Midterm Review

Ben Lenard

blenard@iit.edu

ILLINOIS TECH | College of Computing

Agenda

- Midterm Review
  - These topics will be on the test but not the exact questions
- **Do not give short answers when I ask you to explain something; more then 1 sentence.**

Midterm

The Midterm will be online from 6:25pm to 7:45pm October 16th 2023 online - I gave you 5 extra minutes for opening it etc.

Join Zoom Meeting for questions

https://iit-edu.zoom.us/j/82470358553?pwd=eEl1SHcrWHArQnkzZkw4VlBDMnVkZz09

Virtual memory

- Virtual memory – separation of user logical memory from physical memory
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Allows address spaces to be shared by several processes
  - Allows for more efficient process creation
  - More programs running concurrently
  - Less I/O needed to load or swap processes

Virtual memory (Cont.)

- Virtual address space – logical view of how process is stored in memory
  - Usually start at address 0, contiguous addresses until end of space
  - Meanwhile, physical memory organized in page frames
  - MMU must map logical to physical
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

Virtual Memory Intuition

Idea: OS keeps unreferenced pages on disk

- Slower, cheaper backing store than memory

Process can run when not all pages are loaded into main memory

OS and hardware cooperate to provide illusion of large disk as fast as main memory

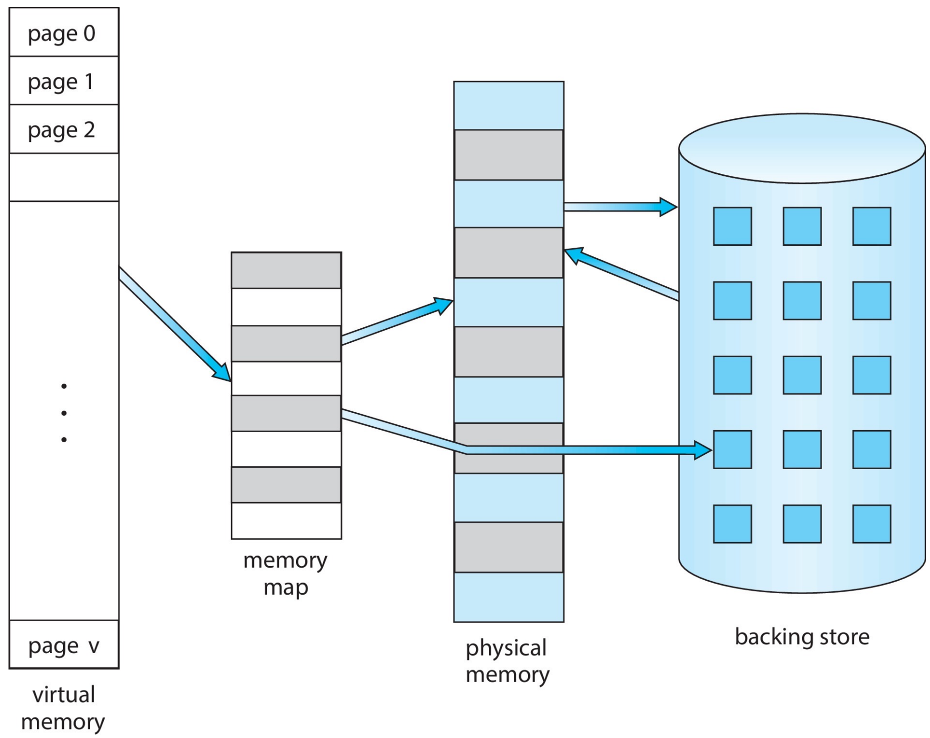
- Same behavior as if all of address space in main memory
- Hopefully have similar performance

Requirements:

- OS must have **mechanism** to identify location of each page in address space → in memory or on disk
- OS must have **policy** for determining which pages live in memory and which on disk

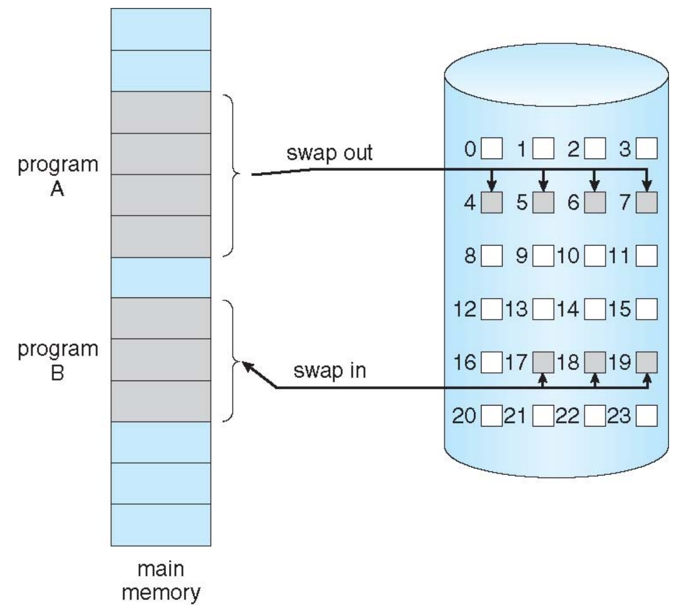
ILLINOIS TECH | College of Computing

Virtual Memory That is Larger Than Physical Memory



Demand Paging

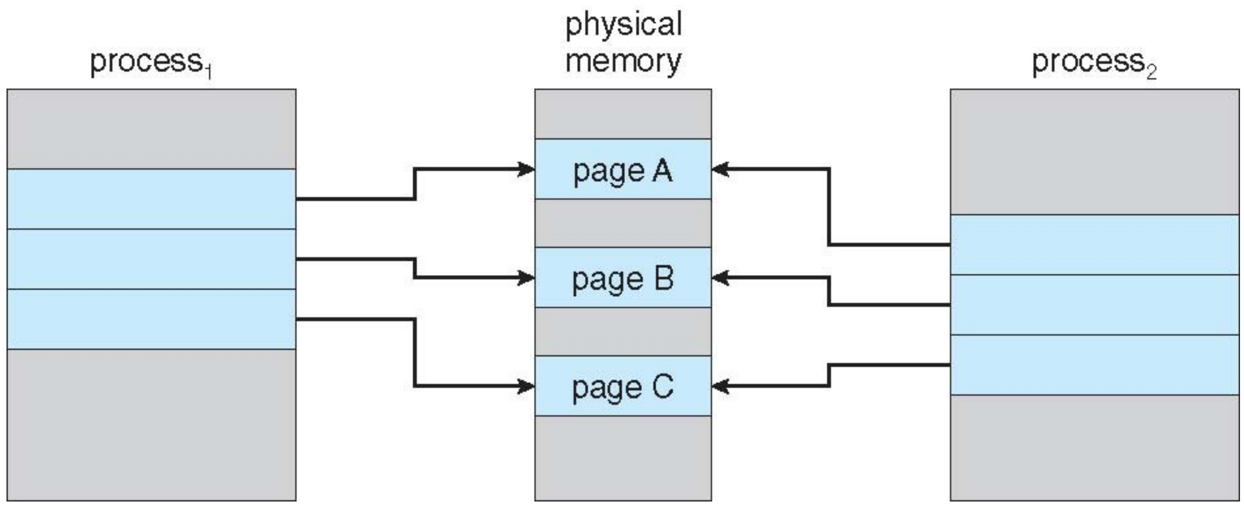
- Could bring entire process into memory at load time
- Or bring a page into memory only when it is needed
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users
- Similar to paging system with swapping (diagram on right)



Copy-on-write (COW)

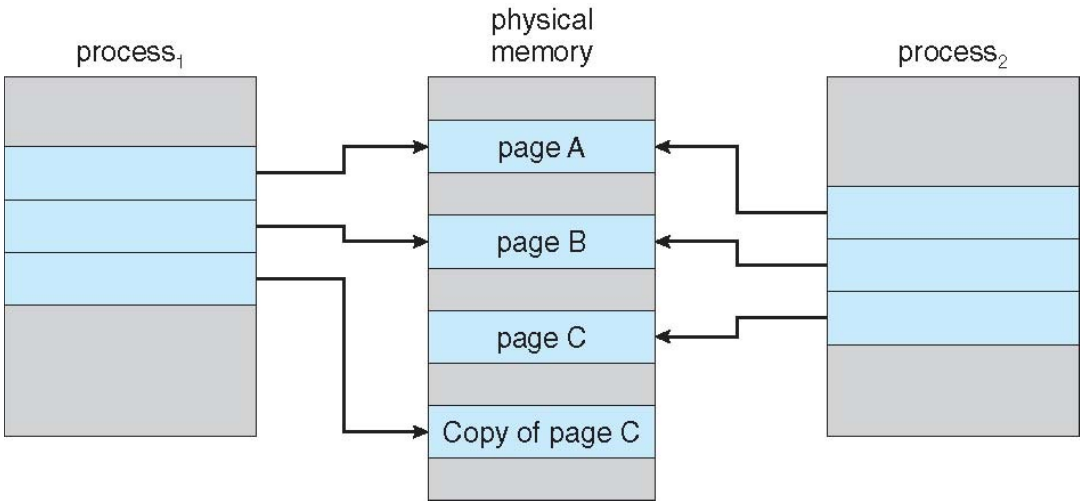
- “Clone” operation is quite common (e.g., used when fork-ing a process)
- But if carried out literally — duplicating entire memory image — is incredibly expensive (and likely unnecessary)
- At clone time, no data is actually copied; simply replicate paging structures and mark pages as read-only
  - Page faults that occur on write accesses trigger copy operation

Before Process 1 Modifies Page C

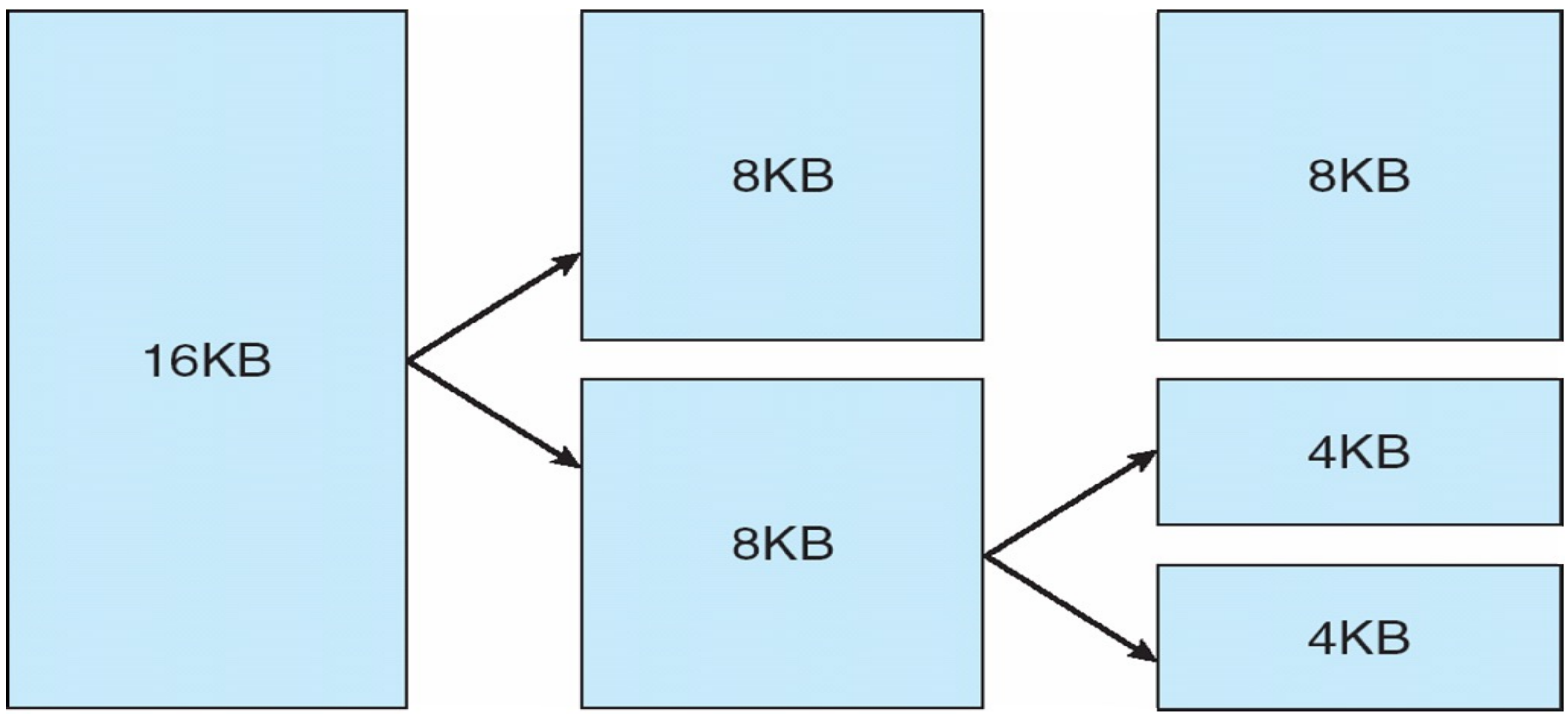


ILLINOIS TECH | College of Computing

After Process 1 Modifies Page C



Splitting of Memory in a Buddy Heap



Buddy system pros & cons

Pros

- Fast allocation — search is easy
- Able to find contiguous blocks
- Good for huge pages
- Can simplify page table updates

Cons

- Small vs. Large blocks creates external fragmentation
- 2^n block sizes can result in significant internal fragmentation
- Compromise: speed vs. efficiency

Where Are Pagetables Stored?

How big is a typical page table?
- assume **32-bit** address space
- assume 4 KB pages
- assume 4 byte entries

Final answer: 2 ^ (32 - log(4KB)) * 4 = **4 MB**

- Page table size = Num entries * size of each entry
- Num entries = num virtual pages = 2^(bits for vpn)
- Bits for vpn = 32– number of bits for page offset
  = 32 – lg(4KB) = 32 – 12 = 20
- Num entries = 2^20 = 1 MB
- Page table size = Num entries * 4 bytes = 4 MB

Implication: Store each page table in memory
- Hardware finds page table base with register (e.g., CR3 on x86)

What happens on a context-switch?
- Change contents of page table base register to newly scheduled process
- Save old page table base register in PCB of descheduled process

Other PT info

What other info is in pagetable entries besides translation?

- valid bit
- protection bits
- present bit (needed later)
- reference bit (needed later)
- dirty bit (needed later)

Pagetable entries are just bits stored in memory

- Agreement between hw and OS about interpretation

Page table walk

- The page table is too large to fit into the MMU, so resides in memory
- Translating a VPN → PPN requires indexing into the page table (known as a page table walk)
  - Performed by MMU
  - Page table is managed by the kernel for each process
    - Current process page table is selected by kernel on each context switch (e.g., by pointing a page table base register at it)

Swapping Motivation

OS goal: Support processes when not enough physical memory

- Single process with very large address space
- Multiple processes with combined address spaces

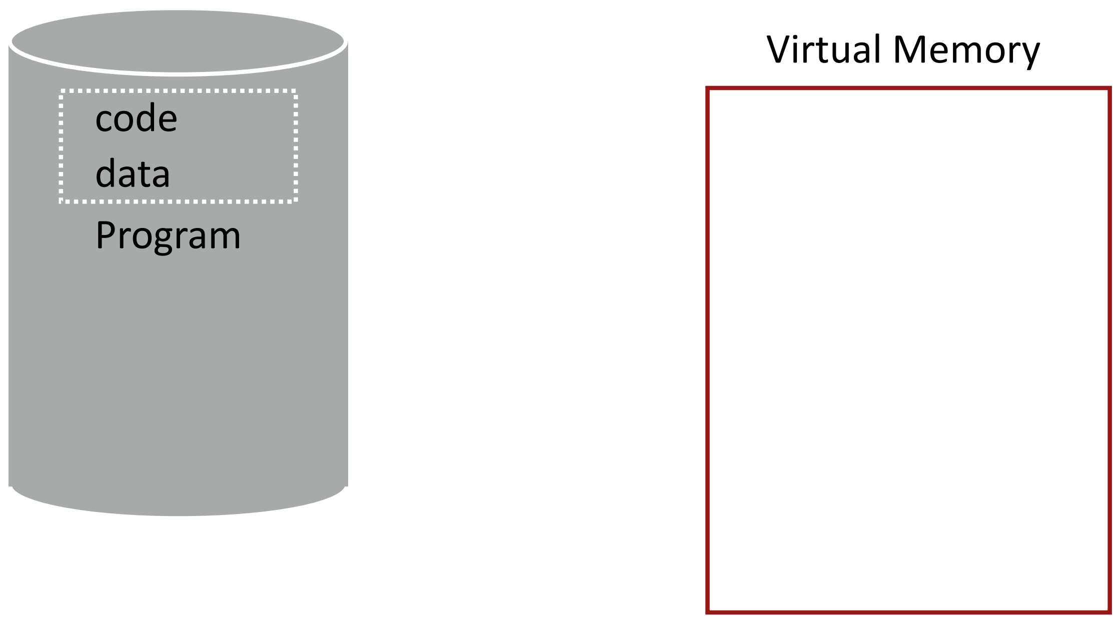
User code should be independent of amount of physical memory

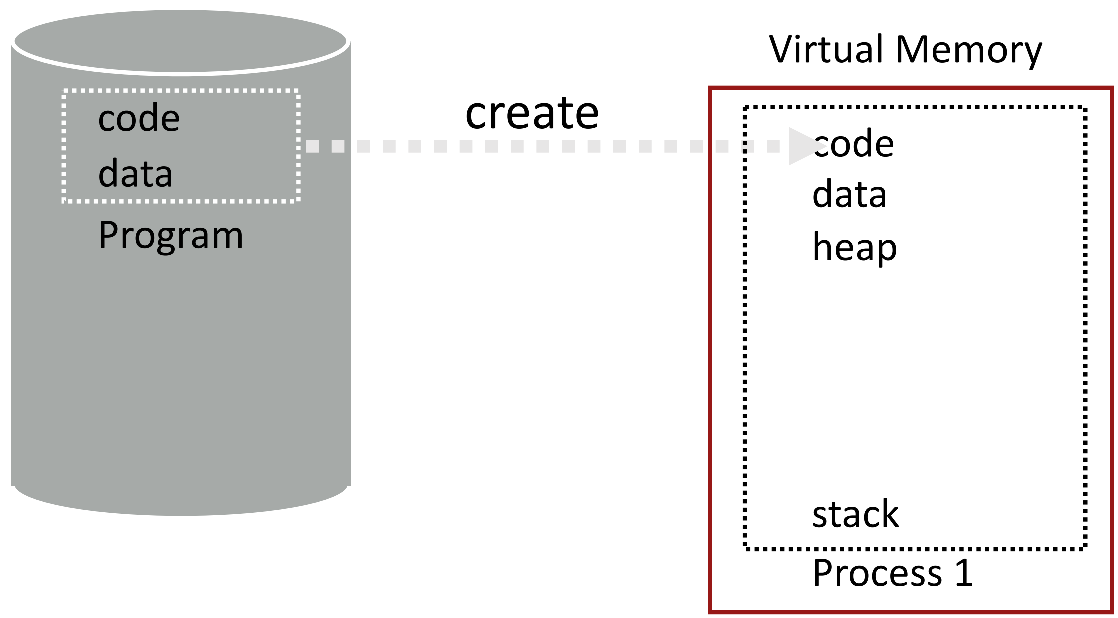
- Correctness, if not performance

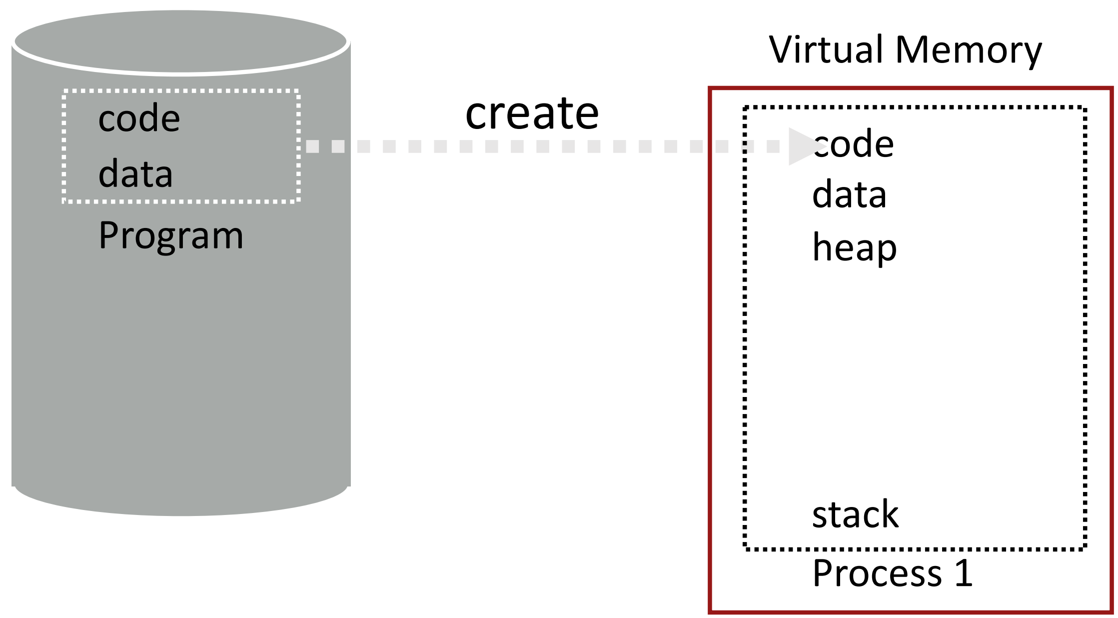
Virtual memory: OS provides illusion of more physical memory

Why does this work?

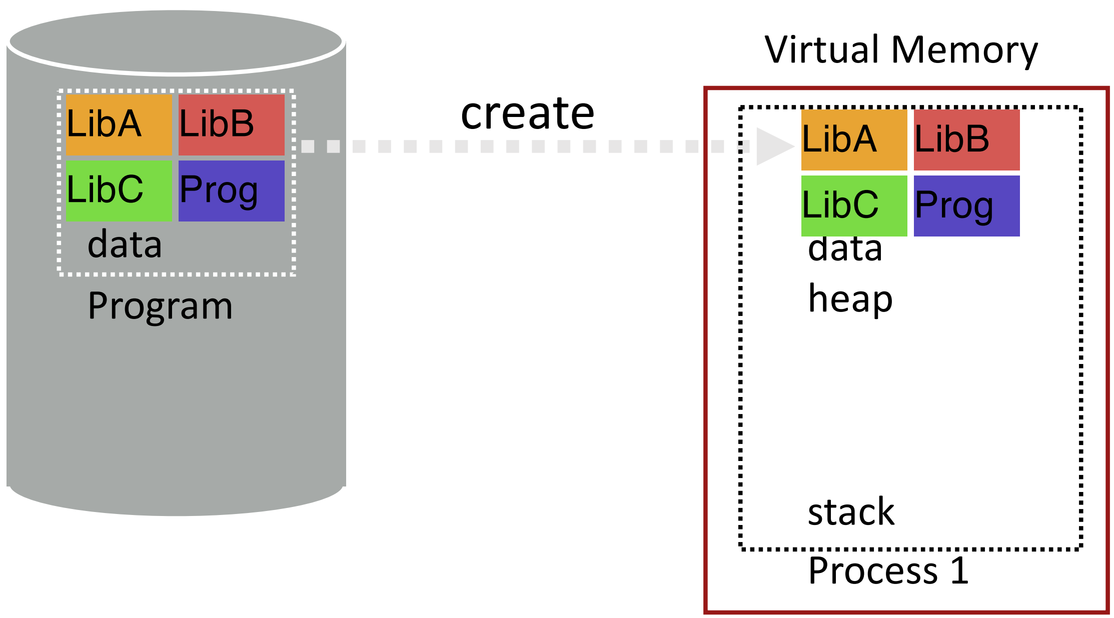
- Relies on key properties of user processes (workload) and machine architecture (hardware)


code
data
Program
Virtual Memory


code
data
Program
create
Virtual Memory
code
data
heap
stack
Process 1

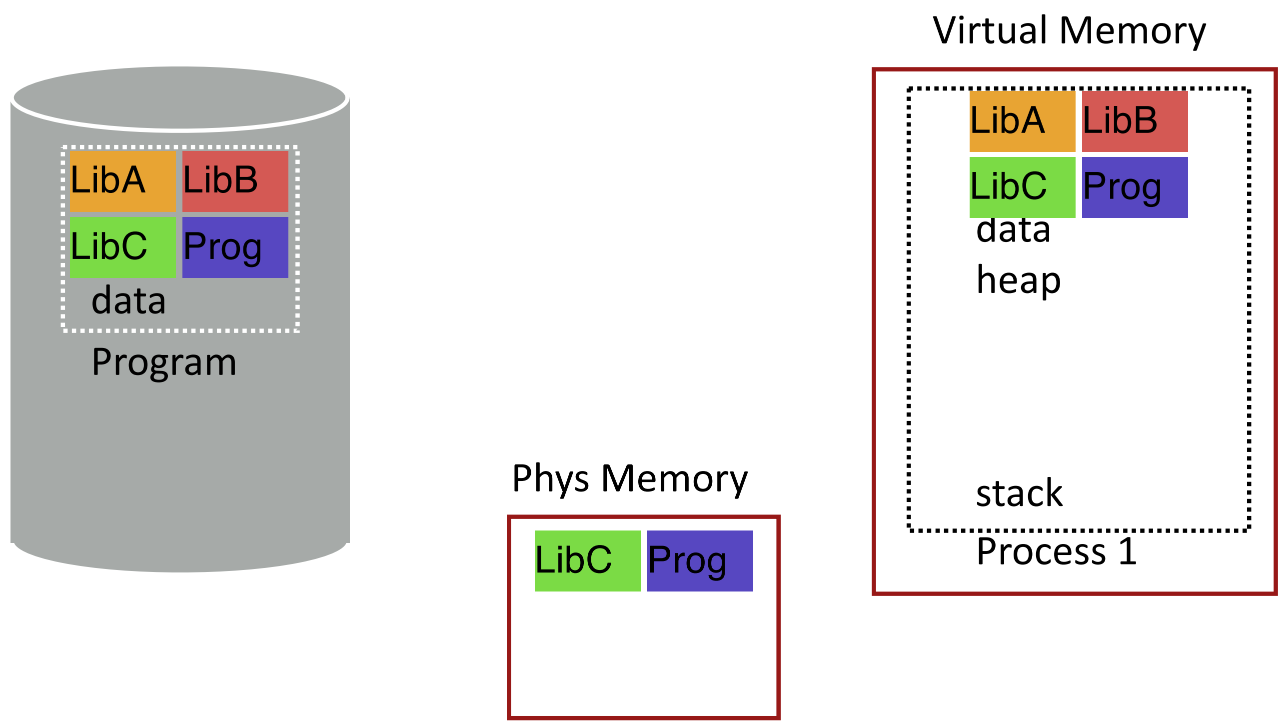


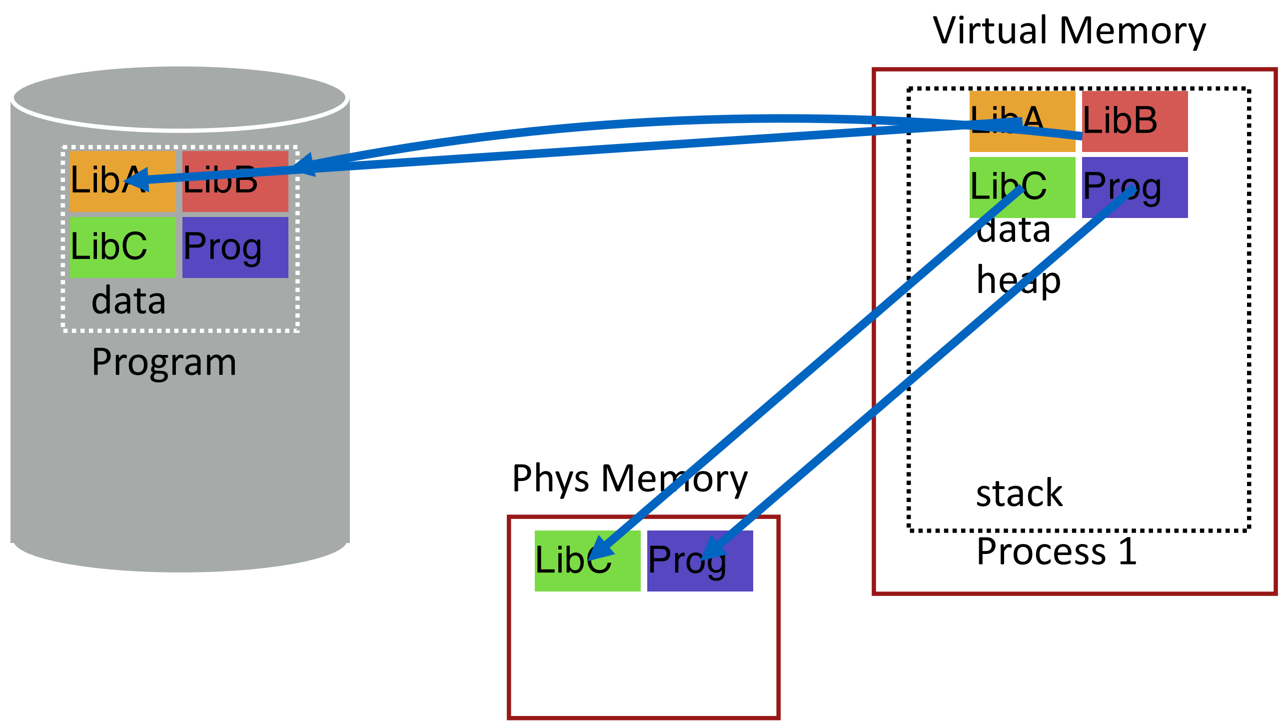
what’s in code?

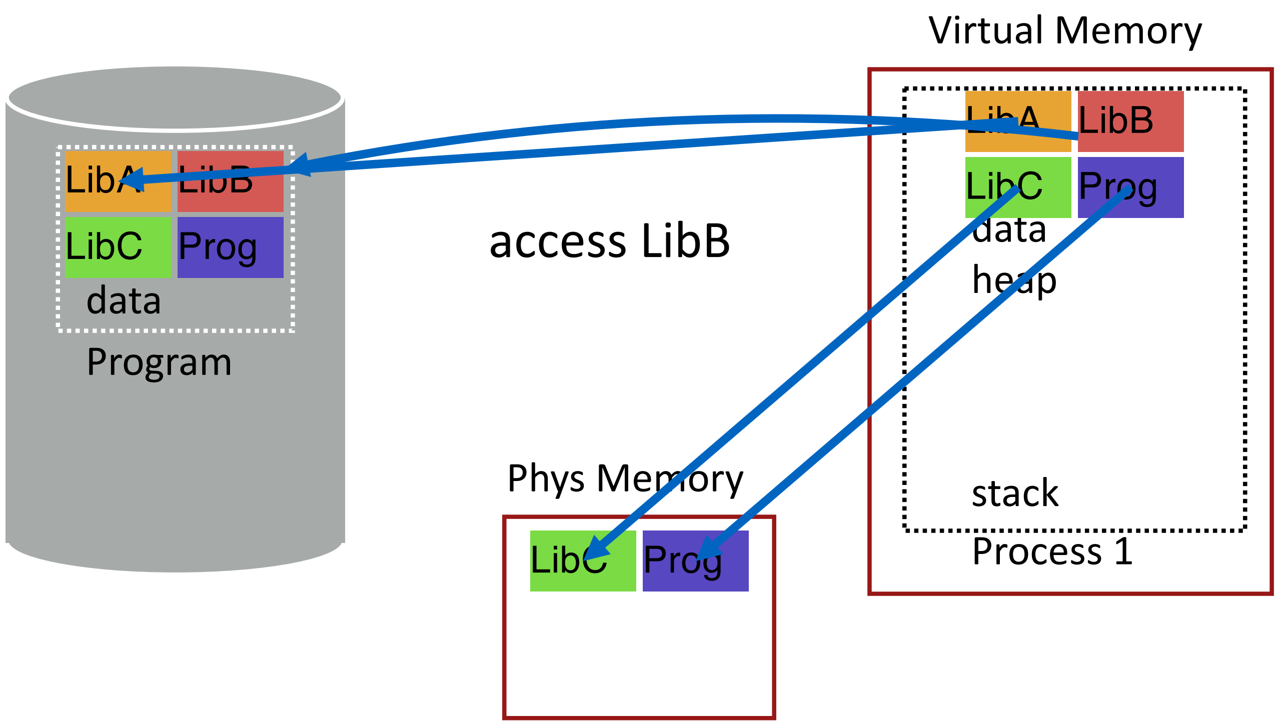


many large libraries, some
of which are rarely/never used

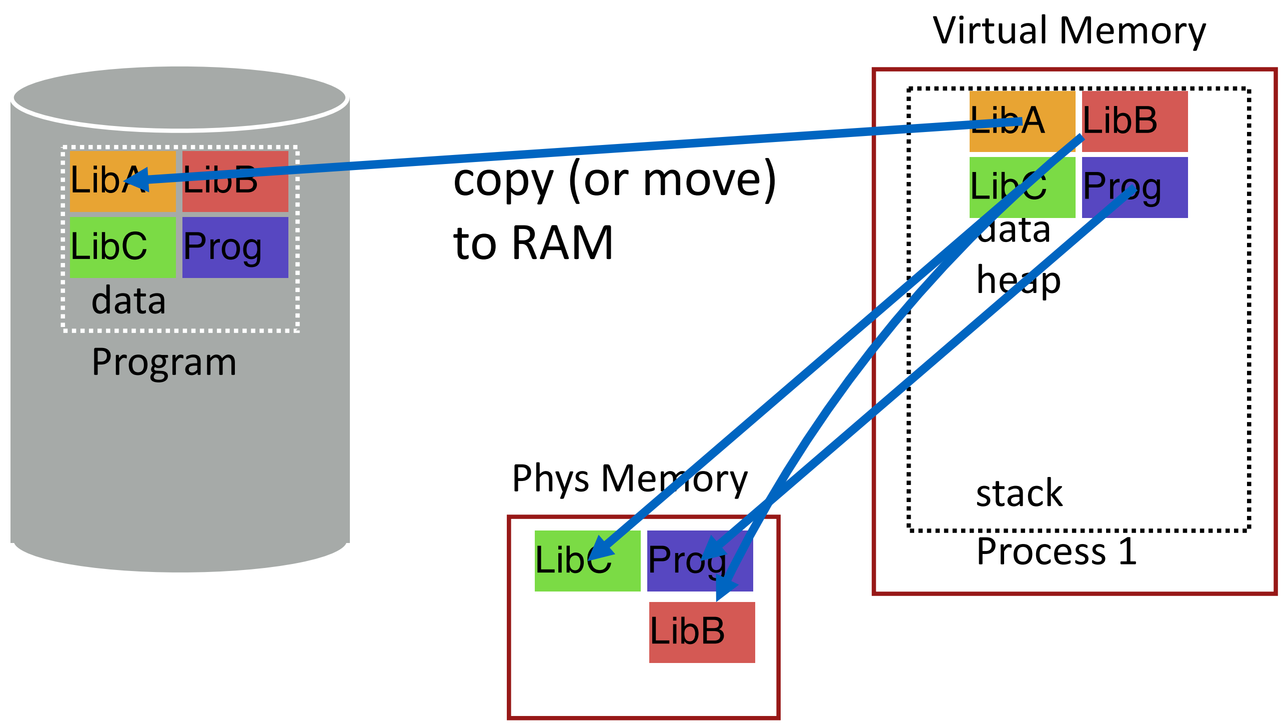
How to avoid wasting physical pages to back rarely used virtual pages?


LibA
LibB
LibC
Prog
data
Program
Phys Memory
LibC
Prog
Virtual Memory
LibA
LibB
LibC
Prog
data
heap
stack
Process 1

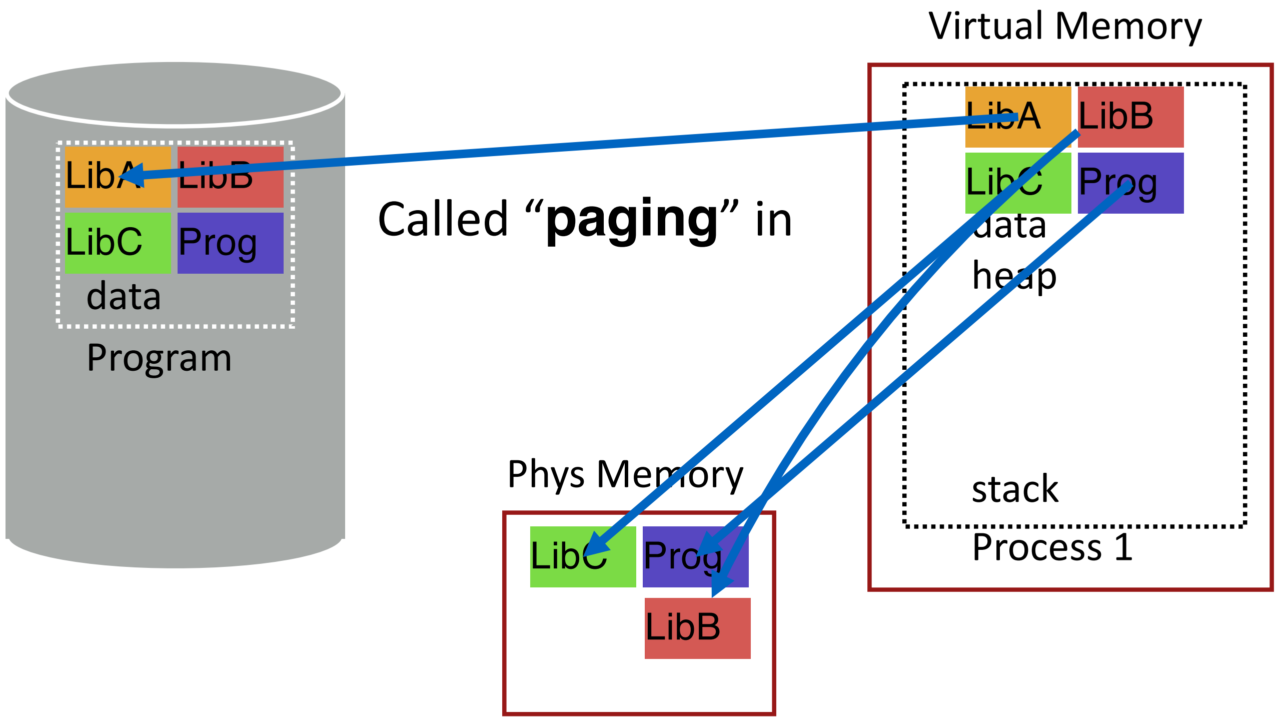

Virtual Memory
LibA
LibB
LibC
Prog
data
Program
LibA
LibB
LibC
Prog
data
heap
stack
Process 1
Phys Memory
LibC
Prog



Virtual Memory
LibA
LibB
LibC
Prog
data
heap
stack
Process 1
LibA
LibB
LibC
Prog
data
Program
access LibB
Phys Memory
LibC
Prog


LibA
LibB
LibC
Prog
data
Program
copy (or move)
to RAM
Virtual Memory
LibA
LibB
LibC
Prog
data
heap
stack
Process 1
Phys Memory
LibC
Prog
LibB

Called “**paging**” in



Locality of Reference

Leverage locality of reference within processes

- Spatial: reference memory addresses **near** previously referenced addresses
- Temporal: reference memory addresses that have referenced in the past
- Processes spend majority of time in small portion of code
  - Estimate: 90% of time in 10% of code

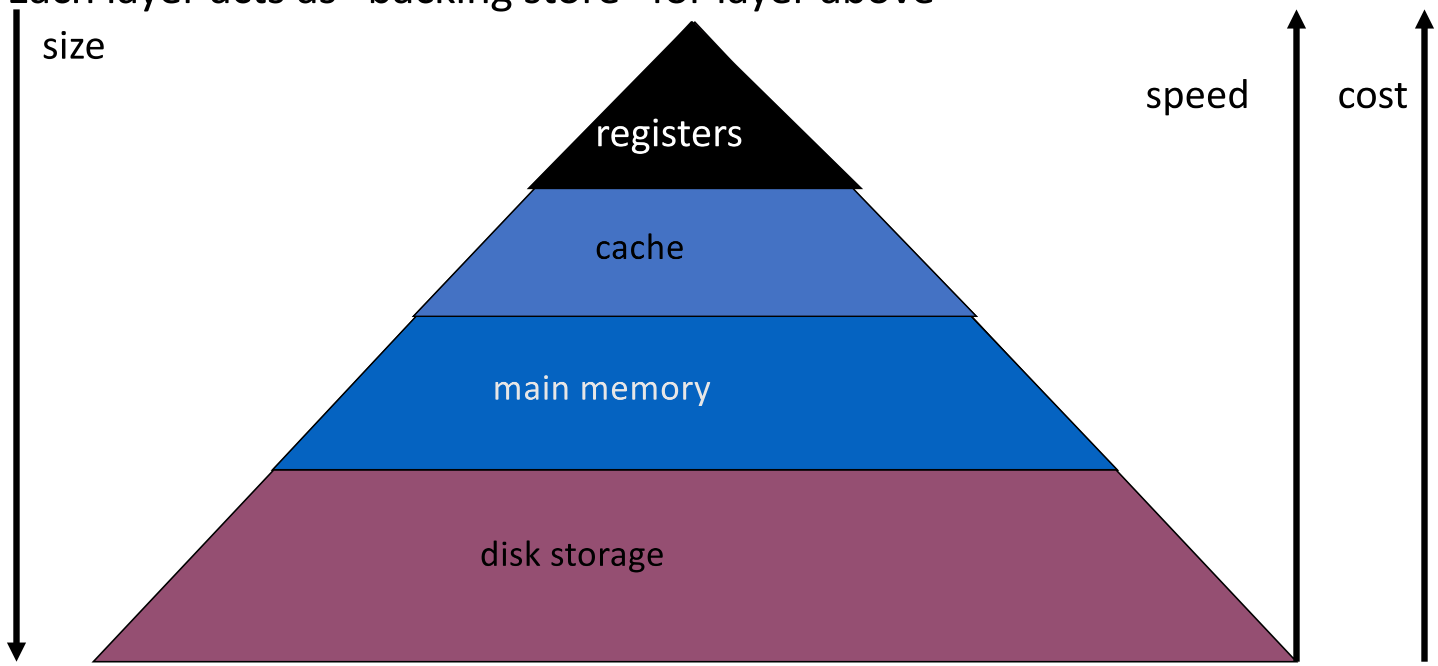
Implication:

- Process only uses small amount of address space at any moment
- Only small amount of address space must be resident in physical memory

Memory Hierarchy

Leverage memory hierarchy of machine architecture

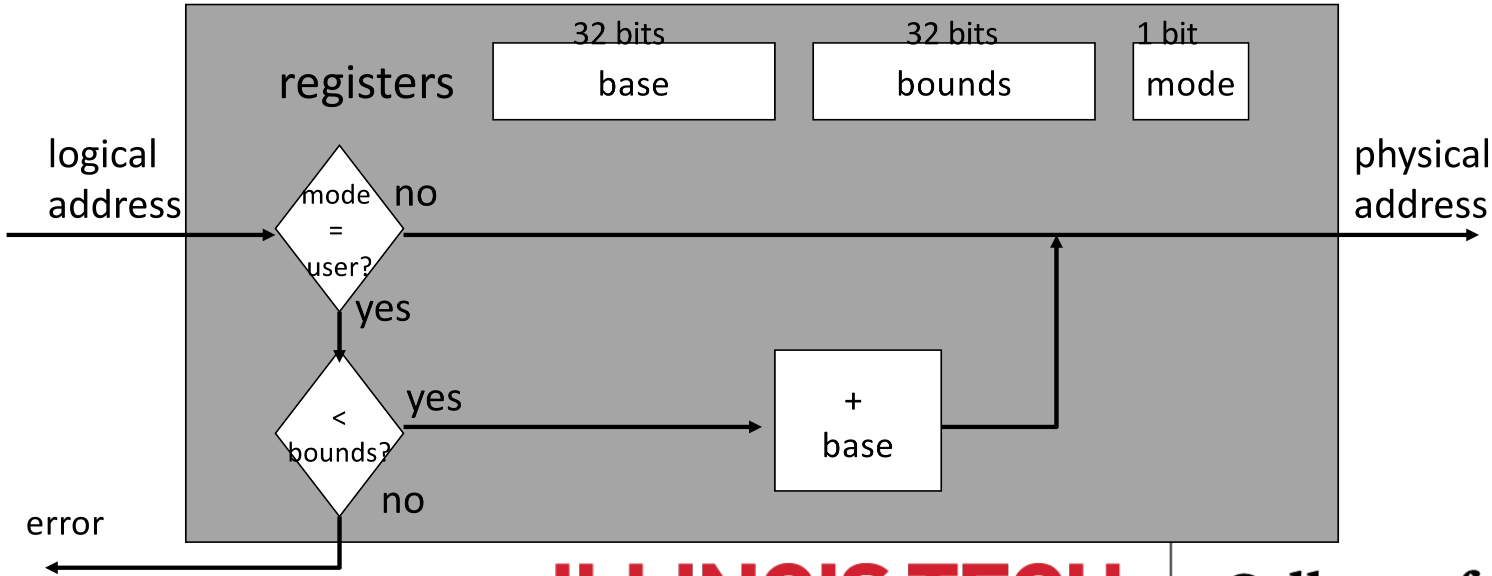
Each layer acts as "backing store" for layer above



Implementation of BASE+BOUNDS

Translation on every memory access of user process

- MMU compares logical address to bounds register
  - if logical address is greater, then generate error
- MMU adds base register to logical address to form physical address



Base address

- Kernel maintains base address of each process in PCB
  - Load into base (address) register in MMU on each context switch
  - Relocation = register access + addition
- Problem: protection not guaranteed!

Base + Limit registers

- Incorporate a limit register to enforce memory protection
- Assertion failure triggers fault (software exception) and loads kernel

Managing Processes with Base and Bounds

Context-switch

- Add base and bounds registers to PCB
- Steps
  - Change to privileged mode
  - Save base and bounds registers of old process
  - Load base and bounds registers of new process
  - Change to user mode and jump to new process

What if don’t change base and bounds registers when switch?

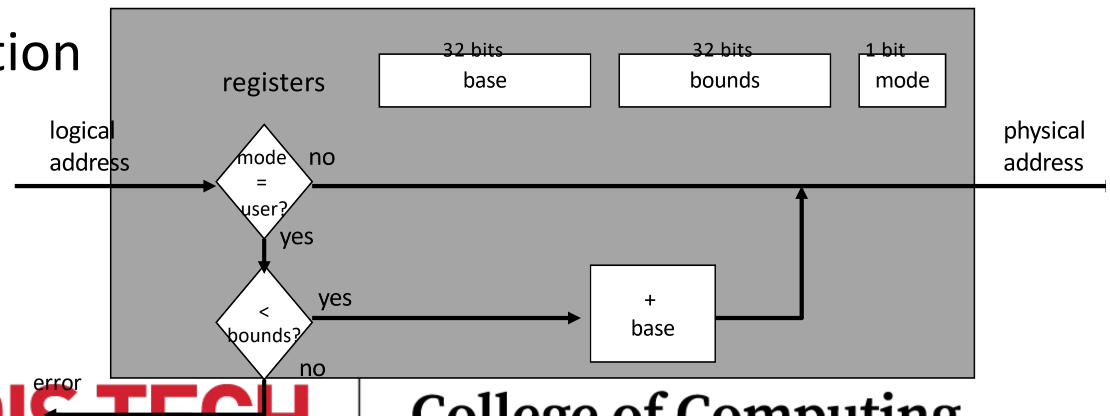
Protection requirement

- User process cannot change base and bounds registers
- User process cannot change to privileged mode

Base and Bounds Advantages

Advantages

- Provides protection (both read and write) across address spaces
- Supports dynamic relocation
  - Can place process at different locations initially and also move address spaces
- Simple, inexpensive implementation
  - Few registers, little logic in MMU
- Fast
  - Add and compare in parallel

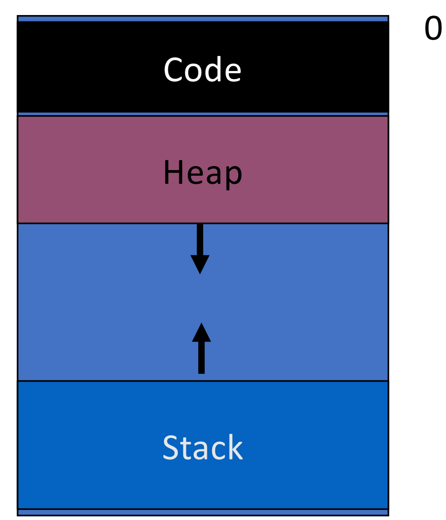


Base and Bounds DISADVANTAGES

Disadvantages

- Each process must be allocated contiguously in physical memory
  - Must allocate memory that may not be used by process

- No partial sharing: Cannot share limited parts of address space



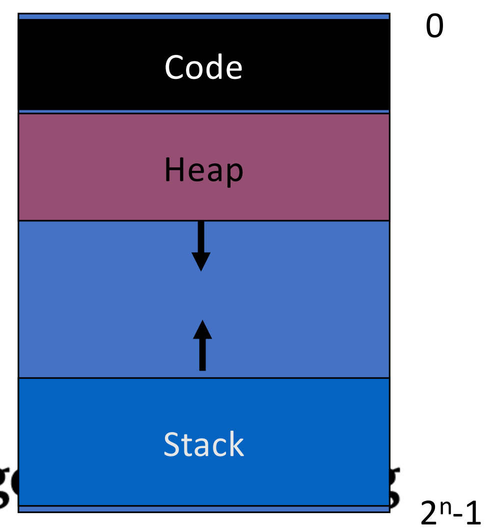
Segmentation

Divide address space into logical segments

- Each segment corresponds to logical entity in address space
  - code, stack, heap

Each segment can independently:

- be placed separately in physical memory
- grow and shrink
- be protected (separate read/write/execute protection bits)



Segmentation (Cont.)

- Partition virtual address space into multiple disjoint segments
  - Individually map onto physical memory with separate base/limit registers
  - Address space info stored in PCB and restored on context switch - Requires that memory requests are for segmented addresses
- Consist of segment selector and offset into segment
- Alternatively: segment can be implied by instruction (e.g., PC always refers to code segment)

Segmented Addressing

Process now specifies segment and offset within segment

How does process designate a particular segment?

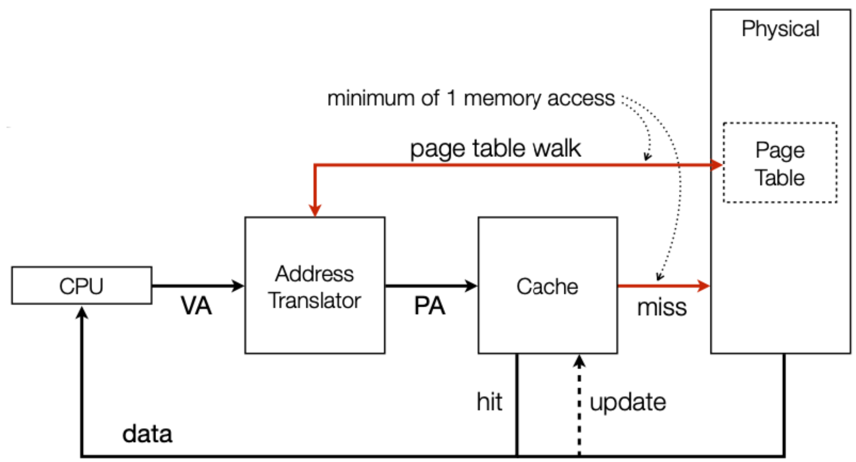
- Use part of logical address
  - Top bits of logical address select segment
  - Low bits of logical address select offset within segment

What if small address space, not enough bits?

- Implicitly by type of memory reference
- Special registers

Page table translations are slow!

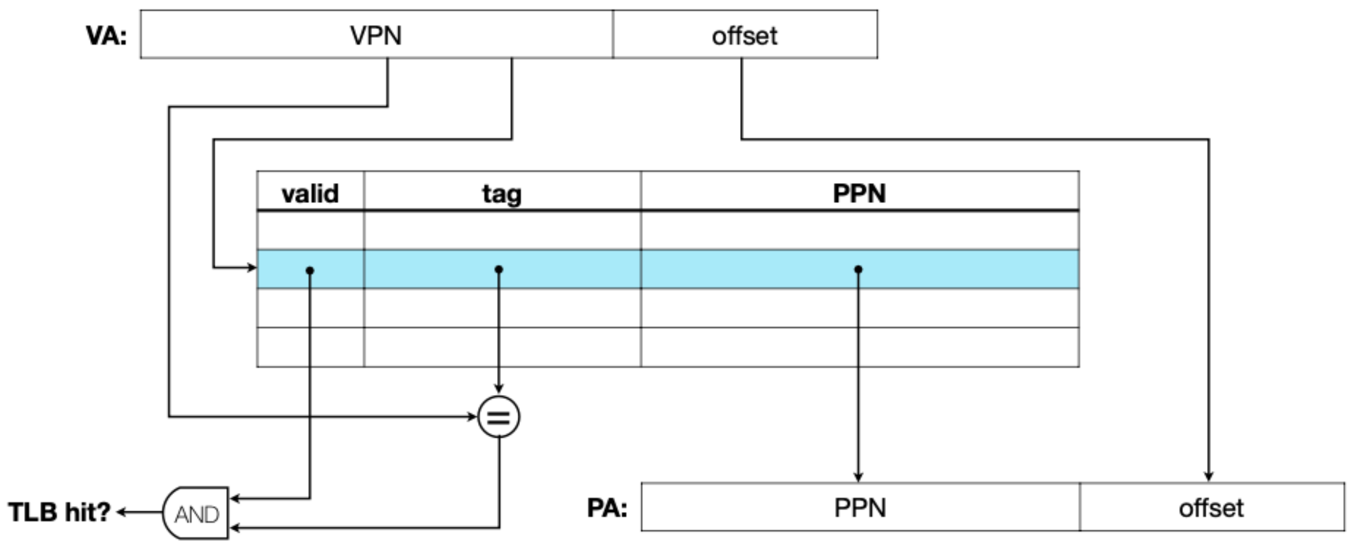
- Most modern caching systems are physically addressed, so we cannot avoid translation before cache lookup
- I.e., each VA access requires up to two memory accesses!



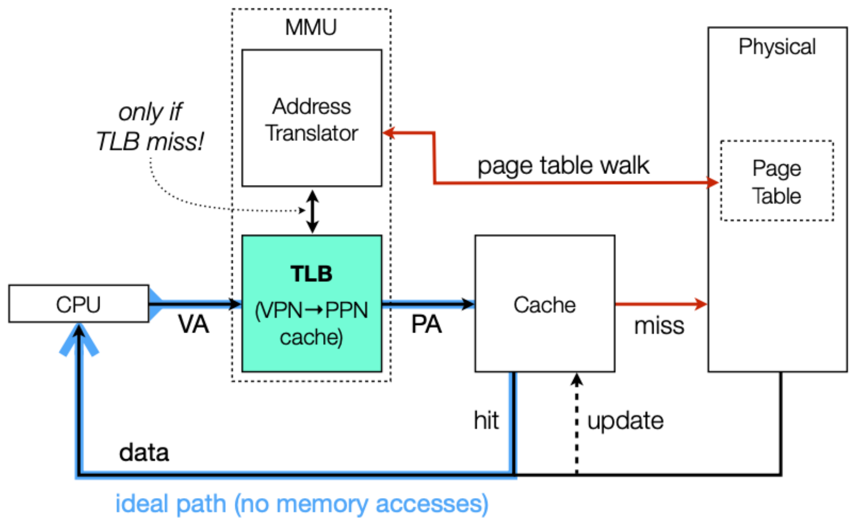
ILLINOIS TECH | College of Computing

Translation Lookaside Buffer (TLB)

- Solution: dedicated cache for VPN → PPN translations
  - Page table walk only performed on TLB miss



TLB / Cache / PT interaction



TLB issues

- TLB mappings are process specific — requires flush on context switch
  - Some architectures store address space identifier per cache line
- TLB only caches a few thousand mappings, at most
  - vs. orders of magnitude more per process, potentially!
  - Effectiveness of TLB can be "tuned" by adjusting number of pages (larger page size = smaller number of pages)
  - Downside to large pages?

ILLINOIS TECH | College of Computing

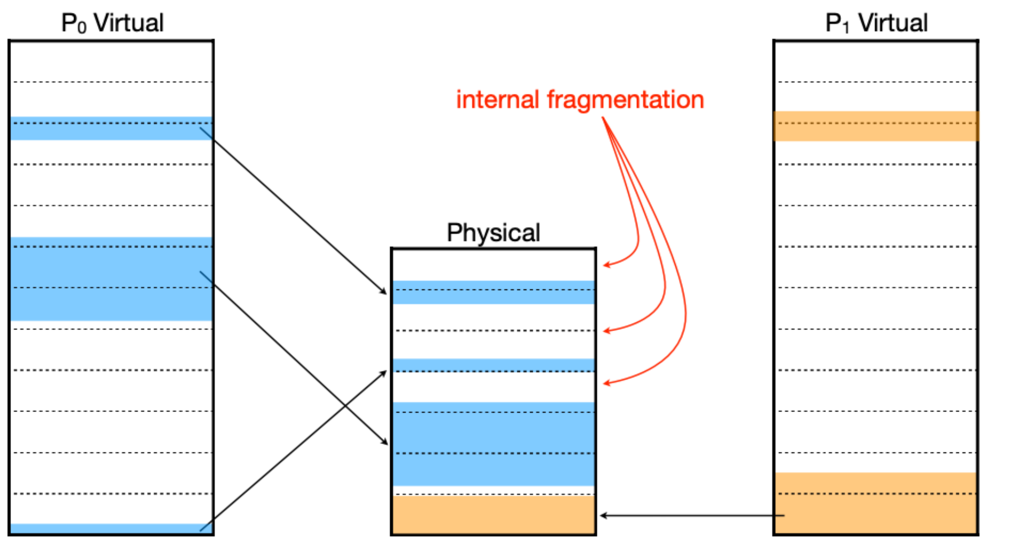
Internal fragmentation

- Large pages result in coarser mapping granularity
  - I.e., larger “chunks” carved out of physical memory at a time
  - May lower utilization, if large portions of pages are not used — known as internal fragmentation
- Must balance TLB effectiveness against memory utilization
- Depends on the application and workload

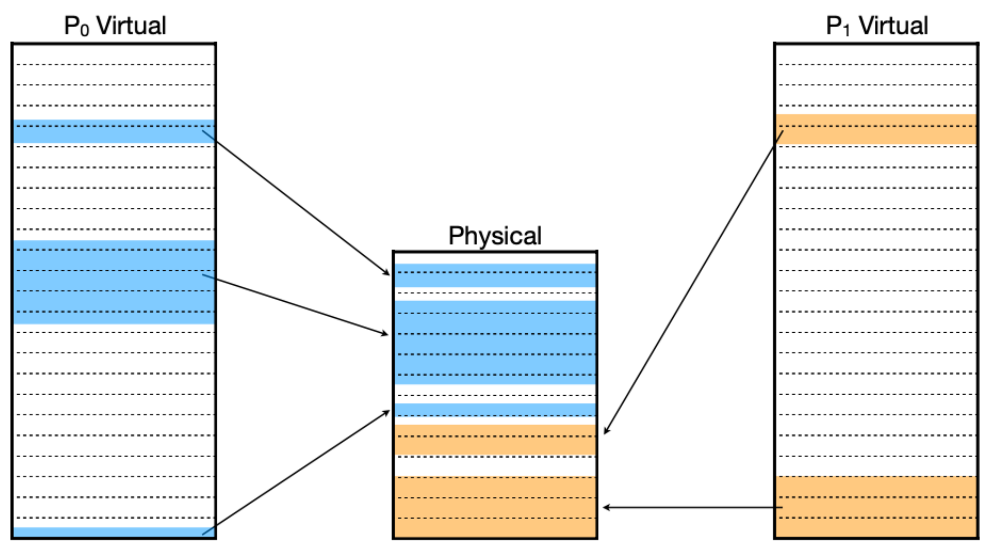
Internal fragmentation

- Large pages result in coarser mapping granularity
  - I.e., larger “chunks” carved out of physical memory at a time
  - May lower utilization, if large portions of pages are not used — known as internal fragmentation
- Must balance TLB effectiveness against memory utilization
- Depends on the application and workload

E.g., large(-ish) pages



E.g., small(-ish) pages



Address space = a lie!

- If processes are simultaneously accessing physical memory ... - Not all text sections can begin at 0x400000
  - Not all data sections can begin at 0x600000
  - Not all heaps can begin at 0x18f0000
  - Not all stacks can begin at 0x7fff00000000
- Uniform process address spaces are an illusion created by the kernel
  - To simplify program loading and execution (among other reasons)

Hardware Support for Dynamic Relocation

Two operating modes

- Privileged (protected, kernel) mode: OS runs
  - When enter OS (trap, system calls, interrupts, exceptions)
  - Allows certain instructions to be executed
    - **Can manipulate contents of MMU**
  - **Allows OS to access all of physical memory**
- User mode: User processes run
  - **Perform translation of logical address to physical address**

Minimal MMU contains **base register** for translation

- base: start location for address space

Scheduling: Proportional Share

- proportional-share scheduler or fair-share scheduler.
- Proportional-share is based around a simple concept: instead of optimizing for turnaround or response time, a scheduler might instead try to guarantee that each job obtain a certain percentage of CPU time.

Lottery Scheduling

- Simple Idea, at a given interval, hold a lottery to determine which process should get to run next; processes that should run more often should be given more chances to win the lottery.
- How can we design a scheduler to share the CPU in a proportional manner?
- What are the key mechanisms for doing so?
- How effective are they?

Tickets Represent Your Share

- Tickets represent your share
- Consider 10 people each one gets 1 ticket, verses 2,3,4,5 for one person
- Let’s say we add timeslices, or cpu time
- Lottery Scheduling is probabilistic not deterministic
- It uses randomness

ILLINOIS TECH | College of Computing

Tickets Represent Your Share (cont.)

- A gets 0 – 74
- B gets 75 - 99

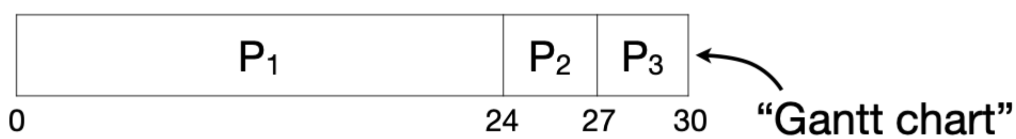
```
63 85 70 39 76 17 29 41 36 39 10 99 68 83 63 62 43  0 49 12
```

Here is the resulting schedule:

```
A     A  A     A  A  A  A  A  A     A     A  A  A  A  A  A
   B        B                    B     B
```

First come first served (FCFS)

Process	***Arrival Time***	***Burst Time***
P_1	0	24
P_2	0	3
P_3	0	3



Wait times: $P_1 = 0$, $P_2 = 24$, $P_3 = 27$

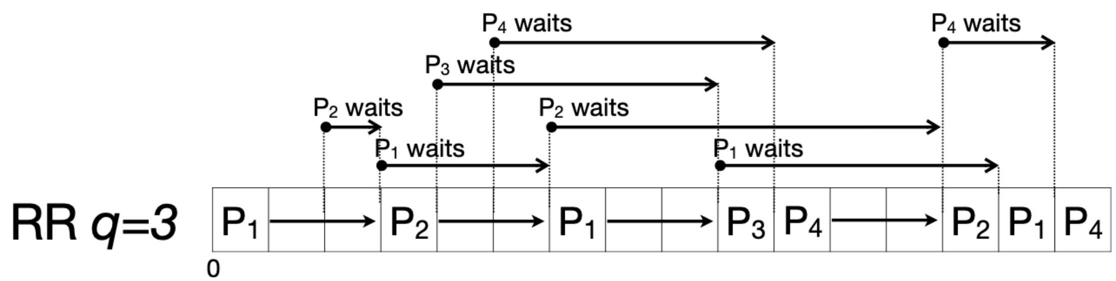
Average: $(0 + 24 + 27) / 3 = 17$

Round Robin (RR)

- The "fairest" of them all
- Uses a FIFO queue:
  - Each job runs for a maximum fixed time quantum q
  - If unfinished, re-enter queue at the tail end
- Given time quantum q and n jobs:
  - max wait time (per cycle) = $q \cdot (n - 1)$
  - each job receives 1/n timeshare
- RR is also used for other applications such as load balancing and dns

RR (cont.)

Process	*Arrival Time*	*Burst Time*
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4



Wait times: $P_1 = 8$, $P_2 = 8$, $P_3 = 5$, $P_4 = 7$

Average: $(8 + 8 + 5 + 7) / 4 = 7$

ILLINOIS TECH | College of Computing

RR (cont.)

Process	*Arrival Time*	*Burst Time*
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

	Avg. Turnaround	**Avg. Wait Time**	
RR *q*=7	8.75	4.75	(FCFS)
RR *q*=4	9	5	
RR *q*=3	11	7	
RR *q*=1	9.75	5.75	

RR (cont.)

Process	***Arrival Time***	***Burst Time***
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

	Throughput	**Utilization**
RR *q*=7	0.25	1.0
RR *q*=4	0.25	1.0
RR *q*=3	0.25	1.0
RR *q*=1	0.25	1.0

Greedy algorithms

- SJF/PSJF are greedy algorithms
  - i.e., they select the best choice at the moment ("local maximum")
- Greedy algorithms don't always produce globally maximal results
  - e.g., naive hill-climbing algorithm (only take a step if it brings me to higher ground) doesn't always find the tallest peak!
- Are SJF/PSJF optimal?

Is It Optimal?

- Consider 4 jobs with burst lengths t0, t1, t2, t3 that just became ready
- What is the average wait time if scheduled in the order given?
  - $= (3 \cdot t0 + 2 \cdot t1 + t2) / 4$
  - Weighted average — clearly minimized by running shortest jobs first!
- SJF/PSJF are provably optimal with respect to average wait time
  - But at what cost?
  - Potential CPU starvation! (e.g., longer jobs keep getting put off)

No way to tell the future

- We've been assuming that job/burst lengths are known in advance
- May be possible in rare circumstances (e.g., repeated jobs, job profiling), but unlikely in practice
- Common approach: predict future burst lengths based on past behavior
  - Simple moving average (sliding window of past values)
  - Exponentially weighted moving average (EMA)

Preemptive SJF (PSJF)

- aka “Shortest Time-to-Completion First” (STCF)
- aka “Shortest Remaining-Time First” (SRTF)
- May preempt running job to schedule a different (ready) job

Some scheduling metrics

- Turnaround time
- Wait time
- Response time
- Throughput
- Utilization

Definition

- Scheduling: policies & mechanisms used to allocate limited resources to some set of entities
- Initial focus: resource & entities = CPU & processes (aka jobs) - other possibilities:
  - resources: memory, I/O bus/devices
  - entities: threads, users, groups
- schedulers for the above may exist in an OS (and must play nice with each other)!

Policy

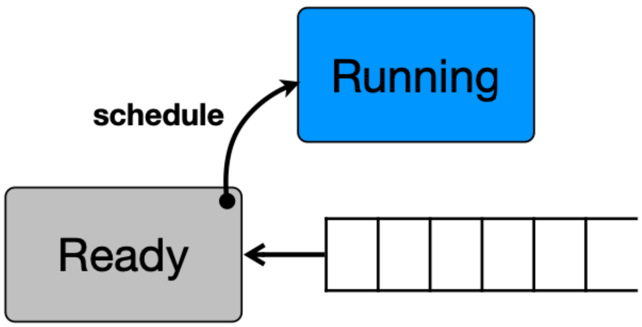
- high-level “what”
- scheduling disciplines
  - e.g., FCFS, SJF, RR, etc.
- driven by a variety of potentially conflicting goals
  - e.g., performance and fairness

Mechanism

- low-level “how”
- combination of HW/SW
  - e.g., clock interrupt, high precision timer, PCB
- scattered throughout kernel codebase

Ready queue isn't always FIFO

- convenient to envision a ready queue (though not necessarily FIFO!)
- the scheduling policy decides which job to select from the set of ready (runnable) jobs to run next



Privilege Escalation

- As it sounds, when a user needs privileges greater then what they have.
- In other words, corner case handling
- Consider sudo, do you need sudo all the time?

Aspects Of The Access Control

- Subjects - A subject is the entity that wants to perform the access, perhaps a user or a process.
- Objects - An object is the thing the subject wants to access, perhaps a file or a device.
- Access - Access is some particular mode of dealing with the object, such as reading it or writing it.

- We sometimes refer to the process of determining if a particular subject is allowed to perform a particular form of access on a particular object as authorization.

Security Goals at a high level

- Confidentiality – Only showing data to the people/apps with the correct permissions – this is different then chmod 777
- Integrity – Ensure something is correct and not altered
- Availability – Ensuring apps/people have access to the data.

- These areas are covered on the CISSP exam

Non-Repudiation

- All about the proof
- Proof that someone received a message or file and cannot say they never received it.
- This will come back with service accounts

Designing Secure Systems

1. **Economy of mechanism** – KISS
2. **Fail-safe defaults** – Default to security, not insecurity. Similar to the credit card networks, default is decline
3. **Complete mediation** – This is a security term meaning that you should check if an action to be performed meets security policies every single time the action is taken. Often ignored.
4. **Open design** – Assume your adversary knows every detail of your design.
5. **Separation of privilege** – Require separate parties or credentials to perform critical actions. Similar to SOX in banking? Why do we have SOX?
6. **Least privilege** – Smaller attack vectors. Similar to why we have userspace.
7. **Least common mechanism** – For different users or processes, use separate data structures or mechanisms to handle them. Aka don't share. We'll talk about containers and switch root.
8. **Acceptability** – If your users won't use it, your system is worthless. Aka the sales pitch

Ulimits – Linux – A user

$ ulimit -a
core file size (blocks, -c) 0
data seg size (kbytes, -d) unlimited
scheduling priority (-e) 0
file size (blocks, -f) unlimited
pending signals (-i) 384666
max locked memory (kbytes, -l) 64
max memory size (kbytes, -m) unlimited
open files (-n) 1024
pipe size (512 bytes, -p) 8
POSIX message queues (bytes, -q) 819200
real-time priority (-r) 0
stack size (kbytes, -s) 8192
cpu time (seconds, -t) unlimited
max user processes (-u) 4096
virtual memory (kbytes, -v) unlimited
file locks (-x) unlimited

Ulimits - root

```
ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 384666
max locked memory       (kbytes, -l) 64
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 384666
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
```

ILLINOIS TECH | College of Computing

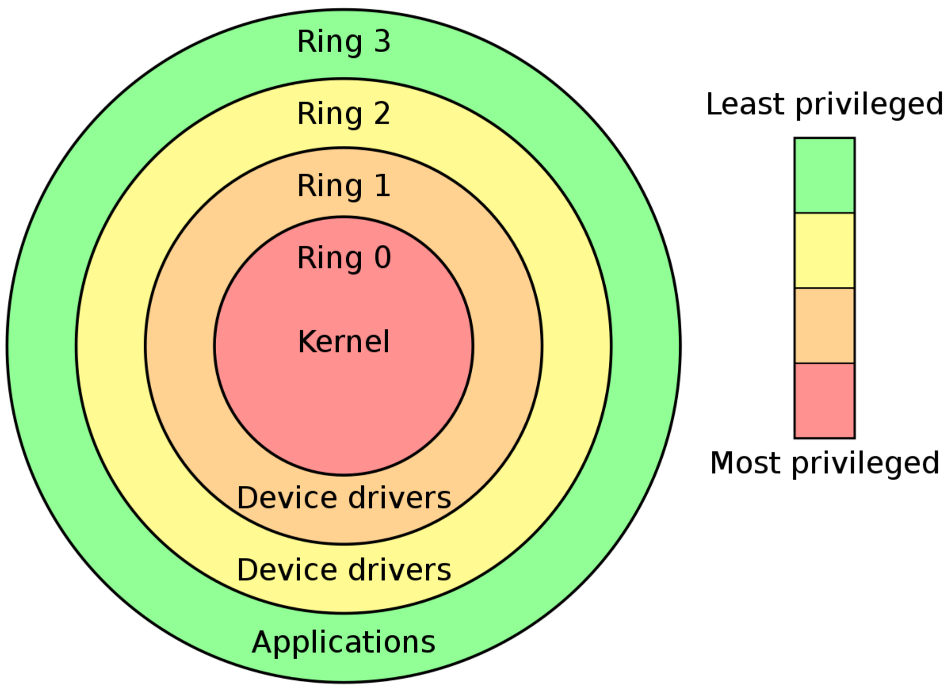
Limited Direct Execution

- Must prevent user from:
  - Accessing arbitrary memory addresses
  - Executing “dangerous” instructions
  - Access to I/O directly
  - System registers

Remember Kernel vs User Mode?

- Privileged instructions can only be executed in kernel mode
  - (what happens when user attempts to run?)
- On x86: ring/CPL flag in Code Selector register, 4 modes but 2 are commonly used: 0 = kernel, 3 = user
- After system boot, OS switches to user mode before delegating control to process

Protection Ring



Restricted Operation

- What if a process wishes to perform some kind of restricted operation such as ...
  - Issuing an I/O request to a disk
  - Gaining access to more system resources such as CPU or memory
- Solution: Using protected control transfer (processor has to support it)
  - User mode: Applications do not have full access to hardware resources.
  - Kernel mode: The OS has access to the full resources of the machine

System Calls

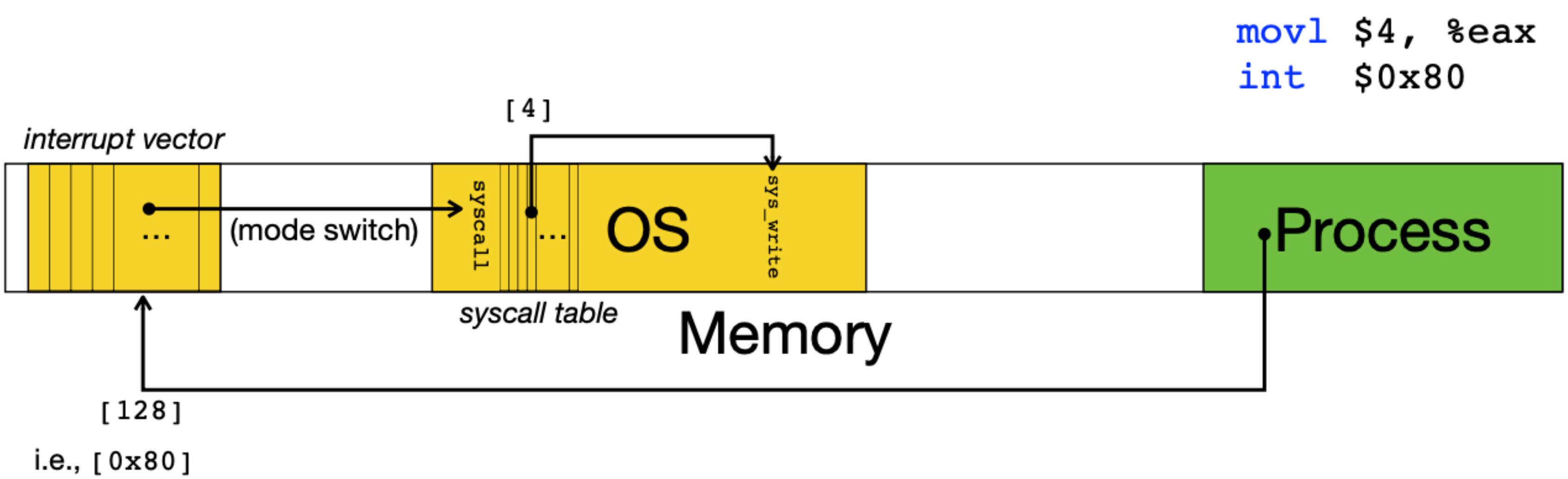
- When user needs to perform I/O, invoke kernel-mode OS functions via system calls:
  - Accessing the file system
  - Creating and destroying processes
  - Communicating with other processes
  - Allocating more memory
- Looks like a regular function call, but isn't!

System Calls (Cont.)

```
char *str = "hello world";
int len = strlen(str);
write(1, str, len);
```

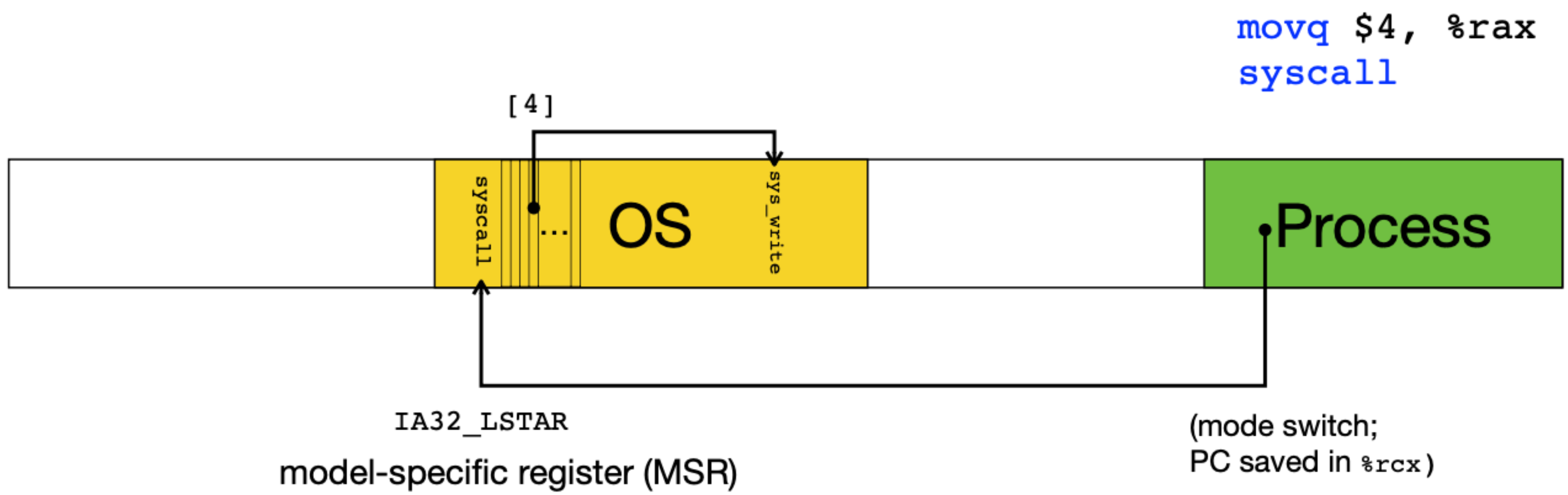
```
movl len, %edx
movl str, %ecx
movl $1, %ebx
movl $4, %eax # syscall num
int $0x80 # trap instr
```

Trap Mechanism (x86)



ILLINOIS TECH | College of Computing

Trap Mechanism (x86-64) syscall added



ILLINOIS TECH | College of Computing

SYSTEM CALLS / TRAPS

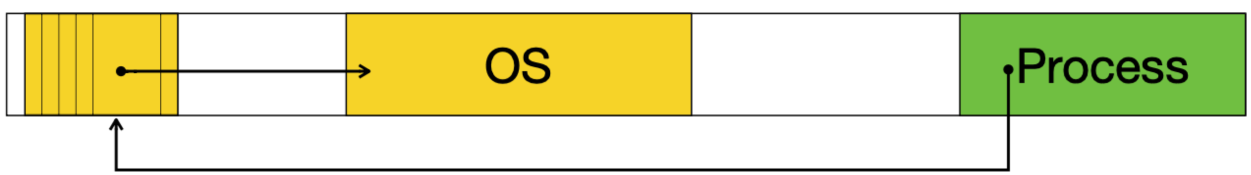
- Defensive programming is key
- Why?
- Each call has its own number

General Interrupt Mechanism

- IDTR (base address register) — populated by privileged lidtr instruction
  - 0-31 reserved for CPU-generated
  - 32-255 software configurable (for sw/hw interrupts) – not all are from User Mode

What to do?

- Problem: when transitioning to OS code, process state may be lost (e.g., PC, SP, etc.)
- Should save in case we return to process after servicing trap

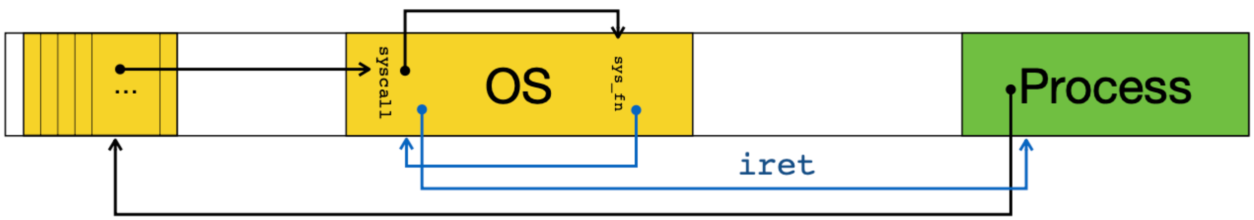


Saving Process State

- Hardware automatically saves current context during trap
- Where?
  - On kernel stack — automatically activated on mode switch
- Every process has its own separate kernel stack and it is used to keep track of kernel state (e.g., while handling I/O)

Restoring Process State

- “return from trap” instruction: iret — pops and restores trap frame and returns to process in user mode



(On x86-64, `sysret` instead; loads PC from `%rcx`)

Do we always immediately return to trapping process?

- Nope:
  - Process may be blocked (due to I/O request)
  - Scheduling decision

System Call (Cont.)

- Trap instruction
  - Jump into the kernel (how to tell where?)
  - Raise (the processor) privilege level to kernel mode
- Return-from-trap instruction
  - Return into the calling user program
  - Reduce (the processor) privilege level back to user mode

System Call (Cont.)

OS @ boot (kernel mode)	Hardware
initialize trap table	
	remember address of … syscall handler

OS @ run (kernel mode)	Hardware	Program (user mode)
Create entry for process list Allocate memory for program Load program into memory Setup user stack with argv Fill kernel stack with reg/PC **return-from -trap**		
	restore regs from kernel stack move to user mode jump to main	
		Run main() … Call system **trap** into OS

System Call (Cont.)

OS @ run (kernel mode)	Hardware	Program (user mode)
	(Cont.)	
	save regs to kernel stack move to kernel mode jump to trap handler	
Handle trap Do work of syscall **return-from-trap**		
	restore regs from kernel stack move to user mode jump to PC after trap	
		… return from main trap (via `exit()`)
Free memory of process Remove from process list		

Switching Between Processes

- How can the OS regain control of the CPU so that it can switch between processes?
  - A cooperative Approach: Wait for system calls
  - A Non-Cooperative Approach: The OS takes control

A cooperative Approach: Wait for system calls

- Processes periodically give up the CPU by making system calls such as yield.
  - The OS decides to run some other task.
  - Application also transfer control to the OS when they do something illegal. Ie. Divide by zero
  - Try to access memory that it shouldn't be able to access
- Early versions of the Macintosh OS, The old Xerox Alto system

What can go wrong?

- A process gets stuck in an infinite loop = Reboot the machine

A Non-Cooperative Approach: OS Takes Control

- A timer interrupt
- During the boot sequence, the OS start the timer (hardware).
- The timer raise an interrupt every so many milliseconds. (hardware)
- When the interrupt is raised :
  - The currently running process is halted.
  - Save enough of the state of the program.
  - A pre-configured interrupt handler in the OS runs.

What does this fix?

- The OS can jump back into execution.

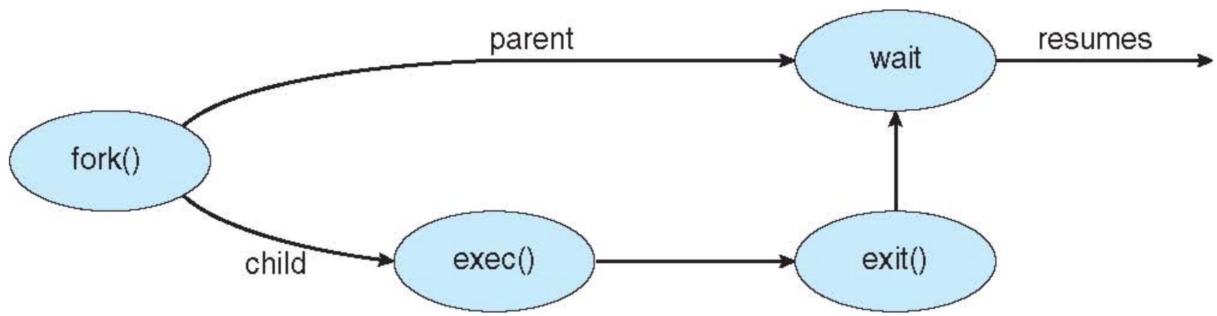
Context Switch

- A piece of assembly code
  - Save a few register values for the current process onto its kernel stack ¢ General purpose registers
  - kernel stack pointer
- Restore a few for the soon-to-be-executing process from its kernel stack
- Switch to the kernel stack for the soon-to-be-executing process

ILLINOIS TECH | College of Computing

Process Creation

- Parent process creates children processes, which, in turn create other processes, forming a tree of processes
- Generally, process identified and managed via a process identifier (pid)
- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate



Process Creation (Cont.)

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - fork() system call creates new process
  - exec() system call used after a fork() to replace the process' memory space with a new program

Metadata about a process

- Metadata examples:
  - PID, GID, UID
  - Allotted CPU time
  - Virtual → Physical memory mapping
  - Pending I/O operations

OS Data Structures

- Critical function of OS is to maintain data structures for keeping track of and managing all current processes
- Layout of many structures are dictated by hardware
  - e.g., VM structures, interrupt stack frame

When do processes end?

- Conditions that terminate processes can be
  - Voluntary
  - Involuntary
- Voluntary
  - Normal exit
  - Error exit
- Involuntary
  - Fatal error (only sort of involuntary)
  - Killed by another process

Process Termination

- Process executes last statement and then asks the operating system to delete it using the exit() system call.
  - Returns status data from child to parent (via wait())
  - Process' resources are deallocated by operating system

- Parent may terminate the execution of children processes using the abort() system call. Some reasons for doing so:
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates

Process Termination (cont.)

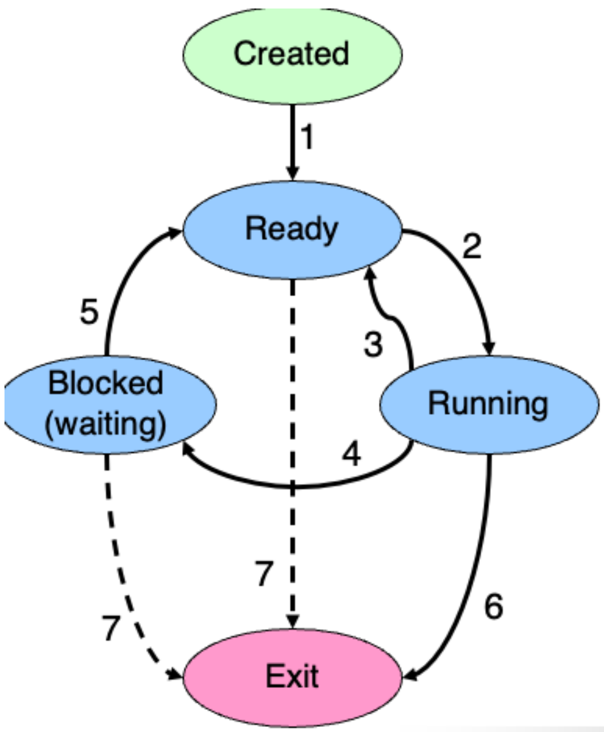
- Some operating systems do not allow a child to exist if its parent has terminated. If a process terminates, then all its children must also be terminated.
  - cascading termination. All children, grandchildren, etc. are terminated.
  - The termination is initiated by the operating system.
- The parent process may wait for termination of a child process by using the wait()system call. The call returns status information and the pid of the terminated process   pid = wait(&status);

- If no parent waiting (did not invoke wait()) process is a zombie
- If parent terminated without invoking wait , process is an orphan

Process hierarchies

- Parent creates a child process
  - Child processes can create their own children
- Forms a hierarchy
  - UNIX calls this a “process group”
  - If a process exits, its children are “inherited” by the exiting process’s parent

Process states



Process in one of 5 states:

- Created
- Ready
- Running
- Blocked
- Exit

Transitions between states:

1 - Process enters ready queue
2 - Scheduler picks this process
3 - Scheduler picks a different process
4 - Process waits for event (such as I/O)
5 - Event occurs
6 - Process exits
7 - Process ended by another process

ILLINOIS TECH | College of Computing

Process Control Block (PCB)

- Aggregate per-process data entry is referred to as the Process Control Block (PCB)
- Implementation likely consists of many disparate structures

Part of the PCB

```c
struct proc {
  uint sz;
  pde_t* pgdir;
  char *kstack;
  enum procstate state;
  int pid;
  struct proc *parent;
  struct trapframe *tf;
  struct context *context;
  void *chan;
  int killed;
  struct file *ofile[NOFILE];
  struct inode *cwd;
  char name[16];
};
```

Size of process memory

Page directory pointer for process

Kernel stack pointer

later

later

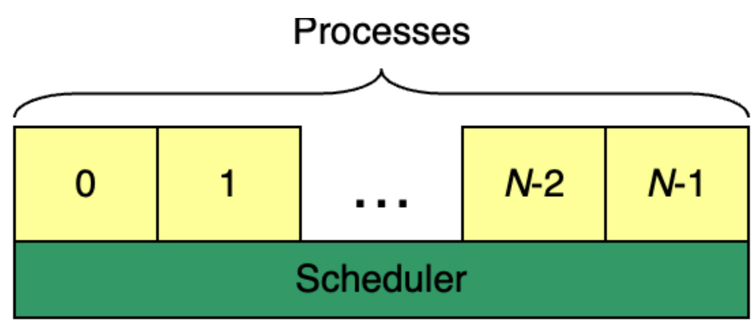
Files opened

Current working directory

Executable name

Processes in the OS

- Two “layers” for processes
- Lowest layer of process-structured OS handles interrupts, scheduling
- Above that layer are sequential processes
  - Processes tracked in the process table
  - Each process has a process table entry

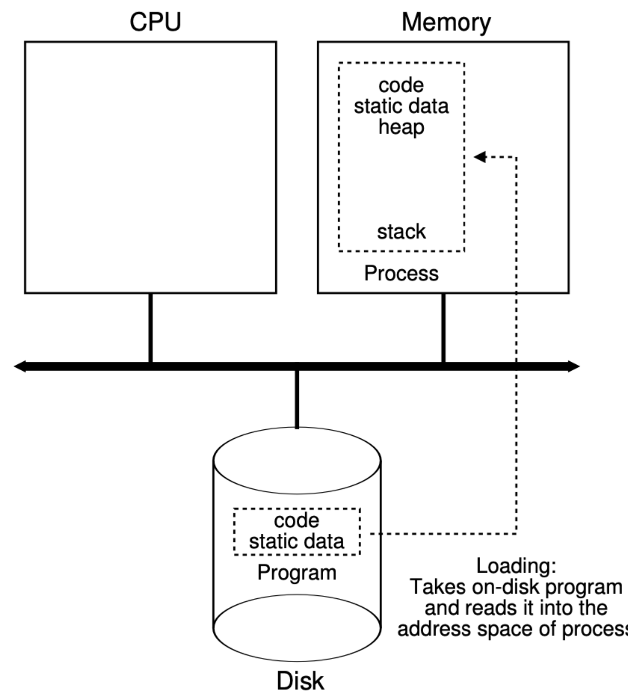


What's in a process table entry?

- Process management
  - Registers
  - Program counter
  - CPU status word
  - Stack pointer
  - Process state .....
- File management
  - Root directory
  - Working (current) directory...
- Memory management
  - Pointers to text, data, stack
  - Pointer to page table

Potatoes vs Potatoes

- Program vs process
- Database vs instance
- Program != process
- Database != instance



ILLINOIS TECH | College of Computing

Context Switches

- Multitasking via virtualization relies on seamlessly switching contexts between processes on hardware
  - Requires frequently saving/loading state to/from PCB
- At any point may have multiple processes ready to run
  - How does the scheduler pick the next process?

What happens on a trap/interrupt?

- Hardware saves program counter (on stack or in a special register)
- Hardware loads new PC, identifies interrupt
- Assembly language routine saves registers
- Assembly language routine sets up stack
- Assembly language calls C to run service routine
- Service routine calls scheduler
- Scheduler selects a process to run next (might be the one interrupted…)
- Assembly language routine loads PC & registers for the selected process

Scheduler

- Scheduler triggered to run when timer interrupt occurs or when running process is blocked on I/O
- Scheduler picks another process from the ready queue
- Performs a context switch

Why schedule processes?

- Bursts of CPU usage alternate with periods of I/O wait
- Some processes are CPU-bound: they don’t make many I/O requests
- Other processes are I/O-bound and make many kernel requests

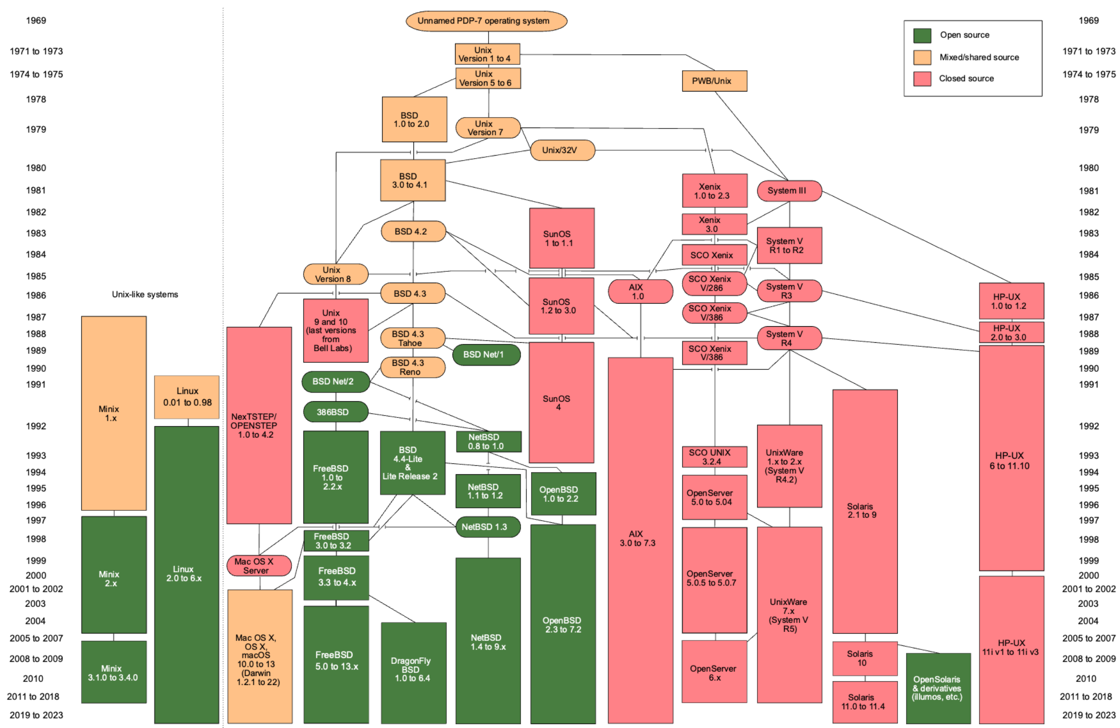
When are processes scheduled?

- At the time they enter the system
  - Common in batch systems
  - Two types of batch scheduling
    - Submission of a new job causes the scheduler to run
    - Scheduling only done when a job voluntarily gives up the CPU (i.e., while waiting for an I/O request)

- At relatively fixed intervals (clock interrupts)
  - Necessary for interactive systems
  - May also be used for batch systems
  - Scheduling algorithms at each interrupt, and picks the next process from the pool of "ready" processes

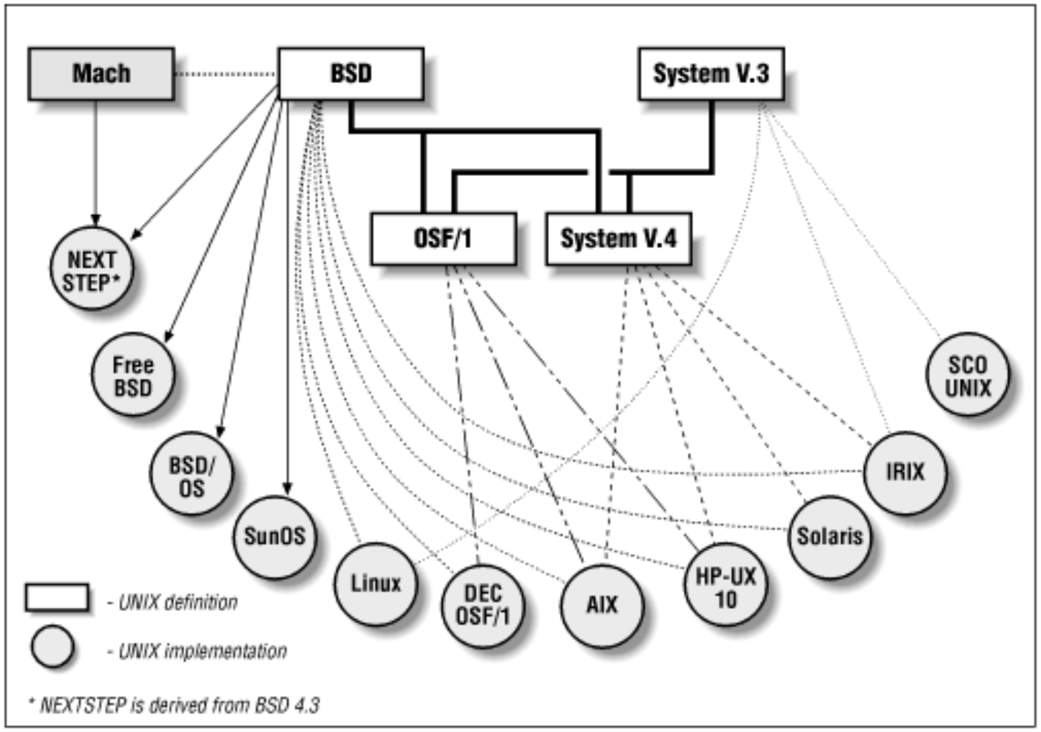
UNIX & AIX vs Linux

- UNIX – Commercial software, HPUX, AIX, Solaris, etc.
  - UNiplexed Information Computing System - Unix is not an acronym; it is a pun on "Multics". Multics is a large multi-user operating system that was being developed at Bell Labs shortly before Unix was created in the early '70s. Brian Kernighan is credited with the name.
  - AIX is developed by IBM and stands for Advanced Interactive eXecutive
  - Proprietary kernels
- Linux – Opensource with sometimes commercial support
  - Redhat, OpenSUSE, Fedora, Rocky, etc
  - Same kernel with tweaks



ILLINOIS TECH | College of Computing

Simple

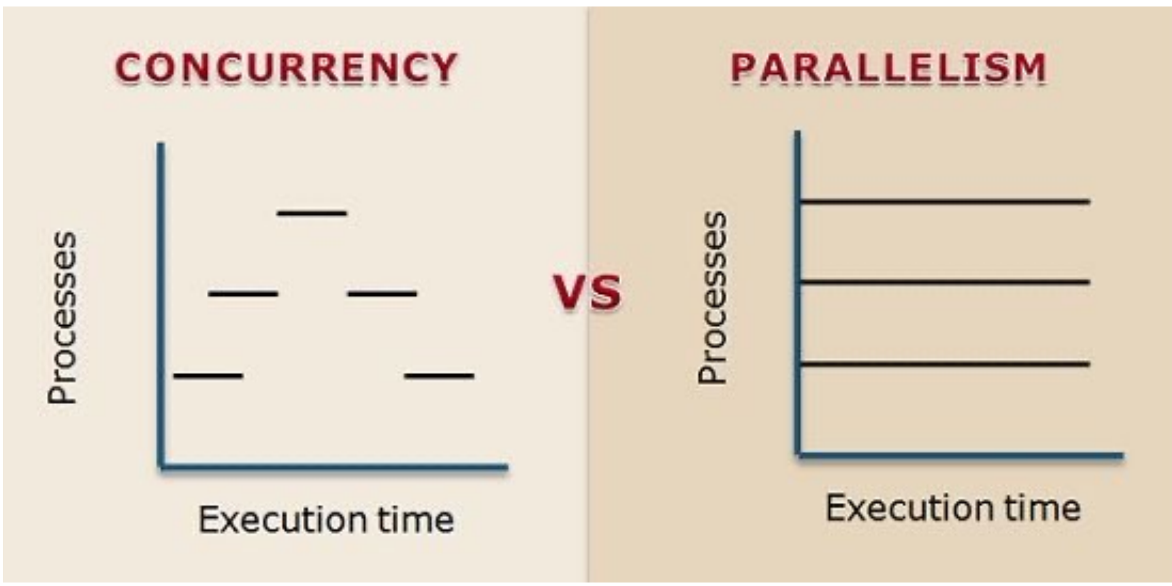


Mach
BSD
System V.3
OSF/1
System V.4
NEXT STEP*
Free BSD
BSD/ OS
SunOS
Linux
DEC OSF/1
AIX
HP-UX 10
Solaris
IRIX
SCO UNIX
- UNIX definition
- UNIX implementation
* NEXTSTEP is derived from BSD 4.3

Concurrency

- Concurrency presents its own challenges and techniques for dealing with them
- Concurrent processes must by isolated from each other
- Nondeterministic execution and access to resources creates race conditions within the OS and between processes
- Dealing with these issues requires special tools and techniques
- Applies to Databases as well

Difference Between Concurrency and Parallelism



ILLINOIS TECH | College of Computing

Persistence

- CPU and Memory state are volatile*
- I/O devices provide support for persistent storage as well as issues:
  - How to namespace persistent data?
  - What OS APIs are needed for accessing persistent data?
  - How to efficiently manage and access data on slow devices? (I/O scheduling and queueing)
  - If processes crash when updating persistent store, how to guarantee consistency?

ILLINOIS TECH | College of Computing