Language Support:

Concepts:

- **Critical Regions:**
    - Semaphores bracket critical sections
        - Start/end exchanged between cooperating processes
    - Simple synchronization has wait(S) in one process and signal(S) in another process
        - better than primitive synchronization
        - Still susceptible to programming errors
    - Critical Region is a higher level construct that removes some of the programmer overhead
        - Higher level construct -> language support!!

Week 4:

Scheduling: ***Try to make one advantage and one disadvantage for a scheduling algo you see empty***

Introduction to Scheduling:

Definitions:

- **Burst:** “time where a process is working on a CPU”

Concepts:

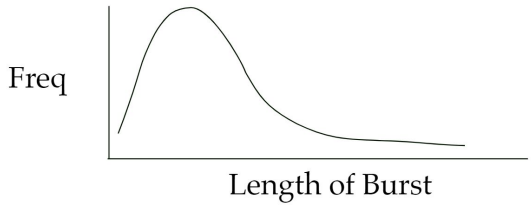
- **Scheduling - Basic Concepts**
    - Goal: Maximum CPU utilization or responsiveness
        - give CPU to another process while one process is waiting I/O
    - Processes proceed in bursts
        - Do some work
        - Do some I/O
        - Repeat
- **Processing Bursts**
    - Most CPU Bursts are short
        - Extensively studied
        - The longer the burst, the less likely it is to occur
        - Follows Hyperexponential distribution (see diagram)
    - Parameters of curve depend on operating system, application
        - Peak shifts to the left(and up) with more I/O
        - Peak shifts to the right(and down) with higher computation stress
- **CPU Scheduler:**

    - Selects processes from ready queue and allocates to CPU
    - Also called the Short term scheduler
    - When does the OS choose a new process?
        1. Processes goes to wait state (e.g. I/O, event wait)
        2. Process terminates
        3. Process is interrupted
            - running process goes to ready state
            - waiting process goes to ready state
    - If the scheduler chooses a new process in all three of the situations above then it is called a preemptive scheduler
    - If the scheduler is only invoked in the first two situations then it is called a Non-preemptive scheduler
- **Scheduling Criteria:**
    - CPU utilization – keep CPU as busy as possible
    - Throughput– # of jobs done per time unit
    - Turn around Time – Time of submission to Time of Completion
    - Waiting Time – amount of time in ready queue
    - Response Time – submit time to time of output request
- **Algorithm Evaluation: Deterministic Modelling (see Diagram)**
    - take an example representative workload
        - a set of cpu burst times, sometimes more than one burst time for each process
    - calculate each of the criteria for each of the algorithms (wait time, turn around time, etc.)
    - trace tapes (generated from real systems)
    - Notes from example: Average Wait time can differ based on order
    - Cons:
        - in general, makes too many assumptions
    - Pros:
        - good for showing how the algorithms work in an undergraduate class
- **Algorithm Evaluation: Simulation**
    - Simulate all of the relevant parts of the system
    - Difficult to link various parts of the model
- **Algorithm Evaluation: Implementation**
    - Actually try scheduler and find out how it works
    - Expensive for developer time and real implementation
- **Multiple Processors:**
    - Scheduling is more complex
        - usually a common queue for all processors (load sharing)
        - actual parallel system, have to watch access to kernel data structures such as PCBs and Queues
            - Accessing ready queues is a critical section and must be synchronized

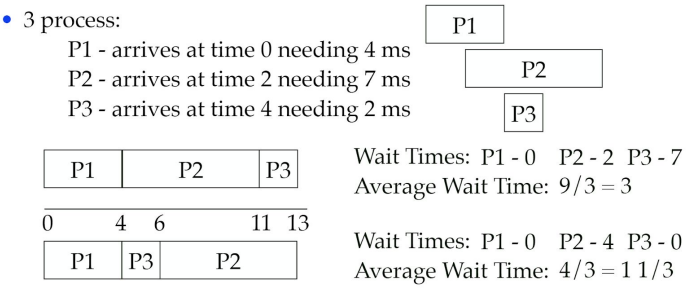
            - May have to use memory barriers to make sure memory is consistent between the cores and the processors
        - sometimes hardware limitations (I/O)
- **Real Time Scheduling**
    - Hard Real Time (result not delivered in time process fails)
        - guaranteed completion times
        - resource reservation
        - dedicated hardware
    - Soft Real Time
        - performance concerns
        - Multimedia
        - priority scheduling required
        - require low context switching overhead
        - kernel preemption points

Diagrams:

- Processing Bursts



- Deterministic Modelling



Scheduling FCFS(First Come, First Serve):

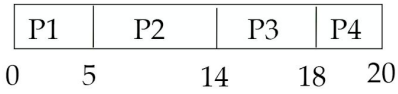
- **Advantage**:
    - Simple, easy to implement
        - Ready Queue is a FIFO Queue
- **Disadvantage**:
    - Average Wait time may be longer
        - Short bursty I/O do not have priority over CPU intensive jobs
        - Variance in wait time/ throughput is large, depending on order of jobs
        - Convoy effect - all I/O jobs end up behind CPU jobs which hog the CPU

    - Tends to make poor response in interactive systems
        - Used in simple OS or in systems with very little variance in CPU burt times

- **Example:**
    - FCFS

#	Arrival	Length
P1	0	5
P2	1	9
P3	2	4
P4	3	2

#	Expr	Wait
P1	0 - 0	0
P2	5 - 1	4
P3	14 - 2	12
P4	18 - 3	15



- Total Wait = 31 ms
- Average Wait = 7.75 ms

    -

- Variance high:

- *Convoy Effect:* I.O Jobs end up behind CPU Jobs, this hogs the CPU, so not very efficient
    - Watch the vid on Onq, you will see how long it takes to do one process because of its intensive nature, and subsequently how long other processes have to wait. This begs the questions, how do we make the short jobs/processes go first?

Scheduling SJF(Short Jobs First):

- **Advantages:**
    - Chooses order based on the shortest length of the next available burst
- **Disadvantages:**
    - must know the burst times ahead of time
- **Estimating CPU Burst Times - Exponential Average**

- Use length of last CPU burst to predict next CPU burst:

  t_n = current CPU burst time

  τ_0 = initial estimate

  τ_n = predicted for current burst

  τ_{n+1} = prediction for next CPU burst

  α = weighting parameter

  $$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\, \tau_n$$

  $\alpha = 0 \rightarrow \tau_{n+1} = \tau_0$ (initial estimate) never changes

  $\alpha = 1 \rightarrow \tau_{n+1} = t_n$ (last time slice) only used

- Advantages:
  - If process spends a reasonable period of time at a constant burst range then estimate approaches current burst time
- Disadvantages:
  - Predicted time always lags real time
- Example:

  $\tau_0 = 10$

  $t_0 = 5, t_1 = 5, t_2 = 5, t_3 = 8, t_4 = 8$

  $\alpha = 0.3$

  $\tau_1 = 0.3 * 5 + (0.7) * 10 = 8.5$

  $\tau_2 = 0.3 * 5 + (0.7) * 8.5 = 7.45$

  $\tau_3 = 0.3 * 5 + (0.7) * 7.45 = 6.715$

  $\tau_4 = 0.3 * 8 + (0.7) * 6.715 = 7.1005$

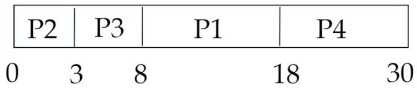
  $\tau_5 = 0.3 * 8 + (0.7) * 7.1005 = 7.37035$

- **Notes:**
  - Nonpreemptive - process gets entire burst time like FCFS (no interrupting?)
  - What is a reasonable period of time for burst range to stay constant?
    - α is the tuning parameter
    - α is low, then past behavior has more weight, est is slower to change (ignore transient behavior)
    - α is high, then last time slice has heavier weight, est is faster to change (more susceptible to transient behavior but tracks process that changes the behavior more accurately)
- **Example:**

- SJF

#	Arrival	Length
P1	0	10
P2	0	3
P3	1	5
P4	3	12

#	Expr	Wait
P1	8 - 0	8
P2	0 - 0	0
P3	3 - 1	2
P4	18 - 3	15



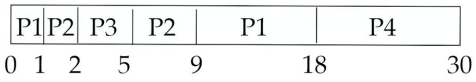
- Total Wait = 25 ms
- Average Wait = 6.25 ms

- **Shortest Remaining Time First (SRTF)**
  - Preemptive version of SJF (where process burst gets entire time needed)
    - after an interrupt, calculate remaining time for current burst
    - will be less than all other process in ready queue
    - if a new process becomes ready with shorter burst than remaining then dispatch the new process
  - Example:

SRTF Example

#	Arrival	Length
P1	0	10
P2	1	5
P3	2	3
P4	3	12

#	Expr	Wait
P1	0 + (9 - 1)	8
P2	(1-1) + (5 - 2)	3
P3	2 - 2	0
P4	18 - 3	15

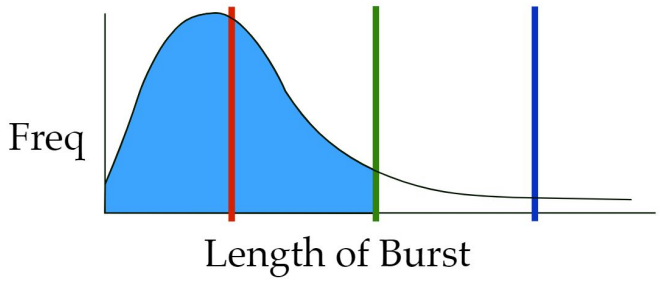


- Total Wait = 26 ms
- Average Wait = 6.5 ms

Scheduling RR (Round Robin):

- **Description:**
  - Similar to FCFS, but with preemption
  - Designed for Time Sharing Systems
  - Time slice(quantum)->maximum time process gets to run
  - If the quantum (q) is large->FCFS
  - If q is small, then appears to be multiple slower CPUs
- **Disadvantage:**
  - Context Switching is not free
    - Shorter q, more context switches to complete a single CPU burst for a given process
    - q must be large with respect to context switch time
- **Round Robin - Quantum Length**
  - If q is small we take the CPU away from many processes before the burst is done which causes extra overhead

- If q is large then we might as well not have quantum
- Ideal: 80% of CPU bursts should be shorter than q



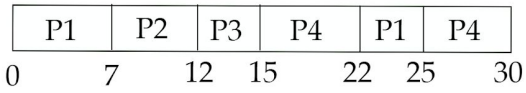
Where red is too small of a q, blue is too large of a q, and green is perfect q

- **Example:**
  - Round Robin

RR Example with q=7, negligible overhead

#	Arrival	Length
P1	0	10
P2	0	5
P3	0	3
P4	0	12

#	Expr	Wait
P1	0 + (22 - 7)	15
P2	7 - 0	7
P3	12 - 0	12
P4	(15 - 0) + (25 - 22)	18



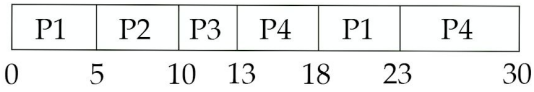
- Total Wait = 52 ms
- Average Wait = 13 ms

Expression formula: (start time-arrival time) + (2nd start time - 1st finish)

RR Example with q=5, negligible overhead

#	Arrival	Length
P1	0	10
P2	0	5
P3	0	3
P4	0	12

#	Expr	Wait
P1	0 + (18 - 5)	13
P2	5 - 0	5
P3	10 - 0	10
P4	(13 - 0) + (23 - 18)	18



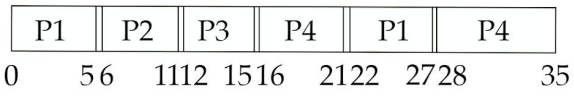
0 5 10 13 18 23 30

- Total Wait = 46 ms
- Average Wait = 11.5 ms

RR Example with q=5, 1 ms overhead

#	Arrival	Length
P1	0	10
P2	0	5
P3	0	3
P4	0	12

#	Expr	Wait
P1	0 + (22 - 5)	17
P2	6 - 0	6
P3	12 - 0	12
P4	(16 - 0) + (28 - 21)	23



0 5 6 11 12 15 16 21 22 27 28 35

- Total Wait = 58 ms
- Average Wait = 14.5 ms

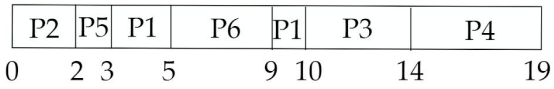
Scheduling Priority & MultiQueue:

- **Priority Scheduling:**
  - Each process has a priority
  - CPU goes to process with highest priority
    - ready queue is sorted based on priority
    - process with the same priority is handled FCFS
    - preemptive / non preemptive versions
    - Priority set Internal (Based on properties of the process)/ external (based on factors such as money or budget etc.)
    - Starvation (if there is always a process of higher priority available the low priority will never be run)

- Example:

Priority Scheduling (Preemptive)

#	Arrival	Length	Pri
P1	0	3	3
P2	0	2	1
P3	0	4	4
P4	0	5	5
P5	0	1	2
P6	5	4	2



#	Expr	Wait
P1	(3 - 0) + (9 - 5)	7
P2	0 - 0	0
P3	10 - 0	10
P4	14 - 0	14
P5	2 - 0	2
P6	5 - 5	0

- Total Wait = 33 ms
- Average Wait = 5.5 ms

- **Multi Level Queues**
  - Each queue has its own scheduling algorithm
  - Interactive (foreground) - probably Round Robin
  - Scheduling must be done between the queues
    - usually fixed priority preemptive scheduling (starvation)
    - time slice between queues (portion time between queues)
  - In simplest form, processes are assigned a queue and remain there until completion
  - Higher priority queues may require more money, or more status

    highest priority

    System Processes
    Interactive Processes
    Interactive Editing Processes
    Batch Processes
    Experimental Processes

    lowest priority
  -
- **Multi Level Feedback**
  - processes move between queues
  - when doing I/O, processes move to higher priority queues
  - When CPU intensive, processes move to lower priority queues
  - Give higher priority queues smaller quanta (preemptive)
  - Processes that use entire quanta are too high priority, bump down to lower priority queue
  - Processes that don't use entire quanta are too low priority and moved up to a higher priority queue
  - Parameters:
    - number of queues
    - the scheduling algorithm for each queue

        - when to upgrade a process
        - when to downgrade a process
        - how to choose the initial queue
    - most complex algorithm, is approximated using priorities

Deadlock:

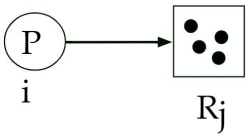
Introduction to Deadlock:

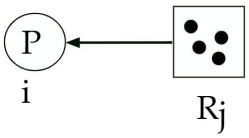
Definitions:

- **Deadlock:** “A set of processes, each holding a resource that another process in the set needs” - all processes are on wait queue
    - Can cause starvation (if a process never gets the resource it needs it can never complete)
    - Rollback (can we take a resource away from a process so the other process can use it)

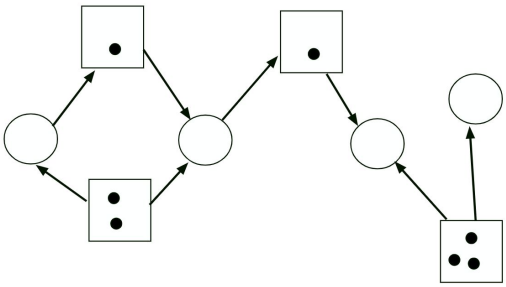
Concepts:

- **System Model**
    - Resource Types R1, R2, ..., Rn
        - Each resource has a number of instances (Wi)
        - Resource instances are indistinguishable (doesn’t matter which one you get)
    - Process resource protocol
        - 1. Request
        - 2. Use
        - 3. Release
- **Four Deadlock Conditions:**
    - **Mutual exclusion:** only a limited number (usually one) process at a time can use a resource
    - **Hold and wait:** a process has (at least) one resource and is waiting for another
    - **No preemption:** we can’t take a resource away from a process
    - **Circular wait:** P0 waits for a resource held by P1, which waits for a resource held by P2, ... Pn, which waits for a resource held by P0
- **Resource Allocation Graph:**

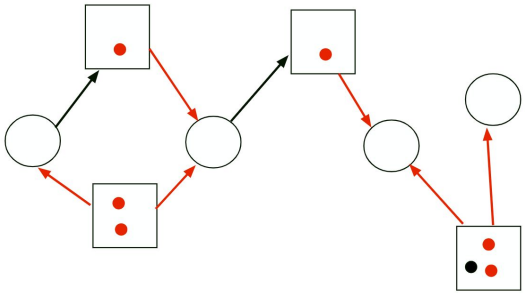
- Each Symbol is described to the left:
  - Process 
  - Resource Type
    - ◊ 4 instances 
  - P_i requests an instance of R_j 

  - P_i holds an instance of R_j 

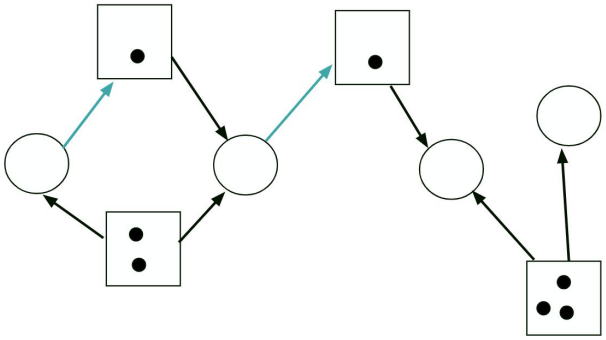
- Example:



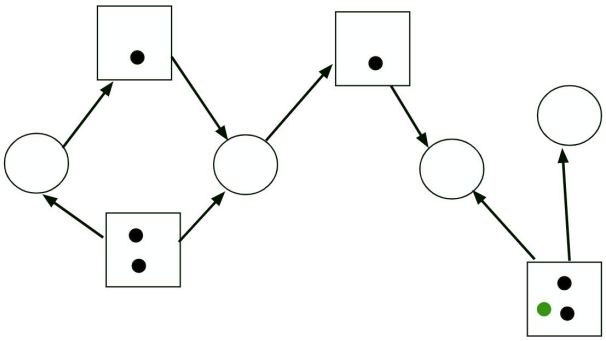
Above is a Resource allocation graph example



Highlighted in red are the Allocated resources



Highlighted in blue are resource requests



Highlighted in green are the free resources

- Deadlock Basics
    - No Cycle->no deadlock
    - Cycle
        - One instance per resource -> deadlock
        - More than one instance -> (maybe deadlock may not)

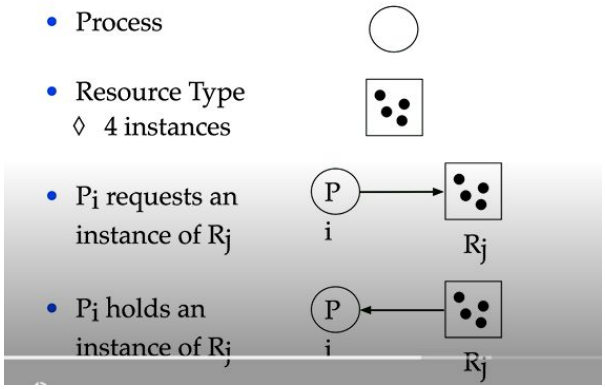
-A set of process, each holding a resource that another process in the set needs
-Process resource protocol:

1. Request
2. Use
3. Release

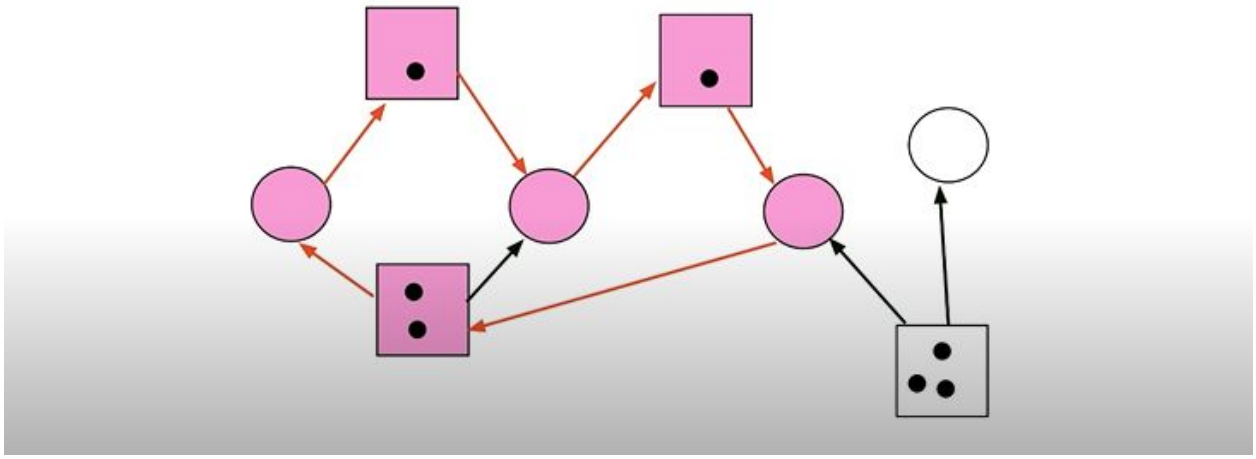
-4 Conditions for deadlock:
Mutual Exclusion: Limited # of processes at a time can use a resource (usually one)
Hold and Wait: Process has at least 1 resource, and is waiting for another
No Preemption: Cannot remove ressource from a process
Circular wait: P0 waits for resource held by P1,which waits for a resource held by P2,...Pn, which waits for a resource held by P0

-**Resource allocation Graph**

-Each bullet point represented by symbol to the right of it (instance = dots #)



Deadlock Example:
-coloured in elements are all in deadlock as they are all in the hold and wait phase.



Week 5:

Deadlock:

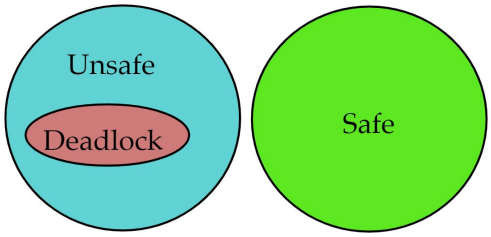
Deadlock Actions:

- **Prevention:**
  - Ensure one of the 4 conditions(mutual exclusion,hold and wait,no preemption,circular wait) never happens
  - **Prevent Mutual Exclusion:**
    - Some resources are shareable
    - Can add spooler or other device driver in some cases
    - Least flexible condition
  - **Prevent hold and wait:**
    - Cannot have another resource when requesting one

        - If need multiple resources, must request them all at same time
        - After using one or more resources, must release them all before requesting new one
        - Worse efficiency as a result (resource utilization is lower: processes have to hold resources longer, over commit resources, might need resources, so take resource)
        - Can lead to starvation since some processes cannot get resources
    - **Relax Preemption:**
        - take away resources when needed (priority)
        - If holding resources and ask for more that are unavailable, must lose the ones you have
        - Wait for entire set to become available
        - Preempt another process that is waiting for the requested resources
        - Rollback - like a save point in a videos game but for processes
    - **Prevent Circular Wait:**
        - Request in same order - no circular requests
        - Impose order on all resource requests

- **Avoidance:**
    - Ask processes to provide extra information before allocating an available resource
        - Information upfront (Processes declare maximum resources needed)
        - Dynamically check current resource allocation to make sure no circular wait condition
        - Resource allocation state: Number of available and allocated resources and a priori known max resources
- **Recovery:**
    - Enter deadlock state and recover
- **Ignore:**
    - Hope it never happens and simply handle it manually when the time comes. Most interactive operating systems use the ignore approach.

- **Safe states:**
    - Safe if no deadlock possible
    - Unsafe if deadlock is possible



- <P1,P2,...Pn> is safe if Pi can satisfy the max resources available and the resources owned by previous processes.
    - -P1 max must be satisfied only with free ressources
    - -P2 max must be satisfied with free + P1
    - -P3 max gets available + P1 + P2
    - -If not, wait until a previous process finishes
- Safe example:

Process	Current	Max
P0	5	10
P1	2	4
P2	2	9

Total = 12, Free = 3
< P1 , P0 , P2 >
P1 (2) + 3 = 5 >= P1Max(4)
5 + P0(5) = 10 >= P0Max(10)
5 + P2(2) = 7 not >= P2Max(9)
10 + P2(2) = 12 >= P2Max(9)

Unsafe example

Process	Current	Max
P0	5	10
P1	2	4
P2	2+1	9

Total = 12, Free = 3-1 = 2 ----> No longer safe
< P1 , P0 , P2 >
P1 (2) + 2 = 4 >= P1Max(4)
4 + P0(5) = 9 not>= P0Max(10)
4 + P2(3) = 7 not >= P2Max(9)

Bankers algorithm: find order of processes so that cascading sum holds in parallel for each resource type.

Deadlock detection: allow system to deadlock. Run a detection algorithm occasionally.
Recovery scheme -----> Analogy: like flat tire, quick change and good to go.

-Terminate processes

-One at a time until deadlock is resolved
-Run deadlock alg. Each time
-Choose by:

1) process priority
2) compute time
3) resource usage
4) resources needed
5) type of process(interactive,batch)

-Resource Preemption:

-Take away resources from other processes

-One or more processes must be rolled back

-Starvation (one process may be chosen as victim)

-Different than prevention case

-In prevention case we just said resources are preemptible and build it into a system so one process can take resources from another.

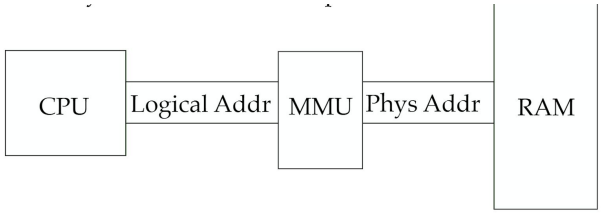
-Here we use it as last resort

Memory Management:

Binding:

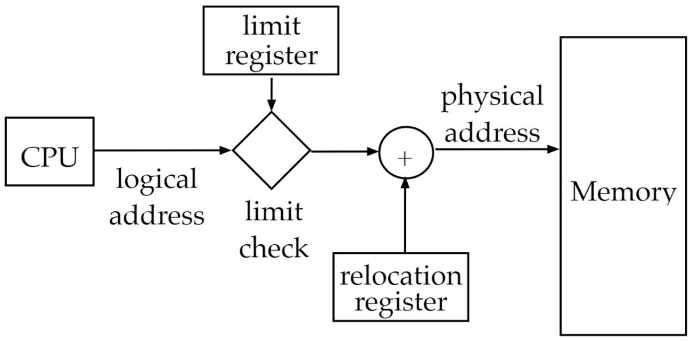
- **Binding Instructions and Data**
  - Entities in the original program must be bound to a location in memory
  - Programs may reside in different parts of memory
  - Three different stages:
    - Compile time: use absolute addressing, code can only be loaded at a particular location
    - Load time - When loader loads program into memory, addresses are resolved; header of load file contains locations of references, loader (part of OS) fixes them at the time they are read into memory
    - Execution time - programs can move into memory (hardware support required); similar to Compile Time, Symbols locations are specified at compile time, Hardware translate the compile time location to runtime location
- **Logical vs Physical Address Space**
  - Central Concept to Memory Management
  - Logical Address
    - address generated by CPU
    - also known as a virtual address
  - Physical Address
    - location in physical memory
  - Logical and Physical address are the same in compile and load time address binding. They differ in execution time binding
  - User program only deals with logical addresses. It never sees the physical address
- **Memory Management Unit (MMU)**
  - Hardware that maps virtual to physical address
    - many different approaches

- Logical addresses may be shared between processes, or may be different for each process.



- One simple approach is to have a single register that is added to every virtual address
  - Similar to the original memory protection scheme talked about in first module
  - Limit register now gives the size of process memory space
  - base register is called the relocation register
  - The PDP-11 divided the 64K logical address space into 8 chunks each with a relocation register and limit (up to 8K)
- Logical addresses range is (0...max)
- Physical address range is (R...R+max)◊ R is the value of the relocation register

Simple MMU



- **Execution Time Binding:** Thus the compiler, assembler and linker can directly use the locations of functions, variables and other memory locations in the logical address space, while the MMU will convert those addresses to the address in physical memory.

Programs may reside in different parts of memory:
-Three diff stages
-compile,load time, execution time

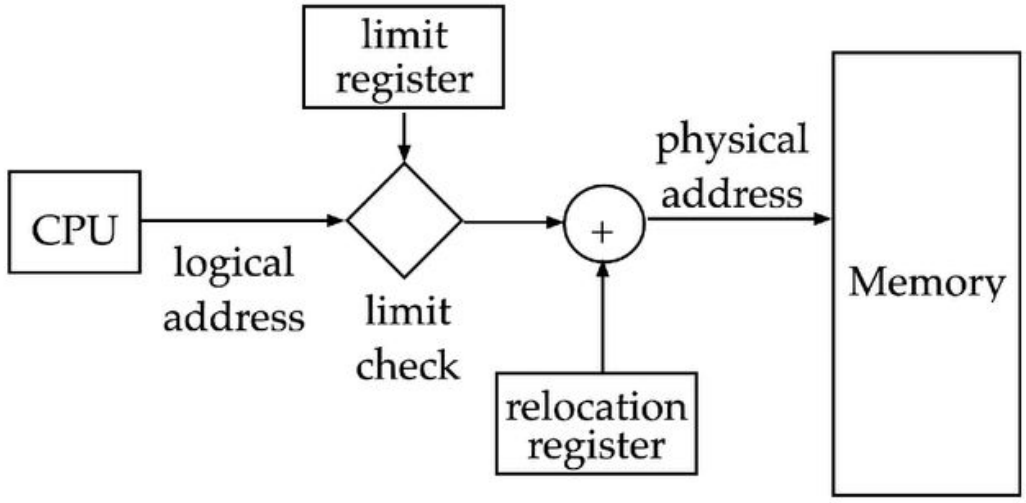
Load time:
-Header of load file contains locations of references
-Loader fixes them at the time they are read into memory

Parts of file:
-Header of load file contains locations of symbols and references to those symbols

-Text segment contains binary machine code
-Data segment contains initialized data

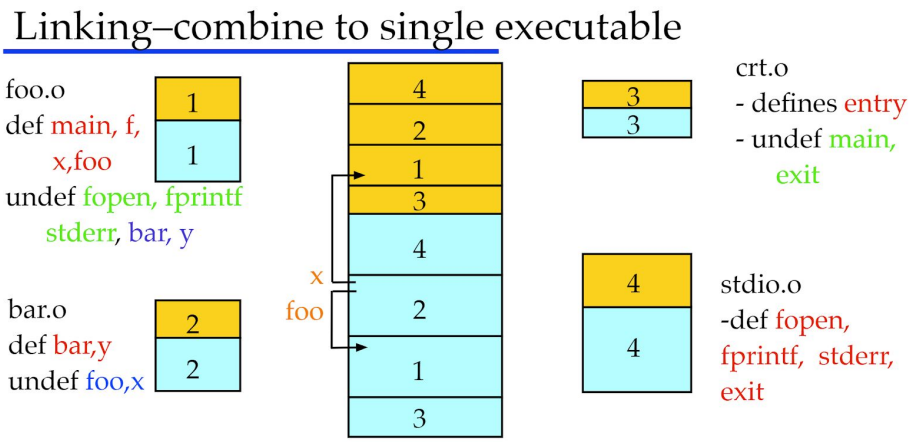
Definitions:
Logical address: generated by Cpu also known as virtual address
Physical address: is location in physical memory
MMU: hardware that maps virtual to phys. Address

Other key points:
-Base register is called the relocation register
-R is the value of the relocation register



Linking (Object Modules):

- **Linking:** how several separate programs are compiled and assembled (linked) into an executable
- **Object Modules (has 3 parts):**
    - Header
        - Tells linker what sort of information is in the file
    - Data Segment
        - Declarations and values for variables
    - Text Segment
        - Machine code for functions



-

Contiguous Memory Allocation:

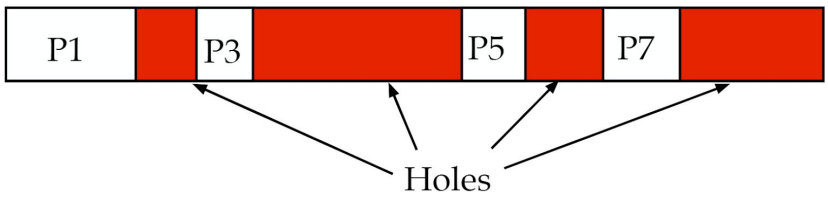
Concepts:

- **Contiguous Memory**
  - Using relocation and limit register assumes that the process is given a contiguous chunk of physical memory.
    - single partition allocation
    - simple allocation strategy, similar to that used by malloc in C or new in Java
  - Main memory is divided into two parts
    - Operating system (usually in same part as the interrupt vector)
    - User memory (divided among processes)
- **Contiguous Allocation (see Diagram)**
  - When system first starts, allocation is simple
    - One block of memory, and as each process starts, allocate the memory to the process
  - Some processes run long, some quit soon after they start
    - Not related to size. A large program can run short or long…
  - User memory must be allocated to processes
    - fixed size segments – IBM MFT – obsolete
    - variable size segments
  - OS keeps list of holes
    - memory not allocated to a process
    - when a process is started find a hole big enough to hold it
    - when a process ends, add the memory to the free list
  - General memory allocation problem
    - When do we merge adjacent holes? on allocation or on free?
- **Storage Allocation (3 general approaches):**
  - **First Fit (see Diagram):**

        - Use the first hole on the free list that is big enough
        - Only look at a part of the list
        - To try to spread processes out through memory we don't always start from the beginning we may start from last place a process was allocated
    - **Best Fit (see Diagram):**
        - Smallest block that is large enough
        - Search entire list
        - Minimizing the amount of memory that is lost
    - **Worst Fit (see Diagram):**
        - Largest block (largest remainder of memory to be used for another process)
        - Worst algorithm
- **Fragmentation (see Diagram):**
    - **Internal Fragmentation:**
        - If we allocate memory in units larger than a single byte (say 1k)
        - It is unlikely that our program is a multiple of 1k in size (may be 6.5k in size for example)
        - Last block is only partially used (in the 6.5k case the last 0.5k would not be used)
    - **External Fragmentation:**
        - Lots of small holes spread throughout memory, none are big enough to satisfy a request
        - Worst fit algorithm tries to reduce this
        - We can remove external fragmentation with Compaction - move blocks (requires execution-time binding)
- **Fragmentation Tradeoff:**
    - Allocate in smaller units (instead of 1k units try 0.5k)
        - less internal fragmentation
        - more overhead in managing memory
    - Allocate in larger units
        - easier to manage
        - more internal fragmentation
    - 50 percent rule - if we have N allocated blocks to processors, we will lose approximately 0.5 N to fragmentation (as a result 1/3 of memory remains unusable in a contiguous allocation scheme)

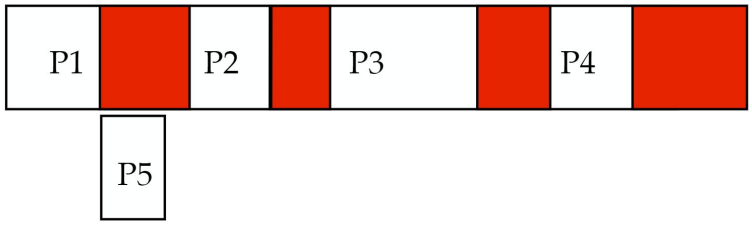
Diagrams:

- Contiguous Allocation

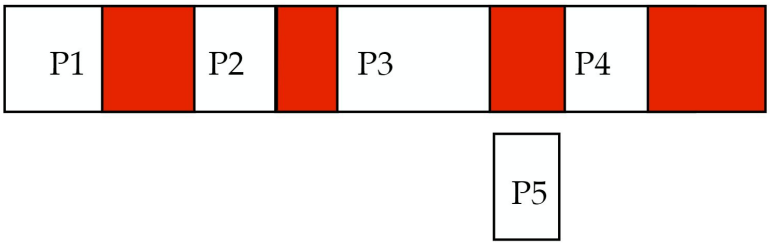


Holes are left after processes get freed

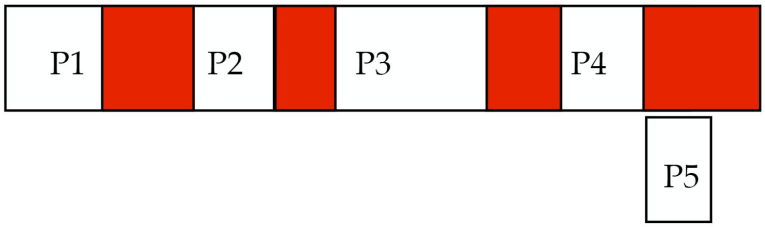
- First Fit



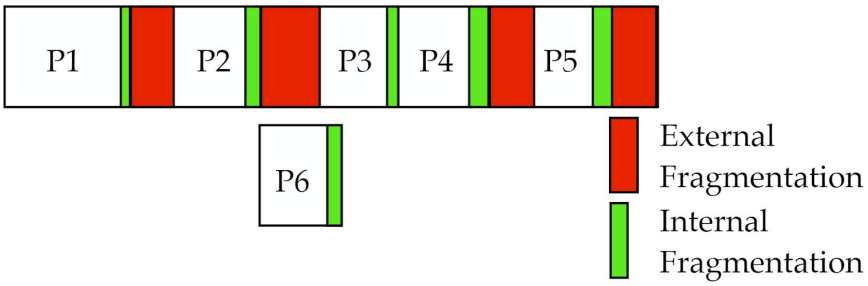
- Best Fit



- Worst Fit



- Fragmentation



-Using relocation and limit register assumes that the process is given a contiguous chunk of physical memory

-single partition allocation

-simple allocation strategy

-One block of user memory and as each process starts, they are allocated to the process when system starts up

-Processes do not run long based on size.

-Memory consists of a set of holes as some processes leave and leave holes in memory chunks

-OS keeps list of the holes present. Which is memory not allocated to a process

-When a process **starts** _> find hole big enough to hold it

-When a process **ends** -> add the memory to the free list

Allocation of memory into existing holes:

1.First fit

-Use the first hole big enough on the list

-Advantage: only look at part of the list

2.Best fit:

-Smallest block that is large enough. But you have to search n elements (whole list)

3.Worst fit

-Largest block (largest remainder)

-Worst algorithm

Memory fragmentation:

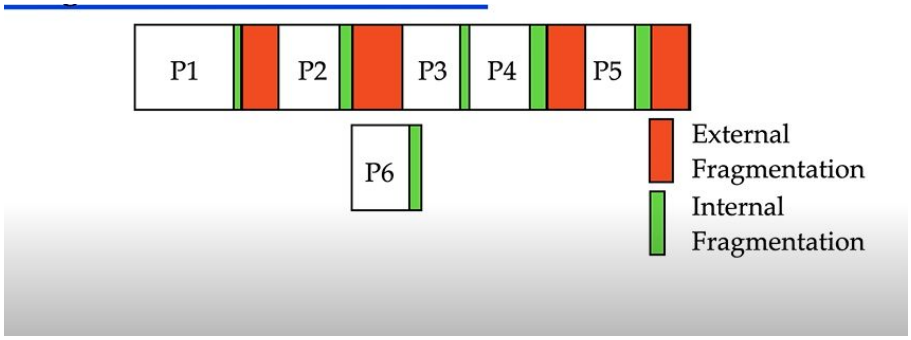
-**Internal fragmentation**

-If we allocate memory in units larger than a single byte (say 1k) , last block is only partially used

-**External fragmentation**

-Lot of small holes spread out which does not have enough space to satisfy ANY requests

-Worst fit actually REDUCES this



Fragmentation trade off

-Allocate in smaller units

-less internal fragmentation

-But more overhead in managing

-Allocate in larger units
    -Easier to manage
    -More internal fragmentation

Therefore we have generalised a 50% rule
-N allocated blocks to processors, we will lose 0.5N to frag. Therefore ⅓ memory is said to be unusable

Dynamic Loading and Linking:

Concepts:

- **Dynamic Loading:**
    - memory is always in short supply
    - not all routines are loaded when the program is loaded
        - only loaded when needed
        - some routines are rarely if ever used
        - does not require any special support from operating system
    - some execution environments do support dynamic loading (IBM mainframe, Java VM)
        - external programs are called by name, OS provides binding
- **Overlays:**
    - common on older systems (MS-DOS)
    - no OS support required (although OS can get in the way)
    - program is broken into multiple parts•one common part of program always in memory
    - other parts of program are replaced as needed
    - common in early games for MS-DOS- different levels of the game might have different code parts, as each level is loaded, the code overlays the previous code
    - was also common in tools like compilers and assemblers
    - complex details in overlays, not common today
- **Dynamic Linking:**
    - Static linking is when the all of the modules including system libraries are linked together at compile time
    - dynamic linking waits to run the linker for some of the system libraries.
        - application object files (i.e. .o files) and static application libraries are linked together to produce an executable
        - When the executable is loaded by the kernel, the dynamic linker is run to include the system libraries and dynamic application libraries
        - primarily used for common libraries
          – libraries commonly used by many programs
          – advantage: allows updates and bug fixes without relinking every application
- **Shared Libraries:**

    - Many system libraries are the same for all applications.
        - The standard c library (fopen, fclose, printf, atoi, etc.) is the same for all applications
    - Ideally, we should only have one copy in memory for all processes.
        - Some applications may depend on a specific version of some specific system libraries.
    - Requires some way to share memory between processes (we will discuss later).
    - **Static linking** is when all of the modules including system libraries are linked together at compile time.
        - shared libraries must be preallocated into common address space at runtime (not common due to complexity)
    - When dynamic linking, the OS invokes the dynamic linker when the process is loaded.
        - linking uses the Program Link Table (PLT)
        - The code must be position independent as the code may be shared in different address locations in different processes
- **Swapping:**
    - processes can be temporarily stored (swapped) from memory to a backing store
        - Backing store is a very fast hard drive
    - If memory binding is not execution time, then process must be swapped back into same place in memory
        - PDP-11 Unix used swapping to relocate and resize processes
    - Make room for higher priority processes
    - Major time is transfer time - amount of memory swapped.

-Memory is always in short supply
-Not all routines are loaded when the program is loaded
    -Only loaded when needed
    -Some routines are rarely if ever used
    -Does not require any special support from OS
-Some Execution environments do support dynamic loading

Static linking is when all modules are linked together at compile time
Dynamic linking waits to run the linker for system libraries

Week 6:

Memory Management:

Introduction to Paging:

Concepts:

- **Paging:**
  - Why should memory have to be contiguous
  - Physical memory is divided into frames
    - typically 512 bytes to 8K in size
    - Linux is 4K
  - Logical memory is divided into pages◊same size as frames
  - If process needs n pages in the logical address space, we find n free frames in memory (physical address space)
    - no need to be contiguous
  - A table translates from each page to the appropriate frame
  - No external fragmentation, but still have internal fragmentation
  - Logical Address Space and Physical Address space may not be the same size!!
    - physical address may be larger(e.g. 32 bit logical, 40 bit physical)
    - physical address may be smaller(64 bit logical = 1.8e19 bytes)
  - Frame and page always the same size– always power a of 2
  - **frame table** - keeps track of allocated and free frames
  - I/O operations have to know memory layout
  - page table has to be switched during context switch
  - increase in context switch time
  - page tables are large, and usually kept in main memory
    - MMU has page table base register and length register for the process that is used to find the table for that process when its executed
    - Extra memory traffic
    - MMU must first use page number to find frame number, then access instruction or data in memory
    - A simple linear table adds one memory access for each data access
- **Page Tables:**
  - Memory is mapped by Page Tables
  - Each process can have its own logical address space
    - exception: Mac OS 7 had a single page table for all processes (no memory protection between processes)
  - Thus each process can have its own page table
  - Thus each process has its own mapping to physical memory
  - address generated by CPU is divided into two parts
    - page number(p) - index into page table

        - page offset(d) – location frames within the page
- **Paging - TLB:**
    - Want to minimize extra memory traffic of page tables
    - Small cache inside of MMU
        - Called TLB - translation look-aside buffer
        - TLB is associative memory that associates a page number with a frame number for a process (see diagram)
    - if the page not in TLB, called a miss
        - Requires one extra memory access cycles in linear page model (one for table, one for memory)
        - new page and frame added to TLB
        - performance sensitive to hit-ratio of the TLB
    - Some entries in TLB are not modifiable (kernel addresses)
    - Some TLBs support multiple processes by adding process ids to the TLB. This allows more than one process in TLB at a time
        - otherwise TLB must be flushed on each context switch
- **Memory Protection:**
    - Since memory is no longer contiguous, base + limit not sufficient for protection
        - could still be used on logical address side
    - memory belonging to other processes not in page table
        - thus not visible!!
    - OS may be in page table for quick access for traps and interrupts
        - add valid-invalid bit to page table
        - process can only access valid pages
- **Sharing Pages:**
    - Shared Libraries
    - Multiple invocations of a given program (e.g. shell, editor)
    - Contiguous memory allocation made sharing difficult
    - Shared code must be reentrant
        - Must not modify itself
        - Must also be position independent
    - Most data is not shared
        - however IPC Shared Segments are now easy!!
    - Page table entries for shared code and data in each process point to the common frames
    - Page table entries for private data point to different frames