Midterm One Notes

Lecture Note One: What is an Operating System

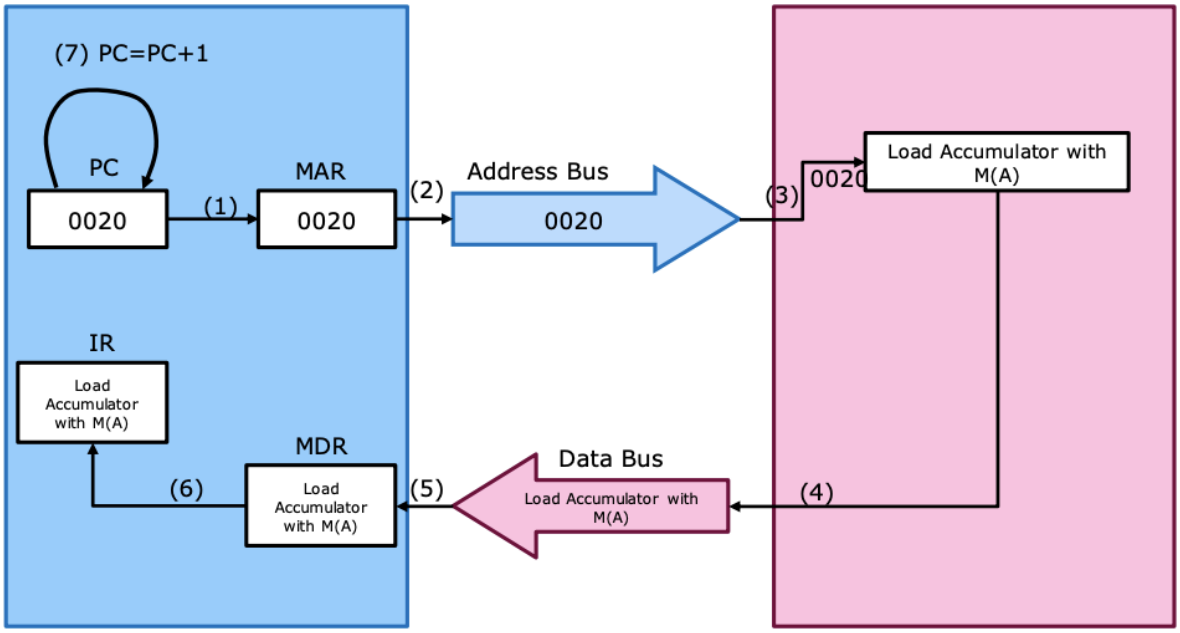
Operating System

- **Modern complex computer system:** processor, memory, I/O devices
- **Operating system:** protected software that provides interface between hardware and software.
- **Hardware:**
  - Physical devices: wires, power supply, IC chips (CPU, memory, I/O)
  - Micro-architecture: physical devices that are grouped together to form functional units
  - Machine language: executing some set of instructions
- **Von Neumann**
  - Von Neumann bottleneck: limitation on throughput caused by the standard personal computer architecture
    - **Throughput:** measure off how many units of information a system can process in a given amount of time
    - Since processor calculation speeds are much faster than data movement between memory and CPU -- causes a bottleneck
  - Programs and data are held in memory; the processor and memory are separate and data moves between the two
- Ways to consider the OS:
  1. **Extended Machine**: provides an interface between user and hahrdware, computer users can use the hardware (CPU, memory, I/O) without knowing the messy details
  2. **Resource Managers**: computer consists of CPU, RAM, and I/O devices
     1. Process management, memory management, file management, I/O management, deadlock management

A Computer System

- **Instruction cycle:** the CPU's main task is to execute instructions
- **Fetch Cycle:**
  1. Reading the address of the instruction in PC to be executed from the memory and
  2. Loading it into the IR

3. PC is modified to point to the next valid instruction



- **Execute Cycle**:
  1. Contents of the IR are decoded and executed
  2. The execution may result in a variety of actions (may be self contained, or involve iteraction with the memory and the ALU)

History of Operating Systems

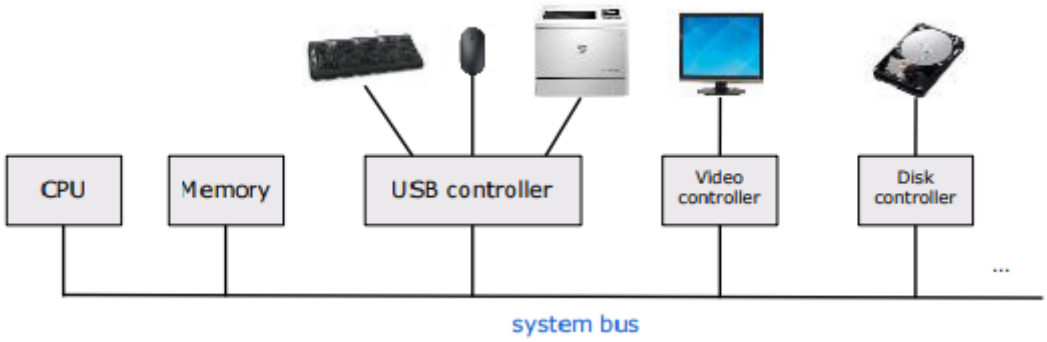
1. First generation (1945-1955): vacuum tubes and plugboards
   - Use vacuum tubes, all programming done in machine language
2. Second generation (1945-1965): transistors and batch system
   - Computers became more reliable
   - Batch system: programmer writes a program on paper, punch program onto cards, bring cards to input room, wait for output
     - Two IBM machines (1401 to read cards onto tape, put tape onto 7094 to do computing)
3. Third generation (1965-1980): IC and multiprogramming
   - Maintaining two computer products was expensive
     - IBM System/360 was made for scientific and commericial purposes
       - All software, including the OS, had to work on all models
     - First major computer to use IC
     - Improve CPU utilization: multiprogramming, spooling, time sharing
   - **Multiprogramming:** multiple jobs are loaded into RAM and run concurrently
     - Once CPU became availabale, one job from the ready queue is selected by the short term scheduler

        - OS needs to keep each job's current status in process table
    - **Spooling:** kind of buffering mechanism for a process in which data is temporarily held as a file to be used and executed by a device, program, or system
    - **Time sharing:** multiple terminals are sharing CPU time
4. Fourth generation (1980 - present): PC build with LSI, VLSI, and ULSI
    - Contains thousands of transistors
    - User types commands from the keyboard
    - Has a GUI
5. Fifth generation (1990 - present): mobile computers

Lecture Note Two: Computer System Organization

Computer System Architecture

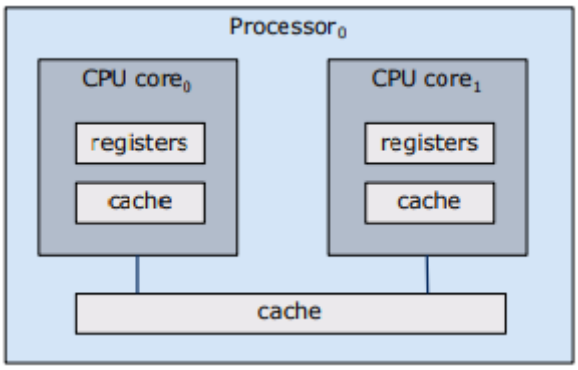
- **Modern general-purpose computer system:** one or more CPUs and a number of device controllers connected through a common bus that provides access between I/O devices and shared memory



General Architecture

- **Device controller**: Each **device controller** is in charge of a specific type of I/O device
    - Maintains some local buffer storage and a set of special purpose registers
    - Device controller is responsible for moving data between the peripheral devices that it controls and its local buffer storage
    - Each I/O device has a **device driver** for each device controller
- **CPU:** the brain of the computer
    - Each type of CPU has a specific instruction set that can be used for executing each instruction
    - All CPUs contain sets of registers and cache to hold instructions, key variables, and temporary results
        - PC, IR, data registers, stack pointer

- Fecthes instructions from memory and executes them
- **Basic instruction cycle executed by CPU:**
  1. Fetch data from memory to register
  2. Decode the instruction
  3. Execute the instruction
- **Components:**
  - ALU, control unit, cache, registers
  - Registers:
    - General registers (instruction, data)
    - Program counter (address of next instruction - virtual address)
    - Stack pointer (address of top of stack for currently running process)
    - Program status word (saves control information for each process)
- The OS must know the content of each register for a process
  - When a process stops running by changing states (from running to ready, running to block), the OS saves content for each register for the process in process table which needs to be used to finish its job
- CPUs might have multiple cores



- **Interrupts:** when an I/O device is ready to receive or send data through a bus, it interrupts the OS by sending a signal
  - For I/O, the device driver loads an instruction (read/write) to the device controller's IR
  - The controller starts the transfer of data from the device to its local bugger
  - Once the data transfer is compelte, check any error then the device controller informs the device driver that it is ready to transfer
  - The device driver gives control to the other parts of the OS
  - Hardware may trigger an interrupt at any time by sending a signal
  - Interrupts are a key part of how the OS and hardware interact
  - When the CPU is interrupted, it stops what it is doing and immediately transfers execution to a fixed location where the service routine for the interrupt is located

    - The interrupt service routine executes; on completion, it resumes the interrupted computation
    - The OS holds a table of pointers in low memory for holding addresses of interrupt service routines
    - **Interrupt Implementation:**
        - Hardware: interrupt-request line that senses after every CPU instruction
            - Types: nonmaskable (unrecoverable hardware error) and recoverable (used by device controllers to request service)
        - CPU reads interrupt number and jumps to the routine using the number as an index into the interrupt vector
- **Memory:** the CPU can load instructions only from RAM
    - **Storage Hierarchy:**
        1. Registers in CPU (volatile)
        2. Cache memory
        3. Main memory - RAM
            1. DRAM (dynamic RAM; capacitors)
            2. SRAM (static RAM; logic gates)
        4. Secondary memory (non-volatile)
            1. HDD, SSD, magnetic tape, USB
- **Input/Output Devices**
    - A large portion of OS code is dedicated to managing I/O
    - Form of interrupt-drive I/O is fine for moving small amounts of data, but has high overhead
        - Instead, use DMA (direct memory access) because it can handle I/O independent from the CPU
        - The CPU is freed from invovlement with data transfer, so it speeds up the overall computer operation
    - OS controls all I/O devices by:
      1. Issue commands (read/write) to devices
      2. Catch interrupts from devices (when devices are ready to send or receive data)
      3. Handle errors
    - OS provides interface between the devices and the rest of the system
    - Parts of I/O devices:
        1. Mechanical components (device itself)
        2. Electrical components (device controller)
        3. Device driver (software)
- Buses: common pathway between the CPU and peripheral devices

    - **Parallel buses:** use slots on the motherboard and provide multiple lines for data between the CPU and the peripheral card
        - Advantage: fast data communication
        - Disadvantage: short distance communication due to crosstalk between the parallel line
        - PCI and AGP
    - **Serial buses:** have external ports and a cable that plugs into them to connect to multiple devices
        - Disadvantage: slower data communication
        - Advantage: long distance communication
        - USB and FireWire
- **Single Processor System:** one CPU with a single processor core
    - **Core:** component that executes instructions and registers for storing data locally; typically capabale of executing a general instruction set
    - May have other special purpose processors, which run on a limited instruction set
    - The OS manages these special purpose processors by sending information about their next task and monitors their status
- **Multiprocessor Systems:**
    - Primary advantage: increased throughput. The speed up ratio with N processors is less than N, because of overhead due to communicatining with the CPU
    - Definition of multiprocessor also includes multicore systems
        - More effecient than multiple chips because on chip communication is faster than between chip
    - **System Types:**
        - **Symmetric Multiprocessing**: each CPU has its own set of registers and possibly cache memory; each CPU performs all tasks including OS functions and user processes.
            - All processors share a common physical memory
            - Additional CPUs will increase computing power but the system bus can act as a bottleneck
        - **Non-uniform memory access**: each CPU has a local memory that is accessed via a small, fast local bus
            - CPUs are connected and share one physical address space
            - Advtange: when a CPU accesses its local memory, its fast with no contention over system interconnect
            - Potential drawback: increased lentecy when a CPU must access remote memory, possible performance penalty

    - **Clustered systems**: two or more individual systems are connected locally and run as one system
        - High availablility service; service will continue if one or more systems fail
        - **Asymmetric clustering**: one machine is on hot stanby - if one node fails, the hot standby host becomes the active server
        - **Symmetric clustering**: two or more hosts are running applications and monitoring eachother; requires more than one application be available to run
- **Multiprogramming**: OS keeps several processes in memory simulatenously
    - The OS chooses one of the processes and lets it use CPU to execute its job
    - Eventualy, the process may have to wait for some task, such as I/O to complete
    - Multiprogramming increases CPU utilization by organizing programs so the CPU always has one to execute

Lecture Three: OS as a Resource Manager

- **Process Managment**: The OS is responsible for the following activities for process managment:
    1. Creating and deleting both user and system processes
    2. Scheduling processes and threads on the CPUs
    3. Suspending and resuming processes and threads
    4. Providing mechanisms for process synchronization (semaphore, mutex, conditional variable)
    5. Providing mechanisms for process communication (PIPE, message queue, shared memory, FIFO, socket)
- **Memory Management:** for a program to be executed, it must be mapped to absolute addresses and loaded into memory
    - As the program executes, it accesses program instructions and data from memory by generating them
    - CPU reads instructions from main memory during the instruction-fetch cycle and both reads/writres data from main memory during the data-fetch cycle
    - The OS is responsible for the following activities for memory managment:
        1. Keeping trak of what parts of memory are currently being used and which process is using them
        2. Allocating and deallocating memory space as needed
        3. Deciding which processes (or parts of processes) and data to move in/out of memory

- **FIle Mangement**: OS provides a logical, uniform view of information storage for users to save a file
  - The OS implements the abstract concept of a file by managing mass storage media and the devices that control them
  - Files are normally organized into directories to make them easier to use
  - The OS is responsible for the following activities for file management:
    1. Creating and deleting files
    2. Creating and deleting directories to organize files
    3. Supporting primitives for manipulating files and directories
    4. Mapping files onto mass sotrage
    5. Backing up files on stable (nonvolatile) storage media
- **Mass Storage Management**: modern computer system use multiple types of secondary storages
  - The OS is responsible for the following activities for secondary storage management
    1. Mounting and unmounted
    2. Free space managment
    3. Storage allocation
    4. Disk scheduling
    5. Partitioning
    6. Protection
- **Cache Management**
  - Hardware or software component that stores data which might be used against soon
  - Cache hit: the requested data can be found in a cache
  - Cache management is an important design problem
  - The movement of information between levels of a storage hierachy may be either implicit or explicit; depends on hardware diesgn
    - Data transfer from cache to CPU and registers is usually a hardware fucntion
    - Data transfer from disk to memory is usually controlled by the OS
- **Deadlock Management**
  - Deadlocks between processes happen since limited number of resources must be shared
  - Processes are sharing resources for finishing their job
  - Four strategies for dealing with deadlock
    1. Ignore
    2. Detection and recover

3. Dynamic avoidance by careful allocation
4. Prevention

- Four necessary conditions to avoid deadlock:
  1. Mutual exclusion
  2. Circular wait
  3. Hold and wait
  4. No preemptve

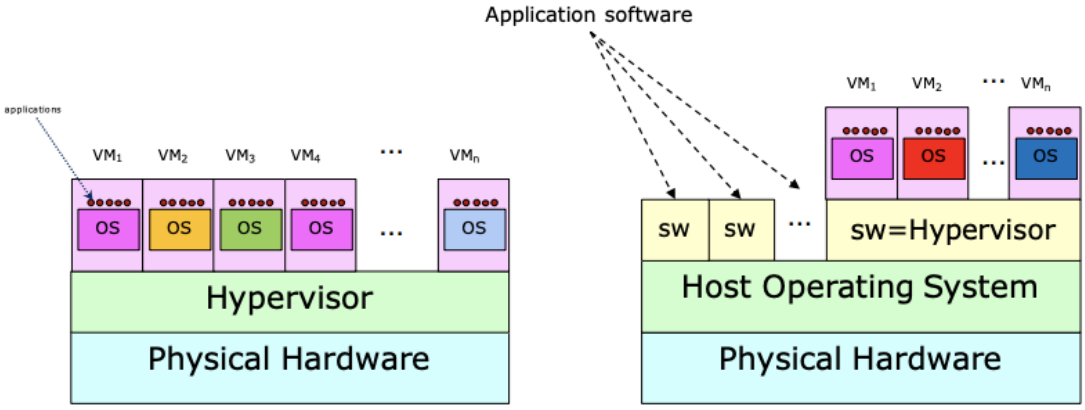
- **Input/Output**:
  - OS manages all kinds of I/O devices such as keyboards, monitors, printers
  - I/O subsystems:
    - A memory mangement components (buffering, spooling, caching)
    - General device drivers
    - Drivers for specific hardware
    - Processor for specific hardware
  - OS controls I/O devices by:
    1. Issue commands to devices
    2. Catch interrupts from devices
    3. Handle errors

Operating System Structures

- **Monolithic**: written as a collection of procedures, each of which can call any of the others whenever it needs to
  - Each procedure in the system as a well-defined interface in tersm of parameters and results, each one is free to call any other one
  - Possible to have some structure for a monolihtic system
    - Main program, service functions, utility functions
- **Layered Systems:** OS is divided into several layers and each later works on different rules:
  1. Layer 0: process management
  2. Layer 1: memory management
  3. Layer 2: inter-process managment
  4. Layer 3: input/output managment
  5. Layer 4: user program
  6. Layer 5: systerm operator process
- **Microkernels**: with the layered approacj, the designers have a choice where to draw the kernel-user boundary
  - Achieve high reliability by splitting the OS up into small well-defined module

- Only one module runs in kernel mode and the rest run as users mode

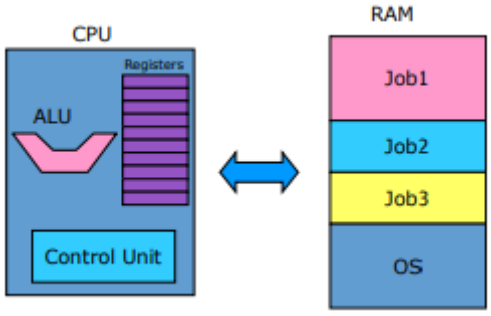
- **Virtual Machine**: runs on bare hardware and does multiprogramming by providing several virtual machines
  - Each virtual machine copies the bare heardware, including kernel/user mode, I/)
  - Different VMs can run on different OS



Type 1 (bare metal hypervisor) Type 2 (Hosted hypervisor)

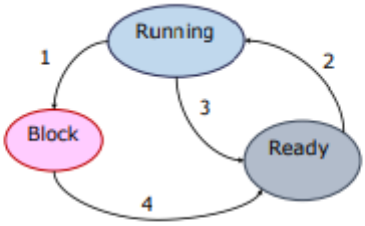
Lecture Note Four: Processes

- **Process**: program in execution
  - **Memory layout of a process**:
    1. Text section (executable code)
    2. Data section (global variables)
    3. Heap (used for dynamic allocation)
    4. Stack section (temporary data storage for local variables - function parameters, return addresses, local variables)
- **The process:** assuming that we have one CPU (single core) in the system
  - **Multiprogramming**: several jobs are loaded into memory, OS simulates pseudo parallelism through virtual memory
  - The OS schedules CPU time for processes by switching from one process to another based on the *scheduling algorithm* and *process state*



- **The Process Model**:
  - **Real model**: multiprogramming
  - **Conceptual (virtual) model**: each process has its own virtual CPU (ALU, PC, registers, stack pointer) and RAM
    - Virtual Machine: each OS runs its own machine
    - Virtual Memory: each process has its own memory
    - Virtual process model: each process runs on its own machine
  - A process can hold CPU during its time term (amount of time allocated to process, not uniform)
    - Time term might be calculated based on the type of jobs (I/O bounded, CPU bounded, or priority)
- **Process Creation**:
  1. System initialization: when an OS is booted
  2. Created by a running process using a system call
  3. Created by system user by executing a program
  4. Initiation of a batch job
  - Child processes: once a child process is created, both parent and child have their own distinct address, requiring inter-process communication for sharing resources.
  - systemd pid = 1
  - When a process creates a new process, two possibilities for execution exist:
    - The parent continues to execute concurrently with its children by using waitpid()
    - The parent waits for some or all of its children to finish executing by wait()
    - Two address-space possibilities:
      - The child is a duplicate of the parent process
      - Child process has a new program loaded using exec into it and run
- **Process Termination**: process may terminate due to the following conditions:
  - **Voluntary**:
    - Normal exit (process finishes its job)
    - Error exit (process itself discoveres a fatal error)
  - **Involuntary**:
    - Fatal error (error cauzed by the process)
    - Killed by another process (when a deadlock is discovered)
  - **Zombie process**: process that has terminated, but whose parent has not yet called wait()
- **Process States**:

- **Running state:** process is currently being executed by using the CPU
- **Block state**: process is waiting for some event to occur (I/O, signal)
- **Ready state**: process is ready to use CPU but it is not currently available



1. Process blocks for input
2. Short-Term Scheduler picks a process since CPU become available
3. A process time out its time term
4. Input becomes available

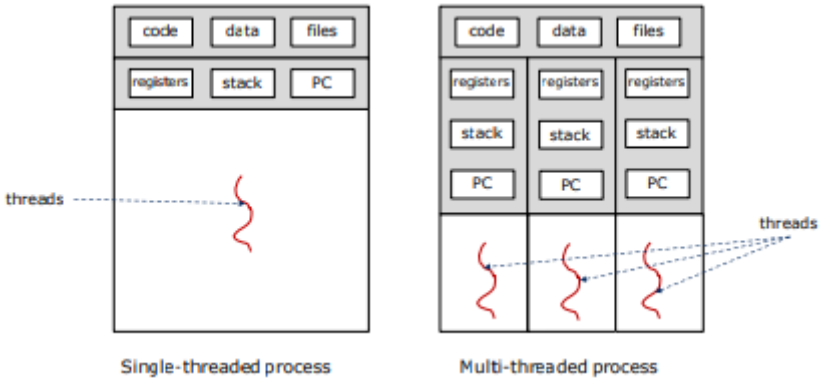
- **Process Table**: when a process is created, the OS stores its run time information in a process table
  - Contains process state, program counter, contents of CPU registers, CPU scheduling information, memory management information, accounting information, and I/O status
- **Multiple Threads**: most OS have processes that have multiple threads of execution. which allows them to perform more than one task at a time; beneficial on multicore systems, since multiple threads can run in parallel.
- **Process Scheduling**
  - **Scheduling Queues**: when the CPU becomes available, the short term scheduler seelcts a process from the ready queue
    - Objective of multiprogramming is to maximize CPU utilitzation
    - Two types of queues to hold pointers to process tables for processes in the block and ready state:
      - **Ready queue** and **wait queue**
      - Generally stored as a linked list
    - Events that can occur once a CPU is allocated and executing:
      1. Process could request I/O and be placed in an I/O wait queue
      2. Process could create a new child process then be placed into wait queue
      3. Process could be forcibly removed as the result of an interrupt
      4. Process could try and a down semaphore whose value is 0, be placed into semaphore wait queue
    - Scheduling algorithms: shortest job first, shortest remaining time, round robin, priority queue, guaranteed scheduling, lottery scheduling
  - **Context Switch**: interrupts cause the OS to change a CPU from the current task to run a kernel routine through a context switch, which is performing a state save of

the current process and a state load of a process from its process table
    - Pure overhead (CPU must be idle)

Lecture Five: Interprocess Communication

- Reasons for allowing process cooperation:
    1. Information sharing
    2. Computational speed up
    3. Modularity
- Fundamental models of interprocess communication:
    - Shared memory: region of memory is shared, OS is only involved in creation but not in synchronization/mutual exclusion
        - Faster than message passing, since shared memory is in user's space after the initial set-up system calls
        - All memory accesses are treated as routine memory accesses, without kernel's assistance; allows processes to access each other's memory without the OS getting involved
        - Other processes that wish to use the shared-memory segment must attach it to their address spaces by key values
        - **Producer-Consumer Problem with Shared Memory**:
            - Has a buffer of items that can be refilled by producer and emptied by consumer
            - Buffer resides in shared memory
            - Producer/consumer must be synchronized so the producer doesn't try and overfill the buffer and the consumer doesn't try to consumer from an empty buffer
    - **Message passing**: communication takes place by means of messages exchanged, easy to implement in large systems, OS involved with creation and synchronization
        - Message queue, FIFO, socket
        - Allows processes to communicate and synchronize their actions without sharing the same address space
        - A communication link must exist between the processes
            - Direct communication requires each process to know the end point address for sending/receiving messages, while with indirect communication, messages are sent/received from mailboxes
                - Direct communication can be full duplex (both processes have endpoint addresses) or half duplex (send to specific address, receive from any)

        - Indirect communication allows a process to communicate with another process via a bunch of different mailboxes, but two processes can only communicate if they have a shared mailbox
    - **Synchronization**: can be blocking (synchronous - sender/receiver are blocked until message is received/sent) or nonblocking (asynchronous - sender sends message then continues and receiver either receives a valid or null message)
        - Rendezvous: both sender and receiver are blocking
    - Buffering:
        - Zero capacity (no messages are queued on a link)
        - Bounded capacity (finite length of n messages)
        - Unbounded capacity (infinite length)
- **Overview of Threads**
    - Each thread runs on a different part of a process, with a thread ID, program counter, register set, and a stack.
        - Shares with other threads belonging to the same process its code section, data section, and OS resources, such as open files and signals



    - Most OS kernels are multithreaded
    - **Benefits with Threads**:
        - Resource sharing (share memory and resources of the process that they belong too; allows an application to have several different threads of activity within the same address space)
        - Economy (more economical to create and context-switch threads)
        - Responsiveness (program may continue to run even if part of a program which is run by a thread is blocked)
        - Scalability (benefits are even greater in a multiprocessor architecture)
    - **Multicore Programming with Threads**: provides improved concurrency
        - Challenges for programming in multicore systems:
            1. Identifiying tasks
            2. Balance (balanced worked load)

3. Data splitting (data is accessed and manipulated into separate tasks)
4. Data dependency
5. Testing and debugging

- Types of parallelism
  1. Data parallelism (distributing subsets of the same data across multiple computing cores)
  2. Task parallelism (distributing tasks/threads across multiple computing cores)

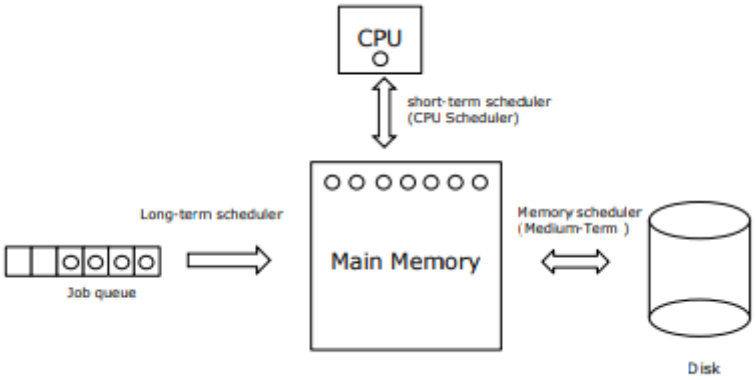
Lecture Six: Thread Implementation

- **User level threads:** kernel is not aware of threads
  - Thread library controls creations, destruction and scheduling
  - Application starts with single thread
  - **Advtantage**:
    1. Thread switching does not requre kernel mode privileges
    2. User level threads can run on any OS
    3. Scheduling can be application specific in user level threads
    4. User level threads are fast to create and manage
  - **Disadvantage**:
    1. Most system calls are blocked
    2. Multithreaded applications cannot take advantage of multiprocessing
- Kernel level threads: thread managment is done by OS
  - Kernel maintains context information for the process as a whole
  - Scheduling by the kernel is done on a thread basis
  - Kernel performs thread creation, scheduling, and management in kernel space
  - Advantages:
    1. Kernel can schedule multiple threads from the same process on multiple processes
    2. If one thread on a process is blocked, the kernel can schedule another thread on the same process
    3. Kernel routines themselves can be multithreaded
  - Disadvantages
    1. Kernel threads are general slower to create and manage than the user threads
    2. Transfer of control from one thread to another within the same process requires a mode switch
- **Multithreading models**

- **Many to one**: maps many user-level threads to one kernel threads
  - Thread managment is done by the thread library is user space, efficient
  - When a thread makes a blocking system call, the entire process will be blocked
  - Only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multiprocessors
- **One to one**: mmaps each user thread to a kernel thread
  - More concurrency than many-to-one model by allowing another thread to run when a thread makes a blocking system call
  - Allows multiple threads to run in parallel
  - A large number of kernel threads may burden the performance of a system
- **Many to many**: multiplexes any number of user threads onto an equal or small number of kernel threads
  - Allows developers to create as many user threads as needed
  - Provides best accuracy on concurrency and when a thread performs a blocking call, the kernel can schedule another thread for execution

Lecture Seven: CPU Scheduling

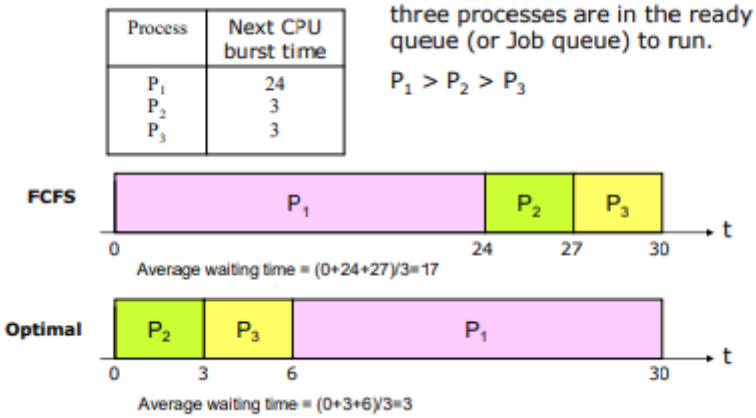
- **Three Level Scheduler**:
  - **Long Term Scheduler**: selects a process from the job queue and loads it into memory for execution
  - **Short Term Scheduler**: selects a process from the ready queue and allocates it to the CPU
  - **Memory Scheduler**: schedules which process is in memory and in disk



- Effective CPU scheduler is essential: process switching is expensive:
  1. All information for the blocked (became ready state or block state) must be saved (state, register, stack pointer, PC)
  2. The memory map must be saved
  3. New process is selected by the scheduler

4. Information for the new process must be loaded
5. Start run new process

- When to make a scheduling decisions:
  - Short term scheduler:
    - When a process switches from running state to blocked state (nonpreemptive)
    - When a process switches from running state to ready state (preemptive)
    - When a process switches from blocked state to ready state (preemptive)
  - Long term scheduler:
    - When a process terminates its job, memory space becomes available (preemptive)
  - Under nonpreemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases it by terminating or by switching to the block state
- **Scheduling Criteria**
  - CPU utilization: want to keep the CPU as busy as possible
  - Throughput: number of processes completed per time unit
  - Turnaround time: interval from the time of submission of a process to the time of completion
  - Waiting time: the sum of the periods spent waiting in the ready queue
  - Response time: the time from submission of a request until the first response is produced
- **First Come, First Serve**
  - Simplest, nonpreemptive
  - Managed by FIFO queue; when a process enters the ready queue, its process table is linked to the tail of the queue
  - When the CPU is free, it is allocated to the process at the head of the queue
  - Drawback: average waiting time is long



- **Shortest Job First**

- Associates each process with the length of the process's next CPU burst
  - When the CPU is available, it is assigned to the process that has the smallest next CPU burst
  - FCFS is used to break ties
- Provably optimal, gives minimal average waiting time
- Cannot be implemented at the level of CPU scheduling since there is no way to know the length of the next CPU burst.
  - Approach: try to approximate based off of previous CPU burst time

  $T_{n+1} = a * t_n + (1 - a) * T_n$

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

  t_n :length of the n^{th} CPU burst, τ_{n+1} : predicted next CPU burst

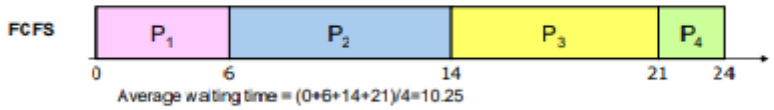
  - $\alpha = 0$: $\tau_{n+1} = \tau_n$:recent history has no effect
  - $\alpha = 1$: $\tau_{n+1} = t_n$:the most recent CPU burst matters.
  - With $\alpha = ½$ and $\tau_0 = 10$, we can estimate sequence of predicted next CPU burst

CPU burst ($t_{i\geq 0}$)		6	4	6	4	13	13	13	..
Predicted τ_n	10	8	6	6	5	9	11	12	..

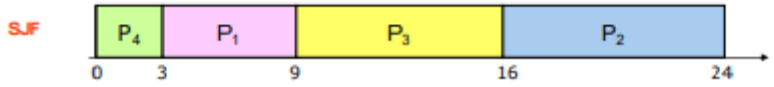
Process	Next CPU burst time
P_1	6 (msec)
P_2	8(msec)
P_3	7(msec)
P_4	3(msec)

Lets assume arriving times as (for FCFS)

$P_1 > P_2 > P_3 > P_4$



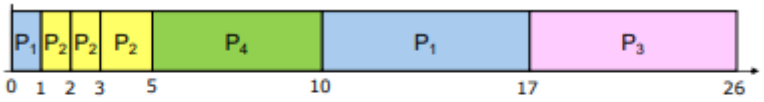
Average waiting time = (0+6+14+21)/4=10.25



Average waiting time = (0+3+9+16)/4=7

- **Shortest Time Remaining**: preemptive version of the SJF
  - When a new process arrives at the ready queue while the previous process is executing, the CPU is preempted from the process and changes to the ready state
  - Short term scheduler selects a process from the ready queue; newly arrived process might be shorter than what is left of the current executing process

Process	Arrival Time	CPU time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5



Average waiting time = ((10 – 1) + 0 + (17 – 2) + (5 – 3))/4 = 6.5

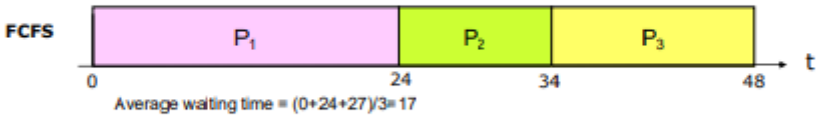
Average Turnaround time = ((17- 0) + (5-1) + (26-2)+(10-3))/4 = 13

- **Round Robin**: similar to FCFS scheduling, but preemption is added to enable to system to switch between processes
  - A small unit of time (time quantum) is defined
  - No process is allocated the CPU for more than 1 time quantum in a row
  - Implemented in a circular queue
  - Average waiting time is long

Process	Next CPU burst time
P_1	24
P_2	10
P_3	14

three processes are in the ready queue to run. With time quantum 4 unit

$P_1 > P_2 > P_3$



Average waiting time = (0+24+27)/3= 17

Average waiting time = ((8+8+6+2)+(4+8+8)+(8+8+6+4))/3=23.3

- **Priority Scheduling**: a priority is assigned with each process, the CPU is allocated to the process with the highest priority; each priority processes are done in FCFS basis
  - Variations: non preemptive priority scheduling, preemptive priority scheduling, priority scheduling with round robin between processes with the same priority
  - Problem: starvation of lower priority processes, who never get CPU time; technique called aging, which gradually increases the priority of processes that have waited in the system for a long time

Process	Arrival Time	priority	CPU time
P_1	0	1	8
P_2	0	3	4
P_3	0	4	9
P_4	0	2	5

High number has high priority

$P_3 > P_2 > P_4 > P_1$

Average waiting time = (18 + 9 + 0 + 13)/4 = 10

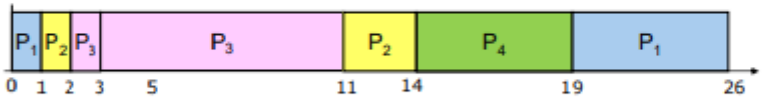
Average Turnaround time = (26 + 13 + 9 + 18)/4 = 16.5

- Each processes has different priorities
- All processes are arrived at time T

Process	Arrival Time	priority	CPU time
P_1	0	1	8
P_2	1	3	4
P_3	2	4	9
P_4	3	2	5

High number has high priority

$P_3 > P_2 > P_4 > P_1$



Average waiting time = ((19 – 1) + (11 – 2) + 0 + (14 – 3))/4 = 9.5

Average Turnaround time = ((26 – 0) + (14 – 1) + (11 – 2) + (19 – 3))/4 =16

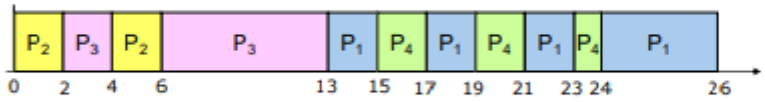
- Each processes has different priorities
- Each process arrived at different time unit

Process	Arrival Time	priority	CPU time
P_1	0	1	8
P_2	0	3	4
P_3	0	3	9
P_4	0	1	5

High number has high priority

$P_2 = P_3 > P_4 = P_1$

R-R time quantum = 2 unit



Average waiting time = ((13+2+2+2+1)+2+(2+2)+(15+2+2)+/4 =

Average Turnaround time = (26 + 6 + 13+ 24)/4 =

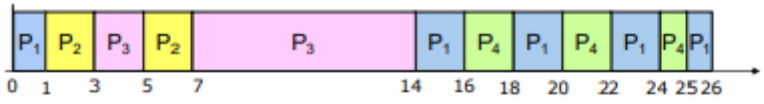
- Some processes might have same priorities
- Every processes are arrived at time T

Process	Arrival Time	priority	CPU time
P_1	0	1	8
P_2	1	3	4
P_3	2	3	9
P_4	3	1	5

High number has high priority

$P_2 = P_3 > P_4 = P_1$

R-R time quantum = 2 unit



Average waiting time = ((13+2+2+2+1)+2+(1+2)+(13+2+2)+/4 =

Average Turnaround time = (26 +(7 – 1) +(14 -2) + (25 -3))/4 =

- Some processes might have same priorities
- Each processes are arrived at different time

- **Priority Scheduling with Multilevel Queue**:

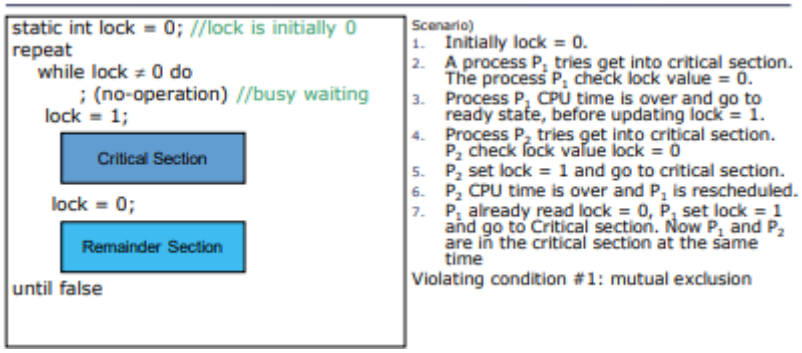
    - Priority scheduling with a singlue queue has O(n) search time; with multilevel queue, each level maintains a pointer to process tables with the same priority
    - Works well when priority scheduling is combined with round robin
    - Processes are permanently assigned to a queue when they enter the system, but with a **multilevel feedback queue**, processes can move in between queues
- **Guaranteed Scheduling**:
    - Guarantees fairness by monitoring the amount of CPU time spent by each user and allocating resources accordingly
    - System must keep track of how much CPU time each process has had since its creation
    - For n processes, each process will receive $1/n$ of the CPU
- **Lottery Scheduling**
    - When a program starts, a lottery ticket is given to each process; whenever a scheduling decision has to be made, a lottery ticket is selected at random and the process holding that ticket gets the resources

Lecture Ten: Interprocess Communication

- **Three Issues with Interprocess Communication**:
    1. How can one process pass information to another? (Communication between processes)
        - Shared memory, message queue, FIFO, PIPE, socket
    2. How to make sure two or more processes do not get into the critical section (mutual exclusion)
        - Mutex, semaphore
    3. Proper sequencing when dependencies are present (synchronization)
- **Race condition**: where two or more processes are reading/writing to some shared data and the final result depends on who runs precisely when
    - How to avoid the race condition? A solution should have the following four conditions:
        1. No two processes may be simulatenously inside their critical sections (mutual exclusion)
        2. No process running outside its critical section may block other processes
        3. No process should have to wait forever to enter the critical region
        4. No assumptions may be made about speed or the number of CPUS
- **Critical section**: part of the program where the shared memory is accessed
- **Mutual exclusion in a critical section an avoid race conditions**.
    - Two approaches for mutual exclusion solutions:

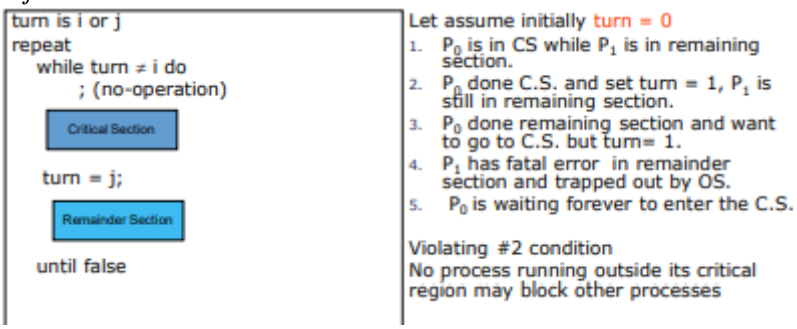
1. Busy wait (a process will wait until the resource becomes available or CPU time term expired)
2. Sleep and wake up (a process checks a resource, if it is not available, go to sleep. When the resource becomes available, the process will be woken up by the system)

- **Mutual Exclusion with Busy Waiting**
  - Each process has a time term; a process keeps checking the possibility to get into the critical section
  - Types: disabling interrupts (non preemptive kernel), lock variables, strict alternation, Peterson's solution, test and set lock (hardware)
  - **Disabling interrupt**:
    - Once the process gets into the critical section, interrupts set to disable
    - Other processes cannot get CPU time until the process finishes its job in the critical section
    - Since each user process has power to control the interrupt, it might cause the end og the system
    - EXAMPLE:
      1. A process gets into the critical section
      2. Disables all interrupts (all other processes are sleeping)
      3. The process is blocked outside the critical section just before it can enable all interrupts again, causing end of system
  - **Lock variable**:
    - A process can enter in its critical section when lock = 0
    - Lock = 0 means that no process is currently running in the critical section, set lock = 1 and enter the critical section
    - Once a process finishes its job in critical section, set lock = 0 and let other processes in the critical section
    - Lock = 1 means that there is a process running in the critical section, a process does busy waiting until lock = 0

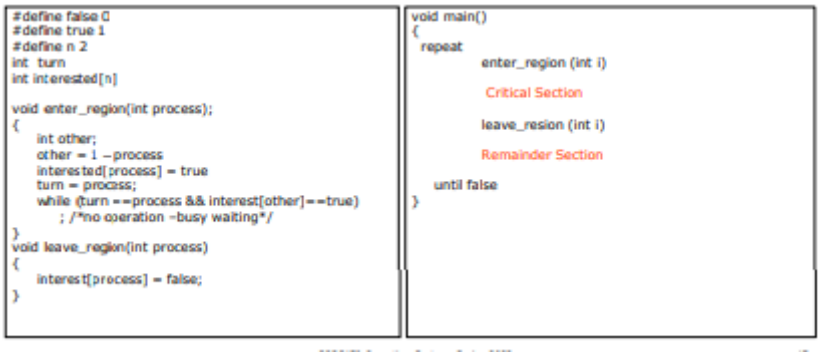


  - **Strict alternation**:
    - Variables can be i or j.

- If turn = i, process P_i can go into the critical section
- Once P_i finishes its job in the critical section, P_i sets turn = j and lets process P_j into the critical section



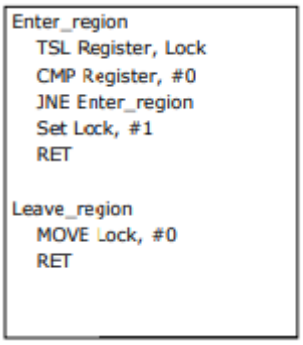
- **Peterson's Solution**
  - Provides a good algorithmic description of solving the critical-section problem
  - Peterson's solution is restricted to two or more processes that alternate execution between their critical sections and remainder section
  - The processes are numbered P_0 and P_1
    - When presenting P_i and P_j , and $j = 1 - i$

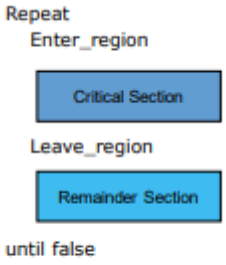


1. Initially, neither process is in the critical section
2. A process P_0 call enter_region (0)
   a) Set interested[0] = true;
   b) Set turn = 0
3. go to critical section
4. the process P_1 call enter_region(1) to get into its critical section
   a) set interested[1] = true;
   b) set turn = 1;
5. since interested[0] = true, it is keep looping for interest [0] = false
6. finally process P_0 finish its critical section and call leave_region(0)
   1. set interested[0] = false
7. now P_1 find out interest[0] = false, P_1 goes to its critical section

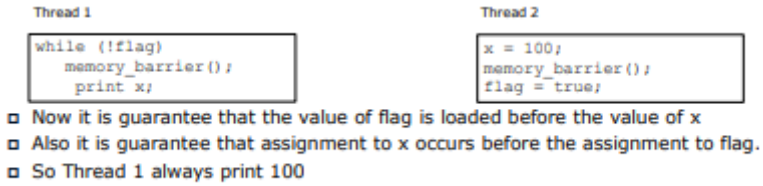
- **Test and Set Lock**:
  - The operations of reading the lock and storing into registers are guaranteed to be individuable
  - Read the content at the memory address of lock into register RX; store a non zero value at the memory address of lock
  - When lock = 0, any process may set lock = 1 by using the TSL instruction and go to its critical section; when the process finishes its critical section,

set lock = 0 by using the original move instruction





- **Memory Barriers**: hardware solution
  - Two general memory models:
    1. Strongly ordered memory: a memory modification on one processor is immediately visible to all other processors
       - Computer architectures provide instructions that can force any changes in memory to be propagated to all other processors
    2. Weakly ordered memory: a memory modification on one processor may not be immediately visible to other processors
  - A memory barrier causes the CPU to enforce an ordering contract on memory operations issued before and after the barrier instruction

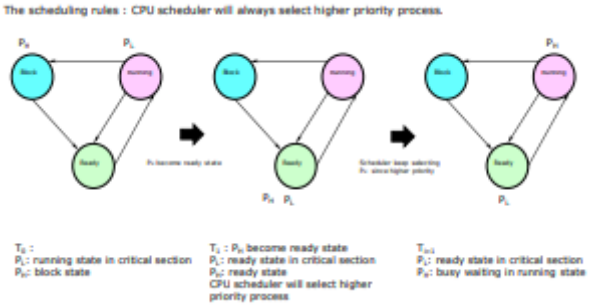


- **Atomic Variables**
  - When a thread/process performs an atomic operation, the oother threads see it as its happening instantaneously
  - Relatively quick compared to locks
  - Only do a limited set of operations; cannot synthesize more complicated operations efficiently

- **Priority Inversion Problem**:
  - A computer with two processes, P_h with high priorities and P_l with lower priority. The scheduling rules state that P_h gets CPU time whenever it is in the ready state

    0. At time T_0, P_l is in the critical section and P_h is in the block state
    1. At time T_1, P_h change state from blocked to ready and try to enter the critical section. P_l is in the critical section
    2. Based on the scheduling rule, the short term scheduler selects P_h, P_h holds the CPU and tries to enter the critical section

3. Since P_l is in critical section, P_h runs busy waiting outside the critical section forever since P_l does not have a chance to get CPU time to finish its critical section



Lecture Eleven: Mutual Exclusion with Sleep and Wakeup

- **Sleep and Wakeup**: a process checks a resource (critical section), if not available, goes to sleep. When the resource is available, the process will be woken up by the system or the process releasing the resources
- **Producer-Consumer Problem**:
  - Description: two processes share a common, fixed-size buffer that the producer puts information into and the consumer takes information out of
  - Trouble arises: when the producer wants to put a new item in but its full; when the consumer wants to remove an item from the buffer but it is empty
    - Solution: Producer goes to sleep and is awakened by the consumer when it has removed items; consumer goes to sleep and is woken up by the produce when it places items into the buffer
  - Race condition for the producer-consumer problem
    1. Initially the buffer is empty (count = 0)
    2. The consumer reads count = 0, but since consumer's time slice is over, CPU assigns time to producer
    3. The producer produces item and checks count = 0, so it places item in buffer and increases count. It then wakes up the consumer, but since it is not sleeping yet, it misses the call
    4. Producer's CPU time is over, so the scheduler assigns CPU time to the consumer
    5. The consumer already read count = 0, so it goes to sleep
    6. The producer keeps producing items until the buffer is full, producer goes to sleep

- The same can be reproduced for the producer if the buffer is initially full

```
#define N 100 //buffer size
int count = 0;//# of item
void producer()
{
   int item
   while (true)
   {
        item = produce_item();
        if (count == N)
             sleep();
        insert_item(item)
        count = count + 1;
        if (count ==1)
             wakeup(consumer);
   }
}
```

```
void consumer()
{
   int item;
   while(true)
   {
        if (count == 0)
             sleep();
        item = remove_item();
        count = count - 1;
        if (count == N - 1)
            wakeup(producer);
        consume_item(item);
   }
}
```

- **Semaphores**: an integer variable which could have the value 0 (no wakeups saved) or +i (wake ups are pending)
  - Can only be accessed through two standard atomic operations (down or up)
  - Modification to the integer value of the semaphore in the down and up operations are indivisibly, which means that when a process is modifying the semaphore value, no process can simultaneously modify that same semaphore value
  - The OS briefly disables all interrupts while it is testing the semapgore, updating it, and putting the process to sleep

```
void down (S)
{
   if S == 0
   {
        1. Add this process to the sleeping list
        2. block;
   }
   S = S – 1;
}
```

```
void up (S)
{
   S = S + 1;
   If S = 1
   {
        1. choose one process P from the sleeping list or let them move to ready state
        2. wakeup(P) to finish down operation
   }
}
```

  - **Proper usage of semaphores**

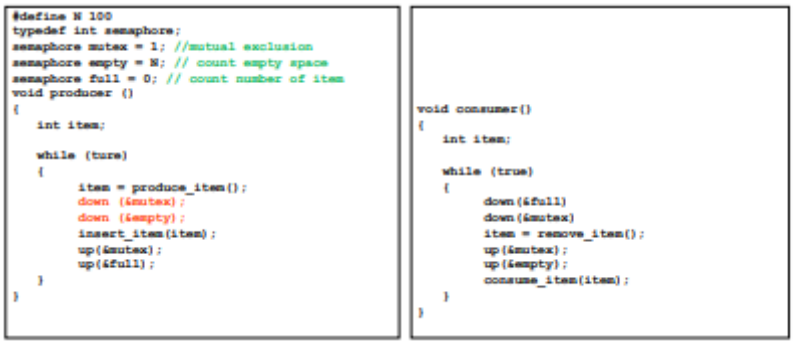
```
#define N 100
typedef int semaphore;
semaphore mutex = 1;  //mutual exclusion
semaphore empty = N; // count empty space
semaphore full = 0; // count number of item
void producer ()
{
   int item;

   while (ture)
   {
        item = produce_item();
        down (&empty);
        down (&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
   }
}
```

```
void consumer()
{
   int item;

   while (true)
   {
        down(&full)
        down(&mutex)
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
   }
}
```

- **Careless usage of semaphores**



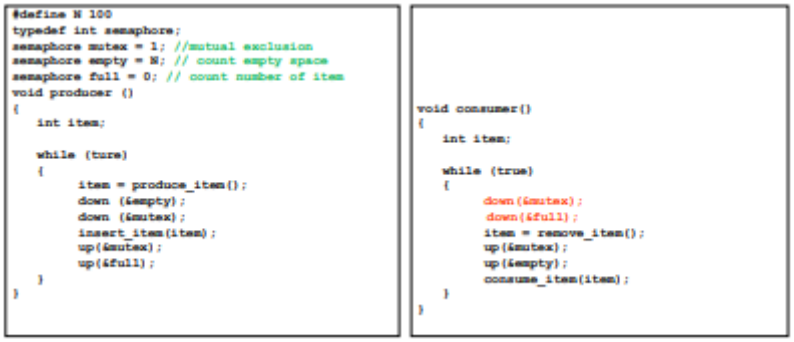
```
#define N 100
typedef int semaphore;
semaphore mutex = 1; //mutual exclusion
semaphore empty = N; // count empty space
semaphore full = 0; // count number of item
void producer ()
{
   int item;

   while (ture)
   {
        item = produce_item();
        down (&mutex);
        down (&empty);
        insert_item(item);
        up(&mutex);
        up(&full);
   }
}
```

```
void consumer()
{
   int item;

   while (true)
   {
        down(&full)
        down(&mutex)
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
   }
}
```

  - In producer, the mutex is downed before the empty semaphore. If empty = 0, meaning the buffer is full, and mutex = 1, then the mutex is downed, then empty is downed. When empty is downed, it will put the producer to sleep before it can up the mutex. When the consumer has CPU time, it will down the full, then try and down the mutex, but since it is already 0, it will be put to sleep. Both processes will be forever sleeping.



```
#define N 100
typedef int semaphore;
semaphore mutex = 1; //mutual exclusion
semaphore empty = N; // count empty space
semaphore full = 0; // count number of item
void producer ()
{
   int item;

   while (ture)
   {
        item = produce_item();
        down (&empty);
        down (&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
   }
}
```

```
void consumer()
{
   int item;

   while (true)
   {
        down(&mutex);
        down(&full);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
   }
}
```

  - In consumer, the mutex is downed before the full semaphore. If full = 0 and mutex = 1, then the mutex is downed. Following that, the full is downed, which puts the consumer to sleep. The producer's turn is next, but when it does to down the mutex after downing the full, it cannot since the mutex is still 0. So, it goes to sleep and both processes will be sleeping.
- **Dining Philosophers Problem:** 5 philosophers are sitting at the table, where there are 5 chopsticks. They alternate between thinking and eating, but need two chopsticks to eat. When they are done eating, they put down their chopsticks. They will never give up a chopstick until they hold two and eat.
- **Readers-Writers Problem**: process reader R and writers W are sharing resources at one time; only one process can access the shared resources at any time.
  - Multiple readers can read at the same time
  - Readers cannot jump ahead of writers
- **Mutexes**:
  - When a semaphore's ability to count is not needed, the simplified version of the semaphore called a mutex is used
  - A variable that can be in one of two states: unlocked (0) and locked (1)

- **Monitor**: high level synchronizing primitive
  - A collection of procdures, variables, and data structures that are all grouped together in a special kind of module
  - One only process can be active in a monitor at any given instance
  - When a process calls a procedure inside a monitor, check whether the process is active within a monitor or not, and if it is, the calling process will be suspended until the other process has left the monitor
  - Uses **conditional variables**:
    - Two operations on a conditional variable (wait, signal)
    - When a monitor procedure discovers it cannot continue, it does wait on some condition variable, which causes the calling process to block; the other process can wake up its sleeping partner by doing a signal on the condition variable its partner is waiting on

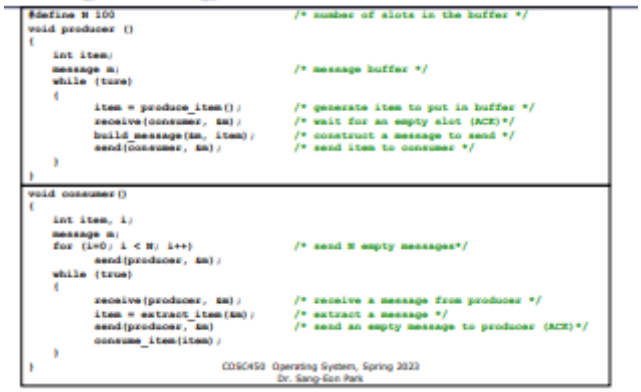
```
monitor ProducerConsumer
condition full, empty;
integer count;
procedure insert (item: integer);
begin
     if count = N then
          wait (full);
     insert_item(item);
     count :=count + 1;
     if count = 1 then
          signal (empty)
end;

function remove: integer;
begin
     if count = 0 then
          wait (empty);
     remove = remove_item;
     count := count - 1;
     if count = N - 1 then
          signal (full);
end;
count := 0;
end monitor
```

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

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

- **Message Passing**:
  - A method of interprocess communication by using two primitive system calls (send, receive)
  - Usually used between processes located in different systems
  - Can used sequence numbers and ACK in order to avoid duplicates/lost messages



```
#define N 100                                   /* number of slots in the buffer */
void producer ()
{
    int item;
    message m;                                  /* message buffer */
    while (true)
    {
         item = produce_item();                 /* generate item to put in buffer */
         receive(consumer, &m);                 /* wait for an empty slot (ACK)*/
         build_message(&m, item);               /* construct a message to send */
         send(consumer, &m);                    /* send item to consumer */
    }
}
```

```
void consumer()
{
    int item, i;
    message m;
    for (i=0; i < N; i++)                       /* send N empty messages*/
         send(producer, &m);
    while (true)
    {
         receive(producer, &m);                 /* receive a message from producer */
         item = extract_item(&m);               /* extract a message */
         send(producer, &m)                     /* send an empty message to producer (ACK)*/
         consume_item(item);
    }
}
```


Lecture Twelve: Memory Managment

Memory Management

- **Ideal Memory**: infinitely large, fast, non-volatile

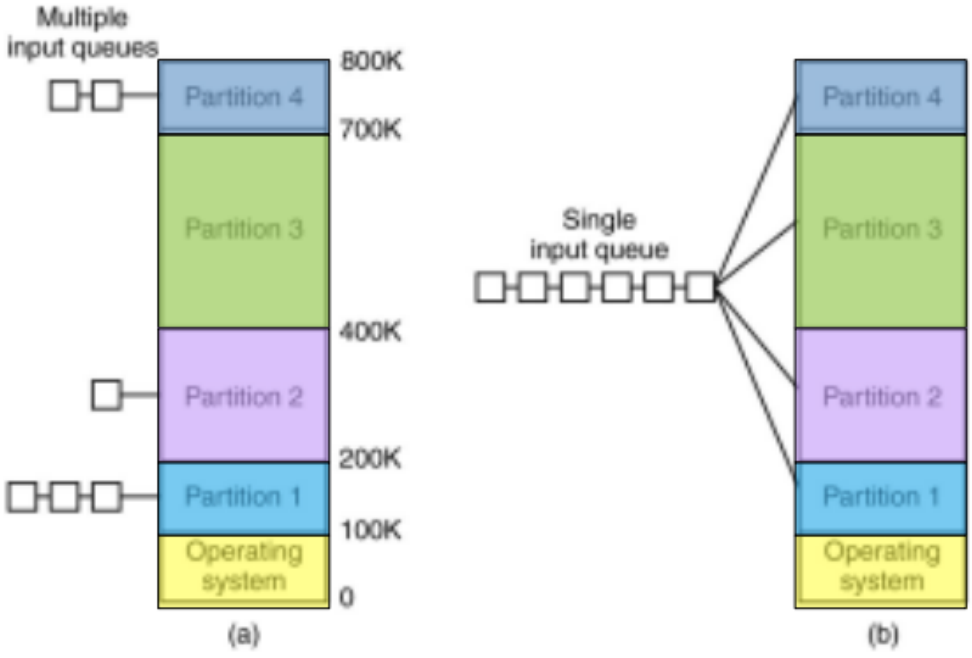
- **Memory Hierarchy**:
  - Registers
  - Cache
  - RAM
  - Hard disks

Mono-process

- Only memory sharing is between a user program and the OS
- Only one program can be loaded into memory
- Long term scheduler chooses job from pool of jobs to be loaded into memory

Multi-process

- **Fixed partition**: memory is divided into n partitions of possibly unequal size
  - **Separate input queues**: when job arrives, can be put into smallest queue large enough for it
    - Issue comes from large queues being empty while small jobs are waiting for smaller queues
  - **Single input queue**: closest queue that fits in it is chosen and loaded into the empty partition
    - Search whole queue and find out what job is a best fit (discriminates against small jobs) $\longrightarrow$ solution is to create a partition for small jobs and keep a counter to make sure that small jobs aren't skipped too often

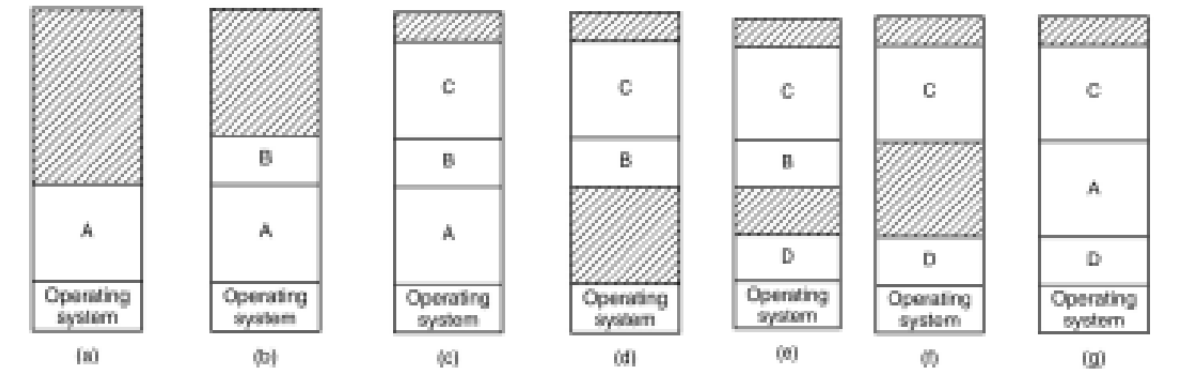


Modeling Multiprogramming

- **Probablistic model**: let p be the fraction of time a process is waiting for I/O. If there are n processes, the probability that all processes are waiting for I/O is p^n.
  - **CPU Utilization** = 1- p^n
- EXAMPLE:
  A computer has 32MB of memory, the OS takes up 16MB and each program takes up 4MB.
  So, with a p = 80%, you have about 60% = $1 - 0.8^4$ of CPU utitlization.

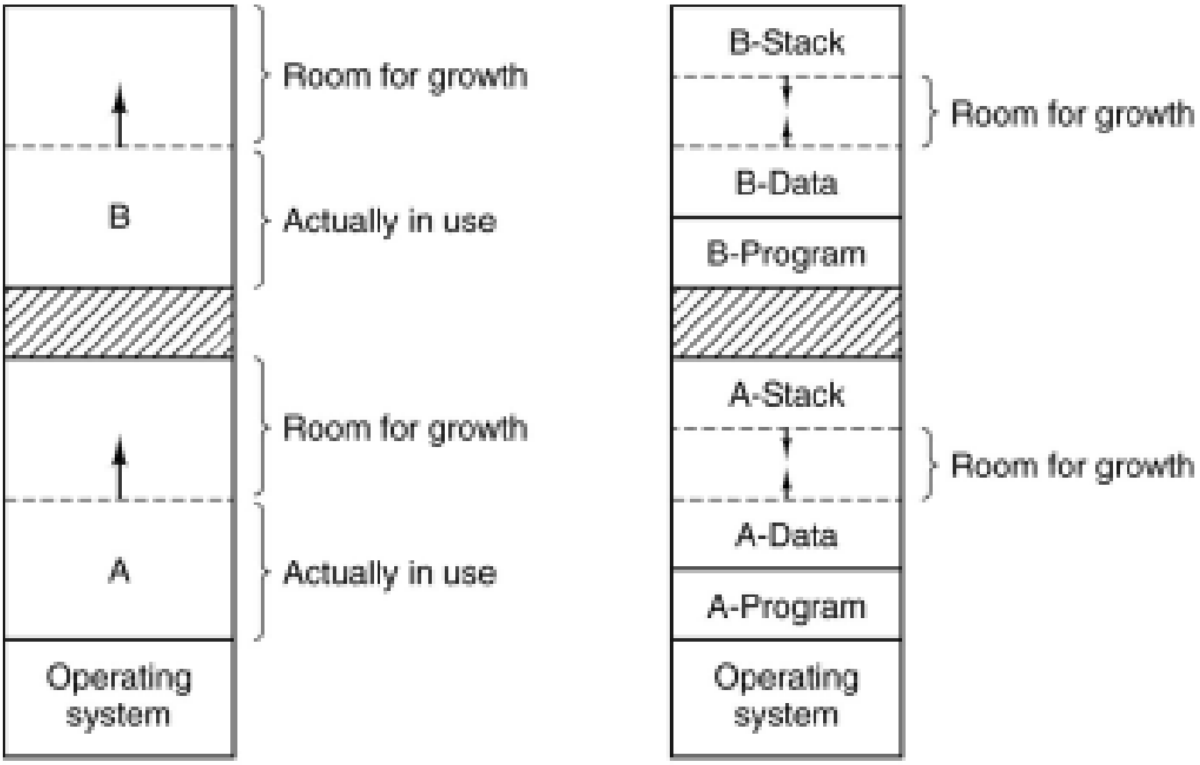
Ways of Memory Management

- **Swapping**: bring each process in its entirety, running it for a while, then putting it back on disk
  - **Fixed Partition**: the number, location, and size of partitions is fixed. OS knows the address of each process, simple to manage the memory, and the internal fragmentation wastes space.
  - **Variable Partition**: number, location, and size of partitions vary dynamically based on the size of the process; OS needs to keep track of partition information dynamically for allocation/deallocvation. Can create multiple holes ⟶ solution can be to combine them into one big hole.



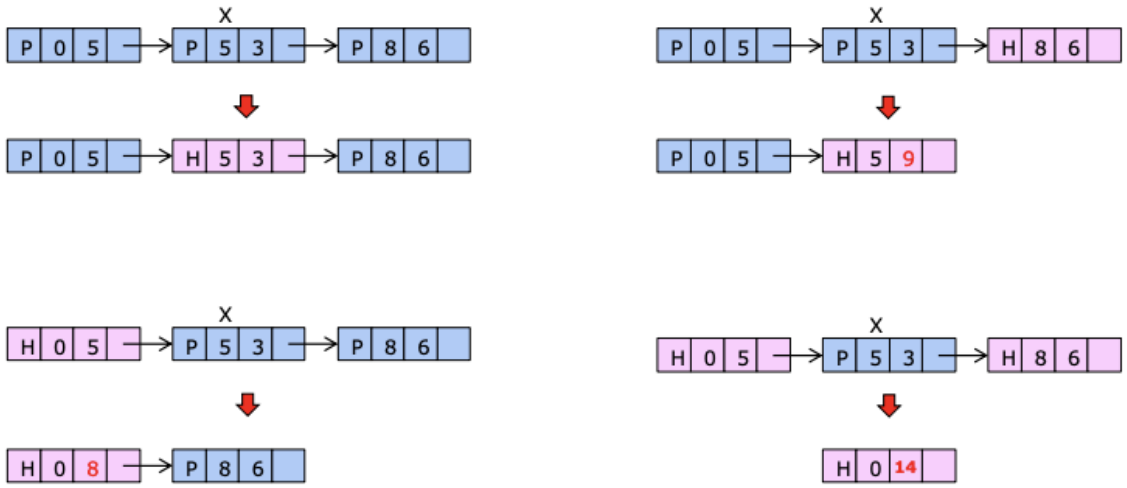
Variable partition example

-
  - How much memory should be allocated for a process when it is created or swapped?
    - Fixed partition: simple, base off that size
    - Dynamic allocation: using the heap/recursion; you have to allocate extra memory for each process, use adjacent hole to manage growing size. Processes may be swapped out or killed.



Free Memory Space Management

- **Bitmap**: memory is divided into allocation units (*words* that are k bytes).
  - Corresponding to each of the allocation units is a bit in the bitmap that show whether the unit is free (0) or occupied (1).
  - Size of allocation unit:
    - Smaller allocation = larger bitmap
    - Large allocation = smaller bitmap, memory may be wasted in the last unit of the process if the allocation units are not a multiple of the process size
  - **Advantages**: simple way to keep track of words in a fixed amount of memory since size of bitmap depends on the size of memory and the size of allocation units
  - **Disadvantages**: to allocate a process withh k unit size, the memory manager has to search for k consecutive 0 bits in the map
- **Free list**: OS maintains a linked list of allocated and free memory segments
  - Each entry has 4 bits: hole or process, starting address, length, and a pointer to the next entry
  - If the segments are sorted by address, then it is simple to update the list when a process terminates or is swapped out of memory



- EXAMPLE
  128MB of memory is allocated in units of 2KB. For the linked list, the memory consists of alternating sequences of segements of processes and holes, each 64KB. Each node in the linked list contains a 32 bit address, 16 bit length, and a 16 bit next node.

Bitmap

# of allocation units = $128MB/2KB = (128 * 2^{20})/(2 * 2^{10}) = 2^{27} * 2^{11} = 2^{16}$
size of bitmap = 2^{16} bits = 2^{13} bytes

Free list

 # number of nodes in linked list = $128MB/64KB$ = $2^{27}/2^{16}$ = 2^{11} nodes
size of each node = 32+16+16 = 64 bits = 8 bytes = 2^3 bytes
total size of linked list = **number of nodes * size of a node** = $2^{11} * 2^3$ = 2^{14} bytes

- Algorithms for memory space allocation with free-list:
  - **First fit**: scans list of segments from the beginning until it finds a whole that is big enough
  - **Next fit**: starts searching the list from where it left off last time
  - **Best fit**: searches entire list and takes the smallest hole
  - **Worst fit**: always takes largest free hole
- Can speed up the algorithms by maintaining two lists, one for the holes that is sorted by size and the other for processes
  - **Quick fit**: takes the separate list for holes which might have a table with n entries, where each entry is a pointer to a head of a certain size ($4KB$ holes, etc.) which means it is quick to find the hole of a required size but it is expensive.

Lecture Thirteen: Virtual Memory

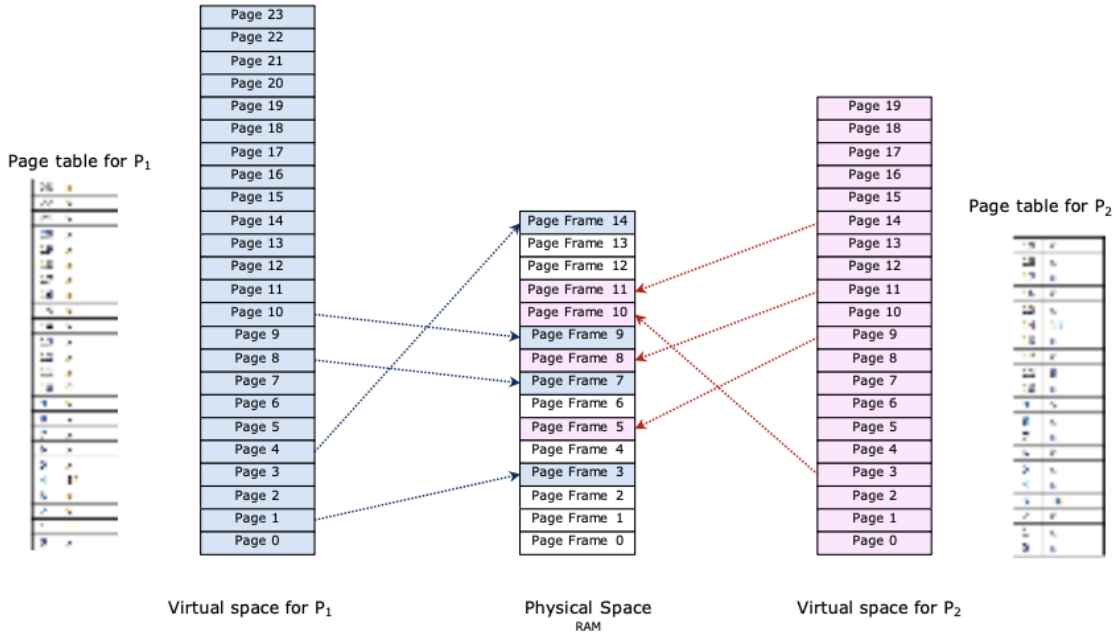
Virtual Memory

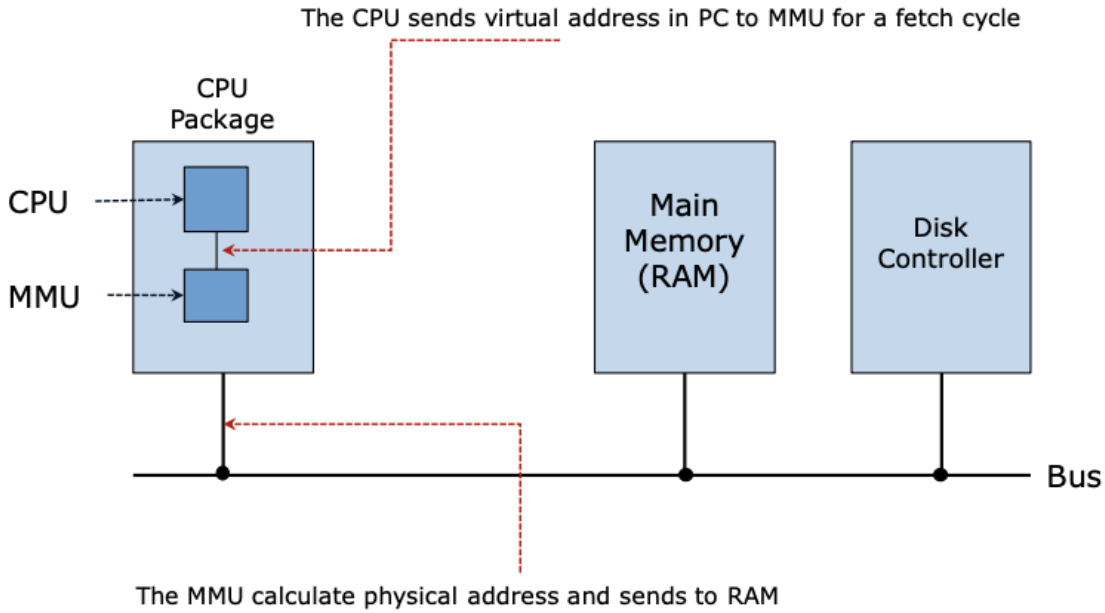
- **Memory sizes in bytes**
  - 1 byte = 8 bits
  - 1 KB = 2^{10} bytes
  - 1 MB = 2^{20} bytes
  - 1 GB = 2^{30} bytes
  - 1 TB = 2^{40} bytes
  - 1 PB = 2^{50} bytes
  - 1 EB = 2^{60} bytes
  - 1 ZB = 2^{70} bytes
  - 1 YB = 2^{80} bytes
- **Motivation**: the entire logical address space of a process must be in physical memory before the process can access, but since the size of a program grows and physical memory is limited, it is not always possible
- Virtual memory is a technique that allows the execution of processes that may not be entirely in memory
  - OS needs to keep track of the parts of the program currently located in main memory and the rest of the disk
  - Can be used in multiprogramming
- Approaches:
  - Paging
  - Segmentation
  - Segmentation with paging

Paging

- **Pages**: units of the virtual address space
  - Size is usually between 512 bytes and $64KB$
- **Page frames**: corresponding units in physical memory

- Page frame size and page size must be thhe same so you can calculate the physical address from the virtual address
- How to map virtual address to physical memory address:
  1. PC saves the address of the next instruction (virtual address in virtual space)
  2. Virtual address goes into the memory management unit (MMU) that maps the virtual address onto the physical memory addresses based on the memory map





- **Finding number of pages:** possible virtual space/page size
- **Finding number of page frames:** size of memory/page size
- EXAMPLE

The computer generates a 16-bit address, so the system supports a 64KB virtual space since: 0 ~ $2^{16} - 1$ = 0 ~ $64 * 2^{10}$. The page size is 4KB and the computer has 32KB

memory.

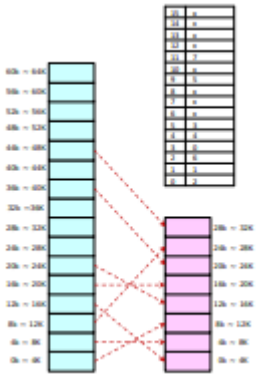
Number of pages: 16 = (64KB/4KB)
Number of page frames: 8 = (32KB/4KB)

- EXAMPLE

The computer generates a 32-bute address, so the system supports a 4GB virtual space since: 0 ~ $2^{32} - 1 = 4 * 2^{30}$. The page size is 4KB and the computer has 2GB of RAM.

Number of pages: 4GB/4KB = $(4 * 2^{30})/(4 * 2^{10})$ = 2^{20} pages
Number of page frames: 2GB/4KB = $(2 * 2^{30})/(4 * 2^{10})$ = 2^{19} page frames \

- EXAMPLE



MOV REG, 8900;;

The virtual address 8900 is sent to the MMU. The MMU then calculates the address 8900 belongs to virtual page 2 (8192 ~ 12288). The MMU then checks the memory map such that virtual page 2 maps to physical frame 6 (24576 ~ 28672).

The MMU then sends the address 24576 + (8900 - 8192) = 25287. This is the physical address.

Calculating physical address = beginning range of physical frame + (location provided - beginning range of virtual frame)

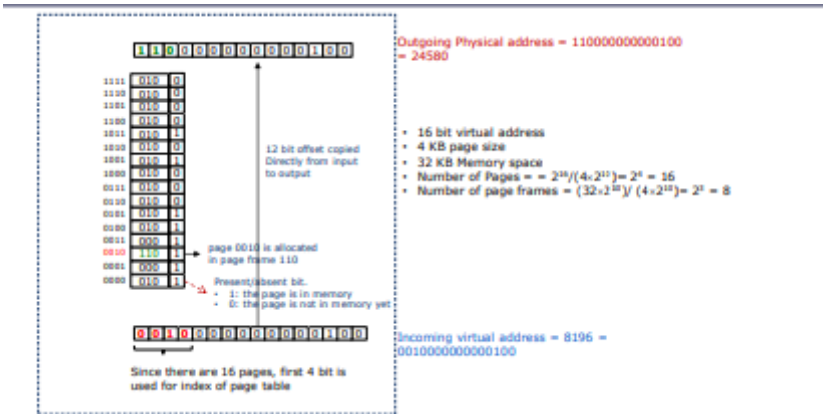
Physical Address Mapping

- How to map virtual address into physical address by the MMU
- EXAMPLE

The computer generates a 16-bit address, with a page size of 4KB. The computer has 32KB of RAM. There are 16 pages and 8 page frames.

The incoming 16 bit address is split into 4-bit page numbers, since there are 16 pages and a 12-bite offset, since the page size is 4KB.

The first 4-bits are used as an index into the page table (since 2^4 = 16). With the 12-bit offset, you can address 2^{12} = 4KB within a page.



The first 4 incoming bits are indexed into the table. In the table, there is a value that then becomes the first 4 bits of the outgoing (physical) address. The 12 bits of the original address are then appended to the new first 4 bits.

- EXAMPLE

System generates a 32-bit virtual address with a memory size of 1GB and a page size of 4KB.

Size of the virtual space = 2^{32}
Number of pages = $2^{32}/(4 * 2^{10})$ = 2^{20} so the page table has 2^{20} entries.
Number of page frames = $(2^{30})/(4 * 2^{10})$ = 2^{18}

For example, if the system generates a virtual address 10101111100010101110111111001111, the first 20 bits are used for the index of the page table, and the remaining 12 bits (page size = 4KB = 2^{12}) will be used for the offset. Because there are 2^{18} page frames, 18 bits must be reserved for saving page frame number in the page table entry.

> Pasted image 20230328224229.png

Lecture Fourteen: Page Tables

Page Table

- The purpose of the page table is to map virtual pages onto page frame
- Two major issues:
  - Page table can be extremely large

- The mapping must be fast
  - Building a single table consisting of an array of fast, hardware registers, with one entry for each virtual page, indexed by virtual page numbers can be very expensive.
  - The page table can be entirely in main memory, all of the hardware needs is a single register that points to the start of the page table, but you need one or more memory references to read page table entries.

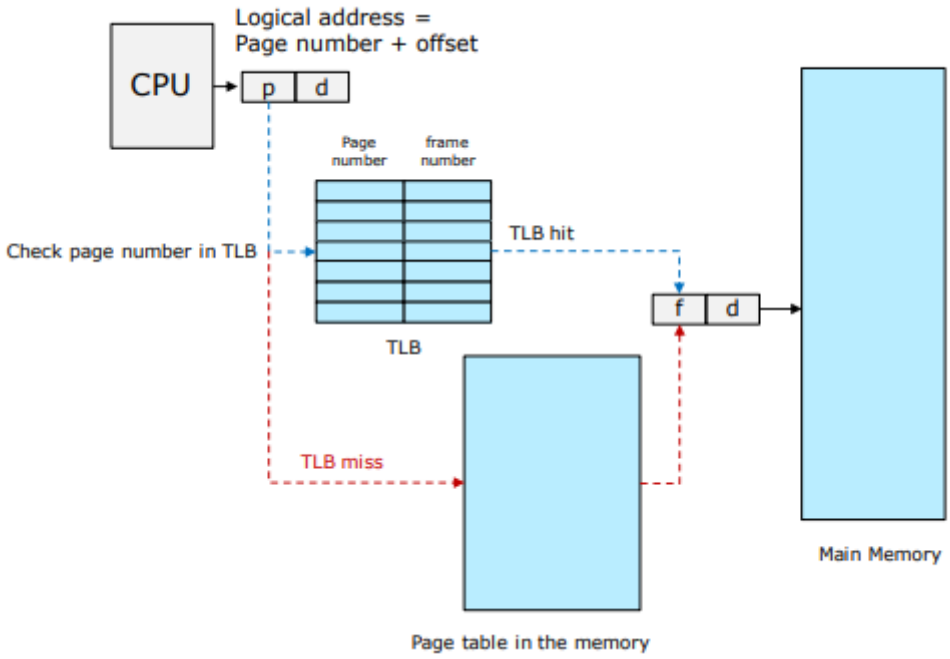
Page Table with Hardware Support

- OS maintains a page table per process; a pointer to the page table is stored in the process table.
- When the short term scheduler selects a process for execution, its page table must be loaded into memory.
- **Hardware implementation:**
  - Dedicated high-speed registers, which makes the page-address translation very efficient, but it increases context-switch time, since each register must be exchanged during a context-switch
  - Page-table based register, which is used to save a pointer to a page table that is kept in memory and changing page table requires only a chance in the content of the register

Translation Look-Aside Buffer

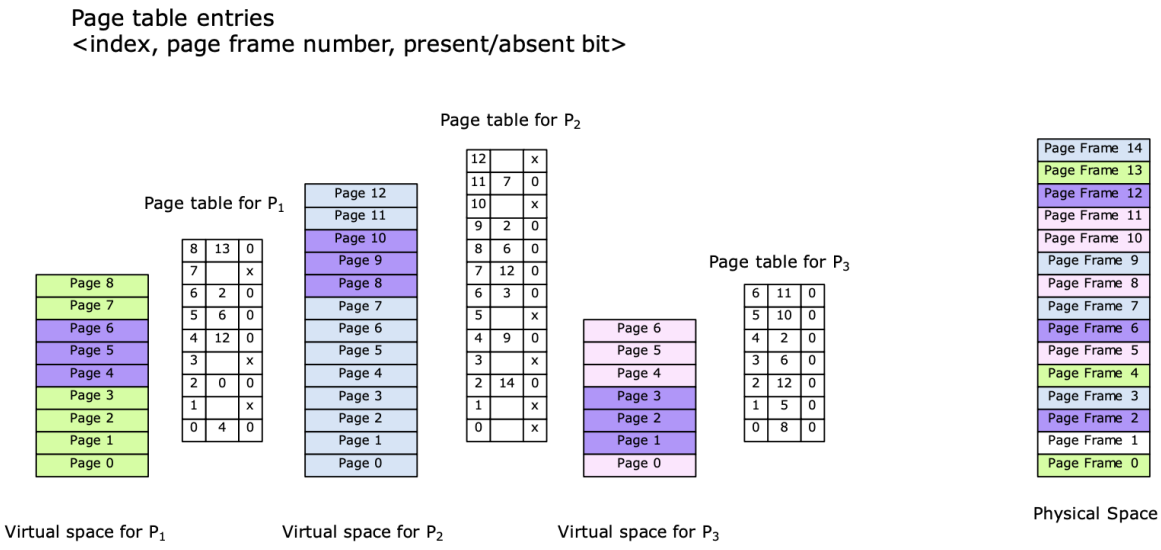
- Small, special, fast-lookup hardware cache with a key and a value.
- Used with a page table by saving a few of the page table entries
- TLB contains only a few of the page-table entries
  - When a logical address is generated by the CPU, the MMU first checks if its page number is present in the TLB; if it is found, then the frame number is immediately available (hit) and the MMU can map the physical address which is used to access memory
  - If a TLB miss happens, then address translation occurs:
  1. Search page table and get a page frame number to map to the physical address
  2. The search result will be added to the TLB, so that they will be used quickly on the next reference
  3. If the TLB is already full of entries, an existing entry must be selected for replacement based on replacement policies (LRU, round-robin, random)
  - **Effective access time:** (hit ratio)$*$ memory access time + (1-hit ratio)$*$ 2$*$ memory

access time



Page Table Continued

- Advantage of paging is sharing common code



One copy of frames in Physical space are shared by P_1, P_2 and P_3

Structure of a Page Table

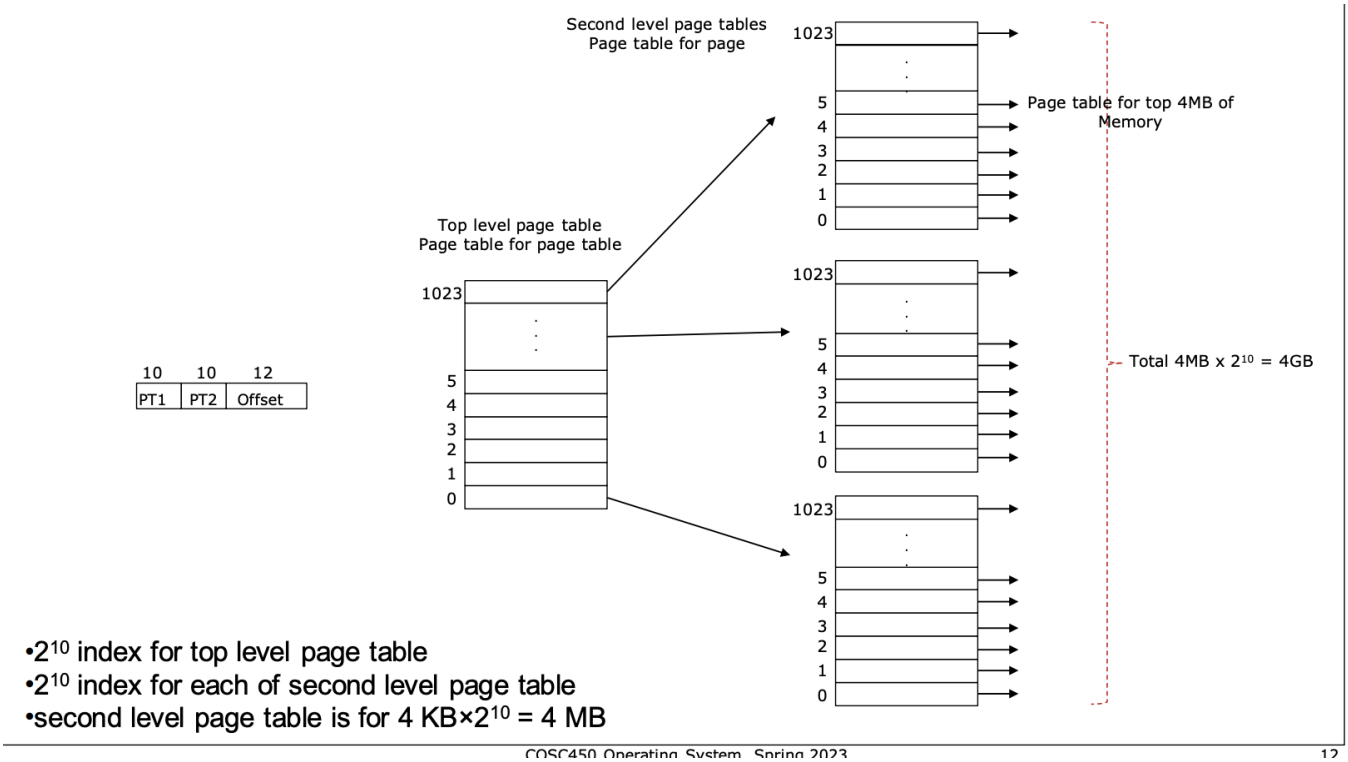
- **Multilevel page table:** avoiding keeping all the page tables in memory all of the time
- EXAMPLE: second level page tables

The system generates a 32-bit virtual address with a page size of 4KB. Since you have a 4KB page size, the offset is 12-bits, and the remaining 20 bits are partitioned into a 10-bit

PT1 field and a 10-bit PT2 field.

The **first level** is a page table for page tables. Since there are 2^{10} entries, there are 2^{10} page tables. The present/absent bit is used to save whether the table is in memory or not.

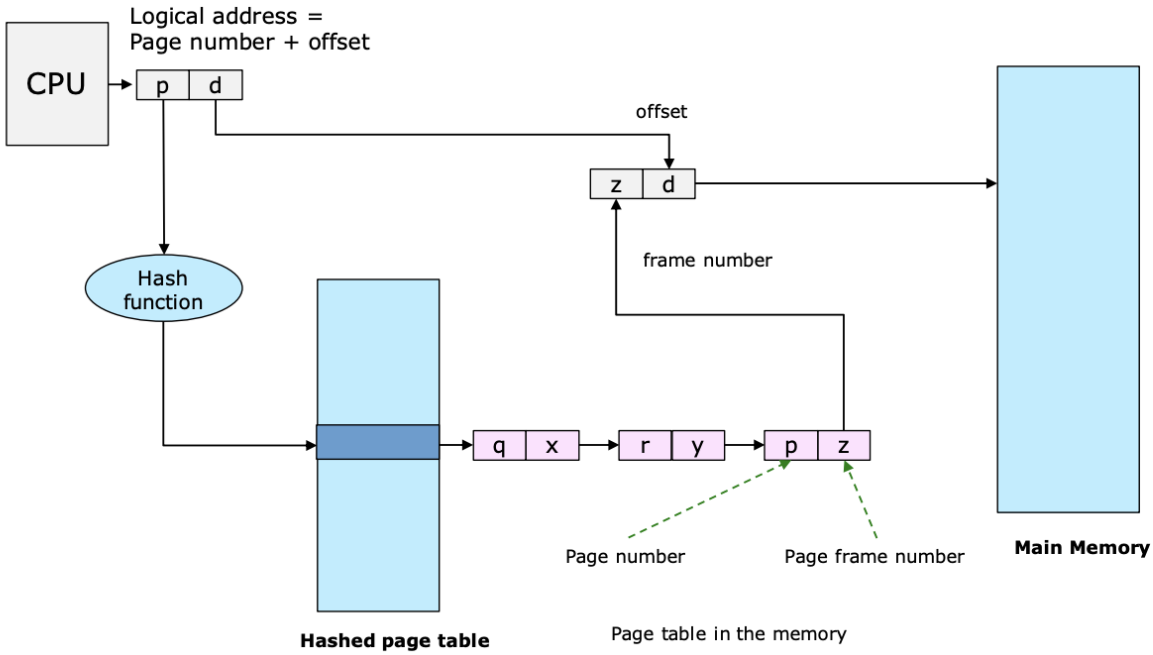
The **second level** is the page table for pages. Each second level page table has 2^{10} entries and since a page size is 4KB, each second level page table supports $2^{10} * 4KB = 4MB$ virtual space.



- **Hashed page table:** one approach for handling address spaces larger than 32 bits; the hash is the virtual page number.
  - Reduces the number of entries in the page table
  - Each entry in the hash table contains a linked list of elements that hash to the same location
  - Each element has three fields (page number, page frame number, pointer to the next element in the linked list)
  - **Algorithm**

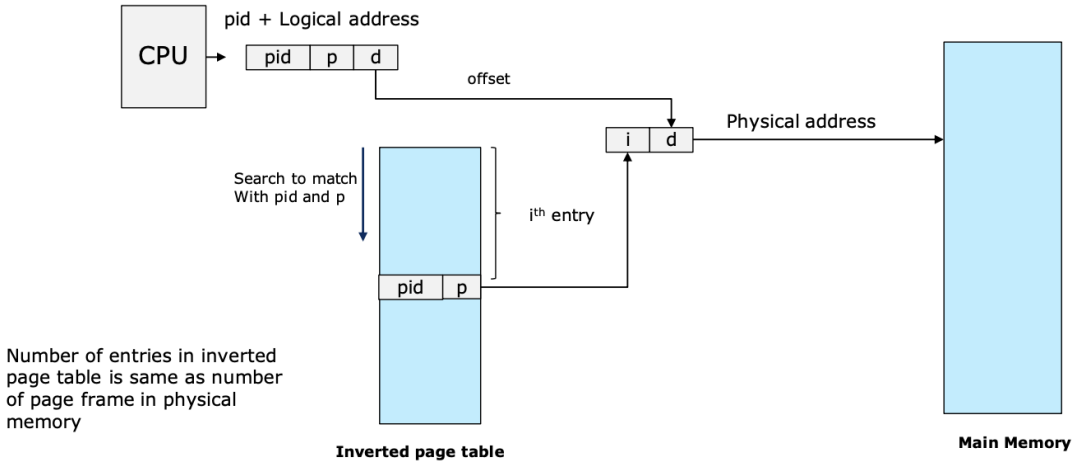
  1. Virtual page number in the virtual address is hashed into the hash table
  2. Virtual page number is compared with field 1 in the first element in the linked list
  3. If there is a match, the corresponding page frame is used to form the desired physical address
  4. If there is no match, subsequent entries in the linked list are searched for a

matching virtual page number



- **Inverted page table:** contains one page table entry for every frame of the main memory
  - Number of page table entries in the inverted page table is the number of page frames in physical memory
  - A single page table is used to represent the paging information of all of the processes
  - Overhead of storing an individual page table for every process gets eliminated and only a fixed portion of memory is required to store all of the paging information
  - Each entry has a virtual page number, process ID, and control bits
    - Given the process ID and virtual page number, you can search the inverted page table for a match
    - If the match is found at the i^{th} entry, then this page is located in the i^{th} frame and you can calculate the physical address to access memory.
  - Increases the amount of time needed to search since it's based on physical address but lookups occur based on virtual address.
    - Can alleviate this problem by using a hash table, though it adds a memory reference (one virtual memory reference requires at least two real memory reads, one for the hash table entry and one for the page table)
  - Can have an issue with shared memory, since there is only one virtual page for every physical page, one physical page cannot have two or more shared virtual pages.
    - A reference by another process sharing the memory will result in a page fault

and will replace the mapping with a different virtual address



Page table in the memory

Lecture Fifteen: Page Replacement Algorithms

Demand Paging and Page Fault

- **Demand paging:** type of swapping done in virtual memory; the data is not copied from the disk to main memory until they are needed or being demanded by some program.
  - While a process is executing, some pages will be in memory and some will be in secondary storage
- **Page fault:** a process tries to access a page that was not brought into memory
- **Algorithm for accessing memory:**
  1. Check the location of the referenced page in the page table
  2. If a page fault occurred, call on the OS to fix it
  3. Using the frame replacement algorithm, find the frame location
  4. Read the data from disk to memory
  5. Update the page map table for the process
  6. The instruction that caused the page fault is restarted when the process resumes execution
- **Pure demand paging:** after a page is brought into memory, the process continues to execute, faulting as necessary until every page that it needs is in memory
  - It can then execute withh no more faults (never bringing a page into memory until it is required)
  - **Algorithm:**
    1. Trap the OS (control change from process to kernel)
    2. Check page fault or not by checking the page table for the process
    3. Check that the page reference was legal and determine the location of the page on the disk

4. Issue a read from the disk to a free frame
5. Coorrect the page table
6. Restore process status
7. The process starts to run the instruction

- Three major task components of the page-fault service time
  1. Service the page-fault interrupt (save the process status, find the free page frame or find victim page frame)
  2. Read in page
  3. Restart process
- **Effective access time with page faults:**
  - p: page fault rate
  - m: memory access rate
  - s: page switch time
  - formula: $(1-p) * m + s * p$
  - Directly proportional to the page fault rate.
- Usage of the swap space: I/O to swap space tends to be faster than to the file system because it is allocated in much larger blocks, and file lookups and indirect allocation methods are not used.

Swapping with Paging

- Process instructions and data must be in memory to be executed, but a process or a portion of it may be swapped temporarily uot of a memory to a backing store and then brought back into memory for continued execution.
  - Swapping makes it possible for the total physical memory address space of all processes to exceed the real physical memory
  - Most systems use a version of swapping that takes out individual pages in a process rather than an entire process

Free-Frame List

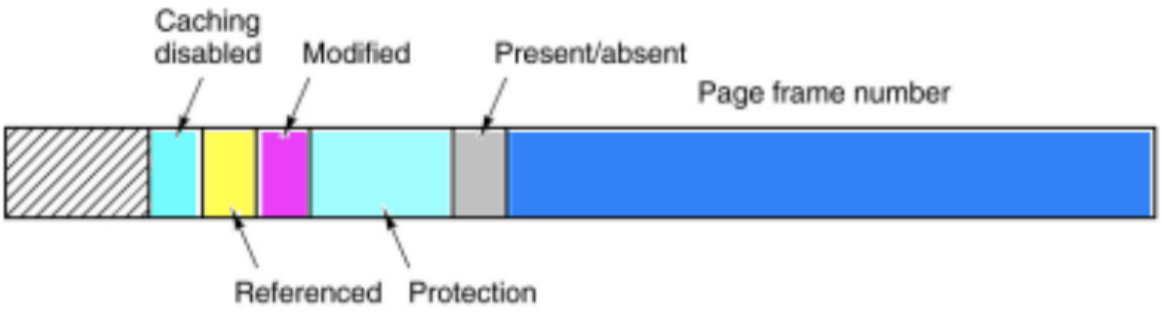
- When a page fault occurs, the OS must bring the desired page from secondary storage into a free page frame in memory
  - OS must create a free frame if there is non
  - Most OS maintains a free-frame list, which is a pool of free frames

Copy-on-Write between parent and child

- When using the fork() command, allows parent and child processes to initially share the same pages.

    - These pages are marked as copy-on-write pages, whichh means that if either process writes a shared page, a copy of the shahred page is created.
- When using vork(), it doesn't use copy-on-write, so if the child proocess modifies any pages of the parent's address space, the modified page will be visible to the parent once it resumes.

Structure of a Page Table Entry

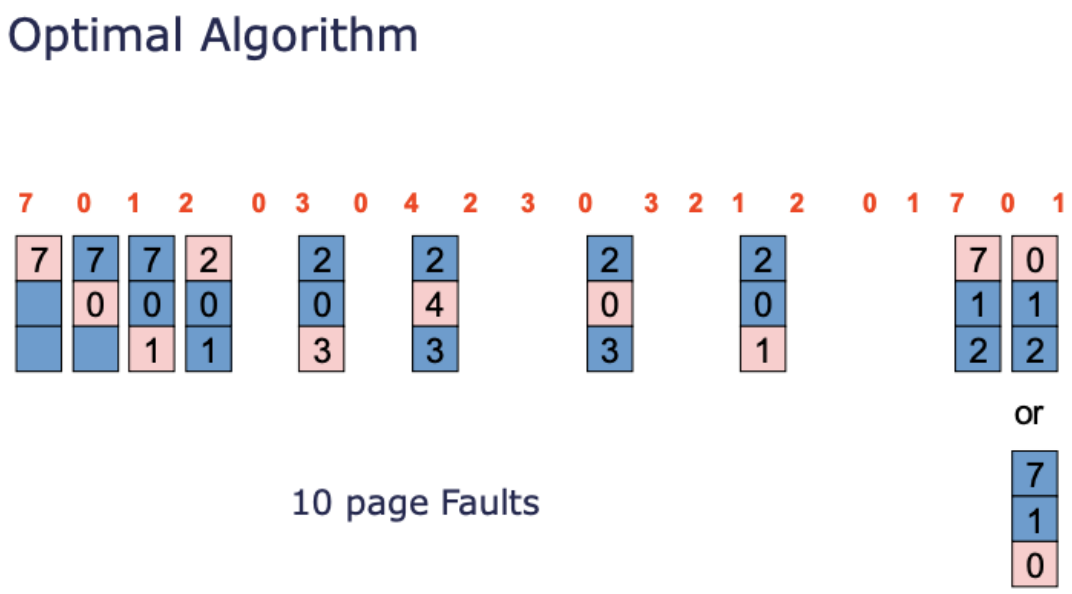


Page Replacement Algorithms

- When a page fault occurs and there is no free frame:
    1. Choose a victim page current allocated in a free frame (dependent on replacement algorithmh)
    2. If the page has been modified in memory, rewrite to the disk
    3. A page is allocated into the page frame which was used by the victim page
    4. Change page table
- If no frames are free, two page transfers (one for page out and one for page in) are required
- A process's memory access can be characterized by a list of page numbers, which is called the **reference string**
- Three items to characterize the paging system:
    1. The reference string of the executing process
    2. The page replacement algoritmh
    3. The number of page frames available in memory for a process

Optimal Algorithm

- Replace the page that will not be used for the longest time

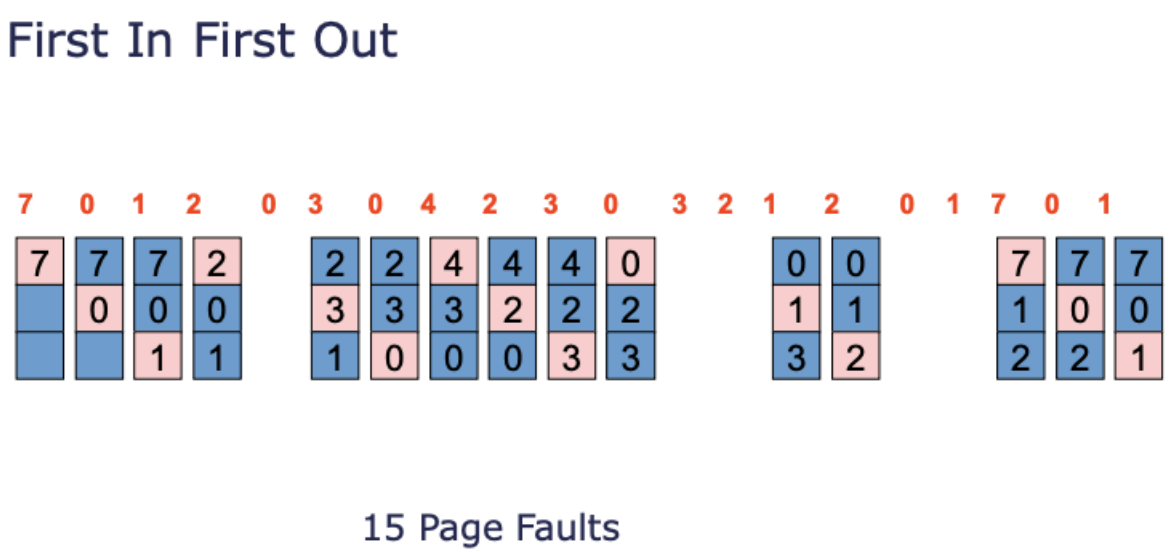


Not Recently Used

- When a page fault occurs, the OS inspects all pages and classifies them into four groups based on the page table information (modified, reference)
  1. Class 0: not referenced, not modified
  2. Class 1: not referenced, modified
  3. Class 2: referenced, not modified
  4. Class 3: referenced, modified
- Removes a page at randdom from the lowest numbered, non empty class

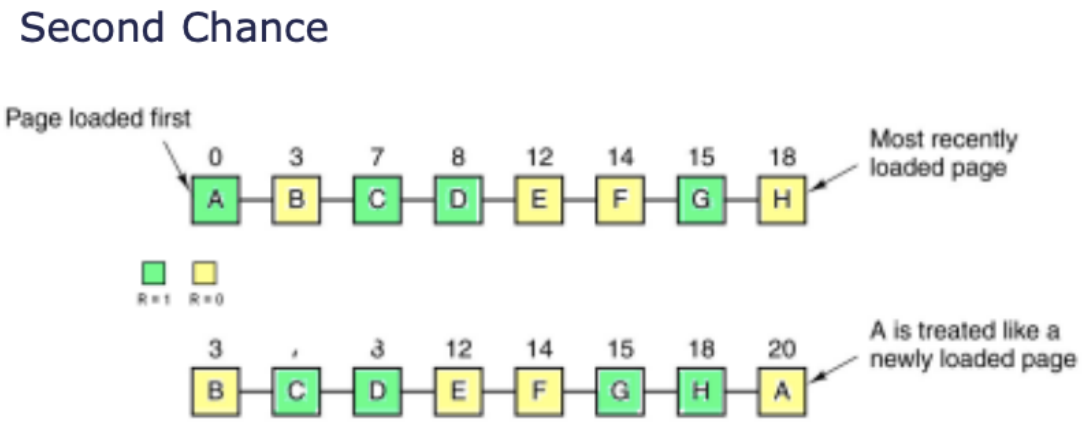
First In, First Out

- Replace the oldest page frame



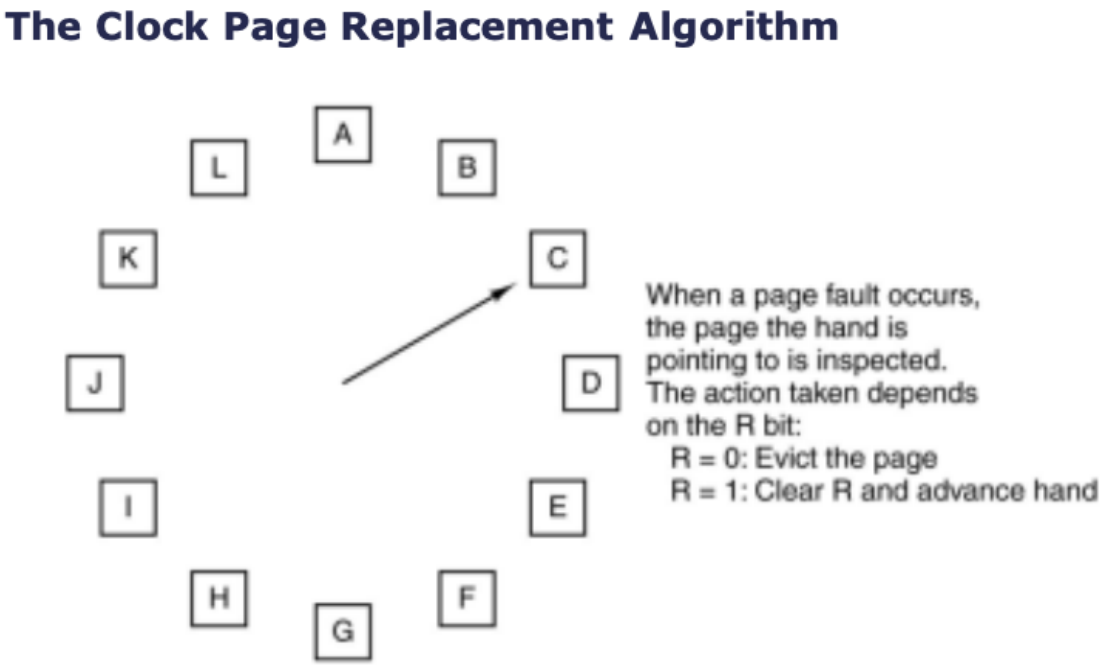
Second Chance

- Simple modification to FIFO that avoids the problem of throwing out a heavily used frame by inspecting the R bit of the oldest page.
  - If the oldest page's R = 0 and is not referenced, it is removed
  - If the oldest page's R = 1, it is old but referenced, so you set R = 0 and it moved from oldest to newest



Clock Replacement Algorithm

- Similar to second change, but you keep all of the page frames in a circular list
- When a page fault occurs, the page the hand is pointing to is inspected
  - If R = 0, evict the page
  - If R = 1, clear R and advance the hand



Least Recently Used

- Replace the page that has not bee used for the longest time
- Optimal page replacement looking backward in time
- Requires use of a counter (to save the reference time) and a stack (to keep a stack of page numbers)