What is a Process?

- Something that has a PID – a "pointer" to a **data structure** in the OS

- A collection of instructions
  - Typically called an **"executable image"**
  - .o file (object file) – or the output of the compiler
  - a.out is the default name, but typically we don't use an extension name for the file used as a command

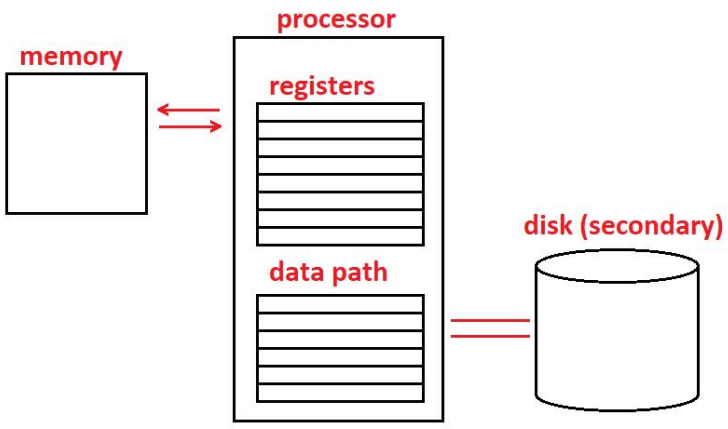
- The executing form of a programming (i.e. running program = process), the contents of **memory** + **registers** + **task structure (archaically known as the Process Control Block)**

Modern Computing Systems

- Secondary storage
  1. Registers – within the CPU
  2. Memory – connected directly to the processor which shares the same clock
  3. Secondary storage
     - Was in the initially read-only during execution (punch cards)
     - Disks + tape could read and write
       ↳ read + write individual **blocks**
- The entire contents of a computer evolves as the program execution proceeds

Process Execution

- Evolving the state of the processor under program control



Small instructions: - load/store, - add

- Purpose of **data path** is to **change the state** of the system

Instructions can be large or small

- Familiar instructions tend to act on a few registers or memory locations
- A single instruction can act on many locations ***(large instructions)***
    - Storing a single value to many locations
    - Add two lists of numbers to produce a list of results
- I/O is a large instruction

[A] + [B] → [C] (Vector operation, example in the GPU)

Von Neumann Computers

- Typically execute **small instructions** of about one or two words.
    - Act on a few registers and memory locations
    - Remember that early computers were small
- Modern architectures are good at executing a lot of small instructions
    - GPU implements **large instructions**
    - Some supercomputers do as well
- Typically, OS processes are executing **large numbers** of **small instructions**

Fine grained execution requires a lot of instructions

- A special purpose machine can have few (or just instruction)
- To be **general purpose:**
    - Needs to execute many **fine-grained** instruction (make **execution fast**)
    - We need to **store** a lot of instructions + **access** them fast
- Modern von Neumann machines have
    - Big memory, **lots** of registers
    - Secondary storage for code and data
    - Fast execution, including memory access
    - [More modern: concurrency in execution]

A modern computer

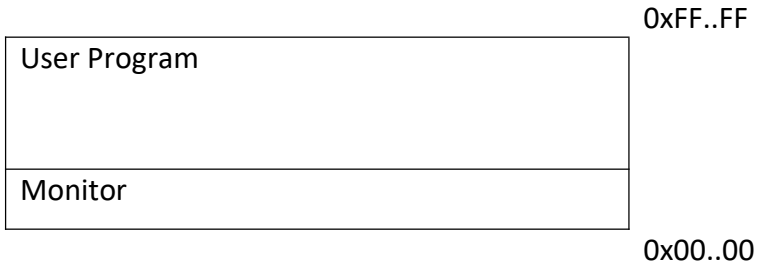
- When you turn it on, RAM is empty.
    - **Volatile memory** – zeros or random at power up of CPU
    - **Nonvolatile memory** – RAM contents are preserved
- We need to load instructions into memory
- Boot loader needs to be **loaded** and executed
    - The boot load is in **read only memory**
    - **Programmable** ROM: “burning”

Boot loader program read in will read from DISK and begin execution

- Things other than the bootloader may be in **READ-ONLY MEMORY (ROM).**
  - Some low level software may be stored in the bootloader
  - Bootloader may be proprietary
- Microsoft and Intel uses a BASIC I/O SYSTEM (BIOS) in ROM.
- Raspberry Pi (open-source)
  - ARM processor
  - Hardware is generically available
  - Raspberry Pi ROM boot software is proprietary
  - Raspberry Pi Alternative: Beaver Board (open-source)

Now we can load + run a longer program

- What about the program that loads and runs other programs
  - Typically called a (batch) monitor
  - Often a program of termination would return control to monitor
- **Monitor** (a simple OS) has to remain in memory during program execution
- The running program cannot overwrite the monitor or its data
- ^ Vulnerable memory, so monitor MUST be well-behaved.



How to keep the program from "killing" the monitor

Approach 1: Don't allow user programs to write values to memory

- Don't execute user code **natively**
- Instead of executing user code, the monitor **interprets** it.
- User code need **not be expressed as machine code**
- Monitor can **refuse** unsafe instructions

- Write code in BASIC, **interpret** it
  - This requires a BASIC interpreter for a **PC (Program Counter)**

Approach 2: Protect the Monitor**:** Call it an "operating system" (abstraction between hardware and software [libraries, utilities, applications] and how shared resources are managed)

Modes of User Program Execution

- Direct execution
    - User program written in "native" machine code
    - Monitor/OS loads user program into memory.
    - Complete control of computer system is to put under control of user program
    - User program may be well-behaved and may return control to monitor on termination
    - "Control": Where the PC (program counter) points

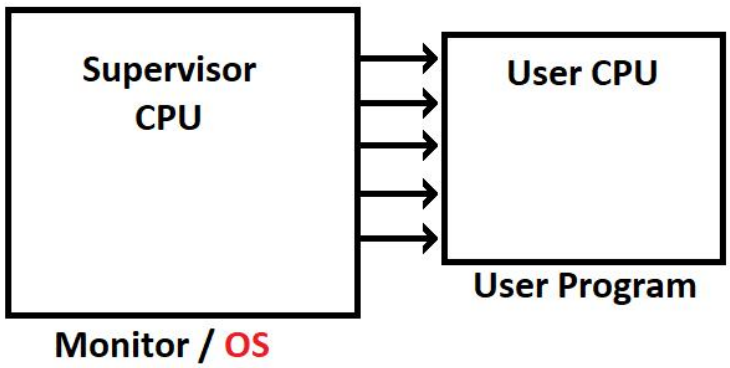
Interpretation/Simulation

- User program **interpreted** by software
    - Not native code (but may *look* like source code)
    - Interpretation can be made safe
    - No direct dereference of pointers
    - Every operation can be inspected by software [**POLICY]** applied.
    - Interpreter is well-behaved, is part of the monitor
    - **DOWNSIDE:** Interpreter is **[SLOW]**

[Policy] ← **Trade-off** → [Speed]

Emulation

- Interpretation with **hardware** assistance
    - Emulab at Utah: network node + **fiber**
    - Quantum computing: electronics + quantum
- We can speed up native code execution using **"emulation"**

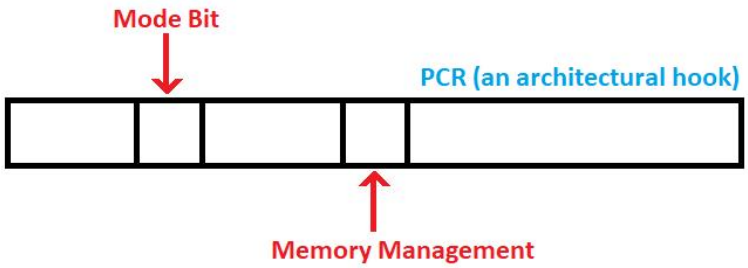


Instead we have:

- User mode
- Supervisory mode (direct execution)

Policies Implemented in User Mode

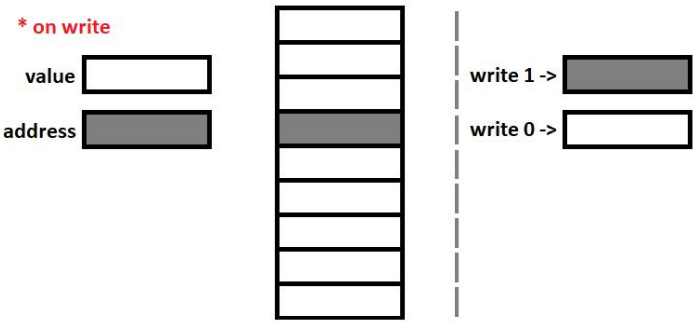
- How does the processor know if it is in user or supervisory mode?
  - The user/supervisory (kernel) mode **bit** which is on/off
  - We need another register to control the **processor**
  - We call this the **Processor Control Register (PCR)**



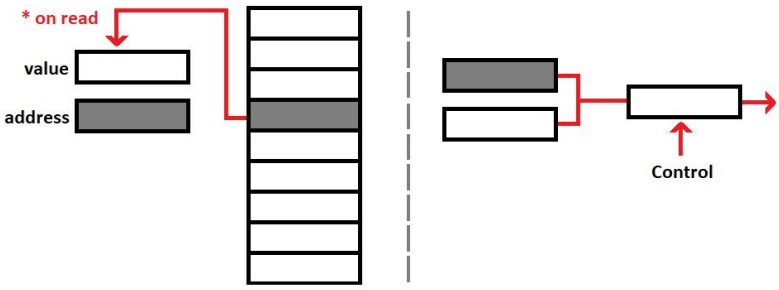
- What does memory management bit do? *[access memory - load/store, we discuss base + limit later on]*

Memory Management Controls Access to Memory

- What is a physical pointer
  - Used to specify which **location** to read or write (load or store)
  - Architecture has addressable memory location

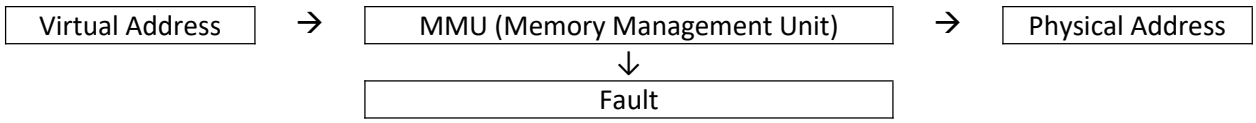


Address on write is input to a **decoder**



On read, the address is control input to **multiplexer**

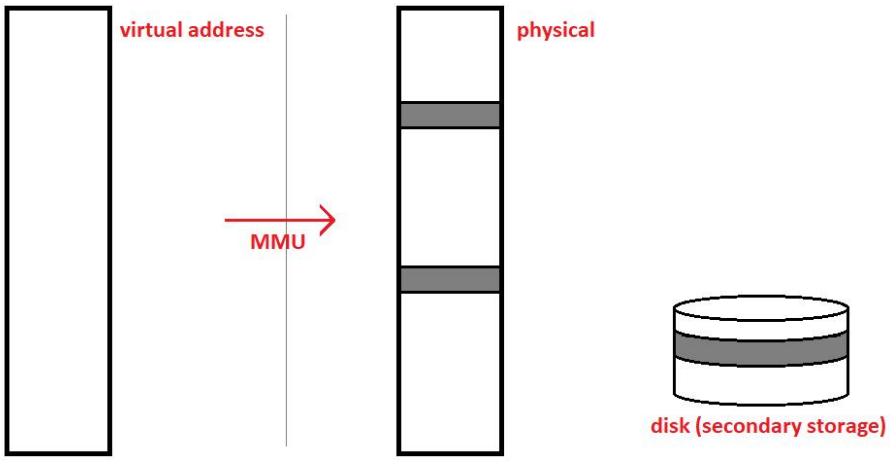
How does memory management apply policy to memory operations?



The MMU adds a level of **SAFETY**

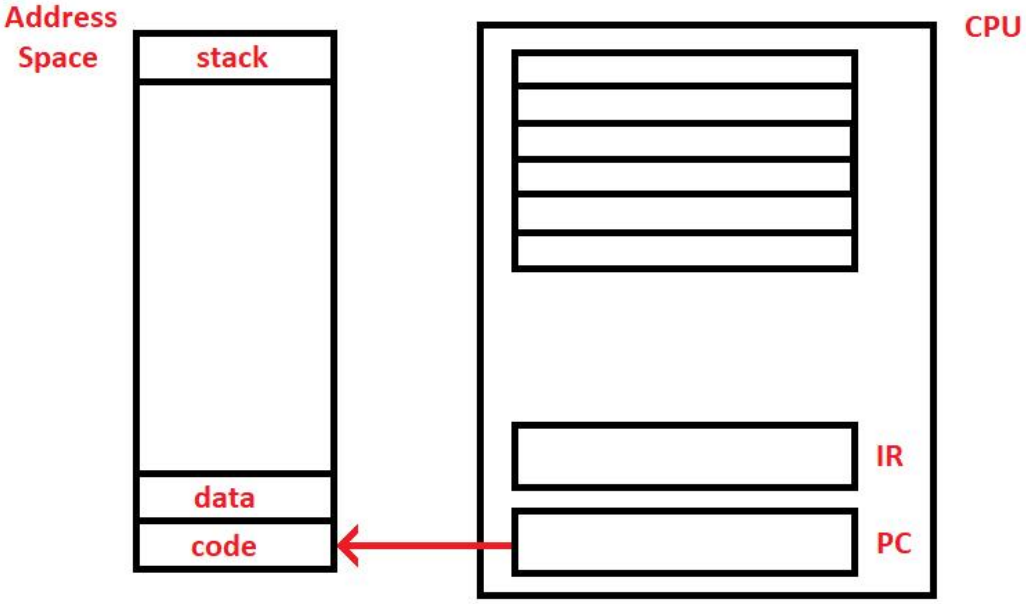
Execution of User Code can be quite FLEXIBLE

- During native execution of code in user mode, there is little flexibility
  - Instruction meaning is set by **architecture**
  - Safety + translation is set by **MMU**
- On a fault, the OS can do anything it wants
  - Only some of the contents of address is in **memory**
  - If the program accessed non-resident part of address space



Executing a program using Native Execution

1. Native execution: the fetch/execute cycle



Fetch-execute cycle:
- **Fetch**: Get contents of memory pointed to by PC into IR
- **Execute**: Do whatever the IR says

2. Suppose our program is in an infinite loop
   loop: br loop

What is a Context Switch?

[Summary: Suspend a process and save its state to be resumed later, and load a different state to resume a different process]

What is an execution context

[HARDWARE CONTEXT]
- Register contents that determine/control the processor's execution of code
  - Program counter (tells what instruction to fetch next, but not IR)
  - General registers (accumulator)
  - **[Processor Control Register]:** PCR

[FIXED]
- Contents of memory is accessible to the program

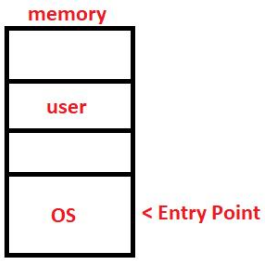
If a program is executing natively, how can execution be suspended?
- Stop fetching – but then processor is "bricked"
- or – store hardware context to memory + execute other code

Switching from User Context to Operating System

1. Operating system regains control from on executing process is to store its hardware context + instantiate its own
2. OS (re)starts execution of a process (which is in memory) is to load its hardware context (stored or **initial**)

Switching from User to OS

1. Store user context
2. Load a PC value corresponding to the OS.
    - Fixed PC value (entry point) then OS figures out what happened
    - A vector of settable PC values – vector is in memory somewhere fixed or pointed to by register



3. Causes of switching **[IMPORTANT TO KNOW for EXAM]**:
    - Hardware Interrupts
    - Clock interrupt (60 Hz)
    - Trap instruction: System calls
    - Software interrupt (segfault or arithmetic error)

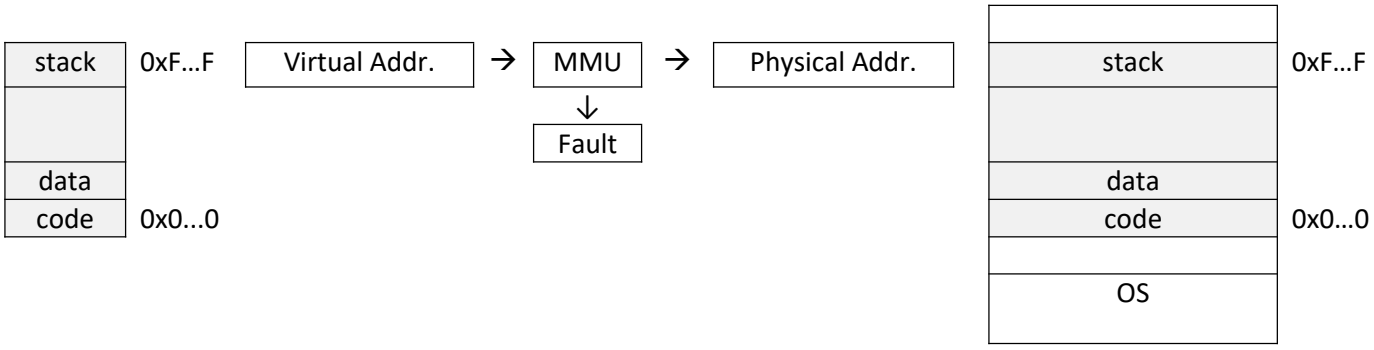
This context switch is an example of an **architectural hook**.

Memory Context

Registers are easy: store to memory
load from memory

- Some architectures have instructions to load/store sets of registers **(performance optimization)**

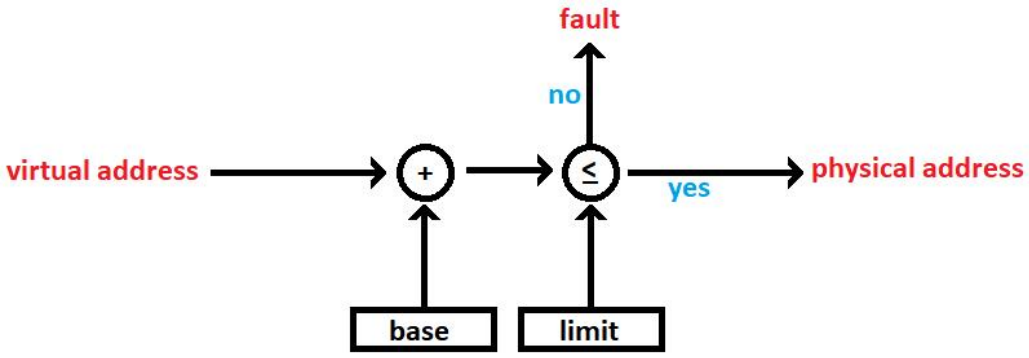
The Memory Management Unit (MMU) controls the view of memory during native execution (user code)



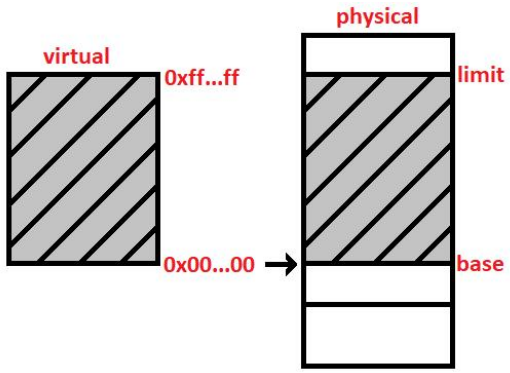
Controlling Memory Context

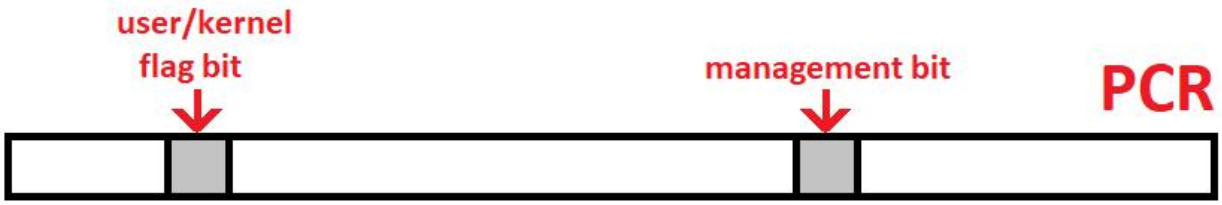
Memory Management Registers (part of hardware context) -- Components

- Base address
- Limit



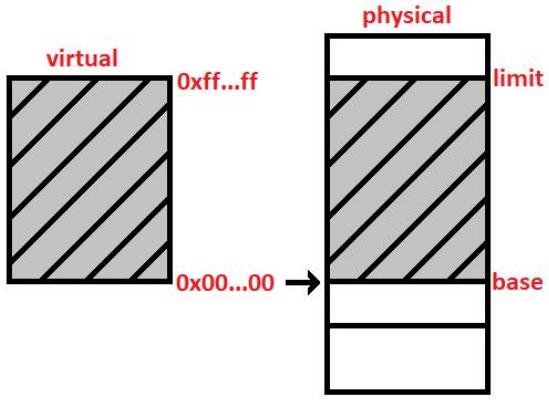
Sandbox safety – There are different images of memory at the same time



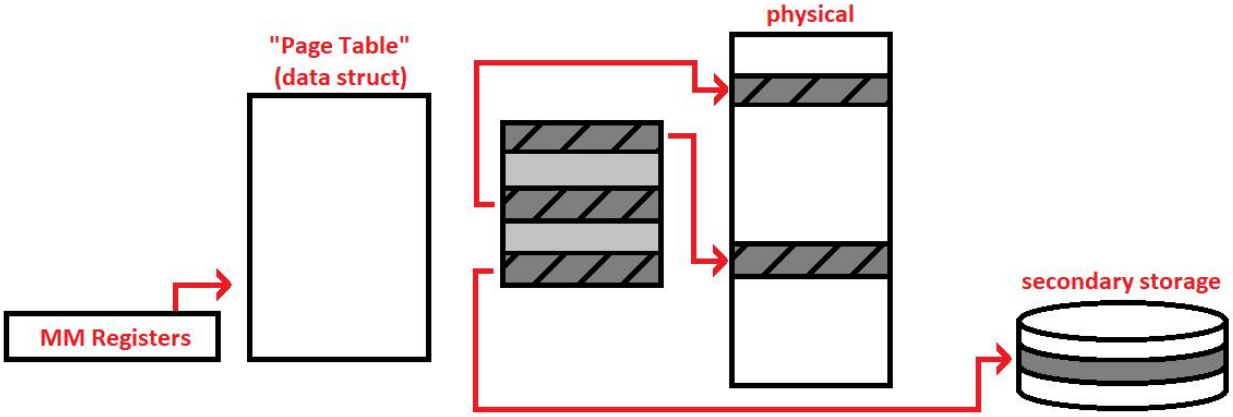


Hardware Context + Switching

- User accessible registers
- User visible but protected registers
- Hidden registers – implement OS control (we don't want to expose pointers to user)
    - Memory management
    - Simple model: **base + limit** ← simple form **more modern, accurate form is **paging**

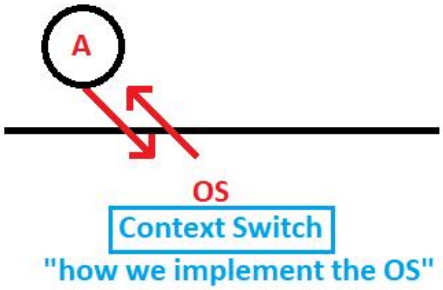


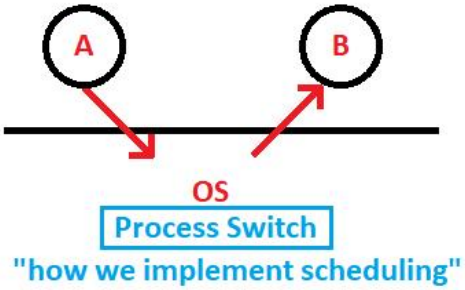
    - More complex memory management is possible (paging)



Possible that dereference can cause a fault

User to OS Transition (Context Switch)





- While user process is running:
  - User context is loaded into registers
  - User code can modify general registers
  - Memory access controlled by MM (Memory Management) registers
  - Processor mode (user/kernel) and MM mode controlled by Processor Control Register
- On switch, OS context is loaded

  Simple approach: save user PC + PCR values

  Load OS PC + PCR – kernel mode, no MM

Managing the User Context

1. We define a data structure within the kernel to hold the user process' context
2. Upon staying in OS upon entry, store user context in Process Control Block **or Task Data Structure**
3. Other OS information relating to a process is also in **fields** of the task data struct.

(task data struct may point to MM struct)

- PID
- Runtime + Execution time (user + system)
- Priority (niceness)

* Process attributes are stored in fields of the task data structure

* Process is the task data structure and the things pointed to by the task data structure (in the context of the OS)

Scheduling Decisions

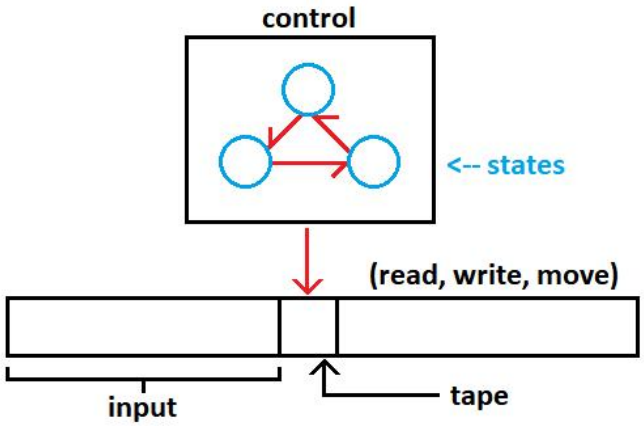
1. When the OS is running and it decided to run a user process.
    - The OS can choose among any process that is ready to run.
    - The choice of which process to run is called process scheduling.
    - The basis for choosing which process to run is called scheduling policy.

The Sandbox architecture hooks *Intel 386/486*

1. While in user mode, a process cannot change the MM registers (which parts of memory it can read or write)
2. In user mode, a process cannot change to kernel mode
3. In user mode, various other features of the processor are modified or restricted (also called trap door)

Process State

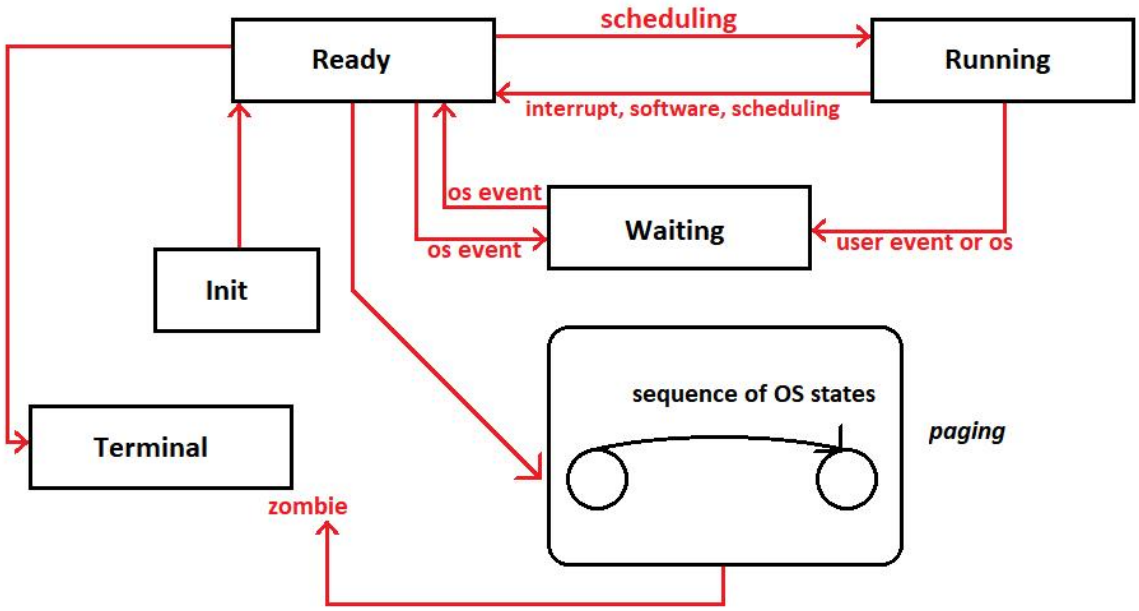
1. Turing machine goal to model human intelligence: "Turing test"



2. Encoding: A **state** is an integer
   - Tells us what is going on with the management of a process at any time

   *** Running:** register context loaded in hardware
   *** Runnable (Ready):** can be made running
   *** Waiting:** will be made runnable by OS eventually

State Transitions



Process Creation

- Fields of the task struct are sensitive/critical
  - User code cannot see or modify
  - Must be interpreted properly by the operating system
- How does parent specify fields of the task struct **safely**?
  - Priority]
  - Sharing |---- process creation *used to be* a privileged operation
  - Contents of memory]

fork()

Process creation does **not** have to be privileged (notion of fork() opened up innovation in multiprocessing)

Memory Allocation within OS

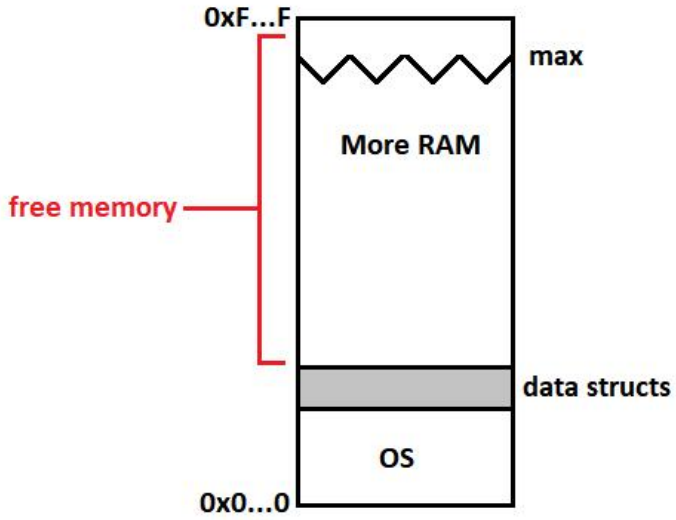
1. Task Control Block (task struct)
2. Open files
3. Memory management data structs (page table)

- What does dynamic memory allocation do within kernel?
- malloc() / sbrk(), mmap(): the heap is used in user code
- UNIX/Linux doesn't use the heap
  - Statically allocated arrays of data structures (size compiled in)
  - Or grab memory at boot time
  - No fragmentation

How fork() works

1. Easy concept: copy parent Task Control Block (TCB) into the child TCB
    - Some fields have to be different
        - PID
        - User + System Times
        - Parent
    - Some data structures cannot be shared between parent and child **(complex examples)**
        - Memory management
        - Code can be shared (but READ-ONLY)
        - Writeable data cannot be shared (including the stack)
        - Contents of data memory must be copied

Free memory is managed on a free list



More TCB fields

- Each TCB field has properties
    - Copied or duped
    - Sometimes allocation
- Exactly the same
    - Priority of child equals that of parent
    - uid and guid
- Some can be changed for administrative purposes or "management"
    - E.g. increasing priority is a "privileged" operation
    - Change user – sudo *[root = superuser or supervisor user]*

Kernel Mode vs. Superuser

1. Kernel mode is a feature of the processor architecture
    - Controlled by mode bit in the PCR
    - Affects instructions + machine behavior
    - Architecture reference manual details mode differences
2. Superuser mode is implemented by software by the OS kernel
    - Controlled by the UID field in task struct
        - *If (t->uid == 0) then { su code }*
          else { nonsu code }
    - Affects system call behavior
    - Look in (Linux) man pages

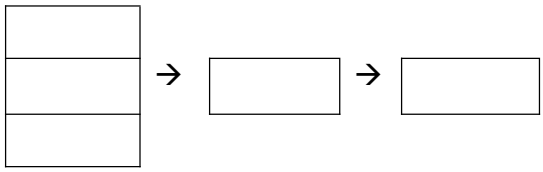
* Important topics to know for Exam: (1) Context Switch, (2) Kernel vs Superuser mode

Managing Task Structs

1. Central data structure of the OS
    - Represents a process (threads)
    - In order to act on a process we have to find its task struct
2. When a process in running, we know how to find it
   (point to task struct of currently running process)
   [in a multiprocessor, we would have an array of pointers]

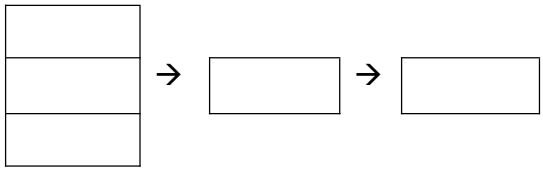
- Other processes are READY or WAITING
- We keep READY processes in a queue
- Other processes are on other queues

Terminal 3



(character device, waiting for the next character)

Disk 0



(some job of looking up)

** which queue depends on which event*

fork() and wait()

- fork() creates a child process
    - The child has a pointer to its parent
    - The parent has a list of its children
- A parent can wait for the child to do something
    - The wait() system call puts the parent in a WAITING state (int field in task struct)
- When child calls exit(), the parent is woken up if it is WAITING

Process Scheduling

1. At any point in time (at most) one process is RUNNING (on a uniprocessor machine)
    - Many processes may be READY
    - Whenever the OS is entered, a process switch is possible
        - Continue running same process
        - Or make the current RUNNING process READY and choose a READY process to put into the RUNNING state and run.
2. That choice is **process scheduling**.

First In First Out Scheduling

1. We actually manage the READY process as a queue
    - When a process becomes READY we put it at the end of the ready-to-run queue
    - When we need to make a new process RUNNING, we use the one at the head of the queue
2. ***ISSUE:*** A **long job** monopolizes the CPU – so short jobs have to wait for it
   (in some situations we want short jobs to complete for good metric of the CPU)

Evaluating Scheduling

1. Turn around time
    - T_complete – T_entry (time complete – time of entry)
    - Incorporate waiting time in queue + execution time
    - Batch processing model (not interactive)
2. First response
    - T_input – T_entry
    - Time in queue + Time until first interaction

Average Turnaround Time

- Assume a fixed set of jobs in queue
- Schedule until ALL have completed
- Take the **average** of turnaround times
- FIFO can do poorly on this metric (shortest jobs first is better than FIFO)
    - If all jobs are same length then any scheduling algorithm has same result.
    - If a long job goes first, then later jobs (shorter) have to wait

Convoy Effect

FIFO:

10	3	2	
10	+13	+15	= 38

NOT FIFO:

3	2	10	
3	+5	+15	= 23

2	3	10	
2	+5	+15	= 22 **(OPTIMAL)**

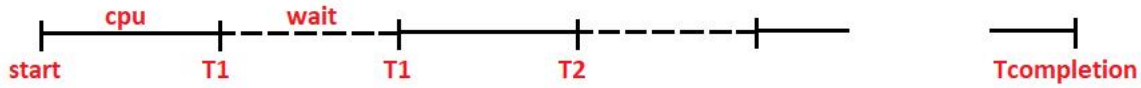
SHORTEST JOB FIRST is optimal with respect to average turnaround time BUT impossible to implement

Obtaining CPU Runtime

1. Actual CPU runtime is not knowable by the scheduler
    - Strategy: ask the user
      (but the user doesn't know)
      (but it is their fault)
    - Other strategies?
        - Use **history** for the same program
        - Perhaps throw in size of data
        - User identity
        - Machine learning??
    - Try to approximate (the overall thing about CPU runtime)

Impact of I/O on Scheduling

1. While I/O is going on, a job is in WAITING state so it cannot run
2. The simple continuous execution model doesn't fit well when programs do I/O during execution



3. But multitasking allows us to switch processes on I/O

What About Shorted Job First?

1. When we switch away due to blocking I/O then the process becomes READY, how do we schedule it?

[* with FIFO (first-in first-out "first come, first served"), stick the process on the end of the queue]
* with SJF (shortest job first), we could use the initial estimate
* we could reduce the estimate by elapsed CPU time
  - Shortest Remaining Time First
* what to do when estimate is negative

Scheduling Policy

- Negative remaining time: kill the job
  * job might be about to complete
- Another approach: negative estimates are ok, just take the most negative first
- Other approaches: charge more over estimate

Scheduling Policies

- FIFO (First Come First Served): convoy problem, nonoptimal
- SJF and SRTF: requires too much knowledge of future
- Response time as a metric
- I/O and CPU Bursts
- Priority
- Fair Scheduling: Similar to Linux

Response Time

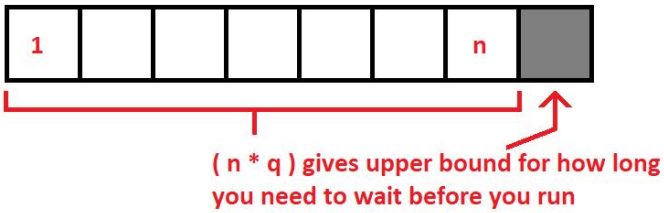
- T_first - T_queue = response time
- Problem: Running every job to completion or until it relinquishes the CPU (e.g. via hardware interrupt, clock interrupt, system call); gives very high variability in response time.
  * A short or highly interactive job can wait a long time before it is first scheduled. High level of **"unfairness"**
  * What about scheduling on device interrupts or clock ticks?

- Scheduling on interrupts or clock: preemptive
  * prevent/limit how long job can wait
  * in SJF (shortest job first), there's unfairness to longer jobs
    - a long job could be constantly pushed to the back of a queue and never runs

    Starvation – doesn't run because job has low priority (^)

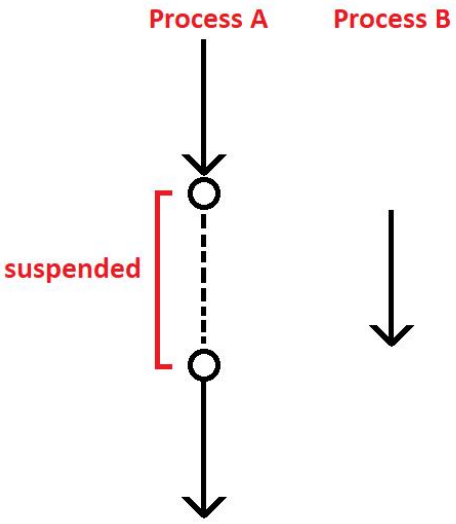
Round Robin Scheduling

- Based on time rather than job length
- Scheduling is based on a time quantum (generally some # of clock ticks)
- A job can run for **at most** (can also run for less) one quantum before it is preempted.
- When a job exhausts its CPU quantum, put it at the end of the FIFO READY queue.



n * q = upper bound on response time

Actual response time still not known, but we've placed an upper bound so **NO STARVATION** in Round Robin scheduling.



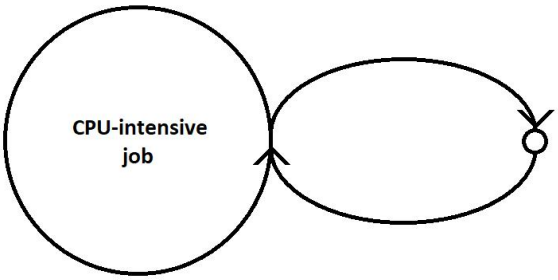
(An example of "concurrency" utilizing early time-sharing)

Scheduling Summary:

Turnaround Time (CPU)	T_comp - T_entry	Shortest Jobs First (Non-preemptive), Shortest Remaining Time First (Pre-emptive)
Response Time (I/O)	T_firstrun – T_queuearrival	Round-Robin (Pre-emptive)

I/O Based Scheduling (Taking I/O into Account)

I/O → relinquishing CPU



- Two categories of job:
    * CPU intensive
    * interactive

    (burst has high priority, but less as CPU goes on until we relinquish it -- hysteresis)

- Each program runs for some period of time before relinquishing the CPU
    * one extreme: never relinquish
    * another extreme: constantly relinquish

- CPU burst: how much total CPU time the process uses before relinquishing

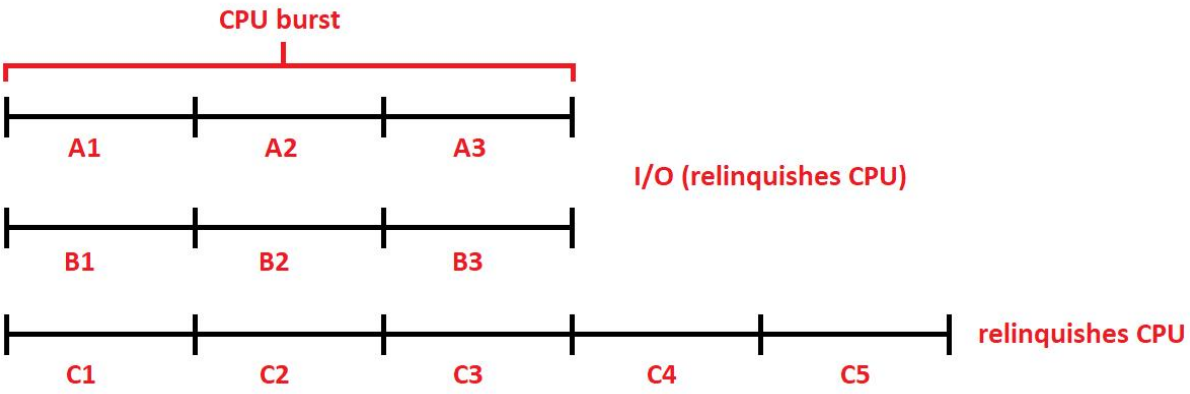
We can keep a history for CPU burst and calculate a metric. Then you can use it in scheduling.

Two Points of View

Machine point of view: jobs run for short periods and process switch interleaves them

1. **Real time**

2. **CPU time**



Scheduling Priority

- Treat READY queue as a priority queue (insert in the middle, keep sorted by some metric or comparison)

- Job priority can be assigned statically
    * job categories: student, researchers, faculty, admin
      0, 1, 2, 3
      (rank from lowest to highest)
    * order queue based on a pair
      (category, priority)
    * sort lexicographically (strict priority → note: starvation to the student jobs)
    * strict has danger of starvation

Multilevel queue: moving jobs between categories

(Summary: Don't do strict priority to avoid starvation)

Priority with Aging

1. Every process has a priority
    * in POSIX (Linux), this is called "niceness", a field in the task struct
    * niceness can be changed through calls to nice() system call
    * non-root processes can only increase niceness (decreases priority)
    * other than calls to nice(), niceness doesn't change
2. Time since process entered READY queue/state is factored in.

Effective priority: A formula that makes sure a process doesn't sit forever in the queue
Thus, the longer you sit in the queue, the higher the effective priority.

Eliminating Complex Scheduling

1. How to give every process a fair shot at the CPU regardless of its history.
2. How about choosing at random?
    - you won't get good effects from complex scheduling policy
    - it's easy to implement correctly
    - it's hard to game
    - useful particularly in the face of competition or malicious actors/processes

"Lottery Scheduling"

Scheduling Approaches

- FIFO: data structure
- Optimality: shortest first
- Approximations to optimal
    - Tradeoffs
    - Irregularities/vulnerabilities

- Policy-driven mechanisms
    - User directed
    - Markets / currencies
- Complex combination of optimality + policy
    - Linux scheduler (open-source allowed people to open up scheduler and modify it)

Linus Torvalds

Lottery Scheduling

- Randomness for simplicity + statelessness
- Priority of different processes
    - Pool of resources allocated to a new process
    - Ability to assign CPU resource from one process to another according to policy (something you cannot do in Round Robin)
    - E.g. When a parent forks, half of its CPU resource
        - Is assigned to a child
- Each process holds some number of "tickets"
    - On a process switch, choose a ticket (not a process)

(The higher the # of tickets, the higher the priority; this is a different form of "niceness")

Implementing Lottery Scheduling

Fairness increases with a larger job length.

- How to assign tickets?
    - Root (supervisory) processes can choose their number of tickets
    - Login can assign a standard number to a new user "shell"
    - We can split up tickets between parent + child
    - Maybe add a parameter to fork saying how to split parent tickets
    - Return child tickets to parent upon termination
    - Request for more tickets

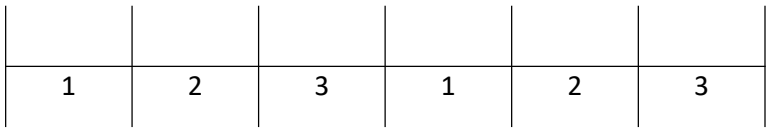
Moving tickets between processes

- Could add a mechanism to send a message between processes and include tickets
- A process could "sell" or "buy" tickets (on what basis?)
- Perhaps allow one process to "loan" CPU resources to another that does work for it
- A compromise is stride scheduling

// Aside:
// Concurrency is largely tied to the logic of the program.
// Giving control to the end user has problems.

Stride Scheduling

- A variant of Round Robin



<-- in-between lines is a fixed quantum

    * a process that relinquishes the CPU gives up the remaining time
    * all process have equal access of CPU

- Stride
    * scheduling quantum is fixed
    * "charge" different processes at different rates for each quantum
    * keep track of how much "charge" (virtual time) each process has used

Who to Run?

- Choose the process with the lowest total charge / "virtual time" to run
    * we have to keep a running total for each process
    * keep a virtual time clock which is incremented by the "stride" of the process (the longer the stride, the lower the priority)
    * order the READY queue by total virtual time used

(STRIDE IS FAIR IN THE SHORT TERM) - Stride length is inverse to the number of tickets allocated
(LOTTERY IS FAIR IN THE LONG TERM)

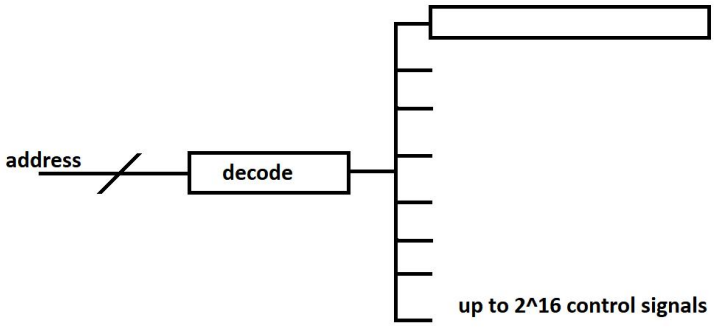
Stride Scheduling and I/O (or anything causing process not to block)

- What happens when a process isn't READY?
    * other processes increase in total virtual runtime
    * What happens when a process (taken out of READY state) returns to the READY state?
    * Will it hog the CPU? (We wouldn't want it to; easy to game the CPU)
- Same problem with new processes?
- Choices (solutions):
    * assign an average total to the new process (stick in the middle of queue)
    * set all processes to zero (get rid of hysteresis by starting over)

    [These choices will **disrupt** scheduling temporarily]

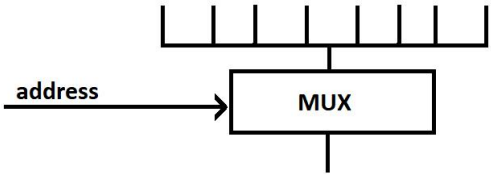
Address Space

- RAM is the only program address space that is a “REAL THING”
    - it has a physical implementation
    - when writing, an address is decoded into a “write” signal on one memory location

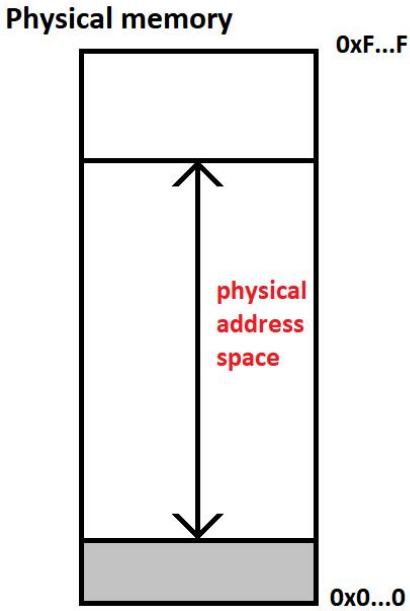


- when reading an addres is the control on a MUX?

“Physical”



up to 2^16 values



(Lower memory is commonly reserved and used to communicate with disk, devices)… so the usable address is somewhere above zero address.

Memory Virtualization

Two functions

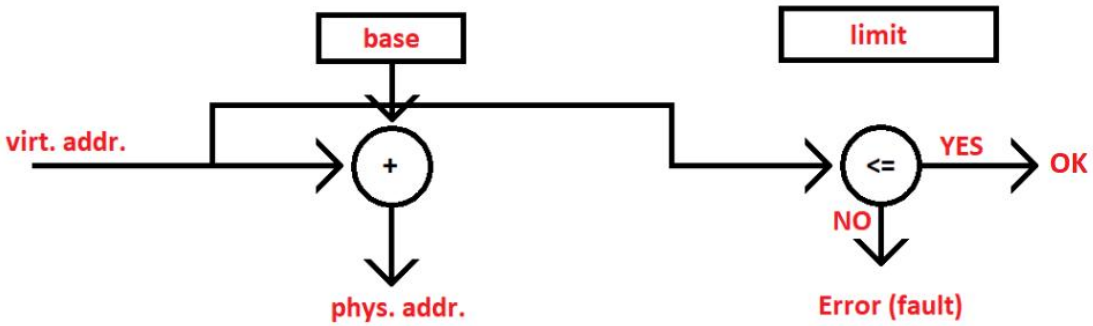
- Relocation
    * A virtual address space starts **close** to zero.
        (Zero is a NULL pointer, so dereferencing is an illegal instruction)
        (But DON'T assume zero will always be NULL)
    * All address spaces start at the same low value
    * This allows a compiler to generate code with the addresses of global variables allocated by the compiler

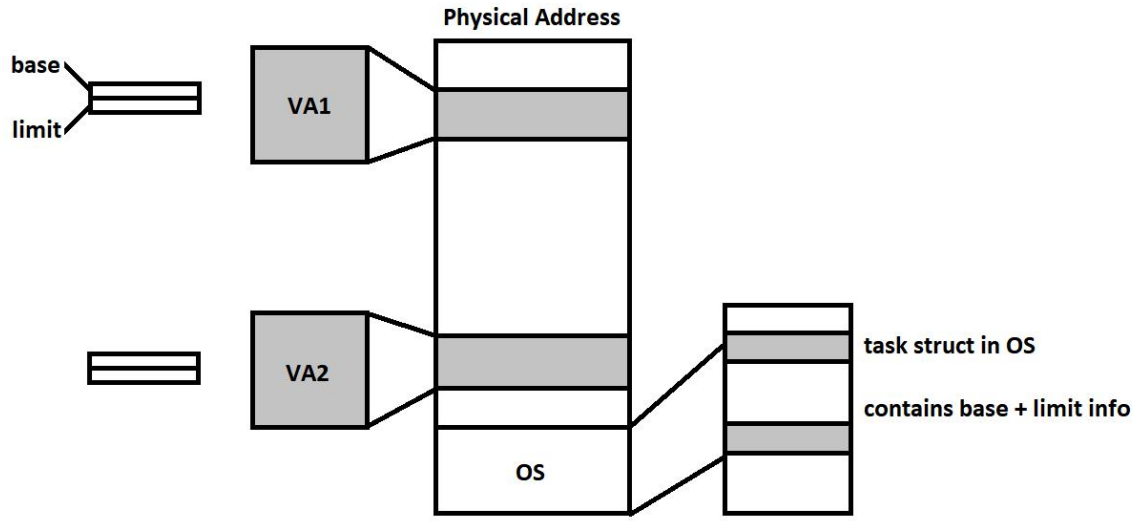
- Memory Protection
    * A specified area of memory is accessible to the program executing with memory management
        - Memory outside of this area is **not** accessible
        - This is important in **isolating** the effect of user program execution
            * from the OS kernel
            * from other user programs

Simple Base/Limit Memory Management

- Base register specifies where in physical memory virtual location **zero** maps to in physical memory.
- Limit register specifies the highest virtual address that is **legal**.
- Memory management bit in the Processor Control Register (PCR) specifies if memory management is on or not.
    - on: virtual addressing is used
    - off: physical addressing is used

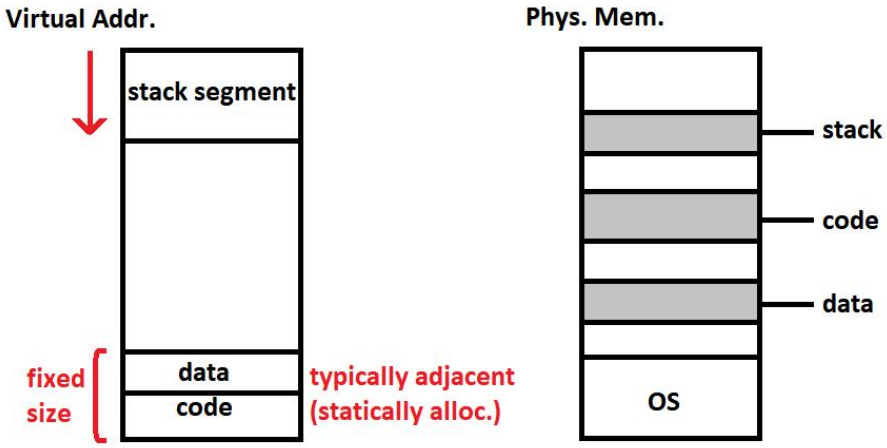
Function of MMU





Base and limit reside in the task struct

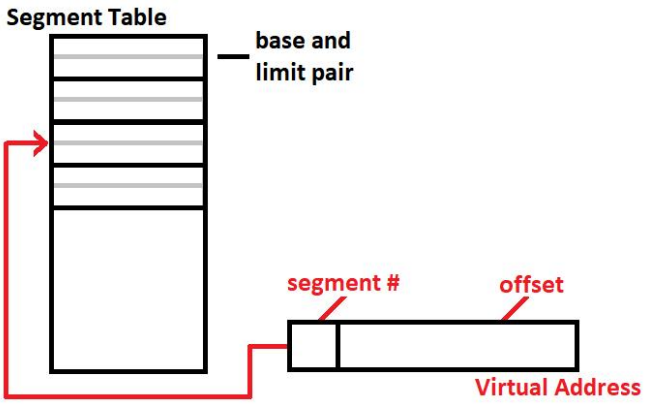
Segmented Virtual Memory (note: today, we use paged virtual memory instead of segmented)
- Multiple contiguous areas of virtual address space, not necessarily adjacent



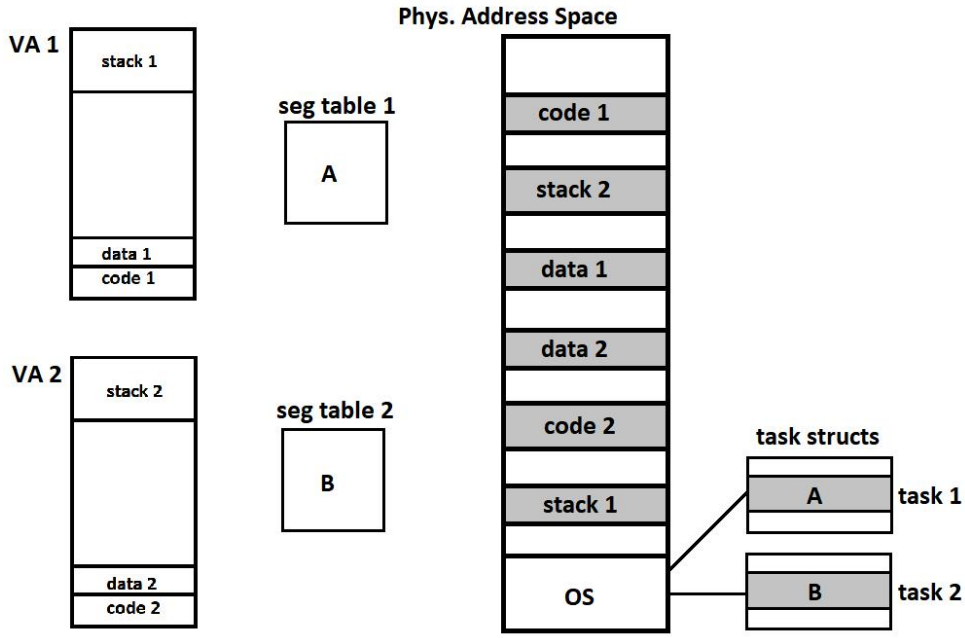
Virtual address:
- Stack segment is read/write: can be fixed size, but typically grows down
- Data segment is read/write: fixed sized
- Code segment is read-only: fixed size
- Only exists through mapping with physical memory

Physical memory:
- Segments (of stack, data, and code) can be allocated anywhere

Segments Implemented Using a Segment Table
- Each entry in the table describes one segment.
    * Each entry consists of a BASE and LIMIT pair
    * MMU has to map each virtual address to a physical address using the appropriate base and limit.



Multiple Segmented Virtual Address Spaces



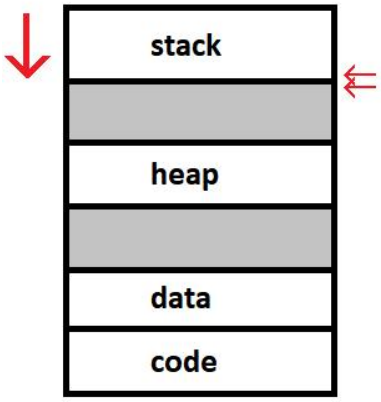
SLAB management's goal is to reduce fragmentation in memory

Writing + validity is controlled by bits in the segment table entry

Address Space Management

- The heap is allocated between data segment + stack segments.
    * both heap and stack can grow
    * on stack overflow, segfault causes entry to OS in kernel mode (stack grows automatically)
        - it could terminate the process
        - it could increase size of the stack & restart execution

        (on segfault, a program count does not occur)



How does the heap grow?

Context Switch:

Malloc knows we ran out of memory, but doesn't segfault
- clock tick wouldn't help (clock tick is used primarily in scheduling)
- there's no external event or interrupt that occurs
- **a trap instruction / syscall**

- The user program (malloc library) requests more heap memory by "making a system call"
    * executing the TRAP instruction
    * sbrk() or mmap()
    * parameters indicating how much memory + which segment
    * OS finds memory and adds to heap segment or indicates it won't (I.e. return a NULL pointer)

How is memory returned to the OS?

- A user program calls free() to return an allocated object to the heap
- Upon process termination all memory is returned to OS

How do Computers Run Fast?

1. Material Science

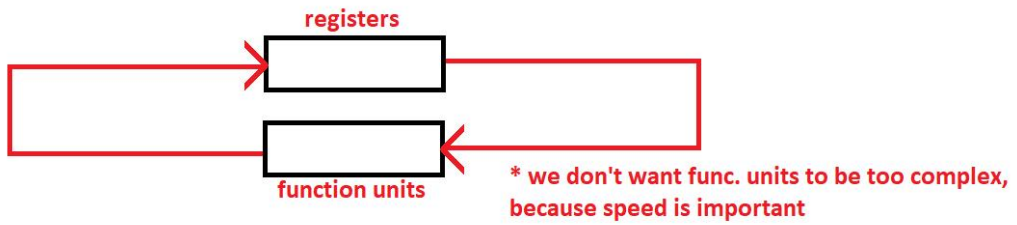
    - Semiconductor logic to change states quickly (fast switching)
    - Miniaturization (switches + wires) => leads to high density on a chip

2. Boolean logic can do a lot quickly In a small space

    - Register values are selected by the multiplexer and are operated on by a function unit (e.g. an adder, rotation, logic ops) and stored back in a register [it all happens in one clock cycle]

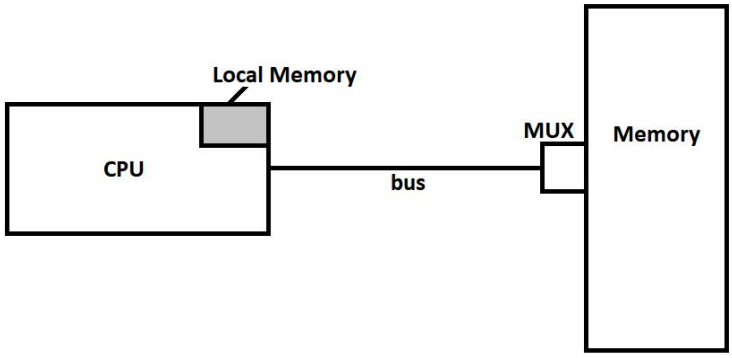
Processor clock is in gigaherts (thousands of times per second) unlike 60 ticks / sec which is in CPU scheduling clock

Data Path:



Access to Main Memory is Slower

- Main memory is “far away” from the CPU in both space and processing time

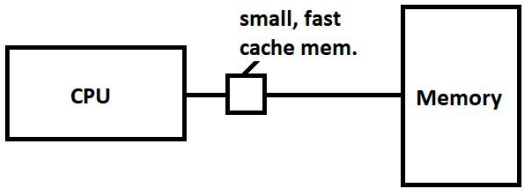



If we try to increase the clock, the CPU gets too hot (risking melting), so there’s a limit to speed

- This is a problem in all computer systems as they scale up
- Distance is related to size (the bigger we ma the resource, the more we move them further away)

Caching Overcomes Latency (a form of buffering)

- Use the local memory in the CPU to hold value that ultimately will be stored in memory



- Data in a cache has the same performance as registers
- Writing to cache also has the performance of registers

How Reading Through A Cache Works

- An address is generated by the CPU to read (load instruction)
- Initially, cache is empty
- If address being read is NOT in the cache, a read is generated to memory (this is what we refer to as a "cache miss")
- If the address is in memory, it is returned immediately (a cache hit)

(If we have a high hit rate, then the average read time is going to be low)

Why do Caches Work Well?

- Temporal ("time") locality: a memory location that has been recently accessed is likely to be accessed again soon.

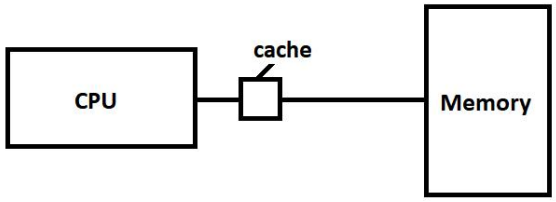
- Spatial locality: a memory location near a location that has been accessed recently is likely to be accessed soon. So we load cache in what are called "lines", sequence in the address space.

What About When Cache Gets Full?

- For values being read, the cache has to choose a line to "evict"
- This cache management policy (like scheduling) has to predict the future
    - Cache management must be compact and fast

What About Writes?

- MMU tables are only written by the kernel, the cache is flushed
- Memory system do write into cache



Policies: Write through (we don't have to wait)
Write back (holds value in case)

These don't occur in caches of MMU tables, because we don't write into the table as a process is running.

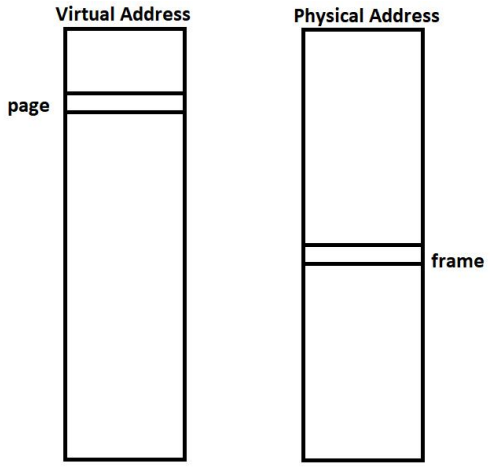
Paged Virtual Memory

- Simple VM had two MMU registers:
    - base address
    - limit
- Segment table can fit in local memory (what we use to implement cache) which is an **array of registers**
    - adds more flexibility
    - independent management of segments (makes some read-only, some write)
    - using segmentation faults to implement memory services (e.g. stack overflow as allocation)
    - BUT variable size to keep segment table small
    - thus, variable size requires memory allocation within the kernel

Fixed Size + Small Pages

- A fixed size region of the virtual address space
    - all pages have the same size
    - each page is identified by a page # which is part of the virtual address

    - each page is managed individually through a table (we call it a page table)
        - either specifies a frame #
        - or that the page is invalid

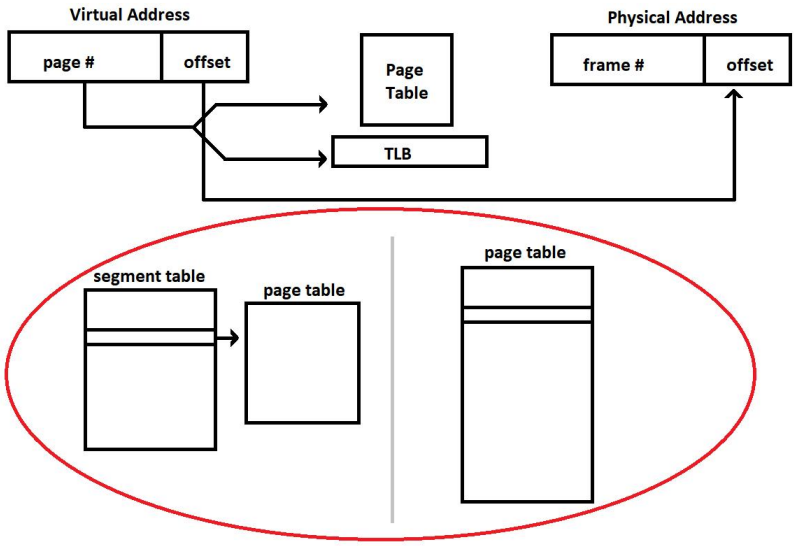


Memory frame vs. Virtual address page

A page table mas the page # to the frame #

Page table implementations: different lookup times, space taken, efficient, but all implement same mapping of page # to frame #

Mapping VA to PA



The TLB

One approach) Put cache between virtual address and the page table, and do a lookup. BUT **we don't do that**, because the translation of virtual address would be SLOW.

What we do is) We do a lookup in the cache of the page table entry. In parallel (simultaneously), the MMU looks up page # in page table entry cache and if found, aborts access to page table. If miss, page table lookup has already been started.

We call this the Translation Lookaside Buffer (TLB). (Lookaside because happens instantaneously, or in parallel with the page table lookup)

Management of Free Memory

- No variable sized slabs of memory
- The free list of frames is uniform
- No fragmentation of coalescing

Storing the Page Table

- It's too big to keep in local memory or registers within MMU
- We have to store the page table in memory because of this (but problematic)
    - Every translation of a virtual address requires an access to the table
    - Register-to-register instructions operate in one cycle (for speed & efficacy)

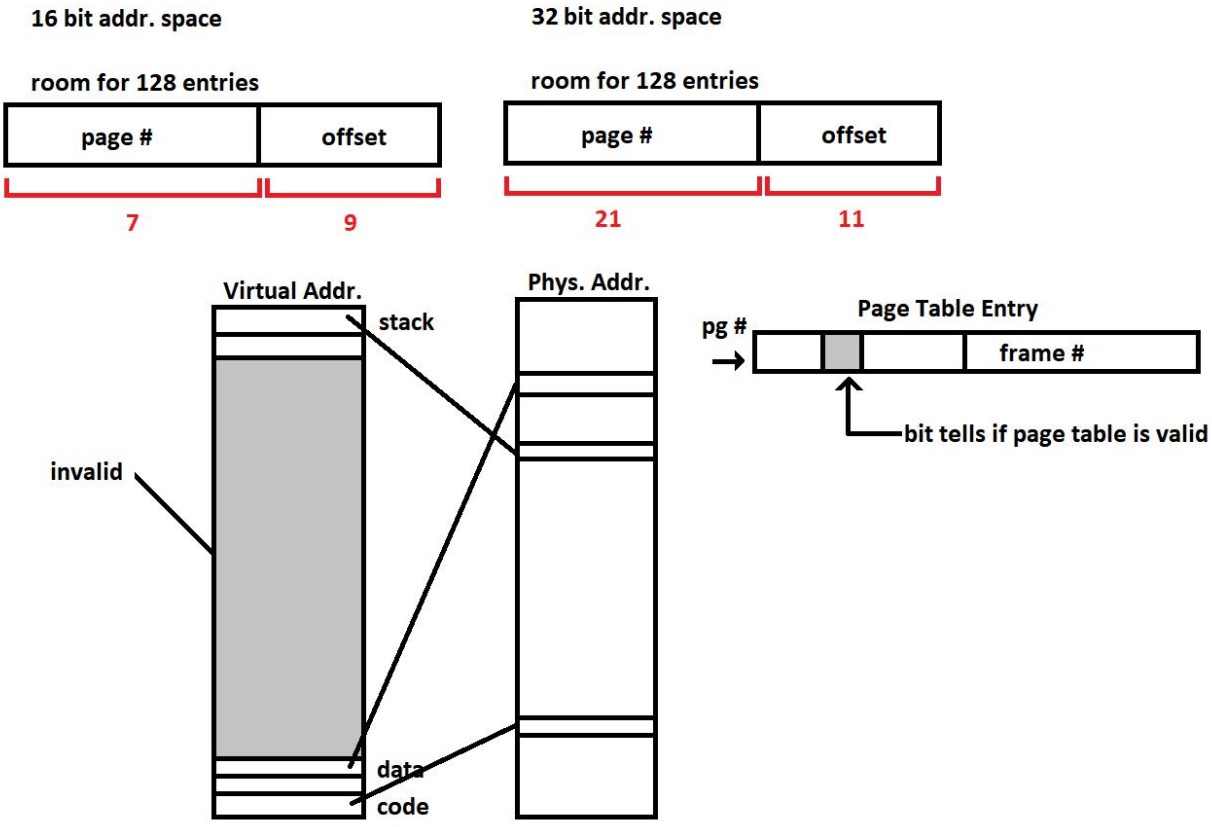
(and we hope to have high hit rate on average)

We can put in instruction and data cache (read) to do fetch and read if we have a cache hit.
(Multiported cache: handle read and write in one instruction)

Plus page table entry reads for every address (cache for page table entry)

Direct (Linear) Page Table

- An array of page table entries stored in the OS's physical memory
- Indexed by page #, with an entry for every page in the virtual address
- 32 bit + 64 bit virtual address direct page tables get very large
  (thus, we need more complex data struct instead of array)

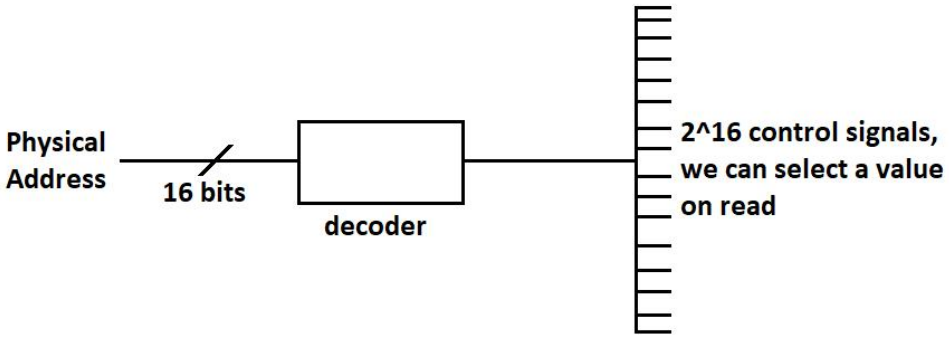


Translation Lookaside Buffer (TLB)

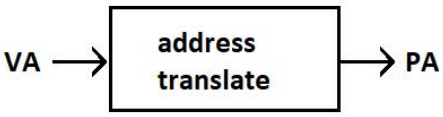
- The TLB is a special-purpose (dedicated) read cache for a page table
- Lookup in the TLB is initiated **at the same time** as address translation
- On TLB hit, address translation is aborted
    - A "speculative" operation is initiated before it's known whether it is required (or even valid)
    - Must be possible for any "speculative" operation to undo (abort or undo is required)
        (in TLD, we just care about abort)

What is a (virtual) address?

- A physical address is decoded into a control signal



- A virtual address is the input to address translation



- So far we use virtual address page number as an index into the page table which leads to direct lookup through memory

(A larger virtual address makes the page table large)
(Speed becomes an issue, so cache (locality) would be needed/optimized)

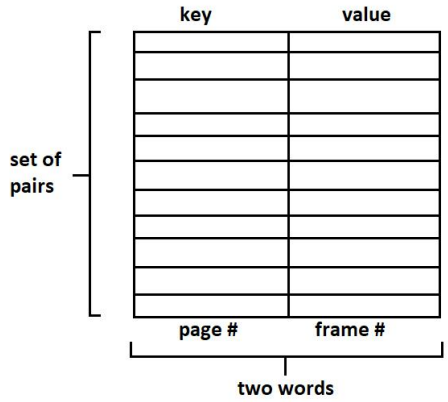
Using A Page Number As A "Key" (Instead of an address)

- Due to temporal locality, we can store a smaller set of page table entry
    - Pages used recently are likely to be accessed again soon
- We can store a small set of keys in a small, fast memory local to the processor (the MMU)
- We can consider the TLB to be a hash table, or a set of (key, value) pairs => **(key: page #, val: frame #)**

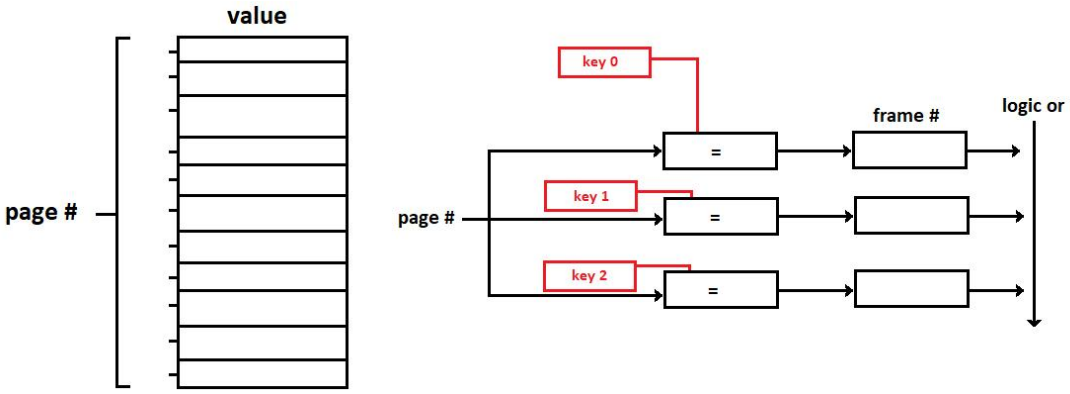
(linear search is slow)
(binary -> logarithmic if sorted) -- multiple cycles
BUT HOW DO YOU MAKE THE LOOKUP FAST (IN ONE CYCLE)?

Content Addressable Memory

- table of keys and values (like a database relation)



- How to implement a lookup by unique key in **one cycle**?
    (values would have to go in ordinary memory and we still have control signals)
    (BUT instead of decoder, we implement a comparator for every entry in the table)



SPEND HARDWARE for SPEED

- expensive, but we have room in the process

Page # fed as input to every comparator and at least one of them will generate a hit; or if all or 0, we generate a miss

Another example:
Translating IP address to a route on a backbone router

What happens on a TLB miss

- Page # cannot be found in the TLB

Approaches:

1) Hardware handling: complex hardware logic keeps control of processor + MMU and does a table lookup in memory (fine for direct mapped table) so we don't abort address translation

2) Software handling: on a miss, raise an exception and enter the OS, handle lookup, and update of TLB in OS software
   * flexible - we can change implementation of page table (no longer fixed) to hardware (may be a bit slower, but hope for high hit rate to pay the cost of a miss)
   * less logic in the processor (because some of the logic is moved into the software)
     Miniaturization of logic made this more possible since historically there weren't a lot of hardware (led to complex architecture)

CISC - 1970s

RISC - 1980s --> reducing logic was good

Less logic makes processor simpler and means more real estate can be put into memory

On a TLB miss a new key-value pair must be loaded

- Where do we load it? To an empty slot in the TLB
  - if there is no empty slot in TLB, we have to overwrite an existing entry

Policies:

1) We can overwrite the least recent used entry
2) We can choose a slot at random

Page Table + Address Translation

- Page table is mapping from virtual address to physical address
- Issues with direct (linear) page tables:
  - time to access (because we read of every address translation; 1 read per translation)
  - size (the table is too big)
- We address time through TLB
  - a high cache rate (in the high 90%, in other words near 100%)
    - this makes miss time much less important
  - a TLB hit reduces access time to 0
- TLB hit rates depends on temporal locality in the TLB
  (what was accessed recently will be accessed again soon with high probability)

Not every program exhibits this.

(if program does poorly with TLB hit rate, you're out of luck)

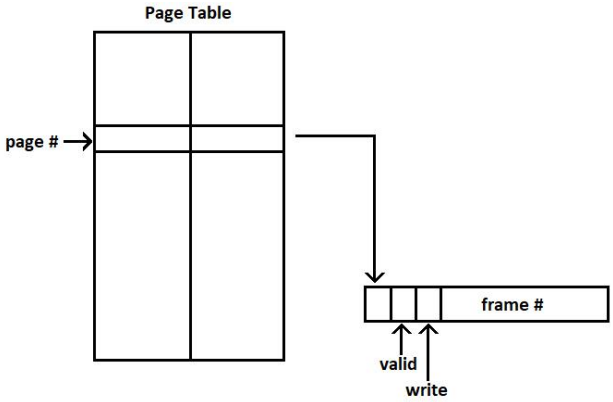
But you can buy memory (increase cache)

With TLB, you can't increase; the TLB is fixed

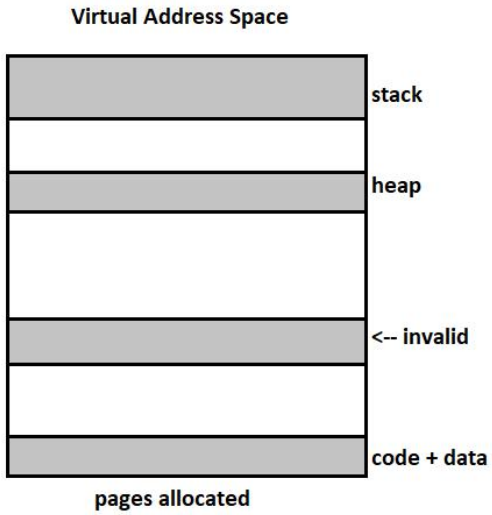
- Now we can address **size** through spend time

How to Implement a Small Sparse Page Table (a data structure)

1. An array is a dense table
   - every possible index value has a memory location statically allocated
   - if lots of entries/indices are not used then this data structure is inefficient
   - in a page table, what is an unused entry/index?
     - indices are page #'s
     - indices are unused when we have invalid pages



If invalid (i.e. the valid bit is 0), then frame # is meaningless, there is no physical frame



How to Deal with Invalid Pages?

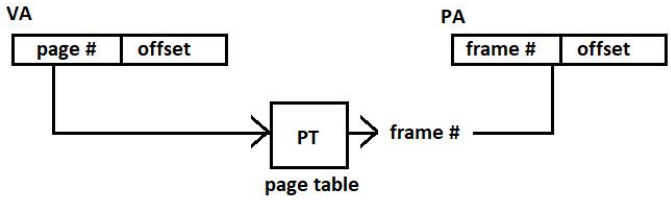
- If a single page is invalid BUT the pages around it are valid, then we need the entry to be able to check the "valid" bit
    * in this case, the page table entry is NOT "unused"
- If many contiguous pages in the virtual address are invalid, then we can optimize to not store their entries.

(to do this, we don't use a direct page table, but a multilevel page table)

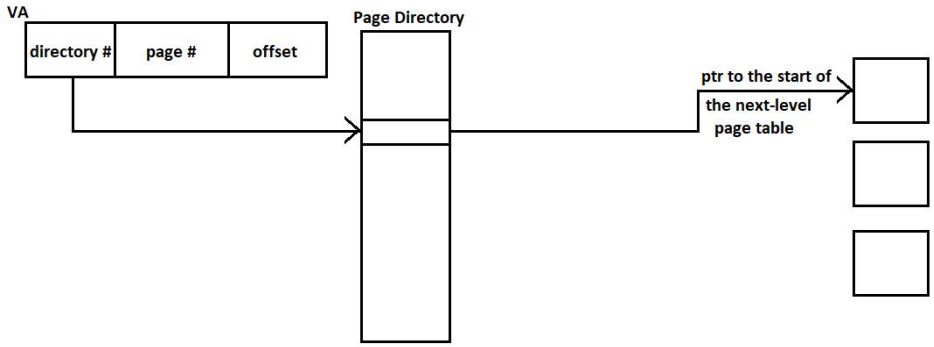
Multilevel Page Table

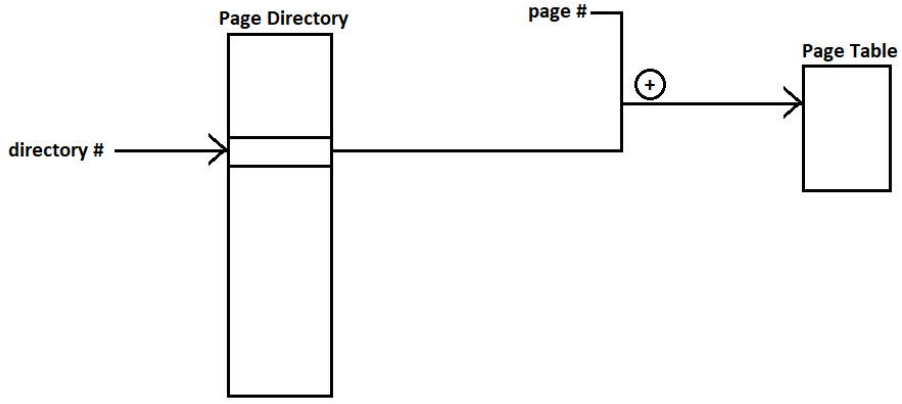
- It's really a tree data structure

Every virtual address has two fields



The frame # and offset together make up the physical address





The sum gives us the index to the page table we want to use

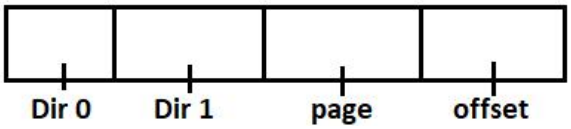
So the page # is the offset into the page table

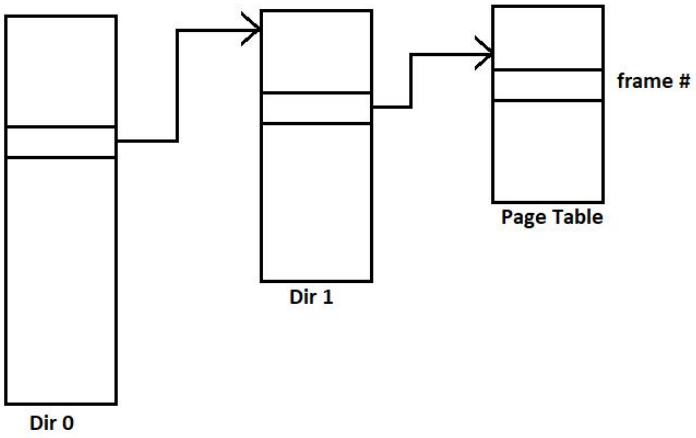
Every table is still a fixed size, but we don't have to implement a new table because the directory table itself can just have valid bit be 0/1 (0 if not valid page table) so then we save space

As we move from 32-bit to 64-bit, we need to manage space more efficiently

3 Level Page Tables

- Virtual address is divided into 4 parts





CISC vs RISC: In CISC implementation,
This is fixed in MMU hardware

First field offset into dir 0
Second field offset into dir 1
Page # is offset into page table (takes 3 reads / memory accesses)

[3 memory accesses on every TLB miss (so we need to rely on low miss rate)]

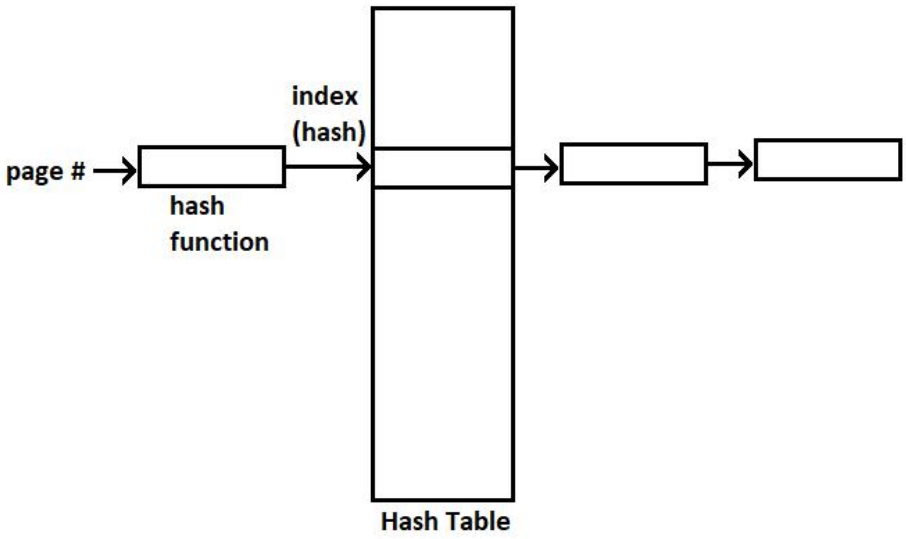
- Small table (sparse) but more costly
- A lot of processor cycle spent doing a TLB miss

Handling TLB Misses in Software (OS)

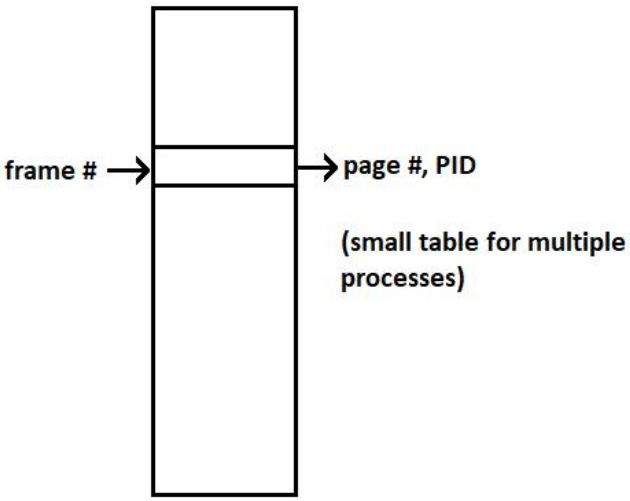
- Instead of having MMU do the work to find the frame # in the page table,
    * simply enter the OS on TLB miss
    * then let software do the lookup (REALLY EXPENSIVE)
    * but it is now under **OS** control
        - **adds flexibility** so then OS can use whichever data structure it wants to do the lookup (direct table, two-level, three-level table, etc.)

Exotic Page Table Implementations

- Hashed page table



- Inverted page table



We can take the time to handle a TLB miss in software because it occurs RARELY!

Paged Virtual Memory

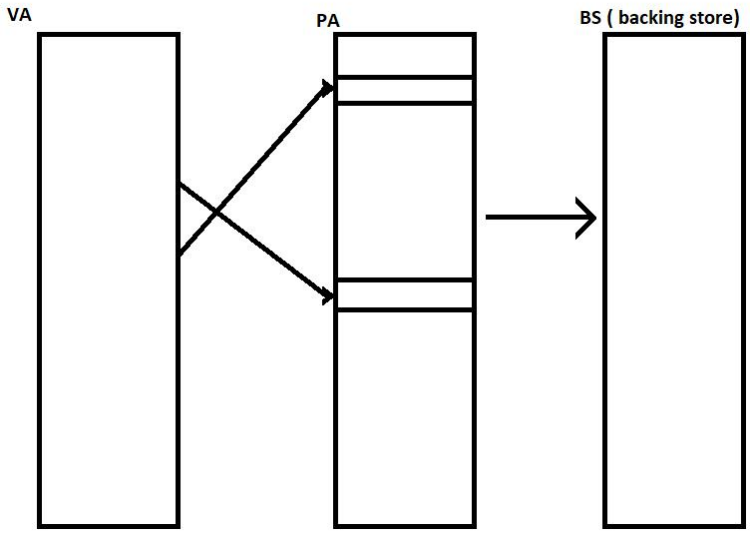
- Start by looking at a **simpler** mechanism which does **not** work on a per-page basis by whole processes or segments.

- Issue: Lots of processes can fill up physical memory.
    * virtual memory translation allows relocation within physical address space
    * it doesn't create more physical memory space

- Multitasking means some processes are not even RUNNABLE/READY (e.g. I/O)

Blocked Processes

- I/O is slow + human interaction can wait indefinitely
- This can result in many processes not READY/RUNNABLE
- Running out of physical memory causes memory allocation and process creation (fork) to fail.
    - Brings Linux to its knees (e.g. fork bomb)
- How to free up memory in processes that are not RUBBABLE
    - move contents of physical memory to secondary storage (disk/SSD) (secondary meaning slow)
    - keep track of where it is



"backing store" -> space on secondary storage

Can be managed in a simple way

Process Swapping

- Processes not READY can be written out to disk
    - their physical memory reused
- When do we swap a process? [Policy-level]
    - immediate swapping
    - triggered by low memory (called "memory pressure")
    - dependent on why a process is wait (character devices only [human interaction])
    - time spent waiting

    <Problem with swapping: Write to disk on hundreds of cycles and process may resolve quickly>
- When a computer system spends all its effort implementing inappropriate policy and cannot make progress it is called thrashing

Housekeeping - moving things around to keep them tidy and things readily available; happens in the background

A process is pinned if it cannot move to secondary memory to achieve low latency

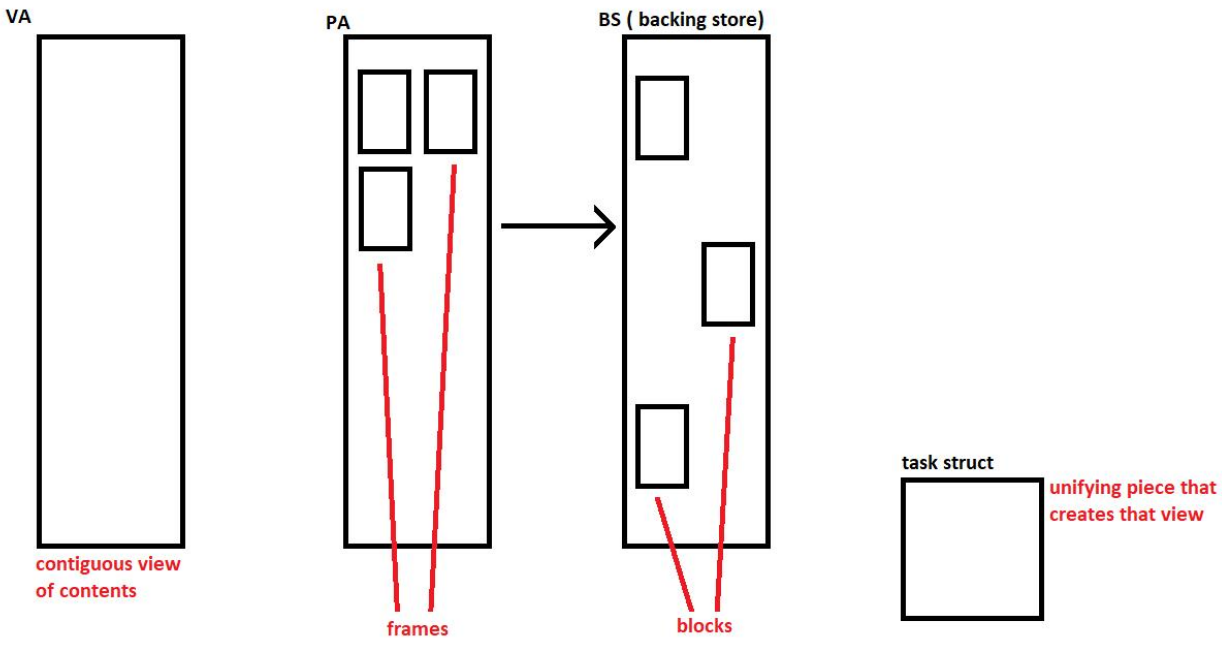
Issues with Process Swapping

1. As processes get large in virtual address space, even the RUNNABLE processes may not fit in memory (32 bit = 4G)
2. With large address spaces, swapping is very slow.
3. How about page swapping?
    - We use invalid bit in the page table for a page that has been swapped out.
    - **Page fault** enters OS and OS can check for swapped page
    - Need to keep track of page location on disk

"Memory scheduling" - What's in physical memory frames and secondary storage (affects speed)
The more page fault, the more overhead, slower

What is a process?

1. A contiguous area of physical memory + the processor state (von Neumann model)
2. Virtual memory mapped to different parts of physical memory
3. Paged virtual memory:
    i. von Neumann model with some pages stored on disk (secondary storage)
    ii. Or a process is a set of physical memory frames and block on disk, using virtual memory to **simulate** von Neumann machine



Paged Virtual Memory

1. Contents of page that make up a process image can reside **EITHER** in RAM or secondary storage/(disk).
    - page table specifies translation for pages resident in memory frames
    - invalid pages may be stored on disk or as OS data struct specifies
    - PT + TLB enables fast fetch/execution as long as all pages accessed are resident
      (this is where we get CPU performance [billions per second])
        - TLB misses slow down the fetch/execution cycle
        - Non-resident pages are very slow to handle

Accessing Non-resident Pages

- starts with page fault (invalid PT entry)
- Enter OS kernel, identify the fault as non-resident access
    - OS data struct (note: this is not the page table) lists all process page
      Resident (the page table) + Non-resident
- we need to make the page resident + restart execution
  --> we need an empty frame
    - worst case: we need to evict a page
        1. First choose a page to evict
        2. Write it to disk
        3. Read from disk into empty frame
    - best case: we have frame available, so skip steps 1 and 2 and just handle 3

Page Fault Handling

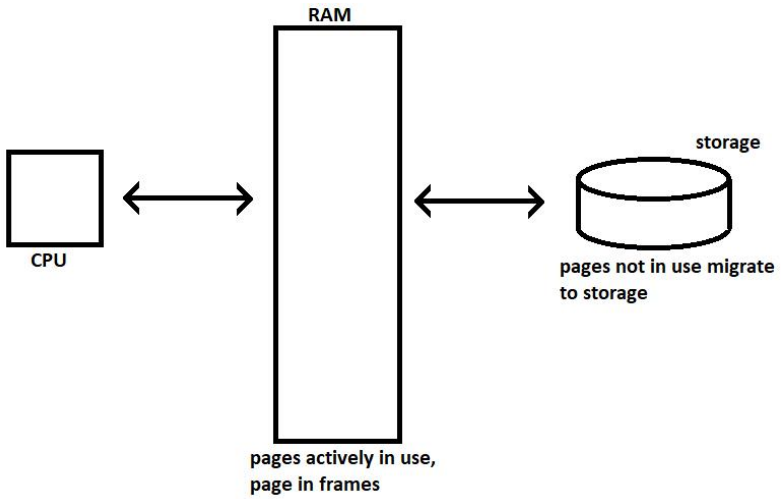
- Restart means setting PC (program counter) appropriately and do a "return from TRAP", loading hardware context + running
- If another READY process runs during page fault handling, process may go on the READY queue

Cause of paging activity: pages being accessed are larger than physical memory

Working set of pages: all page being actively accessed at a point in time

Page Virtual Memory is a Cache

1. Code + data memory caching
2. TLB to cache page table
3. Frames cache pages stored on disk



4. We need an eviction policy
   - We had CPU scheduling
     TLB eviction policy
   - and now we'll have page eviction policy

Which Frame to Evict?

1. Least used --> we have to keep a count
2. Least recently used - temporal locality
   (page transfer gives spatial locality)
   - We have to keep track of when pages are used
   - Page faults only occur on "misses" to page cache
   - We have t keep track even on PT hits
   - How to do LRU fast + cheap (to optimal)? You can't.
   - But we can approximate it

See textbook for more info for below section

AMAT (Average Memory Access Time) = Miss Time + (Probablility of miss * disk time)

Take a trace to know future run to implement an optimal policy for cache replacement

Simple policy using FIFO replacement (implements a queue). All pages have some lifetime and we evict the oldest page frame.
- But faces Belady's anomaly, as we increase size of cache, or increase page frames, we increase the probability of a page fault

Simple policy using random page replacement
- Doesn't face anomaly, so better than FIFO
- choose page at random to evict

Using history (**Least-Recently-Used (LRU)** Policy)
- Keep track of things happening on hit
- Must be fast, in one cycle

Least-frequently used

Information as field to keep in MMU to figure out the most-frequently used to pick out the least
- Each page needs to know each time there was use of a page
    - Increment counter during miss or hit during address translation in the TLB
- Find on a miss the frequently used and kick it out

(If you don't use page for a long time, it's unlikely it'll be used again soon)

Least-recently used says which page has gone the longest

ISSUE: How do you track it?
- We need a timestamp on EACH page (larger, so more expensive) instead of implementing a counter

In case of locality, optimal is better (predicts pattern).
LRU is the second best; does better with hit rate as cache size gets bigger

Loop-sequential can have both temporal and spatial locality in play
- If size of cache not large enough, more eviction at end of loop

LRU would be poorly
FIFO would do WORSE (page gets pushed out before getting used again in loop)
Random does better for loop-sequential

Measuring Least-recently used

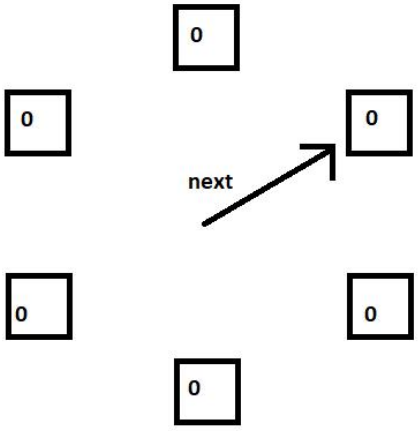
Keep pages ordered (queue). When used, put at back of queue. Otherwise, evict front of queue.

INSTEAD, we could

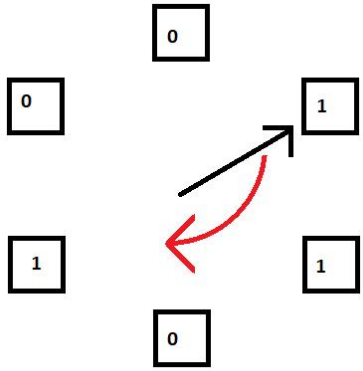
Use counting to measure least-recently used (clock algorithm) -- note **APPROXIMATION**

Approximating LRU: Clock Algorithm (or second chance algorithm)

1. Requires one bit for every page
2. The reference bit is set by the MMU whenever a page is used



Any page that gets referenced, we set the bit to 1.



Go around, skipping 1's, and look for the first instance of zero. Then evict the first unreferenced page.

On a miss, we scan each 1 (in a clock pattern) and change 1 to 0 until we scan a 0. The 0 we scan gets flipped to a 1 (so we ensure that wecan always find a page to evict)

80-20 workload - based on how much locality on a workload

With clock, does almost as well as true LRU algorithm

FIFO and rad do significantly worse

Write into cache takes longer if we have to evict

- We can evict clean pages before dirty pages, and keep dirty pages for longer, reducing chance of a miss

Applications of Virtual Memory

1. (User error) Page fault originally for exception-handling: catching errors in memory access
2. (OS Policy for Page Swapping) Paged virtual memory : catch a fault and potentially trigger page swapping activity
   (catch + handle access to non-resident pages)
3. Improving performance of process creation
   - fork() system call allocates a task struct
   - causes replication of address space
     - sharing code between parent + child
     - allocating and copying data, stack, heap (writable data areas)

PERFORMANCE ISSUES ==> (we don't have to copy code, because that's static (read-only), thus shared)
(because process gets bigger when we make copy)

Performance in Fork()

1. Memory allocation which can cause page swapping
2. Often fork() is followed by exec()
   - exec() reads code + data from a file
   - old contents (parent page contents) are discarded -> (it's a waste to copy then discard)
3. Use virtual memory to resolve
   - avoid copying data *en mass* (for the entire process) (also avoid allocating en mass)
   - so instead of *en mass*, draw it out on demand
   - never unnecessarily allocate memory

Copy-On-Write

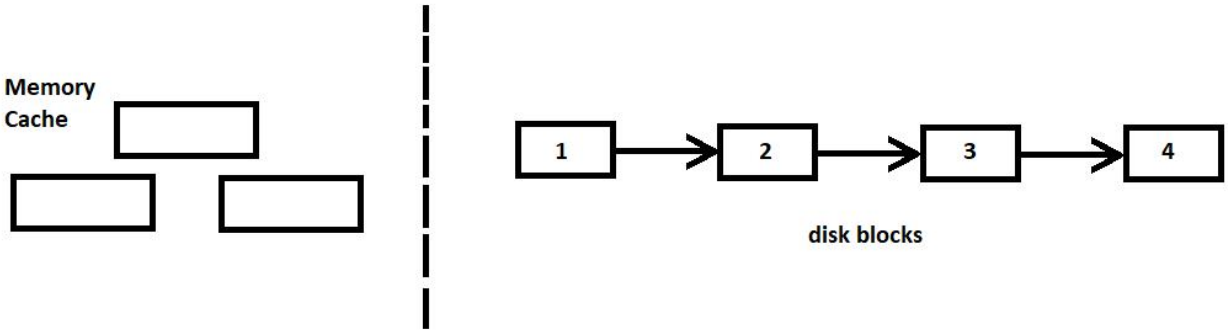
1. Initially on fork() we create a virtual copy of data
   - share writable data pages between parent and child
   - read protect the virtually copied pages (need a write protection bit in PIE)
   - a write to a shared page (protected) causes a page fault
2. On-write to a protected page
   (this is what we mean by drawing out on demand -- we guarantee amortized cost)
   - allocate a new page
   - copy contents
   - remove protection "Lazy copy"
   - restart execution

Other uses for Page Faults

- Filling memory regions with zeros
  - an important security feature
  - memory allocated by kernel but not filled with values
    - e.g. new heap or stack pages
  - the problem with not filling w/ zeros: leakage between user processes
- Freeing memory + filling w/ zero is slow potentially
- Instead memory protect region of virtual address space and fill with zeros on demand

Memory Mapped Files + Objects

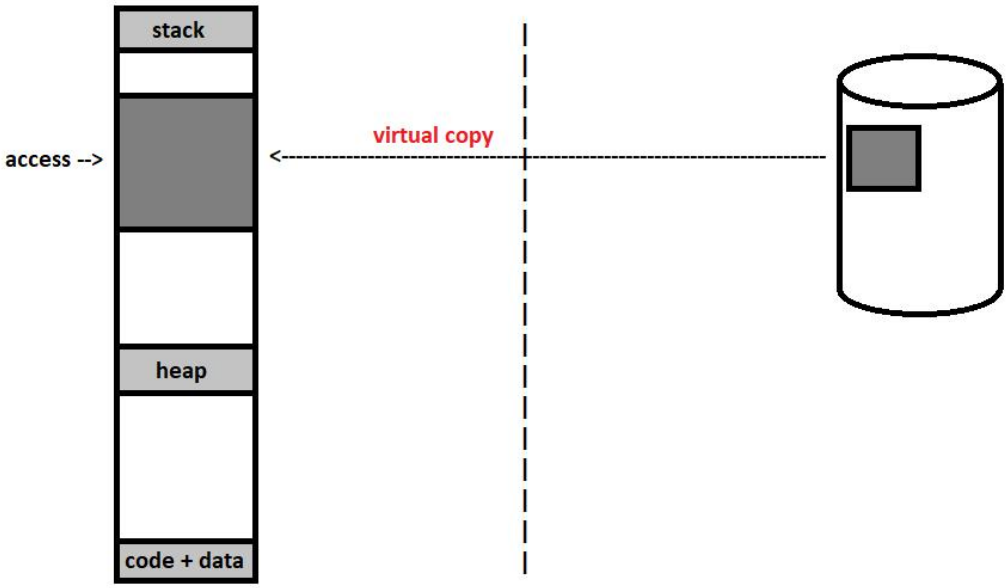
1. File access using open/read/write is sequential
    - seek is possible (moving pointer)
    - between seeks access is linear and the file system optimizes disk activity on that basis.
        (in managing file blocks in cache)
        (spatial local occurs in the file system as well)



    - non-linear access to a file can be inefficient

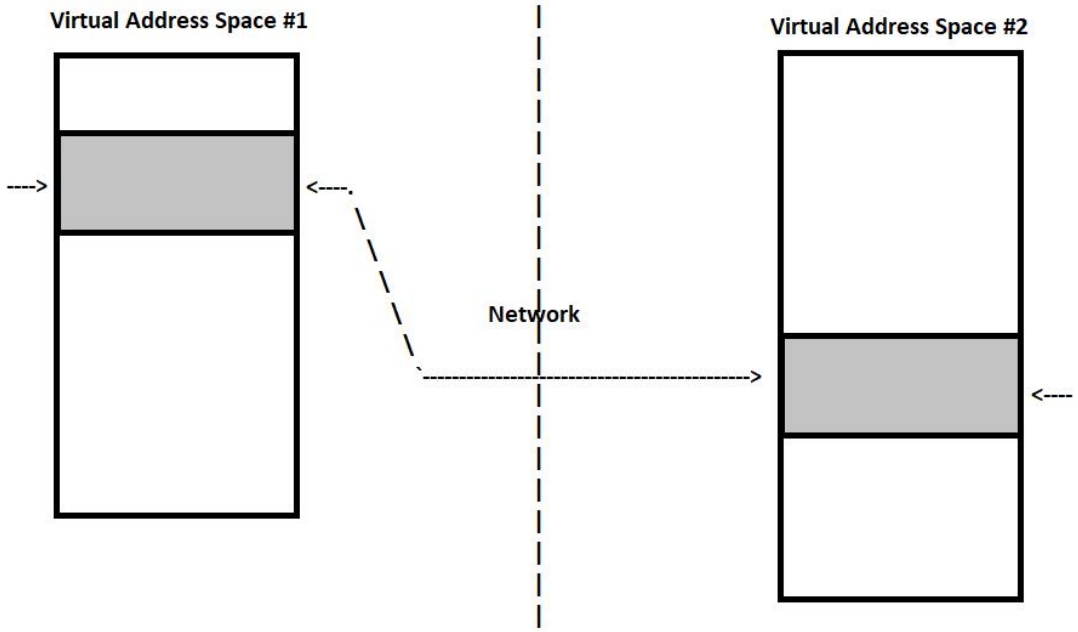
How to Insert a File Into the Virtual Address Space of a Process

1. Use mmap() system call
2. Specify a file (usuall small) that is to be copied (virtually) into an unused area of the virtual address space
    - invalidate area of virtual address space
    - accesses cause page fault
    - reads + writes act on cached blocks
    - non-sequential



Into data space, works just like copy-on-write

Distributed Shared Memory



- **invalidate** pages
- catch access as **faults**
- communicate between processes to move data and replicate between processors

Threads

1. What is a process?
    - threads can be called "lightweight processes"
2. Usual answer: a process is the state + activity **describe** by the task data struct.
    - in modern OSs, a thread is usually also represented by a task struct
3. Historically OSs didn't support threads
    - but developers wanted to use threads

History: Threads Implemented with User Process

- Using resources + functions allowed to a process by the OS (no use of kernel mode)
- Threads are implemented in a library using subroutine calls like in the C library
- Threads are used to represent multiple locations of execution within a process
    - program counter determines location to fetch

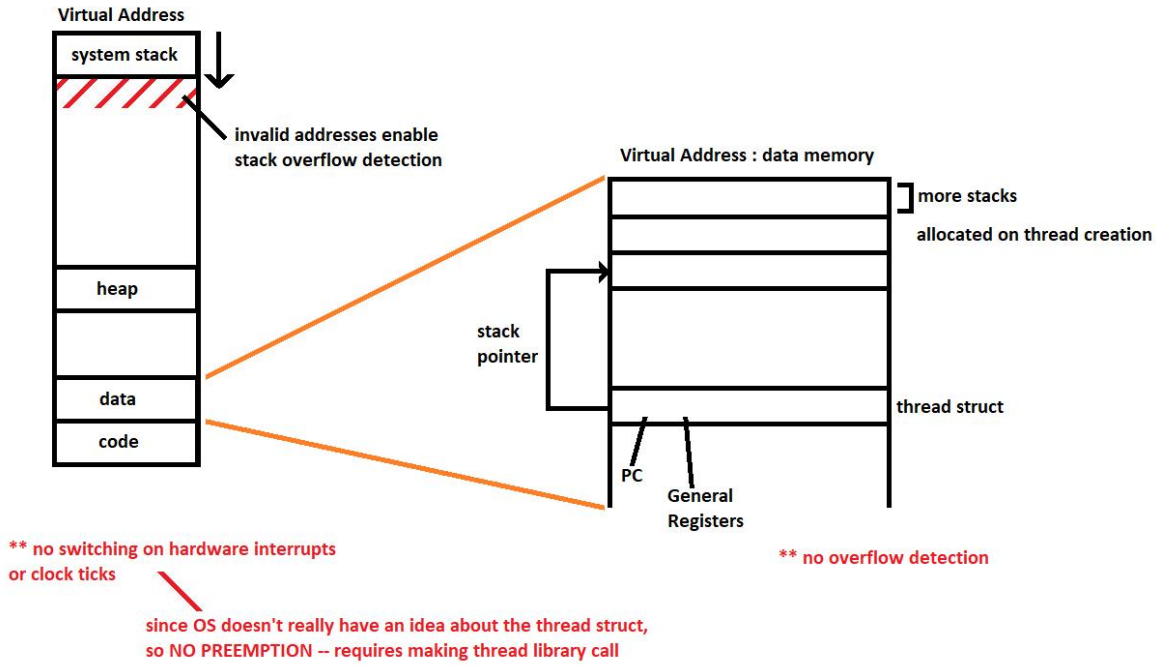
Keeping + Managing Multiple PCs

- subrouting calls store PC (does not go in task data struct during subrouting call)
    --> goes on the **stack** (in a single instruction), loads new PC
    - location of control is program counter and contents of the stack
- What if each PC had a private stack associated with it?
    - each program counter + stack has its own history of control and so a different future
- A PC value + a stack defines a "thread of execution" (since it has past & future)

Thread Switching

- Store current program counter + stack pointer
- Load store program counter + stack pointer
- Must do it safely
    - Switch stack pointer } -- do not use stack pointer in between these instructions
    - Then load new PC }
    - We can do it in ASM (assembler)
    1. This can be do in user mode
    2. Can also use task struct **BUT** then must have OS support if we do it that way

Multiple Threads within a Virtual Address Space



Why Use Thread Programming

1. (Today) It's the only way provided for **parallel** execution within a **single virtual address** space using a **multicore processor**!

2. **Historically** processors were single core
3. Storing the history of a thread execution on the **stack is convenient**
    (stack holds a lot of information about context)
    (co-routines - interleave execution on single data struct.)

Applications for Threads

- Systems that must **react quickly** to input while also performing longer-running computations
    - human-computer interaction
    - early example: character input echoing
    - also: **network traffic**

- Embedded systems + interaction
- Standard thread library as part of POSIX: pthreads
- then OS support for pthreads developed (becoming standard, de facto)
    - if Linux kernel can support pthreads

Virtualization Emphasizes Isolation + Protection

1. Isolation between address space
2. Protection of system resource from user level abuse/misuse

Goal: sharing! (sounds unintuitive)

"Good fences make good neighbors"

\ \

Isolation/protection sharing

Early: physical machine shared by simulating an isolated machine

Problem: wasteful, limiting, inconvenient / low performance

We Share When We Can (When it's Safe)

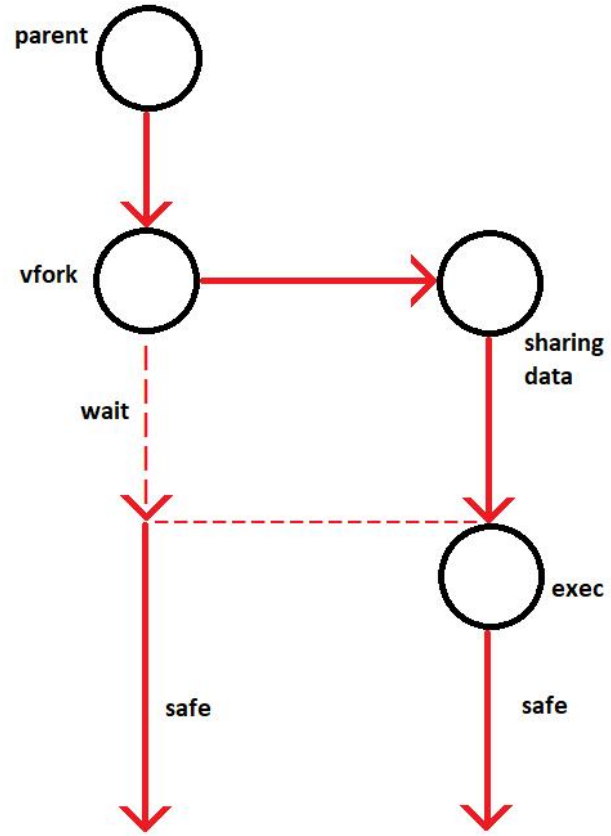
1. Share memory between address spaces
    - e.g. sharing code (not written to): read only
2. vfork() is a mechanism (prior to COW) to speed up fork() through radical + **unsafe** sharing
    - sharing data address space (writable) between processes
        (shared by parent + child)
    - Used only when child will call exec() --> exec escapes unsafe sharing
        almost immediately (very common)
    - After call to exec() conflict is gone.

We don't use memory protection (COW) in vfork, so how do we keep from conflict?

Has to do with the scheduling of processes -- we block the parent (make the parent wait)

Threads are unsafe as vfork(), but are more general

Before COW



OS Support for Threads

1. Threads allow multiple points of control + histories
2. Also enable very intimate sharing between RUNNING or READY processes (unlike vfork)
3. Stacks are separated by simply assigning one stack per thread.
4. What about global variables + the heap
    - We do sometimes separate global variables
    - We could maintain multiple heaps
    - But then communication would be complex (messages)

In Linux An OS Thread is a Lightweight Process --- hardware context + stack

1. Every thread has a task struct
2. Every thread has a PID
3. Threads show up in "ps" (process status) results

    BUT

    Threads created by (within) a process share the code + data segments of the VA space and the heap BUT **not the stack!** (we get memory protection)

    - Local variables + subroutine calls OK (safe) because of private stack

Thread Creation

1. Not through fork()
2. Parent process has a lot of control over thread creation
3. pthreads (POSIX threads) is the standard interface (but not the only one) for thread creation
    - library level that we standardize thread

|| Threads complicate policy.
|| Policies to resource allocation + exception handling

Safe execution through concurrency control

Thread Creation

Thread data structure, the *thread, is passed in to pthread_create and managed by the library. We pass in subroutine pointer *(*start_routine) (code address that's entry point to subroutine) as well which is invoked by library. Also pass in arguments (or NULL); different from fork (pass nothing to the child from parent; child is literal copy of parent).

Threads are created to do a particular task. How does the child know what task to do?
Recall with fork, the child knows it's the child by the PID value (child gets return value 0 from fork()).
How does it know what to do? Look into the data segment; it has it's own image or copy of the data. Can look at what's going on, the state, etc. Parent has its copy of its data and so does child, so no conflict.

Only private space for thread for child is the stack. The stack is empty except for the arg, which is local (not shared with anyone). So the parent can pass void* arg to the child; child has some piece of information (e.g. a simple integer).

If you have a bunch of threads and there's no argument, their symmetric and you can't distinguish them (like their unique identity).

When pthread_create returns, parent is running and child is running as well.

Thread Completion

Equivalent of wait for fork(). pthread_join()
Pass in thread and return value from pthread_create()
    Wait from fork() had to return value except exit value

Concurrent Execution

1. Applies preemptive multitasking and actual parallel execution.
    During develop, first pertained to preemptive multitasking
    When multicore processors, the parallel computation
    - In the von Neumann model, memory locations change only through instruction execution.
        - Some register can change implicitly (e.g. PC, SP), but still through instruction exec.
    - I/O can be an exception (not part of von Neumann model) to this, but not visible in process execution model

- Can have value to register during I/O
- This is why compilers can reason about the contents of memory and can change/optimize
** concurrency can interact badly with compiler optimization

Preemptive Thread Execution

- Switching between threads can occur after **any** instruction!
- If another thread runs (from PoV of one thread), it can change any location in the shared data memory
  - The stability of memory values in the von Neumann model is violated
  - This is also true if two threads run in parallel
  - Two threads interleaving or running in parallel running the same instruction (we think of that as an error since we don't know the outcome)
  (If two instructions ran at same time, what happens? Depends on memory architecture, but generally one of them will win. But we should avoid it: race condition)
  (Race condition: within interleaving or running, memory gets overwritten while two instructions being run)

A C statement can be interrupted

- A statement is compiled into a sequence of instructions
- Some operations are single instructions
  - E.g. addition, subtraction
  - "C is shorthand for assembler" - treating a C operation as if it were an instruction
  - at least at low optimization levels (GCC can be aggressive to the code compilation)

- x = y + z might be (determines on architecture) compiled into a single instruction if x, y, and z are scalar variables (can be put into register).
- x = y (similar)

- x += 1 is likely not to be
- x++

In all of these cases, you should not assume any of these are a single instruction, even if you know something about the architecture.

How Can We Program Threads Safely?

1. What is private to a thread?
   - register context: PC, SP, general registers
   - on a single core, these are saved and restored explicitly on thread switch (thread library does this)
   - on multicore processors, each core has its own hardware context (its own registers)

2. Using a thread id, the data in memory can be partitioned (divide things up, so threads stay out of each other ways, don't touch shared memory locations)
3. Also the stack, which is accessed the the stack pointer (local variables)

How to Access Shared Memory Locations Safely?

1. Let multiple threads (or processes) read and write memory locations within the kernel
    - Then they're accessed through system calls }
    - Write puts data in memory } - this is a pipe
    - Read to get data out }

    Problem with using pipe is that we have to use system call, so inefficient expensive.
    Also inconvenient, having to do these steps.
2. Locking

Locks

- A lock is a way to protect a memory location within an address space from concurrent access
- How? We associate a lock with a memory resource (one location or larger).
- A lock when created is initially an "unlocked" state.
- A thread obtains the lock through a "lock" operation
- Only one thread at a time can hold a lock
    - Lock when we want to access, unlock when we are done
    - Allows threads to run safely

- Locks enable you to control concurrency in threads. High-level, part of pthreads.
- Condition variables (part of pthreads), used to implement semaphores and reader-writer locks
    - Complex to use, not very intuitive
- "Semaphores" generalization of locks.

Thread-safe programming

- The way we reason (think through) about programs depends completely on stability of memory value (they don't change on their own; only through program actions)
- We reason locally about a single thread of execution
- Threads enable the violation of these assumptions.
    - The point of danger is in any writable shared resource (in particular, memory locations)
- Partition resources or control concurrency
    - enforce mutual exclusion (one access of shared resource at a time)

Locking for thread access to "shared variable"

```
balance = balance + 1;      // if variable in memory (rather than register), then multiple instructions
                            // can't assume safety
```

What if compiler puts balance in register? // it's not shared; every register copy can be different

```
volatile int balance;       // tells we'll need it in memory
                            // memory location could be part of the I/O structure
                            //      (writing to it my cause an action)
                            // volatile tells it not to hold it in a general register
```

There's also the "register" keyword which is opposite to volatile. Higher-level languages don't really let you do this.

```
lock_t mutex;              // globally-allocated, visible through program
…                          // mutex is data structure (we have to initialize it so it's at an unlock state)
lock(&mutex);              // simply call lock (if in unlocked state, lock will be acquired)
                           // if someone else holds it then they have access to shared variable,
                           //       so then cause the thread of execution to stop until the lock can
                           //       be acquired (this is a point of synchronization)
balance = balance + 1;     // critical section, the part that needs to be locked
unlock(&mutex);            // release the lock
```

If you had complicated piece of code and unlock was never called.

- Anyone else who wanted to get that lock will have to wait forever;
  failing to unlock

* Note: Circularity is deadlock (not about holding a lock forever), but process stops anyway.

What about calling unlock twice?
- Depends on implementation of locks
- pthreads should be able to detect that, but we shouldn't assume for all concurrency libraries

How do we implement locks?
- Depends on whether you have operating system support for the lock
  - Where does the concurrency come from?
  - Locks generally used for preemptive context switch
- Directly done using shared memory, because multi-core processing

Can you use simple reads and writes to implement a lock for two processes running in parallel? Or if it's hard to do that efficient, can you extend the memory in some way to make it possible to implement a lock?

- Mutex flag
- Loops "**spin locks**" through operation called **spin-wait**
- Can be done efficiently, if you have an operation called **TestAndSet** (single memory operation that does read followed by a write, uninterrupted). TestAndSet is dependent on the architecture.
  - Still have to do a **spin**. Do spin until we return 0 (0 meaning we've freed the lock)

This approach of having two processors that directly uses shared memory (very technical kind of programming). Rare.

Other ways to implement lock:
- CompareAndSwap (similar to TestAndSet); way to implement spin lock

How does Linux implement a lock?
- Avoid spinning altogether
- Use **yield** – the lock itself is implemented by the kernel as a data structure within the kernel

if lock is held and you call lock, you wait (you don't spin or have a location in memory you're going to test and set). You put the process in a waiting state, and put it in a queue. Sleeping (waiting) instead of spinning.

Inefficient (overhead), we have to do a system call to enter the kernel. Having to check whether we need to sleep.

Some implementations may use combination of spinning and sleeping.

When we go over concurrency (don't go by the book)

Lock-Based Concurrent Data Structures -- Classic Concurrency Control
- Global counter, one variable which holds a value, which can be incremented/decremented at any point in the programming, and get it's value (in sequential program can be programmed without locks)
- In thread-based programs, it will not be safe. Non-safety is in update statements (c->value++) or (c->value--). Could be as many as three instructions (read from memory, update, write to memory).

If concurrent increment and decrement, there's no telling what the value will be.

- How do we fix this?
    - In counter type, not just include value, but also a synchronize variable called lock
    - Initialize lock to NULL (lock is free)
    - When want to increment/decrement, acquire the lock using lock()
    - We can safely increment/decrement

    - In some memory architectures, a read and write happening at same time can have unintended effect.
    - So lock even the reads! (Architecture!)

Book brings up important and subtle topic: Implementation of locks
- Occurs when you have lock of activity in the shared memory
- Performance issue (high overhead), suppressing concurrency whenever we're accessing variable
    - Constantly entering the kernel, very expensive all this locking and unlocking
    - We want to reduce the overhead; changing the data structure in some way or the way we lock it in such a way that is preserves the sequential behavior. With a counter, it's almost frankly impossible to do. Solution: Scalable Counting

    Scalable Counting: approximate the counting
    - Do not make every thread have counter, but every processor have a counter.
    - Locally, update a subtotal; periodically (after some threshold), update the subtotal to globally
    - Note: We're not locking across processors.
    - Global value will be approximate

    If we replace with approximate counters, will it still work? Is it still correct? *MAYBE, who knows?*
    - All depends on the correctness of the program (the logic) that depends on the value being exactly correct or approximately correct.
    - Hard to reason about a program, taking into account that certain value will be precise will instead be approximate.
        - Source of error. If you make inaccurate assumption about approximate counter, that it'll be closer to reality than you think it will.
    - If you're willing to settle from approximate counter, you will get good performance.

    In the book:
    ** The approximate counter they implement doesn't have decrement counter.
    - When to add up the subtotal into the total depends on the threshold. They only compare when the threshold is high enough. If decrement counter implemented, hard to check and reason about. Easier to reason about, if you only had increment.

    Implementation:
    - Array of local locks with number of processors.
    - cpu = number of threads modulo by number of CPUs

    - initialize globals to 0, and locks to free

    How we do an update
    - before, set lock for cpu
    - test if local[cpu] meets threshold value, after updating amount of local[cpu]
    - if reach threshold, lock global, take subtotal and add to global total, release lock of global
    - after, release lock for cpu

    Get
    - lock global, set variable to global value, release lock (note, we only get approximate)

Precise counter - time scales up for each thread
Approximate counter - time is almost 0

Better and better performance by letting threshold be off by a larger value (global counter updates less periodically), but less and less precise. (How much precision are you willing to give up for performance)

What about concurrent linked list?

- How do we make sure linked list doesn't become corrupt by mismanagement of pointer fields?
- Modifying head pointer and doing a lookup
    - Also needs to be sequentialized.
- Simple strategy: Locking the entire data structure, would make thread-safe
- Initialization of lock happens once on initialization of list
- When we error check or return success in the middle, don't forget to unlock:
    - If we don't unlock, we have a problem where no one else can do an operation of the shared data structure. Block in the lock forever.
- Optimization locking and unlocking within critical section; using break instead of return

Scaling Linked List:

- Is possible through hand-over-hand locking.
- Individual lock on every node (each of the next pointer), not just head node.
- Can do multiple lookups in parallel
- More complex data structure, but can improve performance

Review

Locks are straightforward user-level synchronization.
Counter - scalar variable

- Other than fact you can only do increments, you can do thresholding in both positive and negative, just a little more complicated.
- Counter doesn't have property that when program ends, you have the right value. The value that returns will be approximate which is in the global variable. Whatever is in the subtotal may never get into the total. Solution is to have a global get which says when you calculate this, don't just get the global variable, but go ahead and lock each of the subtotals and add them all up into the total. More expensive operation, no parallelism in any of the threads.

What about queues?

- Two pointers, head and tail, and also two locks for each of those pointers
- Dummy node used to represent null queue node
    - Initially both the head and tail
- Enqueue, lock the tail, instead next pointer to tmp, point tail to point to tmp
- Dequeue, at head, more pointer manip because dummy node. Release lock and return if queue is empty.

Hand-over-hand locking (fine grain), but we don't need. Worthwhile on linked list though because we modify in the middle.

Concurrent Hash Table

- Application of concurrent lists
- Buckets which are essentially sets
- List insert and list lookup to know if we found value in hash lookup
    - Implementation of list is itself concurrent. All of the locking and unlocking done within list
    - Hash table doesn't need explicit locking-unlocking

100 buckets.
Inserts in thousands (time of concurrent list slower, almost no visible contention when dealing with hash table)

First goto:
- Most thread-safe - lock the WHOLE data structure (minimizes concurrency, but gives safety)
Optimize, break it down and lock pieces.

What did we do about concurrency before threads?
- Use processes not threads

Pipes + Signals

- Pipes are way to implement a buffer between two processes
    - data must flow through the kernel (because data doesn't share address space)
    - it is an area of memory that is managed like an open file, but with FIFO operations
    - reads and writes occur concurrently by different processes
    - pipe() returns two file descriptors to same process

(W) --> [FIFO Buffer] --> (R)

Implemented through forks, parent will close one end, child will close end not using.

Issues (two conditions):

- FIFO is empty and a read occurs (reader can't get any data)
    - Have the reader block on FIFO that cannot shrink
- FIFO gets full (kernel runs out of memory for it)
    - Writer blocks on writing on FIFO that cannot grow
- No possible race conditions; kernel is the manager

Signals Deal with Asynchronous Events

- A software (kernel) analogy to a hardware or software interrupt.
- Ctrl-C configured to interrupt program execution
    - doesn't go into a buffer
    - instead execution is interrupted

    |≥ --> [Character Device] --> (R)

- Character causes hardware interrupt and stops user program running
    - character is processed by kernel
    - if it is ctrl-C, the program associated with the device should be "interrupted" signaled.
        - Kernel sets a bit in the task data structure

Fielding a signal

- Set a bit corresponding with the "signal number" in an array of signal bits in the task struct
- When the process is READY to run and about to be running, the kernel checks the bit set first.
- By default, Ctrl-C the INT signal causes process termination
    - Divide by zero, turn to signal, process termination
    - Segmentation fault, turn to signal, process termination
- But if the process has registered a subprogram to "handle" a signal number, the kernel forces a call to the subprogram.
- One signal cannot be caught: KILL signal (sig 9)

Locks, Condition Variables, Semaphores (concurrency control mechanism)

- Locks are architecturally + historically very important; synchronization at a low level, but not really part of the operating system.

- Semaphores are important theoretically and in programming. More general (because originally theoretical.

- Condition variables are a variant of lock-like **synchronization** (but not the same as locks) that can be used to implement **semaphores**. And for **other things** other than semaphores.

About Locks

- Mutual exclusion ("one at a time")

- A lock is held by a thread, and can only be unlocked by that thread

- But there are situations where one thread can wait for some **global condition** of the **process** (meaning all threads + their shared data structs) which a different thread may cause to hold

- One thread **waits**, another **signals**

- These are the reasons why we need condition variables

Book

- If the parent creates the child, and child goes to run before parent manages to wait, then the parent won't be signed (race condition).

- We declare as a global variable the condition variable

- To combat race condition, we also use a mutex lock, to enable the parent to

set up things so that it can wait, and do so in an atomic way without possibility

of race condition (MUTEX + CV eliminates race condition)

- Every use of CV requires a mutex

thr_join() does the waiting

- parent locks the mutex

- parent goes into waiting state with the mutex locked

- setting done to zero and waiting atomically

- the child cannot get in between setting done to zero and the condition wait

- we call wait and exit will do the signalling

thr_exit() does the signaling

- acquire the same mutex lock (but mutex was acquired in thr_join())

- the call to wait also atomically unlocks the lock (pass in condition variable and mutex to wait). When unlock, then thr_exit() gets to run

- set done to 1 and we signal, and then mutex is unlocked

- once signal is received and thr_join wakes up, and thr_join reeacquires the lock

Why do we say while (done == 0)?

- possible: when wait returns, done hasn't been set to 1

- the reason to test done == 0 at all:

in parents code, we did thread create

then set join

how does it result in done being true, when we get to cond_wait

only place done not be zero is in thr_exit()

what must have occurred: THE CHILD RUNS BEFORE PARENT (race cond.)

child could set to 1

but why is it a while instead of an if?

In some uses of CV, you can have more than one use of CV in the same program and you can't always guarantee therefore that when wait completes that the condition will still hold (has to do with implementation of condition variable)

- possible to get woken up when it's not your turn; have to check again

- could be given a false wakeup

Use of done = 1 and done == 0 is like a lock; use of that in synchronization

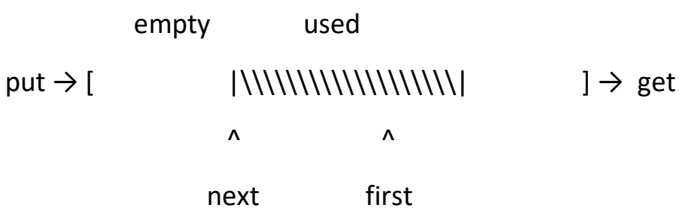
Bounded Buffer Problem – solves same problem that pipe solves

Bounded Buffer Problem

- A buffer resource is shared between threads

- The buffer is of fixed size (different from pipe) and contains records of a fixed size

- The problem is the count how many (and which) slots are used (full of data) and how many are available (empty)



Bounded Buffer Synchronization

- Mutual exclusion on filling/emptying slots and modifying the pointers

- An attempt to put data into a full buffer (no available slots) should block until one is freed

- An attempt to get data from an empty buffer (no full slots) should also block until one record is consumed

Book:

- Solution, we use to CVs: one for empty and one for full

empty producer, when sees that there is a maximum count

- if count is max, then we're going to wait until one is empty

- producer signals that they have filled something

fill consumer, when sees count empty wait fill

- if count is zero, wait until filled

- signals that is has empty (someone is waiting for something to be empty)

Always do signaling when filling and emptying

put/get circular FIFO buffer with "pointers" (modular)

A semaphore is packaging up of condition variable with a count. Count never goes negative (if negative, process blocks until positive again)

Book (Semaphore)
- Dijkstra pioneered; was an abstraction (parallel programming not exist yet)
- Primitives: sem_wait() and sem_post()
    - Similar to condition variables, but doesn't just involve condition, but variable as well
- Semaphore like CV is a type that you declare, we initialize it to a value (the third parameter of sem_init())
- Semaphore is a counter. Sem_t s is an integer value with an initial value.
    - Value of semaphore can be negative, but no process will see negative, so we wait
    - wait is decrement, post is increment

Semaphore that takes 0 and 1 and initial value is 1 (binary semaphore). Then we have a lock (though not a true lock); we get mutual exclusion. A lock is more restrictive in that a lock has to be released, but semaphores can be used to signal among different threads. One thread can do wait and another can do a post (can't do that with true lock).

Know for the exam:
How to use Binary semaphore
How to use lock, when we're talking about true lock used to release

Binary Semaphore use as signaling (parent waiting for a child)
sem_init: have the initial value be 0
- first attempt to decrement will result in block
- parent creates thread and does wait
- child has to do the post (signal)

More common way to use semaphore
Initialize semaphore to some count

Producer/Consumer (Bounded Buffer) problem:

- Use one semaphore to count one slot
- Use another semaphore to count another slot

Need two to handle (block) two different conditions, when full and empty

Same put and get routines
Tricky part in synchronization
- Two semaphores: empty and full
    Sem_init empty to MAX
    Sem_init full to 0

(Producer to put)
sem_wait(&empty) // decrement empty slot
put(i)
sem_post(&full) // increment full slot

(Consumer to get)
sem_wait(&full) // decrement full slot
tmp = get();
sem_post(&empty); // increment empty slot

But there's a problem with this: works if there's exactly one producer and one consumer
- if you have multiple producers or consumers executing same code, the problem is race condition.
    If two instances of put are executed, location of fill is shared. Write is not atomic operation, so two calls to put that happens concurrently may result in race condition.
    - e.g. fill gets modified before read buffer[fill]

Adding mutex (using like a lock) -- keeps the race condition from happening, but problem of **deadlock**
We protected code that needs to happen atomically with a mutex.
Producer
sem_wait(&mutex)
sem_wait(&empty)
put(i)
sem_post(&full)
sem_post(&mutex)

Consumer
sem_wait(&mutex)
sem_wait(&full)
put(i)
sem_post(&empty)
sem_post(&mutex)

New Problem: Deadlock

Mutex is acquired and may never be released
If empty semaphore at 0, the wait is going to block and we won't proceed
For mutex to be released is if there's a post in the call through consumer
But in consumer, it will also try to acquire the mutex and that's going to block

- Process stops running, have to kill the process. In important application, deadlock could cause brick; typically use will turn off system and turn on again and hope deadlock won't again occur.

How do we fix this
- Program carefully
- Move acquisition of mutex inside (between wait and post of empty and full) instead of outside
sem_wait(&empty)
sem_wait(&mutex) **<-- inside**
put(i)
sem_post(&mutex) **<-- inside**
sem_post(&full)

Another way: do wait and post in opposite order.

Semaphores
- A counting condition
- A way to avoid working with raw condition variables
- Can be used to implement other types of locks (reader-writer locks)

Reader-Writer Locks (new synchronization primitive which utilizes semaphores)
- A kind of lock which is not symmetric.
- Symmetric: Lock A, use A, release A
- Any number of readers to read particular resource at same time
- If writer wants to modify, there has to be no other readers reading and no other writers writing
    - Writing is a lock
    - Reading, we want to enable more concurrency

Reader-Writer Locks
- semaphores - way of using condition variables and using mutual exclusion (a more generalized lock)
- have a count involved

Multiple readers can work together (we have many readers, but one writer)

- writelock (used for mutual exclusion)
- count for a number of readers (not implemented as semaphore)
- set # of readers to 0
    - need keep track how many
- reader need to be made atomic because manipulating count (to increment, we need to lock it)
- if first reader, no one's acquired writelock

- read lock (conditional acquision of write lock)
- write lock - decrement number of readers
    If 0, sem_post to release writelock

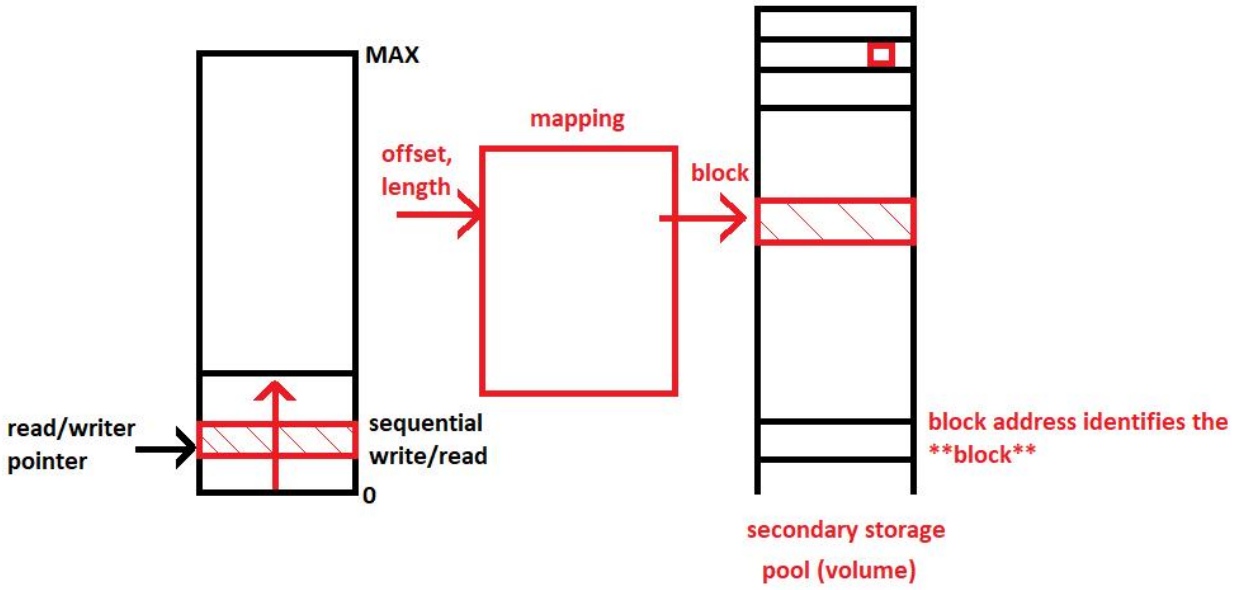
- problem with reader-writer: starvation (we keep reading and reading)
    can't ever acquire the writelock
    - so many readers that there's no opportunity to write
    - no notion of fairness
    - fairness policy (examples)
        - no writer waits for reader to read twice (implement round robin, need keep track of reader) (complicates threaded structure)
        - look at certain amount of time spent
        - choose # of readers and can't ever have more than that # before a write

File Systems
- punch cards (sequential) in the beginning, then magnetic tapes
- storage, computation, network more of less made of same stuff

What Is A File
- Simulation of a card deck
    - early files systems implemented files as sequences of fixed sized records
    - a record is like a struct constructed of printable characters
    - UNIX got rid of records so files are sequences of bytes
    - file type is application level and is not necessarily record-based
- A sequence of bytes of variable length

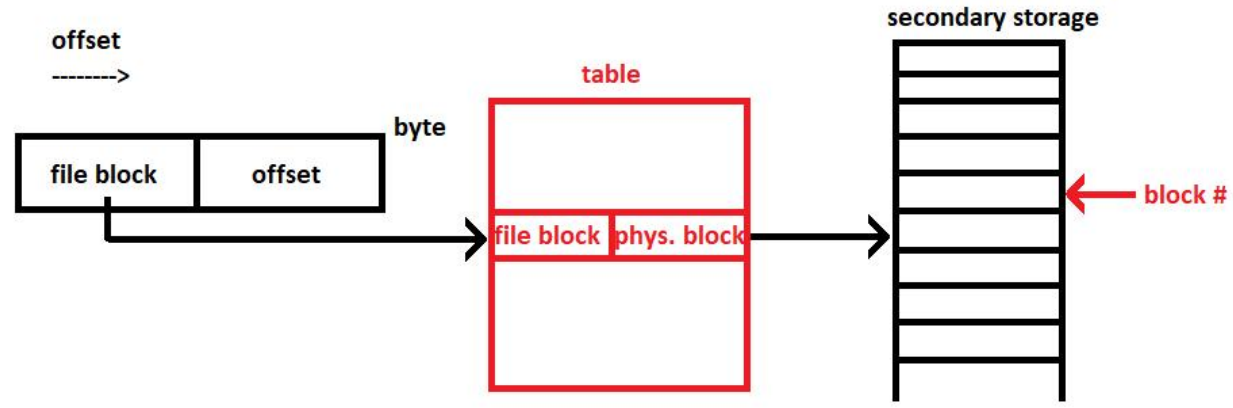
How do we Implement Files
- Early: contiguous allocation of storage medium (type, drum, disk)
- Addressable ***block*** storage
    - addressable tape (DEC tape)
    - drum and disk (particularly disk with moveable heads)
    - ***sequential access*** used to implement random (rotating media)
- Secondary storage is used like RAM which is large, slow and accessed by block (512 bytes to 4 MB)

A File is An Addressable Data Extent (Space)



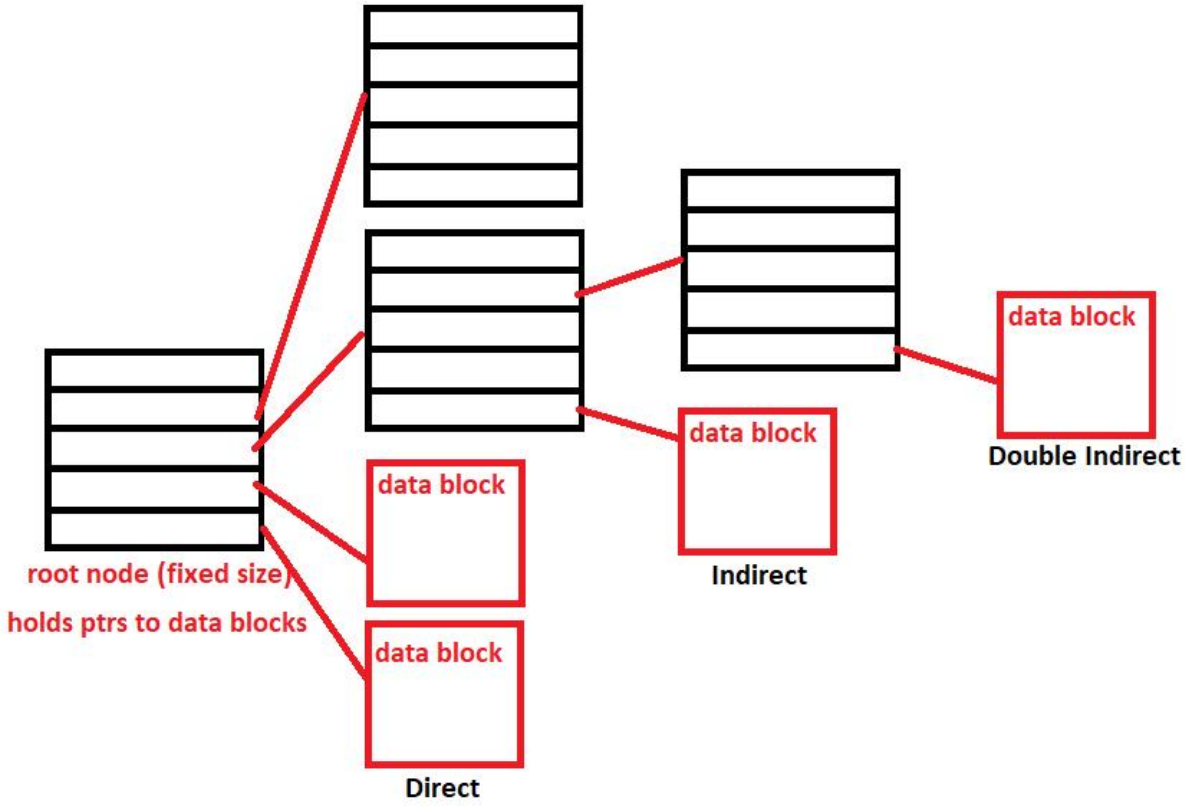
- File offsets not called addresses (but that's what it is)
- Kind of process, but only data part. Just data, no code execution
- Could argument in some sense there is, just not flexible code execution, can't give file system a user program, but there is executable code part of the operating system
- "File is a process with fixed code"

How is the File Mapping from Offset to Block # Implemented



We Implement Table as Tree

A File is a Tree



- This data structure is stored on secondary storage
  (In UNIX, name of this data structure is a struct inode)
- Generally, storage is segmented into inodes and data blocks

How Do We Speed Up Reading + Writing Partial Blocks to A Files?

- We cache file blocks (both data + inode) in RAM
- Recently accessed blocks reside in faster main memory
- Reading a block can require writing back a dirty block or evicting a clean one
- For cache policy we use **LRU** sometimes or something else (generally not)

A File is a lot like a Process

- Except for executing arbitrary code
- Block caching uses secondary storage + main memory with cache policy to gain performance
  - File resides on disk
  - Cached to memory
- Page swapping uses same with cache policy for performance
  - process resides in memory
  - and swapped out for space

File Implementation

- How read/write works
- How we implement concurrency among processes
- How we implement file naming and directories for data management

Files Implemented as Inode Trees

- the term “inode” is often used to refer to the root

(see illustration on previous page)

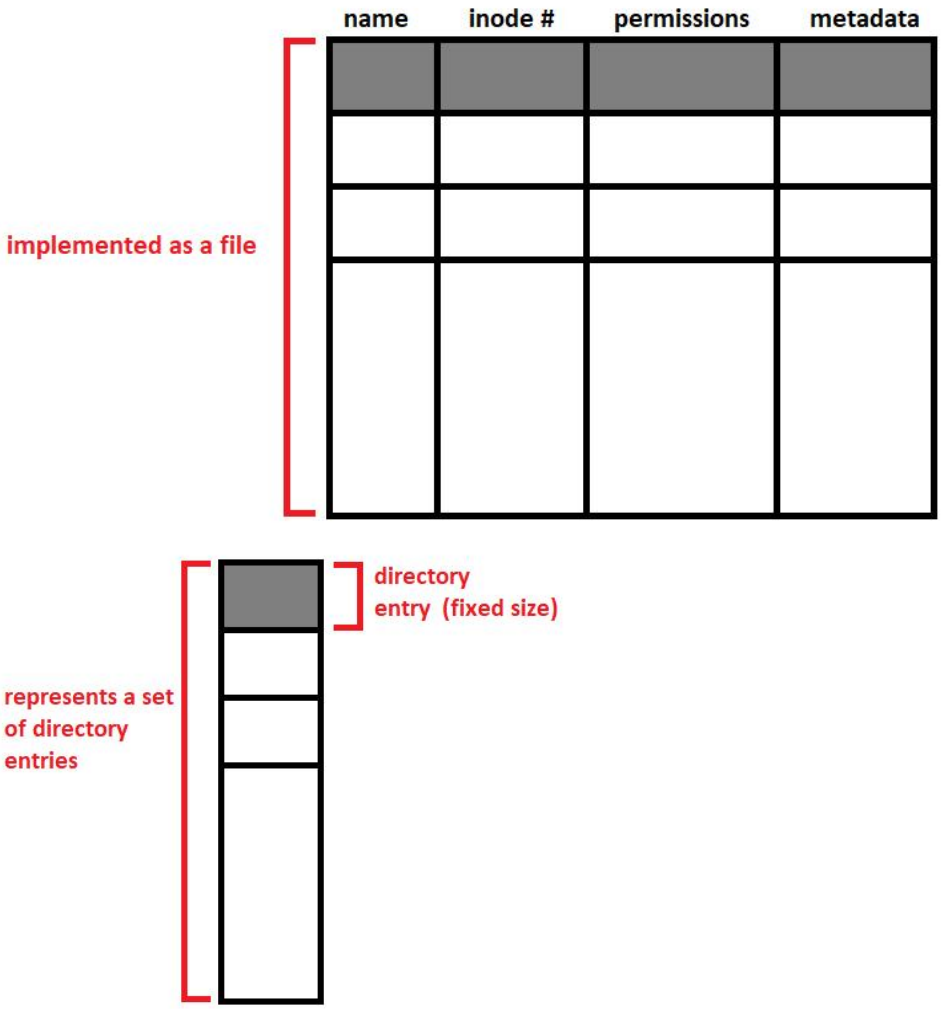
- every block has a block address
- inodes are managed in an array on storage, referred by index in that array (inode number)

(volume, index #) is the “internal” name of a file

Directories Are Files

- Maintained by file system
- Contain file system information
- Map symbolic file names to inode number
    - symbolic file names are human friendly
    - this level of indirection allows file implementation to change (special inode #)

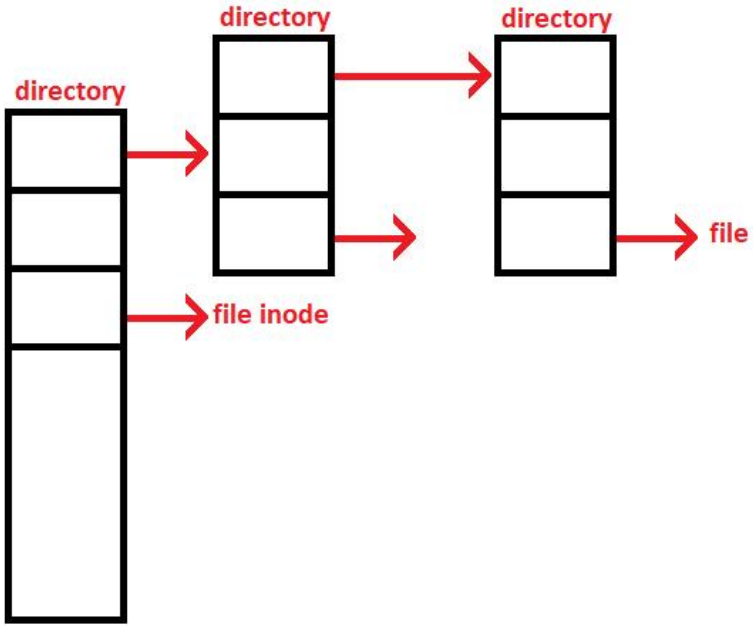
Simple Directories



- identity - metadata
  - user id, group id
- permissions
  - expressed in terms of owner, group, everyone
- read, write, execution
- other forms of meta data
  - time of creation, last modification
- type
  - file or directory

Directory Hierarchy

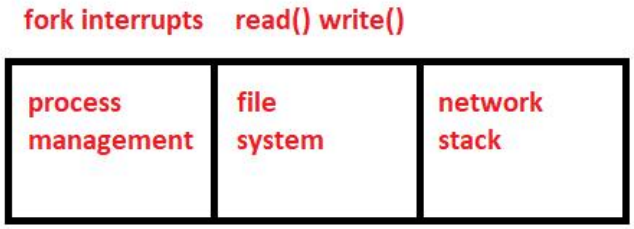
- directory entries can point to other directories



- names within a hierarchy are **paths** through the tree

mbeck/classes/cosc360/index.html

Every File System (volume) Has A Root Directory



OS Kernel

A single storage volume contains a data structure consisting of directories + end user files

This data structure is called a “file system” (another use of the term)

Name of root directory is /

absolute path starts at root

relative path doesn’t

Connecting A Physical Volume To A File + Directory Hierarchy

- We need info about the layout of file of blocks with the volume
- Inode vs. Data blocks
- Inode number of root directory
- Every volume contains a superblock at a known location is the volume (typically block #)
- Superblock + empty file system has to be created by admin

How Does The Kernel (OS) Keep Track of Volumes?

- Every volume is listed in an OS table
- Each type of volume is identified by a major device number
- The specific volume is identified by a minor device number
- We access a volume by (major, minor)

A Whole Volume File System can be MOUNTED within a Directory

- Root of the mounted file system appears as a subdirectory
- Hierarchy of volumes
- Root of that hierarchy is the root file system
- Its root directory is accessed by /

/usr
/etc
…

Storage + File Systems

1. Storage medium (disk or SSD)

2. Formatting (laying out blocks)

    special driver functionality

3. Drivers are **modules** within the kernel

    each driver type has an interface (two types of devices)

        - block devices (e.g. disk, SSD)

        - character devices (e.g. terminal, serial, network, ethernet) – burst traffic

        drivers are linked in with **kernel**

        modern OS can allow dynamic loading of drivers

            + into directory "/dev/____"

File System Data Structure

- Superblock, inodes, data blocks

    this is what's on disk when FS is not mounted.

- Directories built out of files also part of data structure.

- Within this data structure, block number is an address (relative to volume)

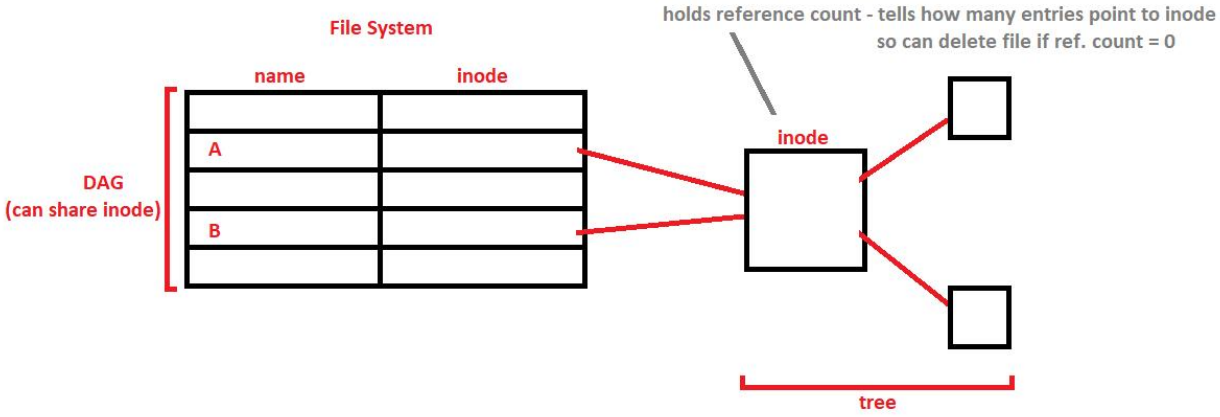
- Inode numbers uniquely identify files

ALL THE FILES, INODES, … WITHIN ONE VOLUME!

File Links + Deletion (directory level)

1. A file system is a tree or a directed graph.

2. Two directory entries can point to (by inode #) the same file.



Only file entries can be non-tree

3. Two paths can specify the same file (inode)

4. File **linking** rather than file **copying** (efficient) (in POSIX, there is no file copy sys. call)

Hard vs. Soft Links

1. Hard links are within a volume.

2. Soft links work across volume

- A soft link is a file of a special directory entry type
- In the file is a path name
- Open of the soft link is redirected to the path it contains

3. Soft links introduce new errors!

- broken soft links (or circular)

Mounted File System

- Root directory of the volume is identified with a "mount point" which is an already-existing directory
- mount device mount point
  * restricted operation
  * any files or subdirectories of mount point will become inaccessible
  * OS maintains a mount table
- On mount, read superblock + root directory information into OS kernel.

Open Files

- Upon open, pathname must be resolved
    * start with / or with current directory
    * the current directory of a process is a directory associated with the process by making its **inode** number a field of the **task data struct**
    * change current directory is a system call **(chdir)**
- During open the kernel follows soft links (extension of open which follows pointers)

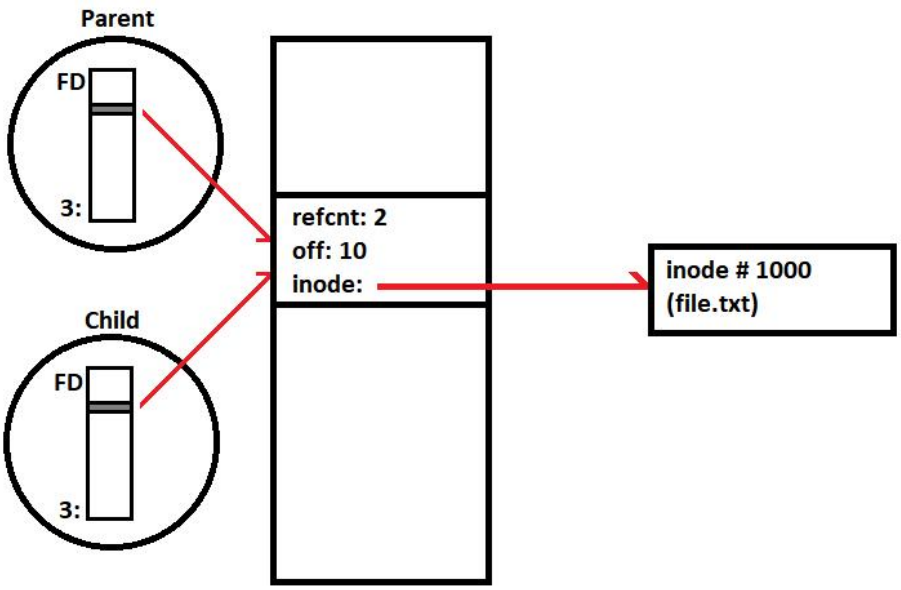
Open File Table

- When open reaches inode of file it reads it into an entry in the open file table (in-core inode)
    > in-core another name for in-memory
- A file table is essentially a cached inode plus some additional information (like the file pointer).
- The opening process puts a pointer to the file table entry into its file descriptor list (part of task data struct)

In the file descriptor table, the offset in this table to the pointer into the open file table is what's called a file descriptor (FD) at user-level and is the return value from open. FD refers to entry in this table, and is index. **Central** part of UNIX process model, but not part of execution model.

This file descriptor is a data structure that includes a type field and that type tells the kernel whether that entry points to open file table, **or a pipe, or even an open device**. If you open a device, you get a FD which is a pointer to an open device (not an open file).

In UNIX everything is a file. Not exactly true. Pipe is not a file. A device is not a file. But they all are accessed through file descriptors and the same system calls as files. Network connections are created by a different syscall (not open) but socket which returns a FD, but it's for a network connection which has to be bound by a network. But everything is named by a FD -- accessed by FD or mapped by process in memory through load/store.



Reading + Writing

- In reading from a file, we use the file pointer (a field of the open file table) and move it
- Upon open, file pointer is at offset zero
- Access multiple inode block on storage + reading data block(s) would be very slow

Buffer Cache Policy

- Use LRU on inode blocks
- Data blocks are managed using a lookahead (readahead) strategy
    - ordinary temporal locality predicts repetitious reading
    - spatial locality says nearby locations will be read soon
    - readahead says sequentially following bytes will be read soon **(sequential access)**
        (THIS IS HOW WE GET HIGH PERFORMANCE)
        - works great **UNLESS you seek**; when you seek, it changes file offset, change file pointer somewhere else.

Reading spans kernel and file, in order to get good performance, on sequential access in particular

Writing Files

- When reading, data starts in storage and is **cached** in memory buffers.
- Reading is sequential by default with lookahead policy, switch to LRU when non-sequential mode (a bunch of seeking)
- When writing, data starts in a user process, is transferred to a memory buffer (which may have to be read from storage, if it's not empty)
- Then written **back** to storage as needed
    (you don't want to write back on every read, too much transfer)

- Dirty block buffers are those that are different than what's being written out to disk. Dirty buffers are **dangerous** because data loss.

    Catastrophic than loss of data:
        Inodes - the tree data structure on disk

        If you've changed inode, you didn't explicitly write to inode, but you made changes to the file.

- no garbage data: when empty, you fill with zeros

What About Dirty Inodes in Memory Buffers

- Take some current block of the inode tree and lose them
    - file can be corrupted without changing any data blocks
    - pointers to inodes or data blocks belonging to **other files** (they've been freed, deallocated).
    - portions of files can be orphaned (disconnected from root)
- So we could just delete bad files
- Collect up fragments

fsck - file system check; if you had a crash, you run fsck twice (1st clean up, 2nd check for correctness)

fsed - file system editor; manually edit inodes, if you really know what you're doing, to get valuable files back

How To Fix The Problem

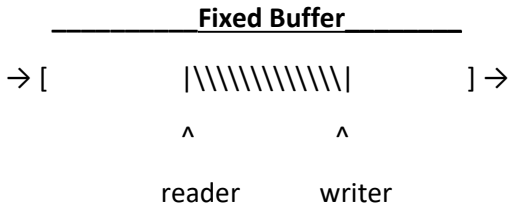
- We can **arrange** for the version of any file in storage to be consistent (a **snapshot** from some point in time)
- In the file system data structure, if block A points to block B, and both are dirty, write B out to a new block (not overwriting) before writing A out to storage.

File Access Concurrency

- Two processes or threads (light-weight processes) can read/write the same file concurrently. (in parallel or interleaved)
- Write to a file is non-atomic (and so is read - has to go to disk and get data blocks)
- Overlapping reading is generally not a problem - familiar with that from reader-writer locks (no changes to stored blocks)
- Single kernel does locking of its data structure as needed

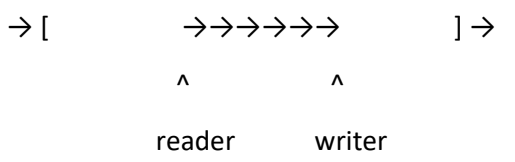
Concurrent Updates to Files

- One sequential writer and one sequential reader of an initially empty file (reader-writer would not get sequentiality, so not similar there)
- To get concurrency is reading + writing
- This is similar to the **bounded buffer problem**



- read + write happened on different locations
- synchronize on **empty** + **full**

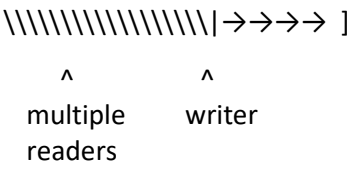
Unbounded Buffer



$ produce > out & (in background)
$ consume < out

Unstructured Concurrent Writes

- The kernel can do concurrency control
- But there will still be **race** conditions between readers + writers



- Brute force: **lock** the entire file
  open **modes**: readonly, writeonly, exclusive

This is all how a monolithic kernel can deal with concurrent writes. But now, what about multiple kernels?

Persistent OS State

- kernel image (in order to boot): nonvolatile storage/memory (does get updated)
- OS utilities (gets updated even more)

  often require reboot; get into mode where utilities are not running in order to change them

- OS configuration has to go somewhere
  - in the **root file system** (this file system is known at boot time)
  - root is mounted during boot sequence
  - configuration files in /etc

    for example /etc/password

- other examples: device configuration and file systems to mount
- more: terminal devices (login not generated by kernel; generated by program called login)
  - login processes

(all depends on **(root file system)**)

The Init Process

- At boot time, load kernel image (e.g. typically from flash memory)
- kernel and file system is executing initially
- goes through hardware initialization
- The first process created is called **"init"** and its purpose is to create other user processes
- Init reads /etc/inittab

    This is a list of files to fork/exec

    All processes are descendants of init

    Login processes are in the inittab, runs at root and uses /etc/passwd to know identity and what shell to run

Other Configure Mechanisms

- Many other subsystems have a configuration
    - traditionally they were all in **/etc**
    - applications may use subdirectories of **/usr** for their executables and configuration files
- This is a mess
- An alternative is to have a single database of configuration information for everyone

    (in Windows it is the registry)

Directories hold metadata

- inode holds structural metadata
- the directory entry contains permissions and ownership information
- every file is "owned" by one user, and by default the identity of the process that created it
- every file has a "group" which is a set of users, defined in /etc/groups
- there are three sets of permissions

    user, group, world

- permissions are READ, WRITE, and EXEC

    - READ is a user process

    - EXEC allows kernel to open file and read it

BUT we've added more permission bits

Other Permission Bits

1. Setuid bit indicates that when a file is executed (exec) the child is given the identity (uid) of the file not its parent

    - a file owned by root but with setuid bit on will run with root (supervisory) permissions

2. The sticky bit on an executable file says when it terminates it is not removed from swap space

    (use the copy on the swap space for programs that are used often such as the shell)

    (shared over and over again by processes using the same file)

Comparing Processes to Files?

- There is no execution model associated with a file.
    - the address of a process changes on its own controlled by register values.
    This we call execution.
        - PC
        - Instruction set
        - Sequential fetch/execution
- The contents of a file doesn't change between writing and reading

Does A File System Compute?

1. Yes - computing hashes for integrity
    managing inodes
    allocating, free, compaction
    error detection, handling

Does Stored Data Change?

2. On a physical level, yes!
    - environmental factors: magnetic field
    - cosmic rays (can flip bits)
    - random changes

Error Correction Within FS

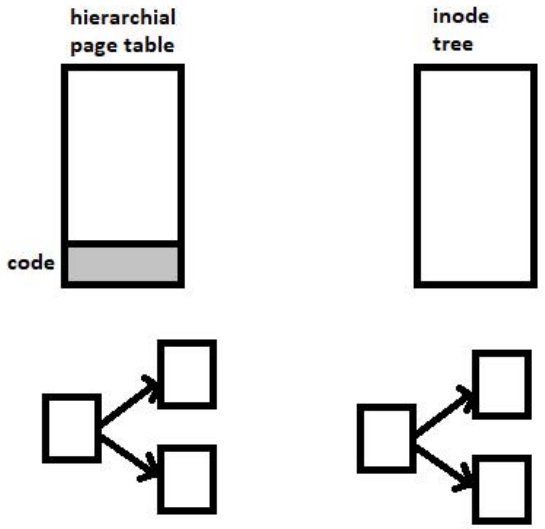
1. Computing checksums for detection.
2. Storing redundantly to correct corruption.
3. It is possible to take data, compute multiple parts which can be stored separately.
    - original data can be reconstructed from a subset of the pieces

RAID - redundant array of inexpensive disks

File system only computes identity function.

File + Process Implementation

- Processes are built out of pages/frames (where frames are in memory)
- Files are built out of storage blocks
- To implement large address spaces, process management uses page table (trees with PTE at the leaves)
- To implement large files, file system use inode trees with pointers to storage blocks at the leaves



How to Associate Executable Code with the Contents of a File?

1. Exec: unpack code + data from a file to create a new virtual address space w/ code
2. How associating the contents of a file with a running process

Memory Mapped Files

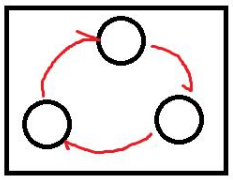
1. Contents of a file appear in an unused part of a process' address space

- the mmap() system call
- splice inode tree into the Page Table

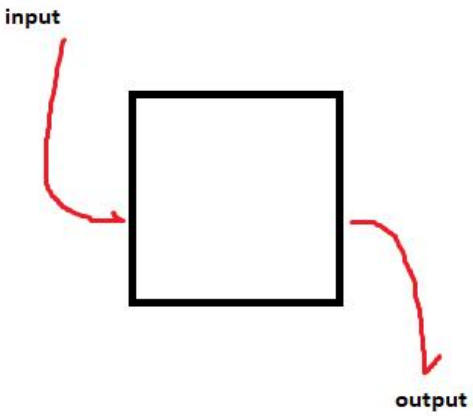
2. Use Page Faults as in Copy-on-Write
   - invalidate memory
   - on access, read a block into a buffer
   - change Page Table to point to buffer
   - restart access instruction

Managing the State of a Computation (state referring to any stored value)

- Computation state was originally considered to be small and local.
  * the finite automaton is contained within one node or computer



  * early computers had small numbers of registers/locations
  * a process captures this idea



The Larger Picture of State/Data Management

- In this course, the lifetime of program states (variables, stack, heap) is the lifetime of the process
  - from creation
  - to termination

- Is it the case that the information has no **value** anymore?

Example: Prime Sieve

1. To decide whether a number N is prime, try to divide it by every number small than it (small than sqrt(N))

2. A prime sieve takes advantage of the fact that you only need to try dividing by primes

3. Whenever a number is found not to be prime, remember that (build up an array of what is/isn't prime)

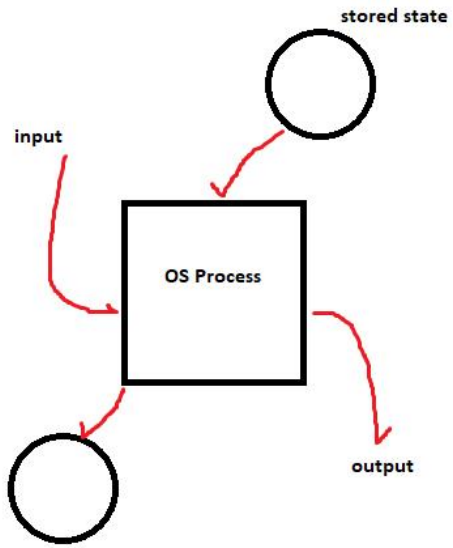
"memoization" - a form of caching within a program

The array/cache of prime information has value (where value is worth)

1. Any program that evaluates primality would run faster if they started with those values.

2. But we throw it away. Why?

3. We can write it to a file.
    - But then the file has to be managed and reread. Cost of transfer.
    - Is it trustworthy?



Long Lived, Less Easily Changed State Not Bound to One Node

1. A database stored in Cloud used by many applications.
    - We have to establish trust.
    - Update may be restricted to authenticated users. (trust)
    - Location of data has to be managed for performance
    - Long-term preservation

Data Ecosystems

1. Data consists of file systems and databases that hold scientific data sets that are identified by metadata (name, how produced, who produced it, etc…)

2. Data sets are distributed across many locations

3. Data may be produced continually from some instruments (Large Hadron Collide, SNS, etc.)
4. Output is produced + stored everywhere.

Cloud has side effect of being expensive and not being accessible everywhere in the world (need excellent high-speed networks)

Trust is in the Cloud data center.