



CSCI-GA.2250-001

Operating Systems

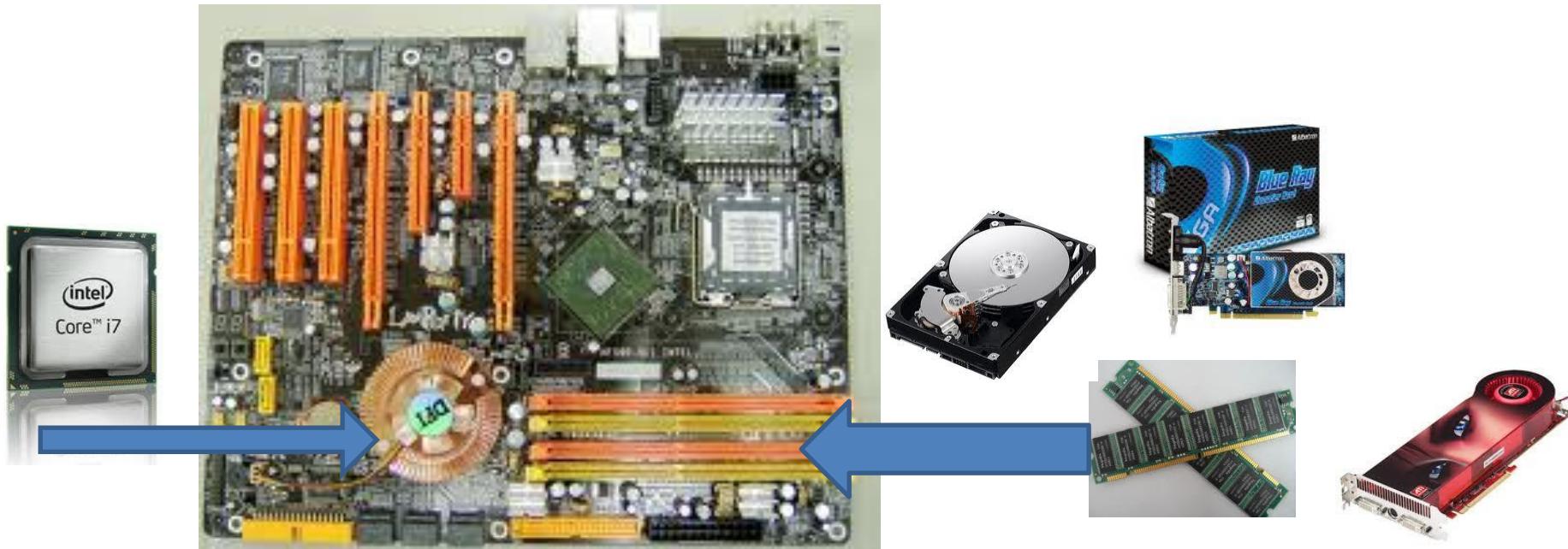
Introduction

Hubertus Franke
frankeh@cims.nyu.edu



Components of a Modern Computer

- One or more processors
- Main memory
- Disks
- Printers
- Keyboard
- Mouse
- Display
- Network interfaces
- I/O devices

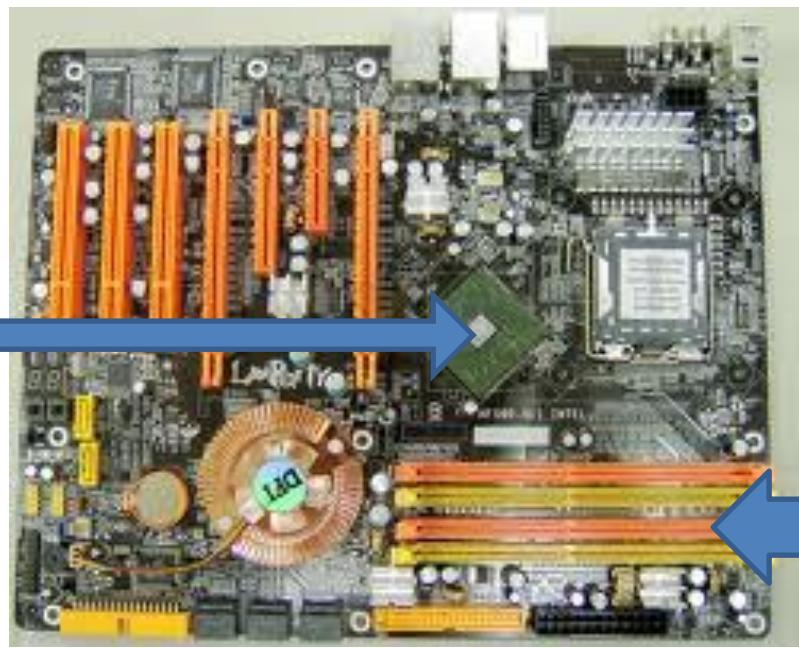


Media
Player

emails

Games

Word
Processing



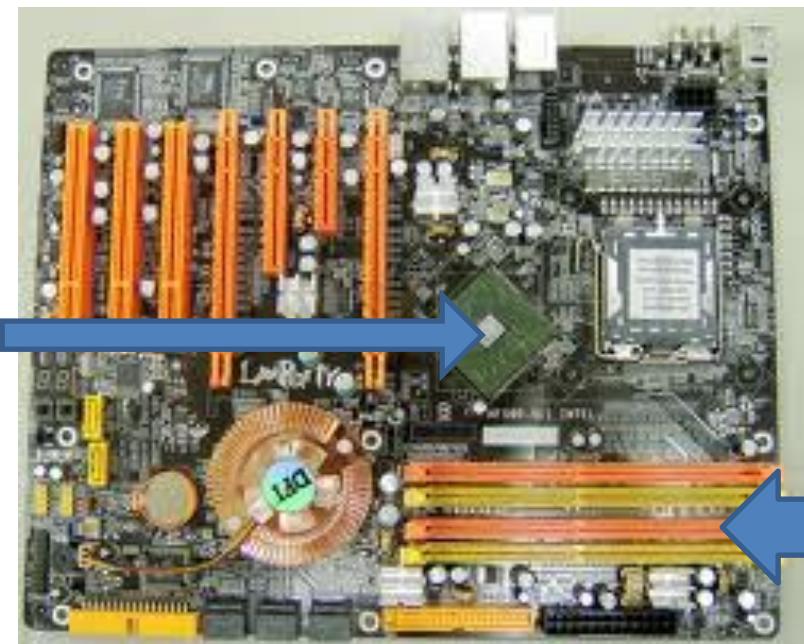
Media
Player

emails

Games

Word
Processing

Does a programmer need to understand all this hardware
in order to write these software programs?



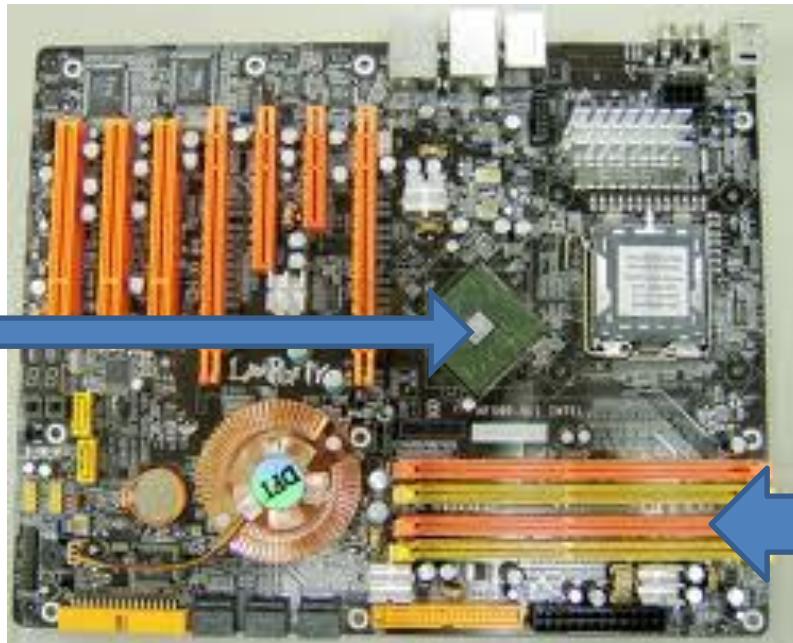
Media
Player

emails

Games

Word
Processing

Operating System



Components of a Modern Computer

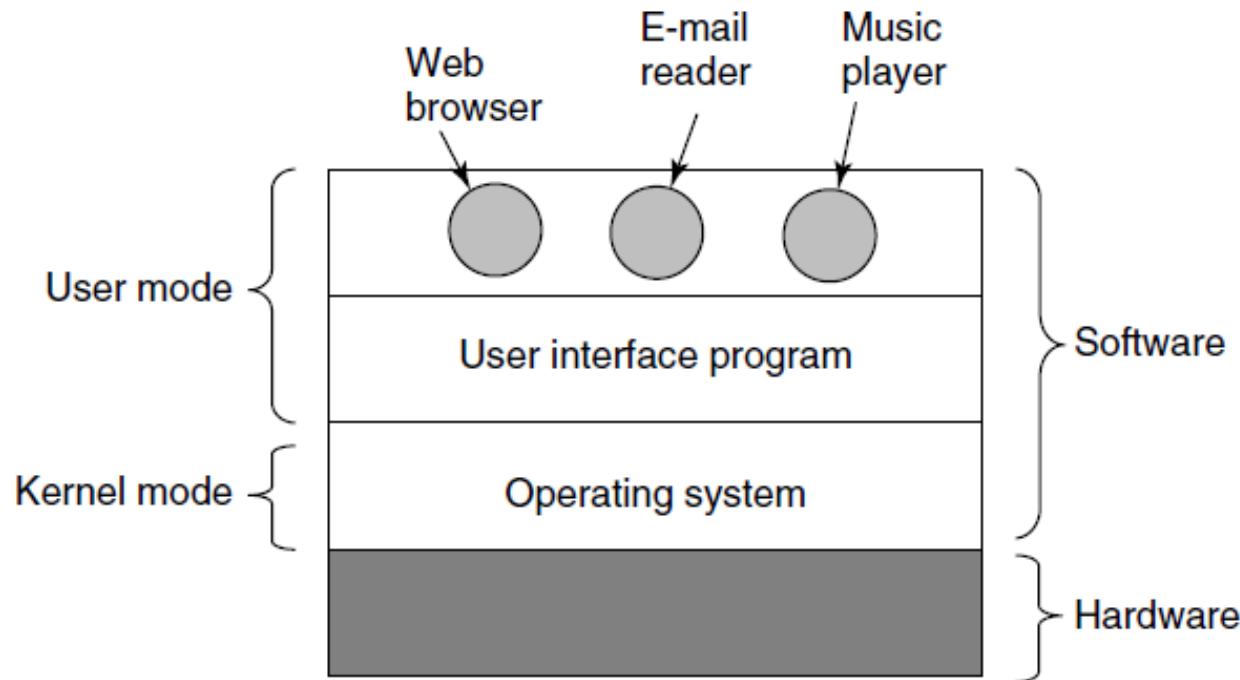
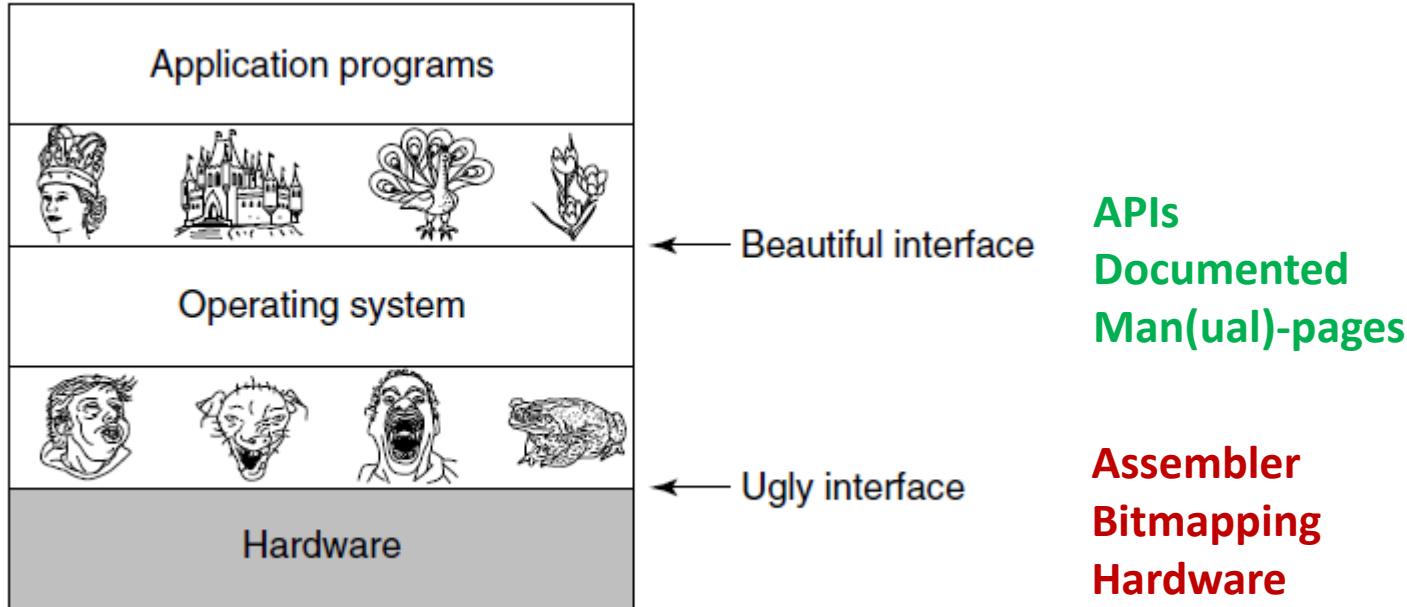


Figure 1-1. Where the operating system fits in.

The Operating System as an Extended Machine



Operating systems turn **ugly hardware** into **beautiful abstractions** (arguable).

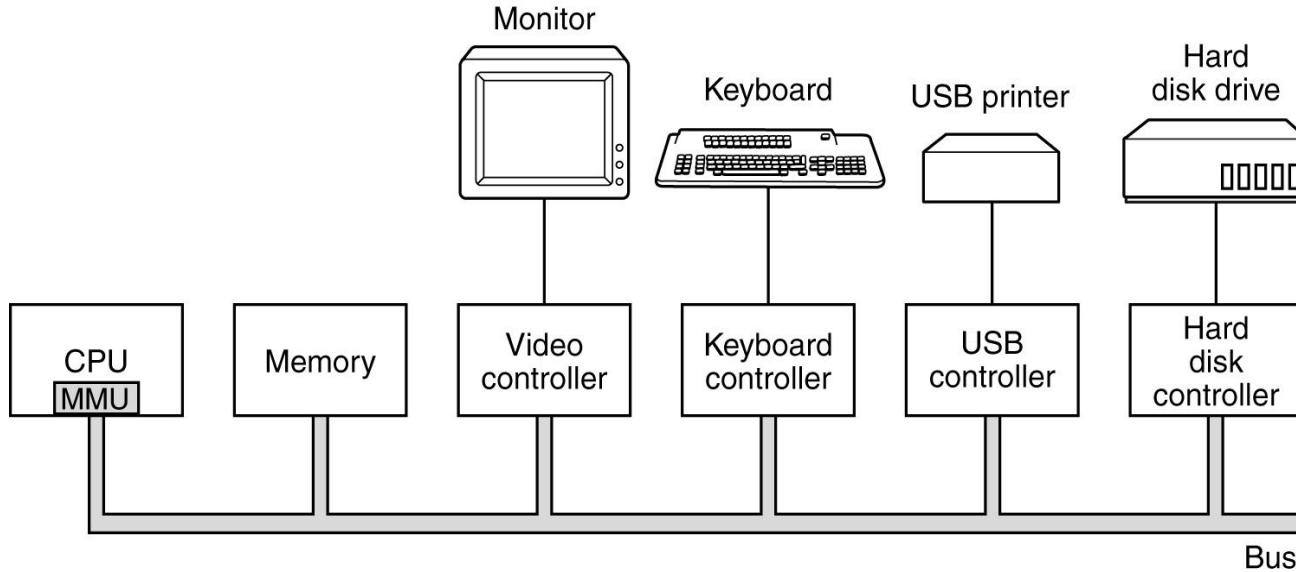
The Operating System as a Resource Manager

- Top down view
 - Provide abstractions to application programs
- Bottom up view
 - Manage pieces of complex systems (hardware and events)
- Alternative view
 - Provide orderly, controlled allocation of resources

The Two Main Tasks of OS

- Provide programmers (and programs) a clean set of abstract resources and services to manipulate these resources
- Manage the hardware resources

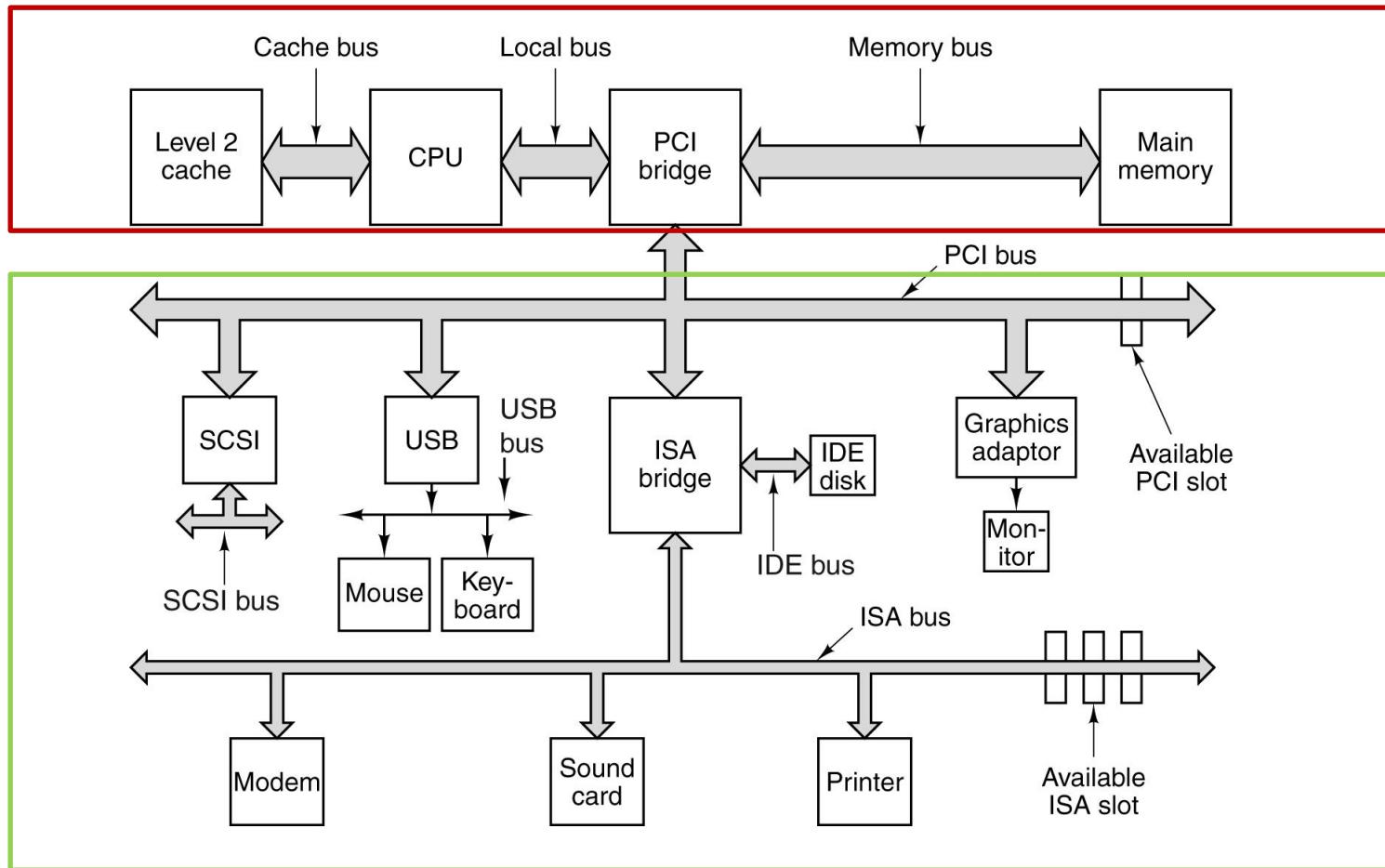
A Glimpse on Hardware



Simplified view: System Components are interlinked through a shared bus and communicate over that bus.

This is done through bus transactions
(e.g. load/store to/from memory, interprocessor notifications,
device accesses, ...)

A Glimpse on Hardware



Reality Check:

In reality there are many buses between the components one set relates to **memory accesses (load/store)** and one to **I/O subsystems**

A few conventions

2^0	=	1
2^1	=	2
2^2	=	4
2^3	=	8
2^4	=	16
2^5	=	32
2^6	=	64
2^7	=	128
2^8	=	256
2^9	=	512
2^{10}	=	1,024
2^{11}	=	2,048
2^{12}	=	4,096
2^{13}	=	8,192
2^{14}	=	16,384
2^{15}	=	32,768
2^{16}	=	65,536
2^{17}	=	131,072
2^{18}	=	262,144
2^{19}	=	524,288
2^{20}	=	1,048,576
2^{21}	=	2,097,152
2^{22}	=	4,194,304
2^{23}	=	8,388,608
2^{24}	=	16,777,216
2^{25}	=	33,554,432
2^{26}	=	67,108,864
2^{27}	=	134,217,728
2^{28}	=	268,435,456
2^{29}	=	536,870,912
2^{30}	=	1,073,741,824
2^{31}	=	2,147,483,648
2^{32}	=	4,294,967,296

- Computer Scientists think in 2^N
- Remember a few tricks
- Know:
 $2^0 .. 2^9$
 $2^{10} \sim 10^3 = 1\text{KB}$
- and the rest is easy (e.g.):
 - $2^{20} \sim 10^6 = 1\text{MB}$
 - $2^{30} \sim 10^9 = 1\text{GB}$
 - $2^{32} = 2^{(30+2)} = 4\text{GB}$
 - $2^{16} = 2^{(10+6)} = 64\text{KB}$

Expected you can do 2^N math on the spot

A few assembler conventions used

- Occasionally we need to use some assembler notation, I utilize a general fictitious notation that should be easily understood
- Loads and stores:
 - basic means to obtain a data item from/to memory
 - `ldw r3,addr` for loading a word of memory (`addr`) into a register (`r3`) (will cover that in a bit)
 - `stw r3,addr` for storing a value in register (`r3`) back to memory (`addr`)
- Arithmetic operations:
 - takes operands and performs basic operations
 - `add r3, r4, r5` ($r3 = r4+r5$)

What Is an OS?

Resources

- Allocation
- Protection
- Reclamation
- Virtualization

Services

- Abstraction
- Simplification
- Convenience
- Standardization

CONTAINER

Makes computer usage simpler

What Is an OS?

Resources

- Allocation
- Protection
- Reclamation
- Virtualization

Finite resources
Competing demands

Examples:

- CPU
- Memory
- Disk
- Network

Government

Limited budget,
Land,
Oil,
Gas,

What Is an OS?

Resources

- Allocation
- **Protection**
- Reclamation
- Virtualization

You can't hurt me
I can't hurt you

Implies some degree of
safety & security

Government

Law and order

What Is an OS?

Resources

- Allocation
- Protection
- **Reclamation**
- Virtualization

The OS gives
The OS takes away

Voluntary at run time
Implied at termination
Involuntary
Cooperative

Government

Income Tax

What Is an OS?

Resources

- Allocation
- Protection
- Reclamation
- **Virtualization**

illusion of infinite
private resources

Memory versus disk
Timeshared CPU

More extreme cases
possible (& exist)

Government

Social security

Operating System

- OS (kernel) is really just a program that runs with special privileges to implement the features of allocation, protection, reclamation and virtualization

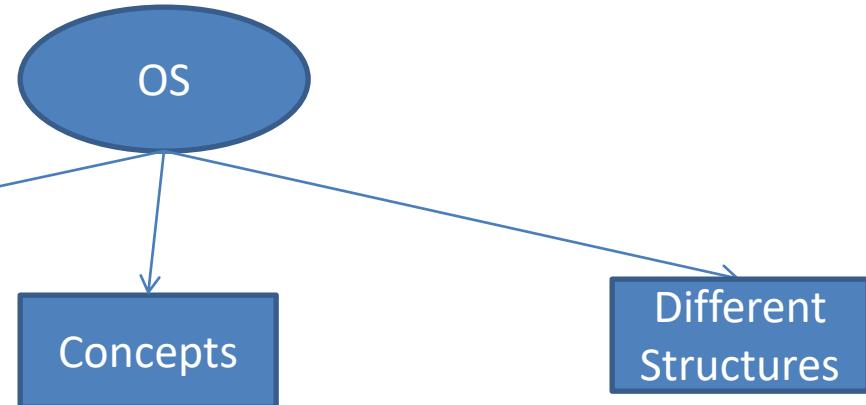
and

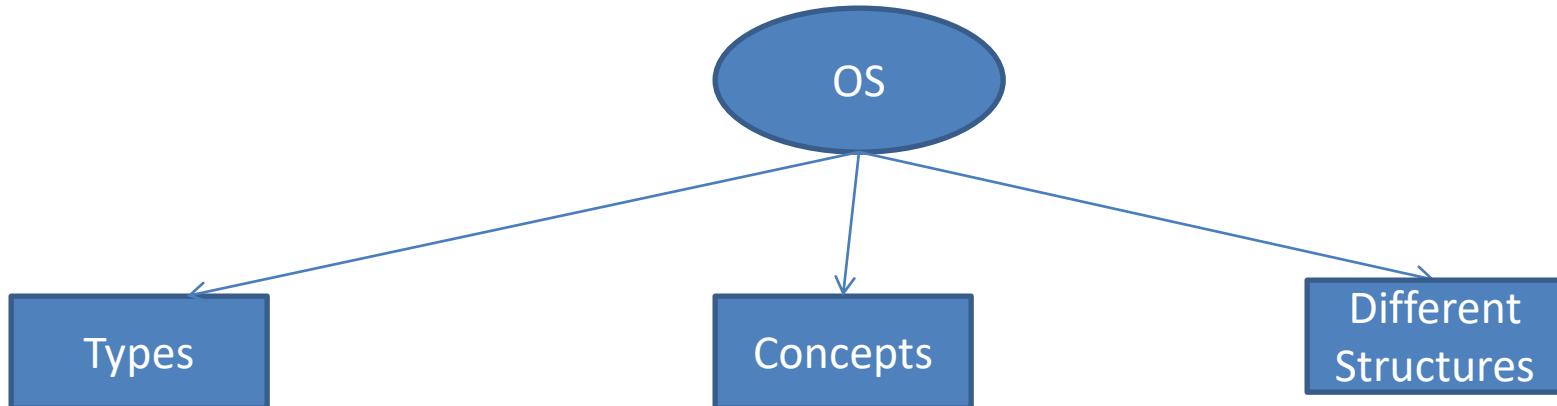
the services that are structured on top of it.

Booting Sequence

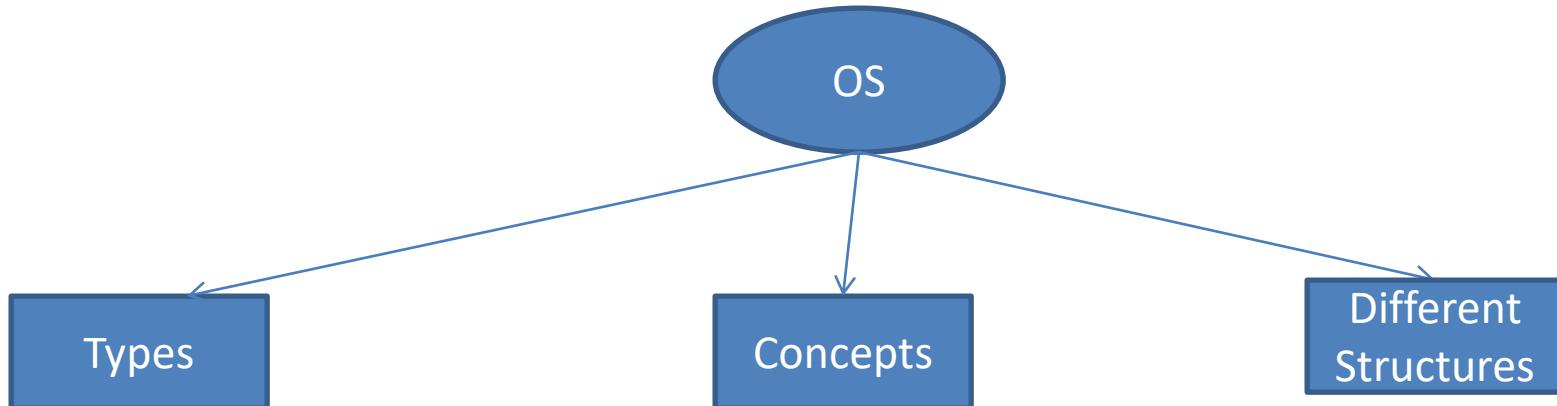
- BIOS starts
 - checks how much RAM
 - keyboard
 - other basic devices
 - BIOS determines boot Device
 - The first sector in boot device is read into memory and executed to determine active partition
 - Secondary boot loader is loaded from that partition.
 - This loaders loads the OS from the active partition and starts it.
 - BIOS: Basic Input/Output System (till ~2010)
 - e.g. UEFI: Unified Extensible Firmware Interface (**UEFI**) is a specification for a software program that connects a computer's firmware to its operating system (OS) and functions as nowadays BIOS
- 
- POST (Power On Self Test)**

Different aspects of
looking at an
operating system



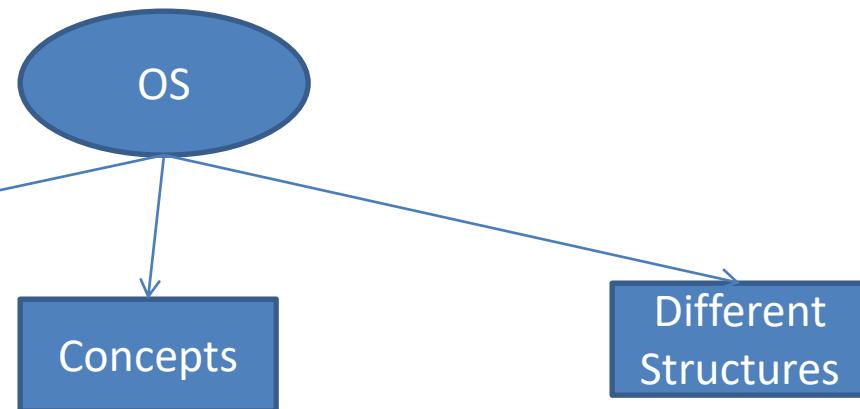


- **Mainframe/supercomputer OS**
 - batch
 - transaction processing
 - timesharing
 - e.g. OS/390
- **Server OS**
- **Multiprocessor OS**
- **PC OS**
- **Embedded OS**
- **Sensor node OS**
- **RTOS**
- **Smart card OS**



- Mainframe OS/supercomputer
 - batch
 - transaction processing
 - timesharing
 - e.g. OS/390
- Server OS
- Multiprocessor OS
- PC OS
- Embedded OS
- Sensor node OS
- RTOS
- Smart card OS

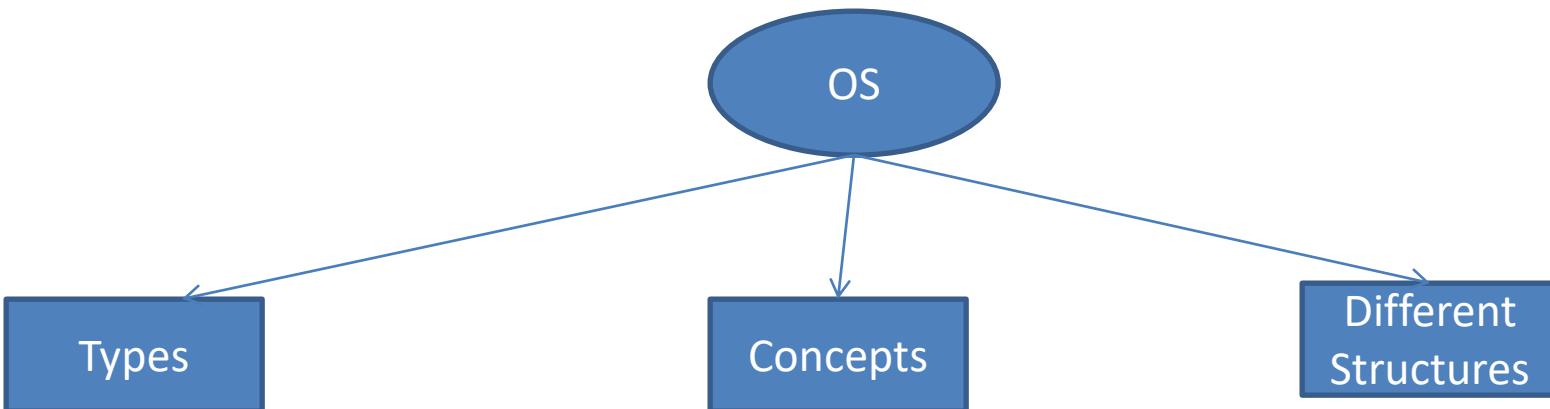
- **Processes**
 - **Its address space**
 - **Its resources**
 - **Process table**
- **Address space**
- **File system**
- **I/O**
- **Protection**



- Mainframe OS/supercomputer
 - batch
 - transaction processing
 - timesharing
 - e.g. OS/390
- Server OS
- Multiprocessor OS
- PC OS
- Embedded OS
- Sensor node OS
- RTOS
- Smart card OS

- Processes
 - Its address space
 - Its resources
 - Process table
- Address space
- File system
- I/O
- Protection

- **Monolithic**
- **Layered systems**
- **Microkernels**
- **Client-server**
- **Virtual machines**



- Mainframe OS
 - batch
 - transaction processing
 - timesharing
 - e.g. OS/390
- Server OS
- Multiprocessor OS
- PC OS
- Embedded OS
- Sensor node OS
- RTOS
- Smart card OS

- Processes
 - Its address space
 - Its resources
 - Process table
- Address space
- File system
- I/O
- Protection

- Monolithic
- Layered systems
- Microkernels
- Client-server
- Virtual machines

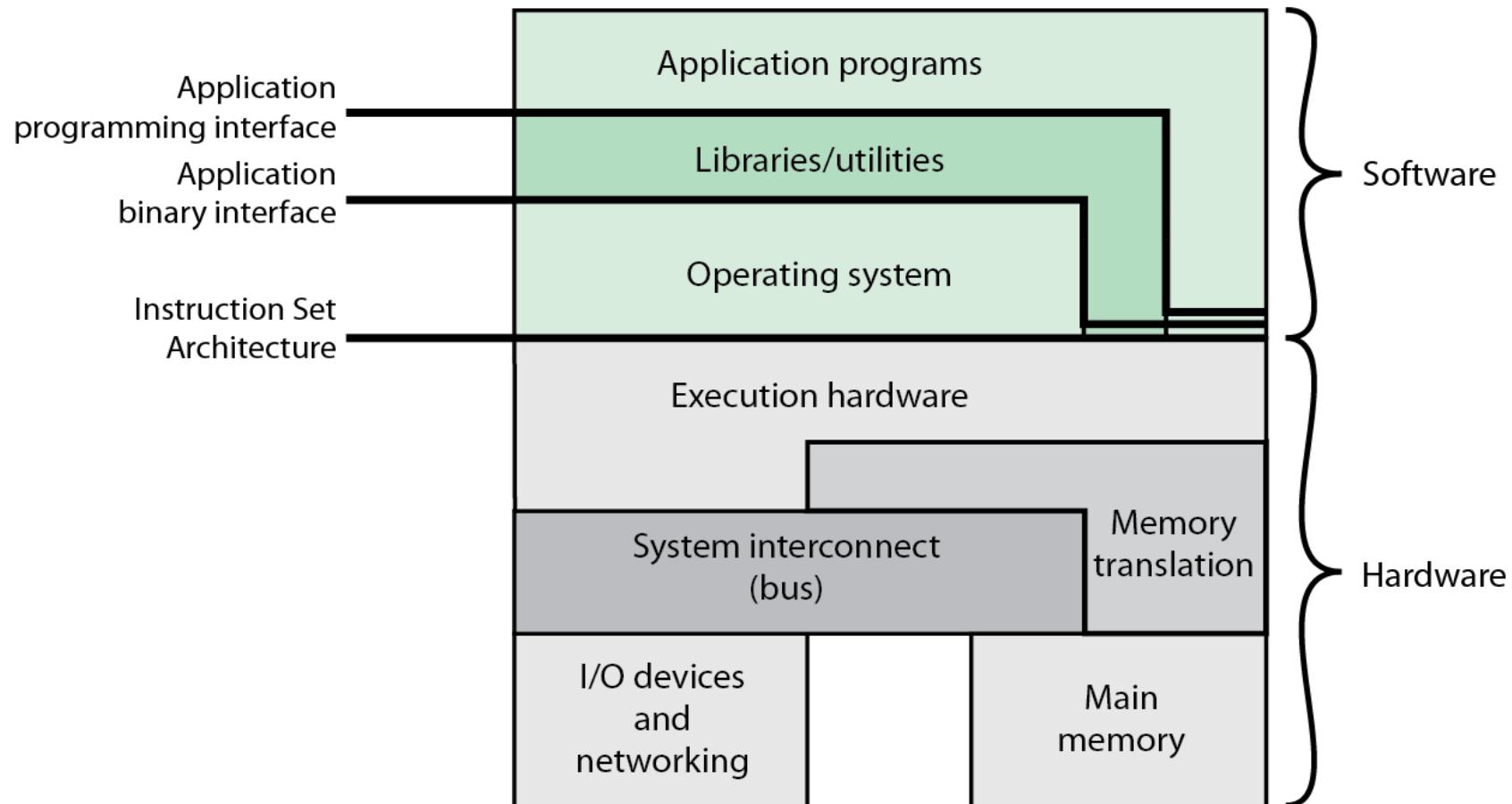
Main objectives of an OS:

- Convenience
- Efficiency
- Ability to evolve

OS Services

- Program development
- Program execution
- Access I/O devices
- Controlled access to files
- System access
- Error detection and response
- Accounting

Hardware and Software Infrastructure



Computer Hardware and Software Infrastructure

In a nutshell



- OS is really a manager:
 - programs, applications, and processes are the customers
 - The hardware provide the resources
- OS works in different environments and under different restrictions
(supercomputers, workstations, notebooks, tablets, smartphones, real-time, ...)

History of Operating Systems

- *"We can chart our future clearly and wisely only when we know the path which has led to the present."*
 - Adlai E. Stevenson, Lawyer and Politician
- First generation 1945 - 1955
 - vacuum tubes, plug boards (no OS)
- Second generation 1955 - 1965
 - transistors, batch systems
- Third generation 1965 - 1980
 - ICs and multiprogramming
- Fourth generation 1980 - present
 - server computers
 - personal computers
- Fifth generation 2005 - present
 - hand-held devices, sensors

History of Operating Systems (1945-55)

- Programming and Control tied to the Computer

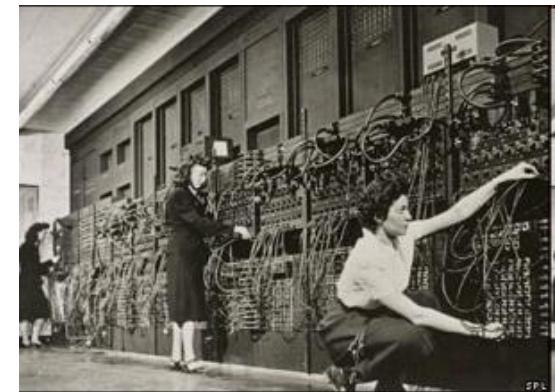
Defining characteristics of some early digital computers of the 1940s (In the history of computing hardware)

Name	First operational	Numerical system	Computing mechanism	Programming	Turing complete
Zuse Z3 (Germany)	May 1941	Binary floating point	Electro-mechanical	Program-controlled by punched 35 mm film stock (but no conditional branch)	Yes (1998)
Atanasoff-Berry Computer (US)	1942	Binary	Electronic	Not programmable—single purpose	No
Colossus Mark 1 (UK)	February 1944	Binary	Electronic	Program-controlled by patch cables and switches	No
Harvard Mark I – IBM ASCC (US)	May 1944	Decimal	Electro-mechanical	Program-controlled by 24-channel punched paper tape (but no conditional branch)	No
Colossus Mark 2 (UK)	June 1944	Binary	Electronic	Program-controlled by patch cables and switches	No
Zuse Z4 (Germany)	March 1945	Binary floating point	Electro-mechanical	Program-controlled by punched 35 mm film stock	Yes
ENIAC (US)	July 1946	Decimal	Electronic	Program-controlled by patch cables and switches	Yes
Manchester Small-Scale Experimental Machine (Baby) (UK)	June 1948	Binary	Electronic	Stored-program in Williams cathode ray tube memory	Yes
Modified ENIAC (US)	September 1948	Decimal	Electronic	Read-only stored programming mechanism using the Function Tables as program ROM	Yes
EDSAC (UK)	May 1949	Binary	Electronic	Stored-program in mercury delay line memory	Yes
Manchester Mark 1 (UK)	October 1949	Binary	Electronic	Stored-program in Williams cathode ray tube memory and magnetic drum memory	Yes
CSIRAC (Australia)	November 1949	Binary	Electronic	Stored-program in mercury delay line memory	Yes

Source: wikipedia

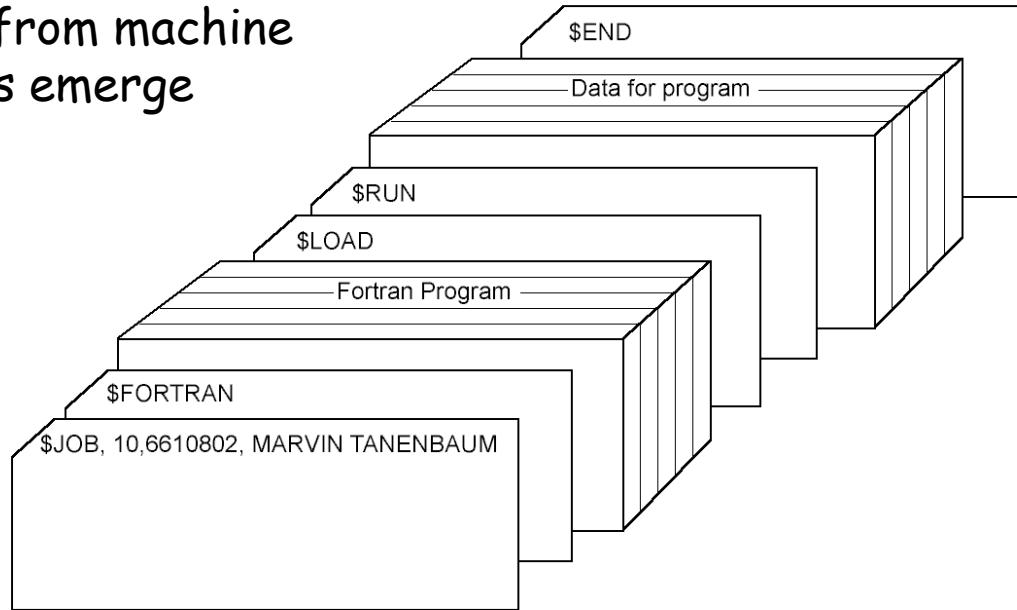
History of Operating Systems (1945-1955)

- Vacuum tubes, plug boards (no OS)
 - ENIAC (UPenn 1944)
 - 30 tons, 150m, 5000calcs/sec
 - Zuse's Z3 (1941)
 - 2000 relays
 - 22 bit words
 - 5-10 Hz
- What's a bug?



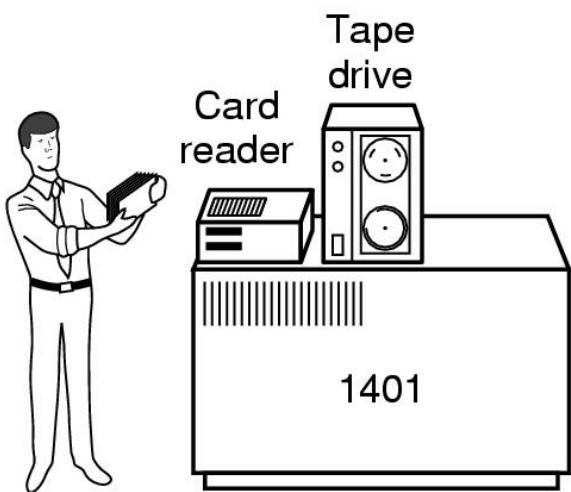
History of Operating Systems (1955-65)

- Emergence of the Mainframe
- Programmers isolated from machine
- Programming Languages emerge
 - Fortran
 - Cobol

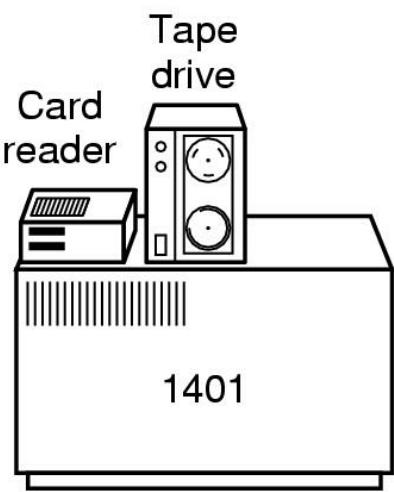


- Structure of a typical JCL job - 2nd generation
- Single user
- Programmer/User as the operator
- Secure, but inefficient use of expensive resources
- Low CPU utilization-slow mechanical I/O devices

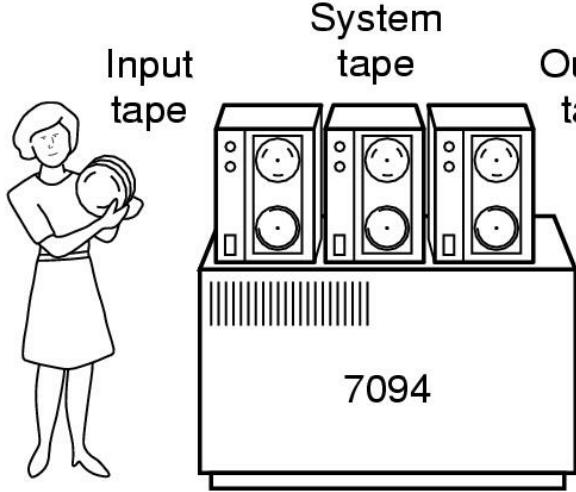
History of Operating System (1955-65)



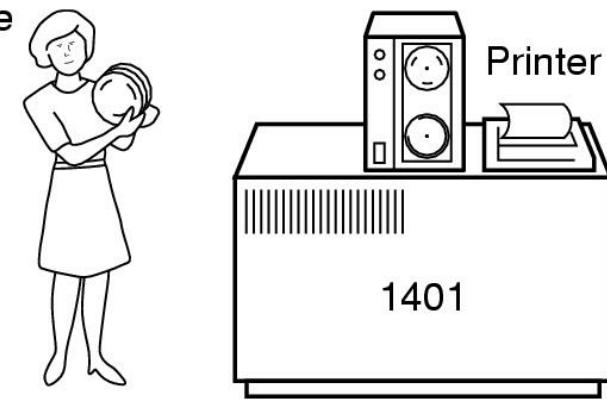
(a)



(b)



(c)



(e)

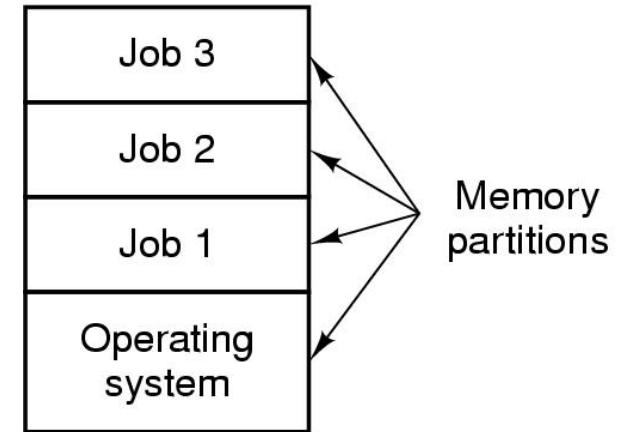


Early batch system

- bring cards to 1401
- read cards to tape
- put tape on 7094 which does computing
- put tape on 1401 which prints output

History of Operating Systems (1965-80)

- Multiprogramming systems
 - Multiple jobs in memory - 3rd generation
 - Allow overlap of CPU and I/O activity
 - Polling/Interrupts, Timesharing
 - Spooling
- Different types
 - Epitomized by the IBM 360 machine
 - MFT (IBM OS/MFT) Fixed Number of Tasks
 - MVT (IBM OS/MVT) Variable Number of Tasks
- Birth of Modern Operating System Concepts
 - Time Sharing: when and what to run → scheduling
 - Resource Control: memory management, protection



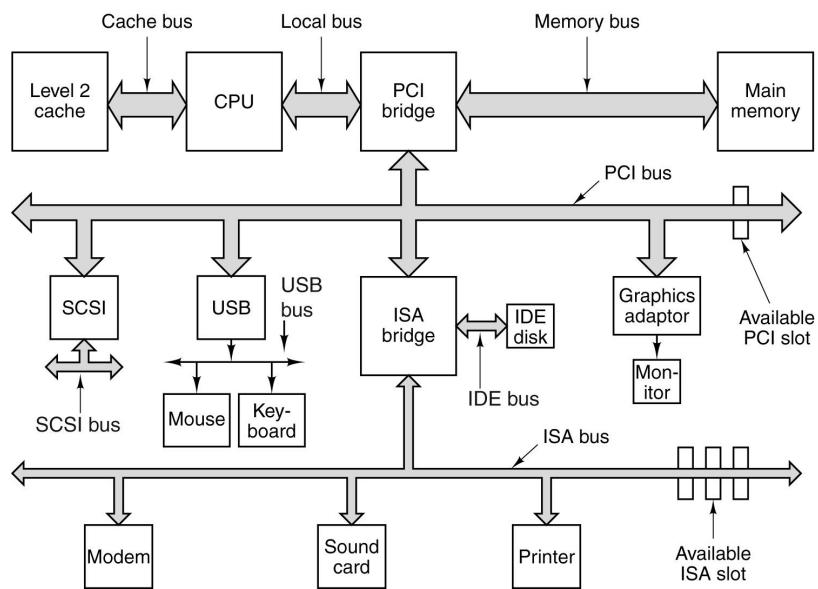
The Operating System Jungle / Zoo (1980-present)

- Mainframe operating systems
- Server operating systems
- Multiprocessor operating systems
- Personal computer operating systems
- Real-time operating systems
- Embedded operating systems
- Smart card operating systems
- Cellphone/tablet operating systems
- Sensor operating systems

Computer Architecture

(a closer look)

We must know and understand
what is actually managed by an OS



Processors

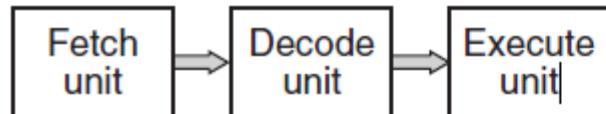
- Each CPU has a specific set of instructions
 - ISA (Instruction Set Architecture) largely epitomized in the assembler
 - RISC: Sparc, MIPS, PowerPC
 - CISC: x86, zSeries
- All CPUs contain
 - General registers inside to hold key variables and temporary results
 - Special registers visible to the programmer
 - Program counter contains the memory address of the next instruction to be fetched
 - Stack pointer points to the top of the current stack in memory
 - PSW (Program Status Word) contains the condition code bits which are set by comparison instructions, the CPU priority, the mode (user or kernel) and various other control bits.

How Processors Work

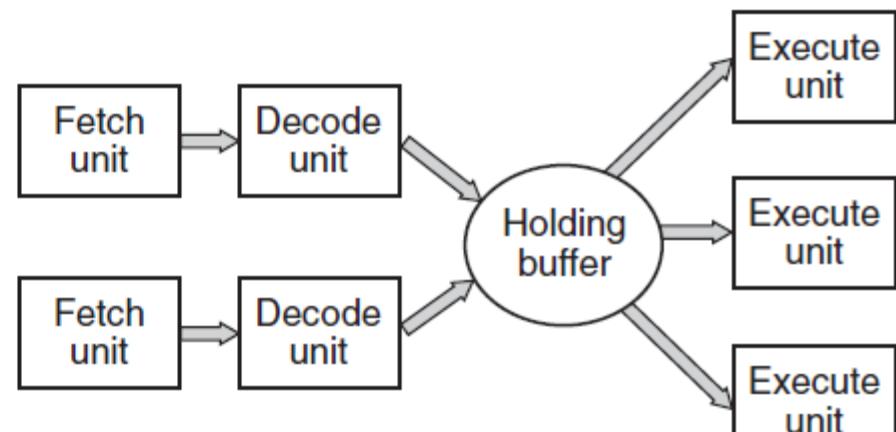
- Execute instructions

- CPU cycles

- Fetch (from mem) → decode → execute
 - Program counter (PC)
 - When is PC changed?
 - Pipeline: fetch n+2 while decode n+1 while execute n



(a)



(b)

(a) A three-stage pipeline.

(b) A superscalar CPU.

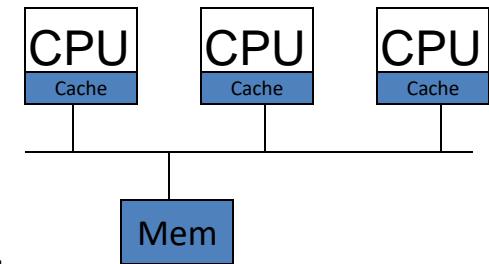
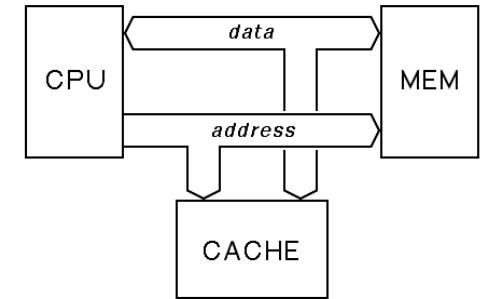
Memory-Storage Hierarchy

Latency		Capacity
1ns	Registers	32+32
10ns	Cache	8KB – 2MB (L1 – L3)
100ns	Main memory	GBs - TBs
10msec	Magnetic disk	10s * TBs
10secs	Magnetic tape	500s * TBs

- Other metrics:
 - Bandwidth (e.g. MemBandwidth 30GB/s → 200GB/s, Disk ~70-200MB/s)
- What can an OS do to increase its “performance”
 - Active management where to place data !!!

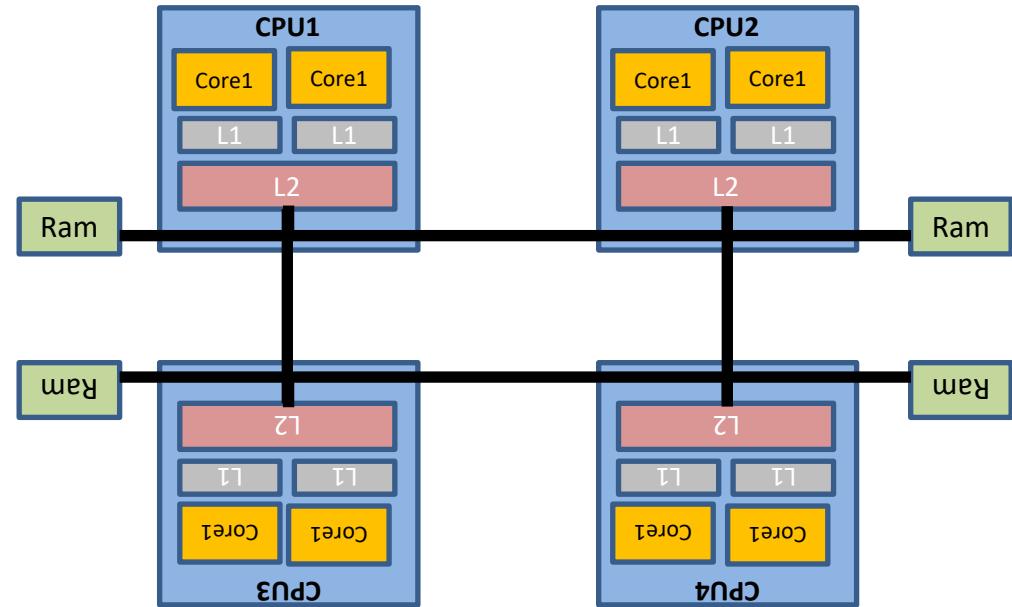
CPU Caches

- Principle:
 - Data/Instruction that were recently used are “likely” used again in short period
 - Caching is principle used in “many” subsystems (I/O, filesystems, ...)
[hardware and software]
- Cache hit:
 - no need to access memory
- Cache miss:
 - data obtained from mem, possibly update cache
- Issues
 - Operation **MUST** be correct
 - Cache management and Coherency for Memory done in hardware
 - Data can be in read state in multiple caches but only in one cache when in write state



Example of real cache/memory access times

- Modern systems have multiple CPUs with their own attached memory and multiple level of caches.
- Non-uniform memory access.



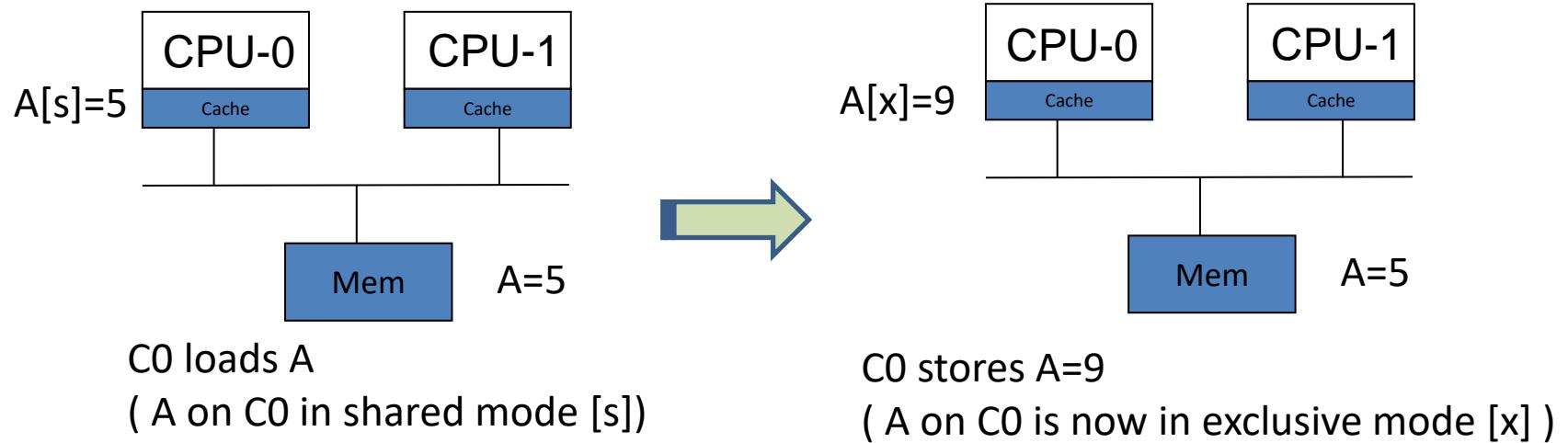
Core i7 Xeon 5500 Series Data Source Latency (approximate)

[Pg. 22]

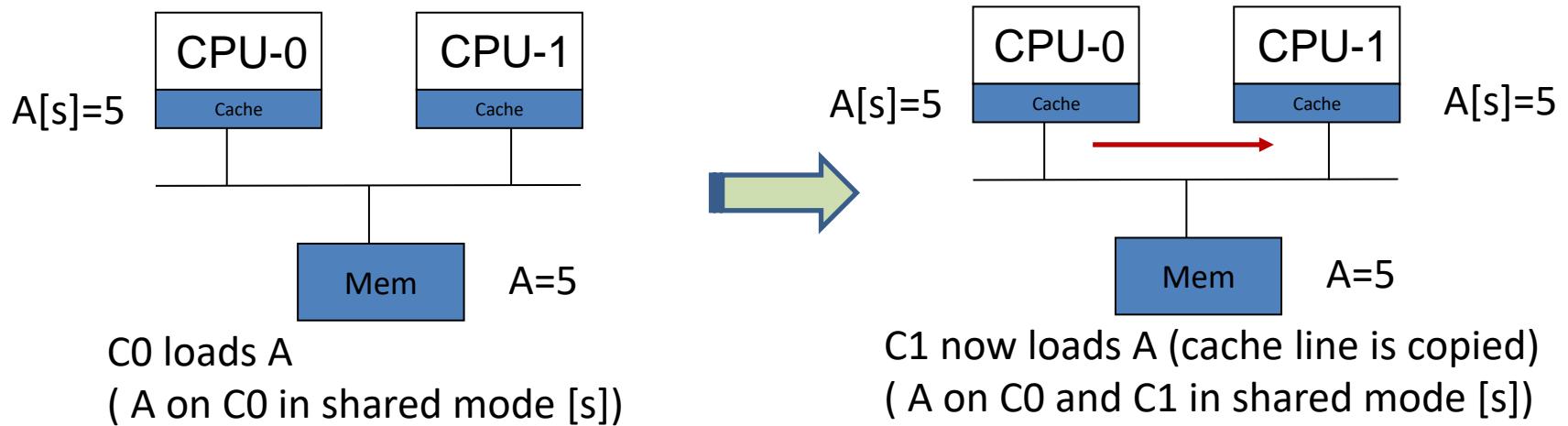
local L1 CACHE hit,	~4 cycles (2.1 - 1.2 ns)
local L2 CACHE hit,	~10 cycles (5.3 - 3.0 ns)
local L3 CACHE hit, line unshared	~40 cycles (21.4 - 12.0 ns)
local L3 CACHE hit, shared line in another core	~65 cycles (34.8 - 19.5 ns)
local L3 CACHE hit, modified in another core	~75 cycles (40.2 - 22.5 ns)
remote L3 CACHE (Ref: Fig.1 [Pg. 5])	~100-300 cycles (160.7 - 30.0 ns)
local DRAM	~60 ns
remote DRAM	~100 ns

Cache Coherency

Scenario 1

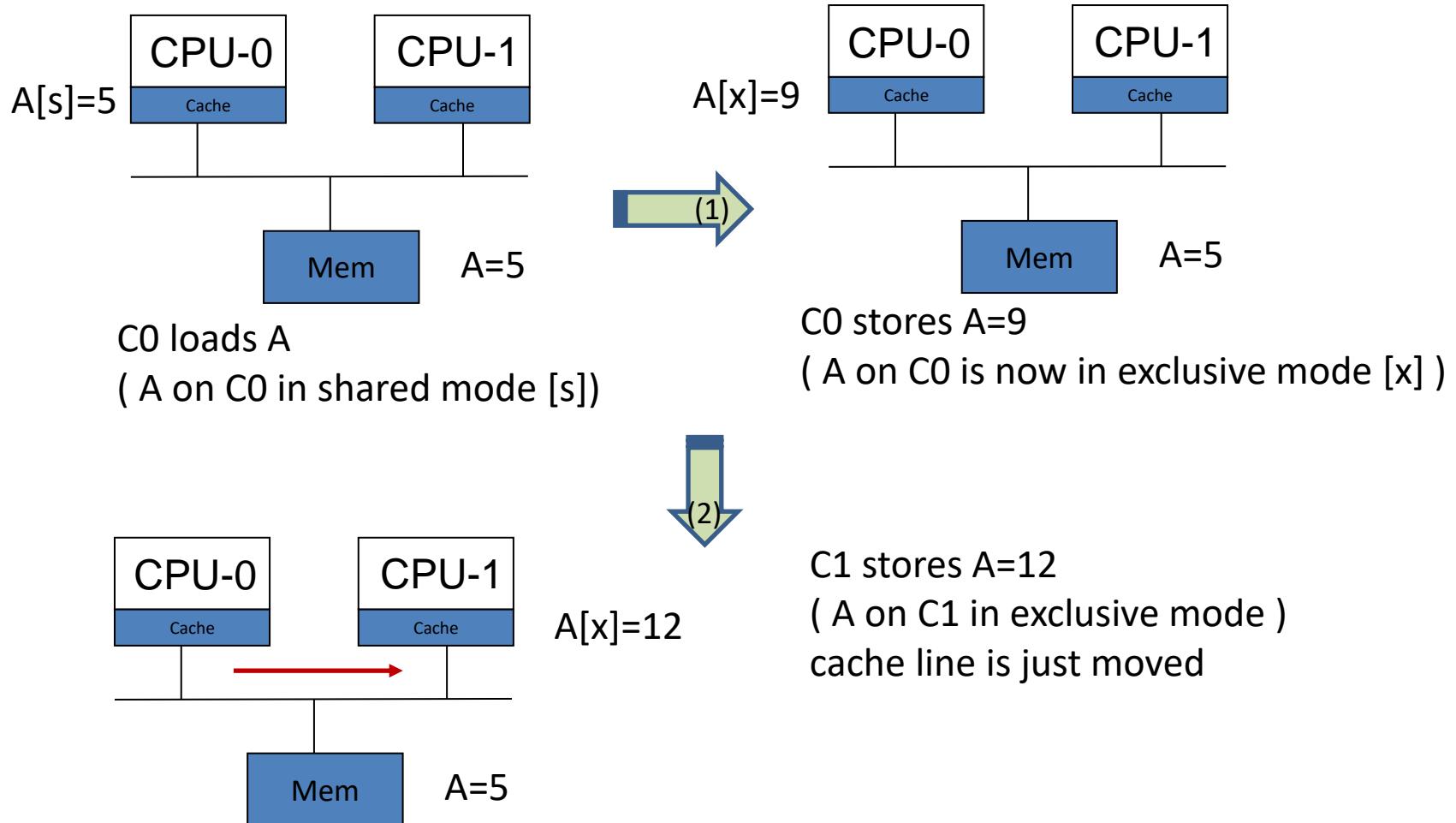


Scenario 2



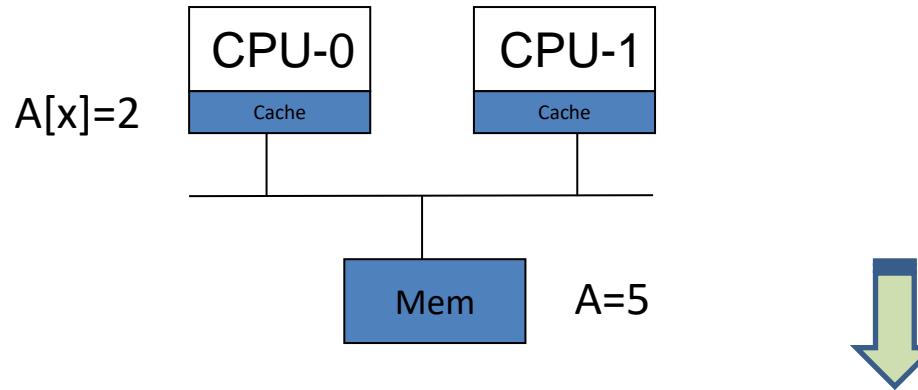
Cache Coherency

Scenario 3



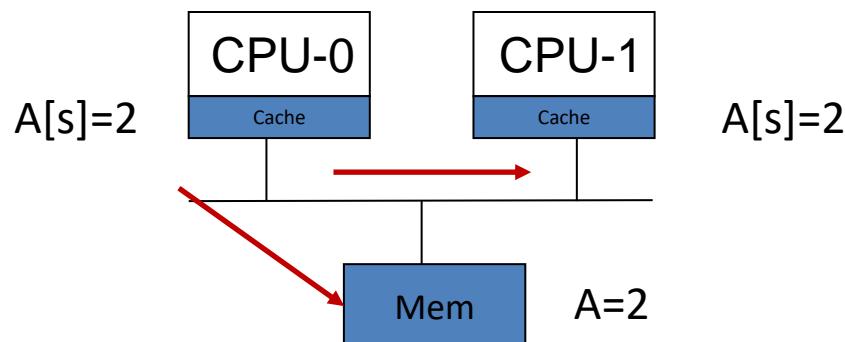
Cache Coherency

Scenario 4



C1 loads A

Forces C0 to write back A to mem
(or lowest level of shared cache)
and put A into shared mode for C0/C1

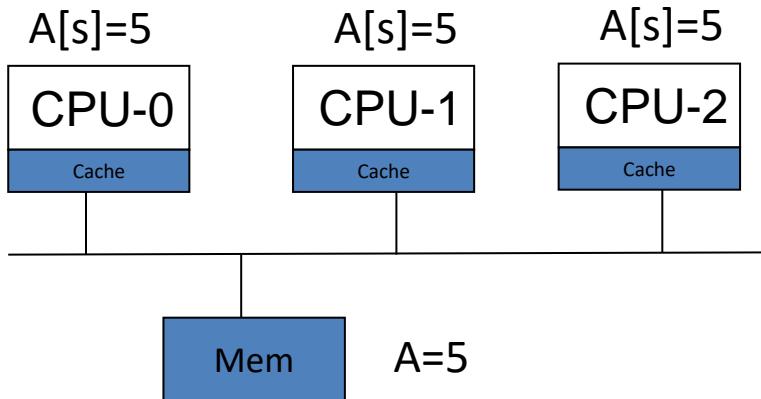


Imperative at all times

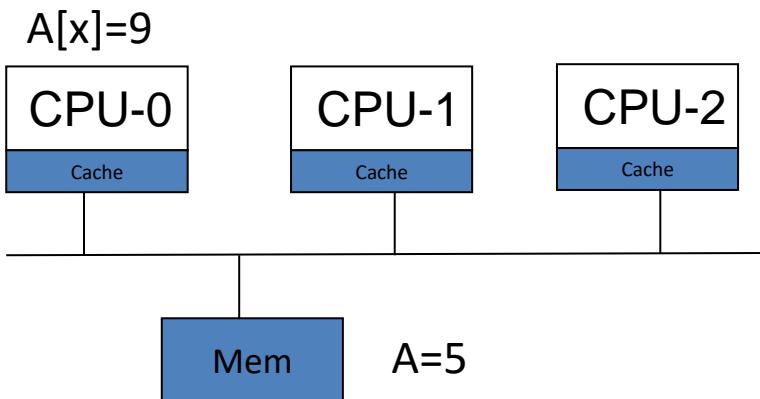
- A can be in [s] across several CPUs
- A can be at most in ONE CPU in [x] and nowhere else in [s]

Cache Coherency

Scenario 5



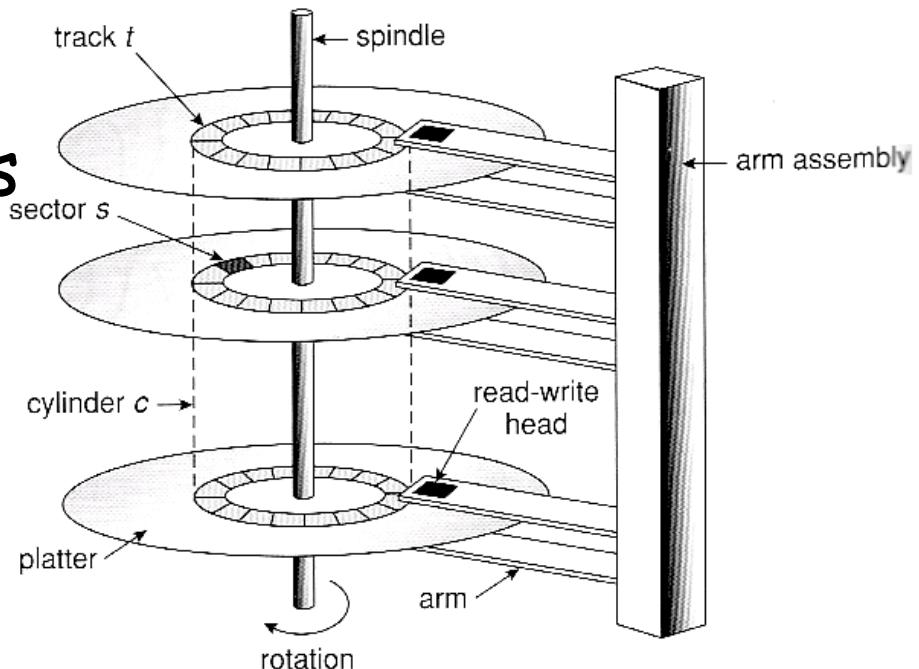
Both thread C0, C1, C2 load A
(A on C0, C1, C2 in shared mode)



Now C0 sets A=9
All cache lines on other cpus must be invalidated and A on C0 in [x]

Example of Device (resource and operation)

- Disk:
 - Multiple-subdevices
 - Translations
 - Block \rightarrow sector
 - Head Movement
 - Seek Time
 - Data Placement



Abraham Silberschatz, Greg Gagne, and Peter Baer Galvin, "Operating System Concepts, Eighth Edition ", Chapter 12

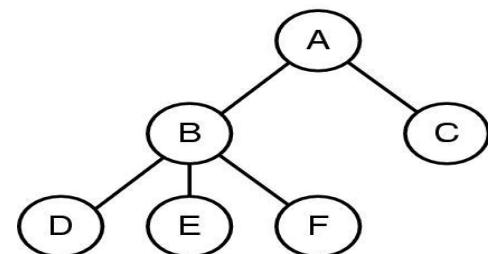
- Power Management

OS Major Components

- Process and thread management
- Resource management
 - CPU
 - Memory
 - Device (I/O)
- File system
- Bootstrapping

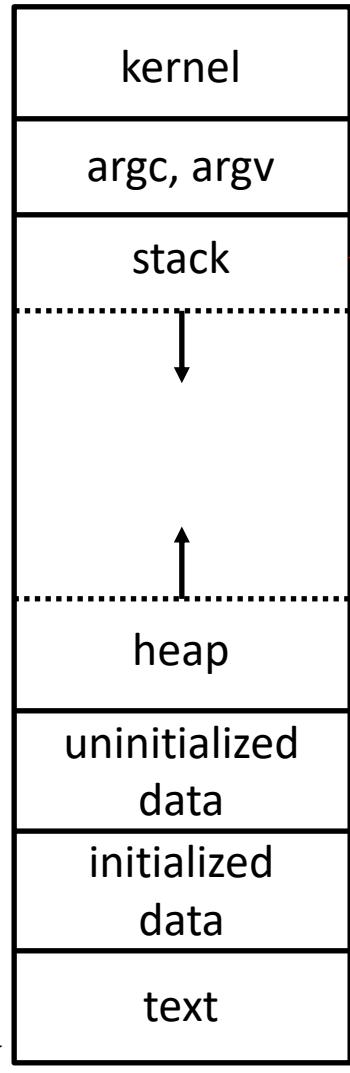
Process: a running program

- A process includes
 - Address space
 - Process table entries (state, registers)
 - Open files, thread(s) state, resources held
- A process tree
 - A created two child processes, B and C
 - B created three child processes, D, E, and F



Address Space: A view of program layout

High memory
0xffffffff



```
#include <stdio.h>
#include <stdlib.h>

#define NUMS (4)

int a;
int b = 2;
int x;
int y = 3;

int main(int argc, char* argv[])
{
    int *values;
    int i;

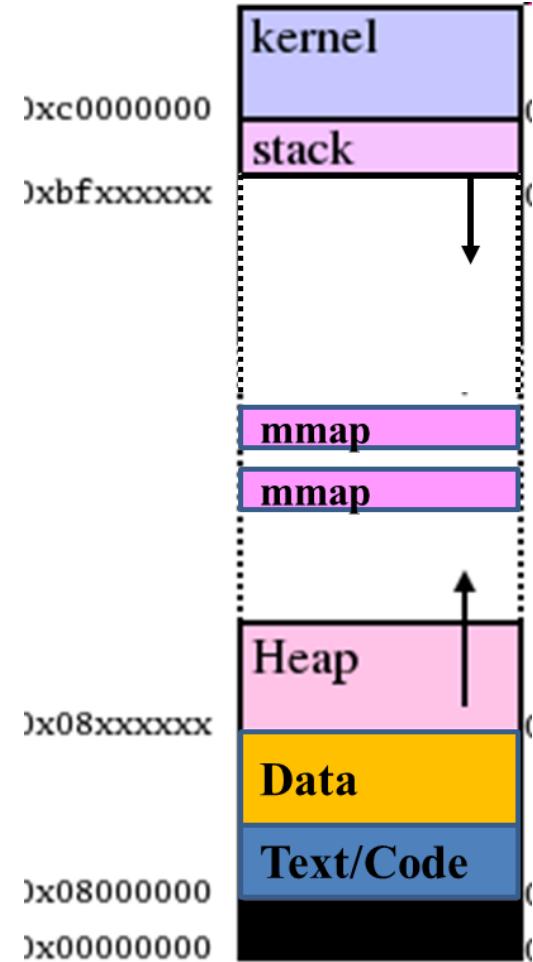
    values = (int*) malloc(NUMS*sizeof(int));

    for (i=0 ; i<NUMS; i++)
        values[i] = i;

    return 0;
}
```

Address Space

- Processes don't Id/st with physical addresses
- Address space defines a "private" addressing concept
 - requires form of address virtualization (will be covered in memory management)
 - We distinguish virtual address vs physical address
- 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`
- Even the kernel has its own address space but typically maps the virt-address → phys-addr but still goes through address translation



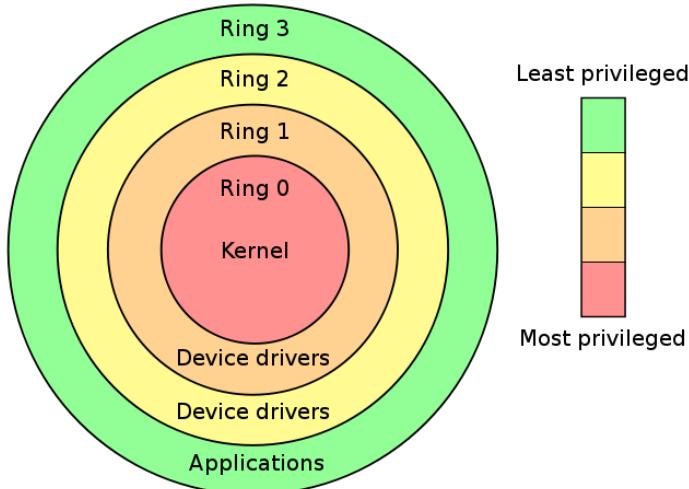
CPU Execution Modes

- Two common modes of CPU to enable protection
 - User mode
 - aka unprivileged or problem mode
 - only capable to running a subset of instructions (almost all) limits (~excludes) user from accessing critical resources
 - Kernel mode
 - aka privileged or supervisor
 - access to all instruction
 - enables the kernel to provide core services in a protected way

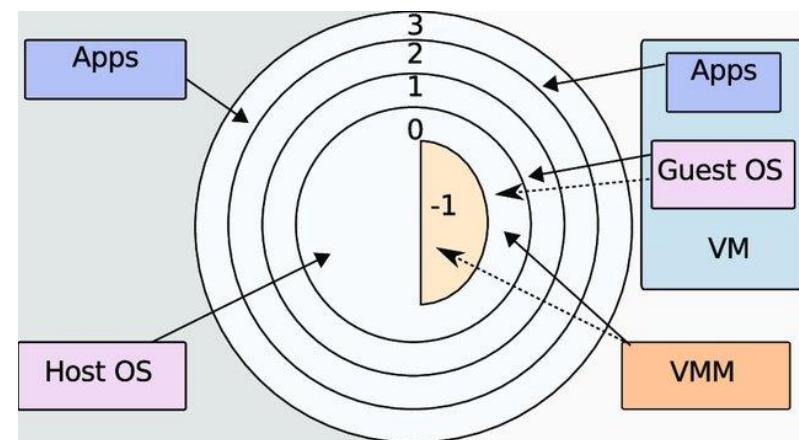
CPU Execution Modes (more general)

- Architectures sometimes provide more extended modes.
- Often referred to as “protection rings”
- Similar concept of privilege limitation on instructions and resource access resulting in traps

x86 protection rings



x86 protection rings with virtualization



CPU Execution Modes Switches

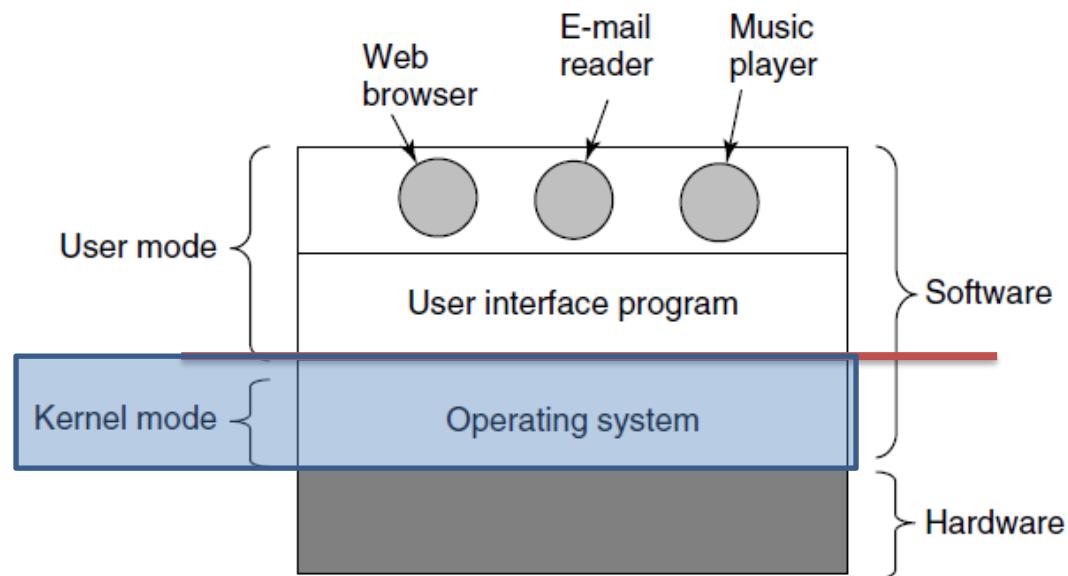
How to switch between the two modes:

UserMode -> KernelMode

- Trap
- Interrupt (also Kernel2Kernel)
- Exception (also Kernel2Kernel)

KernelMode -> UserMode

- rfi (return from interrupt)
(also kernel to kernel)



Exception / Traps / Interrupt

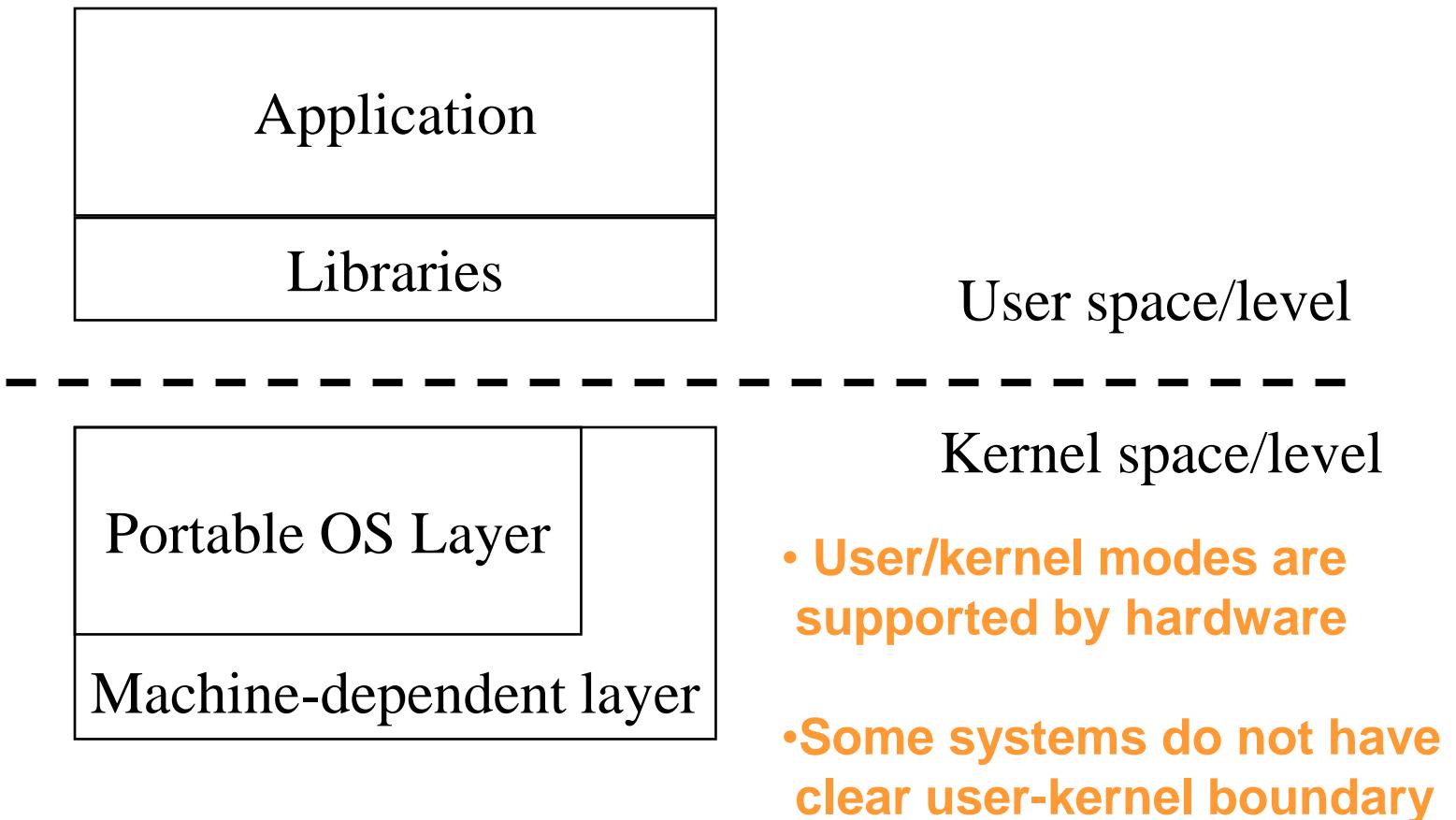
- **Exceptions:**
 - triggered by a "fault condition" of an instruction condition
 - *Synchronous*
- **Traps**
 - special kind of exception → instruction, aka sc [system call], side effect is intended
 - *Synchronous*: triggered by "trap instruction" for syscall
- **Interrupts:**
 - Triggered by a hardware event from a "device" (device needs attention)
 - *Asynchronous*

Understand similarities and differences between these 3 "events"

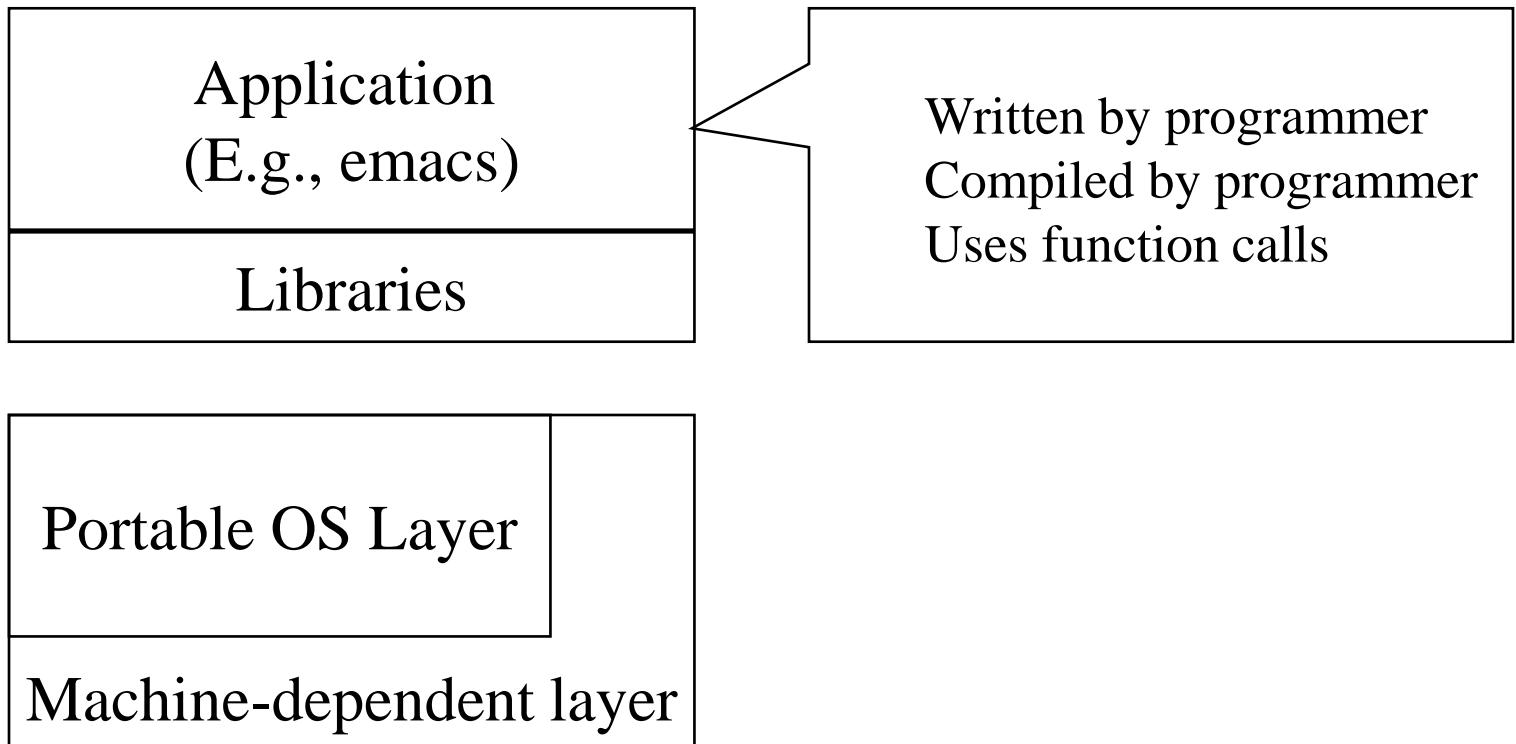
Where do Exceptions/Traps/Interrupts switch to ?

- They all end up in the so called "interrupt handler":
 - (1) Protected Hardware register is initialized in OS bootstrap with the address of kernel entry code (**aka __entry**) , so the hardware knows where to jump to when an Exception or Trap or Interrupt is raised.
 - (2) kernel entry code (__entry) is typically written in assembler because all registers must be accessed and a stack frame must be created to get as quickly as possible into "C" code.
 - (2) on exception / trap / interrupt the CPU changes mode to kernel and starts execution to __entry in the kernel while the storing the failing/interrupted instruction.
 - (3) from there the assembler identifies whether an interrupt, exception, or trap and jumps to their respective handlers.
- __entry is the **ONLY** means to start running code in the operating system kernel

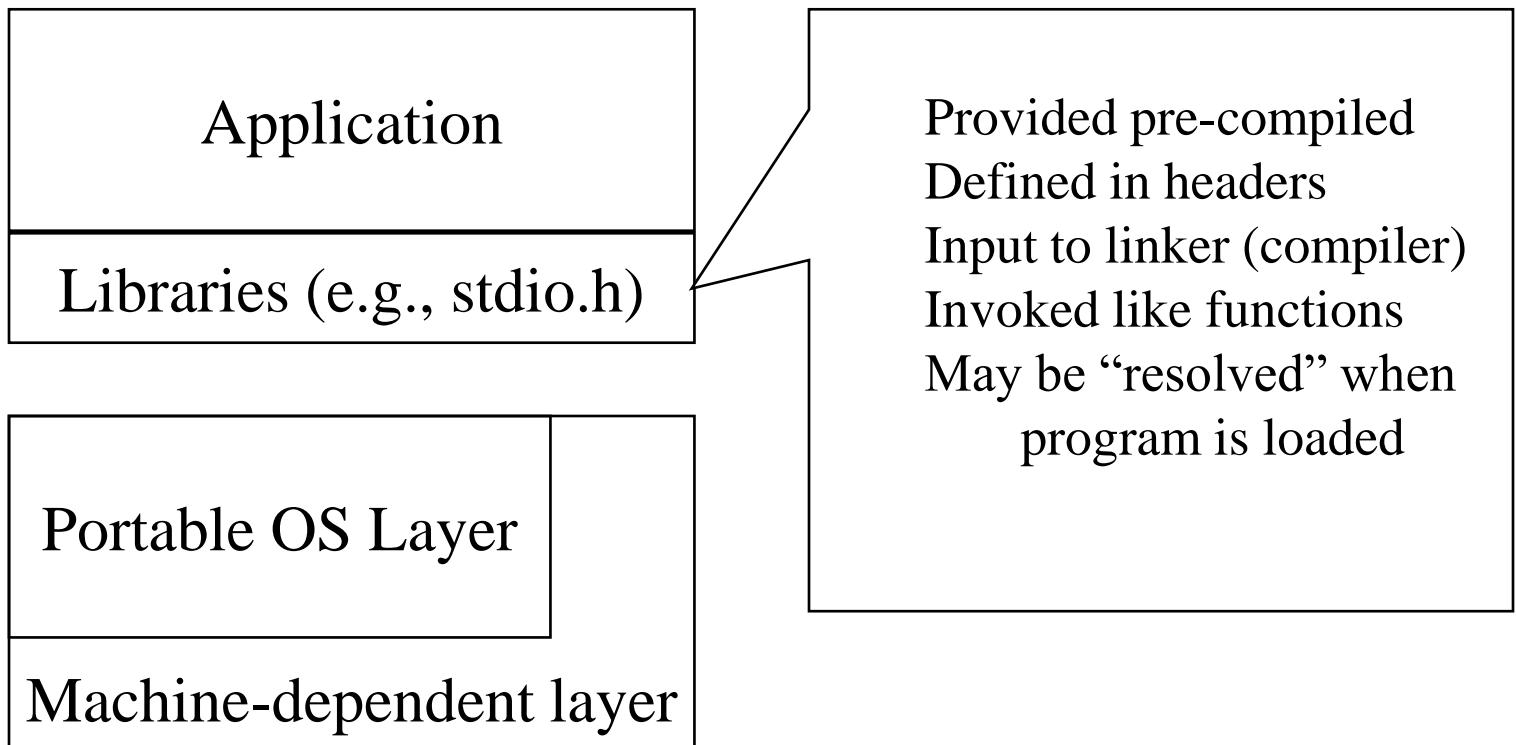
A peek into Unix/Linux



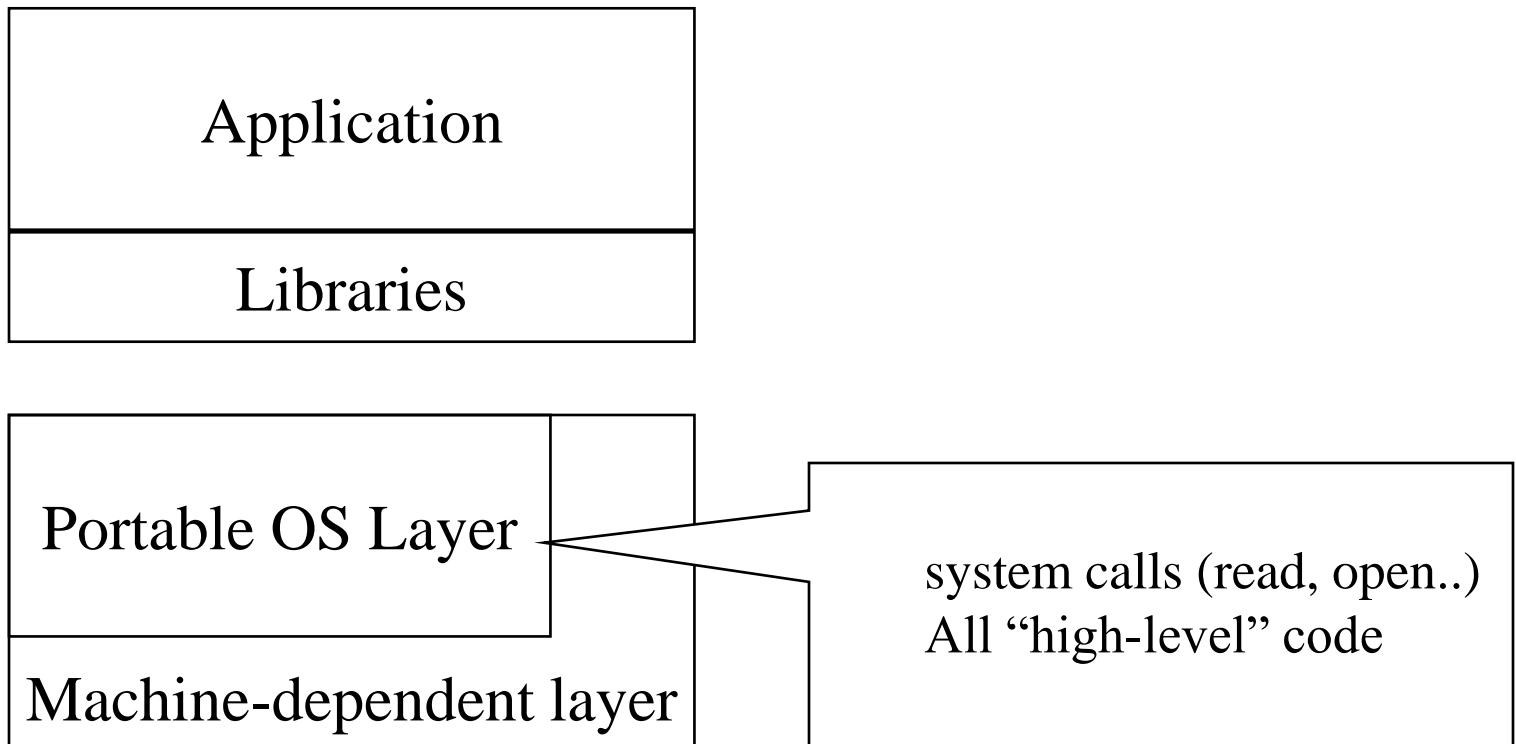
Unix: Application



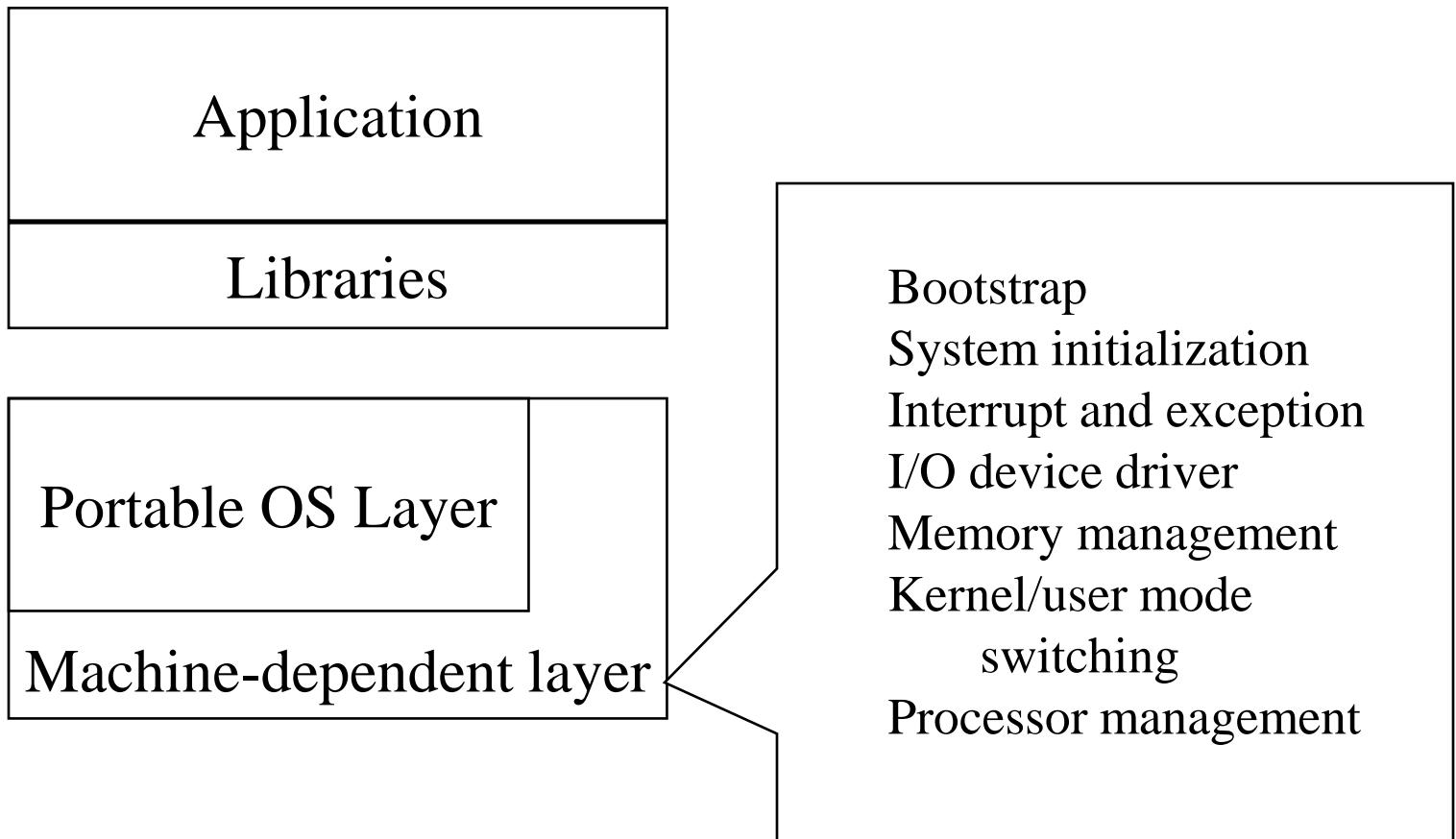
Unix: Libraries



Typical Unix OS Structure



Typical Unix OS Structure



Service Requests from user to kernel (OS)

- Basic means to request services from the operating system kernel is to make **system calls**
(which end up in a “trap / sc” event)
- It's a well architected and “secure” API between kernel and userspace

Some System Calls For Process Management

Process management

Call	Description
pid = fork()	Create a child process identical to the parent
pid = waitpid(pid, &statloc, options)	Wait for a child to terminate
s = execve(name, argv, environp)	Replace a process' core image
exit(status)	Terminate process execution and return status

Signal

Call	Description
kill(pid, signal)	Deliver signal to the process pid
signal(signal, function)	Define which function to call for signal

System Calls (POSIX)

- System calls for process management
- Example of `fork()` used in simplified shell program

```
#define TRUE 1

while(TRUE) {
    type_prompt();
    read_command(command, parameters);
    if (fork() !=0) {
        /* some code*/
        waitpid(-1,&status, 0); }
    else {
        /* some code*/
        execve(command, parameters,0);
    }
}
```

System Calls (POSIX)

- System calls for file/directory management
 - fd = **open**(file,how,...)
 - n = **write**(fd,buffer,nbytes)
 - e = **rmdir**(name)
- Miscellaneous
 - e = **kill**(pid,signal)
 - e = **chmod**(name,mode)

List of important syscalls

Posix	Win32	Description
Process Management		
Fork		Clone current process
exec(ve)	CreateProcess	Replace current process
wait(pid)	WaitForSingleObject	Wait for a child to terminate.
exit	ExitProcess	Terminate process & return status
File Management		
open	CreateFile	Open a file & return descriptor
close	CloseHandle	Close an open file
read	ReadFile	Read from file to buffer
write	WriteFile	Write from buffer to file
lseek	SetFilePointer	Move file pointer
stat	GetFileAttributesEx	Get status info
Directory and File System Management		
mkdir	CreateDirectory	Create new directory
rmdir	RemoveDirectory	Remove <i>empty</i> directory
link	(none)	Create a directory entry
unlink	DeleteFile	Remove a directory entry
mount	(none)	Mount a file system
umount	(none)	Unmount a file system
Miscellaneous		
chdir	SetCurrentDirectory	Change the current working directory
chmod	(none)	Change permissions on a file
kill	(none)	Send a signal to a process
time	GetLocalTime	Elapsed time since 1 jan 1970

System Call == OS kernel service request

- Reminder:

Invoked via non-privileged instruction (trap / sc)

- Treated often like an interrupt, but its “somewhat” different

- Synchronous transfer control from user to kernel
- Side-effect of executing a trap in userspace is that an “exception” is raised and program execution continues at a prescribed instruction in the kernel
see __entry -> syscall_handler

How are syscalls implemented

- First, one has to understand how arguments in any regular function call are passed.
- For this, a **calling code convention** is defined.
- Typically, arguments are passed through registers (sometimes as offsets on the stack)
- Those registers can be modified by the function called, any other registers must be saved and restored by the callee function
 - Volatile register (args,stackptr) and non-volatile registers (callee must save and restore)
- Generally referred to as ABI: **Application Binary Interface**
- Syscalls are simply an extension on this. All compilers need to agree on ABI or code will no cooperate/work.

arch/ABI	arg1	arg2	arg3	arg4	arg5	arg6	arg7
arm/OABI	a1	a2	a3	a4	v1	v2	v3
arm/EABI	r0	r1	r2	r3	r4	r5	r6
arm64	x0	x1	x2	x3	x4	x5	-
blackfin	R0	R1	R2	R3	R4	R5	-
i386	ebx	ecx	edx	esi	edi	ebp	-
ia64	out0	out1	out2	out3	out4	out5	-
mips/o32	a0	a1	a2	a3	-	-	-
mips/n32,64	a0	a1	a2	a3	a4	a5	-
parisc	r26	r25	r24	r23	r22	r21	-
s390	r2	r3	r4	r5	r6	r7	-
s390x	r2	r3	r4	r5	r6	r7	-
sparc/32	o0	o1	o2	o3	o4	o5	-
sparc/64	o0	o1	o2	o3	o4	o5	-
x86_64	rdi	rsi	rdx	r10	r8	r9	-
x32	rdi	rsi	rdx	Ir10	r8	r9	-

arch/ABI	instruction	syscall #	retval	Notes
arm/OABI	swi NR	-	a1	NR is syscall #
arm/EABI	swi 0x0	r7	r0	
arm64	svc #0	x8	x0	
blackfin	excpt 0x0	P0	R0	
i386	int \$0x80	eax	eax	
ia64	break 0x100000	r15	r8	See below
mips	syscall	v0	v0	See below
parisc	ble 0x100(%sr2, %r0)	r20	r28	
s390	svc 0	r1	r2	See below
s390x	svc 0	r1	r2	See below
sparc/32	t 0x10	g1	o0	
sparc/64	t 0x6d	g1	o0	
x86_64	syscall	rax	rax	See below
x32	syscall	rax	rax	See below

Trap/SC instruction

syscall implementation (user side)

/usr/include/asm-generic/unistd.h

unistd.h:extern ssize_t pread64 (int __fd, void * __buf, size_t __nbytes)

```
#define _GNU_SOURCE          /* See feature_test_macros(7) */
#include <unistd.h>
#include <sys/syscall.h>    /* For SYS_xxx definitions */

long syscall(long number, ...);
```



Definition agreed upon by libc and kernel
→ ABI is known. Compiler assembles args

```
0000000000400596 <main>:
400596: 55                      push  %rbp
400597: 48 89 e5                mov    %rsp,%rbp
40059a: 48 83 ec 70              sub    $0x70,%rsp
40059e: 64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
4005a5: 00 00
4005a7: 48 89 45 f8              mov    %rax,-0x8(%rbp)
4005ab: 31 c0                   xor    %eax,%eax
4005ad: 48 8d 45 a0              lea    -0x60(%rbp),%rax
4005b1: ba 50 00 00 00            mov    $0x50,%edx
4005b6: 48 89 c6                mov    %rax,%rsi
4005b9: bf 00 00 00 00            mov    $0x0,%edi
4005be: e8 ad fe ff ff          callq 400470 <read@plt>
4005c3: 89 45 9c                mov    %eax,-0x64(%rbp)
4005c6: 8b 45 9c                mov    -0x64(%rbp),%eax
4005c9: 48 8b 4d f8              mov    -0x8(%rbp),%rcx
4005cd: 64 48 33 0c 25 28 00    xor    %fs:0x28,%rcx
4005d4: 00 00
4005d6: 74 05                   je    4005dd <main+0x47>
4005d8: e8 83 fe ff ff          callq 400460 <__stack_chk_fail@plt>
4005dd: c9                      leaveq 
4005de: c3                      retq 
4005df: 90                      nop
```

ABI: Application Binary Interface

```
/* fs/read_write.c */
#define __NR3264_lseek 62
SC_3264(__NR3264_lseek, sys_llseek, sys_lseek)
#define __NR_read 63
SYSCALL(__NR_read, sys_read)
#define __NR_write 64
SYSCALL(__NR_write, sys_write)
#define __NR_ready 65
SC_COMP(__NR_ready, sys_ready, compat_sys_ready)
#define __NR_writev 66
SC_COMP(__NR_writev, sys_writev, compat_sys_writev)
#define __NR_pread64 67
SC_COMP(__NR_pread64, sys_pread64, compat_sys_pread64)
#define __NR_pwrite64 68
SC_COMP(__NR_pwrite64, sys_pwrite64, compat_sys_pwrite64)
#define __NR_preadv 69
SC_COMP(__NR_preadv, sys_preadv, compat_sys_preadv)
#define __NR_pwritev 70
SC_COMP(__NR_pwritev, sys_pwritev, compat_sys_pwritev)

/* fs/sendfile.c */
#define __NR3264_sendfile 71
SYSCALL(__NR3264_sendfile, sys_sendfile64)
```

#define __SYSCALL(number) syscall(number...)

syscall is implemented as assembler largely taking the arguments already in the right registers and TRAP-ing into the kernel.

syscall implementation (kernel side)

- Kernel defines a table (using the compiler help)

```
int syscall_table [ __NR_SYSCALL_MAX ] = {  
    :  
    [ __NR_READ] = sys_read,  
    [ __NR_WRITE] = sys_write,  
    :  
}
```

The compiler does the magic
and associates the syscall
number with the kernel
internal function

- On system trap, architecture automatically and immediately enters kernel mode and runs a small piece of assembler code that is stored at a machine register address set by the OS at boot time.
- Said trap assembler code (aka interrupt handler) does the following:
 - Checks the syscall number in well known register (see ABI) to be in range
 - Assembler equivalent:
 - Change stack to kernel (more on this in a bit)
 - All arguments are already in right place thanks to the ABI and the compiler's help
 - "Call/jmp" to syscall_table[registers.syscall_number]; // see ABI definition
 - After return from ^^^, switch back from kernel stack to user stack and RFI (return from kernel mode).

Putting it all together (based on ARM/EABI)

Your app.c

```
char buf[128];
int fd;
myfunct()
{
    read(fd,buf,64);
}
```

Regular function call

libc.so

```
#define syscall(sysnum) \
    asm("ldr r7,%d; swi 0x0", sysnum)

int read(int fd, char* buf, in nbytes)
{
    return syscall(__NR_READ);
}
```

```
_entry: // all assembler code,
// save require non-volatile
// figure out what type (intr,trap,exception

    call syscall_table[r7]
    return from interrupt
```

All argument registers must stay intact:

r0,r1,r2,...

Function expects args in r0,r1,r2,...

assembler based on compiler

```
ldr r0, =fd
ldr r1, =buf
ldr r2, =0x40
bl read
bx lr
```

r0 = fd
r1 = buf
r2 = 64

r7 = __NR_READ

User
kernel

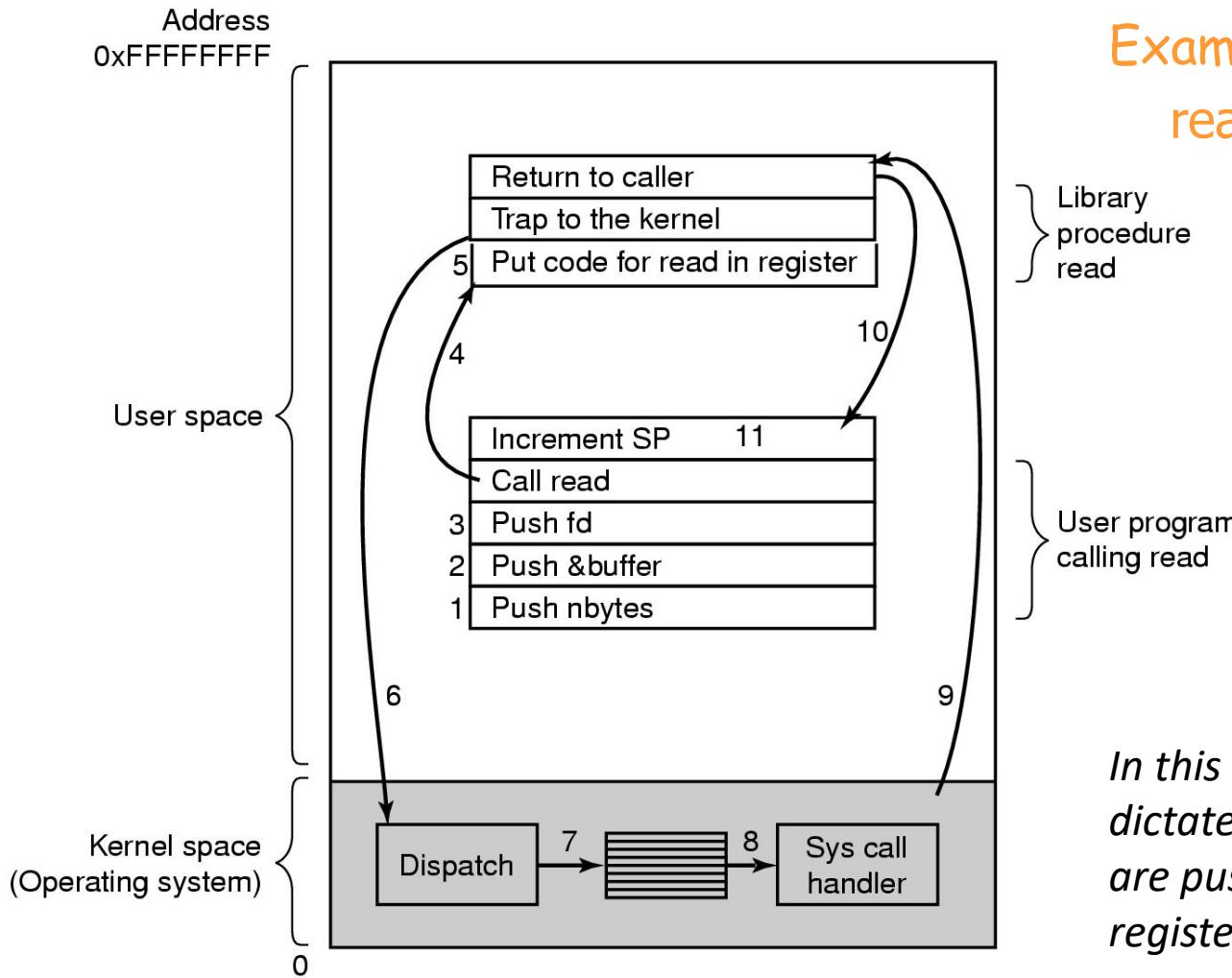
```
ldr r7, =0x3F
swi 0x0
bx lr
```

```
shift r7,2
add r20,r7,=syscall_table
blr r20
bx lr
```

Regular function call

```
int sys_read(int fd, char* buf, in nbytes)
{
    // some sophisticated kernel code
}
```

System Call with args passed over the stack (different ABI)



Example:

read (fd, buffer,nbytes)

In this example the ABI dictates that function arguments are pushed on the stack not through registers

(older architectures)

System Calls (Windows Win32 API)

- Process Management
 - CreateProcess - new process (combined work of fork and execve in UNIX)
 - In Windows - no process hierarchy, event concept implemented
 - WaitForSingleObject - wait for an event (can wait for process to exit)
- File Management
 - CreateFile, CloseHandle,.CreateDirectory, ...
 - Windows does not have signals, links to files, ..., but has a large number of system calls for managing GUI

Other implicit/explicit OS Services Examples

- Services that can be provided at user level (because they only read unprotected data)
 - Read time of the day
- Services that need to be provided at kernel level
 - System calls: file open, close, read and write
 - Control the CPU so that users won't stuck by running

```
while ( 1 ) ;
```
 - Protection:
 - Keep user programs from crashing OS
 - Keep user programs from crashing each other

Is Any OS Complete? (Criteria to Evaluate OS)

Portability

Security

Fairness

Robustness

Efficiency

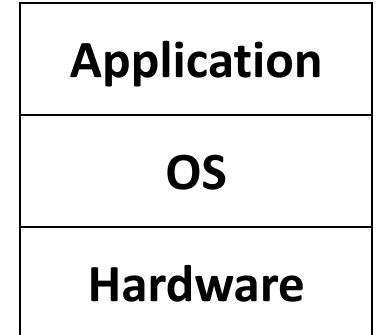
Interfaces

When you look at these criteria you should see that they can't all be satisfied at the same time.

Recap: What is an OS ?

- **Code that:**

- Sits between programs & hardware
- Sits between different programs
- Sits between different users



- **Job of OS:**

- Manage hardware resources
 - Allocation, protection, reclamation, virtualization
- Provide services to app. How? → system call
 - Abstraction, simplification, standardization

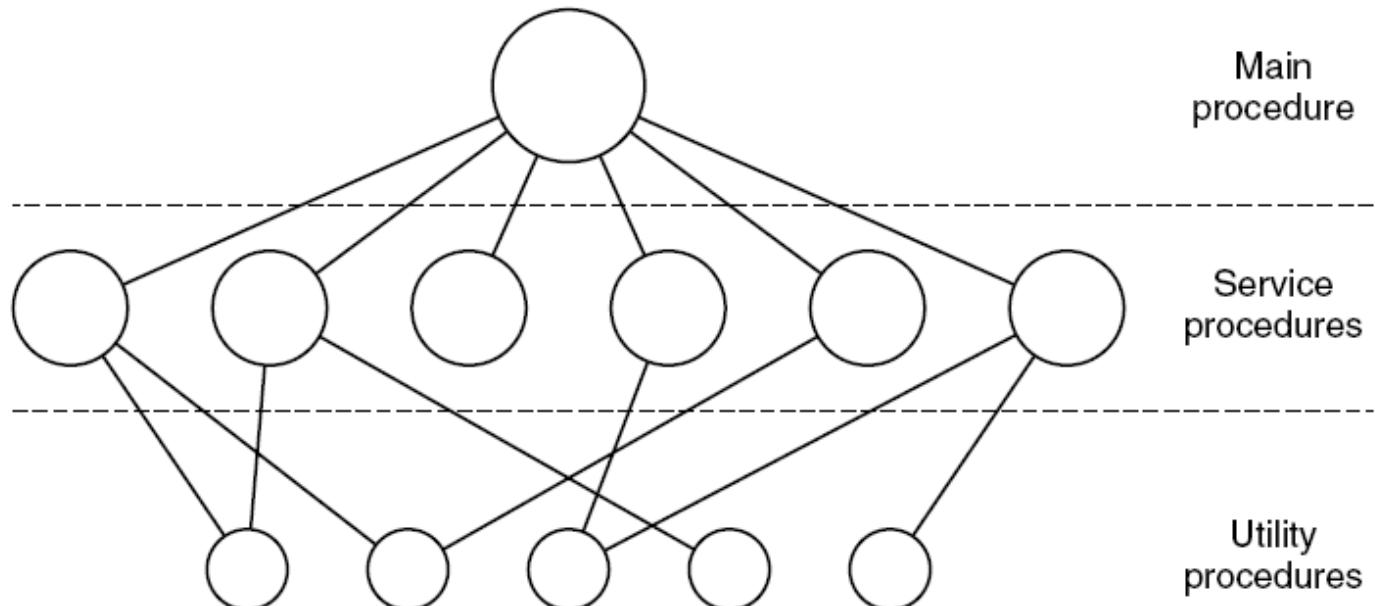
Operating Systems Structure (Chapter 1)

Monolithic systems - basic structure

1. A main program that invokes the requested service procedure.
2. A set of service procedures that carry out the system calls.
3. A set of utility procedures that help the service procedures.

Monolithic Systems

By far the most common OS organization
A simple structuring model for a monolithic system.



Layered Systems

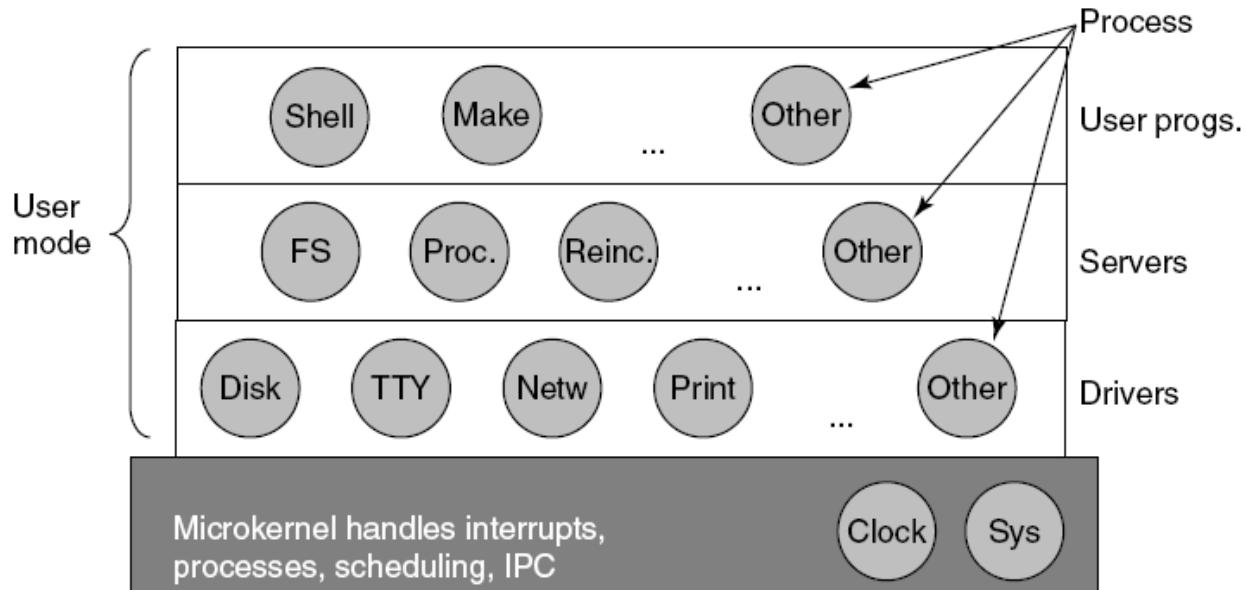
- Layer-n services are comprised of services provided by Layer-(n-1)..
- Structure of the **THE** operating system (Dijkstra 1968)

Layer	Function
5	The operator
4	User programs
3	Input/output management
2	Operator-process communication
1	Memory and drum management
0	Processor allocation and multiprogramming

- **THE** used this approach as a design AID
- Multics Operating System relied on Hardware Protection to enforce layering

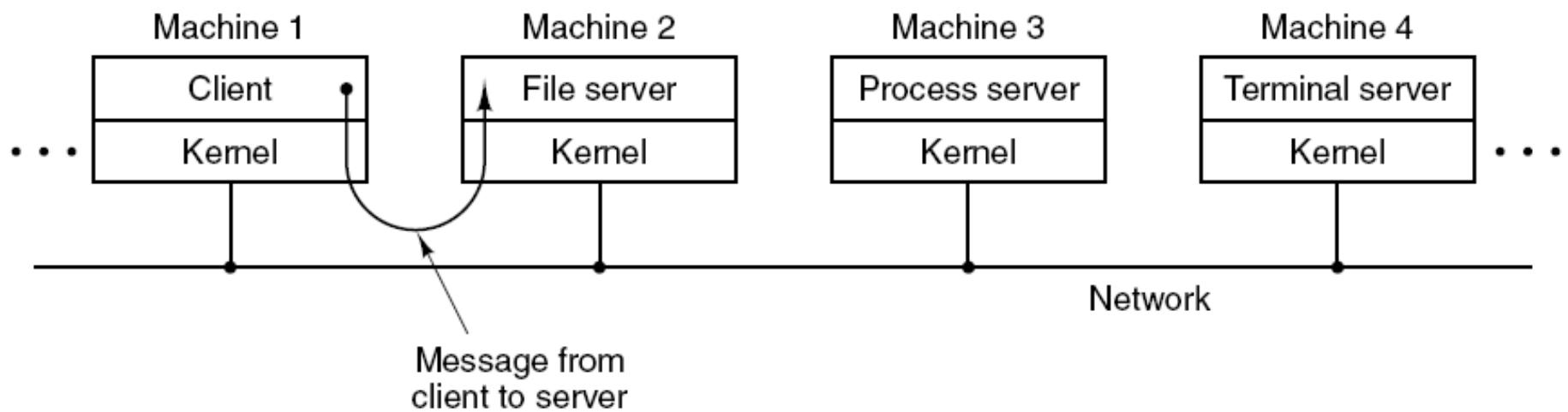
Microkernels

- Microkernels move the layering boundaries between kernel and userspace
- Move only most rudimentary services to kernel
- Move other services to Userspace
- Higher Overhead, but more flexibility, higher robustness
when one OS non-micro kernel component fails it is being restarted.
 - Minix: is only 3200 lines of C and 800 lines assembler)
 - L4, K42,



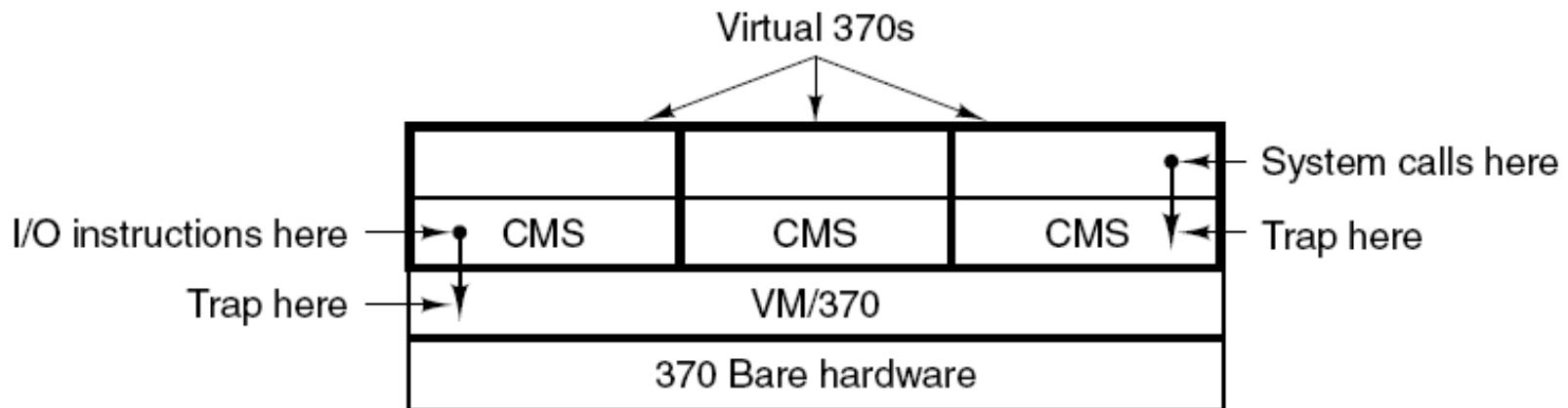
Client-Server Model

- Assumes generic network model (network, bus)
- Communication via message passing



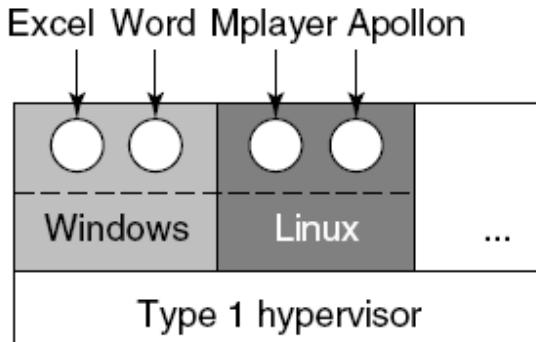
Virtual Machines (1)

- VM/370: Timesharing system should be comprised of:
 - Multiprogramming
 - extended machine with more interface than bare HW
 - Completely separate these two functions
- Provides ability to “self-virtualize”
- Beginning of “modern day” virtualization technology (1970)

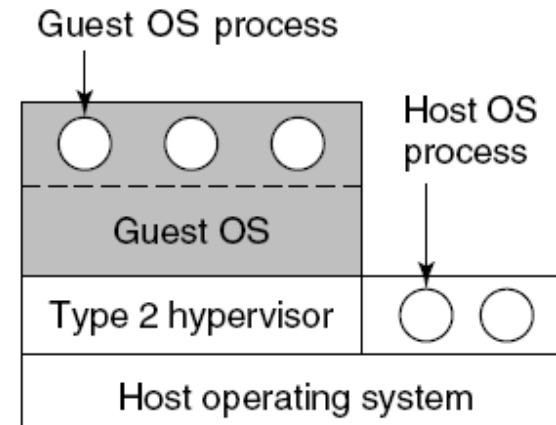


Virtual Machines (2)

- A type 1 hypervisor
(like virtual machine monitor)
 - Privileged instructions are trapped and “emulated”
- A type 2 hypervisor (runs on top of a host OS)
 - Unmodified (trapped)
 - Modified (paravirtualization)



(a)



(b)

Other areas of virtual machines usage

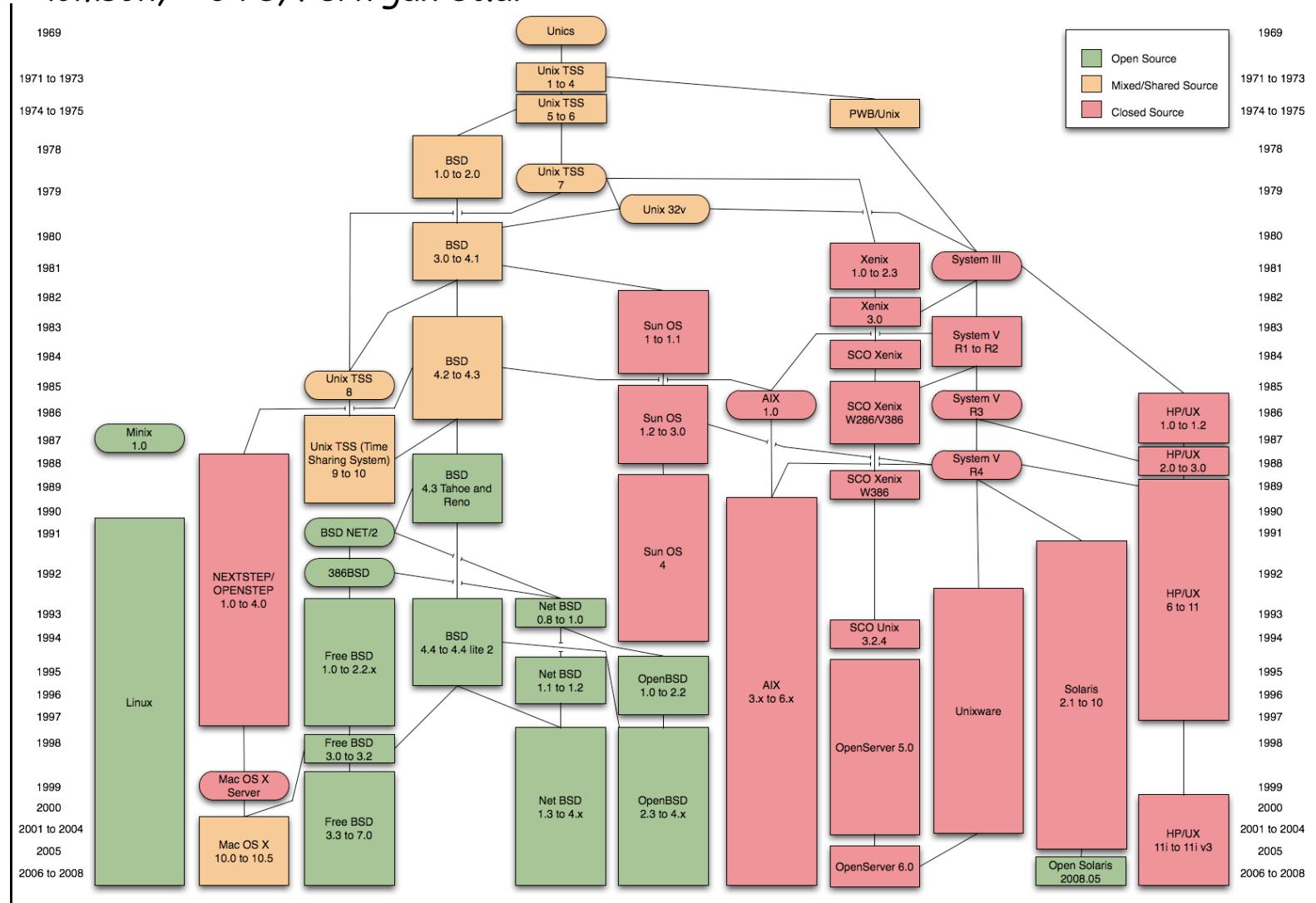
- Java virtual machines
- Dynamic scripting languages (e.g. Python)
- Typically define a instruction set that is "interpreted" by the associate virtual machine
 - JVM, PVM
 - Modern system then JIT (Just In Time) compile the VM instructions into native code.

JIT HAPPENS

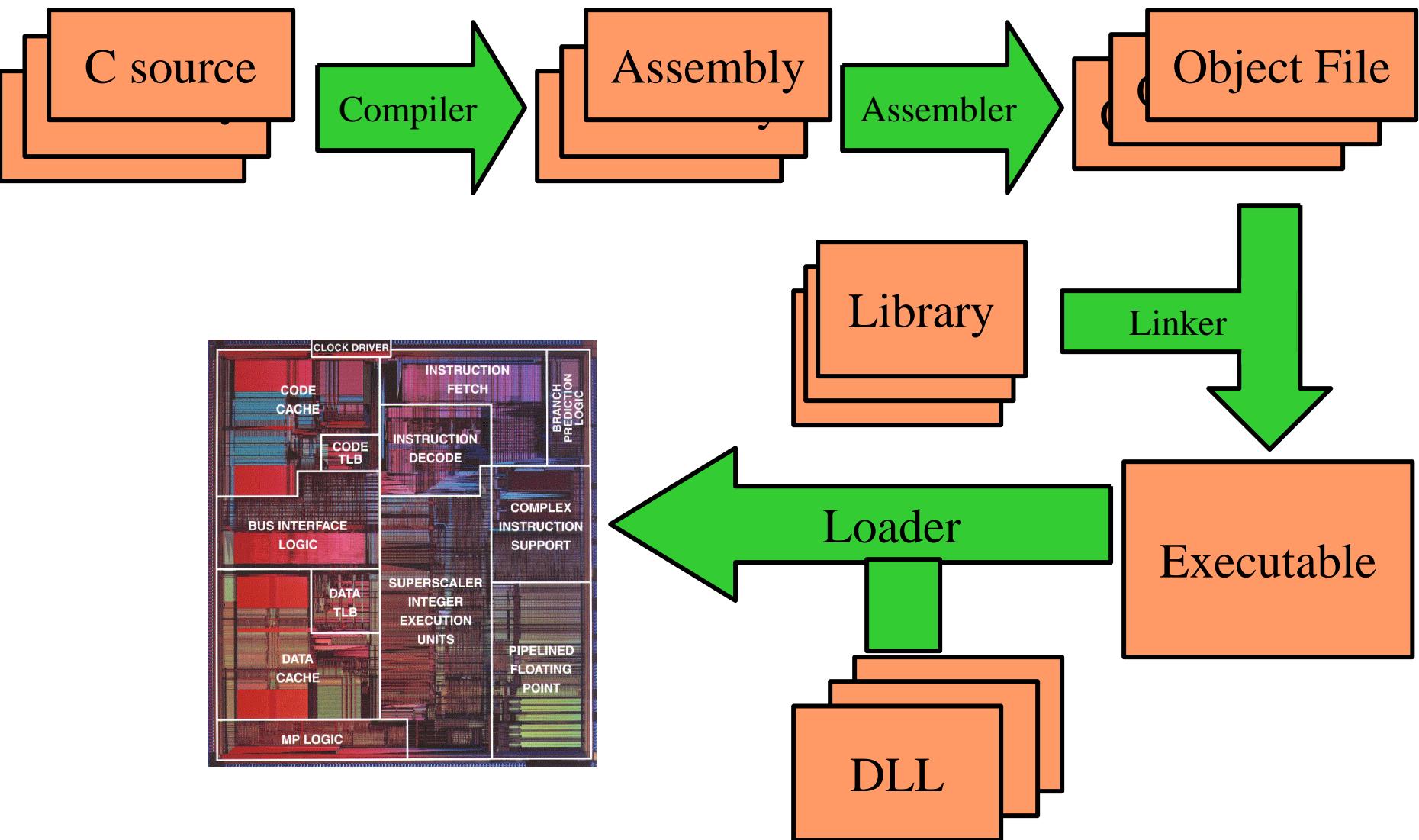
History of the UNIX Operating System

(source: wikipedia)

Bell Labs: Thomson, Richie, Kernigan et.al

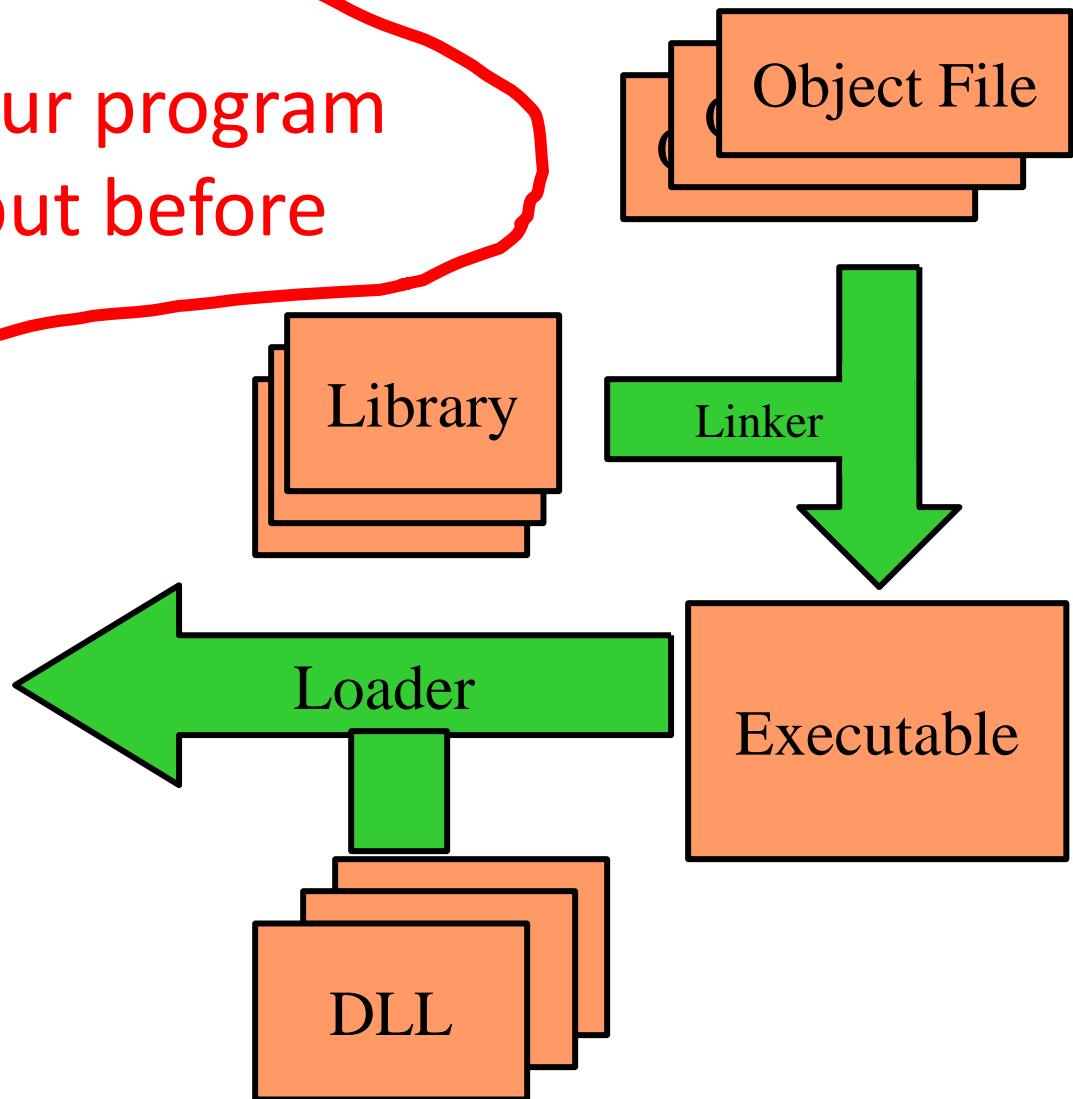


Source Code to Execution



Source Code to Execution

What happens to your program after it is compiled but before it can be executed?



The OS Expectation

- The OS expects executable files to have a specific format
 - Header info
 - Code locations and size
 - Data locations and size
 - Code & data
 - Symbol Table
 - List of names of **things** defined in your program and where they are defined
 - List of names of **things** defined elsewhere that are used by your program, and where they are used.

Example of Things

```
#include <stdio.h>
extern int errno;

int main () {
    printf ("hello,
world\n")
    <check errno for
errors>
}
```

- Symbol defined in your program and used elsewhere
 - main
- Symbol defined elsewhere and used by your program
 - printf
 - errno

Two Steps Operation: Parts of OS

Linking

- Stitches independently created object files into a single executable file (i.e., a.out)
- Resolves cross-file references to labels
- Listing symbols needing to be resolved by loader

Loading

- copying a program image from hard disk to the main memory in order to put the program in a ready-to-run state
- Maps addresses within file to memory addresses
- Resolves names of dynamic library items
- schedule program as a new process

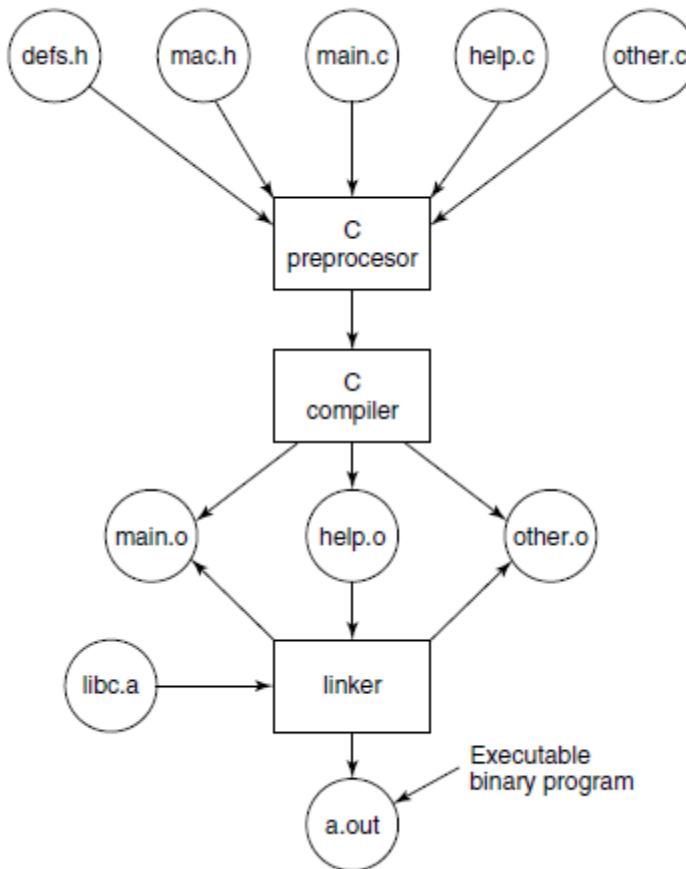
Libraries (I)

- Programmers are expensive.
- Applications are more sophisticated.
 - Pop-down menus, streaming video, etc
- Application programmers rely more on library code to make high quality apps while reducing development time.
 - This means that most of the executable is library code

Libraries (II)

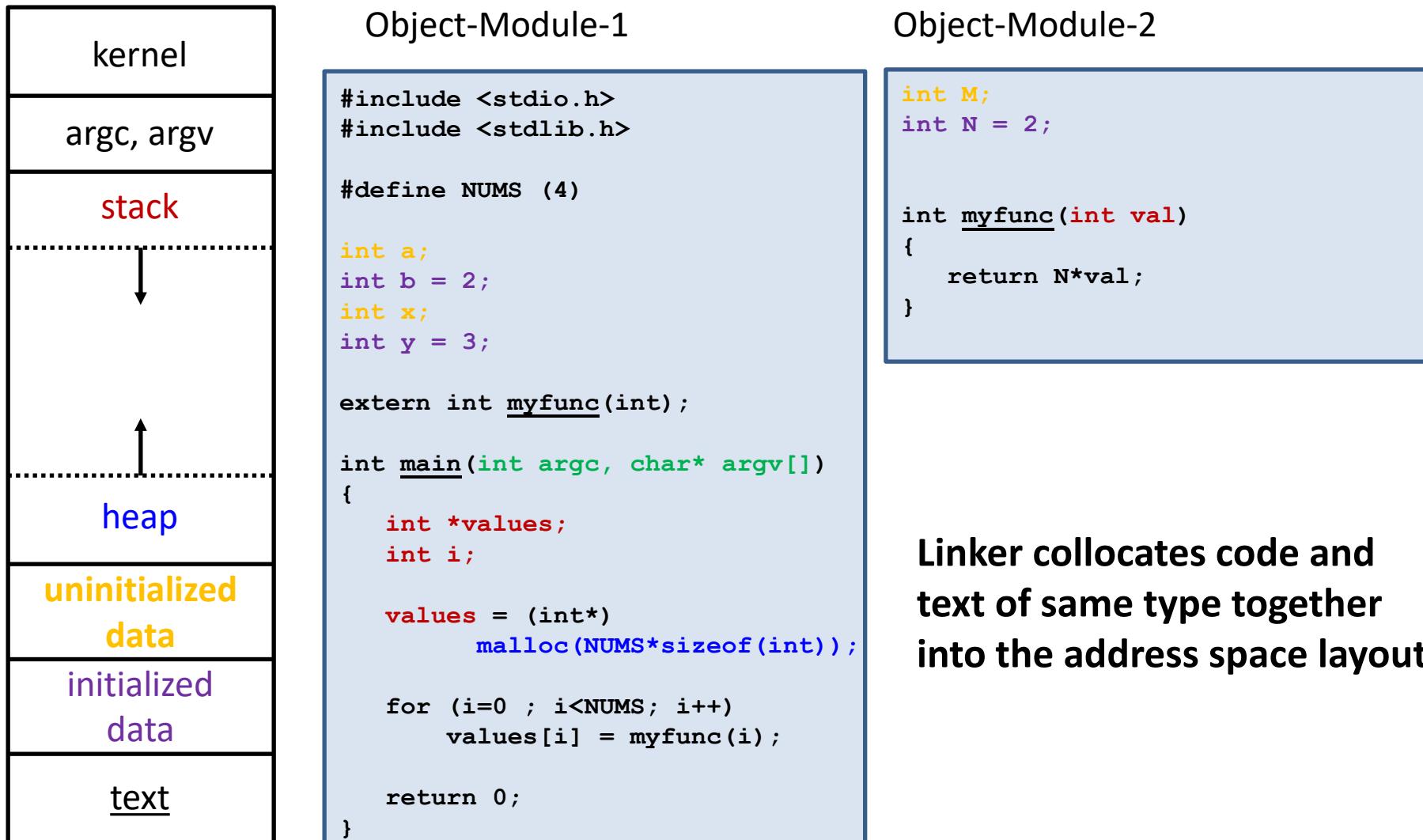
- A collection of subprograms
- Libraries are distinguished from executables in that they are not independent programs
- Libraries are "helper" code that provides services to some other programs
- Main advantages: reusability and modularity

Large Programming Projects



The process of compiling C and header files to make an executable.

Linker: A view of program / object module layout



Object Module Relocation

- modifies the object program so that it can be loaded at an address different from the location originally specified
- The compiler and assembler (mistakenly) treat each module as if it will be loaded at location zero

(e.g. `jump 120`

is used to indicate a jump to location 120 of the current module)

Object Module Relocation

- To convert this **relative address** to an **absolute address**, the linker adds the **base address** of the module to the relative address.
- The base address is the address at which this module will be loaded.

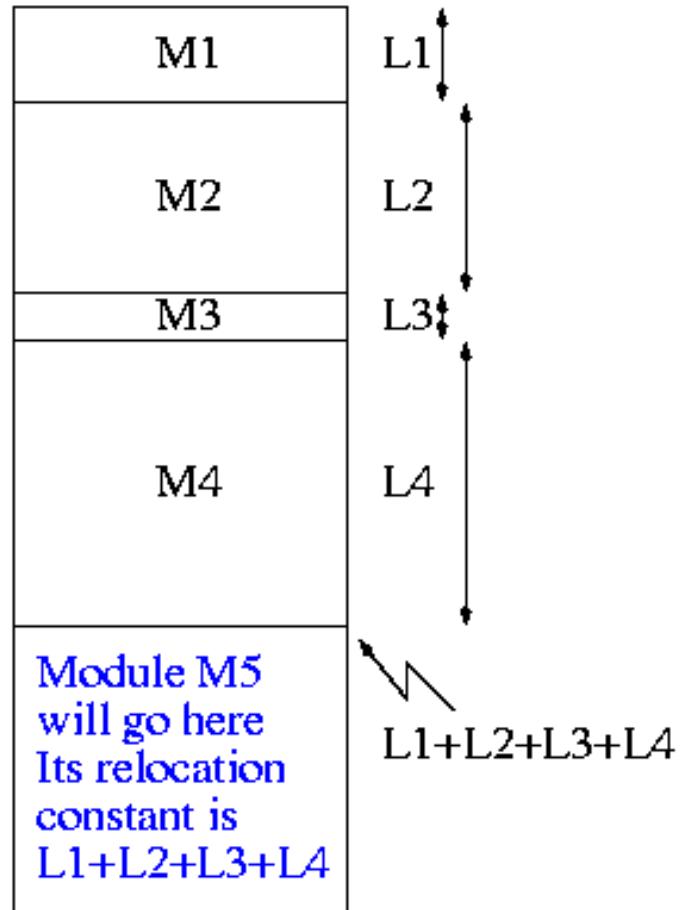
Example: Module A is to be loaded starting at location 2300 and contains the instruction
jump 120

The linker changes this instruction to
jump 2420

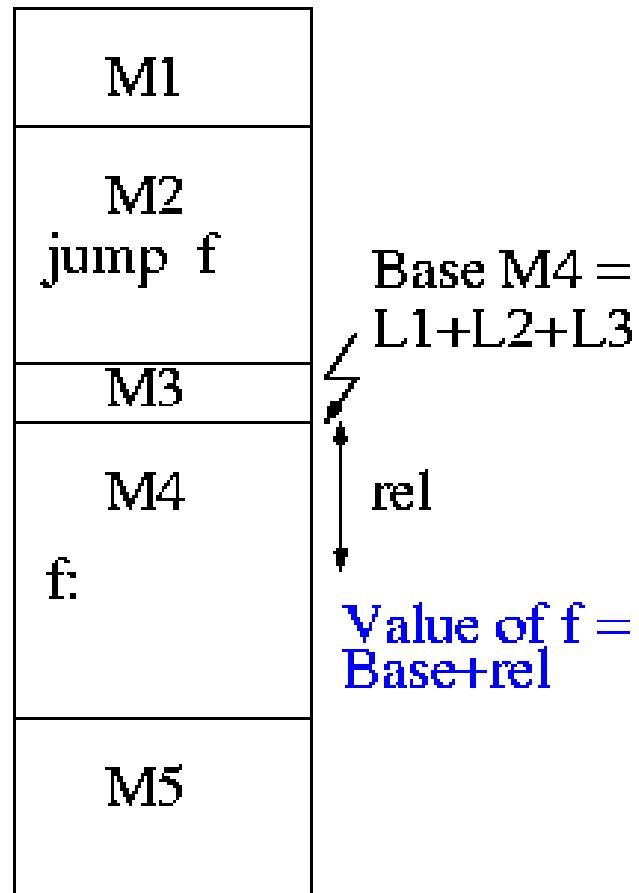
Object Module Relocation

- How does the linker know that Module A is to be loaded starting at location 2300?
 - It processes the modules one at a time. The first module is to be loaded at location zero. So relocating the first module is trivial (adding zero). We say that the relocation constant is zero.
 - After processing the first module, the linker knows its length (say that length is L_1).
 - Hence the next module is to be loaded starting at L_1 , i.e., the relocation constant is L_1 .
 - In general, the linker keeps the sum of the lengths of all the modules it has already processed; this sum is the relocation constant for the next module.

Relocation



Relocation



LAB assignment #1

LAB #1: Write a Linker

- Link “==merge” together multiple parts of a program
- What problem is solved?
 - External references need to be resolved
 - Module relative addressing needs to be fixed

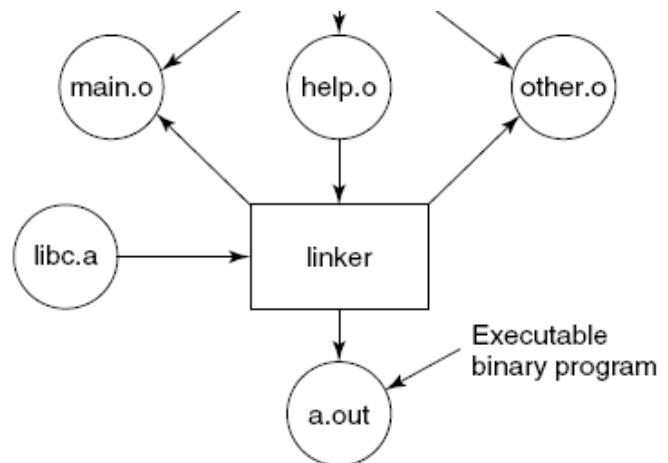
```
Cygwin
289:~/NYU/lab1/example>cat print.c
#include <stdio.h>

void print_hello()
{
    printf("Hello world\n");
}

Cygwin
291:~/NYU/lab1/example>cat main.c
#include <stdio.h>

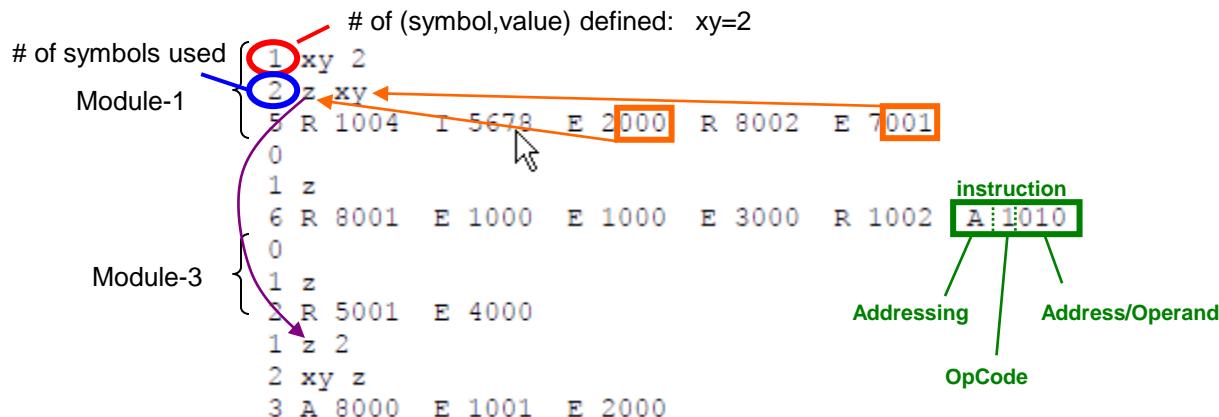
extern void print_hello();

int main(int argc, char **argv)
{
    print_hello();
}
```



LAB #1: Write a Linker

- Simplified module specification
 - List of symbols defined and their value by module
 - List of symbols used in module (including external)
 - List of “instructions”



Addressing

I: Immediate
R: Relative
A: Absolute
E: External

Lab #1: Write a Linker

input

```
1 xy 2
2 z xy
5 R 1004 I 5678 E 2000 R 8002 E 7001
0
1 z
6 R 8001 E 1000 E 1000 E 3000 R 1002 A 1010
0
1 z
2 R 5001 E 4000
1 z 2
2 xy z
3 A 8000 E 1001 E 2000
```

Fancy Output (not req)

```
Symbol Table
xy=2
z=15
Memory Map
+0
0: R 1004 1004+0 = 1004
1: I 5678 5678
2: xy: E 2000 ->z 2015
3: R 8002 8002+0 = 8002
4: E 7001 ->xy 7002
+5
0: R 8001 8001+5 = 8006
1: E 1000 ->z 1015
2: E 1000 ->z 1015
3: E 3000 ->z 3015
4: R 1002 1002+5 = 1007
5: A 1010 1010
+11
0: R 5001 5001+11= 5012
1: E 4000 ->z 4015
+13
0: A 8000 8000
1: E 1001 ->z 1015
2 z: E 2000 ->xy 2002
```

Required output

```
Symbol Table
xy=2
z=15
Memory Map
000: 1004
001: 5678
002: 2015
003: 8002
004: 7002
005: 8006
006: 1015
007: 1015
008: 3015
009: 1007
010: 1010
011: 5012
012: 4015
013: 8000
014: 1015
015: 2002
```



CSCI-GA.2250-001

Operating Systems

Processes and Threads

Hubertus Franke
frankeh@cims.nyu.edu

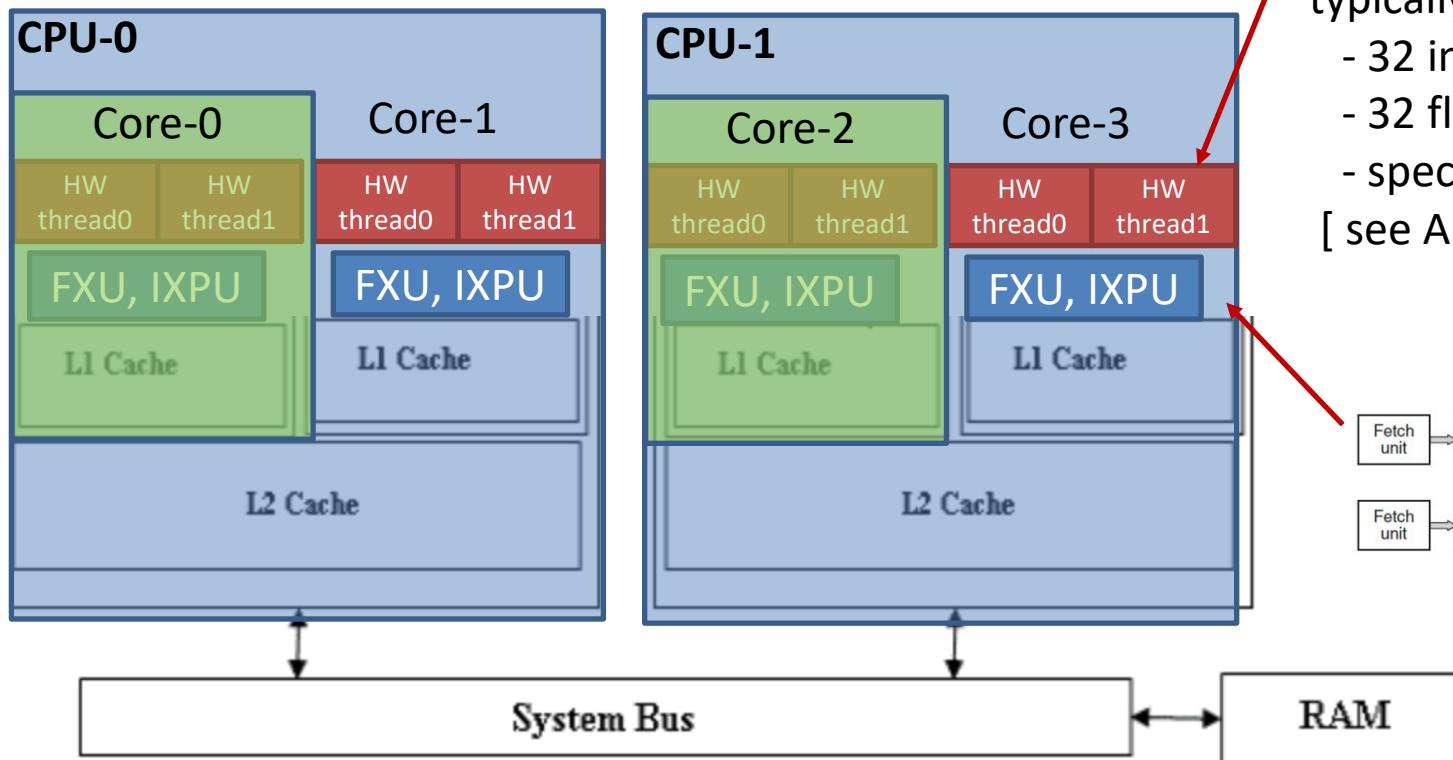


Details of Lecture

- Process Model
- Process Creation (fork , exec)
- Signals
- Process State / Transition Models
- Multi-programming
- Threads

What is exactly a “processor” from an OS perspective

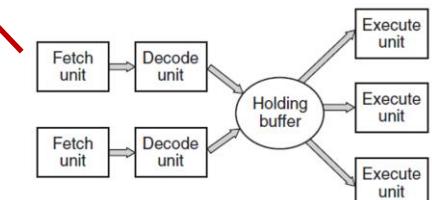
OS can schedule an independent unit of execution (process or thread) onto each Hardware-Thread (HW-Thread)



Each HW-thread exposes register set typically,

- 32 integer regs
- 32 floating point regs
- special regs

[see ABI]



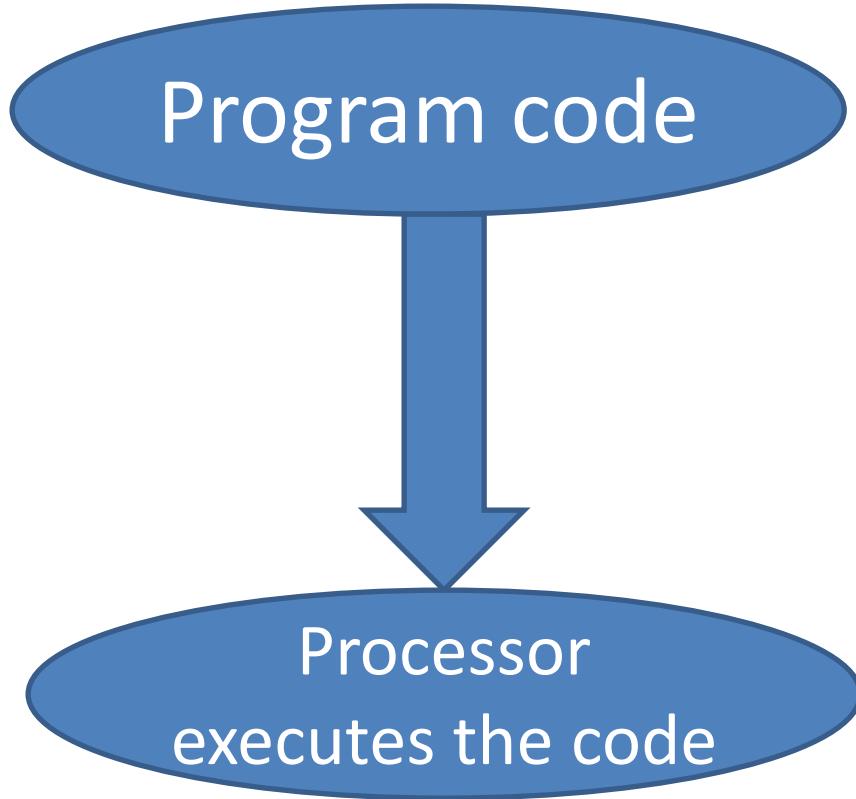
(b)

OS Management of Application Execution

- Resources are made available to multiple applications
- Instruction and registers are not virtualized, applications run natively on the hardware.
- Hence, a “processor” or “core” or “hw-thread” can only run one unit of execution (process/thread) at a time.
- The processor is switched among multiple units of execution, so all will appear to be progressing (albeit potentially at reduced speed)

this “switch” is called a “context switch” and is initiated by the operating system, more on this later

- The processor and I/O devices can be used efficiently
 - When application performs I/O, the processor can be used for a different application.

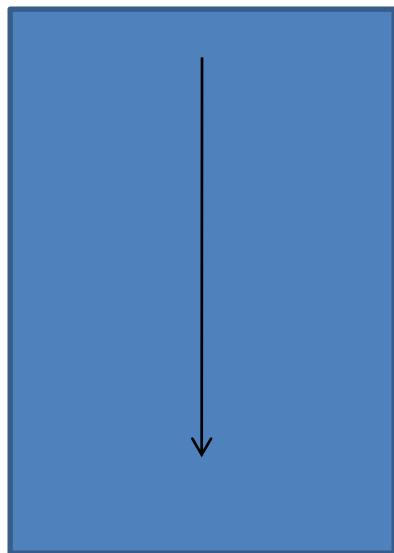


Resides on storage
(just a sequence of bytes
[code , data])

When the processor begins to execute the program code, we refer to this executing entity as a ***process***

What Is a Process?

An abstraction of a running program



Program Counter
(points at current instruction)

The Process Model

- A process has a **program, input, output, and state (data)**.
- A process is an instance of an executing program and includes
 - Variables (memory)
 - Code
 - Program counter (really hardware resource)
 - Registers (- " -)
 - ...

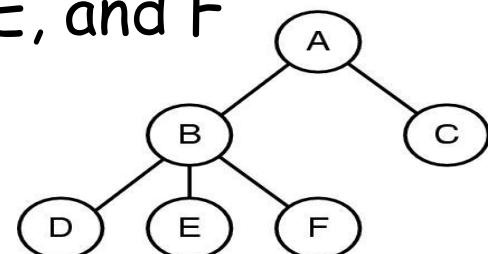
If a program is running twice, does it count as two processes? or one?

Process: a running program

- A process includes
 - **Process State** (state, registers save areas)
 - Open files, thread(s) state, resources held
 - **Address space** (view of its private memory)

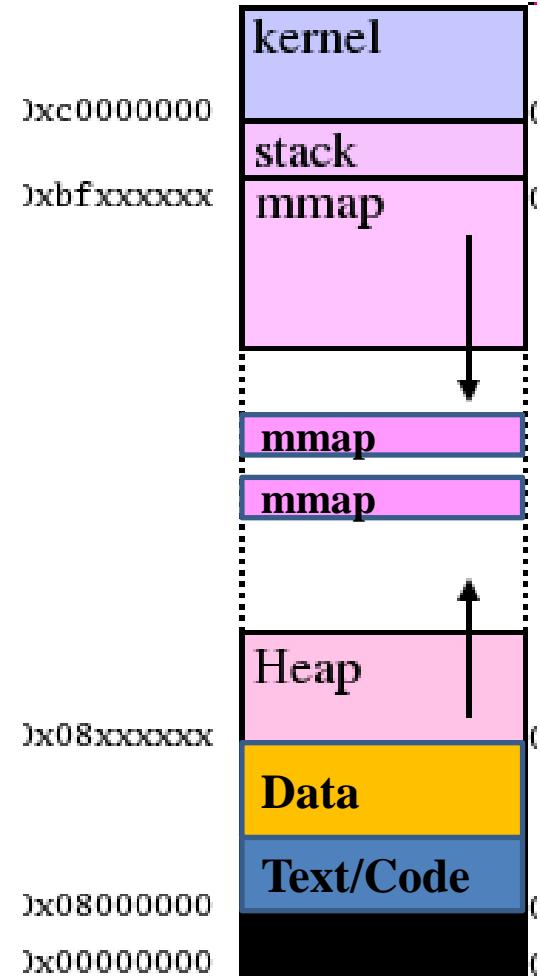
All state is accessible as an entry in the process (entry) table

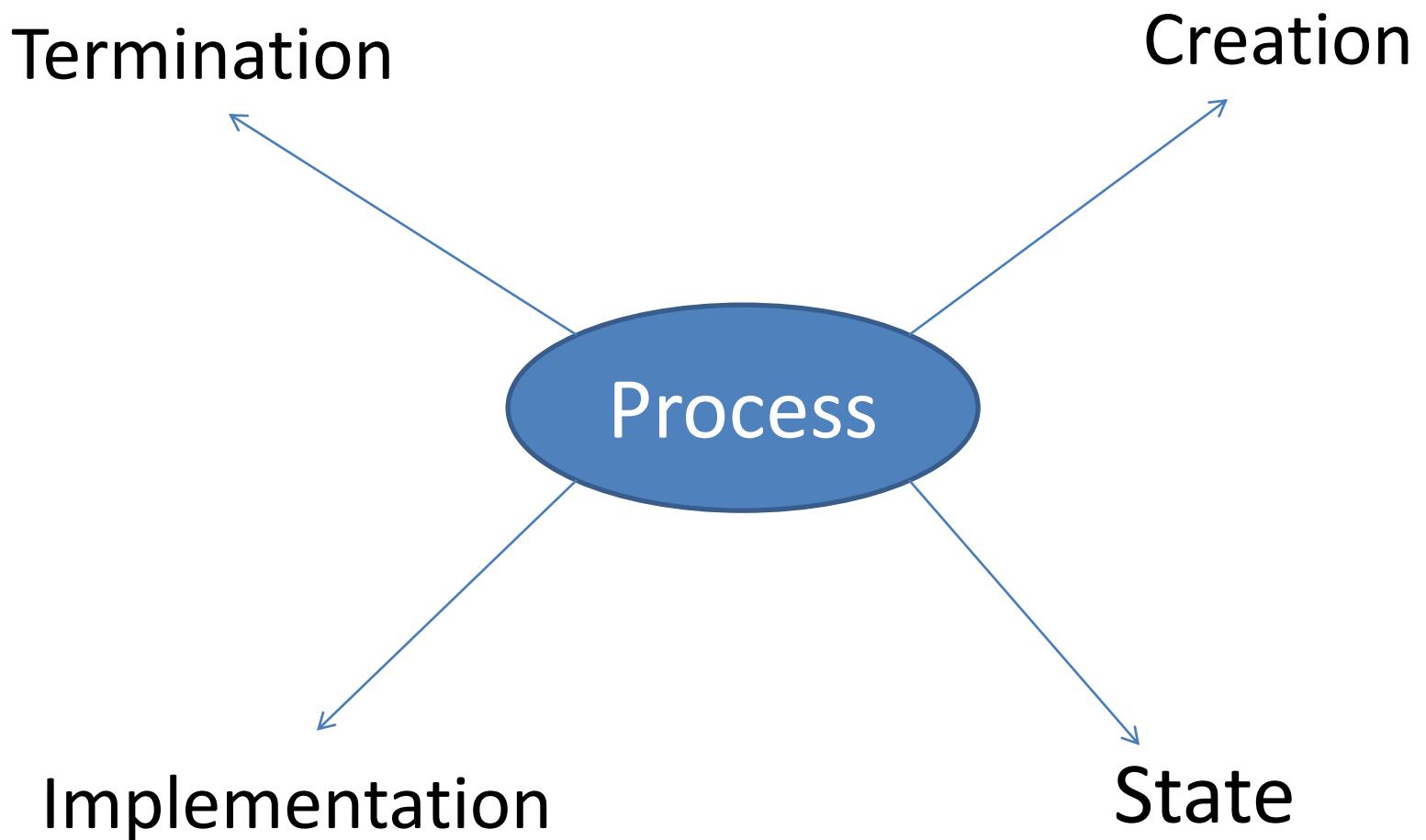
- A process tree
 - A created two child processes, B and C
 - B created three child processes, D, E, and F



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
(will get to it during memory management)





Process Creation

- System initialization
 - At boot time
 - Foreground
 - Background (daemons)
- Execution of a process creation system call by a running process
- A user request
- A batch job
- Created by OS to provide a service
- Interactive login

Process Termination

- Normal exit (voluntary)
- Error exit (voluntary)
- Fatal error (involuntary)
- Killed by another process (involuntary)

Process Termination: More Scenarios

Normal completion	The process executes an OS service call to indicate that it has completed running.
Time limit exceeded	The process has run longer than the specified total time limit. There are a number of possibilities for the type of time that is measured. These include total elapsed time ("wall clock time"), amount of time spent executing, and, in the case of an interactive process, the amount of time since the user last provided any input.
Memory unavailable	The process requires more memory than the system can provide.
Bounds violation	The process tries to access a memory location that it is not allowed to access.
Protection error	The process attempts to use a resource such as a file that it is not allowed to use, or it tries to use it in an improper fashion, such as writing to a read-only file.
Arithmetic error	The process tries a prohibited computation, such as division by zero, or tries to store numbers larger than the hardware can accommodate.
Time overrun	The process has waited longer than a specified maximum for a certain event to occur.
I/O failure	An error occurs during input or output, such as inability to find a file, failure to read or write after a specified maximum number of tries (when, for example, a defective area is encountered on a tape), or invalid operation (such as reading from the line printer).
Invalid instruction	The process attempts to execute a nonexistent instruction (often a result of branching into a data area and attempting to execute the data).
Privileged instruction	The process attempts to use an instruction reserved for the operating system.
Data misuse	A piece of data is of the wrong type or is not initialized.
Operator or OS intervention	For some reason, the operator or the operating system has terminated the process (e.g., if a deadlock exists).
Parent termination	When a parent terminates, the operating system may automatically terminate all of the offspring of that parent.
Parent request	A parent process typically has the authority to terminate any of its offspring.

Implementation of Processes

- OS maintains a **process table**

```
Process* proc_table[];
```

- An array (or a hash table) of structures
- One entry per process (pid is the uniq id)

`proc_table[pid]`



Process management	Memory management	File management
Registers Program counter Program status word Stack pointer Process state Priority Scheduling parameters Process ID Parent process Process group Signals Time when process started CPU time used Children's CPU time Time of next alarm	Pointer to text segment info Pointer to data segment info Pointer to stack segment info	Root directory Working directory File descriptors User ID Group ID

Implementation of Processes: Process Control Block (PCB)

- Contains the process elements
- It is possible to interrupt a running process and later resume execution as if the interruption had not occurred → state
- Created and managed by the operating system
- Key DataStructure that allows support for multiple processes

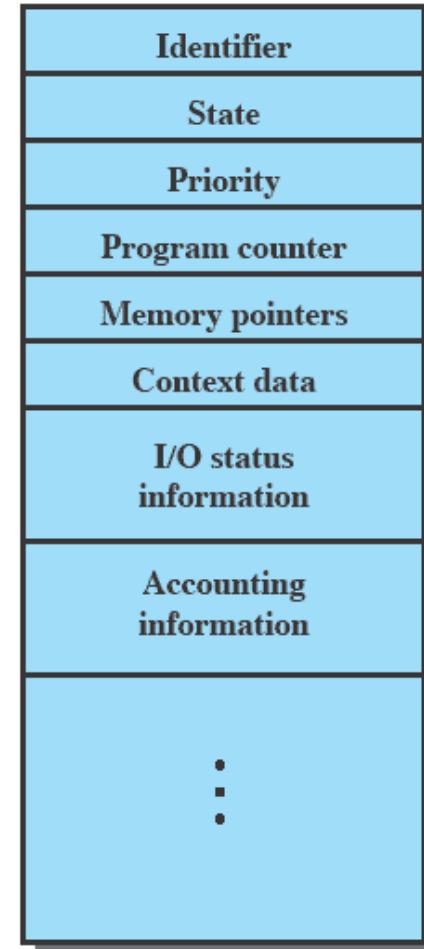
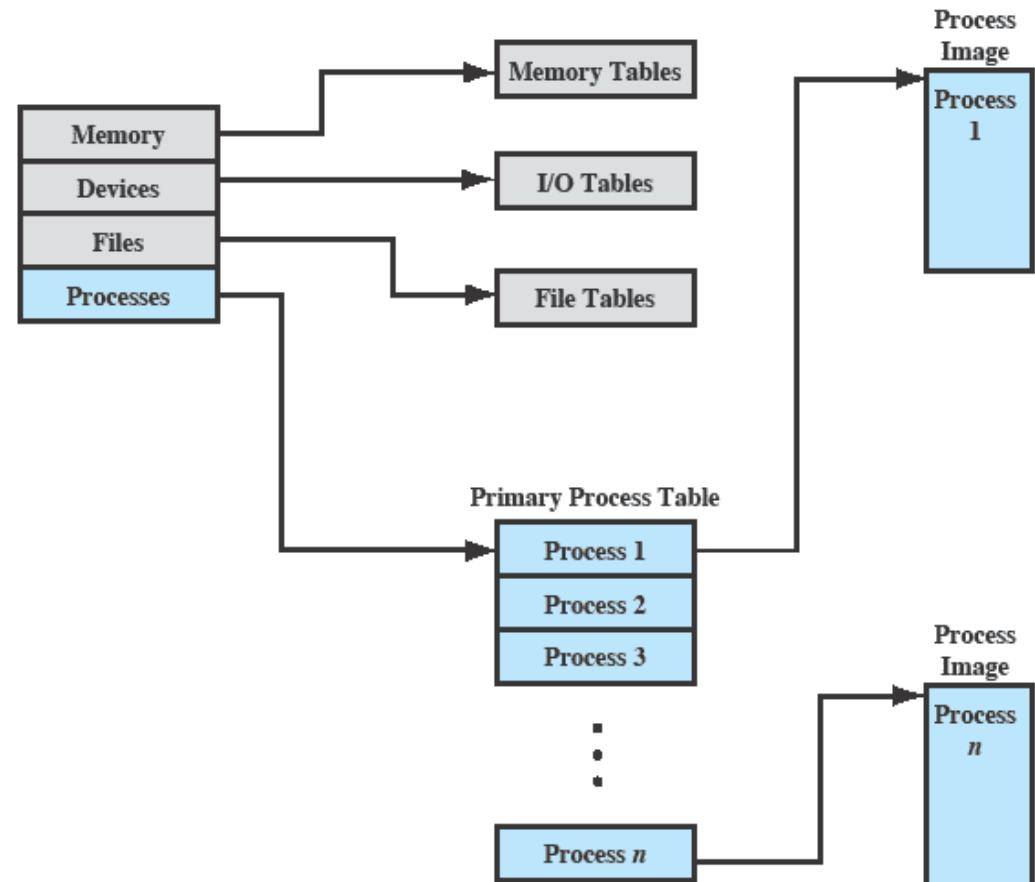


Figure 3.1 Simplified Process Control Block

OS objects related to process

Conceptual view of the tables that OS maintains in order to manage execution of processes on resources.

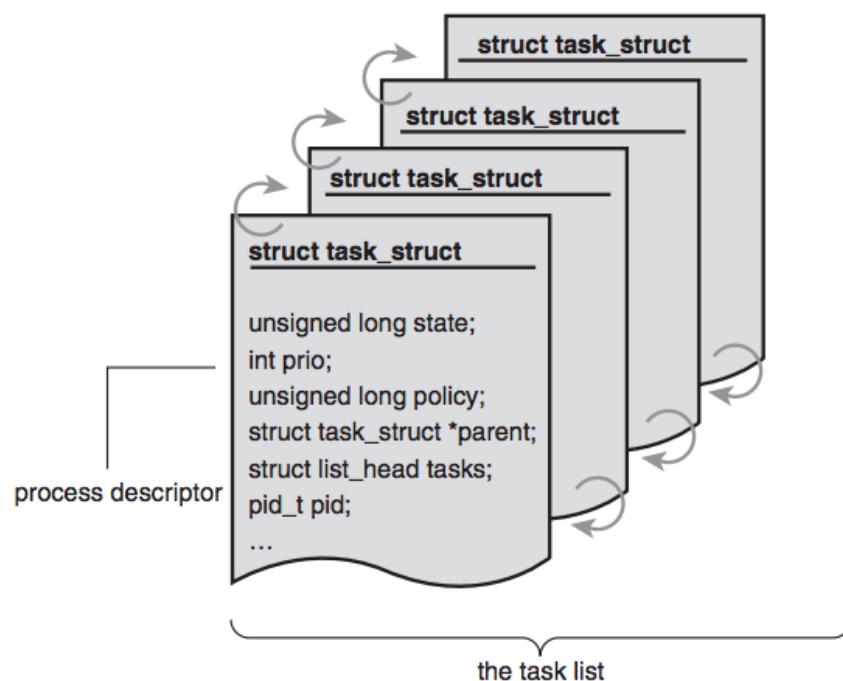


Though there is one PCB per process, there is a “mesh” of multiple objects to allow for sharing between processes where appropriate
- Memory tables, io-tables,

Linux Process (Task) object

- Linux Process \approx struct task

$\sim 1.5\text{KB} + 2^*\text{stacks}$ (user + kernel)



Need two separate stacks per process
to ensure isolation between
Kernel and User
(recall syscall)

user: stack growth till out of space
Kernel: 4KB or 8KB

Some Unix Details

- Creation of a new process: fork()
- Executing a program in that new process
- Signal notifications
- The kernel boot manually creates **ONE** process (the boot process (**pid=0**))
- In Linux , pid=0 spawns the init process (pid=1 and kernel threads)
all regular processes are created by fork() from the init process (pid=1)

UID	PID	PPID	C	S	TIME	STIME	TTY	CMD
root	1	0	0	11:11	?	00:00:02	/sbin/init	
root	2	0	0	11:11	?	00:00:00	[kthreadd]	
root	3	2	0	11:11	?	00:00:00	[rcu_gp]	
root	4	2	0	11:11	?	00:00:00	[rcu_par_gp]	
root	6	2	0	11:11	?	00:00:00	[kworker/0:0H-kblockd]	
root	7	2	0	11:11	?	00:00:00	[kworker/0:1-events]	
root	8	2	0	11:11	?	00:00:00	[kworker/u8:0-events_power_efficient]	
root	9	2	0	11:11	?	00:00:00	[mm_percpu_wq]	
root	10	2	0	11:11	?	00:00:00	[ksoftirqd/0]	

systemd+	626	1	0	11:12	?	00:00:00	/lib/systemd/systemd-resolved
systemd+	627	1	0	11:12	?	00:00:00	/lib/systemd/systemd-timesyncd
root	642	1	0	11:12	?	00:00:00	/usr/sbin/haveged --Foreground --verbose=1 -w 1024
root	670	1	0	11:12	?	00:00:00	/usr/lib/accountsservice/accounts-daemon
root	671	1	0	11:12	?	00:00:00	/usr/sbin/acpid

root	1280	1	0	11:12	?	00:00:00	/usr/sbin/lightdm
root	1296	1	0	11:12	?	00:00:00	/usr/sbin/VBoxService --pidfile /var/run/vboxadd-service.sh
root	1298	1280	0	11:12	tty7	00:00:01	/usr/lib/xorg/Xorg -core :0 -seat seat0 -auth /var/run/lightdm
/root/:0	-nolisten	tcp	vt7	-novtswitch			
rtkit	1358	1	0	11:12	?	00:00:00	/usr/libexec/rtkit-daemon
root	1385	1	0	11:12	?	00:00:00	/usr/lib/bluetooth/bluetoothd
root	1416	1280	0	11:12	?	00:00:00	lightdm --session-child 12 19
frankeh	1445	1	0	11:12	?	00:00:00	/lib/systemd/systemd --user
frankeh	1448	1445	0	11:12	?	00:00:00	(sd-pam)

fork()

```
#include <unistd.h>
```

```
pid_t fork(void);
```

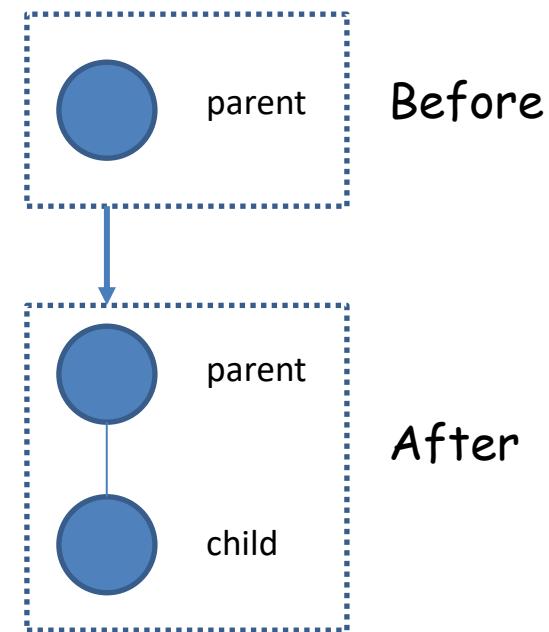
Description

fork() creates a new process by duplicating the calling process. The new process, referred to as the *child*, is an exact duplicate of the calling process, referred to as the *parent*, except for the following points:

- * The child has its own unique process ID, and this PID does not match the ID of any existing process group ([setpgid\(2\)](#)).
- * The child's parent process ID is the same as the parent's process ID.
- * The child does not inherit its parent's memory locks ([mlock\(2\)](#), [mlockall\(2\)](#)).
- * Process resource utilizations ([getrusage\(2\)](#)) and CPU time counters ([times\(2\)](#)) are reset to zero in the child.
- * The child's set of pending signals is initially empty ([sigpending\(2\)](#)).
- * The child does not inherit semaphore adjustments from its parent ([semop\(2\)](#)).
- * The child does not inherit record locks from its parent ([fcntl\(2\)](#)).
- * The child does not inherit timers from its parent ([setitimer\(2\)](#), [alarm\(2\)](#), [timer_create\(2\)](#)).
- * The child does not inherit outstanding asynchronous I/O operations from its parent ([aio_read\(3\)](#), [aio_write\(3\)](#)), nor does it inherit any asynchronous I/O contexts from its parent (see [io_setup\(2\)](#)).

Only **LOGICALLY** duplicates the whole process, otherwise way to expensive (will cover in MemMgmt)

!



Before

After

fork()

```
#include <stdio.h>
#include <unistd.h>

int main(int argc, char **argv)
{
    pid_t pid = fork(); → syscall that creates new PCB and duplicates
    if (pid == 0)           Address Space
    {
        // child process → Child runs now here
    }
    else if (pid > 0)
    {
        // parent process → Parent continues here
    }
    else
    {
        // fork failed
        printf("fork() failed!\n");
        return 1;
    }
}
```

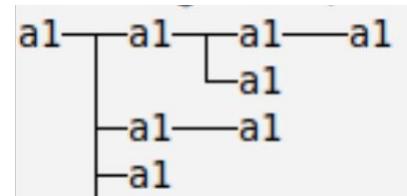
fork()

```
int main(int argc, char **argv)
{
    for (int i=0; i<3; i++) {
        // observe <i>
        fork();
    }
    sleep(10);
}
```

Resulting
Process Tree

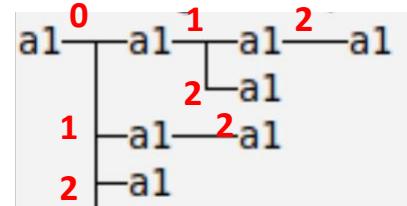


pstree -c <pid-of-first-a1>



On fork() the entire process state is copied (memory, stack , ..) !

Consider variable <i> at “//observe <i>”



Now consider the classical forkbomb():

```
int main(int argc, char **argv)
{
    for (;;)
        fork();
}
```

It will reender your machine unresponsive in seconds.

execv()

exec, execl, execle, execv, execvp, execvpe - execute a file

Synopsis

```
#include <unistd.h>
extern char **environ;

int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg,
..., char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[],
char *const envp[]);
```

execv () replaces the “image==executable” of the current process with a new one.

The execv () functions return only if an error has occurred.

The return value is -1, and errno is set to indicate the error.

Description

The **exec()** family of functions replaces the current process image with a new process image. The functions described in this manual page are front-ends for [execve\(2\)](#). (See the manual page for [execve\(2\)](#) for further details about the replacement of the current process image.)

The initial argument for these functions is the name of a file that is to be executed.

The *const char *arg* and subsequent ellipses in the **execl()**, **execlp()**, and **execle()** functions can be thought of as *arg0*, *arg1*, ..., *argn*. Together they describe a list of one or more pointers to null-terminated strings that represent the argument list available to the executed program. The first argument, by convention, should point to the filename associated with the file being executed. The list of arguments *must* be terminated by a NULL pointer, and, since these are variadic functions, this pointer must be cast (*char **) *NULL*.

The **execv()**, **execvp()**, and **execvpe()** functions provide an array of pointers to null-terminated strings that represent the argument list available to the new program. The first argument, by convention, should point to the filename associated with the file being executed. The array of pointers *must* be terminated by a NULL pointer.

The **execle()** and **execvpe()** functions allow the caller to specify the environment of the executed program via the argument *envp*. The *envp* argument is an array of pointers to null-terminated strings and *must* be terminated by a NULL pointer. The other functions take the environment for the new process image from the external variable *environ* in the calling process.

execv()

```
#include <stdio.h>
#include <unistd.h>
int main(int argc, char **argv)
{
    pid_t pid = fork();           → Creates new PCB and Address Space
    if (pid == 0)
    {
        execv(path,executablename) → Child starts new program image of new process
        this will NEVER return but for an error
    }
    else if (pid > 0)
    {
        int status;
        waitpid(pid,&status,option) → Parent waits for child process to finish
    }
    else
    {
        // fork failed
        printf("fork() failed: \n", strerror(errno));
        return -1;
    }
}
```

Fork , exec, wait (and their variants) are system calls

pid is the object handle that the operating system returns for a process

clone()

See:

NAME [top](#)

clone, __clone2 - create a child process

<http://man7.org/linux/man-pages/man2/clone.2.html>

SYNOPSIS [top](#)

```
/* Prototype for the glibc wrapper function */

#define _GNU_SOURCE
#include <sched.h>

int clone(int (*fn)(void *), void *child_stack,
          int flags, void *arg, ...
          /* pid_t *ptid, struct user_desc *tls, pid_t *ctid */);

/* Prototype for the raw system call */

long clone(unsigned long flags, void *child_stack,
           void *ptid, void *ctid,
           struct pt_regs *regs);
```

DESCRIPTION [top](#)

clone() creates a new process, in a manner similar to [fork\(2\)](#).

This page describes both the glibc clone() wrapper function and the underlying system call on which it is based. The main text describes the wrapper function; the differences for the raw system call are described toward the end of this page.

Unlike [fork\(2\)](#), clone() allows the child process to share parts of its execution context with the calling process, such as the memory space, the table of file descriptors, and the table of signal handlers. (Note that on this manual page, "calling process" normally corresponds to "parent process". But see the description of [CLONE_PARENT](#) below.)

One use of clone() is to implement threads: multiple threads of control in a program that run concurrently in a shared memory space.

For what things can be clone.

This is the bases on how threads are created.

Linux/Unix internally uses objects to create processes / threads

How these objects interrelate depends on fork/clone calls

Signal()

- Means to “signal a process” (i.e. get its attention).
- There are a set of signals that can be sent to a process (some require permissions).
- Process indicates which signal it wants to catch and provides a call back function
- When the signal is to be sent (event or “kill <signal> pid”) the kernel delivers that signal.

No.	Short name	What it means
1	SIGHUP	If a process is being run from terminal and that terminal suddenly goes away then the process receives this signal. “HUP” is short for “hang up” and refers to hanging up the telephone in the days of telephone modems.
2	SIGINT	The process was “interrupted”. This happens when you press Control+C on the controlling terminal.
3	SIGQUIT	
4	SIGILL	Illegal instruction. The program contained some machine code the CPU can't understand.
5	SIGTRAP	This signal is used mainly from within debuggers and program tracers.
6	SIGABRT	The program called the <code>abort()</code> function. This is an emergency stop.
7	SIGBUS	An attempt was made to access memory incorrectly. This can be caused by alignment errors in memory access etc.
8	SIGFPE	A floating point exception happened in the program.
9	SIGKILL	The process was explicitly killed by somebody wielding the <code>kill</code> program.
10	SIGUSR1	Left for the programmers to do whatever they want.
11	SIGSEGV	An attempt was made to access memory not allocated to the process. This is often caused by reading off the end of arrays etc.
12	SIGUSR2	Left for the programmers to do whatever they want.
13	SIGPIPE	If a process is producing output that is being fed into another process that consume it via a pipe (“producer consumer”) and the consumer dies then the producer is sent this signal.
14	SIGNALRM	A process can request a “wake up call” from the operating system at some time in the future by calling the <code>alarm()</code> function. When that time comes round the wake up call consists of this signal.
15	SIGTERM	The process was explicitly killed by somebody wielding the <code>kill</code> program.
16	unused	
17	SIGCHLD	The process had previously created one or more child processes with the <code>fork()</code> function. One or more of these processes has since died.
18	SIGCONT	(To be read in conjunction with SIGSTOP.) If a process has been paused by sending it SIGSTOP then sending SIGCONT to the process wakes it up again (“continues” it).
19	SIGSTOP	(To be read in conjunction with SIGCONT.) If a process is sent SIGSTOP it is paused by the operating system. All its

Signal()

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

void sig_handler(int signo) {
    if (signo == SIGINT)
        printf("received SIGINT\n");
}

int main(void) {
    // install the handler
    if (signal(SIGINT, sig_handler) == SIG_ERR)
        printf("\ncan't catch SIGINT\n");

    // A long long wait so that we can easily
    // issue a signal to this process

    while(1) sleep(1);
    return 0;
}
```

When signal is delivered:

- * kernel stops threads in process
- * kernel "adds a user stack frame"
- * kernel "switches IPC to *sig_handler*"
- * kernel continues process
- * Process will continue with *sig_handler*
- * Process on completion will call back to kernel

No.	Short name	What it means
		state is preserved ready for it to be restarted (by SIGCONT) but it doesn't get any more CPU cycles until then.
20	SIGTSTP	Essentially the same as SIGSTOP. This is the signal sent when the user hits Control+Z on the terminal. (SIGTSTP is short for "terminal stop") The only difference between SIGTSTP and SIGSTOP is that pausing is only the <i>default</i> action for SIGTSTP but is the <i>required</i> action for SIGSTOP. The process can opt to handle SIGTSTP differently but gets no choice regarding SIGSTOP.
21	SIGTTIN	The operating system sends this signal to a backgrounded process when it tries to read input from its terminal. The typical response is to pause (as per SIGSTOP and SIGTSTP) and wait for the SIGCONT that arrives when the process is brought back to the foreground.
22	SIGTTOU	The operating system sends this signal to a backgrounded process when it tries to write output to its terminal. The typical response is as per SIGTTIN.
23	SIGURG	The operating system sends this signal to a process using a network connection when " urgent " out of band data is sent to it.
24	SIGXCPU	The operating system sends this signal to a process that has exceeded its CPU limit. You can cancel any CPU limit with the shell command "ulimit -t unlimited" prior to running make though it is more likely that something has gone wrong if you reach the CPU limit in make.
25	SIGXFSZ	The operating system sends this signal to a process that has tried to create a file above the file size limit. You can cancel any file size limit with the shell command "ulimit -f unlimited" prior to running make though it is more likely that something has gone wrong if you reach the file size limit in make.
26	SIGVTALRM	This is very similar to SIGALRM, but while SIGALRM is sent after a certain amount of real time has passed, SIGVTALRM is sent after a certain amount of time has been spent running the process.
27	SIGPROF	This is also very similar to SIGALRM and SIGVTALRM, but while SIGALRM is sent after a certain amount of real time has passed, SIGPROF is sent after a certain amount of time has been spent running the process and running system code on behalf of the process.
28	SIGWINCH	(Mostly unused these days.) A process used to be sent this signal when one of its windows was resized.
29	SIGIO	(Also known as SIGPOLL.) A process can arrange to have this signal sent to it when there is some input ready for it to process or an output channel has become ready for writing.
30	SIGPWR	A signal sent to processes by a power management service to indicate that power has switched to a short term emergency power supply. The process

A few basics to remember

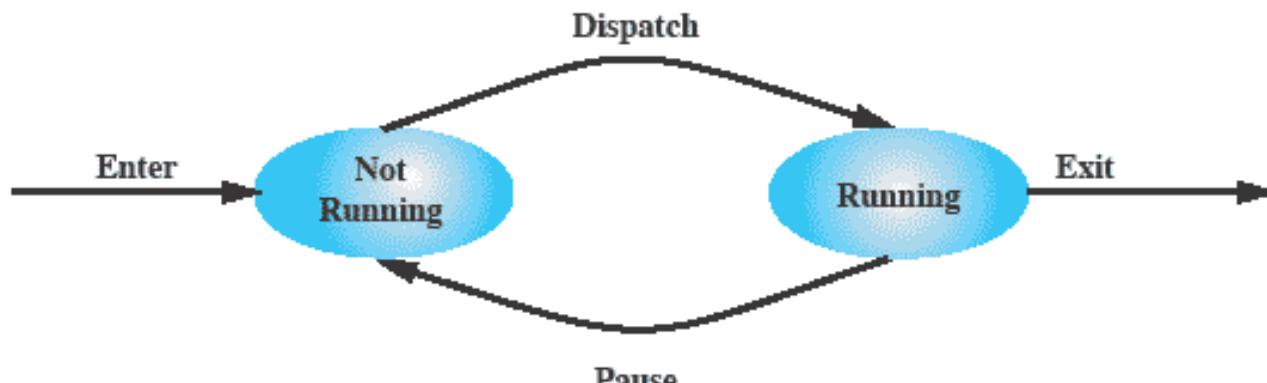
- `fork()`, `exec()`, `wait()` and its variants are OS system calls to create and manipulate processes
- PIDs are *handles* that the operating system identifies life processes by
- There are user processes and system processes

```
File Edit Tabs Help
top - 07:35:00 up 2:07, 1 user, load average: 0.18, 0.05, 0.01
Tasks: 160 total, 1 running, 159 sleeping, 0 stopped, 0 zombie
%Cpu(s): 0.2 us, 0.2 sy, 0.0 ni, 99.7 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem : 2048316 total, 597732 free, 216968 used, 1233616 buff/cache
KiB Swap: 2096124 total, 2096124 free, 0 used. 1650628 avail Mem

PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
 925 root 20 0 281812 75396 28132 S 1.3 3.7 0:36.30 Xorg
 2322 frankeh 20 0 430512 31004 21536 S 1.0 1.5 0:17.76 x-terminal-emul
 1865 frankeh 20 0 116204 2160 1792 S 0.3 0.1 0:17.16 VBoxClient
 2064 frankeh 20 0 379044 19440 15280 S 0.3 0.9 0:01.44 openbox
    1 root 20 0 120004 6148 3968 S 0.0 0.3 0:02.46 systemd
    2 root 20 0 0 0 0 S 0.0 0.0 0:00.00 kthreadd
    3 root 20 0 0 0 0 S 0.0 0.0 0:00.12 ksoftirqd/0
    5 root 0 -20 0 0 0 S 0.0 0.0 0:00.00 kworker/0:0H
    7 root 20 0 0 0 0 S 0.0 0.0 0:00.89 rcu_sched
    8 root 20 0 0 0 0 S 0.0 0.0 0:00.00 rcu_bh
    9 root rt 0 0 0 0 S 0.0 0.0 0:00.07 migration/0
   10 root rt 0 0 0 0 S 0.0 0.0 0:00.06 watchdog/0
   11 root rt 0 0 0 0 S 0.0 0.0 0:00.07 watchdog/1
   12 root rt 0 0 0 0 S 0.0 0.0 0:00.07 migration/1
   13 root 20 0 0 0 0 S 0.0 0.0 0:00.41 ksoftirqd/1
   15 root 0 -20 0 0 0 S 0.0 0.0 0:00.00 kworker/1:0H
   16 root 20 0 0 0 0 S 0.0 0.0 0:00.00 kdevtmpfs
   17 root 0 -20 0 0 0 S 0.0 0.0 0:00.00 netns
   18 root 0 -20 0 0 0 S 0.0 0.0 0:00.00 perf
```

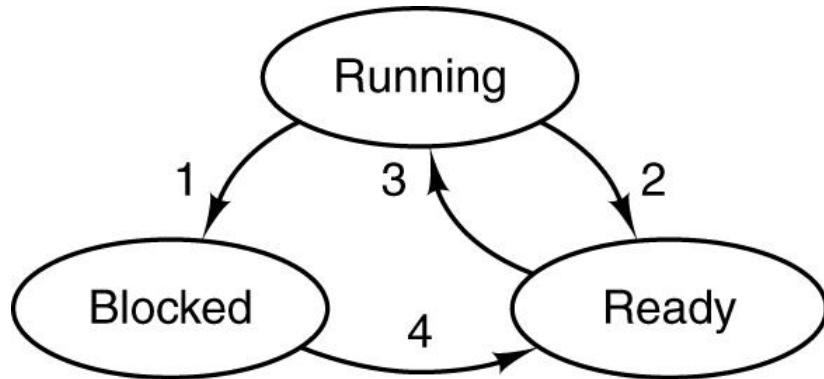
Process State Model

- Depending on the implementation, there can be several possible state models.
- The Simplest one: Two-state diagram

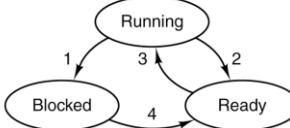
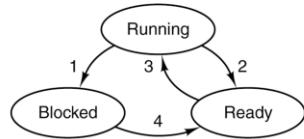


(a) State transition diagram

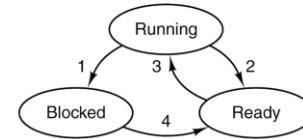
Process State Model: Three-State Model



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available



...



Process_1

Process_2

Process_N

Scheduler (picks process and drives state transition)

Process State Model: Five-State Model

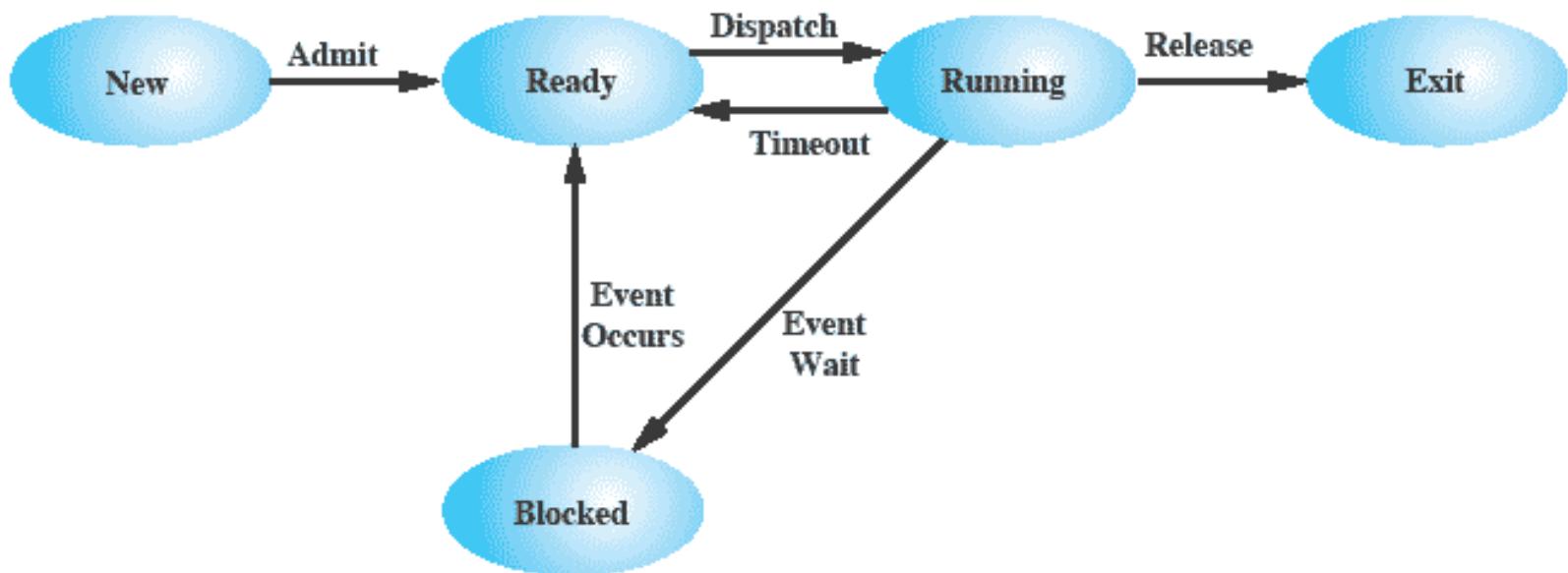
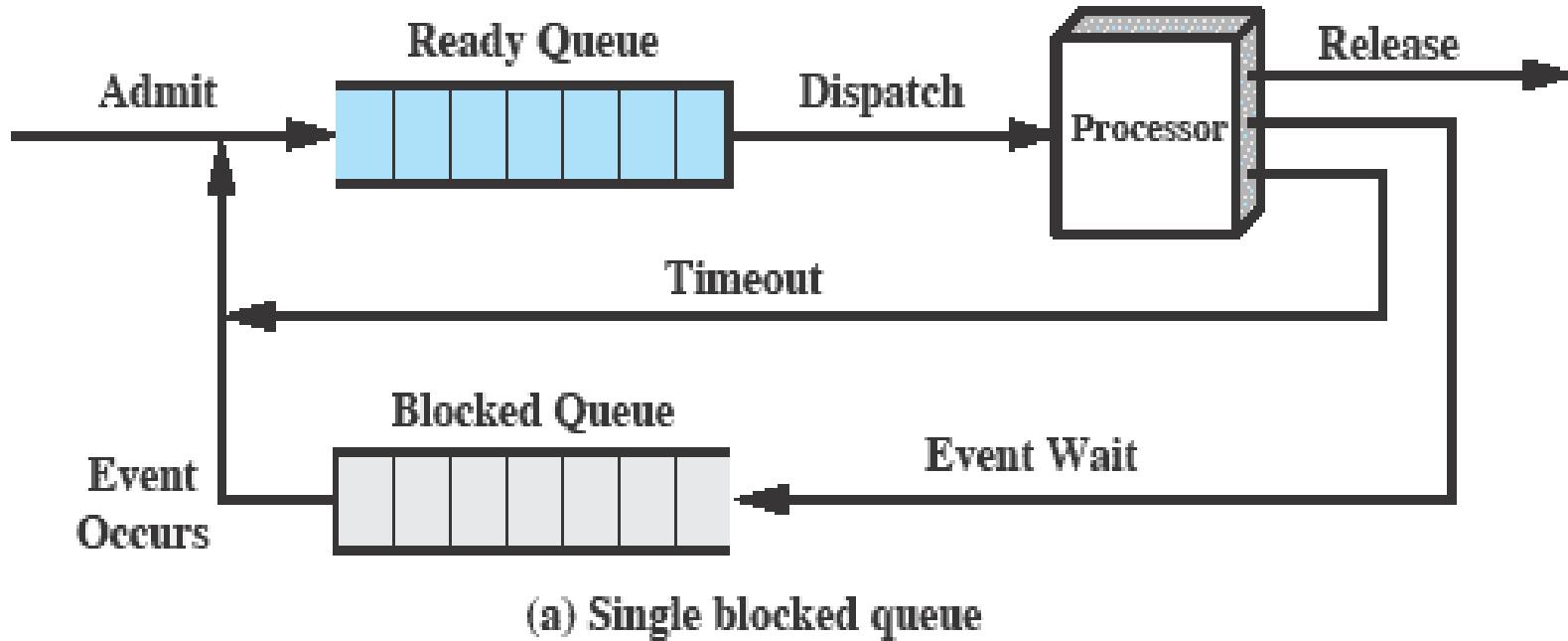
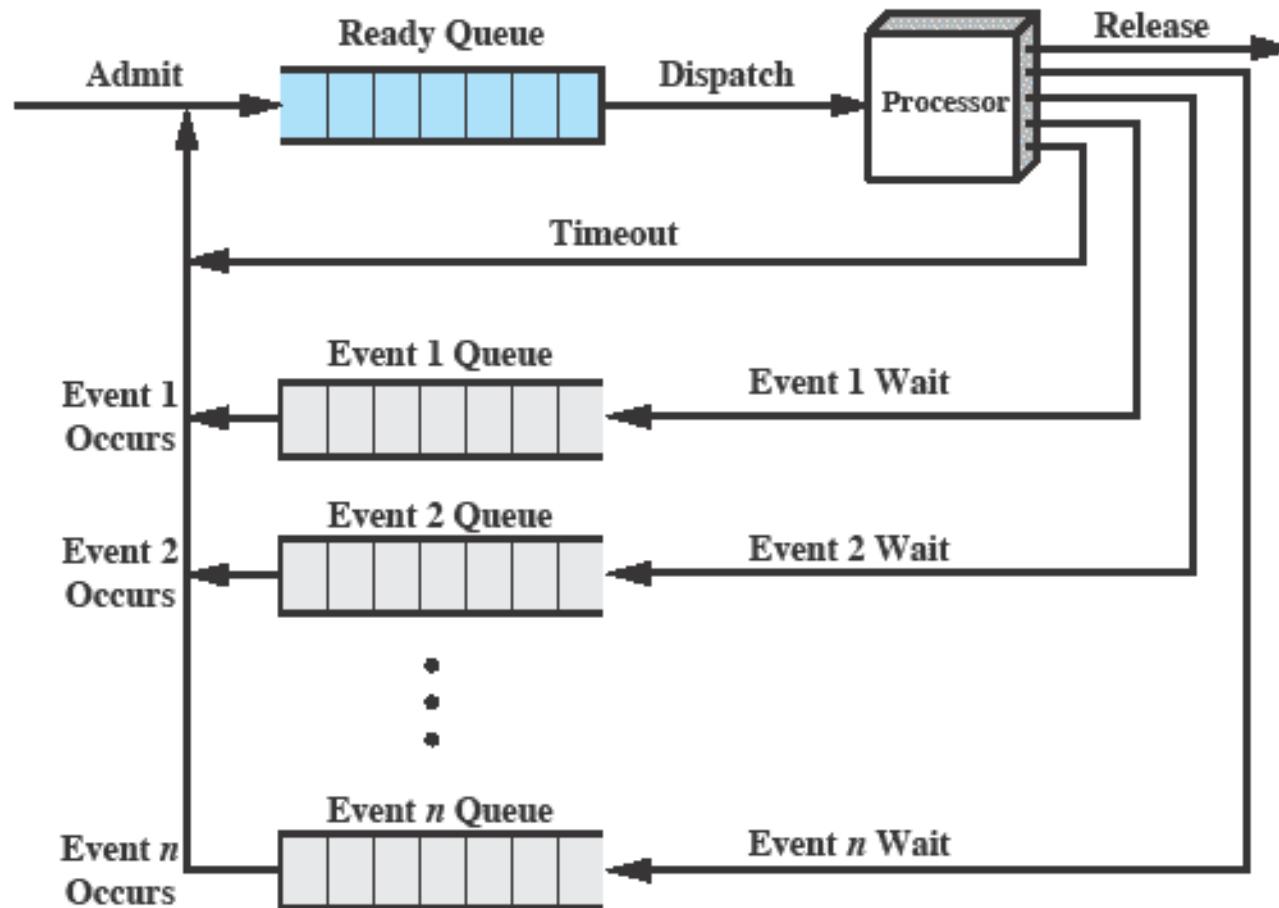


Figure 3.6 Five-State Process Model

Using Queues to Manage Processes

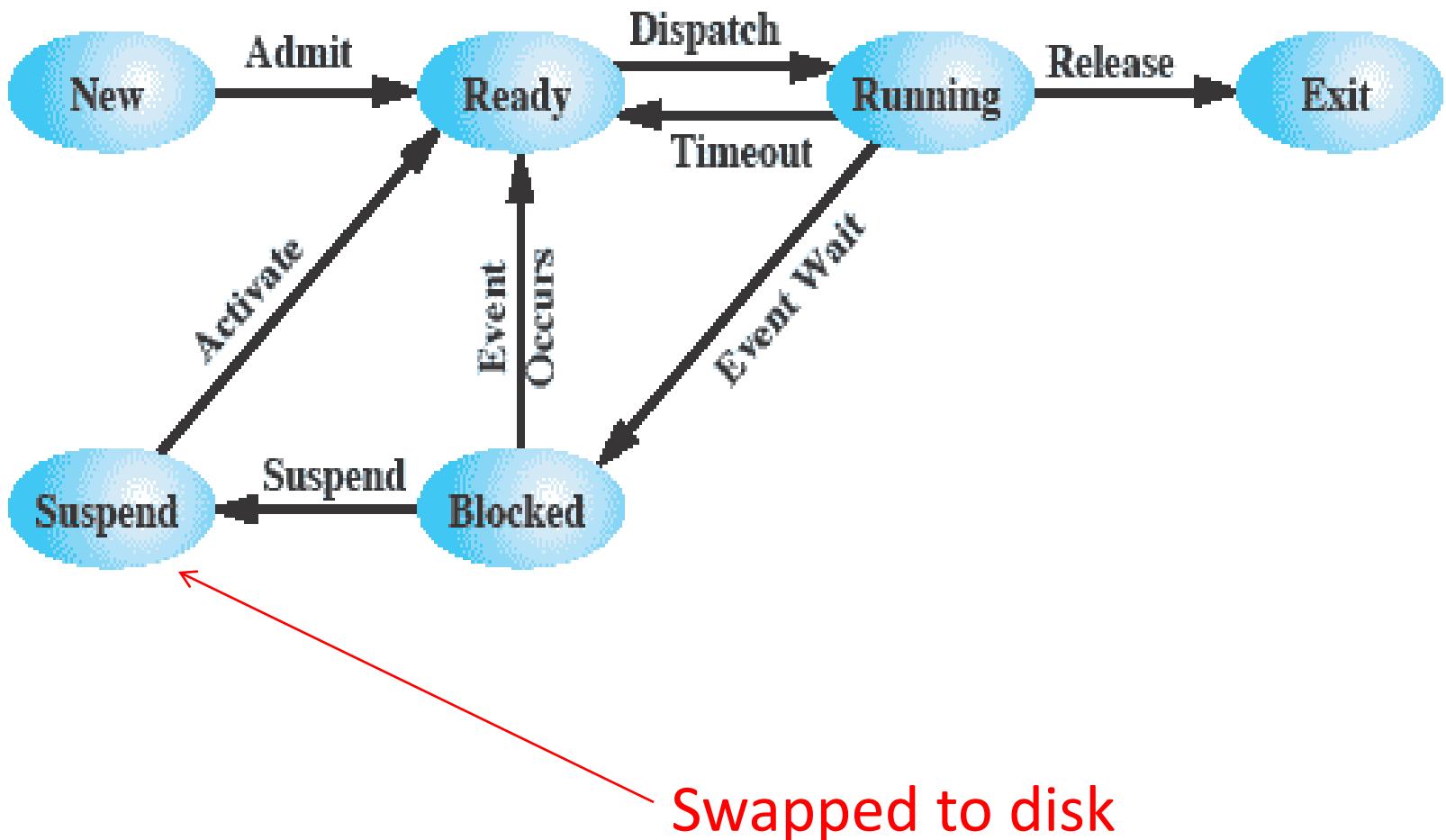


Using Queues to Manage Processes



(b) Multiple blocked queues

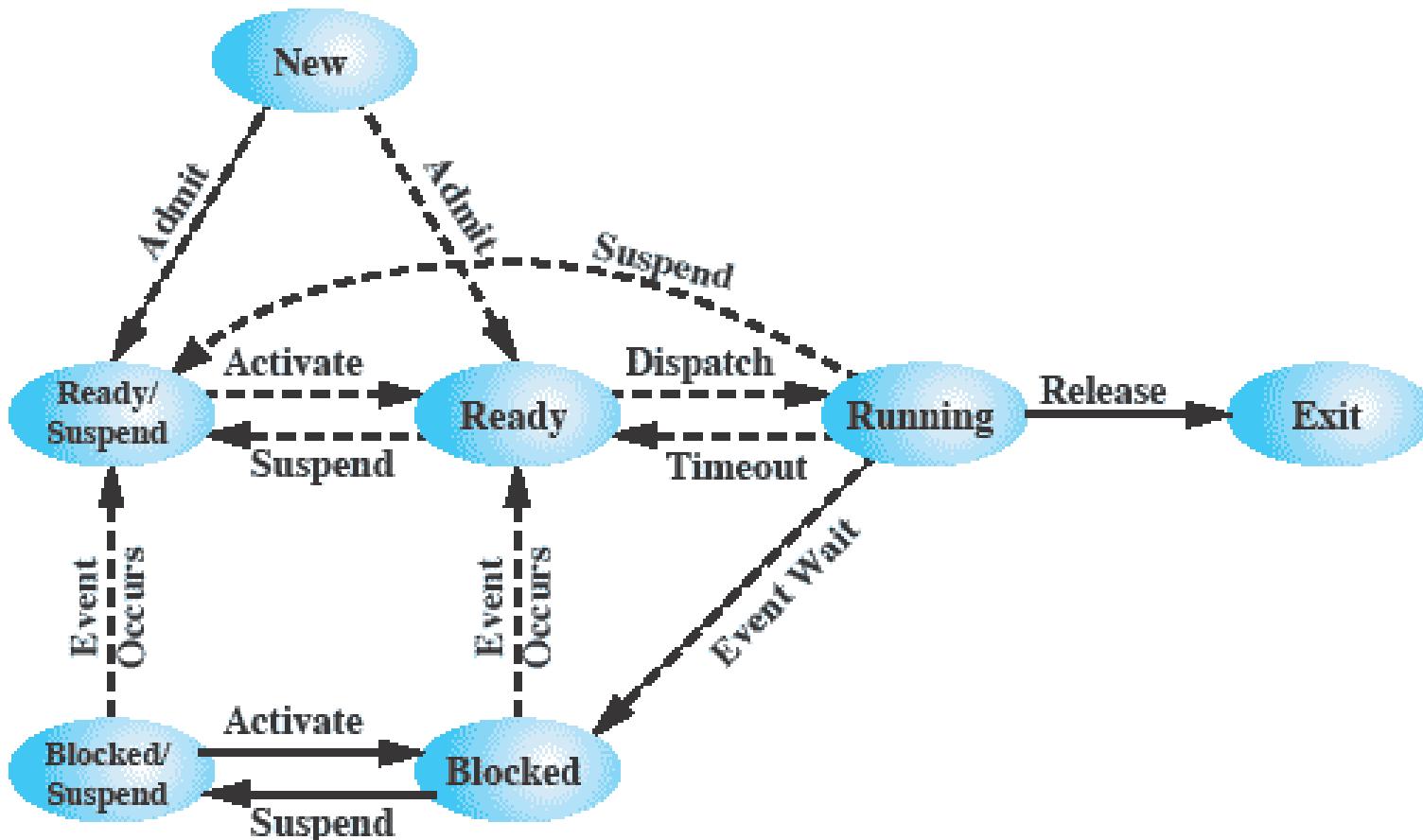
One Extra State!



Swapped to disk

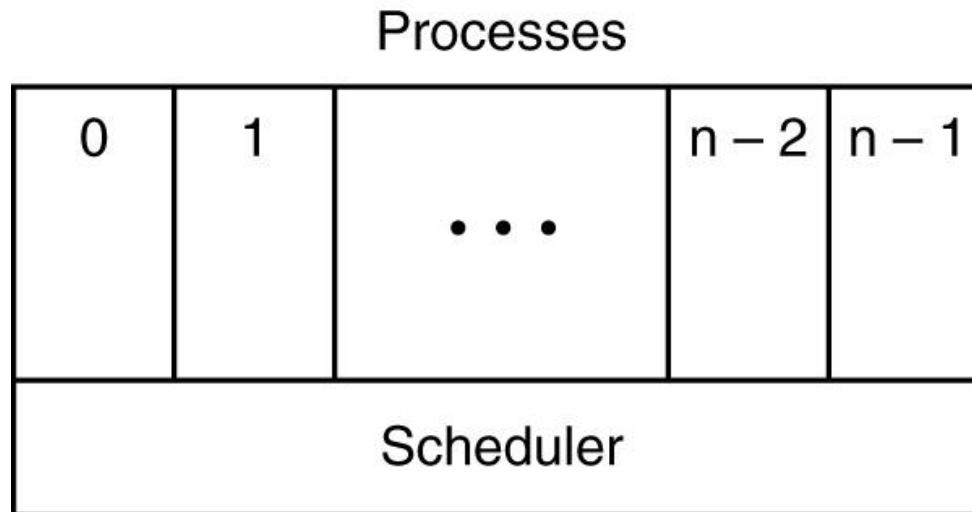
Whole process and its “address space” is moved to disk

One Extra State!

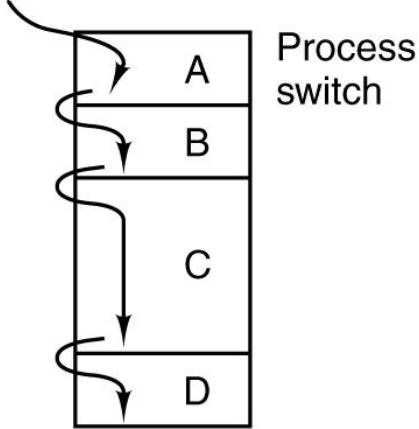


Multiprogramming

- One CPU and several processes
- CPU switches from process to process quickly

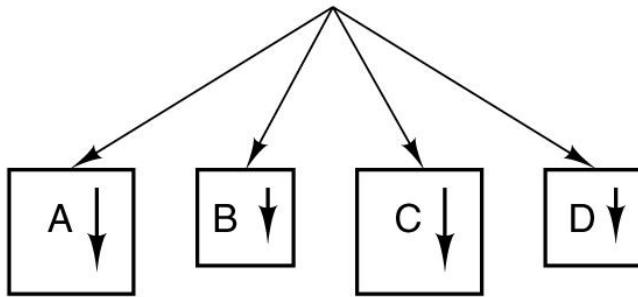


One program counter



(a)

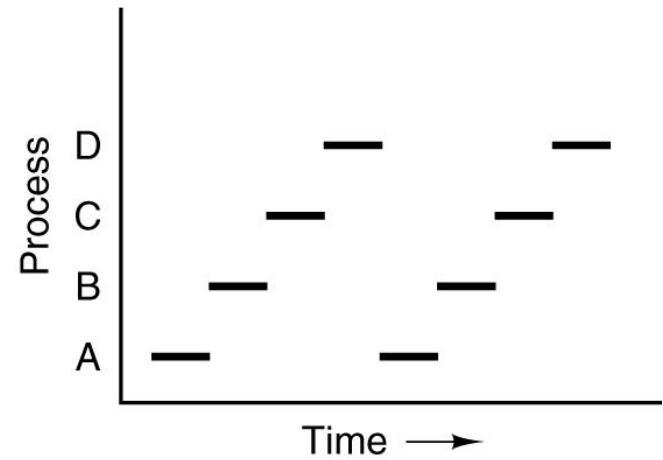
Four program counters



(b)

What Really Happens

What We Think It Happens



(c)

Running the same program several times will not result in the same execution times due to:

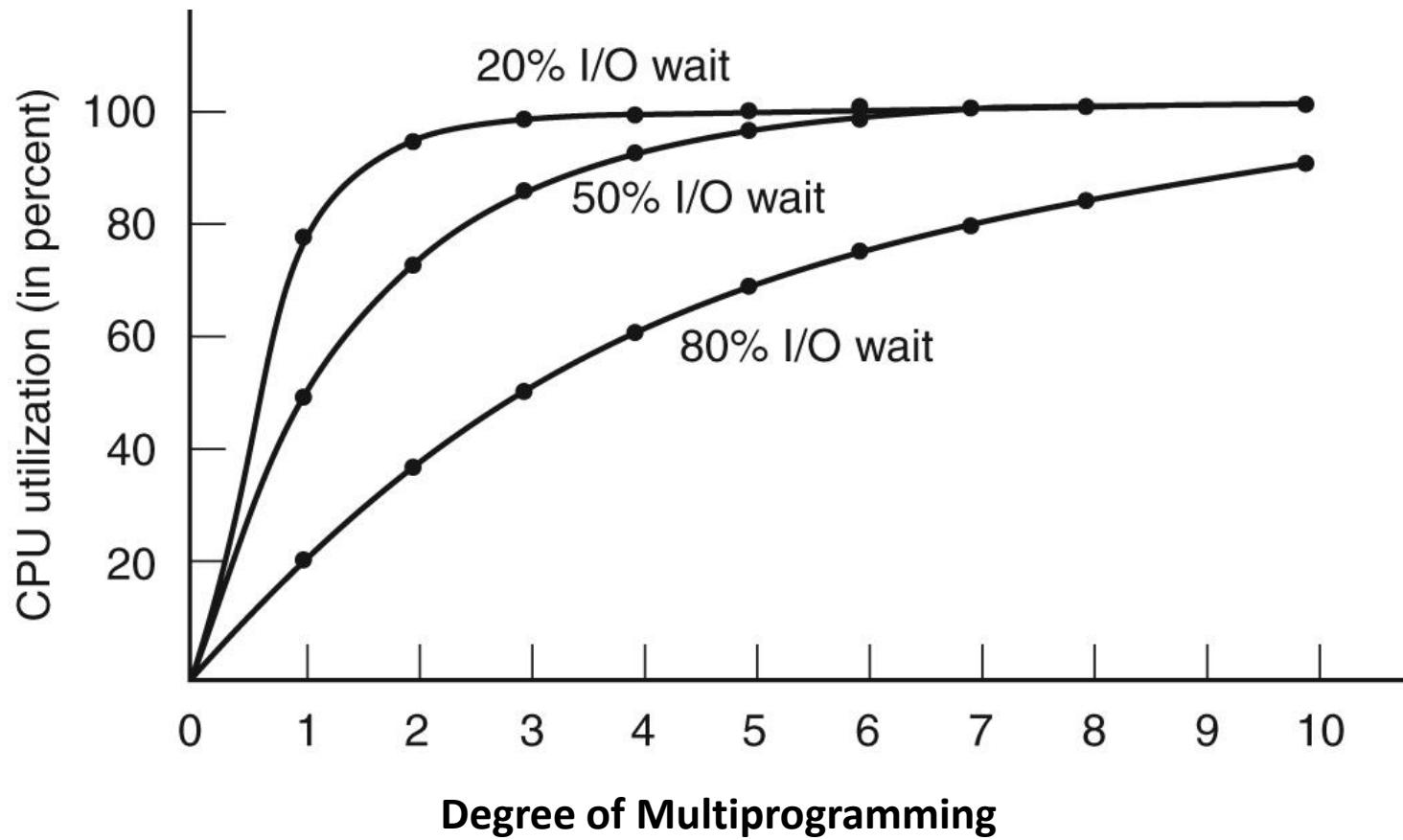
- interrupts
- multi-programming

Concurrency vs. Parallelism

- **Concurrency** is when two or more tasks can start, run, and complete in overlapping time periods. It doesn't necessarily mean they'll ever both be running at the same instant. For example, multitasking on a single-core machine.
- **Parallelism** is when tasks *literally* run at the same time, e.g., on a multicore processor.

Simple Modeling of Multiprogramming

- A process spends fraction p waiting for I/O
- Assume n processes in memory at once
- The probability that all processes are waiting for I/O at once is p^n
- So -> CPU Utilization = $1 - p^n$



Multiprogramming lets processes use the CPU when it would otherwise become idle.

How to do multiprogramming

- Really a question of how to increase concurrency (e.g. multi-core system) and overlap I/O with computation.
- Example Webserver:
 - If single process, every system call that blocks will block forward progress
 - Let's discuss !!!!!!

Solution #1

- Multiple Processes
- What's the issue ?
 - Resource consumption
 - Each process has its own address space
(code, stack, heap, files,)
 - Who owns perceived single resource:
 - E.g. webserver port 80 / 1080 / 8080 ????

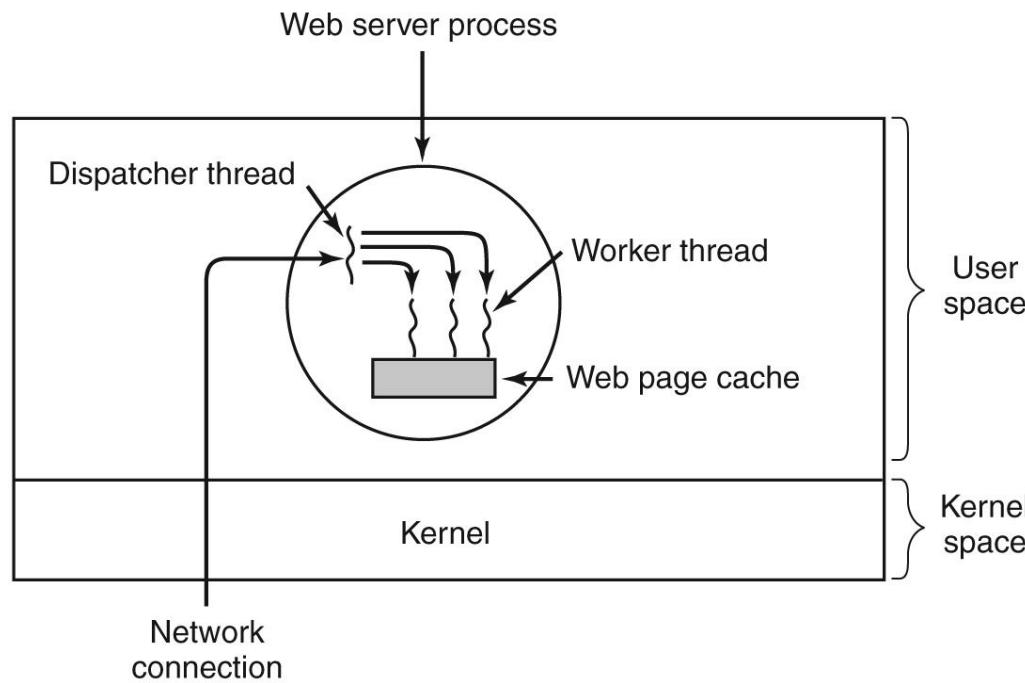
Threads

- Multiple threads of control within a process
 - unique execution
- All threads of a process share the same address space and resources (with exception of stack)

Why Threads?

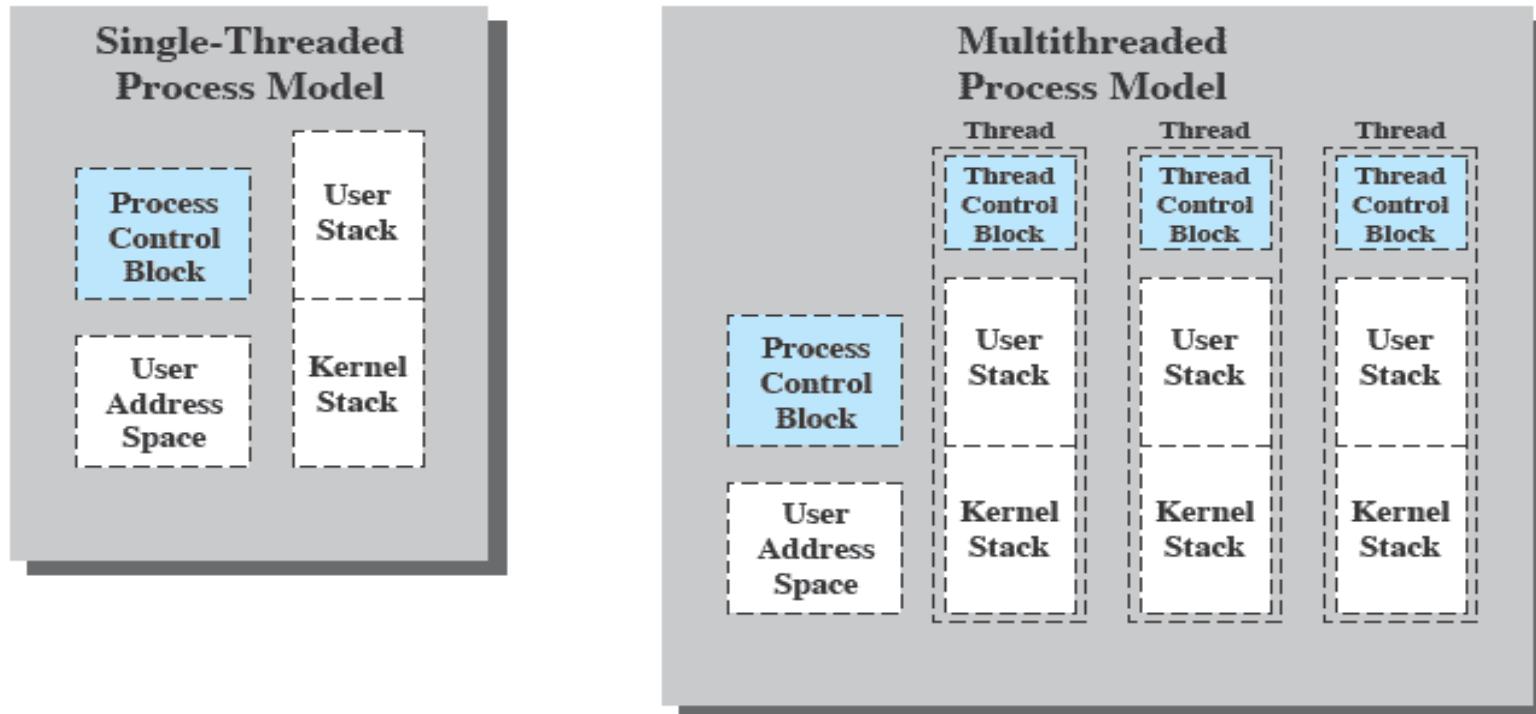
- For some applications many activities can happen at once
 - With threads, programming becomes easier
 - Otherwise application needs to actively manage different logical executions in the process
 - This requires significant state management
 - Benefit applications with I/O and processing that can overlap
- Lighter weight than processes
 - Faster to create and restore
 - (we just really need a stack and an execution unit, but don't have create new address space etc.)

Example : Multithreaded Web Server



Processes vs. Threads

OS internal Data Structure implications:



Processes vs Threads

- Process groups resources
 - (Address Space, files)
- Threads are entities scheduled for execution on CPU
- Threads can be in any of several states: *running, blocked, ready, and terminated* (remember the process state model ?)
- No protections among threads (unlike processes) [Why?] → **this is important**

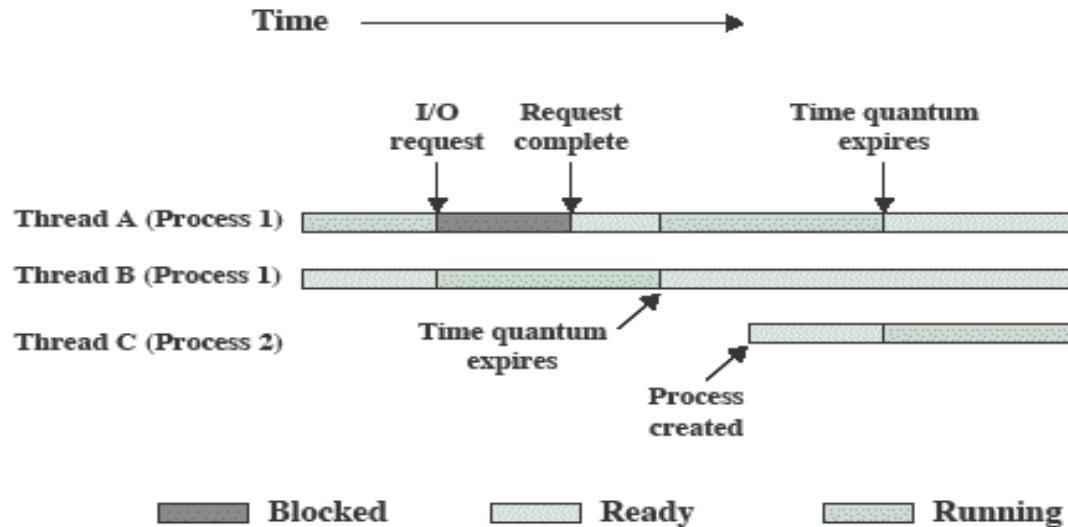
Processes vs Threads

- The unit of dispatching is referred to as a *thread* or *lightweight process (lwp)*
- The unit of resource ownership is referred to as a *process* or *task*
(unfortunately in linux *struct task* represents both a *process* and *thread*)
- *Multithreading* - The ability of an OS to support multiple, concurrent paths of execution within a single process

Processes vs Threads

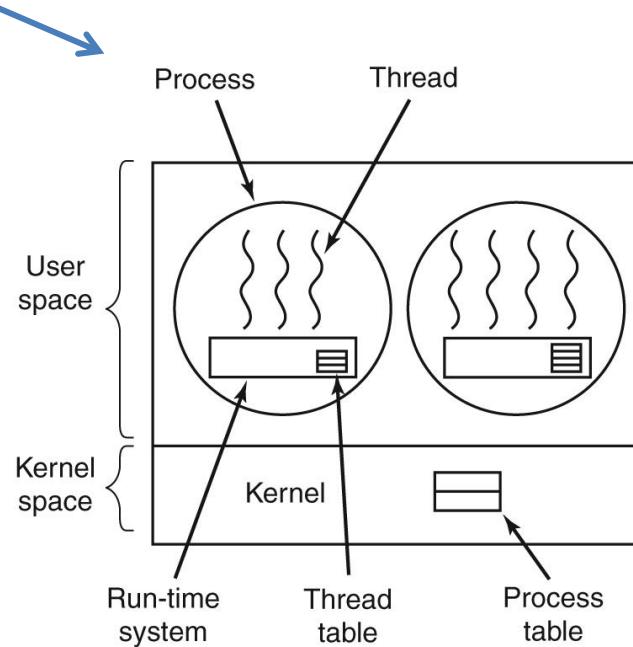
- Process is the unit for resource allocation and a unit of protection.
- Process has its own (one) address space.
- A thread has:
 - an execution state (Running, Ready, etc.)
 - saved thread context when not running
 - an execution stack
 - some per-thread static storage for local variables
 - access to the memory and resources of its process (all threads of a process share this)

Multithreading on Uniprocessor System

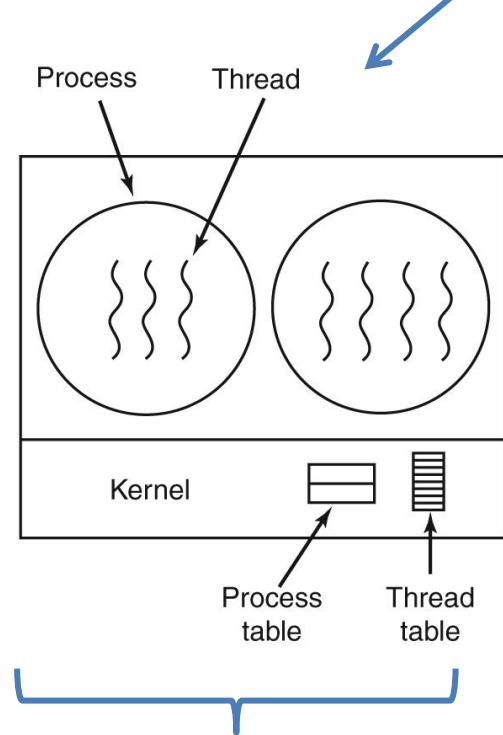


Where to Put The Thread Implementation / Package?

User space



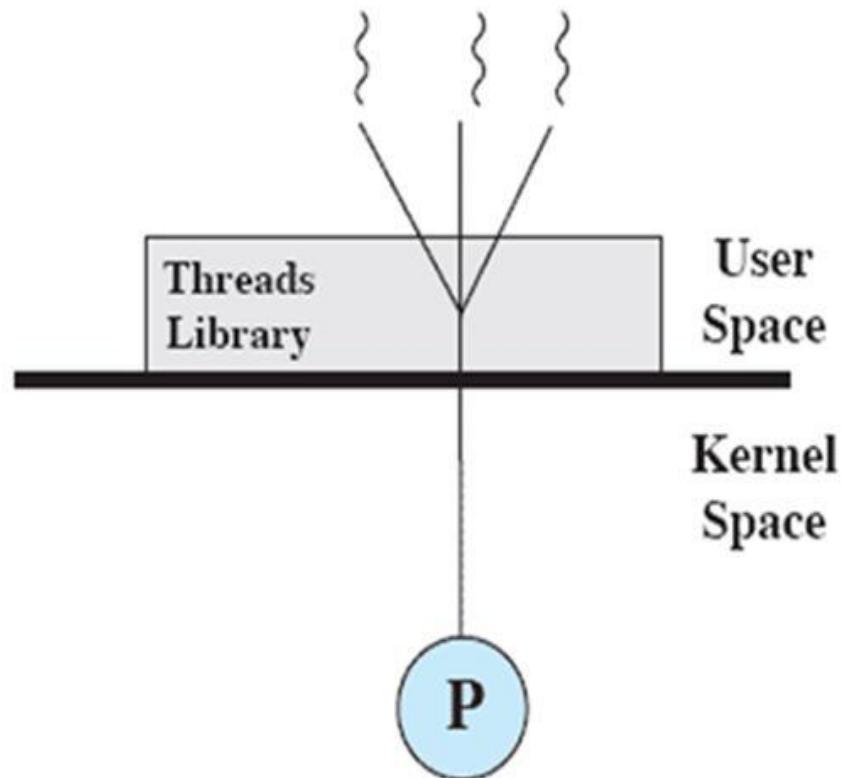
Kernel space



Discussed in previous slides

User-Level Threads (ULT)

- All thread management is done by the application
- Initially developed to run on kernels that are not multithreading capable.
- The kernel is not aware of the existence of threads

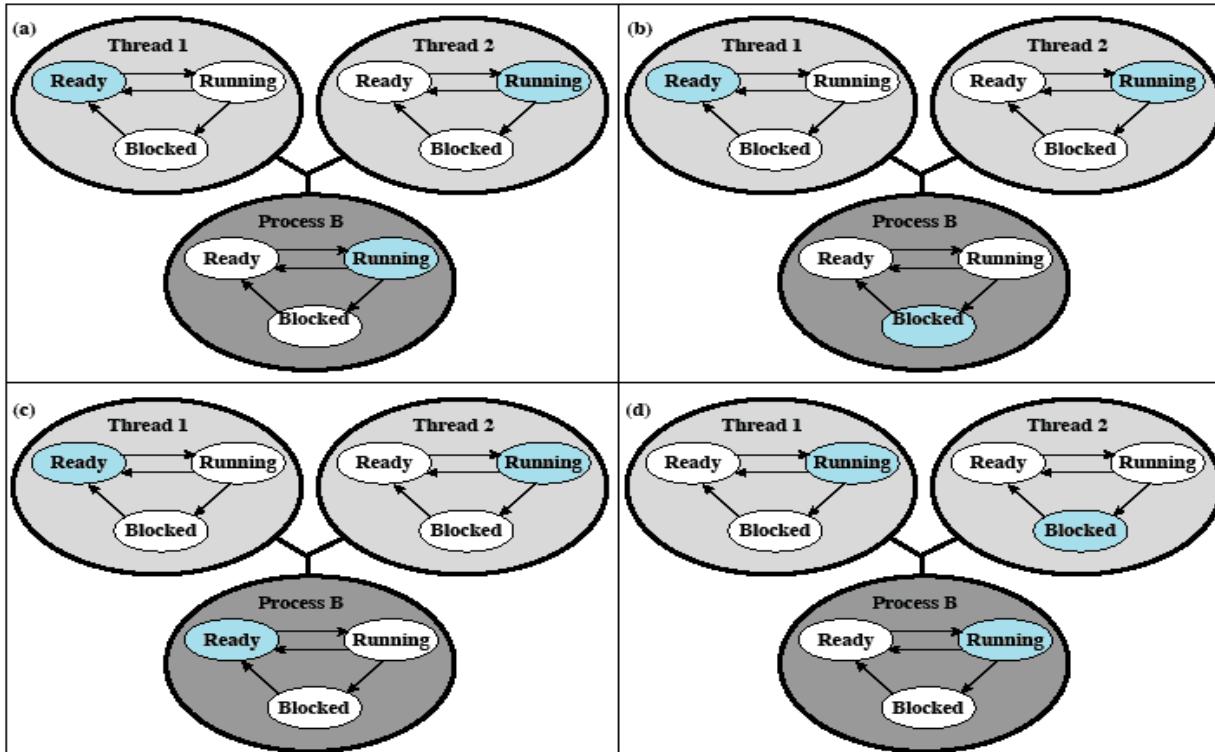


Implementing Threads in User Space

- Threads are implemented by a library
- Kernel knows nothing about threads, only processes and there is one “execution” associated with that process.
- Only one thread of the process can technically be executing at a given time.
- Each process needs its own private **thread table** in userspace
- Thread table is managed by the runtime system

User-Level Threads (ULTs)

- The kernel continues to schedule the process as a unit and assigns a single execution state .



User-Level Threads (ULTs)

Advantages

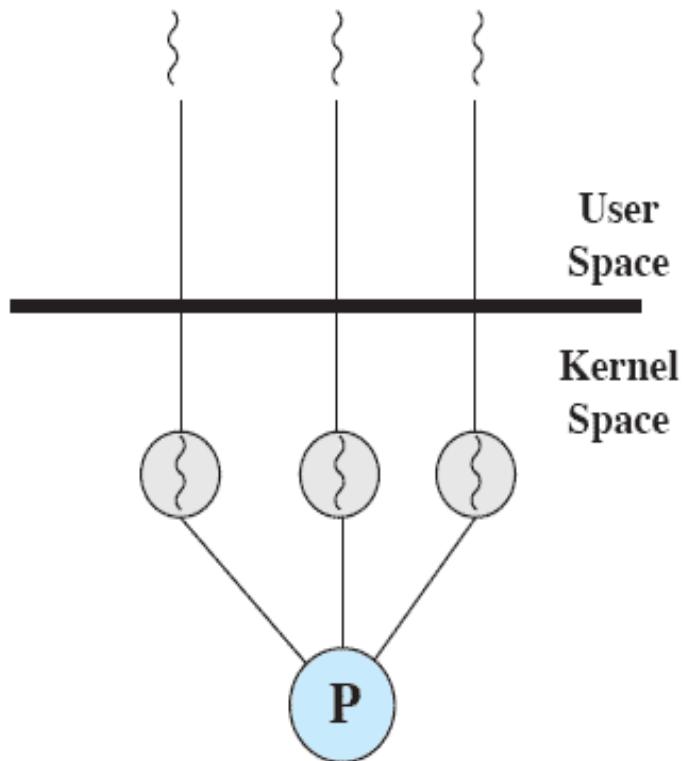
- Thread switch does not require kernel-mode (still similar expense)
- Scheduling (of threads) can be application specific.
- Can run on any OS.
- Scales better.

Disadvantages

- A system-call by one thread can block all threads of that process.
- Page fault blocks the whole process.
- In pure ULT, multithreading cannot take advantage of multiprocessing.

Kernel-Level Threads (KLTs)

- Thread management is done by the kernel
- no thread management is done by the application



Implementing Threads in Kernel Space

- Kernel knows about and manages the threads
- No runtime is needed in each process
- Creating/destroying/(other thread related operations) a thread involves a system call
- In linux a `task_struct` and stacks are allocated and linked to the current process and its address space

Kernel-Level Threads (KLTs)

Advantages

- The kernel can simultaneously schedule multiple threads from the same process on multiple processors/cores/hw-threads
- If one thread in a process is blocked, the kernel can schedule another thread of the same process
- Kernel routines can be multithreaded

Disadvantages

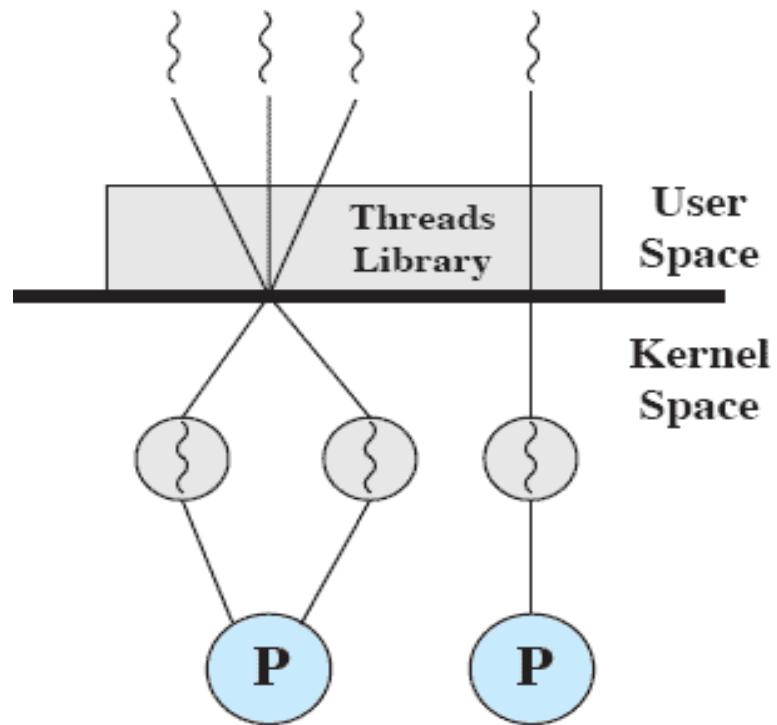
- The transfer of control from one thread to another within the same process requires a mode switch to the kernel



But note, if you want to implement interruption for user level thread scheduling, you have to issue and deliver a "signal(SIGVTALARM)" which is much more expensive.

Combined (Hybrid) Approach

- Thread creation is done completely in user space.
- Bulk of scheduling and synchronization of threads is by the application (i.e. user space).
- Multiple ULTs from a single application are mapped onto (smaller or equal) number of KLTs.
- Solaris is an example



Different Naming Conventions

- Thread Models are also referred to by their general ratio of user threads over kernels threads
- 1:1 : each user thread == kernel thread
- M:1 : user level thread mode
- M:N : hybrid model

PCB vs TCB

- Process Control Block **handles global process resources**
- Thread Control Block **handles thread execution resources**

Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

- pids vs. tid

```
frankeh@NYU2:~$ ps -edfmT | grep --color=no -e "^[f|U]" | head
UID      PID  SPID  PPID  C STIME TTY          TIME CMD
frankeh  1445      -  1431  0 11:46 ?        00:00:00 init --user
frankeh      -  1445      -  0 11:46 -        00:00:00 -
frankeh  1500      -      1  0 11:46 ?        00:00:00 /usr/bin/VBoxClient --clipboard
frankeh      -  1500      -  0 11:46 -        00:00:00 -
frankeh      -  1519      -  0 11:46 -        00:00:00 -
frankeh  1508      -      1  0 11:46 ?        00:00:00 /usr/bin/VBoxClient --display
frankeh      -  1508      -  0 11:46 -        00:00:00 -
frankeh      -  1523      -  0 11:46 -        00:00:00 -
frankeh  1512      -      1  0 11:46 ?        00:00:00 /usr/bin/VBoxClient --seamless
```

How are threads created ?

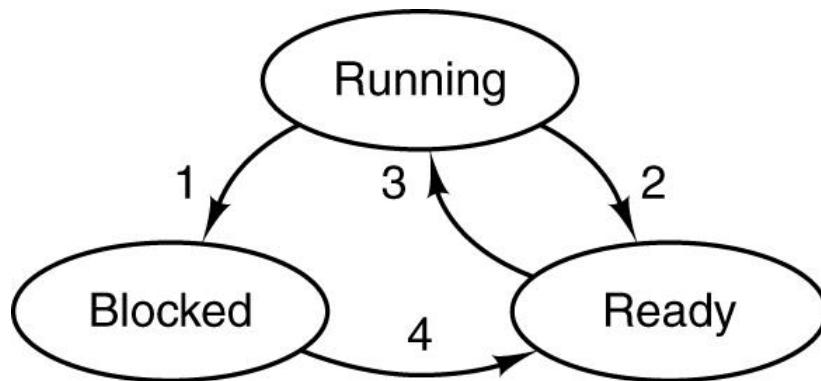
```
int pthread_create(pthread_t *thread,
                  const pthread_attr_t *attr,
                  void *(*start_routine) (void *) ,
                  void *arg );
```

Assuming 1:1 model:

- a) Allocates a new stack via malloc for the user level
- b) Calls clone() to create a new schedulable thread inside current process
(in linux it's a struct task obj)
- c) Sets the thread's stack pointer to (a)
- d) Sets the thread's instruction pointer to
(*start_routine) (arg)
- e) makes thread scheduable

Thread State Model:

- What changes to the state model in a kernel based thread model (1:1 or M:N) ?
- Really replace “process” with “thread” and you are basically there.
- Often we interchangeably use thread and process scheduling.



Thread

1. ~~Process~~ blocks for input
2. Scheduler picks another ~~process~~
3. Scheduler picks this ~~process~~
4. Input becomes available

#> sed -e “s/[P|p]rocess/thread/g”

Threads + fork()

- Linux threads are not replicated with a fork()
only forking thread remains

```
#include <pthread.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

void* thfunc(void* arg)
{
    sleep(2);
    while (1) {
        printf("%4d %lx\n",getpid(), pthread_self());
        sleep(1);
    }
    return NULL;
}

int main()
{
    pthread_t th1, th2;
    pthread_create(&th1,NULL,thfunc,NULL);
    pthread_create(&th2,NULL,thfunc,NULL);
    sleep(1);
    int rc = fork();
    printf("%4d %lx %d\n",getpid(),pthread_self(), rc);
    sleep(10000);
    return 0;
}
```

```
1835 7f690ae14740 1838
1838 7f690ae14740 0
1835 7f690ae13700
1835 7f690a612700
1835 7f690ae13700
1835 7f690a612700
1835 7f690ae13700
1835 7f690a612700
1835 7f690ae13700
1835 7f690a612700
```

```
$ ps -edfmT
```

frankeh	1835	-	1798	0 09:06 pts/0	00:00:00 ./mthread
frankeh	-	1835	-	0 09:06 -	00:00:00 -
frankeh	-	1836	-	0 09:06 -	00:00:00 -
frankeh	-	1837	-	0 09:06 -	00:00:00 -
frankeh	1838	-	1835	0 09:06 pts/0	00:00:00 ./mthread
frankeh	-	1838	-	0 09:06 -	00:00:00 -
frankeh	1839	-	1824	0 09:06 pts/1	00:00:00 ps -edfmT

Threads + fork()

```
#include <pthread.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

void* thfunc(void* arg)
{
    int rc = fork();
    printf("%4d rc=%4d pts=%lx\n",getpid(),rc, pthread_self());
    sleep(2);
    while (1) {
        printf("%4d %lx\n",getpid(), pthread_self());
        sleep(1);
    }
    return NULL;
}

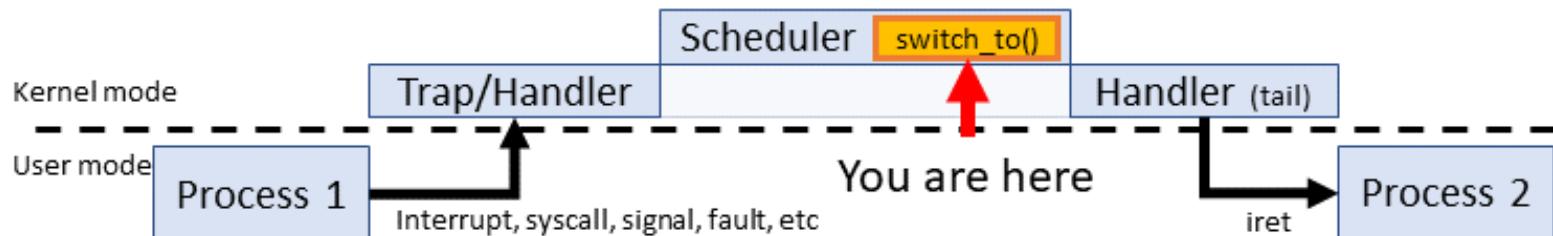
int main()
{
    pthread_t th1, th2;
    printf("main %4d %lx\n",getpid(), pthread_self());
    pthread_create(&th1,NULL,thfunc,(void*)0);
    pthread_create(&th2,NULL,thfunc,(void*)1);
    sleep(10000);
    return 0;
}
```

```
main 3481 7fe413831740
3481 rc=3484 pts=7fe413830700
3484 rc= 0 pts=7fe413830700
3481 rc=3485 pts=7fe41302f700
3485 rc= 0 pts=7fe41302f700
3484 7fe413830700
3485 7fe41302f700
3481 7fe413830700
3481 7fe41302f700
3484 7fe413830700
3481 7fe41302f700
3485 7fe41302f700
3481 7fe413830700
3484 7fe413830700
3485 7fe41302f700
3481 7fe413830700
3484 7fe413830700
3485 7fe41302f700
```

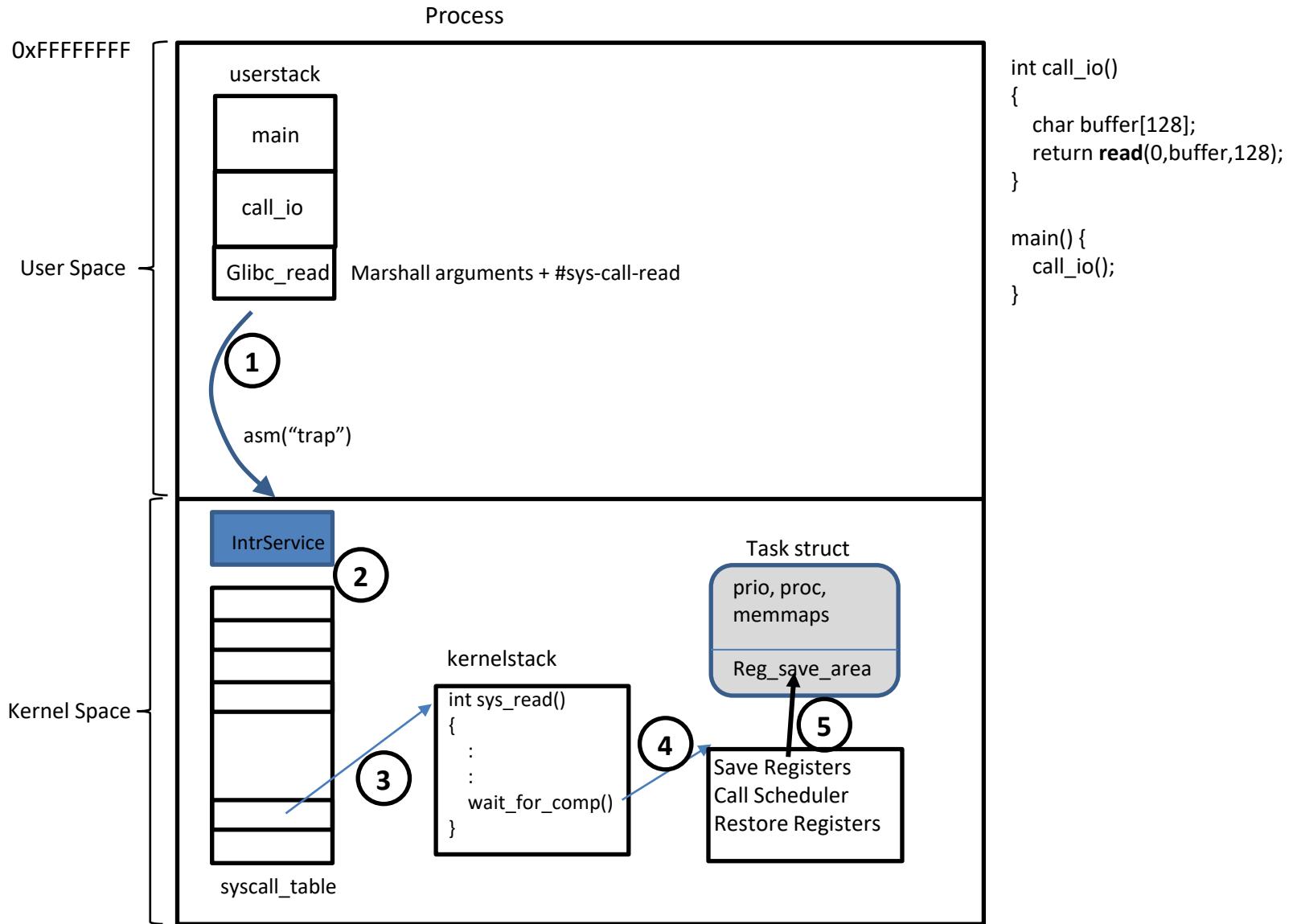
frankeh	3481	-	1972	0	14:39	pts/1	00:00:00	./mthread1
frankeh	-	3481	-	0	14:39	-	00:00:00	-
frankeh	-	3482	-	0	14:39	-	00:00:00	-
frankeh	-	3483	-	0	14:39	-	00:00:00	-
frankeh	3484	-	3481	0	14:39	pts/1	00:00:00	./mthread1
frankeh	-	3484	-	0	14:39	-	00:00:00	-
frankeh	3485	-	3481	0	14:39	pts/1	00:00:00	./mthread1
frankeh	-	3485	-	0	14:39	-	00:00:00	-

Context Switch

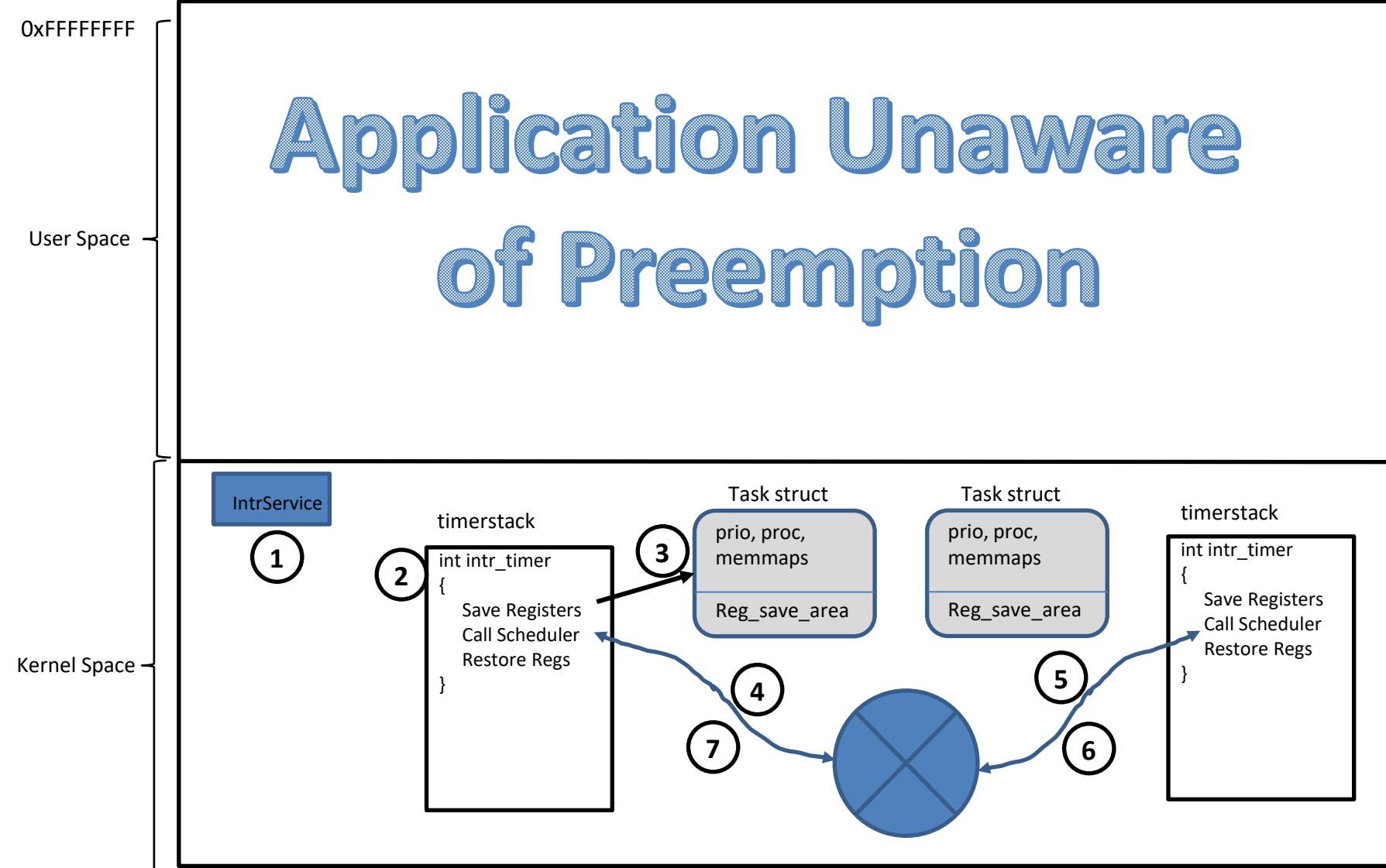
- Scenarios:
 - 1) Current process (or thread) blocks OR
 - 2) Preemption via interrupt
- Operation(s) to be done
 - Must release CPU resources (registers)
 - Requires storing “all” non privileged registers to the PCB or TCB save area
 - Tricky as you need registers to do this
 - Typically an architecture has a few privileged registers so the kernel can accomplish this
 - All written in assembler



CtxSwitch: Process Blocks



CtxSwitch: Preemption



Examples of low-level context switch routine for x86

Details not important,
appreciate the complexity
of the ASM code

```
/** include/asm-x86_64/system.h */

#define SAVE_CONTEXT    "pushfq ; pushq %%rbp ; movq %%rsi,%%rbp\n\t"
#define RESTORE_CONTEXT "movq %%rbp,%%rsi ; popq %%rbp ; popfq\n\t"
#define __EXTRA_CLOBBER
,"rcx","rbx","rdx","r8","r9","r10","r11","r12","r13","r14","r15"

#define switch_to(prev,next,last)
asm volatile(SAVE_CONTEXT
    "movq %%rsp,%[threadrsp](%[prev])\n\t" /* save RSP */
    "movq %P[threadrsp](%[next]),%%rsp\n\t" /* restore RSP */
    "call __switch_to\n\t"
    ".globl thread_return\n\t"
    "thread_return:\n\t"
    "movq %%gs,%P[pda_pcurren],%%rsi\n\t"
    "movq %P[thread_info](%%rsi),%%r8\n\t"
    "btr %[tif_fork],%P[ti_flags](%%r8)\n\t"
    "movq %%rax,%%rdi\n\t"
    "jc ret_from_fork\n\t"
RESTORE_CONTEXT
: "=a" (last)
: [next] "S" (next), [prev] "D" (prev),
[threadrsp] "i" (offsetof(struct task_struct, thread.rsp)),
[ti_flags] "i" (offsetof(struct thread_info, flags)),
[tif_fork] "i" (TIF_FORK),
[thread_info] "i" (offsetof(struct task_struct, thread_info))
[pda_pcurren] "i" (offsetof(struct x86_64_pda, pcurren))
: "memory", "cc" __EXTRA_CLOBBER)
```

```
/** arch/x86_64/kernel/process.c */

struct task_struct *__switch_to(struct task_struct *prev_p, struct task_struct *next_p)
{
    struct thread_struct *prev = &prev_p->thread,
                         *next = &next_p->thread;
    int cpu = smp_processor_id();
    struct tss_struct *tss = init_tss + cpu;

    unlazy_fpu(prev_p);

    tss->rsp0 = next->rsp0;

    asm volatile("movl %%es,%0" : "=m" (prev->es));
    if (unlikely(next->es | prev->es))
        loadsegment(es, next->es);

    asm volatile ("movl %%ds,%0" : "=m" (prev->ds));
    if (unlikely(next->ds | prev->ds))
        loadsegment(ds, next->ds);

    load_TLS(next, cpu);

    /* Switch FS and GS. */
    {
        unsigned fsindex;
        asm volatile("movl %%fs,%0" : "=g" (fsindex));

        if (unlikely(fsindex | next->fsindex | prev->fs))
            loadsegment(fs, next->fsindex);
        if (fsindex)
            prev->fs = 0;
    }
    /* when next process has a 64bit base use it */
    if (next->fs)
        wrmsrl(MSR_FS_BASE, next->fs);
    prev->fsindex = fsindex;
}

unsigned gsindex;
asm volatile("movl %%gs,%0" : "=g" (gsindex));
if (unlikely(gsindex | next->gsindex | prev->gs))
    load_gs_index(next->gsindex);
if (gsindex)
    prev->gs = 0;
}

if (next->gs)
    wrmsrl(MSR_KERNEL_GS_BASE, next->gs);
prev->gsindex = gsindex;

/* Switch the PDA context. */
prev->userrsp = read_pda(oldrsp);
write_pda(oldrsp, next->userrsp);
write_pda(pcurrent, next_p);
write_pda(kernelstack, (unsigned long)next_p->thread_info + THREAD_SIZE - PDA_STACKOFFSET);

/* Now maybe reload the debug registers */
if (unlikely(next->debugreg7)) {
    loaddebug(next, 0);
    loaddebug(next, 1);
    loaddebug(next, 2);
    loaddebug(next, 3);
    /* no 4 and 5 */
    loaddebug(next, 6);
    loaddebug(next, 7);
}

/* Handle the IO bitmap */
if (unlikely(prev->io_bitmap_ptr || next->io_bitmap_ptr)) {
    if (next->io_bitmap_ptr) {
        memcpy(tss->io_bitmap, next->io_bitmap_ptr, IO_BITMAP_BYTES);
        tss->io_bitmap_base = IO_BITMAP_OFFSET;
    } else {
        tss->io_bitmap_base = INVALID_IO_BITMAP_OFFSET;
    }
}

return prev_p;
```

Task switching in ARM

```
extern struct task_struct * __switch_to(struct task_struct *, struct thread_info *, struct thread_info *);

#define switch_to(prev,next,last)
do {
    __complete_pending_tlb();
    if (IS_ENABLED(CONFIG_CURRENT_POINTER_IN_TPIDRURO))
        __this_cpu_write(__entry_task, next);
    last = __switch_to(prev,task_thread_info(prev), task_thread_info(next));
} while (0)

#endif /* __ASM_ARM_SWITCH_TO_H */

        ENTRY(__switch_to)
        .fnstart
        .cantunwind
        add    ip, r1, #TI_CPU_SAVE
        stmia ip!, {r4 - r11}          @ Store most regs on stack
        str   sp, [ip], #4
        str   lr, [ip], #4
        mov   r5, r0
        add   r4, r2, #TI_CPU_SAVE
        ldr   r0, =thread_notify_head
        mov   r1, #THREAD_NOTIFY_SWITCH
        bl    atomic_notifier_call_chain
        mov   ip, r4
        mov   r0, r5
        ldmia ip!, {r4 - r11}          @ Load all regs saved previously
        ldr   sp, [ip]
        ldr   pc, [ip, #4]!
        .fnend
ENDPROC(__switch_to)
```

Conclusions

- Process/Threads are one the most central concepts in Operating Systems
- Process vs. Thread (understand difference)
 - Process is a resource container with at least one thread of execution
 - Thread is a unit of execution that lives in a process (no thread without a process)
 - Threads share the resources of the owning process.
- Multiprogramming vs multithreading



CSCI-GA.2250-001

Operating Systems

Process/Thread Scheduling

Hubertus Franke
frankeh@cs.nyu.edu

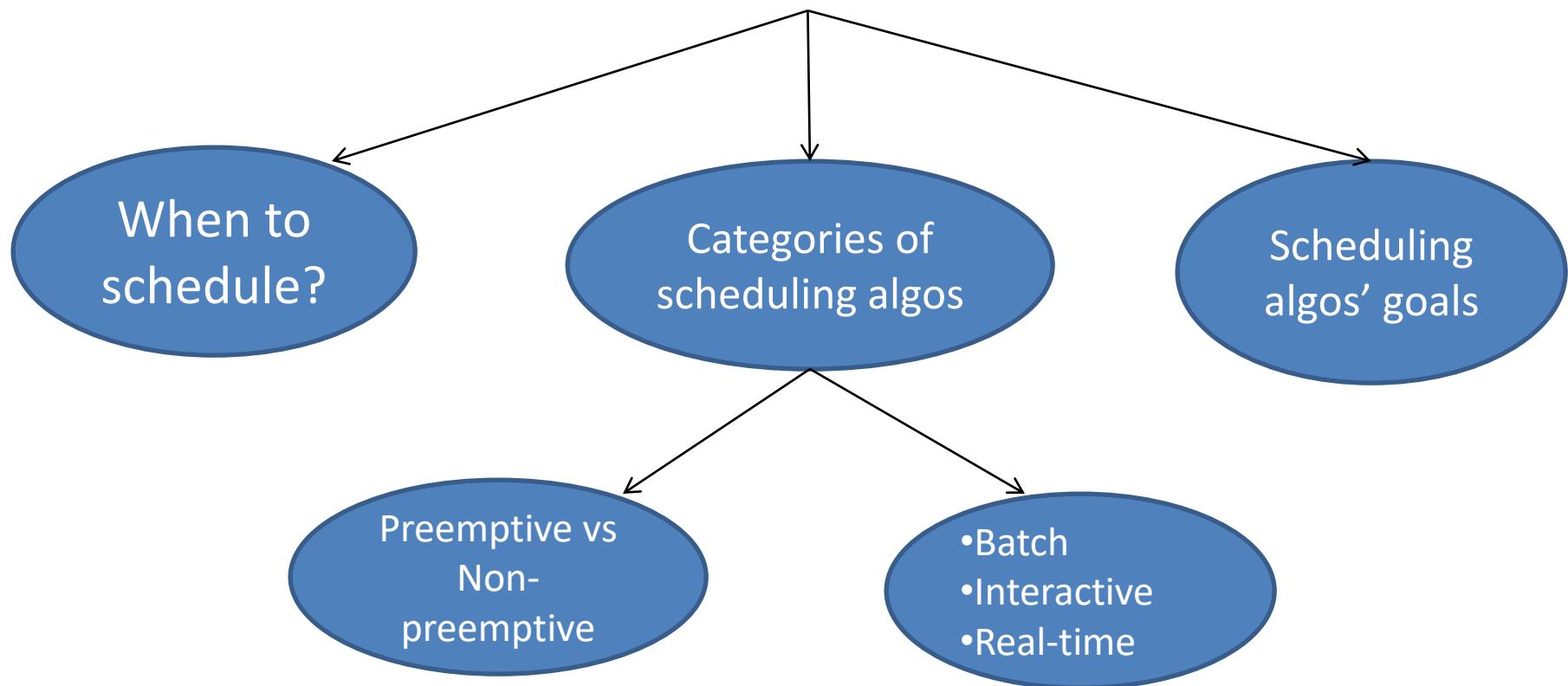


Scheduling

- Whether scheduling is based on processes or threads depends on whether the OS is multi-threading capable.
- In this deck/lab2 we use process as the assumption, just be aware that it applies to threads in a multi-threading capable OS.
- Given a group of ready processes or threads, which process/thread to run?

Scheduling

**Given a group of ready processes,
which process to run?**



When to Schedule?

- When a process is created
- When a process exits
- When a process blocks
- When an I/O interrupt occurs

Categories of Scheduling Algorithms

- Interactive
 - preemption is essential

preemption == a means for the OS to take away the CPU from a currently running process/thread
- Batch
 - No user impatiently waiting
 - mostly non-preemptive, or preemptive with long period for each process
- Real-time
 - deadlines

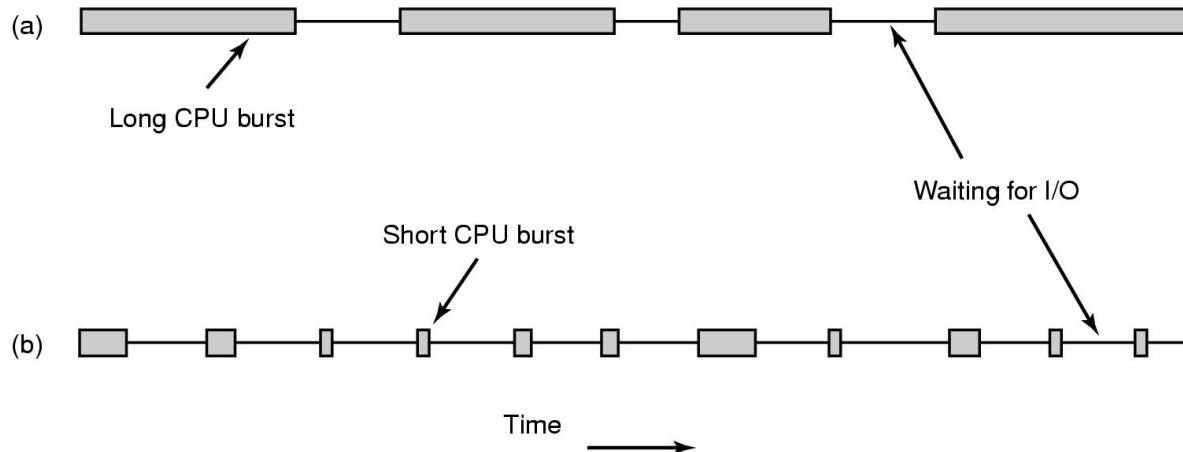
Scheduling Algorithm: Goals and Measures

- Turn Around Time (Batch)
- Throughput (e.g. jobs per second)
- Response Time (Interactive)
- Average wait times (how long waiting in readyQ)

CPU / IO Burst

- CPU Burst:
a sequence of instructions a process runs without requesting I/O.
Mostly dependent on the program behavior.
- IO "Burst":
time required to satisfy an IO request while the Process can not run any code.
Mostly dependent on system behavior
(how many other IOs, speed of device, etc.)

Scheduling – Process Behavior



CPU-Burst and IO-Burst are simple runtime behaviors of applications, they continuously change

Figure 2-38. Bursts of CPU usage alternate with periods of waiting for I/O. (a) A CPU-bound process. (b) An I/O-bound process.

Trick Question: what is the cpu burst of the following program?

```
int main() {  
    int i=0;  
    for (;;) i++;  
    return i;  
}
```

Scheduling Algorithms Goals

All systems

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

Batch systems

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

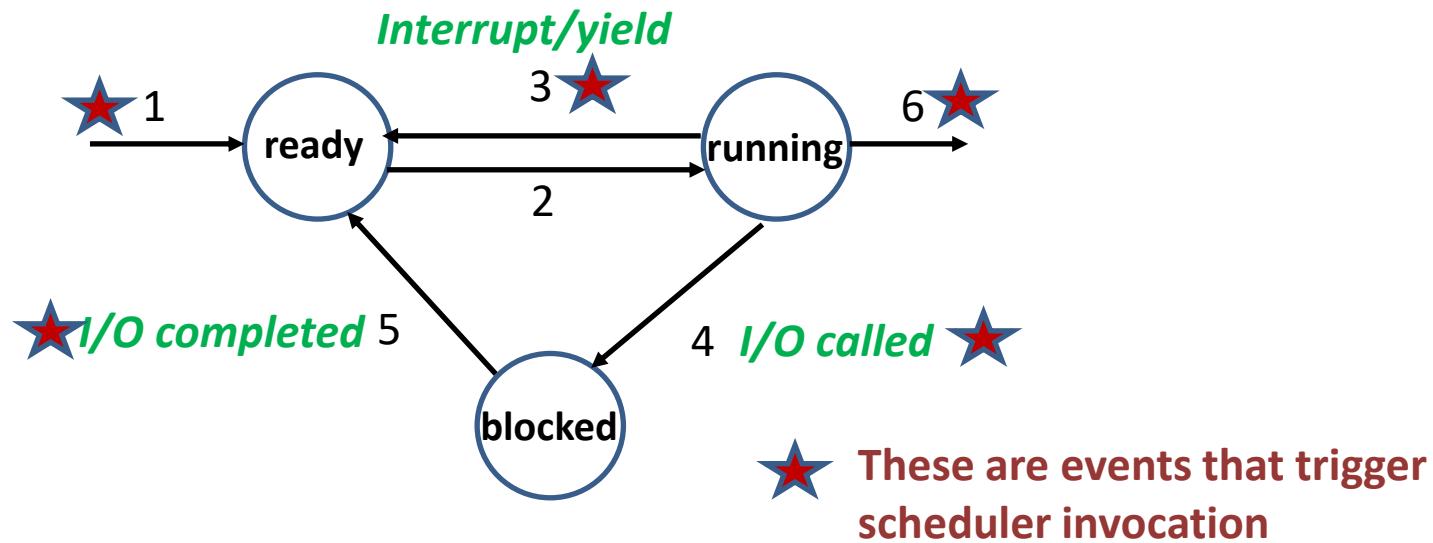
Interactive systems

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

Real-time systems

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

Almost ALL scheduling algorithms can be described by the following process state transition diagram or a derivative of it (we covered some more sophisticated one in prior lecture)



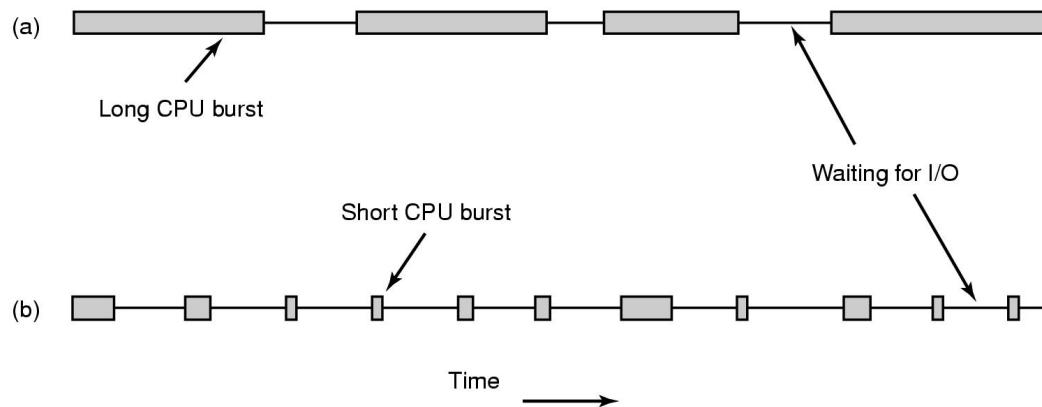
The OS Scheduler maintains a data structure called "RunQueue" or "ReadyQueue" where all processes that are <ready to run> are maintained (doesn't have to be strictly a queue, think "pool")

Scheduling : The Simplest Algorithm of them All

- Random() !!!
- Simply maintain a pool and pick one.
- Things to consider:
 - Starvation
 - Fairness

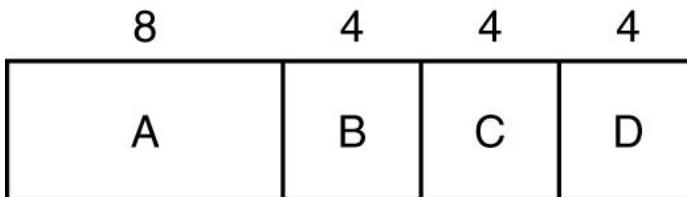
Scheduling in Batch Systems: First-Come First-Served (FIFO / FCFS)

- Non-preemptive (run till I/O or exit)
- Processes ordered as queue
- A new process is added to the end of the queue
- A blocked process that becomes ready added to the end of the queue
- Main disadv: Can hurt I/O bound processes or processes with frequent I/O



Scheduling in Batch Systems: Shortest Job First

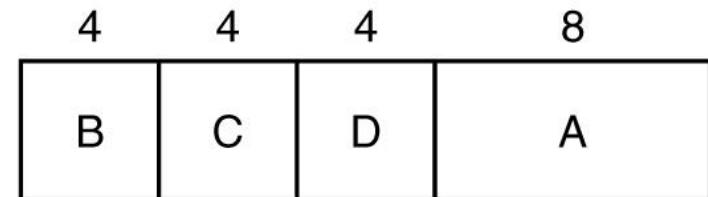
- Non-preemptive
- Assumes runtime is known in advance (☹)
- Is only optimal when all the jobs are available simultaneously



(a)

Run in original order

Avg-waittime:
 $(0+8+12+16) / 4 = 9$



(b)

Run in shortest job first

Avg-waittime:
 $(0+4+8+12) / 4 = 6$

Scheduling in Batch Systems: Shortest Remaining Time Next

- Scheduler always chooses the process whose remaining time is the shortest
- Runtime has to be known in advance (☹)
- Preemptive (see next slide) or non-preemptive (wait till block or done)
- Note: in lab2 we are using the non-preemptive version of "shortest remaining time next" since we have preemption in other schedulers already.
- This typically reduces average turn-around time

Scheduling in Interactive Systems: Round-Robin

- Each process is assigned a time interval referred to as **quantum**
- After this quantum, the CPU is given to another process (i.e. CPU is removed from the process/thread aka **preemption**)
- $RR = FIFO + \text{preemption}$
or ($FIFO = RR$ w huge quantum)
- What is the length of this quantum?
 - too short \rightarrow too many process context switches
 \rightarrow lower CPU efficiency
 - too long \rightarrow poor response to short interactive
 - quantum longer than typical CPU burst is good (why?)

Round-Robin Scheduling (cont)

- Promotes Fairness (due to preemption)

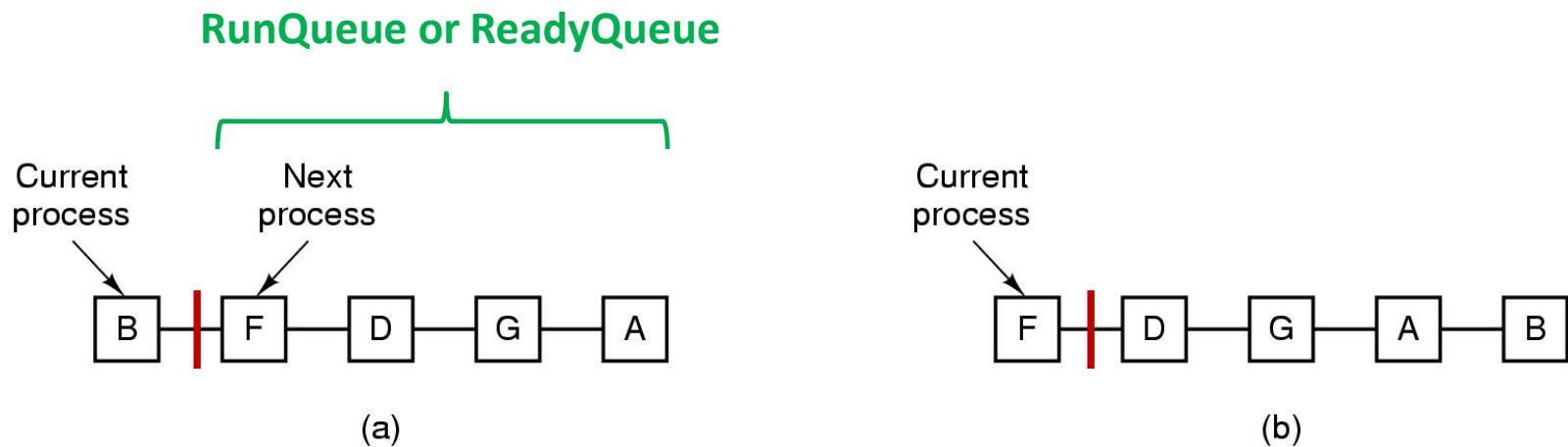


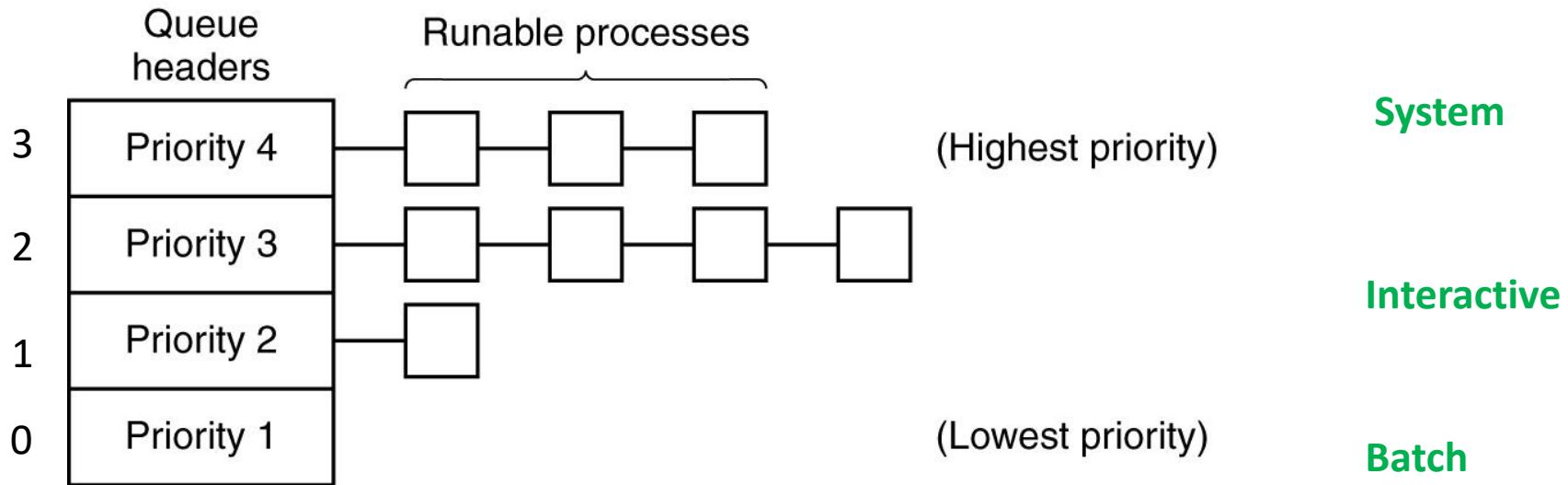
Figure 2-41. Round-robin scheduling.

(a) The list of runnable processes. (b) The list of runnable processes after B uses up its quantum.

Scheduling in Interactive Systems: Priority Scheduling

- Each process is assigned a priority
- runnable process with the highest priority is allowed to run
- Priorities are assigned statically or dynamically
- Must not allow a process to run forever
 - Can decrease the priority of the currently running process
 - Use time quantum for each process

Scheduling in Interactive Systems: Multiple Level Queuing (MLQ)



- `Process::static_priority`: it is a parameter of the process (non-changing unless requested through system call)
- Multiple levels of priority (MLQ) plus each level is run round-robin
- Issue: starvation if higher priorities have ALWAYS something to run

Multi-Level Feedback Queueing

[MLFQ aka priority decay scheduler]

- Multiple levels of priority MLQ , but
- `Process::dynamic_priority`: [0 .. $p \rightarrow \text{static_priority} - 1$]
(this changes constantly based on current state of process)
- If process has to be preempted, moves to (`dynamic_priority--`).
- When it reaches "-1", dynamic priority is reset to (`static_priority-1`)
 - This creates some issues when high prio is reset before other low prio is executing.
- When a process is made ready (from blocked):
it's dynamic priority is reset to (`static_priority-1`)
- What kind of process should be in bottom queue?
 - Best to assign a higher priority to IO-Bound tasks
 - Best to assign a lower priority to CPU-Bound tasks
- Discussion:

Lottery Scheduling

- Each runnable entity is given a certain number of tickets.
- The more tickets you have, the higher your odds of winning.
- Trade tickets?
- Problems?

Fair Share Scheduler

- Schedule not only based on individual processes, but process's owner.
- N users, each user may have different # of processes.
- Does this make sense on a PC?

Policy versus Mechanism

- Separate what is allowed to be done from how it is done
- Scheduling algorithm parameterized
 - mechanism in the kernel (e.g. quantum)
 - Context switches
- Parameters filled in by user processes
 - policy set by user process (priority, scheduling type)

Scheduling in Real-Time

- Process must respond to an event within a deadline
- Hard real-time vs soft real-time
- Periodic vs aperiodic events
- Processes must be schedulable
- Scheduling algorithms can be static or dynamic

Thread Scheduling

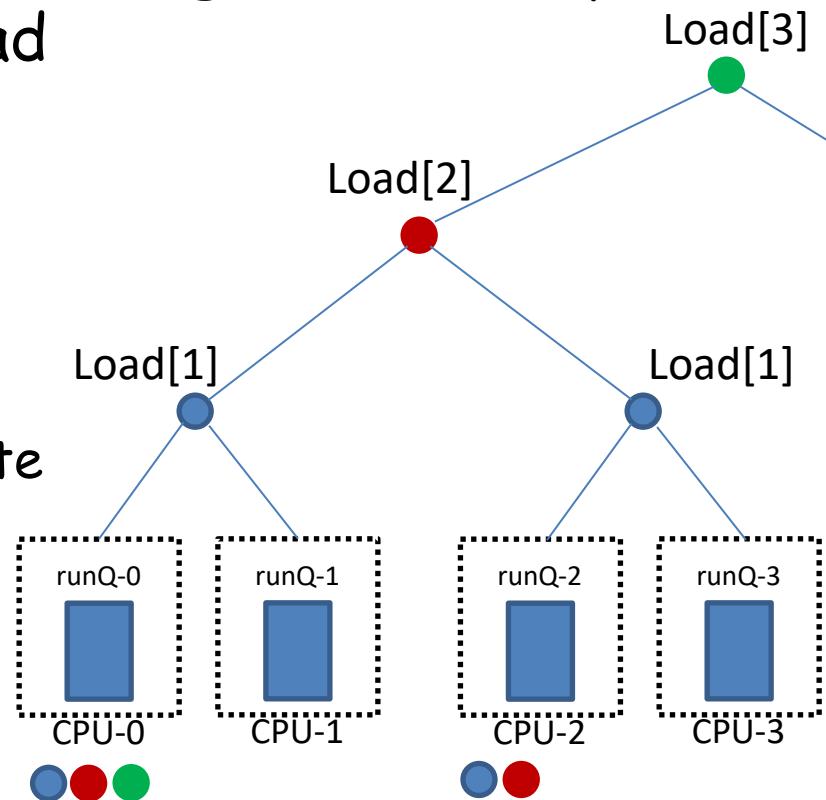
- Already covered in previous lecture:
- Two levels of parallelism: processes and threads within processes
- Kernel-based vs. user-space
(recall that kernel only schedules threads for kernel based thread model)

SMP Scheduling (SMP = Symmetric Multi Processors)

- One Scheduler Data Structure per CPU (hw-thread)
- Note that modern systems can have 100s of "cpus"
- Each CPU schedules for itself and is also triggered by its own timer interrupt (potential preemption)
→ This is known as the local scheduler
- New processes (fork) or threads (clone) are typically assigned the local CPU
- This "obviously" will lead to imbalances which needs to be dealt with
- Imbalances lead to idle resources and reduce fairness.

Load Balancing (LB)

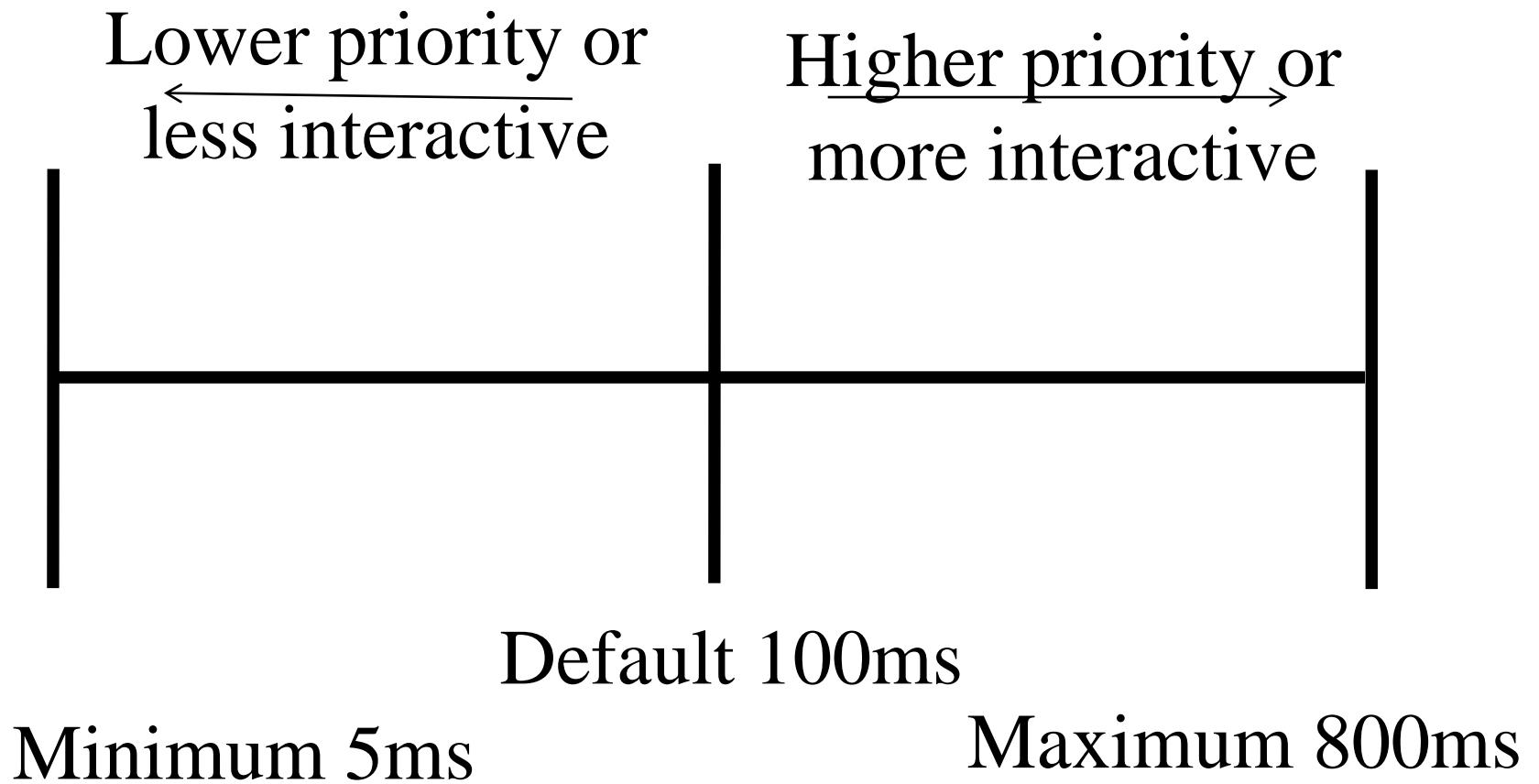
- Occasionally or when no process is runnable, scheduler[i] looks to steal work elsewhere
- Each scheduler maintains a load average and history to determine stability of its load
- LB typically done in a hierarchy
- Frequency of neighbor check:
 - Level in hierarchy ~ cost to migrate
 - Make “small” changes by pulling work from other cpu



Example Linux Scheduling

- Implementation has changed multiple times
- Dynamic Priority-Based Scheduling (classic)
- Two (static) Priority Ranges:
 - **Nice value:** -20 to +19, default 0.
Larger values are lower priority.
Nice value of -20 receives maximum timeslice, +19 minimum.
 - **Real-time priority:** By default values range 0 to 99.
Real-time processes have a higher priority than normal processes (nice).

Linux Timeslice



Scheduler Goals

- $O(1)$ scheduling - constant time
 - vs $O(N)$ which implies dependent on number of ready processes.
- SMP - each processor has its own locking and individual runqueue
- SMP Affinity. Only migrate process from one CPU to another if imbalanced runqueues.
- Good interactive performance
- Fairness
- Optimized for one or two processes but scales

Priority Arrays

- Each runqueue has two priority arrays:
 - (a) active and (b) expired
[note these are just pointers to arrays]
- Each priority array contains a bitmap
 - If bit- i is set in bitmap, it indicates there are processes runnable at given priority- i .
- Allows constant retrieval algorithm to find highest set bit (see next slide)

Runqueues

<kernel/sched.c> struct runqueue

activeQ - active priority array

expiredQ - expired priority array

```
Queue<Process*> *activeQ =
    calloc(sizeof(Queue<Process*>), maxprios);
Queue<Process*> *expiredQ =
    calloc(sizeof(Queue<Process*>), maxprios);
```

OR

```
Queue<Process*> *activeQ =
    new Queue<Process*> [ maxprios ];
Queue<Process*> *expiredQ =
    new Queue<Process*> [ maxprios ];
```

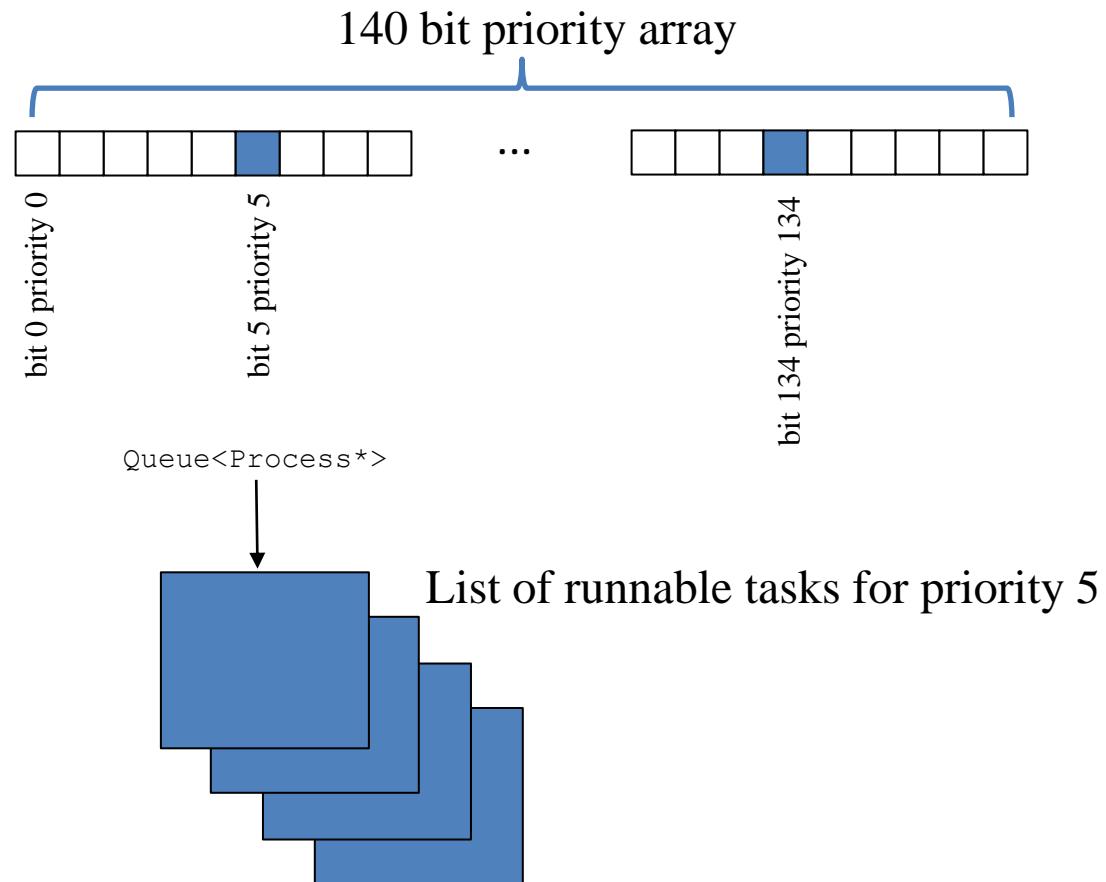
migration_thread

migration_queue

nr_iowait - number of tasks waiting on I/O

Scheduler Algorithm

- Maintain bitmap of state of the queues (in three 64-bit words)
- Identify highest prio queue that is not empty:
 - Use ffz() or ffnz() to find first bit that is set/notset supported by hardware instructions.
 - Only needs 3 64-bits to be examined to return correct priority (queue)
- Then dequeue_front from that queue.



Scheduler Operations

- add_to_rqueue()

```
if (dynamic prio < 0)
    reset and enter into expireQ
else
    add to activeQ
```

- get_from_rqueue()

```
if activeQ not empty
    pick activeQ[highest prio].front()
else
    swap(activeQ,expiredQ) and try again
    [ swapping means swapping the pointers to array
      not swapping the entire arrays .. Smart programming ]
```

Calculating Priority and Timeslice

`effective_prio()` returns task's dynamic priority.

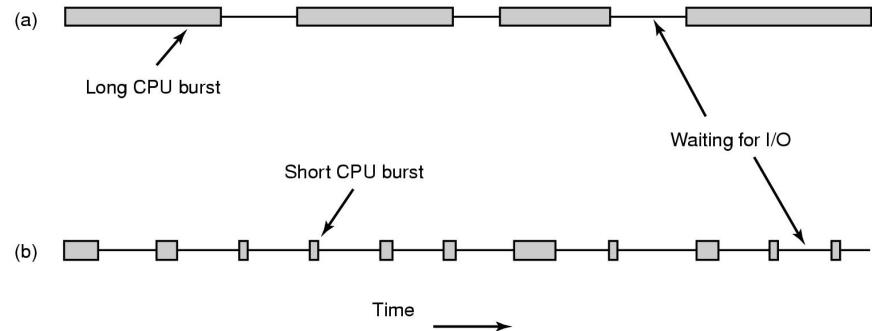
nice value + or - bonus in range -5 to +5

Interactivity measure by how much time a process sleeps. Indicates I/O activity.
`sleep_avg` incremented to `max_sleep_avg` (10 millisecs) every time it does I/O. If no I/O, decremented.

How to measure interactivity ?

How to determine interactivity?

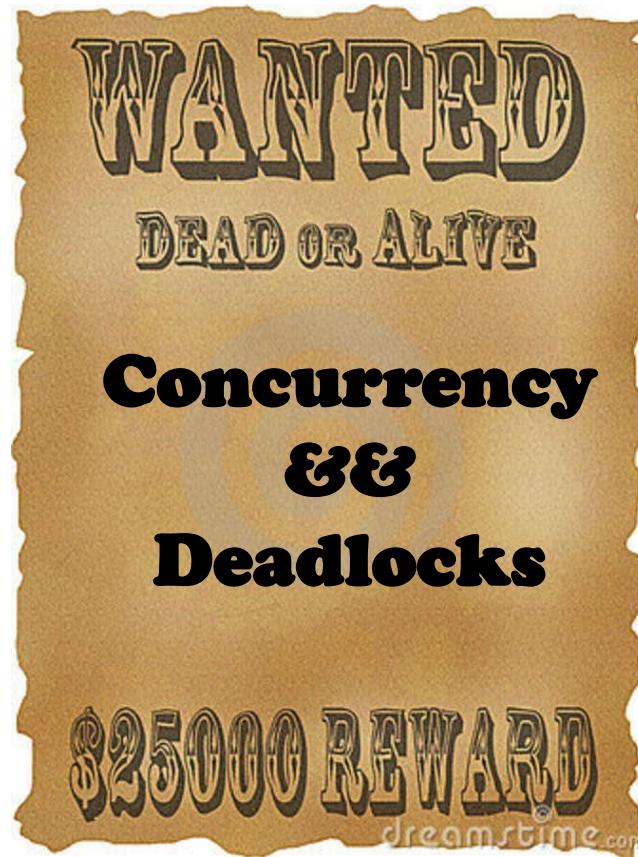
- $\text{Sum(cpu_burst)} / \text{wall time}$
- $\text{Sum(io_burst)} / \text{wall time}$
- $\text{Sum(cpu_burst)} / \text{Sum(io_burst)}$
- Note excludes queue-time in the ready queue
- Need to take “recency” into account:
- Mostly interested in recent history
- Use a weighting mechanism
- $\text{value}(t) = \text{measure}(t) * R + \text{value}(t-1) * (1-R)$
 $R \text{ in } (0 .. 1].$
- If $R == 1$ then only the last period counts. The closer R gets to 0 the more history counts





CSCI-GA.2250-001

Operating Systems

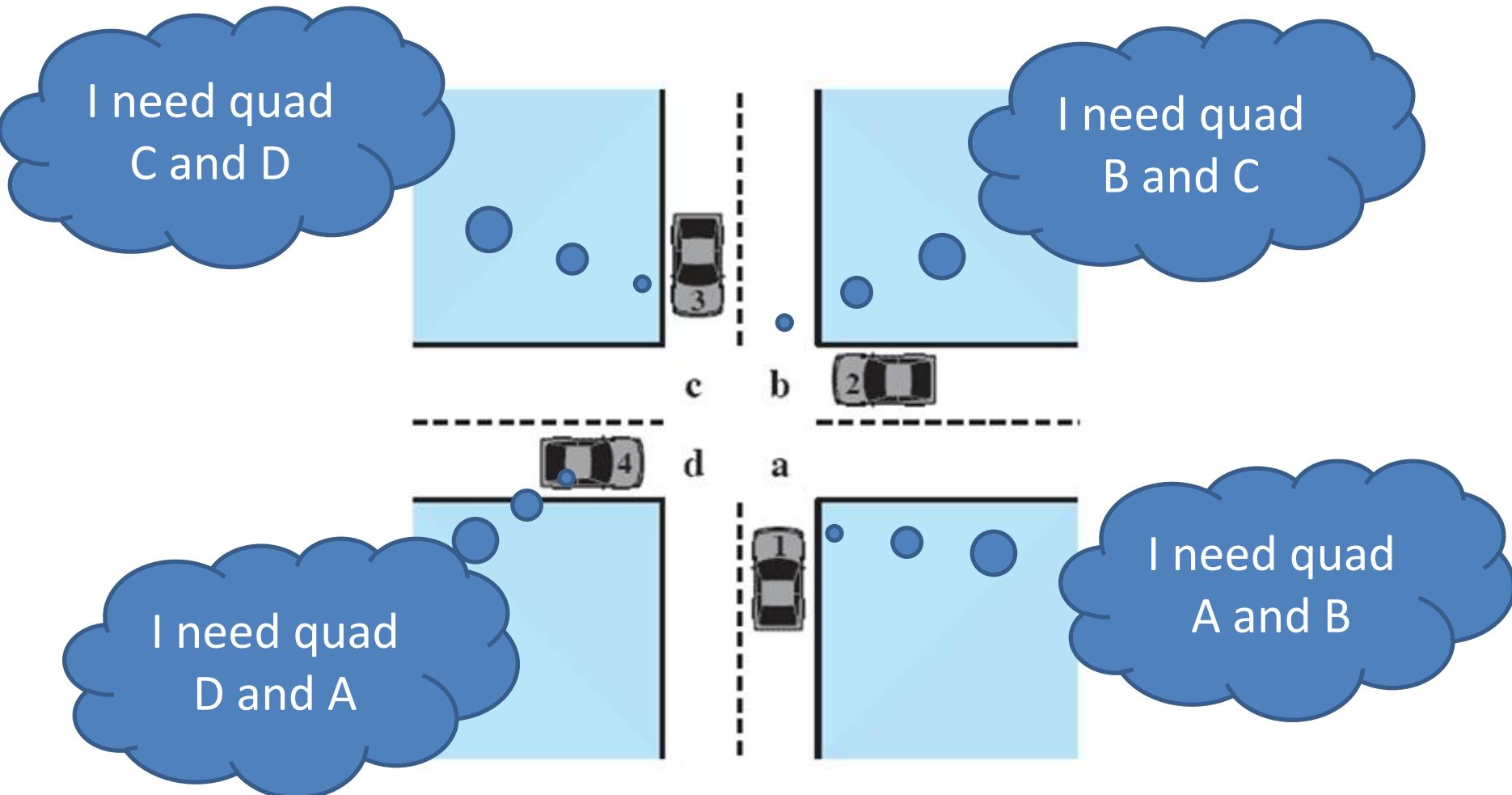


Hubertus Franke
frankeh@cs.nyu.edu

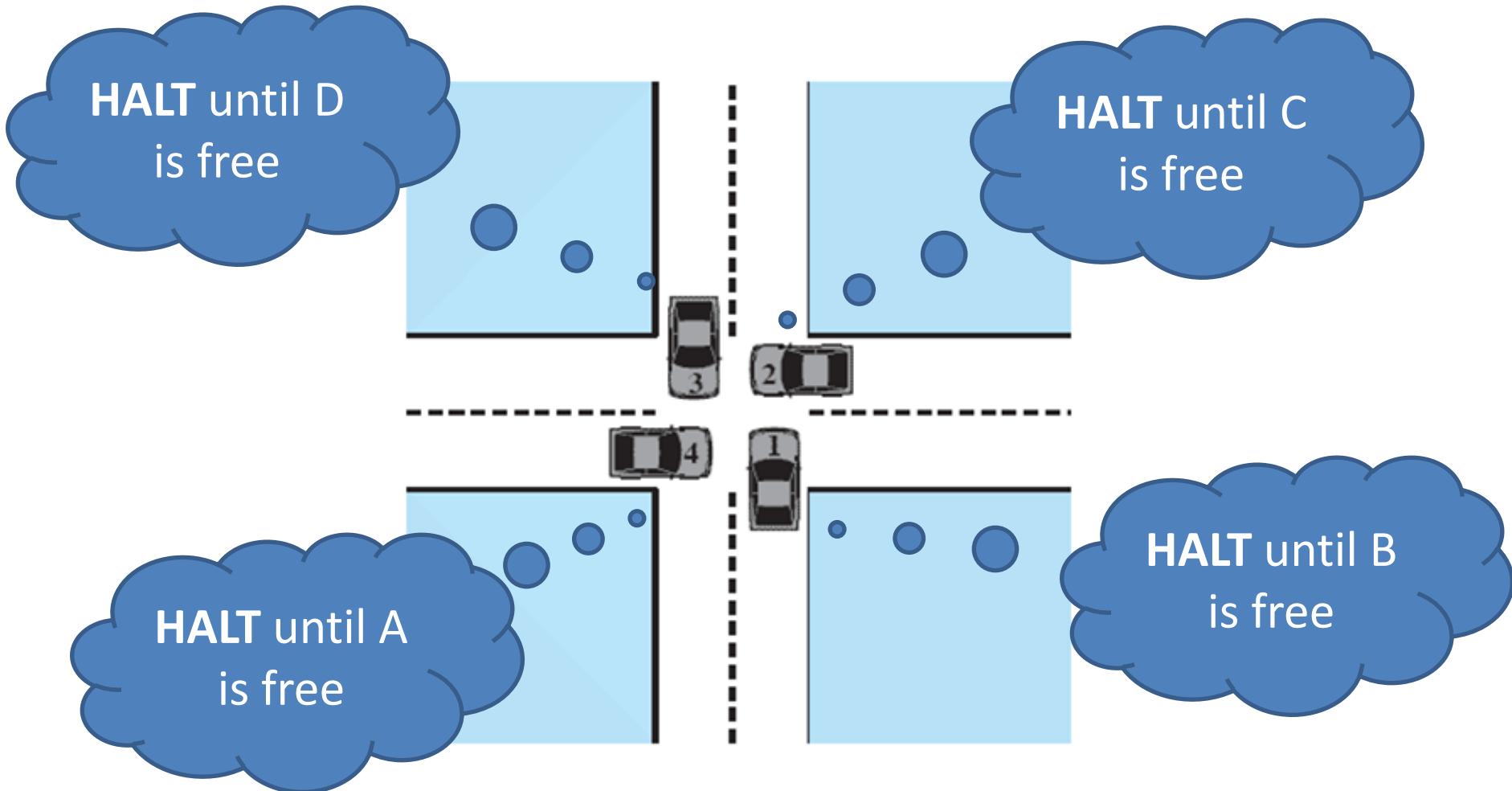
For illustration
purpose only



Potential Deadlock



Actual Deadlock



Reminder

- **Concurrency** is having multiple contexts of execution not necessarily running at the exact same time.
- **Parallelism** is having multiple contexts of execution running at the exact same time.

Inter Process Communication (IPC)

- **Processes** often need to work together or at the very least share resources.

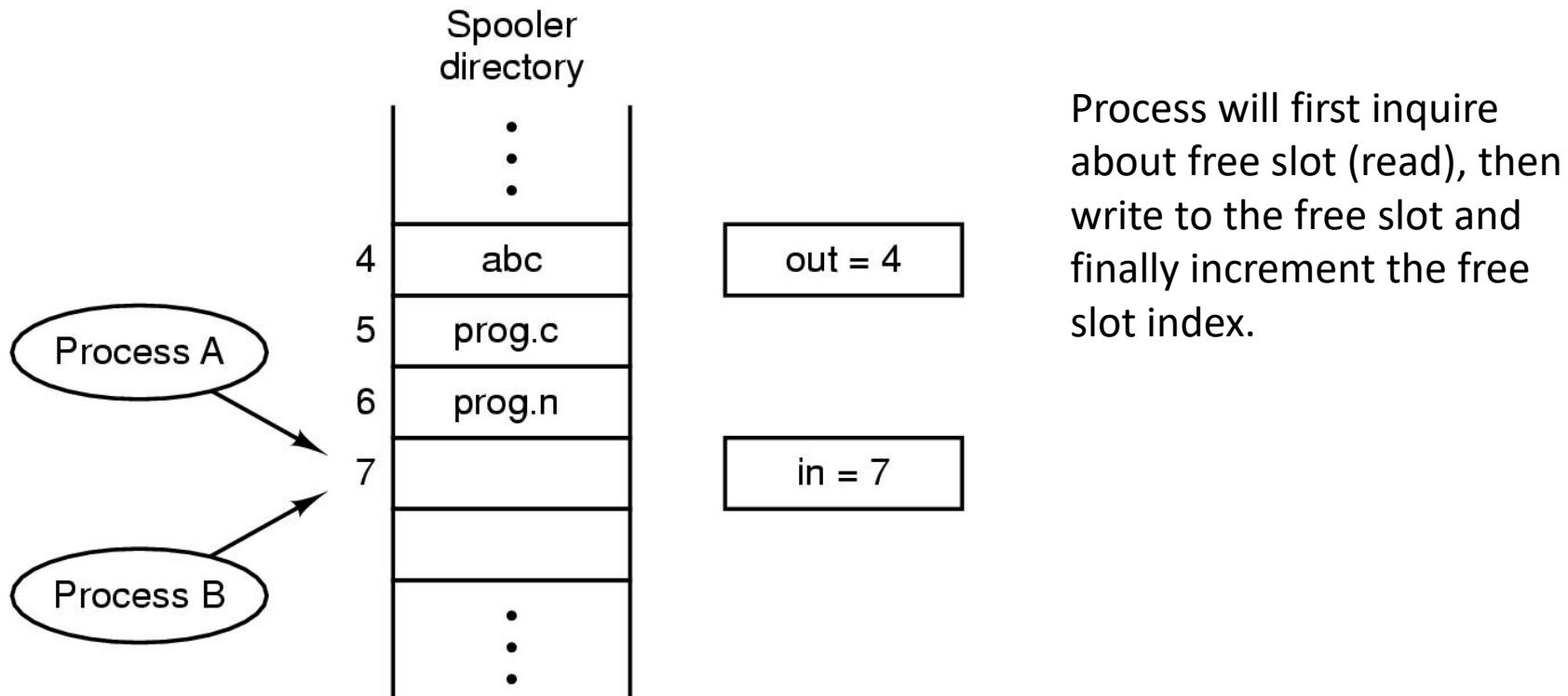
Issues:

- Send information
- Mitigate contentions over resources
- Synchronize dependencies

Inter-Process Communication

Race Conditions:

result depends on exact order of processes running



Two processes want to access shared memory at same time:
Read is typically not an issue but write or conditional execution **IS**

Intra Process Communication

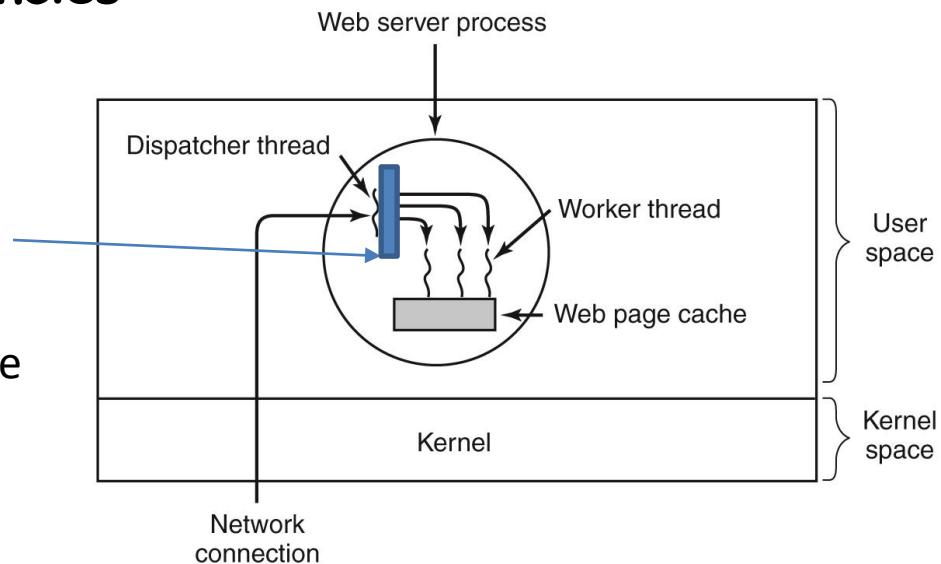
- **Threads** often need to work together and access resources (e.g. memory) in their common address space.

Same Issues:

- Send information
- Mitigate contentions over resources
- Synchronize dependencies

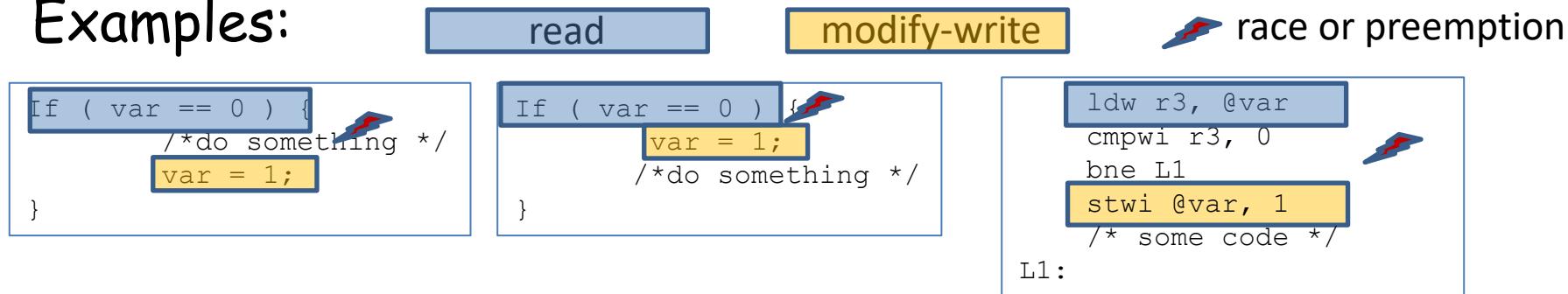
Example: Multi-threaded Webserver

- Dispatcher thread deposits work into queue of requests to be processed
- Worker threads will pick work from the queue



Read-Modify-Write Cycles

- For instance, you read a variable , make a decision and modify the variable
- Examples:



- These will have problems if the same code is executed by two threads concurrently.
- Read-Modify-Write cycles are typically an issue
- Consider the race and preemption case with 2 threads

Expectation is that code has "consistent view" at data

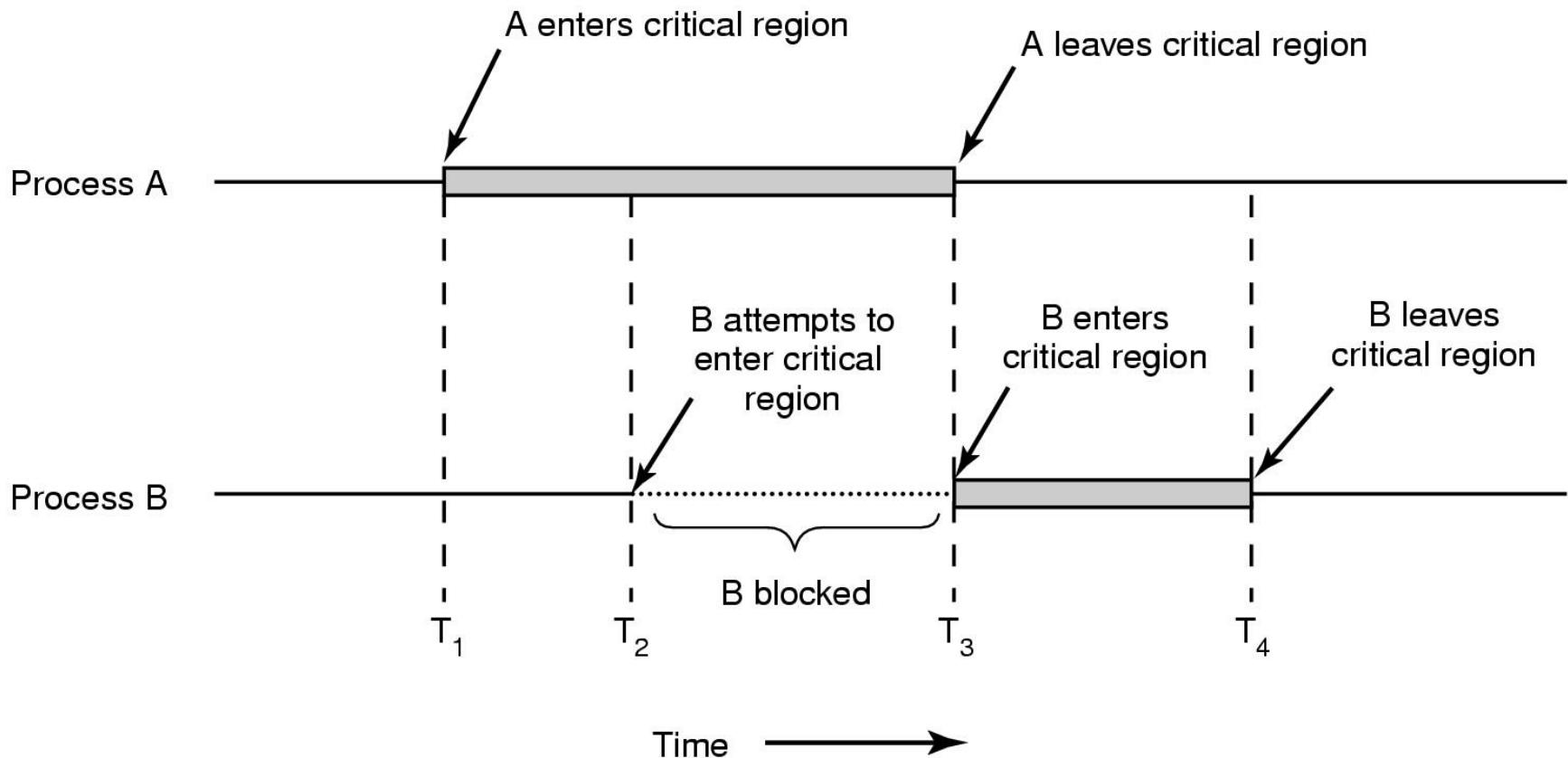
Critical Regions

Mutual Exclusion: Only one process (thread) at a time can access any shared variables, memory, or resources.

Four conditions to prevent errors:

1. No two processes simultaneously in critical region
2. No assumptions made about speeds or numbers of CPUs
3. No process running outside its critical region may block another process
4. No process must wait forever to enter its critical region

Critical Regions



Mutual exclusion using **critical regions**

Mutual Exclusion with Busy Waiting

Simplest solution?

- How about disabling interrupt?
 - User program has too much privilege.
 - Won't work in SMP (multi-core systems)

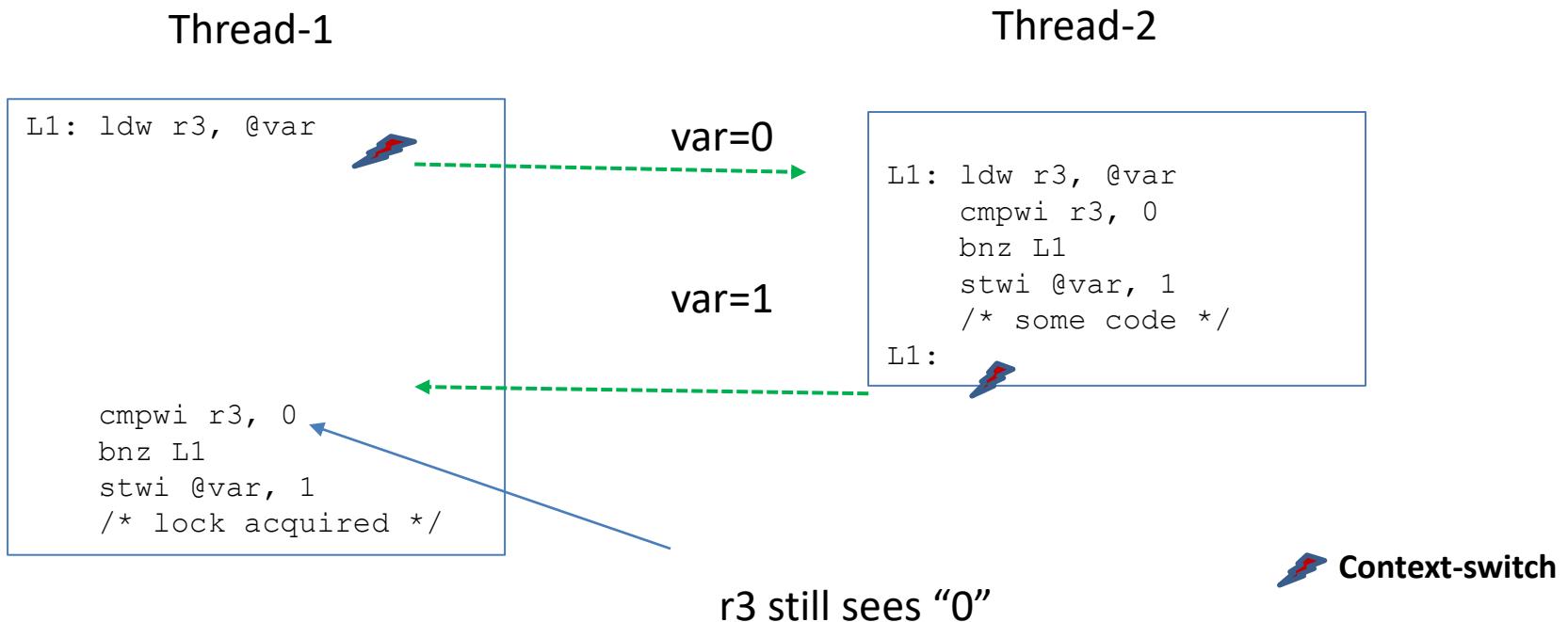
Lock variable?

- If the value of the lock variable is 0 then process sets it to "1" and enters. Other processes have to wait.
- What's the problem here?

--> READ-MODIFY-WRITE cycle

Read-Modify-Write Cycles During Locks

- Using a simple Lock Variable <var>



- Both Threads assume now they hold the lock

Mutual Exclusion with Busy Waiting

```
while (TRUE) {  
    while (turn != 0)      /* loop */ ;  
    critical_region();  
    turn = 1;  
    noncritical_region();  
}
```

(a)
Process 0

```
while (TRUE) {  
    while (turn != 1)      /* loop */ ;  
    critical_region();  
    turn = 0;  
    noncritical_region();  
}
```

(b)
Process 1

Proposed solution to critical region problem

But that's just a toy and in reality not useful

Mutual Exclusion with Busy Waiting

```
#define FALSE 0
#define TRUE 1
#define N      2           /* number of processes */

int turn;                  /* whose turn is it? */
int interested[N];         /* all values initially 0 (FALSE) */

void enter_region(int process); /* process is 0 or 1 */
{
    int other;               /* number of the other process */

    other = 1 - process;    /* the opposite of process */
    interested[process] = TRUE; /* show that you are interested */
    turn = process;          /* set flag */
    while (turn == process && interested[other] == TRUE) /* null statement */;
}

void leave_region(int process) /* process: who is leaving */
{
    interested[process] = FALSE; /* indicate departure from critical region */
}
```

Peterson's solution for achieving mutual exclusion

The TSL (Test and Set Lock) Instruction

- With a “little help” from Hardware

enter_region:

```
TSL REGISTER,LOCK  
CMP REGISTER,#0  
JNE enter_region  
RET
```

```
| copy lock to register and set lock to 1  
| was lock zero?  
| if it was nonzero, lock was set, so loop  
| return to caller; critical region entered
```

leave_region:

```
MOVE LOCK,#0  
RET
```

```
| store a 0 in lock  
| return to caller
```

Entering and leaving a critical region using the TSL instruction.

The XCHG Instruction (other arch have cmp_and_swap)

```
enter_region:  
    MOVE REGISTER,#1  
    XCHG REGISTER,LOCK  
    CMP REGISTER,#0  
    JNE enter_region  
    RET  
  
        | put a 1 in the register  
        | swap the contents of the register and lock variable  
        | was lock zero?  
        | if it was non zero, lock was set, so loop  
        | return to caller; critical region entered  
  
leave_region:  
    MOVE LOCK,#0  
    RET  
        | store a 0 in lock  
        | return to caller
```

Figure 2-26. Entering and leaving a critical region using the XCHG instruction.

Load/Store Conditional

- Modern processors resolve cmp_and_swap through the cache coherency and instructions
- Reservation (~~ldwx~~) remembers ONE address on a CPU and verifies on (~~stwx~~) whether still held otherwise store will fail

```
L1: ldwx r3, @lockvar      // load r3 and set reservation register on CPU with &lockvar  
    add r3,r3,1           // r3 = r3+1          |  
    stwx r3, @lockvar     // store conditionally r3 back to lockvar if reservation is still held  
    bcond L1              // if store conditionally failed try again
```

- Reservation is lost on interrupts or if another CPU steals cacheline holding <lockvar> (see discussion in first lecture on cache coherency) or if another ldwx is issued.

Solutions so far

- Both TSL (xchg/cmpswp) and Peterson's solutions are correct, but they
 - rely on busy-waiting during "contention"
 - waste CPU cycles
 - Priority Inversion problem

Priority Inversion Problem

- Higher priority process can be prevented from entering a critical section (CS) because the lock variable is dependent on a lower priority process.
 - Priority P1 < Priority P2
 - P1 is in its CS but P1 is never scheduled. Since P2 is busy in busy-waiting using the CPU cycles

Lock/Mutex Implementation with semi busy waiting

mutex_lock:

```
TSL REGISTER,MUTEX  
CMP REGISTER,#0  
JZE ok  
CALL thread_yield  
JMP mutex_lock  
ok: RET
```

| copy mutex to register and set mutex to 1
| was mutex zero?
| if it was zero, mutex was unlocked, so return
| mutex is busy; schedule another thread
| try again
| return to caller; critical region entered

mutex_unlock:

```
MOVE MUTEX,#0  
RET
```

| store a 0 in mutex
| return to caller

Thread gives up the CPU and upon reschedule it will attempt again.

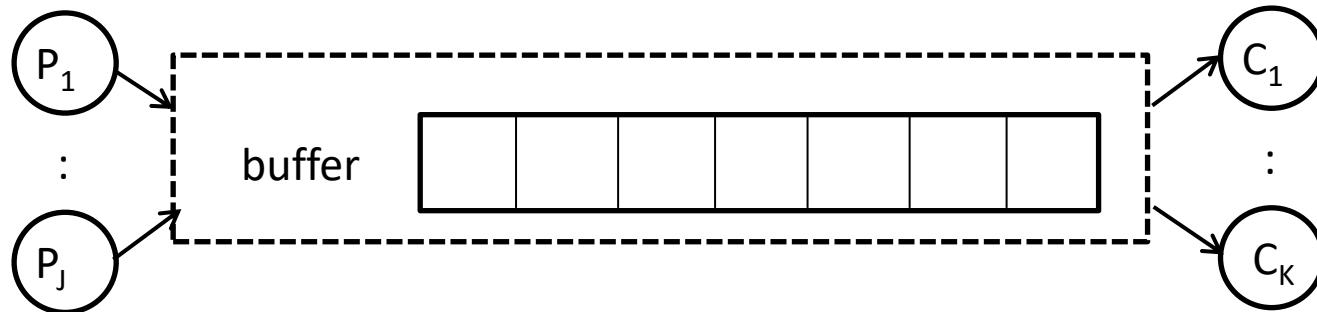
Figure 2-29. Implementation of mutex lock and mutex unlock.

Lock Contention

- Lock Contention arises when a process/thread attempts to acquire a lock and the lock is not available.
- This is a function of
 - frequency of attempts to acquire the lock
 - Lock hold time (time between acquisition and release)
 - Number of threads/processes acquiring a lock
- Lock contention is a function of lock hold time and lock acquisition frequency.
- If lock contention is low, TSL is an OK solution.
- The linux kernel uses it extensively for many locking scenarios.
- Alternative approaches will be discussed next.

Producer-Consumer problem

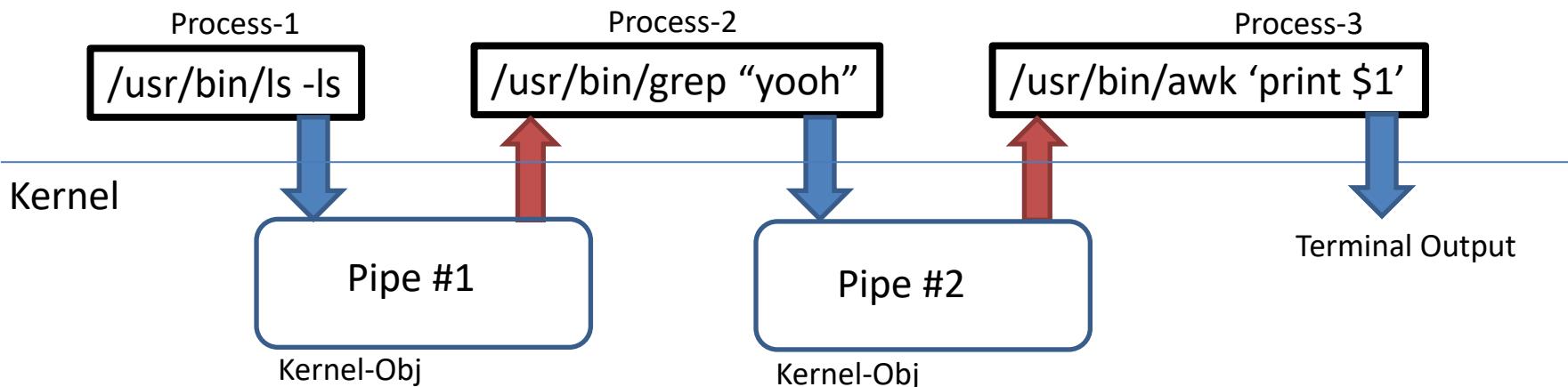
Buffer with N slots



- Producer: (1 .. j)
 - How do I know a slot is free ?
 - How do I know a slot became free ?
- Consumer: (1 .. k)
 - How do I know nothing is available?
 - How do I know something became available ?

How can mutual exclusion solutions be used in every day OSs?

- Example:
 - UNIX: `ls -ls | grep "yooh" | awk '{ print $1 }'`



- Responsibility of a Pipe
 - Provide Buffer to store data from stdout of Producer and release it to stdin of Consumer
 - Block Producer when the buffer is full (because consumer has not consumed data)
 - Block Consumer if no data in buffer when the consumer wants to read(stdin)
 - Unblock Producer when buffer space becomes free
 - Unblock Consumer when buffer data becomes available

Pipes

- Pipes not just for stdin and stdout
- Pipes can be created by applications
- All kind of usages for pipes.



- PipeBuffer typically has 16 write slots
- 4KB guaranteed to be atomic

```
frankeh@GCD:~$ cat /proc/sys/fs/pipe-max-size  
1048576
```

```
NAME  
    pipe, pipe2 - create pipe  
  
SYNOPSIS  
    #include <unistd.h>  
  
    int pipe(int pipefd[2]);  
  
    #define __GNU_SOURCE           /* See feature_test_macros(7) */  
    #include <fcntl.h>            /* Obtain O_* constant definitions  
*/  
    #include <unistd.h>  
  
    int pipe2(int pipefd[2], int flags);  
  
DESCRIPTION  
    pipe() creates a pipe, a unidirectional data channel that can be  
    used for interprocess communication. The array pipefd is used to  
    return two file descriptors referring to the ends of the pipe.  
    pipefd[0] refers to the read end of the pipe. pipefd[1] refers  
    to the write end of the pipe. Data written to the write end of  
    the pipe is buffered by the kernel until it is read from the read  
    end of the pipe. For further details, see pipe(7).
```

```
#define N 100  
int count = 0;
```

```
void producer(void)  
{  
    int item;  
  
    /* loop forever */ {  
        item = produce_item();  
        if (count == N) sleep();  
        insert_item(item);  
        count = count + 1;  
        if (count == 1) wakeup(consumer);  
    }  
}
```

```
void consumer(void)  
{  
    int item;  
  
    /* loop forever */ {  
        if (count == 0) sleep();  
        item = remove_item();  
        count = count - 1;  
        if (count == N - 1) wakeup(producer); /* was buffer full? */  
        consume_item(item);  
    }  
}
```

```
/* number of slots in the buffer */  
/* number of items in the buffer */
```

```
/* repeat forever */  
/* generate next item */  
/* if buffer is full, go to sleep */  
/* put item in buffer */  
/* increment count of items in buffer */  
/* was buffer empty? */
```

*Not sure why the book put a while loop here,
we just assume that these function
produce/consume one elem*

```
/* repeat forever */  
/* if buffer is empty, got to sleep */  
/* take item out of buffer */  
/* decrement count of items in buffer */  
/* was buffer full? */  
/* print item */
```

```
#define N 100
int count = 0;
/* number of slots in the buffer */
/* number of items in the buffer */

void producer(void)
{
    int item;

    /* repeat forever */
    /* generate next item */
    /* if buffer is full, go to sleep */
    /* put item in buffer */
    /* increment count of items in buffer */
    /* was buffer empty? */

    item = produce_item();
    if (count == N) sleep();
    insert_item(item);
    count = count + 1;
    if (count == 1) wakeup(consumer);
}

void consumer(void)
{
    int item;

    /* repeat forever */
    /* if buffer is empty, got to sleep */
    /* take item out of buffer */
    /* decrement count of items in buffer */
    /* was buffer full? */
    /* print item */

    if (count == 0) sleep();
    item = remove_item();
    count = count - 1;
    if (count == N - 1) wakeup(producer); /* was buffer full? */
    consume_item(item);
}
```

Called after pre-emption
but before sleep in the consumer

Pre-emption!!!
Count == 0

Fatal Race Condition

1. Let count = 1
2. Consumer begins loop, decrements count == 0
3. Consumer returns to loop beginning and executes:
if(count == 0), then
pre-emption happens.
4. Producer gets to run, executes
 $\text{count} = \text{count} + 1;$
if(count == 1) and calls wakeup(consumer)
5. Pre-emption Consumer calls sleep(consumer)

Requirements

- Need a mechanism that allows synchronization between processes/threads on the base of shared resource.
- Synchronization implies interaction with scheduling sub-system.
- Led to the innovation of semaphores.

Semaphore Data Structure

Dijkstra 1965

```
class Semaphore
{
    int value;           // counter
    Queue<Thread*> waiting; // queue of threads
                           // waiting on this sema
    void Init(int v);   // initialization
    void P();            // down(), wait()
    void V();            // up(), signal ()
}
```

Initially developed as a “construct” to ease programming.

This version is based on multi-threaded capable OS or thread library
Older version used Process as the object , same principle

Semaphore implementations

```
void Semaphore::Init(int v)
{
    value = v;
    Queue<Thread*>.init(); // empty queue
}
```

Semaphore implementations

```
void Semaphore::P() // or wait() or down()
{
    value = value - 1;
    if (value < 0)
    {
        waiting.add(current_thread);
        current_thread.status = blocked;
        schedule(); // forces wait, thread blocked
    }
}
```

AKA “acquiring or grabbing the semaphore”
think of it as obtaining access rights

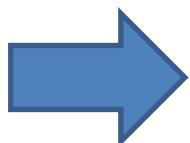
Semaphore implementations

```
void Semaphore::V() //      or      signal()      or up()
{
    value = value + 1;
    if (value <= 0)
    {
        Thread *thd = waiting.getNextThread();
        scheduler->add(thd); // make it scheduable
    }
}
```

AKA “releasing the semaphore”

Semaphore Solution

How do P() and V() avoid the race condition?



P() and V() must be **atomic**.

e.g.

- First line of P() & V() can disable interrupts.
- Last line of P() & V() re-enables interrupts.

However: disabling interrupts only works on single CPU systems

Atomic lock variable an option on entry and exit,

but must release the lock as part of call to schedule() in P()

or must reacquire the lock as part of call to schedule() in V()

Semaphore Data Structure (with atomicity)

```
class Semaphore
{
    int lockvar;           // to guarantee atomicity
    int value;             // counter
    Queue<Thread*> waiting; // queue of threads
                           // waiting on this sema
    void Init(int v);     // initialization
    void P();              // down(), wait()
    void V();              // up(), signal ()
}
```

Semaphore implementations

```
void Semaphore::P() // or    wait()   or down()
{
    lock(&lockvar);
    value = value - 1;
    if (value < 0)
    {
        waiting.add(current_thread);
        current_thread.status = blocked;
        unlock(&lockvar);
        schedule(); // forces wait, thread block
    } else {
        unlock(&lockvar);
    }
}
```

Semaphore implementations

```
void Semaphore::V() //      or      signal()      or up()
{
    lock(&lockvar);
    value = value + 1;
    if (value <= 0)
    {
        Thread *thd = waiting.getNextThread();
        scheduler->add(thd); // make it scheduable
    }
    unlock(&lockvar);
}
```

Two kinds of semaphores

- **Mutex semaphores**
(or **binary** semaphores or **LOCK**):
for mutual exclusion problems:
value initialized to 1
- **Counting semaphores:**
for synchronization problems.
Value initialized to any value 0..N
Value shows available tokens to enter
or number of processes waiting when
negative.

Semaphore Solution To the Producer Consumer Problem

```
#define N <somenumber>
Semaphore empty = N;
Semaphore full = 0;
Semaphore mutex = 1;
T buffer[N];
int widx = 0, ridx = 0;
```

3 Semaphores required

LOCK {
signaling →

```
Producer(T item)
{
    P(&empty); // Lock
    P(&mutex); // Lock
    buffer[widx] = item;
    widx = (widx + 1) % N;
    V(&mutex); // Unlock
    V(&full);
}
```

```
Consumer(T &item)
{
    P(&full); // Lock
    P(&mutex); // Lock
    item = buffer[ridx];
    ridx = (ridx + 1) % N;
    V(&mutex); // Unlock
    V(&empty);
}
```

Semaphore Solution To the Producer Consumer Problem

```
#define N <somenumber>
Semaphore empty      = N;
Semaphore full       = 0;
Semaphore mutex_w   = 1;
Semaphore mutex_r   = 1;
T buffer[N];
int widx = 0, ridx = 0;
```

```
Producer(T item)
{
    P(&empty);
    P(&mutex_w); // Lock
    buffer[widx] = item;
    widx = (widx + 1) % N;
    V(&mutex_w); // Unlock
    V(&full);
}
```

4 Semaphores for lower lock contention

LOCK {
signaling →

```
Consumer(T &item)
{
    P(&full);
    P(&mutex_r); // Lock
    item = buffer[ridx];
    ridx = (ridx + 1) % N;
    V(&mutex_r); // Unlock
    V(&empty);
}
```

Semaphore Solution To the Producer Consumer Problem

```
#define N <somenumber>
Semaphore empty      = N;
Semaphore full       = 0;
Semaphore mutex_w   = 1;
Semaphore mutex_r   = 1;
T buffer[N];
int widx = 0, ridx = 0;
```

```
Producer(T item)
{
    P(&empty);
    P(&mutex_w); // Lock
    int wi = widx;
    widx = (widx + 1) % N;
    V(&mutex_w); // Unlock
    
    buffer[wi] = item;
    V(&full);
}
```

Nasty condition on wi

4 Semaphores for lower lock contention

This example doesn't work; too aggressive !!!!

```
Consumer(T &item)
{
    P(&full);
    P(&mutex_r); // Lock
    int ri = ridx;
    ridx = (ridx + 1) % N;
    V(&mutex_r); // Unlock
    item = buffer[ri];
    V(&empty);
}
```

LOCK {
signaling →

Shorter lock hold times by only protecting the indices not the buffer themselves

Mutexes in Pthreads

Thread call	Description
Pthread_mutex_init	Create a mutex
Pthread_mutex_destroy	Destroy an existing mutex
Pthread_mutex_lock	Acquire a lock or block
Pthread_mutex_trylock	Acquire a lock or fail
Pthread_mutex_unlock	Release a lock

Figure 2-30. Some of the Pthreads calls relating to mutexes.

Mutexes in Pthreads

Thread call	Description
Pthread_cond_init	Create a condition variable
Pthread_cond_destroy	Destroy a condition variable
Pthread_cond_wait	Block waiting for a signal
Pthread_cond_signal	Signal another thread and wake it up
Pthread_cond_broadcast	Signal multiple threads and wake all of them

Figure 2-31. Some of the Pthreads calls relating to condition variables.

Mutexes in Pthreads

```
#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp;
int buffer = 0;
/* how many numbers to produce */

/* buffer used between producer and consumer */
/* produce data */

void *producer(void *ptr)
{
    int i;

    for (i= 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
        buffer = i; /* put item in buffer */
        pthread_cond_signal(&condc); /* wake up consumer */
        pthread_mutex_unlock(&the_mutex);/* release access to buffer */
    }
    pthread_exit(0);
}

void *consumer(void *ptr) /* consume data */
{
    int i;

    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer == 0 ) pthread_cond_wait(&condc, &the_mutex);
        buffer = 0; /* take item out of buffer */
        pthread_cond_signal(&condp); /* wake up producer */
        pthread_mutex_unlock(&the_mutex);/* release access to buffer */
    }
    pthread_exit(0);
}

int main(int argc, char **argv)
{
    pthread_t pro, con;
    pthread_mutex_init(&the_mutex, 0);
    pthread_cond_init(&condc, 0);
    pthread_cond_init(&condp, 0);
    pthread_create(&con, 0, consumer, 0);
    pthread_create(&pro, 0, producer, 0);
    pthread_join(pro, 0);
    pthread_join(con, 0);
    pthread_cond_destroy(&condc);
    pthread_cond_destroy(&condp);
    pthread_mutex_destroy(&the_mutex);
}
```

Using threads to solve
the producer-consumer
problem.

Problems with semaphore?

- It can be difficult to write semaphore code (arguably)
- One has to be careful in code construction
- If a thread dies and it holds a semaphore, the implicit token is lost

Deadlock-free code

```
typedef int semaphore;  
    semaphore resource_1;  
    semaphore resource_2;  
  
void process_A(void) {  
    down(&resource_1);  
    down(&resource_2);  
    use_both_resources( );  
    up(&resource_2);  
    up(&resource_1);  
}  
  
void process_B(void) {  
    down(&resource_1);  
    down(&resource_2);  
    use_both_resources( );  
    up(&resource_2);  
    up(&resource_1);  
}
```

Code with potential deadlock

```
semaphore resource_1;  
semaphore resource_2;  
  
void process_A(void) {  
    down(&resource_1);  
    down(&resource_2);  
    use_both_resources( );  
    up(&resource_2);  
    up(&resource_1);  
}  
  
void process_B(void) {  
    down(&resource_2);  
    down(&resource_1);  
    use_both_resources( );  
    up(&resource_1);  
    up(&resource_2);  
}
```

Some Advice for locks

- **Always** acquire multiple locks in the same order
- Preferably release in reverse order as acquired: not required but good hygiene
 - lots of discussion on this, there are scenarios where that is not desired but highly optimized implementation*
- Example: SMP CPU scheduler where load balancing is required.

Example

```
doit()  
{  
    s1.P();  
    s2.P();  
    "do something awesome"  
    s1.V();  
    s2.V();  
}
```

```
doit()  
{  
    s1.P();  
    s2.P();  
    "do something awesome"  
    s2.V();  
    s1.V();  
}
```

```
doit()  
{  
    s1.P();  
    {  
        s2.P();  
        "do something awesome"  
        s2.V();  
    }  
    s1.V();  
}
```

OK

Better

Making nesting clearer

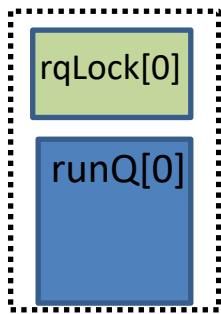
Example

```
doit()
{
    s1.P();
    do {
        s2.P();
        do {
            "do something awesome"
            break;
            "do something else awesome"
        } while(0);
        s2.V();
    } while (0);
    s1.V();
}
```

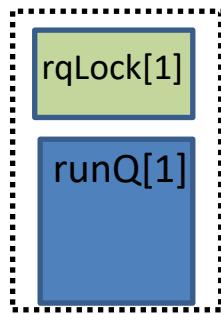
Making nesting clearer

Example: SMP Scheduler

- Each `cpu_i` has (`runQ[i]` and `rqLock[i]`)



CPU-0



CPU-1

```
Schedule(int i)
{
    lock(rqlock[i]);
    :
    { // load balance with j
        lock(rqlock[j]);
        ; // pull some threads
        unlock(rqlock[j]);
    }
    :
    unlock(rqlock[i]);
}
```

- `cpu0 (i=0, j=1) ; cpu1 (i=1, j=0)`
- Can lead to deadlocks → must use same order

Example SMP Scheduler

- So how do we get correct order?
- We force it !!
if necessary release owned lock first and reacquire.

```
Schedule(int i)
{
    lock(rqlock[i]);
    :
    { // load balance with j
        add_lock(i,j);
        ; // pull some threads
        unlock(rqlock[j]);
    }
    :
    unlock(rqlock[i]);
}
```

```
add_lock(int hlv, int alv)
{
    if ( hlv > alv ) {
        // first unlock hlv
        unlock(rqlock[hlv]);
        lock(rqlock[alv]);
        lock(rqlock[hlv]);
    } else {
        lock(rqlock[alv]);
    }
}
```

- We just need some order (could be "<" or based on addresses, it doesn't matter)

So which lock should I use

- Busy Lock vs. Semaphores ?
- If lock hold time is short and/or code is uninterruptible, then lock variables and busy waiting are OK (linux kernel uses it all the time)
- Otherwise use semaphores

Other Unix/Linux Mechanisms

- File based:

`flock()`

- System V semaphores

heavy weight as each call is a system call going into the kernel

`semget()`, `semop()` [P and V]

- Futexes

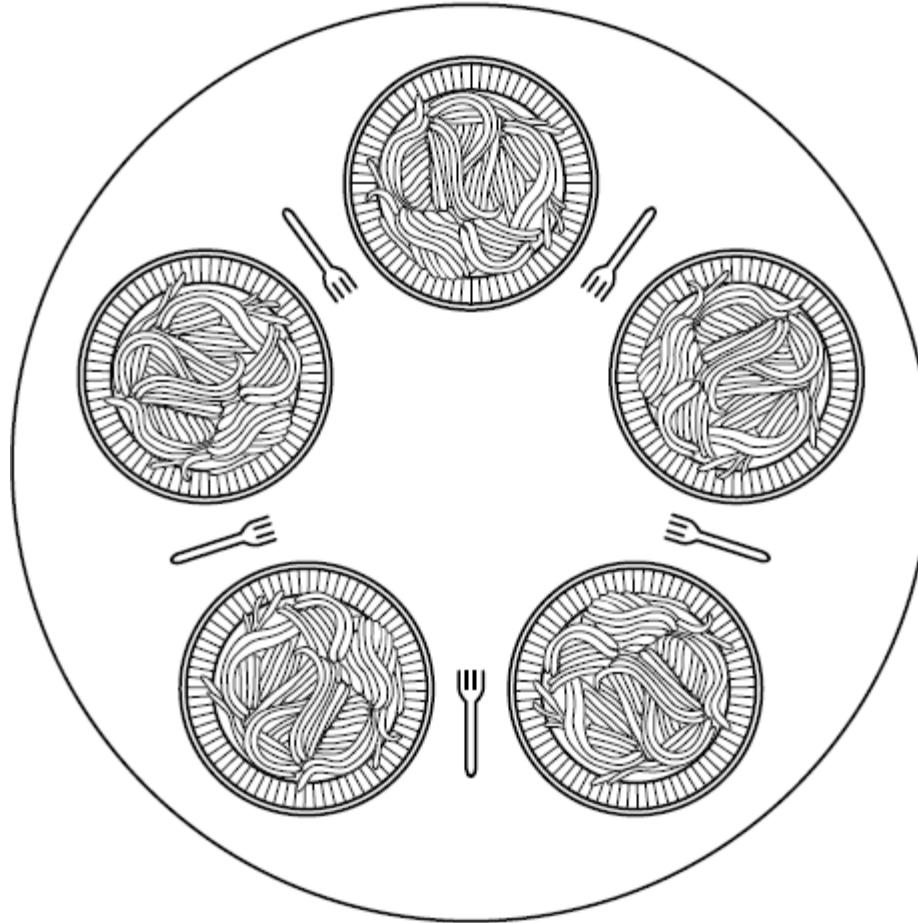
Lighter weight as uncontested cases are resolved done using `cmpxchg` in userspace and if race condition is recognized it goes to kernel

`futex()`

- Message queues

`mq_open()`, `mq_close()`, `mq_send()`, `mq_receive()`

The Dining Philosophers Problem (1)



Lunch time in the Philosophy Department.

The Dining Philosophers Problem (2)

```
#define N 5                                     /* number of philosophers */  
  
void philosopher(int i)                         /* i: philosopher number, from 0 to 4 */  
{  
    while (TRUE) {  
        think();  
        take_fork(i);  
        take_fork((i+1) % N);  
        eat();  
        put_fork(i);  
        put_fork((i+1) % N);  
    }  
}
```

A nonsolution to the dining philosophers problem.

The Dining Philosophers Problem (3)

```
#define N      5          /* number of philosophers */
#define LEFT    (i+N-1)%N  /* number of i's left neighbor */
#define RIGHT   (i+1)%N   /* number of i's right neighbor */
#define THINKING 0         /* philosopher is thinking */
#define HUNGRY   1         /* philosopher is trying to get forks */
#define EATING   2         /* philosopher is eating */
typedef int semaphore;
int state[N];
semaphore mutex = 1;
semaphore s[N];

void philosopher(int i)
{
    while (TRUE) {
        think();
        take_forks(i);
        eat();
        put_forks(i);
    }
}
```

/* semaphores are a special kind of int */
/* array to keep track of everyone's state */
/* mutual exclusion for critical regions */
/* one semaphore per philosopher */
/* i: philosopher number, from 0 to N-1 */
/* repeat forever */
/* philosopher is thinking */
/* acquire two forks or block */
/* yum-yum, spaghetti */
/* put both forks back on table */

A solution to the dining philosophers problem.

The Dining Philosophers Problem (4)

```
~~~~~ put_forks(i); /* put both forks back on table */
}
}

void take_forks(int i) /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);
    state[i] = HUNGRY;
    test(i);
    up(&mutex);
    down(&s[i]);
}

void put_forks(i) /* i: philosopher number, from 0 to N-1 */
~~~~~
```

A solution to the dining philosophers problem.

The Dining Philosophers Problem (5)

```
}

void put_forks(i)                                /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);                               /* enter critical region */
    state[i] = THINKING;                      /* philosopher has finished eating */
    test(LEFT);                                /* see if left neighbor can now eat */
    test(RIGHT);                               /* see if right neighbor can now eat */
    up(&mutex);                               /* exit critical region */
}

void test(i) /* i: philosopher number, from 0 to N-1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

A solution to the dining philosophers problem.

The Multiple Readers and Writer Problem (1)

It should be allowed for multiple readers to be active on a data as long as the data is not modified

```
typedef int semaphore;
semaphore mutex = 1;
semaphore db = 1;
int rc = 0;

void reader(void)
{
    while (TRUE) {
        down(&mutex);
        rc = rc + 1;
        if (rc == 1) down(&db);
        up(&mutex);
        read_data_base();
        down(&mutex);
        rc = rc - 1;
        if (rc == 0) up(&db);
        up(&mutex);
        use_data_read();
    }
}

void writer(void)
```

```
/* use your imagination */
/* controls access to 'rc' */
/* controls access to the database */
/* # of processes reading or wanting to */

/* repeat forever */
/* get exclusive access to 'rc' */
/* one reader more now */
/* if this is the first reader ... */
/* release exclusive access to 'rc' */
/* access the data */
/* get exclusive access to 'rc' */
/* one reader fewer now */
/* if this is the last reader ... */
/* release exclusive access to 'rc' */
/* noncritical region */
```

A solution to the multiple readers and writer problem.

The Multiple Readers and Writer Problem (1)

Only one writer can be active but then no reader can be active
Writer starvation still possible

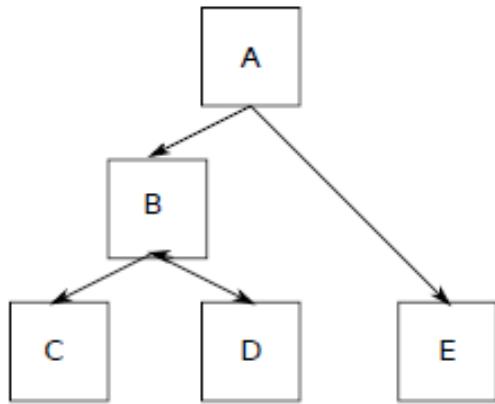
```
        use_data_read();      /* noncritical region */
    }
}

void writer(void)
{
    while (TRUE) {          /* repeat forever */
        think_up_data();    /* noncritical region */
        down(&db);          /* get exclusive access */
        write_data_base();   /* update the data */
        up(&db);            /* release exclusive access */
    }
}
```

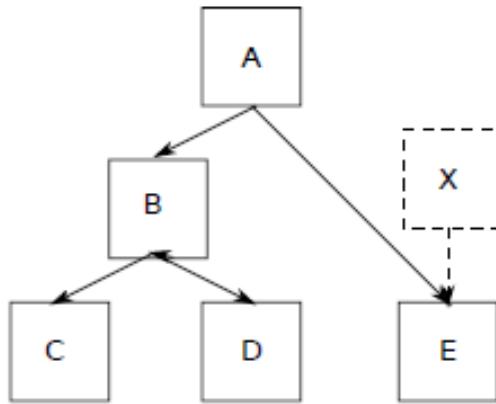
A solution to the multiple readers and writer problem.

Avoiding Locks: Read-Copy-Update (1)

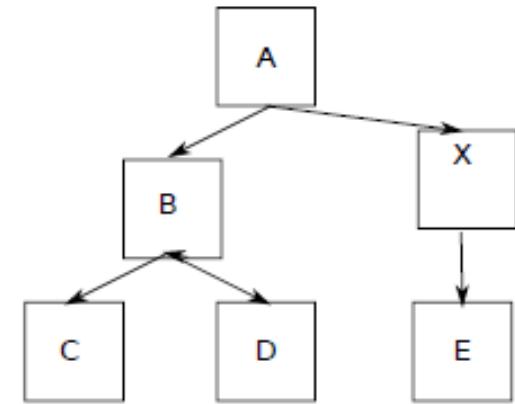
Adding a node:



(a) Original tree



(b) Initialize node X and connect E to X. Any readers in A and E are not affected.

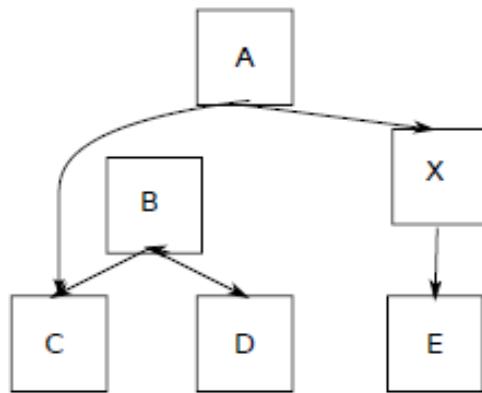


(c) When X is completely initialized, connect X to A. Readers currently in E will have read the old version, while readers in A will pick up the new version of the tree.

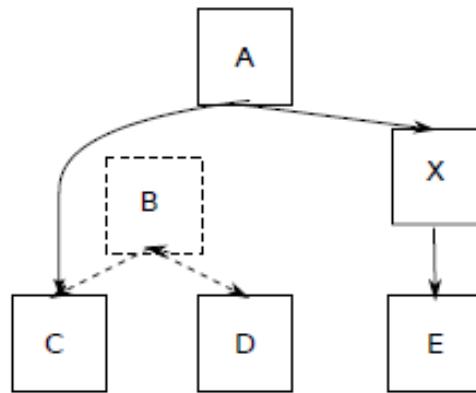
Read-Copy-Update: inserting a node in the tree and then removing a branch—all without locks

Avoiding Locks: Read-Copy-Update (2)

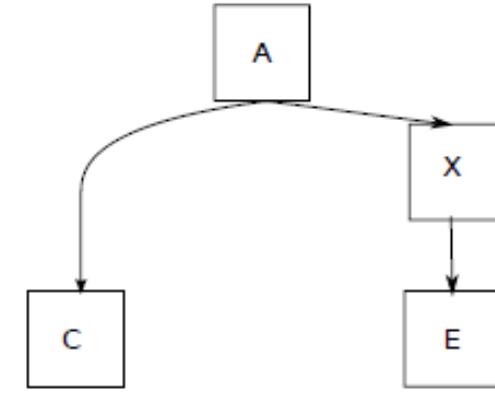
Removing nodes:



(d) Decouple B from A. Note that there may still be readers in B. All readers in B will see the old version of the tree, while all readers currently in A will see the new version.



(e) Wait until we are sure that all readers have left B and C. These nodes cannot be accessed by anymore.



(f) Now we can safely remove B and D

Read-Copy-Update: inserting a node in the tree and then removing a branch—all without locks

Monitors

- Hoare and Brinch Hansen proposed a higher-level synchronization primitive: **monitor**
- Only **ONE** thread allowed inside the Monitor
- **Compiler** achieves Mutual Exclusion
- Monitor is a **programming language** construct like a class or a for-loop.
- We still need a way to synchronize on events!
 - Condition variables: wait & signal
 - Not counters: signaling with no one waiting → event lost
 - Waiting on signal releases the monitor and wakeup reacquires it.

Monitor Example

```
monitor example
    integer i;
    condition c;

    procedure producer( );
    .
    .
    .
    end;

    procedure consumer( );
    .
    .
    .
    end;

end monitor;
```

Figure 2-33. A monitor.

Monitors

- What to do when a thread A in a monitor signals a thread B sleeping on a condition variable?
 - Ensure that signal is the last command in the monitor?
 - Block A while B runs?
 - Let A finish then run B?

Monitors

```
monitor ProducerConsumer
    condition full, empty;
    integer count;
    procedure insert(item: integer);
    begin
        if count = N then wait(full);
        insert_item(item);
        count := count + 1;
        if count = 1 then signal(empty)
    end;
    function remove: integer;
    begin
        if count = 0 then wait(empty);
        remove = remove_item;
        count := count - 1;
        if count = N - 1 then signal(full)
    end;
    count := 0;
end monitor;

procedure producer;
begin
    while true do
        begin
            item = produce_item;
            ProducerConsumer.insert(item)
        end
    end;
procedure consumer;
begin
    while true do
        begin
            item = ProducerConsumer.remove;
            consume_item(item)
        end
    end;

```

Figure 2-34. An outline of the producer-consumer problem with monitors.

Producer-Consumer Problem with Message Passing

```
#define N 100                                     /* number of slots in the buffer */

void producer(void)
{
    int item;
    message m;                                    /* message buffer */

    while (TRUE) {
        item = produce_item();                   /* generate something to put in buffer */
        receive(consumer, &m);                  /* wait for an empty to arrive */
        build_message(&m, item);                /* construct a message to send */
        send(consumer, &m);                    /* send item to consumer */
    }
}

. . .
```

Figure 2-36. The producer-consumer problem with N messages.

Producer-Consumer Problem with Message Passing

```
void consumer(void)
{
    int item, i;
    message m;

    for (i = 0; i < N; i++) send(producer, &m); /* send N empties */
    while (TRUE) {
        receive(producer, &m);                  /* get message containing item */
        item = extract_item(&m);                /* extract item from message */
        send(producer, &m);                    /* send back empty reply */
        consume_item(item);                  /* do something with the item */
    }
}
```

Figure 2-36. The producer-consumer problem with N messages.

Message Passing

- No shared memory on distributed systems...
- Instead we can send LAN messages.

```
send(destination, &message);  
receive(destination, &message);
```

Message Passing Issues

- Guard against lost messages (**acknowledgement**)
- **Authentication** (guard against imposters)
- Addressing :
 - To processes
 - Via **Mailbox** (place to buffer messages)
 - Send to a full mailbox means block
 - Receive from an empty mailbox means block

Message Passing Issues

- What about buffer-less messages?
- Send and Receive wait (block) for each other to be ready to talk: **rendezvous**

Barriers

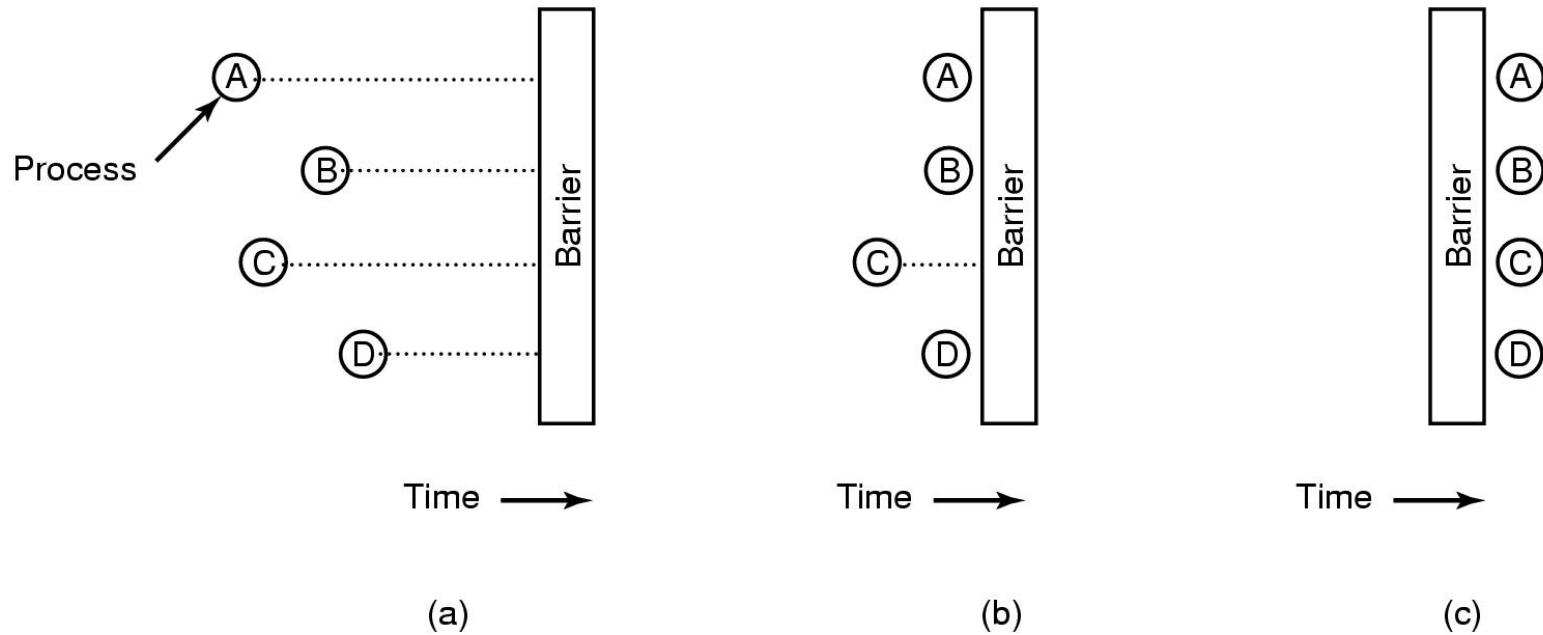


Figure 2-37. Use of a **barrier**. (a) Processes approaching a barrier. (b) All processes but one blocked at the barrier. (c) When the last process arrives at the barrier, all of them are let through.

Deadlock vs Starvation

- **Deadlock** - Process(es) waiting on events (resources) that will never happen:
 - Can be system wide or just one process
[will define this more precisely later]
- **Starvation** - Process(es) waiting for its turn but never comes:
 - Process could move forward, the resource or event might become available but this process does not get access.
 - Printing policy prints smallest file available. 1 process shows up with HUGE file, will not get to run if steady stream of files.

Deadlocks

Occur among **processes** who need to
acquire **resources** in order to **progress**

Resources

- Anything that must be acquired, used, and released over the course of time.
- Hardware or software resources
- Preemptable and Nonpreemptable resources:
 - Preemptable: can be taken away from the process with no ill-effect
 - Nonpreemptable: cannot be taken away from the process without causing the computation to fail

Resource Categories

Reusable

- can be safely used by only one process at a time and is not depleted by that use
 - processors, I/O channels, main and secondary memory, devices, and data structures such as files, databases, and semaphores

Consumable

- one that can be created (produced) and destroyed (consumed)
 - interrupts, signals, messages, and information
 - in I/O buffers

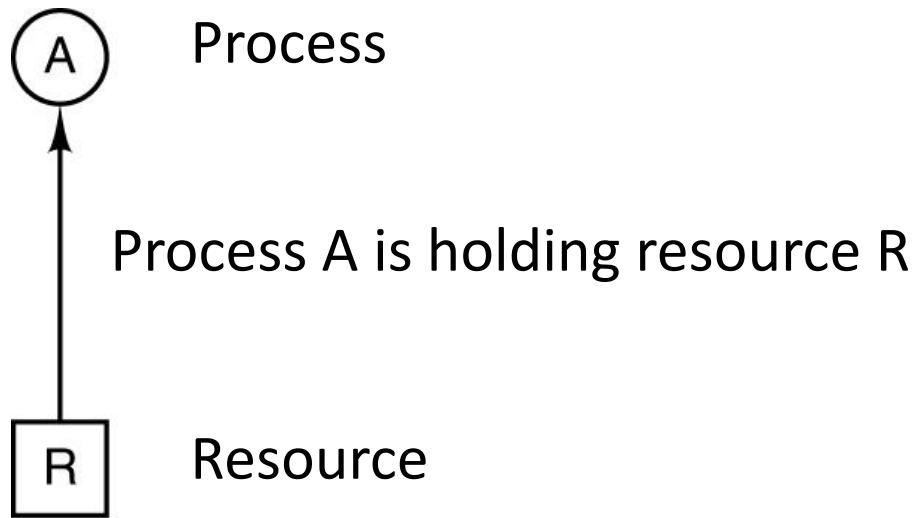
So ...

- A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.
- Assumptions
 - If a process is denied a resource, it is put to sleep
 - Only single-threaded processes
 - No interrupts possible to wake up a blocked process

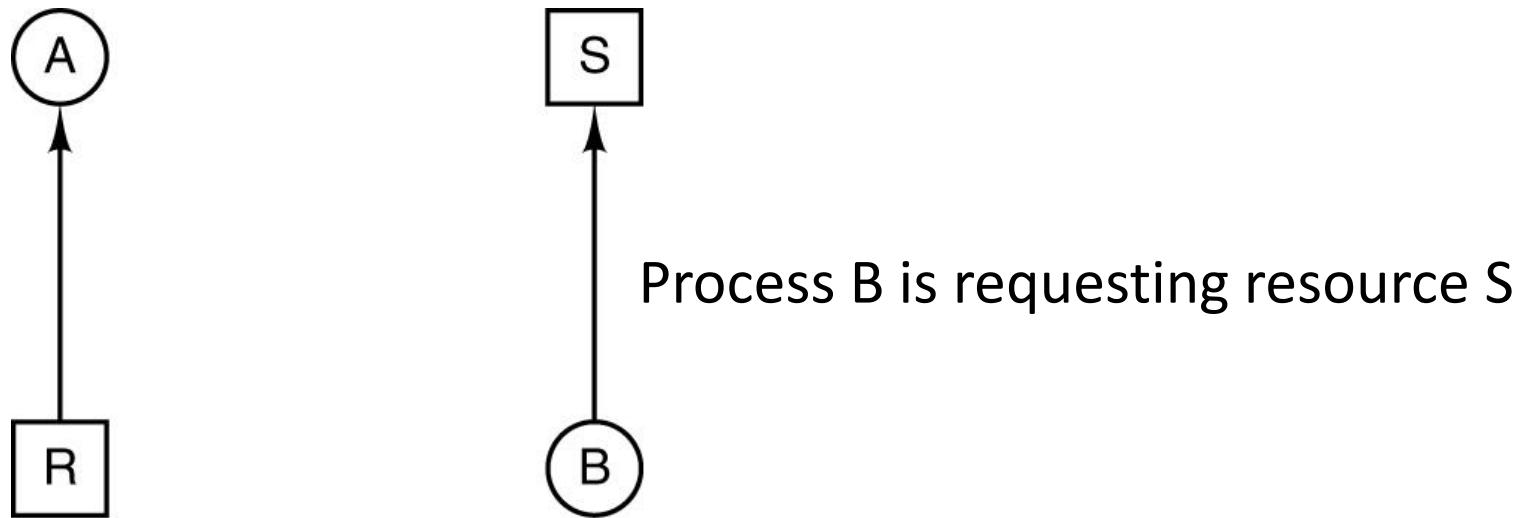
Conditions for Resource Deadlocks

1. Each resource is either currently assigned to exactly one process or is available.
2. Processes currently holding resources that were granted earlier can request new resources.
3. Resources previously granted cannot be forcibly taken away from a process. They must be explicitly released by the process holding them.
4. There must be a circular chain of two or more processes, each of which is waiting for a resource held by the next member of the chain.

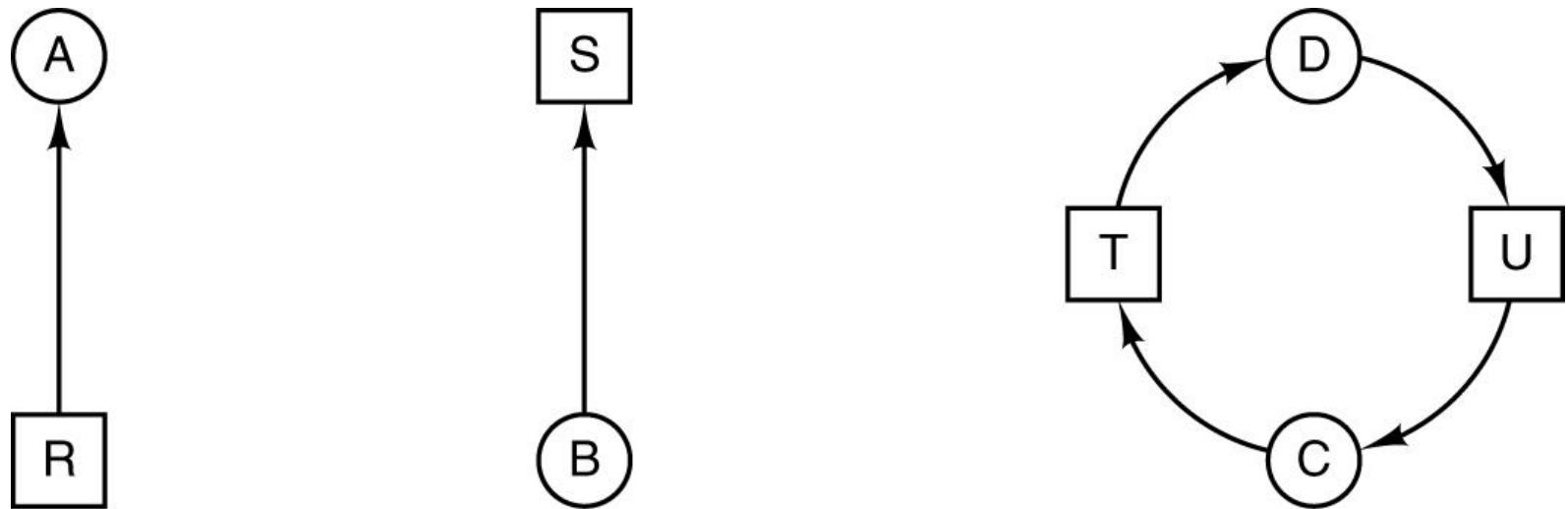
Resource Allocation Graph



Resource Allocation Graph



Resource Allocation Graph



Deadlock!

A

Request R
Request S
Release R
Release S

(a)

B

Request S
Request T
Release S
Release T

(b)

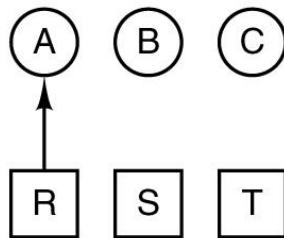
C

Request T
Request R
Release T
Release R

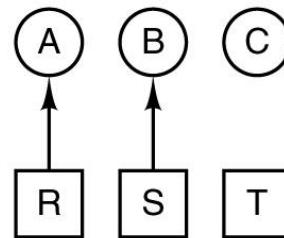
(c)

1. A requests R
 2. B requests S
 3. C requests T
 4. A requests S
 5. B requests T
 6. C requests R
- deadlock

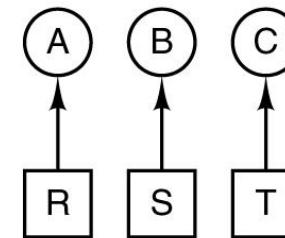
(d)



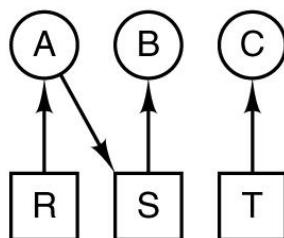
(e)



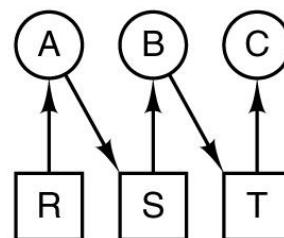
(f)



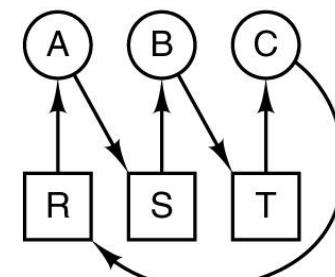
(g)



(h)



(i)



(j)

How to Deal with Deadlocks

1. Just ignore the problem !
2. Let deadlocks occur, detect them, and take action
3. Dynamic avoidance by careful resource allocation
4. Prevention, by structurally negating one of the four required conditions.

The Ostrich Algorithm



However: remember Murphy's law !!!

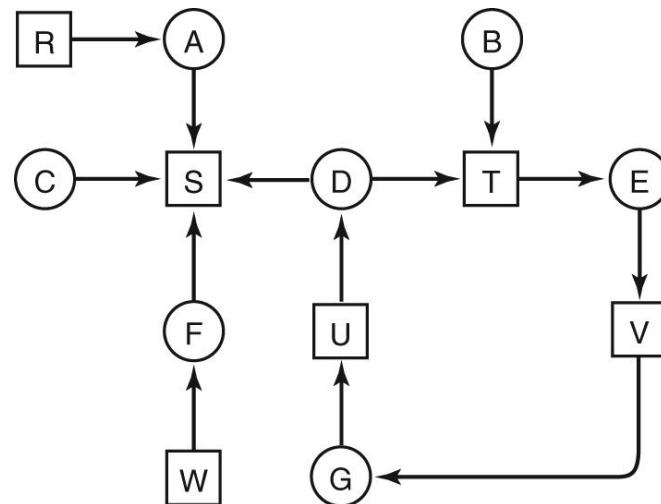
What can go wrong, will

Deadlock Detection and Recovery

- The system does not attempt to prevent deadlocks.
- It tries to detect it when it happens.
- Then it takes some actions to recover
- Several issues here:
 - Deadlock detection with one resource of each type
 - Deadlock detection with multiple resources of each type
 - Recovery from deadlock

Deadlock Detection: One Resource of Each Type

- Construct a resource graph
- If it contains one or more cycles, a deadlock exists



Formal Algorithm to Detect Cycles in the Allocation Graph

For Each node N in the graph do:

1. Initialize L to empty list and designate all arcs as unmarked
2. Add the current node to end of L. If the node appears in L twice then we have a cycle and the algorithm terminates
3. From the given node pick any unmarked outgoing arc. If none is available go to 5.
4. Pick an outgoing arc at random and mark it. Then follow it to the new current node and go to 2.
5. If the node is the initial node then no cycles and the algorithm terminates. Otherwise, we are in dead end. Remove that node and go back to the previous one. Go to 2.

When to Check for Deadlocks?

- Check every time a resource request is made
- Check every k minutes
- When CPU utilization has dropped below a threshold

Recovery from Deadlock

- We have detected a deadlock ... What next?
- We have some options:
 - Recovery through preemption
 - Recovery through rollback
 - Recovery through killing processes

Recovery from Deadlock: Through Preemption

- Temporary take a resource away from its owner and give it to another process
- Manual intervention may be required (e.g. in case of printer)
- Highly dependent on the nature of the resource.
- Recovering this way is frequently impossible.

Recovery from Deadlock: Through Rollback

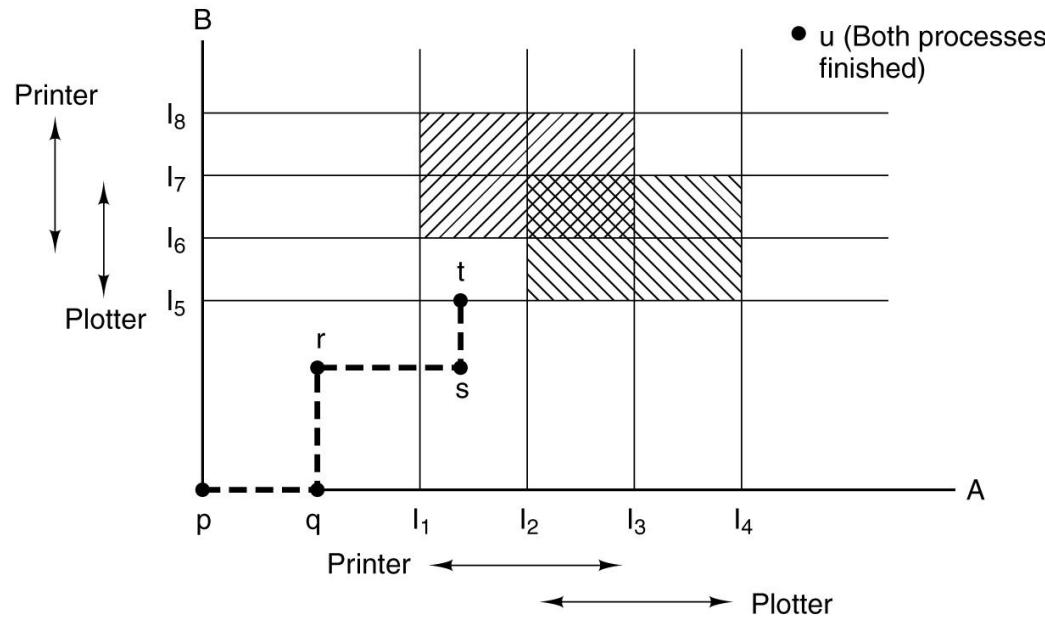
- Have processes **checkpointed** periodically
- Checkpoint of a process: its **state** is written to a file so that it can be restarted later
- In case of deadlock, a process that owns a needed resource is rolled back to the point before it acquired that resource

Recovery from Deadlock: Through Killing Processes

- Kill a process in the cycle.
- Can be repeated (i.e. kill other processes) till deadlock is resolved
- The victim can also be a process NOT in the cycle

Deadlock Avoidance

- In most systems, resources are requested one at a time.
- Resource is granted only if it is **safe** to do so



Safe and Unsafe States

- A **state** is said to be safe if there is one scheduling order in which every process can run to completion even if all of them suddenly request their maximum number of resources immediately
- An **unsafe** state is NOT a deadlock state

Safe and Unsafe States

Maximum the process will need (e.g. A will need 6 more).

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3
(a)

	Has	Max
A	3	9
B	4	4
C	2	7

Free: 1
(b)

	Has	Max
A	3	9
B	0	-
C	2	7

Free: 5
(c)

	Has	Max
A	3	9
B	0	-
C	7	7

Free: 0
(d)

	Has	Max
A	3	9
B	0	-
C	0	-

Free: 7
(e)

Assume a total of 10 instances of the resources available

This state is **safe** because there exists a sequence of allocations that allows all processes to complete.

Safe and Unsafe States

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3
(a)

	Has	Max
A	4	9
B	2	4
C	2	7

Free: 2
(b)

	Has	Max
A	4	9
B	4	4
C	2	7

Free: 0
(c)

	Has	Max
A	4	9
B	—	—
C	2	7

Free: 4
(d)



How about this state?

The difference between a safe and unsafe state is that from a safe state the system can guarantee that all processes will finish; from an unsafe state, no such guarantee can be given.

The Banker's Algorithm

- Dijkstra 1965
- Checks if granting the request leads to an unsafe state
- If it does, the request is denied.

The Banker's Algorithm: The Main Idea

- The algorithm checks to see if it has enough resources to satisfy some customers
- If so, the process closest to the limit is assumed to be done and resources are back, and so on.
- If all loans (resources) can eventually be repaid, the state is safe.

The Banker's Algorithm: Example (single resource type)

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

Free: 10

(a)

Safe

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

Free: 2

(b)

Safe

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

Free: 1

(c)

Unsafe

The Banker's Algorithm: Example (multiple resources)

Process
Tape drives
Plotters
Printers
CD ROMs

A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

Resources assigned

Process
Tape drives
Plotters
Printers
CD ROMs

A	1	1	0	0
B	0	1	1	2
C	3	1	0	0
D	0	0	1	0
E	2	1	1	0

Resources still needed

$$\begin{aligned}E &= (6342) \\P &= (5322) \\A &= (1020)\end{aligned}$$

The Banker's Algorithm

- Very nice theoretically
- Practically useless!
 - Processes rarely know in advance what their maximum resource needs will be.
 - The number of processes is not fixed.
 - Resources can suddenly vanish.

Deadlock Prevention

- Deadlock avoidance is essentially impossible.
- If we can ensure that at least one of the four conditions of the deadlock is never satisfied, then deadlocks will be structurally impossible.

Deadlock Prevention: Attacking the Mutual Exclusion

- Can be done for some resources (e.g the printer) but not all.
- Spooling
- Words of wisdom:
 - Avoid assigning a resource when that is not absolutely necessary.
 - Try to make sure that as few processes as possible may actually claim the resource

Deadlock Prevention: Attacking the Hold and Wait Condition

- Prevent processes holding resources from waiting for more resources
- This requires all processes to request all their resources before starting execution
- A different strategy: require a process requesting a resource to first temporarily release all the resources it currently holds. Then tries to get everything it needs all at once

Deadlock Prevention: Attacking No Preemption Condition

- Virtualizing some resources can be a good strategy (e.g. virtualizing a printer)
- Not all resources can be virtualized (e.g. records in a database)

Deadlock Prevention: The circular Wait Condition

- Method 1: Have a rule saying that a process is entitled only to a single resource at a moment.
- Method 2:
 - Provide a global numbering of all resources.
 - A process can request resources whenever they want to, but all requests must be done in numerical order
 - With this rule, resource allocation graph can never have cycles.

Deadlock Prevention: Summary

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

Conclusions

- Deadlocks can occur on hardware/software resources
- OS needs to be able to:
 - Try to avoid them if possible
 - Detect deadlocks
 - Deal with them when detected



CSCI-GA.2250-001

Operating Systems

Lecture 5: Memory Management

Hubertus Franke
frankeh@cims.nyu.edu



Programmer's dream



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

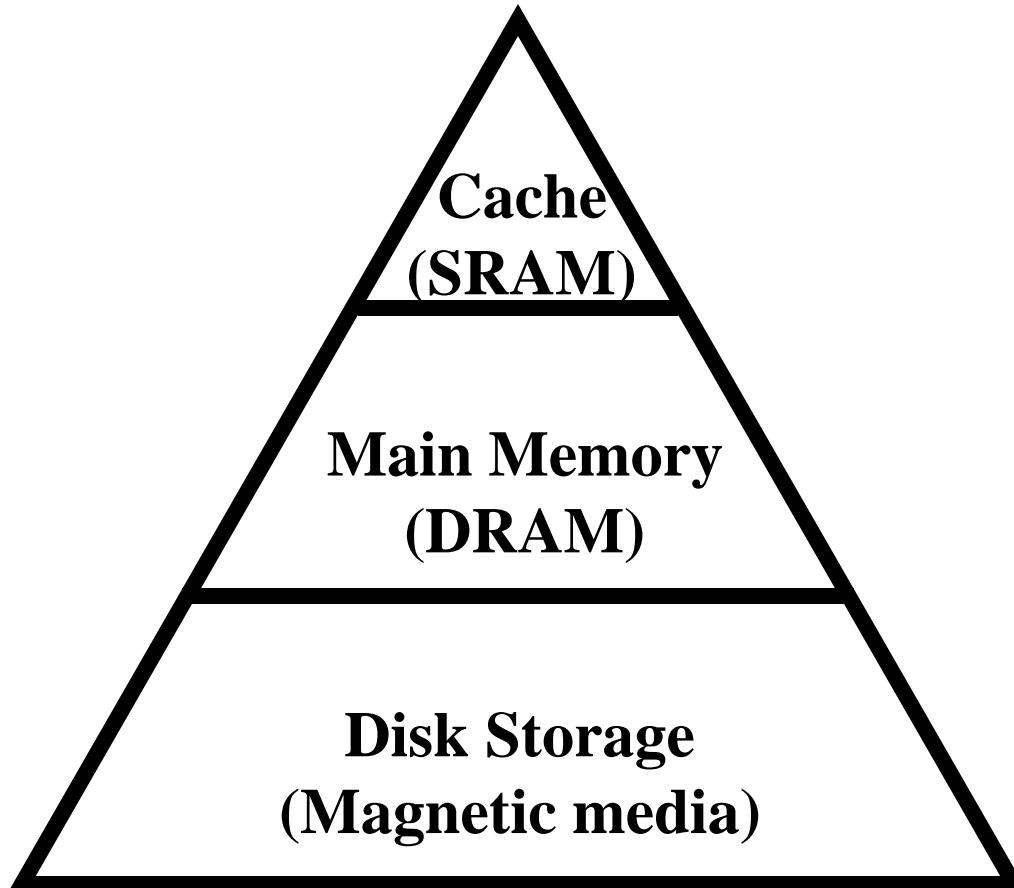
Programmer's Wish List



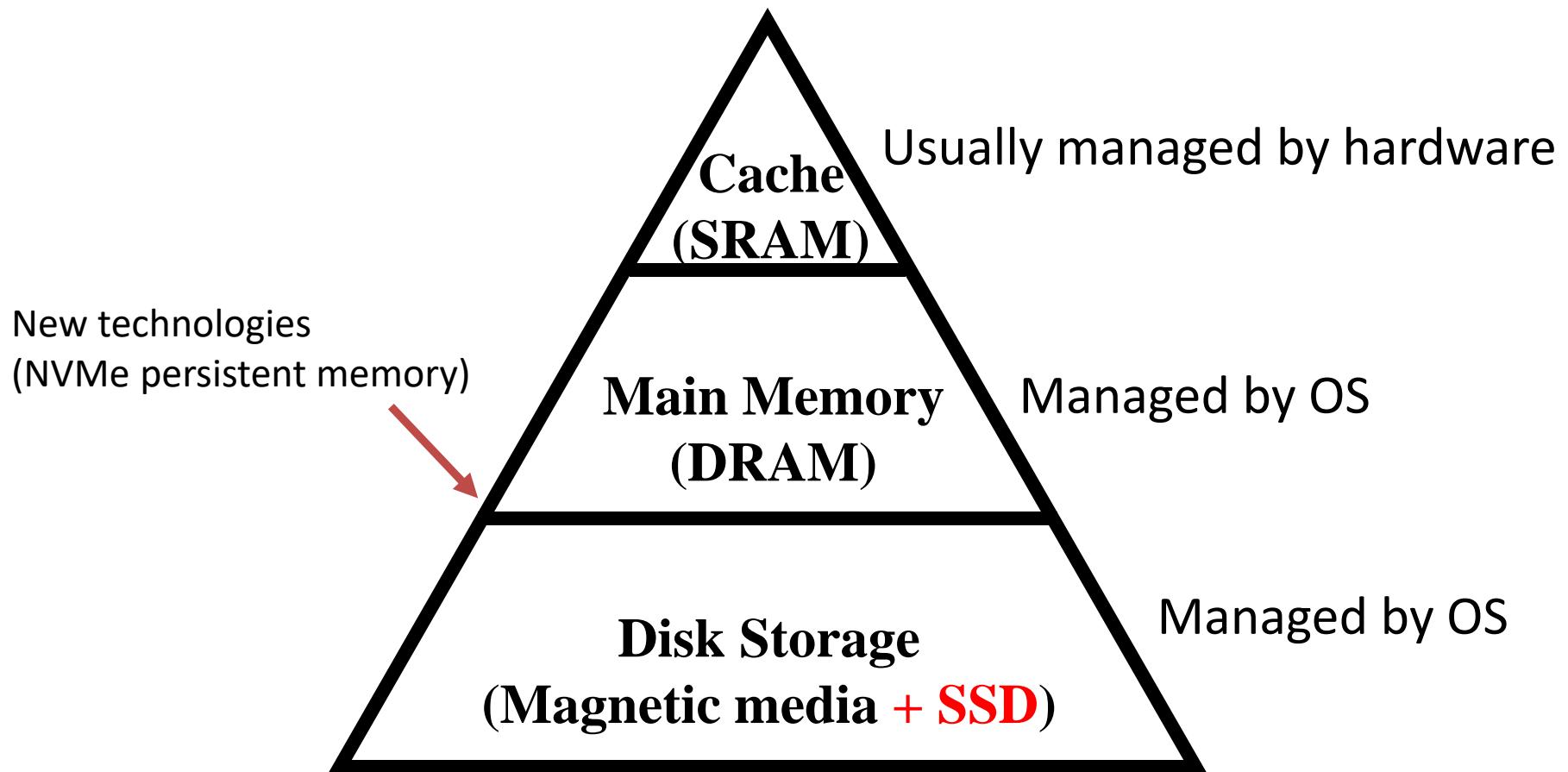
- Private
- Infinitely large
- Infinitely fast
- Non-volatile
- Inexpensive

Programs are getting bigger faster than memories.

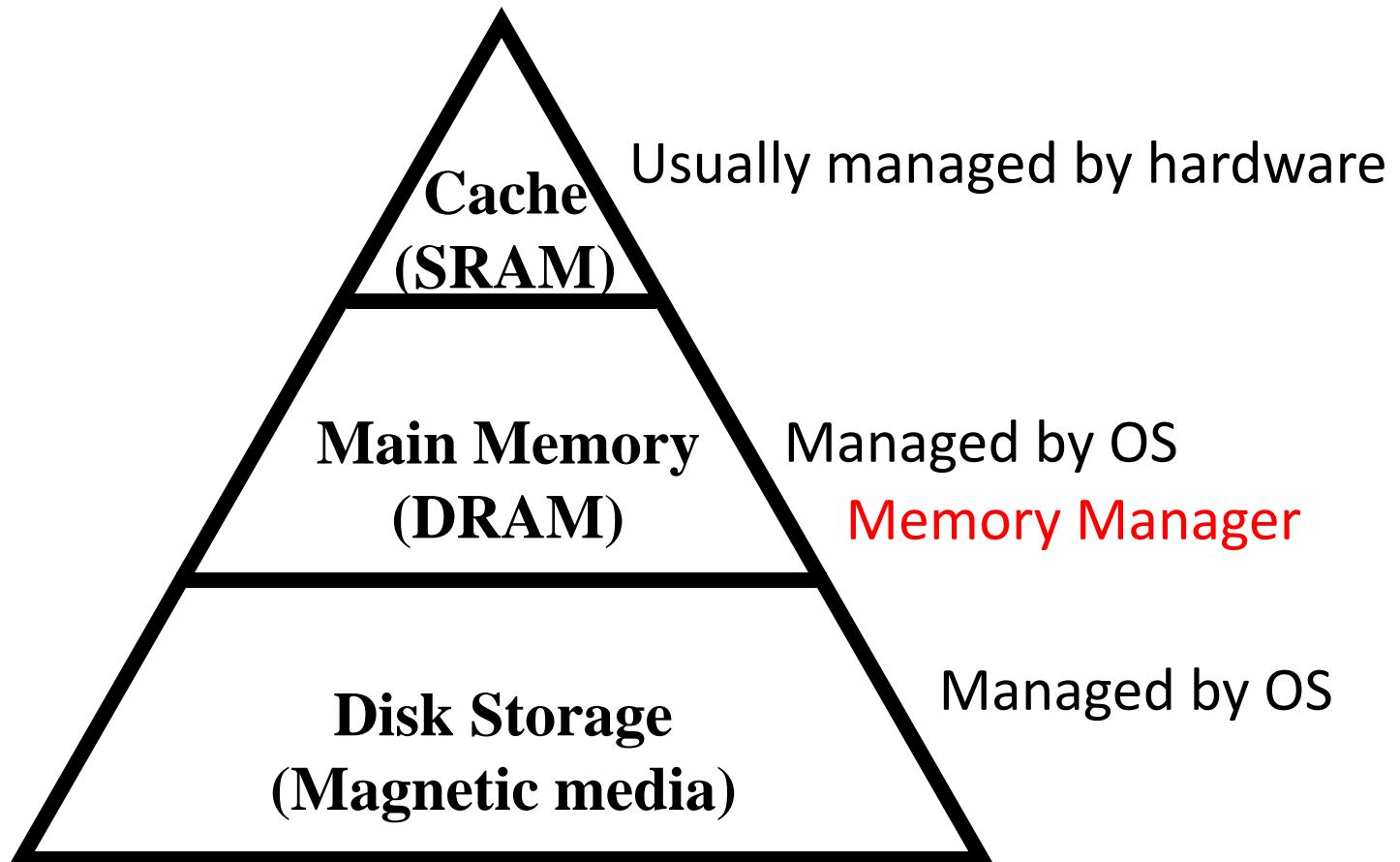
Memory Hierarchy



Memory Hierarchy

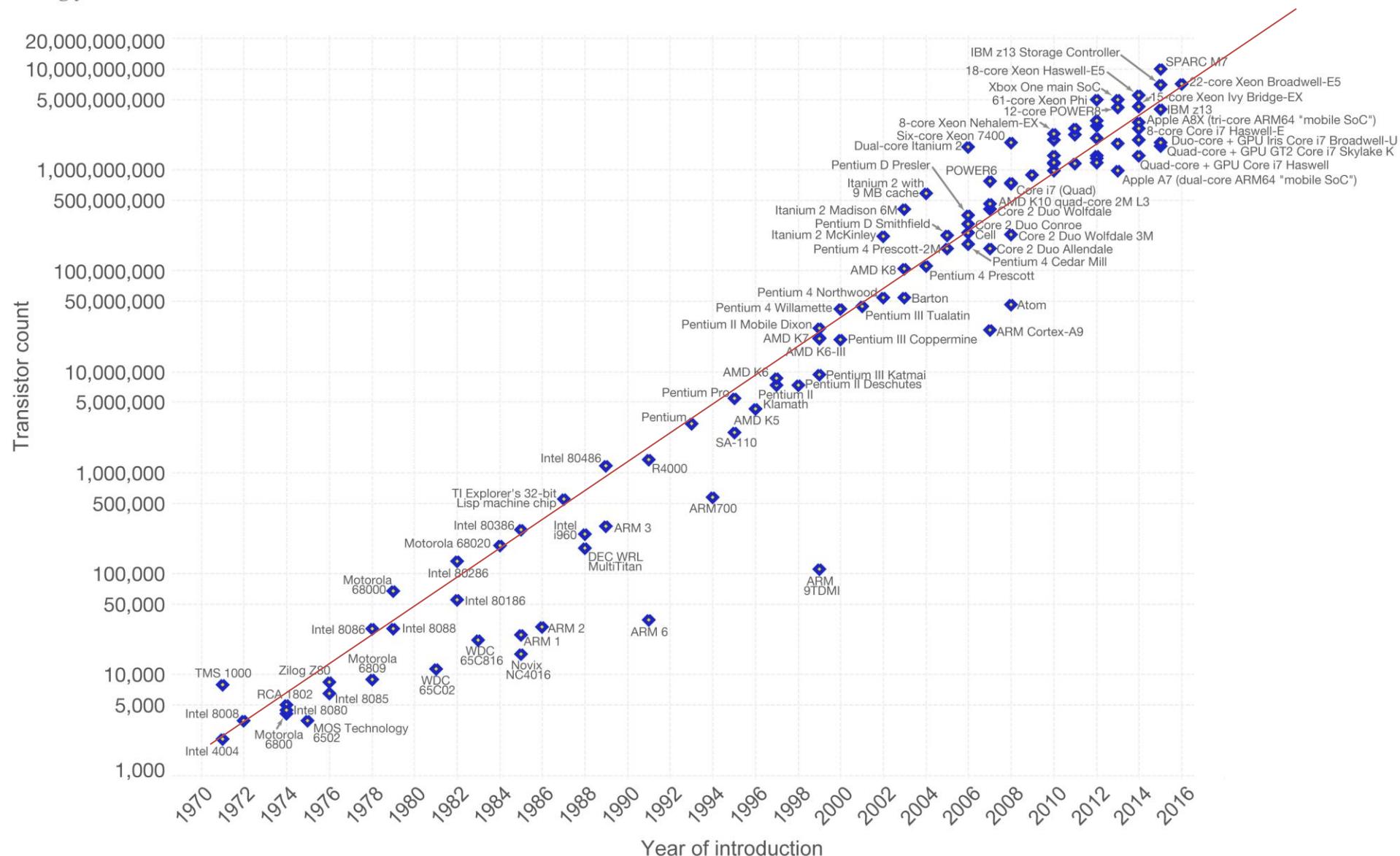


Memory Hierarchy



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

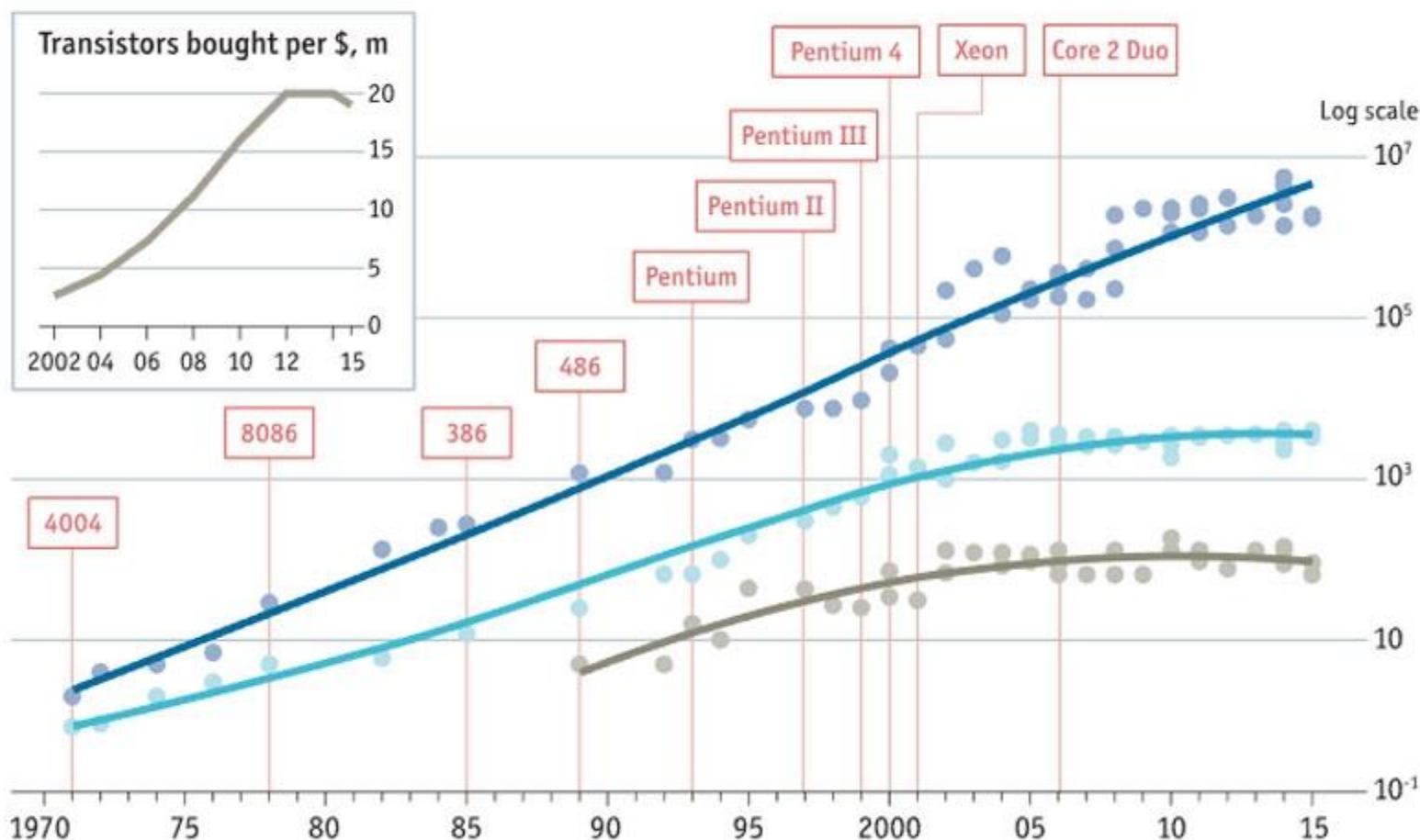
Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are strongly linked to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



Stuttering

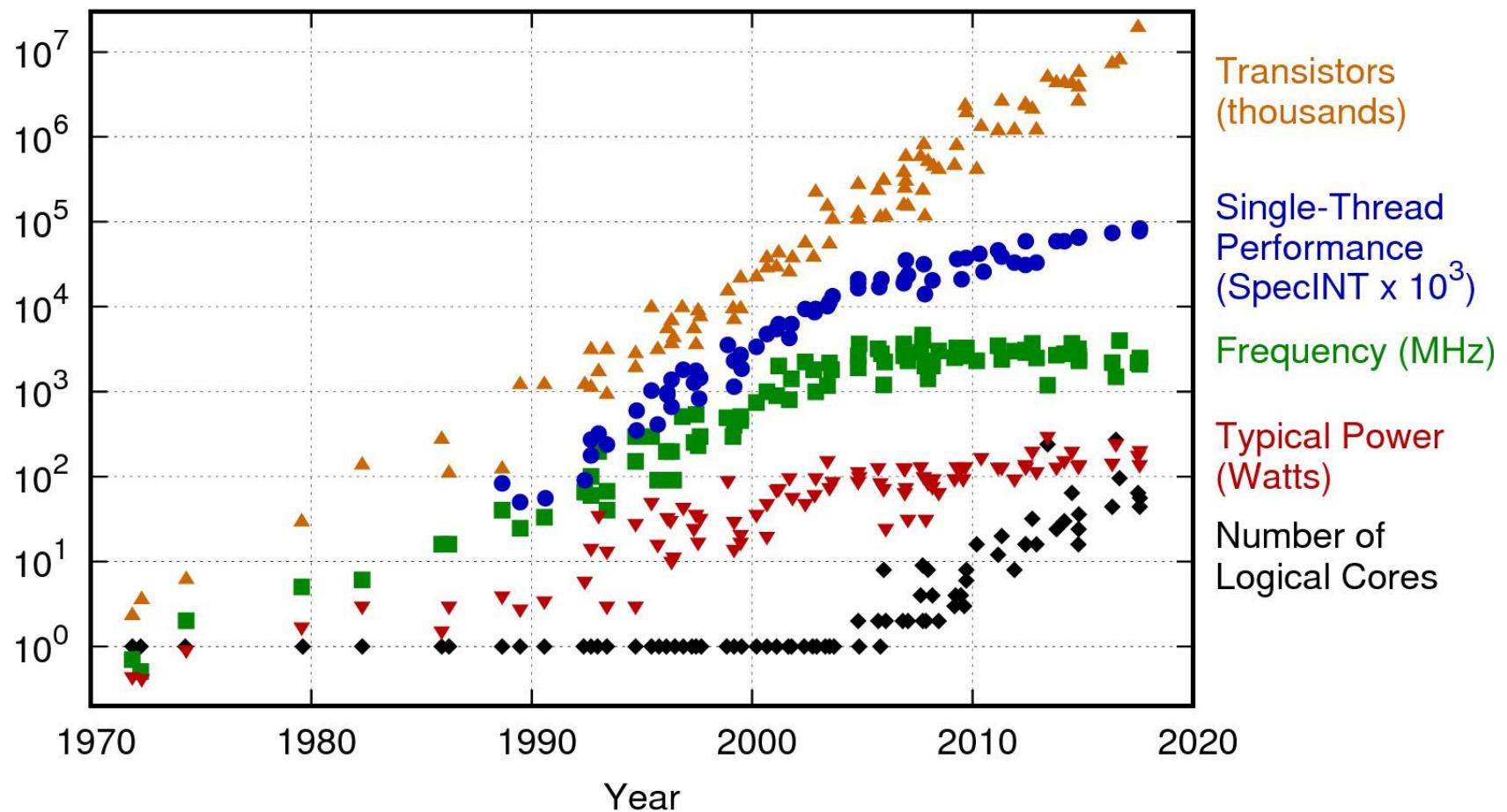
● Transistors per chip, '000 ● Clock speed (max), MHz ● Thermal design power*, w

Chip introduction
dates, selected



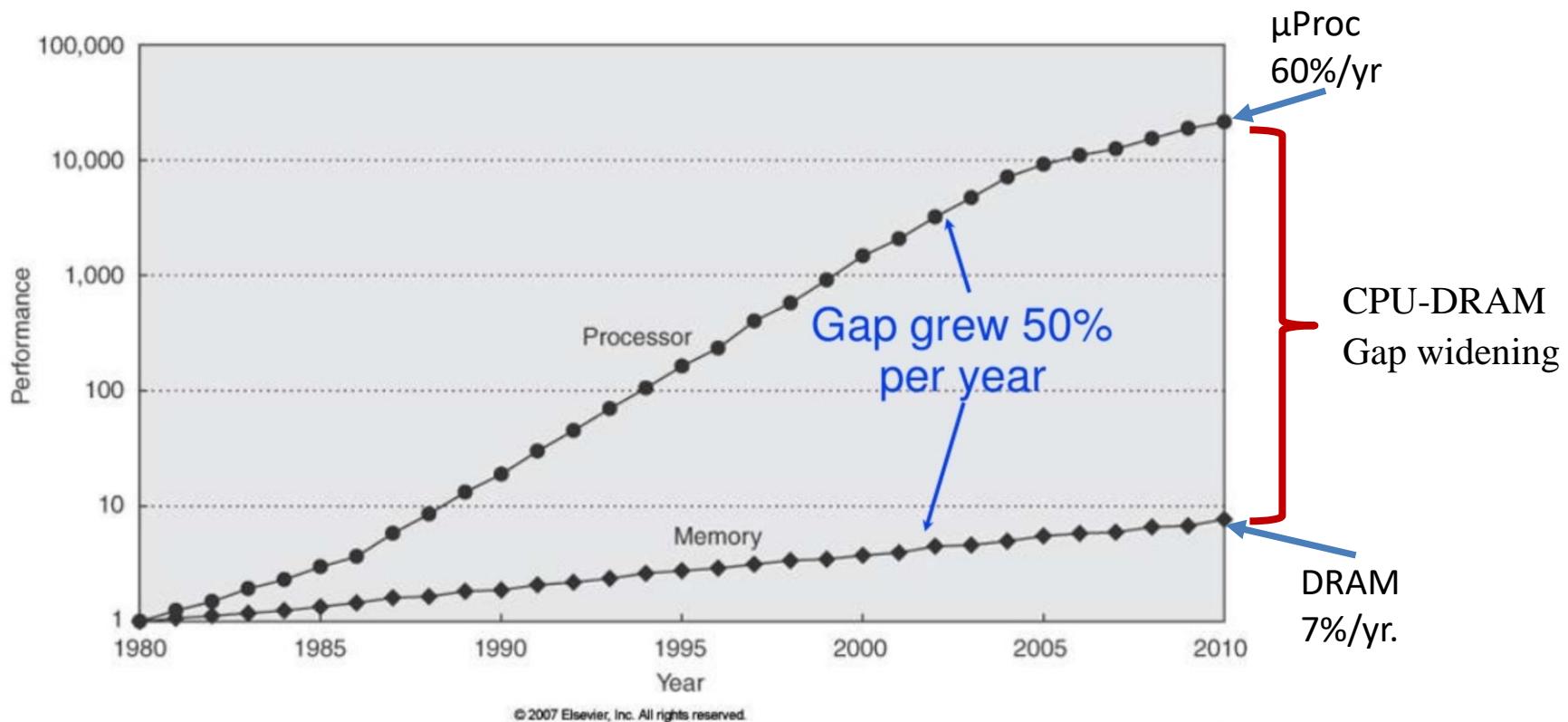
Sources: Intel; press reports; Bob Colwell; Linley Group; IB Consulting; *The Economist*

*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:

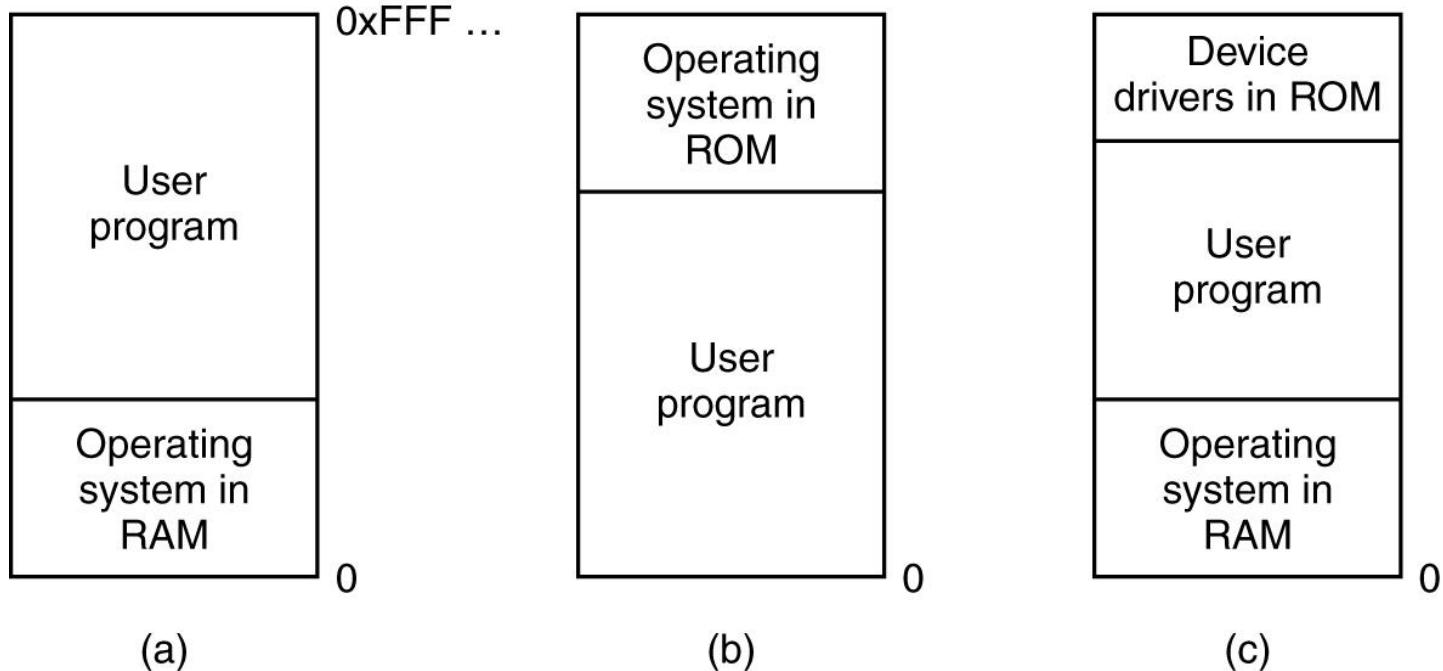
- Longer per cycle memory access times → Memory seems “further away”
- New Technologies: SDR → DDR-1/2/3/4/5 → QDR (technologies)
Still problem widens

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)

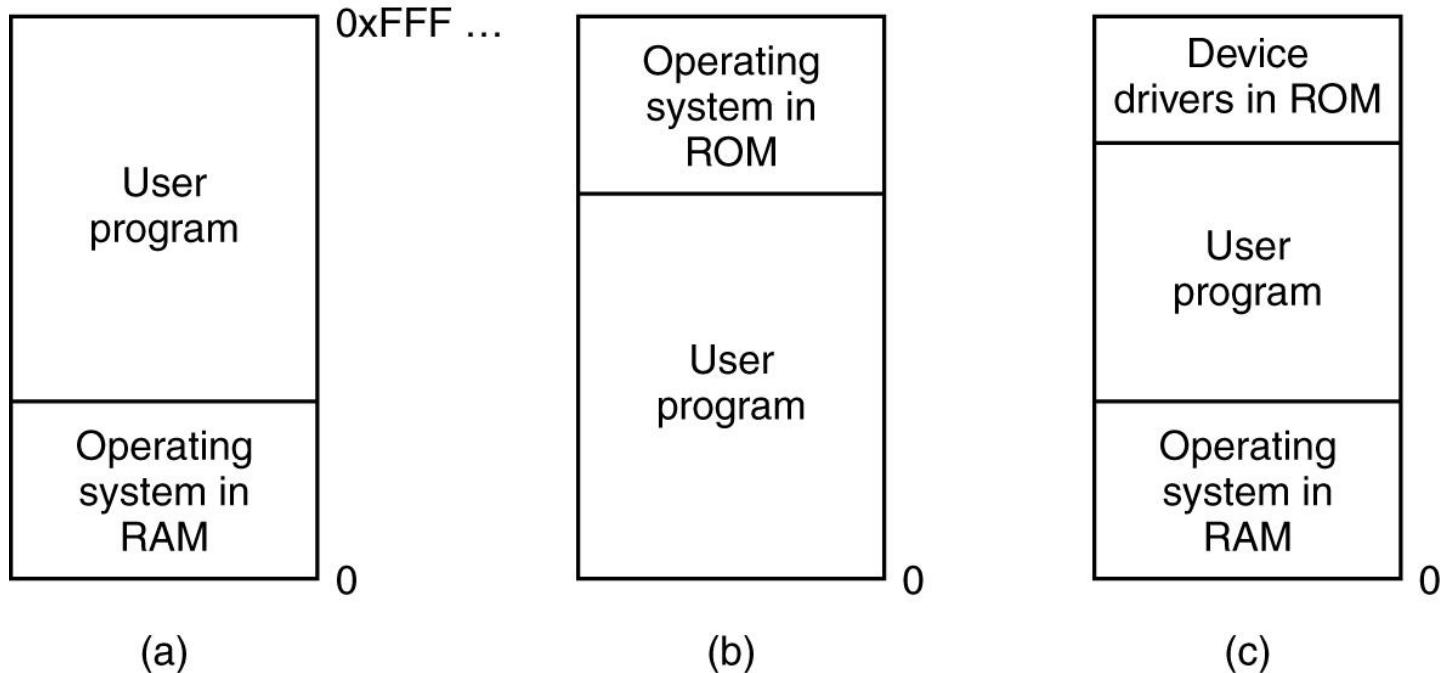
Is abstraction really necessary?

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)

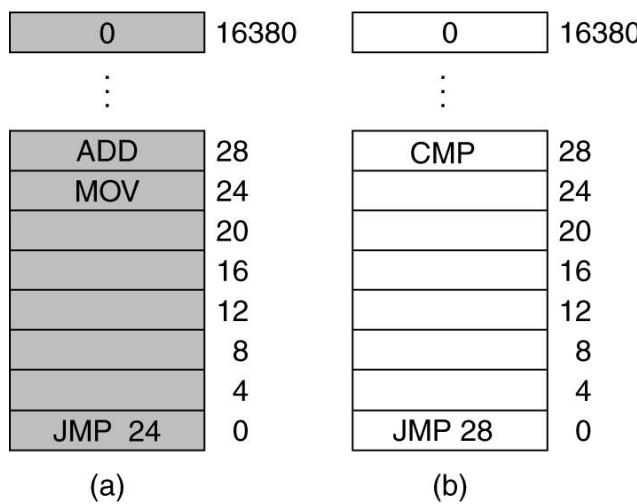
No Memory Abstraction

- 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

:

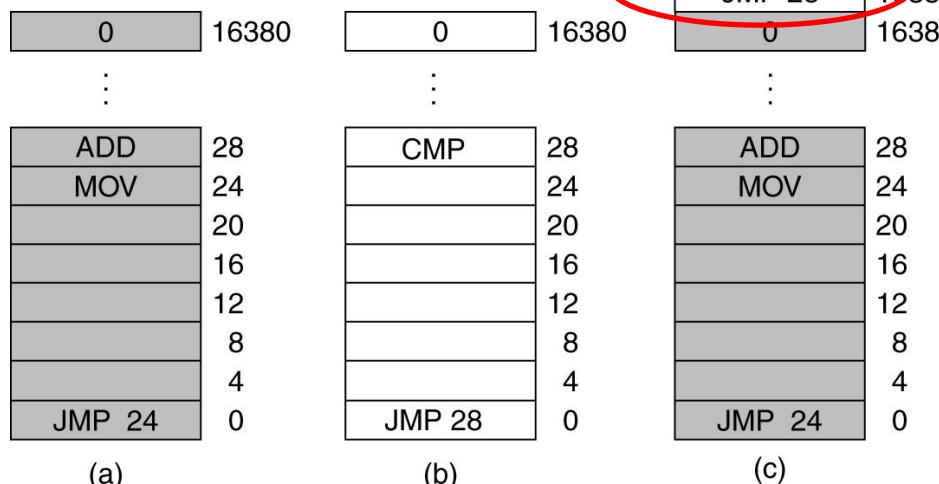
ADD	28
MOV	24
	20
	16
	12
	8
	4
JMP 24	0

(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

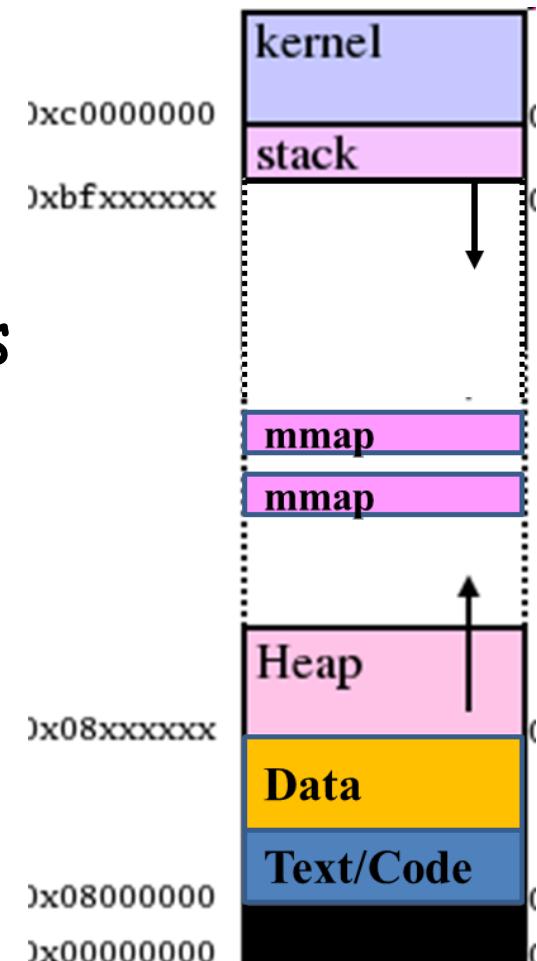
- It is advantageous to allow each process access to the same copy of the program rather than have their own separate copy
- Memory management must allow controlled access to shared areas of memory without compromising protection

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

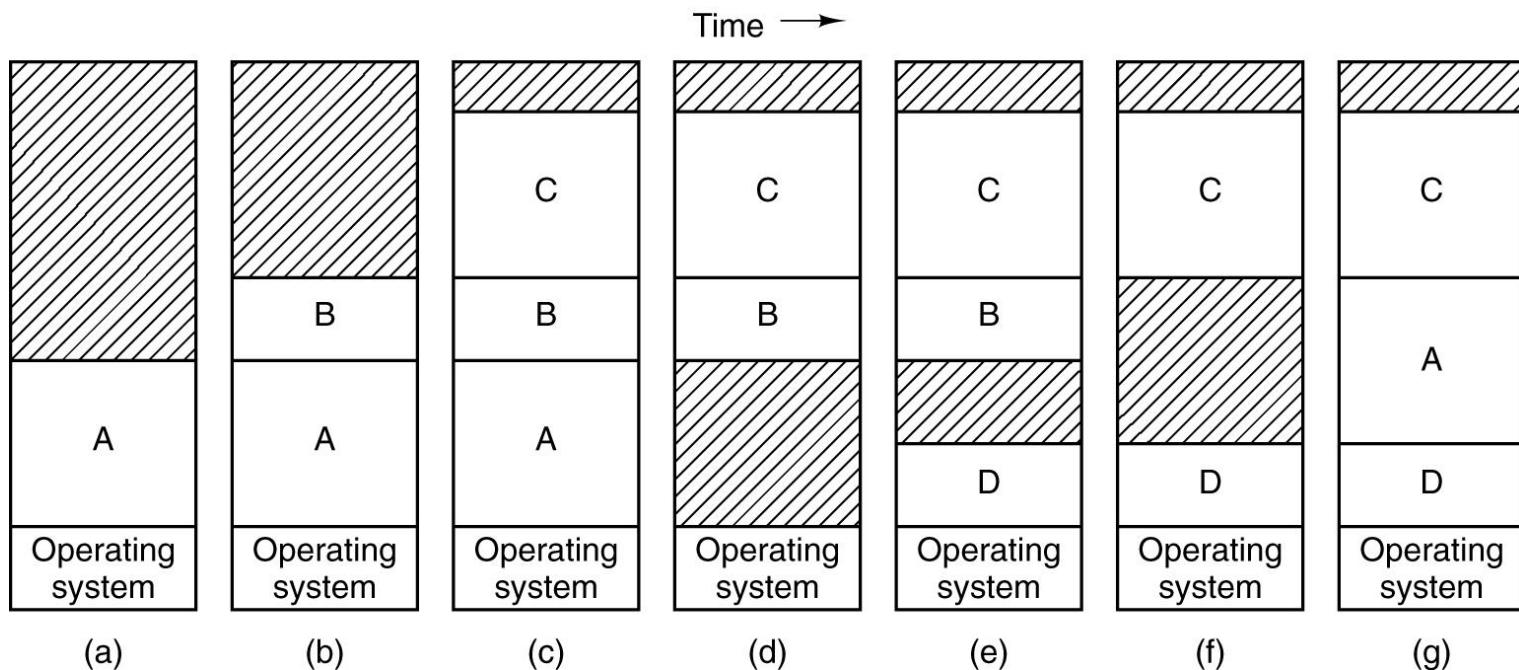
Address Space: Base and Limit

- 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

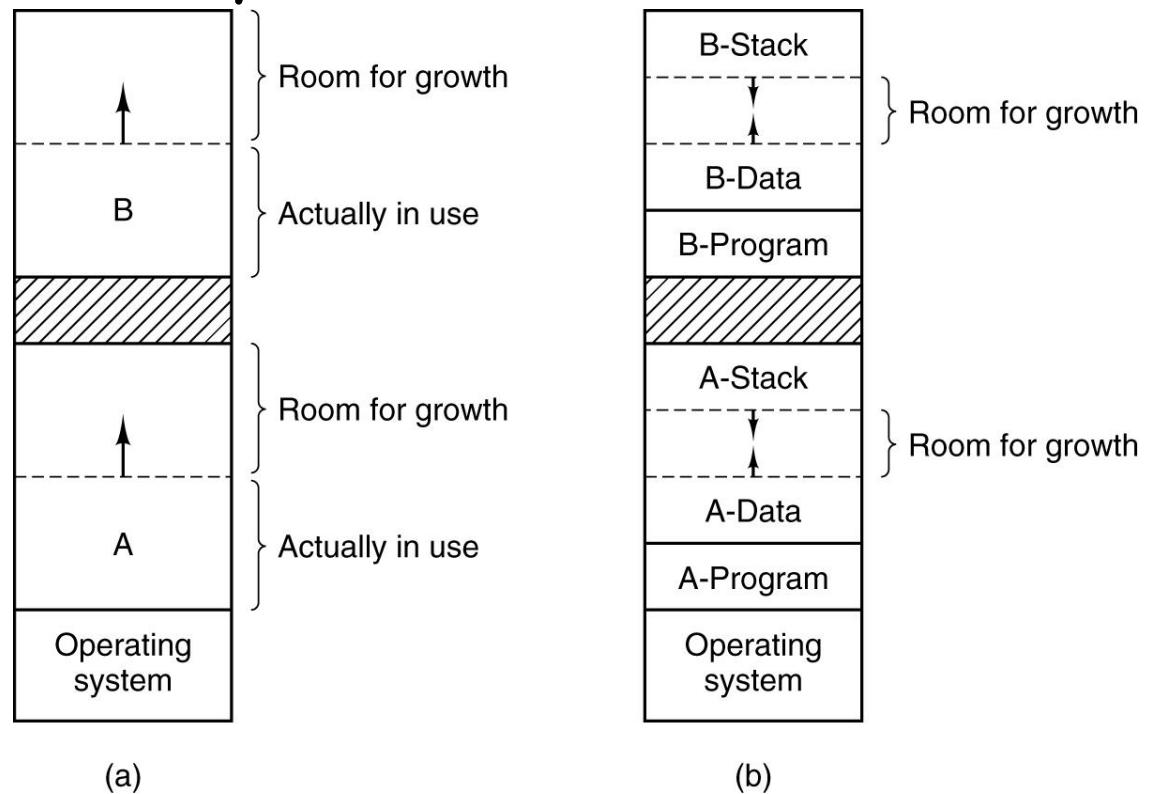


Swapping

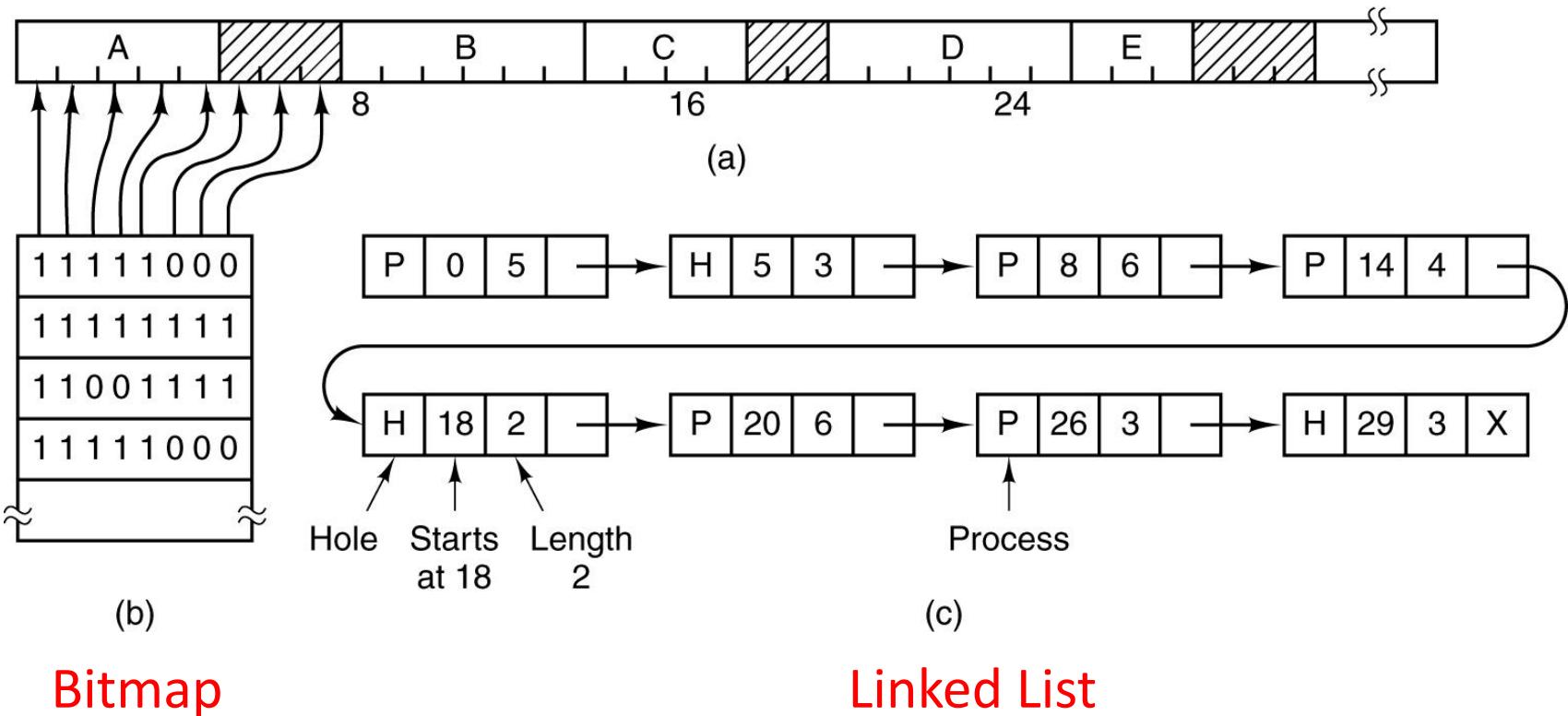
- Programs move in and out of memory
- Holes are created
- Holes can be combined -> memory compaction
- What if a process needs more memory?
 - If a hole is adjacent to the process, it is allocated to it
 - Process has to be moved to a bigger hole
 - Process suspended till enough memory is there

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.

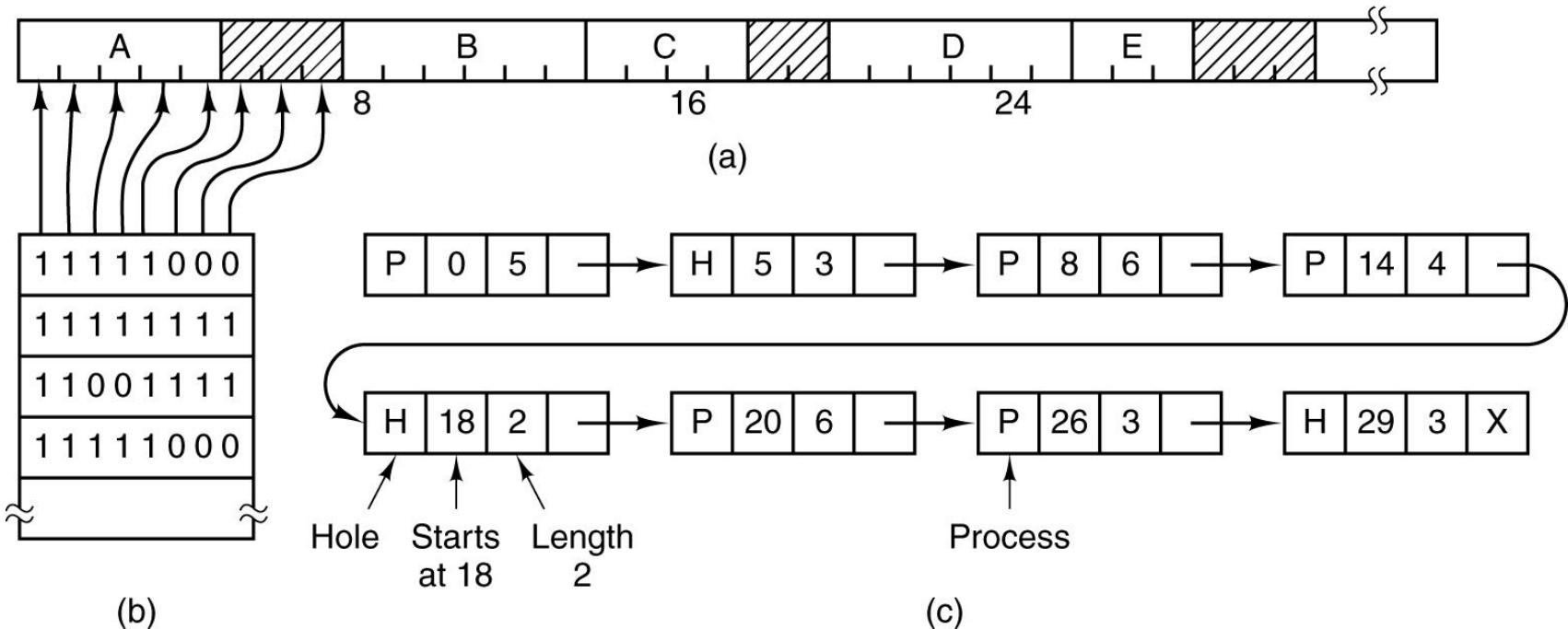


Managing Free Memory



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

Managing Free Memory



Bitmap

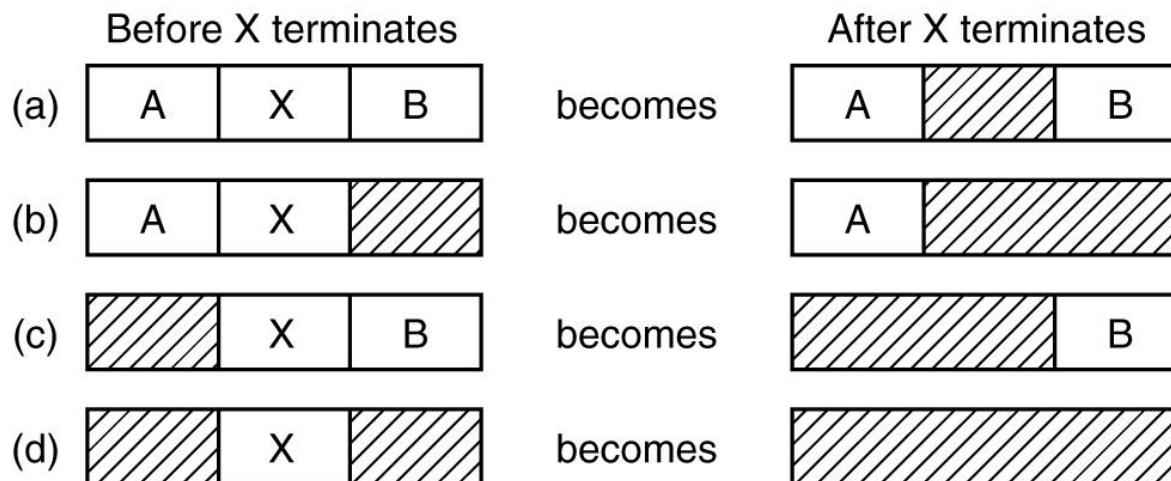


Slow: To find k-consecutive 0s for a new process

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 ?

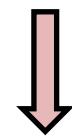
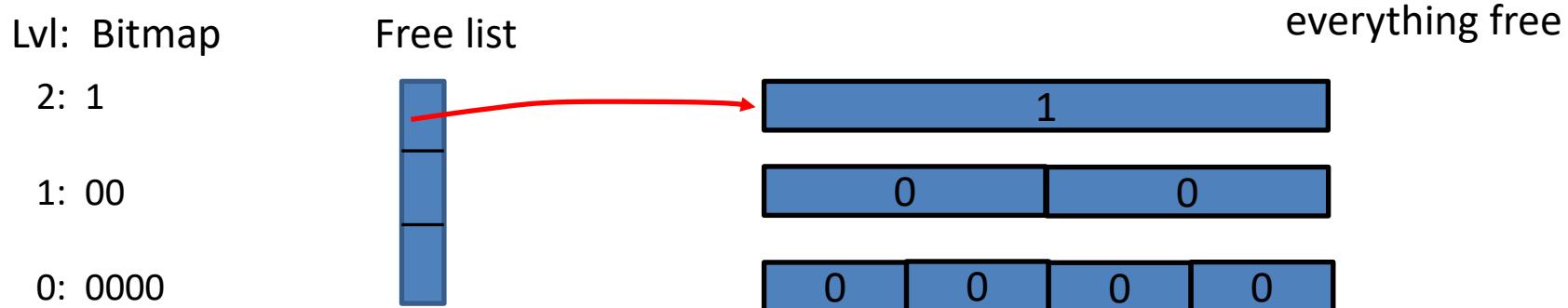
- How to allocate?
 - First fit
 - Best fit
 - Next fit
 - Worst fit
 - ...
- Each has their philosophy to why and what their pros and cons are.
At the end you must weight efficiency of space (fragmentation) and time (to manage)

Buddy Algorithm

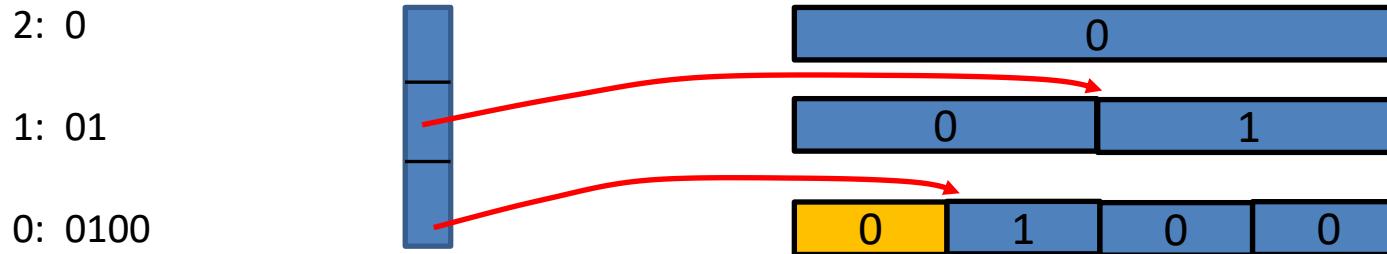
- Considers blocks of memory only as 2^N
- Potential for fragmentation (drawback)
- If no block of a size is available it splits higher blocks into smaller blocks
- Easy to implement and fast
 $O(\log_2(\text{MaxBlockSz}/\text{MinBlockSize}))$
e.g. 4K .. 128B = $2^{12-7} = 5$ steps

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

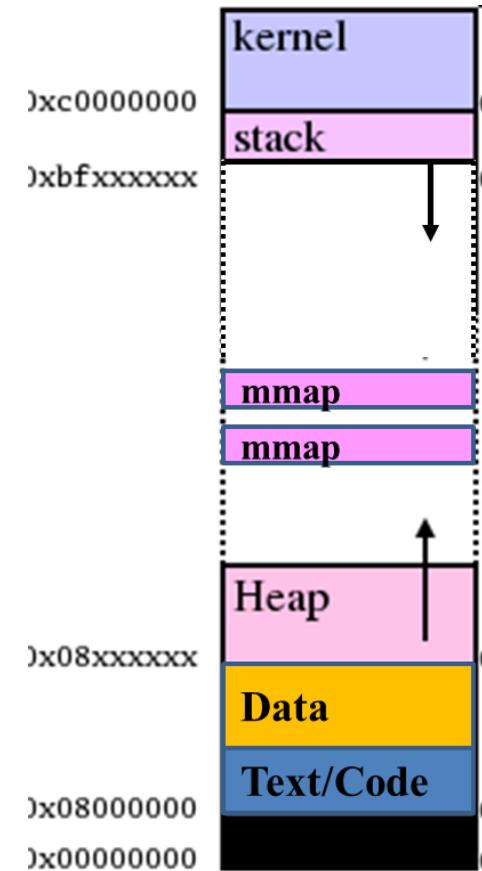
Mapping from
logical (virtual) address space
(user / process perspective)

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

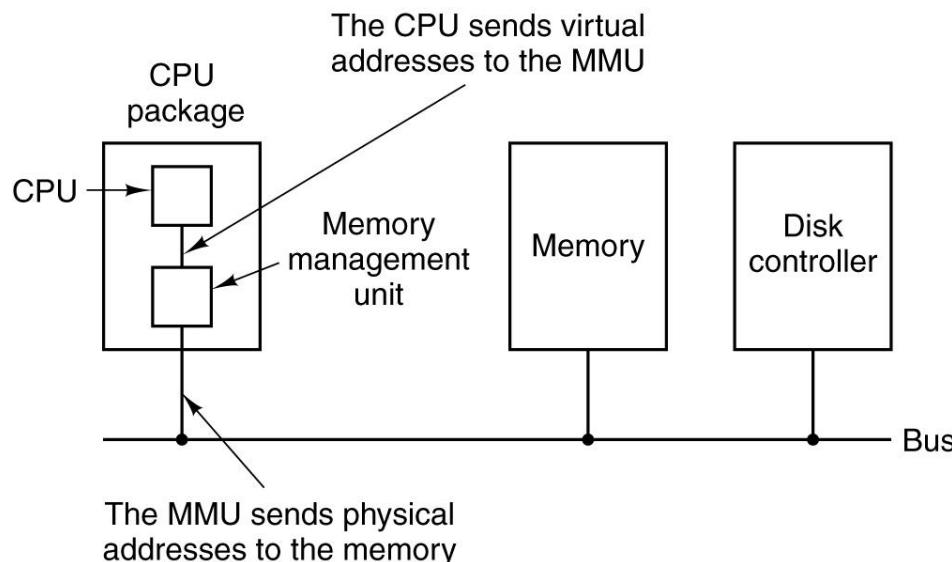
1. Operating system brings into main memory a few pieces of the program.
2. An exception is generated when an address is needed that is not in main memory (will see how).
3. Operating system places the process in a blocking state.
4. Operating system allocates memory and optionally issues a disk I/O 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)



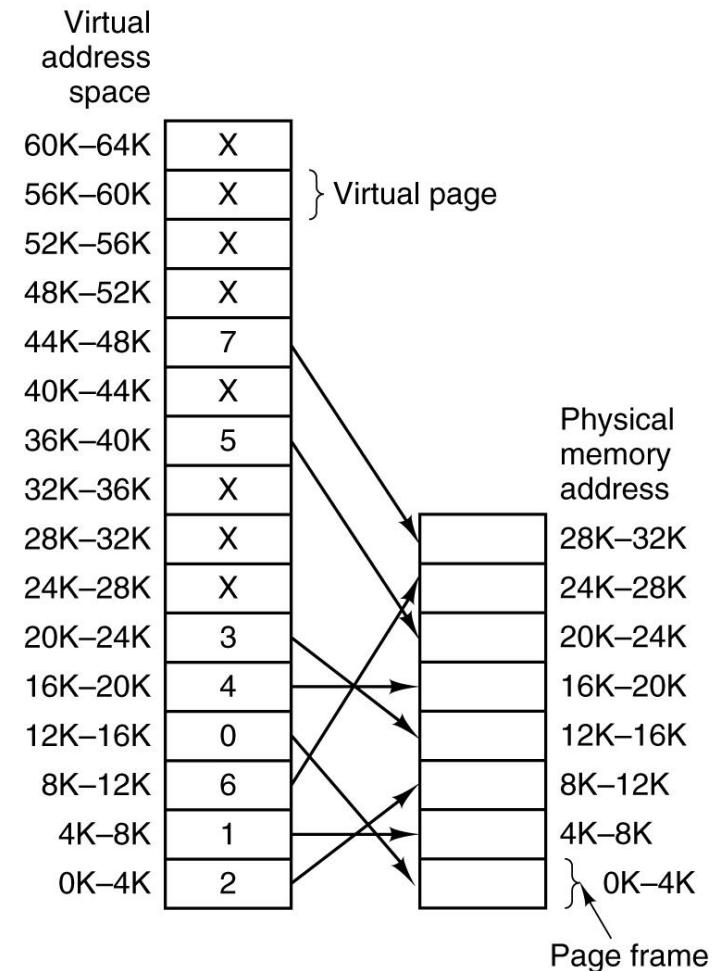
Virtual Memory

Single Process / Address Space view first

Allows to have larger virtual address space than physical memory available

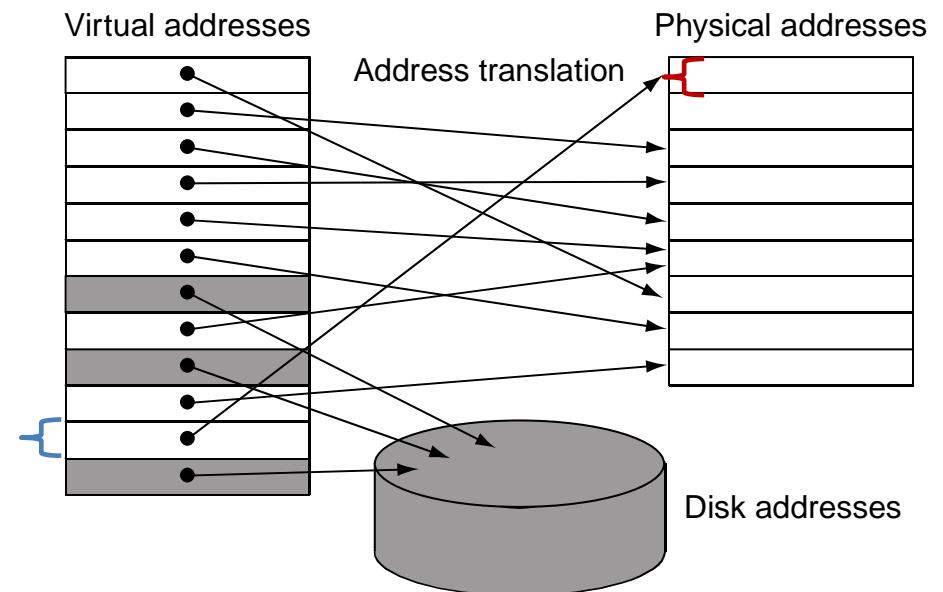
"X" → when you access a virtual page you get a page fault (see prior story) that needs to be resolved first

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



Virtual Memory

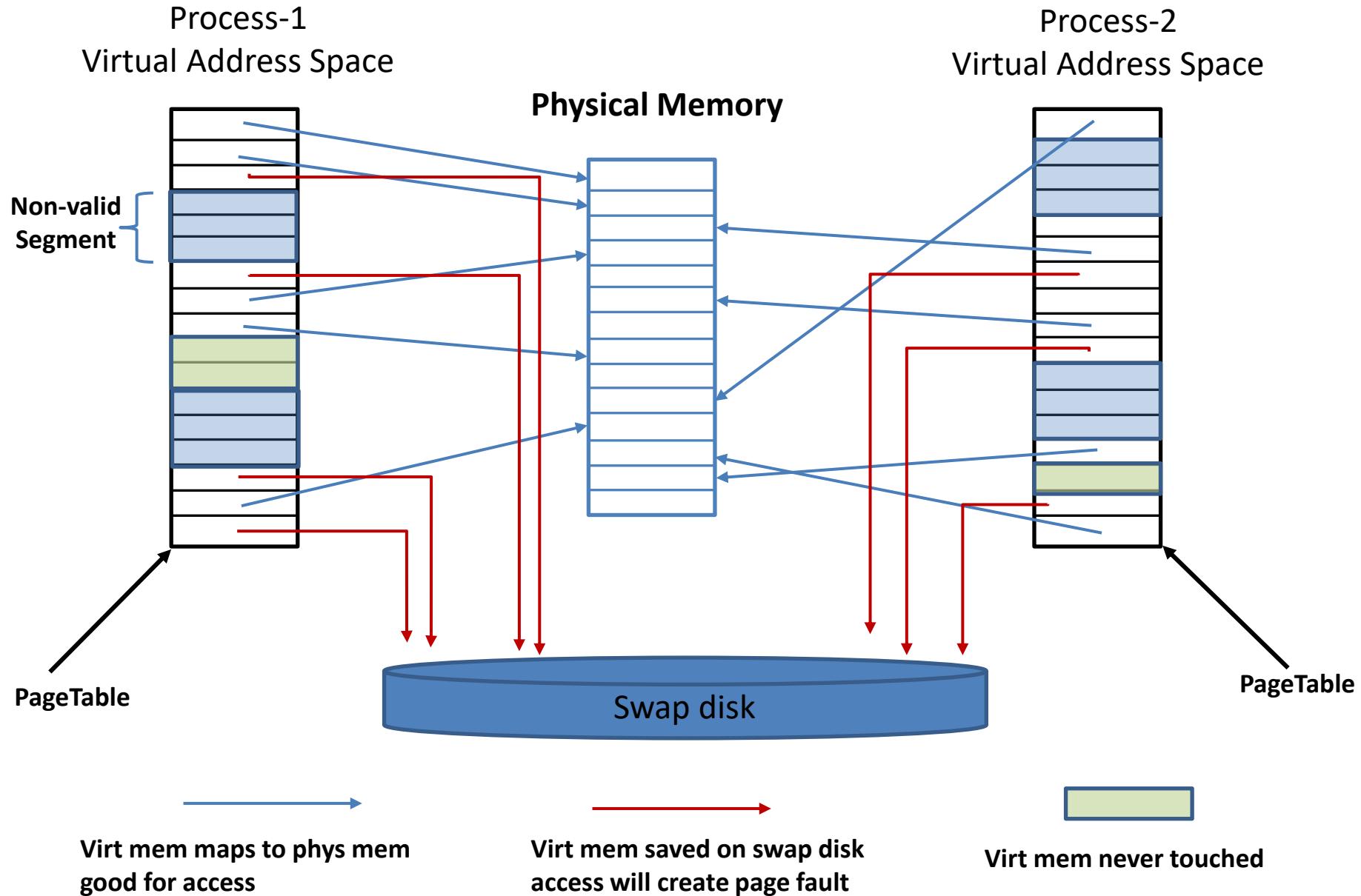
- Main memory can act as a cache for the secondary storage (disk)



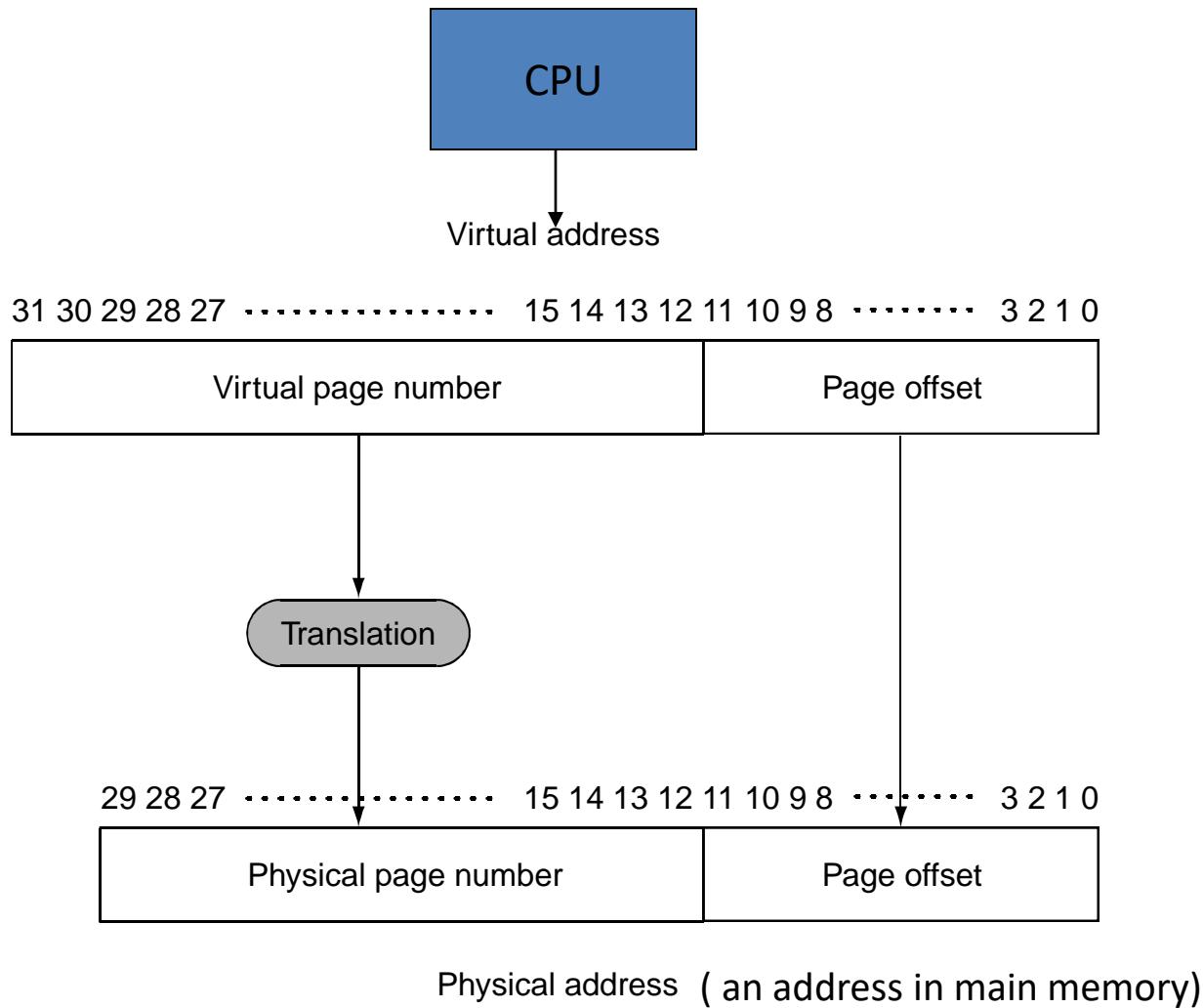
- Advantages:
 - illusion of having more physical memory
 - program relocation
 - protection

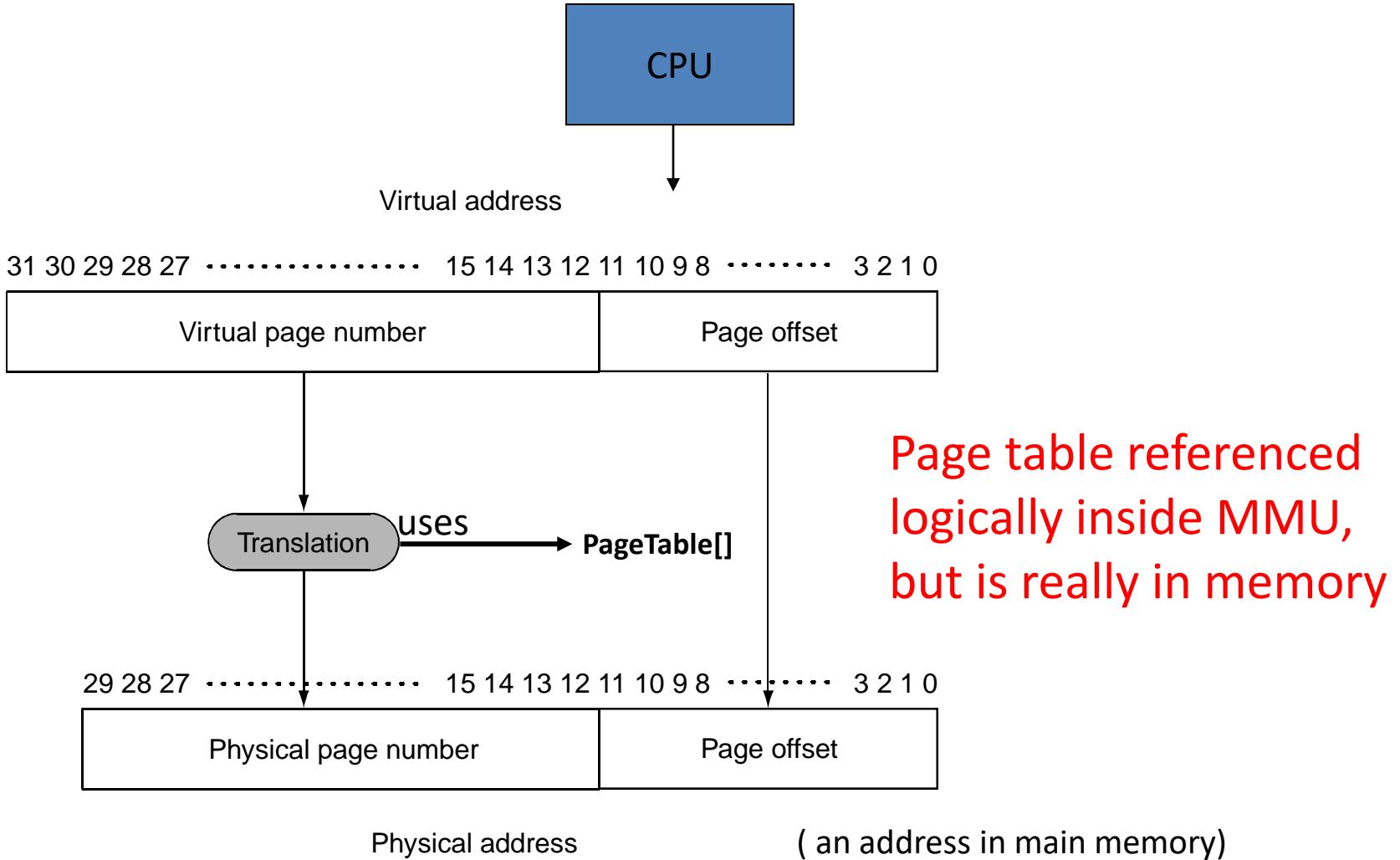
Block-of-mem
Virtual
Physical

Virtual Memory



How does address translation work?

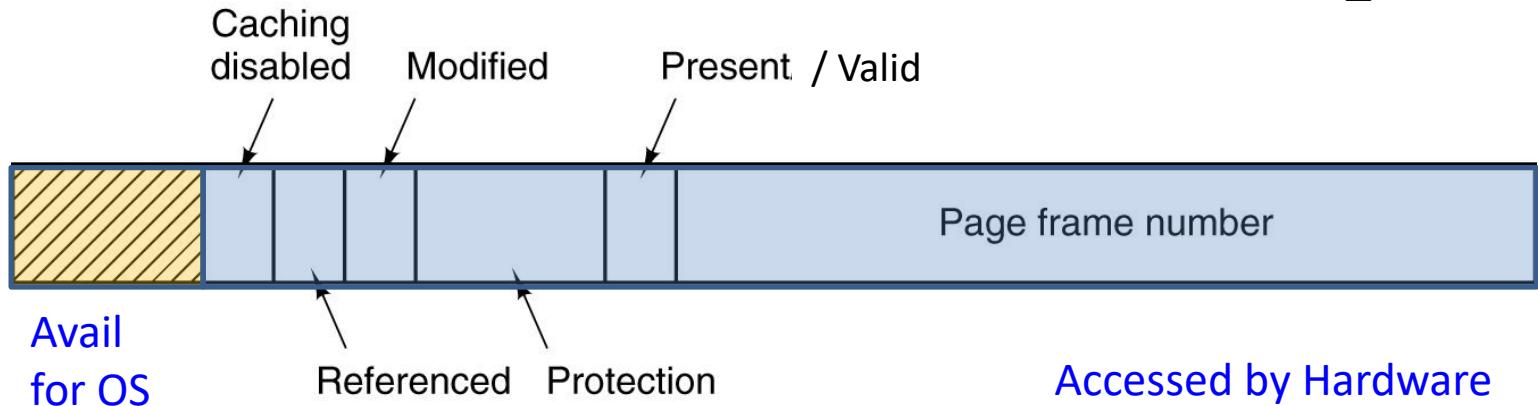




MMU = Memory Management Unit

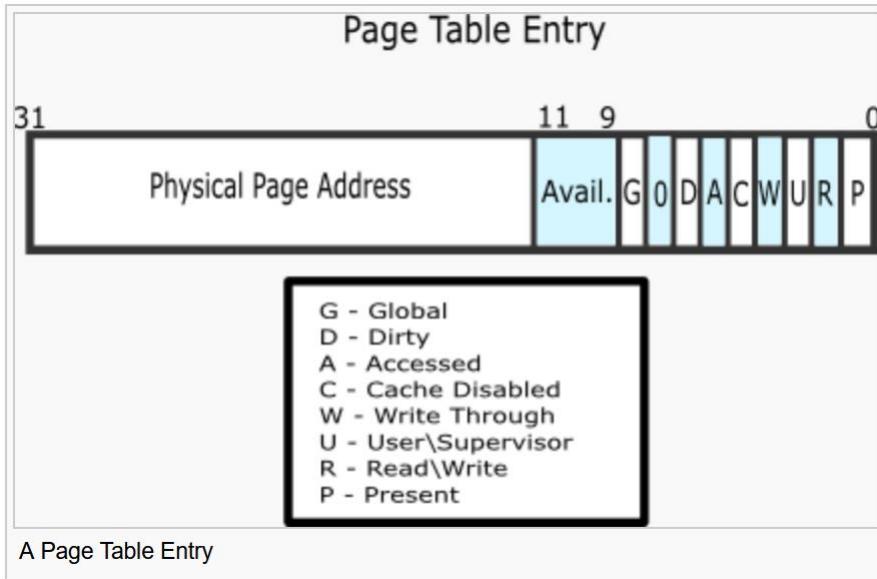
Structure of a Page Table Entry

```
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



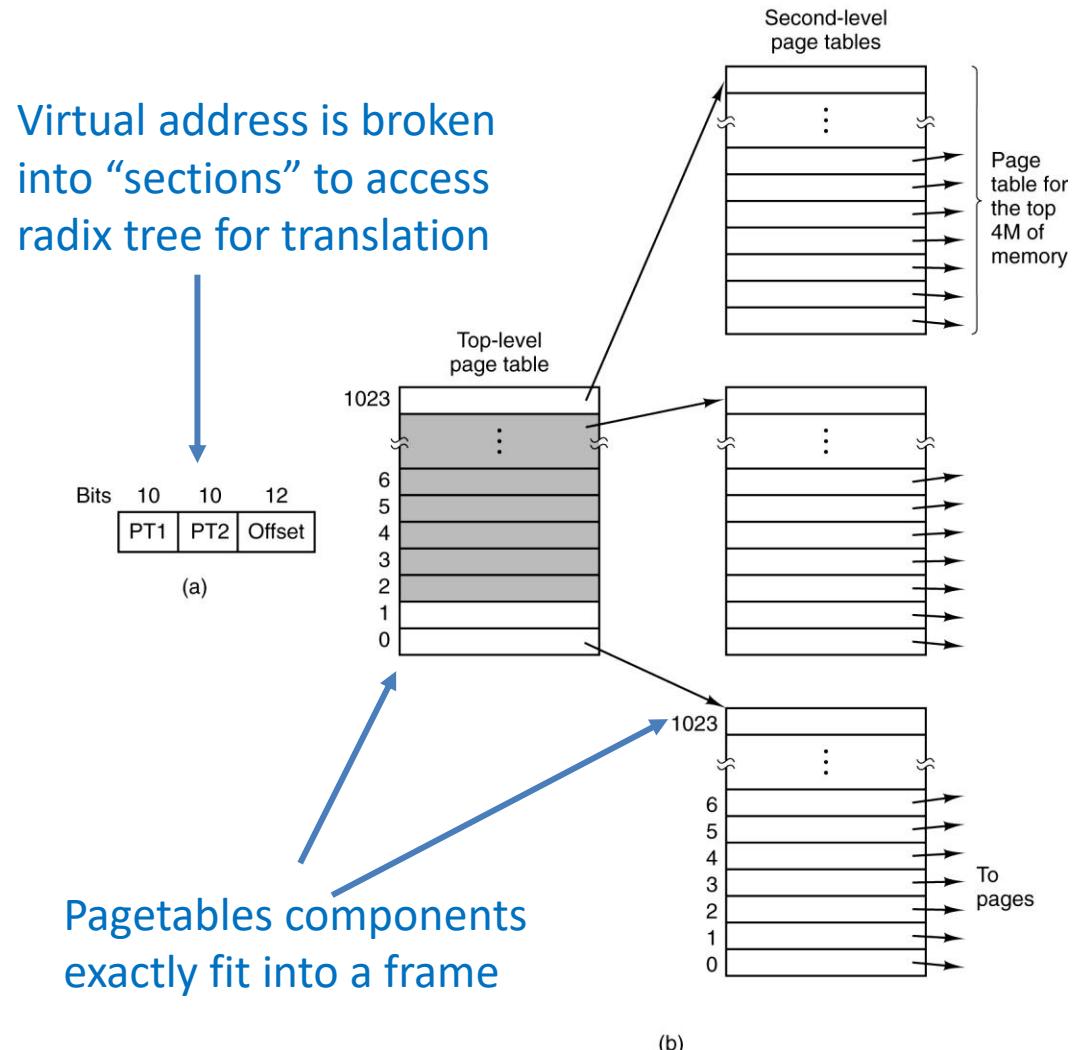
32-bit

```
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;
```

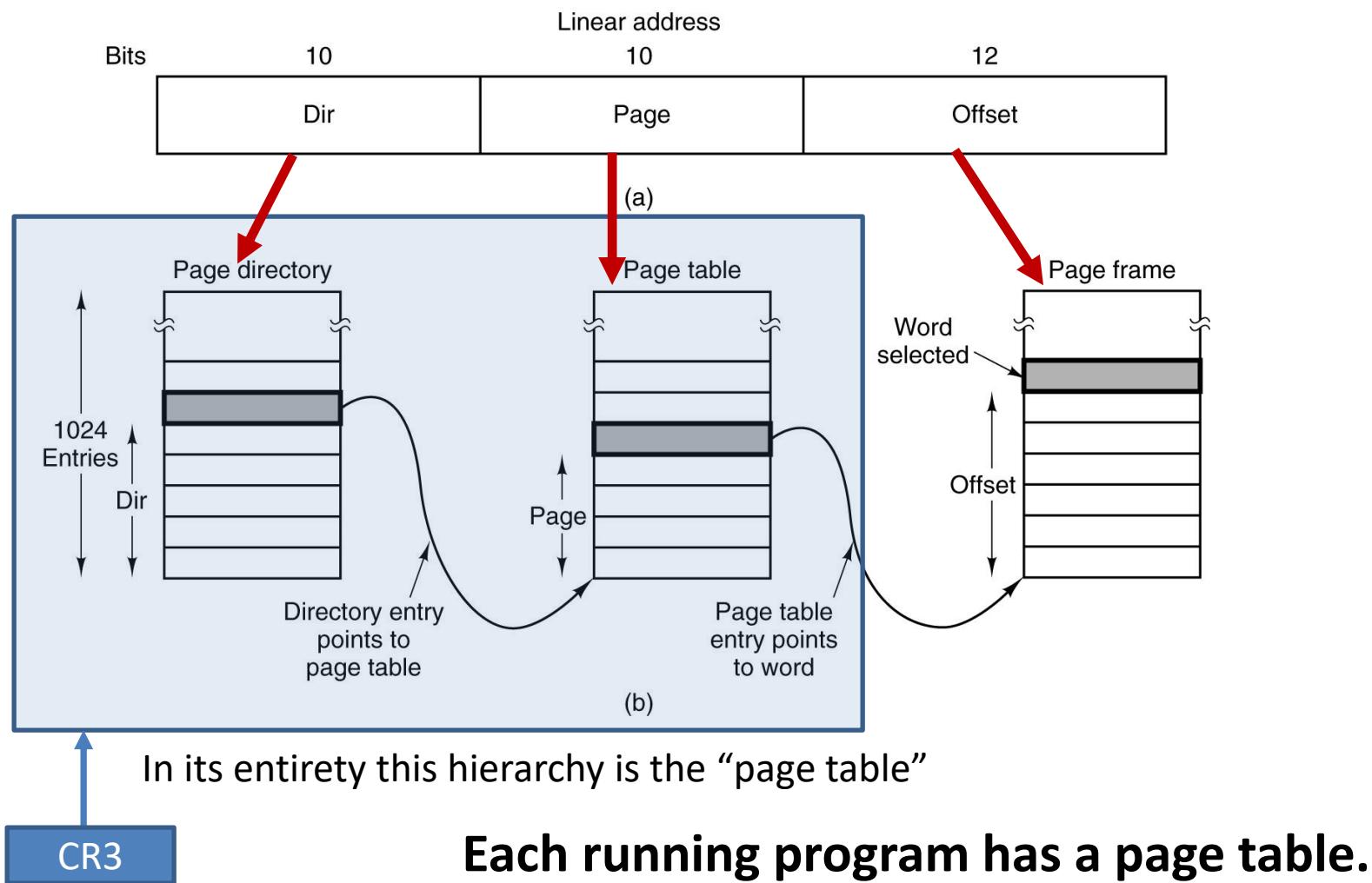
64-bit

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 2nd tables required !!!

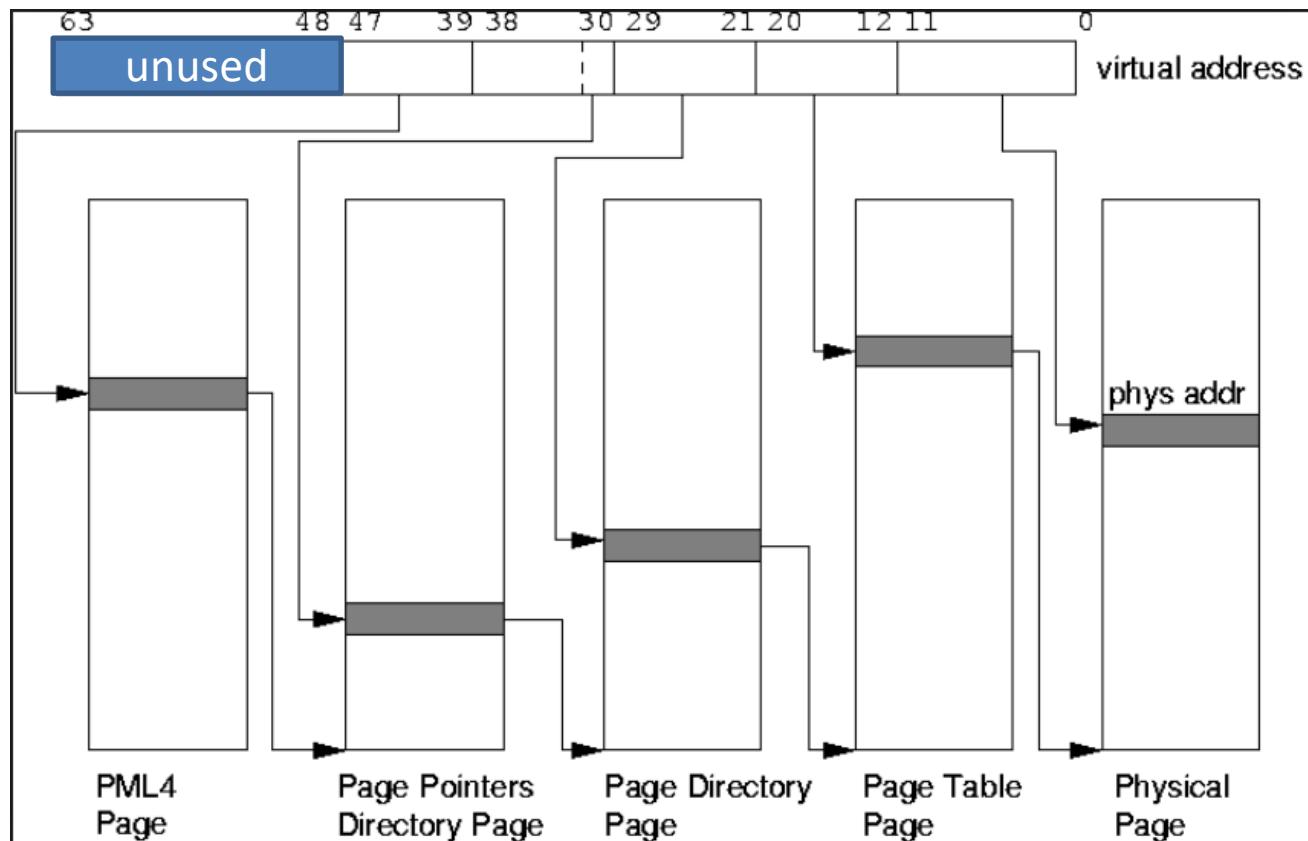


The Intel Pentium



CR3: Privileged hardware register

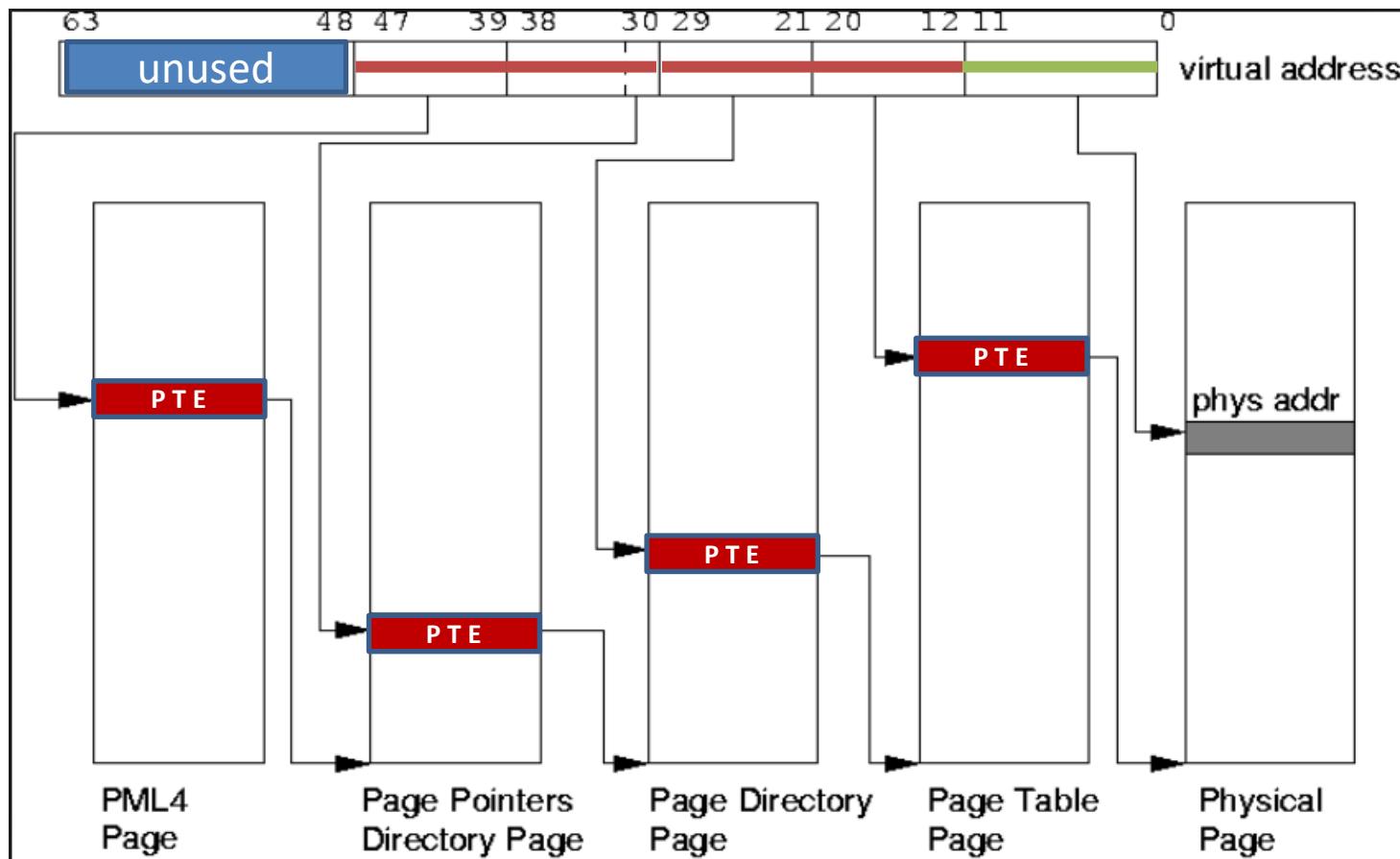
4 level PageTable for 64-bit arch



- 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

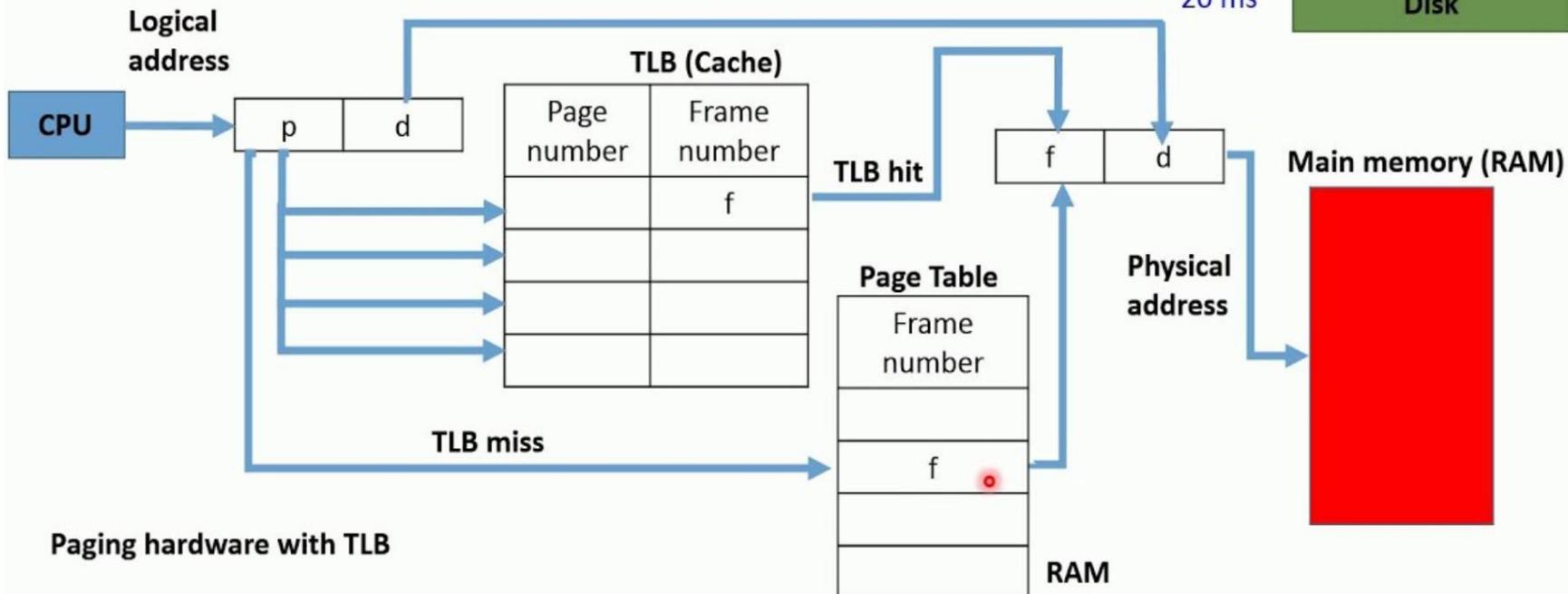
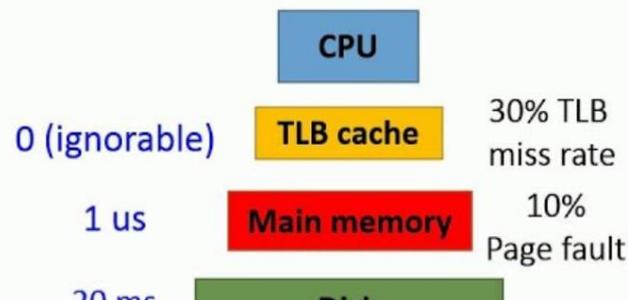
- **Observation:** most programs tend to make a large number of references to a small number of pages over a period of time -> only fraction of the page table is heavily used
(data and instruction locality !!!)
- TLB
 - Hardware cache inside the MMU
 - **Caches** PageTable translations (VA -> PA)
 - 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

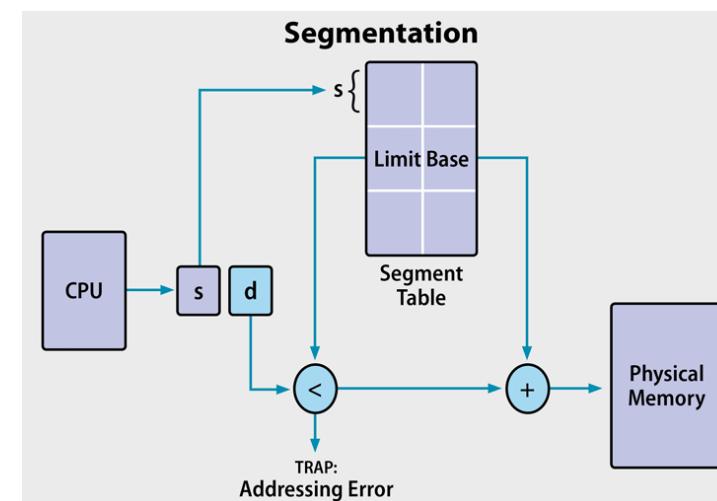
Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

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

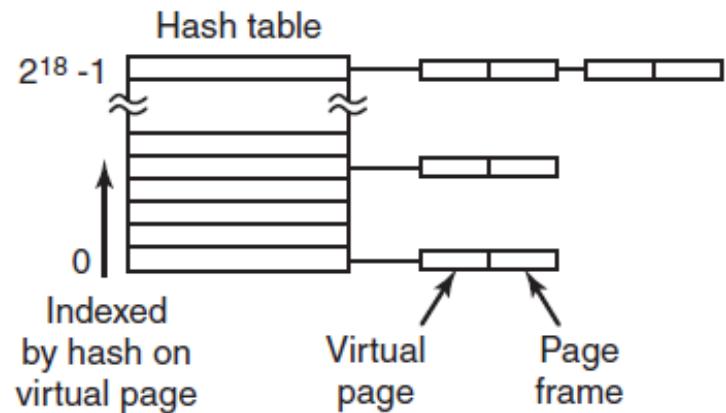
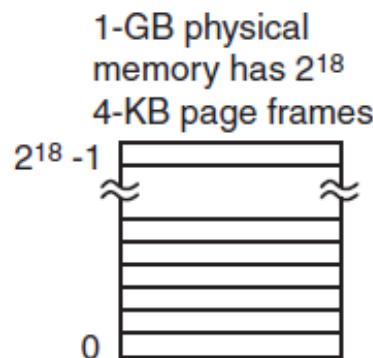
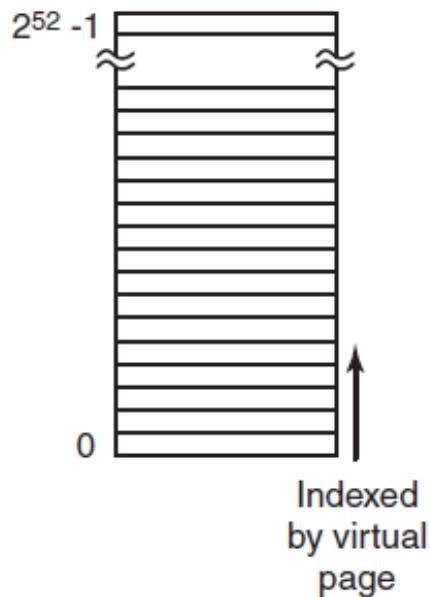
Other Translation Organizations

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



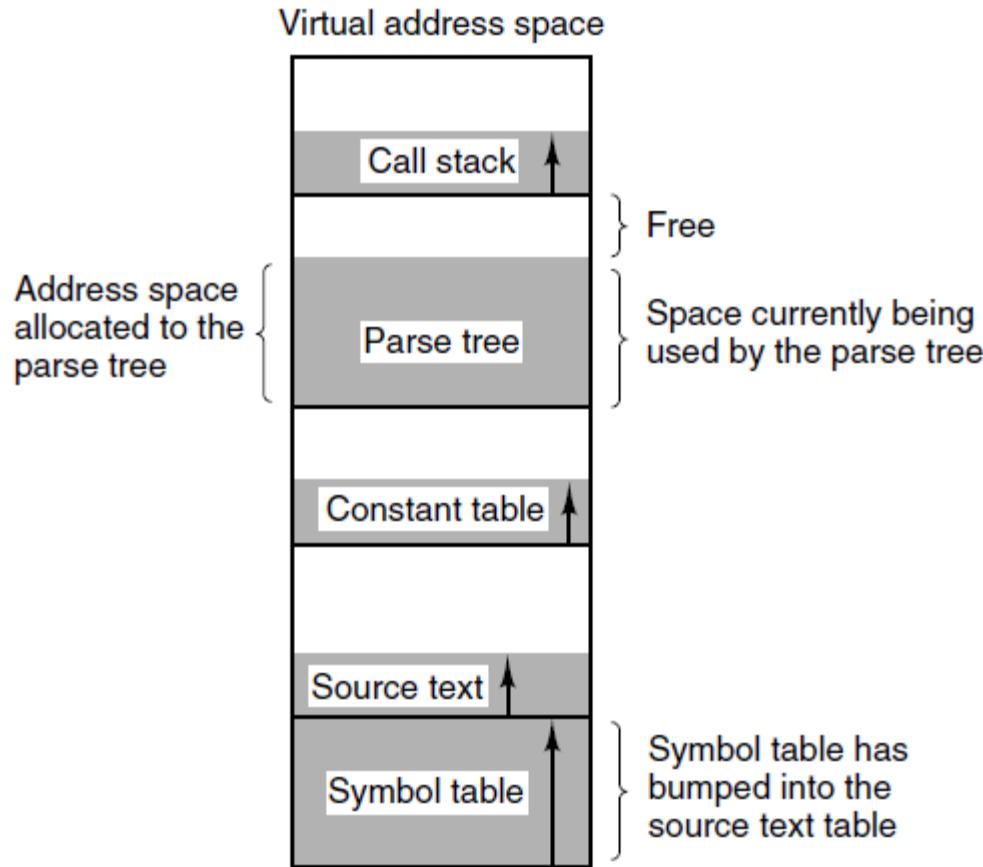
Inverted Page Tables

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)

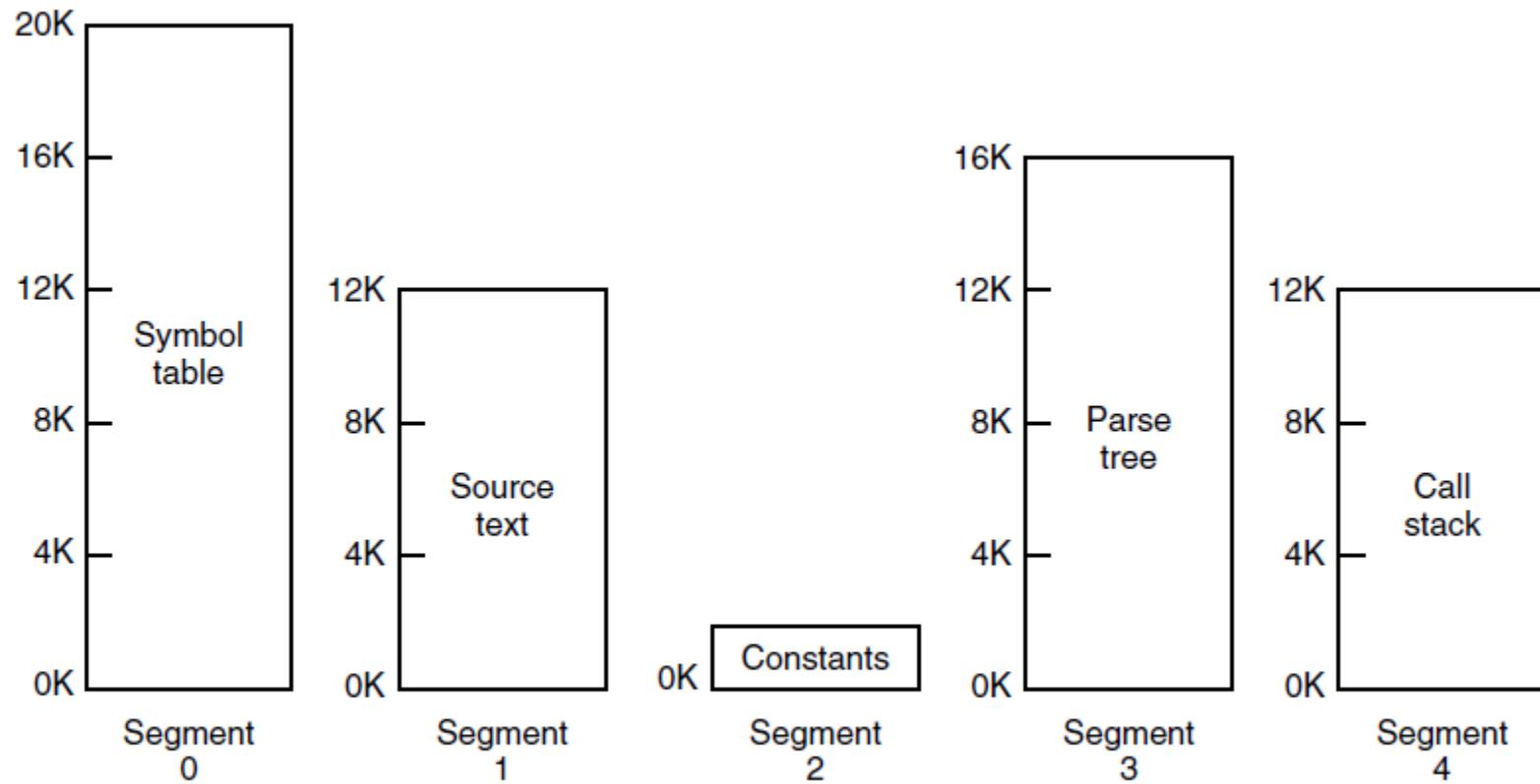


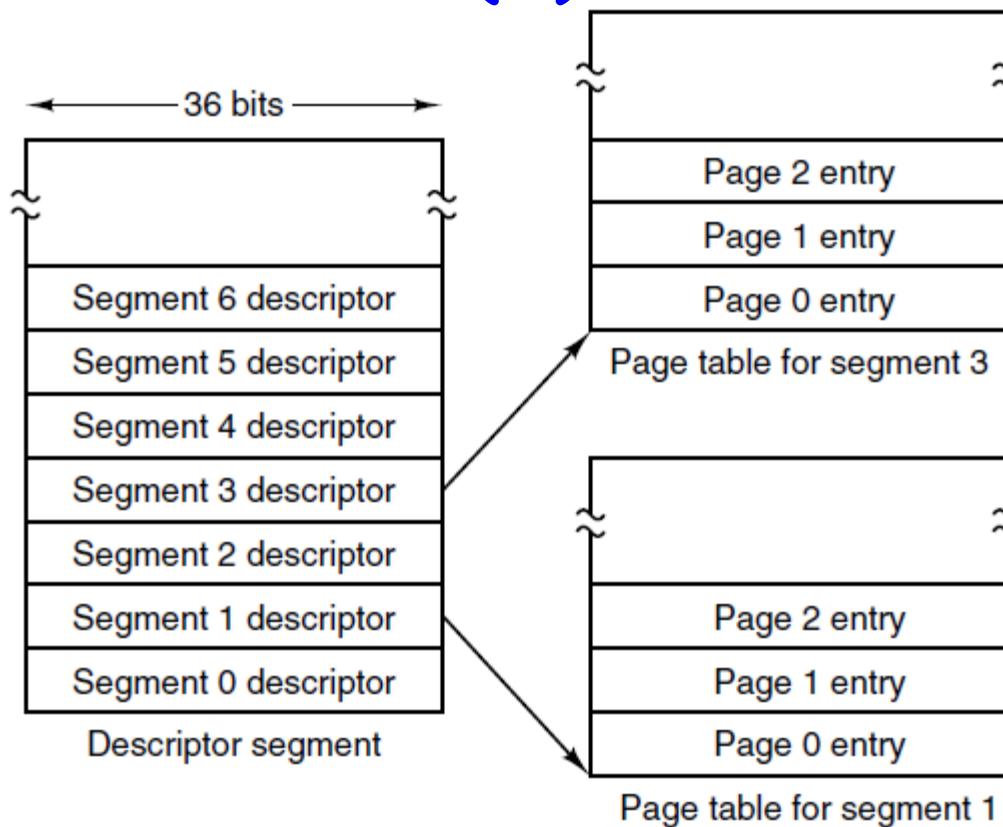
Figure 3-31. A segmented memory allows each table to grow or shrink independently of the other tables.

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

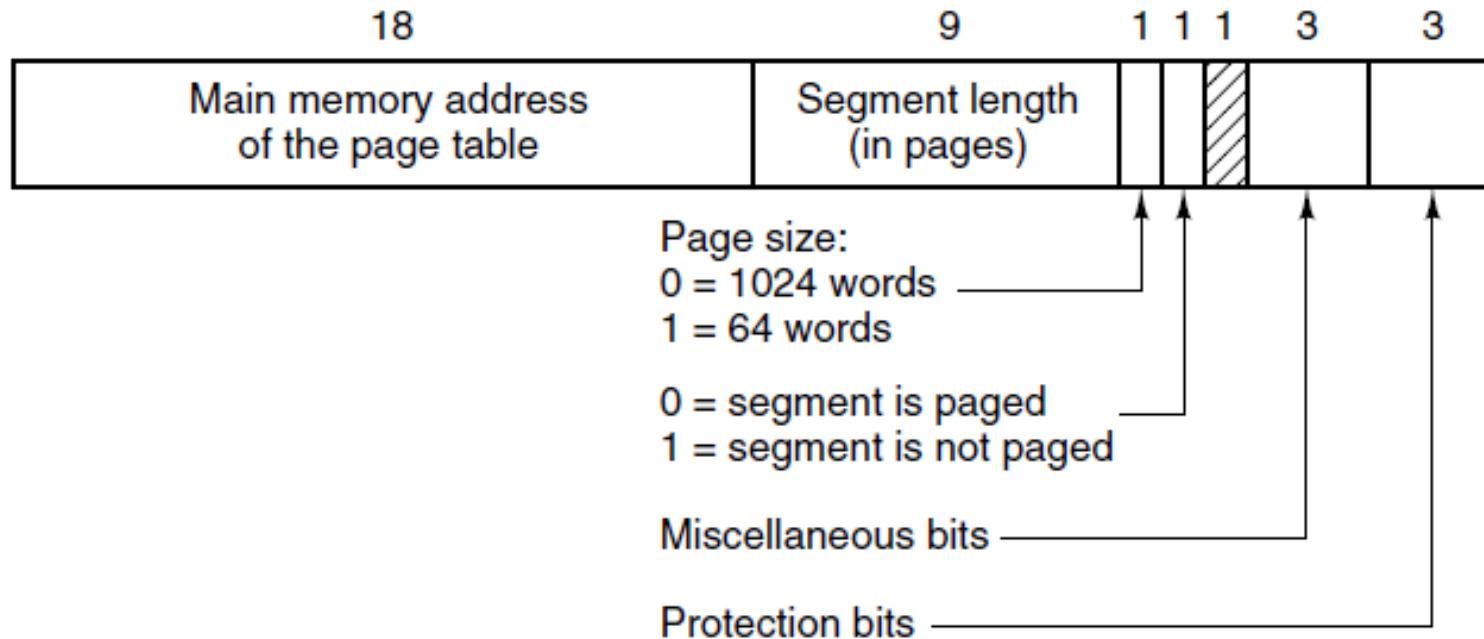
Comparison of paging and segmentation

Segmentation with Paging: MULTICS (1)



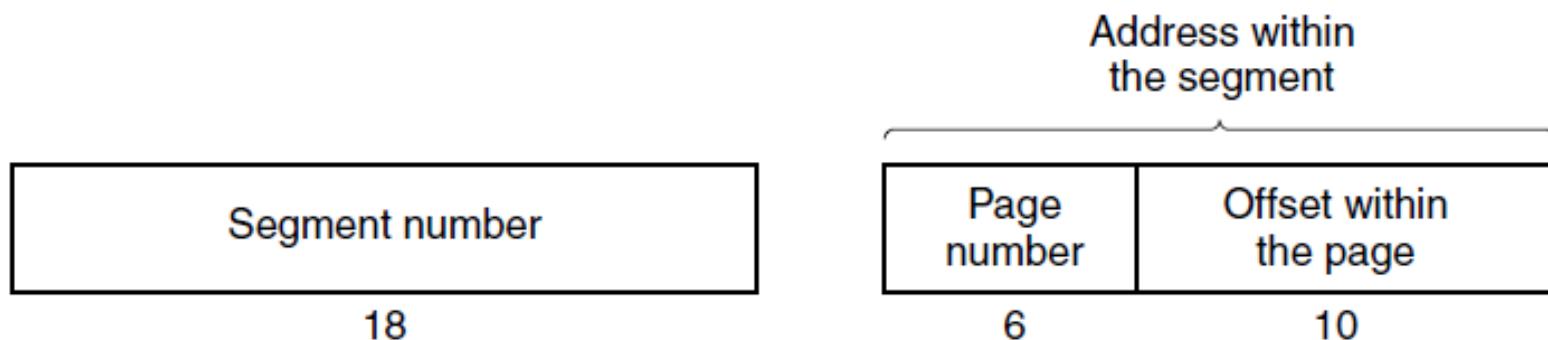
The MULTICS virtual memory. (a) The descriptor segment pointed to the page tables.

Segmentation with Paging: MULTICS (2)



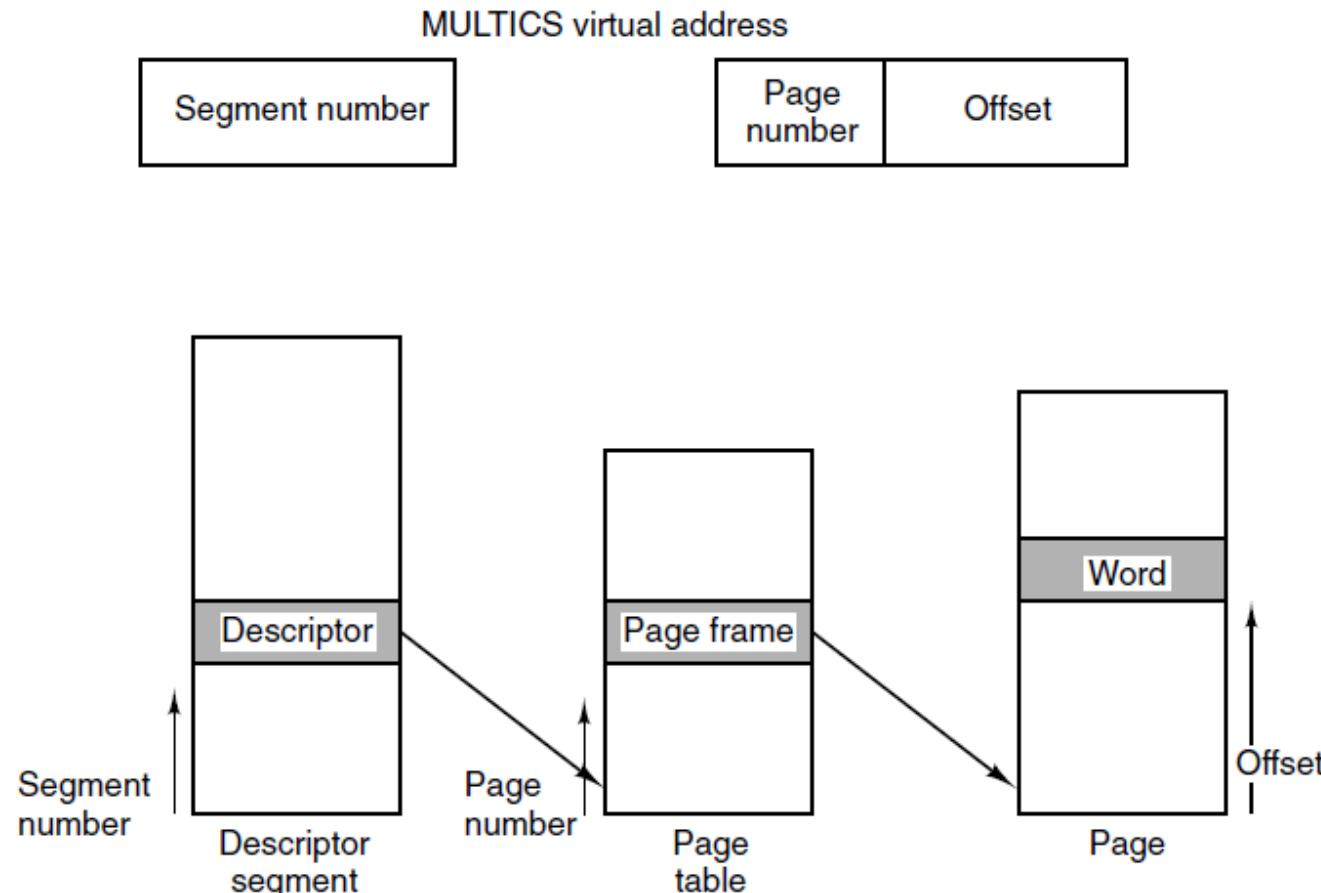
The MULTICS virtual memory. (b) A segment descriptor. The numbers are the field lengths.

Segmentation with Paging: MULTICS (3)



A 34-bit MULTICS virtual address.

Segmentation with Paging: MULTICS (4)



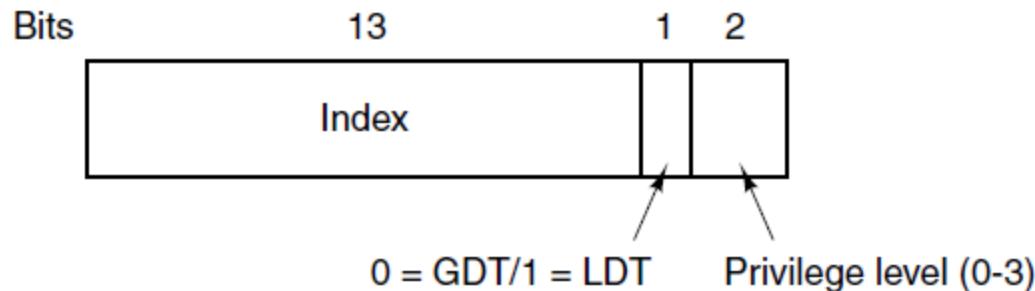
Conversion of a two-part MULTICS address into a main memory address.

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

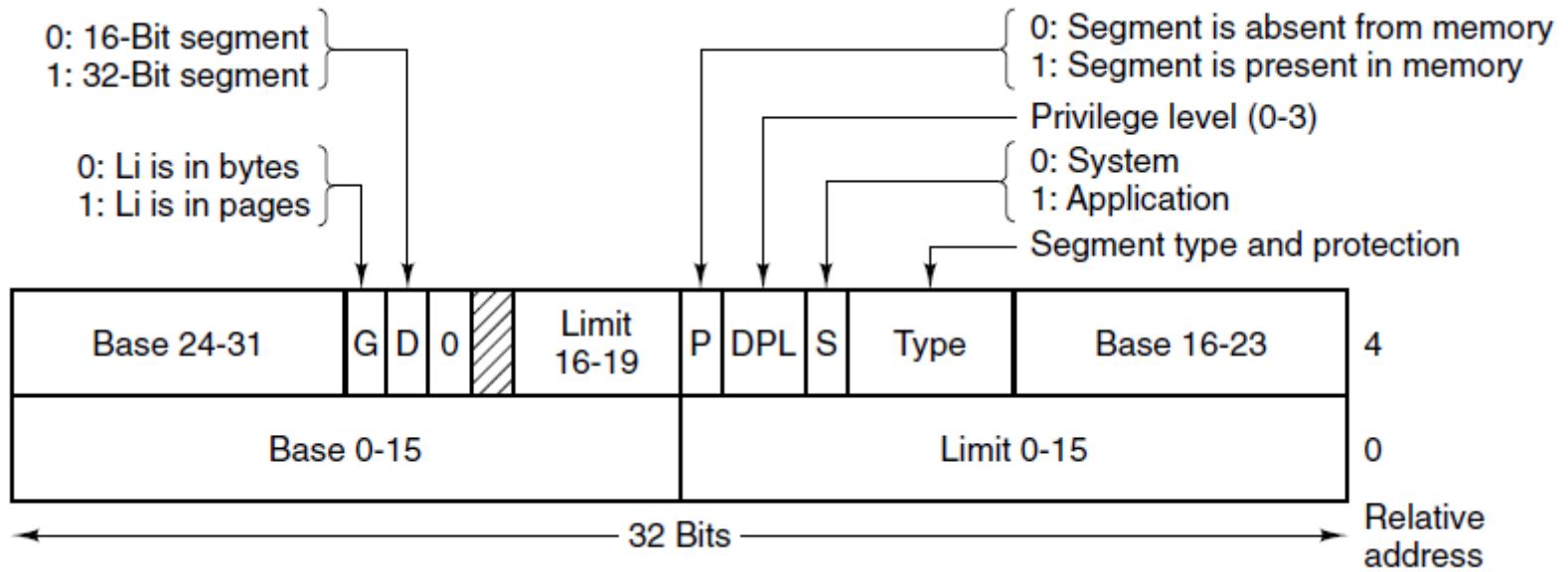
A simplified version of the MULTICS TLB. The existence of two page sizes made the actual TLB more complicated.

Segmentation with Paging: The Intel x86 (1)



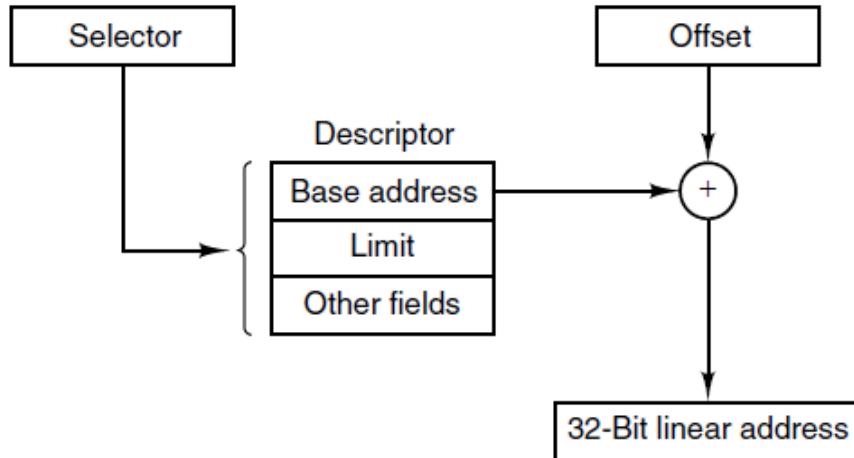
An x86 selector.

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**.

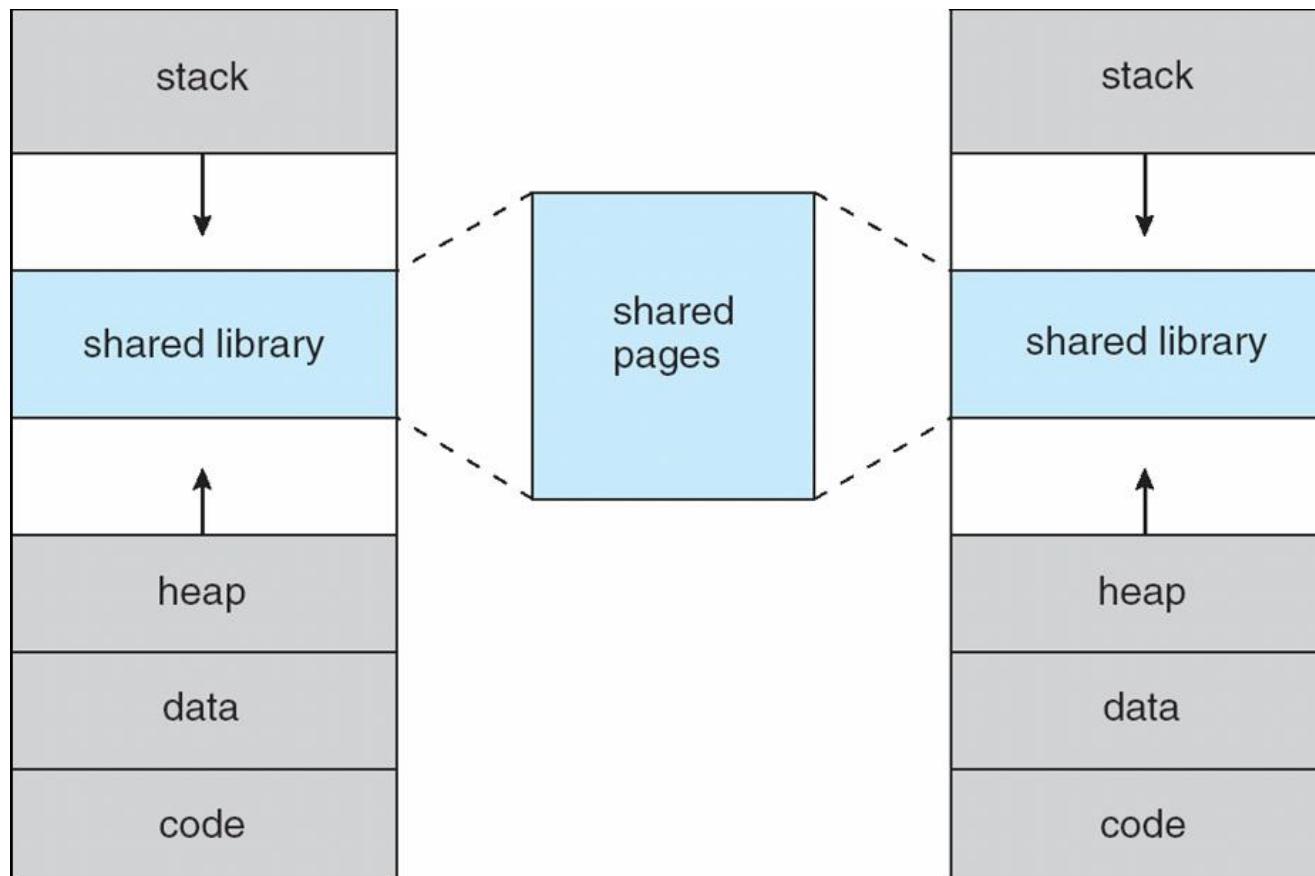
```
mov dword [fs:eax], 42
```

```
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



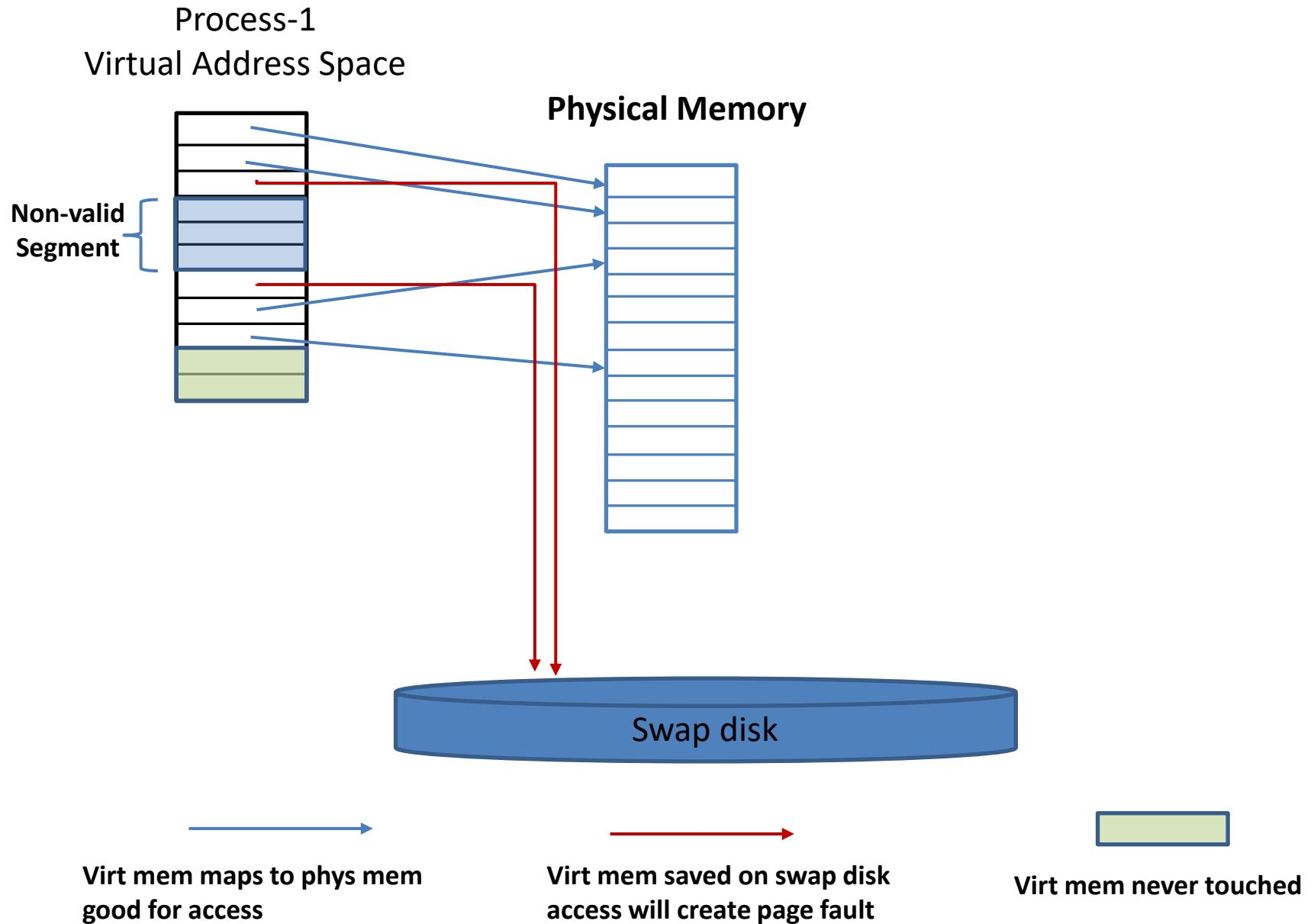
Copy-on-Write

- **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

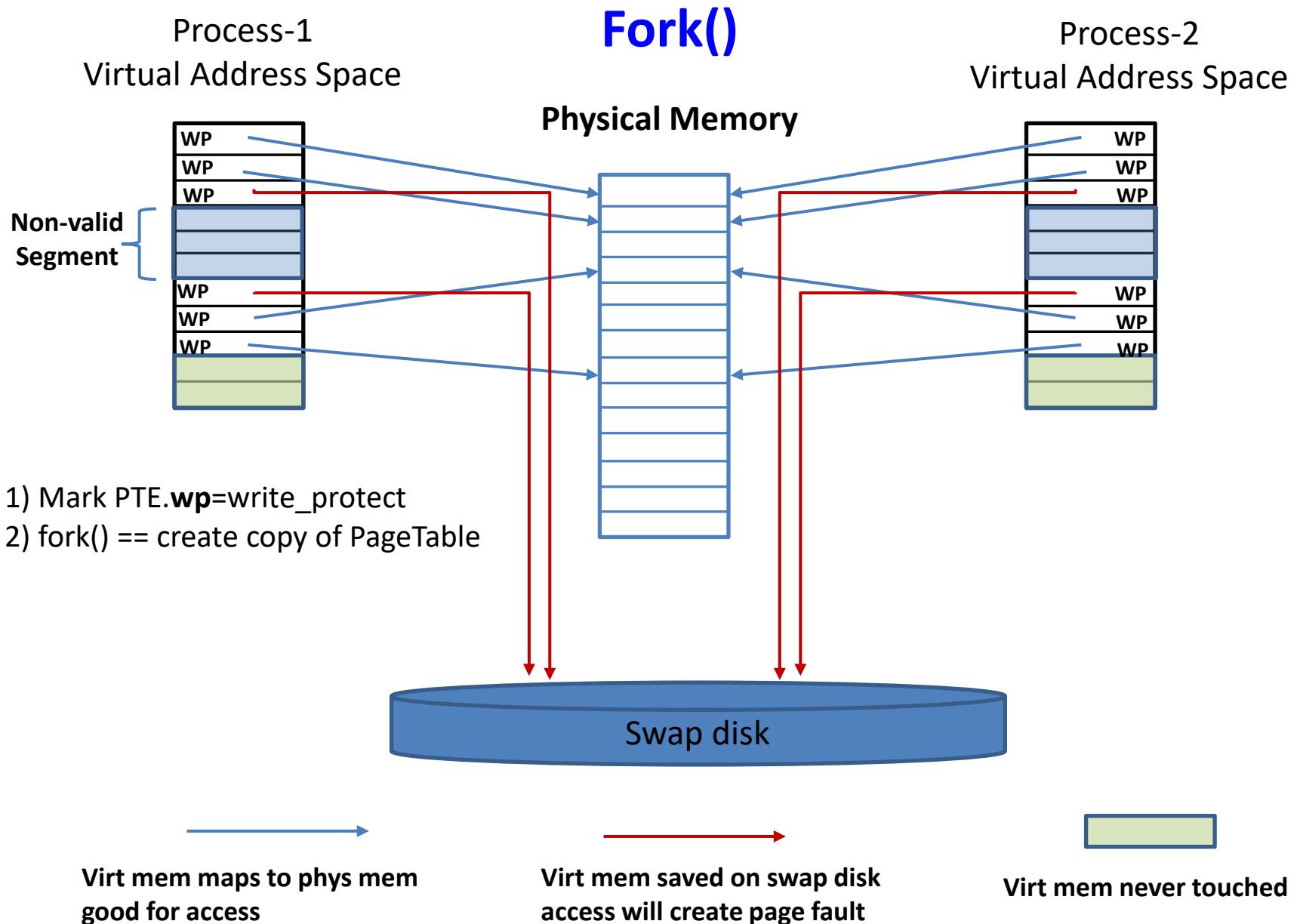
Copy-on-Write (cont)

- 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

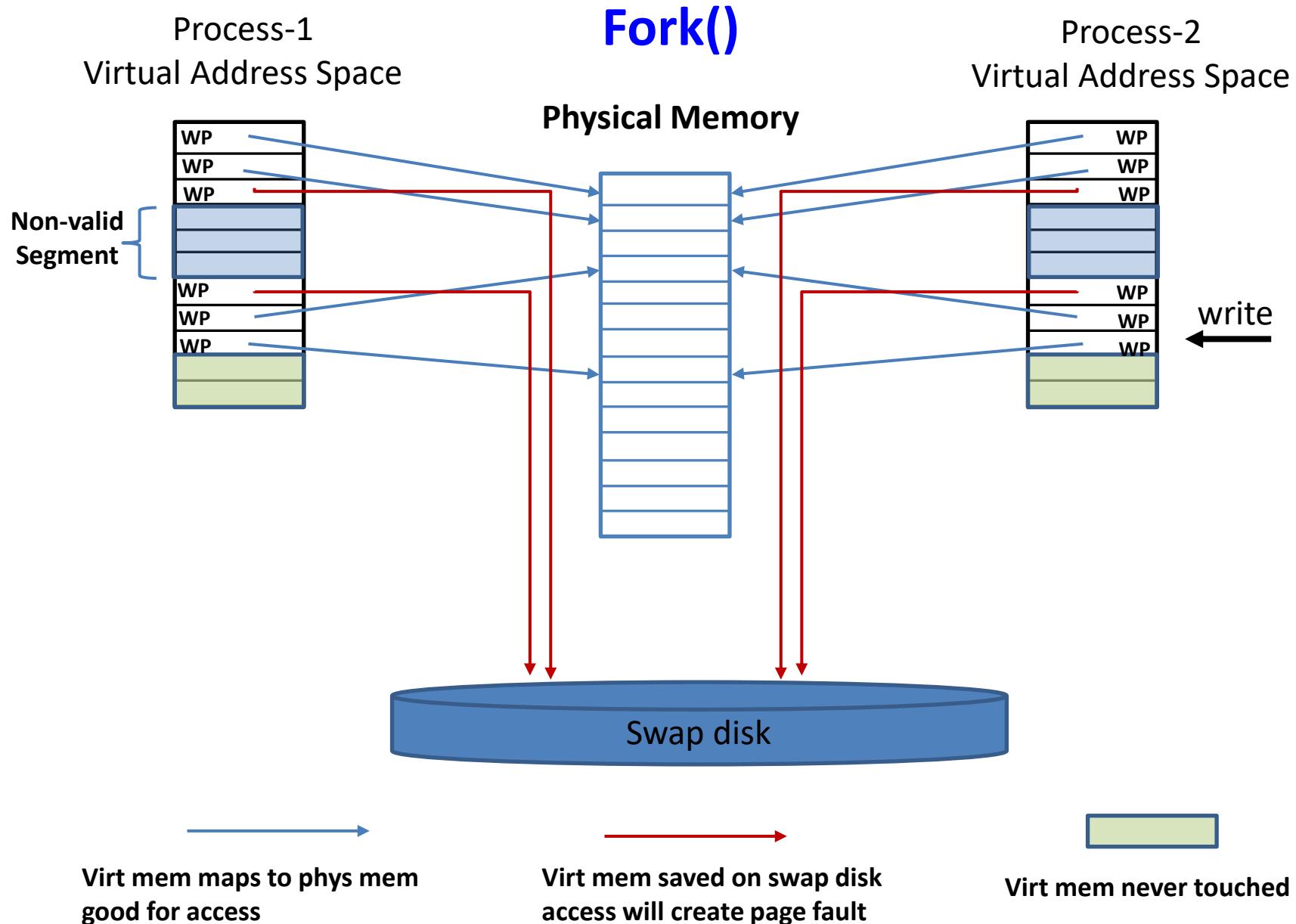
Virtual Memory



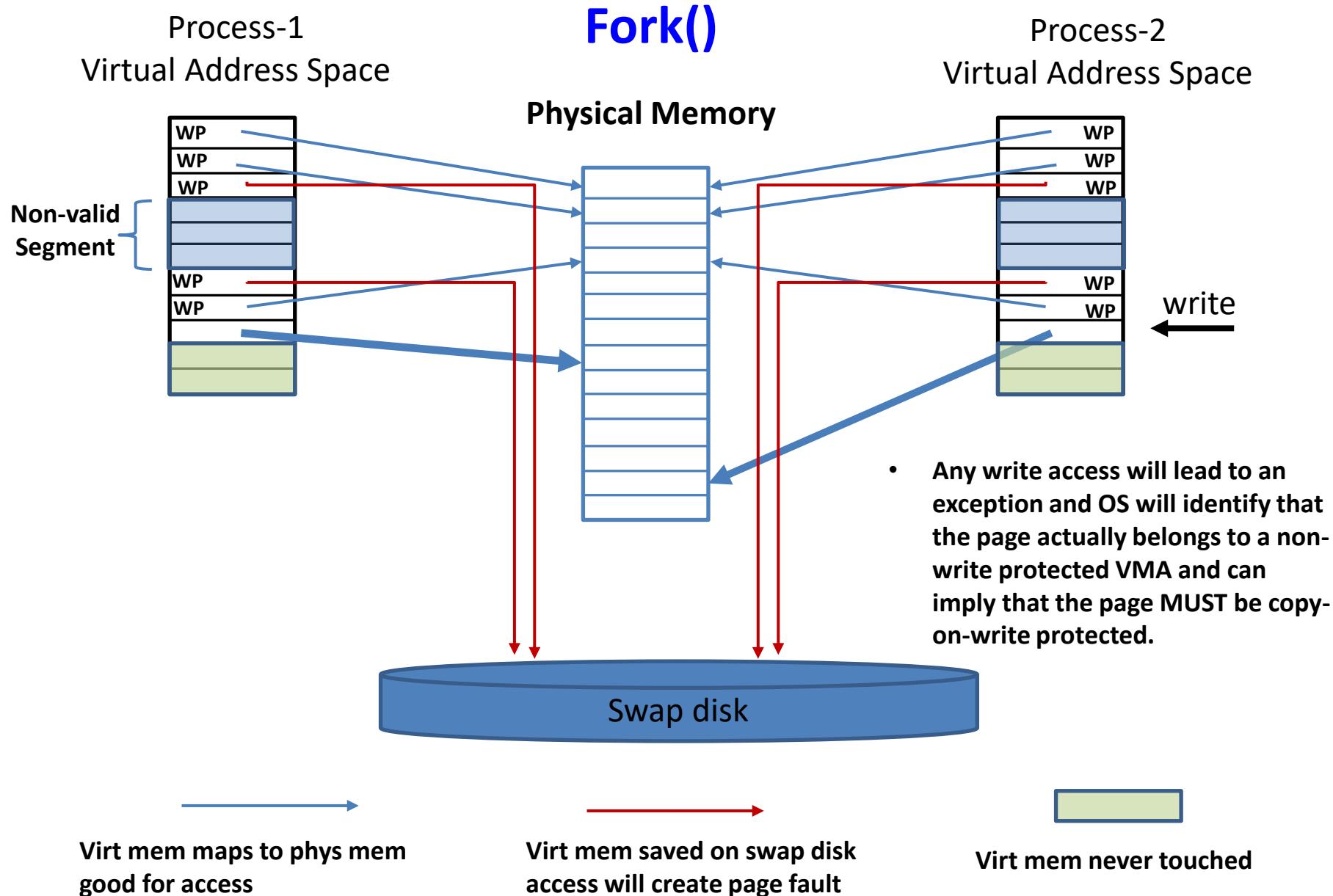
Virtual Memory



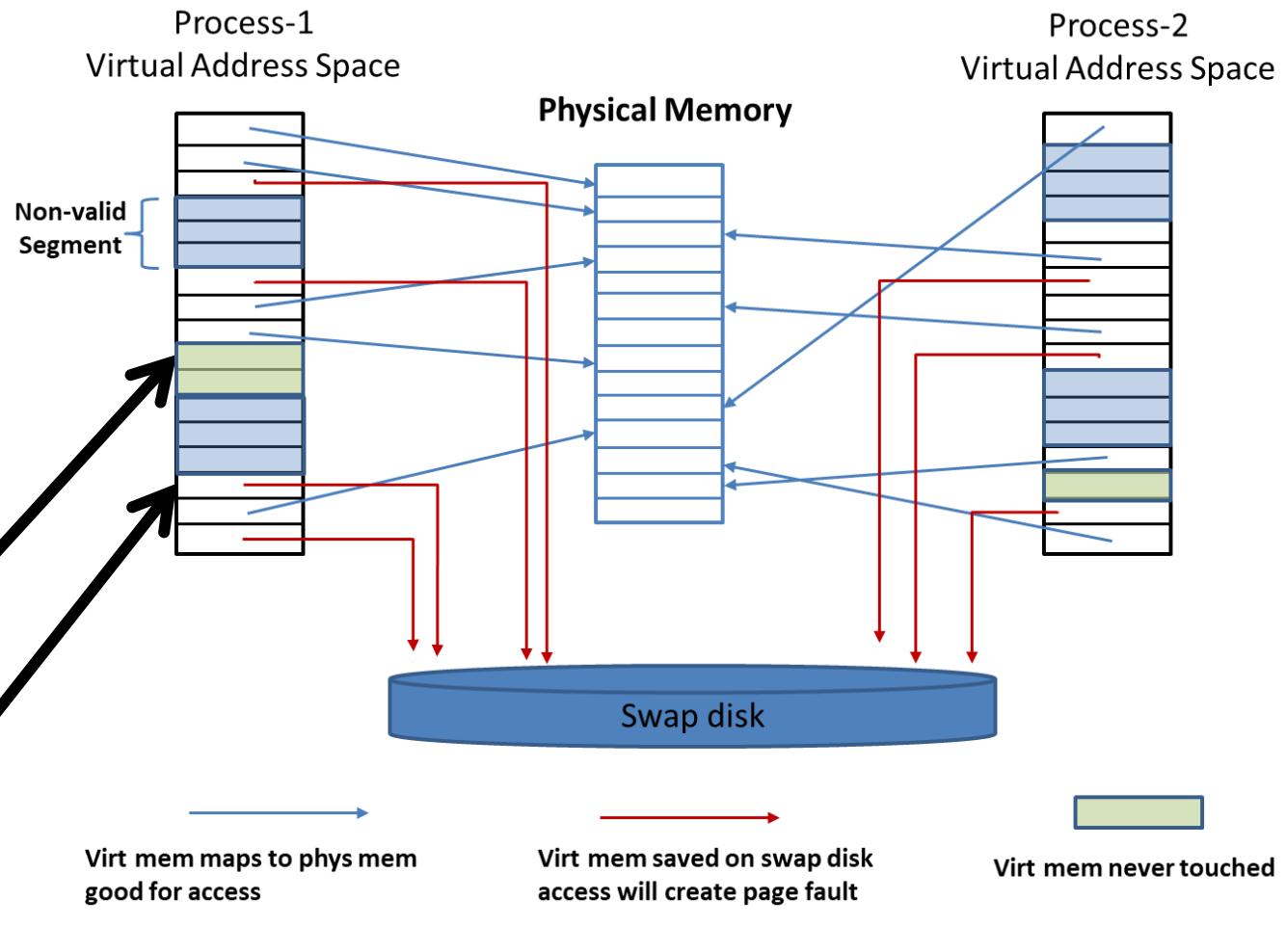
Virtual Memory (COW)



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 .. \rightarrow 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

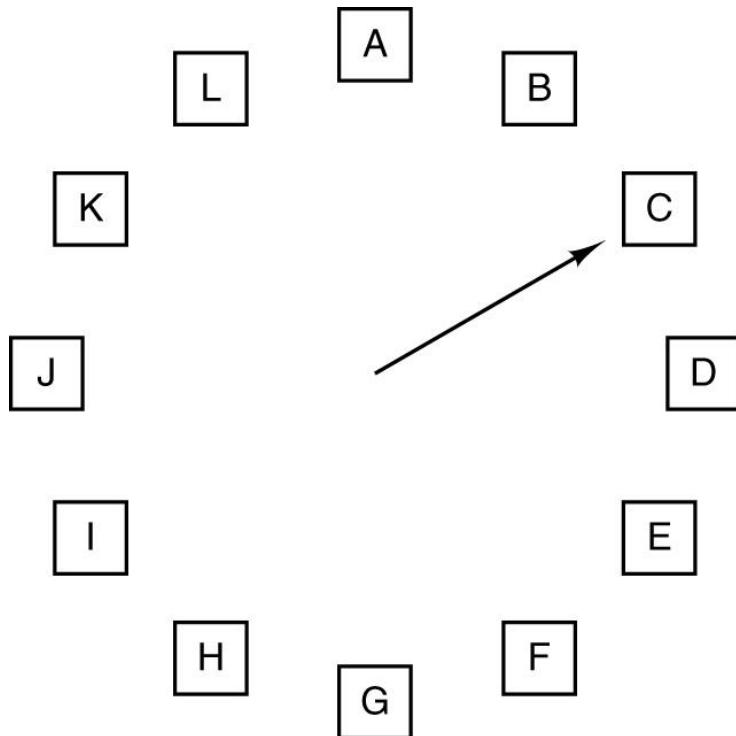
The Second-Chance Page Replacement Algorithm

- 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 2nd-Chance

The Clock Page Replacement Policy

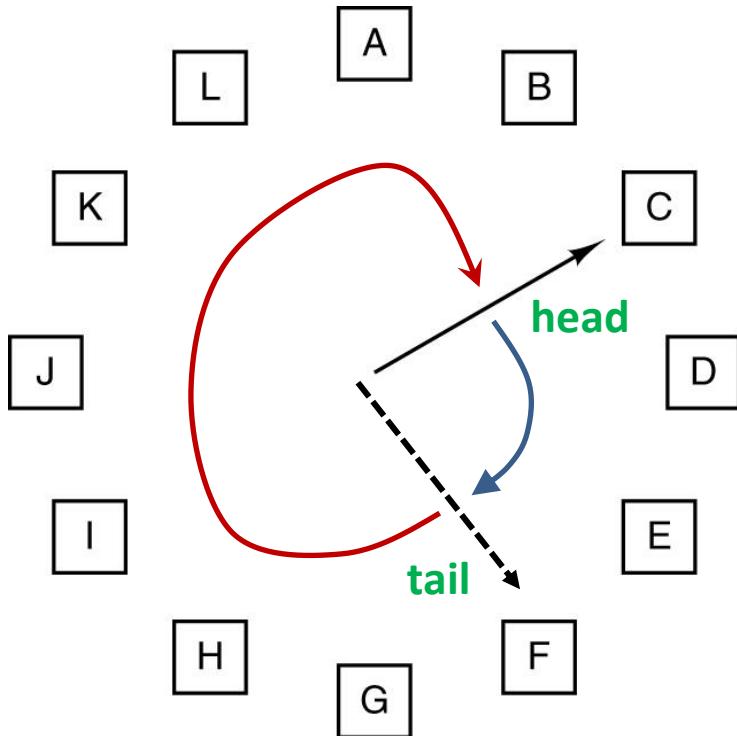


When a page fault occurs,
the page the hand is
pointing to is inspected.
The action taken depends
on the R bit:

- 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.
 - After each access, the counter of the accessed entry
 - At page fault, the page with lowest value is discarded
- Too expensive!
- Too Slow!

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

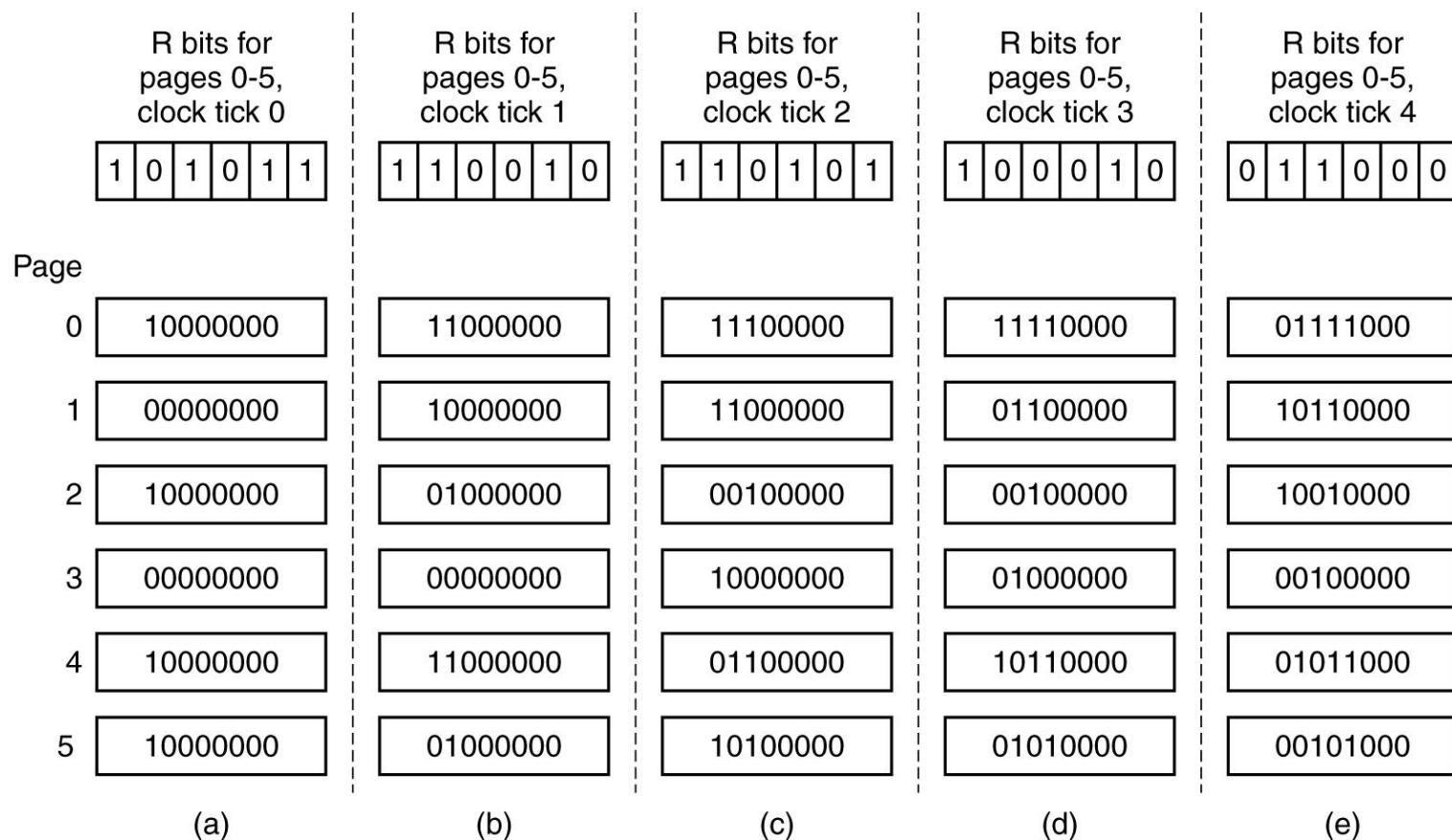
- Not Frequently Used (NFU) algorithm
- Software counter associated with each page, initially zero
- At some periodicity (e.g. a second), the OS scans all page table entries and adds the R bit to the counter of that PTE
- At page fault: the page with lowest counter is replaced

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

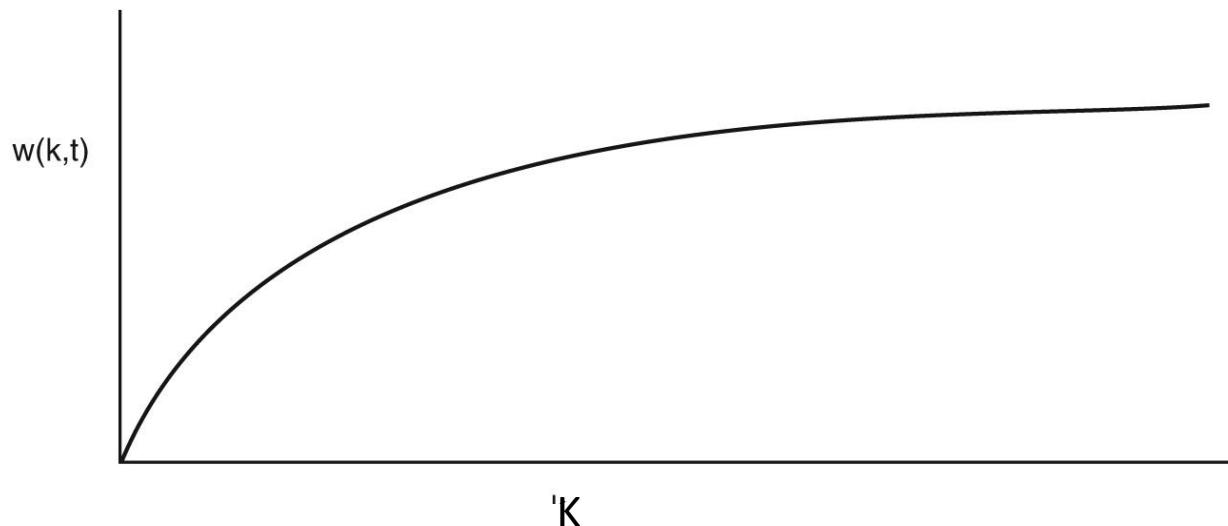
- **Working set**: the set of pages that a process is currently using
- **Thrashing**: a program causing page faults at high rates (e.g. pagefaults/instructions metric)

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.

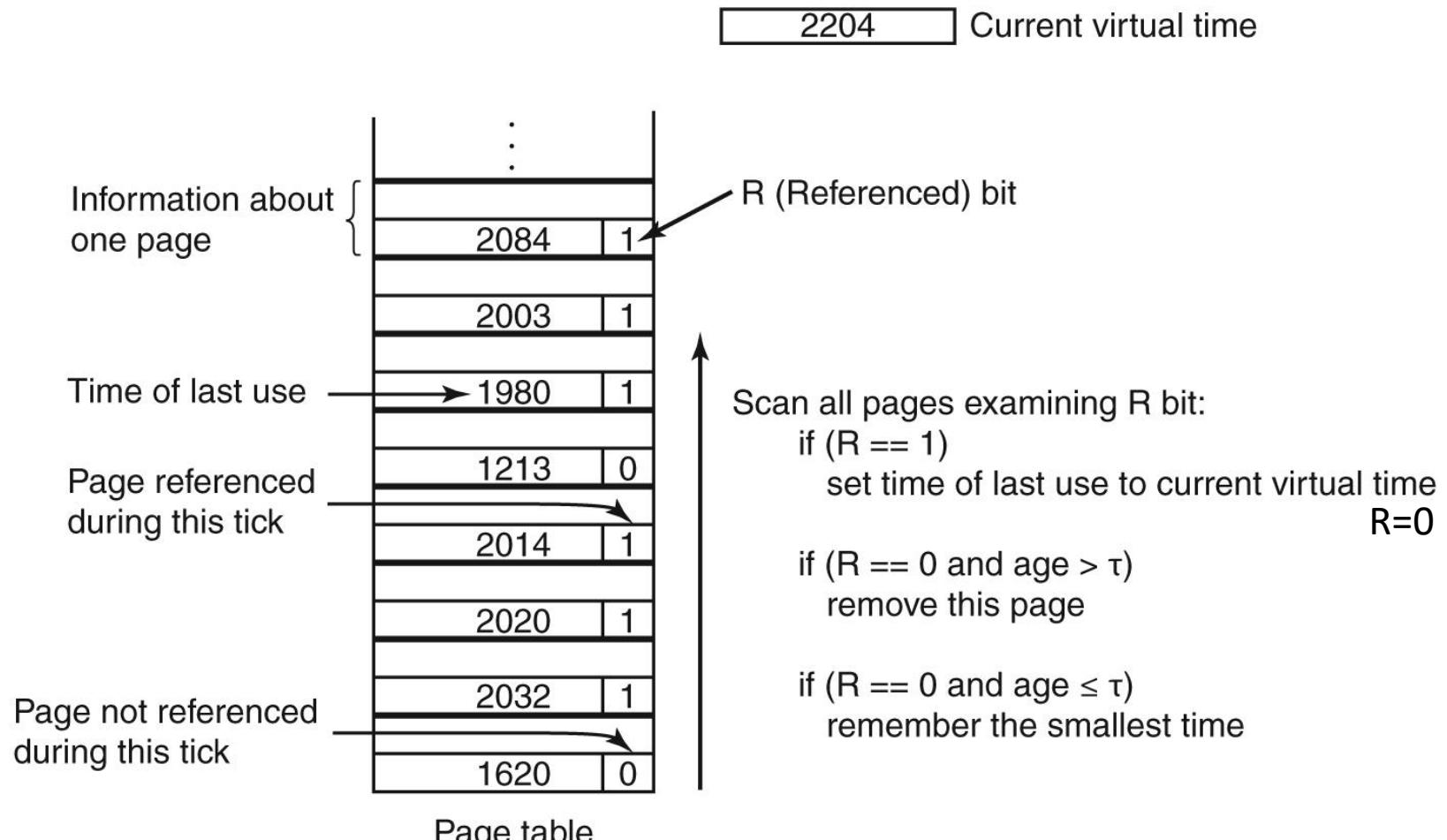


$w(k,t)$: the set of pages accessed in the last k references at instant t

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

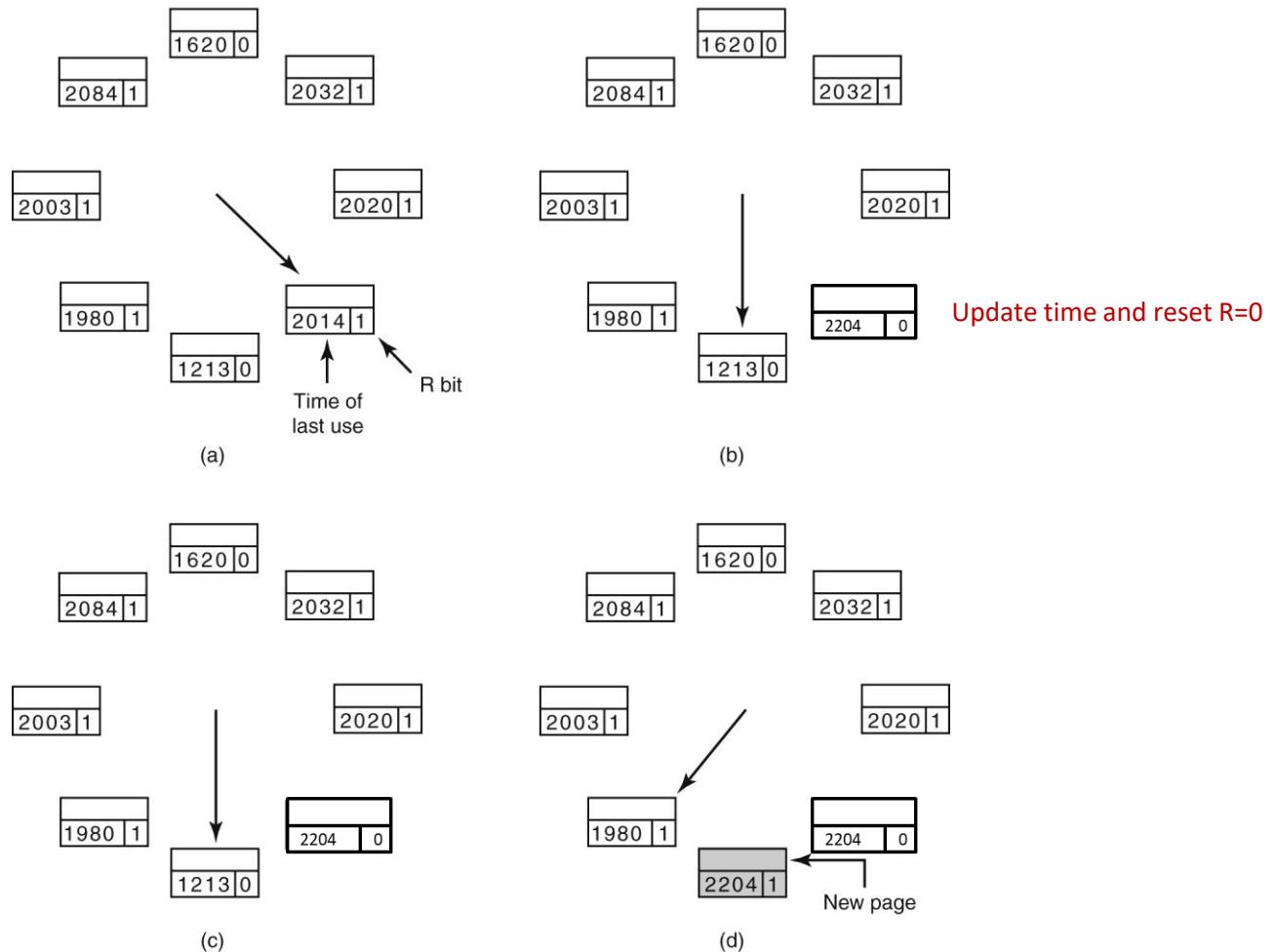
The WSClock Page Replacement Algorithm (specific implementation of Working Set Replacement)

- 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



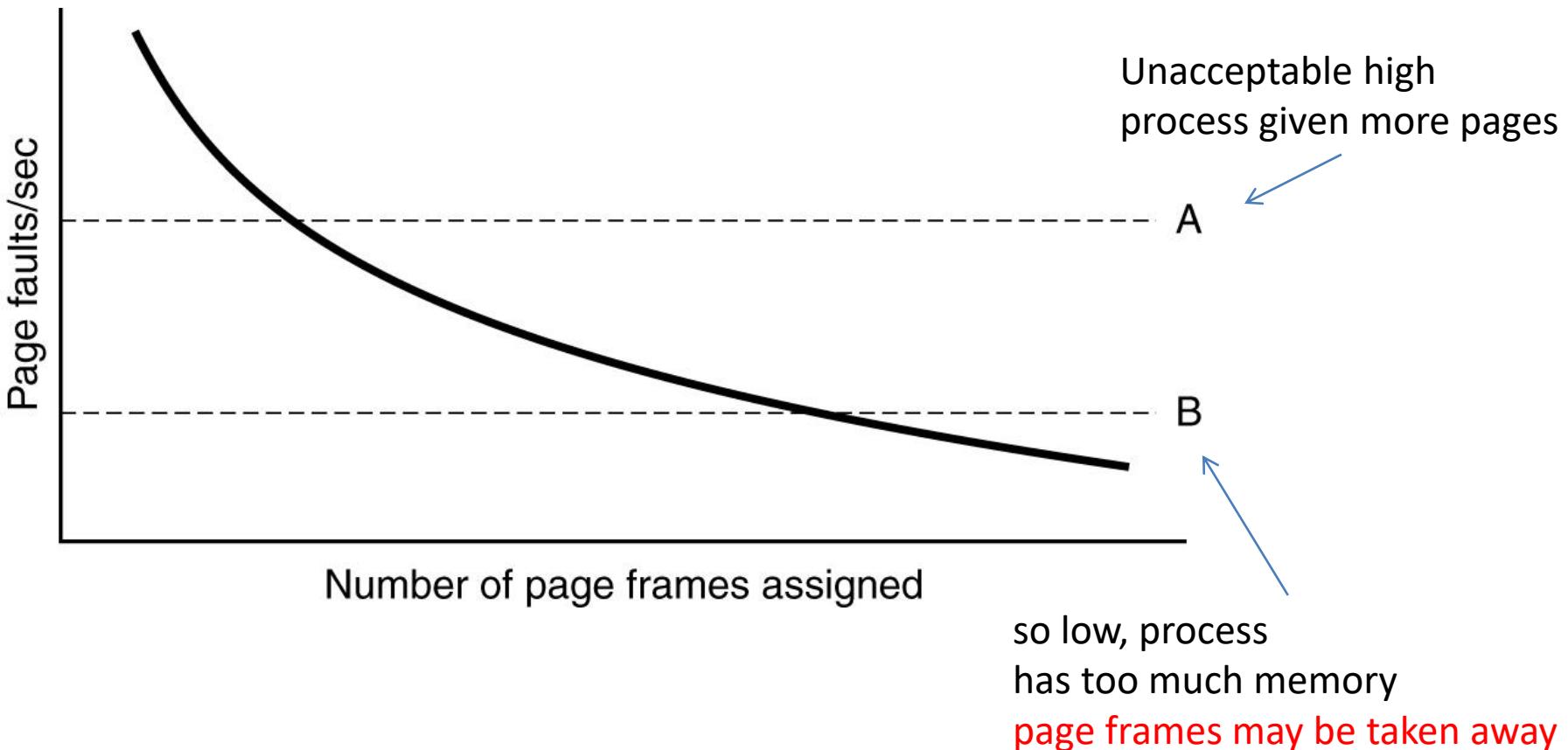
Design Issues for Paging: Local vs Global Allocation

- 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**

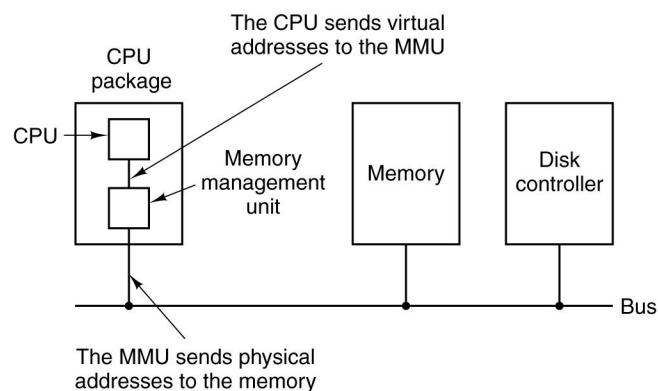
Design Issues: Cleaning Policy

- 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.

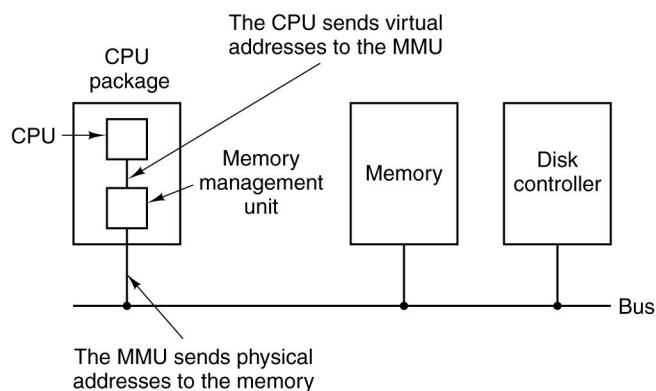
Summary of Paging

- 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

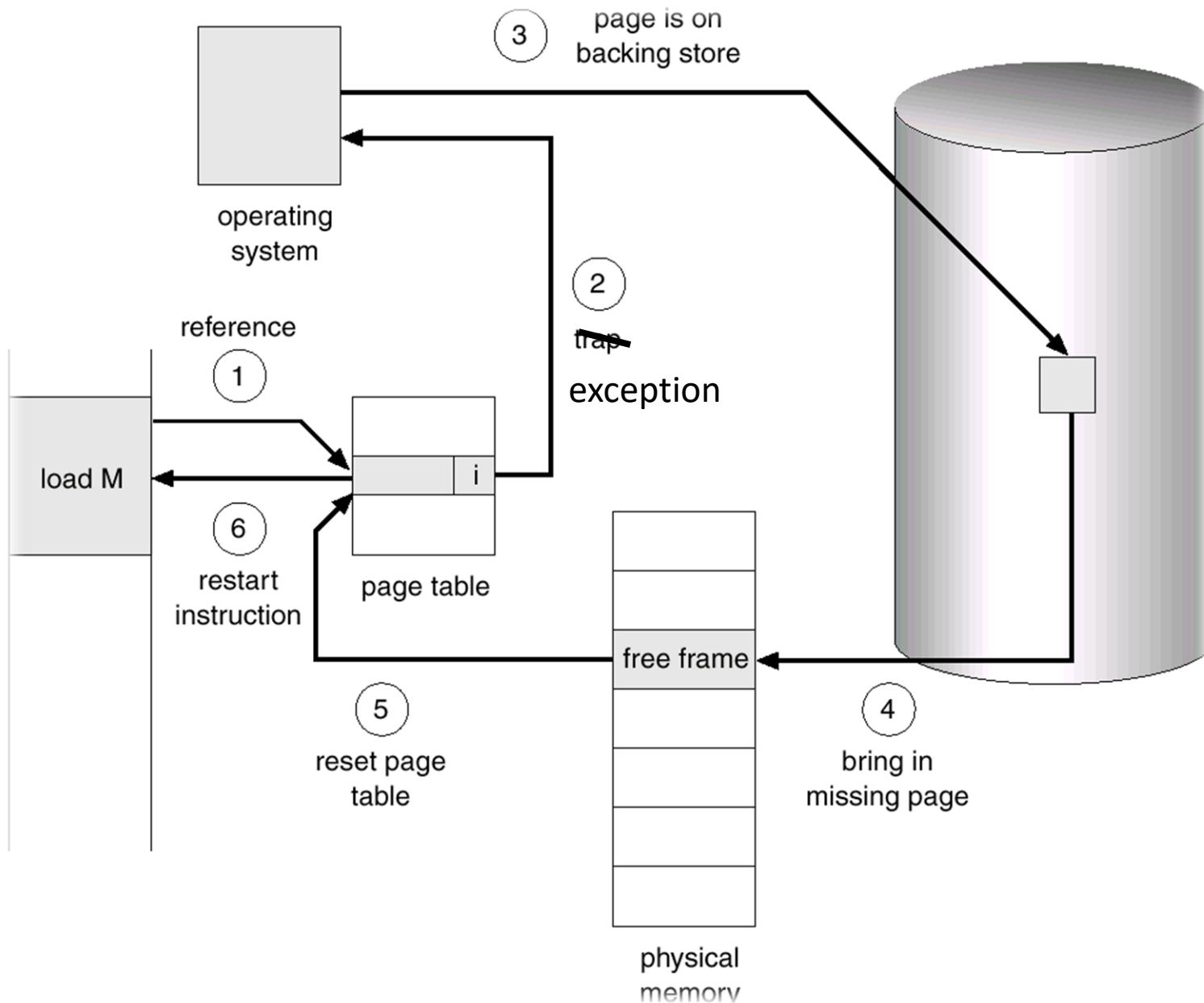
- When a new process is created
- When a process is scheduled for execution
 - TLB flushed for current process
 - MMU reset for the process
 - Process table of next process made current
- When process exits
- When page fault occurs

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)

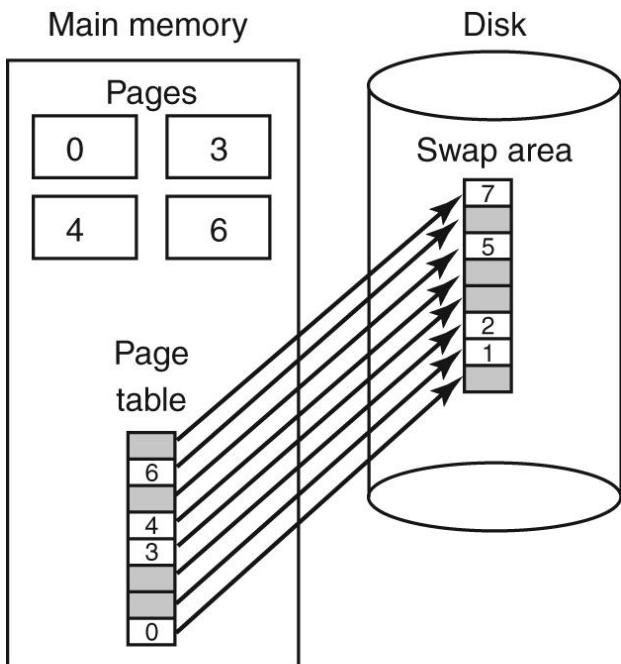
- Process issues a `syscall()` to read a file into a buffer
- Process suspended while waiting for I/O
- New process starts executing
- This other process gets a page fault
- If paging algorithm is global there is a chance the page containing the buffer could be removed from memory
- The I/O operation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

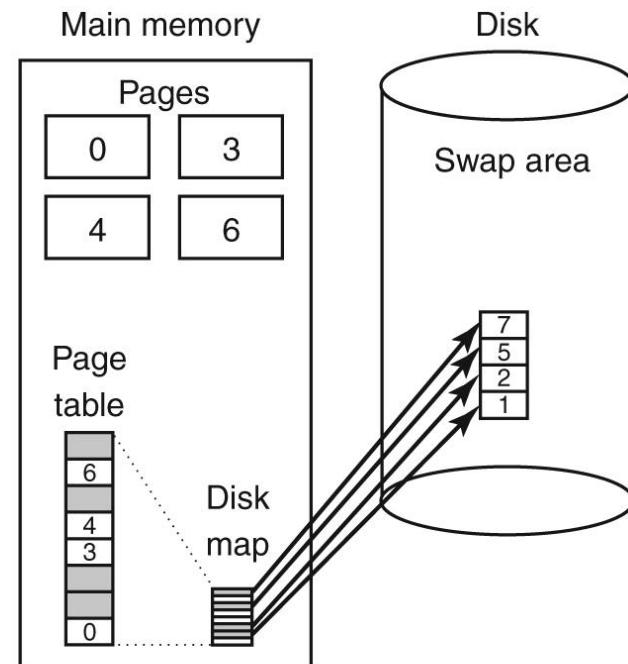
- Swap area: not a normal file system on it.
- Associates with each process the disk address of its swap area; store in the process table
- Before process starts swap area must be initialized
 - One way: copy all process image into swap area [**static swap area**]
 - Another way: don't copy anything 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



(a)

Static Swap Area



(b)

Dynamic

Page Fault Handler

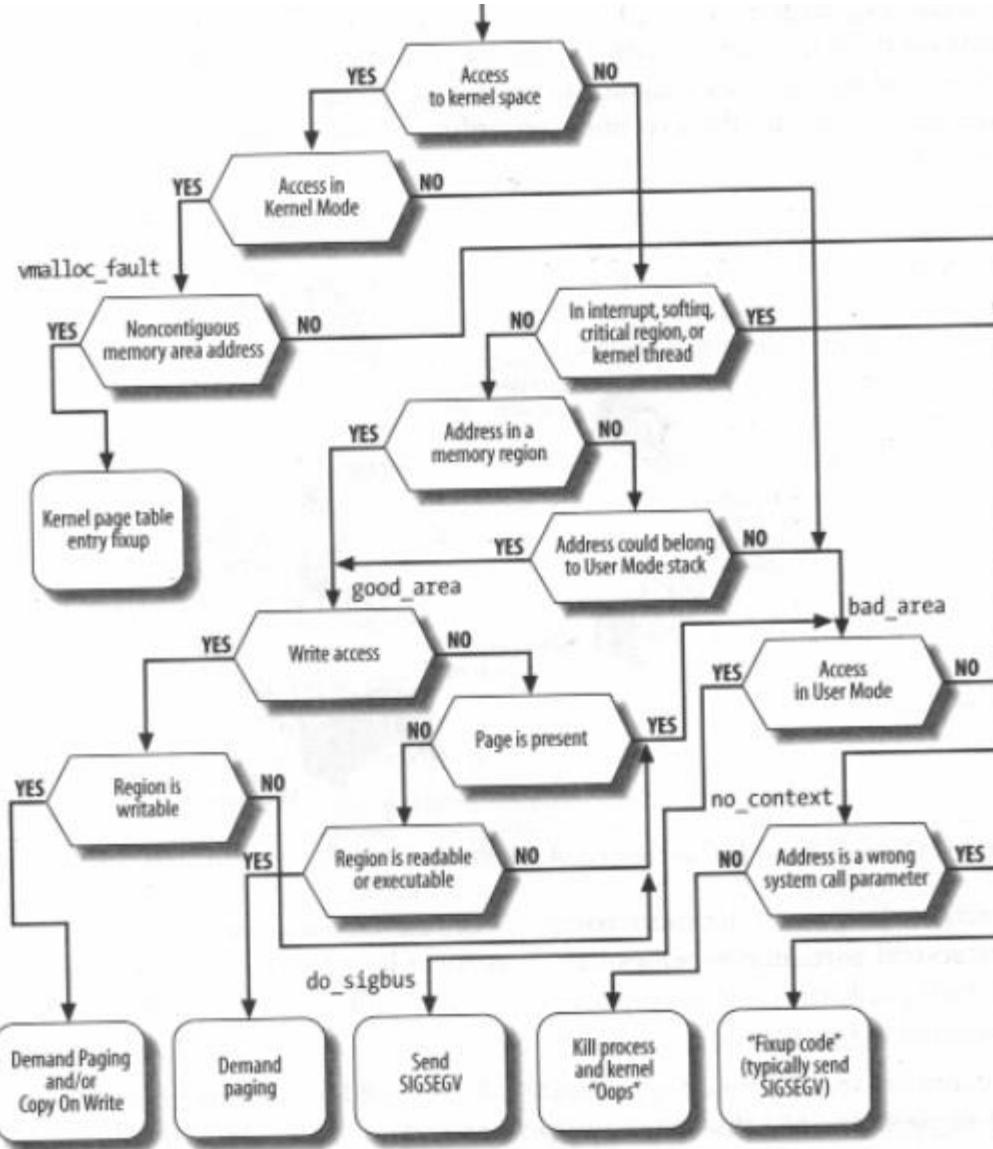


Figure 9-5. The flow diagram of the Page Fault handler

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]
```

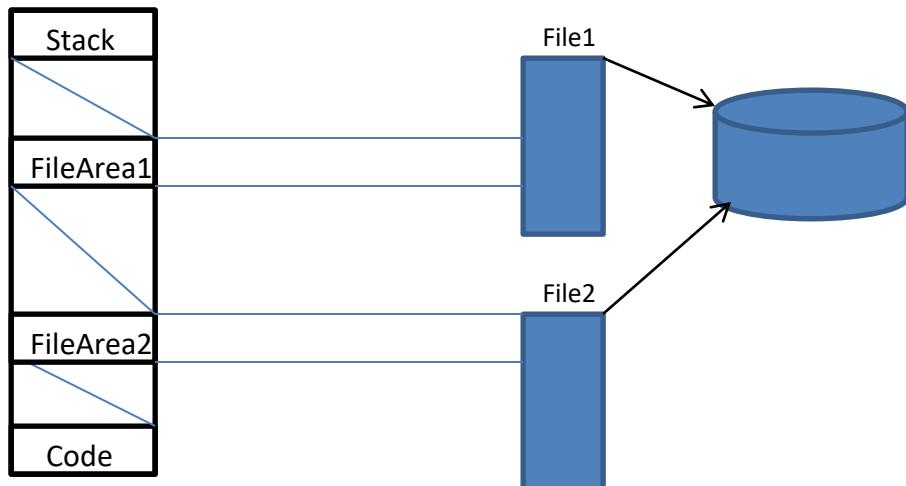
Anonymous memory:
- Swap space required

```
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

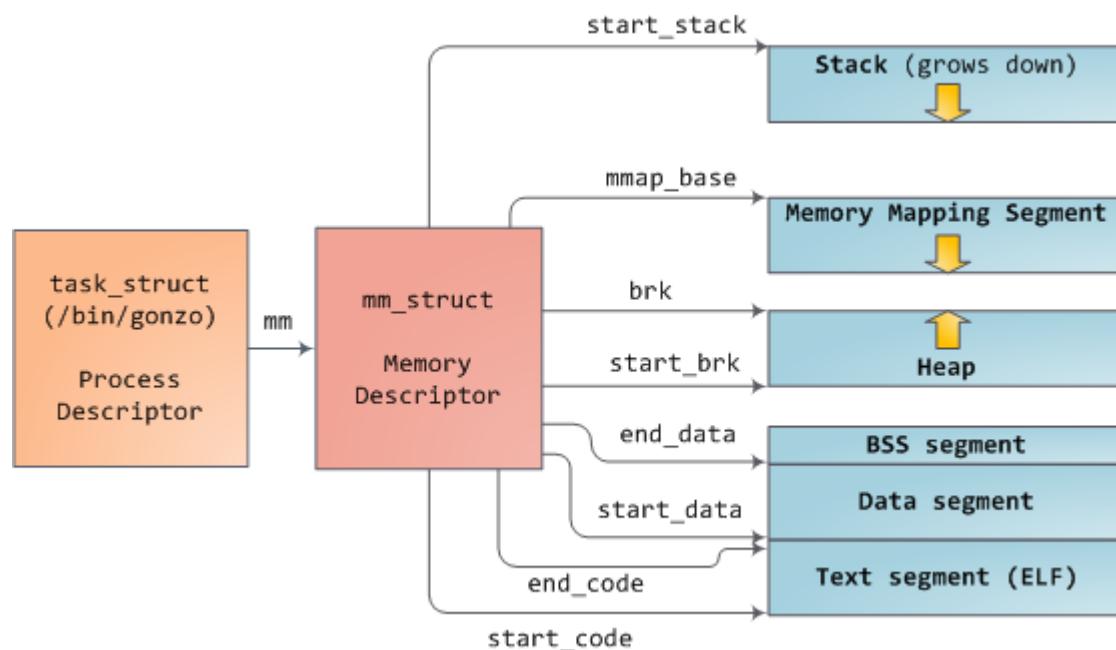


Benefits

- Can perform load/store operations vs input/output

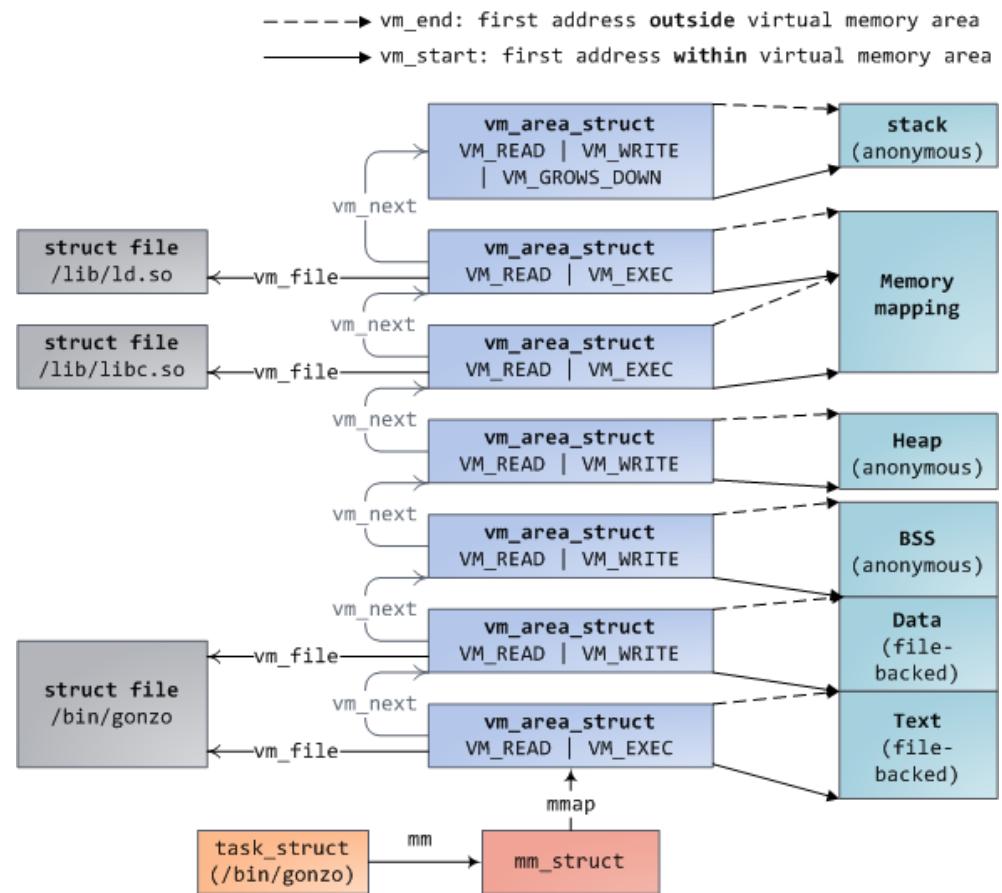
Linux Example #1

- Objects:



Linux Example

- VMA areas
- Anonymous vs. filebacked

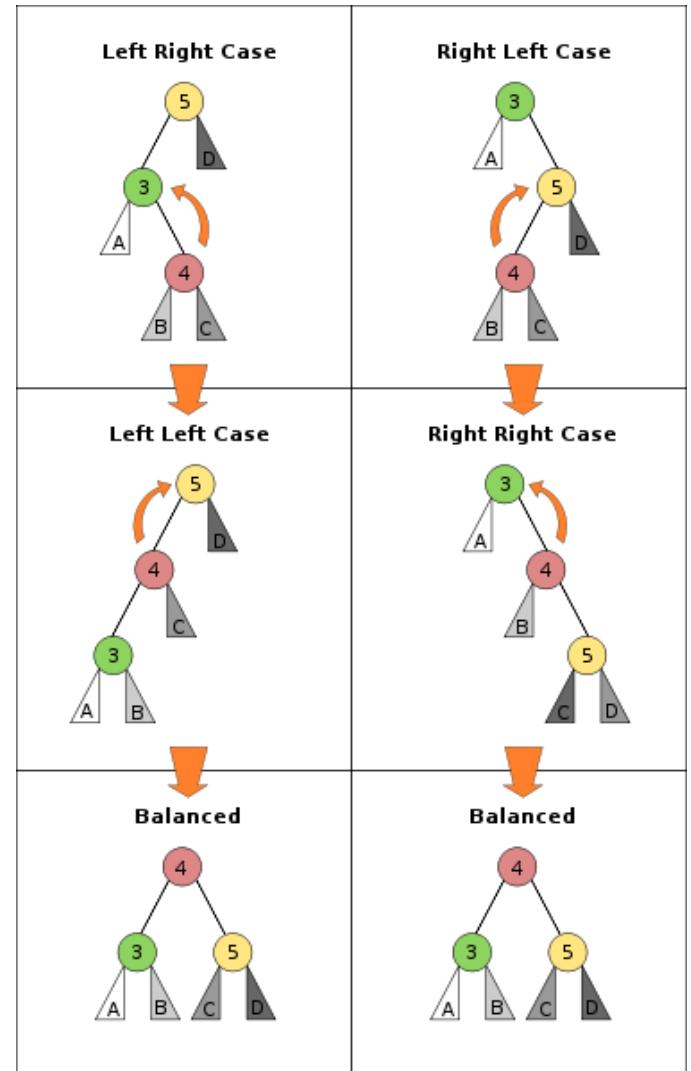


Organization of Memory Regions

- 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

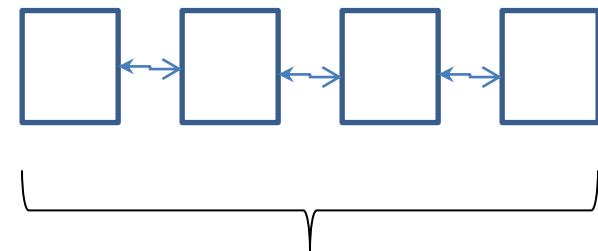
VMA organization

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

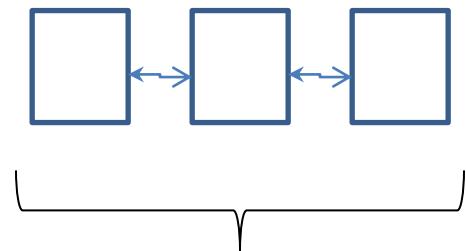


Linux MemMgmt LRU Approximation

- 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

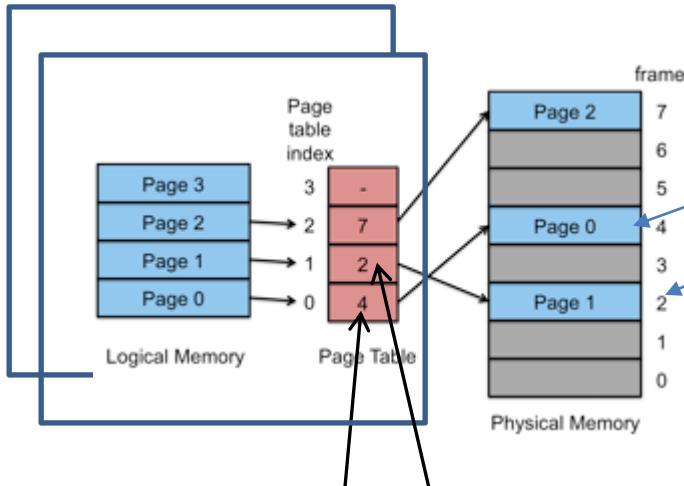
Lab3 (how to approach)

- 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

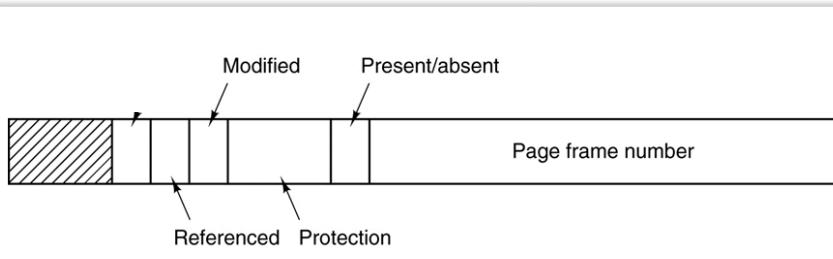
- Create your frame table
- Make all algorithms run through frame table
- Make all algorithms maintains a "hand" from where to start the next "selection"
- On process exit() return all used frames to a free pool and start getting a free frame from there again till free pool is empty; then continue algorithm 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)

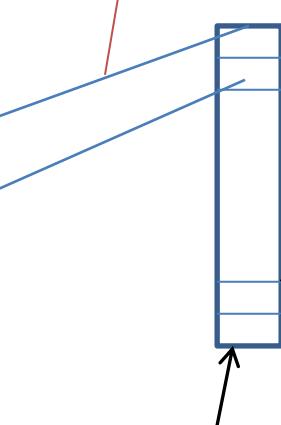


PageTable (one per process)

PageTable Entry (one per virtual page)



reverse mappings



Inverse mapping
Reference count
Locked
Linkage

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