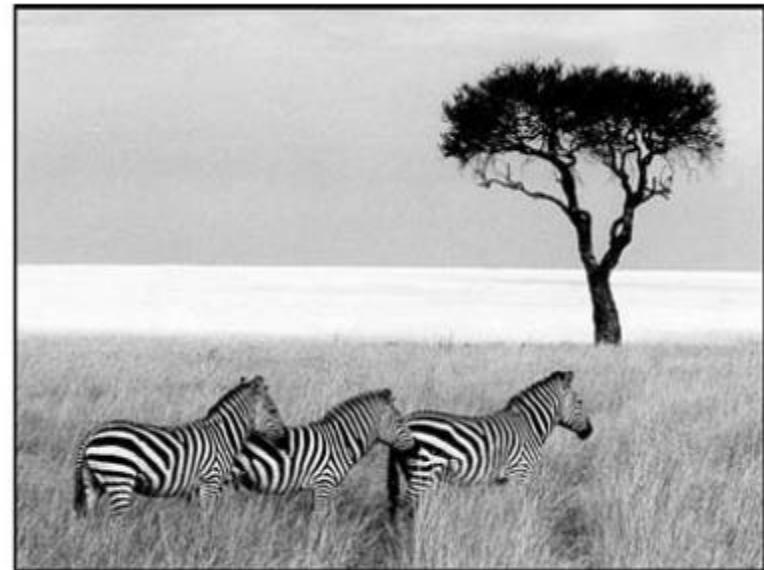


Figure 9-13. A covert channel using file locking.

# Steganography



(a)



(b)

Figure 9-14. (a) Three zebras and a tree. (b) Three zebras, a tree, and the complete text of five plays by William Shakespeare.

# Basics of Cryptography

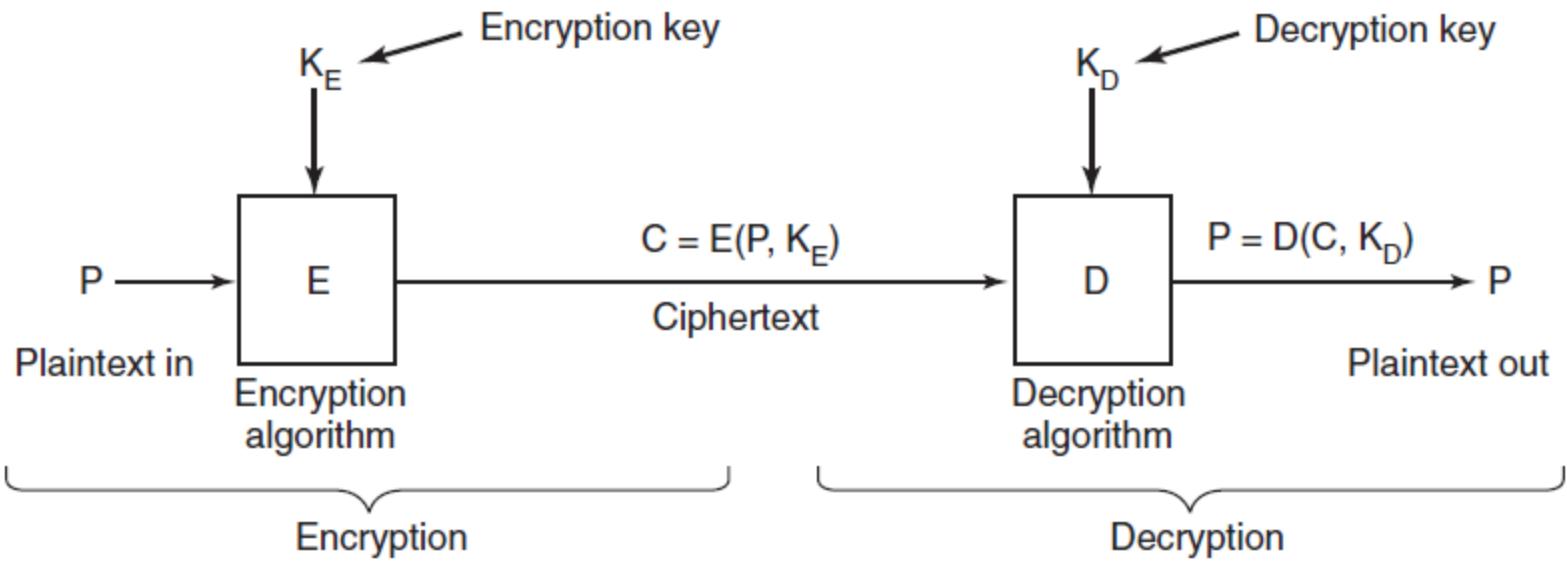
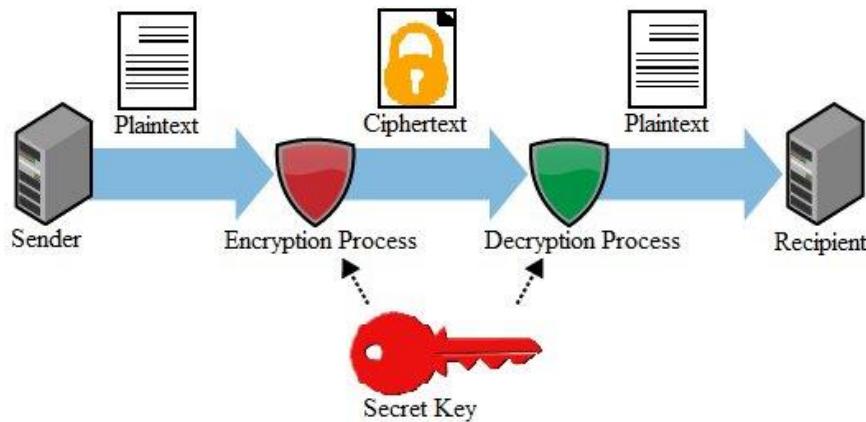
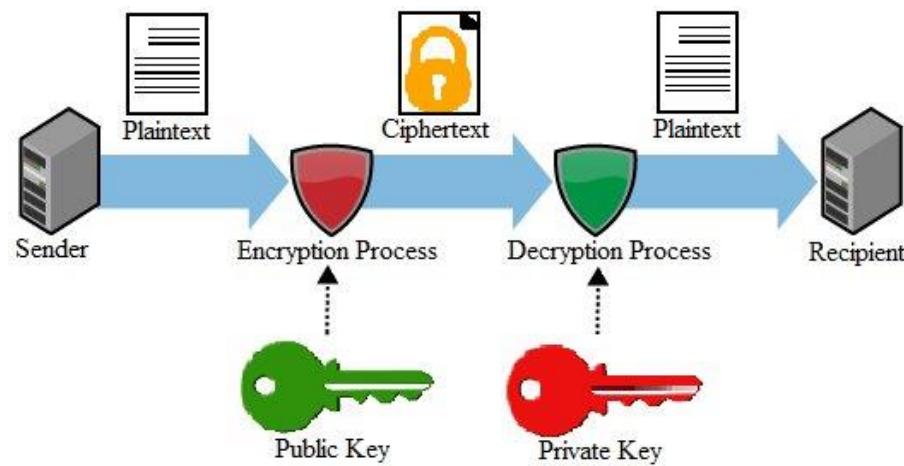


Figure 9-15. Relationship between the plaintext and the ciphertext.

# Types of Encryption Keys



Symmetric Key  
(e.g. DES [ Data Encryption Standard ] )



Asymmetric Key  
(e.g. RSA [ Rivest, Shamir, Adleman ] )

# Secret-Key Cryptography

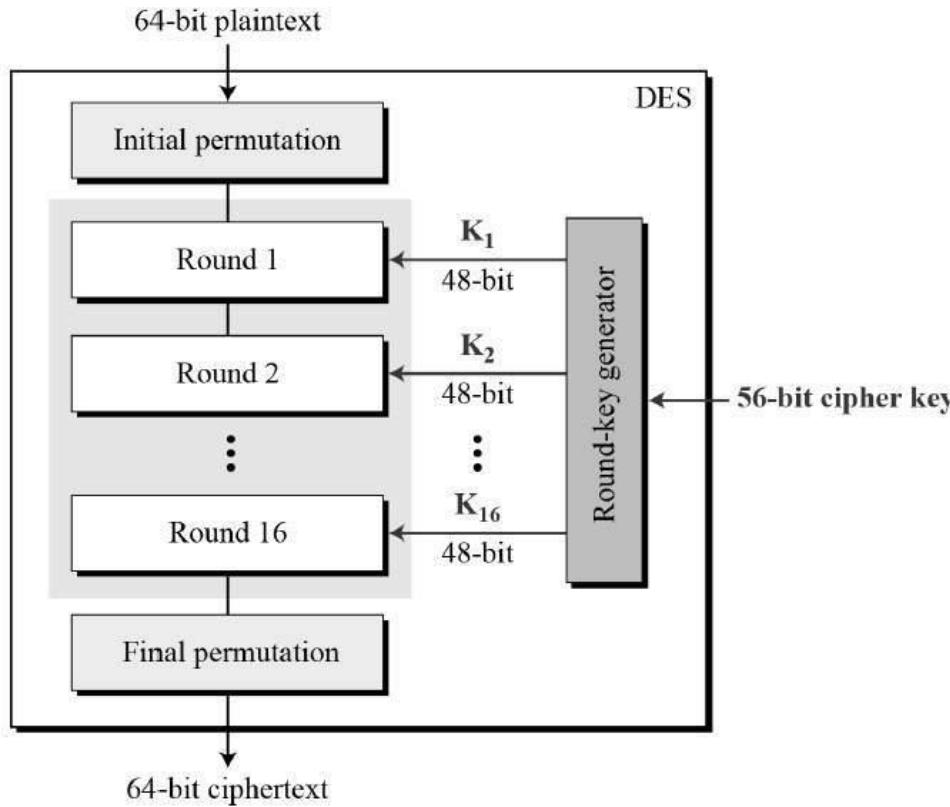
plaintext:      ABCDEFGHIJKLMNOPQRSTUVWXYZ

ciphertext:     QWERTYUIOPASDFGHJKLZXCVBNM

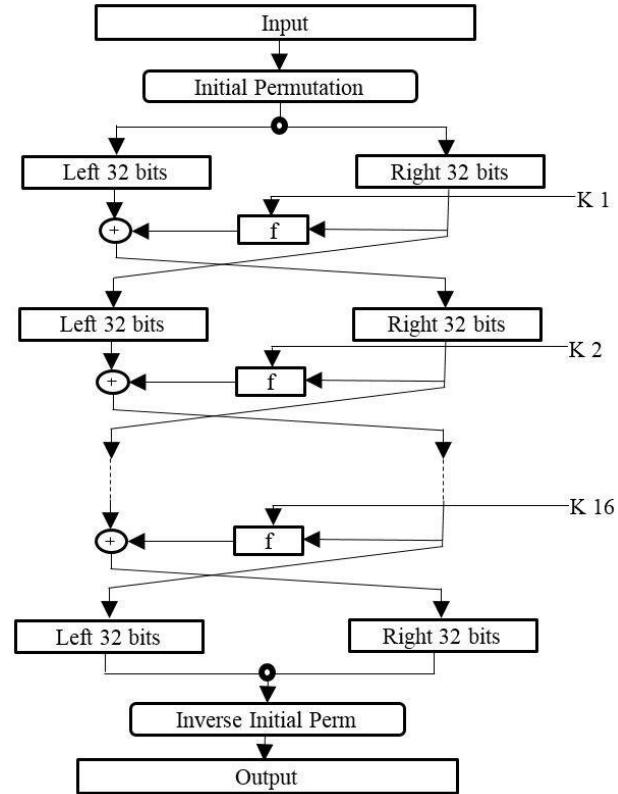
An encryption algorithm in which each letter is replaced by a different letter.

- Above is a symmetric key as, the function is reversible
- This particular function is "trivial"

# DES (Data Encryption Standard)



General idea



Detailed Execution

Decryption is the inverse operation (bottom up)

# Digital Signatures

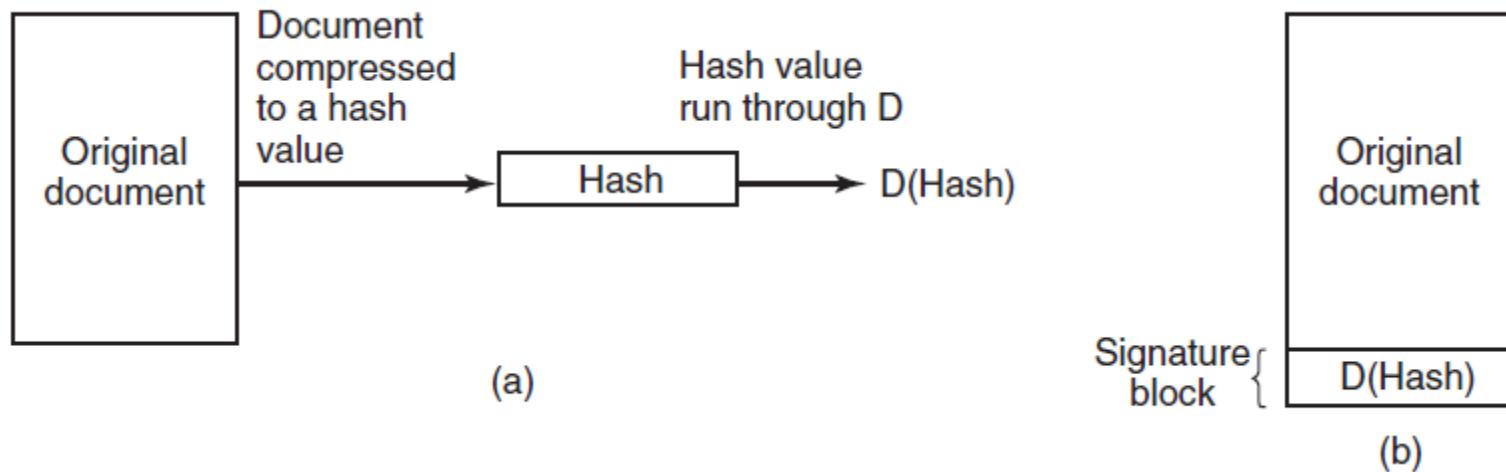


Figure 9-16. (a) Computing a signature block.  
(b) What the receiver gets.

# Authentication (1)

Methods of authenticating users when they attempt to log in based on one of three general principles:

1. Something the user knows.
2. Something the user has.
3. Something the user is.

# Challenge-Response Authentication

Questions should be chosen so that the user does not need to write them down.

Examples:

1. Who is Marjolein's sister?
2. On what street was your elementary school?
3. What did Mrs. Ellis teach?
4. Two factor authentication

# Authentication Using a Physical Object

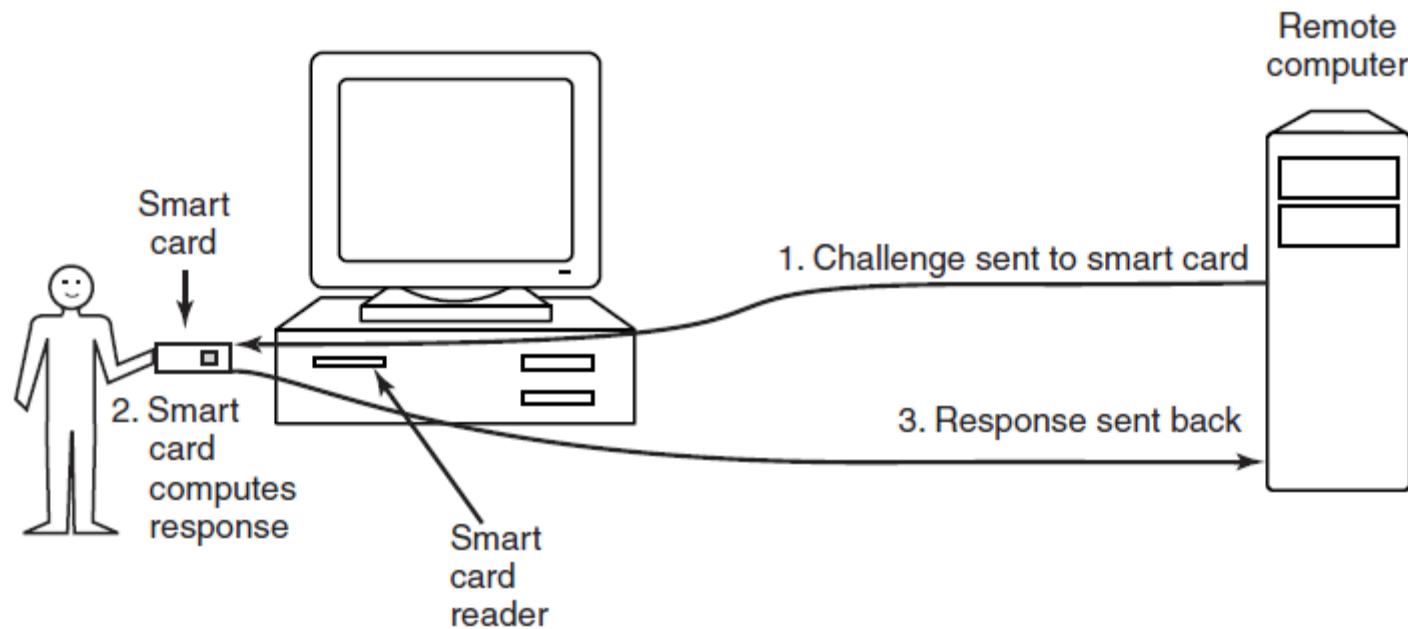


Figure 9-19. Use of a smart card for authentication.

# Authentication Using Biometrics

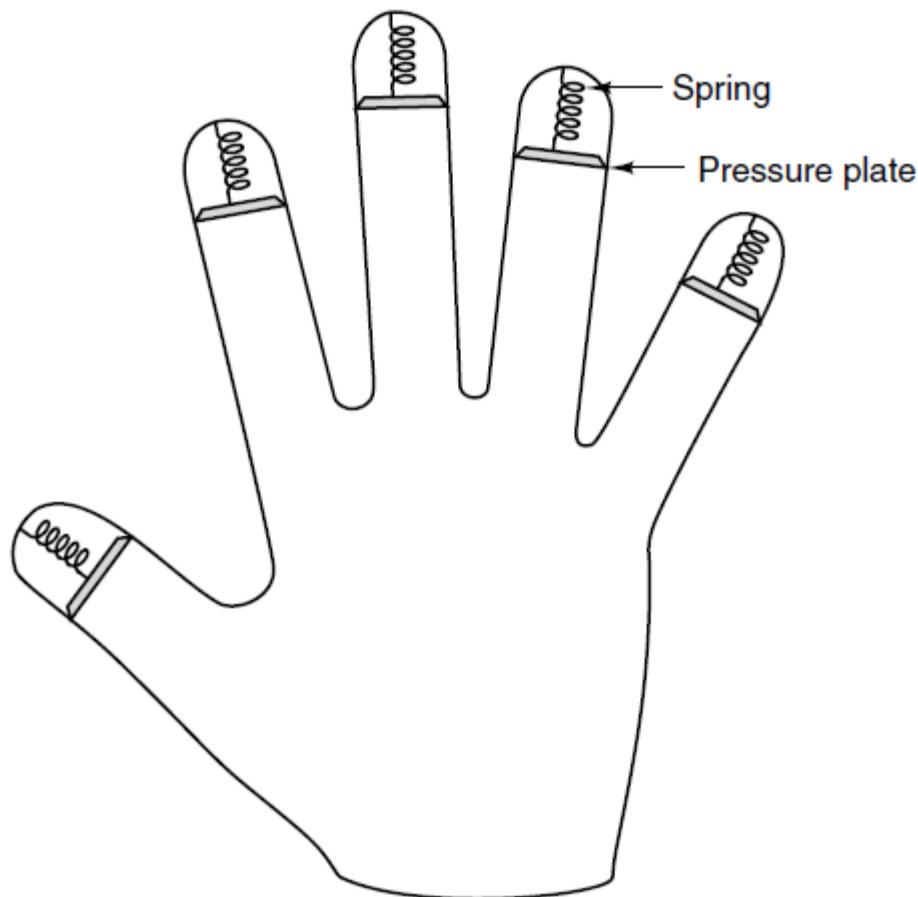


Figure 9-20. A device for measuring finger length.

# Buffer Overflow Attacks

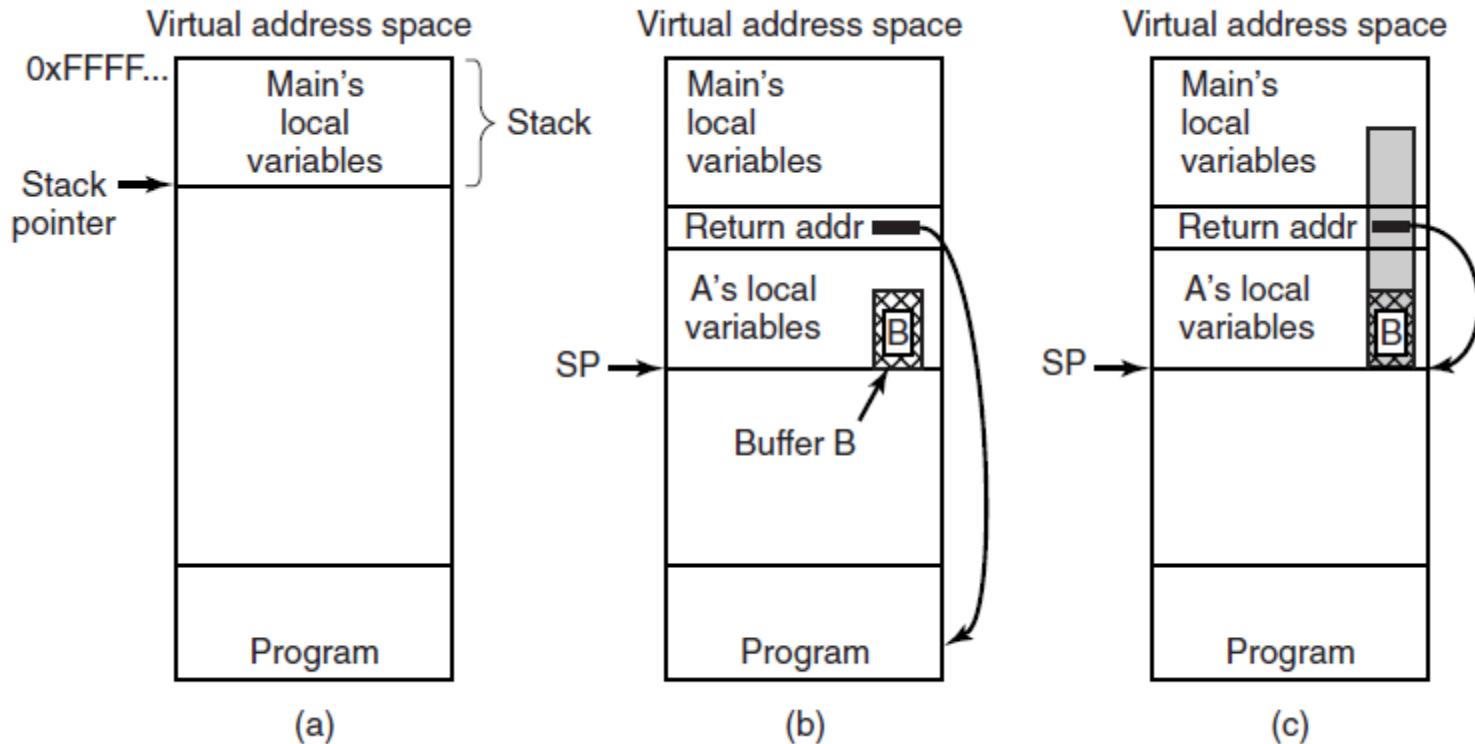


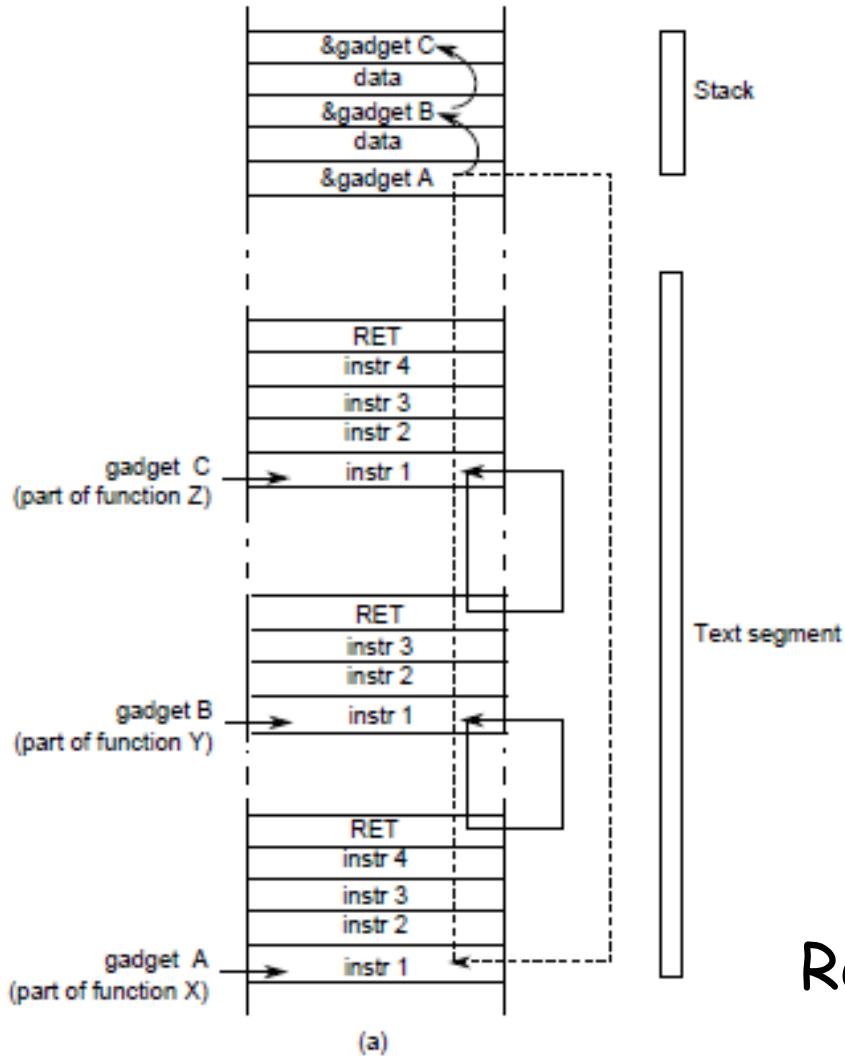
Figure 9-21. (a) Situation when the main program is running. (b) After the procedure A has been called. (c) Buffer overflow shown in gray.

# Avoiding Stack Canaries

```
01. void A (char *date) {  
02.     int len;  
03.     char B [128];  
04.     char logMsg [256];  
05.  
06.     strcpy (logMsg, date); /* first copy the string with the date in the log message */  
07.     len = strlen (date);    /* determine how many characters are in the date string */  
08.     gets (B);             /* now get the actual message */  
09.     strcpy (logMsg+len, B);/* and copy it after the date into logMessage */  
10.     writeLog (logMsg);   /* finally, write the log message to disk */  
11. }
```

Figure 9-22. Skipping the stack canary: by modifying *len* first, the attack is able to bypass the canary and modify the return address directly.

# Code Reuse Attacks



## Example gadgets:

### Gadget A:

- pop operand off the stack into register 1
- if the value is negative, jump to error handler
- otherwise return

### Gadget B:

- pop operand off the stack into register 2
- return

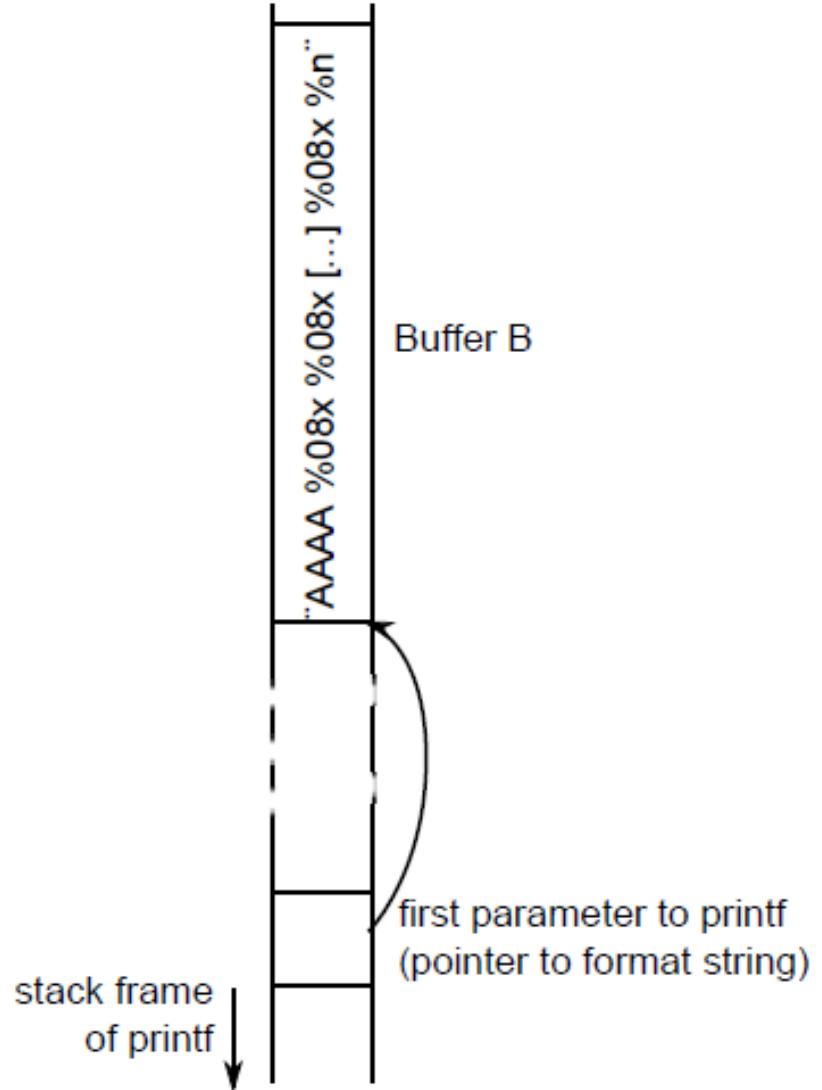
### Gadget C:

- multiply register 1 by 4
- push register 1
- add register 2 to the value on the top of the stack and store the result in register 2

Figure 9-23. ROP  
Return-oriented programming  
linking

# Format String Attacks

Figure 9-24. A format string attack. By using exactly the right number of %08x, the attacker can use the first four characters of the format string as an address.



# Command Injection Attacks

```
int main(int argc, char *argv[])
{
    char src[100], dst[100], cmd[205] = "cp ";
    printf("Please enter name of source file: ");
    gets(src);
    strcat(cmd, src);
    strcat(cmd, " ");
    printf("Please enter name of destination file: ");
    gets(dst);
    strcat(cmd, dst);
    system(cmd);
}
```

/\* declare 3 strings \*/  
/\* ask for source file \*/  
/\* get input from the keyboard \*/  
/\* concatenate src after cp \*/  
/\* add a space to the end of cmd \*/  
/\* ask for output file name \*/  
/\* get input from the keyboard \*/  
/\* complete the commands string \*/  
/\* execute the cp command \*/

Figure 9-25. Code that might lead to a command injection attack.

# Firewalls

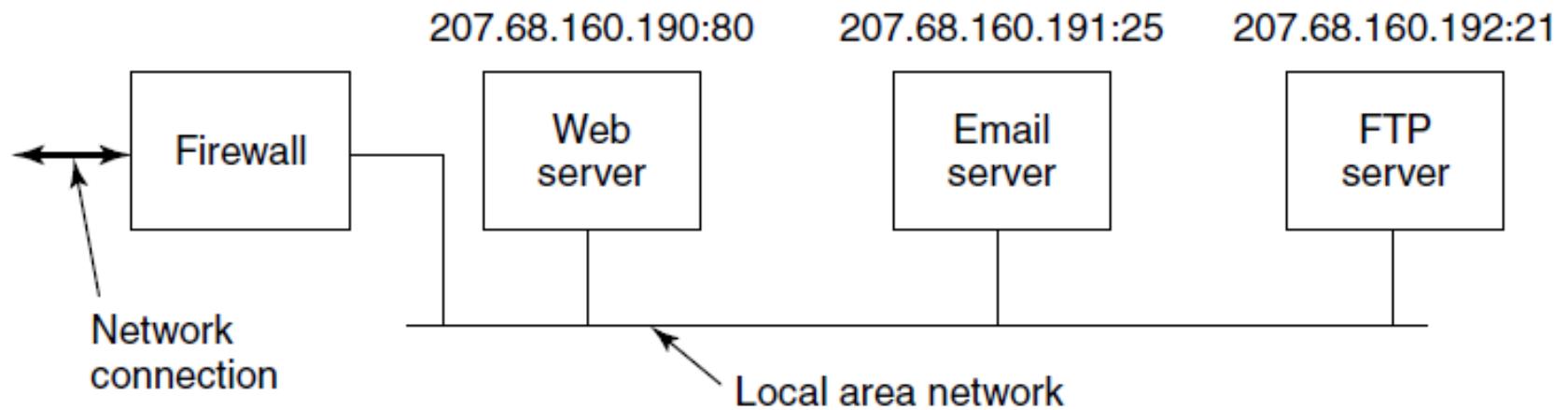


Figure 9-32. A simplified view of a hardware firewall protecting a LAN with three computers

# Code Signing

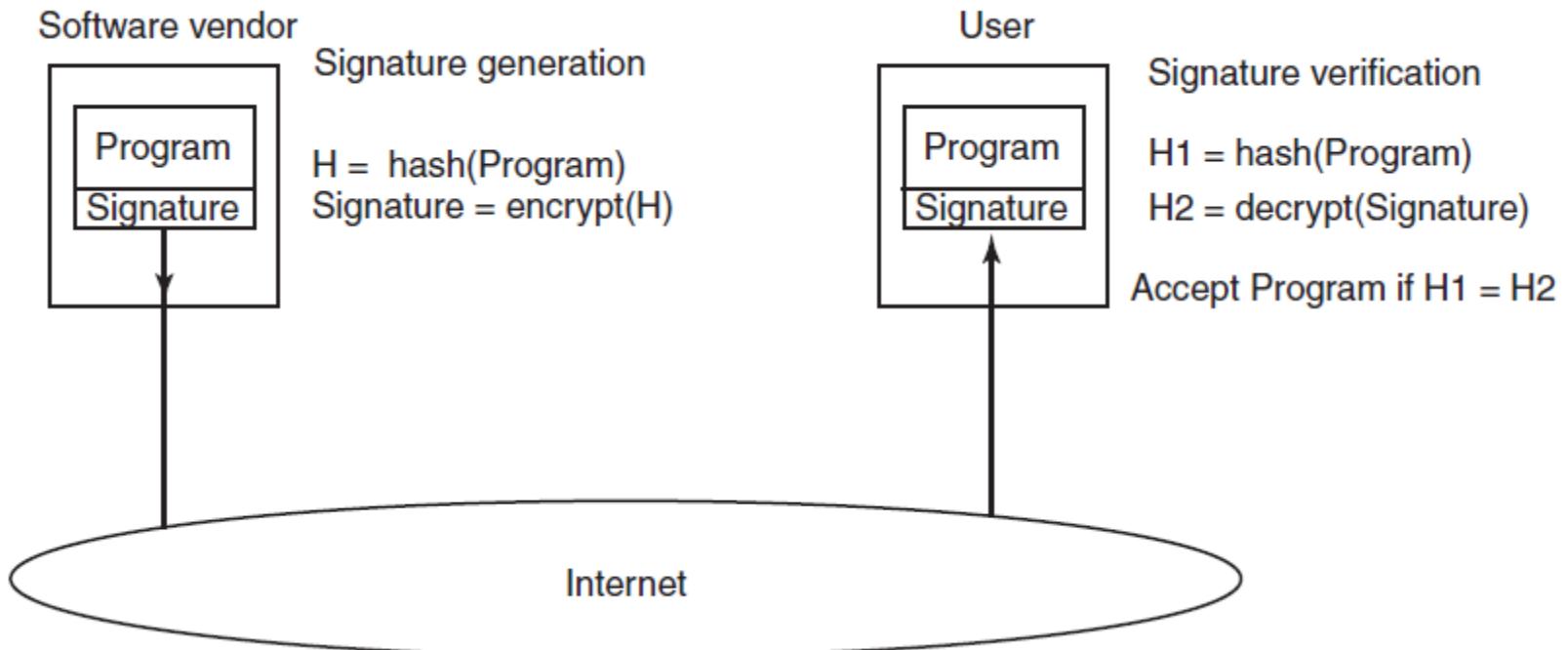


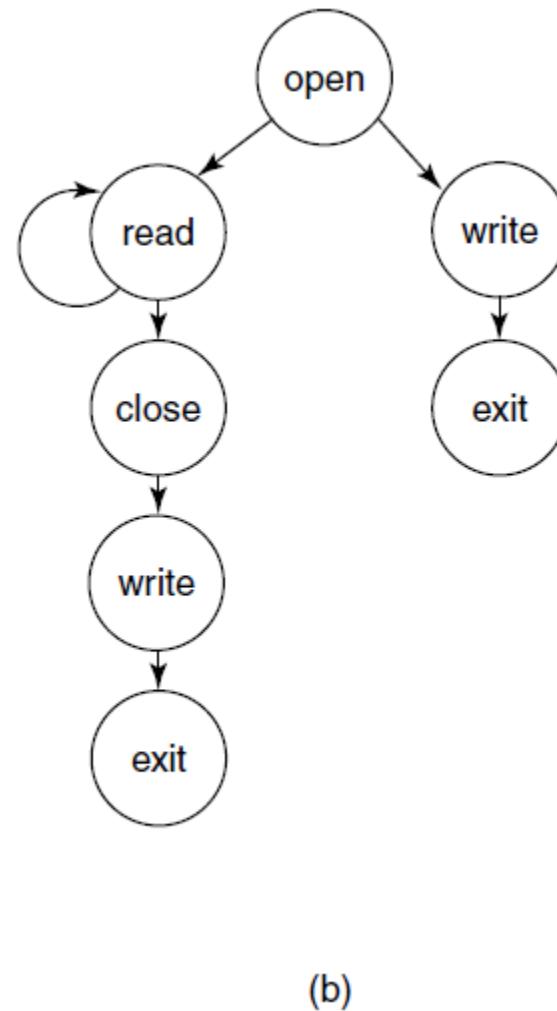
Figure 9-35. How code signing works.

# Model-Based Intrusion Detection

```
int main(int argc *char argv[])
{
    int fd, n = 0;
    char buf[1];

    fd = open("data", 0);
    if (fd < 0) {
        printf("Bad data file\n");
        exit(1);
    } else {
        while (1) {
            read(fd, buf, 1);
            if (buf[0] == 0) {
                close(fd);
                printf("n = %d\n", n);
                exit(0);
            }
            n = n + 1;
        }
    }
}
```

(a)



(b)

Figure 9-37. (a) A program. (b) System call graph for (a).

# Sandboxing

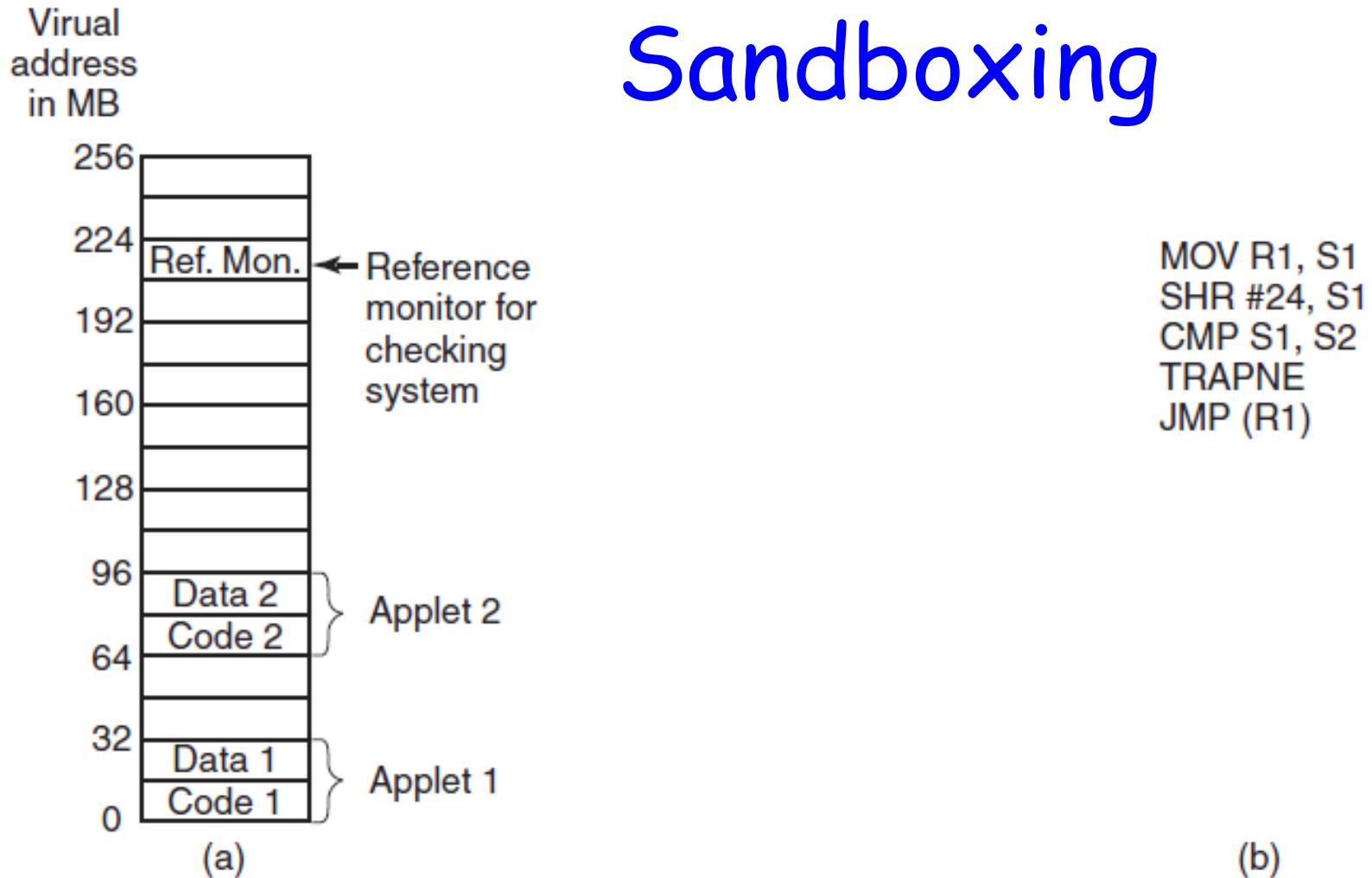


Figure 9-38.

- (a) Memory divided into 16-MB sandboxes.
- (b) One way of checking an instruction for validity.



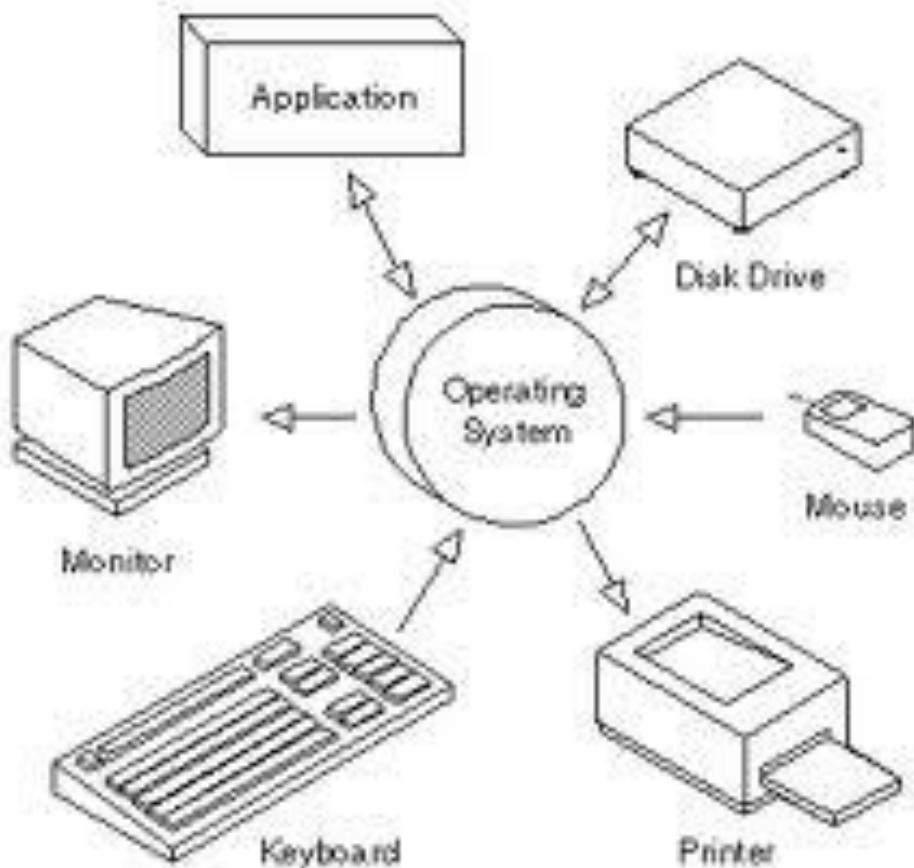
CSCI-GA.2250-001

# Operating Systems

Input / Output → I/O

Hubertus Franke  
frankeh@cs.nyu.edu





# Categories of I/O Devices

External devices that engage in I/O with computer systems can be grouped into three categories:

## **Human readable**

- suitable for communicating with the computer user
- printers, terminals, video display, keyboard, mouse

## **Machine readable**

- suitable for communicating with electronic equipment
- disk drives, USB keys, sensors, controllers

## **Communication**

- suitable for communicating with remote devices
- modems, digital line drivers

# A Simple Definition

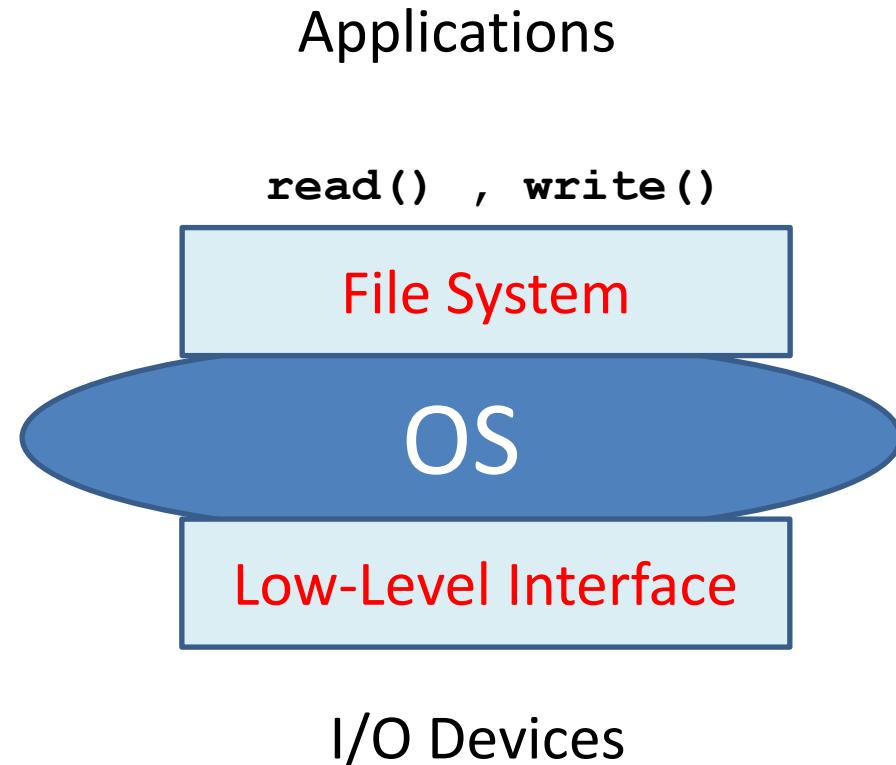
- The main concept of I/O is to move data from/to I/O devices to the processor using some modules (code) and buffers.
- This is the way the processor deals with the outside world.
- Examples: mouse, display, keyboard, disks, network, scanners, speakers, accelerators, GPUs, etc.

# The OS and I/O

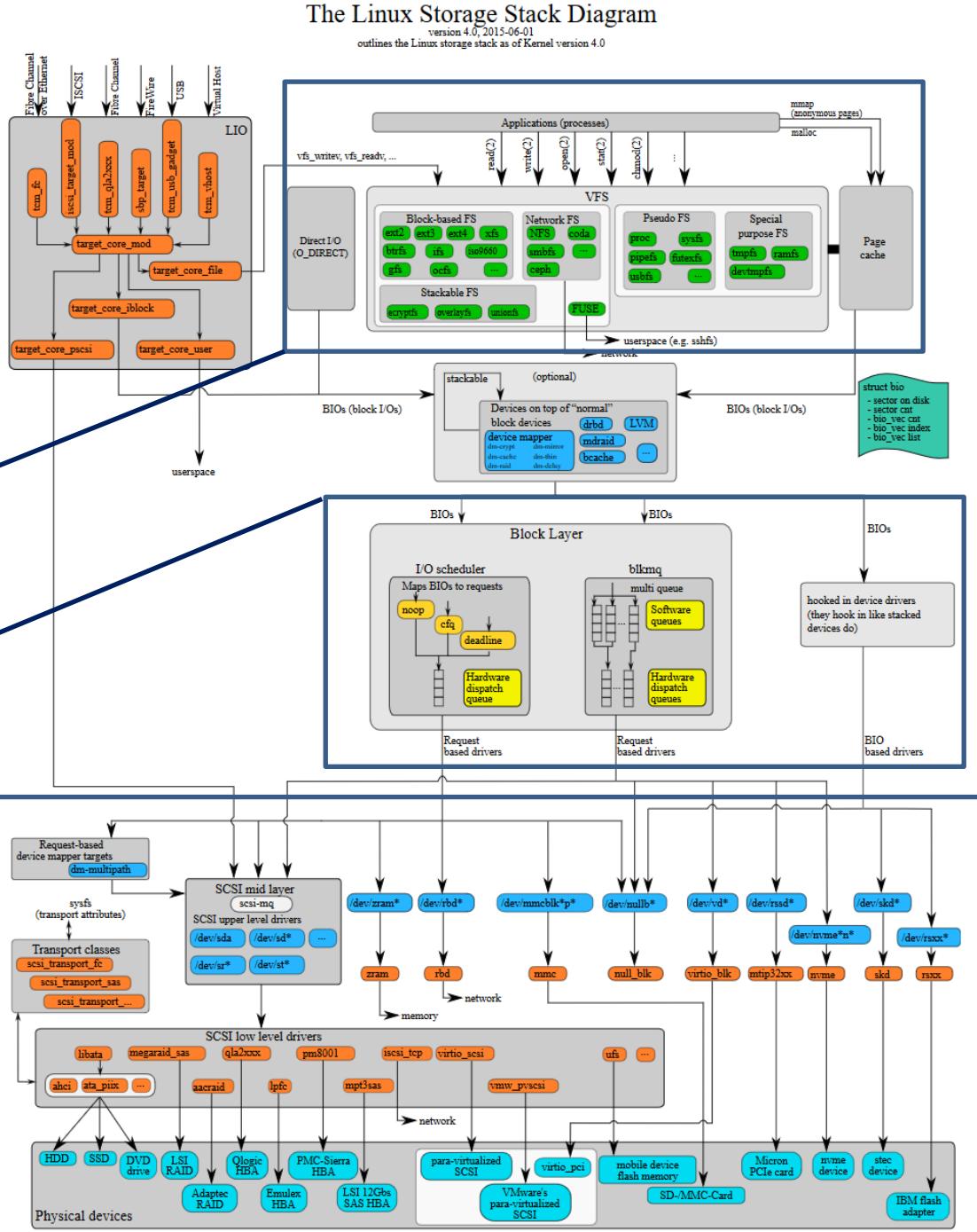
- The OS controls all I/O devices
  - Issue commands to devices
  - Catch interrupts
  - Handle errors
- Provides an interface between the devices and the rest of the system
- A few exceptions are processor built-in accelerators ( encryption, compression engines) that often can be invoked by applications.
  - But think of these less as I/O but as a different kind of functional units in the CPU

# Simplified View

- Applications rarely communicate directly with Devices
- Would be too hardware specific
- OS responsible of translating filesystem requests (read/write) into low level device specific I/O requests

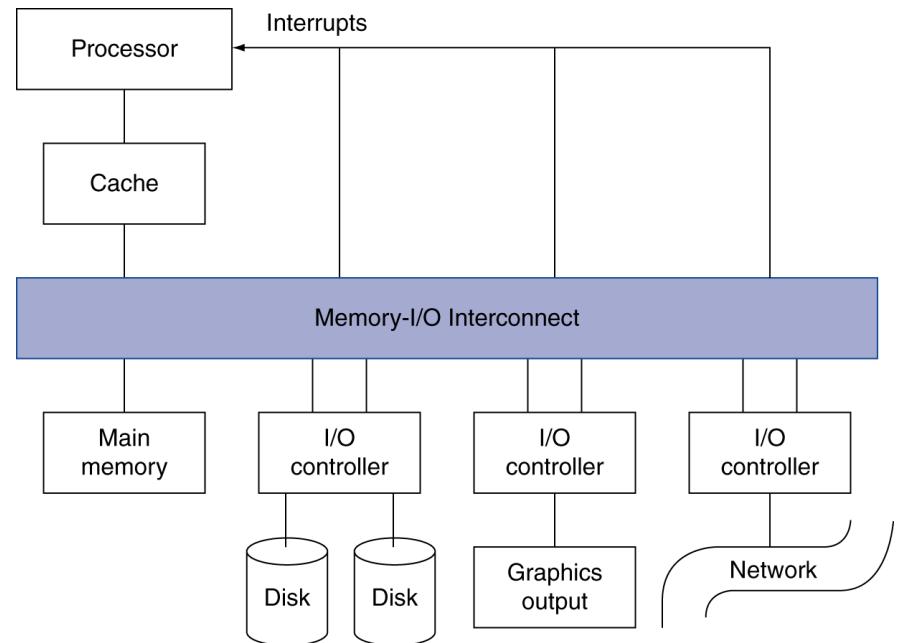


# Full Linux I/O Stack



# I/O Devices: Challenges

- Very diverse devices
  - behavior (i.e., input vs. output vs. storage)
  - partner (who is at the other end?)
  - data rate
- I/O Design affected by many factors (expandability, resilience)
- Performance:
  - access latency
  - throughput
  - connection between devices and the system
  - the memory hierarchy
  - the operating system
- A variety of different users

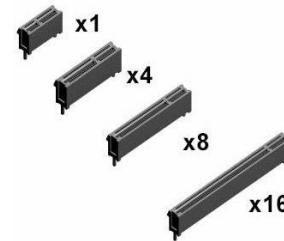


# I/O Devices

- **Block device**
  - Stores information in fixed-size blocks
  - Each block has its own address
  - Transfers in one or more blocks
  - Example: Hard-disks, CD-ROMs, USB sticks
- **Character device**
  - Delivers or accepts stream of character
  - Is not addressable
  - Example: mice, printers, network interfaces

# Devices ( Feed and Speed) !!!

Device	Data rate
Keyboard	10 bytes/sec
Mouse	100 bytes/sec
56K modem	7 KB/sec
Scanner	400 KB/sec
Digital camcorder	3.5 MB/sec
802.11g Wireless	6.75 MB/sec
52x CD-ROM	7.8 MB/sec
Fast Ethernet	12.5 MB/sec
Compact flash card	40 MB/sec
FireWire (IEEE 1394)	50 MB/sec
USB 2.0	60 MB/sec
SONET OC-12 network	78 MB/sec
SCSI Ultra 2 disk	80 MB/sec
Gigabit Ethernet	125 MB/sec
SATA disk drive	300 MB/sec
Ultrium tape	320 MB/sec
PCI bus	528 MB/sec



peripheral component interconnect express

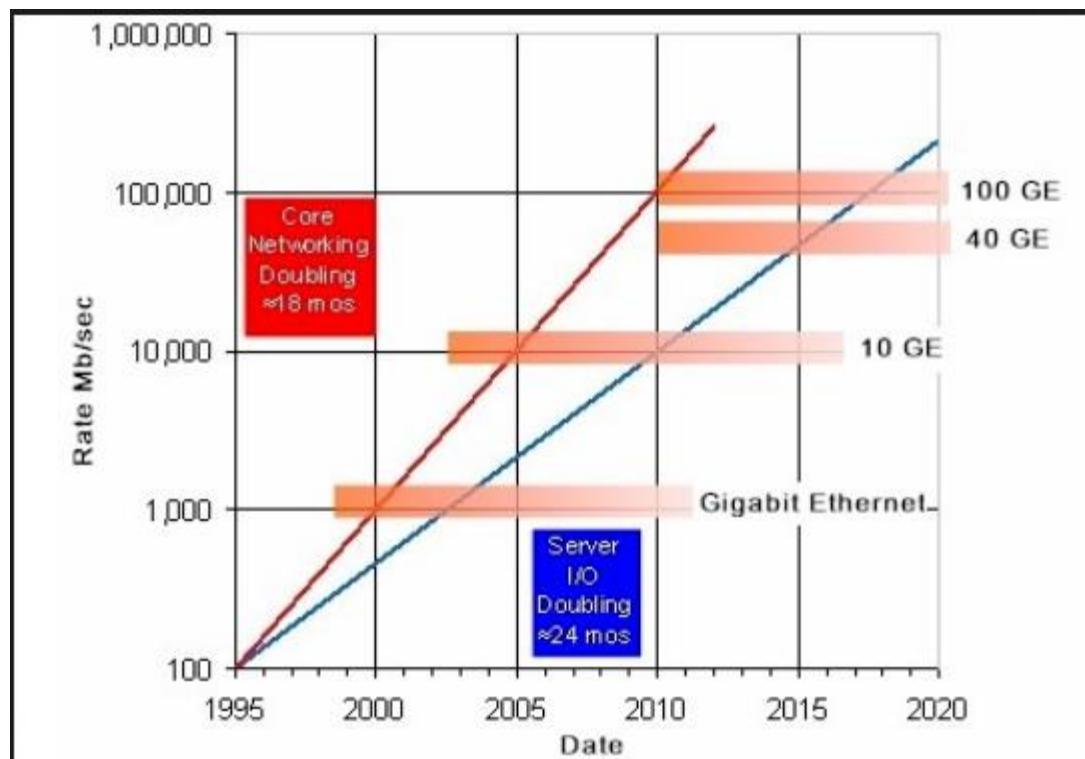
## PCIe Bandwidth & Frequency



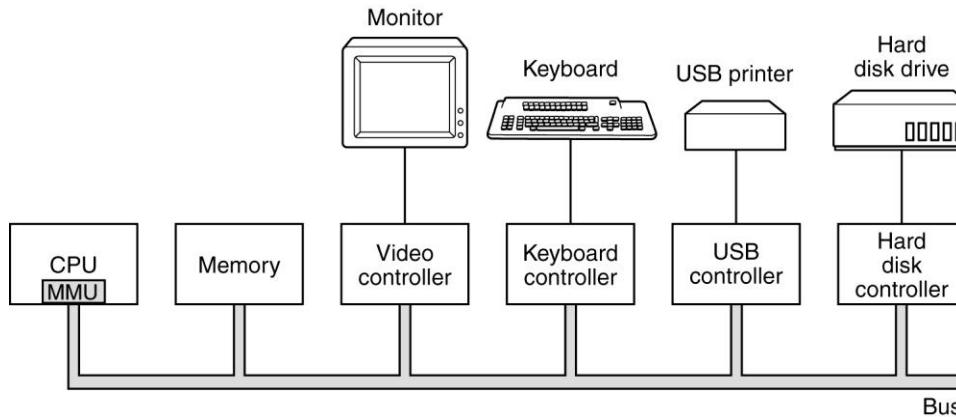
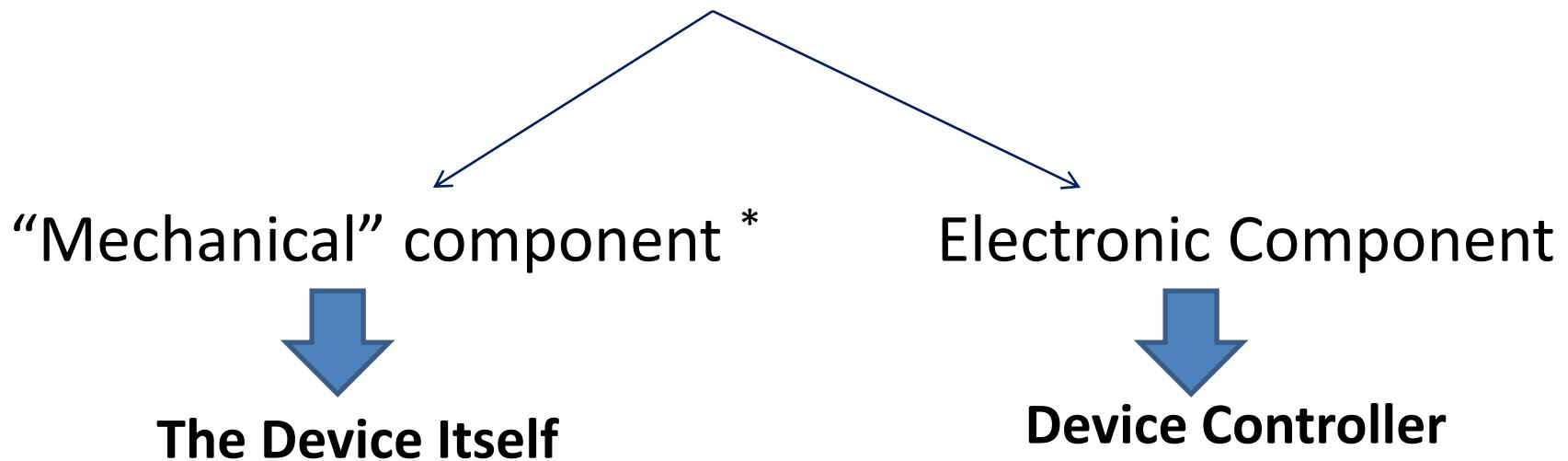
Year	Bandwidth	Frequency/Speed
1992	133MB/s (32 bit simplex)	33 Mhz (PCI)
1993	533MB/s (64 bit simplex)	66 Mhz (PCI 2.0)
1999	1.06GB/s (64 bit simplex)	133 Mhz (PCI-X)
2002	2.13GB/s (64 bit simplex)	266 Mhz (PCI-X 2.0)
2002	8GB/s (x16 duplex)	2.5 GHz (PCIe 1.x)
2006	16GB/s (x16 duplex)	5.0 GHz (PCIe 2.x)
2010	32GB/s (x16 duplex)	8.0 GHz (PCIe 3.x)
2017	64GB/s (x16 duplex)	16.0 GHz (PCIe 4.0)
2019	128GB/s (x16 duplex)	32.0 GHz (PCIe 5.0)

# Devices change !!!

- Network speed doubling every 18-24 month
- Approaching 400Gbps ( we already have 200)

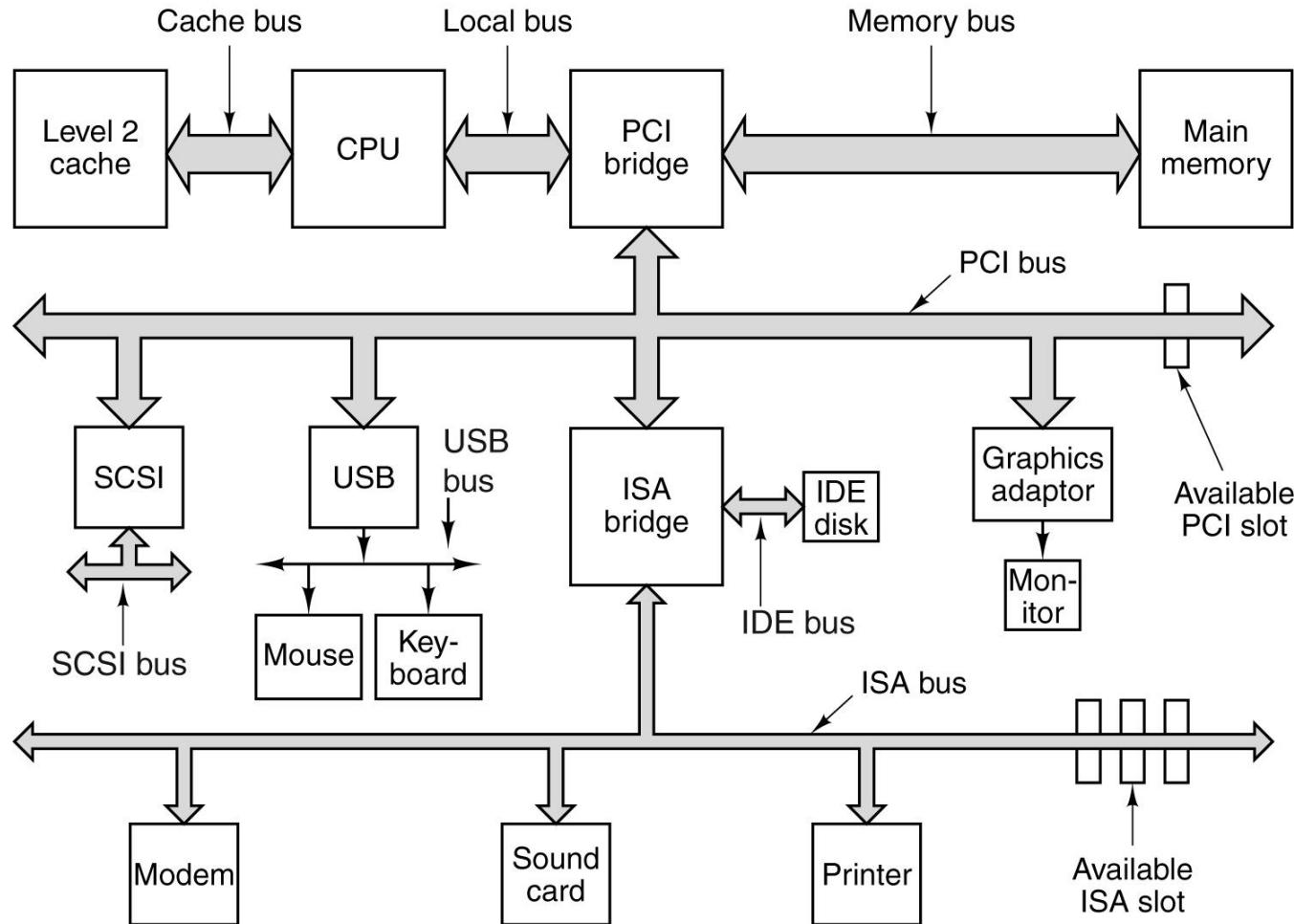


# I/O Units



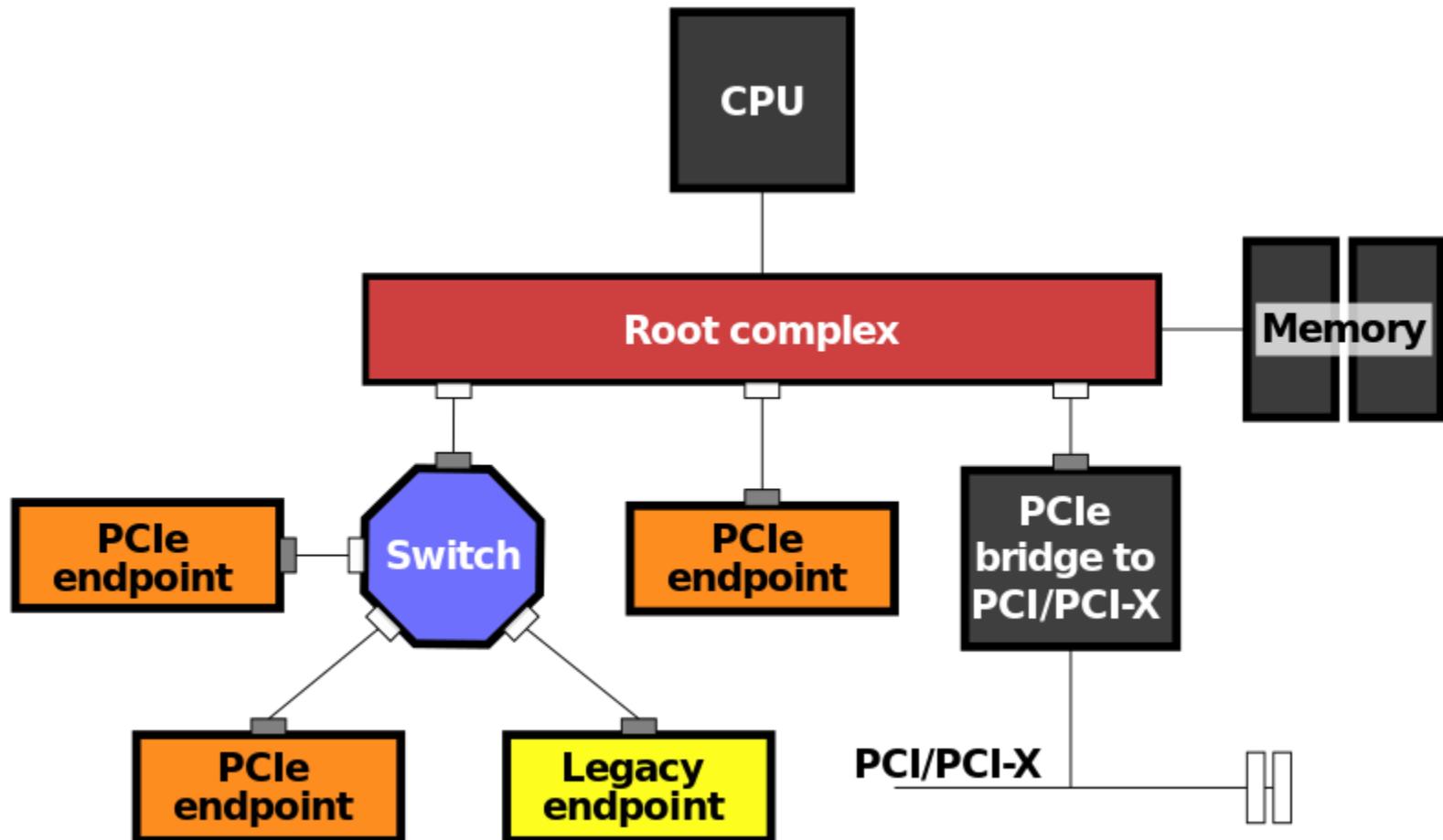
\* Many devices don't have mechanical components anymore

# A more realistic hierarchy of I/O



# PCI-e

## Peripheral Component Interconnect Express



Standard means to connect modern devices to the system

# Controller and Device

- Each controller has few registers used to communicate with CPU
- By writing/reading into/from those registers, the OS can control the device.
- (1) There are also data buffers in the device that can be read from or written to by the OS
  - Location/address of buffer
  - Location/address of I/O “object”
  - Size of data to transfer
- (2) cmd buffers to start/stop operations

# How does CPU communicate with control registers and data buffers?

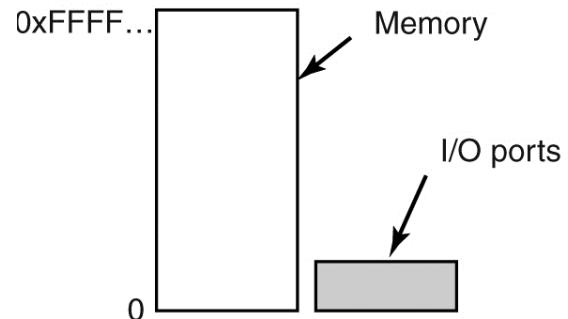
Two main approaches:

- I/O port space
- Memory-mapped I/O

# I/O Port Space

- Each control register is assigned an **I/O port number**
- The set of all I/O ports form the I/O port space
- I/O port space is protected → only kernel can access
  - execute privileged instructions:
    - `in portnum, target`
    - `out portnum, source`

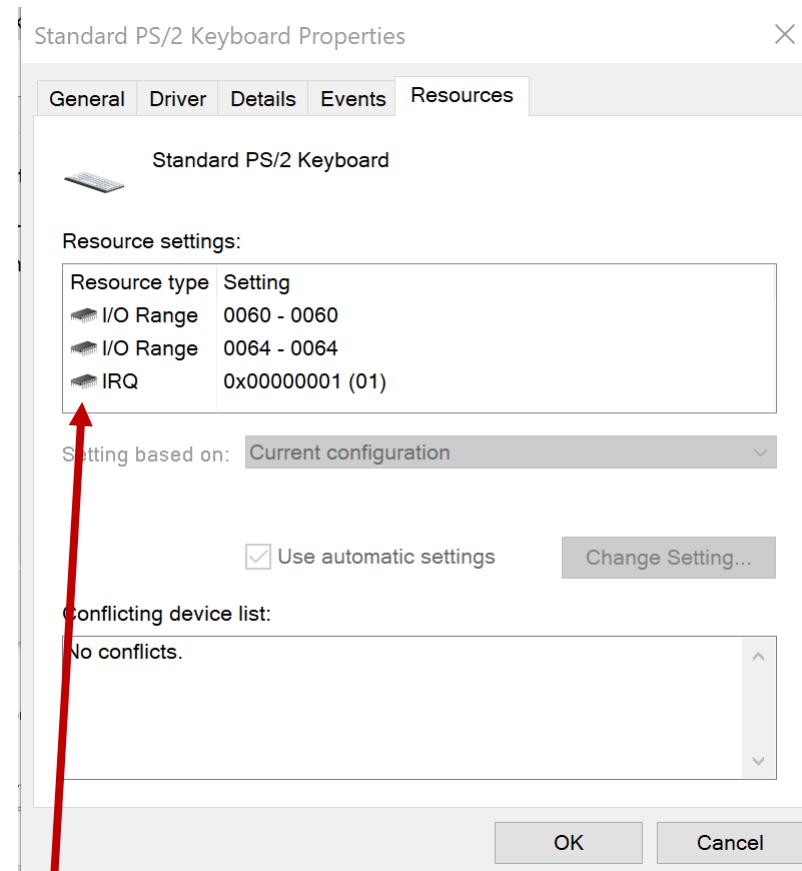
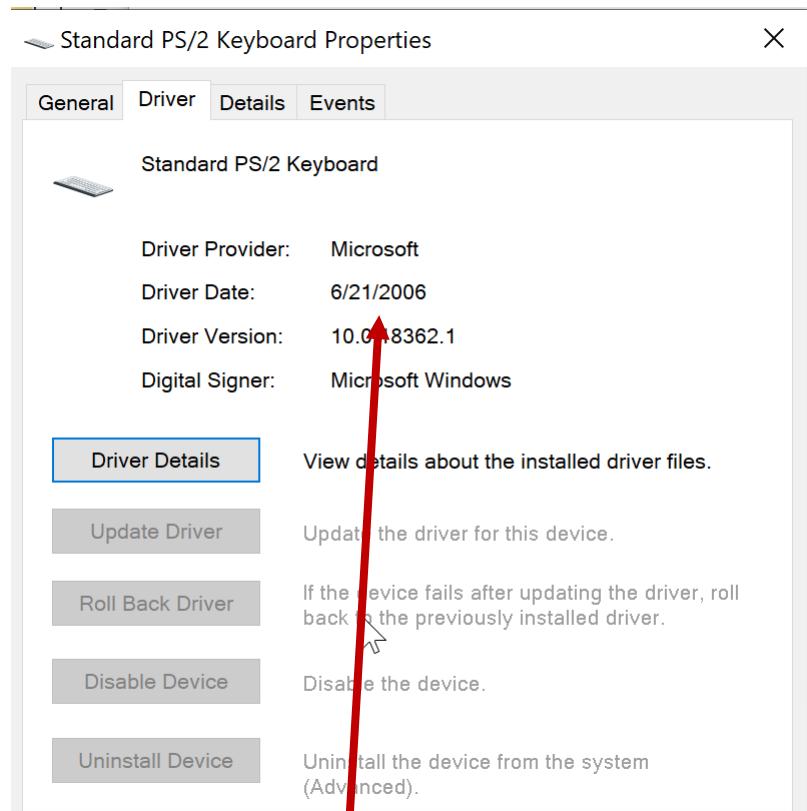
Two address



Opcode	Mnemonic	Description
E4 ib	IN AL,imm8	Input byte from imm8 I/O port address into AL.
E5 ib	IN AX,imm8	Input byte from imm8 I/O port address into AX.
E5 ib	IN EAX,imm8	Input byte from imm8 I/O port address into EAX.
EC	IN AL,DX	Input byte from I/O port in DX into AL.
ED	IN AX,DX	Input word from I/O port in DX into AX.
ED	IN EAX,DX	Input doubleword from I/O port in DX into EAX.

- Classical x86 way
- Largely done only for "legacy" devices

# Example Win Keyboard



Talk about legacy software

two io-ports and one interrupt

# Example: Serial and Parallel Ports

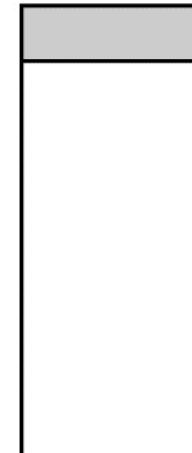
<b>COM Port #</b>	<b>IRQ</b>	<b>I/O Port Address</b>
1	4	3F8-3FFh
2	3	2F8-2FFh
3	4	3E8-3EFh
4	3	2E8-2EFh

<b>LPT Port #</b>	<b>IRQ</b>	<b>I/O Port Address Range</b>
LPT1	7	378-37Fh or 3BC-38Fh
LPT2	5	278-27Fh or 378-37Fh
LPT3	5	278-27Fh

# Memory-Mapped I/O

- Map control registers into the memory space
- Each control register is assigned a unique memory address
- Load/stores to the memory are “snooped” on by device and acted upon, PCI device is told the range to snoop
- Every modern architecture basically does it this way

One address space



# Advantages of Memory-Mapped I/O

- Device drivers can be written entirely in C (since no special instructions are needed)
- No special protection is needed from OS, just refrain from putting that portion of the address space in any user's virtual address space
- Every instruction that can reference memory can also reference control registers

# PCI-Configuration Space

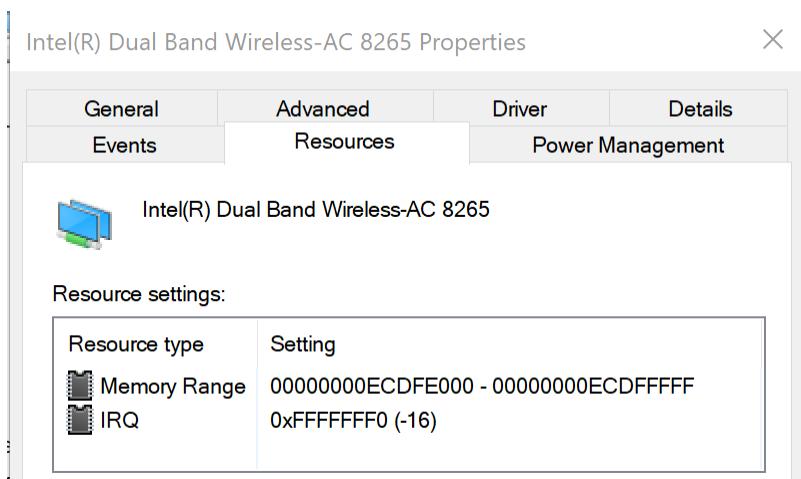
Table 2-22: Type 0 PCI Configuration Space Header

31	16	15	0	
Device ID		Vendor ID		
Status		Command		
Class Code		Rev ID		00h
BIST	Header	Lat Timer	Cache Ln	04h
Base Address Register 0				08h
Base Address Register 1				0Ch
Base Address Register 2				10h
Base Address Register 3				14h
Base Address Register 4				18h
Base Address Register 5				1Ch
Cardbus CIS Pointer				20h
Subsystem ID	Subsystem Vendor ID			24h
Expansion ROM Base Address				28h
Reserved		CapPtr		2Ch
Reserved				30h
Max Lat	Min Gnt	Intr Pin	Intr Line	34h
				38h
				3Ch

```
struct pci_config {
    uint16 devid;
    uint16 vendorid;
    :
    uint32 bar0;
    :
    uint32 bar1;
    :
    uint8 maxlat;
    :
    uint8 intr;
};
```

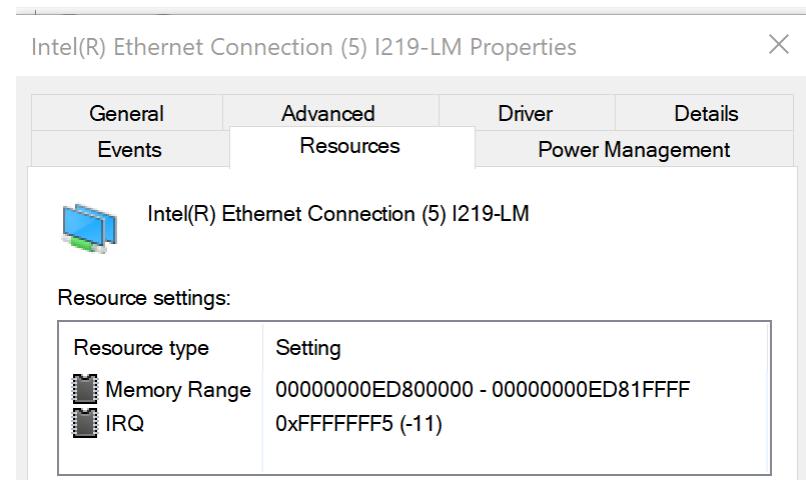
Standard PCI Control Space

# Examples for network devices



Range: 0x2000 == 8KB

```
struct wlac_8265 {  
    struct pci_config pcie_config;  
    <specific structure for 8265 dev>  
};  
  
struct wlac_8265 *wldev =  
    (struct wlac_8265*) kern_addr(0xECDFE000);  
wldev->memberelem = .....
```



Range: 0x2000 == 128KB

```
struct lan_I219 {  
    struct pci_config pcie_config;  
    <specific structure for I219ev>  
};  
  
virtual kernel address  
    ↓  
struct lan_I219 *landev =  
    (struct lan_I219*) kern_addr(0xED800000);  
landev->memberelem = .....
```

physical bus address

# Inspecting your PCI subsystem

What devices are connected to my system ?

```
[frankeh@linserver1 ~]$ lspci | grep -v "^\^00"
01:00.0 Ethernet controller: Broadcom Inc. and subsidiaries NetXtreme II BCM5709 Gigabit Ethernet (rev 20)
01:00.1 Ethernet controller: Broadcom Inc. and subsidiaries NetXtreme II BCM5709 Gigabit Ethernet (rev 20)
02:00.0 Ethernet controller: Broadcom Inc. and subsidiaries NetXtreme II BCM5709 Gigabit Ethernet (rev 20)
02:00.1 Ethernet controller: Broadcom Inc. and subsidiaries NetXtreme II BCM5709 Gigabit Ethernet (rev 20)
03:00.0 PCI bridge: PLX Technology, Inc. PEX 8624 24-lane, 6-Port PCI Express Gen 2 (5.0 GT/s) Switch [ExpressLane] (rev bb)
04:00.0 PCI bridge: PLX Technology, Inc. PEX 8624 24-lane, 6-Port PCI Express Gen 2 (5.0 GT/s) Switch [ExpressLane] (rev bb)
04:01.0 PCI bridge: PLX Technology, Inc. PEX 8624 24-lane, 6-Port PCI Express Gen 2 (5.0 GT/s) Switch [ExpressLane] (rev bb)
04:04.0 PCI bridge: PLX Technology, Inc. PEX 8624 24-lane, 6-Port PCI Express Gen 2 (5.0 GT/s) Switch [ExpressLane] (rev bb)
04:05.0 PCI bridge: PLX Technology, Inc. PEX 8624 24-lane, 6-Port PCI Express Gen 2 (5.0 GT/s) Switch [ExpressLane] (rev bb)
05:00.0 RAID bus controller: Broadcom / LSI MegaRAID SAS 2108 [Liberator] (rev 05)
0a:03.0 VGA compatible controller: Matrox Electronics Systems Ltd. MGA G200eW WPCM450 (rev 0a)
20:00.0 Host bridge: Advanced Micro Devices, Inc. [AMD/ATI] RD890 Northbridge only dual slot (2x8) PCI-e GFX Hydra part (rev 02)
20:00.2 IOMMU: Advanced Micro Devices, Inc. [AMD/ATI] RD890S/RD990 I/O Memory Management Unit (IOMMU)
20:02.0 PCI bridge: Advanced Micro Devices, Inc. [AMD/ATI] RD890/RD9x0/RX980 PCI to PCI bridge (PCI Express GFX port 0)
20:03.0 PCI bridge: Advanced Micro Devices, Inc. [AMD/ATI] RD890/RD9x0 PCI to PCI bridge (PCI Express GFX port 1)
20:0b.0 PCI bridge: Advanced Micro Devices, Inc. [AMD/ATI] RD890/RD990 PCI to PCI bridge (PCI Express GFX2 port 0)
```

```
[frankeh@linserver1 ~]$ lspci -n | grep -v "^\^00"
01:00.0 0200: 14e4:1639 (rev 20)
01:00.1 0200: 14e4:1639 (rev 20)
02:00.0 0200: 14e4:1639 (rev 20)
02:00.1 0200: 14e4:1639 (rev 20)
03:00.0 0604: 10b5:8624 (rev bb)
04:00.0 0604: 10b5:8624 (rev bb)
04:01.0 0604: 10b5:8624 (rev bb)
04:04.0 0604: 10b5:8624 (rev bb)
04:05.0 0604: 10b5:8624 (rev bb)
05:00.0 0104: 1000:0079 (rev 05)
0a:03.0 0300: 102b:0532 (rev 0a)
20:00.0 0600: 1002:5a12 (rev 02)
20:00.2 0806: 1002:5a23
20:02.0 0604: 1002:5a16
20:03.0 0604: 1002:5a17
20:0b.0 0604: 1002:5a1f
```

[Add item](#)  
[Discuss](#)  
[Help](#)  
[ID syntax](#)

## The PCI ID Repository

The home of the `pci.ids` file

[Log in](#)

[Main](#) > [PCI Devices](#) > **Vendor 14e4**

**Name:** Broadcom Inc. and subsidiaries

### Discussion

**Name:** Broadcom Corporation

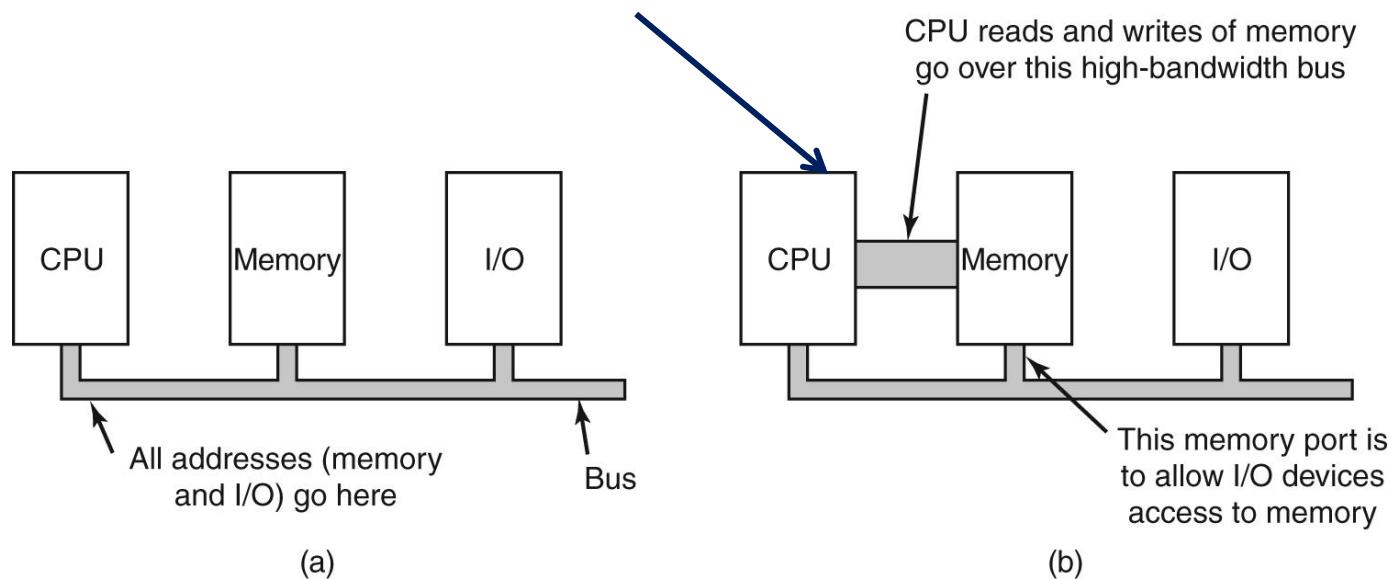
*mj*

2002-08-14 18:16:25

Executed on “linserver1.cims.nyu.edu”

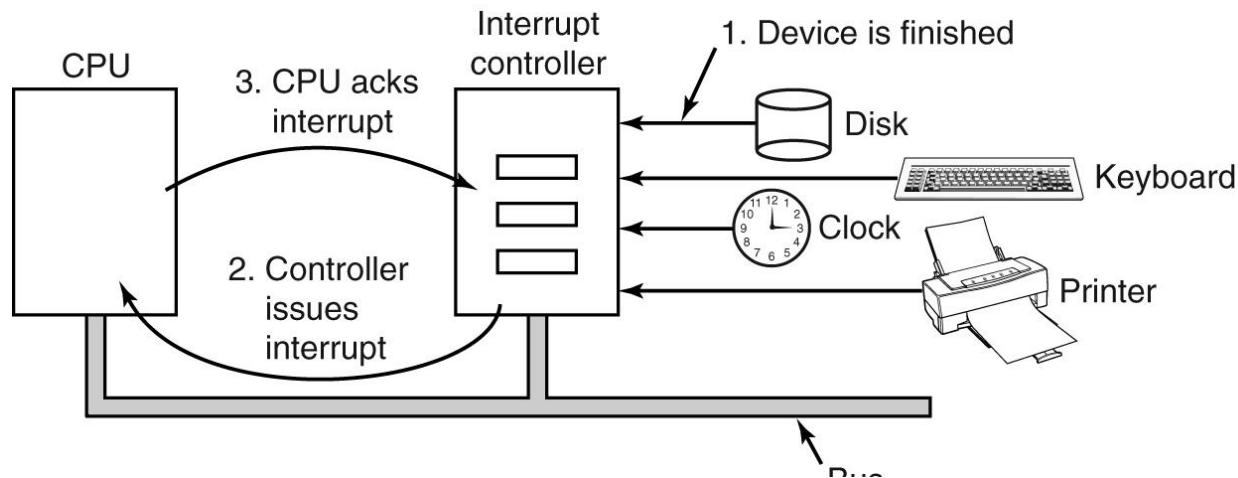
# Something to be conscious about with Memory-Mapped I/O

- Caching a device control register can be disastrous, after all it is just a “memory load”
- Remember the “cache disabled bit” in PTE ?

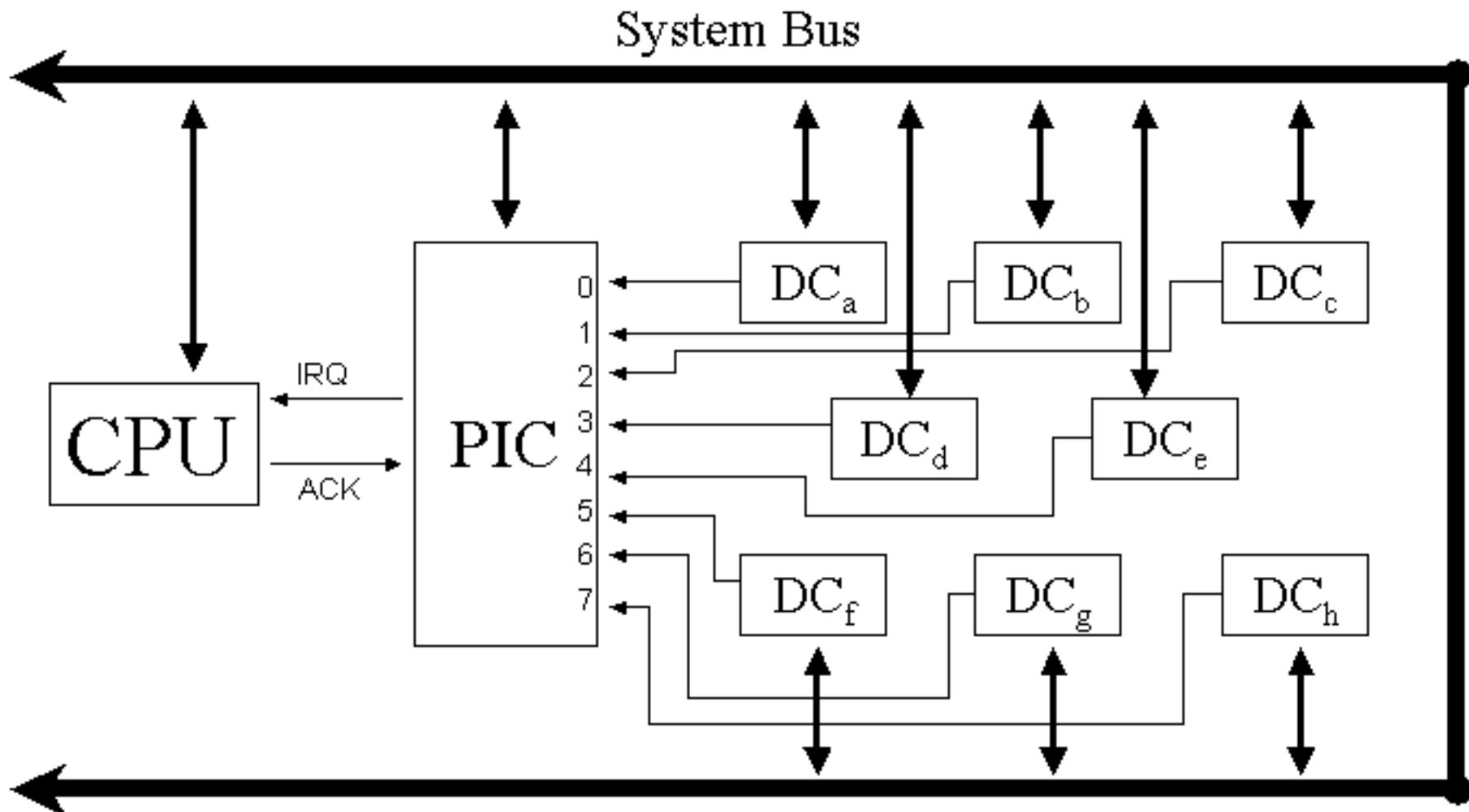


# Interrupts

- When an I/O device has finished the work given to it, it causes an interrupt.
- More generally, anytime a device wants attention it causes an interrupt.



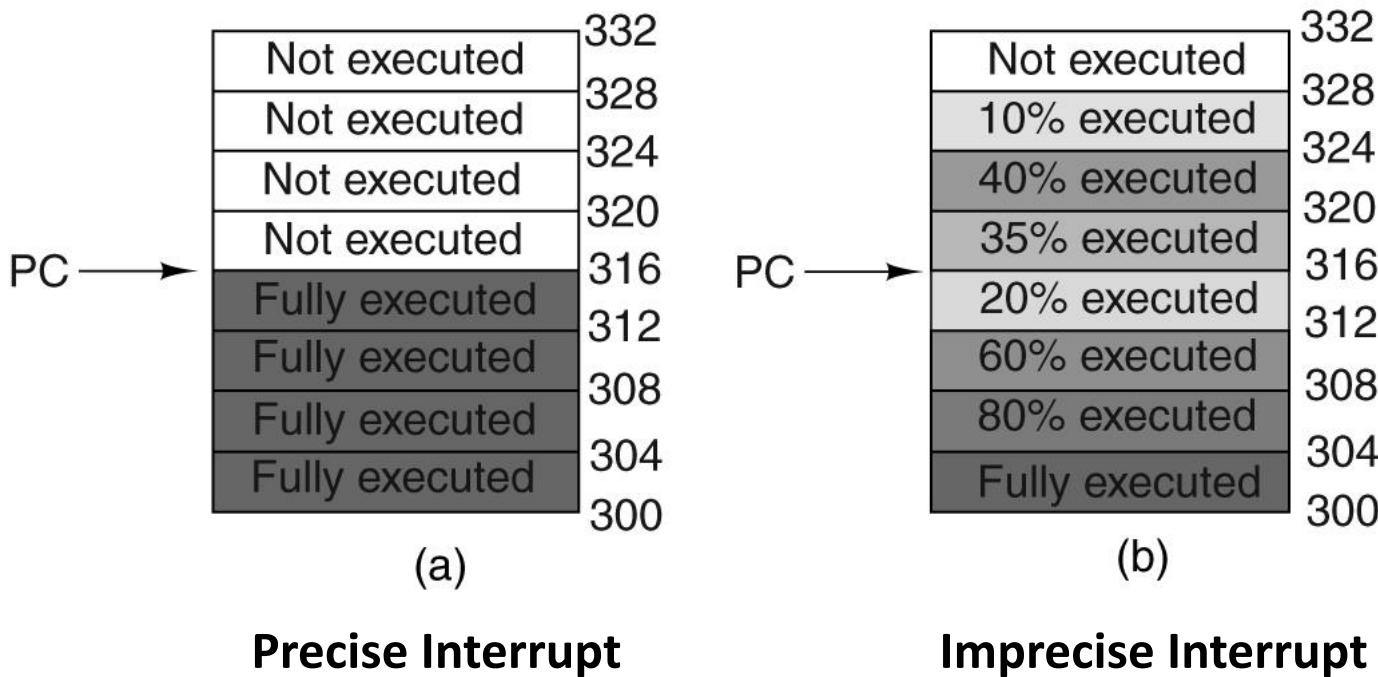
# (Advanced) Programmable Interrupt Controllers [ APIC ]



DC == Device Controller

# Precise Interrupts

- Makes handling interrupts much simpler
- Has 4 properties
  - The program counter (PC) is saved in known place (typically a special register only accessible through kernel mode)
  - All instructions before the one pointed by PC have fully executed
  - No instruction beyond the one pointed by PC has been executed (or has any sideeffects)
  - The execution state of the instruction pointed to by the PC is known
- Hugely important with out of order / superscalar processors



# I/O Software

- Device independence
- Uniform naming
- Error handling
  - Should be handled as close to the hardware as possible
- Synchronous vs asynchronous (interrupt-driven)
- Buffering
- Sharable versus dedicated devices

# Three Ways of Performing I/O

- Programmed I/O
- Interrupt-driven I/O
- I/O Using DMA

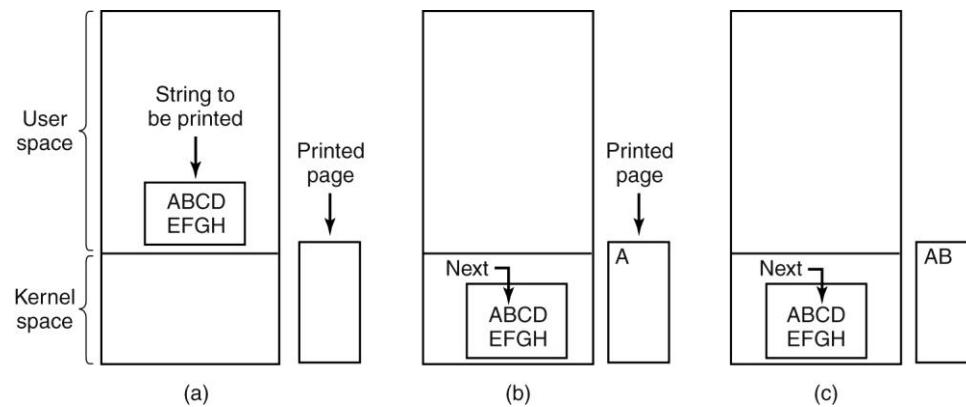
# Programmed I/O

- CPU does all the work
- Busy-waiting (polling)

Example:

```
copy_from_user(buffer, p, count);
for (i = 0; i < count; i++) {
    while (*printer_status_reg != READY) ;
    *printer_data_register = p[i];
}
return_to_user();
```

/\* p is the kernel buffer \*/
/\* loop on every character \*/
/\* loop until ready \*/
/\* output one character \*/

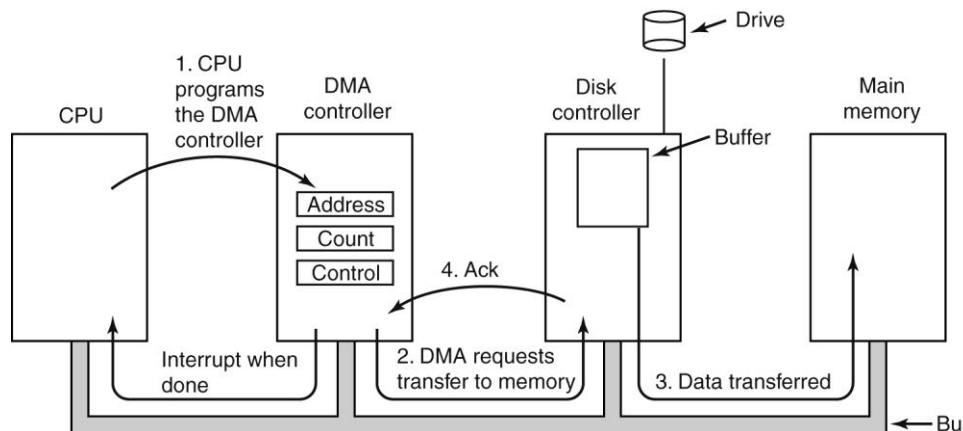


# Interrupt-Driven I/O

- Waiting for a device to be ready, the process is blocked and another process is scheduled.
- When the device is ready it raises an interrupt.
- Then data is copied by CPU as previous (avoids polling)
- Upon completion of the transfer the blocked process can be made ready again.

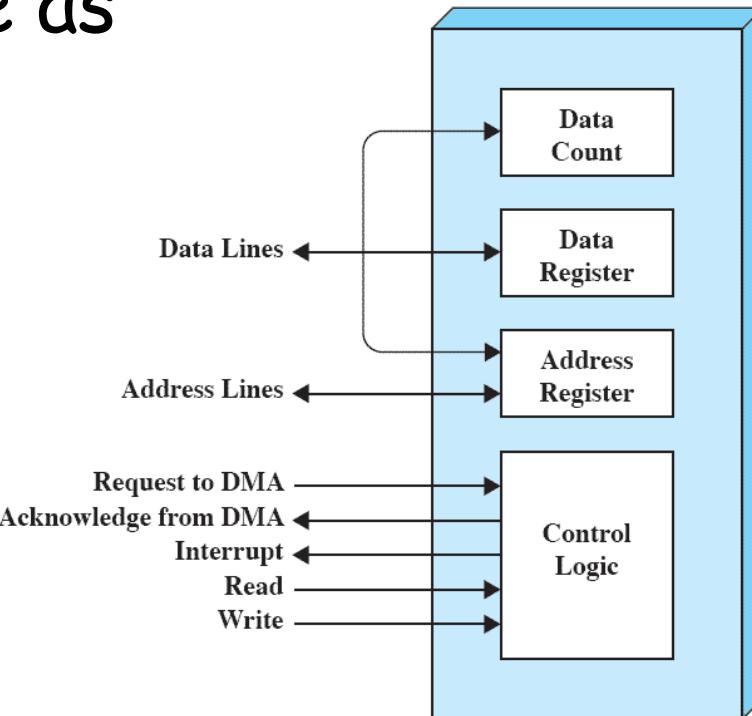
# Direct Memory Access (DMA)

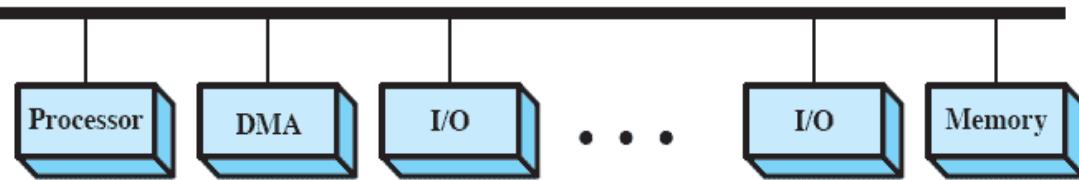
- It is not efficient for the CPU to request data from I/O one byte/word at a time
- DMA controller has access to the system bus independent of the CPU



# I/O Using DMA

- DMA does the work instead of the CPU
- Think of DMA engine as a simple “copy-core”
- Let the DMA do its work and then use interrupts to notify CPU

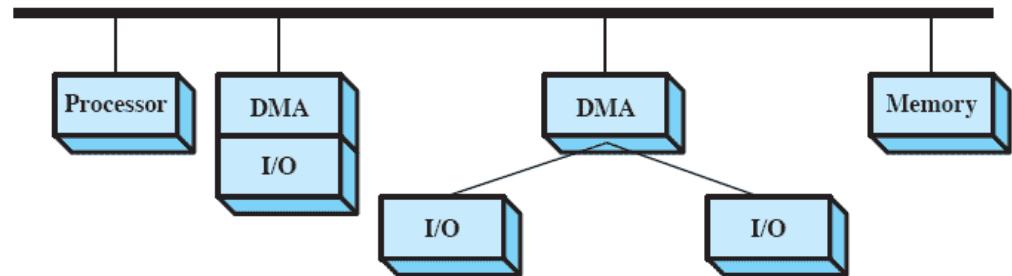




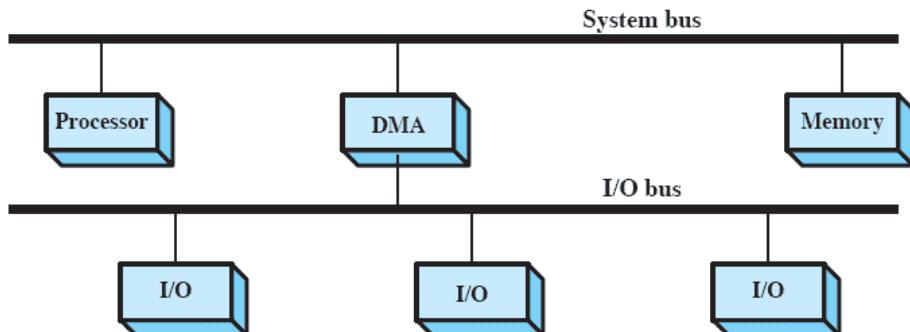
**DMA**

(a) Single-bus, detached DMA

# Alternative



(b) Single-bus, Integrated DMA-I/O

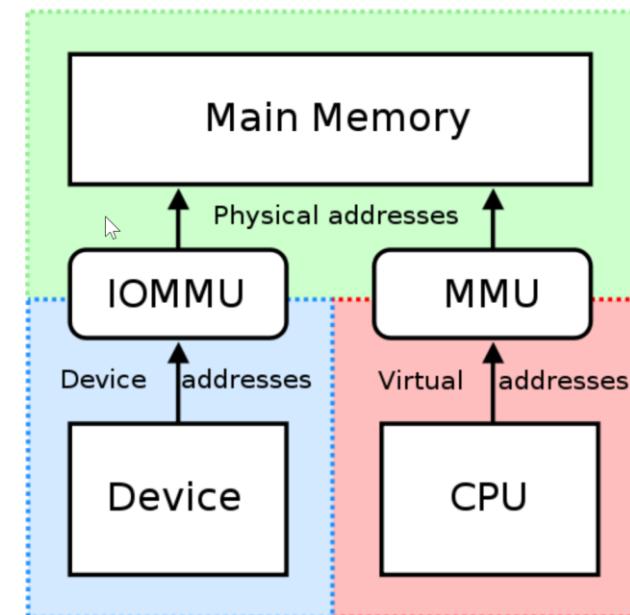


(c) I/O bus

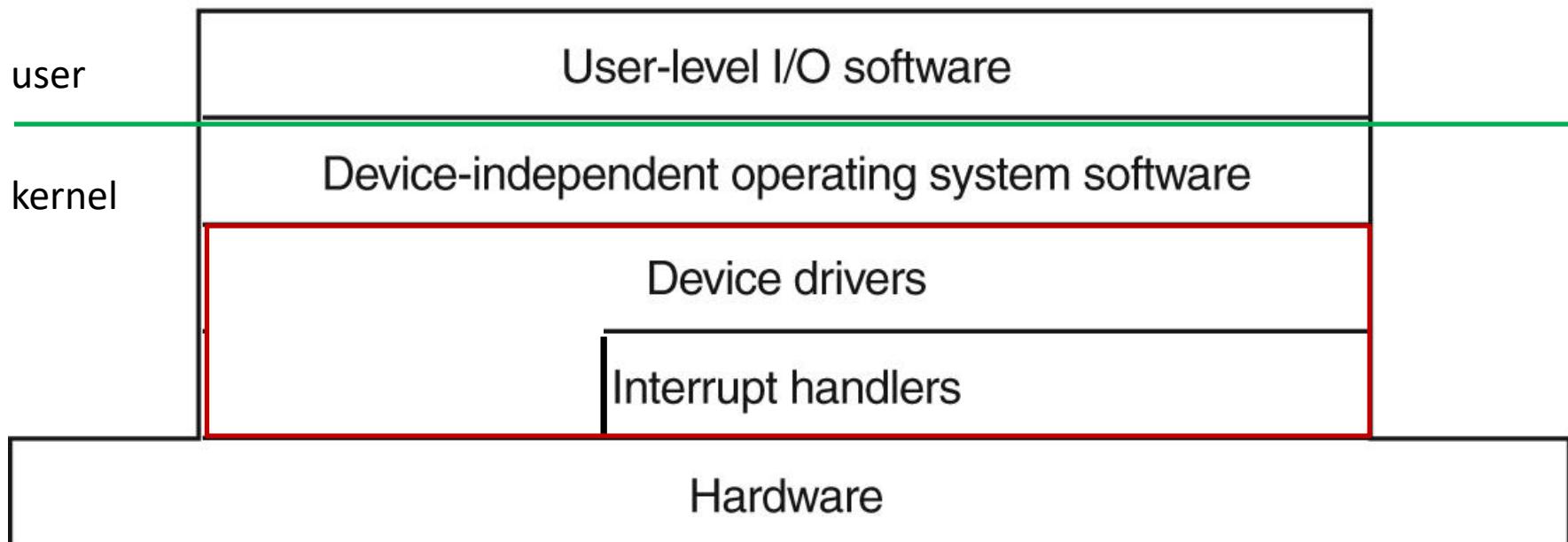
# Configurations

# DMA and Memory Protection

- Since DMA allows devices access to memory it can be a source of BUGs (and considerable headaches)
- System and Operating System protects itself with another translation table → IOMMU
- Enables OS to install only buffer addresses in IOMMU
- If Device attempts DMA to an address and there is no translation, an interrupt/exception is raised
- It is similar how apps are restricted to what memory they can access using a pagetable (MMU).



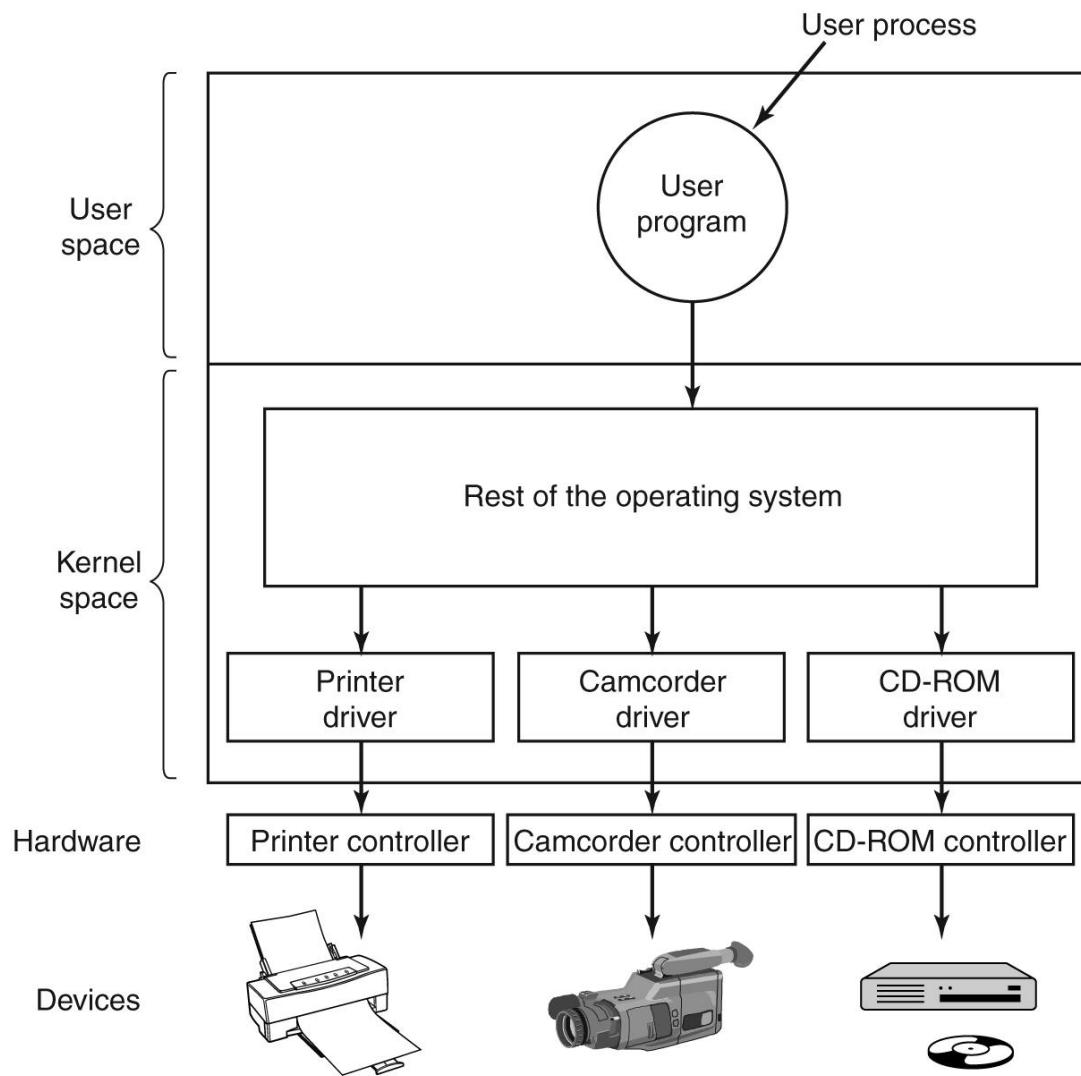
# OS Software Layers for I/O



# Device Drivers

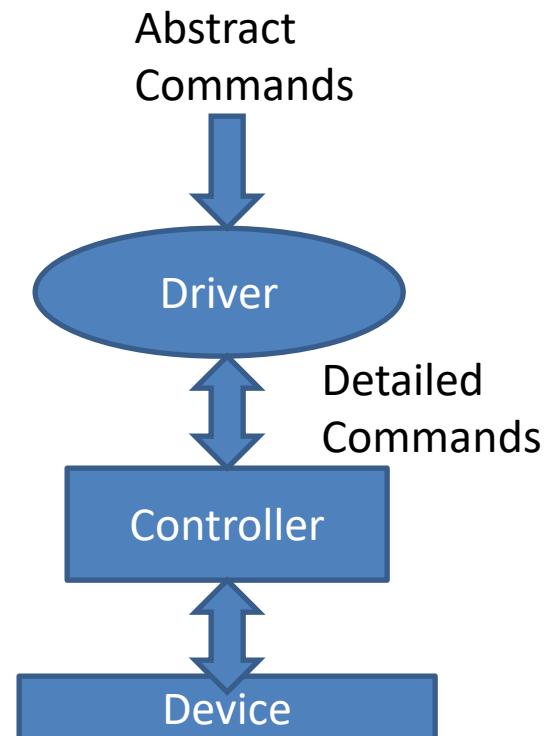
- Device specific code for controlling the device
  - Read device registers from controller
  - Write device registers to issue commands
  - Handle Interrupts
- Usually supplied by the device manufacturer
- Can be part of the kernel or at user-space (with system calls to access controller registers)
- OS defines a standard interface that drivers for block devices must follow and another standard for driver of character devices

# Device Drivers



# Device Drivers

- Main functions:
  - Initialize the device
  - Receive abstract read/write from layer above and execute them
  - Handle Interrupts
  - Log events
  - Manage power requirements
- Driver Code must be **reentrant**
  - Deal with multiple devices of same type
  - As a result they always pass a `struct dev *devobj` object to each function  
all state must be maintained in this object
- Drivers must deal with events such as a device removed or plugged



# Device Independent I/O Software

Uniform interfacing for device drivers

Buffering

Error reporting

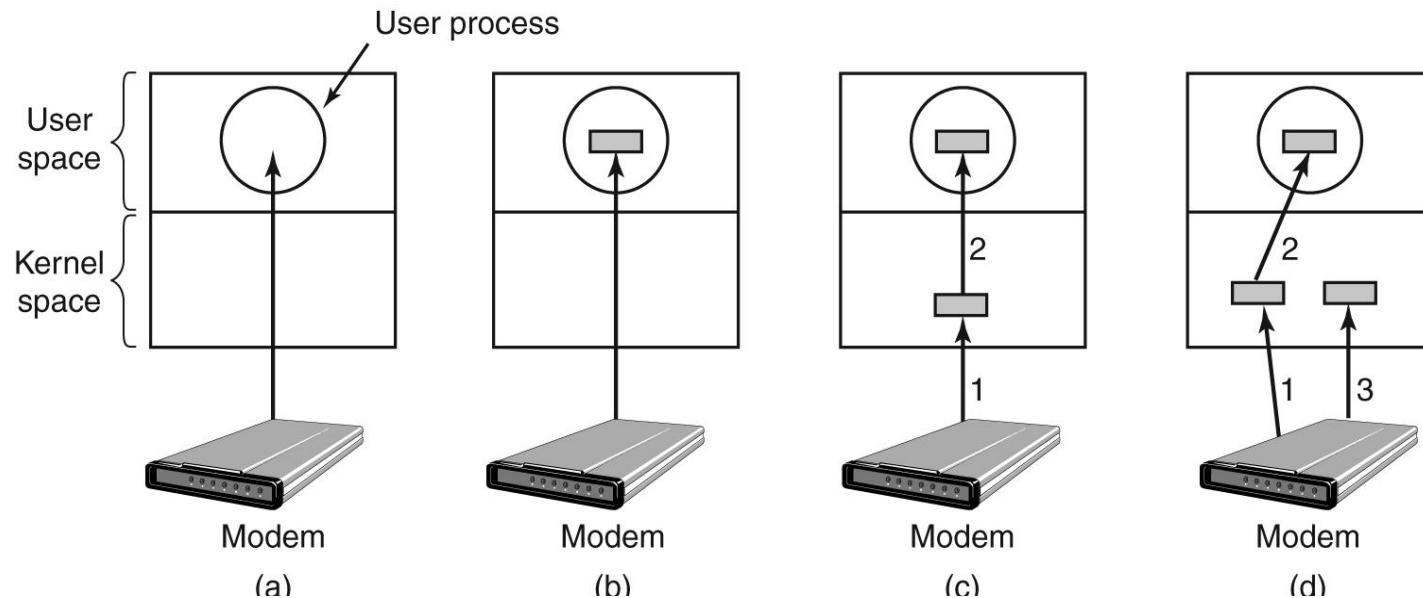
Allocating and releasing dedicated devices

Providing a device-independent block size

# Device Independent I/O Software

- Uniform interfacing for device drivers
  - Trying to make all devices look the same
  - For each class of devices, the OS defines a set of functions that the driver must supply.
  - This layer of OS maps symbolic device names onto proper drivers

# Device Independent I/O Software



Interrupt with every character

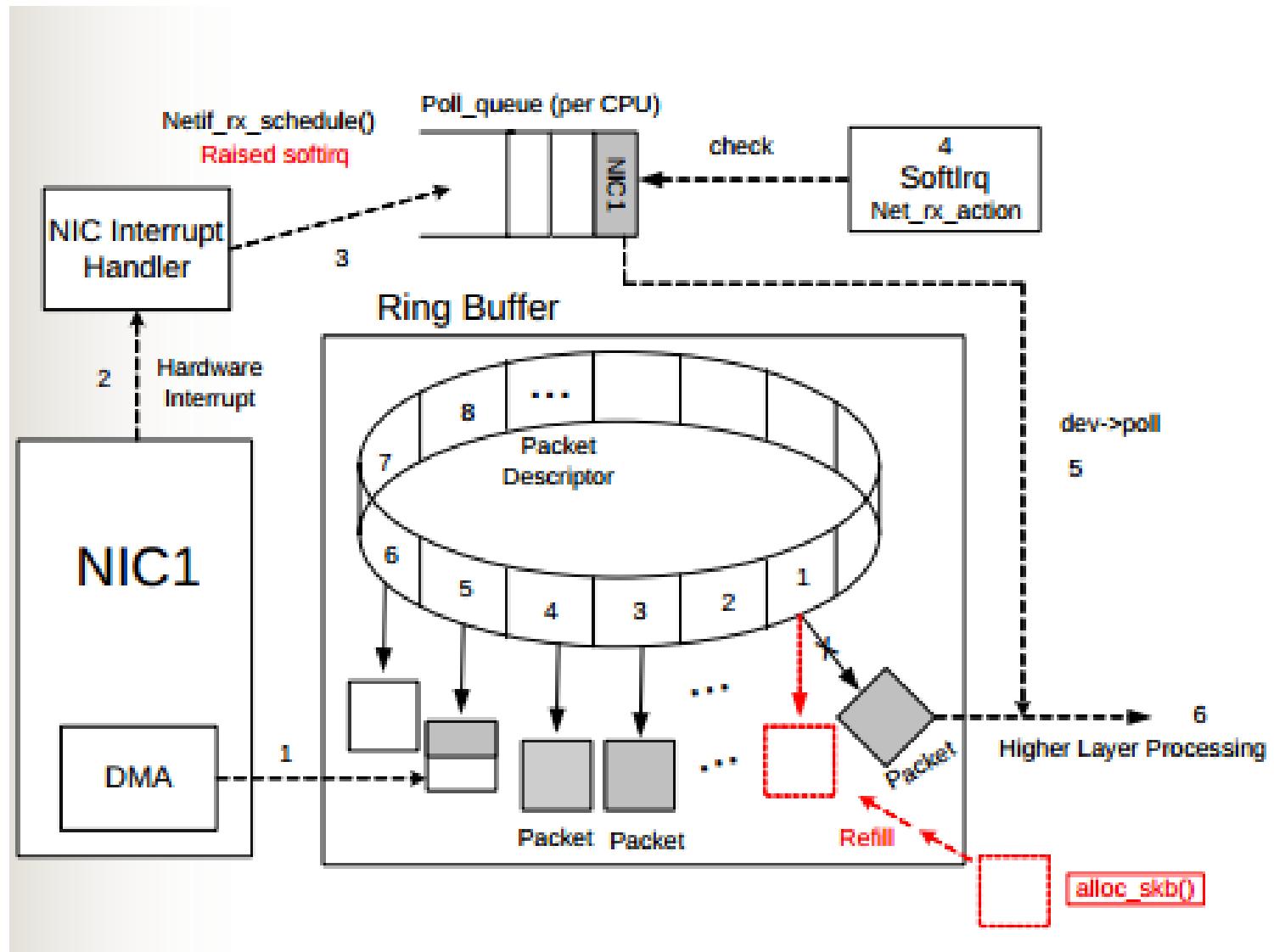
Very inefficient

Buffering n char.

What if buffer is paged out?

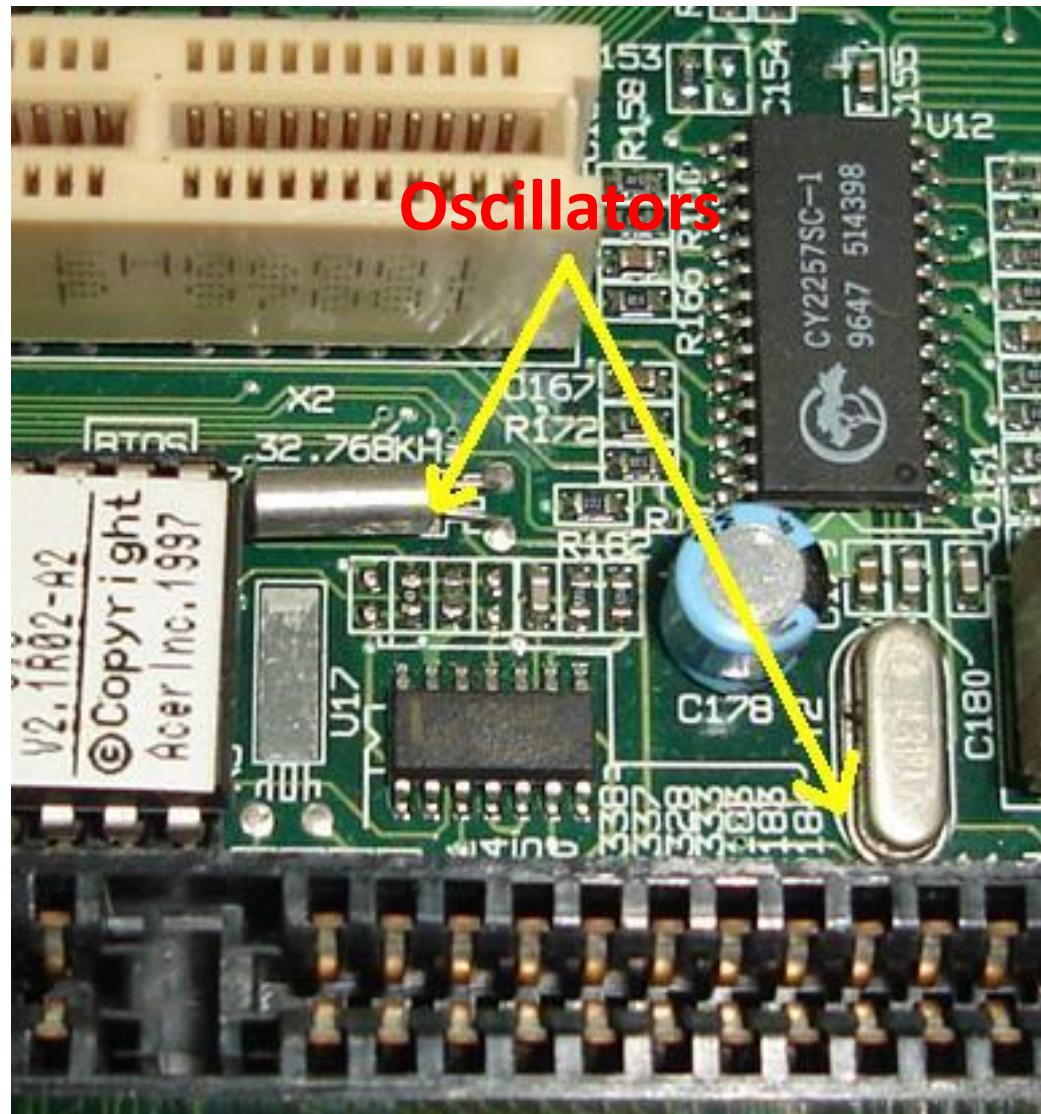
The need for buffering → buffering now is N buffers

# Linux Networking Driver



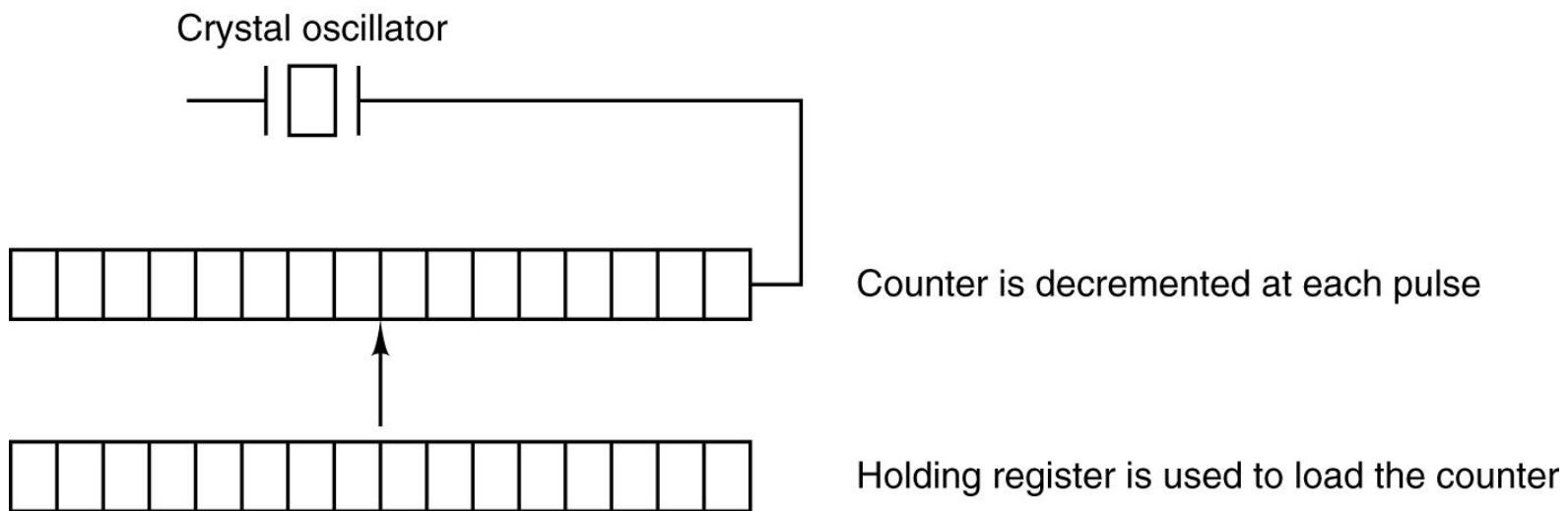
# Example of Devices

# Clocks



# Clock Hardware

- Old: tied to the power line and causes an interrupt on every voltage cycle
- New: Crystal oscillator + counter + holding register



# Clock Software

- Maintaining the time of the day
- Preventing processes from running longer than they are allowed to
- Accounting for CPU usage
- Handling alarm system call made by user processes
- Providing watchdog timers for parts of the system itself
- Doing profiling, monitoring, and statistics gathering

# User Interfaces

- Keyboard
- Mouse
- Monitor

As examples, we will take a closer look at the keyboard and mouse.

# Keyboards

- Contains simplified embedded processor that communicates through a port with the controller at the motherboard
- An interrupt is generated whenever a key is struck and a second one whenever a key is released
- Keyboard driver extracts the information about what happens from the I/O port assigned to the keyboard
- The number in the I/O port is called the **scan code** (7 bits for code + 1 bit for key press/release)

# Keyboards

# **Microprocessor of the keyboard**



# Key matrix

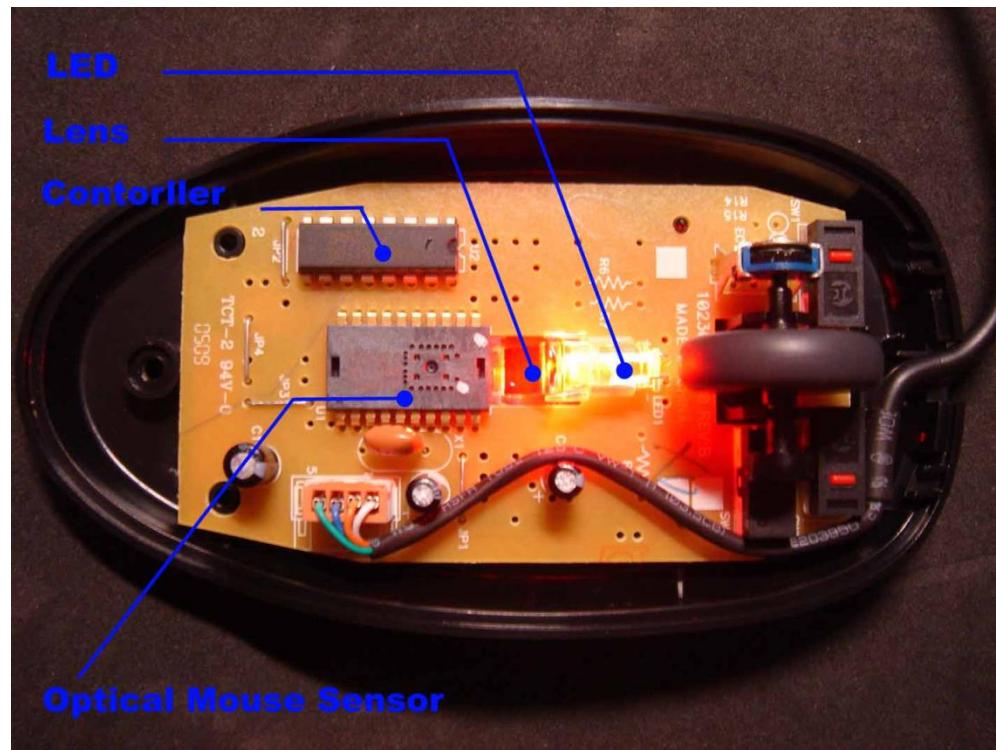


# Keyboards

- There are two philosophies for programs dealing with keyboards
  1. The programs gets a raw sequence of ASCII codes (raw mode, or noncanonical mode)
  2. Driver handles all the intraline editing and just delivered corrected lines to the user programs (cooked mode or canonical mode)
- Either way, a buffer is needed to store characters

# Mouse

- Mouse only indicates changes in position, not absolute position (`delta_x` and `delta_y`)



# I/O Software Layer: Principle

- Interrupts are facts of life, but should be hidden away, so that as little of the OS as possible knows about them.
- The best way to **hide interrupts** is to have the **driver starting an IO operation block** until IO has completed and the interrupt occurs.
- OS registers assigned Interrupt with Device's interrupt handler and when interrupts happens, the associated interrupt handler is invoked.
- Once the handling of interrupt is done, the **interrupt handler unblocks the thread in the device driver** that started it.
- This model works if **drivers are structured as kernel processes** with their own states, stacks and program counters.

# More complex real device drivers

- Character Drivers
  - Discovery
  - Read / write data
- Disk storage
  - discovery
  - read block
  - write block
  - Interrupt management
- Network Cards (will talk about it in last class)
  - Send packet
  - Receive packet
  - Interrupt management
  - Throttling

# Character Device Driver

- Let's do a life walk through from the linux-kernel-labs
- [https://linux-kernel-labs.github.io/refs/heads/master/labs/device\\_drivers.html](https://linux-kernel-labs.github.io/refs/heads/master/labs/device_drivers.html)

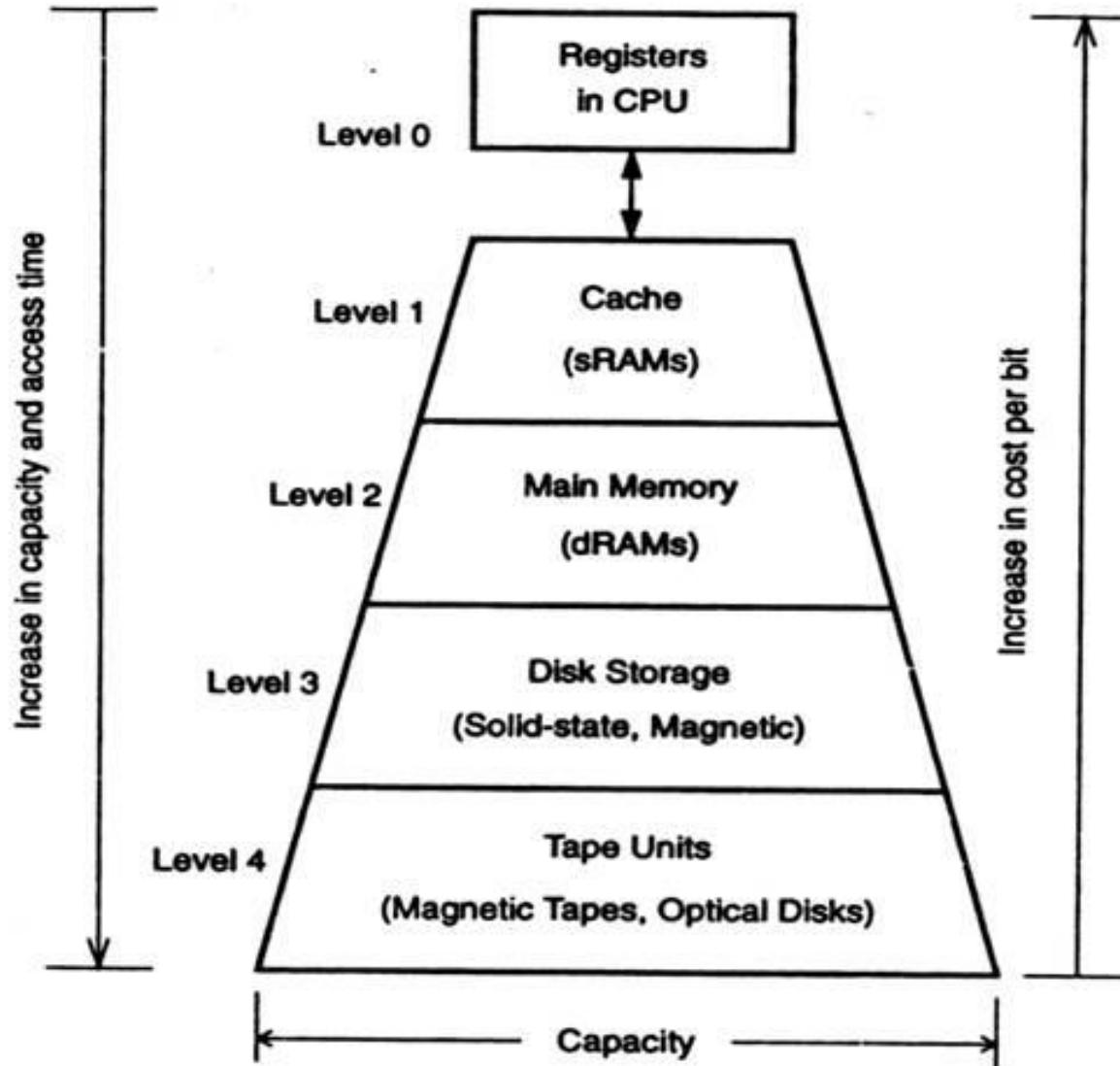
# Block Device Driver

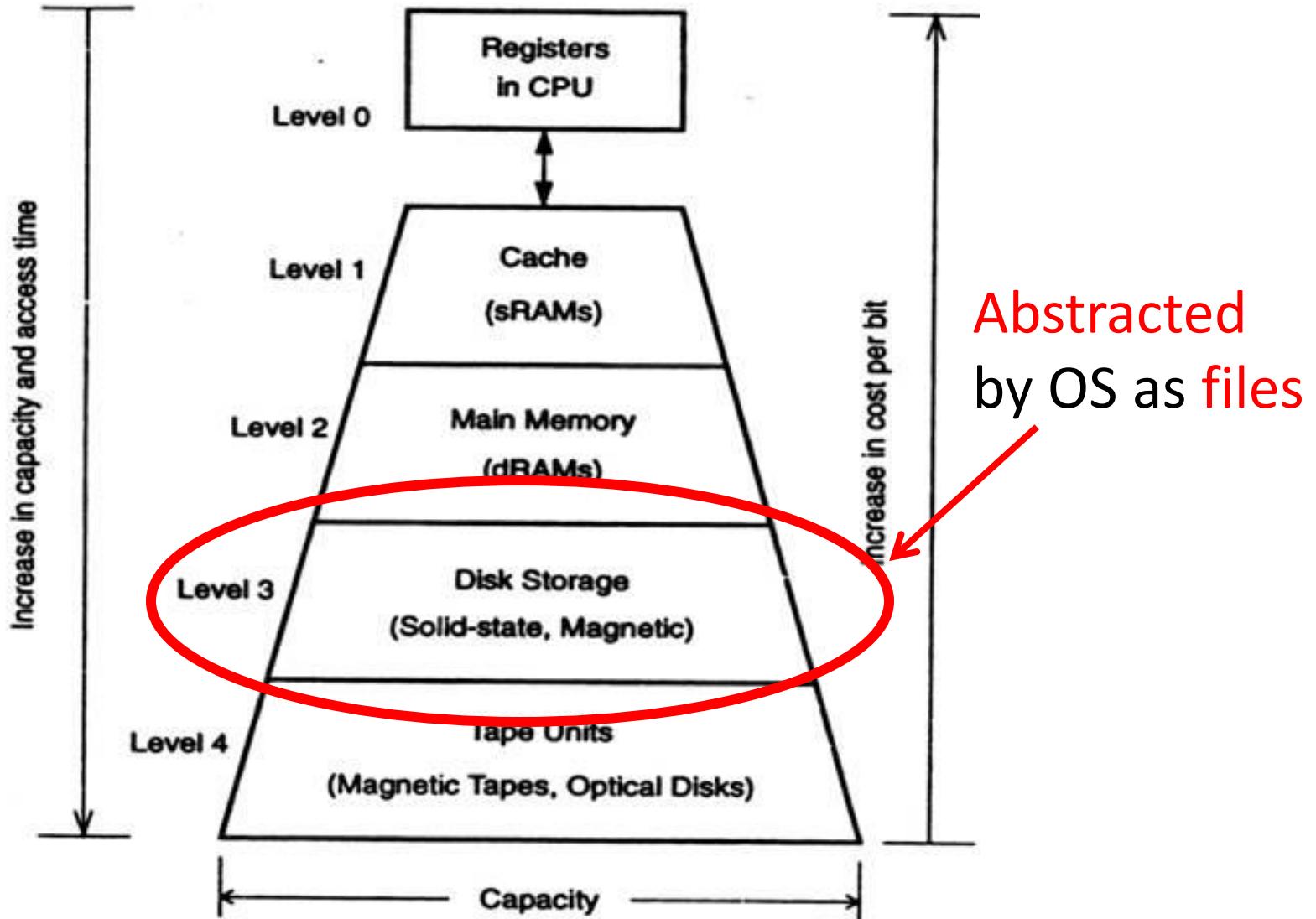
- Let's do a life walk through from the linux-kernel-labs
- [https://linux-kernel-labs.github.io/refs/heads/master/labs/block\\_device\\_drivers.html](https://linux-kernel-labs.github.io/refs/heads/master/labs/block_device_drivers.html)

# Conclusions

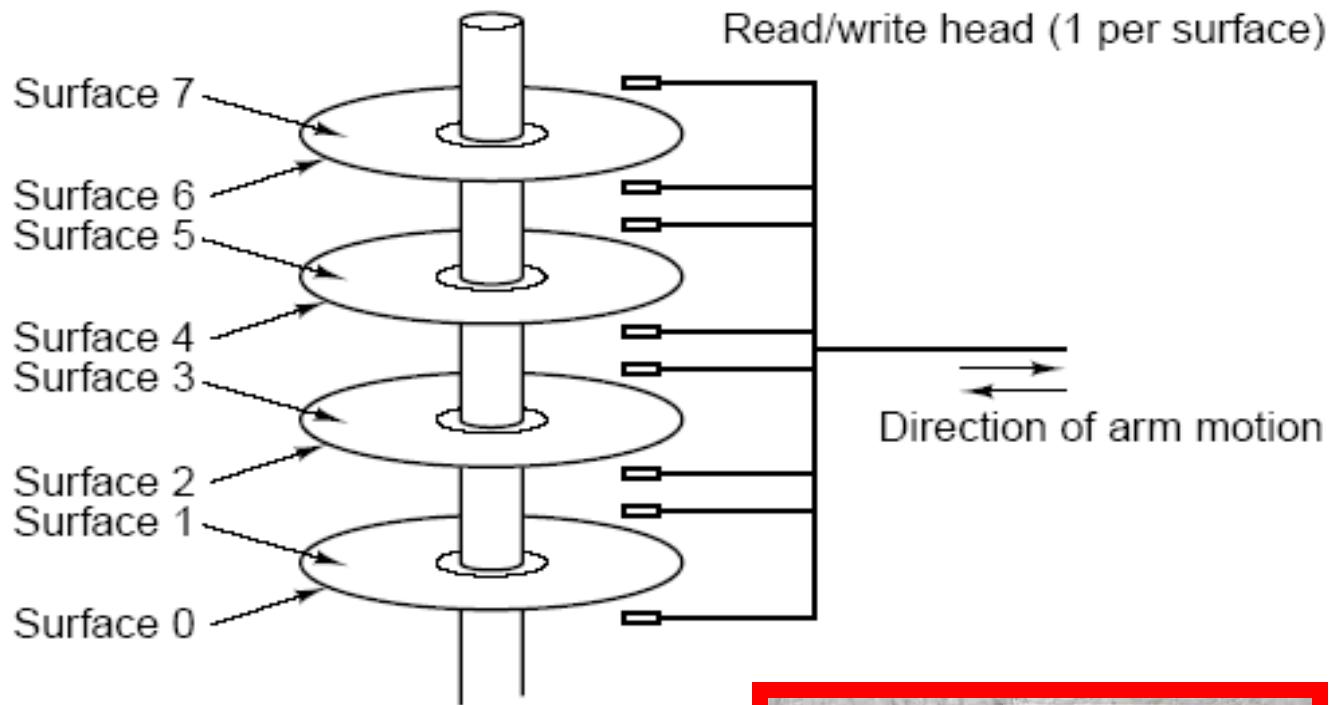
- The OS provides an interface between the devices and the rest of the system.
- The I/O part of the OS is divided into several layers.
- The hardware: CPU, programmable interrupt controller, DMA, device controller, and the device itself.
- OS must expand as new I/O devices are added

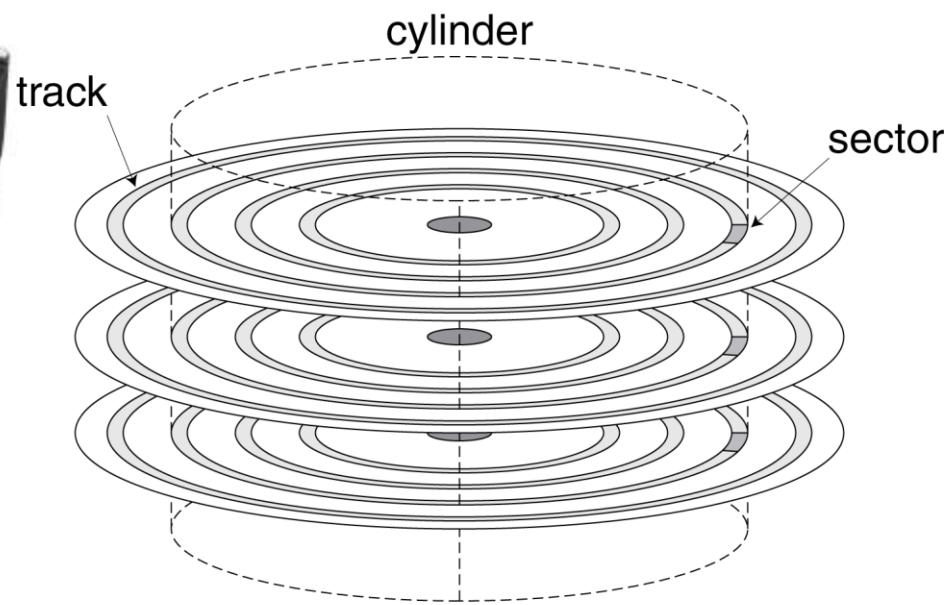
# Disks Scheduling





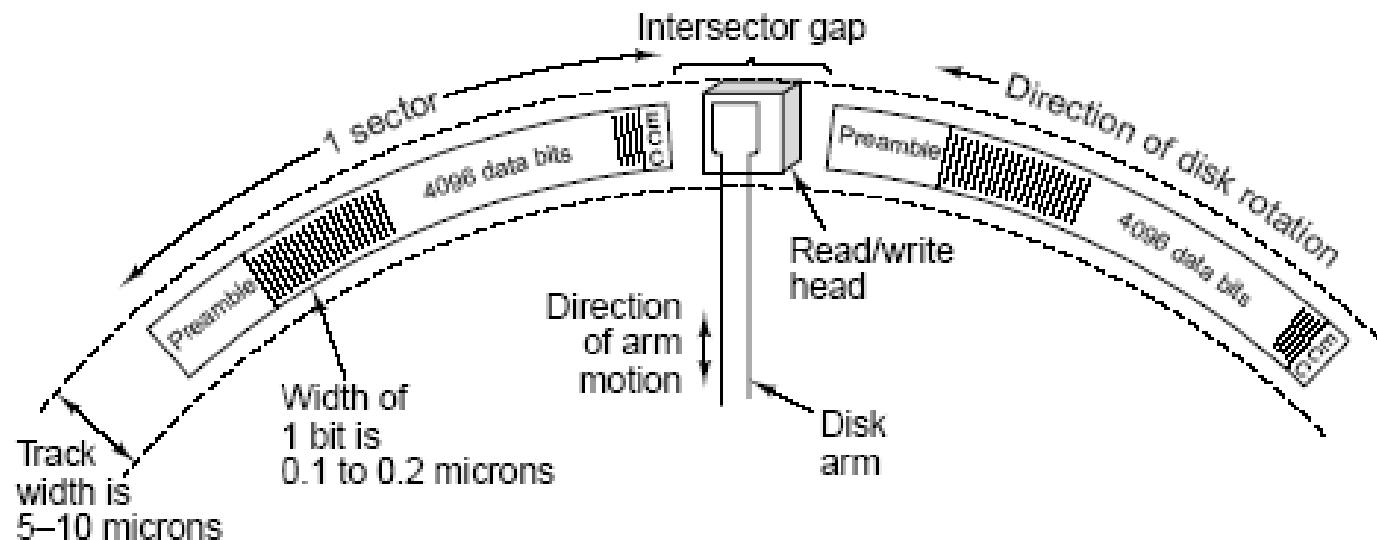
# A Conventional Hard Disk (Magnetic) Structure





# Hard Disk (Magnetic) Architecture

- **Surface** = group of tracks
- **Track** = group of sectors
- **Sector** = group of bytes
- **Cylinder**: several tracks on corresponding surfaces



# Hard Disk Performance

- **Seek**
  - Position heads over cylinder, typically we assume 10msec avg:
    - 4 msecs for highend disk drives to
    - 15 msecs for mobile devices
- **Rotational delay**
  - Wait for a sector to rotate underneath the heads
  - Typically 7.14 – 2.0 ms (4,200 – 15,000 rpm)  
[ or  $\frac{1}{2}$  rotation ]
- **Transfer bytes**
  - Average transfer bandwidth (50-80 Mbytes/sec @4KB block sz)

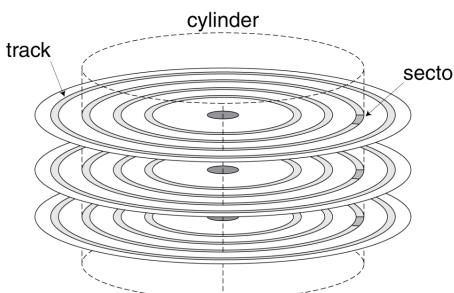
HDD spindle speed [rpm]	Average rotational latency [ms]
4,200	7.14
5,400	5.56
7,200	4.17
10,000	3.00
15,000	2.00

# Hard Disk Performance

- Performance of transfer 1 Kbytes
  - Seek (5.3 ms) + half rotational delay (3ms) + transfer (0.04 ms)
  - Total time is 8.34ms or 120 Kbytes/sec!
- What block size can get 90% of the disk transfer bandwidth?

# Disk Behaviors (example)

- There are more sectors on outer tracks than inner tracks
  - Read outer tracks: 37.4MB/sec
  - Read inner tracks: 22MB/sec
- Seek time and rotational latency dominate the cost of small reads
  - A lot of disk transfer bandwidth is wasted
  - Need algorithms to reduce seek time



Block Size (Kbytes)	% of Disk Transfer Bandwidth
1Kbytes	0.5%
8Kbytes	3.7%
256Kbytes	55%
1Mbytes	83%
2Mbytes	90%

# Observations

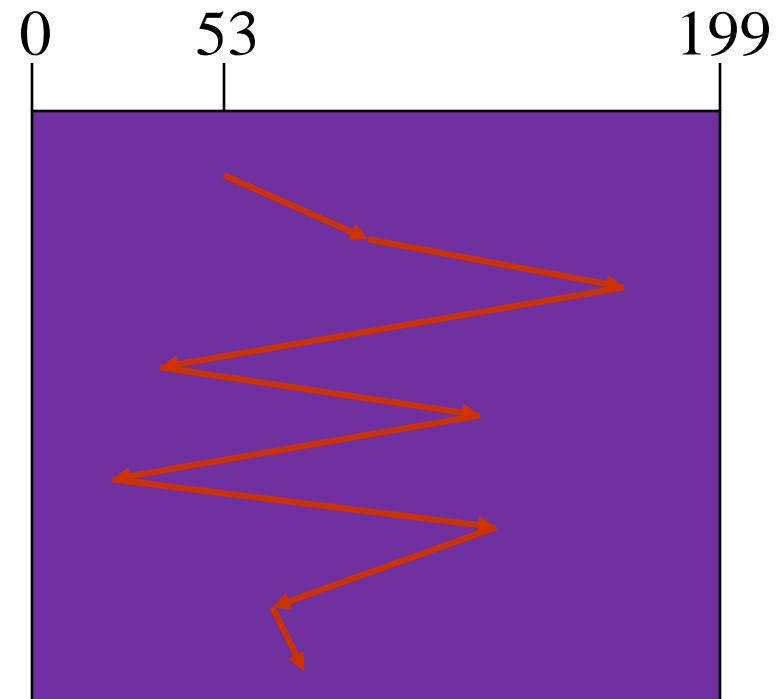
- Getting first byte from disk read is slow
  - high latency
- Peak bandwidth high, but rarely achieved
- Need to mitigate disk performance impact
  - Do extra calculations to speed up disk access
    - Schedule requests to shorten seeks
  - Move some disk data into main memory - file system caching

# Disk Scheduling

- Which disk request is serviced first?
  - FCFS (FIFO)
  - Shortest Seek Time First (SSTF)
  - Elevator (SCAN)
  - LOOK
  - C-SCAN (Circular SCAN)
  - C-LOOK
- Looks familiar?

# FIFO (FCFS) order

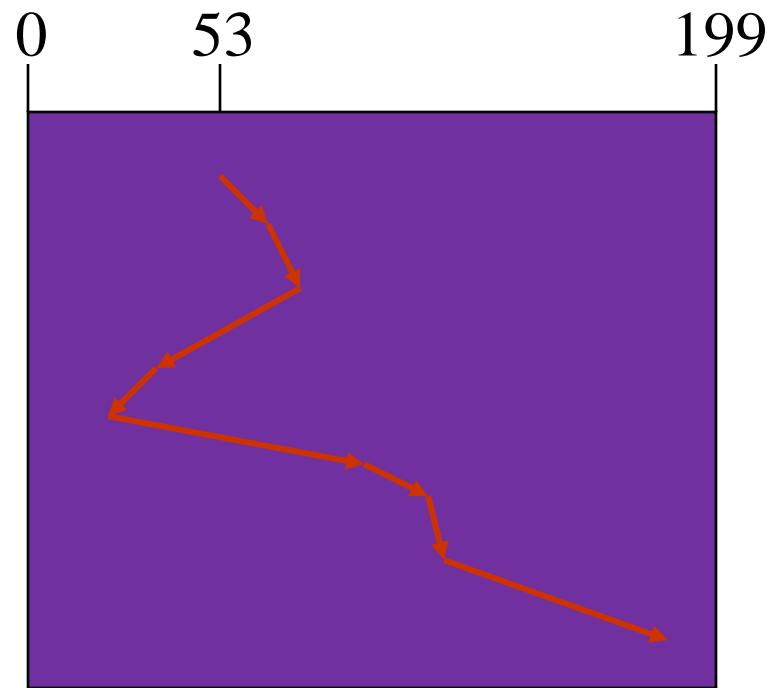
- Method
  - First come first serve
- Pros
  - Fairness among requests
  - In the order applications expect
- Cons
  - Arrival may be on random spots on the disk (long seeks)
  - Wild swing can happen
- Analogy:
  - Can elevator scheduling use FCFS?



98, 183, 37, 122, 14, 124, 65, 67

# SSTF (Shortest Seek Time First)

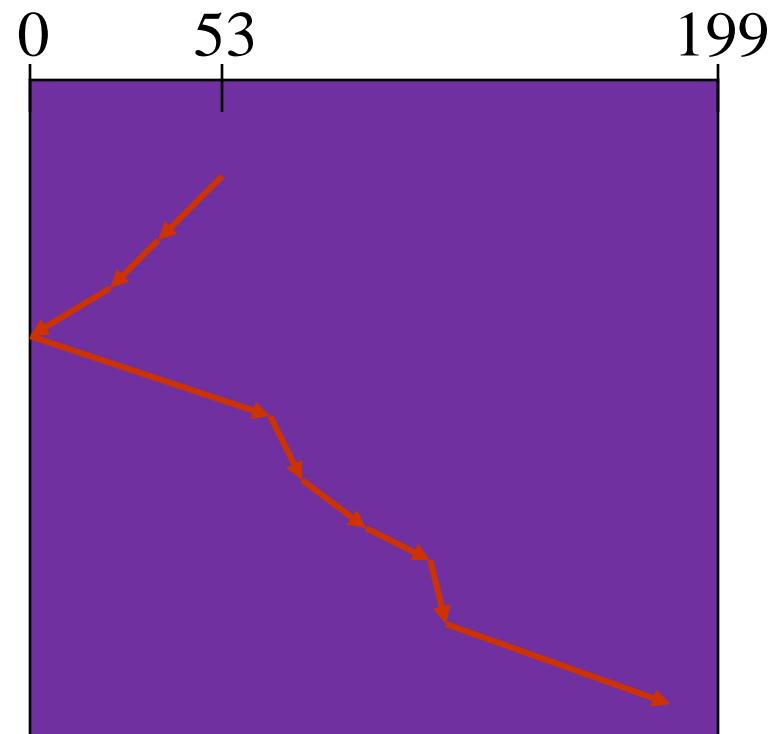
- Method
  - Pick the one closest on disk
  - Rotational delay is in calculation
- Pros
  - Try to minimize seek time
- Cons
  - Starvation
- Question
  - Is SSTF optimal?
  - Can we avoid starvation?
- Analogy: elevator



98, 183, 37, 122, 14, 124, 65, 67  
(65, 67, 37, 14, 98, 122, 124, 183)

# Elevator (SCAN)

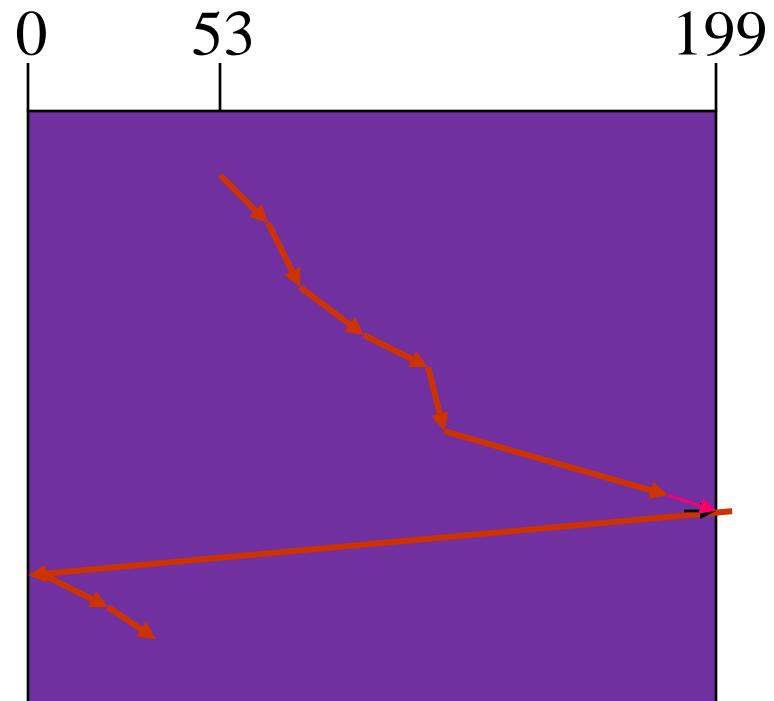
- Method
  - Take the closest request in the direction of travel
  - Real implementations do not go to the end (called LOOK)
- Pros
  - Bounded time for each request
- Cons
  - Request at the other end will take a while



98, 183, 37, 122, 14, 124, 65, 67  
(37, 14, 0, 65, 67, 98, 122, 124, 183)

# C-SCAN (Circular SCAN)

- Method
  - Like SCAN
  - But, wrap around
  - Real implementation doesn't go to the end (C-LOOK)
- Pros
  - Uniform service time
- Cons
  - Do nothing on the return



98, 183, 37, 122, 14, 124, 65, 67  
(65, 67, 98, 122, 124, 183, 199, 0, 14, 37)

# LOOK and C-LOOK

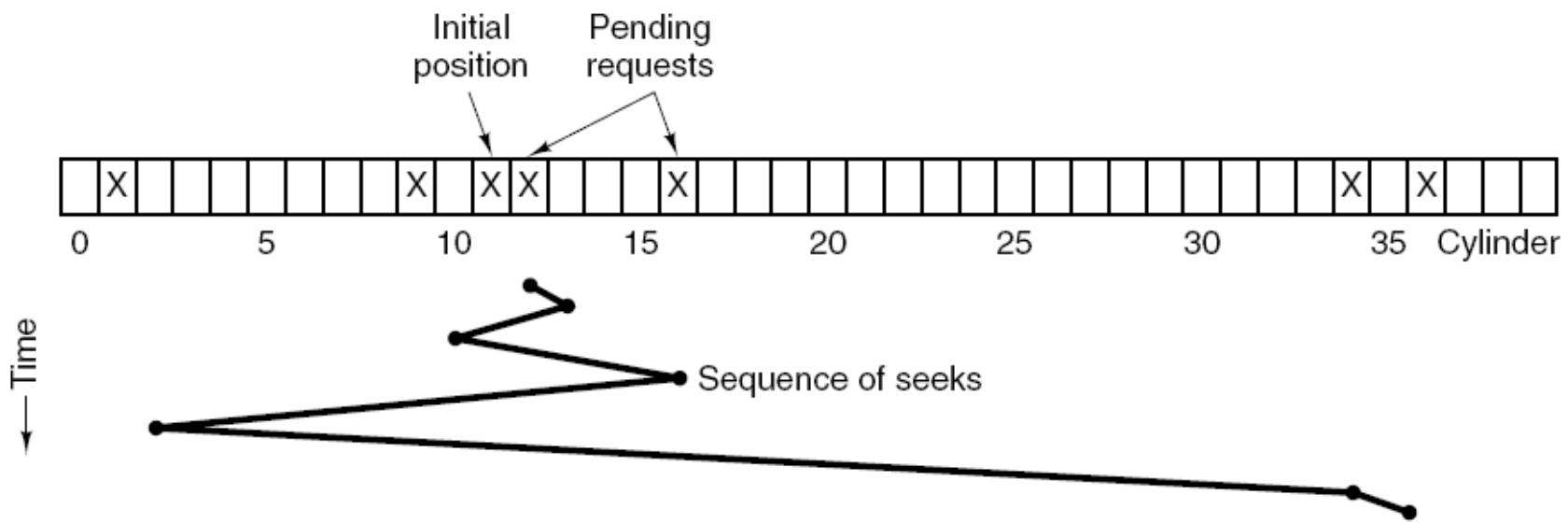
- SCAN and C-SCAN move the disk arm across the full width of the disk
- In practice, neither algorithm is implemented this way
- More commonly, the arm goes only as far as the final request in each direction. Then, it reverses direction immediately, without first going all the way to the end of the disk.
- These versions of SCAN and C-SCAN are called LOOK and C-LOOK

# FSCAN / FLOOK

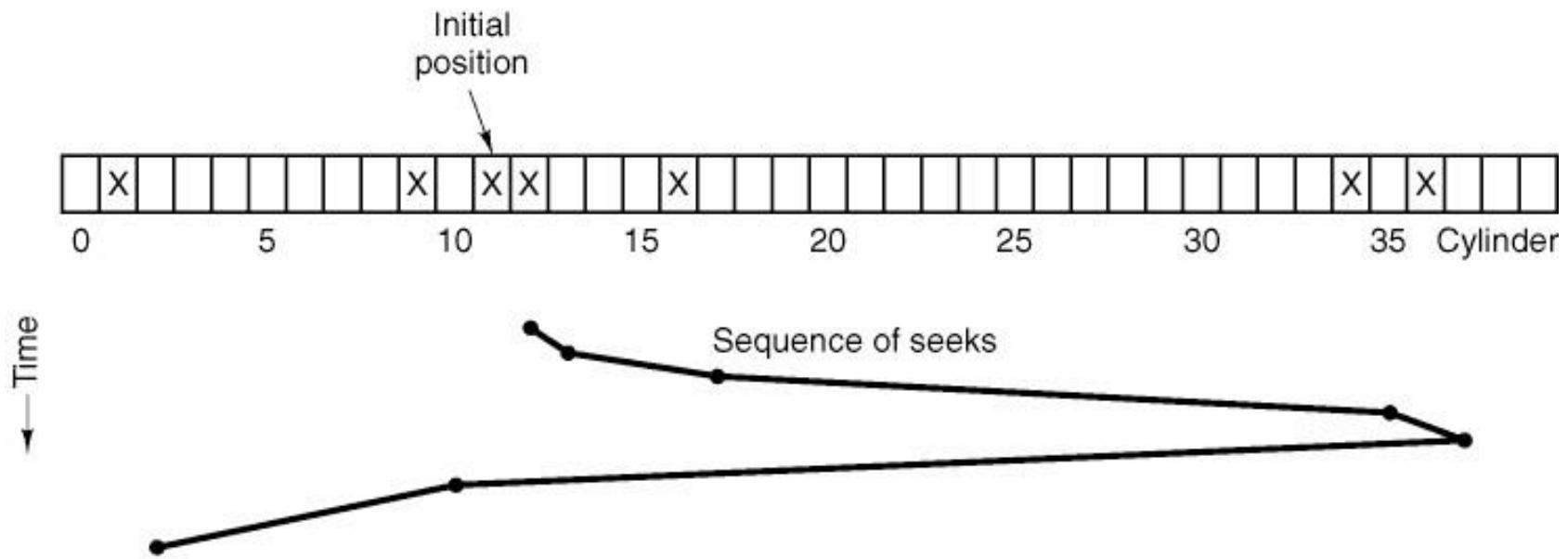
- Like SCAN / LOOK respectively... but
- Operates with two queues:
  - active-queue
  - add-queue
- any new I/O request is entered into add-queue.
- I/O is only scheduled from active-queue, when active-queue is empty, the queues are swapped (pointers !!) and tried one more time.

# Which algorithm is this?

(assume I/O arrived in random)



# Which algorithm is this?



# Other modern schedulers (1)

- Deadline Scheduler
  - Guarantees a start time for each I/O requests → deadline
  - Read requests have an expiration time of 500 ms, write requests expire in 5 seconds.
  - Maintains read + write deadline queues + sorted (by sector) queue
  - If a deadline expired then it is served otherwise, a batch is served from sorted queue.

# Other modern schedulers (2)

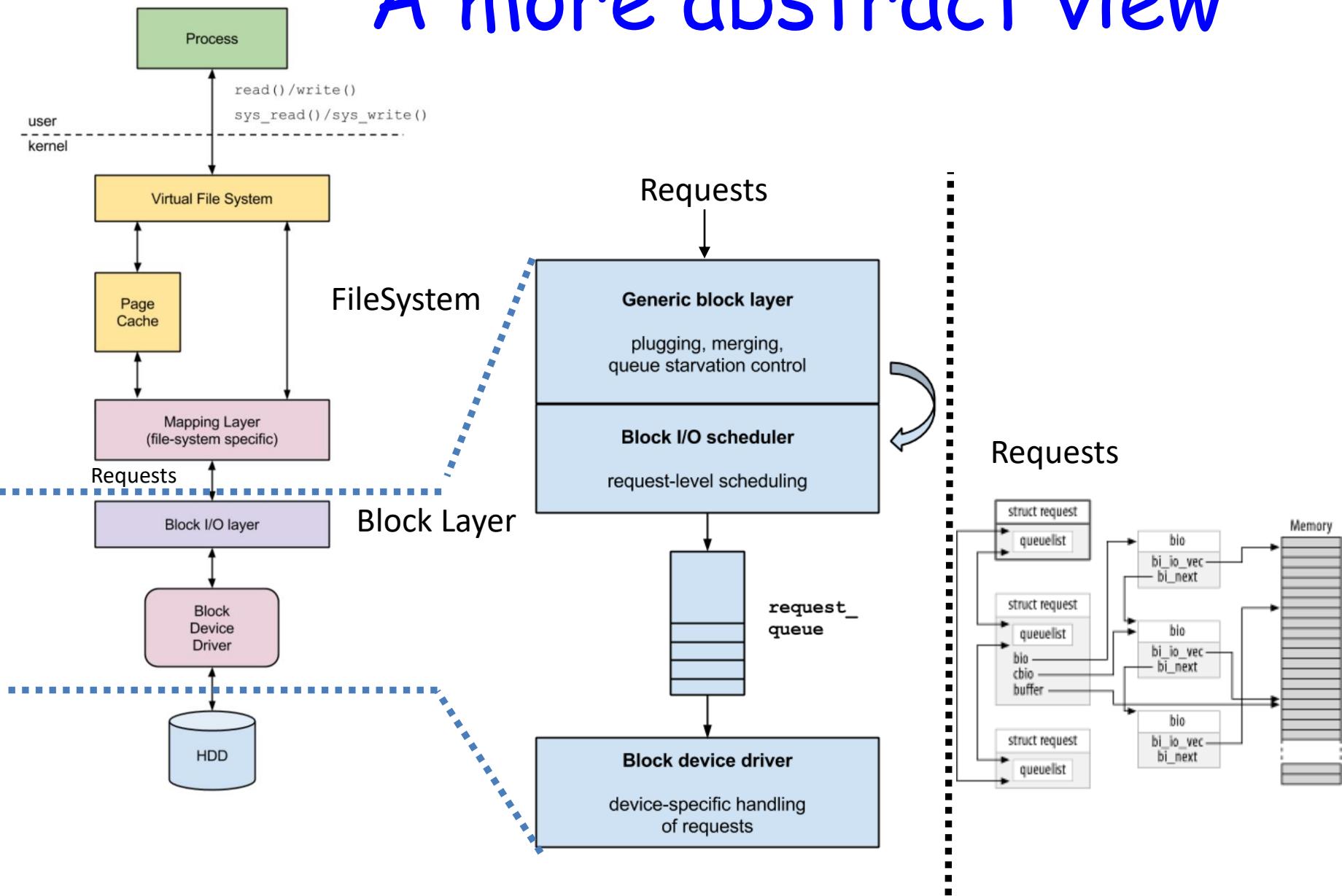
- CFQ (completely fair queueing)
  - places requests submitted into a number of per-process queues
  - allocates timeslices for each queue to access the disk.
  - The length of the time slice and the number of requests a queue is allowed to submit depends on the I/O priority of the given process.
  - Implicitly provides “idle” time at end of I/O request to allow subsequent requests to be issued and bundled

# Other modern schedulers (3)

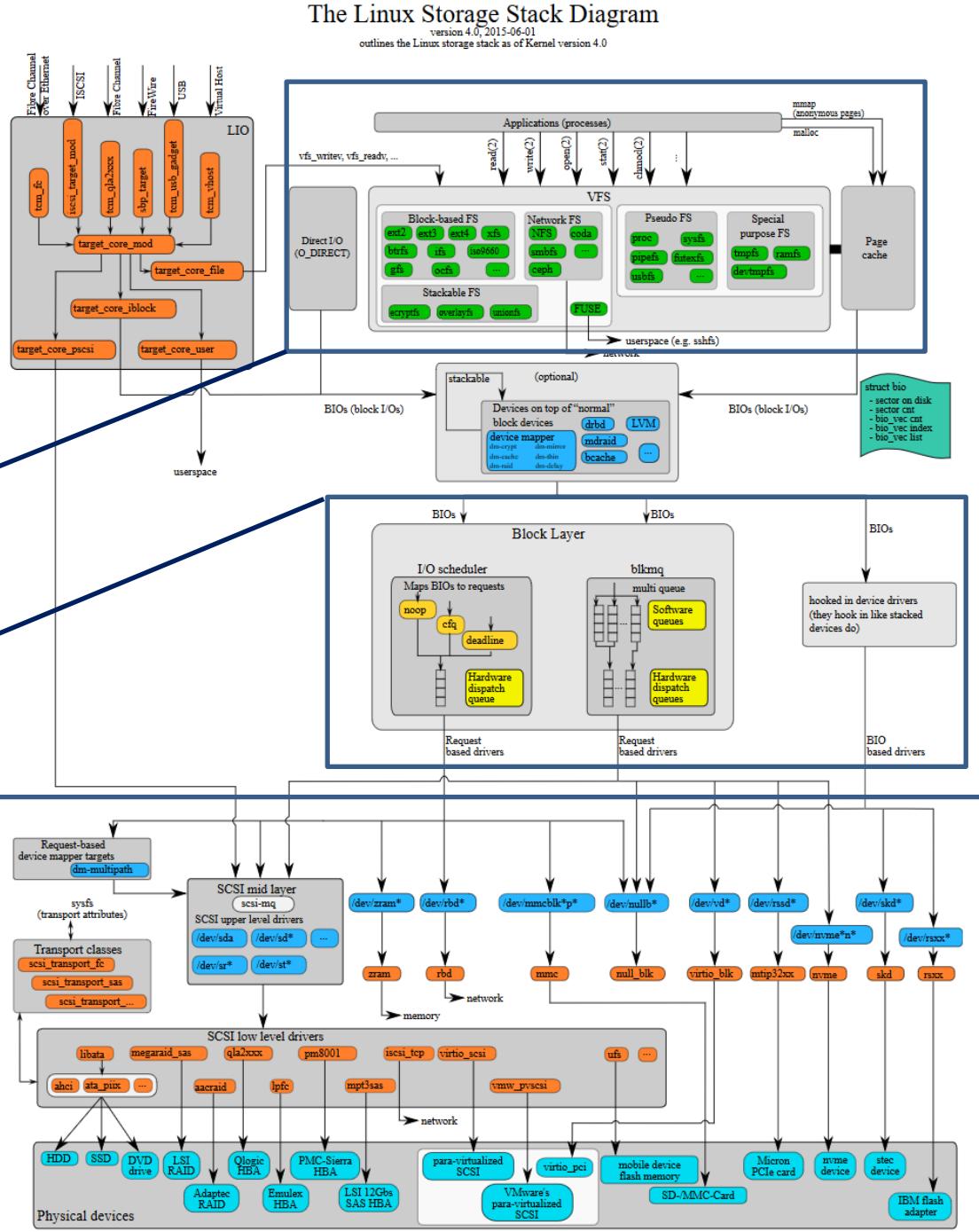
- Linux allows selection (dynamically) on a per I/O device
- LIVE DEMO

File: /sys/block/sda/queue/scheduler

# A more abstract view



# Full Linux I/O Stack



RAID



# Common Hard Drive Errors

1. Programming error
  - request for nonexistent sector
2. Transient checksum error
  - caused by dust on the head
3. Permanent checksum error
  - disk block physically damaged
4. Seek error
  - the arm sent to cylinder 6 but it went to 7
5. Controller error
  - controller refuses to accept commands

# RAID

- *Redundant Array of Independent Disks* OR  
*Redundant Array of Inexpensive Disks*
- Use parallel processing to speed up CPU performance
- Use parallel I/O to improve disk performance, reliability (1988, Patterson)
- Design new class of I/O devices called RAID - Redundant Array of Inexpensive Disks (also Redundant Array of Independent Disks)
- Use the RAID in OS as a SLED (Single Large Expensive Disk), but with better performance and reliability

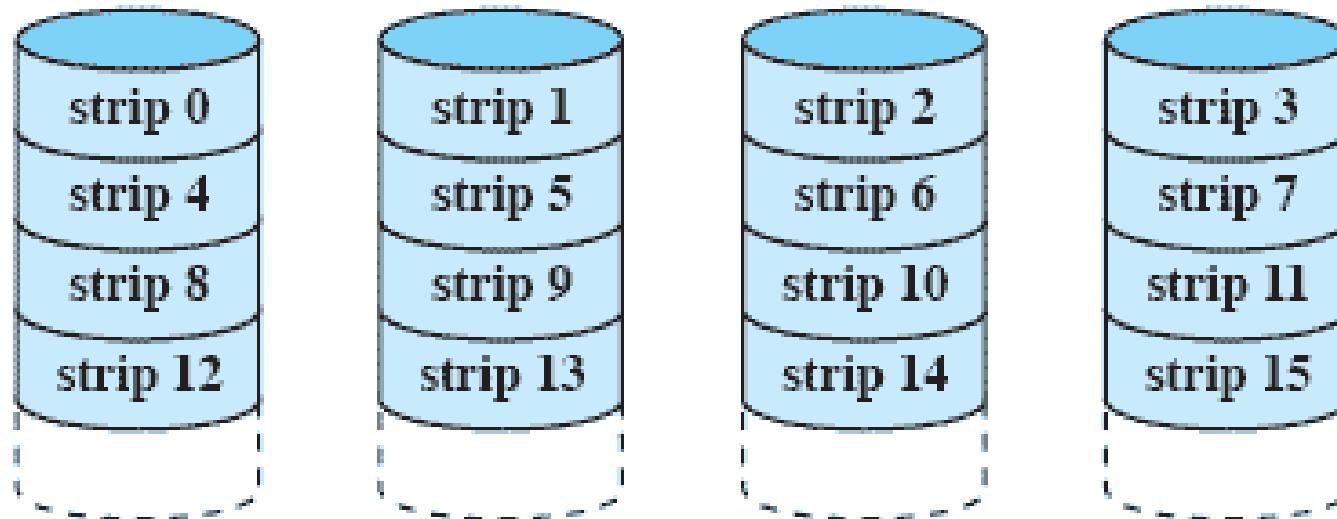
# RAID

- RAID consists of RAID SCSI controller plus a box of SCSI disks
- Data are divided into strips and distributed over disks for parallel operation
- RAID 0 ... RAID 5 levels
- RAID 0 organization writes consecutive strips over the drives in round-robin fashion - operation is called striping
- RAID 1 organization uses striping and duplicates all disks
- RAID 2 uses words, even bytes and stripes across multiple disks; uses error codes, hence very robust scheme
- RAID 3, 4, 5 alterations of the previous ones

Small Computer System Interface (SCSI, [/skʌzi/ \*\*SKUZ-ee\*\*](#)) is a set of standards for physically connecting and transferring data between computers and [peripheral devices](#). The SCSI standards define [commands](#), [protocols](#), electrical and optical [interfaces](#). (source Wikipedia)

# RAID Level 0

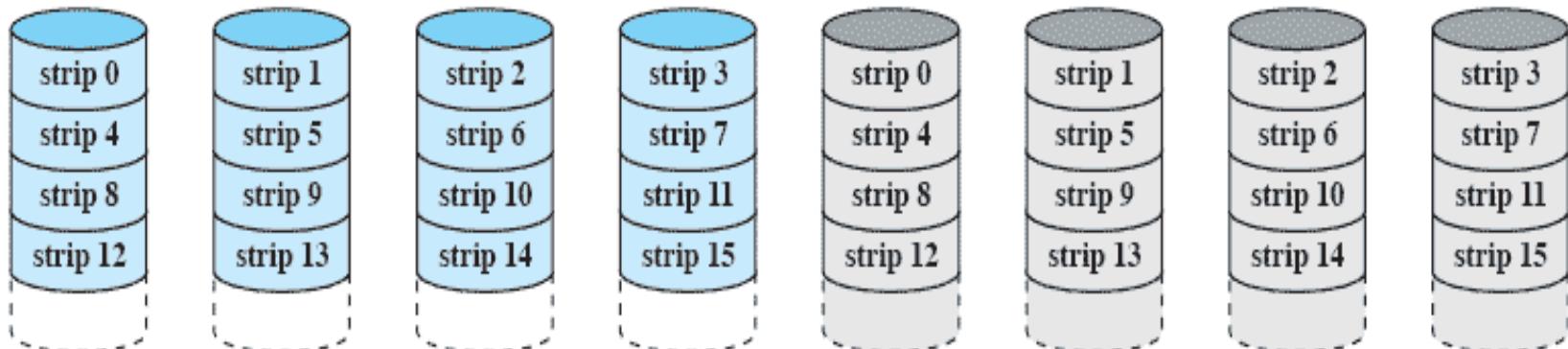
- Not a true RAID because it does not include redundancy to improve performance or provide data protection
- User and system data are distributed across all of the disks in the array
- Logical disk is divided into strips



(a) RAID 0 (non-redundant)

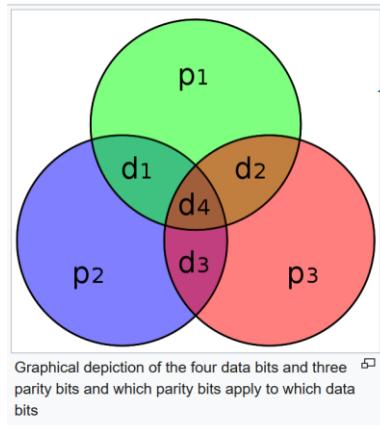
# RAID Level 1

- Redundancy is achieved by the simple expedient of duplicating all the data
- There is no “write penalty”
- When a drive fails the data may still be accessed from the second drive
- Principal disadvantage is the cost



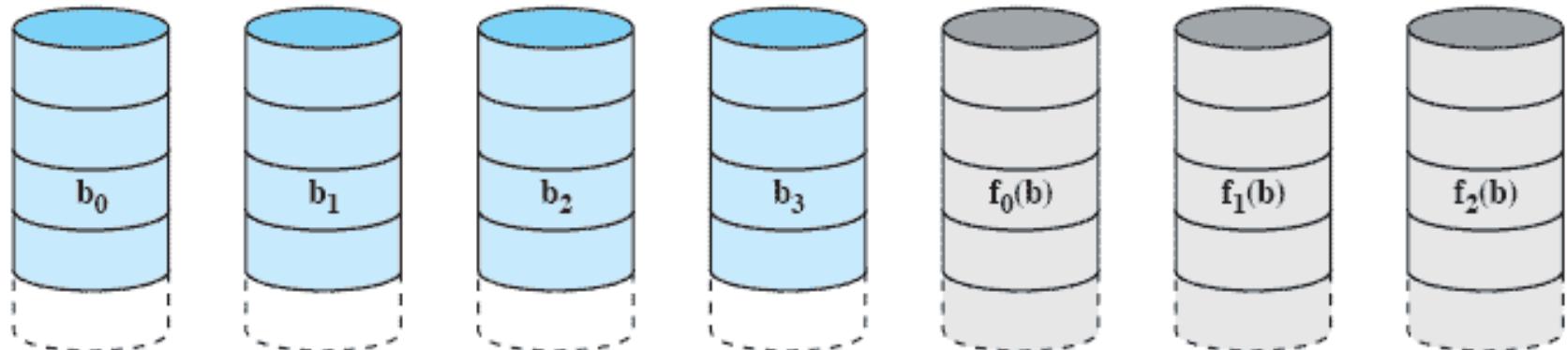
(b) RAID 1 (mirrored)

# RAID Level 2



Requires  $> \log(N)$  redundant disks

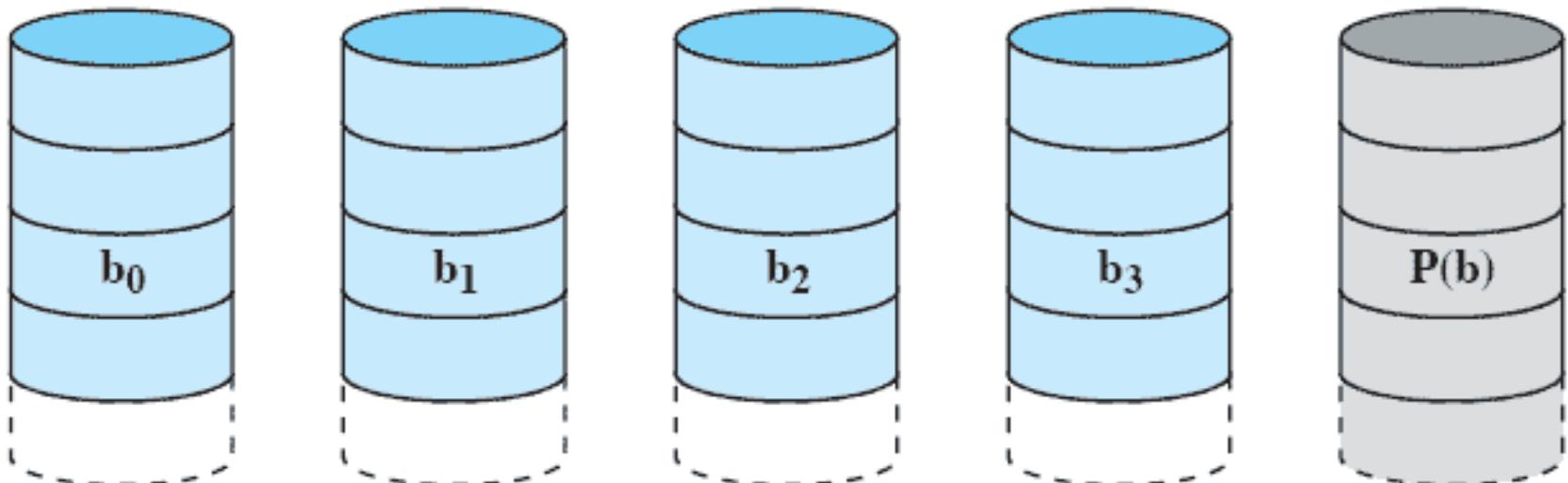
- Makes use of a parallel access technique, all disks participate
- Data striping is used, strips are very small, sometimes byte/words.
- Typically a Hamming code is used
- Effective choice in an environment in which many disk errors occur
- Can correct-single bit, detect 2-bit
- On read all disks are read



(c) RAID 2 (redundancy through Hamming code)

# RAID Level 3

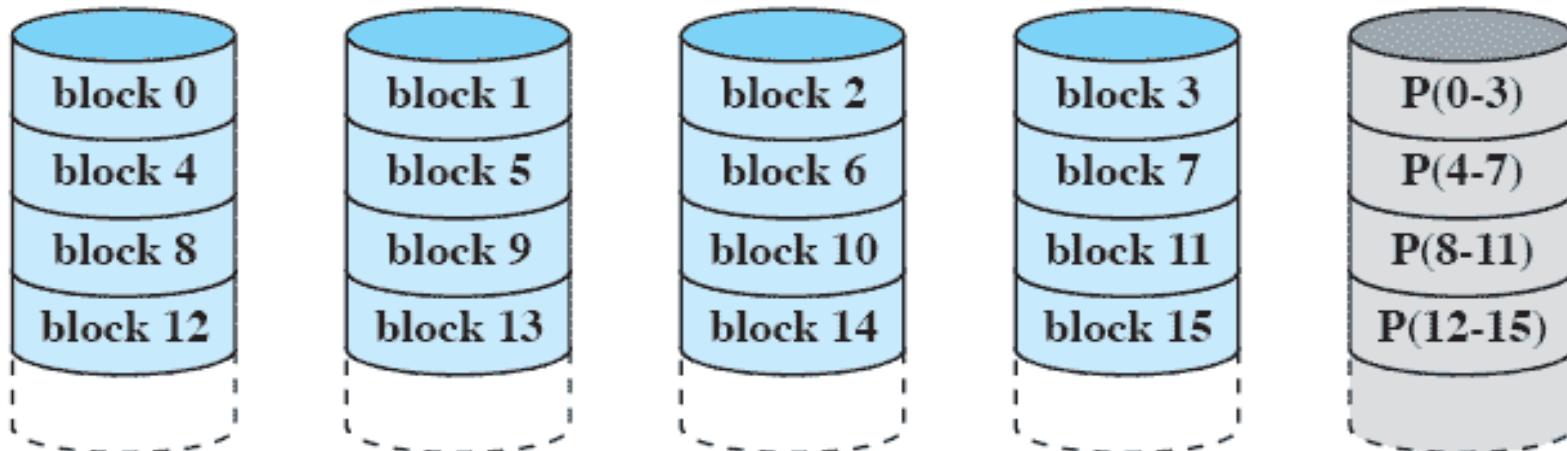
- Similar to Level2, but requires only a single redundant disk, no matter how large the disk array
- Employs parallel access, with data distributed in small strips
- Can achieve very high data transfer rates, but only one I/O at a time



(d) RAID 3 (bit-interleaved parity)

# RAID Level 4

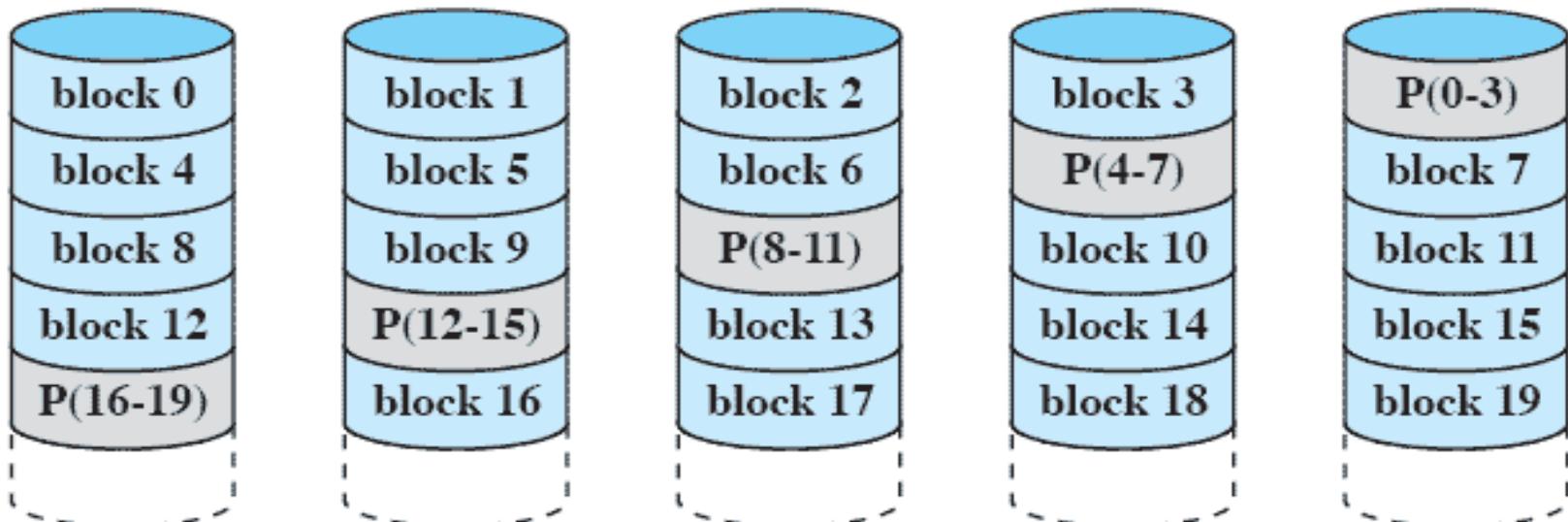
- Makes use of an independent access technique and strips are large.
- A bit-by-bit parity strip is calculated across corresponding strips on each data disk, and the parity bits are stored in the corresponding strip on the parity disk
- Involves a write penalty when an I/O write request of small size is performed



(e) RAID 4 (block-level parity)

# RAID Level 5

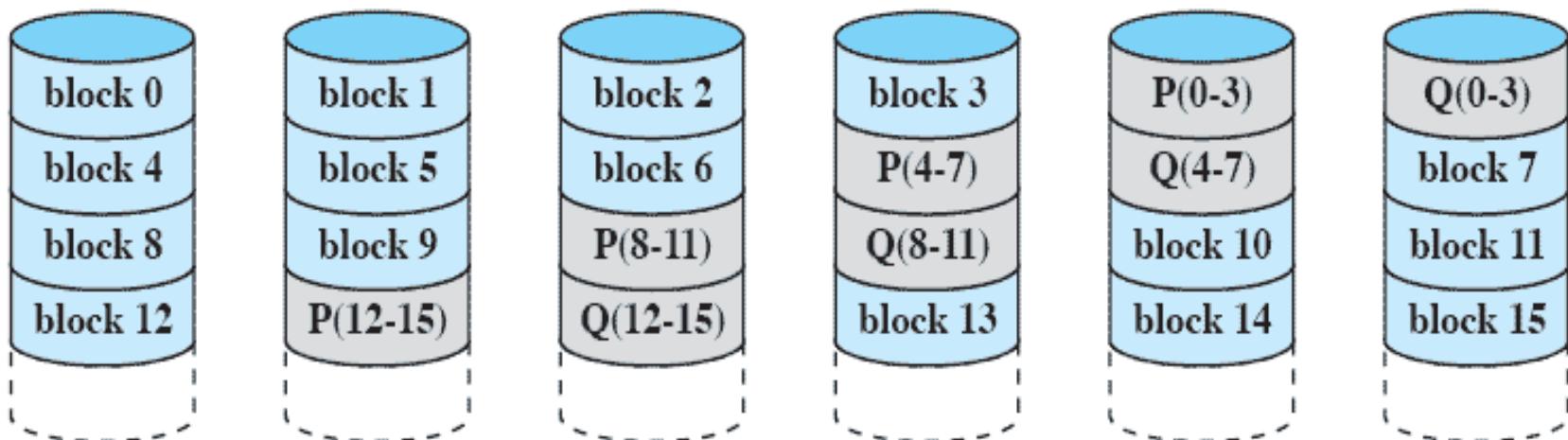
- Similar to RAID-4 but distributes the parity bits across all disks
- Typical allocation is a round-robin scheme
- Has the characteristic that the loss of any one disk does not result in data loss



(f) RAID 5 (block-level distributed parity)

# RAID Level 6

- Two different parity calculations are carried out and stored in separate blocks on different disks
- Provides extremely high data availability
- Incurs a substantial write penalty because each write affects two parity blocks



(g) RAID 6 (dual redundancy)

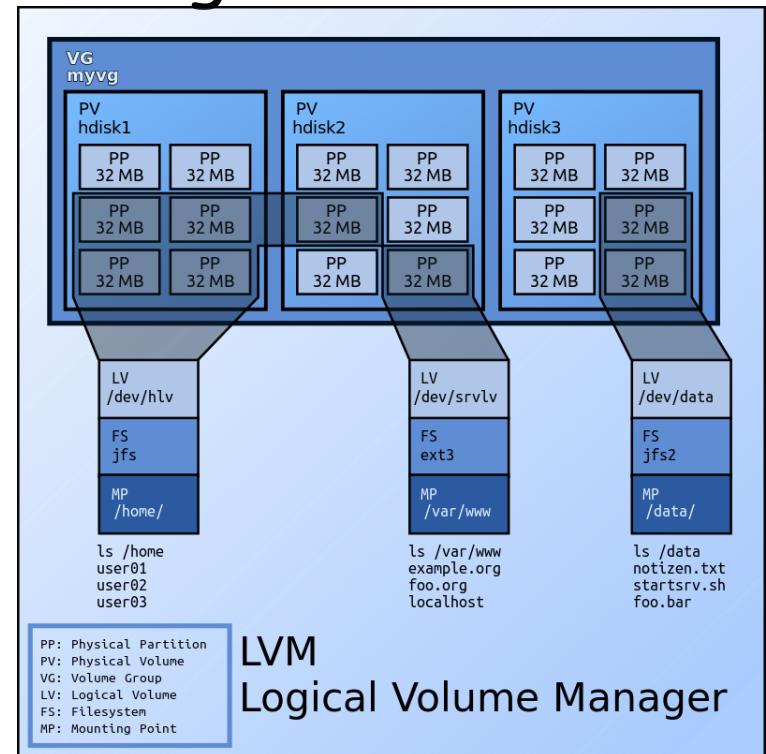
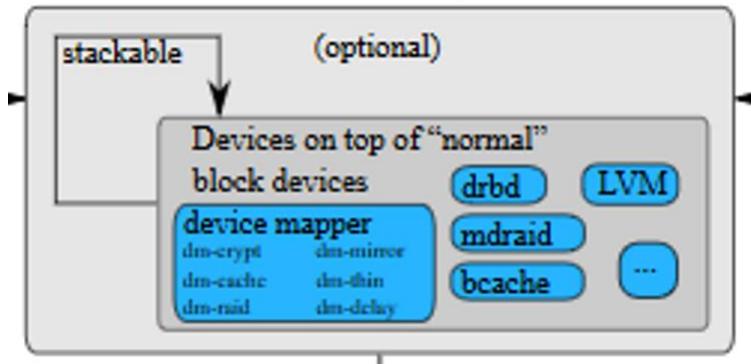
## Table 11.4 RAID Levels

Category	Level	Description	Disk required	Data availability	Large I/O data transfer capacity	Small I/O request rate
Striping	0	Nonredundant	$N$	Lower than single disk	Very high	Very high for both read and write
Mirroring	1	Mirrored	$2N$	Higher than RAID 2, 3, 4, or 5; lower than RAID 6	Higher than single disk for read; similar to single disk for write	Up to twice that of a single disk for read; similar to single disk for write
	2	Redundant via Hamming code	$N + m$	Much higher than single disk; comparable to RAID 3, 4, or 5	Highest of all listed alternatives	Approximately twice that of a single disk
Parallel access	3	Bit-interleaved parity	$N + 1$	Much higher than single disk; comparable to RAID 2, 4, or 5	Highest of all listed alternatives	Approximately twice that of a single disk
	4	Block-interleaved parity	$N + 1$	Much higher than single disk; comparable to RAID 2, 3, or 5	Similar to RAID 0 for read; significantly lower than single disk for write	Similar to RAID 0 for read; significantly lower than single disk for write
	5	Block-interleaved distributed parity	$N + 1$	Much higher than single disk; comparable to RAID 2, 3, or 4	Similar to RAID 0 for read; lower than single disk for write	Similar to RAID 0 for read; generally lower than single disk for write
Independent access	6	Block-interleaved dual distributed parity	$N + 2$	Highest of all listed alternatives	Similar to RAID 0 for read; lower than RAID 5 for write	Similar to RAID 0 for read; significantly lower than RAID 5 for write

$N$  = number of data disks;  $m$  proportional to  $\log N$

# Logical Volume Manager (LVM)

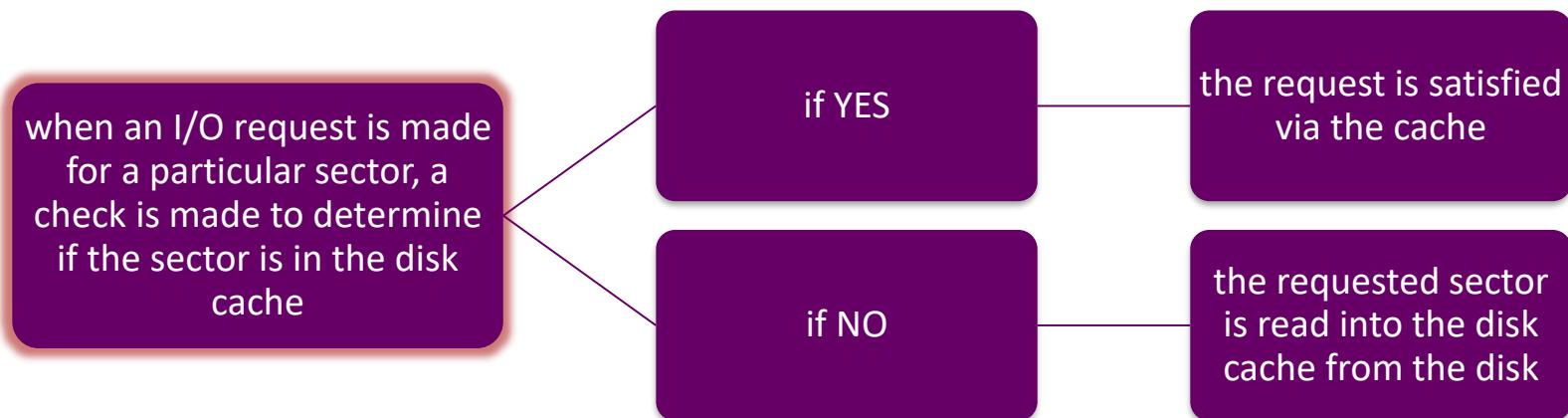
- Physical Volume ≈ HDD or SSD or partition
- LVM allows to combine a set of physical volumes to be treated as a volume group ( via software )
- LVM allows flexible creation of logical volumes (think virtual disks) out of the volume group
- I/O requests are directed to correct physical disk



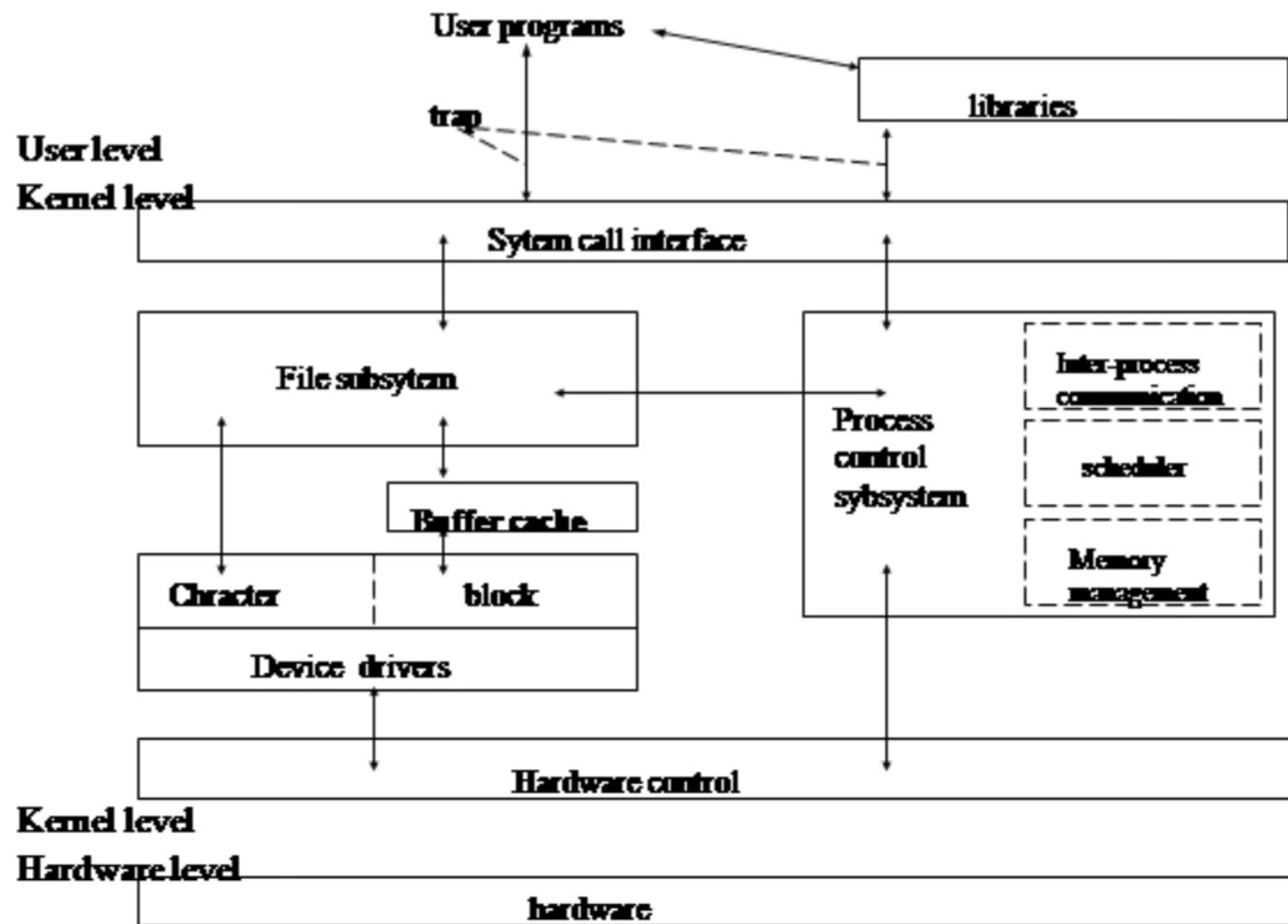
# Other I/O related functions and Technologies

# Disk Cache (aka buffer cache)

- *Cache memory* is used to apply to a memory that is smaller and faster than main memory and that is interposed between main memory and the processor
- Reduces average memory access time by exploiting the principle of locality
- *Disk cache* is a buffer in main memory for disk sectors
- Contains a copy of some of the sectors on the disk



# Disk or Buffer Cache



# Solid State Disks and FLASH Memory

- Solid State Disks ( all electronic , no mechanical parts )
  - Made out of FLASH Memory
  - Genealogy of FLASH
    - RAM, EPROM, EEPROM
      - RAM needs power
      - EPROM needs no power but could be programmed only once
      - EEPROM → erased and programmed, maintain the programmed value without power
      - 'ROM' → Read-Only-Memory: this term is used because although we could read any arbitrary location, entire 'block' needs to be erased at once.
  - Usage of EEPROMs
    - Historically, programs for embedded processors [which needed to be programmed once in a while]
    - Programming an EEPROM was not a time critical event in the past
    - Limitations on the number of times EEPROMs could be programmed ranged from 100s to 1000s and that was not a problem as typical embedded systems were programmed just a dozen times.
  - Evolution of Technology
    - Thousands of EEPROM circuits (called "blocks" in FLASH) are arrayed in order to randomly program and erase blocks

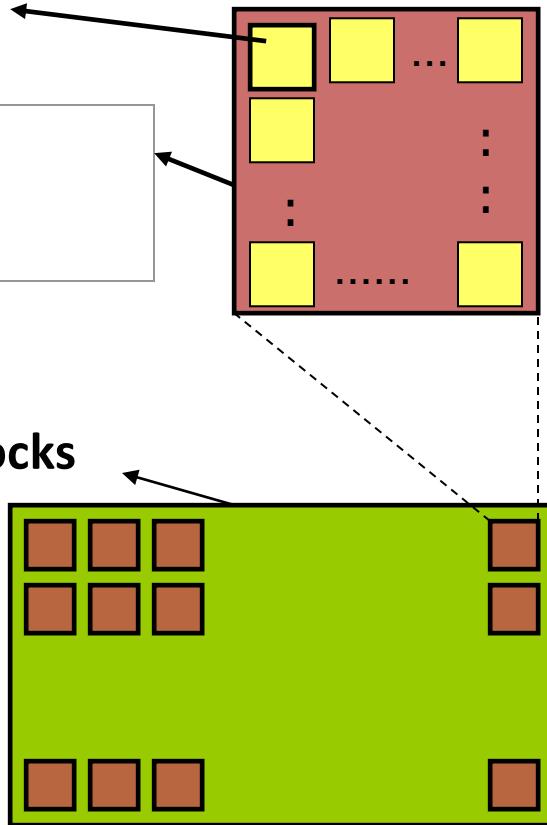
## Organization of a Typical FLASH (Single-Layered) Chip

- 1 Page = 2KB

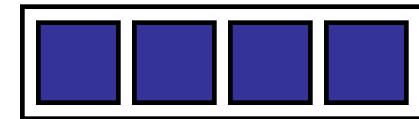
- 1 Block = 128KB  
(64 pages)

- 1 Plane = 2K Blocks  
= 256MB

- 1 Die = 2 Planes  
= 512MB



- 1 Flash Chip = 2GB  
(4 Dies)



- Functions supported
  - Read
  - Erase
  - Program

# FLASH Chip: A few details

- Operations on a FLASH Chip
  - 3 main operations: Read/Erase/Program. **Write = Erase + Program.**
  - Each set of dies in a chip-enable group operate independently as long as the shared data pins are not busy
  - Within a chip-enable group, only one of these operations can take place on a plane
  - Separate operations can take place in separate planes in parallel as long as they are submitted to the chip-enable group at once (this is because once a command is sent to the chip-enable group, it signals busy until the command is complete)
- Types of FLASH Chips
  - SLC: Single Layered Chip
    - Stores 0/1 in each cell [2 voltages]
    - Faster and more reliable
  - MLC: Multi Layered Chip
    - Stores 00/01/10/11 in each cell [4 voltages] → 2 bits per cell
    - Fails 10 times sooner!
    - Cost Advantage of roughly 2 to 1.
  - Manufacturing process almost same for both SLC and MLC.

# Quick Look at various operations on FLASH

	<b>Read</b>	<b>Erase</b>	<b>Program</b>
<b>Granularity</b>	Page	Block	Set of Pages with in a Block (entire Block need not be programmed at once.)
<b>Access</b>	Random	Random blocks within a Chip	<ul style="list-style-type: none"> <li>❑ Any block in a chip can be programmed randomly</li> <li>❑ Set of pages with in a block need to be programmed sequentially</li> </ul>
<b>Time</b>	< 0.1ms	1.5ms	$\geq 0.3\text{ms}$
<b>'Wear Out'?</b>	No	Yes	No (but you can program only once before erase)
<b>Bit Change</b>	-	Changes all Bits $\rightarrow$ '1'	Only op to change Bits $\rightarrow$ '0'
<b>Other Notes</b>		<ul style="list-style-type: none"> <li>❑ 'Stresses' the block and will eventually cause it to fail. SLC Chips: 98% of blocks will last at least 100,000 erase cycles</li> <li>❑ Large systems have 'wear-leveling' algorithms built in so that 'system write' has high endurance than that of write to each cell</li> </ul>	

## Example: Sample Write Operation (source: Texas Memory Systems 2015)

- Writing one 8KB random write would require the following steps:

<b>Step 1</b>	Read 128KB (1 block)	<b>0.1ms + transfer time (trsfr time = 128KB/(20MB/s) = 6.25ms)</b>	6.4ms
<b>Step 2</b>	Erase 128KB Block		1.5ms
<b>Step 3</b>	Program 128KB Block (with 120KB of unchanged data and 8KB of new data)	<b>0.3ms + 6.25ms + (?)</b>	6.7ms
<b>Total Time</b>			<b>14.6ms</b>
Data Transfer rate of this FLASH chip ~ 20MB/s			

### Write Performance (of 14-15ms)

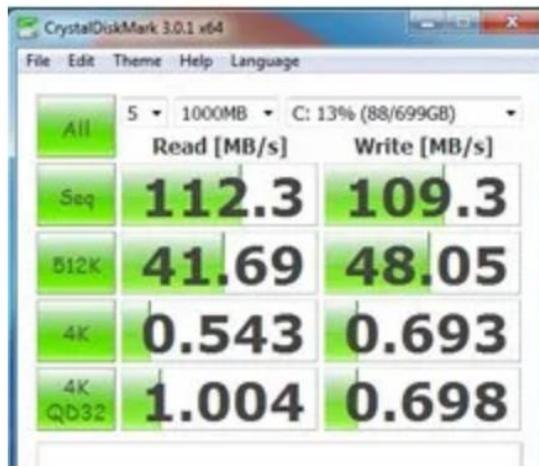
- Acceptable in consumer markets like thumb (USB) drives
- Not Acceptable in Enterprise markets → *one of the reasons for not deploying Flash drives in enterprises*

# Difference between SSDs and traditional Hard Drives

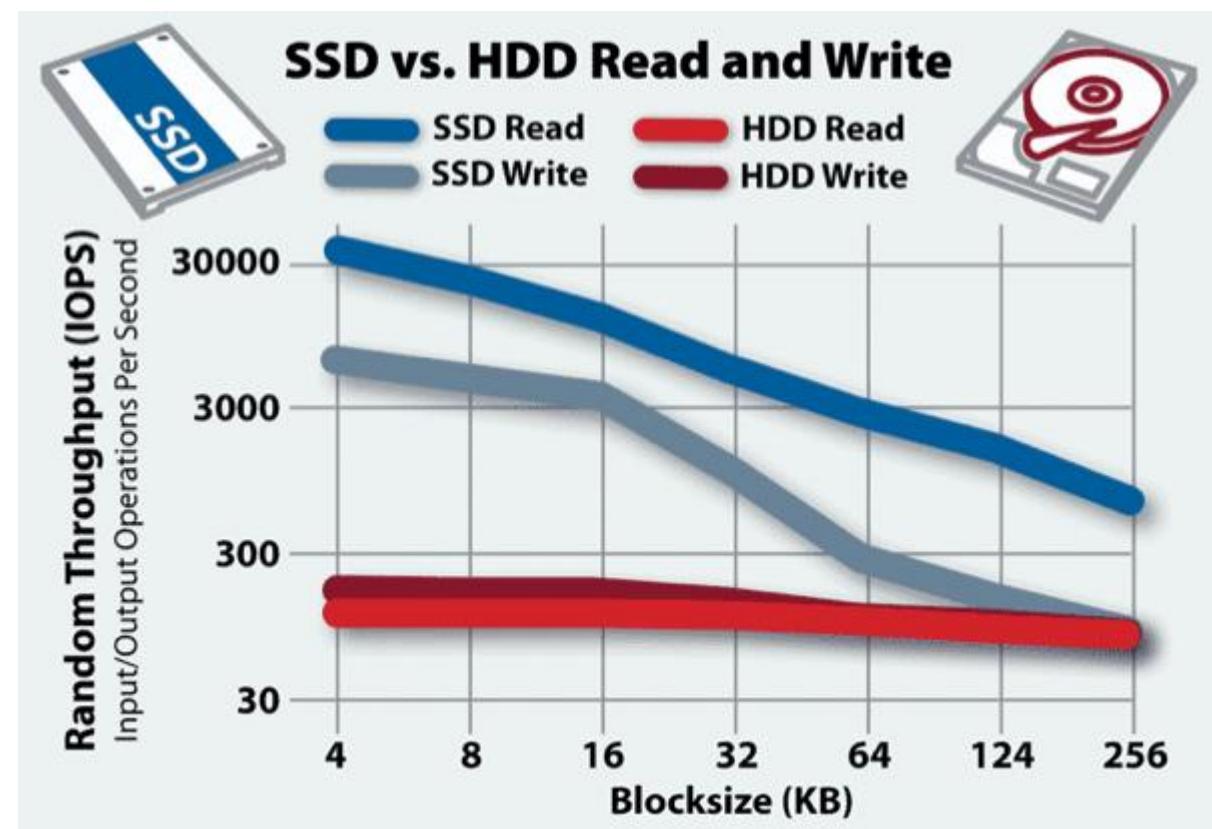
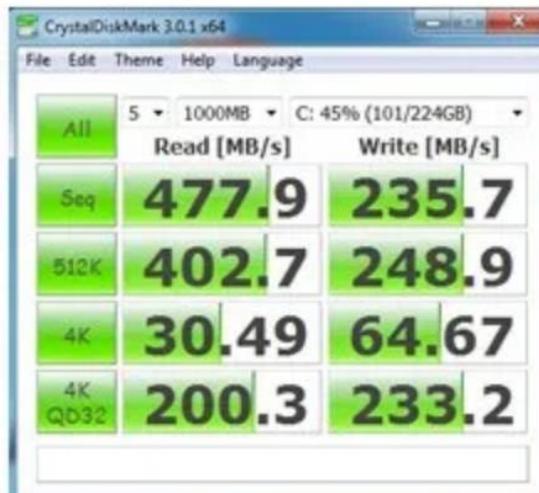
	Hard Drive	Solid State Drive
<b>Construction/ Data Organization</b>	Mechanical Motor/Spindle, Head, Cylinders. Data organized as Tracks and Sectors	FLASH Memory. Constructed using 'blocks'. No spinning. Electronic Reading and Writing.
<b>Reads</b>	<input checked="" type="checkbox"/> Slow Random Reads	Fast Random/Sequential Reads
	<input checked="" type="checkbox"/> Pre-fetching to help Sequential Reads (initiated by OS/present in Disk itself)	—
<b>Writes</b>	Slow Random Writes	Slow Random Writes (slower than current state-of-the-art Hard Disks) → 14-15ms for 128KB.
	Disk Cache/OS Coalescing could help buffer a few writes	Remapping blocks to convert random writes to sequential writes is one trend
<b>Failures</b>	Complete Disk Failure	Block Failure
<b>Recovery/Failure prevention method</b>	Implemented at an array level. RAID is the normal way to re-construct failed disks	Can be done at a disk level or array level <input checked="" type="checkbox"/> Current methods at a disk level
	—	Wear-leveling algorithms to reduce possible failures of blocks
<b>Striping for arrays</b>	Typically Large Stripe Sizes preferred	Smaller Stripe Sizes preferred

# Performance comparison (2018)

## Hard Drive



SSD



# I/O Scheduler Changes

- Since there is no seek time and I/O to any block is equal latency, I/O schedulers optimize for write reduction.
- Imagine writing a multiple consecutive 4KB blocks (since that's what the operating system will assume)
- 1<sup>st</sup> 4KB: read 128K, erase 128K, prog 128K  
2<sup>nd</sup> 4KB: read 128K, erase 128K, prog 128K  
:  
:
- Reduces efficiency. I/O Scheduler should buffer requests for a "bit" assuming there might be subsequent write request.
- Coalesce into a fewer "read/erase/program cycle"

# Current Industry Trends...

- Increasing Performance (Read/Write)
  - Array of FLASH memories to utilize the parallel bandwidth and reduce the latencies
  - A cache (DDR RAM) in front of the array to further minimize the latencies
  - Optimum Stripe Sizes (transfer rate is important)
- Increasing Write Performance
  - Erase as a parallel background operation
- Increasing Endurance
  - A cache (DDR RAM) in front of the array to buffer writes (write-back cache)
  - Remapping for writes
  - Wear-leveling algorithms
  - Disks contain more space than advertised (of the order 20%)
- New technology hitting the market
  - NVMe : allows load/store operations ( Intel Optane )
  - SSD behaves like memory → Storage class memory byte addressable



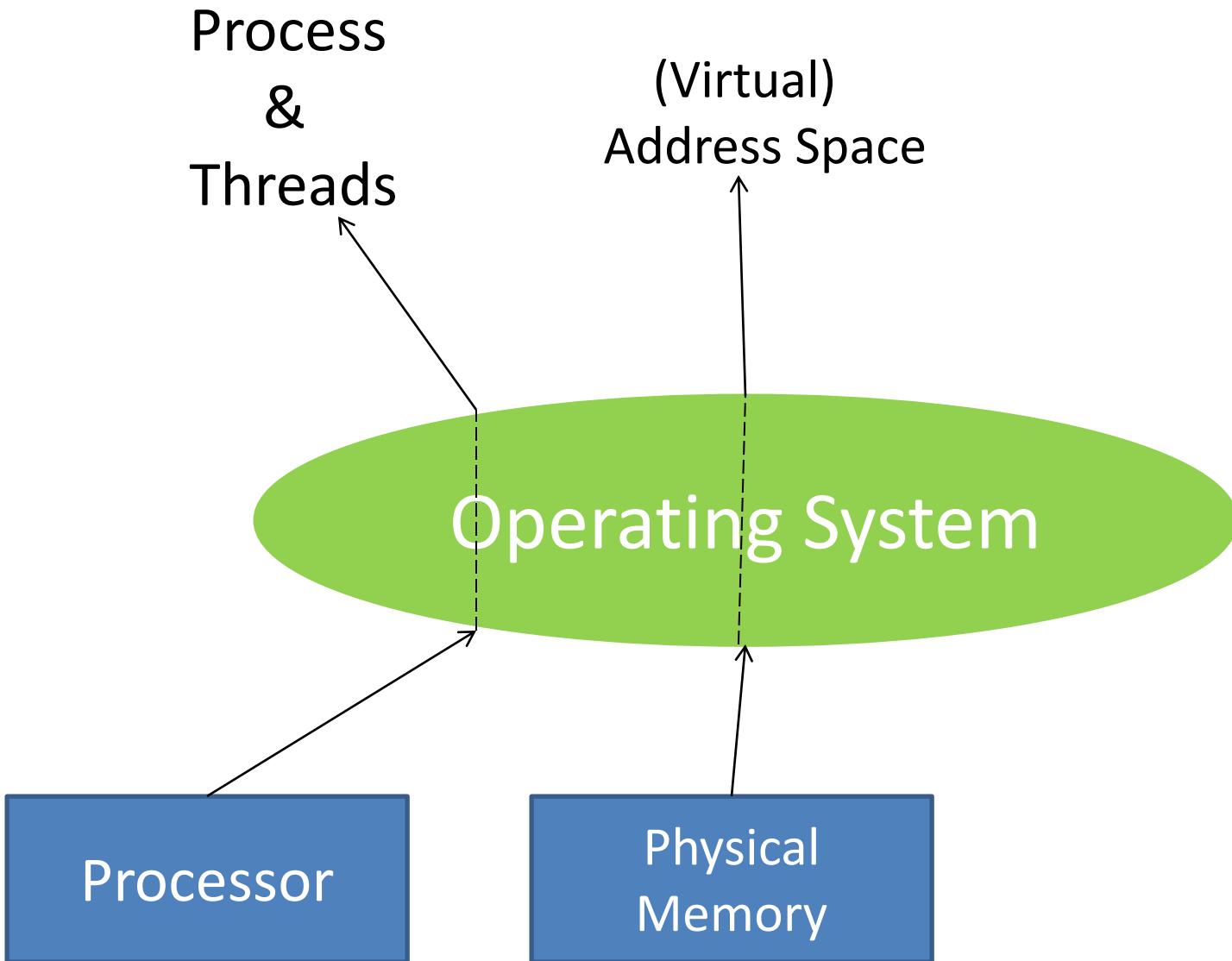
CSCI-GA.2250-001

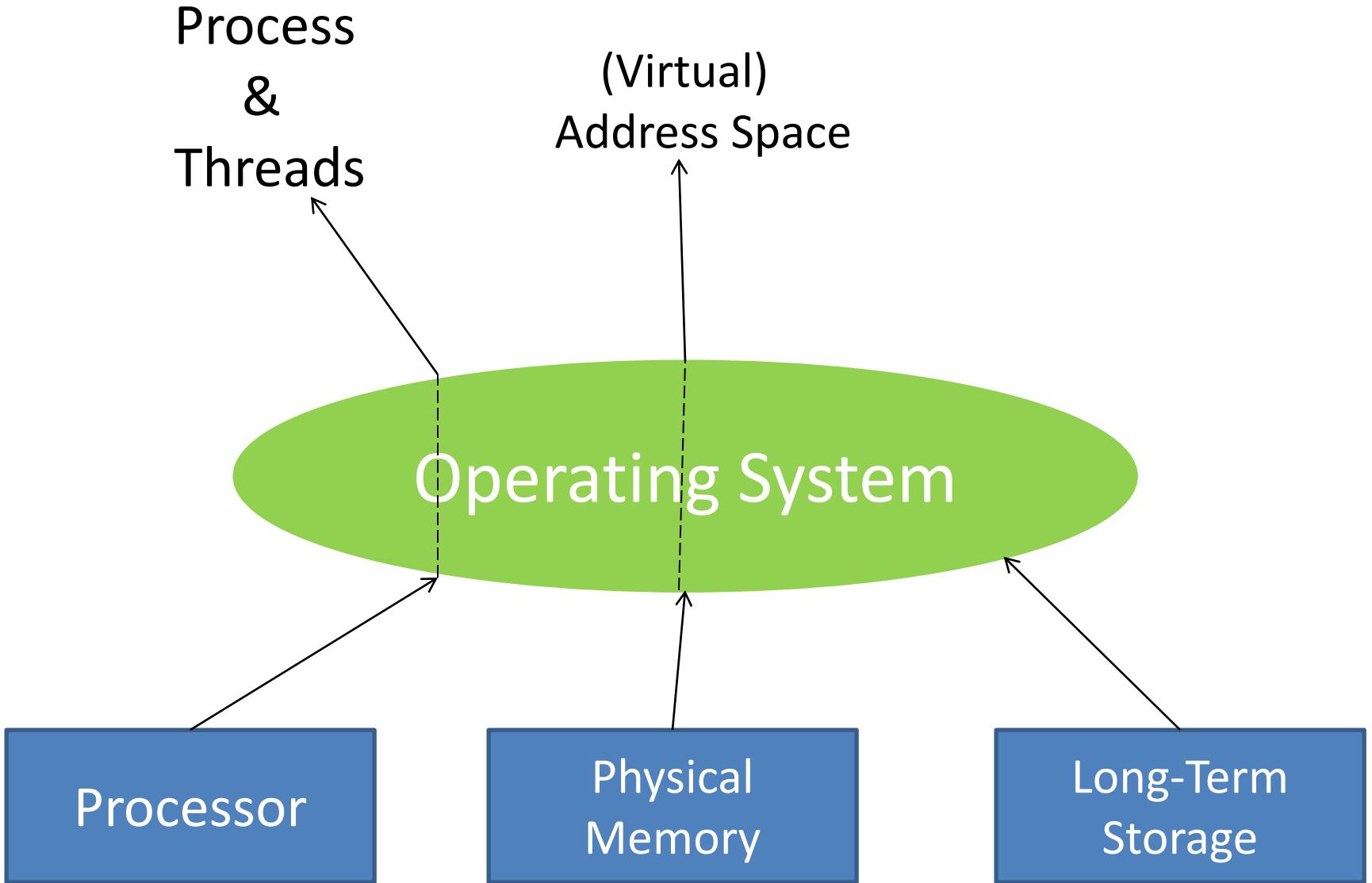
# Operating Systems

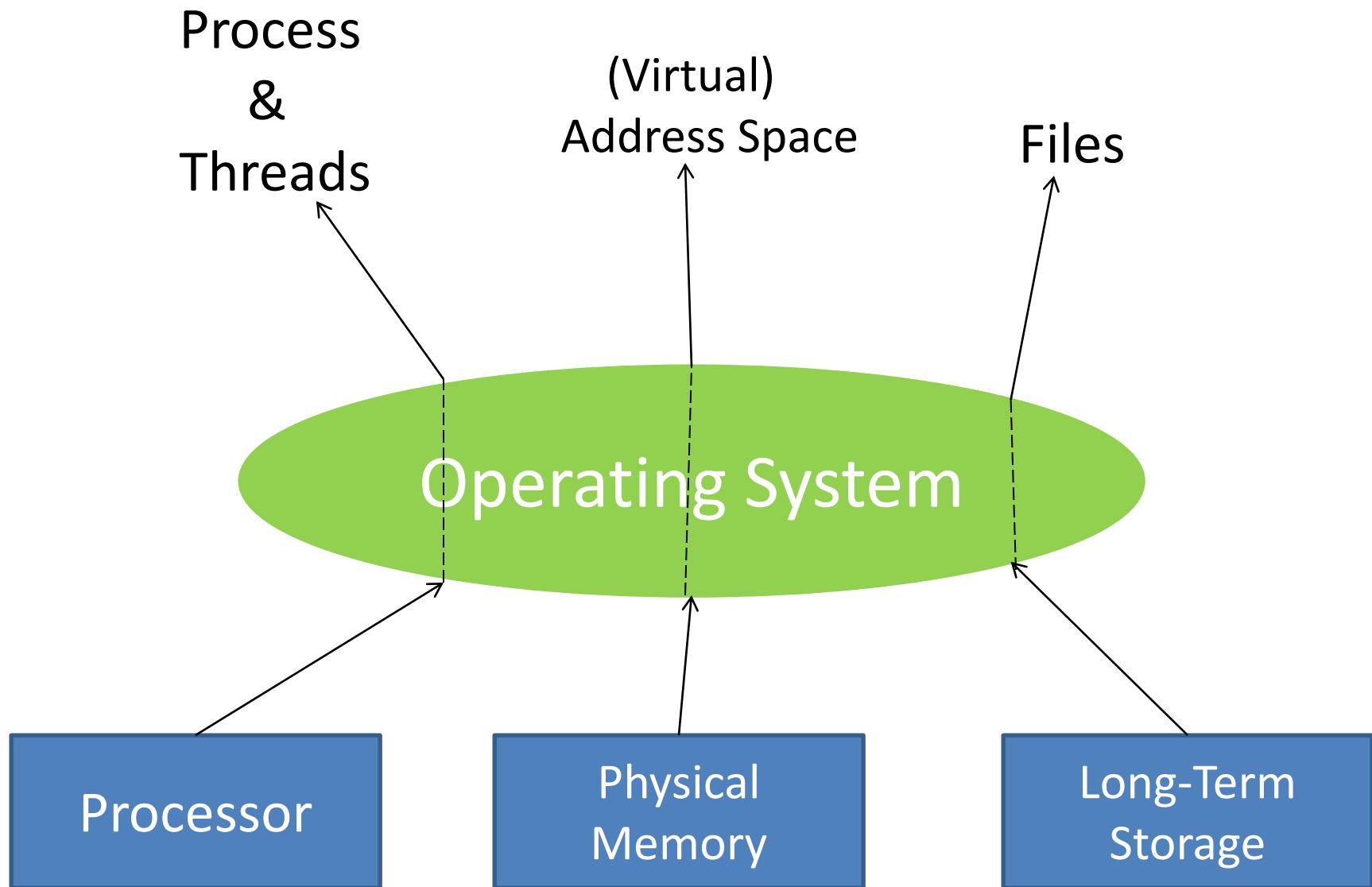
## File Systems

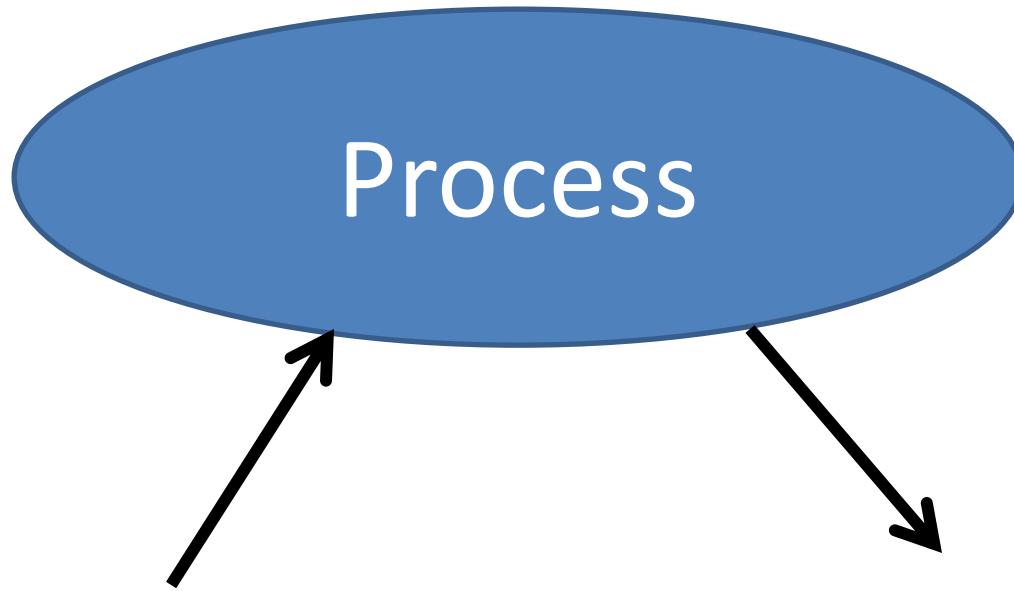
Hubertus Franke  
[frankeh@cs.nyu.edu](mailto:frankeh@cs.nyu.edu)











Is it OK to keep this information only in the process address space?

# Shortcomings of Process Address Space

- Virtual address space may not be enough storage for all information
- Information is lost when process terminates, is killed, or computer crashes.
- Multiple processes may need the information at the same time
- Information might be obtained from 3<sup>rd</sup> sources

# Requirements for Long-term Information Storage

- Store very large amount of information
- Information must survive the termination of the process using it
- Multiple processes must be able to access the information concurrently

# Files

- Data collections created by users
- The File System is one of the most important parts of the OS to a user
- Desirable properties of files:
  - Long-term existence
    - files are stored on disk or other secondary storage and do not disappear when a user logs off
  - Sharable between processes
    - files have names and can have associated access permissions that permit controlled sharing
  - Structure
    - files can be organized into hierarchical or more complex structure to reflect the relationships among files

# Files

- Logical units of information created by processes
- Used to model disks instead of RAM (memory)
- Information stored in files must be **persistent** (i.e. not affected by processes creation and termination)
- Managed by OS
- The part of OS dealing with files is known as the **file system**

# Full Linux I/O Stack

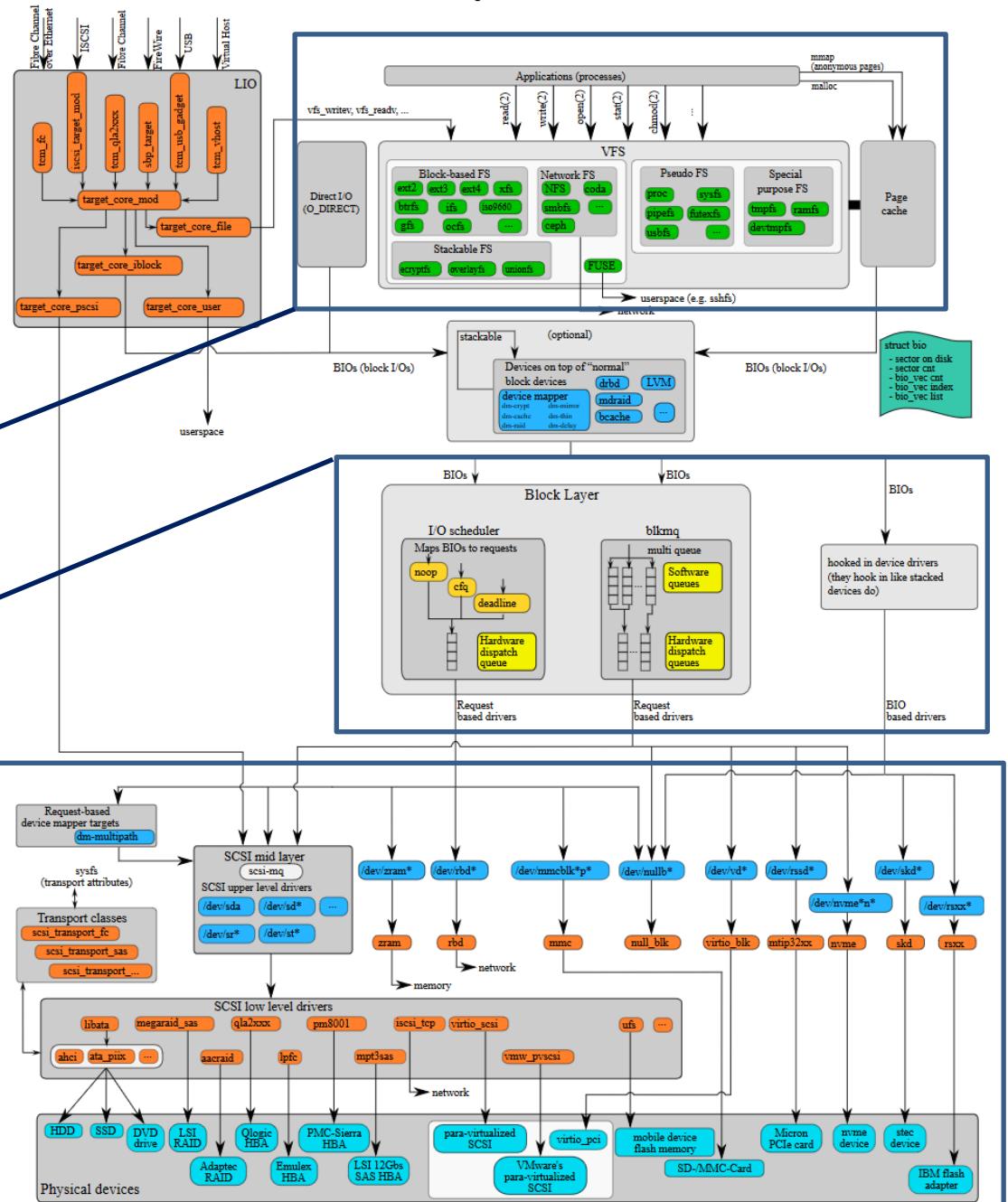
File System

IO Schedulers

Device Drivers

The Linux Storage Stack Diagram

version 4.0, 2015-06-01  
outlines the Linux storage stack as of Kernel version 4.0



*Filesystem maps file content and hierarchy to blocks and I/O requests*

# File Structure ( a walk down memory lane)

From the early days: most of terms related to databases

Four terms are commonly used when discussing files:

Field

Record

File

Database

# Structure Terms

## Field

- collection of related data
- basic element of data
- contains a single value
- fixed or variable length

## Record

- collection of related data
- collection of related fields that can be treated as a unit by some application program
- fixed or variable length



## Database

- collection of related data
- relationships among elements of data are explicit
- designed for use by a number of different applications
- consists of one or more types of files

## File

- collection of similar records
- treated as a single entity
- may be referenced by name
- access control restrictions usually apply at the file level

Example: (1) Telephone Books  
(2) Airlines

# Issues

- How do you find information?
- How do you keep one user from reading another user's data?
- How do you assign files to disk blocks?
- How do you know which disk **blocks** are free?

# Files from The User's point of View

- Files
  - Naming
  - Structure
  - Types
  - Access
  - Attributes
  - Operations
- Directories
  - Single-level
  - Hierarchical
  - Path names
  - Operations

# File Systems

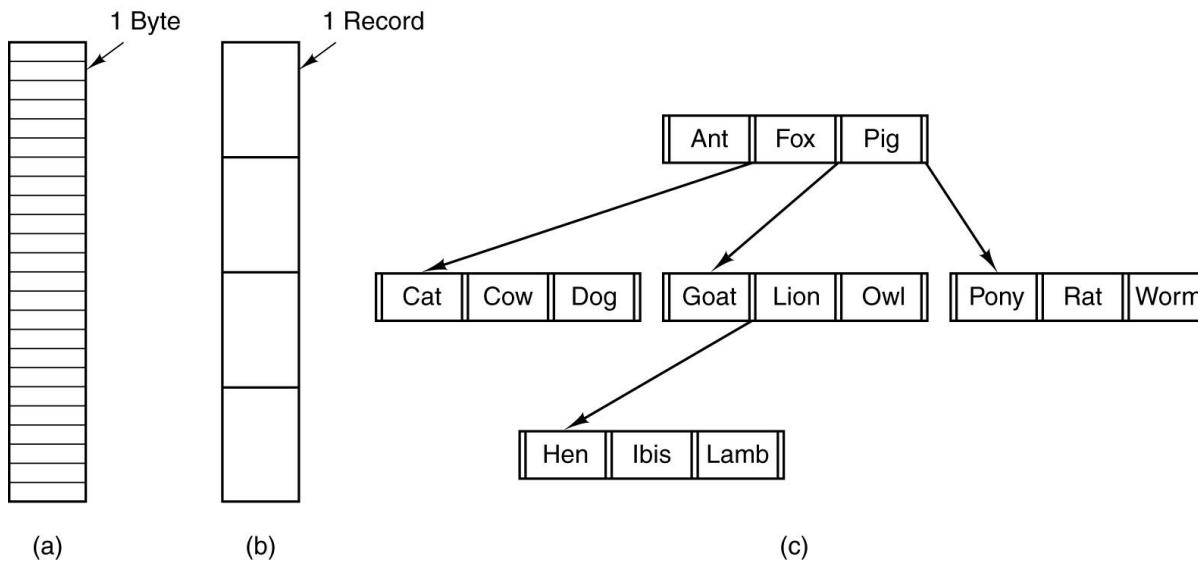
- Provide a means to store data organized as files as well as a collection of functions that can be performed on files
- Maintain a set of attributes associated with the file
- Typical operations include:
  - Create
  - Delete
  - Open
  - Close
  - Read
  - Write

# FILES

## NAMING

- Shields the user from details of file storage.
- In general, files continue to exist even after the process that creates them terminates.

# FILES



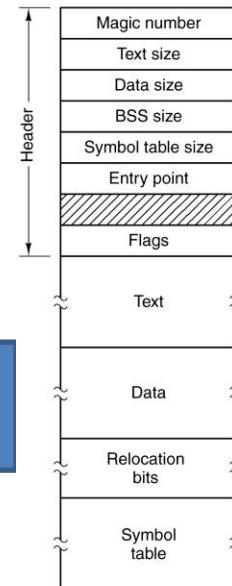
# FILES

Naming

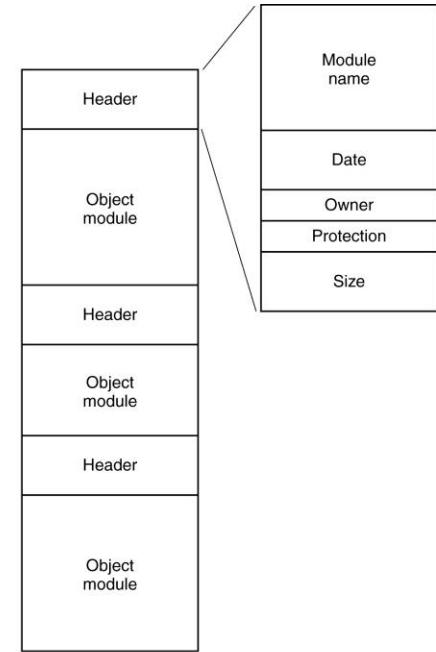
Structure

Types

- Regular files
  - ASCII
  - Binary
- Character special
  - to model serial devices  
(printers, networks, ...)
- Block special
  - to model disks



(a)



(b)

# FILES

Naming

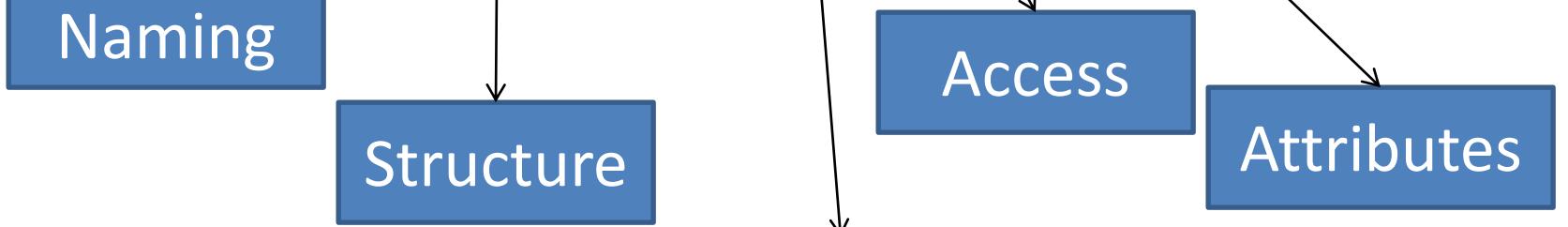
Structure

Access

Types

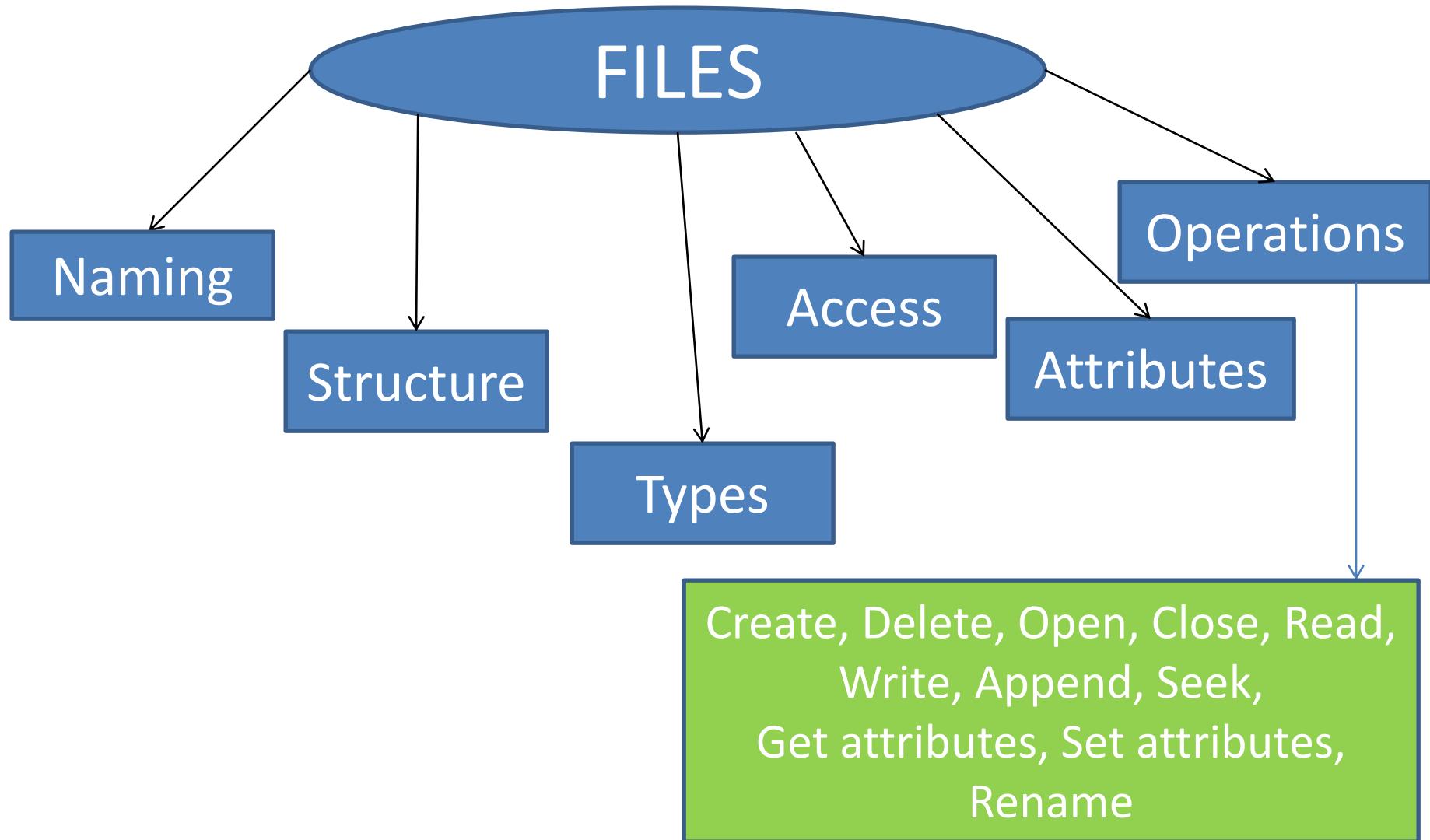
- Sequential
- Random access

# FILES



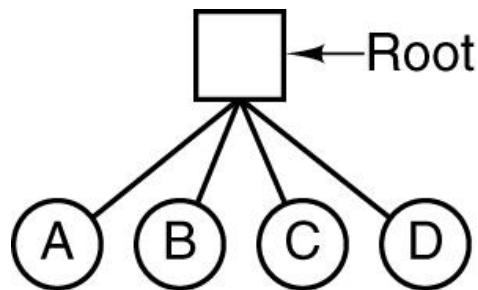
T

Attribute	Meaning
Protection	Who can access the file and in what way
Password	Password needed to access the file
Creator	ID of the person who created the file
Owner	Current owner
Read-only flag	0 for read/write; 1 for read only
Hidden flag	0 for normal; 1 for do not display in listings
System flag	0 for normal files; 1 for system file
Archive flag	0 for has been backed up; 1 for needs to be backed up
ASCII/binary flag	0 for ASCII file; 1 for binary file
Random access flag	0 for sequential access only; 1 for random access
Temporary flag	0 for normal; 1 for delete file on process exit
Lock flags	0 for unlocked; nonzero for locked
Record length	Number of bytes in a record
Key position	Offset of the key within each record
Key length	Number of bytes in the key field
Creation time	Date and time the file was created
Time of last access	Date and time the file was last accessed
Time of last change	Date and time the file was last changed
Current size	Number of bytes in the file
Maximum size	Number of bytes the file may grow to





# Directories: Single-Level Directory Systems



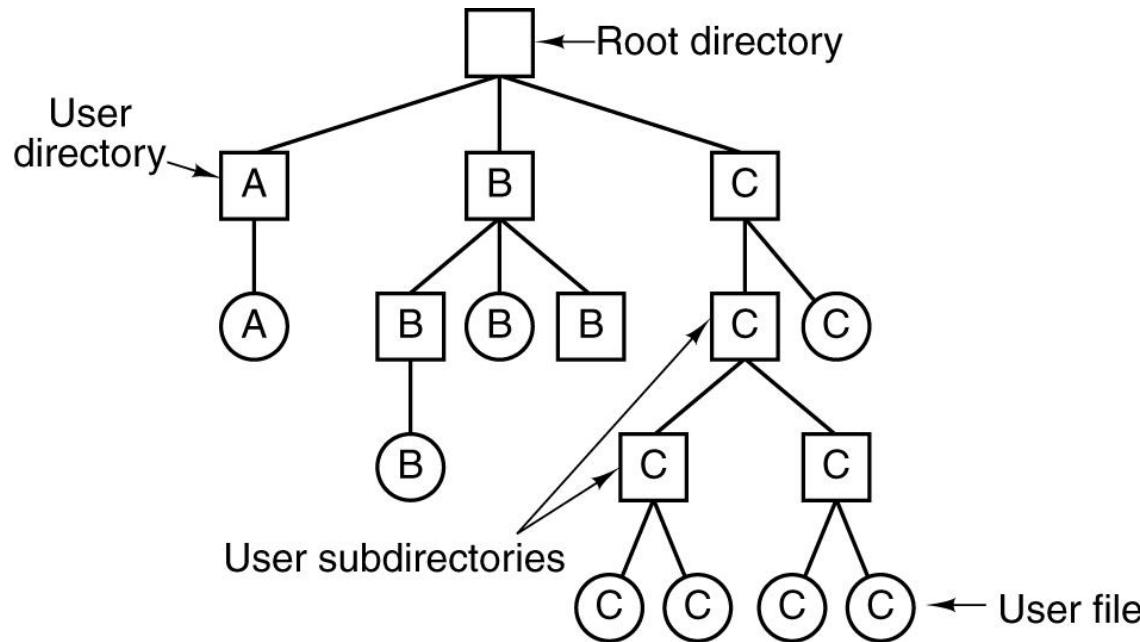
+ Simplicity

-Not adequate for large number  
of files.

Used in simple embedded devices

# Directories: Hierarchical Directory Systems

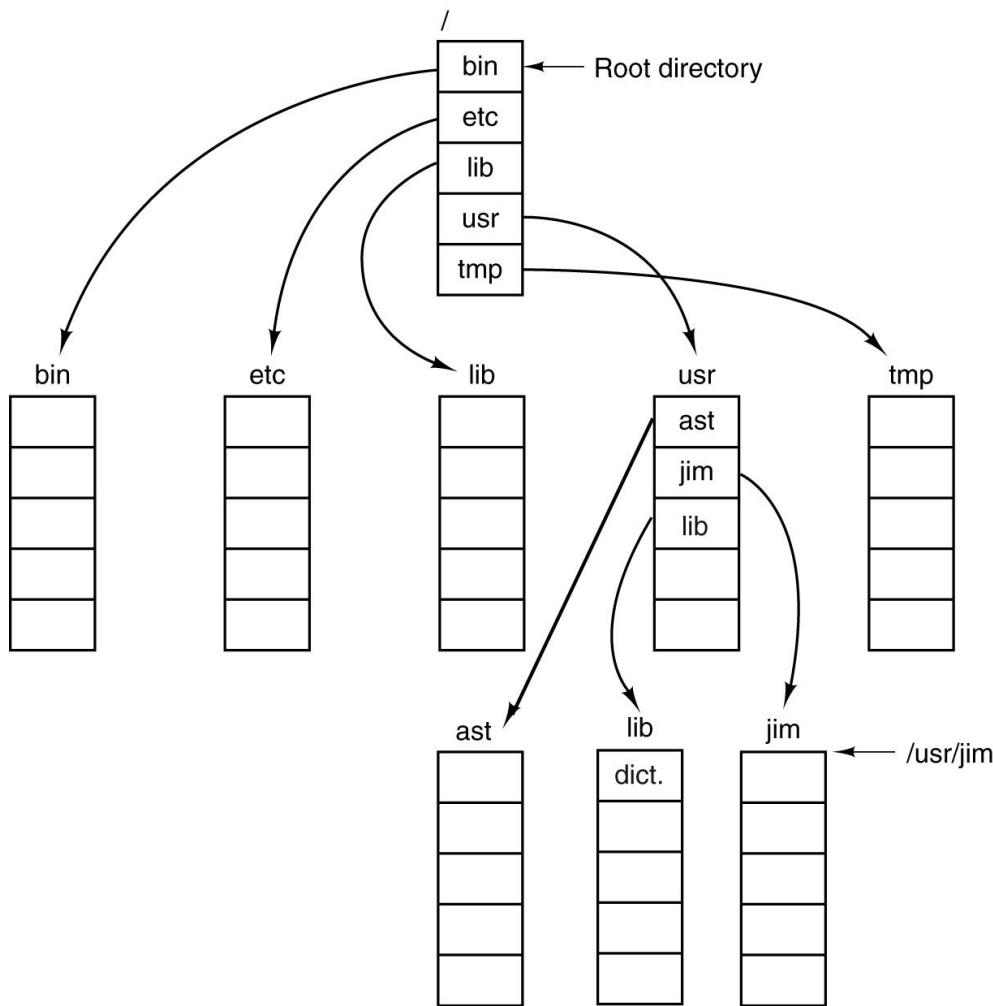
- Group related files together
- Tree of directories



# Directories: Path Names

- Needed when directories are used
- Absolute path names
  - Always start at the root
  - A path from the root to the specified file
  - The first character is the separator
- Relative path names
  - Relative to the **working directory**
  - Each process has its own working directory

# Directories: Path Names



# Directories: Operations

- More variations among OSes than file operations
- Examples (from UNIX):
  - Create, deleted
  - Opendir, closedir
  - Readdir
  - Rename
  - Link, unlink

# Reminder of the hierarchy

## The Linux I/O Stack Diagram

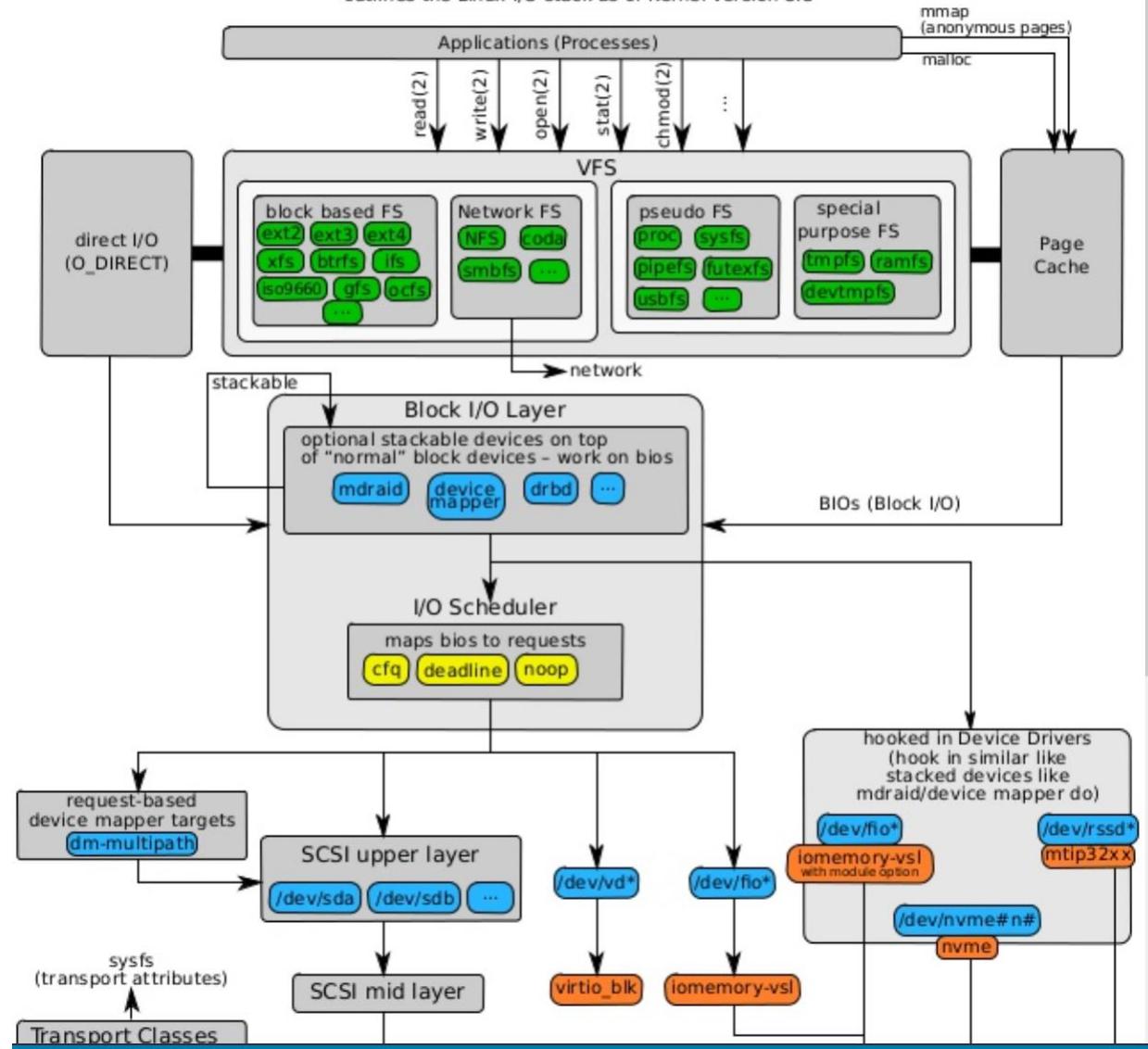
version 1.0, 2012-06-20  
outlines the Linux I/O stack as of Kernel version 3.3

Application

Filesystem

Block I/O

Device Drivers



# Some Example Syscalls

access modes: `O_RDONLY`, `O_WRONLY`, or `O_RDWR`.  
`O_CLOEXEC`, `O_CREAT`, `O_DIRECTORY`, `O_EXCL`,  
`O_NOCTTY`, `O_NOFOLLOW`, `O_TMPFILE`, and `O_TRUNC` ...

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);

int creat(const char *pathname, mode_t mode);
```

0x777 ( rwx , rwx , rwx )

# Intermediate Conclusions

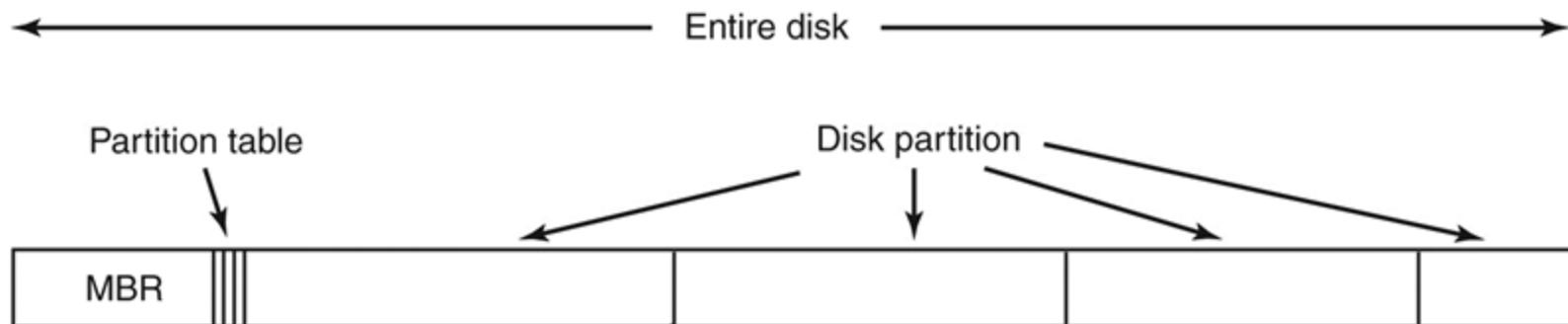
- Files are OS abstraction for storage, same as address space is OS abstraction for physical memory, and processes (& threads) are OS abstraction for CPU.
- So far we discussed files from user perspective.
- Next we will discuss the implementation.

# Questions that need to be answered:

- How does the system boot
- How files and directories are stored ?
- How disk space is managed ?
- How to make everything work efficiently and reliable ?

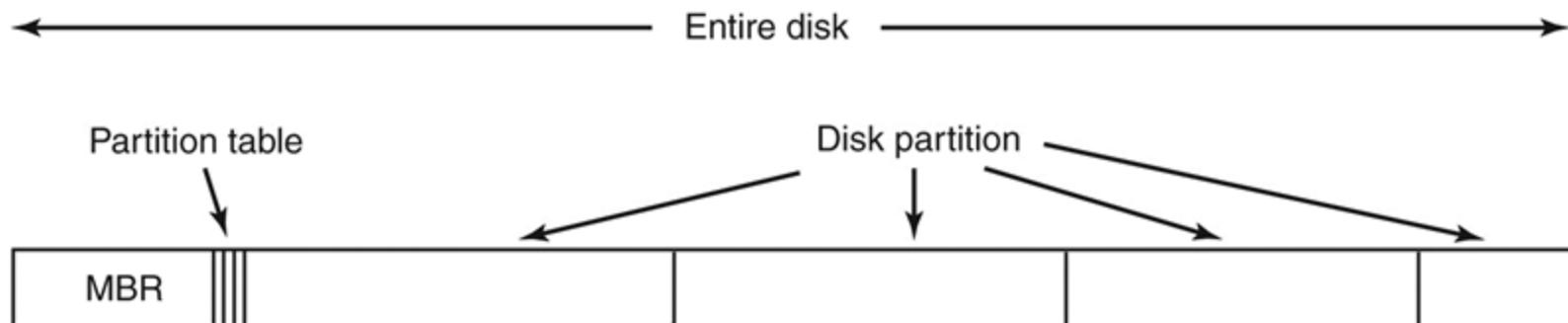
# Disk Layout

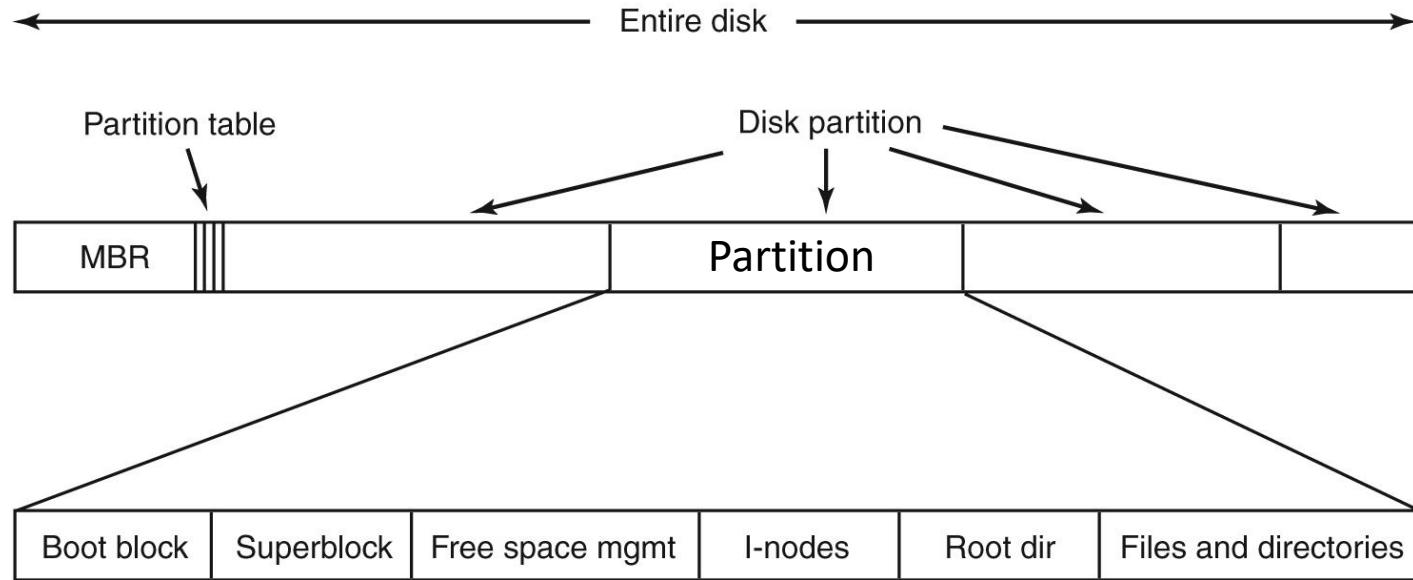
- Stored on disks
- Disks can have partitions with different file systems
- Sector 0 of the disk called **MBR** (Master Boot Record)
- MBR used to boot the computer
- The end of MBR contains the **partition table**

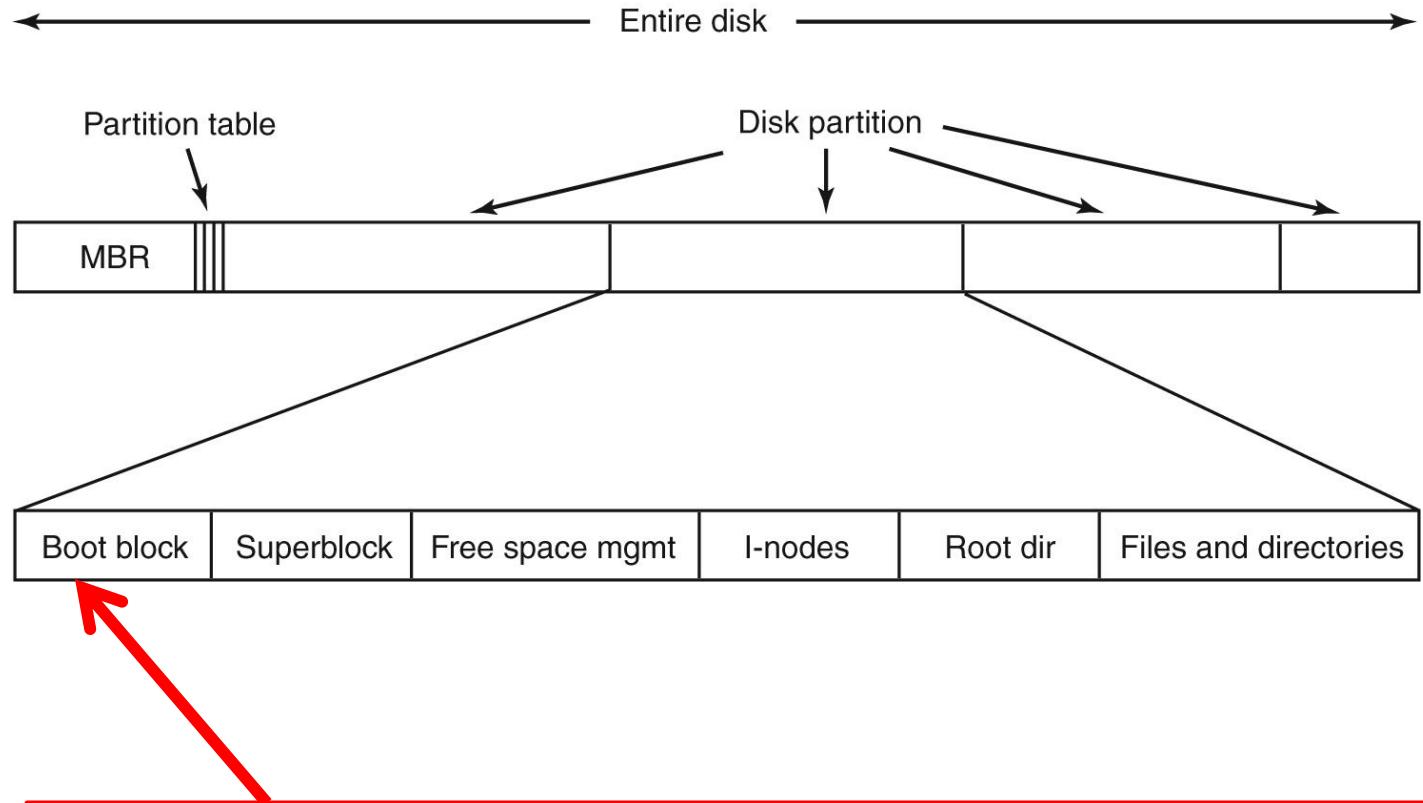


# MBR and Partition Table

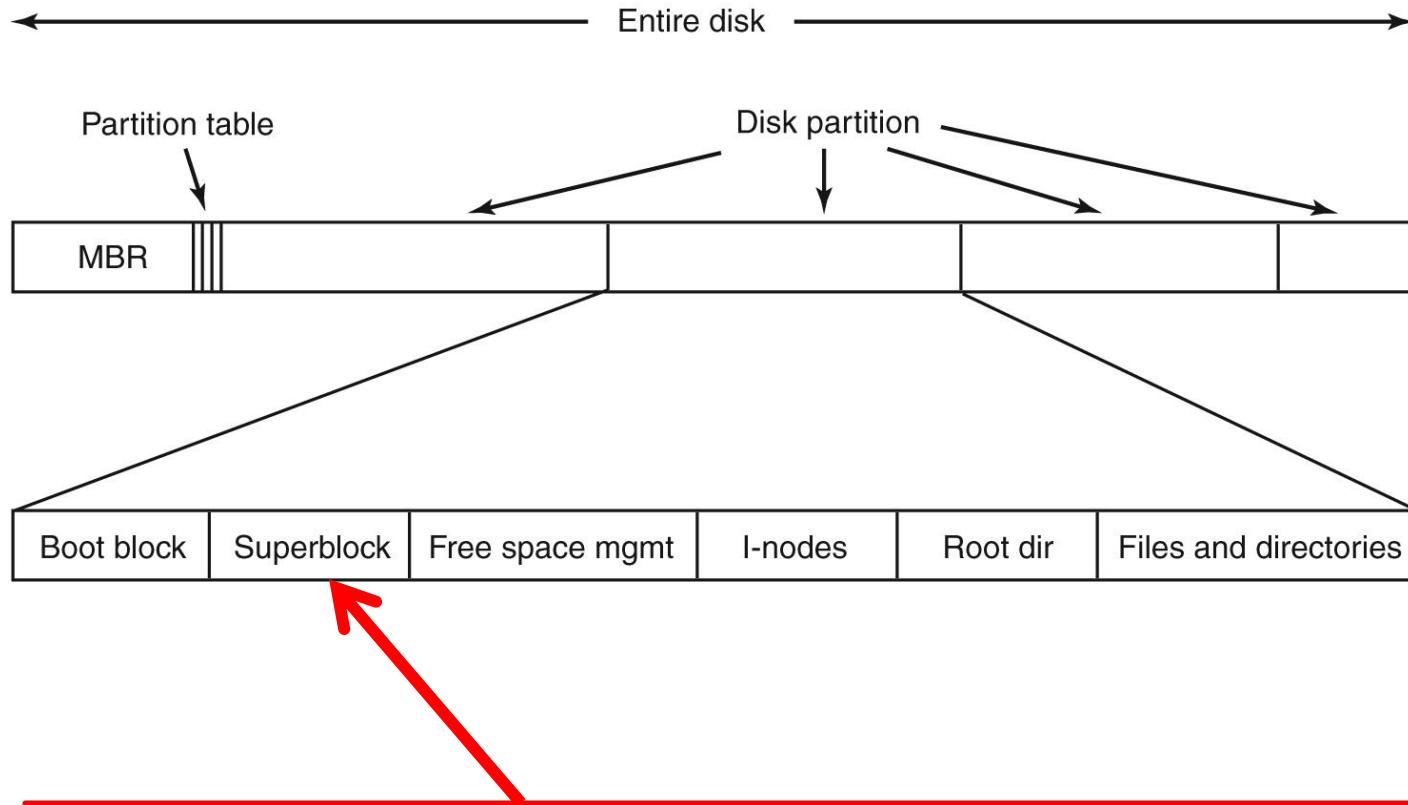
- Gives the starting and ending **addresses** of each partition
- One partition in the table is marked as **active**.
- When the computer is booted, BIOS executes MBR.
- MBR finds the active partition and reads its first block (called **boot block**) and executes it.
- Boot block loads the “OS” contained in that partition.



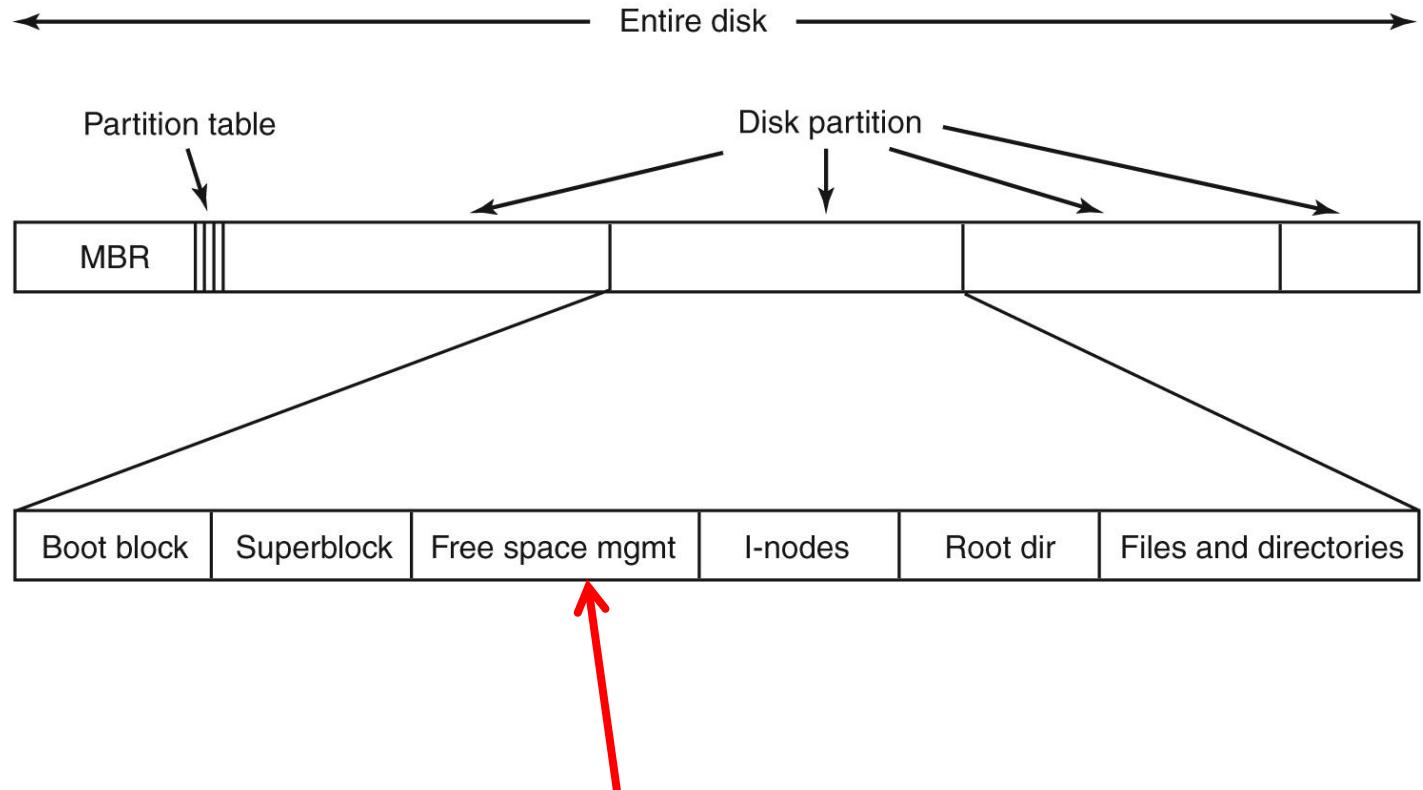




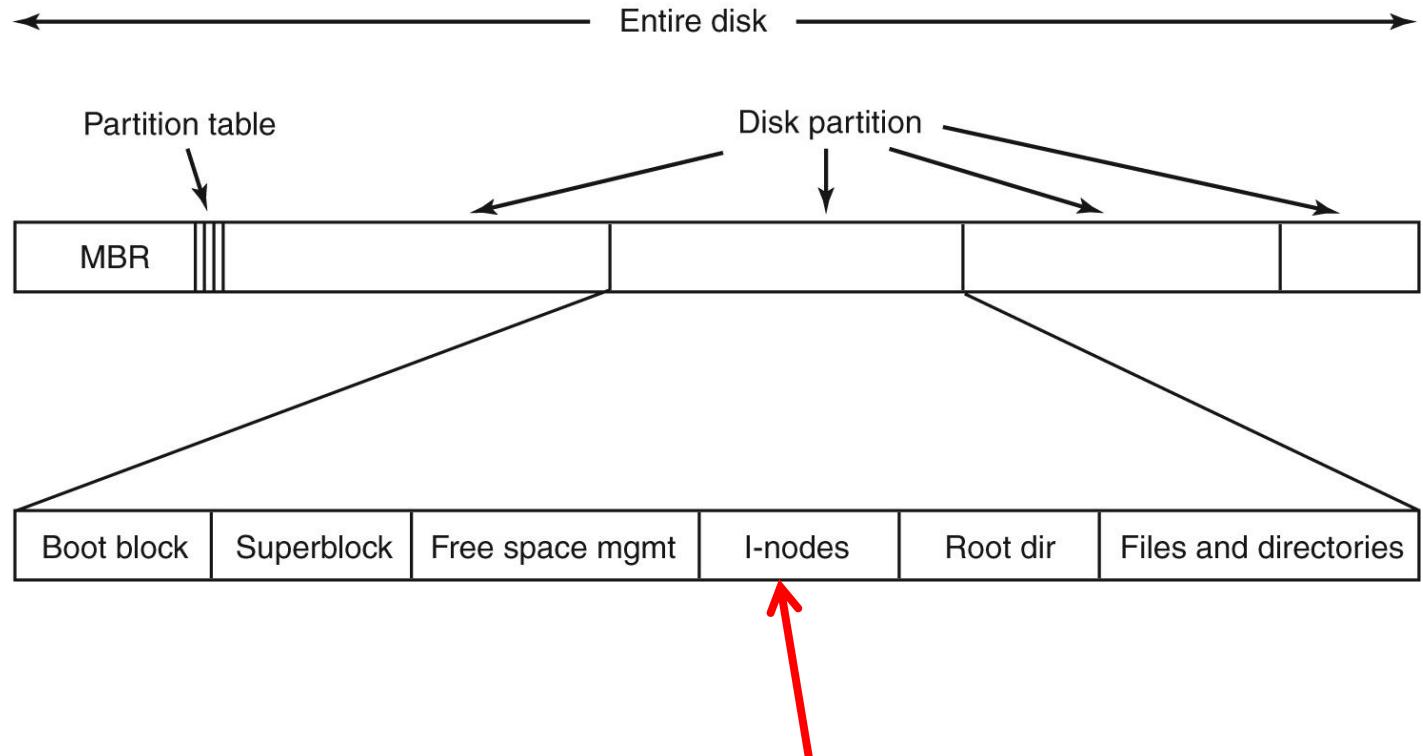
- Contains the bootable code (e.g. operating system)



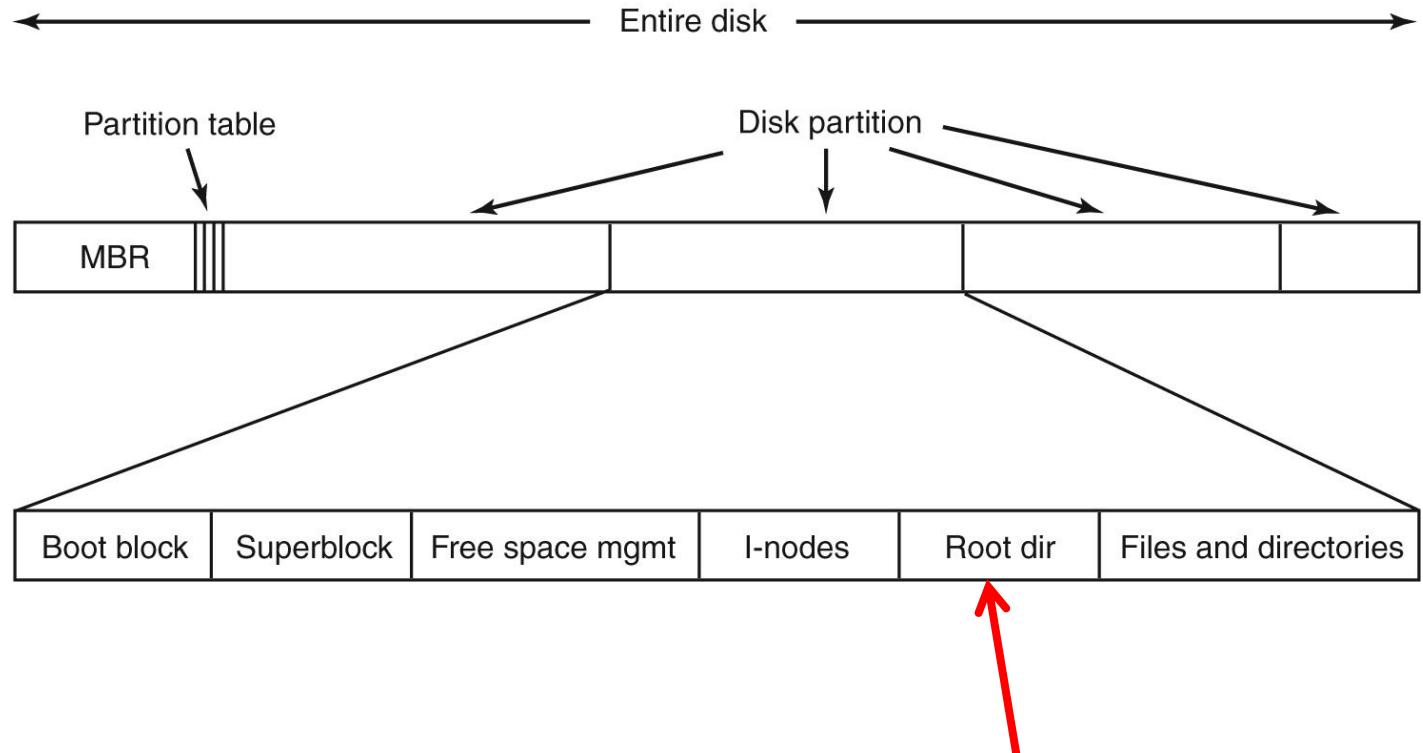
- Contains all key parameters about the file system
  - (e.g. filesystem type (magic, #-blocks, ..))
  - Is read into memory when computer is booted or file system is touched.



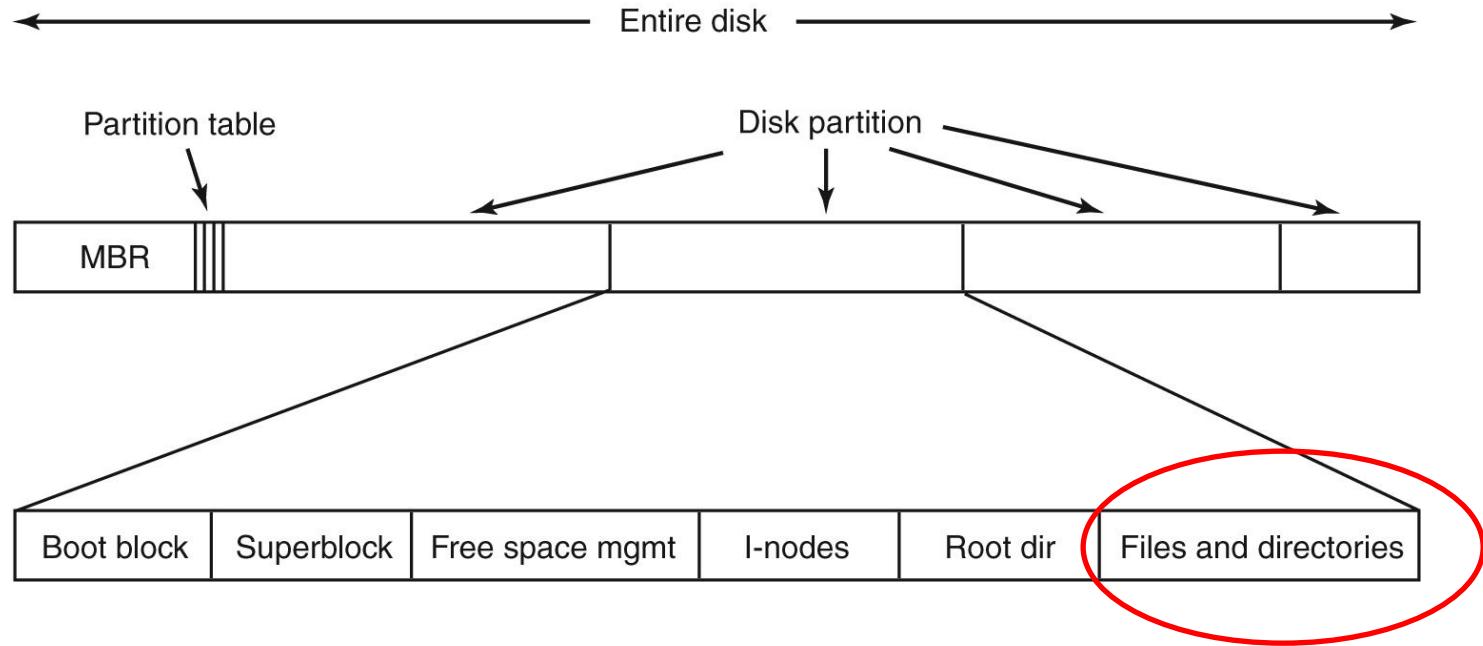
Bitmap or linked list



An array of data structures, one per file, telling about the file



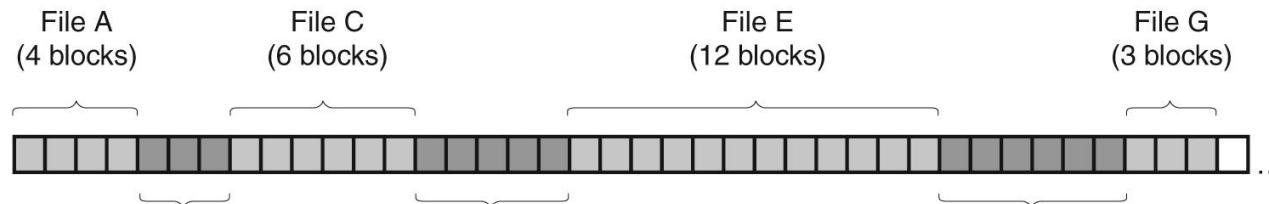
Root Directory .. Think '/' (as in '/home/user/jamesbond')



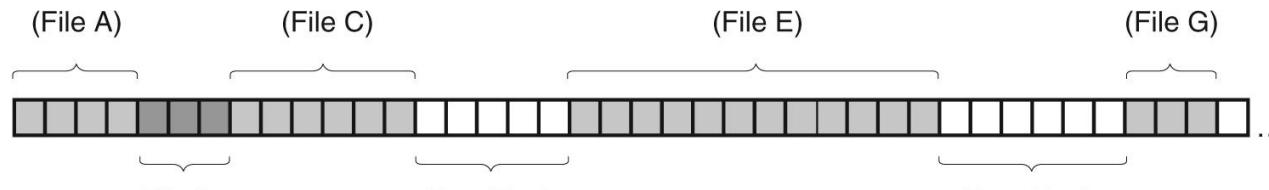
Which disk blocks go with which files?

# Implementing Files: Contiguous Allocation

- Store each file as a contiguous run of disk blocks



(a)



(b)

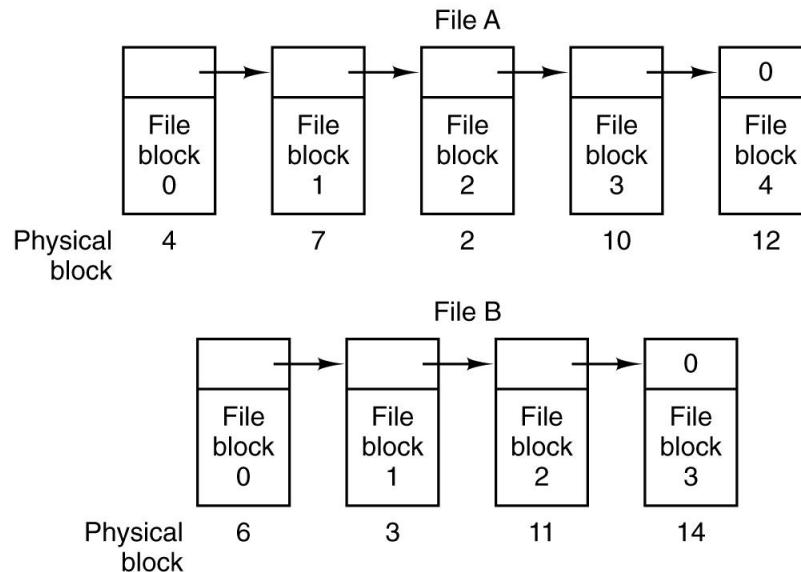
After files D and F were deleted

# Implementing Files: Contiguous Allocation

- + Simple to implement:  
Need to remember starting block address  
of the file and number of blocks
- + Read performance is excellent  
The entire file can be read from disk in a  
single operation.
- Disk becomes fragmented
- Need to know the final size of a file when  
the file is created

# Implementing Files: Linked List Allocation

- Keep a file as a linked list of disk blocks
- The first word of each block is used as a pointer to the next one.
- The rest of the block is for data.

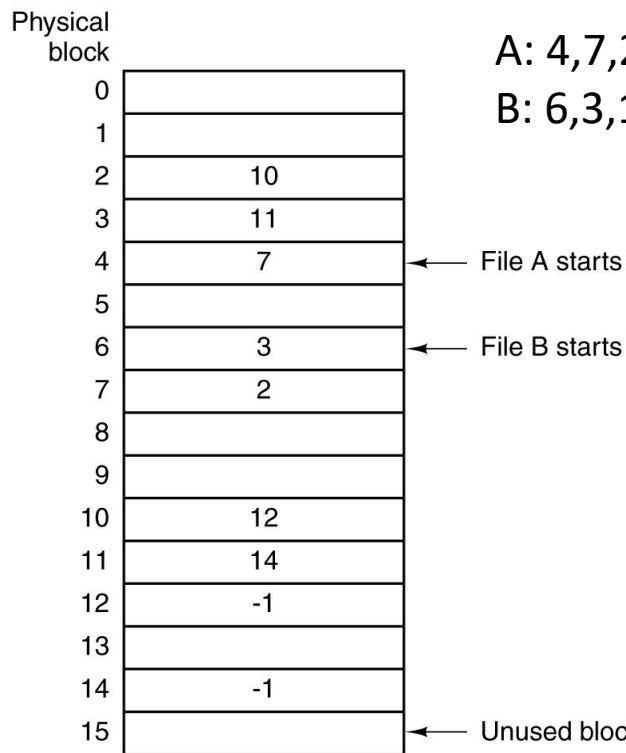


# Implementing Files: Linked List Allocation

- + No (external) fragmentation
- + The directory entry needs just to store the disk address of the first block.
- Random access is extremely slow because we need to start a random request from the beginning half of the time.
- The amount of data storage is no longer a power of two, because the pointer takes up a few bytes (not to bad though)

# Implementing Files: Linked List Allocation Using a Table in Memory

- Take the pointer word from each block and put it in a table in memory.



- This table is called:  
**File Allocation Table (FAT)**
  - Directory entry needs to keep a single integer:  
(the start block number)
- Main drawback: Does not scale to large disks because the table needs to be in memory all the time.**

**FAT12, FAT16, FAT32**

# Limits of FAT

- Limits:

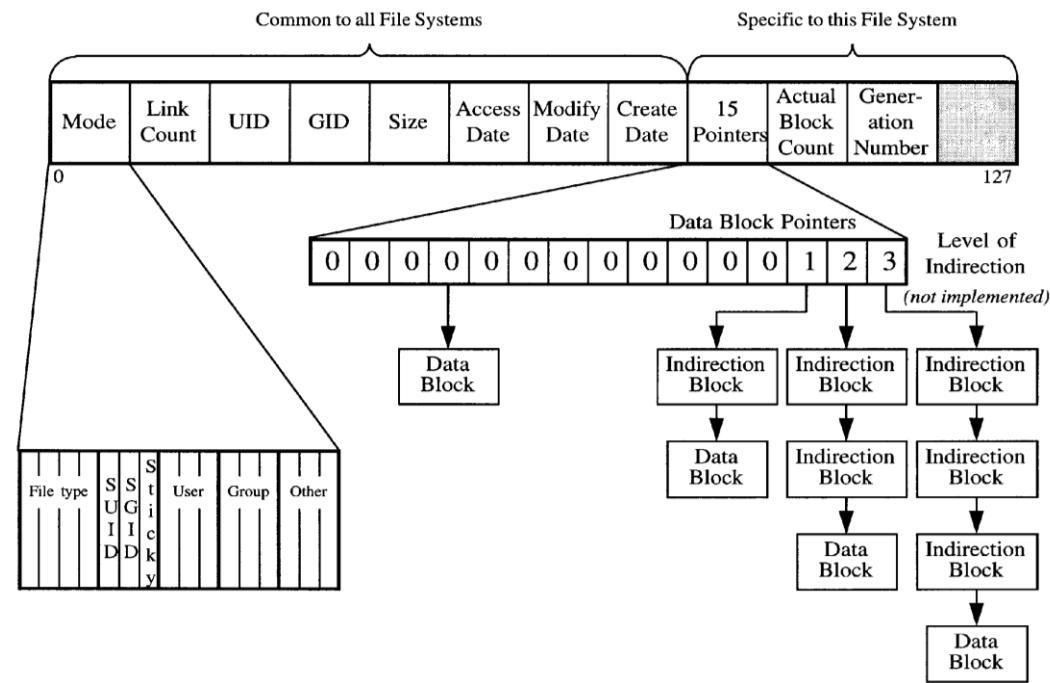
[http://en.wikipedia.org/wiki/Comparison\\_of\\_file\\_systems#cite\\_note-7-14](http://en.wikipedia.org/wiki/Comparison_of_file_systems#cite_note-7-14)

File system	Maximum filename length	Allowable characters in directory entries <sup>[5]</sup>	Maximum pathname length	Maximum file size	Maximum volume size <sup>[6]</sup>
FAT12	8.3 (255 UTF-16 code units with LFN) <sup>[14]</sup>	Any byte except for values 0-31, 127 (DEL) and: " * / : < > ? \   + , . ; = [] (lowercase a-z are stored as A-Z). With VFAT LFN any Unicode except NUL <sup>[14][15]</sup>	No limit defined <sup>[16]</sup>	32 MB (256 MB)	32 MB (256 MB)
FAT16	8.3 (255 UTF-16 code units with LFN) <sup>[14]</sup>	Any byte except for values 0-31, 127 (DEL) and: " * / : < > ? \   + , . ; = [] (lowercase a-z are stored as A-Z). With VFAT LFN any Unicode except NUL <sup>[14][15]</sup>	No limit defined <sup>[16]</sup>	2 GB (4 GB)	2 GB or 4 GB
FAT32	8.3 (255 UTF-16 code units with LFN) <sup>[14]</sup>	Any byte except for values 0-31, 127 (DEL) and: " * / : < > ? \   + , . ; = [] (lowercase a-z are stored as A-Z). With VFAT LFN any Unicode except NUL <sup>[14][15]</sup>	No limit defined <sup>[16]</sup>	4 GB (256 GB <sup>[22]</sup> )	2 TB <sup>[23]</sup> (16 TB)
ext2	255 bytes	Any byte except NUL <sup>[15]</sup> and /	No limit defined <sup>[16]</sup>	2 TB <sup>[6]</sup>	32 TB
ext3	255 bytes	Any byte except NUL <sup>[15]</sup> and /	No limit defined <sup>[16]</sup>	2 TB <sup>[6]</sup>	32 TB
ISO 9660:1988	Level 1: 8.3, Level 2 & 3: ~ 180	Depends on Level <sup>[51]</sup>	~ 180 bytes?	4 GB (Level 1 & 2) to 8 TB (Level 3) <sup>[52]</sup>	8 TB <sup>[53]</sup>
XFS	255 bytes <sup>[57]</sup>	Any byte except NUL <sup>[15]</sup>	No limit defined <sup>[16]</sup>	8 EB <sup>[58]</sup>	8 EB <sup>[58]</sup>
ZFS	255 bytes	Any Unicode except NUL	No limit defined <sup>[16]</sup>	16 EB	16 EB

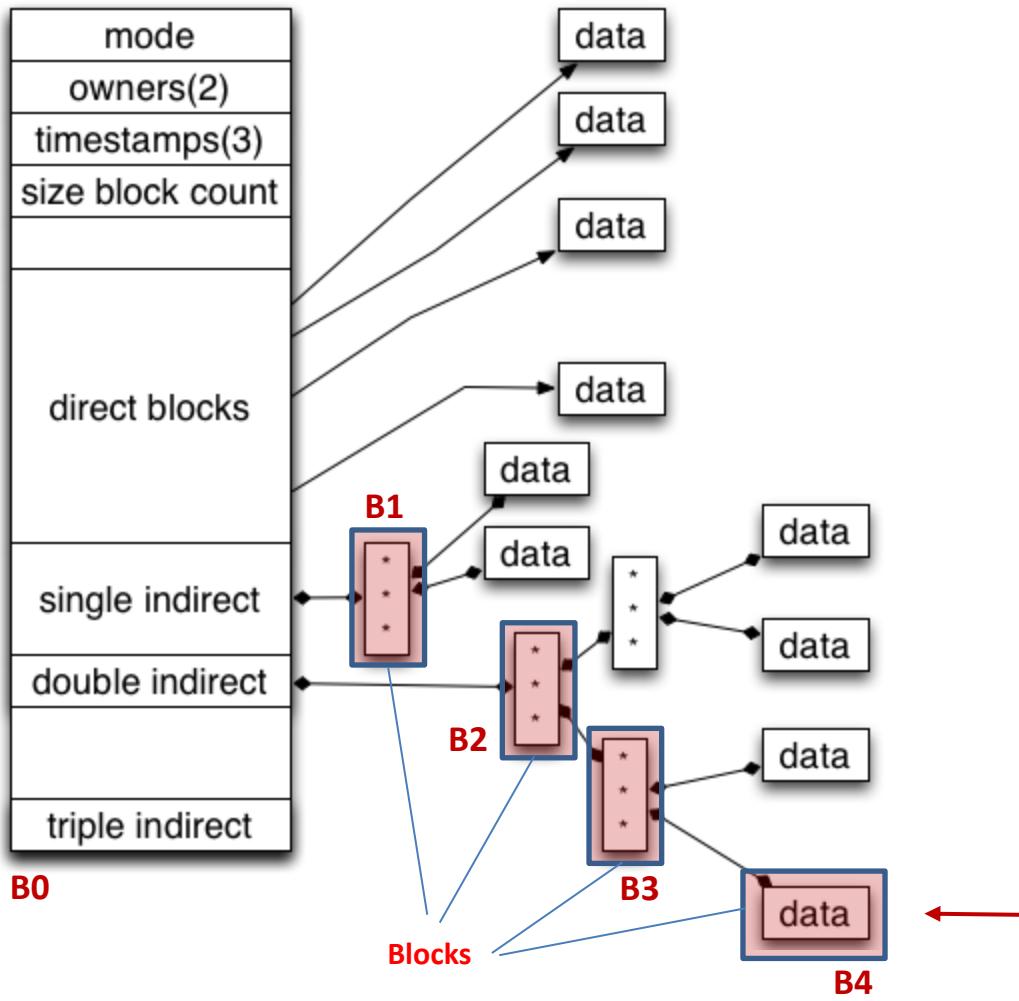
# Implementing Files: I-nodes

- A data structure associated with each file (and directory)
- Lists the attributes and disk addresses of the file blocks
- Need only be in memory when the corresponding file is open
- Aka **FILE META DATA** ( typically 128 bytes )

```
i-node=#
frankeh@frankeh-vb1:~$ ls -li
803294 cloudOE
668584 CodeSourcery
654758 Desktop
654877 Documents
654874 Downloads
659792 DVSDK
668397 examples.desktop
654878 Music
668582 NYU
806338 Papers
654879 Pictures
654876 Public
1064546 SDET
654875 Templates
803184 tmp
654880 Videos
799328 workdir
676014 xyz
frankeh@frankeh-vb1:~$
```



# Implementing Files: i-nodes



Properties:

- Small file access fast
- Everything a block
- Huge files can be presented
- Overhead (access, space) proportional to file size

To get to this data we must read

- 1) inode (B0)
- 2) double indirect block (B2) and (B3)
- 3) data block (B4)

→ if uncached, 4 disk block reads

# Example for File access

- Assume blksz=1K ( $2^{10}$ ) and block number a 4 byte integer so the data is stored in 1 KB blocks and  $256$  ( $=2^8$ ) block id's can fit in an indirection block

How much data range is covered at each level ??

- Direct Access (12 entries) cover  $12 * 2^{10}$  = 12KB
- Indirect:  $2^8 * 2^{10}$  =  $2^{18}$  = 256KB
- Double Indirect:  $2^8 * 2^8 * 2^{10}$  =  $2^{26}$  = 64MB
- Triple Indirect:  $2^8 * 2^8 * 2^8 * 2^{10}$  =  $2^{34}$  = 16GB
- Quad Indirection:  $2^8 * 2^8 * 2^8 * 2^8 * 2^{10}$  =  $2^{42}$  = 4TB
- Then: fileofs=260KB is covered by Indirect as it covers 12KB - 268KB
- Note: the math is different if you go to quad pointers (only 11 direct then) or if blksz is different.

# syscalls to retrieve meta data

```
int stat(const char *pathname, struct stat *statbuf);
int fstat(int fd, struct stat *statbuf);
int lstat(const char *pathname, struct stat *statbuf);
```

ls -ls translates to

```
struct stat {
    dev_t      st_dev;          /* ID of device containing file */
    ino_t      st_ino;          /* Inode number */
    mode_t     st_mode;         /* File type and mode */
    nlink_t    st_nlink;        /* Number of hard links */
    uid_t      st_uid;          /* User ID of owner */
    gid_t      st_gid;          /* Group ID of owner */
    dev_t      st_rdev;         /* Device ID (if special file) */
    off_t      st_size;         /* Total size, in bytes */
    blksize_t  st_blksize;      /* Block size for filesystem I/O */
    blkcnt_t   st_blocks;       /* Number of 512B blocks allocated */

    /* Since Linux 2.6, the kernel supports nanosecond
       precision for the following timestamp fields.
       For the details before Linux 2.6, see NOTES. */

    struct timespec st_atim;    /* Time of last access */
    struct timespec st_mtim;    /* Time of last modification */
    struct timespec st_ctim;    /* Time of last status change */

#define st_atime st_atim.tv_sec      /* Backward compatibility */
#define st_mtime st_mtim.tv_sec
#define st_ctime st_ctim.tv_sec
};
```

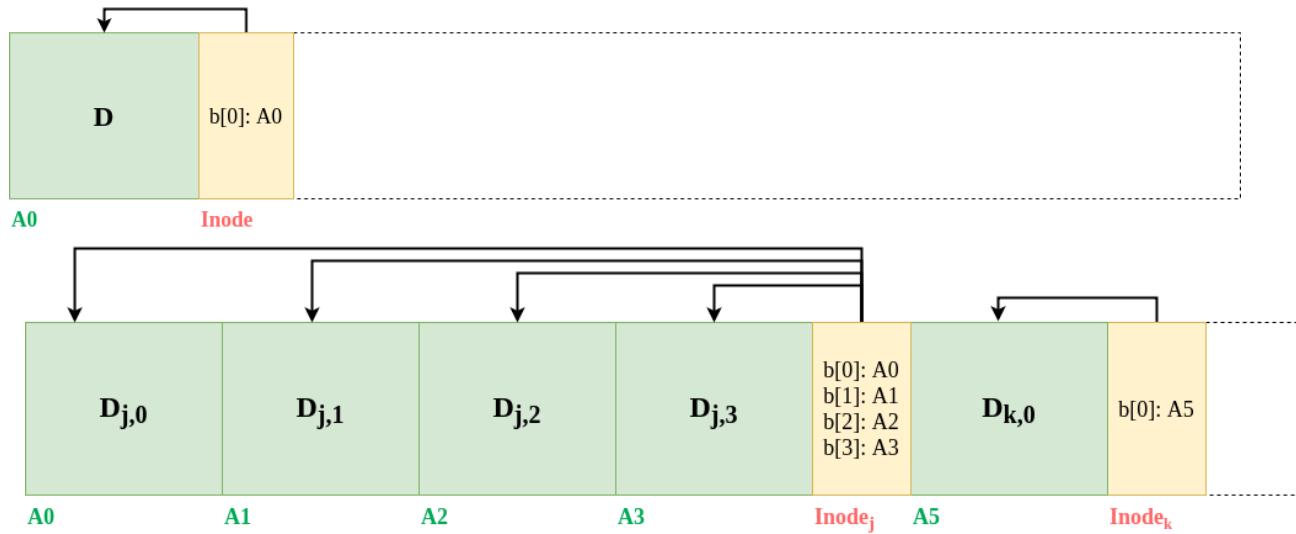
# Log-Structured File Systems

- Disk seek time does not improve as fast as relative to CPU speed, disk capacity, and memory capacity.
  - **Disk caches** can satisfy most requests (e.g. buffer cache )
- In the future:  
most disk accesses will be writes
- LSF optimizes for writes !

# Log-Structured File Systems

- Basic idea: Buffer all writes (data + metadata) using an in-memory segment; once the segment is full, write the segment to a log (aka checkpoint)
- The segment write is one sequential write, so its fast
- We write one large segment (e.g., 1 or 4 MB) instead of a bunch of block-sized chunks to ensure that, at worst, we pay only one seek and then no rotational latencies (instead of one seek and possibly many rotational latencies)
- There are no in-place writes as in previous discussed filesystem implementation
- Reads still require random seeks, but physical RAM is plentiful, so buffer/page cache hit rate should be high (more on that later)

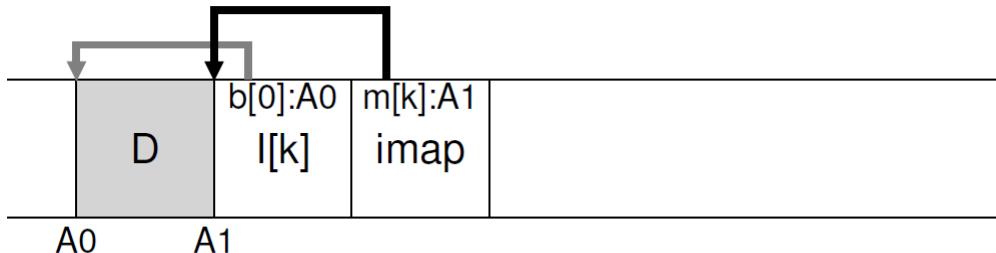
# LSF: Segments



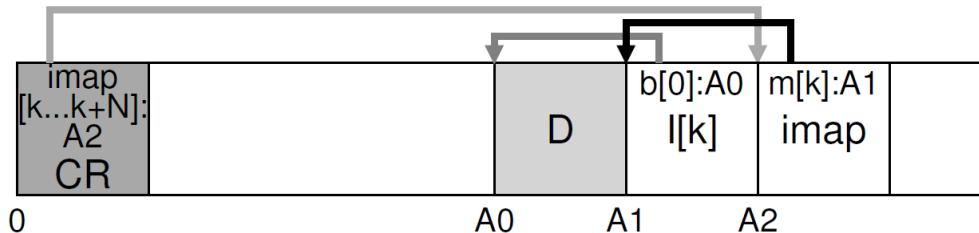
In order to determine whether a block is stale, you need to know its identity. This is stored in an additional part of the segment called the **segment table**. The segment table contains an entry for each block in the segment, which identifies which file (inode number) the block is part of, and which part it is (e.g. "direct block 37", or "the inode").

# Log-Structured File Systems

- i-nodes now scattered over the disk
- Part of the i-node map is written with inode/block to log



- An i-node map, indexed by i-number, is maintained.
- The map is kept on disk at fixed location and is also cached.



- Checkpoint region (CR) only update periodically (30secs)

# Log-Structured File Systems

- Disks are not infinite, hence at some point no further segment can be written to the log
  - But many segments contains blocks that are no longer needed.
    - Block overwritten with a new block (now in a different segment)
    - E.g. file created then deleted (think compile)
- Cleaner Thread (kernel)

# LSF ( Cleaner Thread )

- Scan the log circularly to compact it
- Reads in the summary of the next segment to identify inodes and blocks
- Looks up inode map to check whether
  - inode is still in the current inode-map ( deleted ?)
  - inode is still current ( current pending write , so will be overwritten )
- Inodes and blocks still in use will go into memory and written to next segment
- Original Segment (and now compacted) is marked free
- This way disk is a big circular buffer
  - Writer Thread adds new segments to the front
  - Cleaner Thread removing old ones from the back
- Crash in period between checkpoints results in data loss or inconsistency

# Journaling File System

- LSF not widely used, though principles continue in JFS
- Example: Microsoft NTFS, Linux ext3
- Deals with consistency issues
- Consider "rm /home/franke/nofreelunch"
  - Release the i-node
  - Release all the disk blocks to the free block pool
  - Remove the file from its directory
- Order of operation matters in the presence of crashes
  - Regardless of order, resources might become orphaned

# Journaling File System

- The JFS first writes an entry of the three actions to be taken to the journal.
- Commit (write) journal to disk (now we have a record)
- Only after the journal is written, do the operations begin
- After operations are completed the journal entry is released
- If the system crashes before it is done, then after rebooting the journal is checked and the operations are rerun.
- Note: the journaled operations must be idempotent (i.e. can be repeated as often as necessary without harm).

# Implementing Directories

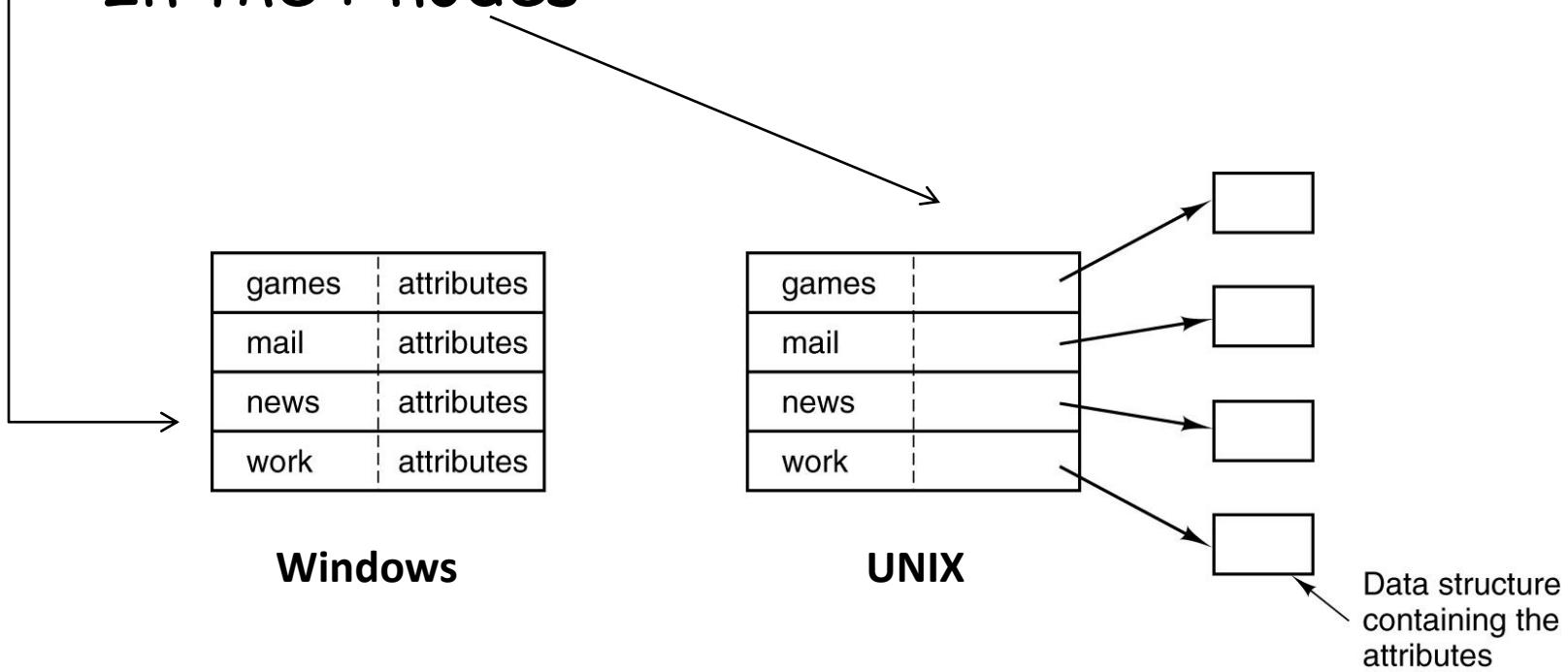


Example:

- Disk address of the file (in contiguous scheme)
- Number of the first block (in linked-list schemes)
- Number of the i-node,

# Implementing Directories

- Where the attributes should be stored?
  - Directly in the directory entry
  - In the i-nodes



# Implementing Directories: Variable-Length Filenames

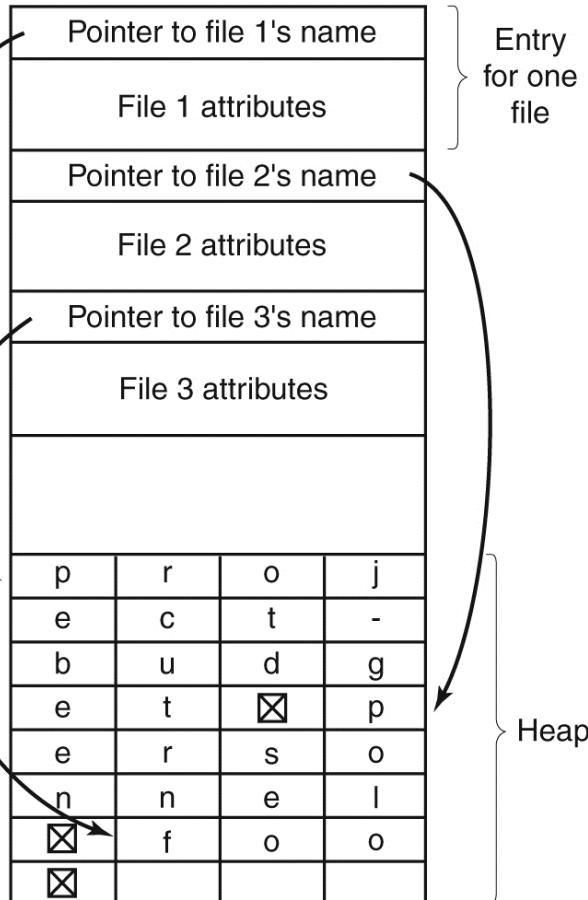
Entry  
for one  
file

File 1 entry length			
File 1 attributes			
p	r	o	j
e	c	t	-
b	u	d	g
e	t	☒	
File 2 entry length			
File 2 attributes			
p	e	r	s
o	n	n	e
l	☒		
File 3 entry length			
File 3 attributes			
f	o	o	☒
⋮			

Disadvantages:

- Entries are no longer of the same length.
- Variable size gaps when files are removed
- A big directory may span several pages which may lead to page faults.

# Implementing Directories: Variable-Length Filenames



- Keep directory entries fixed length
- Keep filenames in a heap at the end of the directory.
- Page faults can still occur while accessing filenames.

# Implementing Directories

- For extremely long directories, linear search can be slow.
  - Hashing can be used
  - Caching can be used
- Note: A directory is nothing but a file, where the filedata represents the lookup table and the “directory bit” is set in the inode

# Listing content of directory

```
DIR *opendir(const char *name);
struct dirent *readdir(DIR *dirp);

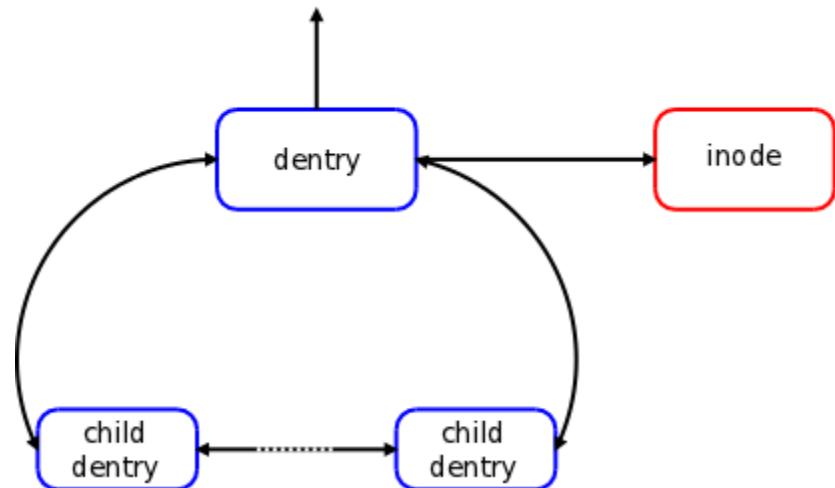
    struct dirent {
        ino_t          d_ino;           /* Inode number */
        off_t          d_off;          /* Not an offset; see below */
        unsigned short d_reclen;      /* Length of this record */
        unsigned char   d_type;        /* Type of file; not supported
                                         by all filesystem types */
        char            d_name[256];    /* Null-terminated filename */
    };
}
```

```
main() {
    DIR *dir;
    struct dirent *entry;

    if ((dir = opendir("/")) == NULL)
        perror("opendir() error");
    else {
        puts("contents of root:");
        while ((entry = readdir(dir)) != NULL)
            printf(" %s\n", entry->d_name);
        closedir(dir);
    }
}
```

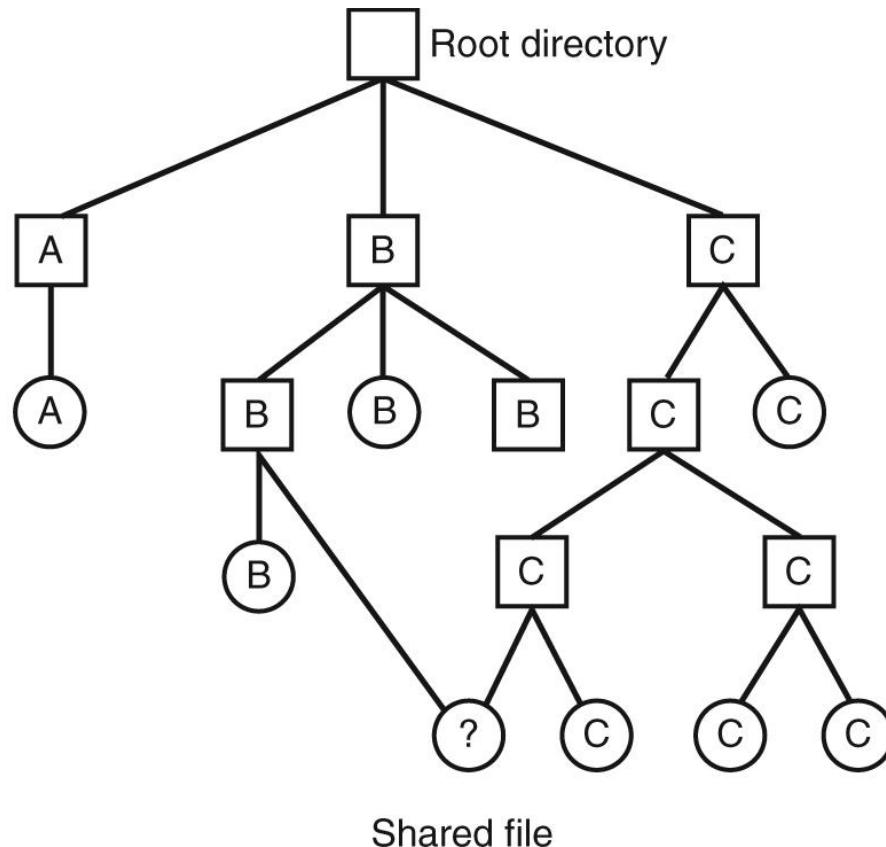
# Speeding it up

- Continuously going to the disk is expensive  
e.g. "/home/frankeh/nyu/best/class/ever"
- Root -> home -> frankeh -> nyu -> best -> class -> ever  
multiple inodes/blocks and dentries (directories) need  
to be read
- Same old story → caching, caching, caching
- **DENTRY cache**  
paths that are walked are  
stored as an *in-memory*  
directory entry tree



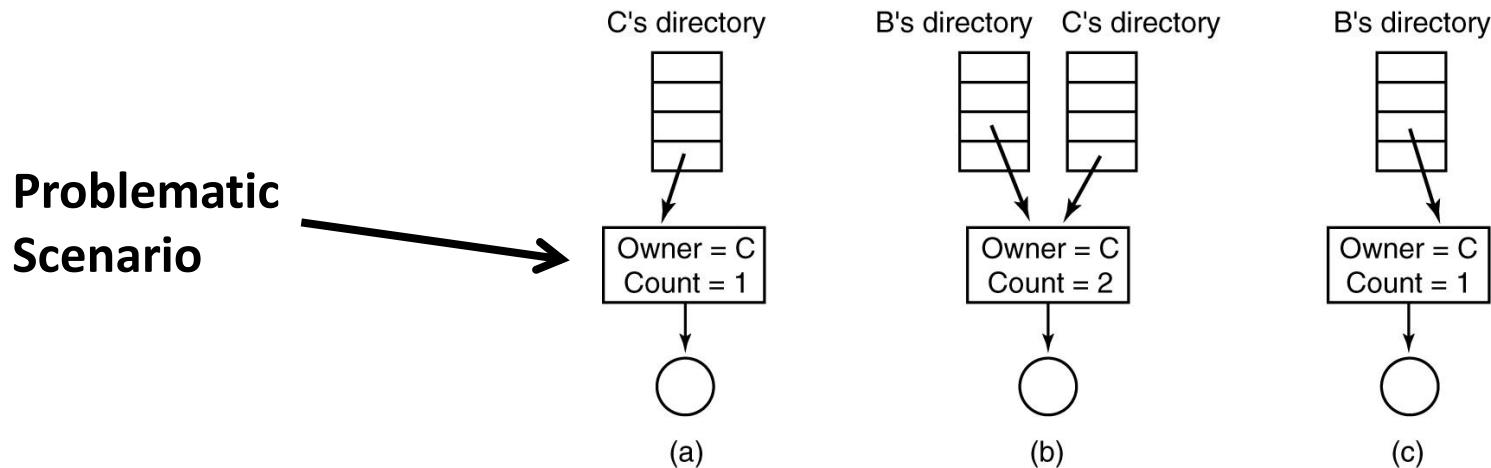
# Shared Files

- Appear simultaneously in different directories



# Shared Files: Method 1

- Disk blocks are not listed in directories but in a data structure associated with the file itself (e.g. i-nodes in UNIX).
- Directories just point to that data structure.
- This approach is called: **static linking**
- “`ln <origfile> <newfilename>`” or “`ln -P`”

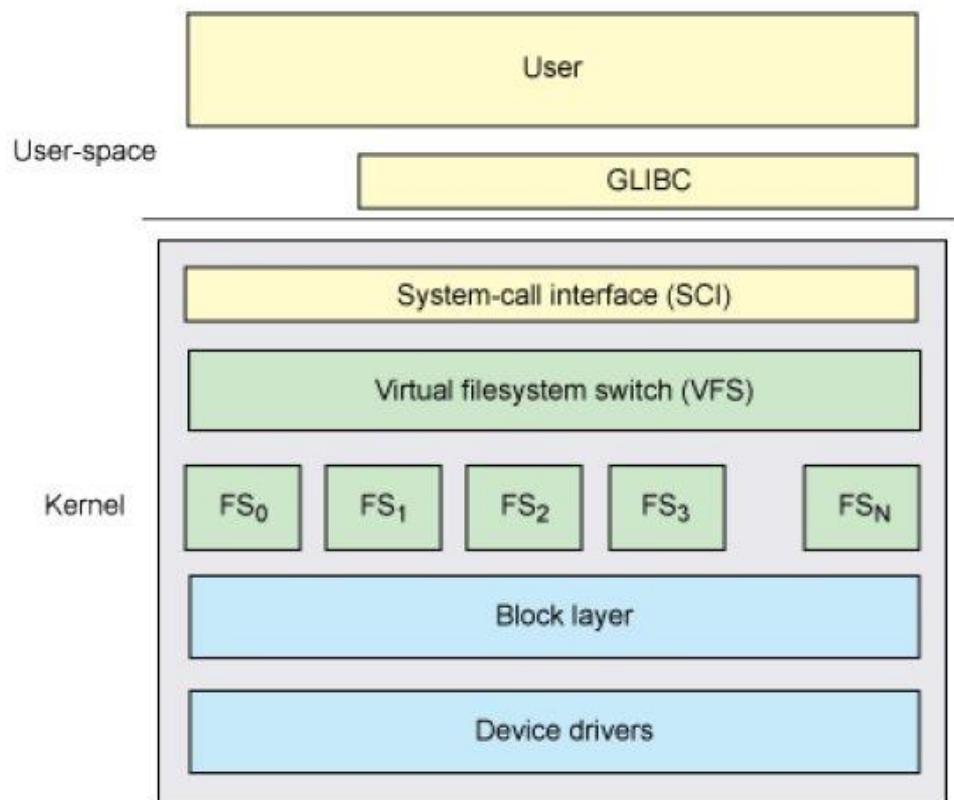


# Shared Files: Method 2

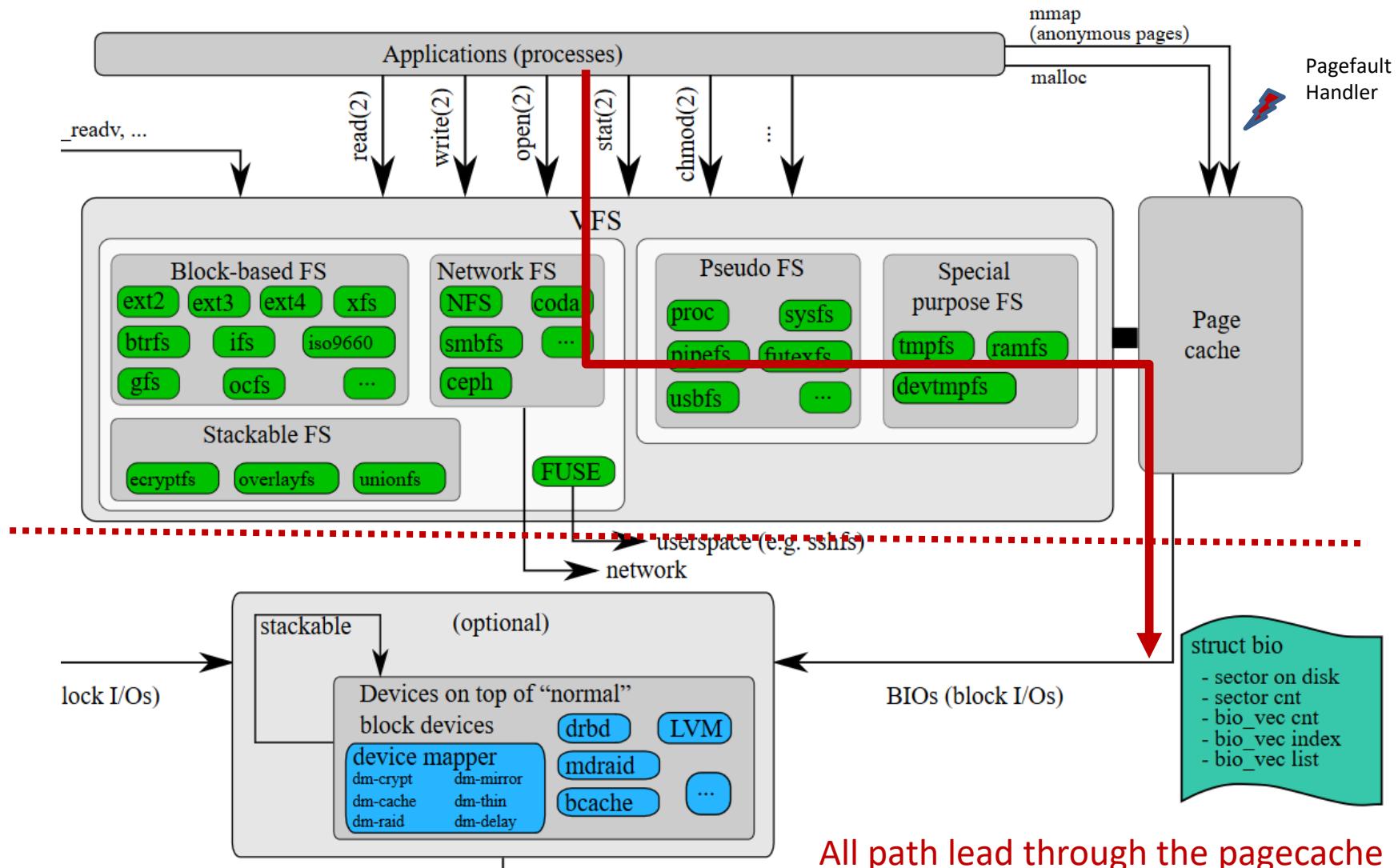
- Have the system create a new file (of type LINK). This new file contains the path name of the file to which it is linked.
- This approach is called: **symbolic linking**
- The main drawback is the extra overhead.
  
- “`ln -s <origfile> <newfilename>`”

# Virtual File Systems

- Integrating multiple file systems into an orderly structure.
- Provides a common layer to the SCI and services to switch to different filesystems underneath
- Said filesystems then take advantage of common OS service such as caching (e.g. block cache) and (block I/O layer)

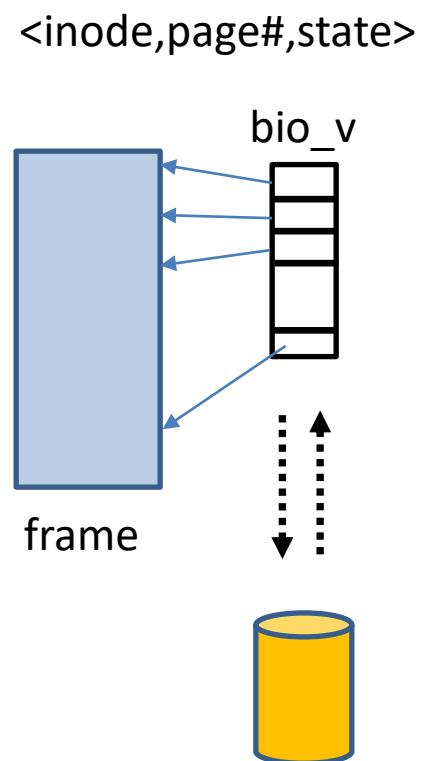


# Page Cache and Other types of FileSystems



# Page Cache

- Integrates I/O caching (buffer cache) and memory mapped file
- Page cache indexed by <inode,vpageofs>  
vpageofs is the virtual page relative to the beginning of the file.
- If no hit, frame is allocated and I/O is issued as set of bio's (block IOs), when all are completed, the data is in cache and the file can be mapped or data can be provided to application.
- Order of the bio's can depend on whether data is requested via mmaps or read/write.
- Page Cache responsible for flushing data (sync) in due time (write backs)



# Page Cache

- PageCache pervasively used in OS
- If there is no other use for memory, why not for cache ?
- Freshly booted system (4GB), mostly free but already 710MB cached

```
root@lnx1:~# free -m
      total        used        free      shared  buff/cache   available
Mem:       3944        263      2970          1      710        3459
Swap:      1942          0      1942
```

- Search the entire disk for something -> access all data

```
root@lnx1:/# find . -type f -print | xargs grep some_string_i_wont_find
```

- PageCache now heavily loaded, almost no memory free

```
root@lnx1:/# free -m
      total        used        free      shared  buff/cache   available
Mem:       3944        292        149          1      3502        3392
Swap:      1942          4      1938
```

- When memory is required for apps (2GB) the page cache will be reduced

```
root@lnx1:/# free -m
      total        used        free      shared  buff/cache   available
Mem:       3944      2306        144          1      1493        1385
Swap:      1942          11      1930
```

# Effects of PageCache

- Significantly speeds up searching through 2.4GB disk space
- Start with a freshly booted system and page cache small

```
frankeh@lnx1:~/NYU$ free -m
total        used        free      shared  buff/cache   available
Mem:       3944         258       2971          1       714       3464
Swap:      1942          0       1942
frankeh@lnx1:~/NYU$ time grep -nr some_bizarre_foobar_garbage
```

```
real    0m14.858s
user    0m1.632s
sys     0m5.444s
```

```
frankeh@lnx1:~/NYU$ free -m
total        used        free      shared  buff/cache   available
Mem:       3944         257       467          1       3219       3438
Swap:      1942          0       1942
frankeh@lnx1:~/NYU$ time grep -nr some_bizarre_foobar_garbage
```

```
real    0m4.523s
user    0m4.074s
sys     0m0.443s
```

```
frankeh@lnx1:~/NYU$ du -ms .
2435 .
```

Cache now loaded

# Network File System

- Files are located on a different server
  - CIFS, NFS, ....
- Requests are sent over to server with name and block requests
- Data can be cached, but be aware of sharing

# Pseudo FileSystem

- /proc or /sys
- Means to access/manipulate OS internals  
(state, parameters, algos, settings)
- Allows scripting (vs. a special syscall interface)
- The hierarchy is created dynamically and on access, there is no real disk !

# Proc fs (inspecting state)

- Inspecting a single process or system statistics
- Registers read/write functions per “node”

```
frankeh@lnx1:~$ ps -edf | grep 1692
frankeh 1692 1509 0 Apr27 ? 00:00:19 x-terminal-emulator

frankeh@lnx1:~$ ls /proc/1692/
attr      coredump_filter  gid_map    mountinfo   oom_score   sched     stat      uid_map
autogroup  cpuset          io         mounts     oom_score_adj schedstat  statm    wchan
auxv       cwd              limits    mountstats pagemap    sessionid  status
cgroup     environ          loginuid  net        patch_state setgroups syscall
clear_refs exe              map_files ns        personality smaps    task
cmdline    fd               maps      numa_maps projid_map smaps_rollup timers
comm       fdinfo          mem      oom_adj    root      stack    timerslack_ns

frankeh@lnx1:/proc/1692$ cat stat
1692 (x-terminal-emul) S 1509 1397 1397 0 -1 4194304 13687 13059 12 49 1472 604 61 132 20 0 3 0 3444 524214272 10142
18446744073709551615 94007262363648 94007262436208 140727902463216 0 0 0 0 4096 65536 0 0 0 17 2 0 0 4 0 0 9400726453
5656 94007264542176 94007273050112 140727902464728 140727902464748 140727902464748 140727902466011 0
frankeh@lnx1:/proc/1692$ cat statm
127982 10142 5828 18 0 12981 0

frankeh@lnx1:~$ cat /proc/1692/maps | head
557fc57ac000-557fc57be000 r-xp 00000000 08:01 1194341          /usr/bin/lxterminal
557fc59be000-557fc59bf000 r--p 00012000 08:01 1194341          /usr/bin/lxterminal
557fc59bf000-557fc59c0000 rw-p 00013000 08:01 1194341          /usr/bin/lxterminal
557fc61dd000-557fc72da000 rw-p 00000000 00:00 0                  [heap]
7fc988000000-7fc988021000 rw-p 00000000 00:00 0
7fc988021000-7fc98c000000 ---p 00000000 00:00 0
7fc98c000000-7fc98c021000 rw-p 00000000 00:00 0
```

# Sys-fs

Read and modify system status and behavior

```
frankeh@lnx1:/sys$ ls
block bus class dev devices firmware fs hypervisor kernel module power
frankeh@lnx1:/sys$ cd bus
frankeh@lnx1:/sys/bus$ ls
ac97      container    gpio  iscsi_flashnode  mipi-dsi  nvmem        platform  sdio   usb      xen
acpi      cpu          hid   machinecheck    mmc       pci          pnp     serial  virtio   xen-backend
clockevents edac        i2c   mdio_bus       nd        pci-epf    rapidio  serio   vme
clocksource event_source isa   memory        node      pci_express scsi     spi     workqueue
frankeh@lnx1:/sys/bus$ cd pci
frankeh@lnx1:/sys/bus/pci$ ls
devices  drivers  drivers_autoprobe  drivers_probe  rescan  resource_alignment  slots  uevent
frankeh@lnx1:/sys/bus/pci$ ls devices/
0000:00:00.0  0000:00:01.1  0000:00:03.0  0000:00:05.0  0000:00:07.0  0000:00:0d.0
0000:00:01.0  0000:00:02.0  0000:00:04.0  0000:00:06.0  0000:00:08.0
```

Example: change the scheduling algorithm for a particular disk device (`/dev/sda`):

```
root@lnx1:~# cat /sys/block/sda/queue/scheduler
noop deadline [cfq]
root@lnx1:~# echo noop > /sys/block/sda/queue/scheduler
root@lnx1:~# cat /sys/block/sda/queue/scheduler
[noop] deadline cfq
```

# Special Purpose Filesystem

- RamFS ( builds a ram disk )
- Backed by memory not disk
- Instead of issuing block I/O requests you memcpy(4K) to from memory
- Obviously
  - significantly faster than going to any real device
  - But no persistence
  - Reduces available RAM on your system
- Linux : tmpfs

# File System Performance

- Caching:
  - **Block cache**: a collection of blocks kept in memory for performance reasons
  - **Page cache**: pages of files are kept in memory
- Block Read Ahead:
  - Get blocks into the cache before they are needed
  - See (page cache)
  - Recognize sequential access and pre-read.
- Reducing Disk Arm Motion:
  - Putting blocks that are likely to be accessed in sequence close to each other, preferably in the same cylinder
- Defragmentation

# Example: fs creation and mounting

- Create a new filesystem on a particular device:
  - Allocate inodes and blocks

```
# mkfs -t ext3 /dev/sda6
mke2fs 1.42 (29-Nov-2011)
Filesystem label=
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
Stride=0 blocks, Stripe width=0 blocks
1120112 inodes, 4476416 blocks
223820 blocks (5.00%) reserved for the super user
First data block=0
Maximum filesystem blocks=0
137 block groups
32768 blocks per group, 32768 fragments per group
8176 inodes per group
Superblock backups stored on blocks:
      32768, 98304, 163840, 229376, 294912, 819200, 884736, 1605632, 2654208,
      4096000

Allocating group tables: done
Writing inode tables: done
Creating journal (32768 blocks): done
Writing superblocks and filesystem accounting information: done
```

- Associating a device with a particular filesystem and providing an access point "mount point"

```
mount -t type device directory
```

```
~]# mount -t vfat /dev/sdc1 /media/flashdisk
```

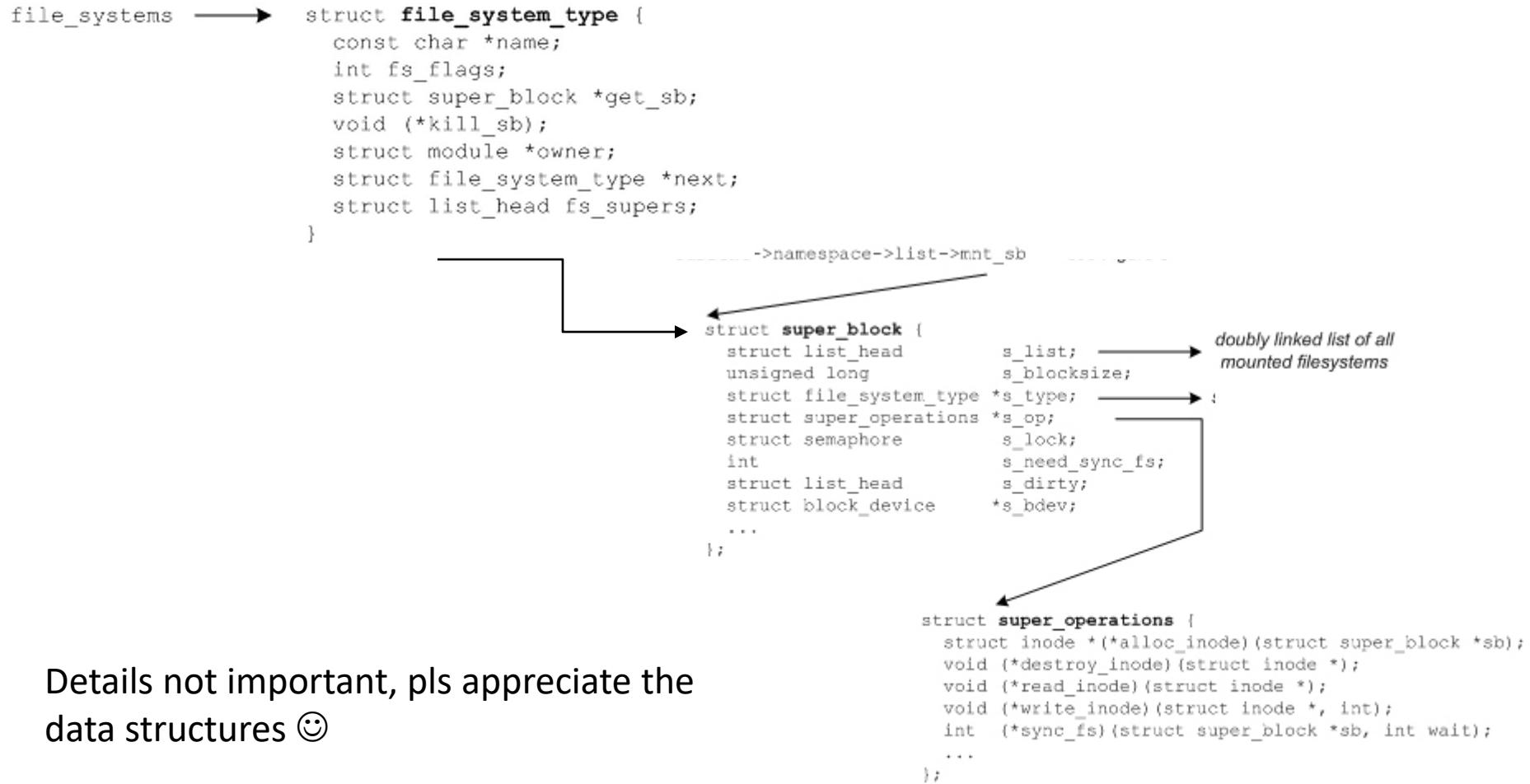
Type	Description
ext2	The <b>ext2</b> file system.
ext3	The <b>ext3</b> file system.
ext4	The <b>ext4</b> file system.
iso9660	The <b>ISO 9660</b> file system. It is commonly used by optical media, typically CDs.
jfs	The <b>JFS</b> file system created by IBM.
nfs	The <b>NFS</b> file system. It is commonly used to access files over the network.
nfs4	The <b>NFSv4</b> file system. It is commonly used to access files over the network.
ntfs	The <b>NTFS</b> file system. It is commonly used on machines that are running the Windows operating system.
udf	The <b>UDF</b> file system. It is commonly used by optical media, typically DVDs.
vfat	The <b>FAT</b> file system. It is commonly used on machines that are running the Windows operating system, and on certain digital media such as USB flash drives or floppy disks.

# “mount” Example

```
root@lnx1:~# mount
sysfs on /sys type sysfs (rw,nosuid,nodev,noexec,relatime)
proc on /proc type proc (rw,nosuid,nodev,noexec,relatime)
udev on /dev type devtmpfs (rw,nosuid,relatime,size=1986852k,nr_inodes=496713,mode=755)
devpts on /dev/pts type devpts (rw,nosuid,noexec,relatime,gid=5,mode=620,ptmxmode=000)
tmpfs on /run type tmpfs (rw,nosuid,noexec,relatime,size=403920k,mode=755)
/dev/sda1 on / type ext4 (rw,relatime,errors=remount-ro,data=ordered)
```

- Note the top entry “/” is mapped to disk
- At various mount points different filesystems are “attached”, e.g. all pseudo file systems or network file systems.

# Example: Linux Internals



Details not important, pls appreciate the  
data structures 😊

# Example: Linux Internals

```
current->namespace->list —→ struct vfsmount {  
    struct list_head mnt_hash;  
    struct vfsmount *mnt_parent;  
    struct dentry *mnt_mountpoint;  
    struct dentry *mnt_root;  
    struct super_block *mnt_sb;  
    struct list_head mnt_mounts;  
    struct list_head mnt_child;  
    atomic_t mnt_count;  
    int mnt_flags;  
    char *mnt_devname;  
    struct list_head mnt_list;  
}
```

*mounted filesystem list*

# Example: Linux Internals

```
struct inode {
    unsigned long          i_ino;
    umode_t                i_mode;
    uid_t                  i_uid;
    struct timespec        i_atime;
    struct timespec        i_mtime;
    struct timespec        i_ctime;
    unsigned short          i_bytes;
    struct inode_operations *i_op;
    struct file_operations *i_fop;
    struct super_block      *i_sb;
    ...
}

struct inode_operations {
    int (*create) (struct inode *, struct dentry *,
                   struct nameidata *);
    struct dentry *(*lookup) (struct inode *,
                             struct dentry *,
                             struct nameidata *);
    int (*mkdir) (struct inode *, struct dentry *, int);
    int (*rename) (struct inode *, struct dentry *,
                  struct inode *, struct dentry *);
    ...
}

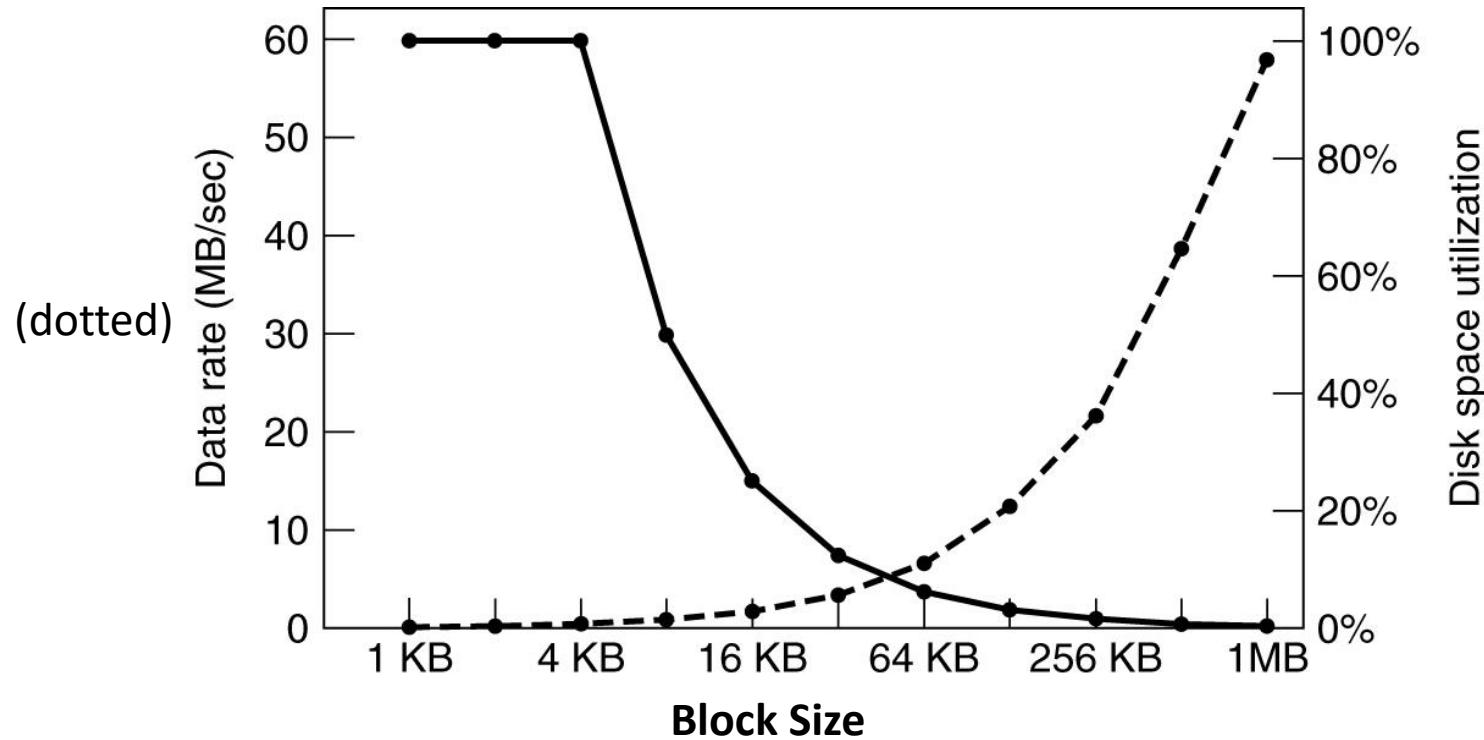
struct file_operations {
    struct module *owner;
    ssize_t (*read) (struct file *, char __user *,
                    size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *,
                     size_t, loff_t *);
    int (*open) (struct inode *, struct file *);
    ...
}
```

The diagram illustrates the inheritance relationship between the `struct inode` and `struct file_operations` structures. The `struct inode` structure is defined on the left, containing fields for inode number, mode, owner, timestamps, bytes used, operation pointers, file operation pointers, and a superblock pointer. The `struct file_operations` structure is also defined on the left, containing pointers to read, write, open, and other file-related functions. A large bracket on the left side of the slide spans from the start of the `struct inode` definition to the end of the `struct file_operations` definition. Two arrows point from the top and bottom of this bracket to the start of the `struct inode_operations` definition, indicating that both `struct inode` and `struct file_operations` inherit from `struct inode_operations`.

# Disk Space Management

- All file systems chop files up into fixed-size blocks that need not be adjacent.
- Block size:
  - Too large → we waste space
  - Too small → we waste time

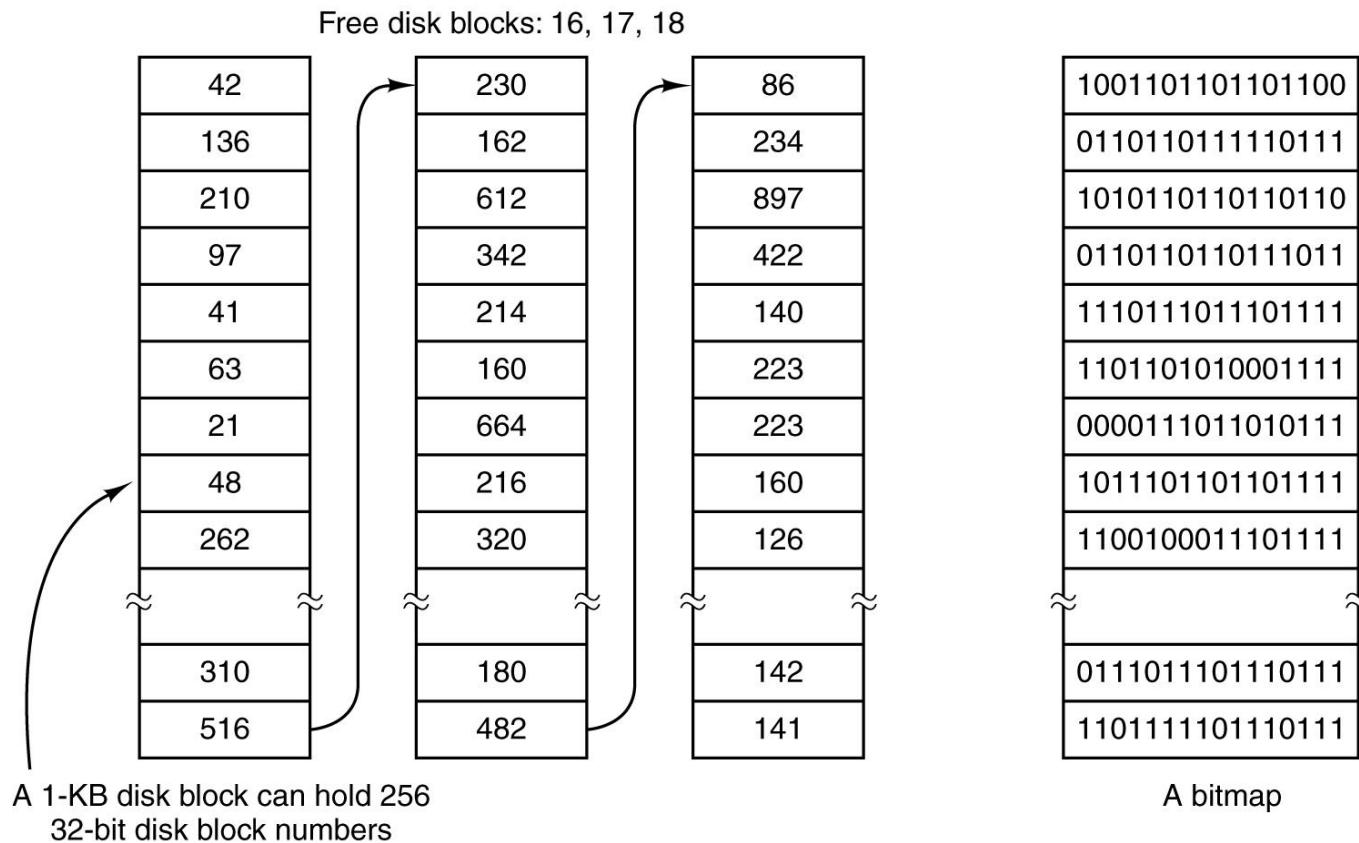
Access time for a block is completely dominated by the seek time and rotational delay.  
So ... **The more data are fetched the better.**



# Disk Space Management: Keeping Track of Free Blocks

- **Method 1:** Linked list of disk blocks, with each block holding as many free disk block numbers as possible.
- **Method 2:** Using a bitmap

# Disk Space Management: Keeping Track of Free Blocks

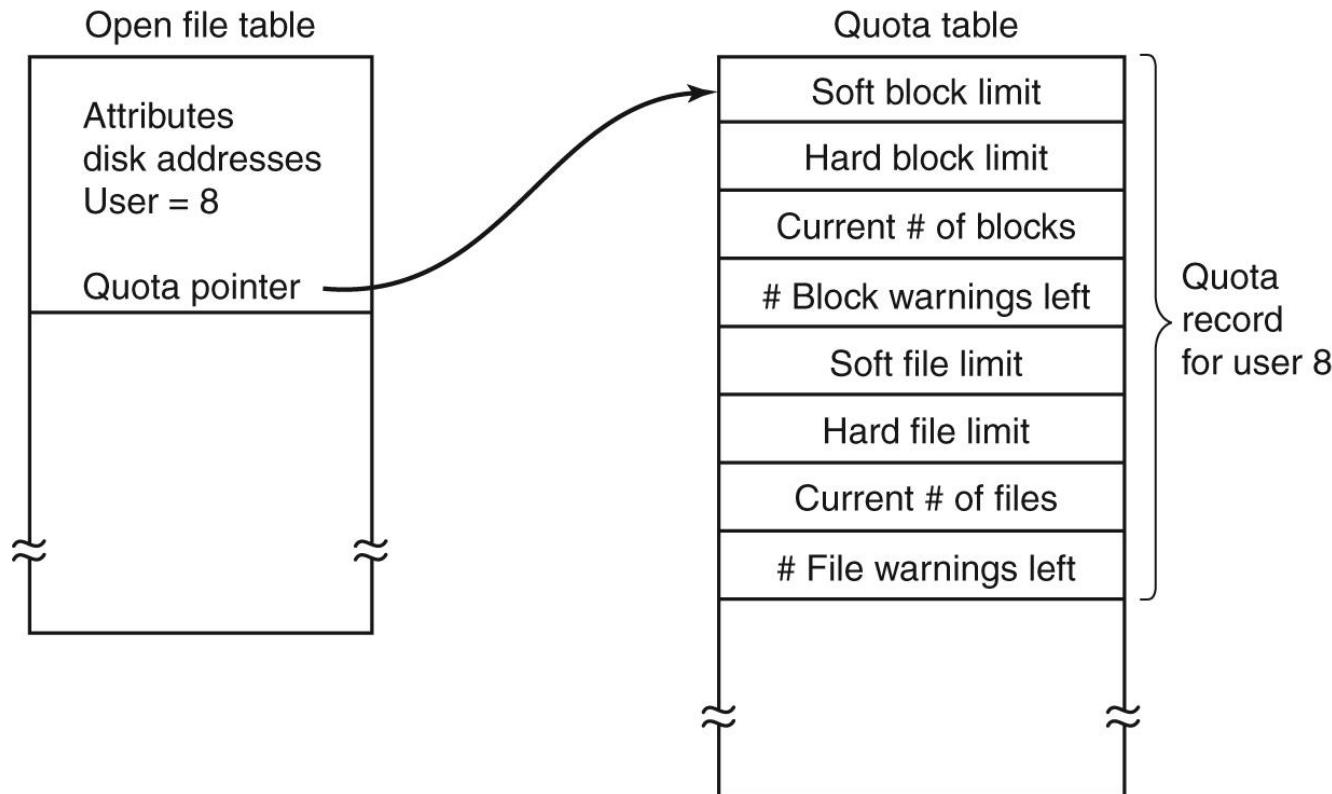


**Free blocks are holding the free list.**

# Disk Space Management: Disk Quotas

- When a user opens file
  - The attributes and disk addresses are located
  - They are put into an **open file table** in memory
  - A second table contains the quota record for every user with a currently open file.

# Disk Space Management: Disk Quotas



# File System Backups

- It is usually desirable to back up only specific directories and everything in them than the entire file system.
- Since immense amounts of data are typically dumped, it may be desirable to compress them.
- It is difficult to perform a backup on an active file system.
- **Incremental dump:** backup only the files that have been modified from last full-backup

# File System Backups: Physical Dump

- Starts at block 0 of the disk
  - Writes all the disk blocks onto the output tape (or any other type of storage) in order.
  - Stops when it has copied the last one.
- + Simplicity and great speed**
- Inability to skip selected directories and restore individual files.**

# File System Backups: Logical Dump

- Starts at one or more specified directories
- Recursively dumps all files and directories found there and have changed since some given base date.

# File System Consistency

- Two kinds of consistency checks
  - Blocks
  - Files

# File System Consistency: Blocks

- Build two tables, each one contains a counter for each block, initially 0
- Table 1: How many times each block is present in a file
- Table 2: How many times a block is present in the free list
- A consistent file system: each block has 1 either in the first or second table

# File System Consistency: Blocks

Block number																
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
1	1	0	1	0	1	1	1	1	0	0	1	1	1	0	0	Blocks in use
0	0	1	0	1	0	0	0	0	1	1	0	0	0	1	1	Free blocks

(a)

Block number																
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
1	1	0	1	0	1	1	1	1	0	0	1	1	1	0	0	Blocks in use
0	0	0	0	1	0	0	0	0	1	1	0	0	0	1	1	Free blocks

(b)

Block number																
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
1	1	0	1	0	1	1	1	1	0	0	1	1	1	0	0	Blocks in use
0	0	1	0	2	0	0	0	0	1	1	0	0	0	1	1	Free blocks

(c)

Block number																
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
1	1	0	1	0	2	1	1	1	0	0	1	1	1	0	0	Blocks in use
0	0	1	0	1	0	0	0	0	1	1	0	0	0	1	1	Free blocks

(d)

# File System Consistency: Blocks

Block number	Blocks in use	Free blocks
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	1 1 0 1 0 1 1 1 1 0 0 1 1 1 1 0 0	0 0 1 0 1 0 0 0 0 0 1 1 0 0 0 1 1

(a)

Add the block to the free list

Block number	Blocks in use	Free blocks
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	1 1 0 1 0 1 1 1 1 1 0 0 1 1 1 1 0 0	0 0 0 0 1 0 0 0 0 0 1 1 0 0 0 1 1

(b)

Block number	Blocks in use	Free blocks
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	1 1 0 1 0 1 1 1 1 1 0 0 1 1 1 1 0 0	0 0 1 0 2 0 0 0 0 0 1 1 0 0 0 1 1

(c)

Block number	Blocks in use	Free blocks
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	1 1 0 1 0 2 1 1 1 1 0 0 1 1 1 1 0 0	0 0 1 0 1 0 0 0 0 0 1 1 0 0 0 1 1

(d)

Rebuild the free list

Allocate a free block, make a copy of that block and give it to the other file.

# File System Consistency: Files

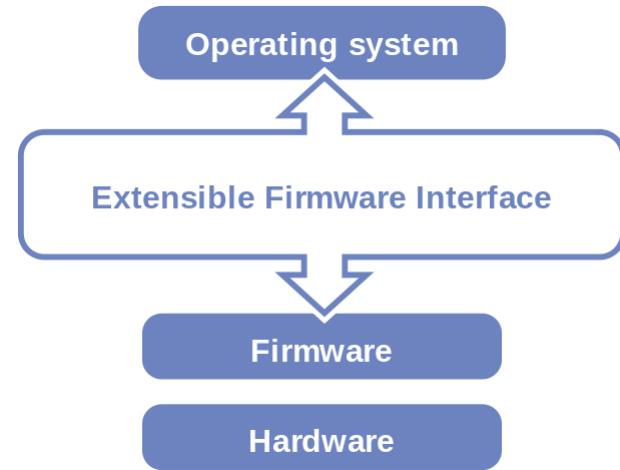
- Table of counters; a counter per file
- Counts the number of that file's usage count.
- Compares these numbers in the table with the counts in the i-node of the file itself.
- Both counts must agree.

# File System Consistency: Files

- Two inconsistencies:
  - count of i-node > count in table
  - count of i-node < count in table
- Fix: set the count in i-node to the correct value

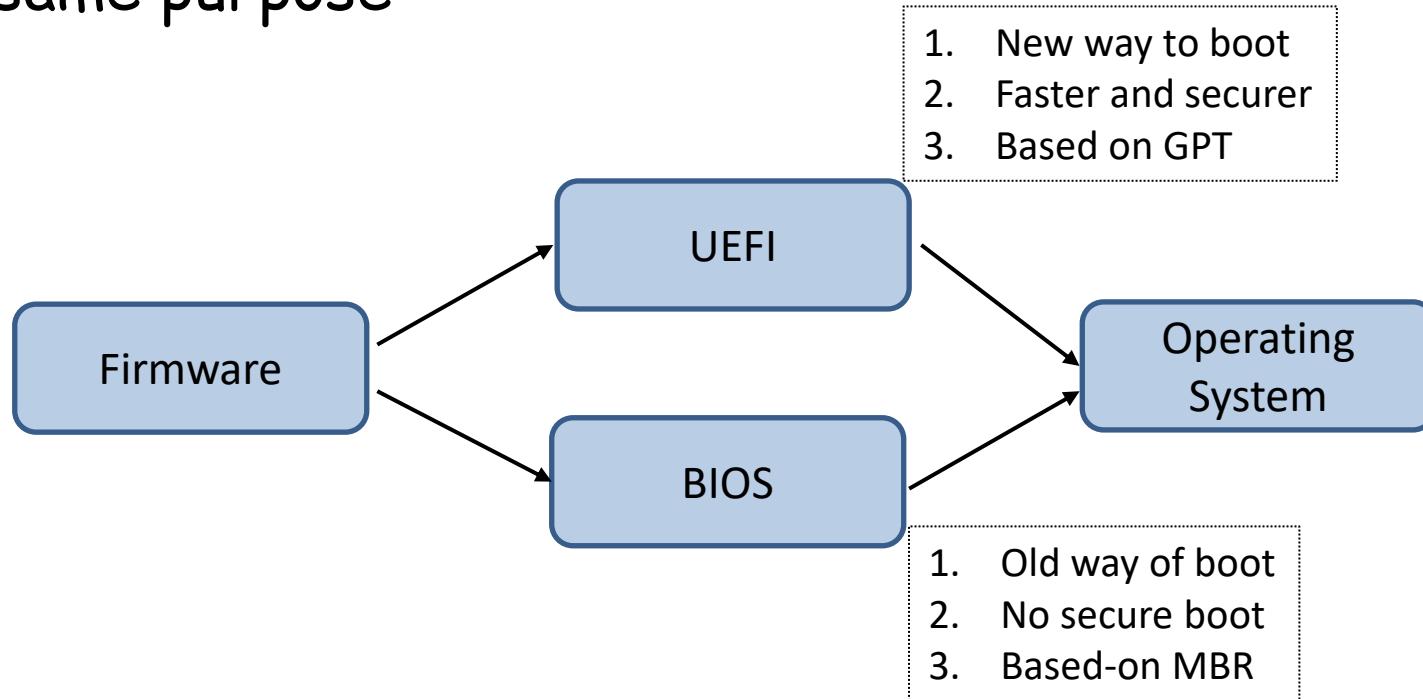
# Booting an Operating System

- **Firmware** is a generic term for embedded software (and its included data) to run something. System controllers in large computer systems that control power up etc have a **mini operating system** (typically a **mini linux**) that's referred to as **firmware**.
- Firmware provides standard API to operating system
- Shields Operating System from low level specifics of a platform
- Assists bringup of physical system to enable the boot process
- Enables settings of the platform (boot order, CPU features )



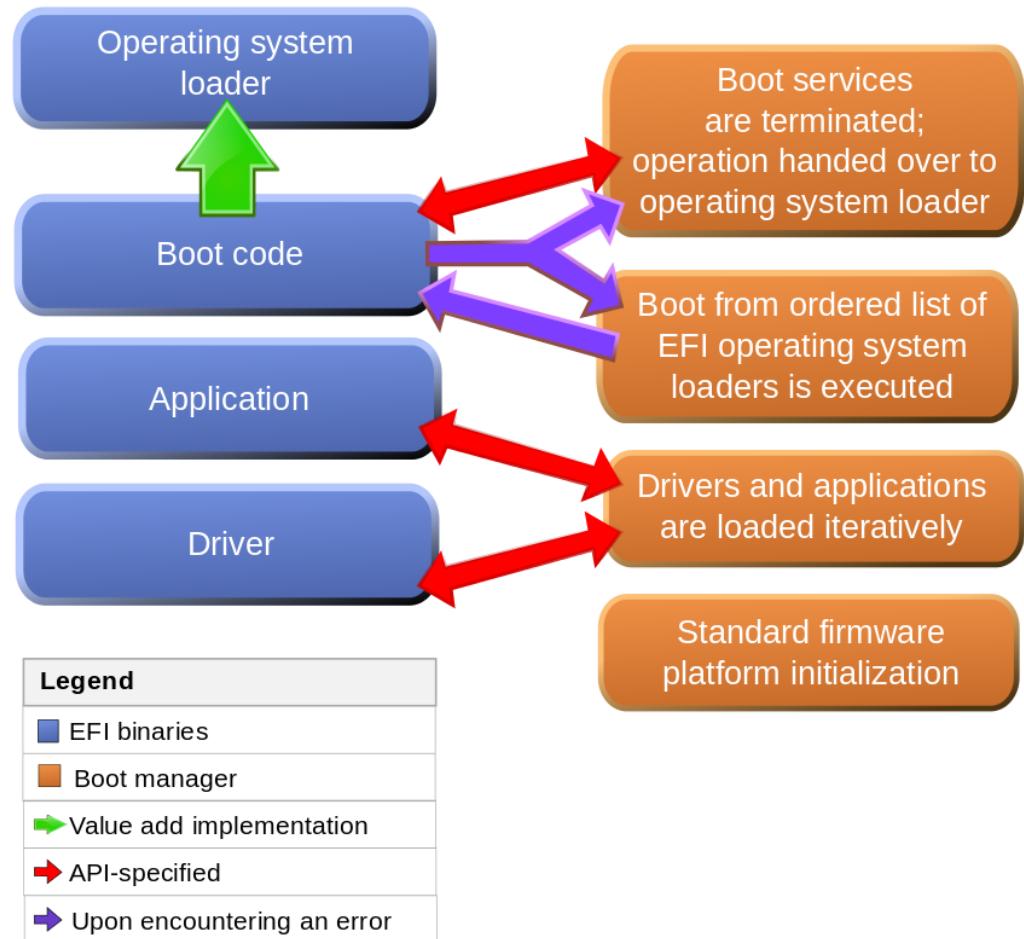
# Booting an Operating System

- Two types of firmware based on features
  - BIOS (legacy till about 2012) : Basic Input Output System
  - UEFI (modern system): Unified Exensible Firmware Interface
- different layouts of the disk and components, though same purpose



# UEFI

## (Universal Extensible Firmware Interface )

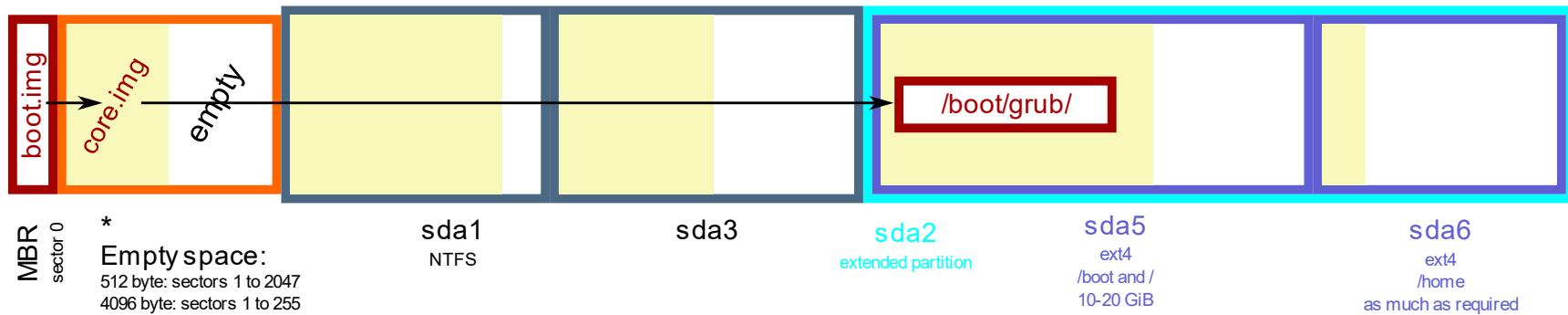


# Disk Layout of bootable Disk

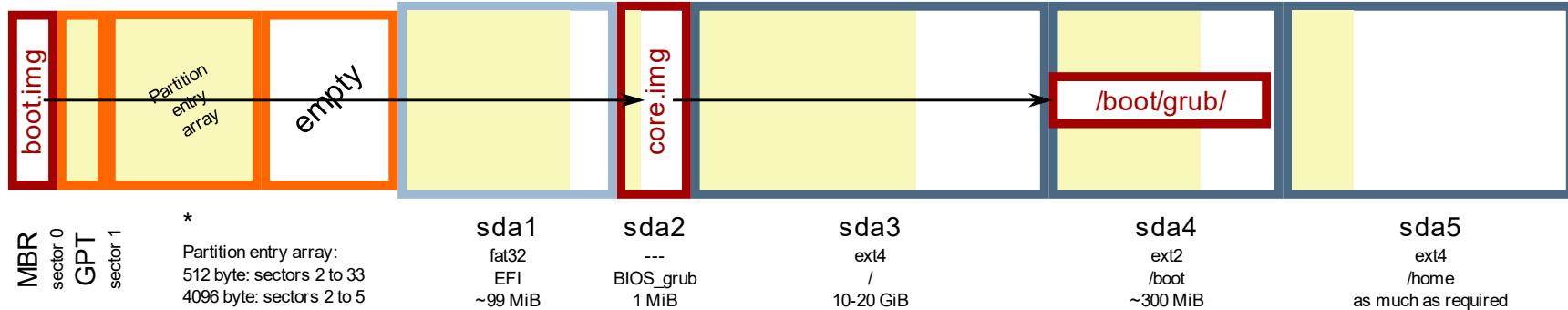
- Need to get to the bootloader (grub) which ultimately loads the OS

Locations of *boot.img*, *core.img* and the */boot/grub* directory

Example 1: An MBR-partitioned hard disk with sector size of 512 or 4096 bytes



Example 2: AGPT-partitioned hard disk with sector size of 512 or 4096 bytes



# Bootstrapping (booting in steps)

- **boot.img:**

This image is the first part of GRUB to start. It is written to a master boot record (MBR) or to the boot sector of a partition. Because a PC boot sector is 512 bytes, the size of this image is exactly 512 bytes. The sole function of boot.img is to read the first sector of the core image from a local disk and jump to it. Because of the size restriction, boot.img cannot understand any file system structure, so grub-install hardcodes the location of the first sector of the core image into boot.img when installing GRUB.

- **core.img:**

This is the core image of GRUB. It is built dynamically from the kernel image and an arbitrary list of modules by the grub-mkimage program. Usually, it contains enough modules to access `/boot/grub`, and loads everything else (including menu handling, the ability to load target operating systems, and so on) from the file system at run-time. The modular design allows the core image to be kept small, since the areas of disk where it must be installed are often as small as 32KB.

- **/boot/grub:**

holds all the modules required for boot including reading disk, menus, graphics handling ..

# Booting (kernel)

- The boot loader (grub) identifies the **kernel** that is to be loaded and copies this kernel image and any associated initrd into memory.
- **vmlinuz** : compressed kernel code
- The **initial RAM disk (initrd)** is an initial file system that is mounted prior to when the real root file system is available. The initrd is bound to the kernel and loaded as part of the kernel boot procedure. The kernel then mounts this initrd as part of the two-stage boot process to load the modules to make the real file systems available and get at the real root file system.
- The initrd contains a minimal set of directories and executables to achieve this, such as the insmod tool to install kernel modules into the kernel.

```
frankeh@lnx1:/boot$ ls -ls | egrep "0-91"
216 -rw-r--r-- 1 root root 217457 Feb 28 05:45 config-4.15.0-91-generic
60784 -rw-r--r-- 1 root root 62235093 Apr 15 19:27 initrd.img-4.15.0-91-generic
3976 -rw----- 1 root root 4069388 Feb 28 05:45 System.map-4.15.0-91-generic
8180 -rw----- 1 root root 8375960 Feb 28 05:51 vmlinuz-4.15.0-91-generic
```

- Config = kernel compile configuration ; System.map = symboltable

# Conclusions

- Files and file systems are major parts of an OS
- Files and File system are the OS way of abstracting storage.
- There are different ways of organizing files, directories, and their attributes.
  - However, VFS is a unifying way to represent a common API
- Remember Murphy's law: What can go wrong will.
  - Please backup your System / Data !!!!



CSCI-GA.2250-001

# Operating Systems

## Lecture 5: Memory Management

Hubertus Franke  
[frankeh@cims.nyu.edu](mailto:frankeh@cims.nyu.edu)



# Programmer's dream



- Private
- Infinitely large
- Infinitely fast
- Non-volatile
- Inexpensive

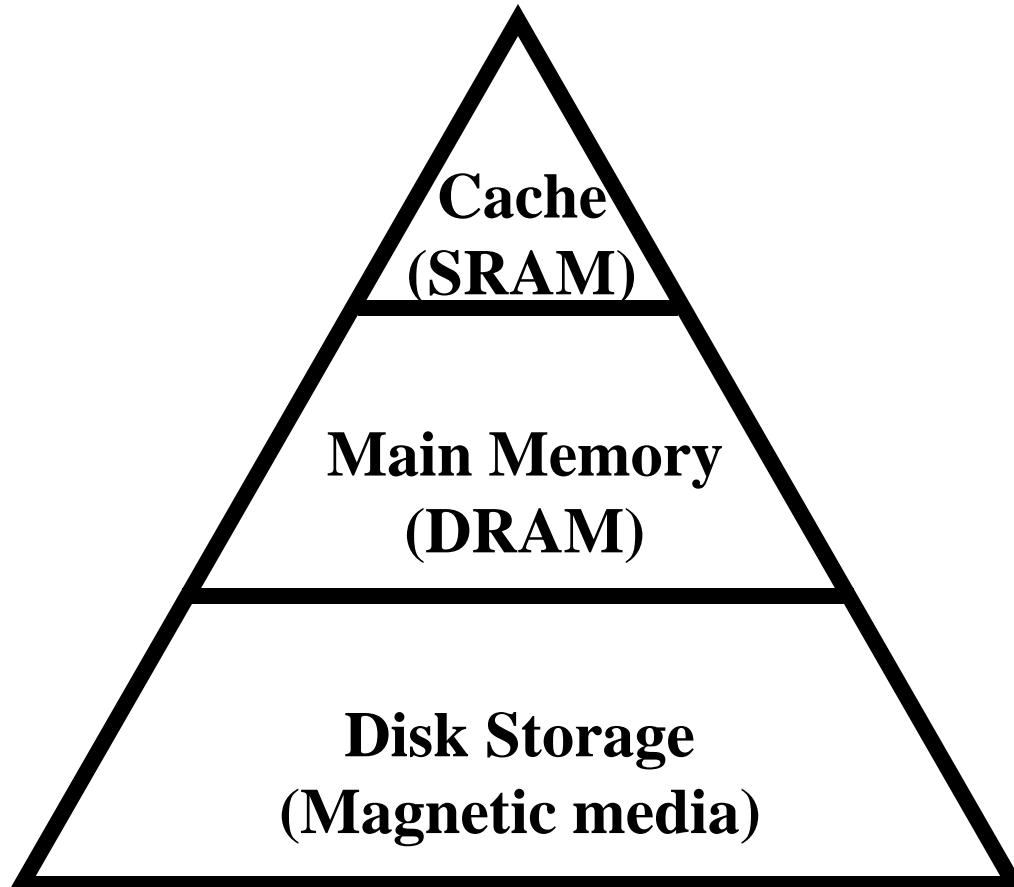
# Programmer's Wish List



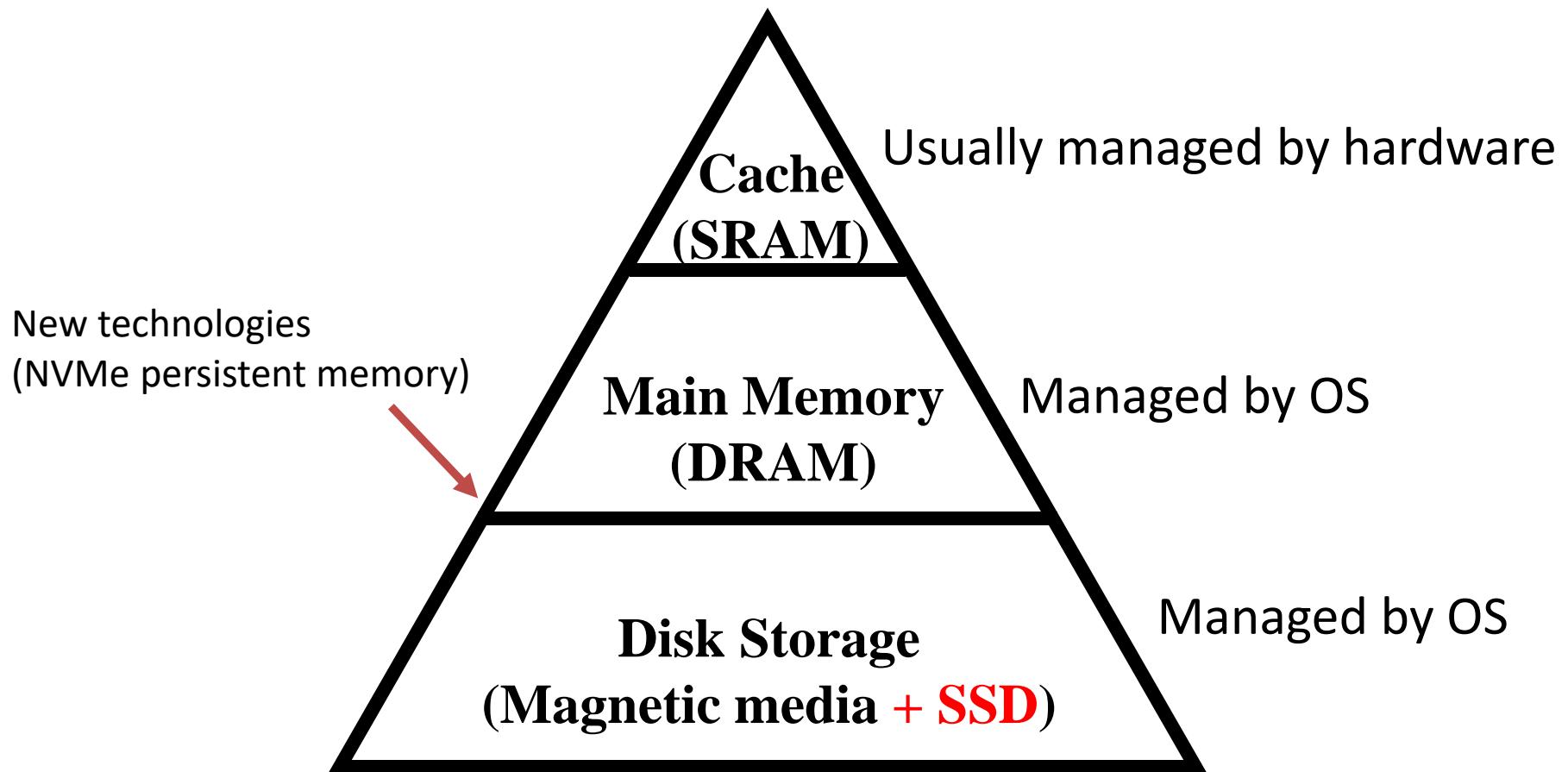
- Private
- Infinitely large
- Infinitely fast
- Non-volatile
- Inexpensive

Programs are getting bigger faster than memories.

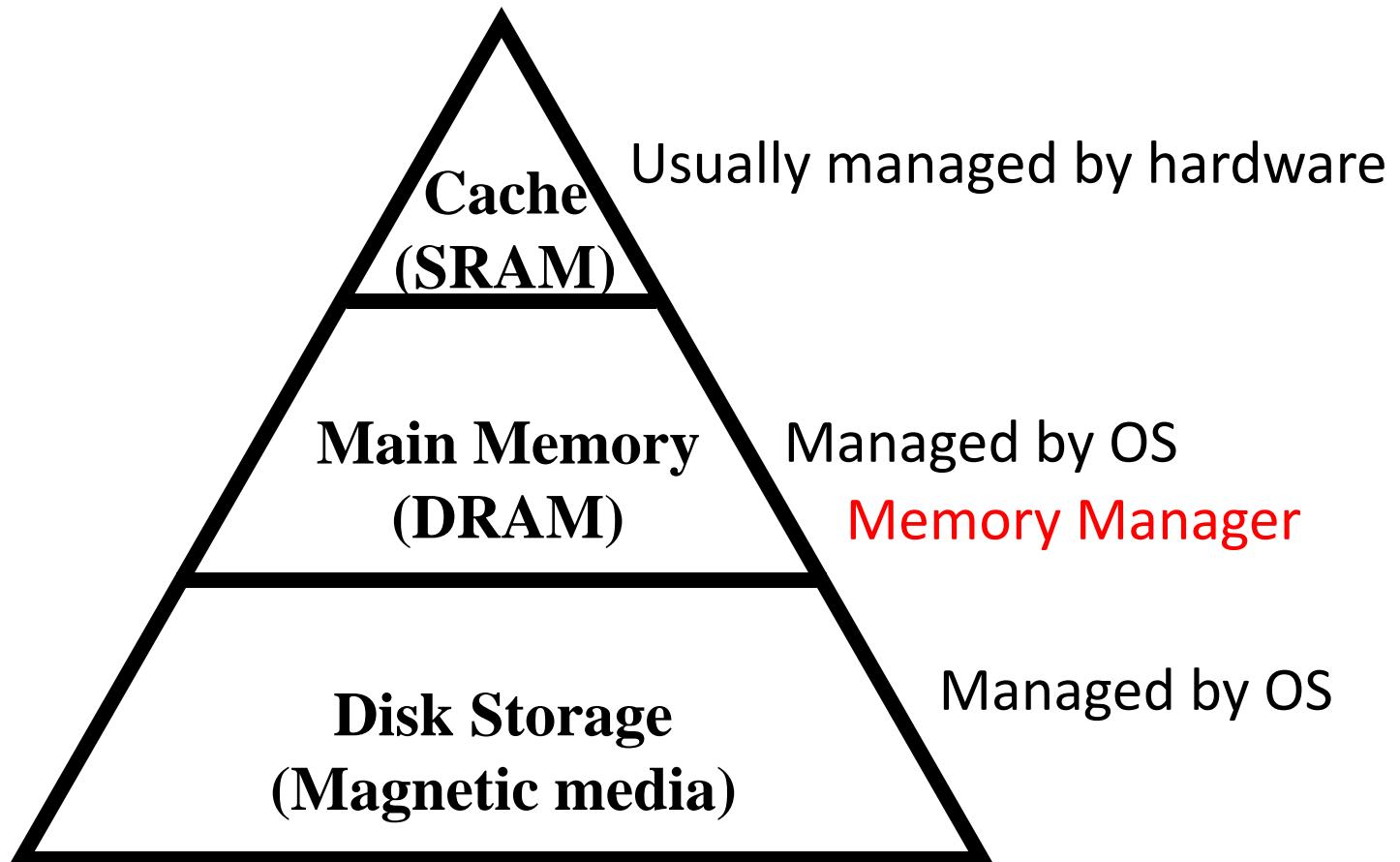
# Memory Hierarchy



# Memory Hierarchy

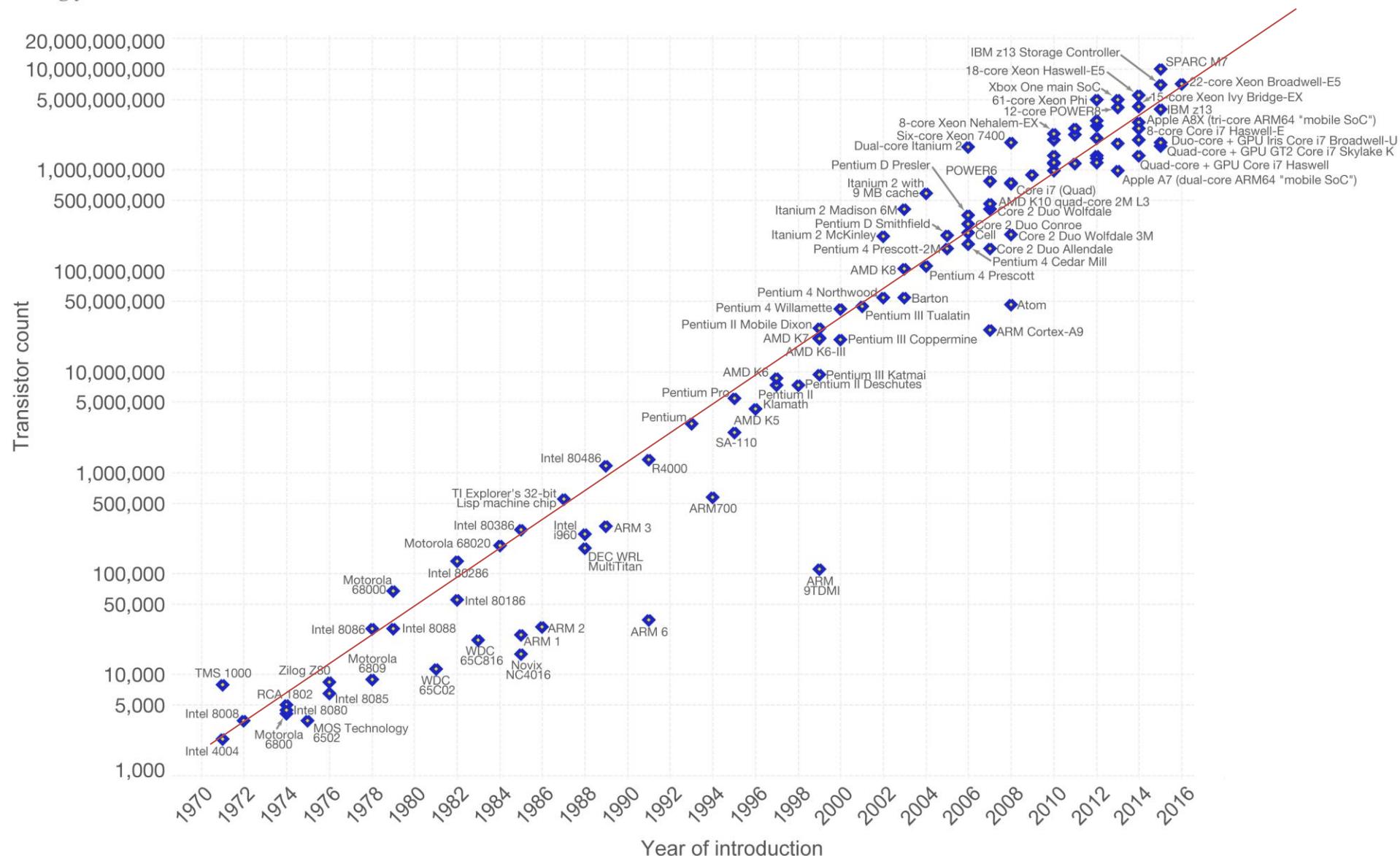


# Memory Hierarchy



# Moore's Law – The number of transistors on integrated circuit chips (1971-2016)

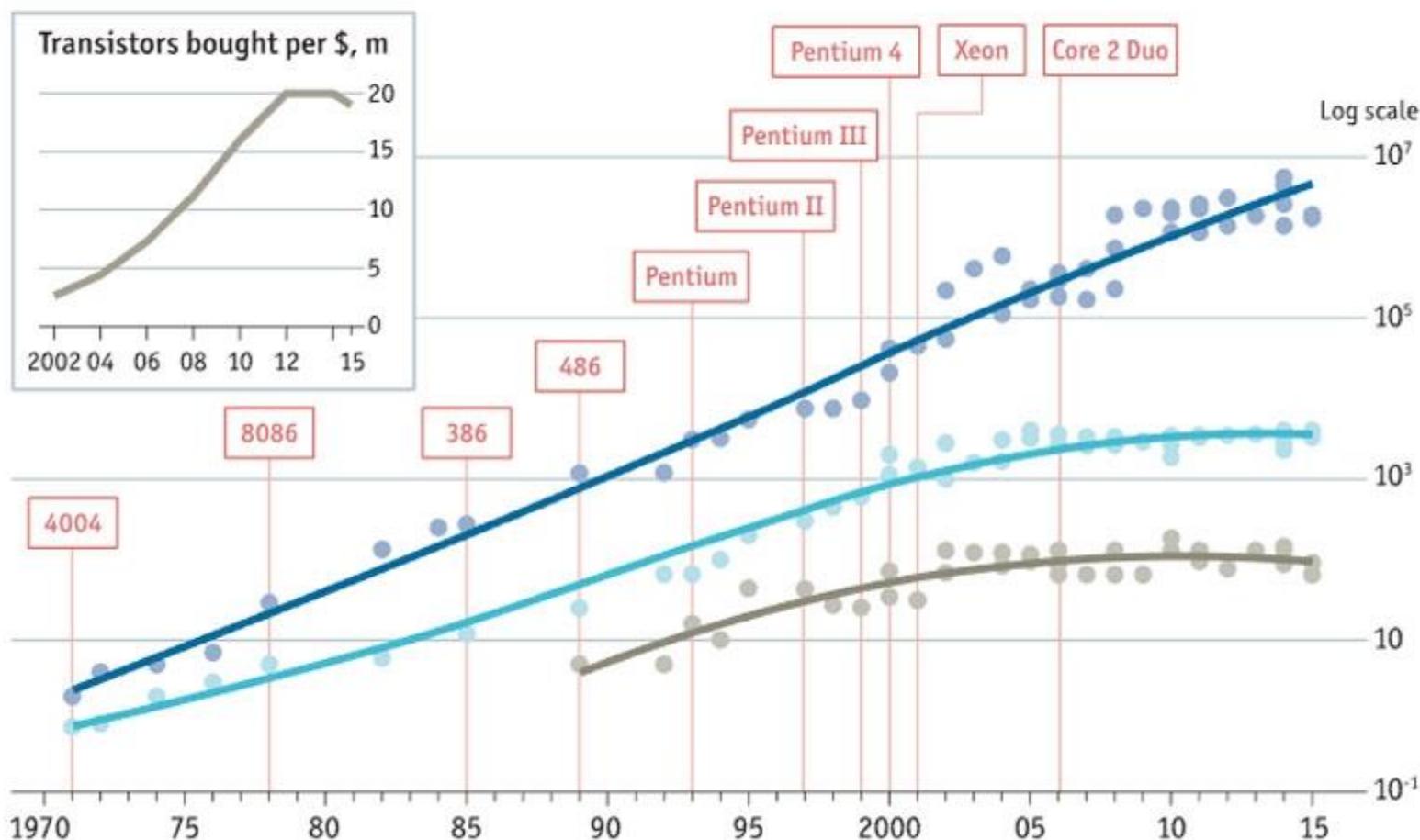
Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are strongly linked to Moore's law.



## Stuttering

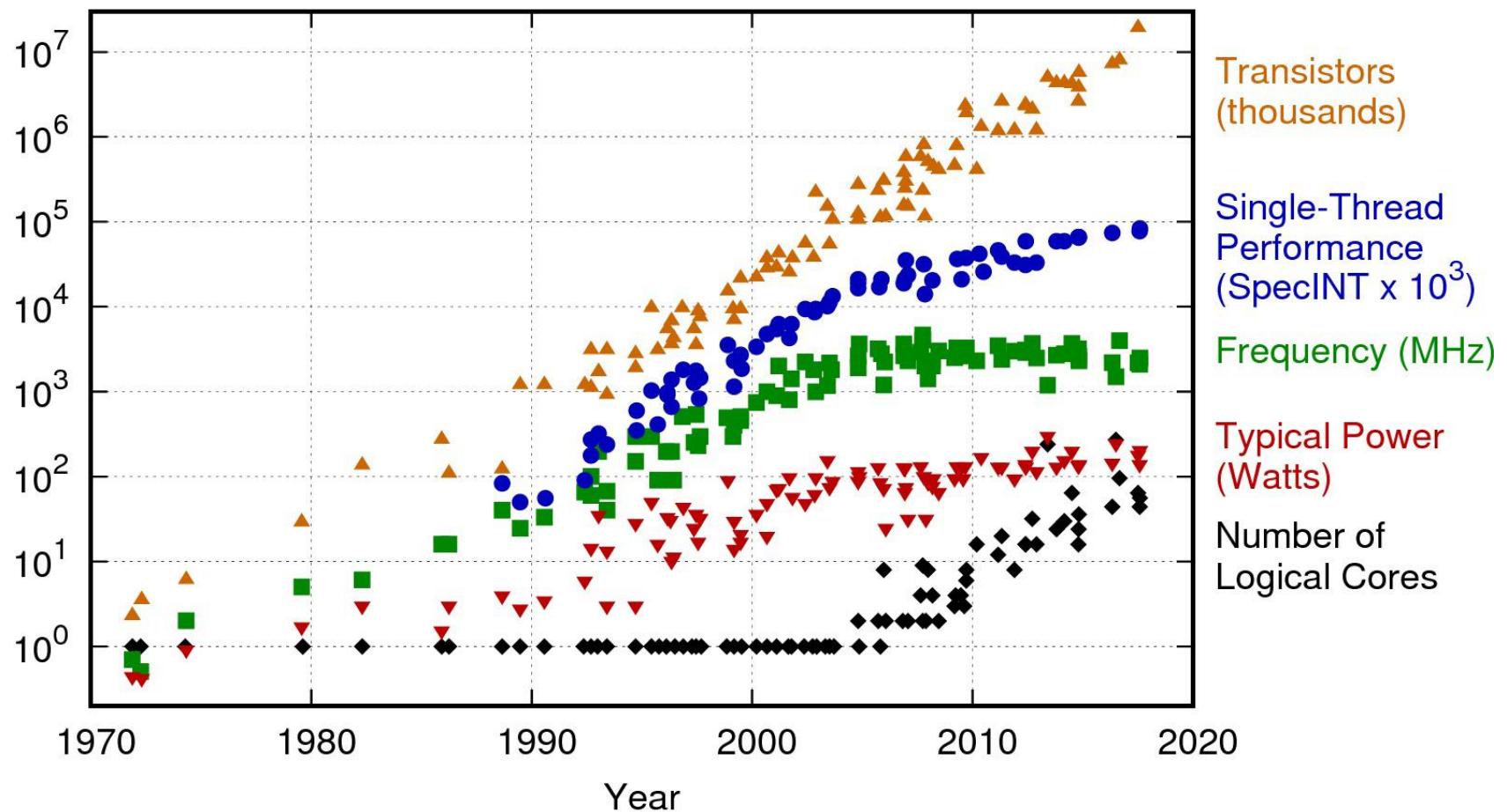
● Transistors per chip, '000 ● Clock speed (max), MHz ● Thermal design power\*, w

Chip introduction  
dates, selected



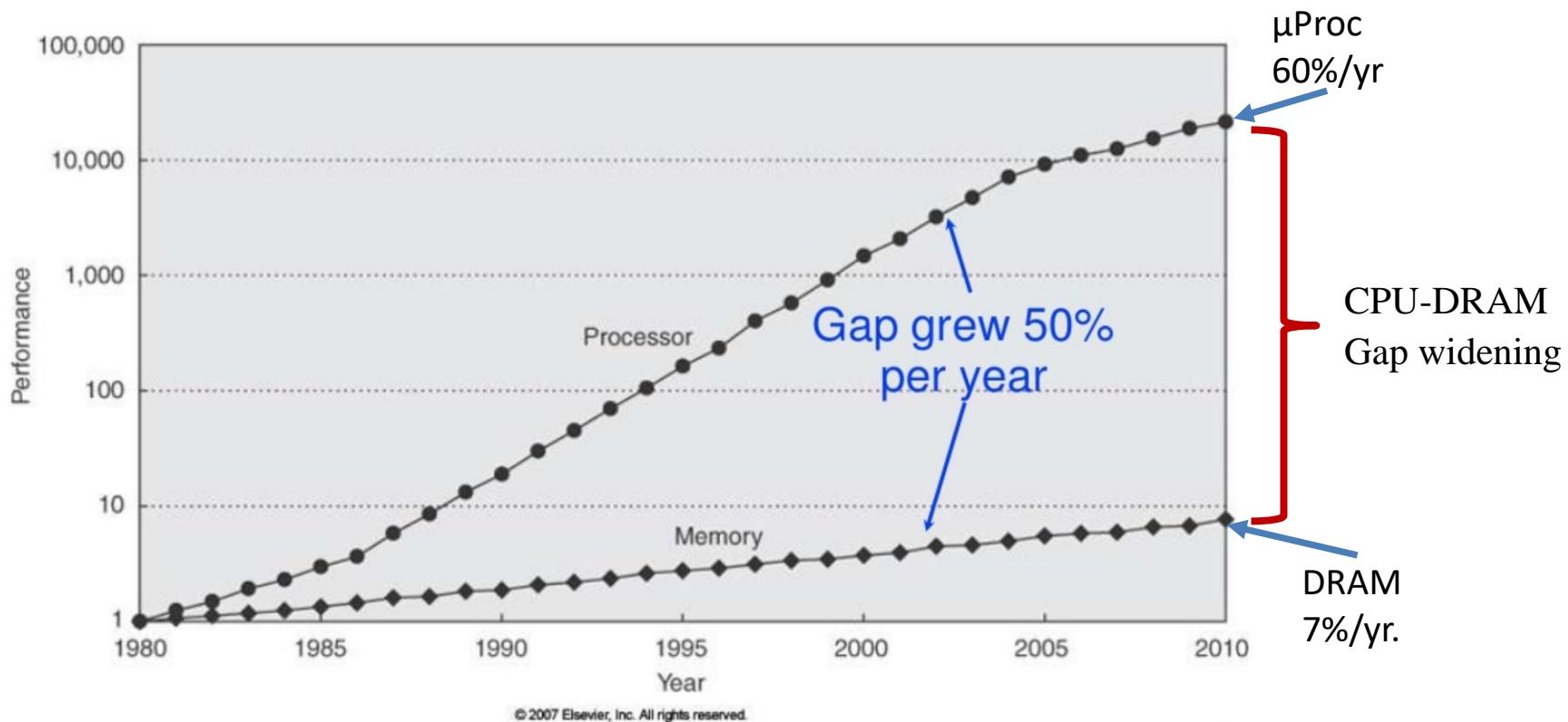
Sources: Intel; press reports; Bob Colwell; Linley Group; IB Consulting; *The Economist*

\*Maximum safe power consumption



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten  
New plot and data collected for 2010-2017 by K. Rupp

# Question: Who Cares About the Memory Hierarchy?



## Implications:

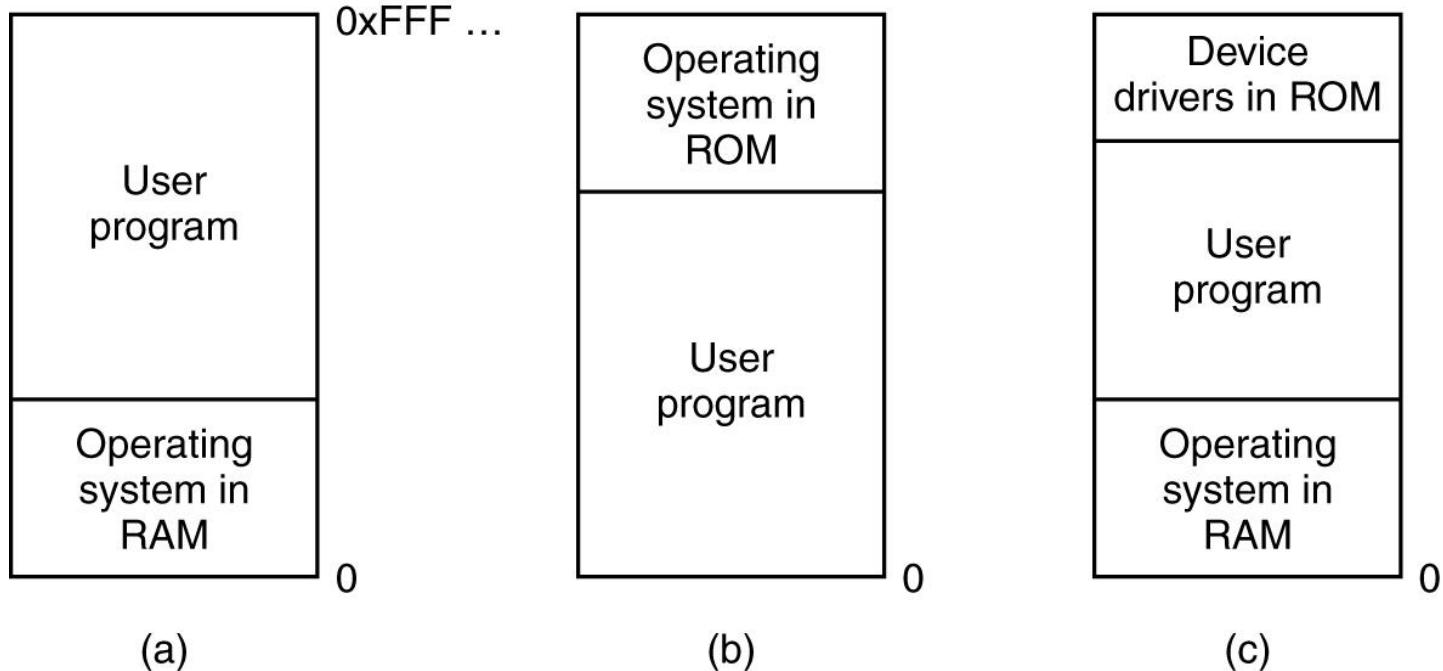
- Longer per cycle memory access times → Memory seems “further away”
- New Technologies: SDR → DDR-1/2/3/4/5 → QDR (technologies)  
Still problem widens

# Memory Abstraction

- The hardware and OS memory manager makes you see the memory in your process as a single contiguous entity
- How do they do that?
  - Abstraction
- Multiple processes of the same program see the same address space (in principle).
  - Addresses start at 0 ..  $2^{^N}$  ( N=32 or N=64)

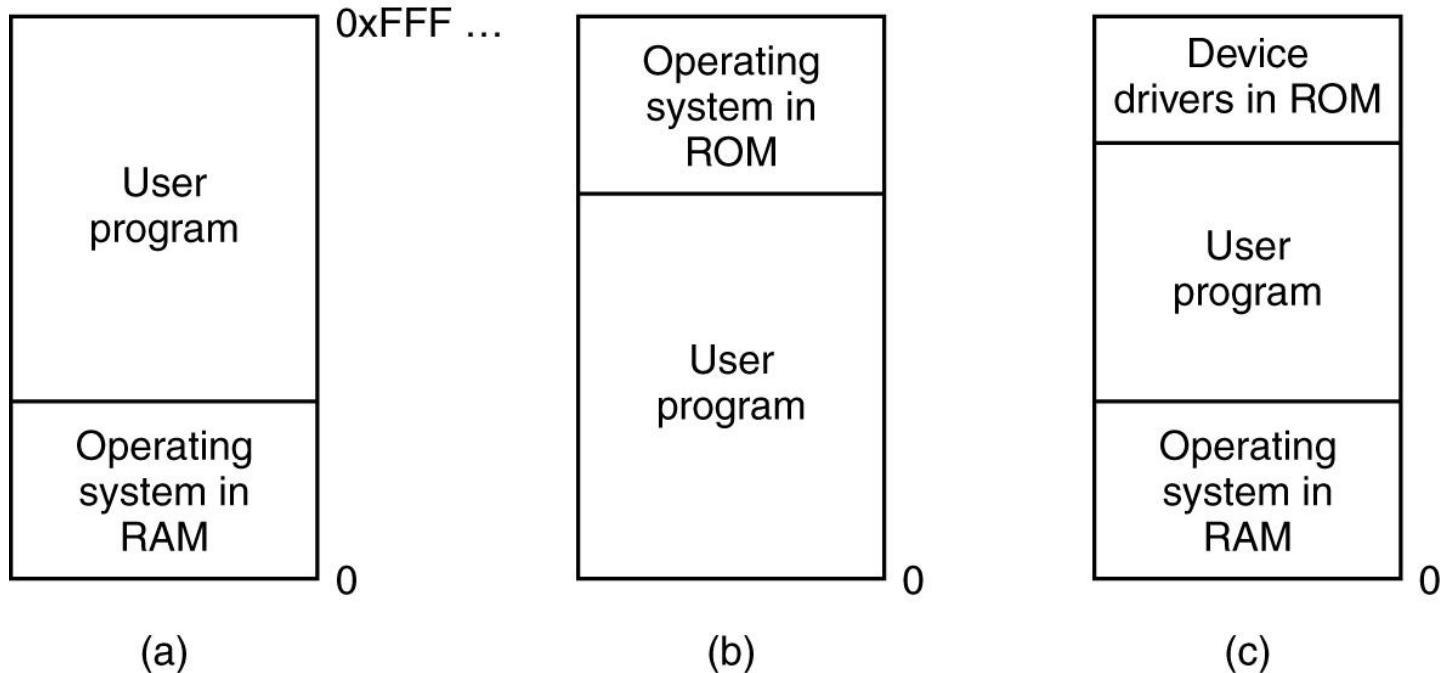
Is abstraction really necessary?

# No Memory Abstraction



**Even with no abstraction, we can have several setups!**

# No Memory Abstraction



Only one process at a time can be running  
(remember threads share the address space of their process)

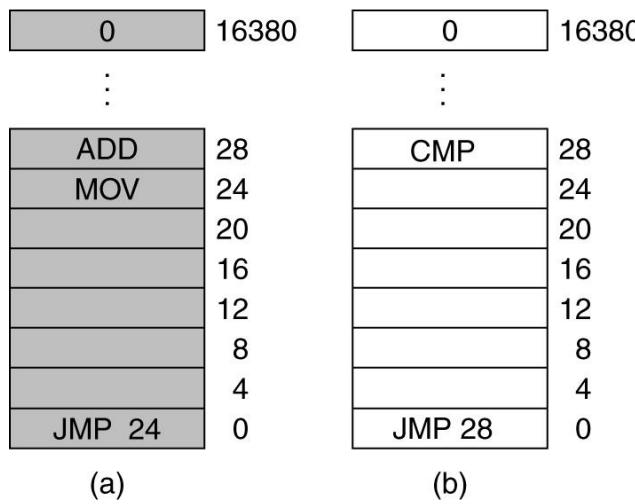
# No Memory Abstraction

- What if we want to run multiple programs?
  - OS saves entire memory on disk
  - OS brings next program
  - OS runs next program
- We can use swapping to run multiple programs concurrently
  - Memory divided into blocks
  - Each block assigned protection bits
  - Program status word contains the same bits
  - Hardware needs to support this
  - Example: IBM 360

Swapping

# No Memory Abstraction (in the presence of multiple processes)

- Processes access physical memory directly



# No Memory Abstraction (in the presence of multiple processes)

- Processes access physical memory directly, so they need to be relocated in order to not overlap in physical memory.

0	16380
:	

ADD	28
MOV	24
	20
	16
	12
	8
	4
JMP 24	0

(a)

0	16380
:	

CMP	28
	24
	20
	16
	12
	8
	4
JMP 28	0

(b)

0	32764
:	
CMP	16412
	16408
	16404
	16400
	16396
	16392
	16388
JMP 28	16384
0	16380

:

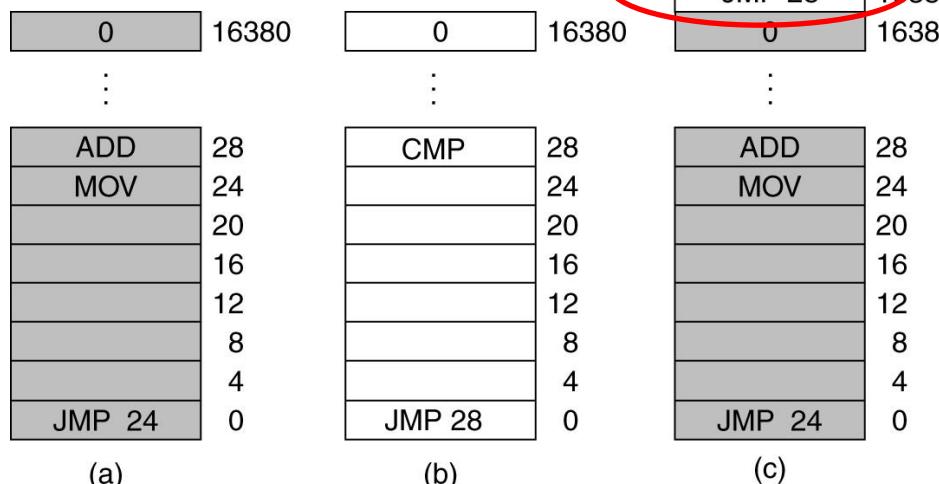
ADD	28
MOV	24
	20
	16
	12
	8
	4
JMP 24	0

(c)

Using absolute address is wrong here

# No Memory Abstraction (in the presence of multiple processes)

- Processes access physical memory directly, so they need to be relocated in order to not overlap in physical memory.
- Can be done at program load time
- Bad idea:
  - very slow
  - Require extra info from program



Using absolute  
address is wrong here

Bottom line: Memory abstraction is needed!

# Memory Abstraction

- To allow several programs to co-exist in memory we need
  - Protection
  - Relocation
  - Sharing
  - Logical organization
  - Physical organization
- A new abstraction for memory:
  - Address Space
- Definition:

Address space = set of addresses that a process can use to address memory

# Protection

- Processes need to acquire permission to reference memory locations for reading or writing purposes
- Location of a program in main memory is unpredictable
- Memory references (load / store) generated by a process must be checked at run time

# Relocation

- Programmers typically do not know in advance which other programs will be resident in main memory at the time of execution of their program
- Active processes need to be able to be swapped in and out of main memory in order to maximize processor utilization
- Specifying that a process must be placed in the same memory region when it is swapped back in would be limiting
  - may need to relocate the process to a different area of memory

# Sharing

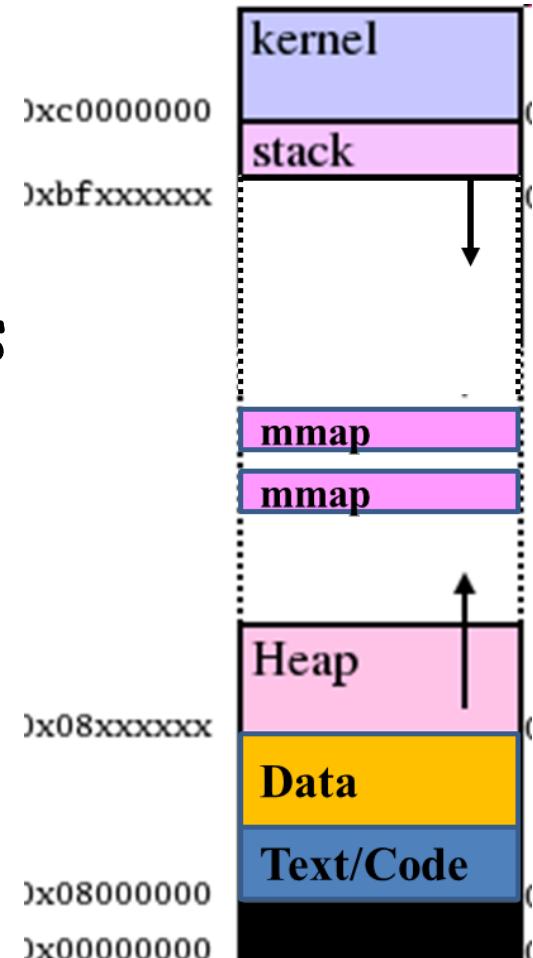
- It is advantageous to allow each process access to the same copy of the program rather than have their own separate copy
- Memory management must allow controlled access to shared areas of memory without compromising protection

# Logical Organization

- We see memory as linear one-dimensional address space.
- A program = code + data + ... = modules
- Those modules must be organized in that logical address space

# Address Space

- Defines where sections of data and code are located in 32 or 64 address space
- Defines protection of such sections  
*ReadOnly, ReadWrite, Execute*
- Confined “private” addressing concept  
→ requires form of address virtualization



# Physical Organization

- Memory is really a hierarchy
  - Several levels of caches
  - Main memory
  - Disk
- Managing the different modules of different programs in such a way as:
  - To give illusion of the logical organization
  - To make the best use of the above hierarchy

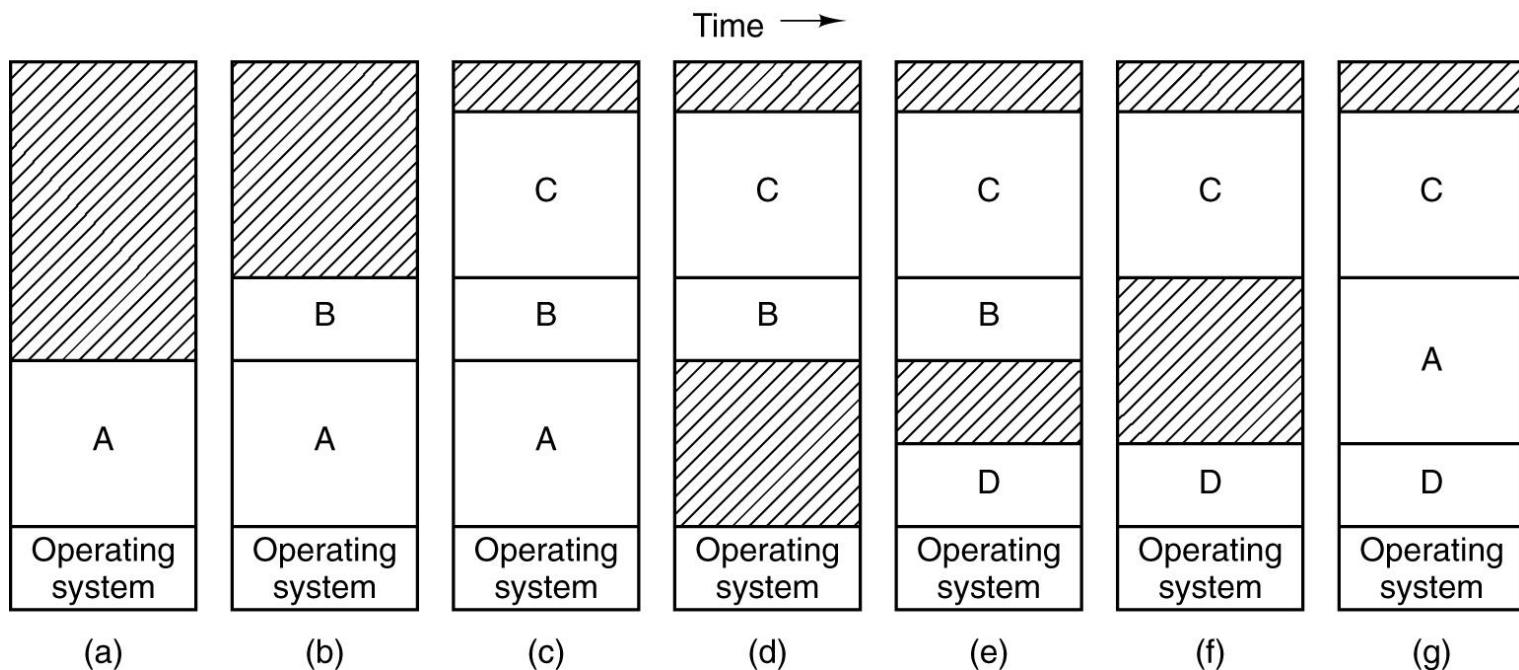
# Address Space: Base and Limit

- Map each process address space onto a different part of physical memory
- Two registers: Base and Limit
  - Base: start address of a program in physical memory
  - Limit: length of the program
- For every memory access
  - Base is added to the address
  - Result compared to Limit
- Only OS can modify Base and Limit

# Address Space: Base and Limit

- Need to add and compare for each memory address:
  - can be done in HW
  - doesn't significantly add to latency
- What if memory space is not enough for all programs?
  - We may need to **swap** some programs out of the memory.
  - remember swapping means moving entire program to disk → expensive

# Swapping

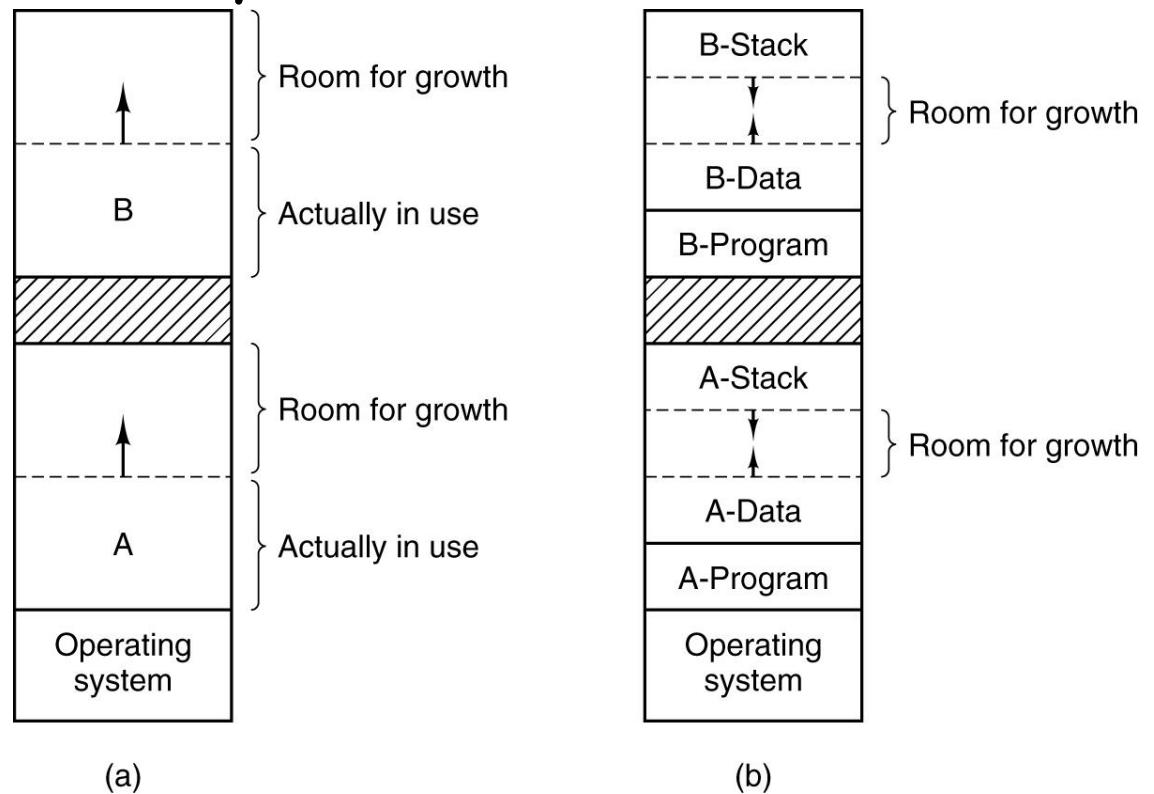


# Swapping

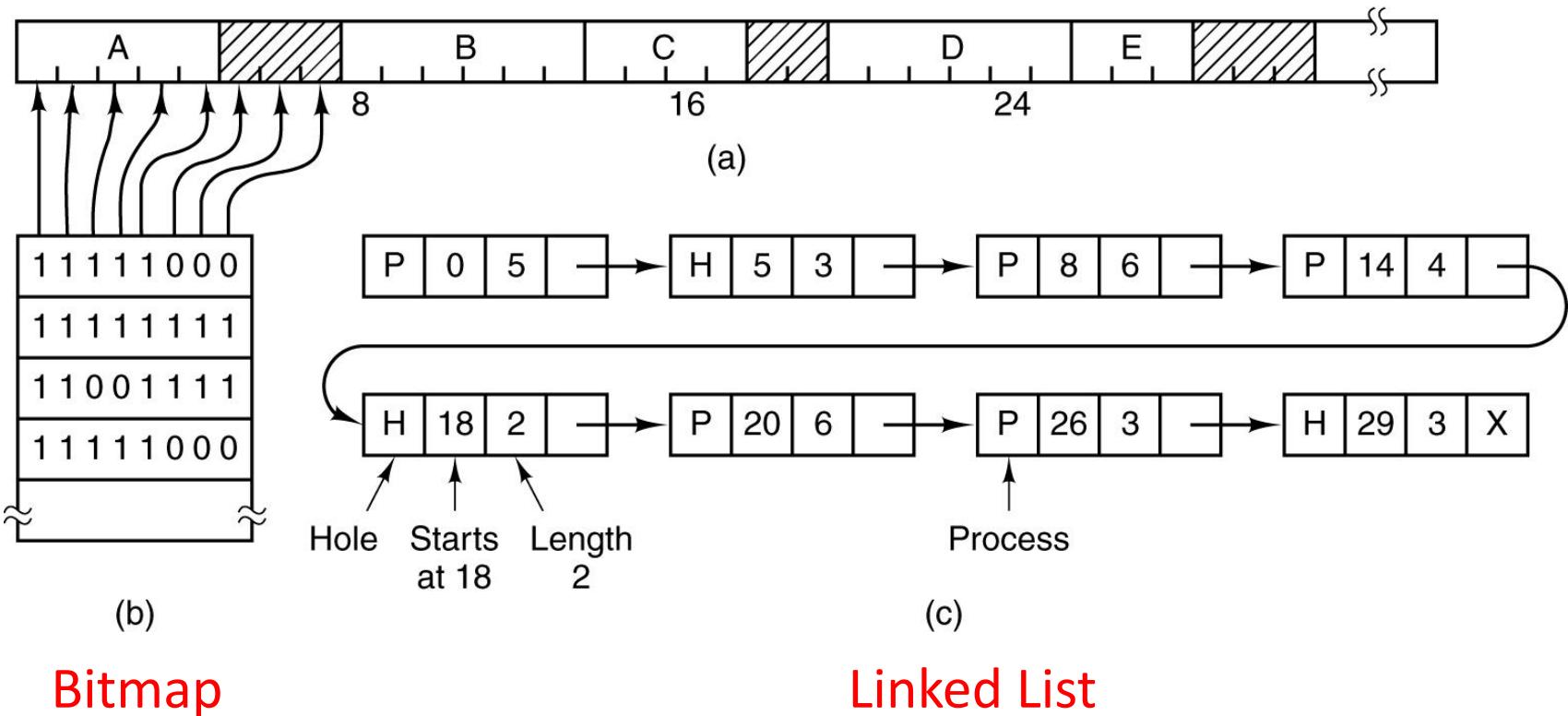
- Programs move in and out of memory
- Holes are created
- Holes can be combined -> memory compaction
- What if a process needs more memory?
  - If a hole is adjacent to the process, it is allocated to it
  - Process has to be moved to a bigger hole
  - Process suspended till enough memory is there

# Dealing with unknown memory requirements by processes

- Anticipate growth of usable address space and leave gaps.
- Waste of phys memory as it will be allocated whether used or not.

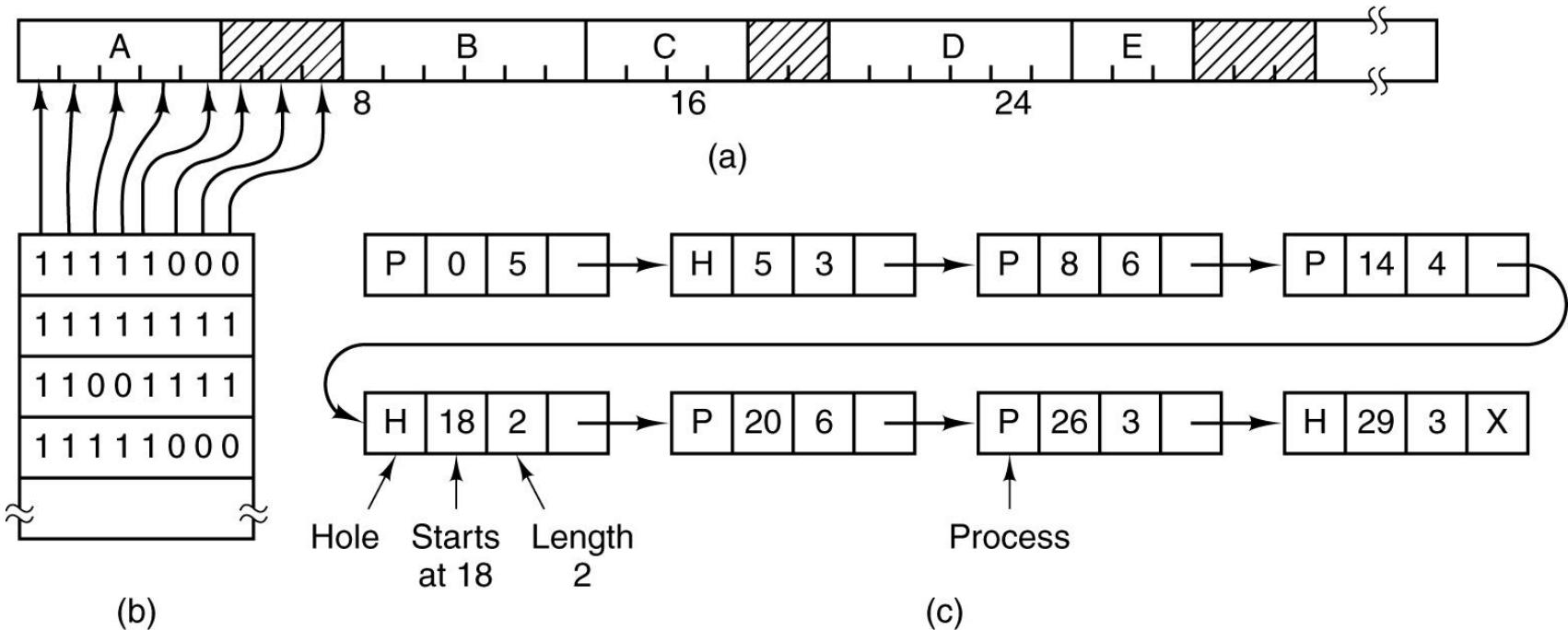


# Managing Free Memory



- These are universal methods used in OS and applications. Other methods employ HEAPS.

# Managing Free Memory



Bitmap

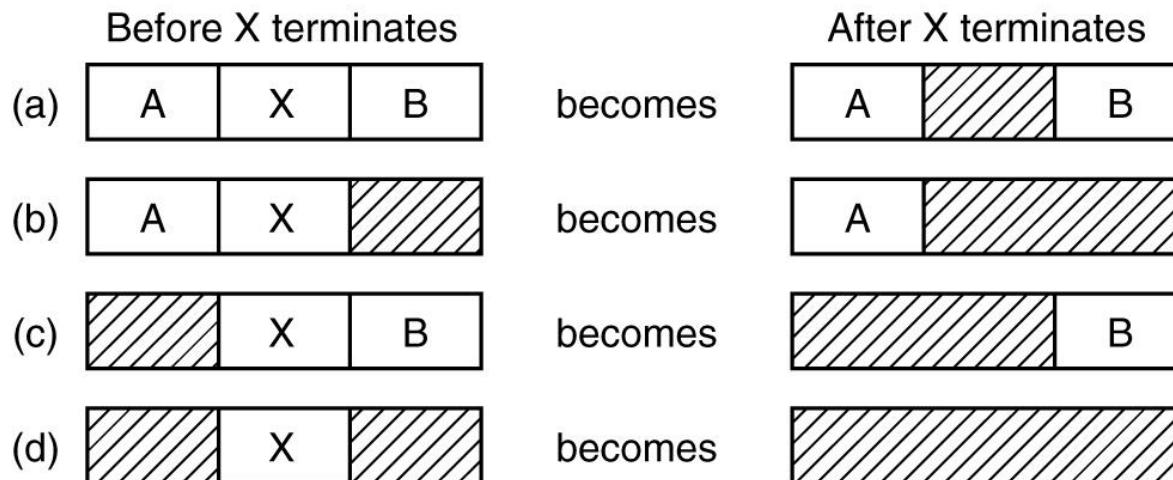


Slow: To find k-consecutive 0s for a new process

Linked List

# Managing Free Memory: Linked List

- Linked list of allocated and free memory segments
- More convenient be double-linked lists



# Managing Free Memory: Which Available Piece to pick ?

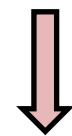
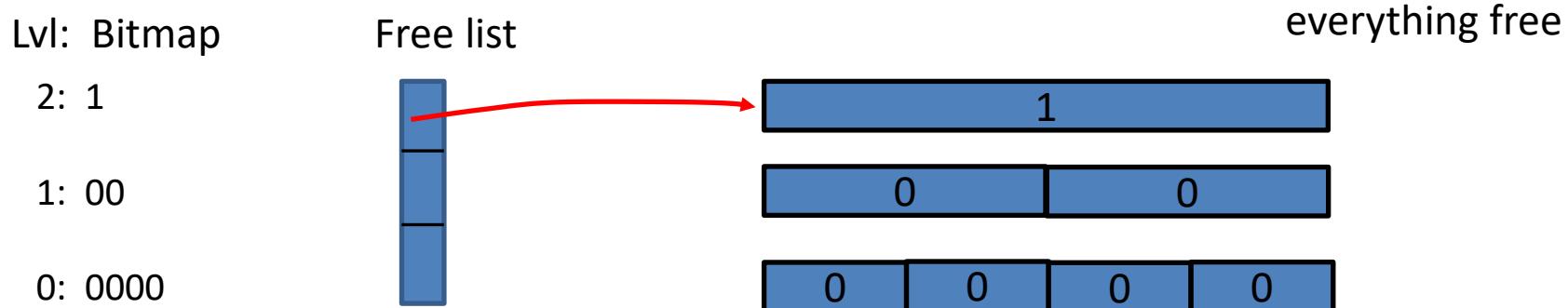
- How to allocate?
  - First fit
  - Best fit
  - Next fit
  - Worst fit
  - ...
- Each has their philosophy to why and what their pros and cons are.  
At the end you must weight efficiency of space (fragmentation) and time (to manage)

# Buddy Algorithm

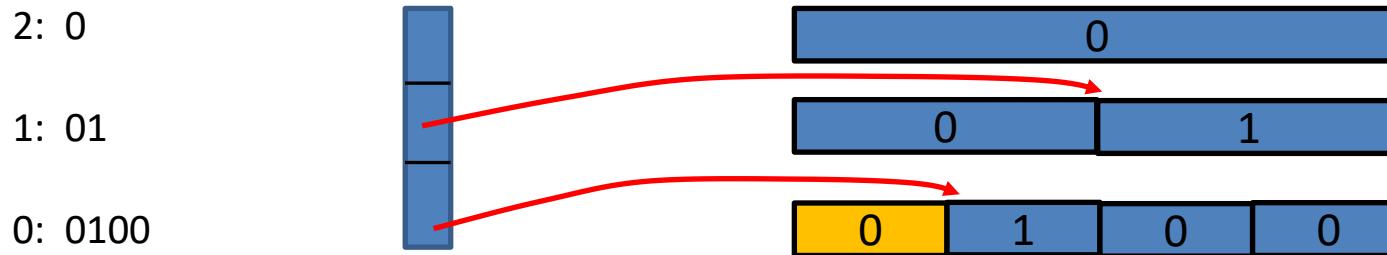
- Considers blocks of memory only as  $2^N$
- Potential for fragmentation (drawback)
- If no block of a size is available it splits higher blocks into smaller blocks
- Easy to implement and fast  
 $O(\log_2(\text{MaxBlockSz}/\text{MinBlockSize}))$   
e.g. 4K .. 128B =  $2^{12-7} = 5$  steps

# Buddy Algorithm

- Allocation at level 0



```
int sz = somesize corresponding to <= 2^(logminsz+0);  
void *ptr = kmalloc(sz);
```



0 = notavail ; 1 avail

\* In-use

# Buddy Algorithm

- Freeing "X" at level 2 leading to coalescing

Lvl: Bitmap

2: 0

1: 00

0: 1001

Free list



kfree(X,sz)

void \*X , \*Y;

From X determine bitofs → determine buddy  
if (bitofs&1) buddy= bitofs + (bitofs&1) ? -1 : +1;  
parbitofs = bitofs >> 1;  
if (buddy is free) merge\_up else add to free;

0: 0

1: 10

2: 0001



merged again  
into level 1

# What are really the problems?

- Memory requirement unknown for most apps
- Not enough memory
  - Having enough memory is not possible with current technology
    - How do you determine "enough"?
- Exploit and enforce one condition:

Processor does not execute or access anything that is not in the memory  
(how would that even be possible ?)

Enforce transparently ( user is not involved )

# Memory Management Techniques

- Memory management brings processes into main memory for execution by the processor
  - involves **virtual memory**
  - based on paging and **segmentation**

# But We Can See That ...

- All memory references are **logical addresses** in a process's address space that are dynamically translated into **physical addresses** at run time
- An address space may be broken up into a number of pieces that don't need to be contiguously located in main memory during execution.

So:

**It is not necessary that all of the pieces of an address space be in main memory during execution. Only the ones that I am “currently” accessing**

# Scientific Definition of Virtual Memory

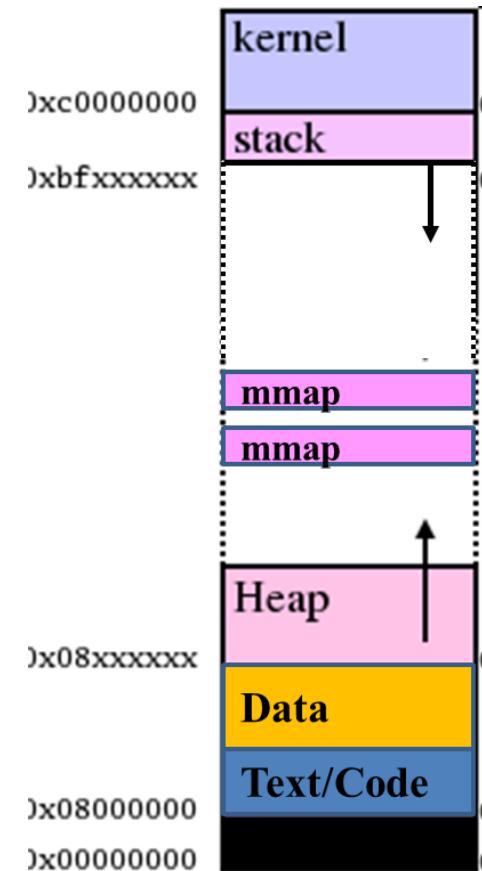
Mapping from  
logical (virtual) address space  
(user / process perspective)

to

physical address space  
(hardware perspective)

# Address Space (reminder)

- Defines where sections of data and code are located in 32 or 64 address space
- Defines protection of such sections
- **ReadOnly, ReadWrite, Execute**
- Confined “private” addressing concept
  - → requires form of address virtualization



# The Story

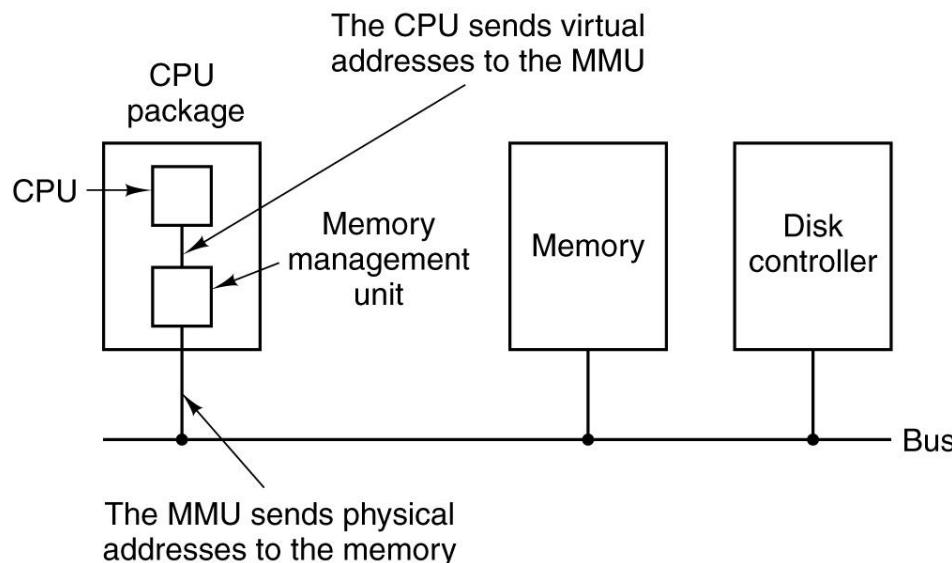
1. Operating system brings into main memory a few pieces of the program.
2. An exception is generated when an address is needed that is not in main memory (will see how).
3. Operating system places the process in a blocking state.
4. Operating system allocates memory and optionally issues a disk I/O Read request to retrieve the content of that memory
5. Another process is dispatched to run while the disk I/O takes place.
6. An interrupt is issued when disk I/O is complete, which causes the operating system to place the affected process in the Ready state
7. Piece of process that contains the logical address has been brought into main memory.

# The Story Questions over Questions

1. Operating system brings into main memory a few pieces of the program.  
*What do you mean by "pieces"?*
2. An exception is generated when an address is needed that is not in main memory (will see how).  
*How do you know it isn't in memory?*
3. Operating system places the process in a blocking state.
4. Operating system allocates memory and optionally issues a disk I/O Read request to retrieve the content of that memory or set it otherwise.  
*How do I know which one?*  
*What if memory is full and I can't allocate anymore ?*
5. Another process is dispatched to run while the disk I/O takes place.
6. An interrupt is issued when disk I/O is complete, which causes the operating system to place the affected process in the Ready state
7. Piece of process that contains the logical address has been brought into main memory.

# Virtual Memory

- Each program has its own **address space**
- This address space is divided into **pages** (e.g 4KB)
- Pages are mapped into physical memory chunks (called **frames**)
- By definition: `sizeof(page) == sizeof(frame)`  
(the size is determined by the hardware manufacturer)



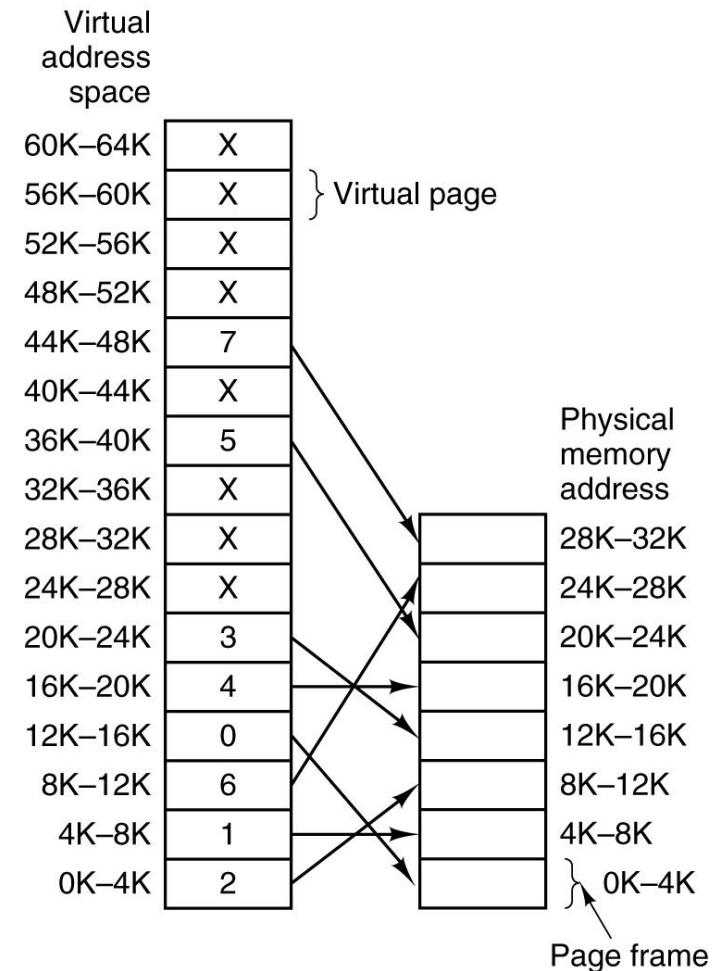
# Virtual Memory

Single Process / Address Space view first

Allows to have larger virtual address space than physical memory available

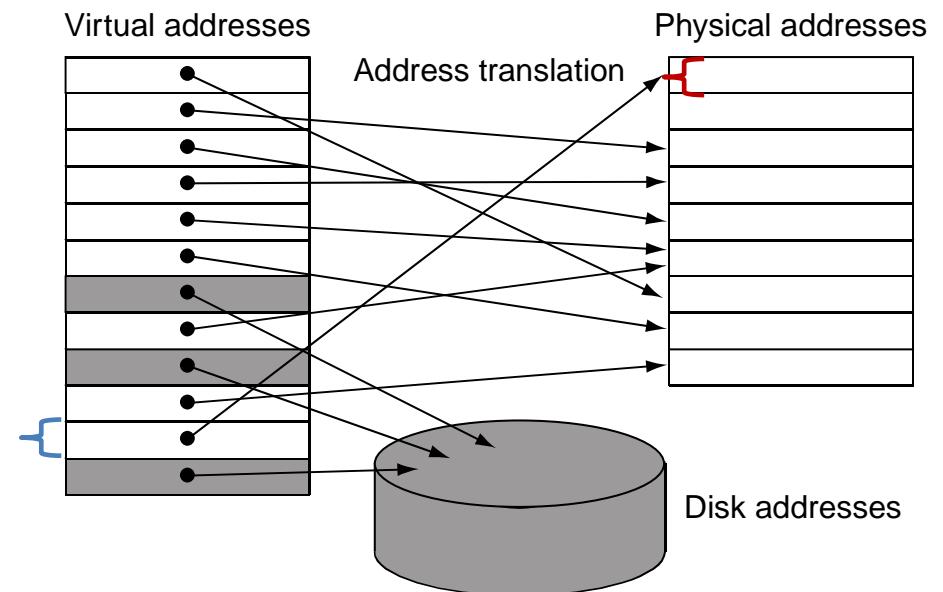
"X" → when you access a virtual page you get a page fault (see prior story) that needs to be resolved first

All other accesses are at hardware speed and don't require any OS intervention (with the exception of protection enforcement)



# Virtual Memory

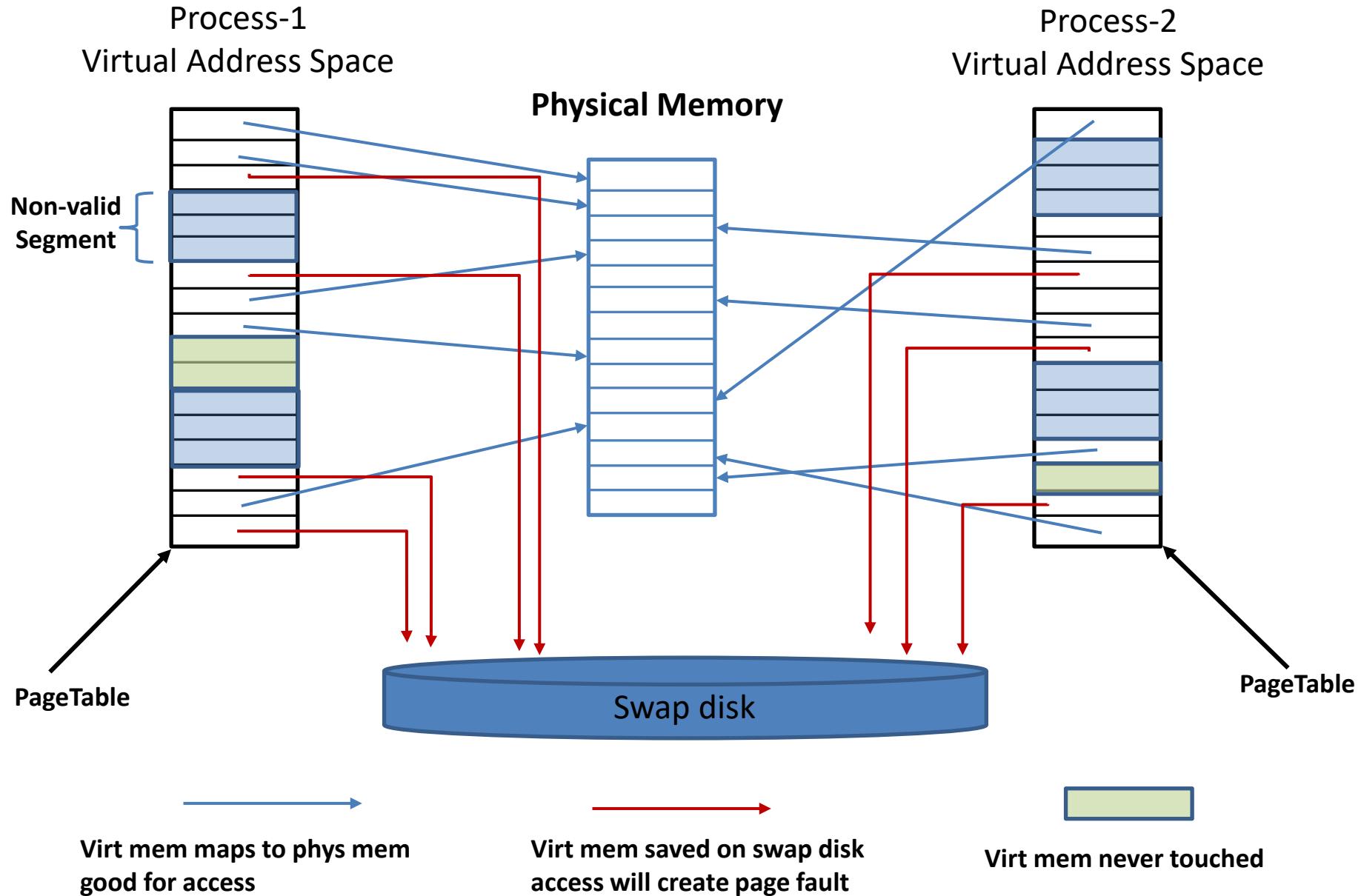
- Main memory can act as a cache for the secondary storage (disk)



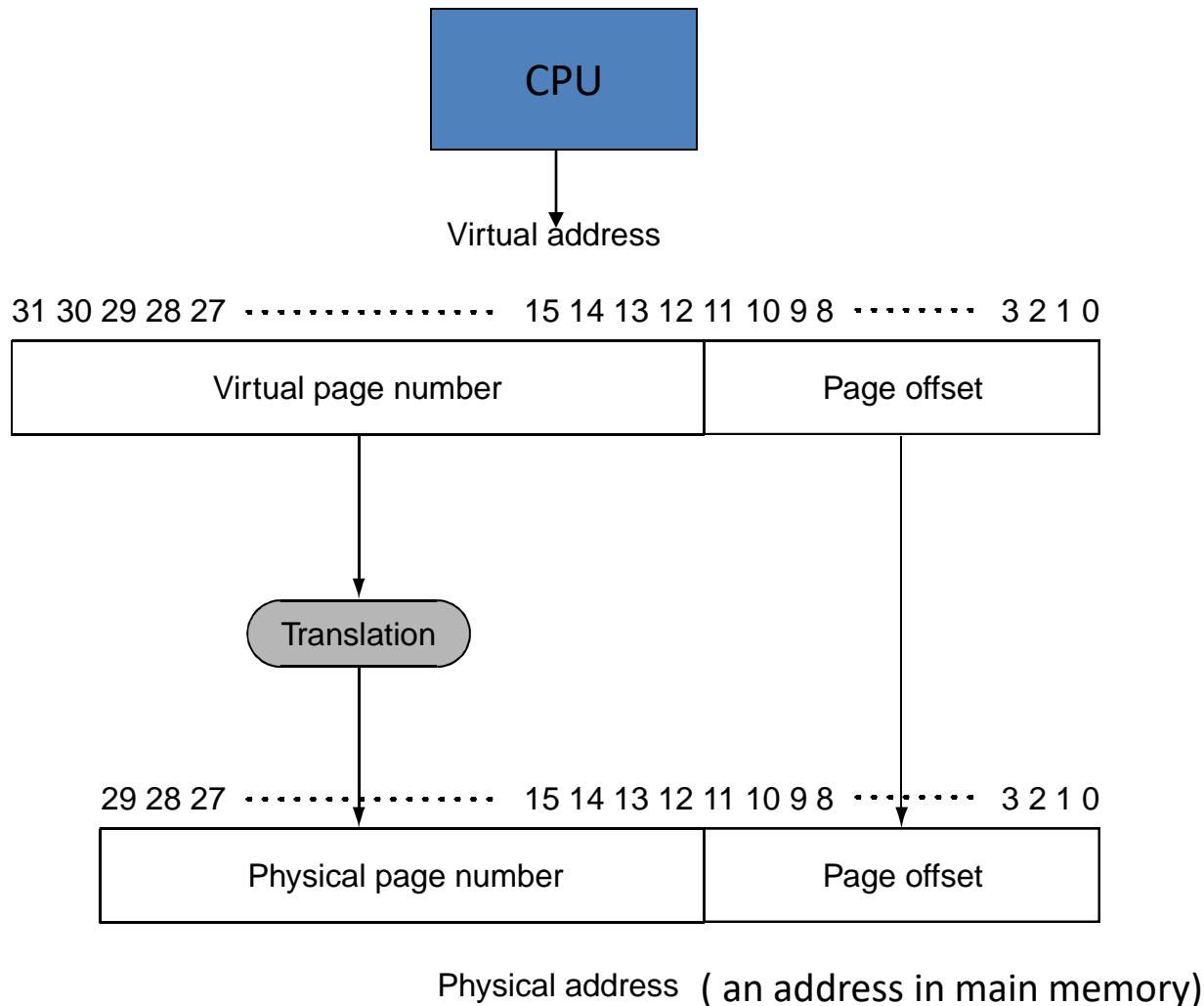
- Advantages:
  - illusion of having more physical memory
  - program relocation
  - protection

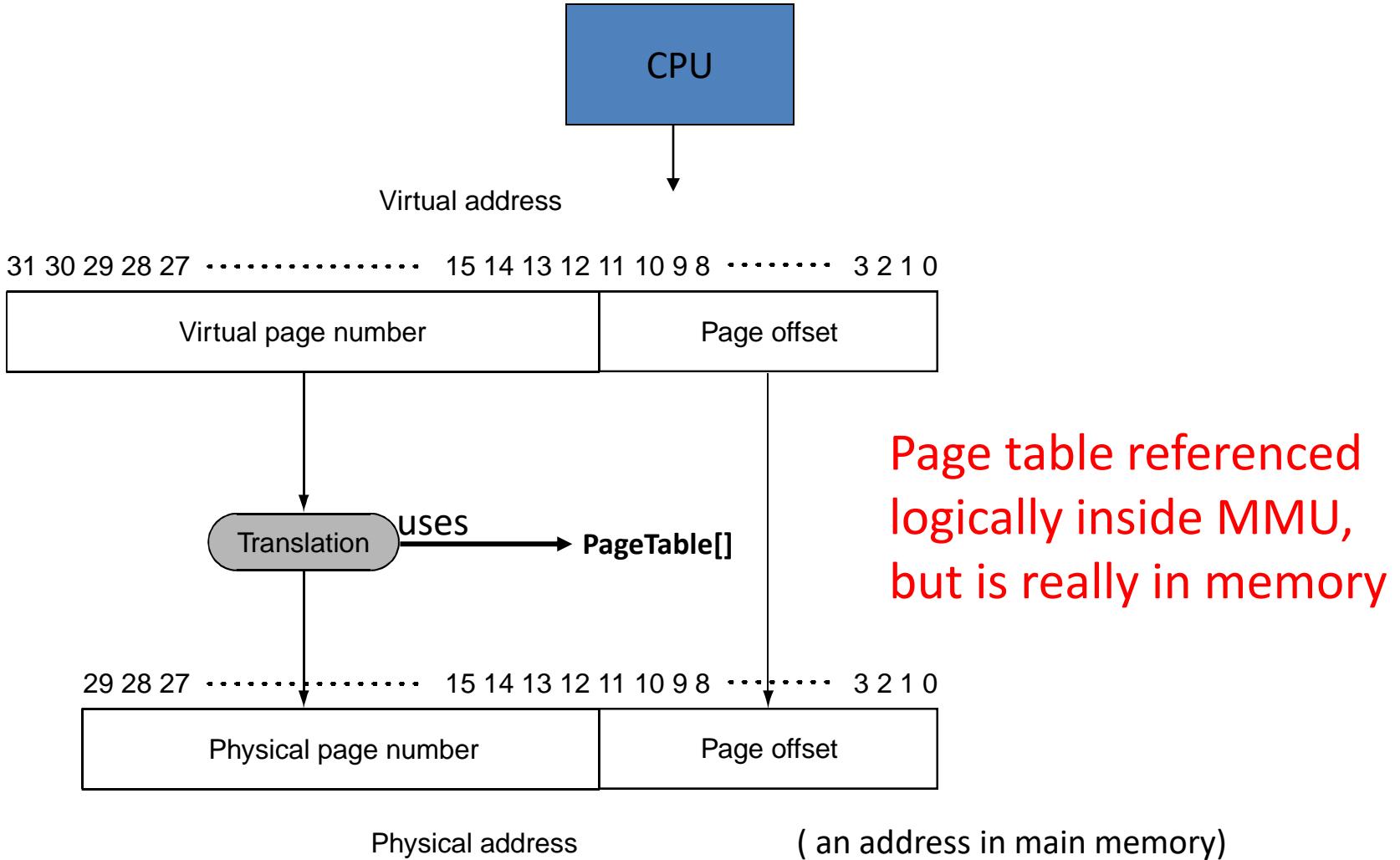
Block-of-mem  
Virtual  
Physical

# Virtual Memory



# How does address translation work?

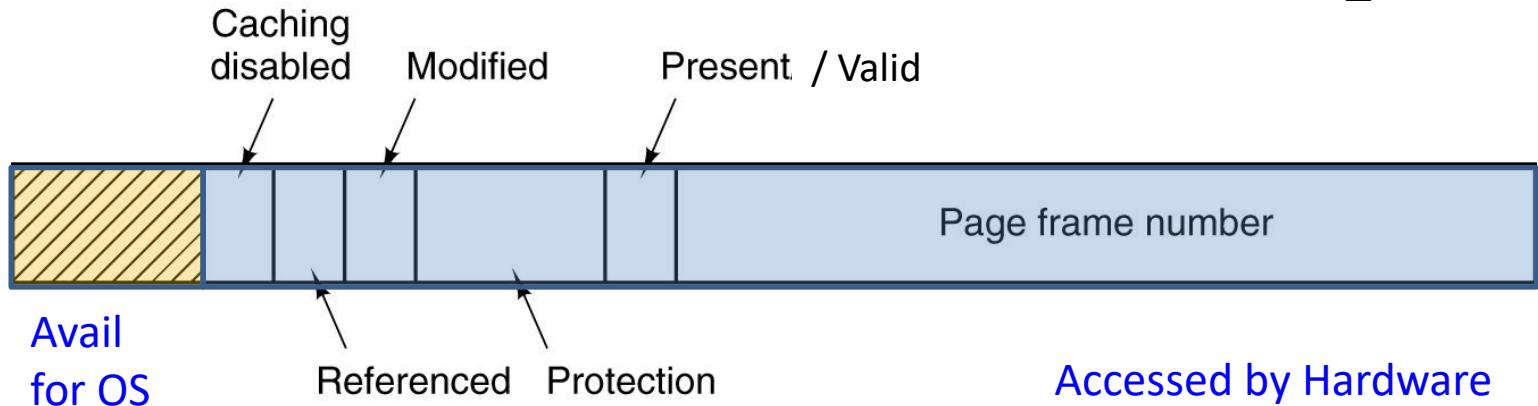




MMU = Memory Management Unit

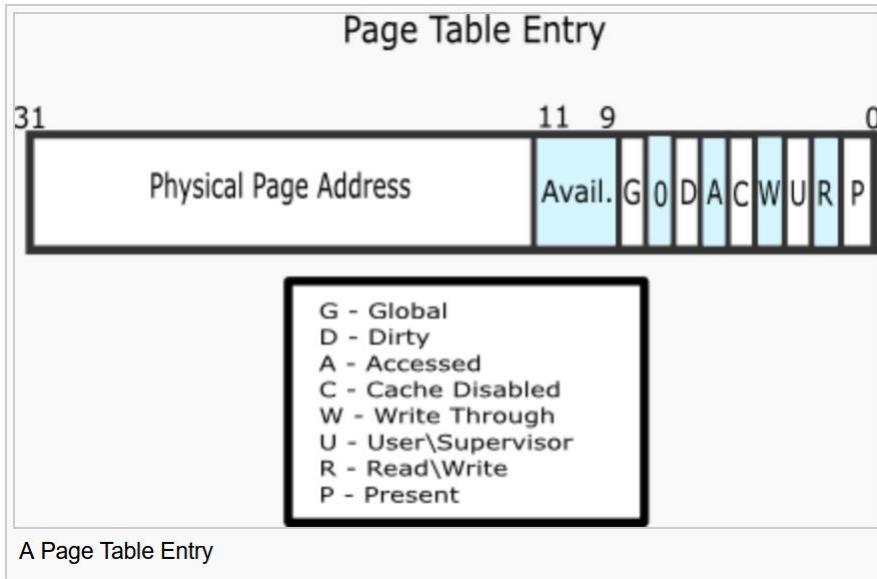
# Structure of a Page Table Entry

```
struct PTE page_table[];
```



- **Present/Valid bit:** '1' if the values in this entry is valid, otherwise translation is invalid and a pagefault exception will be raised
- **Frame Number:** this is the physical frame that is accessed based on the translation.
- **Protection bits:** 'kernel' + 'w' specifies who and what can be done on this page if *kernel bit* is set then only the kernel can translate this page. If user accesses the page a 'privilege exception' will be raised.  
If *writeprotect bit* is set the page can only be read (load instruction). If attempted write (store instruction), a write protection exception is raised.
- **Reference bit:** every time the page is accessed (load or store), the reference bit is set.
- **Modified bit:** every time the page is written to (store), the modified bit is set.
- **Caching Disabled:** required to access I/O devices, otherwise their content is in cpu cache
- Other unused bits typically available for the operating system to "remember" information in

# X86 examples for PTE



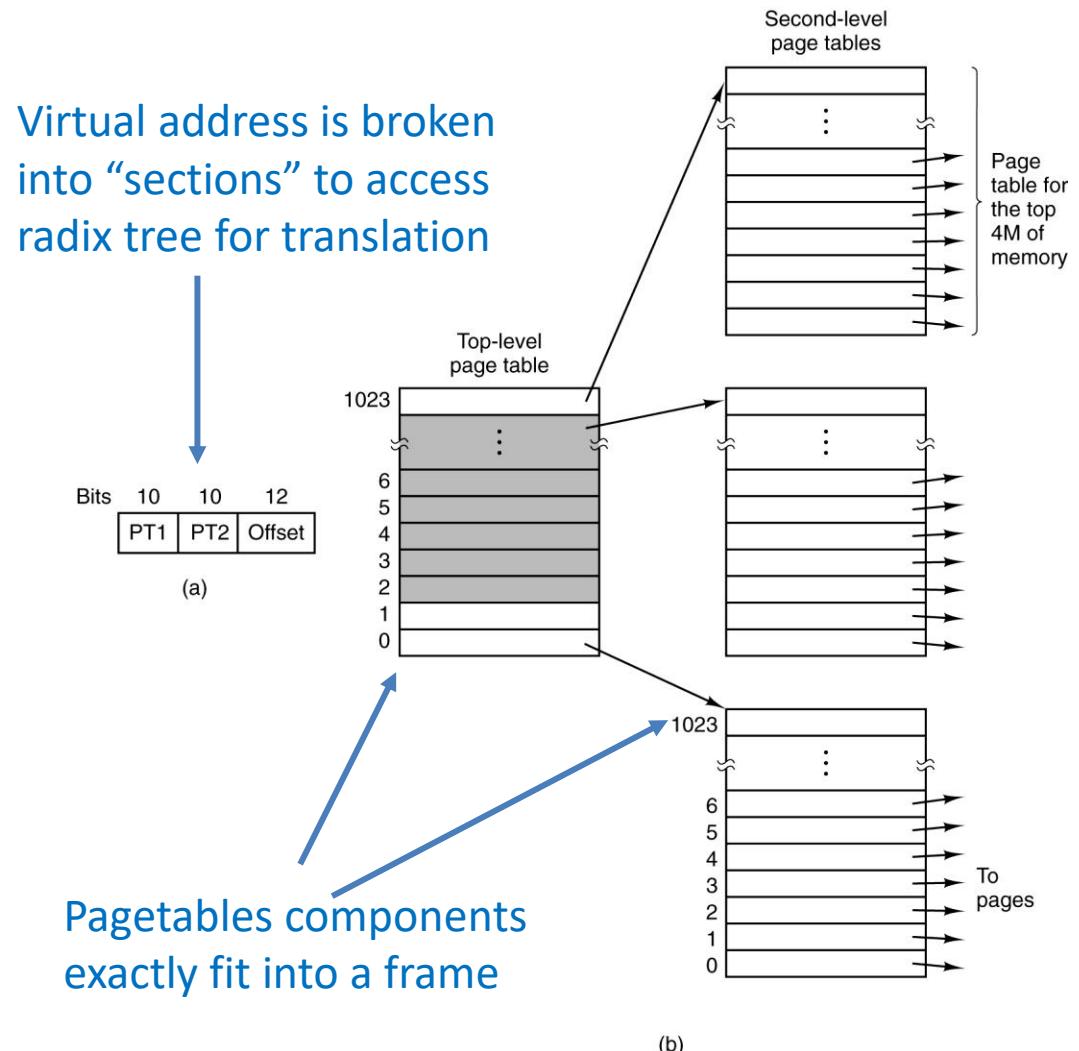
32-bit

```
typedef struct
{
    uint64 present          :1;
    uint64 writeable        :1;
    uint64 user_access      :1;
    uint64 write_through    :1;
    uint64 cache_disabled   :1;
    uint64 accessed         :1;
    uint64 ignored_3        :1;
    uint64 size              :1; // must be 0
    uint64 ignored_2        :4;
    uint64 page_ppn         :28;
    uint64 reserved_1       :12; // must be 0
    uint64 ignored_1         :11;
    uint64 execution_disabled :1;
} __attribute__((__packed__)) PageMapLevel4Entry;
```

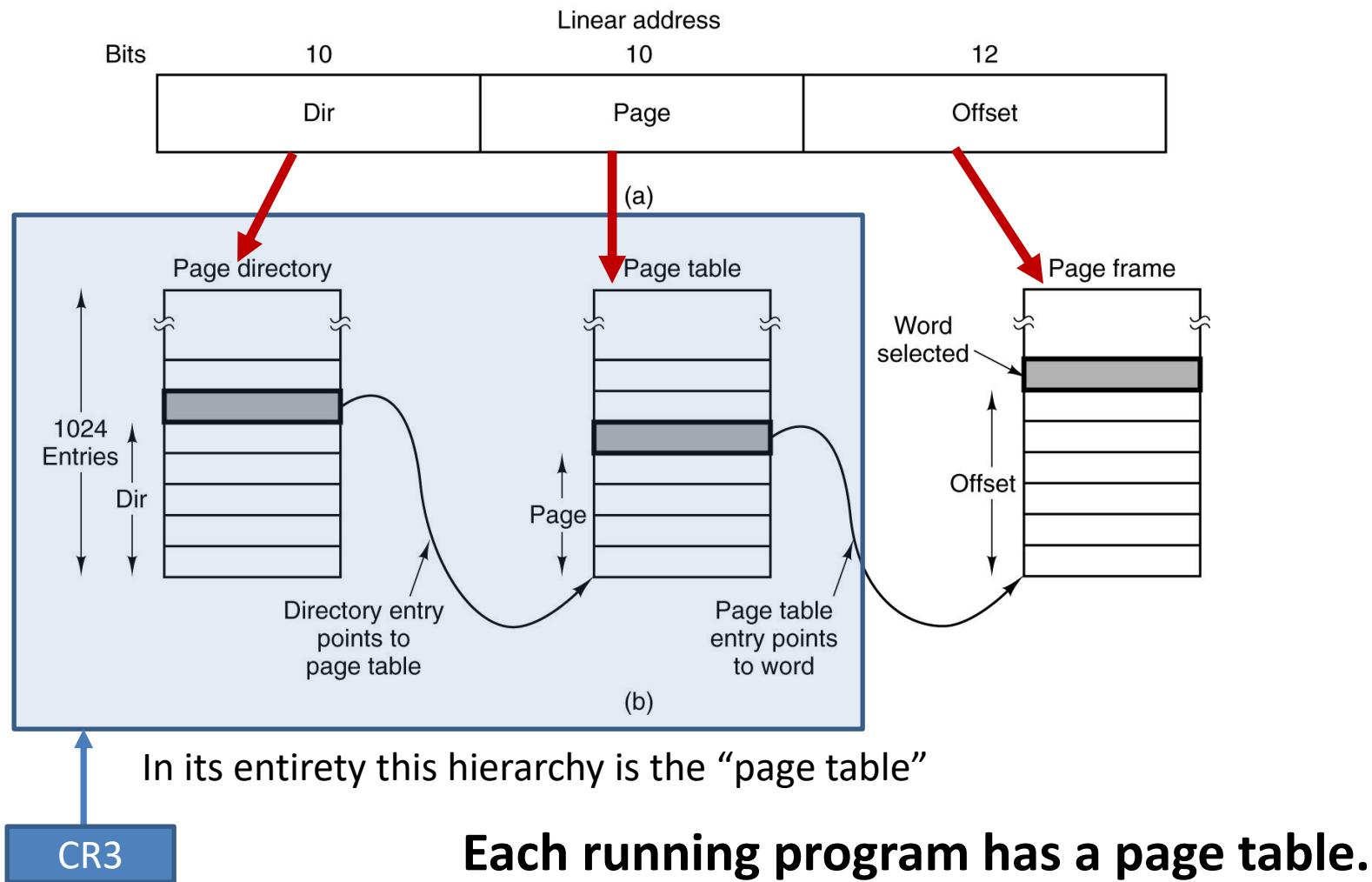
64-bit

# Multi-Level Page Table aka Radix Tree or Hierarchical Page Table

- To reduce storage overhead in case of large memories
- For sparse address spaces (most processes) only few 2<sup>nd</sup> tables required !!!

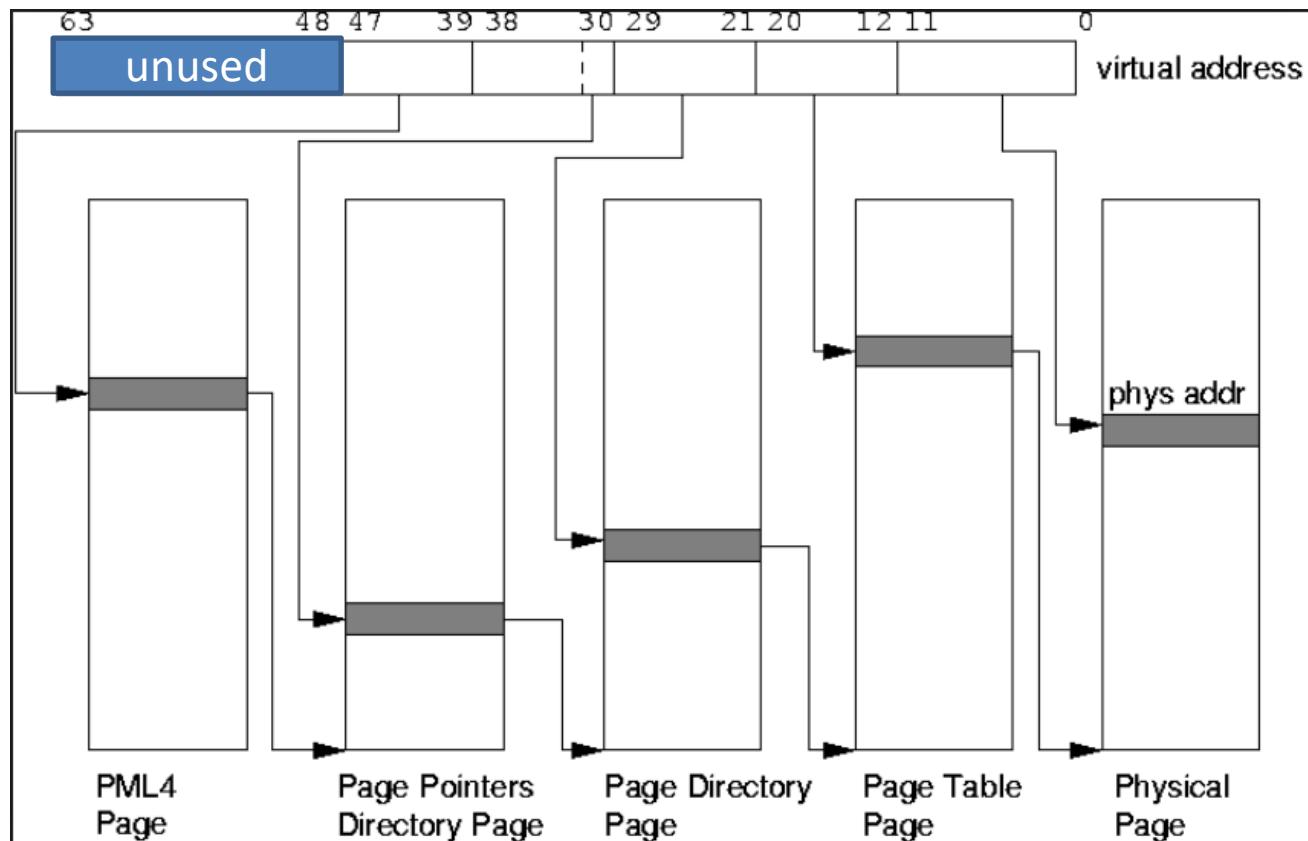


# The Intel Pentium



CR3: Privileged hardware register

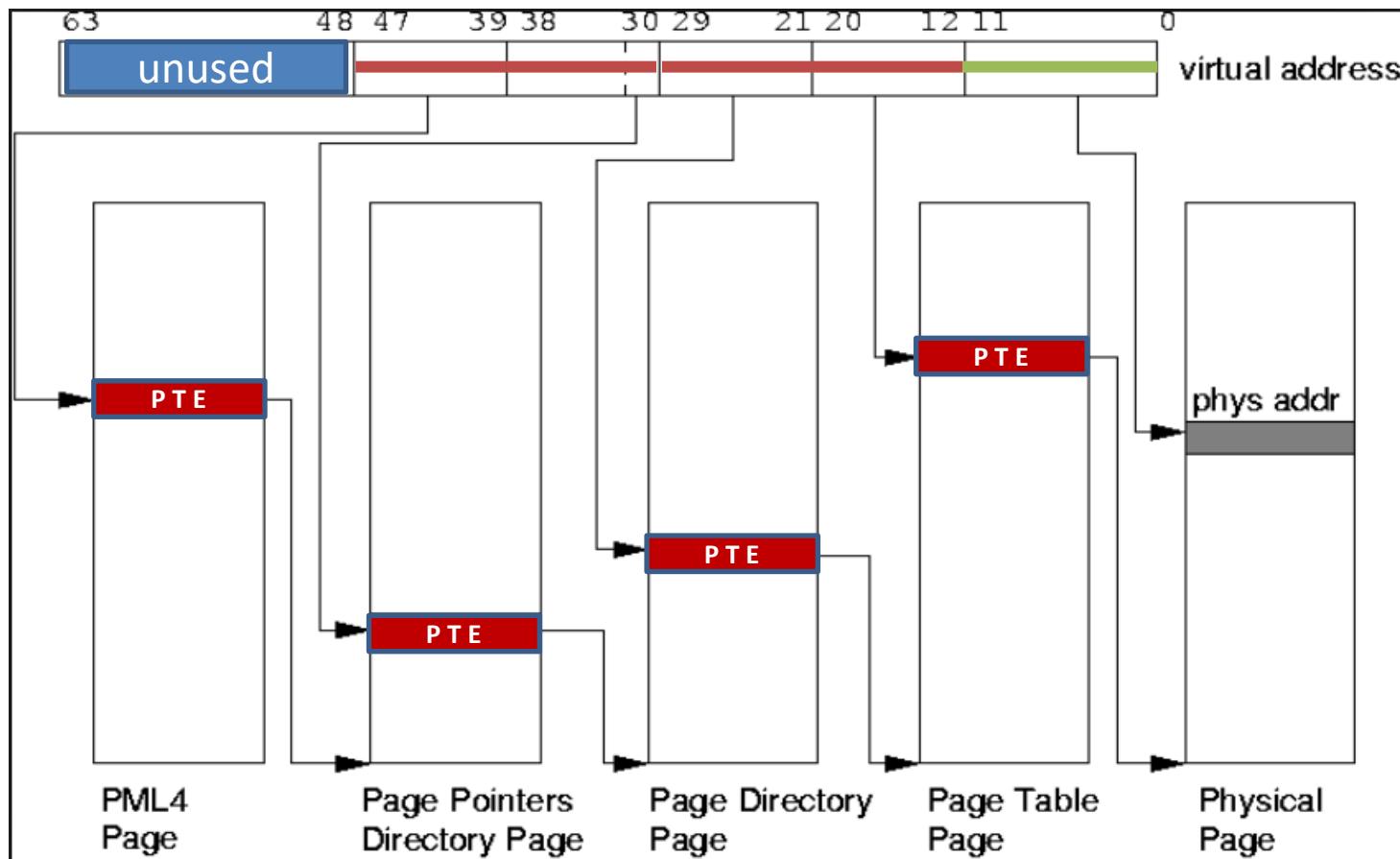
# 4 level PageTable for 64-bit arch



- This is a 48-bit virtual address space architecture.
- OS has to make sure no segment is allocated into high range of address space ( 63-48 bits )
- If bits are set the MMU will raise an exception ( this is really an OS bug then )

# Animation of Hardware

ldw r3, vaddr



# General Formula of Page Table Management

- **Hardware defines the frame size** as  $2^{**N}$
- (virtual) page size is typically equal frame size (otherwise it's a power-of-two multiple, but that is rare and we shall not base on this exception)
- Everything the OS manages is based on pagesize (hence framesize)
- You can compute all the semantics with some basic variables defined by the hardware:
  - Virtual address range ( e.g. 48-bit virtual address, means that the hardware only considers the 48-LSB bits from an virtual address for ld/st, all other raise a SEGV error)
  - Frame size
  - PTE size (e.g. 4byte or 8byte)
  - From there you can determine the number of page table hierarchies, offsets
- Example:
  - Framesize = 4KB → 12bit offset to index into frame 12bits
  - PTE size = 8Bytes → 512 entries in each page table hierarchy 9bits / hierarchy
  - Virtual address range 48-bits: →  $12 + N \cdot 9$  bits must add up to 48 bits  $N=4$  hierarchies
  - ^^^^^^ the hardware does all the indexing as described before
- The OS must create its page table and VMM management following the hardware definition.

# Speeding Up Paging

- Challenges:
  - Mapping virtual to physical address must be fast
  - If address space is large, page table will be large

# Speeding Up Paging

- Challenges:
  - Mapping virtual to physical address must be fast
    - we can not always traverse the page table to get the VA → PA mapping **Translation Lookaside Buffer(TLB)**
  - If address space is large, page table will be large (but remember the sparsity, not fully populated) **Multi-level page table**

# TLB

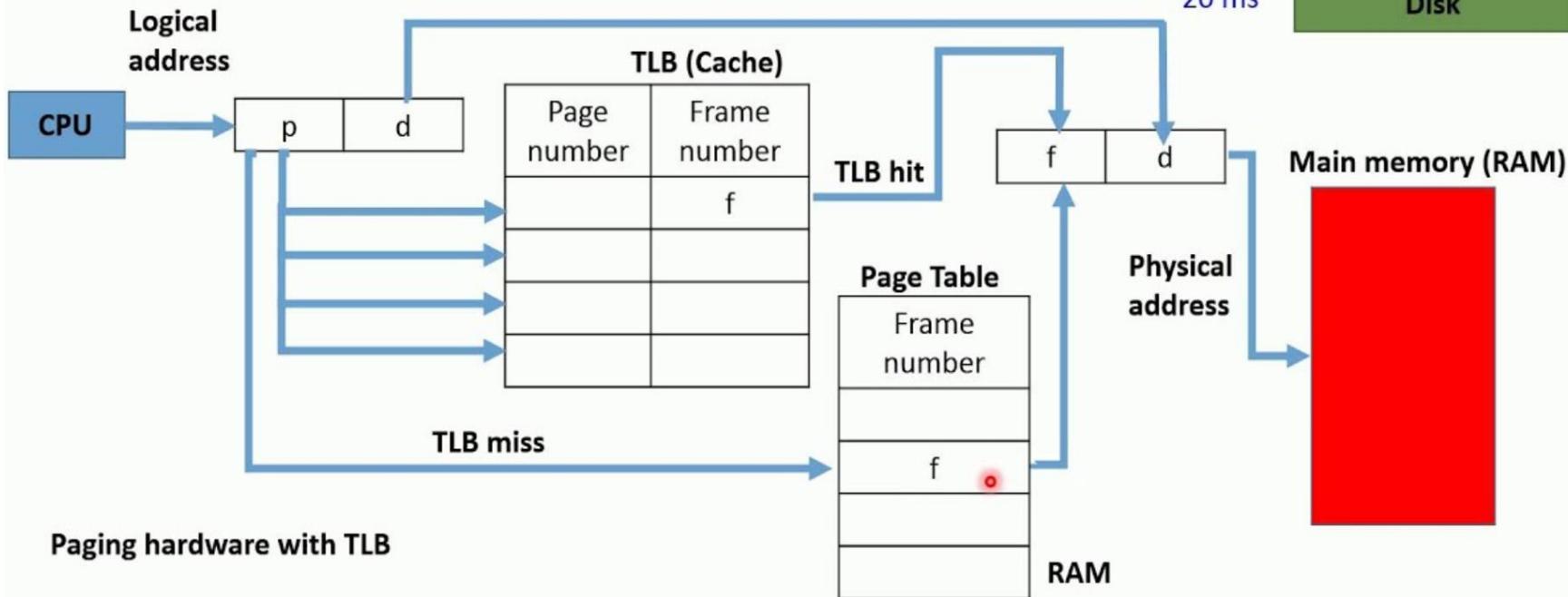
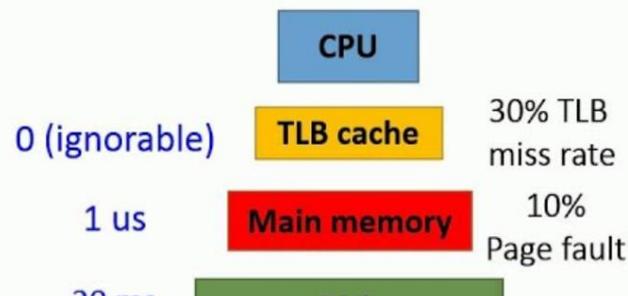
- **Observation:** most programs tend to make a large number of references to a small number of pages over a period of time -> only fraction of the page table is heavily used  
( data and instruction locality !!!)
- TLB
  - Hardware cache inside the MMU
  - **Caches** PageTable translations ( VA -> PA )
  - Maps virtual to physical address without going to the page table (unless there's a miss)

# TLB based translation

- Concept: logical addr -> page table (frame number) -> physical addr

- Q: Given a demand-paging system

- One memory operation is 1 us
- Each memory access through the page table takes two accesses
- Average access time of a page in a paging disk is 20 ms
- TLB hit rate = 70 %, page fault rate of remaining access = 10 %



# TLB

- In case of TLB miss -> MMU accesses page table and load entry from pagetable to TLB
- TLB misses occur more frequently than page faults
- Optimizations
  - Software TLB management
    - Simpler MMU
    - Becomes rare in modern system

# TLB

- Sample content of a TLB
- Size often 256 - 512 entries
- $512 * 4\text{KB} = 2^9 * 2^{12} = 2^{21} =$   
2MB address coverage at any given time

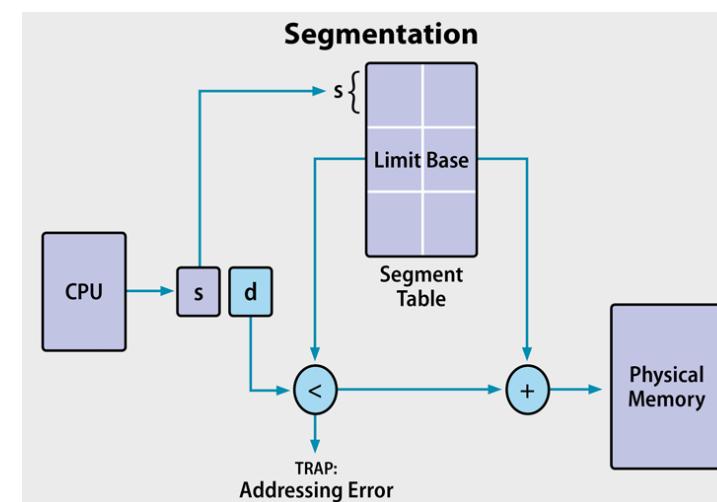
Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

# TLB management

- TLB entries are not written back on access, just record the R and M bits in the PTE (it is a cache after all) upon access
- On TLB capacity miss, if the entry is dirty, write it back to its associated PageTableEntry (PTE)
- On changes to the PageTable, potential entries in the TLB need to be flushed or invalidated
  - “TLB invalidate” e.g. when mmap area disappears.
  - “TLB flush” write back when changes in TLB to PageTable (either global or per address) must be recorded, think when a “TLB invalidate” might be insufficient

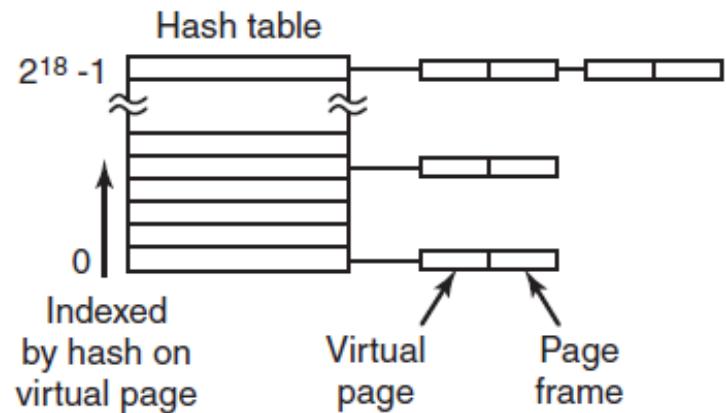
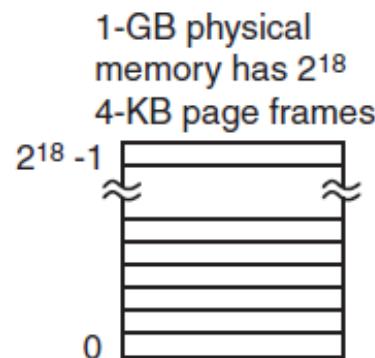
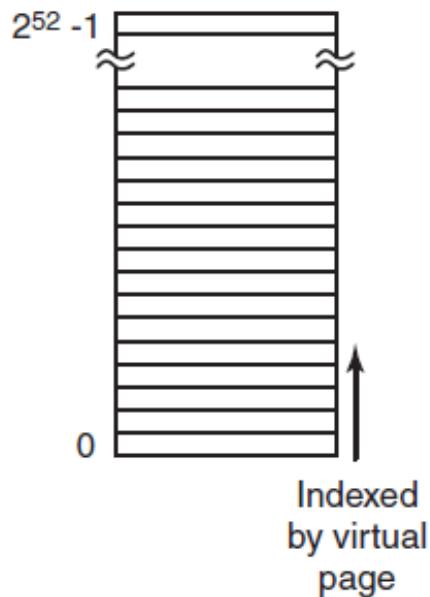
# Other Translation Organizations

- Inverted Page Tables
  - PowerPC , Sparc, IA64
- Segmentation
  - 80x86 ( but not x86-64 )



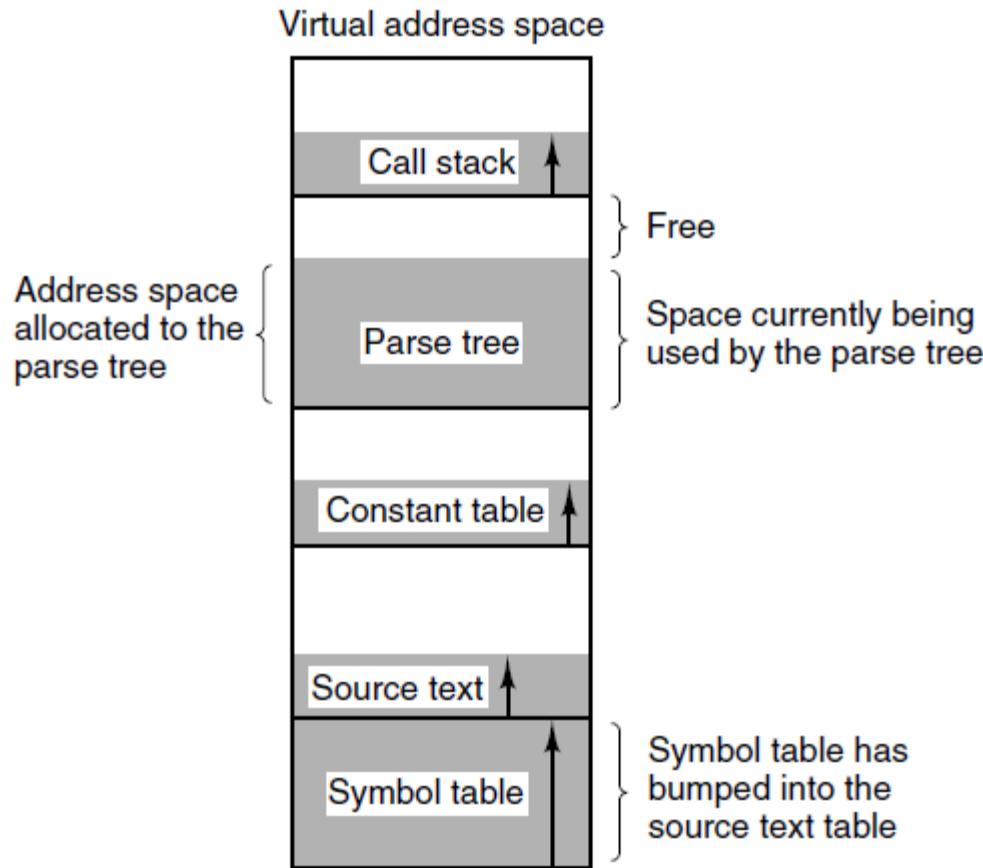
# Inverted Page Tables

Traditional page table with an entry for each of the  $2^{52}$  pages



Comparison of a traditional page table with an inverted page table.

# Segmentation (1)



In a one-dimensional address space with growing tables, one table may bump into another.

# Segmentation (2)

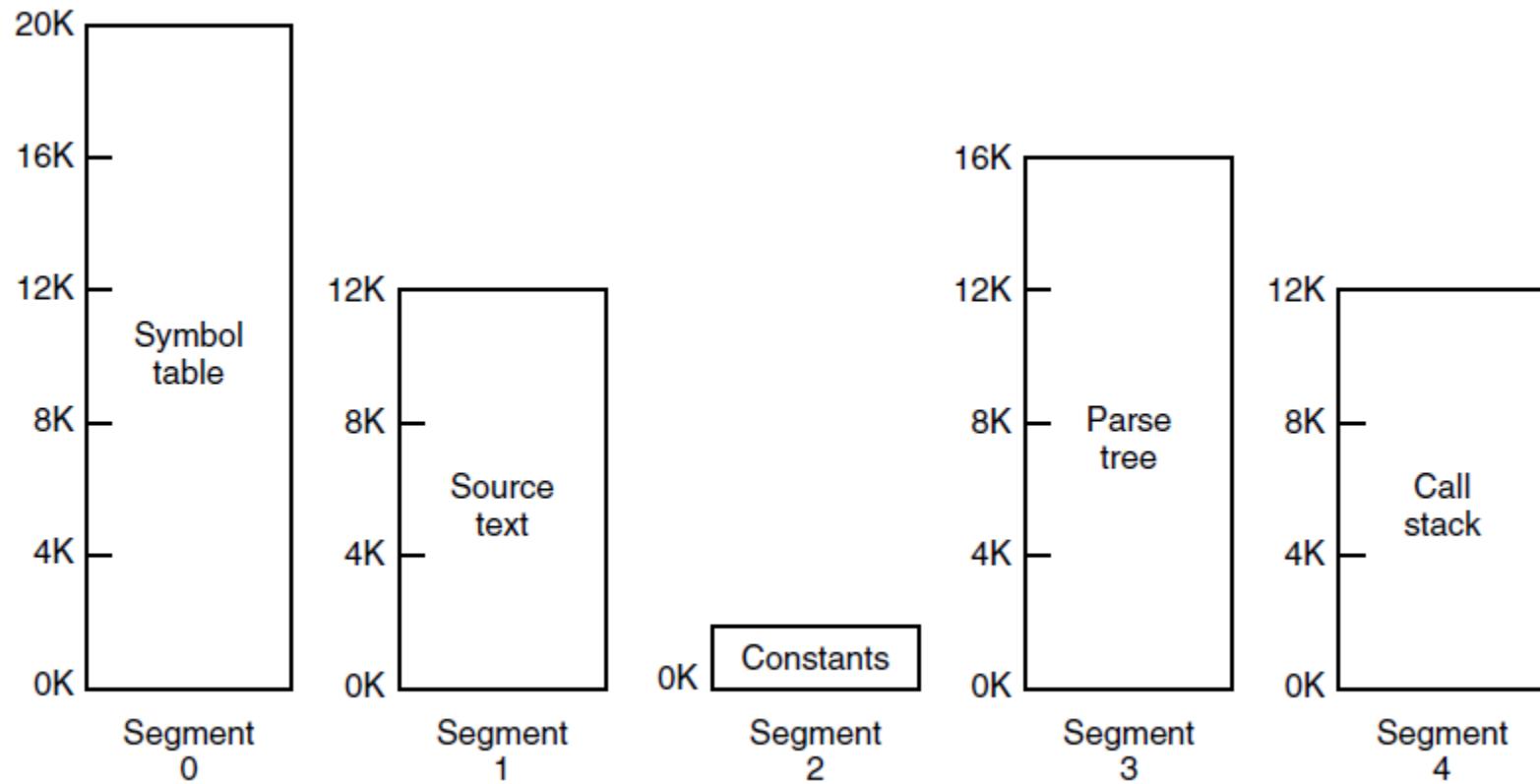


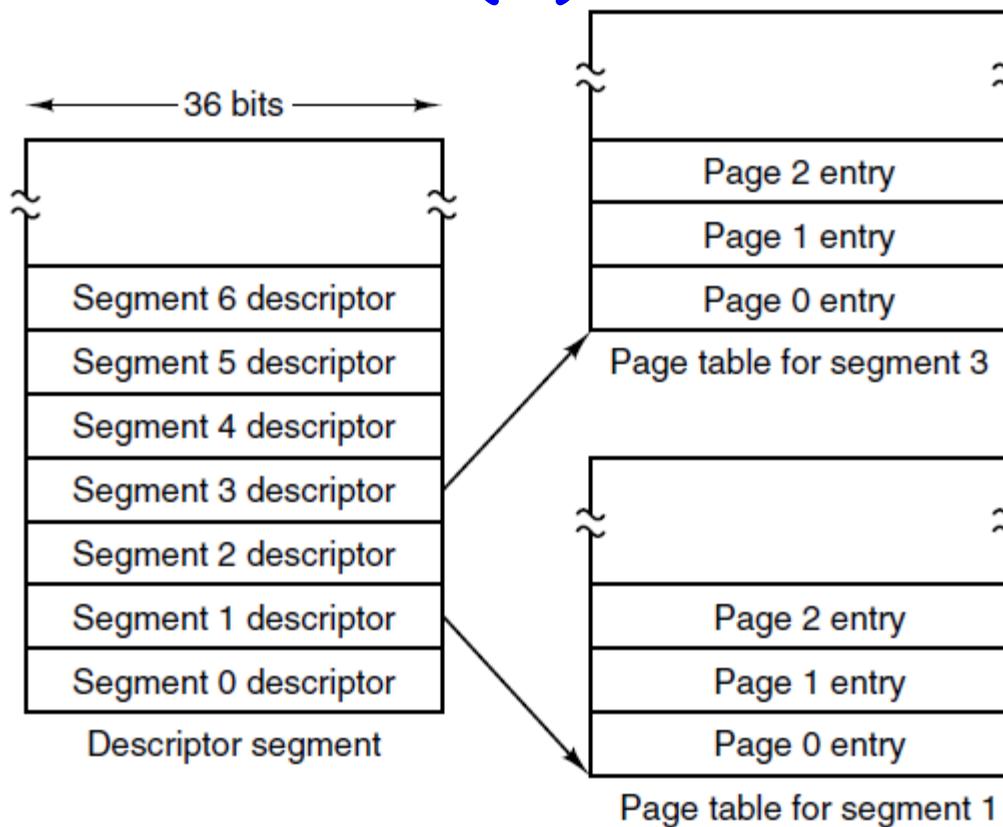
Figure 3-31. A segmented memory allows each table to grow or shrink independently of the other tables.

# Segmentation (3)

Consideration	Paging	Segmentation
Need the programmer be aware that this technique is being used?	No	Yes
How many linear address spaces are there?	1	Many
Can the total address space exceed the size of physical memory?	Yes	Yes
Can procedures and data be distinguished and separately protected?	No	Yes
Can tables whose size fluctuates be accommodated easily?	No	Yes
Is sharing of procedures between users facilitated?	No	Yes
Why was this technique invented?	To get a large linear address space without having to buy more physical memory	To allow programs and data to be broken up into logically independent address spaces and to aid sharing and protection

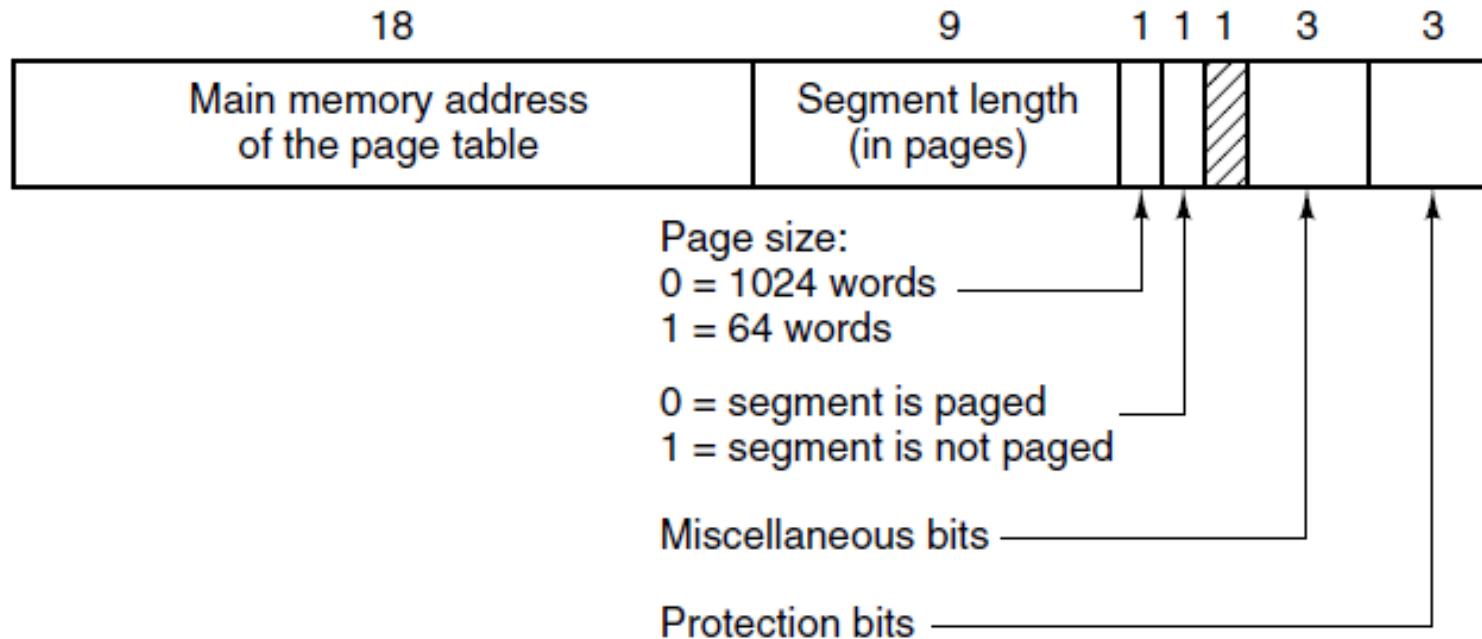
Comparison of paging and segmentation

# Segmentation with Paging: MULTICS (1)



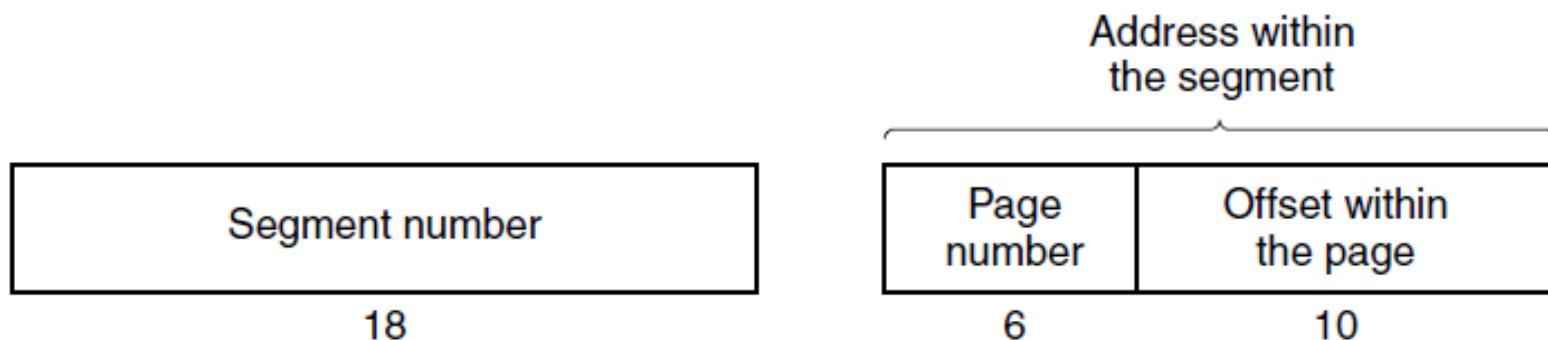
The MULTICS virtual memory. (a) The descriptor segment pointed to the page tables.

# Segmentation with Paging: MULTICS (2)



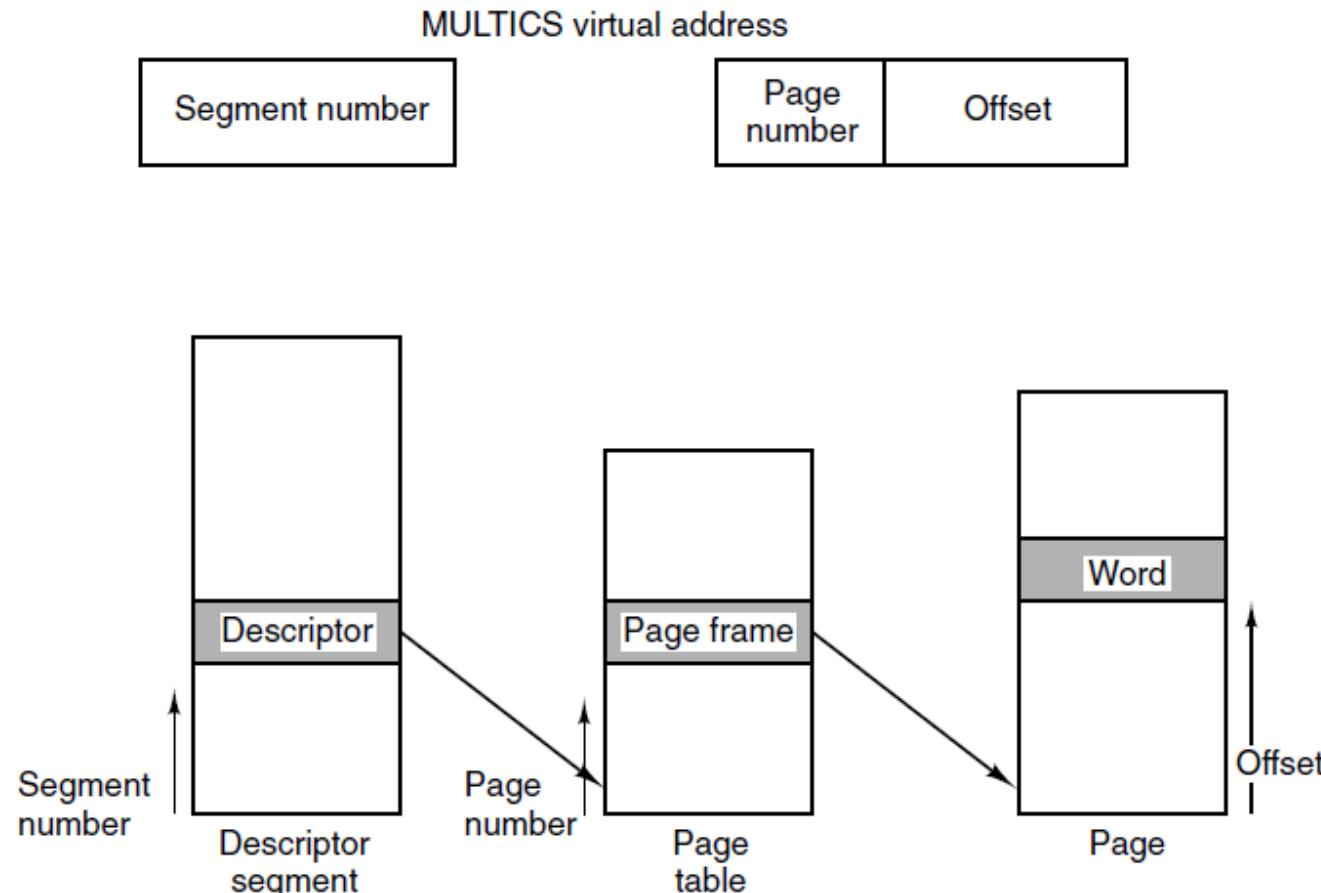
The MULTICS virtual memory. (b) A segment descriptor. The numbers are the field lengths.

# Segmentation with Paging: MULTICS (3)



A 34-bit MULTICS virtual address.

# Segmentation with Paging: MULTICS (4)



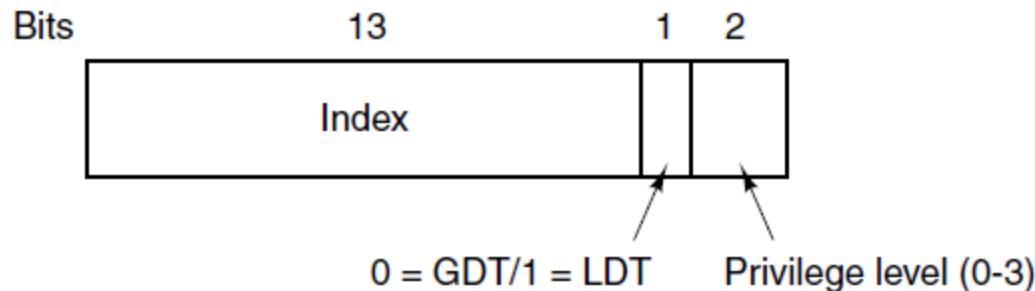
Conversion of a two-part MULTICS address into a main memory address.

# Segmentation with Paging: MULTICS (5)

Comparison field					Is this entry used?
Segment number	Virtual page	Page frame	Protection	Age	
4	1	7	Read/write	13	1
6	0	2	Read only	10	1
12	3	1	Read/write	2	1
					0
2	1	0	Execute only	7	1
2	2	12	Execute only	9	1

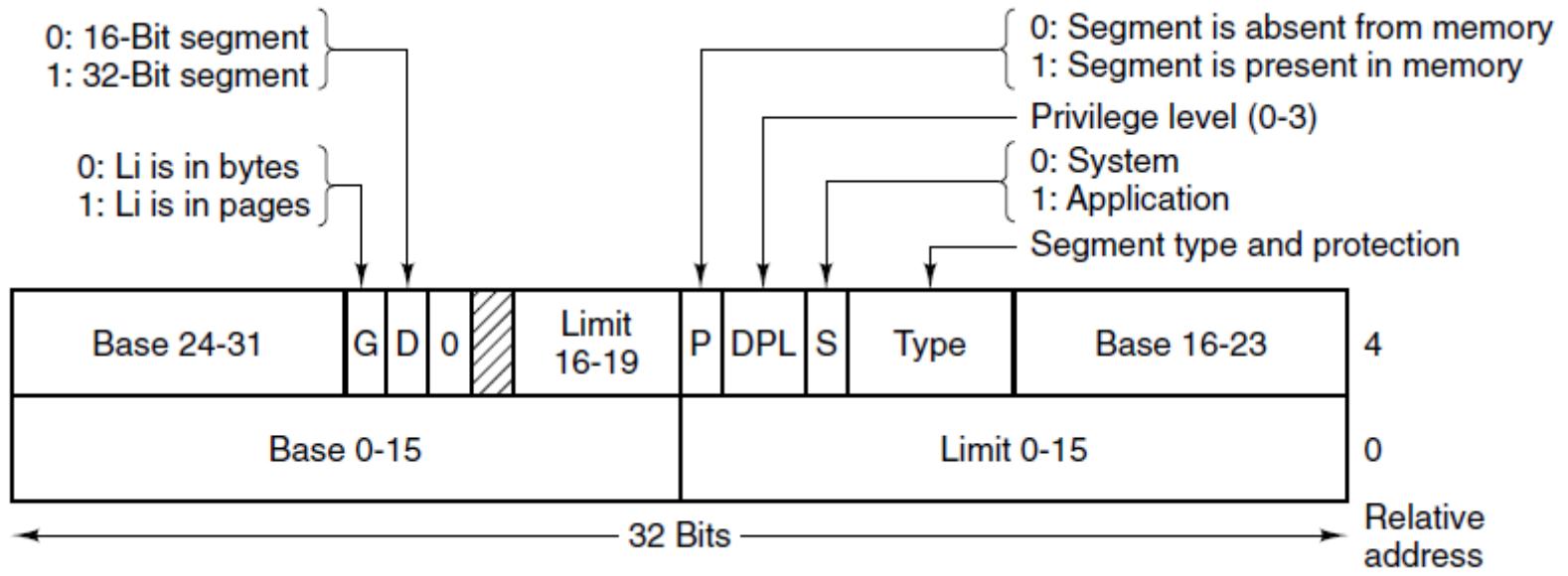
A simplified version of the MULTICS TLB. The existence of two page sizes made the actual TLB more complicated.

# Segmentation with Paging: The Intel x86 (1)



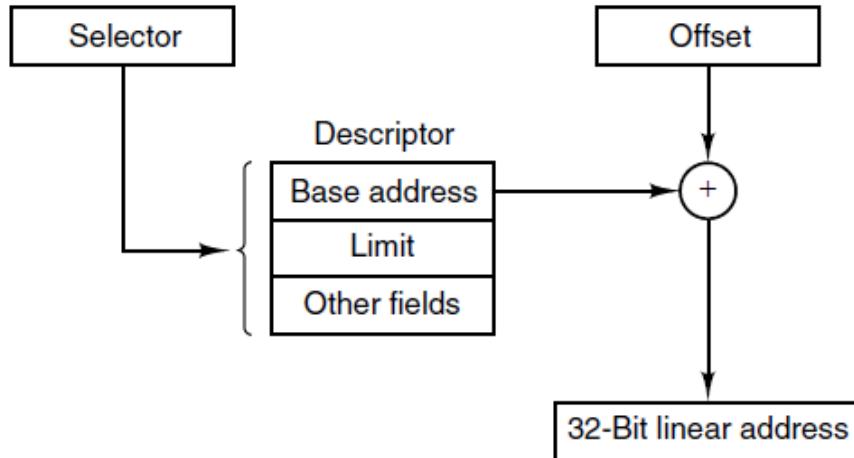
An x86 selector.

# Segmentation with Paging: The Intel x86 (2)



x86 code segment descriptor.  
Data segments differ slightly.

# Segmentation with Paging: The Intel x86 (3)



Conversion of a  
(selector, offset)  
pair to a linear  
address.

The 8086 architecture introduced 4 segments:

**CS** (code segment)

**DS** (data segment)

**SS** (stack segment)

**ES** (extra segment)

the 386 architecture introduced two new general segment registers **FS**, **GS**.

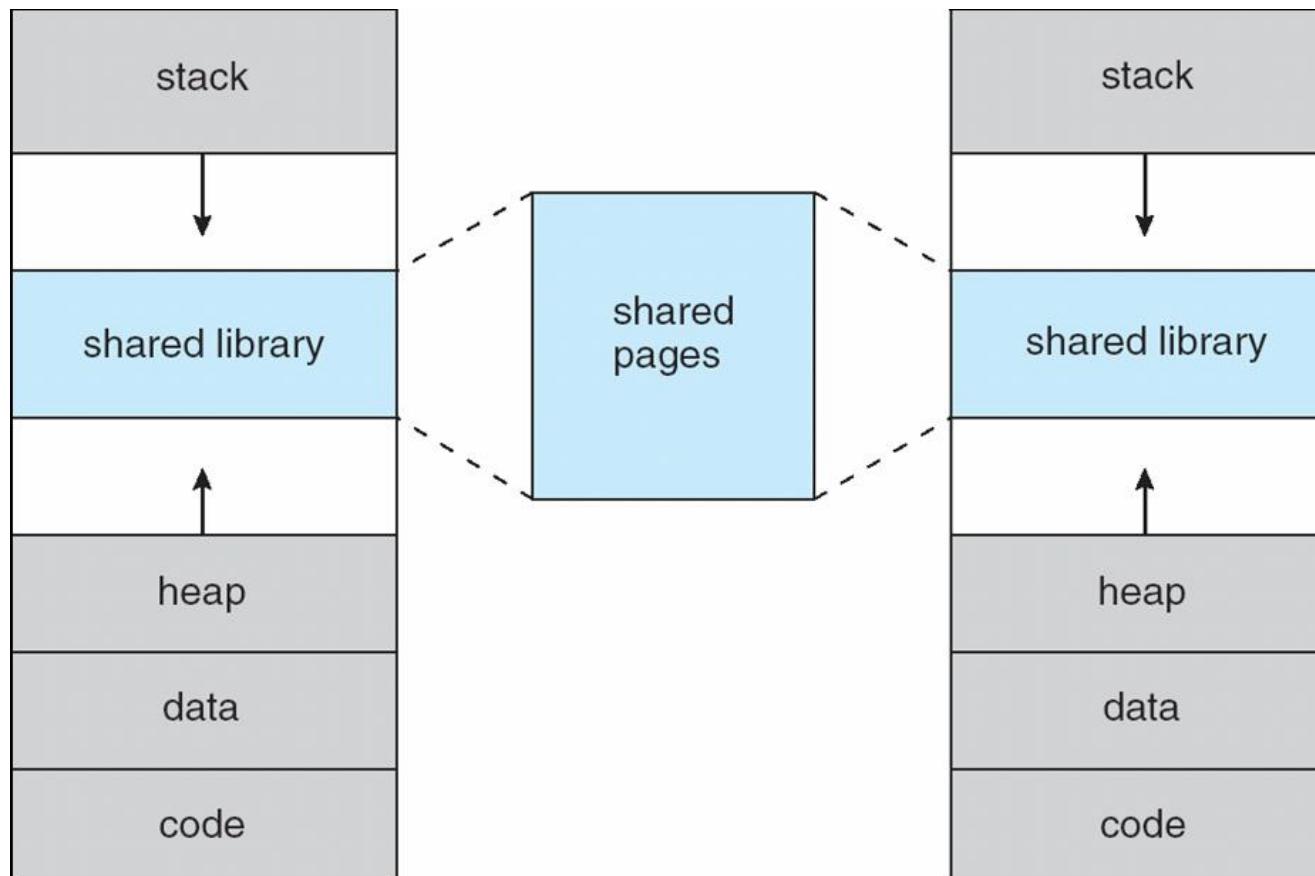
```
mov dword [fs:eax], 42
```

```
mov dx, 850h
mov es, dx ; Move 850h to es segment register
mov es:cx, 15h ; Move 15 to es:cx
```

# Shared Pages (efficient usage of memory)

- **Shared code**
  - One copy of read-only (**reentrant**) code shared among processes (i.e., text editors, compilers, window systems)
  - Similar to multiple threads sharing the same process space
  - Also useful for inter-process communication if sharing of read-write pages is allowed
- **Private code and data**
  - Each process keeps a separate copy of the code and data
  - The pages for the private code and data can appear anywhere in the logical address space

# Shared Library Using Virtual Memory



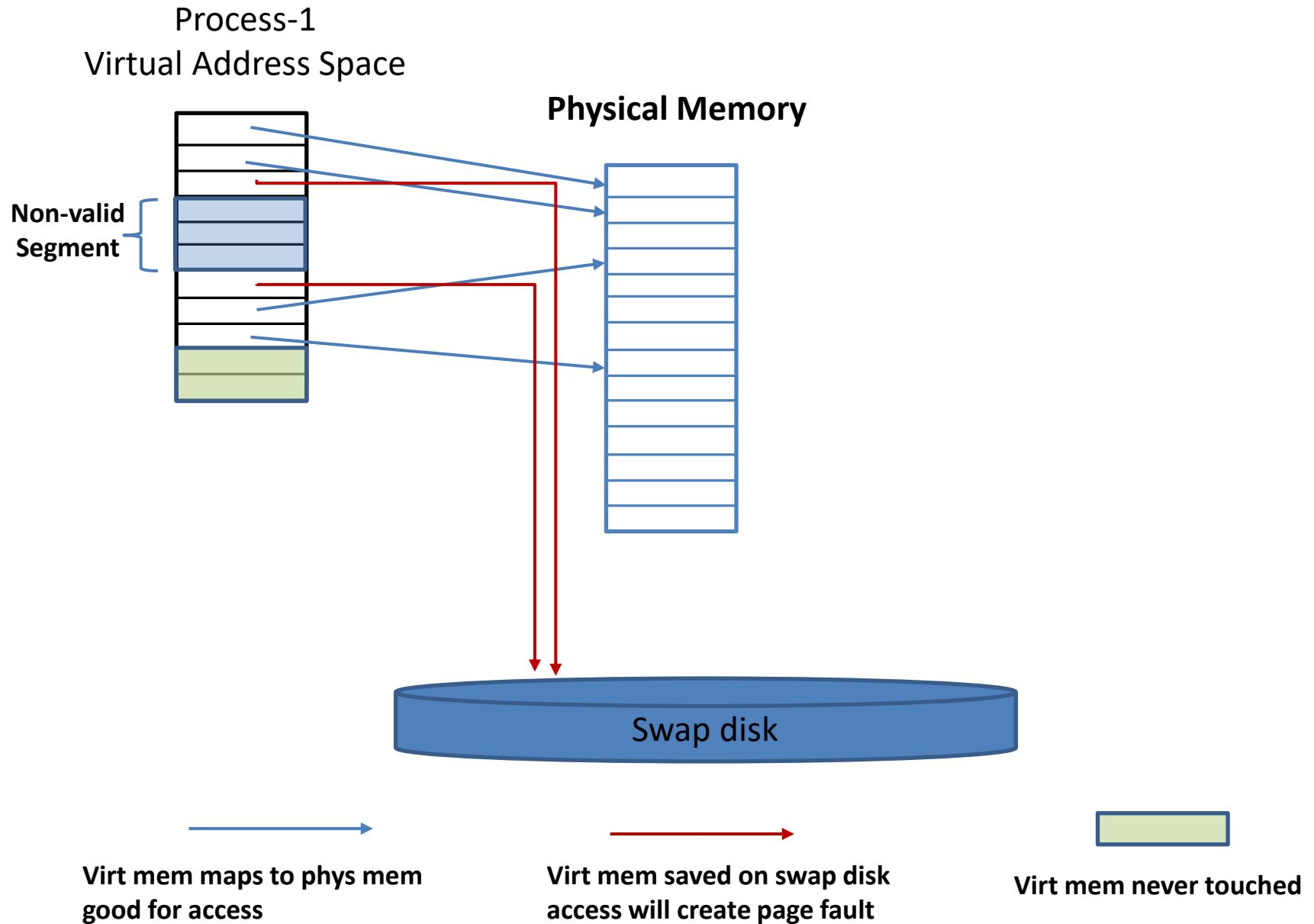
# Copy-on-Write

- **Copy-on-Write (COW)** allows both parent and child processes to initially *share* the same pages in memory
  - If either process modifies a shared page, only then is the page copied
- **COW** allows more efficient process creation as only modified pages are copied

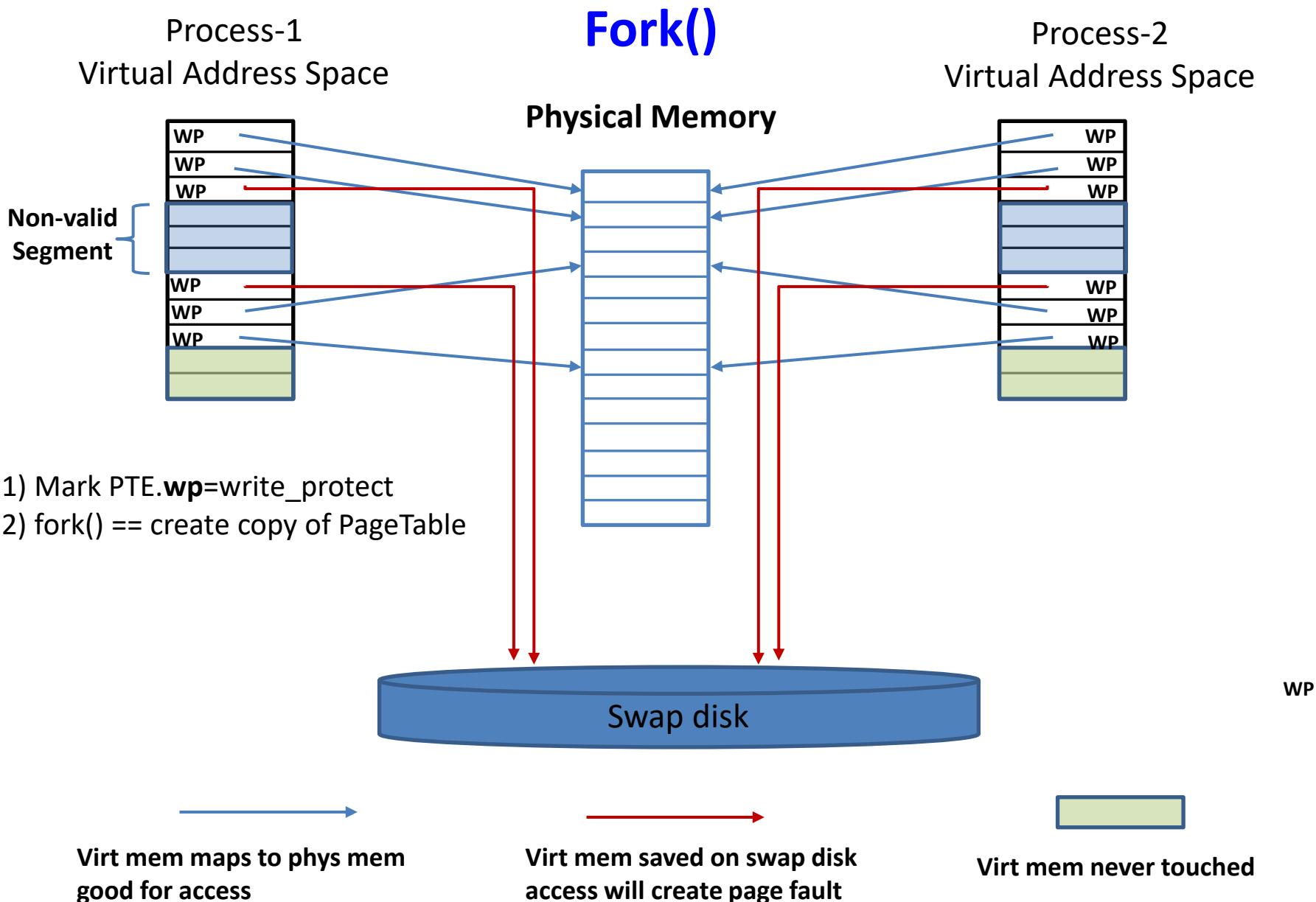
# Copy-on-Write (cont)

- In general, free pages are allocated from a **pool** of **zero-fill-on-demand** pages
  - Pool should always have free frames for fast demand page execution
    - Don't want to have to free a frame when one is needed for processing on page fault
  - Why zero-out a page before allocating it?  
-> so we get a known state of a page.
- vfork() variation on fork() system call has parent suspend and child using copy-on-write address space of parent
  - Designed to have child call exec()
  - Very efficient

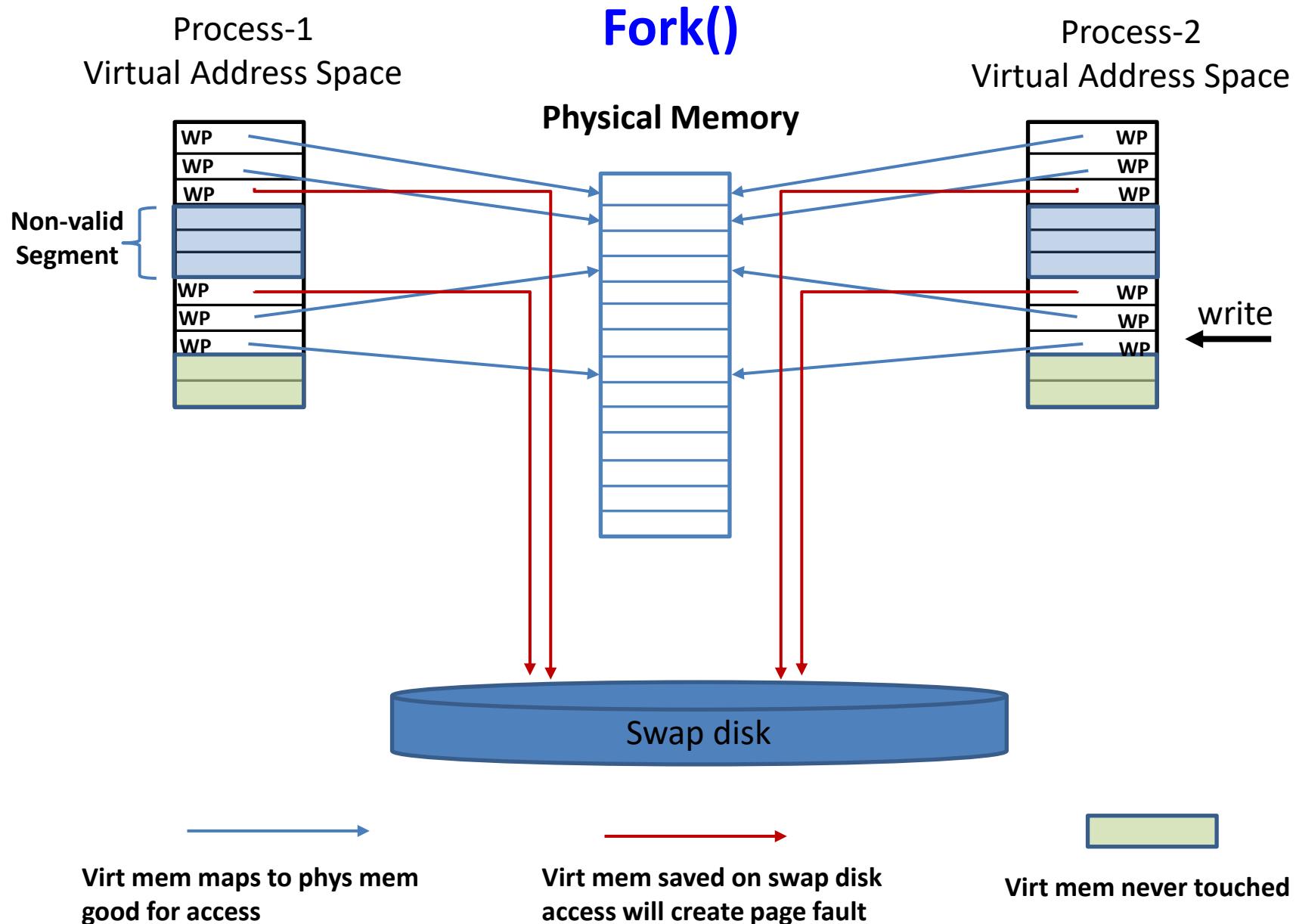
# Virtual Memory



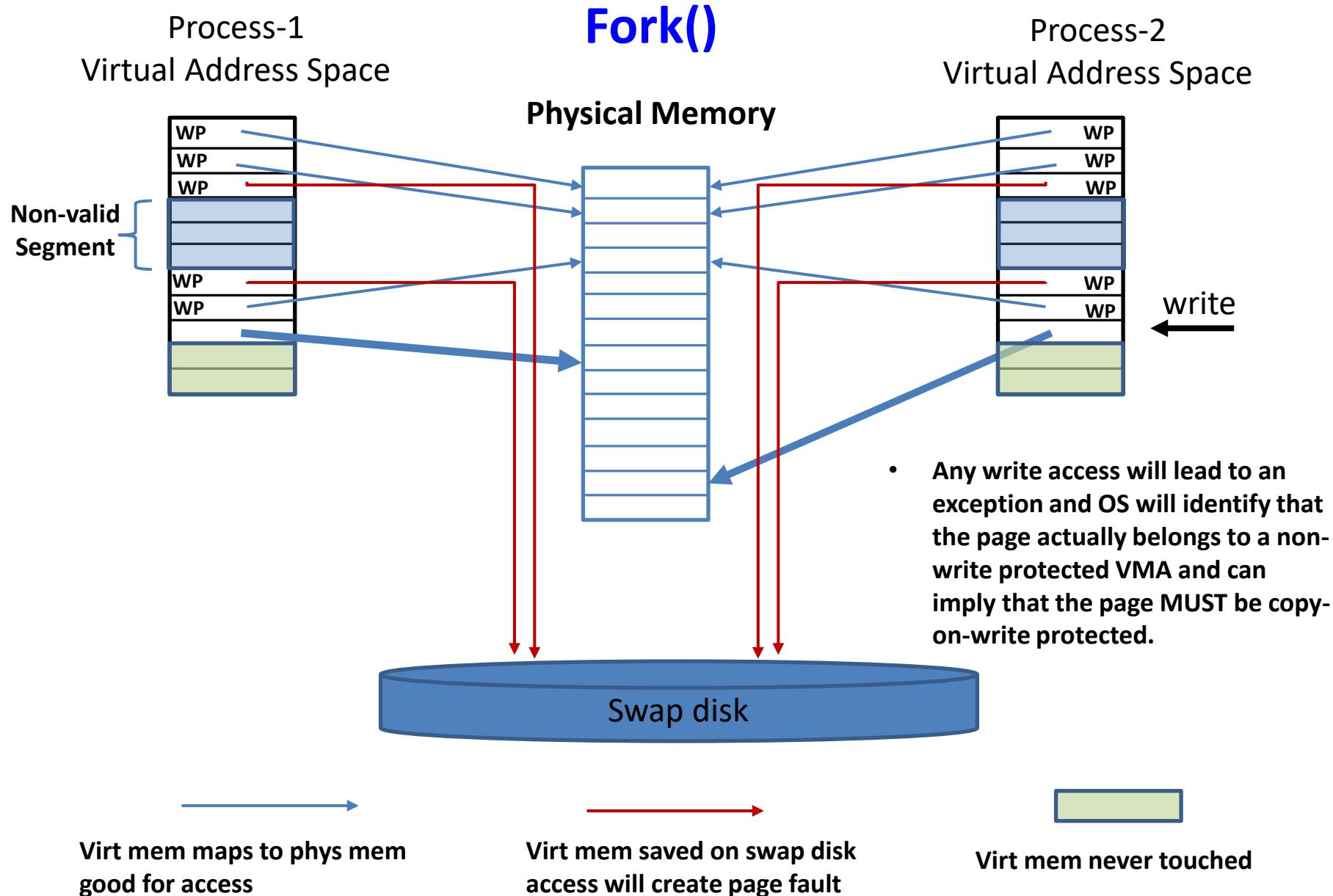
# Virtual Memory



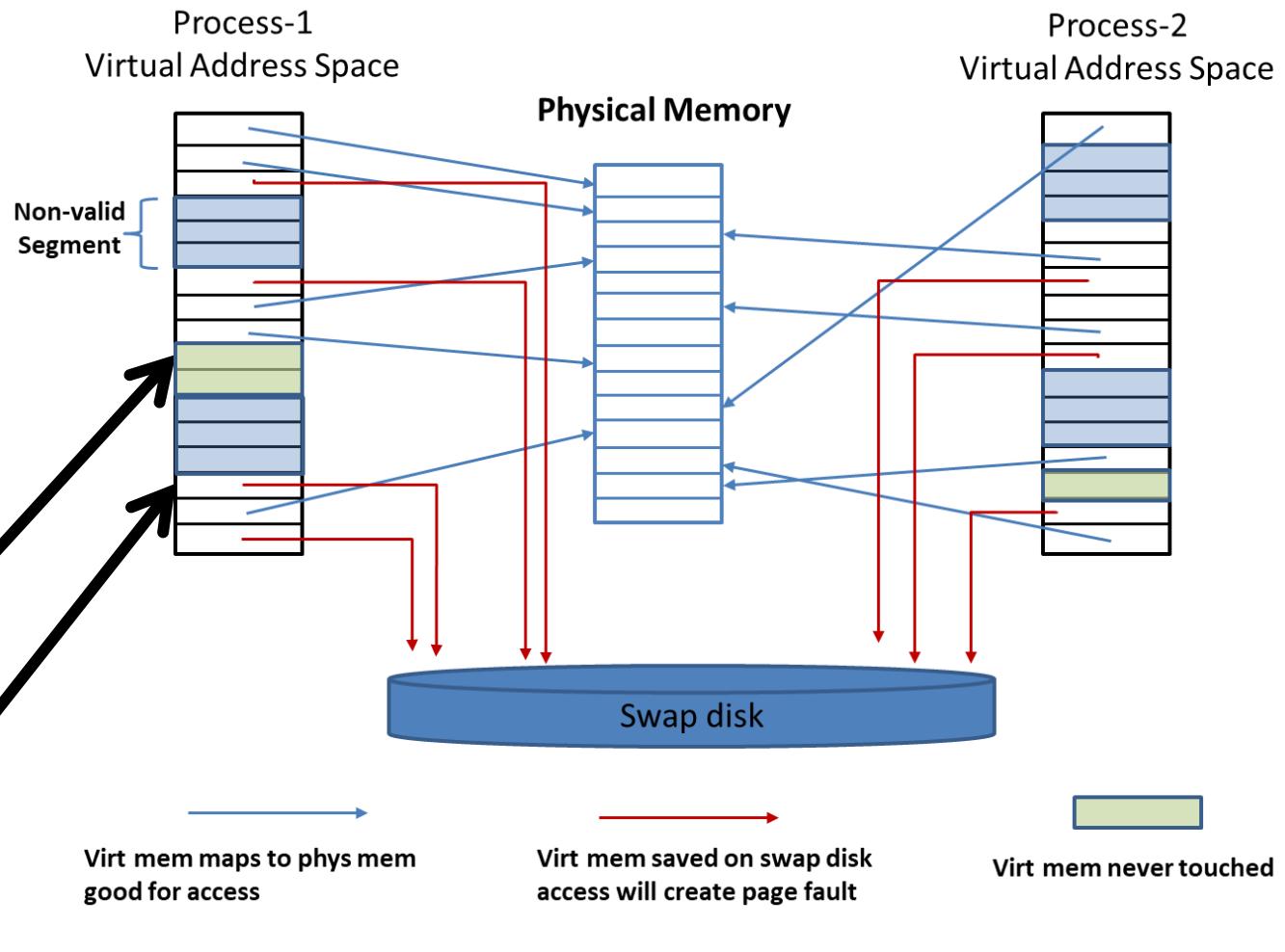
# Virtual Memory (COW)



# Virtual Memory (COW)



# We are running out of memory now what ?



In case of page fault:

**which page to remove from memory and swap → stealing from self or some other process ?**

# Replacement Policies

- Used in many contexts when storage is not enough (not just operating systems)
  - caches
  - web servers
  - Pages
- Things to take into account when designing a replacement policy
  - measure of success
  - cost

# Thin[gk]s to consider

- Local Policies:
  - search processes' pagetables for candidates
  - $O(N * \text{sizeof(PageTable})$ , where  $N$  = number of processes,  
 $\text{sizeof(PageTable)} \approx$  size of Virtual address space
  - Does not have a global view of usage
  - Enables running "global" algorithms for page replacement which are now  $O(F)$ , where as  $F$  is #frames vs  $O(N * \text{sizeof(PageTable})$ , where as  $N$  is number of processes.
- Global Policies:
  - search frames for candidates
  - $O(F)$  where  $F$  is number of physical frames ( $\approx$  size of physical memory )
- In general global policies are preferred as  $O(F) \ll O(N * \text{sizeof(PageTable})$ )

for now let's first talk about

- a single page table
- talk about page replacement in that single page table
- Describe this algorithm that way .. → local policy with  $N=1$
- Later we generalize with global policy and multiple processes.

# Data Structures Required

- Address Space (one per process)
  - Represented as a sequence of VMAs (virtual memory areas)
    - [ start-addr, length, Read/Write/Execute, ... ]
  - PageTable
- PageTable ( struct PTE pgtable[] )
- Frame Table (struct frame frame\_table[])
  - each chunk of physical memory (e.g. 4K) is described by meta data [ struct frame ]
  - Used and how often, locked ?
  - Back reference to pte(s) that refer to this frame. **Required because we need access to PTE's R and M bits to make replacement decisions.**
  - In global algorithms we loop through the frametable but reach back to the PTEs
  - Note only pages that are mapped to frames can be candidates for paging !!!!

# Optimal Page Replacement Algorithm

- Each page labeled with the number of instructions that will be executed before this page is referenced
- Page with the highest label should be removed
- Impossible to implement  
(just used for theoretical evaluations)

# More realistic Algos

- We don't have to make the best, optimal decision
- We need to make a reasonable decision at a reasonable overhead
- Its even OK to make the absolute worst decision occasionally as the system will implement correct behavior nevertheless.

# The FIFO Replacement Algorithm

- OS maintains a list of the pages currently in memory [ that would be for instance frame table ]
- The most recent arrival at the tail
- On a page fault, the page at the head is removed
- In lab3: for simplicity you implement FIFO with a simple round robin using a HAND == pointer to the frametable

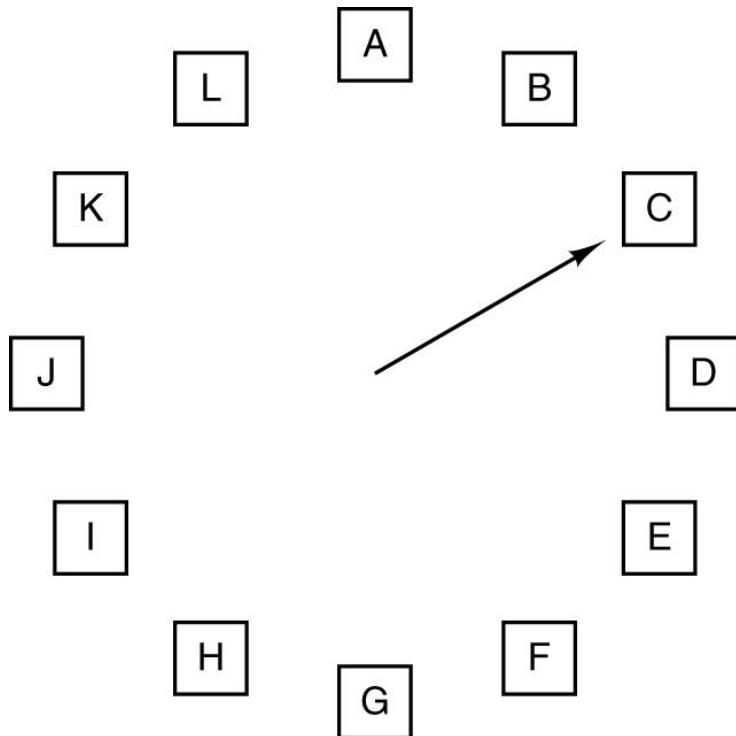
# The Second-Chance Page Replacement Algorithm

- Modification to FIFO
- Inspect the R bit of the oldest page
  - If  $R=0$  page is old and unused  $\rightarrow$  replace
  - If  $R=1$  then
    - bit is cleared
    - page is put at the end of the list
    - the search continues
- If all pages have  $R=1$ , the algorithm degenerates to FIFO

# The Clock Page Replacement Policy

- Keep page frames on a circular list in the form of a clock
- The hand points to the oldest uninspected page
- When page fault occurs
  - The page pointed to by the hand is inspected
  - If  $R=0$ 
    - page evicted
    - new page inserted into its place
    - hand is advanced
  - If  $R = 1$ 
    - $R$  is set to 0
    - hand is advanced
- This is essentially an implementation of 2<sup>nd</sup>-Chance

# The Clock Page Replacement Policy

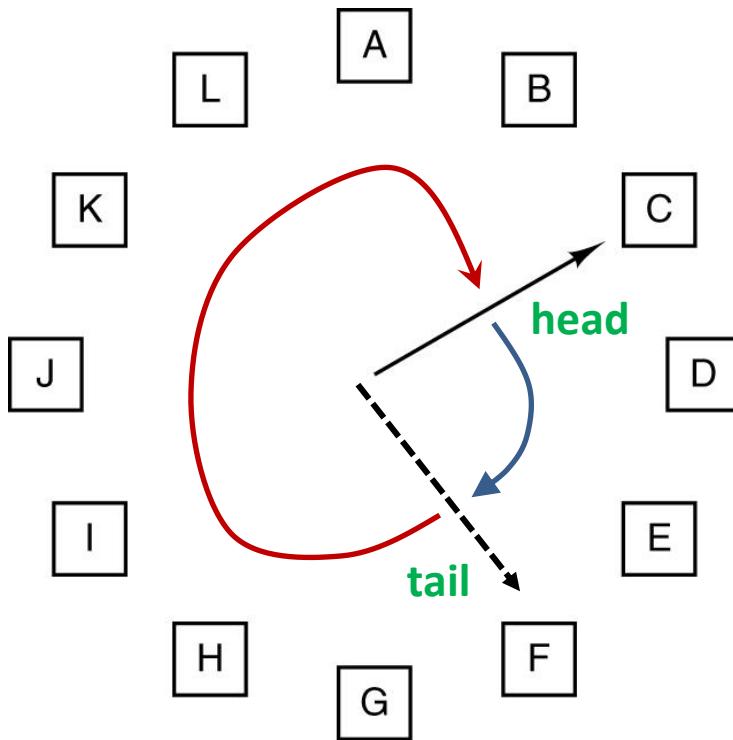


When a page fault occurs,  
the page the hand is  
pointing to is inspected.  
The action taken depends  
on the R bit:

- R = 0: Evict the page
- R = 1: Clear R and advance hand

# The Clock

## Page Replacement Policy ( an optimization )



When a page fault occurs,  
the page the hand is  
pointing to is inspected.  
The action taken depends  
on the R bit:

- R = 0: Evict the page
- R = 1: Clear R and advance hand

Tail moves in front and “processes” candidates

- writing optimistically modified pages back to swap and clear “M” bit and R bit

# The Not Recently Used (NRU) Replacement Algorithm

Sometimes called: Enhanced  
Second Chance Algorithm

- Two status bits with each page
  - R: Set whenever the page is referenced (used)
  - M: Set when the page is written / dirty
- R and M bits are available in most computers implementing virtual memory
- Those bits are updated with each memory reference
  - Must be updated by hardware
  - Reset only by the OS
- Periodically (e.g. on each clock interrupt) the R bit is cleared
  - To distinguish pages that have been referenced recently

# The Not Recently Used Replacement Algorithm

	R	M
Class 0:	0	0
Class 1:	0	1
Class 2:	1	0
Class 3:	1	1

NRU algorithm removes a page at random from the lowest numbered un-empty Class.

Note: class\_index = 2\*R+M

Note: in lab3 to we select the first page encountered in a class so we don't have to track all of them, we also use a clock like algorithm to circle through pages/frames

Ref-Bit reset doesn't happen each and every search.  
Typically done at specific times through a daemon.

# The Least Recently Used (LRU) Page Replacement Algorithm

- Good approximation to optimal
- When page fault occurs, replace the page that has been unused for the longest time
- Realizable but not cheap

# LRU

## Hardware Implementation 1

- 64-bit counter increment after each instruction
- Each page table entry has a field large enough to include the value of the counter
- After each memory reference, the value of the counter is stored in the corresponding page entry
- At page fault, the page with lowest value is discarded

# LRU

## Hardware Implementation 1

- 64-bit counter increment after each instruction
- Each entry needs to increment its counter though to indicate it has been used.  
Too expensive!
- After each access, the counter of the accessed entry needs to be updated.  
Too Slow!
- At page fault, the page with lowest value is discarded

# LRU: Hardware Implementation 2

- Machine with n page frames
- Hardware maintains a matrix of  $n \times n$  bits
- Matrix initialized to all 0s
- Whenever page frame k is referenced
  - Set all bits of row k to 1
  - Set all bits of column k to 0
- The row with lowest value is the LRU

# LRU: Hardware Implementation 2

	Page			
	0	1	2	3
0	0	1	1	1
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0

(a)

	Page			
	0	1	2	3
0	0	0	1	1
1	0	1	1	1
2	0	0	0	0
3	0	0	0	0

(b)

	Page			
	0	1	2	3
0	0	0	0	1
1	0	0	0	1
2	1	1	0	1
3	0	0	0	0

(c)

	Page			
	0	1	2	3
0	0	0	0	0
1	0	0	0	0
2	1	1	0	0
3	1	1	1	0

(d)

	Page			
	0	1	2	3
0	0	0	0	0
1	0	0	0	0
2	1	0	0	0
3	1	1	0	1

(e)

0	0	0	0
1	0	1	1
1	0	0	1
1	0	0	0

(f)

0	1	1	1
0	0	1	1
0	0	0	1
0	0	0	0

(g)

0	1	1	0
0	0	1	0
0	0	0	0
1	1	1	0

(h)

0	1	0	0
0	0	0	0
1	1	0	1
1	1	0	0

(i)

0	1	0	0
0	0	0	0
1	1	0	0
1	1	1	0

(j)

Pages referenced: 0 1 2 3 2 1 0 3 2 3

# LRU

## Hardware Implementation 3

- Maintain the LRU order of accesses in frame list by hardware



- After accessing page 3



# LRU Implementation

- Slow
- Few machines (if any) have required hardware

# Approximating LRU in Software

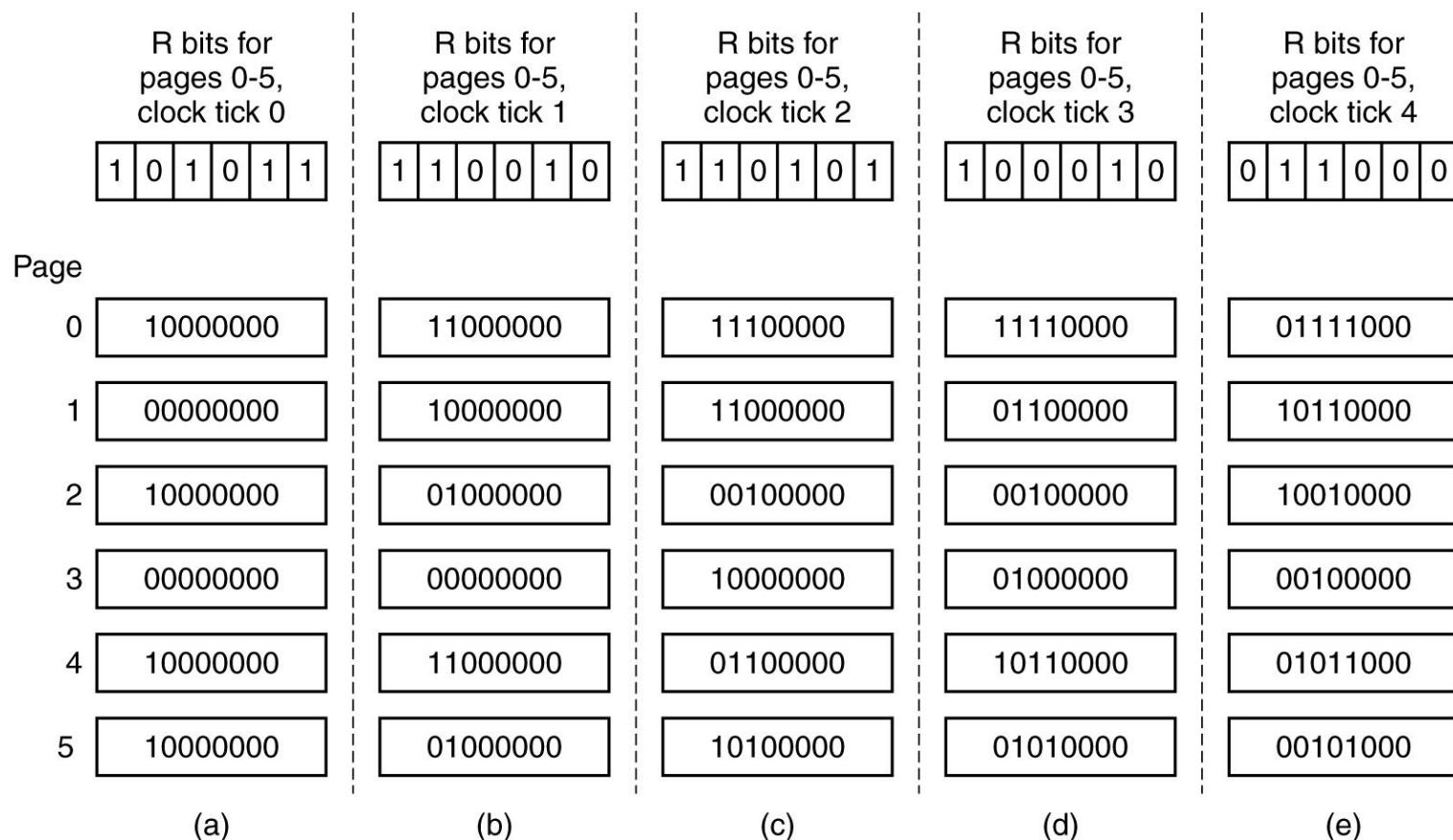
- Not Frequently Used (NFU) algorithm
- Software counter associated with each page, initially zero
- At some periodicity (e.g. a second), the OS scans all page table entries and adds the R bit to the counter of that PTE
- At page fault: the page with lowest counter is replaced

Still a lot of overhead (not really practical either)

# Aging Algorithm

- NRU never forgets anything  
→ high inertia
- Modifications:
  - shift counter right by 1
  - add R bit as the leftmost bit
  - reset R bit
- This modified algorithm is called **aging**
- Each bit in vector represents a period
- The page whose counter is lowest is replaced at page replacement

# Aging Algorithm



# The Working Set Model

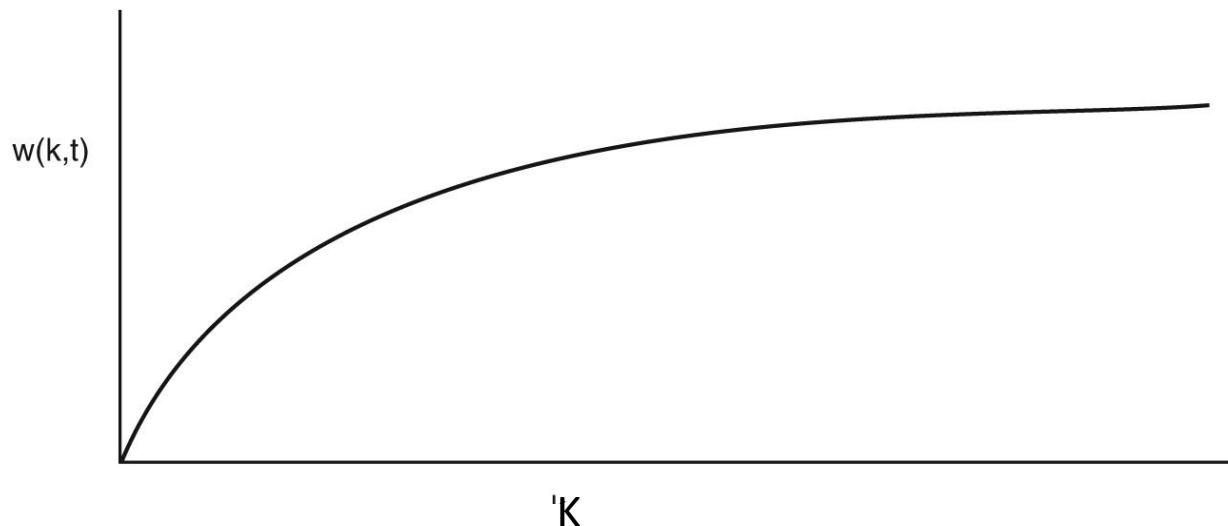
- **Working set**: the set of pages that a process is currently using
- **Thrashing**: a program causing page faults at high rates (e.g. pagefaults/instructions metric)

An important question:

In multiprogramming systems, processes are sometimes swapped to disk (i.e. all their pages are removed from memory). When they are brought back, which pages to bring?

# The Working Set Model

- Try to keep track of each process' working set and make sure it is in memory before letting the process run.

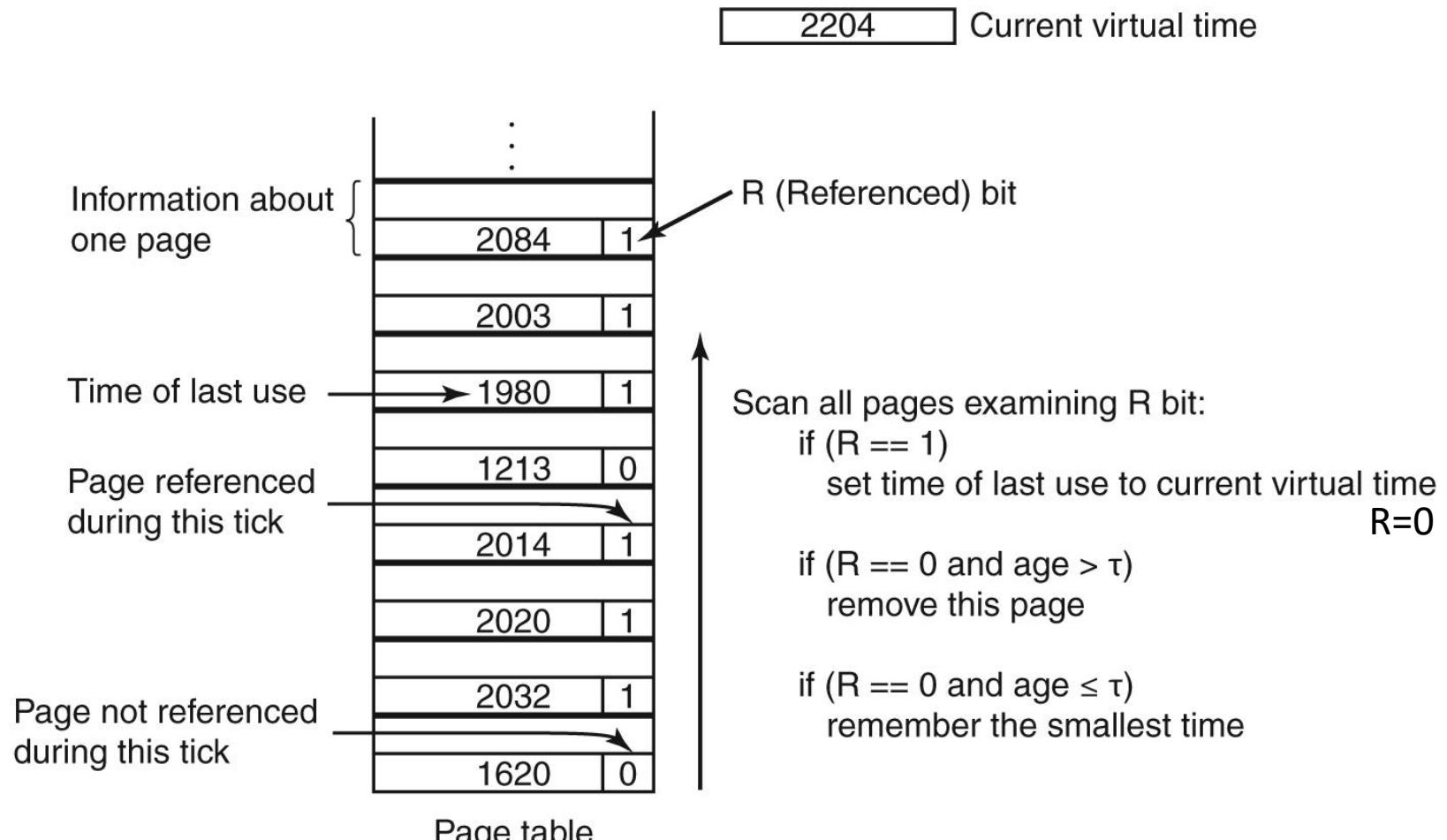


$w(k,t)$ : the set of pages accessed in the last  $k$  references at instant  $t$

# The Working Set Model

- OS must keep track of which pages are in the working set
- Replacement algorithm: evict pages not in the working set
- Possible implementation (but expensive):
  - working set = set of pages accessed in the last k memory references
- Approximations
  - working set = pages used in the last 100 msec or 1 sec (etc.)

# Working Set Page Replacement Algorithm



age = current virtual time – time of last use  
time of last use == ref-bit was reset

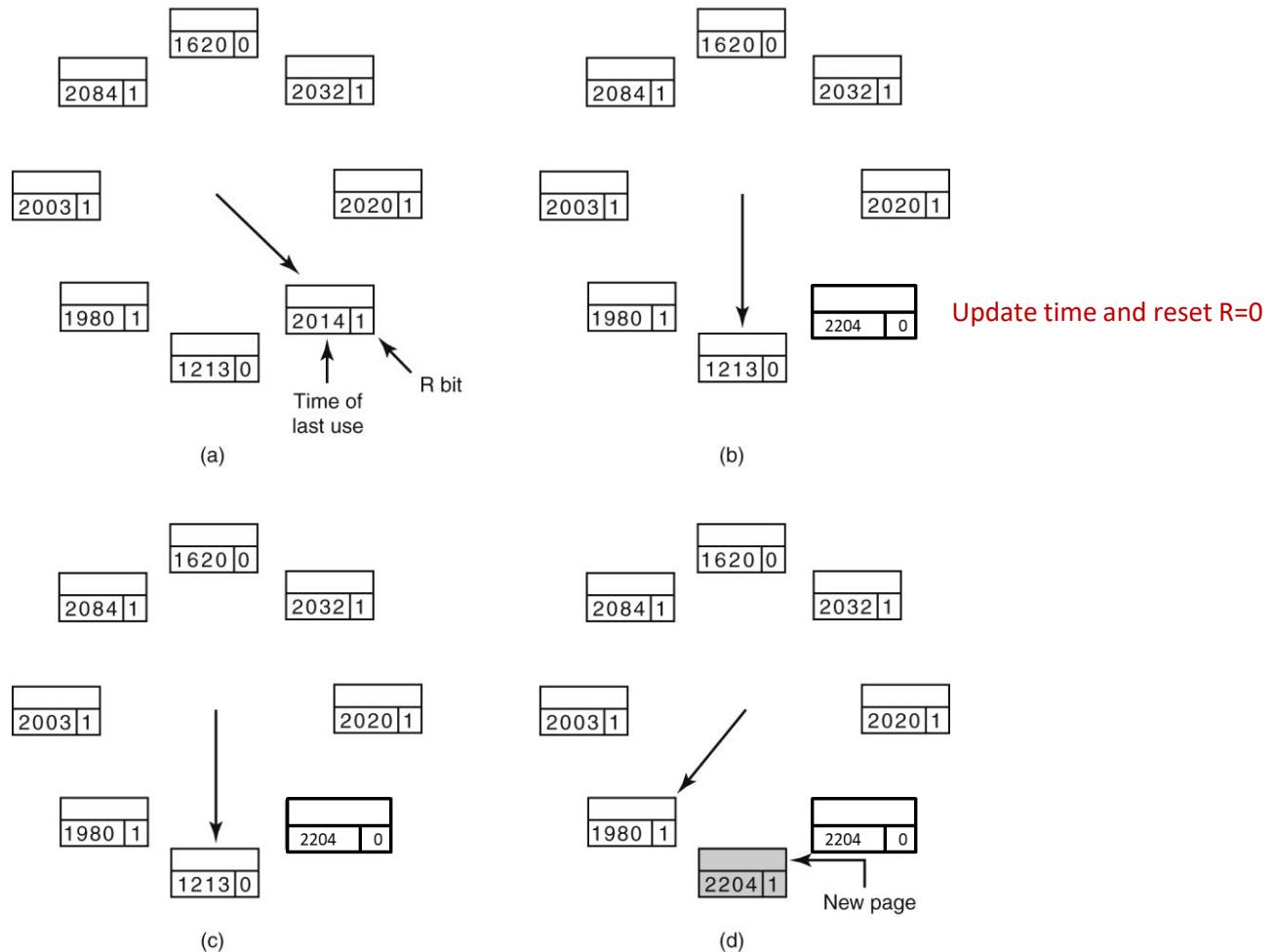
# The WSClock Page Replacement Algorithm (specific implementation of Working Set Replacement)

- Based on the clock algorithm and uses working set
- data structure: circular list of page frames (clock)
- Each entry contains: time of last use, R bit
- At page fault: page pointed by hand is examined
  - If  $R = 1$ :
    - Record current time , reset R
    - Advance hand to next page
  - If  $R = 0$ 
    - If age > threshold and page is clean -> it is reclaimed
    - If page is dirty -> write to disk is scheduled and hand advances  
(note in lab3, we skip this step for simplicity reasons, but think why this is advantageous ?)
    - Advance hand to next page

# The WSClock Page Replacement Algorithm

2204 Current virtual time

Let Tau = 500



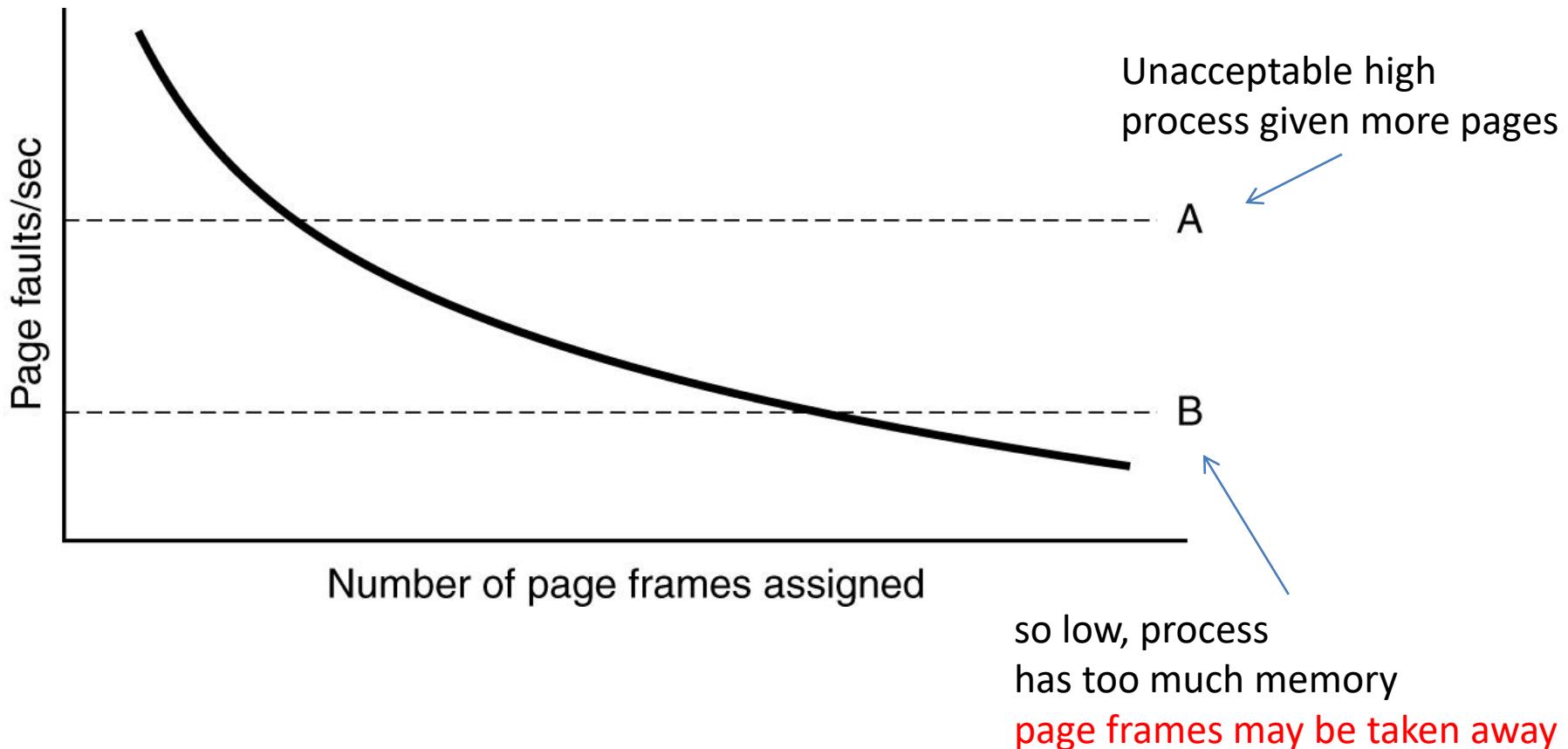
# Design Issues for Paging: Local vs Global Allocation

- How should memory be allocated among the competing runnable processes?
- Local algorithms: allocating every process a fixed fraction of the memory
- Global algorithms: dynamically allocate page frames
- Global algorithms work better
  - If local algorithm used and working set grows → thrashing will result
  - If working set shrinks → local algorithms waste memory

# Global Allocation

- Method 1: Periodically determine the number of running processes and allocate each process an equal share
- Method 2 (better): Pages allocated in proportion to each process total size
- Page Fault Frequency (PFF) algorithm: tells when to increase/decrease page allocation but says nothing about which page to replace.

# Global Allocation: PFF



# Design Issues: Load Control

- What if PFF indicates that some processes need more memory but none need less?
- Swap some processes to disk and free up all the pages they are holding.
- Which process(es) to swap?
  - Strive to make CPU busy (I/O bound vs CPU bound processes)
  - Process size

# Global Policies and OS Frame Tables

- In general the OS keeps meta data for each frame, typically 8-16 bytes.  
→ 2-4% of memory required to describe memory.
- Meta Data includes
  - reverse mapping ( pid, vpage) of user
  - Helper data for algorithm (e.g. age, tau)
- Enables running “global” algorithms for page replacement which are now  $O(F)$ , where as F is #frames vs  $O(N * \text{sizeof(PageTable)})$ , where as N is number of processes.
- Replacement algo can now run *something similar* to this.

```
for ( i=0 ; i<num_frames ; i++ ) {  
    derive user's PTE from frametable[i].<pid,vpage>  
    do what ever you have to do based on PTE  
    determine best frame to select  
}
```

# Design Issues: Page Size

- Large page size → internal fragmentation
- Small page size →
  - larger page table
  - More overhead transferring from disk
- Remember the Hardware Designers decide the frame/page sizes !!

# Design Issues: Page Size

- Assume:
  - $s$  = process size
  - $p$  = page size
  - $e$  = size of each page table entry
- So:
  - number of pages needed =  $s/p$
  - occupying:  $se/p$  bytes of page table space
  - wasted memory due to fragmentation:  $p/2$
  - overhead =  $se/p + p/2$
- We want to minimize the overhead:
  - Take derivative of overhead and equate to 0:
  - $-se/p^2 + \frac{1}{2} = 0 \rightarrow p = \sqrt{2se}$

# Design Issues: Shared Pages

- To save space, when same program is running by several users for example
- If separate I and D spaces: process table has pointers to Instruction page table and Data page table
- In case of common I and D spaces:
  - Special data structure is needed to keep track of shared pages
  - **Copy on write** for data pages

# Design Issues: Shared Libraries

- Dynamically linked
- Loaded when program is loaded or when functions in them are called for the first time
- Compilers must not produce instructions using absolute addresses in libraries → **position-independent code**

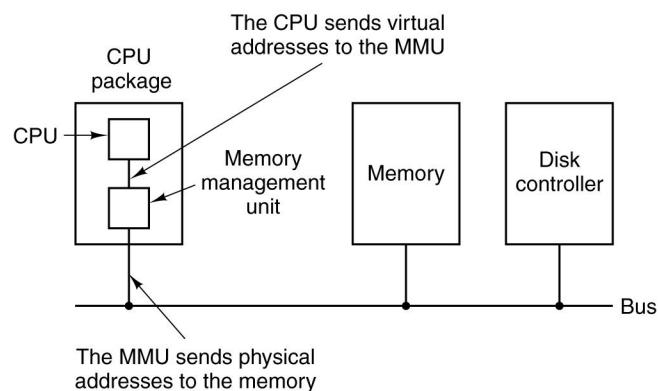
# Design Issues: Cleaning Policy

- Paging daemon
- Sleeps most of the time
- Awakened periodically to inspect state of the memory
- If too few pages/frames are free -> daemon begins selecting pages to evict and gets more aggressive the lower the free page count sinks.

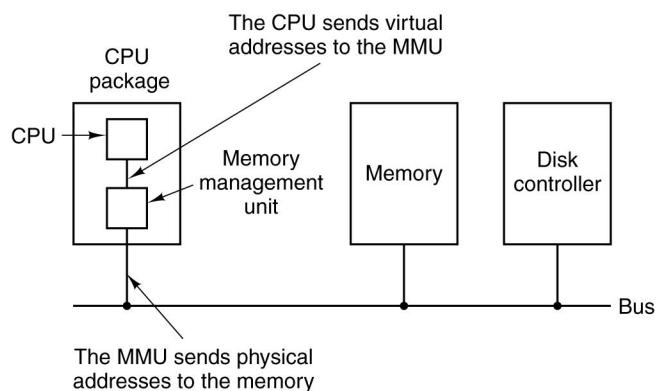
# Summary of Paging

- Virtual address space bigger than physical memory
- Mapping virtual address to physical address
- Virtual address space divided into fixed-size units called **pages**
- Physical address space divided into fixed-size units called pages **frames**
- Virtual address space of a process can be and are non-contiguous in physical address space

# Paging



# Paging



# OS Involvement With Paging

- When a new process is created
- When a process is scheduled for execution
- When process exits
- When page fault occurs

# OS Involvement With Paging

- When a new process is created
  - Determine how large the program and data will be (initially)
  - Create page table
  - Allocate space in memory for page table
  - Record info about page table and swap area in process table
- When a process is scheduled for execution
- When process exits
- When page fault occurs

# OS Involvement With Paging

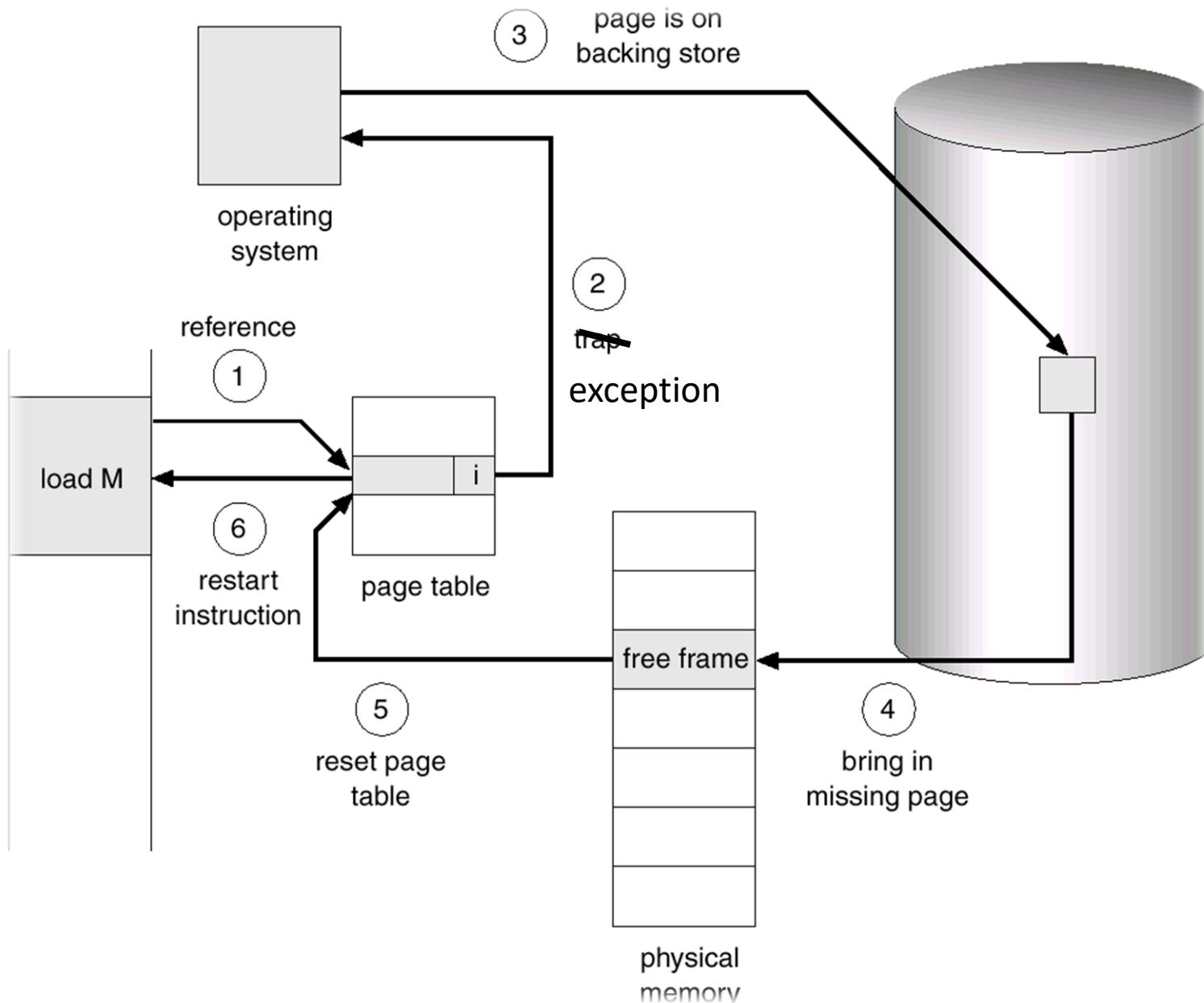
- When a new process is created
- When a process is scheduled for execution
  - TLB flushed for current process
  - MMU reset for the process
  - Process table of next process made current
- When process exits
- When page fault occurs

# OS Involvement With Paging

- When a new process is created
- When a process is scheduled for execution
- **When process exits**
  - OS releases the process page table
  - Frees its pages and disk space
- When page fault occurs

# OS Involvement With Paging

- When a new process is created
- When a process is scheduled for execution
- When process exits
- When page fault occurs



# Page Fault Exception Handling

1. The hardware:
  - Saves program counter
  - Exception leads to kernel entry
2. An assembly routine saves general registers and calls OS
3. OS tried to discover which virtual page is needed
4. OS checks address validation and protection and assign a page frame (page replacement may be needed)

# Page Fault Handling

## 5. If page frame selected is dirty

- Page scheduled to transfer to disk
- Frame marked as busy
- OS suspends the current process
- Context switch takes place

## 6. Once the page frame is clean

- OS looks up disk address where needed page is
- OS schedules a disk operation
- Faulting process still suspended

## 7. When disk interrupts indicates page has arrived

- OS updates page table

# Page Fault Handling

8. Faulting instruction is backed up to its original state before page fault and PC is reset to point to it.
9. Process is scheduled for execution and OS returns to the assembly routine.
10. The routine reloads registers and other state information and returns to user space.

# Virtual Memory & I/O Interaction (Interesting Scenario)

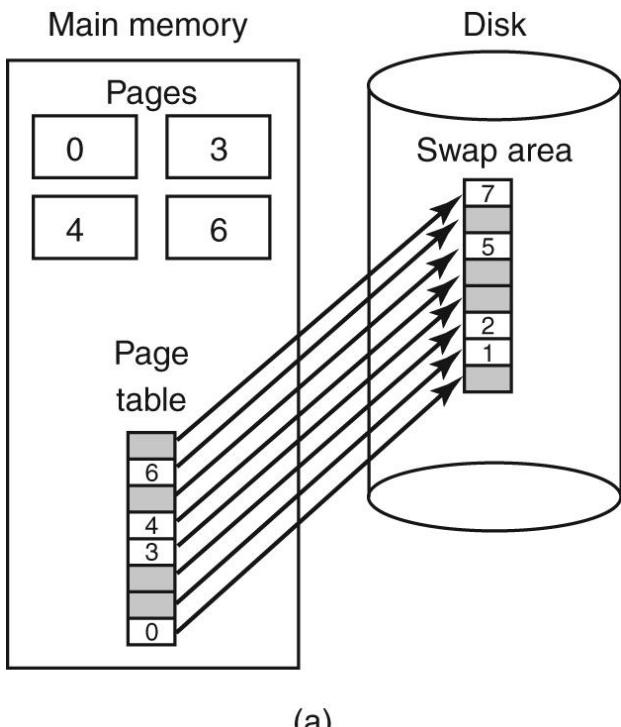
- Process issues a `syscall()` to read a file into a buffer
- Process suspended while waiting for I/O
- New process starts executing
- This other process gets a page fault
- If paging algorithm is global there is a chance the page containing the buffer could be removed from memory
- The I/O operation of the first process will write some data into the buffer and some other on the just-loaded page !

One solution: **Locking (pinning)** pages engaged in I/O so that they will not be removed by replacement algo

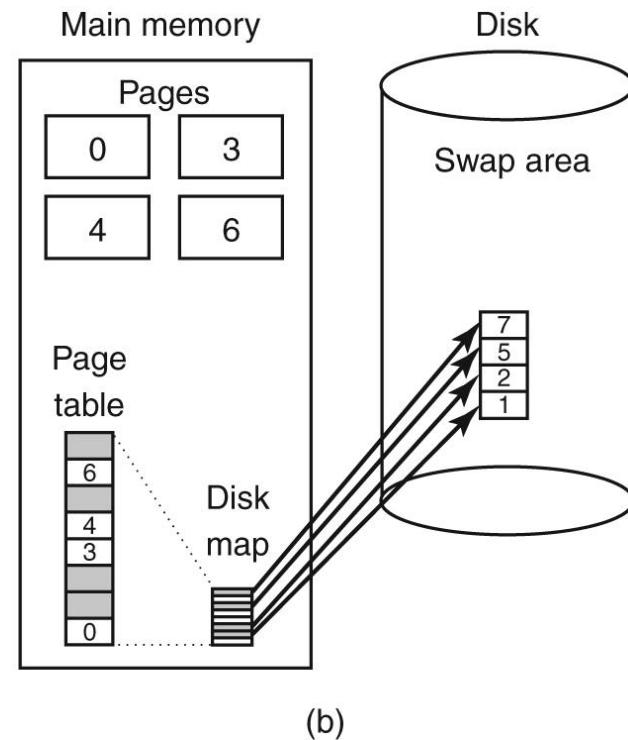
# Backing Store

- Swap area: not a normal file system on it.
- Associates with each process the disk address of its swap area; store in the process table
- Before process starts swap area must be initialized
  - One way: copy all process image into swap area [**static swap area**]
  - Another way: don't copy anything and let the process swap out [**dynamic**]
- Instead of disk partition, one or more preallocated files within the normal file system can be used [Windows uses this approach.]

# Backing Store



**Static Swap Area**



**Dynamic**

# Page Fault Handler

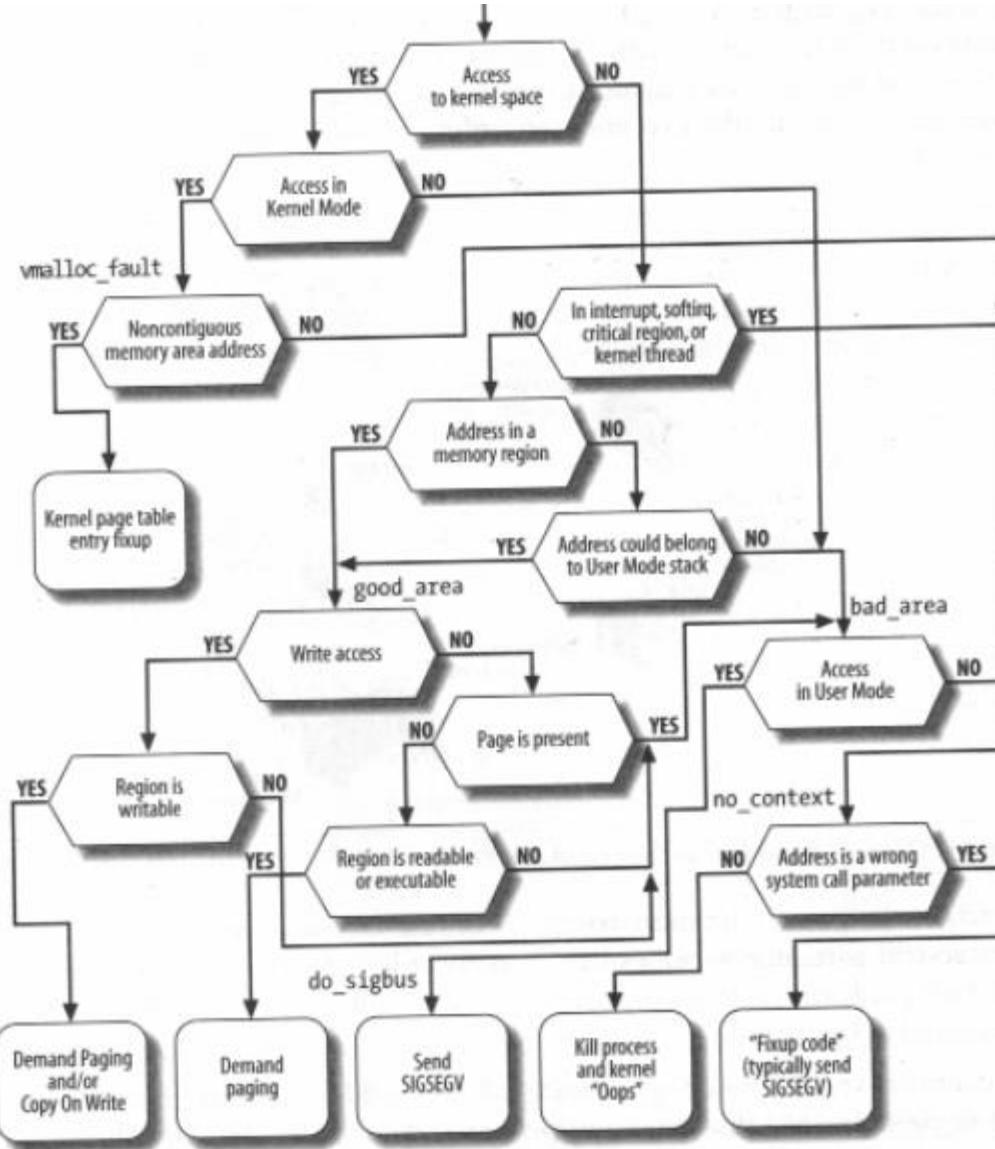


Figure 9-5. The flow diagram of the Page Fault handler

# Memory Mappings

- Each process consists of many memory areas:
  - Aka:
    - segments
    - regions
    - VMAs virtual memory areas.
  - Ex: Heap, stack, code, data, ronly-data, etc.
- Each has different characteristics
  - Protection (executable, rw, rdonly )
  - Fixed, can grow (up or down) [ heap, stack ]
- Each process can have 10s-100s of these.

# Example: emacs VMAs

```
00110000-001a3000 r-xp 00000000 08:01 267740 /usr/lib/libgdk-x11-2.0.so.0.2000.1
001a3000-001a5000 r--p 00093000 08:01 267740 /usr/lib/libgdk-x11-2.0.so.0.2000.1
001a5000-001a6000 rw-p 00095000 08:01 267740 /usr/lib/libgdk-x11-2.0.so.0.2000.1
001a6000-001bf000 r-xp 00000000 08:01 267488 /usr/lib/libatk-1.0.so.0.3009.1
001bf000-001c0000 ---p 00019000 08:01 267488 /usr/lib/libatk-1.0.so.0.3009.1
001c0000-001c1000 r--p 00019000 08:01 267488 /usr/lib/libatk-1.0.so.0.3009.1
001c1000-001c2000 rw-p 0001a000 08:01 267488 /usr/lib/libatk-1.0.so.0.3009.1
001c2000-001cc000 r-xp 00000000 08:01 265243 /usr/lib/libpangocairo-1.0.so.0.2800.0
001cc000-001cd000 r--p 00009000 08:01 265243 /usr/lib/libpangocairo-1.0.so.0.2800.0
001cd000-001ce000 rw-p 0000a000 08:01 265243 /usr/lib/libpangocairo-1.0.so.0.2800.0
001ce000-001d1000 r-xp 00000000 08:01 267773 /usr/lib/libgmodule-2.0.so.0.2400.1
001d1000-001d2000 r--p 00002000 08:01 267773 /usr/lib/libgmodule-2.0.so.0.2400.1
001d2000-001d3000 rw-p 00003000 08:01 267773 /usr/lib/libgmodule-2.0.so.0.2400.1
001d3000-001e8000 r-xp 00000000 08:01 267367 /usr/lib/libICE.so.6.3.0
001e8000-001e9000 r--p 00014000 08:01 267367 /usr/lib/libICE.so.6.3.0
001e9000-001ea000 rw-p 00015000 08:01 267367 /usr/lib/libICE.so.6.3.0
001ea000-001ec000 rw-p 00000000 00:00 0
001ec000-001fb000 r-xp 00000000 08:01 267433 /usr/lib/libXpm.so.4.11.0
001fb000-001fc000 r--p 0000e000 08:01 267433 /usr/lib/libXpm.so.4.11.0
001fc000-001fd000 rw-p 0000f000 08:01 267433 /usr/lib/libXpm.so.4.11.0
```

336 VMAs

```
b75fc000-b763b000 r--p 00000000 08:01 395228 /usr/lib/locale/en_US.utf8/LC_CTYPE
b763b000-b763c000 r--p 00000000 08:01 395233 /usr/lib/locale/en_US.utf8/LC_NUMERIC
b763c000-b763d000 r--p 00000000 08:01 404948 /usr/lib/locale/en_US.utf8/LC_TIME
b763d000-b775b000 r--p 00000000 08:01 395227 /usr/lib/locale/en_US.utf8/LC_COLLATE
b775b000-b776c000 rw-p 00000000 00:00 0
b776c000-b776d000 r--p 00000000 08:01 404949 /usr/lib/locale/en_US.utf8/LC_MONETARY
b776d000-b776e000 r--p 00000000 08:01 404950 /usr/lib/locale/en_US.utf8/LC_MESSAGES/SYS_LC_MESSAGES
b776e000-b776f000 r--p 00000000 08:01 395442 /usr/lib/locale/en_US.utf8/LC_PAPER
b776f000-b7770000 r--p 00000000 08:01 395102 /usr/lib/locale/en_US.utf8/LC_NAME
b7770000-b7771000 r--p 00000000 08:01 404951 /usr/lib/locale/en_US.utf8/LC_ADDRESS
b7771000-b7772000 r--p 00000000 08:01 404952 /usr/lib/locale/en_US.utf8/LC_TELEPHONE
b7772000-b7773000 r--p 00000000 08:01 395529 /usr/lib/locale/en_US.utf8/LC_MEASUREMENT
b7773000-b777a000 r--s 00000000 08:01 269322 /usr/lib/gconv/gconv-modules.cache
b777a000-b777b000 r--p 00000000 08:01 404953 /usr/lib/locale/en_US.utf8/LC_IDENTIFICATION
b777b000-b777d000 rw-p 00000000 00:00 0
bfd9a000-bfe5d000 rw-p 00000000 00:00 0 [stack]
```

# More on memory regions

Memory mapped files:

- Typically **no** swap space required
- The file is the swap space

```
b7773000-b777a000 r--s 00000000 08:01 269322 /usr/lib/gconv/gconv-modules.cache  
b777a000-b777b000 r--p 00000000 08:01 404953 /usr/lib/locale/en_US.utf8/LC_IDENTIFICATION  
b777b000-b777d000 rw-p 00000000 00:00 0  
bfd9a000-bfe5d000 rw-p 00000000 00:00 0 [stack]
```

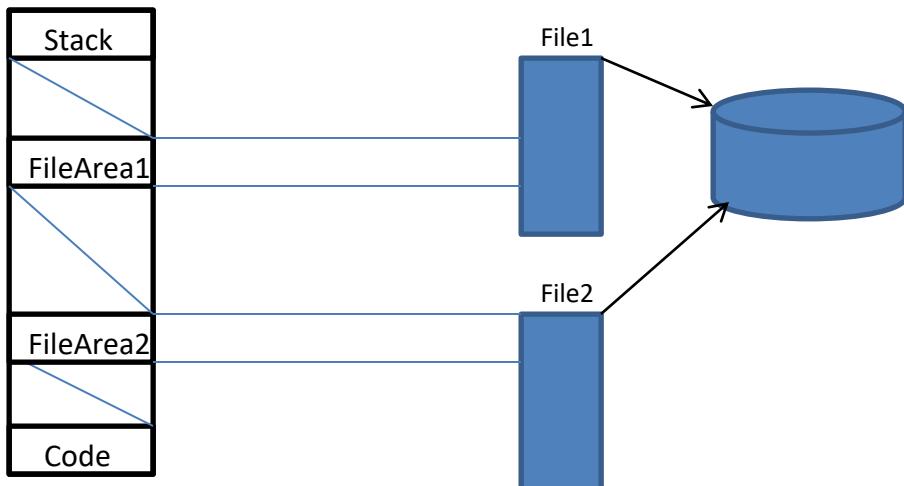
Anonymous memory:  
- Swap space required

```
NAME  
mmap, munmap - map or unmap files or devices into memory  
  
SYNOPSIS  
#include <sys/mman.h>  
  
void *mmap(void *addr, size_t length, int prot, int flags,  
           int fd, off_t offset);  
int munmap(void *addr, size_t length);  
  
DESCRIPTION  
mmap() creates a new mapping in the virtual address space of the calling process.  
The starting address for the new mapping is specified in addr. The length argument  
specifies the length of the mapping.
```

# Memory Mapped Files

- Maps a VMA → file segment ( see `mmap()` `fd` argument )
- It's contiguous
- On PageFault, it fetches the content from file at the appropriate offset into a virtual page of the address space
- On "swapout", writes back to file ( different versions though )

AddressSpace

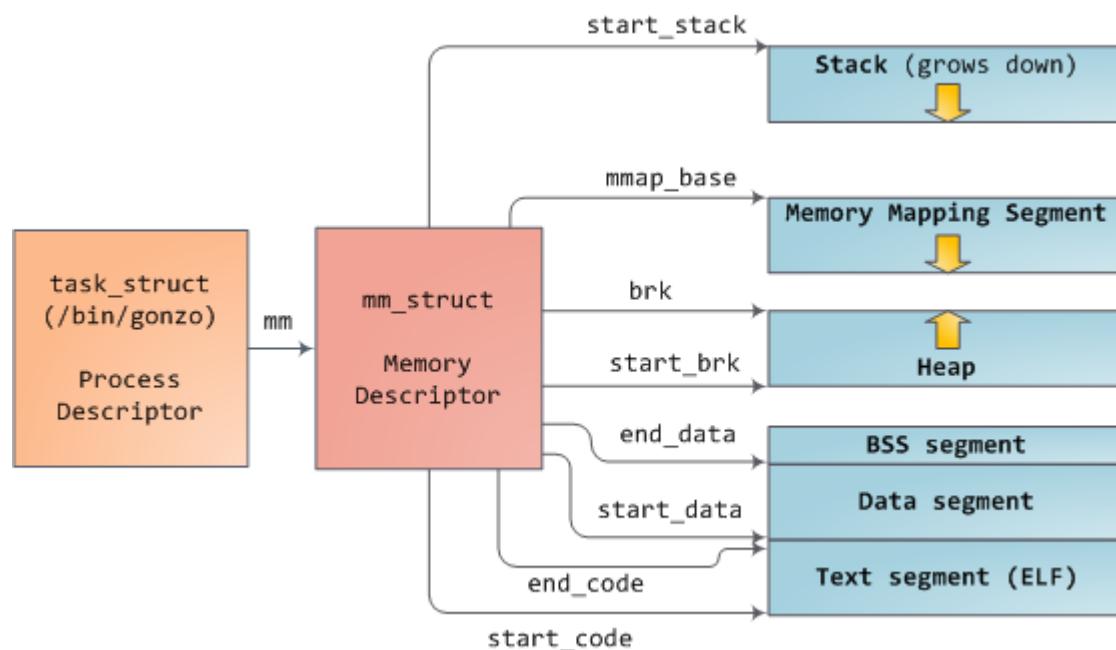


## Benefits

- Can perform load/store operations vs input/output

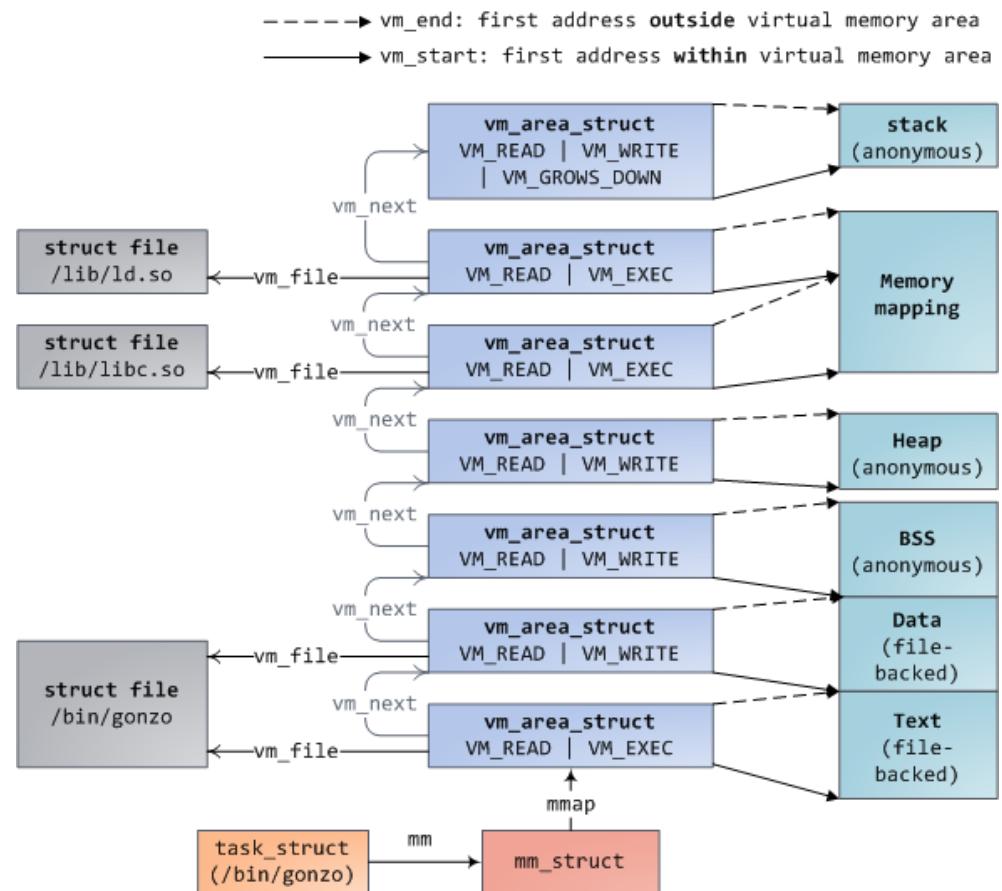
# Linux Example #1

- Objects:



# Linux Example

- VMA areas
- Anonymous vs. filebacked

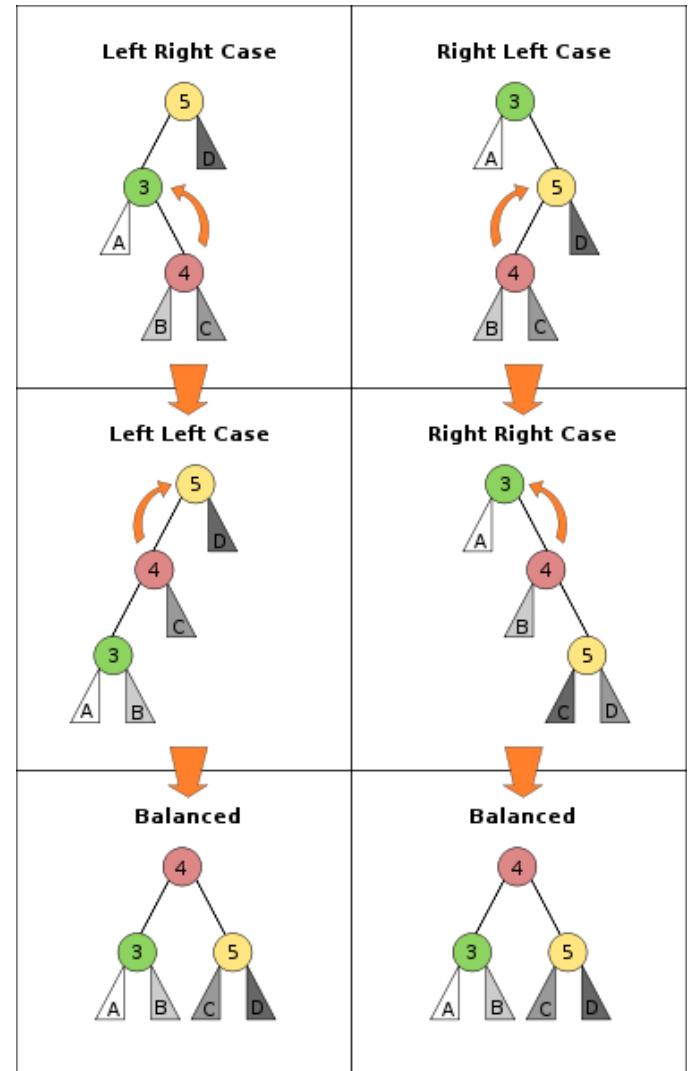


# Organization of Memory Regions

- Cells are by default non-overlapping
  - Called VMA (Virtual Memory Areas)
- Organized as AVL trees
- Identify in  $O(\log N)$  time during pgfault
  - Pgfault(vaddr)  $\rightarrow$  VMA
- Rebalanced when VMA is added or deleted

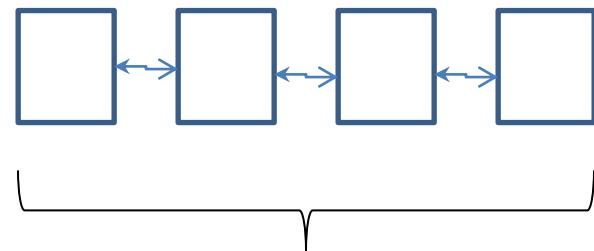
# VMA organization

- Organized as balanced tree
- Node [ start - end ]
- On add/delete  
→ rebalance
- Lookup in  $O(\log(n))$

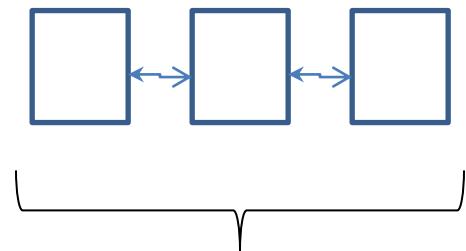


# Linux MemMgmt LRU Approximation

- Arrange the entries into LRU
- Two queues



Active Queue



Inactive Queue

Maintain Ratio

Inactive Queue .. → clean proactively

Inactive Queues create minor faults

FrameTable  
(one entry related to  
each physical frame)

**Minor Faults** forced by setting present=0  
and setting “minor-fault sw-bit” in PTE.  
Move page from inactive to active and set present

# Conclusions

- We are done with Chapter 3
- Main goal
  - Provide Processes with illusion of large and fast memory
- Constraints
  - Speed
  - Cost
  - Protection
  - Transparency
  - Efficiency
- Memory management
  - Paging

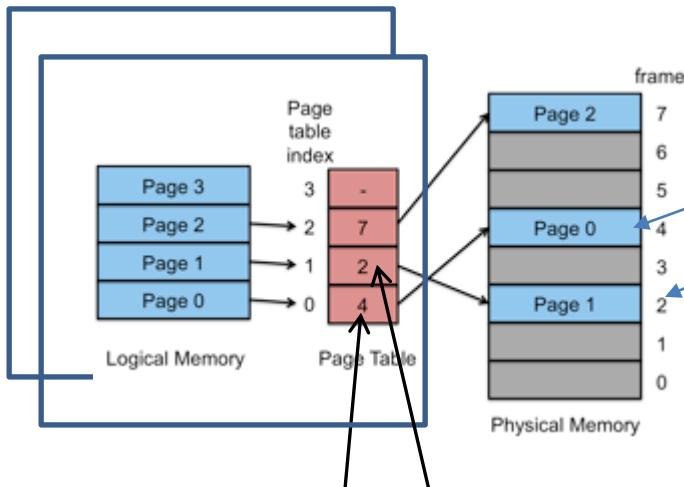
# Lab3 (how to approach)

- You are emulating HW / SW behavior:
- For every instruction simulated
  - Check whether PTE is present
  - If not present → pagefault
    - > OS must resolve:
      - select a victim frame to replace  
(make pluggable with different replacement algorithms)
      - Unmap its current user (UNMAP)
      - Save frame to disk if necessary (OUT / FOUT)
      - Fill frame with proper content of current instruction's address space (IN, FIN,ZERO)
      - Map its new user (MAP)
      - Its now ready for use and instruction
    - Mark reference/modified bit as indicated by instruction

# Lab3

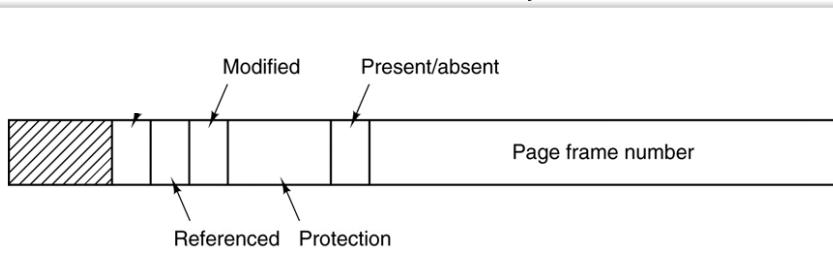
- Create your frame table
- Make all algorithms run through frame table
- Make all algorithms maintains a "hand" from where to start the next "selection"
- On process exit() return all used frames to a free pool and start getting a free frame from there again till free pool is empty; then continue algorithm at hand.  
(this can at times select a frame that was just used ..  
C'est la vie .. You can make algorithms to complicated,  
rather make simple and occasionally not make an optimal  
decision)

# Data Structures (also lab3)

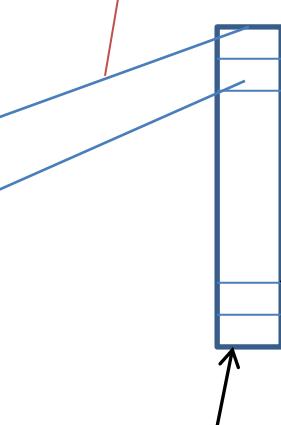


PageTable (one per process)

PageTable Entry (one per virtual page)



reverse mappings



FrameTable:

- This is a data structure the OS maintains to track the usage of each frame by a pagetable (speak reverse mapping).
- one entry related to each physical frame
- Used by OS to keep state for each frame

Inverse mapping  
Reference count  
Locked  
Linkage