CSE120 Final Review

This includes detailed summary of lecture slides and content. It also refers to the essential problems in review and research and sample midterm/final. Feel free to contribute and collaborate.

Logistic

- CSE120, Winter 2019, Professor Pasquale, Joseph.
- Exam date: **3-6pm, March 16, 2019**
- Format: 40-60 multiple choice problems, with explanation
- Coverage
    - 55-60% on post-midterm material (Lecture 8 to Lecture 16)
    - 25-30% on pre-midterm material (Lecture 1 to Lecture 7)
    - 15% on all the PAs

Topics Summary

- Processes
    - definition, scheduling, synchronization, inter-process communication
- Memory
    - Memory management, physical vs. logical vs. virtual, segmentation and paging, page replacement
- File system
    - File system interface, name space & directories, file system structure, block allocation and management, block cache
- I/O
    - Structure of I/O system software (functionality of layers and interoperation), device drivers, buffering (why and why not buffer, where to buffer).
- Protection/Security
    - General domain/resource protection model, capability lists vs. access control lists, protected subsystems, attacks on security, cryptograph
- Network/Distributed Systems
    - Protocol, network protocol layers, The Internet, distributed systems vs. centralized systems, fundamental distributed algorithms, problems: two generals, black/red hats.

Resources

- Collaborative review and research problems
- Sample Midterm
- Questions on the book
- Another Review doc by Yilin X

- PA 划重点式复习
- Pa_notes_github
- concise_version

LECTURE 1 - Introduction

- Why Study OS
  - Know how your computer works
  - System programmers get respect
  - Classic area of CS
  - Intellectually challenging

- Operating System
  - Software that makes computers easier to use by allowing users to interact with programs and programmers to use machine resources (CPU, memory, storage, I/O devices)
  - Improve performance, reliability (fault tolerance), security

- **Kernel**
  - A subset of operating system: parts that **absolutely critical** to the functionality of computer
  - Work with hardware by accessing registers, responding to interrupts
  - Allocate basic resources such as CPU, memory, and I/O
  - Provides basic functionality: run a program
  - Support process to run
  - Text, data and multiple stacks

- Two purpose(for OS)
  - **Provide abstract machine** → Simplicity, convenience
    - interface
    - function and resources
  - **Manage resources** → Performance, reliability and security
    - Mechanisms and policies
    - Allocate usage of space and time

- Undesirable into Desirable

Reality	Illusions
Complexity of hardware	Simple, easy to use resources
Limited processors and memory	Unlimited processors and memory

    - **Abstracts the actual behavior** of the computer and **provides users the functionality and interface** they need to use the computer.
    - Programmers focus on algorithm design, not machine
    - Minimize accounting for computer limitations

- Key Ideas

Abstraction	Desired illusion: representation without exposing background details, e.g. process and files.
Mechanism	How to do : determines how to do something, e.g. implementation of context switch
Policy	Ways to achieve mechanism: determines what will be done, e.g. scheduling policy

- Measure of performance
    - Speed: how fast computer can accomplish given task
    - Efficiency: how fully the computer manages to use its resources

- Measure of reliability
    - Correctness: given a certain input, how well can it produce the correct output
    - Fault tolerance: in case of error and failure, OS can continue to work correctly

- Measure of security
    -
    - Protection of data and information from access to and/or modification by unauthorized user
    - **Protection** is a part of security which controls access to a system by refining the types of file access allowed to the users.
    - **Privacy**: the ability to determine what data in a computer system can be shared with third parties.
    - **Authenticity**: the assurance that a message, transaction, or other exchange of information is from the source it claims to be from.
    - Integrity: the assurance that data is real, accurate, and safeguarded from unauthorized user modification.

LECTURE 2 - Process

- Goal
    - Given limited CPU and memory, user wants to run multiple programs using kernel

- Process

- Abstraction of a running program, **program in execution**
- Dynamic: has state and changes
  - **Memory** to maintain state (space)
  - **CPU** to execute instructions (time)

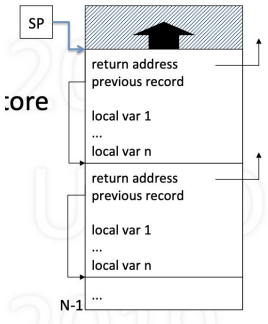
- Context

	Machine and kernel **related states**
CPU	Register values: PC, SP, FP(frame pointer), GP(general pointer)
Memory	Pointers to memory area: code, static variables, heap, shared, stacks of activation record

- Process memory structure

Text	Code
Data	Global variables, dynamic allocation heap
↕	↕
Stack	Activation records, automatic growth and shrink

- Process stack - <u>stack of activation records</u>
  - One per pending procedure （function call).
  - Where to return to
  - Link to previous records
  - Local variables
  - Stack pointer points to top



- Multiprogramming
  - Running multiple programs on a single CPU by quickly context switching between programs to give the **illusion** of simultaneous execution
  - ***Voluntarily*: process wait for resources, not actively using CPU,** so the CPU is not forcibly taken.

- **Context Switching**
  - Each process has its own text, data, and stack sections.
  - Allocate CPU from one to another
    - Save context of currently running process
    - Restore (load) context of the next process to run
      - Load registers, stack pointer
      - Load program counter (run the process)
  - Yield:
    - A process voluntarily give up CPU (may be implicit)
    - **Common code put in the kernel**
    - **Kernel**
      - **Stack:** hold the stack of **EACH PROCESSES**. Kernel allocates a separate stack for each process.
      - **Data:** global data that does not change
      - **Text:** hold code for the program
  - Trap:
    - Cause kernel entry
    - TRAP20 is a compiler generated code
    - Generally don't directly call yield by programmers, system calls call yield for us. Yield has TRAP20.
    - Process may not know any other processes from user space.

  - Kernel:
    - Text, data, and multiple stacks
    - After jumping to the system call's fixed location, kernel takes control.
    - Separate **stacks** per process **inside** the kernel
      - User space and kernel space separate
      - Protect the kernel from dynamically allocating new memory
  - Routine

```
magic = 0;           // local variable
save A's context:    // current process
  asm save GP;       // general purpose registers;
  asm save SP;       // stack pointer
  asm save PC;       // program counter, note value!
if (magic == 1) return;
else magic = 1;
restore B's context: // process being yielded to
  asm restore GP;
  asm restore SP;
  asm restore PC;    // must be last!
```

- Machine language to modify the registers

While executing Process 1...

1) Set magic to 0
2) Save context of current process (Process 1) into PCB_1
    i) PC points to the checking magic statement
3) Check if magic == 1 {return}, else {set magic to 1}
    i) Flag the calling to see if it had already yielded to another process
    ii) Make it possible for a process to return from Yield ().
4) Restore context of other process (Process 2) from PCB_2
    i) The context is stored in a shared memory, usually a static array
5) Execute Process 2
6) Save state into PCB_2
7) Reload state from PCB_x
8) Execute Process x

- Activation records
    - Before context switch:
        - The current running process
        - Top of the stack pointing to r**eturn address of yield**
    - After context switch:
        - The process that is yielded to
        - *? Top of stack correspond to the **return address of Yield() of the new process***

- savecontext(C) return twice, first from explicit call it, second from restore context()

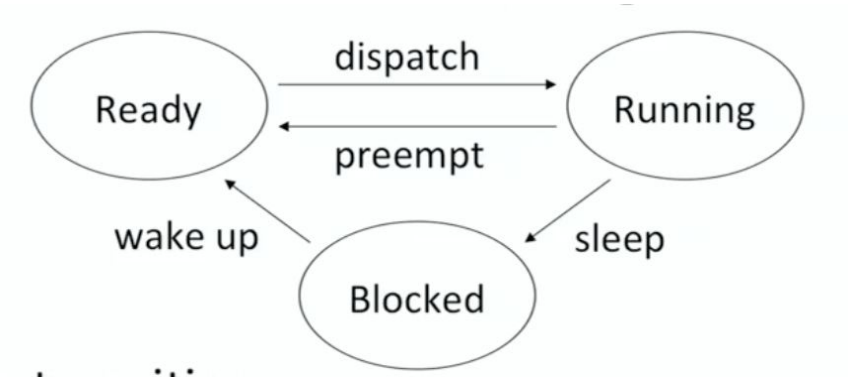
<u>LECTURE 3 - Time Sharing</u>

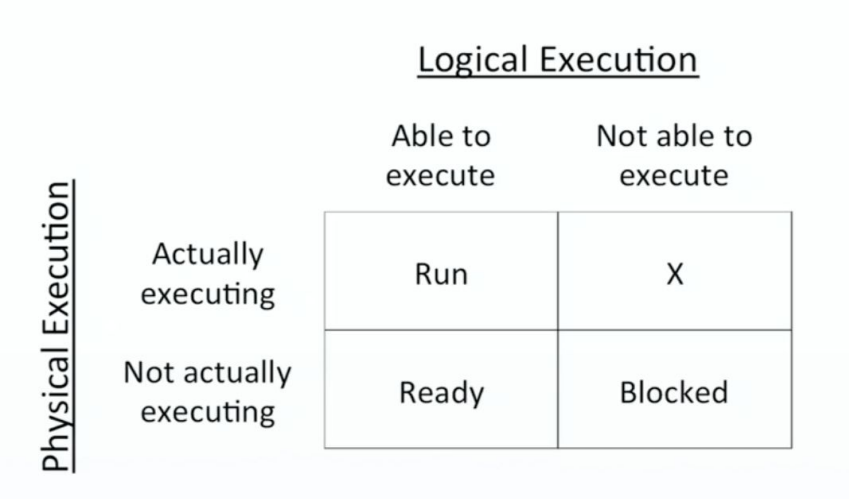
- **Time Sharing**
    - **Rapidly switching the CPU** to create illusion of simultaneous running
    - Multiple processes, single CPU
        - Single CPU **cannot** run multiple processes simultaneously
    - Illusion of parallel progress
        - Conceptual: each process makes progress over time
        - Reality: each process **periodically gets quantum** of CPU time

- Quantum: smallest unit of CPU time.

- Implementation
    - **Kernel keeps track of progress**
        - Every time a process gets a quantum, we keep track of dat
    - States of process
        - Running, Ready, Blocked
    - Select ready processes to run based on scheduling implementation

- Process state transition





  - Running → Blocked
    - Make resources request, etc.
  - Blocked → Ready
    - The system generates interrupt, causing kernel to run
    - Kernel checks registers and figure out to wake up the blocked process
  - Scheduling: how to select the next process to run
  - **Preemption**
    - The kernel takes away the CPU
    - Interrupt physically implemented in the hardware as **clock**

- **Kernel**
  - Code that supports process
    - system calls like fork(), exit(), read(), write()
    - management code such as context switching and scheduling
  - It runs when
    - system call

    - hardware interrupt occurs: **clock** (modern machines have different levels of interrupt Lec3 S9 QA)
- The kernel runs as part of the running process
    - Although the CPU time of kernel is charged for the current process, it's good as long as the time is spread evenly for each process. (*Lec3 S7)*
    - The process jump into the kernel's code and then the kernel does the rest (Lec3 S9 QA)

- Kernel stores information in *Data Section* for system management
    - List of processes (PID, state, other info)
    - Contents of CPU contexts
    - Areas of memory used
    - Reason for being blocked (condition for unblocking)

- Kernel gets control by
    - Process give up control voluntarily: explicit or implicit system calls
    - Preemption:
        - Hardware timer.
        - When running, resets the timer
        - Then call Yield()

- Kernel calls are like procedure calls, but it needs to cross the boundary (take more time)

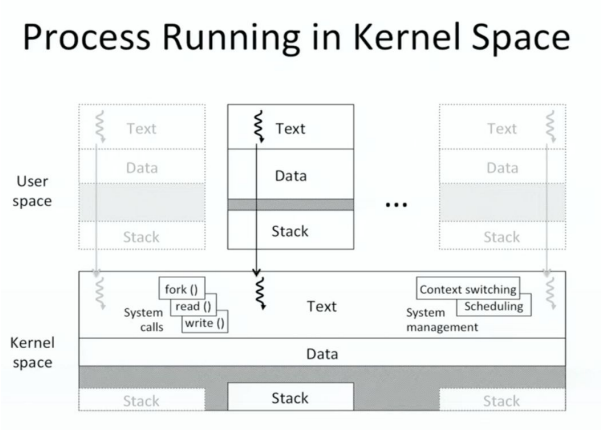
- **Context Switch**
  When system calls or interrupt occurs (HARD WIRED)

Hardware	**Software**
Switch from user to kernel mode	Save context, select process, restore
Go to **fixed** kernel location	Return from interrupt/trap (RIT)

    - Mode
        - bit in the hardware to indicate the mode
        - The process typically runs in user mode
    - Difference
        - Hardware has different modes
        - **All instructions** can be run in kernel mode, while user can execute some of the instructions
    - Kernel will set the state of current running process as ready

- Kernel on system call and clock interrupt example *Lec 3, S13-36*
    - **Yield: a new process' stack become visible (always visible to kernel)**

- **Change the stack pointer and program counter**
- **Context switching when interrupt occurs**



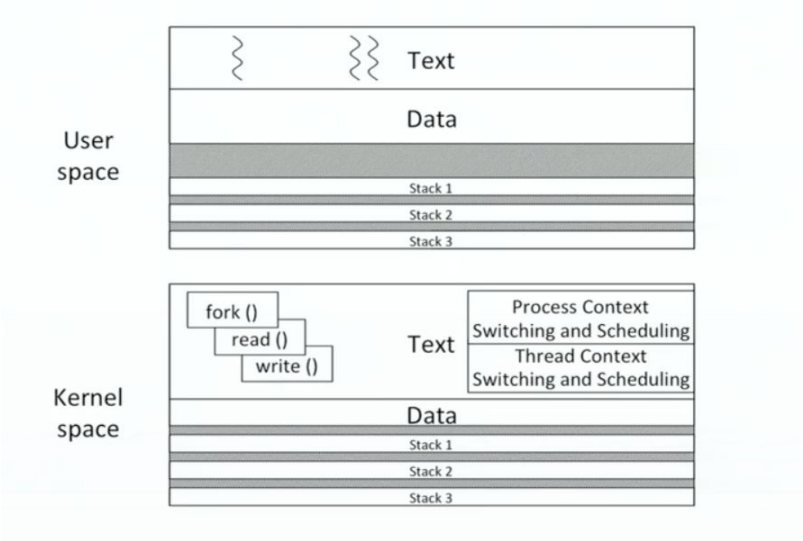
- Parallelism in process
  - Process as a **single path** of execution, **with memory** composed of text, data and stack
  - Multiple paths of execution
    - Single text and multiple execution
    - Single data and multiple execution can see others' update
    - Seperate stacks one per ongoing execution
    - CAN'T use multiple process because each process has its own text and shared global area. Although fork make copy of text, it doesn't work

  - **Threads**
    - Single **sequential path of execution**, abstraction **independent of memory.**
      - Process: path of execution + memory
    - Part of a process
      - Live in the memory of a process
      - Allow multiple threads in a process
    - To user as unit of **parallelism**
    - To kernel as unit of **schedulability**

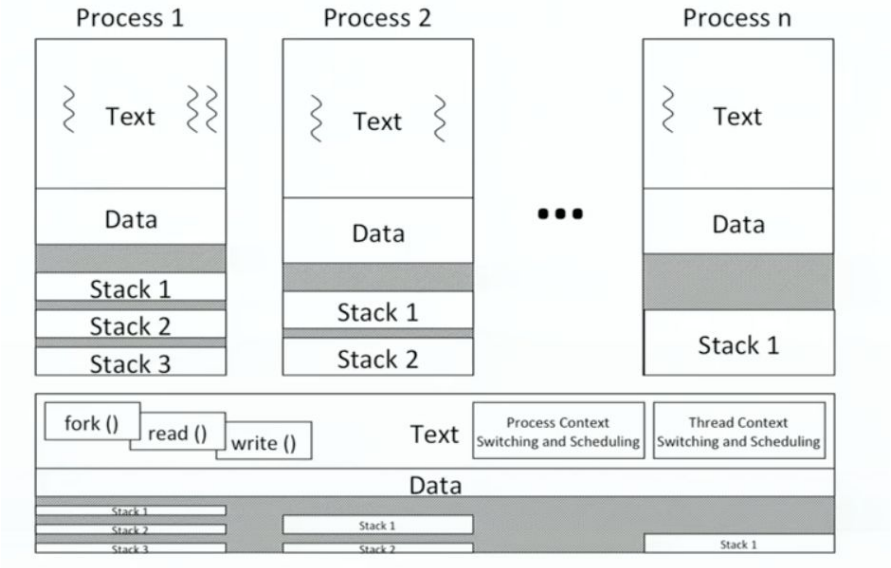
  - **Kernel-level** Thread Implementation
    - In the kernel
      - Thread calls (system calls: ForkThread())
      - Thread management functions (context switch, scheduling, etc.)
    - Each thread requires **both** user and kernel stacks
    - Kernel can schedule threads on separate CPUs.

  - Single Process, Multiple threads

- Each process's user and kernel space have multiple stacks. They may collide (generally large enough).



- Many process with multiple threads
  - Each process may have multiple stacks, many stacks inside the kernel
  - Kernel decides which thread runs on which CPU, if there are multiple
  - Kernel **schedules** the **threads**
  - CPU gets interrupt individually
  - If a thread calls Fork(), exactly the same memory (stacks)
  - Allow dynamic allocation, as long as it doesn't cause problems.



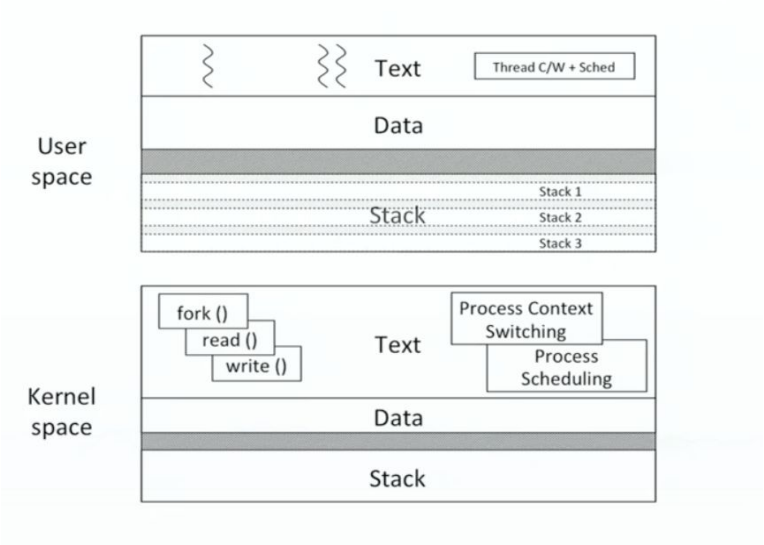
- **User-level** threads

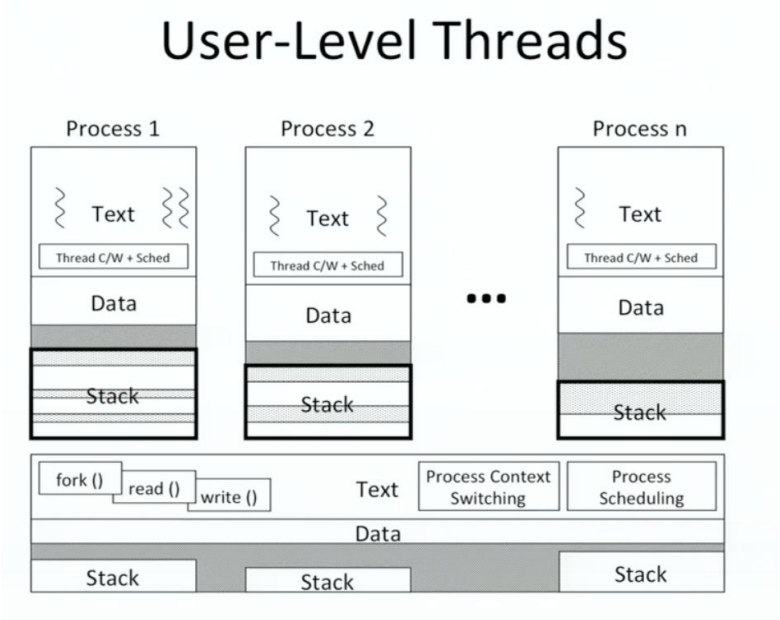
  - Programmers want to **express set of parallelism in programs**.
  - Language construct as multiple threads

    - If OS don't support, then can't assign to multiple CPU
    - But don't want to be limited the way of programming by understanding machines
- Have a primitive form of parallelism

- Support threads at user-level via **thread library**, regardless of the kernel supports it or not. The library automatically divide up the stack kernel gives the process to achieve one stack per thread.

- At user level
    - Thread calls
    - Thread management
- No true parallelism





- When a particular thread runs, the corresponding stack would be used.
- When a system call, whichever the thread makes the system call would run. Upon return, possibly yielding CPU between threads

- Advantage of multiple threads on a single CPU
  - Not only parallelism in using CPU, but also parallelism in using other devices
  - E.g. a thread makes resources request. The kernel, instead of schedule another process, it gives back the CPU time, Then the user level could schedule another thread to run.
  - Operating system has to be **designed specifically** → give back CPU instead of taking away.

- Pros and Cons of User/kernel level threads

	Pros	Cons
User-level threads	- **Portability** as works on any kernel - **Efficient** as thread switching occurs in user space - User decide on **scheduling**	- No true parallelism
Kernel-level threads	- Achieve **true parallelism**	- Overhead as thread switch requires kernel call - Costly, heavy weight

  - Most modern os allow you to specify the user/kernel level threads, allow you to have software clock; context switching occurs at user level.

- **Thread Support and Execution**
  - User-level vs. kernel-level threads
    - Thread **support part of** user or kernel code
  - Running in user space vs. kernel space
    - Thread **running in** user or kernel space
  - E.g. User level threads can also run in the system, via system calls.

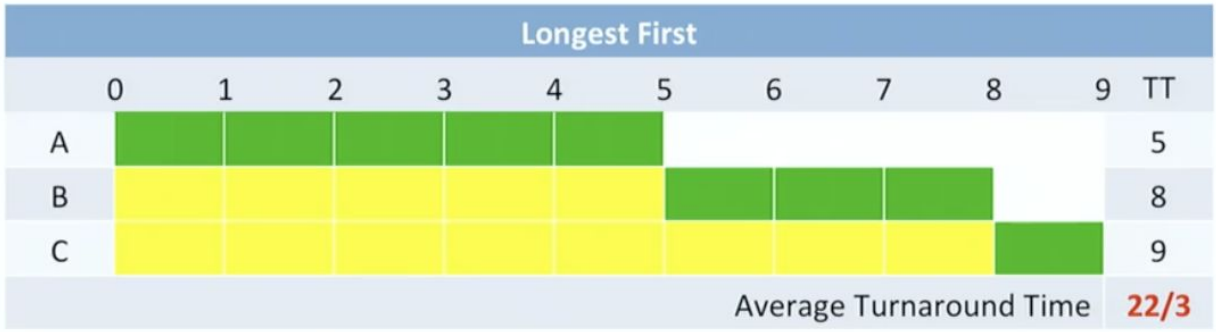
LECTURE 4/5 - Scheduling

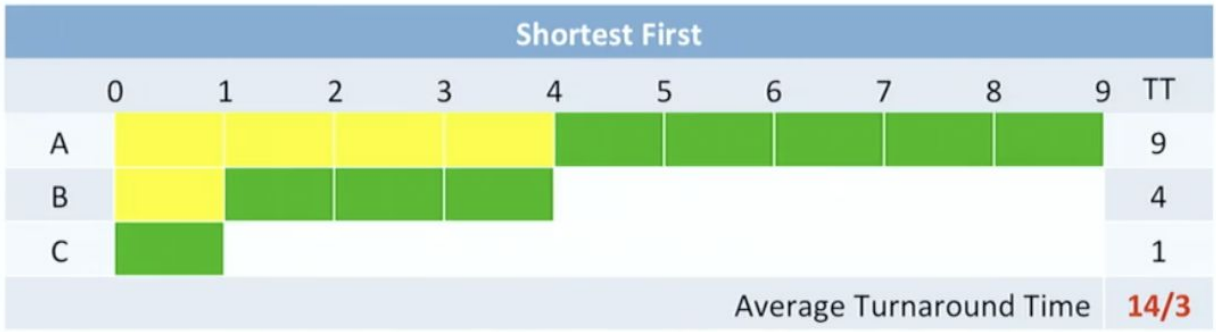
- Given multiple process and one CPU, decides which process gets CPU and when

- **No single best policy:** different and even conflicting goals
- Mechanism: the best way to do something

- Example Format
  - Arrival time
  - Service time: CPU time each process needed to complete
  - Turnaround time: wait for CPU + uses CPU

- Longest First vs. Shortest First





  - Assume we know the execution time.
  - In general **shortest first is provably optimal**, although both takes up the CPU
    - Given n processes with service time S_1, S_2, …, S_n; then average turnaround time is
    - $[S_1 + (S_1 + S_2) + \ldots + (S_1 + \ldots + S_n)] / n = (n * S_1 + (n - 1) * S_2 + ... S_n) / n$
  - That's why supermarket have express lane O. O.

- **F**irst **C**ome **F**irst **S**erved - NONPREEMPTIVE
  - Allocate CPU to processes in order of arrival
  - **Pros**: non-preemptive, simple, no starvation
  - **Cons**: poor for short processes (don't get special attention)

- **S**hortest **P**rocess **N**ext - NONPREEMPTIVE
  - Select process with the shortest service time

- Generally preemptive does better than non-preemptive
- **Pros: optimal** for non-preemptive
- **Cons**: allow starvation
- Assume **service times are known, theoretical**

- **R**ound **R**obin - PREEMPTIVE
  - Each process gets quantum in turn. Needs a clock.
  - In general better than FCFS in terms of average turnaround time. Modern machines have neglectable context switch time.
  - **Pros:** preemptive, simple, **no starvation**, wait at most (n - 1) * quantum
  - Wait **(n - 1) quantum** in the **worst** case
    - Processes may not use up the entire quantum.
    - In this lecture, assumed *compute bound*, used up entire quantum.

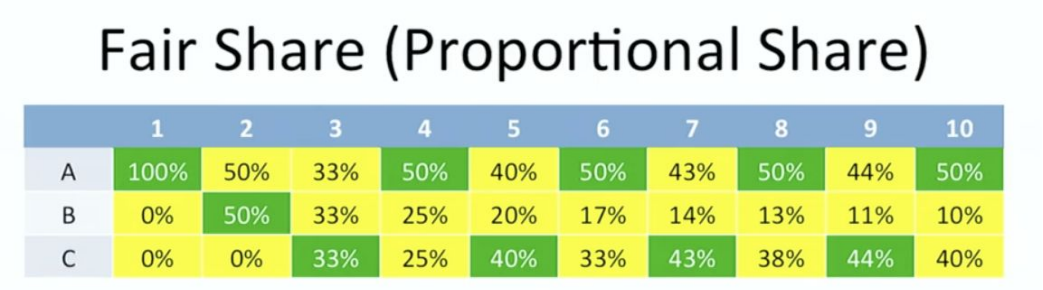
- **S**hortest **R**emaining **T**ime - PREEMPTIVE
  - Select process with the shortest remaining time
  - **Pros**: optimal for preemptive
  - **Cons**: allow starvation
  - Assume all service time are known, can't implement in real machines

- **Multilevel Feedback queues -** PREEMPTIVE
  - N different priority queues, each has a priority associated with it: 0 (high), .., N (low)
  - New process arrive on the highest priority.
  - **Strict rule**: Always select processes from **the highest priority** queue

- Run for T = 2^k quantums
    - Case 0: used up T, move to next lower queue, FIFO
    - Case 1: used < T, back to the same queue, RR
- Periodically boost
    - E.g. suffers from starvation
    - Thus, periodically boost up the priority
- **Pro:** complex, **adaptive**, highly responsive
    - Learn the CPU time along the way
    - Sorting over time
- **Cons**: favors shorter over longer, possible starvation
- In general, does better than RR, but not always.

- Priority Scheduling
    - Select process with highest priority, an **external criteria**
        - E.g. priority = 1 / cputime
    - Achieved different goals of the scheduling

- **Fair Share (Proportional Share)**
    - Each process requests some CPU utilization (percentage of time resource is used)
    - **Pros:** over the long run, actual ~ request
    - Determine which runs next quantum: select **minimum actual/request ratio**
    - **先算A，再B，再把剩下的给C**



    - **Cons:** inefficient by calculating actual / request and select min.

- **Stride Scheduling**

- Code
  - For process A, B, C with request R_A, R_B, …
  - Calculate **strides** $S_A = L / R_A$, ….
  - Each process x maintains **pass** value P_x (init 0)
  - For every quantum
    - Select process x with **minimum pass** value P_x, run
    - Increment the pass by $P_x = P_x + S_x$
- Optimization: $S = L / R_{x;}$ L as a large num

Stride Scheduling Example

Process *x*	Runs Now	A	B	C	To Run Next
Requested Utilization: R_x (in %)		50	10	40	
Stride: $S_x = L/R_x$ (L = 100000)		2000	10000	2500	
Pass: $P_x = P_x + S_x$; init = 0		8000	10000	10000	A
quantum 1	A	2000	0	0	B
quantum 2	B	2000	10000	0	C
quantum 3	C	2000	10000	2500	A
quantum 4	A	4000	10000	2500	C
quantum 5	C	4000	10000	5000	A
quantum 6	A	6000	10000	5000	C
quantum 7	C	6000	10000	7500	A
quantum 8	A	8000	10000	7500	C
quantum 9	C	8000	10000	10000	A
quantum 10	A	10000	10000	10000	… will repeat

ber

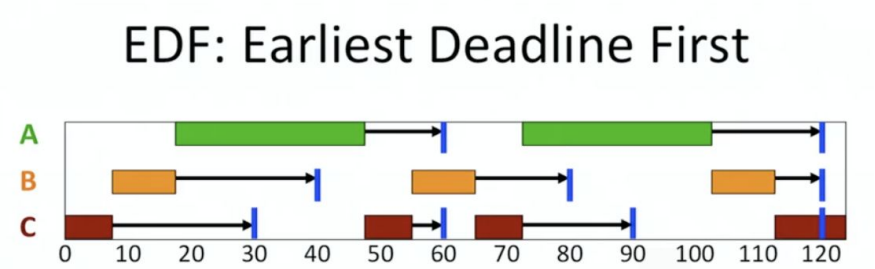
- <u>**R**eal **T**ime **S**cheduling</u>

  - Correctness of real-time systems depend on
    - **Logical result** of computations
    - **Timing** of these results
  - Types
    - Hard (weapon) vs. soft (stream video): must meet every deadline if hard.
    - Periodic vs. aperiodic

  - Periodic Processes
    - Computation is cyclic
    - C = CPU burst, T = period, U = C/T = utilization
    - May be possible to schedule: **sum of utilizations** doesn't exceed **100%**

  - **E**arliest **D**eadline **F**irst

- Schedule process with earliest deadline. If earlier deadline appears, **preempt**.
- **Pros:**
  - Work for both periodic and aperiodic processes, achieve 100% (if no overhead)
  - If new process came in, request for some U, we can **guarantee to meet all deadlines** as long as ≤ **100 %**
- **Cons: expensive,** requires ordering by deadlines, **nlog(n)**.
- OS can only goes with linear time solution



	C	T	U
A	30	60	50%
B	10	40	25%
C	7.5	30	25%

- **R**ate **M**onotonic **S**cheduling
  - Works only for **periodic**
  - **Prioritize** based on rates (**1 / T**), reduce the overhead of EDF
    - Select the **highest priority** first, preemptive if necessary
    - When burst down, wait till next period
  - Meet deadlines if (≤100%)

$$U_1 + \ldots + U_n \leq n\left(2^{1/n} - 1\right)$$

    - Monotonic decreasing function, max = 100%
    - If don't meet the constraints, it's not guaranteed to succeed or fail.
  - Sorting occurs **only once**

	C	T	U	Rate	Prio
A	20	60	33%	1/60 = 0.017	Low
B	10	40	25%	1/40 = 0.025	Med
C	5	30	17%	1/30 = 0.033	High

- **Optimal** but limited
  - **Optimal** for **static** priority scheduling (simple and efficient static priority based on rates)
  - **Limited**
    - Utilization lower bounded by **n ($2^{1/n}$ - 1) > ln 2 ~ 69%**
    - If the test passes, the deadlines will be guaranteed to be met (If **below the 69%**, it must work)
    - **Periodic** process only

- Summary
-

name	preemption	characteristic	starvation
First Come First Serve	NO	simple	NO
RR	YES	simple	NO
Shortest process next	NO	theoretical	allows
Shortest remaining time	YES	theoretical	allows
Multi-level feedback	YES	Adaptive, responsible, complex	allows
Priority	YES	External critertial	allows
Fair share	YES	Proportional allocation	allows
Stride share	YES	Proportional allocation	allows

Earliest deadline first(real time)	YES	100% utilization high overhead	N/A
RMS(real time)	**Preempt if necessary**	< 100% utilization low overhead Simple and efficient Optimal for **static priority algorithm** **Periodic only**	N/A

LECTURE 5/6 Synchronization

- Definition
  - Events happen at the same time
  - Process synchronization: when one process **waits for** another
    - E.g. A trigger some event, then wait B to trigger this event, synchronize
  - **Prevent race conditions**
  - Wait for resources when they become available

- Credit/Debit Problem
  - Deposit and withdraw 100 USD at the same time
  - Context switch and process writes incorrect amount of money
  - **Race condition**

- Critical Section
  - Interleaving these sections of code may cause problems
  - Sections of code executed by **different processes**
  - Run **automatically**, with respect to each other (whoever goes first, cannot be divided up)

- Avoid Race condition
  - Only programers can identify the critical section
  - Enforce mutual exclusion: only **one process active** in a critical section

- Atomicity
  - Indivisible
  - Seek **effective** atomicity: <u>can interrupt, as long as interruption has no effect</u>
  - Consider effect of critical section in insolation, then consider interruptions. if result OK, then OK.
  - Respective to others

- Achieve Mutual Exclusion
  - **Entry/exit** code: supplied by operating system
  - Entry act as a barrier: if another process is in critical section, block; else allow process to proceed
  - Exit code release the barriers for other PIDs

- **Four Requirements For Mutual Exclusion Solution**
  Given multiple cooperating processes, each with related CS
  - **At most one process in CS**
    - E.g. software lock doesn't work
  - **Can't prevent entry if no others in CS**
    - Processes outside the CS can't prevent processes entering the CS>
    - E.g. Take turns doesn't work
  - **Should eventually be able to enter CS (no starvation)**
    - E.g. State intention doesn't work
  - **No assumptions about CPU speed or number**
    - E.g. disable interrupt doesn't work (only work on one CPU)

- Variables:
  - Local variable: only accessible within the function of the local variable
  - Global variable: visible to any function in the entire program
  - Shared variable: visible to other processes

- Software lock
  - Use shared variable to flag open/close to indicate if any process in CS

```
shared int lock = OPEN;
```

P_0

```
while (lock == CLOSED);
lock = CLOSED;
< critical section >
lock = OPEN;
```

P_1

```
while (lock == CLOSED);
lock = CLOSED;
< critical section >
lock = OPEN;
```

  - **Issue**: Lock itself is a **race condition**
  - Break the **first rule** (at most one process in CS)

- Take turns
  - Shared variable turn indicate which process' turn

```
shared int turn = 0;        // arbitrary set to P0
```

P_0	P_1
`while (turn != 0);`	`while (turn != 1);`
`< critical section >`	`< critical section >`
`turn = 1;`	`turn = 0;`

- **Busy Waiting**: process is executing, but all it does is waiting
- **Issue**: if process 1 runs first (process 0 has not yet entered), but it gets stuck by busy waiting.
- Break the **second rule** (can run if no other process)

- State intention
  - Process states intent to enter the critical section. While other processes' intent is false, enter the CS; else, busy waiting.

```
shared boolean intent[2] = {FALSE, FALSE};
```

P_0	P_1
`intent[0] = TRUE;`	`intent[1] = TRUE;`
`while (intent[1]);`	`while (intent[0]);`
`< critical section >`	`< critical section >`
`intent[0] = FALSE;`	`intent[1] = FALSE;`

  - **Issue**: after stating true intend, switch context. Then processes would never enter the critical section.
  - Break the **third rule** (should eventually enter the CS)

- **Peterson**
  - Combine stating intentions and taking turns
  - Check **if another process's turn and it intents to enter**. If so, busy wait; else enter the critical solution.

```
shared int turn;
shared boolean intent[2] = {FALSE, FALSE};
```

P_0	P_1
`intent[0] = TRUE;`	`intent[1] = TRUE;`
`turn = 1;`	`turn = 0;`
`while (intent[1] && turn==1);`	`while (intent[0] && turn==0);`
`< critical section >`	`< critical section >`
`intent[0] = FALSE;`	`intent[1] = FALSE;`

  - Satisfies all the requirements; works for n processes.

- Unsatisfactory: **busy waiting, no way to get away from busy waiting.** (waste a whole quantum)

- Disable interrupts: no **clock interrupt**
  - No uncontrolled context switch → no races → no mutual exclusion
  - **Issue:**
    - A bug in the program that causes infinite loop, etc.
    - Assumption: only single CPU, can only disable interrupt on one CPU.
  - Break the **fourth rule** by assuming that there is only one CPU

- **Test-and-Set Lock Instruction** (hardware instruction)
  - Does two machine language instructions toge

```
do atomically (i.e., locking the memory bus)
      [ test if mem == 0 AND set mem = 1 ]
```

    ther, indivisible.

  - Occur without interruptions
    - One instructions that does two things
    - No hardware interrupts
    - Locked memory bus
  - C function that is **atomic**
    - Whatever in the lockptr is set to 1
    - We learned the value of the original lockptr is 0 or 1.

```
TSL(int *lockptr)
{   int oldval;

    oldval = *lockptr
    *lockptr = 1;
    return ((oldval == 0) ? 1 : 0);
}
```

Effectively, this is what the TSL instruction does in one atomic instruction

  - **Single, atomic machine instruction**: test if the lock is 0, if so return 1; else return 0; before returning set the lock to 1.
  - Works for any number of threads

```
shared int lock = 0;
```

P_0	P_1
`while (! TSL(&lock));`	`while (! TSL(&lock));`
`< critical section >`	`< critical section >`
`lock = 0;`	`lock = 0;`

  - **Issue:** busy waiting

- **Semaphores**
    - Abstraction of synchronization
    - Synchronization variable that
        - Takes on non-negative integer values
            - 1 = GO (can go but changes to stop)
            - 0 = STOP
        - Causes block or unblock
    - Wait and signal operations
        - **Wait(s)**: block if zero, else decrement
        - **Signal(s)**: unblock a process if any that is blocked by wait(), else increment
    - No other operations: **can't test values**
    - Used for **only synchronization,** can't **transfer information**

    - Alternative definition (classical)

Synchronization variable
– Takes on integer values
– Can cause a process to block/unblock
wait and signal operations
– wait (s) decrement s; if s < 0, block
– signal (s) increment s; unblock a process if any
No other operations allowed
– In particular, cannot test value of semaphore!

        - Same behavior

- Mutual Exclusion with Semaphores
    - Mutex semaphore, initialized to 1
        - Represent how many processes can enter a critical section simultaneously
    - Only **one process** can enter the critical section, and the rest processes get into **block** state.
    - Work for n processes

```
sem mutex = 1;          // declare and initialize
P0                      P1
wait (mutex);           wait (mutex);
< critical section >    < critical section >
signal (mutex);         signal (mutex);
```

- Oder how processes execute with Semaphores
  - Want P_0 executes before P_1
  - Semaphore "*cond"* as 0, indicating has P_0 done its job

```
sem cond = 0;
P0                                  P1
< to be done before P1 >            wait (cond);
signal (cond);                      < to be done after P0 >
```

  - If P_1 runs first, P_1 blocks because cond = 0, then P_0 runs, signal would unblock P_1
  - If P_0 runs first, then signal increments cond and P_1 can run
  - **No way** for P_1 to learn about anything of P_0, or whether it's blocked or not; no information transfer

- Semaphores Implementation
  - S = [n (non-negative integer values), L(list of processes blocked on s)]

```
wait (sem s) {
    if (s.n == 0) add calling process to s.L and block;
    else s.n = s.n – 1; }

signal (sem s) {
    if (s.L !empty) remove/unblock a process from s.L;
    else s.n = s.n + 1; }
```

- Wait and Signal Must be **Atomic**
  - Bodies of wait and signal are **critical sections**
  - Use lower-level mechanisms **(test and set lock, peterson's solution)**
  - Busy waiting still exists → lower level

- Lower-Level busy waiting
  - When process A executes wait() or signal(), **OS switches to process B wait/signal, B is busy waiting**
  - Much smaller time of the busy waiting; also happen very rarely

- **Never get rid of busy waiting**

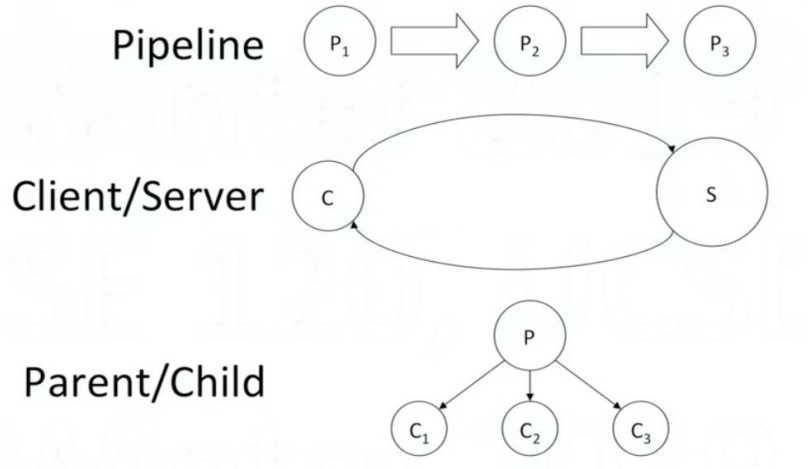
- Summary:

- Synchronization: process waiting for another
- Critical section: code allowing race condition
- Mutual exclusion: one process excludes others
- Mutual exclusion mechanism: obey four rules
- Peterson's solution: all software, but complex
- Semaphores: simple flexible synchronization
  - wait and signal must be atomic, thus requiring lower-level mutual exclusion (Peterson's, TSL)

LECTURE 6 Inter Process Communication

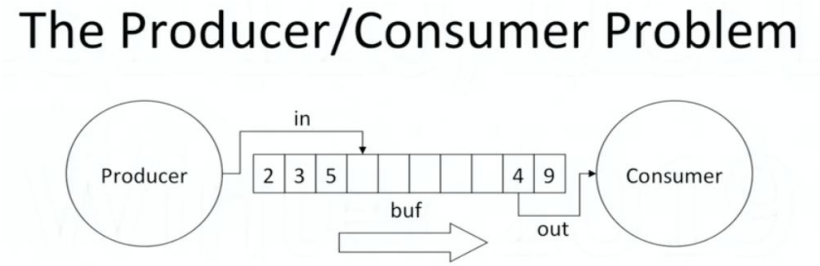
- Structure a computation as a set of cooperating processes
  - Performance (speed)
    - Exploit inherent parallelism of computation
    - Allow some parts to proceed while others do I/O
  - Modularity: reusable self-contained programs (UNIX philosophy)
    - Each do a useful task on its own
    - Useful as a sub-task for others

- Examples



- **I**nter-**P**rocess **C**ommunication
  - **Data transfer**
  - **Synchronization**

- Producer/Consumer Problem
  - Producer produces data and consumer removes data from that buffer



  - Transfer information by shared memory as **a buffer (temporary holding area)**

- Shared memory only

```
shared int buf[N], in = 0, out = 0;

Producer                        Consumer
while (TRUE) {                  while (TRUE) {
    buf[in] = Produce ();           Consume (buf[out]);
    in = (in + 1)%N;                out = (out + 1)%N;
}                               }
```

  - No synchronization
  - No mutual exclusion: multiple produces producing

- **Shared memory + Semaphores**

```
shared int buf[N], in = 0, out = 0;
sem filledslots = 0, emptyslots = N;

Producer                        Consumer
while (TRUE) {                  while (TRUE) {
    wait (emptyslots);              wait (filledslots);
    buf[in] = Produce ();           Consume (buf[out]);
    in = (in + 1)%N;                out = (out + 1)%N;
    signal (filledslots);           signal (emptyslots);
}                               }
```

  - Filledslots, Emptyslots both as semaphores.
  - Buffer empty (filledslots == 0), consumer wait, blocked
  - Buffer full (Emptyslots == 0), producer wait, blocked

- **Multiple Producers**

```
shared int buf[N], in = 0, out = 0;
sem filledslots = 0, emptyslots = N;

Producer1                Producer2                Consumer
while (TRUE) {           while (TRUE) {           while (TRUE) {
 wait (emptyslots);       wait (emptyslots);       wait (filledslots);
 buf[in] = Produce ();    buf[in] = Produce ();    Consume (buf[out]);
 in = (in + 1)%N;         in = (in + 1)%N;         out = (out + 1)%N;
 signal (filledslots);    signal (filledslots);    signal (emptyslots);
}                        }                        }
```

  - Inconsistent updating of variable buf and in → race condition
  - **Solution:** surround the critical section with another semaphores

```
shared int buf[N], in = 0, out = 0;
sem filledslots = 0, emptyslots = N, mutex = 1;

Producer1, 2, …                  Consumer1, 2, …
while (TRUE) {                   while (TRUE) {
    wait (emptyslots);               wait (filledslots);
    wait (mutex);                    wait (mutex);
    buf[in] = Produce ();            Consume (buf[out]);
    in = (in + 1)%N;                 out = (out + 1)%N;
    signal (mutex);                  signal (mutex);
    signal (filledslots);            signal (emptyslots);
}                                }
```

  - Critical sections of one type associated with producer and another type for consumers. We can do two different mutex to protect them separately
  - Need separate queues if each consumer needs from particular producer

- **Monitor**
  - Programming language construct for IPC
    - Shared variables requiring controlled access
    - Accessed via procedures (automatic mutual exclusion)
    - Condition variable (wait, cond) → sleep or wake up on the condition variable
  - Only **one process** and be running or able to run

Producer/Consumer using a Monitor

```
monitor ProducerConsumer {
   int buf[N], in = 0, out = 0, count = 0;
   cond slotavail, itemavail;

 PutItem (int item) {
    if (count == N)
       wait (slotavail);
    buf[in] = item;
    in = (in + 1)%N;
    count++;
    signal (itemavail);
    }

                              GetItem () {
                                 int item;
                                 if (count == 0)
                                    wait (itemavail);
                                 item = buf[out];
                                 out = (out + 1)%N;
                                 count--;
                                 signal (slotavail);
                                 return (item);
                                 }
}
```

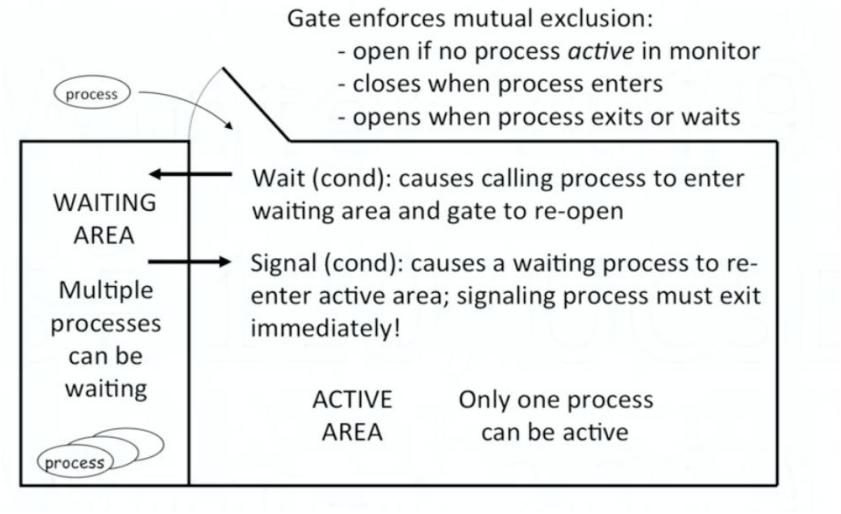
Producer

```
while (TRUE) {
   PutItem (Produce ());
}
```

Consumer

```
while (TRUE) {
   Consume (GetItem ());
}
```

- **Monitor wait means block; monitor signal means wake up exactly one waiting process**
- The processes are the one that get **blocked**, not the functions
- Monitors behave like a big box:
    - Automatically mutual exclude, **only one process in the active area**
    - Programming language understands the concept of lock



    - Signal as the last thing a previous process in the active area executes. (no two processes can exist in the active area).

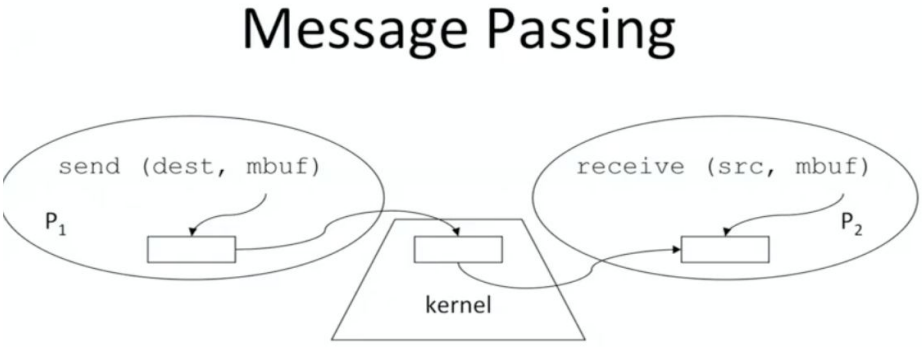
- Condition variables in monitors have **no values**, **no memory**
    - **Wait means block, signal means unblock**
    - Act as labels to allow to signal which **conditions** (distinguish why a process is waiting and who should wait up)
    - Associate **a queue** with wait and signal for each condition

    - If called without effect, then it's lost.

- Issues:
    - Package up the whole section
    - Compiler makes sure the section has mutual section
    - Compiler guarantee the **signal as the last statemen**t before returning
    - Count has to be added to indicate buffer full/not
    - Monitors force the structure of IPC
    - **Only one processes can run in the monitor at the same time**

- **Message Passing**

    - Two methods
        - Send (destination, message_buffer)
        - Receive (source, message_buffer)
    - Data Transfer: kernel message buffers
    - Synchronization: **receive blocks to wait for message** (kernel makes the process to block or wait)



    - Receive(): synchronous, include wait;
    - Send(): asynchronous
    - **But receive and send shouldn't be both asynchronous(ONLY ONE OF THEM）**

```
/* NO SHARED MEMORY */

Producer                        Consumer
int item;                       int item;

while (TRUE) {                  while (TRUE) {
    item = Produce ();              receive (Producer, &item);
    send (Consumer, &item);         Consume (item);
}                               }
```

    - **Kernel copy the communication between processes' local stack**

- Kernel could sense too many sending messages and block sender
- Kernel implement the reading and writing to **its buffer as synchronized critical sections**
- **Pros:**
  - Reduce complexity for programmers
  - Run on the systems **without shared memory**
  - **Safer** than shared memory paradigms (The process controls the writing to memory)

- Flow control
  - Consumer send the producer empty message (empty boxes), then the producer could send back to the consumer
  - Could have at most N messages
  - Don't need to rely on the operating system (may packaged in send and receive) → network protocol

```
Producer
int item, ready;

while (TRUE) {
    receive (Consumer, &ready);
    item = Produce ();
    send (Consumer, &item);
}
```

```
Consumer
int item, ready;

do N times {
    send (Producer, &ready);
}

while (TRUE) {
    receive (Producer, &item);
    Consume (item);
    send (Producer, &ready);
}
```

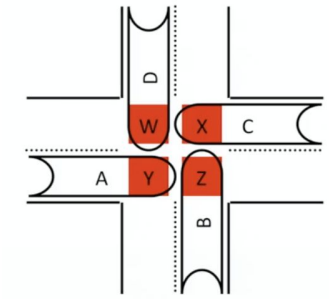
- Ports: **Address the messages to the mailboxes (ports), don't need to know the process ID.**
- Receiver: indicate to receive from anyone, **get PID by return values,** etc.
- Kernel buffering: outstanding messages → block sender

LECTURE 7 Deadlock

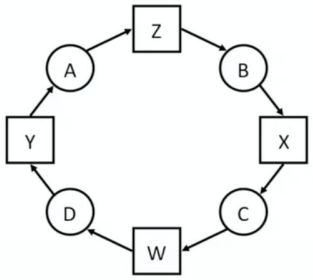
- Programmers identify the critical sections and then apply IPC.

- Definition: a set of processes are **permanently blocked** (unblocking of one relies on the progress of another)

- Examples circles



Cars deadlocked in an intersection



Resource Allocation Graph

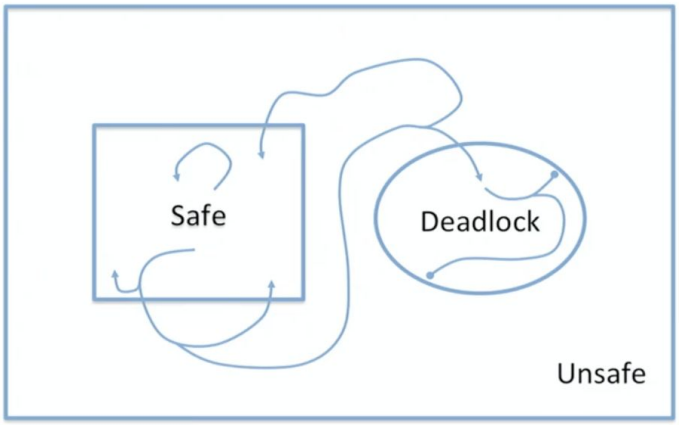
- Example: memory (deadly embraced)

- **Four conditions of Deadlock** (all to be present)
  - **Mutual exclusion:** only one process may use a resource at a time
  - **Hold and wait:** holding and waiting for another
  - **No preemption**: can't take resources away from process
  - **Circular wait:** waiting processes form a cycle

- **Deadlock Prevention** (make any of the conditions unsatisfied)
  - Can't force mutual exclusion (printer and etc.)
  - Get all resources all at one, but **process don't know what they need**
  - Allow resources to take away (printer and etc.) **can't support preemption**
  - Identify all resources and give a number; when process ask for resources in the ascending order → prevent circler. **Can't identify all resources.**
  - E.g. traffic light remove hold and wait

- **Deadlock Avoidance**
  - Avoid situations that lead to deadlock "safe"
    - Selective prevent condition for a period of time
    - Dynamic
  - Works with incremental resources requests
  - Need **maximum resources needed**

  - <u>Banker's algorithm</u>
    - Fixed number of processes and resources
    - System states
      - Safe: can **avoid deadlock** by certain order of execution
      - Unsafe: deadlock is possible, but no guarantee



- Given
  - Claim matrix
  - Resource matrix
  - Resource availability vector
- Whenever a new process request resources, then check if there exists **a process ordering** such that
  - A process can run to **completion, return resources**
  - Resources can then be used by another process
  - Eventually all the processes complete

Example of a Safe State

	Claim				Allocation				Avail-ability	Total
	P_1	P_2	P_3	P_4	P_1	P_2	P_3	P_4		
R_1	3	6	3	4	1	6	2	0	0	9
R_2	2	1	1	2	0	1	1	0	1	3
R_3	2	3	4	2	0	2	1	2	1	6

  - Avoid the system from becoming unsafe, even if unsafe state **may not** lead to deadlock.
- E.g. traffic lights: if only allow at most three cars into intersection

- **Deadlock Detection and Recovery**
  - Do nothing special to prevent and avoid deadlocks
  - Reasoning
    - Deadlocks rarely happen
    - Cost of prevention or avoidance not worth it
    - Deal with them in another way
  - Most general purpose OS's take this approach: user detects the deadlock.

  - Other: detect the "wait for" cycle

    - Identifying all resources, tracking their use, periodically running detection algorithm

LEC 8 - 10 MEMORY

- Memory Issues:
    - Where should process memories be placed: memory management
    - How does compiler model memory: logical memory model, segmentation
    - How to deal with limited physical memory: virtual memory, paging
    - Mechanisms and policies
    - *Abstraction by abstraction*, based on assumption of each level.

LEC 8 - Memory Management (*Feb 6th)*

- Definition
    - How to allocate and free portions of memory
    - Occurs:
        - Process created
        - Request more memory
    - Freeing:
        - Process exits
        - Process no longer needs memory
    - E.g. malloc()
    - E.g. stack grows and shrinks

- **Process Memory**

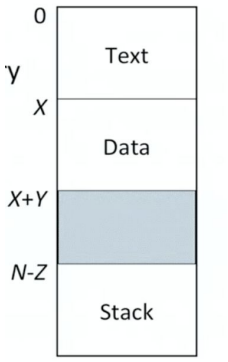
    - Region

Text	Code of program
Data	Static variable, heap
Stack	Automatic variables, activation records
Other	**Shared memory regions**

    - Characteristics:
        - size: fixed or variable
        - Execution rights: r, w, x.

- Process Memory's **Address Space**
    - Address Space
        - Set of addresses to access memory (set of all locations named)

- Memory:
  - Start from 0, increasing going down, sequential.

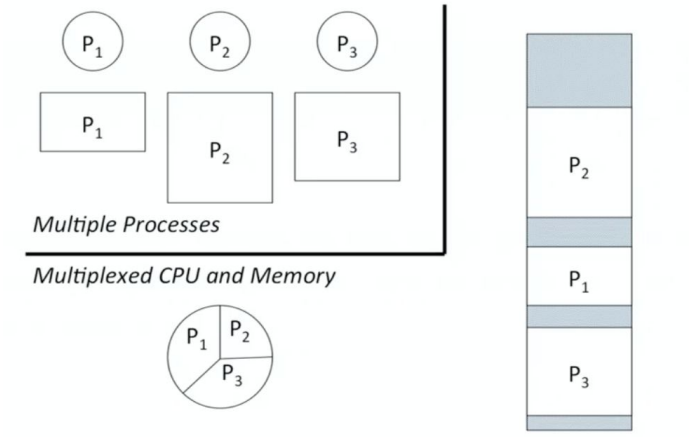


    - Text: 0 to X - 1

  - The process's view of the memory: simplified abstraction

- Complier's Model of Memory
  - These 0 - N modal could not possibly be physical addresses
  - Compiler generates memory addresses
    - Address ranges for text, data, stack
    - Allow data and stack to grow
  - Not known
    - Physical memory size (to place stack at high end)
    - Allocated regions of physical memory (to avoid)

- Goal: support **multiple processes**
  - Support running programs "simultaneously"

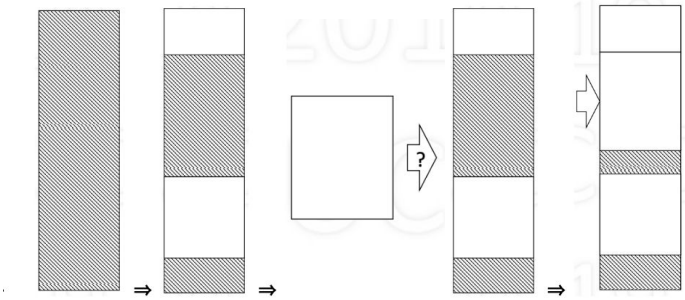


    - P_1 thinks it starts from 0, but in fact not

- Sharing physical memory
  - Problem:

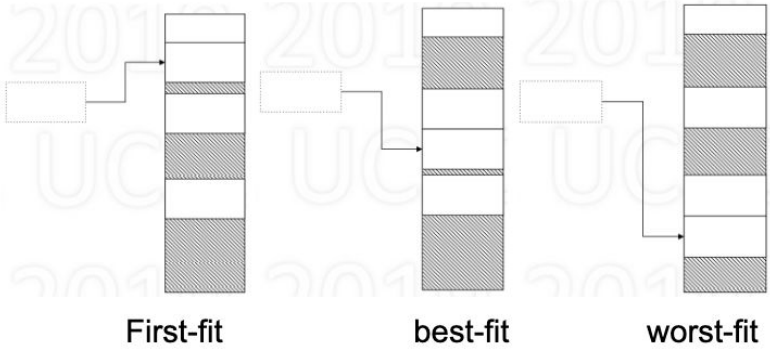
    - If process given CPU, **must** also **be** in memory.
    - Context switching is very fast, but loading from **disk** is very **slow** (10,000 times CST).
  - Solution:
    - keep **multiple** processes **in memory** and context switch only between processes in memory.

- **Memory Issues**
- E.g. physical memory starts as one empty “hole” and allocate over time.



  - To allocate memory:
    - Find a large enough hole (otherwise may not be able to run the process)
    - Typically leaves (smaller) hole
    - Rarely can we find perfect fit
  - When no longer needed:
    - Release
    - Creates a hole, coalesce with adjacent

- **Selecting the Best Hole**



  - First (next) Fit:
    - Pros: simple, fast
  - Best Fit:
    - **Smallest** holes: Pick holes as tight as possible: leaves very small fragments
    - Cons: must check every hole, useless tiny fragments - no use
  - Worst Fit: *actually the worst*

        - **Largest** holes: Pick holes as large as possible
        - Cons: must check every hole, leaves large fragments
    - *Note*: is region fixed or variable.
    - *Tradeoff*: fit vs. search time:
        - memory is cheap and **time** is expensive → First/Next Fit

- **Fragmentation**
    - Memory becomes fragmented: organized by little useless pieces
    - ***Internal:*** unused space within (allocated) block, a process asked for more than it needs, but didn't use). Come in handy for growth.
    - ***External*****:** unused space outside any blocks (result of allocation and deallocation). Can be allocated but too small so not useful.

- What if no holes
    - Compaction:
        - Move all holes to a big block of memory
        - Pros: simple idea
        - Cons: very time consuming: copy memories from one place to another
    - **Break Block into sub-block**
        - Pros: easier to fit, use memory more efficiently, doesn't require much time
        - Cons: complex

- **Given n blocks, how many holes**
    - There are n blocks allocated
    - On average, how many holes are there: **50% Rule**
        - M = N / 2.
        - Number of holes = Number of blocks / 2

- **How much memory lost to holes**
    - Average size of holes = **H,** Average size of blocks = **B**.
    - Fraction of memory lost to holes: $0 \leq f(b, h) \leq 1$
    - Unused Memory Rule:
        - **K = H / B**, ratio of average hole to block size
        - Fraction of **space lost** to holes: **F = k / (k + 2)**. F = mh / (mh + nb) and m = n / 2 since no perfect fit → f = mh / (mh + 2mb) = h / (h + 2b) → Let k = h/b, then f = k / (k + 2)
        - Wasted memory fraction: memory that is not used.
        - Steady state: run for a long time and then take snapshot.

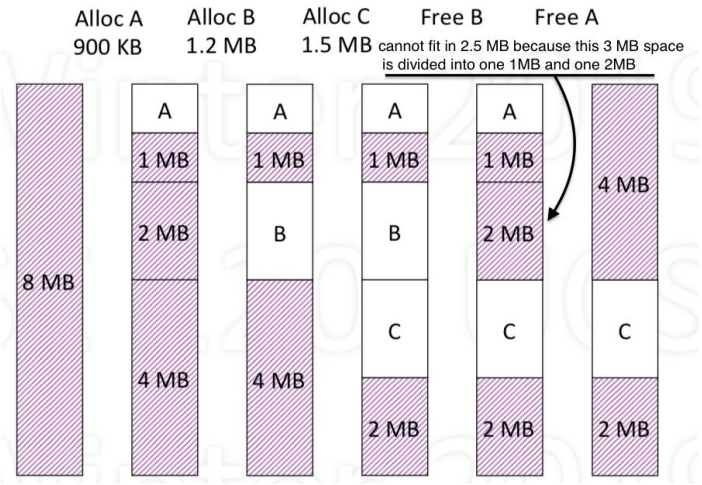
- **How efficient is the memory usage**
    - K = 1, f = ⅓, waste 33% of memory. The bigger k, the more waste space
    - Average hole size has to be small **relative** to block size
    - Limits:

    - In general, **f increases with increasing k**: the larger the average hole size to average block size, the larger is the fraction of wasted memory.
    - Alternatively, **f decreases with decreasing k**: the smaller average hole size is to average block size, the smaller the fraction of wasted memory.

- Pre-sized holes
    - All holes of the same fixed size
    - Pros:
        - All holes the same, easy allocation, no tiny fragmentation.
        - Any block fit in any hole
    - Cons:
        - Inflexible, a size too small.
        - Even with a variety of sizes (small, medium, large, ..), more flexible but more complex
    - **NOT** adaptable, **internal fragmentation**

- Buddy System
    - Holes partition into power-of-2 size chunks

```
Find chunk larger than r (else return failure)
while (r ≤ sizeof(chunk) / 2)
      Divide chunk into 2 buddies (each 1/2 size)
Allocate the chunks
```

    - **Allocate:** Keep dividing up the buddy until fit the requested memory, as much as possible
    - 

    - **Free:** When **both** buddies are freed, the memory chunks are collapsed.
    - Data structure for buddy system:
        - Binary tree
            - When allocate, find the best fit
            - When free, check if the buddy is free (leaf node is freed), if so,

merge.
- Fast allocation mechanism
  - Faster than first fit: searching for pre-sized holes
  - Less wasted memory
  - More internal fragmentation

LEC 9 - Logical Memory *(Feb 11)*

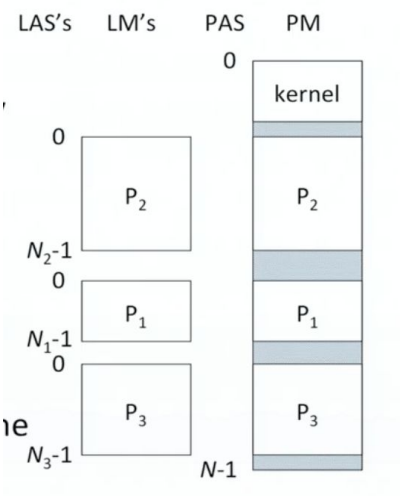
- Definition
  - A process's memory
  - As viewed (reference) by a process
  - Allocated **without** regard to physical memory

- Aspects

Addressing	Compiler generate memory references Unknown where process will be located
Protection	Modifying another process's memory
Space	More processes there are, less memory each one can have

- Address Space
  - Set of addresses for memory
  - Usually linear from 0 to N - 1 (size N)

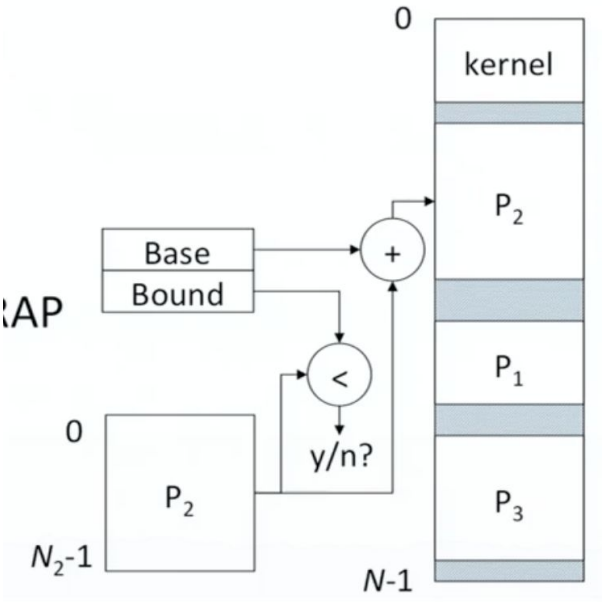
- **Two Types of Address Space**:
  - Physical Address Space:
    - 0 to N - 1, N = size
    - **Kernel** will be put at **location 0** in **physical memory**
  - Logical Address Space:
    - Assume separate memory **starting** at 0
    - Compiler generated
    - **Independent** of location in physical memory



- **Converting logical to physical**
  - Software at load time
    - Look at the object code and then offset by a certain amount
    - Doesn't work because the loader couldn't find all addresses: some languages can create address (pointer), only determined at runtime
    - Rarely used
  - Hardware at access time
    - Constant struggle between OS and hardware
      - OS designer → hard work → hardware
      - Hardware → concern of cost → refuse
      - OS: we cannot do it, or do it with highly cost

- Hardware for Logical Addressing
  - **Base register** with start address
  - To translate logical address, add base
  - Achieves relocation: to move process by changing base

- Protection
  - **Bound register** works with **base register**: avoid accessing memory that is not supposed to access.
  - Check if address < bound: yes add to base; no invalid address TRAP
  - Achieves protection: kernel is protected



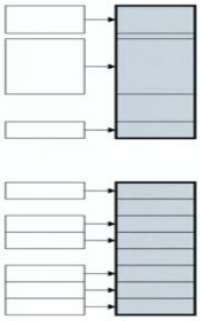
- Base/Bound register: one set each machine

- Memory Registers as Part of Context
  - On **context switch**
    - Load base/bound registers (one set each machine, shared) for selected process
    - Only kernel does loading of these registers
  - **Pro**:
    - Allows each process to be separately located
    - Protects each process from all others

- Fitting Process into Memory
  - Process Address Space
    - Text: execute only, fixed size
    - Data: variables (static, heap): read/write, variable size, dynamic allocation
    - Stack: activation records (auto): read/write, variable size, automatic growth
  - The logical memory of each process is huge and can't be fit in any of the holes in physical memory
    - Even if successfully → inefficient because space must be allocated for potential growth
  - Solution: **break process into pieces**: distribute into available holes

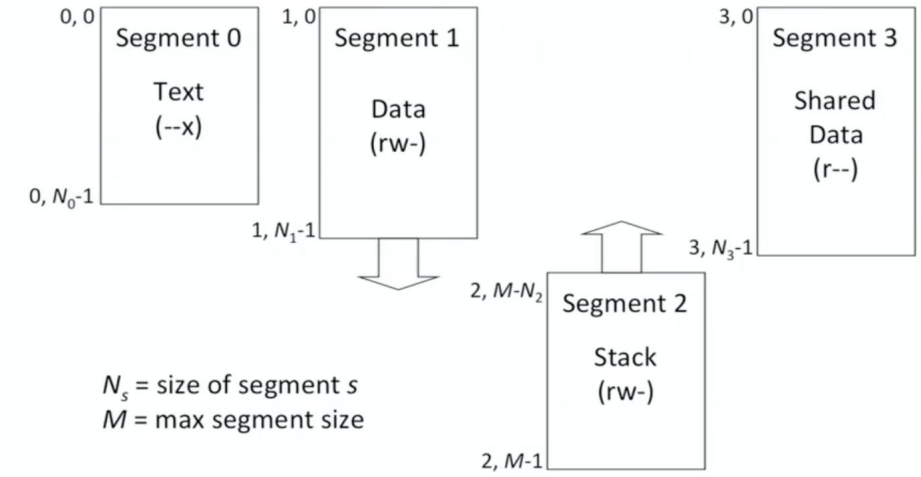
- **Two Approaches** to break process memory into pieces

Segmented	Partition into segments Segments can be **arbitrary** sizes
Paged	Breaks up into pieces with **same** size

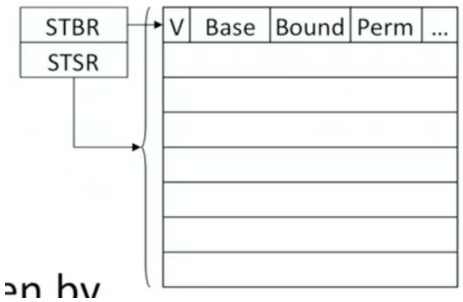


- **Segmented Address Space**

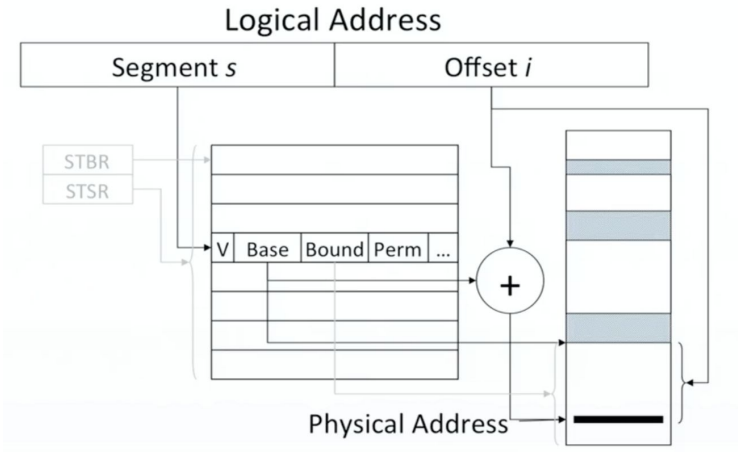
  - Address space is a set of segments
  - Segments: a linearly addressed memory, typically contains **logically related** information. (program code, data, stack, etc.)
  - Each segment has an identifier s and a size N (MAX segments)
  - Logical addresses of the **form (s, i)**
    - generated by compiler: segment number s and offset (s, i).



  - **Translation:**
    - Segment (translation) table ST:
      - **One** per process
      - **Static data** area (tables for each process)
    - Entries: valid bits, base for segment location, bound for segment size and permissions.
    -



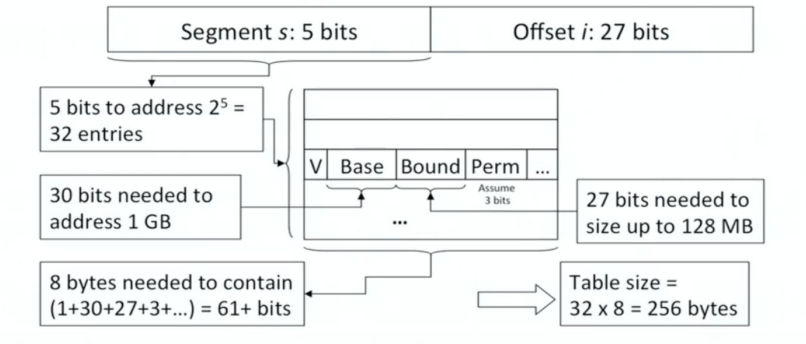
- Physical address = ST(s) + i
    - Check the size of segment identifier
    - Check the valid bit of the segment table entry
    - Check i is within bound
    - Check operations permission
    - Finally access physical address by base + offset
    - *Done in hardware*



- **Sizing the segment table**

Segment Identifier has n bits	Max size of the segmentation table = 2^n
Base	Number of bits to address physical memory
Bound	Number of bits for the max segment size
Offset	Number of bits for max segment size

    - E.g. physical memory of 1GB and 32 bit logical memory



- 2^{30} addresses 1 GB
- Round each segment table entry from 61 bits to 64 bits → **8 bytes**
- Total table size = 8 bytes * 32 entries = **256 bytes**

- Compiler will determine the size of segmentation table

- **Pros:**
  - each segment can be located independently (different base)
  - Separately protected (data, stack, etc)
  - grown/shrunk independently (change the bound value)
  - Segments can be shared by process (share the same text, etc.)
- **Cons**:
  - Variable size allocation
  - Difficult to find holes in physical memory
  - External fragmentation

- **<u>Paged Address Space</u>**

  - Logical memory broken into fixed size pages
    - Linear sequence of **pages**
  - Physical memory
    - Linear sequence of **frames**
  - Size of frames = size of pages
    - A page fits exactly fit into a frame
    - Frame as a physical unit of information
    - **No bound** value needed

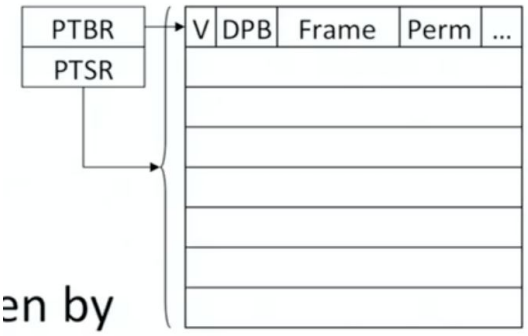
  - **Translation**

    - Forms and size between logical and physical addressing

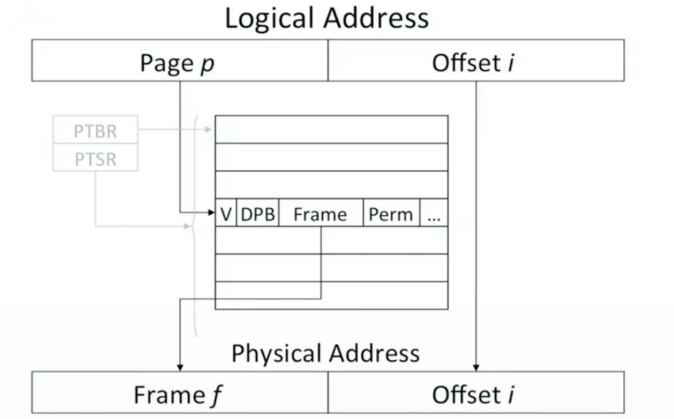
	Form	Size
Logical Addresses	(p, i), where p is page number and i is the offset within the	Max number of pages * page size

	page	
Physical Addresses	(f, i), where f is the frame number and i is the offset within frame	Max number of frames * frame size

- Page table:
  - **One** per process, in **kernel's data** area
  - Table entries: valid bit, demand paging bits (reference bit, modified bit), frame as page location
  - Register: **base** register, **size** register



- Given a page number, translate to frame number
  - **F = PT(p),** then concatenate f and i: PT(p) || i
  - Check if page p is within PTSR (**size** register - page table size)
  - Check page p's valid bit
  - Check the permissions of the operation
  - Concatenate the frame with the offset.

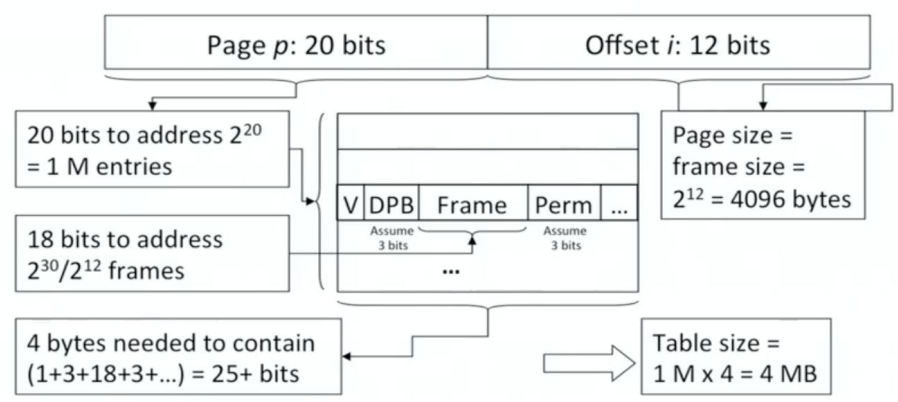


- **Sizing the page table**

The page identifier has	Max size of the table = 2^n

n bit	
Frame	Number of bits needed to address physical memory in **units of frames**
Offset	Number of bits n specifies page size

- E.g. given 32 bits logical, 1GB physical memory (max)

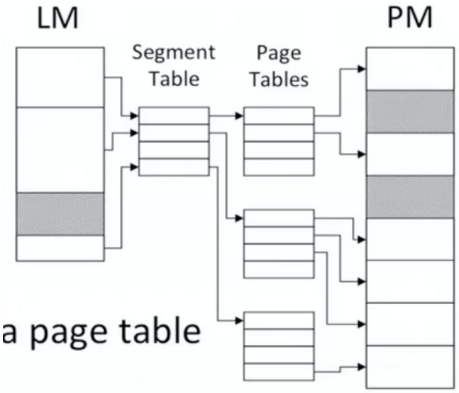


  - 25 bits is rounded to 32 bits → **4 bytes**

- **Comparison** between **Page** and **Segmentation**
  - Segmentation: *logical* unit of information
    - Sized to fit any content (content is homogeneous)
    - Makes sense to share (e.g. code and data)
    - Protected based on content
  - Page: *physical* unit of information
    - Simple memory management

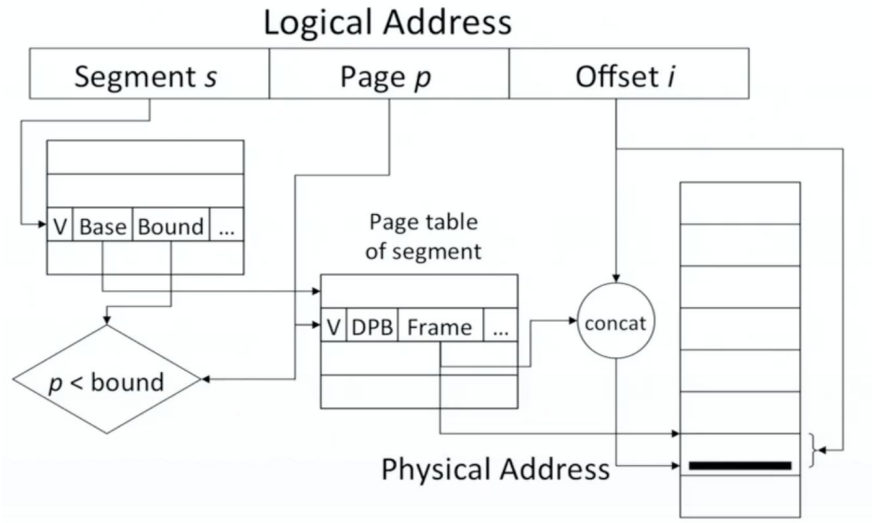
- **Combination** of Page and Segmentation

  - Logical memory composed of segments, each segment composed of pages



- Segment table: maps each segment to a page table
- Page table: map each page to physical page frames

- **Cost** of Multiple translation

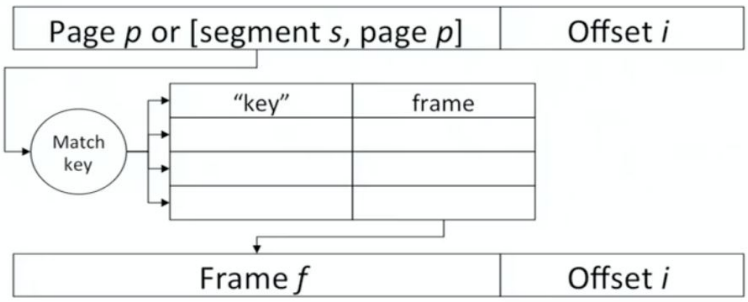


  - Each lookup costs another memory reference
  - For each reference, additional references required, slow down the machine by a **factor of 2** or more

- **Solution**: take advantage of **locality of reference**, but don't know which page until accessed.
  - **Most programs access small number of pages**
  - May jump around memory section, but most times access memory close in space
  - Keep translation in high-speed memory

(Feb 13)

- **T**ranslation **L**ook-aside **B**uffer



  - **Fast memory** (hardware) keeps most recent translations
  - Key matches, get frame number, without doing normal memory access; else wait for normal translation

- Bypass the normal mechanisms

- **Cost**: e.g. speed of memory ~100 nsec, speed of TLB ~ 5 nsec, hit ratio by TLB ~ 99%
    - Average without address translation: 100 nsec
    - Average address translation: 106 nsec
        - TLB miss: 200 nsec (100% slowdown)
        - TLB hit: 105 nsec (5% slowdown)

- **Design** Issues
    - The larger the TLB → Higher hit rate, slower response, greater expense.
    - Major effect on performance
        - Flushed: on context switch, the machine gets slower; Then gradually becomes faster
        - Tag: tagging entries with PIDs

- Summary
    - Logical memory
        - Memory that structured in a way that makes sense
        - Organize the memory in a logical way
    - Addressing Problem, Protection Problem
    - Organization: **segmented**, **paged**
    - Logical-to-physical address translation
    - High cost of translation: **locality**, **TLB**

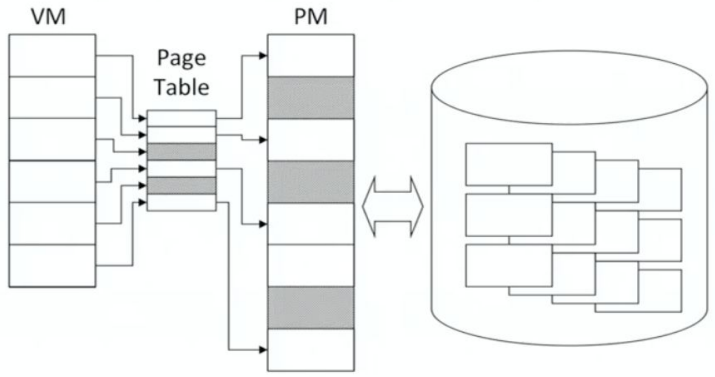
LEC 10 - Virtual Memory

- Logical Memory
    - A logically organized memory (segment and page): partition memory for convenient allocation & reorganizing memory for convenient usage.
        - Relocation via address translation (TLB) (base and bound based on segmentation or page).
        - Protection via matching operations with objects [embed access right]
    - Implications
        - Not all pieces need to be in memory. Need **only piece being referenced**, while others can be on the disk. Bring pieces in only when needed.
        - **Illusion**: there is much more memory
        - **Mechanism**: identify whether a piece is in memory, bring in a piece, relocation (address translation).

- **Virtual Memory**

- "Virtual" work as if it's real
- Logical memory becomes virtual memory: **still logical** (separate organization from physical) and virtual (seem to exist, regardless of how).
- Virtual:
  - Keep only portion of logical memory in physical
  - Rest is kept on disk (larger, slower, cheaper)
  - Unit of memory is segment or page (or both)
- Logical address space → **virtual address** space

- Virtual Memory Based on Paging



  - All pages in virtual memory reside on disk and some reside in physical memory
  - Contents of page table entry

Valid	Ref	Mod	Frame number	Prot: rwx

- Address translation
  - Index page table with page number
  - If the valid bit is off, **page fault**: trap into kernel
    - Find page on disk (kept in kernel data structure)
    - Read it into free frame (may need to make room by page replacement)
    - Record frame number in page table entry
    - Set the ref/mod = 0 and set valid bit to 1
  - Retry instructions

- Faults Under Segmentation/Paging
  - Virtual address: <segment s, page p, offset i>
  - Use s to index segment table (to get page table)
    - May get a **segment fault**
    - Bring in the page table from the disk

- Convert the page number into frame:
    - Check bound (ls p < bound): may get a **segmentation violation** (unrecoverable)
- Use p to index into page table (to get frame f)
    - May get a **page fault**
    - Bring in the frame from the disk to memory
- Eventually get the physical address by concatenating f and offset i.

- Page Faults are **Expensive**
    - Disk: 5 - 6 magnitude slower (1-10 day) than RAM (1s).
    - Very expensive, but if very rate tolerable.
    - Much more expensive than factor of 2 memory reference.
    - *Note:* even page table (~MB per segment, cost very high) in the RAM, can still get page fault.

- **Principle of Locality**
    - Not all pieces referenced uniformly over time
        - Make sure **most referenced pieces** in memory
        - Thrashing: constant fetching of pieces
    - References **cluster in time/space**
        - Will be to the same or neighboring area
        - Allows prediction based on past
    - *Page fault is inevitable

- Page Replacement Policy
    - Goal: remove page not in locality reference
    - The better we do page replacement, the lower probability we get page fault
    - Cheapest way possible: least additional hardware and least software overhead

- **Replacement Algorithm**

    - Number of page faults is at least the number of pages.

    - **FIFO**: select the page that is oldest to remove

Reference string	2	3	2	1	5	2	4	5	3	2	5	2
FIFO →	2*	2	2	2	5*	5	5	5	3*	3	3	3
		3*	3	3	3	2*	2	2	2	2	5*	5
9 faults				1*	1	1	4*	4	4	4	4	2*

        - Discriminate against old pages
        - **Pros**: simple to implement (keep pointer to next frame after last loaded)
        - **Cons**: old doesn't imply useless

- **OPT**: select a page to be used furthest in future

Reference string	2	3	2	1	5	2	4	5	3	2	5	2
OPT	2*	2	2	2	2	2	4*	4	4	2*	2	2
→		3*	3	3	3	3	3	3	3	3	3	3
6 faults				1*	5*	5	5	5	5	5	5	5

  - **Pros:** optimal, used as benchmark
  - **Cons:** requires predicting the future, NOT REALISTIC

- **L**east **R**ecently **U**sed: select page that was least recently used

Reference string	2	3	2	1	5	2	4	5	3	2	5	2
LRU →	2*	2	2	2	2	2	2	2	3*	3	3	3
		3*	3	3	5*	5	5	5	5	5	5	5
7 faults				1*	1	1	4*	4	4	2*	2	2

  - **Pros:** works well, use the past to predict the future
  - **Cons**: locality assumption, requires hardware support (complex), expensive

- Summary: OPT ≥ LRU (assuming locality) ≥ FIFO

(Feb 20)

- **Clock Algorithm**

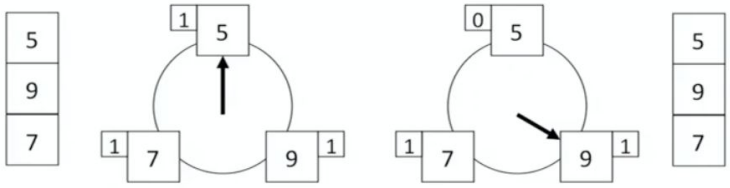
  - Select page that is **old** and **not recently used**: approximates the *LRU*
  - Hardware support: reference bit
    - Associated with each frame is a **reference bit**, stored in page table entry
  - When frame is filled with page, **OS set the bit to 0**; When frame is accessed (by executing a hardware instruction), **hardware turn the bit to 1**.
  - Clock hand points to a frame (page) in physical memory. If the valid bit in the page table is fault, then the kernel runs and finds the page is not in the memory. Then the kernel looks at the clock data structure. The kernel load the page and set the reference bit to 0. Finally the process runs and the hardware sets the bit to 1.

  - Page fault: find frame

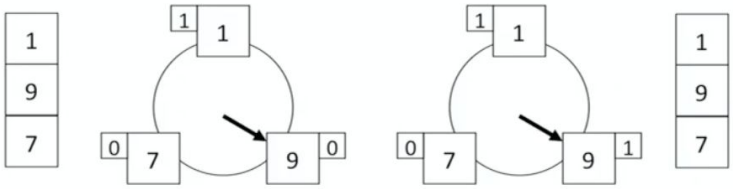
```
While (not found frame) {
```

```
      If (ref == 0)
            select frame;
            advance clock hand;
            break;
      Else:
            ref = 0; advance clock hand;
}
```

- If all pages have been accessed, then find the oldest page
- Ir reference bit is 0, the page is not recently accessed



Ref string: 5 9 7 1 9 5 9



- When accessing page 9, the kernel is not invoked because page 9 is in the memory. No clock hand movement. No page fault, hardware turn the bit to 1

- Performance
  - All memory operations are **atomic**
  - The clock algorithm is done by the kernel and kernel make sure it's atomic.
  - Swape about page is a disk access based on the modified page.
  - Close to the optimal solution, but many have other sophistication. (some operating systems have two hands, keep pool of pages)

- **Resident Set Management**

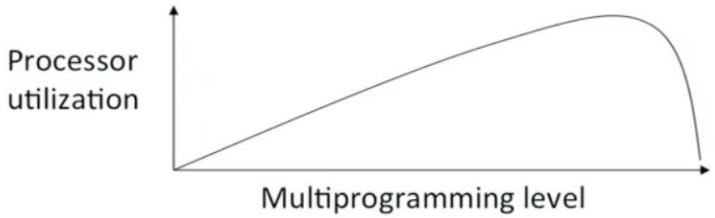
  - Resident set: **process's pages in physical memory: size, which, provider**

  - Local: limit frame selection to requesting process
    - Only allows you to replace from the **same** process's memory
    - **Pros:** Isolates effects of page behavior on process

    - **Cons:** Inefficient: some processes **have unused frames**

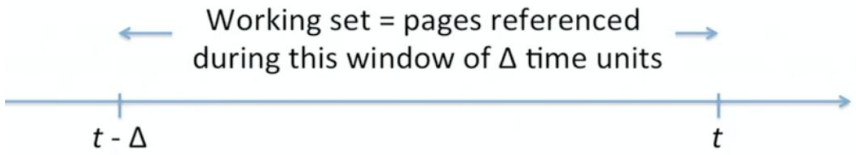
- Global: select any frame (from any process)
    - Replace any frame that is in the memory
    - **Pros:** *Clock Algorithm* is a global policy
    - **Cons:** No isolation: process can negative affect another

- Multi-programming level and process utilization
    - Multiprogramming level: non-empty resident sets)
    - Goal: increase multiprogramming level
    - Process utilization:



        - As multiprogramming level initially increases, the processor utilization increases (never reach 100% because of other things like IO)
        - **Thrashing**: too many processes are in the physical memory. **Chain of kicking out of pages.**

- **Working Set Model** (Denning's)



    - Working set as **pages referenced during last delta** (process time)
    - Process given some amount of memory
    - Add/remove pages according to W(t, delta)
    - Either the entire working set in memory, or **all the process's pages swap out**, put all the pages in disk. Then you cannot run this process again. Then at some point we may run this process again.
    - Allows starvation so needs some scheduling algorithm

- Comparison between Working Set and Clock Algorithm
    - Working Set
        - **Local** replacement policy: process's page fault behavior doesn't affect others
        - Difficult to implement

            - timestamp pages, determine if timestamp older than t - delta;
            - Delta is experimentally determined
        - Starvation for process with large memory
            - Memory scheduling algorithm as part of CPU scheduling algorithm.
    - Clock algorithm
        - **Global** policy
        - Simple, easy to implement
        - Good approximation of Least Recently Used

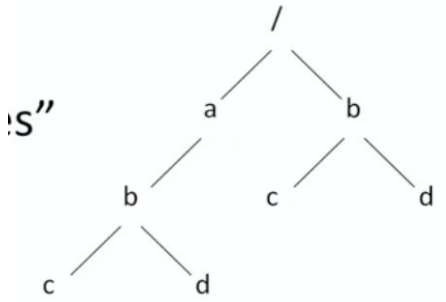
- Summary
    - Virtual memory
        - Logical memory: some in physical, all in secondary
            - E.g. store money in the bank; not storing actual amount but when needed we could have it
            - E.g. kernel makes process think they have huge memory, but indeed give them what they need
        - Effective size of disk, effective speed of RAM
        - Efficient because of locality
    - Performance of page replacement:
        - OPT ≥ LRU ≥ Clock ≥ FIFO
    - Goal: keep working set in memory: if working set cannot be resident, swap out.

LEC 11 - File System

- Definition
    - **File**: logical unit of storage, container of data; accessed by <name, region within file>
    - **File System**: a structured collection of files; <access control, name space (structure of naming), persistent storage>. **INSIDE KERNEL**

- Abstraction
    - **Repository of objects** (data, program for system, users; referenced by name, to be read/written)
        - **All objects** <accessed by name, read/written, protected, shared, locked, etc>: IO devices like disk, keyboard, display and processes: memory.
        - Object as something encapsulated data, via function, interface, etc.
    - Persistent: remains “forever”
    - Large: “unlimited” size
    - Sharing: controlled access
    - Security: protecting information

- Hierarchical File Name Space
    - Name space is organized as a tree.
    - Ex: unix pathnames: absolute and relative. Allow links.



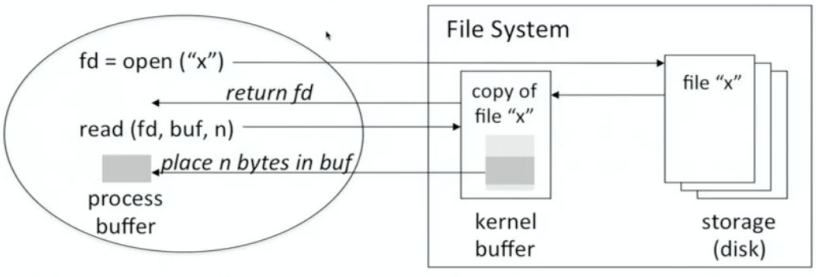
- File Attributes
    - Type, Times, Sizes, Access Control
    - Container with certain characteristics that the system cares about
    - Optimize the usage of file

(Feb 25)

- File operations
    - Creation: create, delete
    - Prepare for access: open, close, memory map
    - Access: read, write
    - Search: move to location
    - Attributes: get, set (e.g. permissions)
    - Mutual exclusion: lock, unlock
    - Name management: rename

- **Buffer**: temporary holding area

- **Read/write Model**



    - File descriptor: fd = open(fname, usage)
        - Small integer each corresponding to each open
        - Handed out by the kernel
    - Read: number read # bytes nr = read (fd, buf, size)

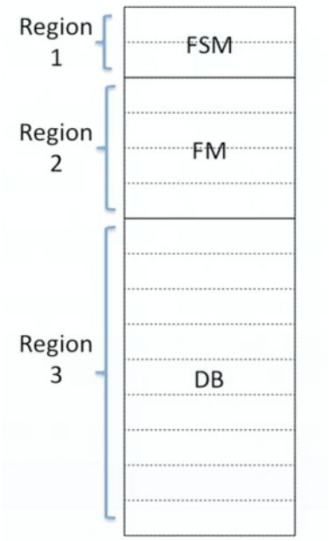
- Write: number write # bytes nw = write (fd, buf, size)
- Closure: close(fd)
- Imposing **open/close**: make copy from disk to **kernel's buffer** so that read and write operates on the kernel buffer temporarily, a lot **faster**.
- The kernel owns the file system. Read/write then load file from disk to buffer.

- **Memory-mapped Model**
  - Map file into process's address space
    - Mmap (addr, n, ... , fd, …) / addr mmap (NULL, n, …, fd, ..)
    - X = addr[5] (virtual address space)
    - Strcpy = (addr, "hello")
  - Can only map a small portion
  - Modify the address automatically modifies the file (OS done)
  - **Pros**: as soon as the address modified, the file get updated (unlike read/write)
  - **Issues**: efficient for multiple processes sharing memory <every process see the update immediately>, but how is the file actually updated?

- **Access Control**
  - How files are shared, to varying degrees
  - Access control: users, operations, user interface must be simple and intuitive
  - E.g. r/w/x permissions for owner, group, everyone + more complicated case
  - 9 bits can specify the access rights

- **File System Implementation: Goals**
  - Archival storage: keep forever, including previous versions
  - Support various storage technologies (invisible to the users)
  - Balance (can't max all at one time): performance, reliability, security

- Storage Abstraction
  - Hide complexity of device: **machine independent**
    - Model as **array of blocks** of data
    - Randomly addressable by block number
    - Typical block size: 1 KB
  - Simple interface: read(block_num, mem_addr), write(block_num, mem_addr)
  - Sub-divide the structure (three major regions),



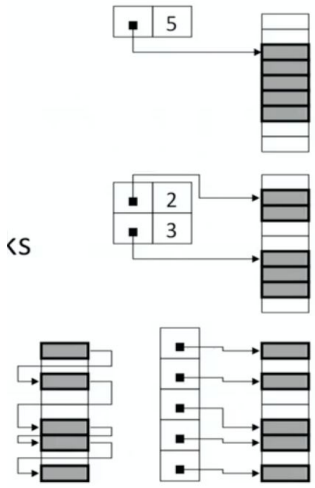
- **File System Metadata:** information about the file system (Meta: about):
    - Size: file in uses, data blocks in use, free entries
    - Free lists (or bitmaps): file control blocks, data block

- **File Metadata**: information about file

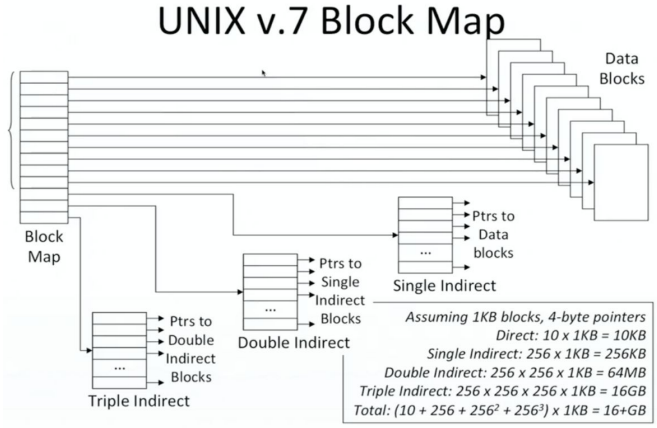
    - On the storage device but may cached in memory.
    - Array of file control blocks: attributes (size, permission, etc.), references to data blocks (disk block map) where to find the data.
    - **Many** file control blocks may fit in single storage block.
    - Has to be preallocated
    - E.g. file control block
        - number 88 as index in file control block array
        - Size: 4096 bytes
        - Permission rw-r--r--
        - Data block: set of indexes into storage array
        - File name stored in *separate way*

    - **Keeping track of allocated blocks**
        - *Contiguous block:* single sequence of blocks (start + size of the file)
            - Store a file of a certain size (must find a contiguous space)
        - *Non-contiguous blocks:* blocks individually named (has to be built in many pointers)
            - More flexibility
            - More space
        - *Alternative*: keep the pointers in the data block; but has to go through all intermediate blocks to visit a certain region.

- *Extents:* groups of contiguous blocks (contains pointer to each other)



- E.g. UNIX v.7 Block Map **inside** File Control Block
    - **Engineering of UNIX**: concisely fit mechanism; optimized for common case, without limiting the ability for rare tasks.
    - Array of pointers to data blocks with **13** pointers
        - 10 direct: references 10 data blocks
        - 1 singly indirect (n), doubly-indirect (n^2), triply-indirect (n^3)
            - Blocks that filled with pointers to data blocks
    - Illustration



    - *Note:* 256 4-byte pointers will fit in 1KB block: 1024 / 4 = 256.
    - Most files in file systems are small, but there are large files.
    - Accessing a disk file is very expensive; accessing small files is fast (common case); the system still supports very big files (rarely uncommon accessing large file is slow).

- **Keeping track of Free Blocks**
    - *Free block map***:** compact if lots of free regions of space; array of pairs (extents)

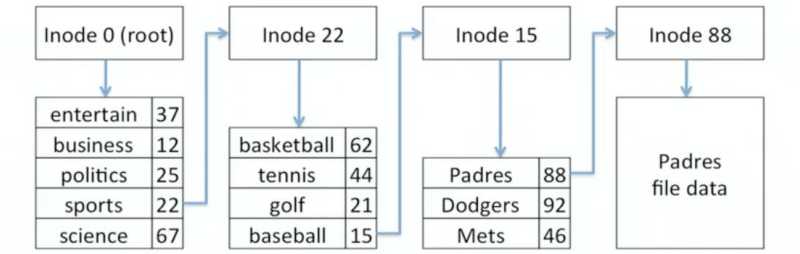
- *Doubly linked list:* easy to keep ordered due to fast inserts and deletes, can be grown and shrink.
- *Bitmap*: fixed size regardless of fragmentation

- **File Name** to File Control Block

  - User access files using file names
  - Translate from “filename” to file control block number index
  - File control block as internal node (i-node)

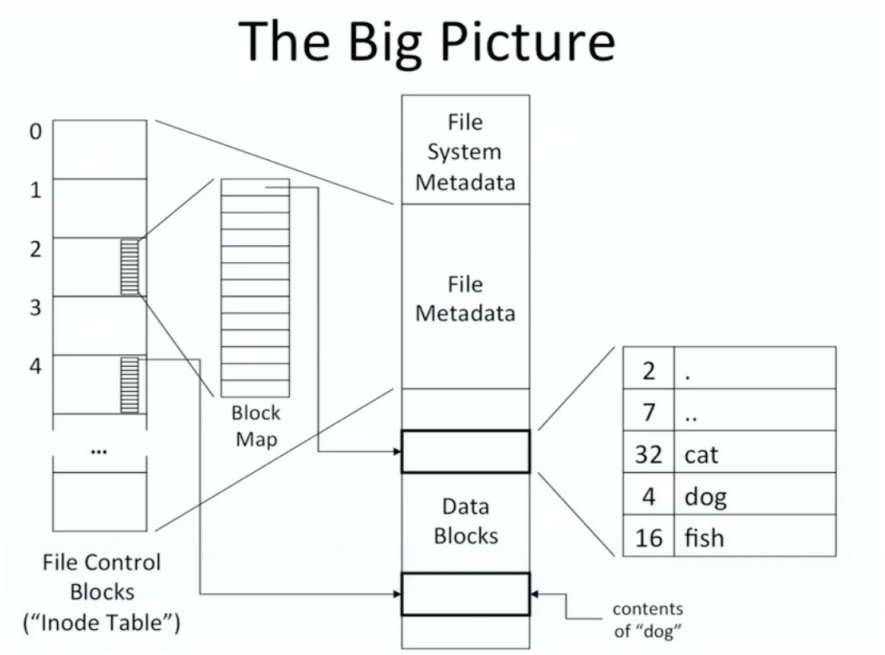
  - **Directories:** mapping of a part of filename into a file control block number (i-node, internal node)
    - Each entry: branch name, i-node number
    - Translate symbolized name to i-node number.

  - Example of *Parsing Names* in UNIX



    - Root directory by default is **0**
    - Each node contains data blocks array (13 pointers) [every directory is itself a file)
    - Get file by translating every part of the path
    - *If store name in the data control block, then name’s length is not known. → pre-allocated space. The kernel can’t know about the size*.

- The Big Picture



- **Physical Device Storage**:

    - **Disk**: Magnetic Drive
        - A set of platters covered with material that can be magnetized, with either north or south polarity (0 or 1).
        - Disk head with read/write heads (floating), disk arm can go in and out
        - Sectors 512 bytes
        - Platter rotating

    - Definition
        - Sector: a path of the head
        - Track: cycle of bytes
        - Cylinder: track of the same positions across platters

    - Why
        - Persistent: the information doesn't reply on continuous electricity
        - Random Access: (quartz) don't need to wait all that much; move head
        - Cheap: 1992 → entire research team for 1TB storage

    - File System Performance: disk access very **SLOW**
        - Disk accesses are time expensive: 5 - 20 msec!
            - Rotational latency: 2 - 6 msec (5400-15000 RPM)
            - Seek time: 3 -13 msec (move the disk head)
            - Transfer rate: 100+ MB/sec (read and transfer to the computer's

memory)
- Reduce accesses by
  - Reading multiple blocks in one access (read ahead)
  - Maintaining a block cache (in physical memory → make a copy)
- Cluster related blocks to reduce seek time

- **Solid State Drives**

  - NAND-based flash memory, non-volatile
  - Unaffected by shock, magnetic fields, no noise
  - Limited number of writes: wears out with age (number so huge, rarely reached)

  - Performance

Attribute	Solid State Drive	Hard Disk Drive	SDD vs HDD
Access time	0.1 msec	10 msec	** 100x **
Throughput	500 MB/sec	100 MB/sec	5x
Power	< 2 watts	5-6 watts	1/2x
Capacity	64 GB – 1 TB	500 GB – 8 TB	1/10x
Cost	$50-100 for 100 GB	$5-10 for 100 GB	10x

- **Performance**: caching (*software cache, physical cache)*
  - Attempt to caching everything, concerned about the cost of disk access
  - Data blocks of files
  - File system metadata (keep in memory)
  - File metadata
    - Currently active files
    - Recently used
  - Block maps
  - File names
    - Name to file metadata translations (keep name cache in the memory)

- **Performance**: clustering
  - Strategically place data on the disk
  - Blocks that exhibit locality of reference
    - Director and files within that directory
    - The i-nodes of the directory and files (physically stored the file control and file data blocks) close to each other.
  - Minimize the disk head movement

- **Performance**: block size

- **Larger the block, the better the throughput**
- **The smaller the block, the less wasted space**
- Trends: disk density is increasing faster than disk speed
  - Make disk blocks larger: 1kb → 8 kb, 64kb
- Sacrifice the space for time
- **#1 problem**: disk is the major issue for speed
  - Reformat the file system with a larger block size

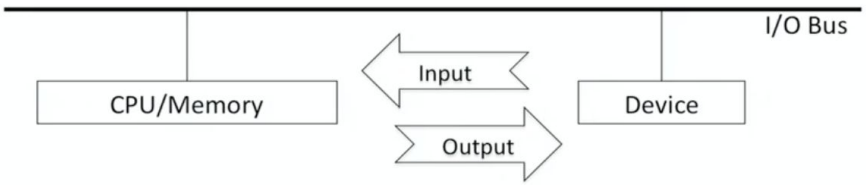
- **Reliability:** consistency
  - Buffer cache reduces disk accesses. But if system crashes, block modification lost
  - To improve file system consistency
    - Write out **modified blocks** (prioritize)
    - Write out critical blocks (write-through)
  - Critical blocks: file system **meta-data**
    - Directories, i-nodes, free block lists
  - Every 30s a process wake up to perform disk writes
  - Sacrifice the time for consistency

- **Reliability:** journaling
  - **Log** of file (or file system) updates
  - For every update, create log entry
  - Write log entry out to disk (part of journa**l)**
    - If there is a system crash: look at journal entries, check if mods properly reflected in file system, update appropriately
    - Have two separate disks, one keeping log of file system (append-only disk, disk head keep at the same location)
    - Recreate anything before the crash by reading the log

- Summary
  - File System: abstraction for object repository
    - Static and dynamic objects
  - Name space: organized hierarchically
  - Access models: read/write, memory-mapped
  - Access control: read, write, sharing
  - Implementation structure: FSM, FM, data
  - Performance (caching, clustering), reliability

LEC 12 - Input/Output System

- I/O = Input/Output



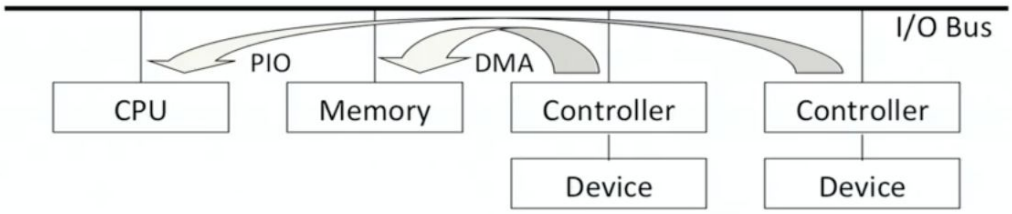
  - Input from attached device to CPU/memory
  - Output from CPU/memory to device
  - *Note*: CPU can only access the memory
  - Synchronization and transferring data *how we synchronize between devices*

- Issues
  - Many different types of I/O devices (in the future)
  - Wide ranges: speed, operation, data transfer units
  - How is synchronization achieved

- Separated into two portions: device dependent and device independent

- Questions:
  - How does a process initiate I/O
  - How is synchronization achieved
  - How is data transferred

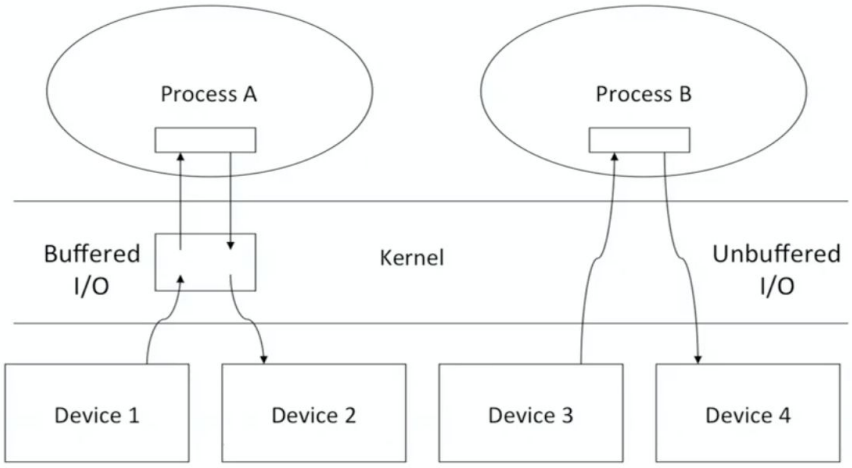
- I/O Hardware



  - Controller: solid state electronics
    - How to control the devices
    - Interface to the I/O bus
    - Have registers where different values lead to different functions
  - Process and device (controller) communicate via
    - I/O instruction: from CPU registers into controller
    - memory instructions (memory mapped): view device and memory
  - Data Transfer:
    - **P**rogrammed **I/O**
      - Move data from CPU's register to controller registers
      - Polling, not interrupt (the interrupt is expensive)
      - Move from this location in memory to disk In small units
    - **DMA:** transfer data between controller and memory;

        - Move certain block memory to controller
        - while that, CPU yields to another process; then device generates interrupt and kernel interrupt handler → wake up a process
    - Synchronization: polling vs. interrupts
        - Polling: CPU constant loop to check the status (controller) PIO: check if the bits.
        - Interrupt: once the device is done, then interrupt the process to indicate DMA is completed. Wake up a process.
        - Time for the device to get data; bit of device to see the bit wrote or not

- Buffered and Unbuffered I/O
    - Example: reading from one device to another: read from camera and write to another



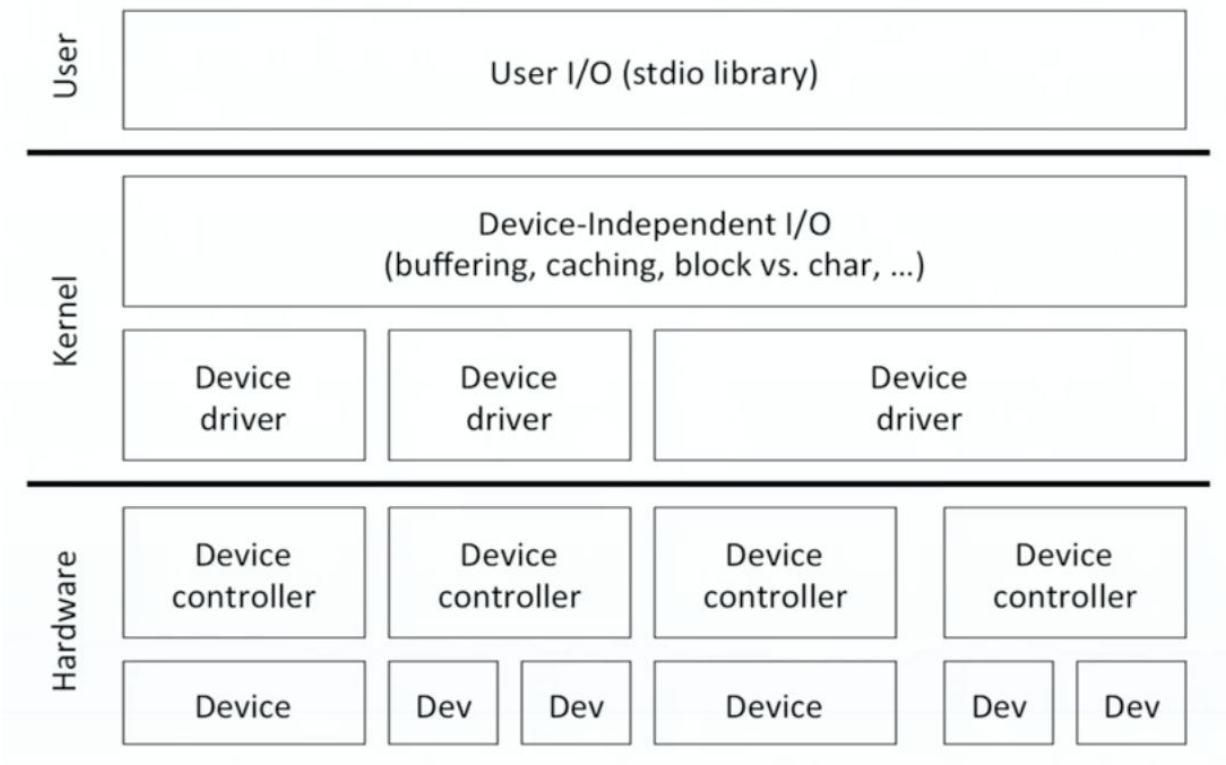
    - **Kernel buffer:** temporarily storage device
        - Take advantage for kernel to share data across process
        - **Pro:**
            - Maybe we want to kick out the process from memory (swap out the process), because the I/O is taking a long time.
            - Keep around for other processes (cache)
            - Decouple the speed of writing/reading from kernel and device
    - **Unbuffered**: bypass kernel, go directing memory ←> device
        - Moving **a lot of data** at high rate → very slow because of copying
        - Connect diskes directly together (eliminate by not using the process's memory at all)
        - Wasted by the coping a lot of data.
        - By pass the kernel
    - **Pros and Cons Kernel Buffering**
        - **Pros:**
            - Pages containing buffer are paged out → OK

- Entire process is swapped out → OK
- Pin page (take a page from a process and don't allow it to swap out. But too many processes' pining prevents processes from running. [resvering]

- **Cons:**
  - Memory copy is expensive [the more the worse]
  - Effect on caches: whenever a memory is accessed, *physical caches* are being filled out; if filling out caches, we may be kicking out information that may be useful.
    - E.g. Camera data copying blowing away useful information
  - *Physical caches:* actual physical caches to make read/write faster. → kill the speed.

(March 4)

- I/O System Structure: Layered

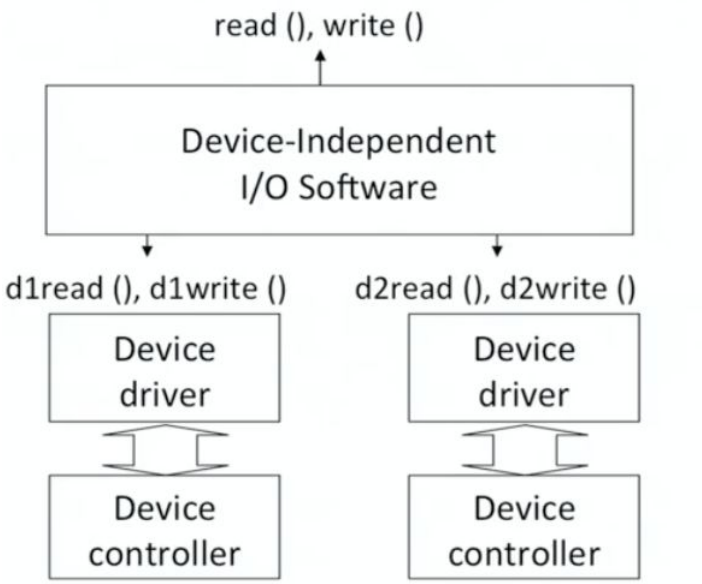


  - A single controller for multiple devices
  - Stdio: code for help for system io

- Device Dependent: **Device Drivers**
  - Encapsulates device-dependent code (provided by the disk manufacturer)
    - Contains device-specific register reads/writes
    - Designed by disk provider, installed as part of the kernel

- Implements a standard interface
    - Open(), close(), read(), write()
        - dread() device read as **device-specific code** for read
    - When called, translate into device-specific instructions
- Interrupt handlers (part of device driver)
    - Device to communicate with the software
    - Devcie's finishing the I/O is encoded as an interrupt, the interrupt handler updates data structures; wakes up processes
- Wrote by the company provided the hardware
    - Allowing third-party codes to run in the kernel

- Device Independent I/O

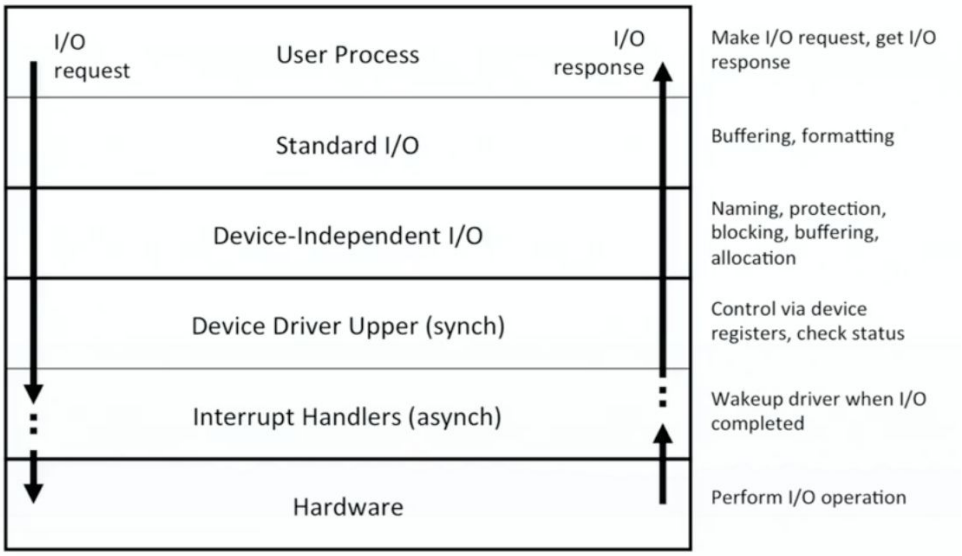


    - **File system as part of Device independent I/O**
    - Uniform interfacing for device drivers
    - Naming,protection (part of file system → device independent I/O)
    - Uniform block size (up to the device driver to cover the size)
    - Buffering, caching
    - Storage allocation
    - Locking
    - Error handling
    - **Reason**: has to rely on third party code, but limit the fraction of device driver's codes, even though the device driver can include these codes.

- User Space I/O
    - User call read(), device-independent I/O calls device drivers corresponding to the file descriptor.
    - Convenient interface:

    - **printf() vs. write()**: not part of the kernel
    - Printf as user code, ultimately may call write
    - Only one way to “write” to a device
  - **User-Level buffering**
    - Standard I/O library may buffer the characters, until it collects enough characters and then do the system to write. → Save a lot of work
    - When read one character, reading the entire 1kb and store into the buffer → routine check the buffer before actually accessing the disk
  - Standard I/O: improves **performance** and provides **convenience**

- **Overall Operation**



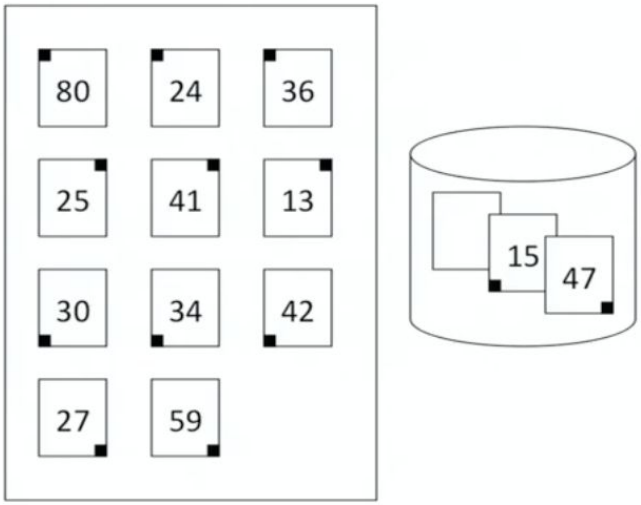
  - Standard I/O → Device-independent: trap into the kernel
  - Device driver upper: synchronized with processes;
  - Interrupt Handlers: asynchronized; happen and then interrupt handler run, wake up and return to the standard I/O

- Example: UNIX I/O Model
  - Uses file system interface
  - Stdio.h: C standard I/O library, compiler adds in the codes
  - Block devices (e.g. disk)
    - Fixed size blocked: can only send fixed size block to disk
    - Randomly addressable: index into array, as opposed to sequential access
    - Uses buffer cache: just by declaring as the block devices
  - Character devices
    - Variable sequence of bytes
    - For non-block devices
    - Cannot use the system buffer cache
- Standard I/O

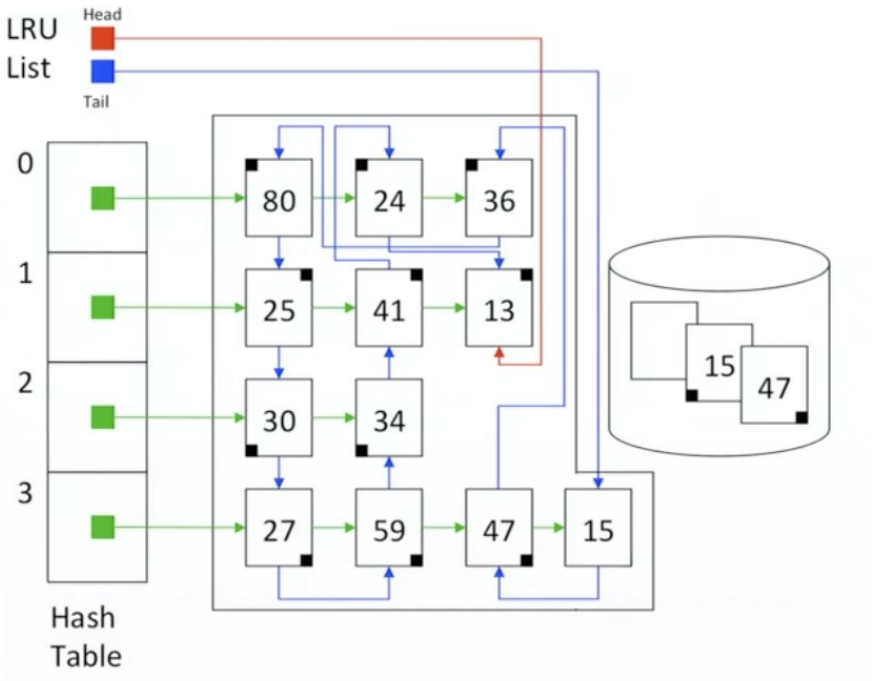
  - Interfaces: read, write, but can also uses (fd, cmb, buff) for device dependent instructions.
  - Private buffer in user space
  - Minimize the number of I/O system calls

- Read / Write + Open / Closed
  - Disk access is very expensive: open() only done once
  - File descriptor of a i-node already hold 13 pointers to access the disk; or device driver routine
    - File open: get the i-nodes (not 13 pointers → normal file, device driver routine)
  - Open/Closed: specify the usage of the file
    - Open() call: do the expensive operation to find the i-node. Also check the permissions. Then later don't need to get.
  - Closed: freed up the memory kept for the read and write

- Software Block Cache Design **(buffer cache)** - Kernel buffer



  - Has copies of blocks that are also on disk
  - In kernel, keeps an array of blocks as cache
  - Upon read or write
    - check if the buffer contains the block
    - If not, get from disk
    - To make room, remove LRU block



  - Cache as **hashtable** to quickly find the page and **linked list** to keep LRU order
  - When removing a block, writes to disk if it has been modified
  - Kick something out the least recently used page. Copy out only if the block is modified.

- Summary
  - I/O system structured in layers
    - User-level I/O libraries
    - Kernel-Level device-independent
    - Kernel-level device-dependent
      - Upper-level device drivers
      - Lower-level interrupt handlers
  - Hardware effect on device drivers
    - I/O vs. memory-mapped, Programming IO, interrupts, DMA, polling

LEC 13 - Protection

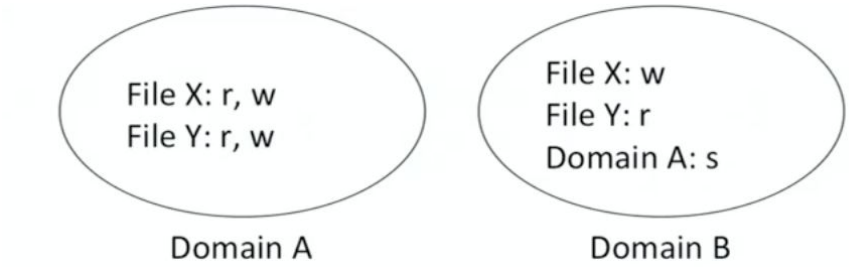
- Introduction
    - Processes access resources, resources are shared, and need to be protected
        - From process without permission
        - From improper access by a process

- Kernel Enforces Protection
    - Kernel “own” the resources (allow access)
    - To access resources, a process must ask for it and be given acess

- Protecting the kernel (mechanism)
    - Memory protection
    - Protected mode of operation: kernel (execute any codes) vs. user
    - Clock interrupt, so kernel eventually gets control

- Goals Supported by Kernel
    - Allow range of permissions
    - Allow user to get/set permissions
    - Be fast/simple for common case
    - Supporting user expressing complex permissions

- Formal Model
    - Protection: how to limit access to a resource
    - Resource: object that requires protection
    - Domain: set of (resource, permission) pairs



    - Process: accesses resources within domain

- Protection Matrix

Protection Matrix

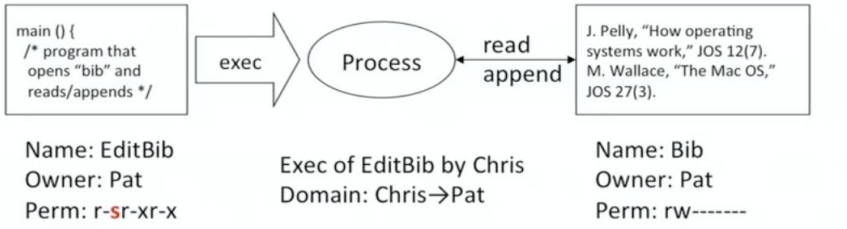
Domains	Resources: X	Y	A	B
A	r, w	r, w		
B	w	r	s	

- Matrix entry [d, r] contains permissions/rights
- Rows are domains and columns are resources
- The matrix is very **sparse**, so inefficient to store

- Efficient Representation
  - **Access Control List**: each resource, list (domain, permissions) pairs
    - ACL associated with resources
    - Like a registry: if name is on list, ok to access
    - **Cons:** Can be inefficient: must lookup on each access
    - **Pros:** Revocation is easy: remove from the list
  - **Capability List:**
    - Associated with each domain
    - Like key/ticket: if you have it, you get access
    - **Pros:** Efficient: on access, just produce capability
    - More efficient: something is empty, don't include in the list
    - **Cons:** Hard to revoke

- UNIX Protection
  - Associated with each file a set of permissions
    - Permissions bit r/w/x for owner, group, world
    - Limited form of access control list
  - Protection domain: UID (user account ID) + …
    - A process is always in some domain
  - When process **opens** (expensive) file, check permission
  - If ok, provides capability (fast) for future access

- Extending protection in UNIX
  - For special cases, can extend via user program
  - Domain switch: SETUID bit
  - E.g. Pat and Chris



  - Chris have access to the Pat's file, through the provided EditBib method. SETUID bit is on. The permission is valid for the during of the program. Permission is **limited to what program does**.
  - The process operates under the Pat's domain

LEC 14 - Security

- Introduction
    - Protect computer systems from outside entity: contents, operation and services
    - Various threats: theft, damagem, disruption

- Aspects of security
    - Confidentiality: disclose **only to those authorized**
    - Integrity: **only authorized changes**
    - Authenticity: is it really who/what it claims to be
    - Availability: can I access the service

- Security Threats
    - Interception: eavesdroppings, communication be listened (extract information)
    - Interruption: destroying, denial of service
    - Modification: tampering with data or programs
    - Fabrication: new data/programs, replaying messages
    - *Note:* deals with determined attackers

- User **Authentication**
    - Passwords are the most common method:
        - user and computer knows the secret and proves the knowledge to check
    - Encrypted passwords
        - Computer stores only encrypted passwords
        - User provides password and then computer encrypts check
    - Problem
        - Password length vs. password strength

- Challenge/Response Protocol
    - Challenge/response, algorithmic passwords
        - User and system know secret algorithms
        - System challenges user's knowledge, user responds
    - Example: $f(x) = x^2$
    - Example: favorite car, table lookup, etc.
    - Secret is never sent, only challenge/response

- **Attacks**
- Trojan Horse
    - Program that contains hidden malicious code
    - Program runs as process in user's domain

- Trap Door
    - Secret access point in program

  - Designer develops program for someone else
  - Once loaded in system, designer can access
  - Consider if trap door is added by **compiler**

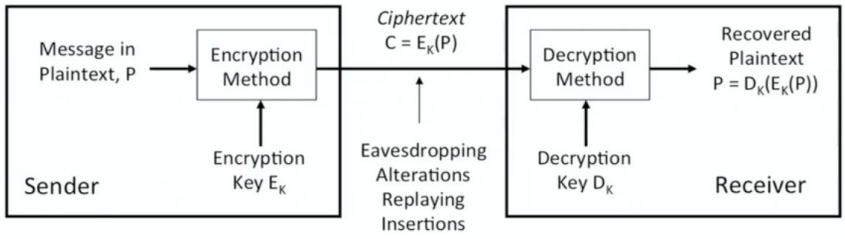
- Virus
  - Code attached to legitimate program
  - When the program runs, the virus runs → causes damage and spreads
  - Disinfectants: check for known pattern
  - Biological system, DNA attacks but DNA is different

- Internet worm
  - Program that copies itself over the network
  - Takes advantage of the tools that already existed
  - E.g. finger program with certain parameters → override the local parameter

- Denial of Service
  - Preventing others from using system: by using a lots of resources, by bombarding with network requests or traffic

- Intrusion detection
  - Detecting if there is an intruder or attack
  - Signature-based: look for specific patterns of attack behavior
  - Anomaly-based: look for unusual behavior
  - Solution: create audit trail (log), then analyze; “honey pot”

- **Cryptography**
  - Encoding messages to limit who can view the original messages, determine who send the message



- Secret Key Encryption (symmetric)

  - Same key to encrypt and decrypt
  - Well known published algorithm → key to secure the information
  - DES: data encryption standard (56 bits)
  - AES: advanced encryption standard (128, 256)
  - **Pros:** past

    - **Cons:** impractical to distribute key
- Public key (asymmetrix)
    - Different keys to encrypt and decrypt
    - Each user has two keys: one public, one private
    - E.g. A sends to B
        - A encrypts with **B's public key**
        - **B decrypt using its private key**
    - RSA
    - **Pros:** convenient for distribution
    - **Cons:** slower

- Combination:
    - Public key start session: exchange secret key
    - During session, use secret key

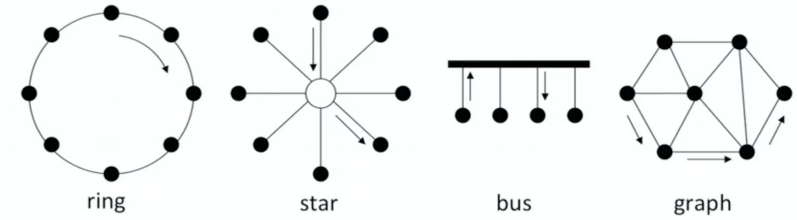
- Digital Signatures:
    - Encrypt M using p**rivate key of A**
    - **Bob decrypt using Alice's public key**
    - Include random number to prevent replaying the message

LEC 15 - Networks

- Definition
    - Set of computing nodes
    - Connected by communication links
    - Allows data transfer by sender to receiver

- Internetwork: a network of networks
    - Internet as a global internetwork
    - Nodes communicate using IP (Internet Protocol)

- Types of Networks
    - Topology: ring (binary or uni), star (central as hub, send to hub and hub sends out), the same amount of time), bus (wire that connects all nodes, shared; coordinate with each one), graph (general collection of nodes with link)



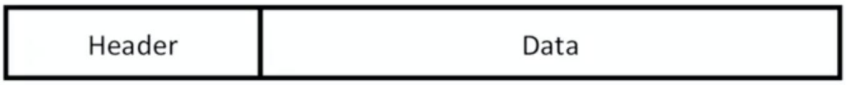
    - By geographic coverage:
        - LAN: local area network (spanning floor, building): limited to space
        - WAN: wide area network (spanning state, country)

- The methods for Data Transfer:
    - **Circuit Switching**: establish path and send data
        - Reserve resources provide performance control
        - Example: telephone system
        - Takes time to set up, but once setup, the data transfers through the wire
    - **Packet switching**: forward packets hop to hop
        - Fair sharing despite burst, statistical multiplexing
        - Example: postal system
        - Time is not constant and packets get delayed; won't wasted when in use
        - Delay rarely happen

- **Protocol**
    - Goal: message from sender to receiver
    - Protocol: agreed message format and transfer procedure
    - Multi-party: no central thread of control
        - Agreed on the sequences of steps
        - Sender and receiver are separate processes

- Expectations of operations
    - First you do x, then I do y, them you do z. If you do q, I'll do p.
- Any communication involves protocol (dial number, ring, hello, name, etc.)

- **Message**



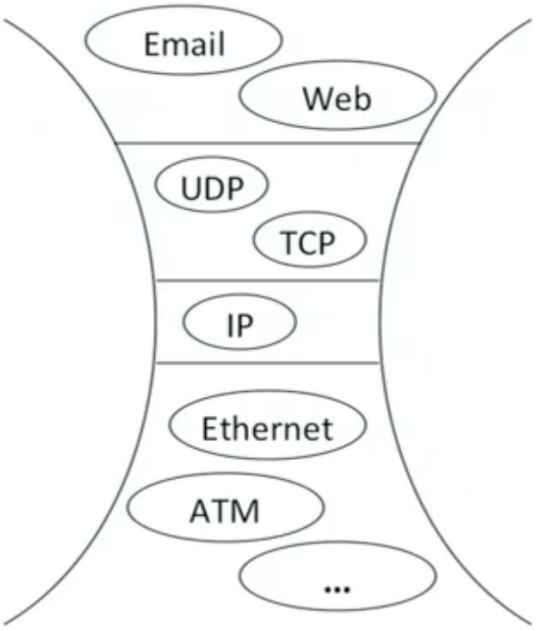
- Contains header and data. (packet, datagram, frame)
- Data: what the sender wants the receiver to know
- Header: *meta* information to support protocol
    - Source, destination
    - State of protocol operation: distributed system, represent the state.
    - Error control (check integrity of the data)
- Encapsulation: put data into another message
- Layering: separation of functions:
    - Ann and Bob: don't know about delivery
    - Postal system: only know addresses and how to deliver

- **7 Layers of OSI Reference Model**

Abstract model

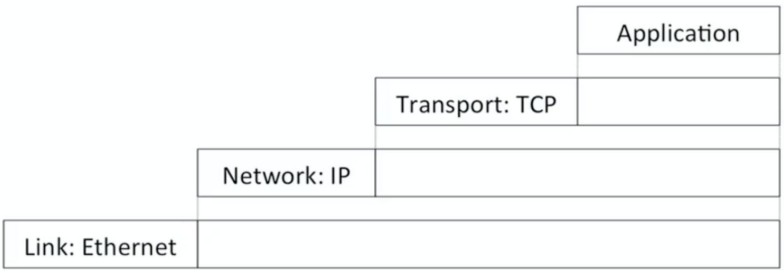
Application	Everything includes application is a protocol. E.g. HTTP
Presentation	Defines the format of messages and information for Application. JPEG as presentation layer protocol. Agreement.
Session	Start/stop/management connections
Transport	Segmenting: sending segments and groups of information of logical meaning. Reliability and flow control (fast or slow)
Network	Many physical networks connected together → all nodes use a certain type of addressing. Logical addressing and routing. (Conversion to physical address)
Link	Package out bits and frame them: start and end. Physical addressing: identify source and destination
Physical	Transmission of bits: interpret wire's 0 and 1 and how to implement 0,1.

- Internet Protocol Stack



- "Hourglass" design
- Application: email, web
- Session: socket (Unix: end point that allows to connect other end point)
- Transport: TCP, UDP, RTP
- Network: IP (narrow)
- Link: Ethernet (link and physical protocol combined together in hardware)
- Physical (support any physical layers)
- ***Note***: When people want to connect to the internet, either you write your own (as long as you follow the rules) and then connect to another node; then connect to entire node. Connect *simply*. Most complexity is inside the end point.
- ***Diff*****:** telephone network: most of the complexity is inside the network

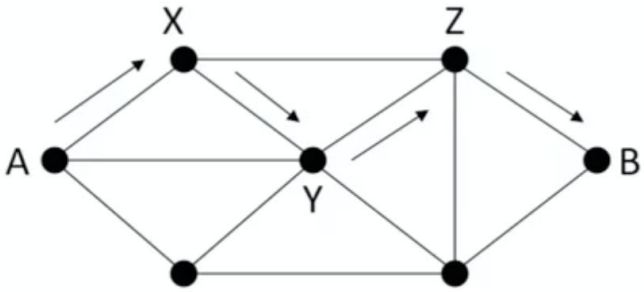
- Encapsulation



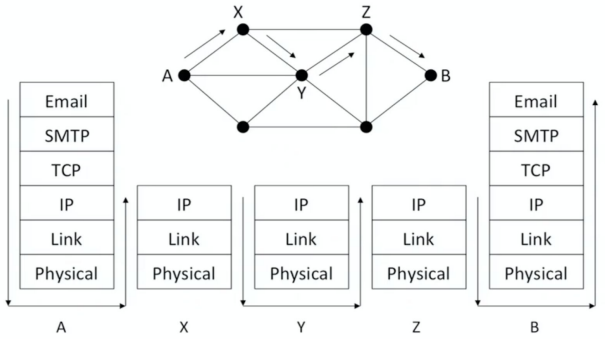
- Higher level n within lower level n - 1
- Hieracharly group of messages

- Addresses
  - Identify the other node
  - Three levels of addresses
    - Domain names: cs.ucsd.edu
    - Logical Address (IP): 128.53.27.92
    - Physical addresses (Ethernet): 0x27A5BB170…
  - Address resolution
    - Mapping higher level name to lower level name
    - Techniques: table lookup, formula, protocol
  - IPv4 (version 4, current/past)
    - 32 bit addresses
    - 2^{32} = 4 billion addresses
  - IPv6 (version 6, future/current)
    - 128 bis addresses
    - $2^{128} = 2.56 * 10^{38}$
    - Number of addresses per nm^2 ~ 500,000
    - Cross section of hair is 5 billion nm^2
  - The longer the address, then the overhead is heavier

- Routing
  - How to get packet from A to B: A forwards to X, X to Y, Y to Z, Z to B



  - Each intermediate node can be a decision point
    - Static: always same decision
    - Dynamic: decision can change based on state
  - Examples



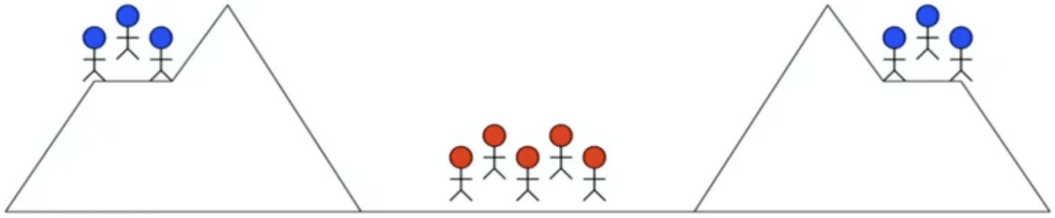
        - Email: Application: how to collect information (destination, source, subject, click send), but doesn't know how to get. Feed into lower layer protocol.
        - SMTP: domain resolution, etc. Doesn't know how to transfer with IP
        - TCP: break information into pieces, send each piece individually. 10kb → 1kb and attached the header a sequence number.
        - *Notes:* these layers are all codes runs **on your machine**
        - IP: routing, decide the transfer
        - Link: transmit the bits in terms of 0 and 1 over the physical layer
        - Then B's physical layer receives voltages
        - Link layer frame the bits of 0 and 1
        - IP then check if this packet for current node
        - *Note:* Router don't need anything above IP
        - Finally TCP package up the packets
        - SMTP gives the information to email server

- Scalability
    - How well does system grow: performance, reliability, etc.
    - Ramifications of adding node or link
        - local effect or global effects
        - **As localize as possible**
    - Information growth: important to reduce
        - amount stored at nodes
        - amount exchanged between nodes.
    - E.g. poor scalable system
        - One node affects all the others (global effect)
    - E.g. routing
        - Poorly scalable: each node needs to know every other node' information
        - In practice: Hierarchy structure

- Error Control
    - Add information to the header of the messages
    - Parity: even, odd, two-dimensional
        - E.g. 11 **0** → even number of ones.
        - **Cons:** Can't correct what went wrong
        - **Cons:** If two bits are changed
    - CRC (cyclic redundancy code)
        - A group of bits that will be a function of the actual information you are sending
        - **Pros:** Powerful
        - **Cons:** complex
    - Checksum
        - Between parity and CRC: adding all the bits in the information; then add

up all the bits to check matching
    - **Pros:** more powerful than parity, cheaper than CRC
- Automatic repeat request (ARQ)
    - More complex → use protocol to ensure the messages sent

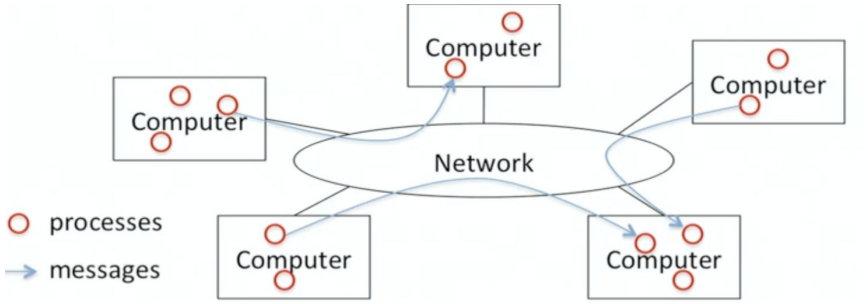
- The Two Generals' Problem



    - Goal: blues has to coordinate to attach at the same time, but messengers may be caught (unreliable channel).
    - **Fundamental Problem:**
        - Can't create **perfect channel** out of the **imperfect one**.
        - Technology doesn't help
        - Can only increase the probability of success

LEC 16 - Distributed Systems

- Definition



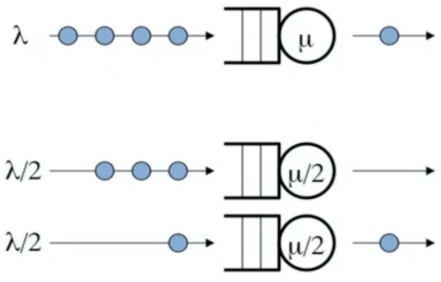
    - Cooperating **processes** (*not only computer*) in a computer network
    - Degree of integration

Loose	Communicate: Internet applications, email, web browsing
Medium	Know a lot about each other: Remote execution, remote file systems - (know it's remote)
Tight	Process migration (start command and then move), distributed file systems (looks like it's local)

- **Advantages**
  - Speed: parallelism, less contention
    - No resources need to be very busy
  - Reliability: redundancy, fault tolerance, “NSPF”
    - Many machines: error may happen but we can survive
    - ***N**o **S**ingle **P**oint **F**ailure*: no single point that it breaks down then the entire system breaks down
  - Scalability: incremental growth, economy of scale
    - Super powerful machines vs. 10 * uniprocessors
  - Geographic distribution: low latency, reliability
    - Local individual machines have faster communication

- **Disadvantages**
  - Fundamental problems of *decentralized control*
    - State uncertainty: no shared memory or clock
      - No node knows what’s going on in other places
    - Action uncertainty: mutually conflicting decisions
      - No node knows what others may do
    - No way to overcome these problems
  - Distributed algorithms are complex
    - Harder to program
    - E.g. load balancing system: start a process on a computer right beside me; there is a node that is idle.
      - Advantage to move the process to unbusy nodes
      - State uncertainty: can’t know the state, because the more communication about workload, the larger the overhead.
      - Action uncertainty: local decision - offload to a certain nodes, while other local decisions also lead to the same. → Global vs. local
  - Note: benefits > disadvantages

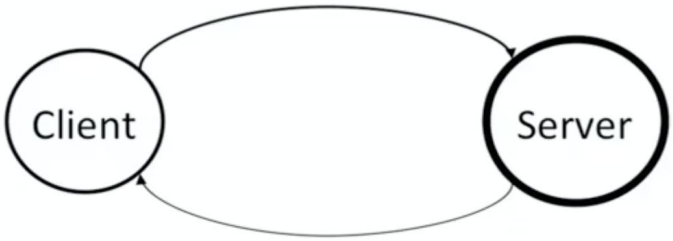
- Is distributed system better?
  - Queue system: arrival rate, miu as the processing power



    - Single one with fast server is the **better** one

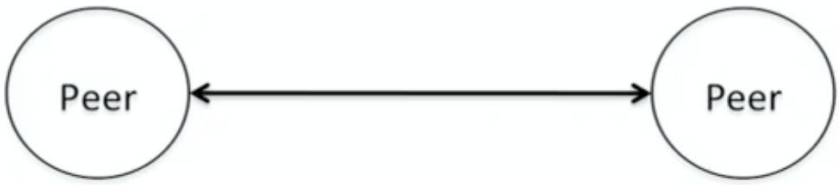
    - Multiple slower servers with separate queues: one line may idle and the other may operate at half speed.
    - *Note:* centralization is better. If continuous work and always busy, the average is the same, but the standard deviation is different. Variability in the second is lower than that in the first.
    - **Optimal solution**: multiple slower serves and single queue.
        - Little's Law: N = lambda * W
          number of processors in system = arrival rate * time spent

- The Client-Server Model



    - Client:
        - Short-lived process that makes requests
        - "User-side" of application
        - Web browsing
    - Server:
        - Exports well-defined requests/response interface
        - Long-lived process that waits for requests
        - Upon receiving request, carries it out (may spawn a thread)

- Peer-to-Peer

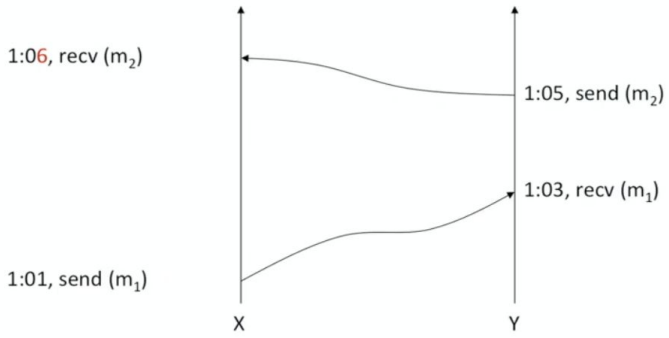


    - A peer talks directly with another peer
        - No intermediary (e.g. central server) involved
        - Symmetric (unlike asymmetric client and server)
    - In actuality, may be dynamic client and server
        - A request file from B, then A acts as client and B as server.
        - C can now requests file from A, A acts as server
    - P2P: file sharing servers

- Distributed Algorithms

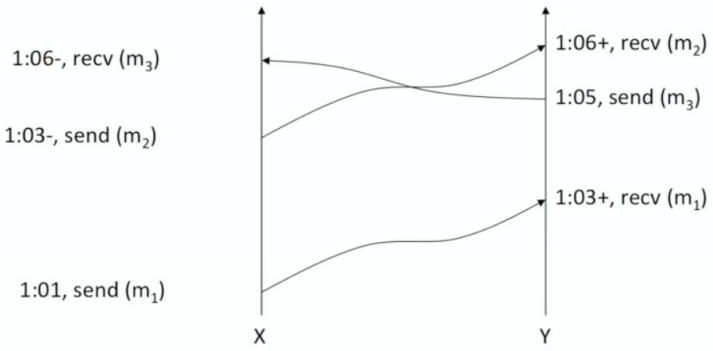
- Idea: the basic algorithms for more complex one. Some assumptions can't be made in distributions

- **Event ordering**: order events given no shared memory/clock
  - Happened before relation: →
    - A, B events in *same process* (sequential) and A before B: A → B
    - A is a send event and B is a receive event: A → B
    - If A → B then B → C, then A → C
  - Implementation:
    - Timestamp all events based on local clock
    - Upon receiving a message, advance local clock
      - Later than the send event
    - Resolve ties by ordering machines
      - Arbitrarily break the ties
      - A, B happen with same local time, then machine ID
  - E.g.



1:01 send (m_1); 1:03 recv (m_1); 1:05 send (m_2); 1:06 recv (m_2)

  - E.g. concurrent (*March 13, p. 16)*

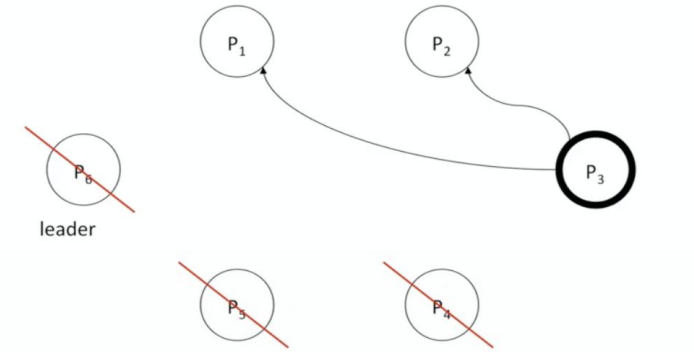


1:01 send (m_1); 1:03- send (m_2); 1:03+ recv (m_1); 1:05 send (m_3); 1:06- recv (m_3); 1:06+ recv (m_2)

    - Only resolves the order if the event is known.

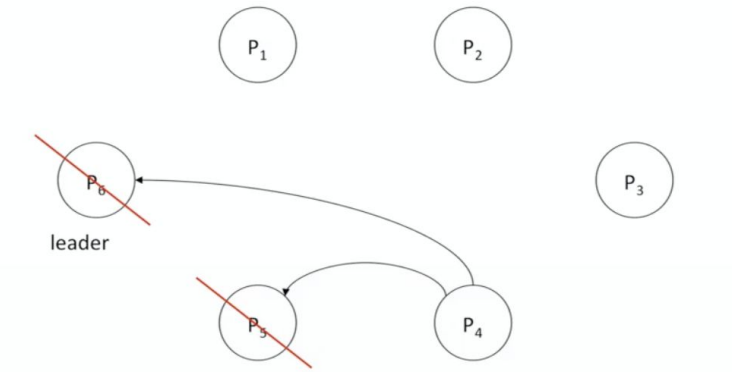
- Local decision makes sure the order of event
- It's not agreed upon the true order of events → there is **NO** true ordering of event. Two observers have different time.

- **Leader Election**: ensure no Single Point Failure
  - Many distributed algorithms rely on leader
  - Need to determine if leader exists: if not elect
  - **Bully** algorithm (elects leader)
    - Every process is numbered P_1, P_2, …, etc.
    - P_j sends request to L, no reply; tries to elect itself
    - P_j sends "Can I be leader" to all $P_{k>j}$
    - No replies, P_j sends to all $P_{i<j}$, "I am leader" done
    - If some $P_{k>j}$ replies, P_j lets P_k try to elect itself
    - If no message from P_k, P_j tries to elect itself again
  - Note: messages are prioritized
  - E.g. Case 1: P_6 as leader is down, then P_3 request to be the leader.



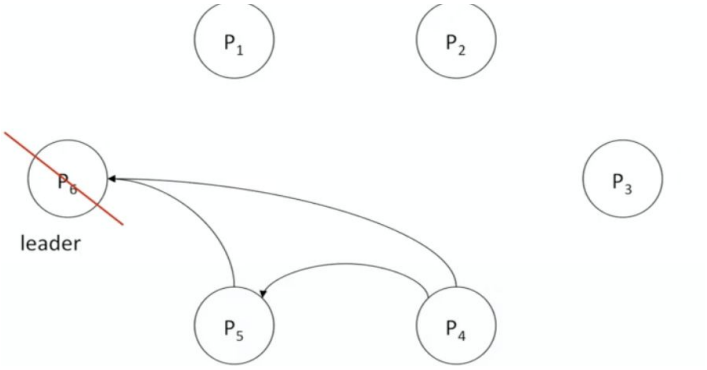
If P_3 were to not hear from P_4 and P_5, it would tell P_1 and P_2 "I'm the leader"

  - E.g. Case 2: P_4 is alive, then P_4 becomes the leader



P_4 tries to elect itself as leader, sending messages to P_5 and P_6; no replies

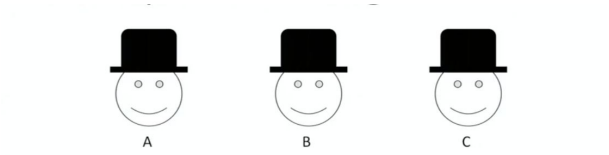
  - E.g. Both P_5 and P_4 sends back to become leader, then both asks P_6, P5 becomes the leader



- Not instantaneous and eventually would stabilize

- **Mutual Exclusion**
  - Centralized approach
    - Single process acts as coordinator server
    - Request, reply (to allow entrance), release
    - E.g. Peterson solution, TSL, advantage of memory bus
    - Another: designate a **coordinated process**: could say yes/no.
      - Two processes send the same requests; Then use agreement on time to order the event
  - Distributed approach
    - Process sends time-stamped request to all others
    - Waits until it receives replies from all (ok to enter)
    - Enter critical section (may get requests, defers)
    - Upon exiting, responds (to release) to all deferred
    - Use timestamps to order “simultaneous” requests

- **Common Knowledge Problem**
  - N people, each wearing red or black ha. Each can see the others’ hats, but not own hat.
  - E.g. Three people wearing black hat and then told **publicly “at least one black hat”**



All told: “At least one of you has a black hat”

Q1. “Is your hat black?” All: “I don’t know”

Q2. “Is your hat black?” All: “I don’t know”

Q3. “Is your hat black?” All: “Yes!”

  - Case 1: if A sees 2 red hats

- A sees 2 red hats
- Can answer "Yes" after 1st question
  - Was told "at least one of you has a black hat"

- Case 2: if two has black hats

A sees 1 red hat and 1 black hat
All answer "I don't know" to the first question
A: If I had red, B would have said "Yes"
So, after 2nd question, A (and B) can say "Yes"

- Case 3: three black hats → after third question

What If Three Black Hats?

- All see 2 black hats
- All answer "I don't know" to 1st, 2nd questions
- After 3rd question, each can say "Yes"
  - If I had red, others will have said "Yes"
  - Therefore, I must have black

- The hint is needed: allows each other to tell the difference between "all red hats" and "one black hat"

- Provides information about what others know.

- Common Knowledge
  - Knowledge of x:
    - K(x) = everyone knows x
    - $K^2(x)$ = everyone knows that everyone knows x
  - Common knowledge of x:
    - K(x) and $K^2(x)$ and $K^n(x)$ as n → infinity
  - Hint is common knowledge:
    - Announced publicly

        - Each must assume all heard it and are all smart

- **Connection With Distributed System**
    - When build a multi-process system, no way to know about every system. Has to rely on certain shared memory.
    - Can’t learn knowledge without knowing knowledge.

Programming Assignment Summary

PA1 - Context Switch

- UMIX: instructional operating system
  - Main function and all system calls are capitalized
  - Each process has a single thread of control and its own private memory

- Fork(): create a process
  - Parent: who called the fork(); Child: the new process being created
  - Parent and child have **identical memory**, with child start executing by returning from Fork()
  - **Return:** parent - id of the child being created; child - 0.
  - *Note:* UMIX always schedules the parent first

- Yield(pid): causes the running process to yield to the process whose identifier is pid
  - Entered kernel and eventually calls MySwitchContext
  - **Return**: PID that **yielded to the one** that is now running (and returning from Yield(pid)).

- MyContextSwitch(p): causes a context switch to process p
  - SaveContext of current, check return, RestoreContext of the new process
  - Magic number to **distinguish between returns** so that you can determine whether or not to call restore context.
  - **Return:** the process id that **just called yield**

- Void SaveContext (CONTEXT *c)
  - Save the context of currently running process into a CONTEXT variable pointed to by c. This context comprises the current values of registers (stack pointer, program counter).
  - **Return: TWICE** 1st - explicitly called SaveContext; 2nd - process's context restored

- Void RestoreContext (CONTEXT *c)
  - Restores the context of the process whose previously saved context is in the CONTEXT variable pointed to by c.
  - The saved context is result of
    - Call to SaveContext(c) at some point in the past
    - Call to NewContext(p, c) that was called when the process was created

- NewContext
  - Memory copy to copy context into the context table

PA2 - Scheduling

- SlowPrintf (n, format): takes in a delay parameter to specify the delay

- InitSched(): system start-up time
    - Initialize the process table
    - Initialize the fields of requested cpu, etc.
    - SetTimer for scheduling

- StartingProc(int p): called by the kernel when process starts up
    - Parameter: the pid of the process created
    - Find entry in the process table and record the PID

- EndingProc(int p): called by the kernel process p is ending
    - Update the process table

- Int SchedProc(): called by the kernel when it needs a decision for which process to run next
    - FIFO
    - LIFO
    - ROUNDROBIN
    - PROPORTIONAL: stride scheduling: pass and stride value.
    - **Return:** PID for the process to run; 0 if no processes left to run

- DoSched(): cause the kernel to make a scheduling decision at **the next opportune time**, at which point SchedProc() will be called to determine which process to select.

- Preemptive: timing
    - SetTimer(t), HandleTimeInterrupt()
-