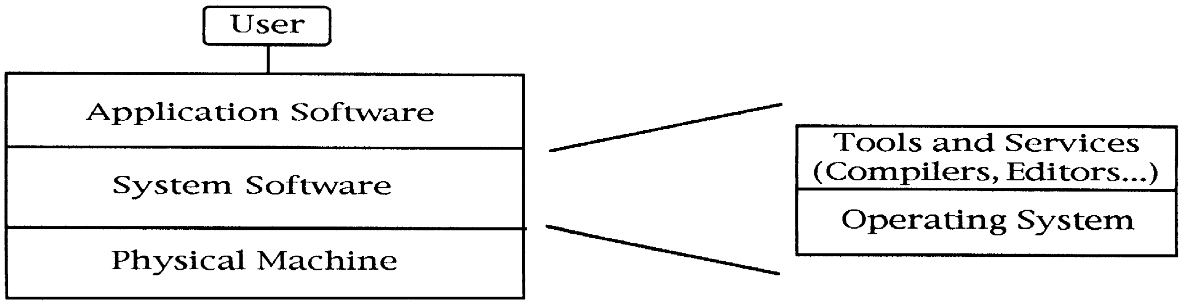
- A Computer is a Complex System
  Low-Level components require detailed, properly timed operating instructions.
  Without built-in complexity management:
    - Programmer productivity would be very low.
      Need to shield programmers from the complexity of the hardware.
    - End User knowledge would have to be very high.
      Need to present users with a simpler interface or simpler virtual machine.

- A Computer System is a synergistic combination of Hardware + Software.
  Without software, a computer is useless, uncoordinated pieces of electronic parts.

- There are two major types of computer software:
  1) Application Programs which solve problems for users in various domains.
  2) System Programs which manage the operation of the computer itself.

- A Computer System has 4 major components:
  1) Users
  2) Application Programs
  3) Systems Programs
  4) Hardware

- Most digital computers operate under the stored program concept.
  Program is stored in the same memory that contains the data.
    Generally attributed to von Neumann.
  An OS is Software but has a tighter integration with the hardware.
  Software can be run in two modes:
    User Mode: Hardware does not prevent user from making modifications.
      e.g.) User can write his own compiler, or use another commercial compiler.
    Kernel Mode or Supervisor Mode: S/W protected from user tampering by H/W.
      e.g.) User cannot modify or write his own disk interrupt handler.
  Therefore,
    Command Interpreters, Compilers, and Editors are *not* considered part of the OS.
    These are typically supplied by computer manufacturer, but run in user mode.



Structure of a Computer System and The Components of System Software

- An Operating System serves two roles:
  1) As a Resource Manager (Bottom-Up View).
     - Goal is to Make the computer system *efficient* .
       - This is especially important for large shared multi-user systems.
     - Provide orderly/controlled allocation of CPU, memory, and I/O to competing jobs.
       - e.g.) Manage 3 jobs attempting to write to the same printer at the same time.
     - Controls all of the computer's resources.
     - OS allocates resources to specific programs and users fairly and efficiently.
     - Gets hardware subsystems and application software to cooperate harmoniously.
       - OS Serves as the interface between application software and hardware.
       - Provides the base upon which application programs can be written.
     - Alternative view and name:
       - "OS" = "Control Program" which controls user programs, I/O Devices, etc.
         - e.g.) CP/M: Control Program for Microcomputers
  2) As a Virtual Machine (Top-Down View).
     - Goal is to Make the computer system *convenient* to use.
       - Especially important for small personal computers.
     - Hide the low-level machine complexity from the programmer and user.
     - Provide programmer with simple, high-level abstractions to deal with.
       - Hides the truth about the hardware from the programmer.
         - e.g.) Disk becomes a collection of named files; not records/blocks/sectors.
       - Provides a more convenient set of instructions to program in and to work with.
         - e.g.) "READ FILE" vs. "Move Disk Head to Track 129 of Drive 2, Wait 6ms"
     - Provide the user with a virtual machine that is easier to program than raw H/W.
       - OS acts as an interface between a user and the computer hardware.
       - OS provides reasonably high-level services with unreasonably low-level H/W.

- Technology has changed operating systems design.

Before:	Expensive, large, time-shared mainframes. Resource Optimization and Accounting were important.
Now:	Cheap, small, single-user personal computers and workstations. User-friendliness is most important. Hardly any accounting.

Note: Convenience and Efficiency are often contradictory goals.

- The most Important System Program is the Operating System.

Question:	"What is an Operating System ?"
Answer (1970's):	"All Programs Written and Supplied by IBM"
Answer (1990's):	"All Programs Written and Supplied by Microsoft"

- Operating Systems have continually evolved.

Question:	"What is a popular, state-of-the-art Operating System ?"	
Answer (1970's):	"IBM/OS360 JCL, DEC/VAX-VMS, DOS"	[Text Based]
Answer (1980's):	"Macintosh, Windows 3.1"	[GUI Based]
Answer (1990's):	"Mac/System 7"	[Networking, File Sharing]
Answer (8/24/95):	"Windows 95"	[Plug & Play, Internet WWW]
Answer (1996 ?)	"Mac/System 8"	[PowerPC Optimization]

- An Operating System has four major components:
  Viewed as concentric circles with hardware at the core, working away from center.
  This is also the order in which we will study material for our class.

  1) Memory Management.
     Memory is most expensive hardware in computer.
     Need Fast Speed *and* Large Size *and* Low Cost.
     The OS must utilize and manage the memory system effectively.
  2) Processor Management.
     Mainframe may cost $500 per hour.
     The OS must have efficient Scheduling Algorithms for the CPU.
  3) Resource and Process Management.
     Processes need to share resources, communicate, coordinate, and synchronize
  4) File System and Input / Output Management.
     The OS should organize information and protect it.
     The OS must schedule and coordinate Device Drivers for I/O.

  Note: In a strictly hierarchical sense, a given level can only call an inner level.
  But, This is not the case with the OS concentric circle view.
  Circle view helps identify which part of OS is closest to the low-level hardware.
  This is more of a layering approach to organization.
  We look at one layer at a time.

- Why Study Operating Systems ?
  - To create high-performance computer *systems* at a cost-effective price,
    Knowledge of the H/W + S/W design alternatives and tradeoffs is important.
  - Every computer specialist should know something about operating systems.
    Can help a software engineer write more efficient programs.
    Can help hardware engineer design better machines.

- Because operating systems work at the hardware / software boundary,
  Some details of the hardware need to be discussed to understand OS algorithms.

- Outline of Major topics:
  - Memory Management
    - Hierarchy
    - Locality of Reference
    - Virtual Memory
    - Paging, Segmentation
    - Replacement Algorithms
  - Processor Management
    - Scheduling
    - Properties of Distribution
    - Multi-Level Scheduling
  - Resource Management
    - Deadlock
    - Mutual Exclusion
    - Resource Graphs
    - Resource Trajectories
    - Avoidance vs. Prevention
    - Avoidance/Prevention Schemes
      - Hierarchy, Ordering of resources
      - Bankers Algorithm
  - Process Management
    - Concurrent Processes
    - Communication Primitives
      - Semaphores
      - Critical Regions
      - Test and Set
      - Producer/Consumer
      - Reader/Writer
  - File Management
    - Non-Hierarchical vs. Hierarchical
    - Protection
      - Authentication, Control Policies
      - Capabilities vs. Access Lists
  - Virtual Machines
    - Emulation
  - Performance
    - Reliability / Fault-Tolerance

- A computer system should satisfy several general design objectives:
  1) Provide an "easy to use" (user-friendly) appearance in S/W and H/W operation.
  2) Provide low-cost computing by sharing resources and information.
  3) Provide an efficient environment for program development, debugging, execution.

  - These functions must be provided by the system as integral characteristics.
  - The system software that provides these functions is called the Operating System.
  - Therefore, the basic functions performed by the OS must include:
    - Job Scheduling
    - Error, I/O, and Interrupt handling
    - Resource control
    - Protection
    - Good user interface
    - Accounting of computing resources
  - In general, an OS today assists in the construction and execution of user programs.
    Virtually all phases of program development and execution are supported by OS.
    - Example: PL/1 execution environment
      - A PL/1 compiler needs OS to read source input, and print its output listing.
      - Object code produced by PL/1 compiler needs run-time environment.
        - e.g.) ALLOCATE and FREE need the OS's Memory management.
        - e.g.) ON CONDITIONS need the OS's interrupt handlers.
  - How the need for these functions arose can be seen in the history of computers.

- The History and Evolution of Operating Systems
  Focus on evolution of: 1) Efficiency of Resource Allocation, and 2) Ease of Use.

- Before 1950's: The Age of Hands-on operation.
  Job-by-job processing.
  No operating system.
  Operation of loading and running a user program performed entirely by human.
  Each programmer operated the computer personally (loaded cards, pushed buttons).
  - Typical tedious sequence of tasks:
    - Place the source program cards in the card reader.
    - Initiate a program to read in the cards.
    - Initiate a compiler to compile the source program.
    - Place the data cards, if any, in the card reader.
    - Initiate execution of the compiled program.
    - Remove results from the line printer.

  Single User had one-at-a-time exclusive access to the entire machine's resources.
  Problem: Very uneconomical since expensive hardware was being underused.

- 1950's: The Age of Job Sequencing Monitors (Early Batch Processing).
  Computers getting faster and more expensive.
  Too costly to wait for a slow human to manually load his jobs in and push buttons.
  - First improvement:
    Let machine itself read-in, compile, load, and execute a program automatically.
    Requires that part of the machine be dedicated for this control (embryonic OS).
  - This allowed a series of jobs prepared in advance to be executed sequentially.
    Allowed automatic transition from one job to the next without long loading times.
  - Monitor had to provide inherent protection functions against errors such as:
    Limitation on processor times.
      Run time limits to prevent an infinite loop from blocking whole system.
    Supervision of input-output.
      Page limits to stop I/O loops.
    Protection of memory area reserved for the monitor.
      Avoids accidental/deliberate modification of kernel by a user program.
  - Problem: Processor was wholly occupied during execution of input-output.

- 1960's: The Age of Batch Processing Systems (Off-Lining).
  - Use two systems:
    1) Main computer for program execution.
      Executes jobs read off a magnetic tape; writes results back to magnetic tape.
    2) I/O computer(s) for preparing input onto or printing results off of magnetic tape.
      Took information from card reader and wrote it to magnetic tape.
      This I/O computer could be operated in parallel with the main computer.
  - Allows Main computer to process at full speed without having to wait for slow I/O.
    Requires an extension to the embryonic OS of 1950's.
      Need routines for the coding and packing of data on magnetic tape.
  - Off-Lining only reduced I/O dependency, but did not eliminate it.
    Main computer was no longer constrained by the speed of the card reader.
    But still constrained by speed of a tape reader (which is faster than card reader).
  - Example: IBM Fortran Monitor System (FMS).

- 1970's: The Age of Spooling, Multiprogramming, and Time-Sharing.
  Primarily Large (IBM) Mainframe technology.
  Computer technology "takes off" and many improvements in H/W and S/W occur.
  - One of the first improvements to early batch systems was spooling.
    Spool: *S*imultaneous *P*eripheral *O*peration *O*n *L*ine.
    Disk Drive Technology enables mass storage at fast access speeds.
    Spooling allows CPU to read in future job(s) while processing current one.
    Thus, computer can rapidly switch to a new job when current job completes.
    Execution order of spooled batches can even be different from the order read in.

- Multiprogramming gives each user the illusion of having whole system to himself.
  - Idea of concurrency (either real or imagined) enables interactive time-sharing.
  - Significantly increased operating system complexity.
    - In order for all jobs to be ready to run, they must all be kept in memory.
    - Rapid context switch mechanisms needed.
    - Also, need to provide for memory sharing and mutual protection of programs.

- Timesharing allows interactive, on-line communication between user and computer.
  - Main advantage: Enabled programmer to interactively debug program.
    - Much more productive than static snapshot dumps from a batch system.

- Also, to eliminate I/O dependency (not just reduce it), overlap I/O with processing.
  - Channels and interrupts were therefore developed (e.g. printer controllers)

- Main Characteristics of machines in the 1970's:
  - Specialized processors were used for information transfer.
    - Enabled system to perform I/O concurrently with CPU processing.
  - Memory could be shared between several jobs.
    - Enabled many jobs to be 'active' so CPU can choose from among several jobs.
  - Users were given an interactive mode of operation.
    - Each user had the "whole machine to himself" at a fraction of the cost.

- These characteristics required the operating system to be able to:
  - Buffer input/output from slower (factor of 10,000) devices as compared to the CPU.
    - OS had to store large amounts of temporary input and output.
  - Allow one job to use CPU while another performs I/O.
    - OS should keep the expensive CPU busy at all times to maximize throughput.
  - Reorder jobs to minimize overall system response times.
    - OS should not let one large job stop many small jobs from ever executing.
  - Keep accurate accounting of resources used by each process.
    - Total time in system is not equal to total CPU time and memory expended.

- Example: IBM OS/360.
    - Two to Three orders of magnitude larger than FMS.
    - Millions of lines of assembly language written by thousands of programmers.
    - Continuous stream of new releases were required to patch thousands of bugs.
    - Described as "a herd of prehistoric beasts stuck in a tar pit".

• 1980's: The Age of Workstations, Personal Computers.
  Microprocessors are introduced giving increased performance at lower costs.
  Prices for machines fall to within reach of the average household.
  - Emphasis shifts towards "personal", user-friendly computing.
    - Graphical User Interfaces for the masses replace cryptic command-line prompts.

Advances in both H/W and S/W technology were needed to "bring machines home".
Microprocessor Hardware brought down the cost making it affordable.
Operating System Software brought down the complexity for the 'average user'.
Desirable qualities in a personal computer include:
- Simplicity in use.
- Ease of extension by adding new utility programs (Setup program).
- Ease of extension by adding, adapting to new peripherals (e.g. Plug and Play).
- Efficiency (since performance of hardware is low, need to use it efficiently).
- Reliability.

• 1990's: The Age of Networks (especially Internet), Distributed systems, Laptops, PDAs
Portable devices require the need to "keep in touch" from anywhere in world.
Emphasis is on communications protocols (modems), file sharing/synchronization.
OS starts to handle some aspects of Networking (e.g., AppleTalk, Internet Access).
OS even responsible for handwriting recognition in PDAs.

• In summary, operating systems continue to grow in responsibility.
As computer power grew, resources required more sophisticated management.
Keeping track of resource usage and status became increasingly complex.
Therefore, the OS became, and will continue to become, more and more complex.

• We shall emphasize the study of *algorithms* for:
- Managing main and auxiliary memory devices.
  e.g.) Memory Manager.
- Managing, coordinating, and scheduling CPU and I/O processes.
  e.g.) Process Scheduler.
- Managing information flow among various devices/processes in a computer system.
  e.g.) Resource Manager, Interprocess Communication.

• We will study these algorithms from the perspective of both:
- Small systems (PC, workstations).
- Large systems (mainframes, distributed computer networks).

• Some algorithms meet acceptable standards, some are awkward, some are elegant.
Performance issues are also discussed, including:
Response Times (in multi-access systems).
Turn-around time for jobs (in batch systems).
Central Processor Unit Utilization time.
Resource Utilization.

=> A good OS increases efficiency and consequently decreases cost of using computers.

- **Memory**

  Digital computer is based on the stored program concept.

  Depends on memory to store both data and instructions needed for its operation.

  Even the Operating System resides in Memory.

  => Memory is a critical resource in the computer and it needs to be managed.

- Current technology trend and applications demand even faster and larger memories.

  Parameters that affect the memory design of a system includes:

  - Speed of the processor.
  - Speed of the I/O devices.
  - Expected application type and program size. (e.g., multimedia)
  - Expected degree of concurrency in a multiprogramming environment.
  - Expected number of instructions needed to process a transaction.
    (e.g., RISC architectures need more instructions per transaction.)
  - Size required for underlying systems software (e.g., Resident OS code).

  - The problem:

    There is a disparity between CPU logic speed and memory speed.

    Logic advances proceeded at a faster rate than advances in memory.

    The CPU can usually process instructions and data faster than they can be fetched from compatibly priced conventional main memory units.

  => Main Memory is the bottleneck in terms of system speed and cost.

    Speed of computer is limited by time needed to store and retrieve information.

    Cost of a system is determined by the data storage capacity of its memory.

- The importance of good memory design cannot be overstated.

  It can determine the useful lifetime of a machine.

    Bell and Strecker [1976]:

      "There is only one mistake that can be made in computer design that is difficult to recover from -- not having enough address bits for memory addressing and memory management".

  Address size is hard to change since it determines width of <u>all</u> H/W data items.

    e.g.) Registers, program counters, effective address arith., bus width.

  => If there are no plans made to expand the address size from the start,

    Then the end of the computer family will occur when the address size changes.

- Considerable effort is often devoted to the development and improvement of memory.

  The Memory Manager of the Operating System plays a critical role.

    Goal: Make up in S/W what cannot be attained in memory H/W technology.

  A tremendous amount of internal processing is used for memory management.

  Emphasis should be on the efficient and economical use of storage and the CPU.

• Memory Terminology Background

- Random Access Memory: All words are equally accessible in same time. So time required for addressing and locating a word is constant and independent of its address. RAM is generally electronic (no physically moving parts) and fast.

- Direct Access Memory (or Cyclic Access): Stores information in a continuously repetitive loop. Each word is accessible only as it passes the reading/writing station. Access time is dependent upon the address or position of the data. Access times for cyclic stores is inherently longer since, on the average, half a cycle must elapse before the desired word becomes available at the read/write station. Cycle time is usually due to some mechanical movement time (e.g. rotational speed of disk).

- Sequential Access Memory: Data are read or written in a longitudinal serial fashion along a finite length of media (e.g. tape). Can have very long latency times.

• Basic goals of memory design:

- Minimize the impact of the processor / memory speed imbalance.
  Generally, the CPU is much faster than (comparably priced) memory.
  Want the system to run at processor speeds rather than at memory speeds.
  Without instructions or data, CPU has no choice but to idle and wait.
  [Present-day solution: Memory hierarchy, Cache memory]

- Provide "enough" memory for the processor at a cost-effective price.
  Affects the extent to which the speed of the system is memory bound.
  General rule:
    A fast processor should have more memory than a slow processor.
    Fast CPU can read and write more memory per unit time.
    So, given same speed memory, faster CPU is more likely memory bound.
    Fast CPU needs constant input of instructions and data to keep busy.
  [Present-day solution: Virtual memory]

- Make memory optimization <u>automatic</u> without need for programmer intervention
  [Present-day solution: Memory management algorithms built into the OS]

- Note: All of above "solutions" must also inherently address cost, complexity, etc.]

• The Grand Design Challenge For Memory:
  Find a memory architecture that is fast and large and cheap.
  But no single memory architecture / technology can satisfy all three constraints.
  Simple axiom of hardware design: Smaller is Faster.
  Implication: It is impossible to build a memory that is large <u>and</u> fast <u>and</u> cheap.
  Fortunately, there is a saving grace: Principle of Locality.

- Principle of Locality of Reference

  The most important program property that enables designers to optimize memory.

  Fact: The data most recently used is likely to be accessed again in the near future.

  Programs tend to reuse data and instructions they have accessed recently.

  Rule of thumb:

  A program spends 90% of its execution time in only 10% of its code.

  Favoring accesses to such a small set of data will improve performance.

  Therefore, it is best to keep recently accessed items in the fastest memory.

  i.e.) Put the 10% of highly used code in faster memory; the rest in slower mem.

  This will give a speedup over the slower memory 90% of the time.

  Intuitive Justification of Locality of Reference for:

  Instructions: Structured and Modular programming (esp. Loops, Subroutines).

  Data: Matrices / Arrays and Record Data Structures.

  Two types of Locality:

  - Temporal: Locality in Time.

    If an item is referenced, it will tend to be referenced again soon.

    e.g.) Loop Counter Variable.

  - Spatial: Locality in Space.

    If an item is referenced, nearby addresses will tend to be referenced soon.

    e.g.) Element i + 1 of the same matrix.

  Note: An implication of locality is that, based on a program's recent past,
  one can predict what instructions and data it will need in the near future.
  A good memory manager will make use of this 'predictive' capability.

  Using locality of reference to our advantage, we can design:

  - A Memory Hierarchy.

    Which enables us to blend large, cheap, slow memories with
    small, expensive fast memories.

  - A Memory Management Algorithm.

    Which dynamically moves data to the proper "level" in the
    hierarchy as needed to optimize performance / cost.

- Memory Hierarchy

  A wide range of various memory technologies are available.

  Each has variations (advantages/disadvantages) in terms of cost and performance.

  It is not economically (or technically) feasible to use just one type of memory.

  Performance/Cost requirements vary tremendously within a computer system.

  e.g.) Always used (OS), Frequently used (apps), Seldom used (archived data).

  Therefore, memory systems consist of a mixture of several different memory types.

- **Central Idea of a Memory Hierarchy:**
  - Provide memories of various speed and size at different points in the system.
  - Organize the memory system into several levels by their speed, size, and cost.
  - Since smaller memories are faster, these should be located closer to the CPU.
    - These memories are the most costly on a per bit basis.
  - Memories become larger and slower as one moves away from the processor.
    - Lower cost per bit media can be used for levels furthest from the CPU.
  - Use a memory management scheme which will move data between levels.
    - Those items most often used should be stored in faster levels.
    - Those items seldom used should be stored in slower levels.
    - Ideally, this is done dynamically, automatically, and with little overhead.
      - >> This is the job of the OS memory manager (the focus of our study).
      - >> We will look at different memory management schemes.

- **A Typical Memory Hierarchy Consists of the following Levels:**
  - - Scratch Pad memory:
    - Consists of registers for control and arithmetic operations.
      - Accumulator, Program Counter (Generally on-chip).
    - Uses the fastest (and most expensive) semiconductor storage devices.
  - - Cache store:
    - Serves as a high speed buffer between slower main memory and CPU.
    - Technology and cost is between that of Main store and Scratch pad.
    - Note: Although cache is generally off-chip, some systems have on-chip cache.
  - - Main Memory:
    - Holds current data and program being executed.
    - Uses fast, high capacity (lower cost) random access storage (e.g. core or LSI).
  - - Permanent Store:
    - Holds program/data not currently in use; available to main store on demand
    - Requirement is for medium speed, high capacity, low cost storage.
  - - Mass Memory:
    - Used to store archival information.
    - High capacity, low speed, very low cost.

Category	Capacity (bits)	Access time	Access	Technology
Scratch pad	500-200K	5ns-10ns	Random	LSI
Cache store	50K-200K	10ns-50ns	Random	LSI
Main store	10K-10M	70 ns	Random	Core/LSI
Perm. store	100M-1G	10 ms	Direct	Disks
Mass store (old)	10**9-10**12	seconds	Sequential	Tape
Mass (new)	*17 G*	*200 ms*	*Direct*	*CD-DVD*

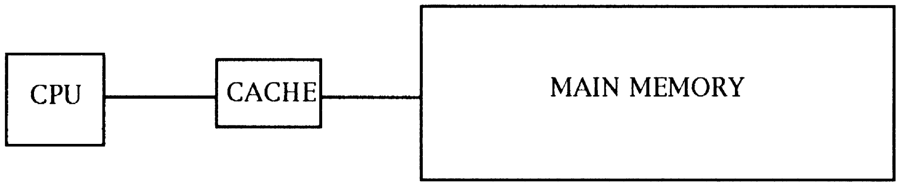
- Memory Management:
  - Using a pyramid structure, smaller memory is at top; larger memory is at bottom.
  - The levels of the hierarchy generally subset one another.
    - All data in one level is also found in the level below it.
      - e.g.) A subset of disk is in RAM; A subset of RAM is in cache.
  - Hierarchy normally consists of many levels.
    - Technology can introduce new levels (e.g., Level 2 cache, CD, DVD)
    - It is usually managed between two adjacent levels at a time.
  - Every pair of levels can be thought of as having an upper and lower level.
    - Upper level:
      - Smaller and faster and more expensive memory.
      - Closer to the processor.
    - Lower level:
      - Larger and slower and less expensive memory.
      - Further from the processor.
  - An effective MM scheme would arrange the data in the hierarchy such that
    whenever the CPU needs to access storage, it is able to find the data
    it needs in the upper levels of the hierarchy the majority of the time.
    - This would make the effective speed of the memory hierarchy nearly as fast as that of the fastest memory at the highest level of the hierarchy.
  - Hit ratio:
    - Percentage of memory accesses found in the upper level.
  - Miss rate:
    - (1.0 - Hit rate)
  - Since performance is main goal, speed of hits and misses is important.
    - Hit time: Time to access the upper level of the memory hierarchy.
      - Includes the time to determine whether the access is a hit or a miss.
    - Miss penalty: Time to move data in lower level up to a higher one.
      - If miss penalty is 10's of clock cycles, processor usually just waits.
      - If 1000's of clock cycles, CPU is task switched to another process.
    - A 1% decrease in hit ratio causes almost 10% drop in effective performance.

Hit Ratio (%)	Effective MIPS
100	5.0
99	4.4
98	3.9
97	3.6
96	3.3
95	3.1
90	2.2

- There are three issues when dealing with memory hierarchy management:
  1) Placement: <u>Where</u> to put lower level information in the higher level
  2) Replacement: <u>What</u> information in higher level to replace
  3) Fetch: <u>When</u> to perform update of upper level with lower level

- Again, Locality enables a memory hierarchy, including cache memory, to work
  Two empirical observations on program behavior are made:
  1) Programs tend to reuse instructions and data.
     => Once information is requested from lower levels, it should be moved to upper level and be kept there as long as possible to let subsequent accesses to it be performed at the faster, upper level speeds.
     Within loops, cache benefits are due to reusing previously accessed words.
  2) Programs tend to use instructions and data near recently used instructions/data.
     => If a block of memory larger than what is immediately needed is moved to the upper level, this additional information is likely to be needed soon, and its presence in the upper level will save one or more future references to the lower level, thereby speeding up future references.
     In straight line code, cache benefits are due to lookahead effect of locality.
     In data access, cache benefits are due to sequential storage of neighbors.

- Although memory concepts apply between any two levels, Cache/MM is usual focus.
  Cache Memory:
     A small, fast "buffer" that lies between the CPU and the Main Memory (MM).
     It holds the Most Recently Accessed Data (and instructions).



  Cache is generally only a small fraction of the size of main memory.
  Generally considered the first level in the hierarchy.
     Registers and accumulators are considered part of the internal CPU.
     [Note: Distinction is "blurred" since ICs today often include on-chip Cache]
  The term "cache" was coined by Conti.
     First realization appeared in IBM System 360 Model 85.
     Since then, caches have been present in most medium to large computers.
  Idea: Provide a copy of recently accessed MM words in a fast cache memory.
     Avoid accessing the slower main memory whenever possible.
  Goal: Make effective memory speed run at full speed of cache.
     Also make effective memory size as large as the entire main memory.
  High performance obtained by cache; Low cost obtained by lower levels (MM).

- Cache Operates as follows:
  When CPU needs to read/write to Main Memory, the cache is checked first.
    If data is found in cache a hit occurs and memory works at cache speed.
    If data not found, a miss occurs and memory works at the main memory speed.
      Also, the data from MM is written to the cache.
- Hit Ratio
  Hit ratio is the number of processor requests satisfied by the cache divided
    by the total number of processor requests.
  By definition, hit ratio is always less than or equal to 1.0 or 100%.
  Define:
    Hit Ratio = h = Probability that next memory access is found in the cache.
    T_a = Average effective memory access time as seen by CPU.
    T_c = Cache access time (access time for a hit).
    T_m = Main memory access time (access time for a miss).
  Then effective memory time is:

$$T_a = hT_c + (1 - h)\,T_m$$

  Speedup due to the cache is:

$$S_c = T_m / T_a$$

  - Example:
    Assume MM access time of 100ns and cache access time of 10ns
    Hit ratio of 0.9 would give an average access time of:
      $T_a = .9(10ns) + (1-.9)(100ns) = 19ns$
    Speedup is:
      $S_c = 100ns / 19ns = 5.26$

  - Example:
    Same as above only hit ratio is now .95 instead:
      $T_a = .95(10ns) + (1-.95)(100ns) = 14.5ns$
    Speedup is:
      $S_c = 100ns / 14.5ns = 6.9$

    => S_c, the speedup, is highly sensitive to small increases in cache hit ratio.

  If h is low, cache memory will not be effective.
    e.g.) If h = .5, we cannot achieve a speedup of more than 2, regardless of the speed of the cache.
  Goal of Cache Memory is to run the system at close to processor speeds
  The importance of getting high cache hit rates can be clearly seen
    Need to achieve high cache hit rates (at least 90%-95% of the time)
    Fortunately, typical hit ratios <u>do</u> range on the order of .9+

- Cache organizations differ primarily in the way blocks in MM are mapped to cache.

  We examine three mapping schemes: 1) Fully Associative
  2) Direct Mapping
  3) Set Associative

  For 1 and 3, Associative Mapping must be done fast since goal of cache is speed.
  Must be able to search and find words (if present) in cache quickly.
  Generally implemented in H/W using an associative (content-addressable) memory.

- Associative Memory or Content-Addressable Memory (CAM)

  Accesses words based on their contents rather than their location (address).
  Memory has extra hardware to perform fast search on its contents.
    Parallel AND gate logic is employed to find matches against a key register.
    e.g.) "Retrieve all memory cells with data equal to value x."
    e.g.) "Retrieve all memory cells with x < data < y ."
  Small associative memories for management of caches have been built.
    In cache, some address bits are stored along with data bits.
    Match is performed on the address bits when a search is done.
      "Does any cell in cache have this addresses' data? If so, what is the data?"
  Associative memory is extremely expensive because of parallel H/W match logic.
    Limited by cost to be very small in size.

- Fully Associative Mapping

  A main memory block can map into any block in cache.
  Theoretically optimum system: Fastest and most flexible.
  Conceptually simplest form of cache organization.
  Store both the entire MM address and its data in cache.
    Each processor request requires a complete search of the cache.
      When given a content value (address), CAM returns "associated" data.
      Labels of all cells of the CAM are tested simultaneously to find a match.
  - Advantages:
    No contention; a block of MM can be put in "best" cache cell possible.
  - Disadvantages:
    Very expensive because of the required large CAM to perform matching in H/W.
      Search would be too slow if done by software.
    Very wasteful of cache storage: Must store the <u>full</u> primary memory address.

Main Memory

Block 1	000	*Prog A*
Block 2	001	*Prog B*
Block 3	010	*Prog C*
Block 4	011	*Prog D*
Block 5	100	*Data A*

Cache Memory

Block 1	*100*	*Data A*
Block 2	*010*	*Prog C*

Italics: Stored in Memory

- **Direct Mapping**

  Opposite design extreme from fully associative; effectively a form of hashing.

  Cheaper to implement; no CAM memory required.

  A subfield of the MM address is used to index the cache in the regular manner.

  Least significant address bits are usually used for the index subfield.

  Store higher order "tag" bits along with data in cache.

  So, If MM Address is of form (t, i)

  then i bits designate the index to the cache block (the cache address)

  t bits are used for the tag

  The number of i bits needed depends on size of cache (e.g., i=3 if size = 8).

  Several MM words can have same i bits; Only one can be in cache at a time.

  Hit occurs only if tag at indexed cache location matches desired MM address tag.

  This allows consecutive blocks to reside simultaneously in cache but,

  All of the addresses in MM with same i subfield must share cache space.

  Assume Cache has K blocks:

  Main Memory blocks are effectively mapped into cache block via MOD function.

  Cache block = (Main memory block number) MOD (K)

  e.g.) Suppose Cache has K = 4 blocks and Main memory has 8 blocks.

  So Main Memory blocks 1 and 5 would map into cache block number 1.

  Only one MM address bit (MSB) needs to be stored in cache as tag.

Main Memory

Block 1	**000**	***Prog A***
Block 2	**001**	***Prog B***
Block 3	**010**	***Prog C***
Block 4	**011**	***Prog D***
Block 5	**100**	***Data A***
Block 6	**101**	***Data B***
Block 7	**110**	***Data C***
Block 8	**111**	***Data D***

Cache Memory

Block 1	**00**	***0***	***Prog A***
Block 2	**01**		
Block 3	**10**	***1***	***Data C***
Block 4	**11**	***0***	***Prog D***

Italics: Stored in Memory

- Potential Problem:

Suppose cache is 4 blocks and Main Memory holds 4 program + 4 data.

Excess swapping occurs if Prog A reads/writes Data A in a loop.

- Disadvantage:

Contention with MM data with same index bits.

- Advantage:

Low cost; doesn't require an associative memory in hardware.

Don't need to store low-order i bits of MM address label in cache memory.

Saves valuable cache space since MM i bits equal the cache's address.

- **Set Associative Mapping**

Intermediate, compromise solution between Fully Associative and Direct Mapping.

Maintains simplicity of the direct mapping approach.

Not as expensive and complex as a fully associative approach.

Allows for some flexibility in block placement.

Not as much contention as in a direct mapping approach.

Technique: Put a fully associative cache within a direct-mapped cache.

Organize the cache into K sets; Let each set contain P blocks.

MM address is directly mapped modulo K (via i bits) into a specific cache set.

But, it can be placed associatively in any of the P cache blocks within that set.

Typically, P (the number of "ways" or "degrees" in a set) is kept between 2 and 8.

Solves the contention problem in previous example with pure direct mapping:

Can put MM segments Prog A & Data A in different blocks within the same set.

Main Memory

Block 1	000	*Prog A*
Block 2	001	*Prog B*
Block 3	010	*Prog C*
Block 4	011	*Prog D*
Block 5	100	*Data A*
Block 6	101	*Data B*
Block 7	110	*Data C*
Block 8	111	*Data D*

Cache Memory

Set 1	0	*00*	*Prog A*	*10*	*Data A*
Set 2	1	*11*	*Data D*	*10*	*Data B*

- Advantages:

A generalized version of both fully associative and direct mapping.

Total number of cache blocks = N = KP.

If P = 1, then this is just direct mapping.

If P = N = entire cache size, then this is just fully associative mapping.

Main cost is in the complexity ("degrees") of the associative memory.

Statistical Hit/Miss rate as a function of P:

Degree Associativity	Miss Rate
1-way	6.6%
2-way	5.4%
4-way	4.9%
8-way	4.8%

=> Degree of associative search need not be over more than 2 to 8 tags.

Performs close to theoretical optimum of a fully associative approach.

Cost is only slightly more than a direct mapped approach.

Set-Associative cache offers best compromise between speed and performance.

The most favored design among manufacturers.

- **Replacement Algorithm:**
  Since the cache memory is very small, it is usually kept full.
  Eventually, a new block from MM will need to be brought into a full cache.
  Replacement algorithm determines which block in cache is removed to make room.
  - Direct mapping:
    Replacement algorithm is trivial: No choice.
  - Fully Associative:
    Many alternatives for replacement: Can choose over whole cache.
  - Set-Associative:
    Some freedom: No choice of set but can choose within a set.

  For fully and set-associative, a wide range of replacement algorithms is possible.
  However, complex replacement algorithms may be too difficult to implement.
  The decision of which block to replace has to made very quickly (nanoseconds).
  Maximize hit ratio by preserving in cache the elements that are in locality of ref.

- **Two main policies in use today:**
  - Least Recently Used (LRU) Algorithm.
    Attempts to reduce chance of throwing out block that will be needed soon.
    Accesses to blocks are recorded.
    The block replaced is the one unused for the longest time.
    e.g.) Attach an age counter (usually via hardware) to each block.
      When a block is loaded into cache, set its counter to 0.
      When a hit occurs for a block, also set its counter to 0.
      Bump counter for all other not-used blocks on each access by 1.
      If cache is full on a miss, remove the block with highest counter value.
    Becomes increasingly expensive to implement as number of blocks grows.
    Most popularly used (e.g. IBM 360 family).
  - Random
    Clark in 1983 showed only a small penalty for using Random instead of LRU.
    A counter-intuitive empirical result.
    Trivial to implement (e.g., random number generator).
    Used in DEC VAX 11 family.

Cache Size	Miss Rate: LRU	Miss Rate: Random
16KB	4.4%	5.0%
64KB	1.4%	1.5%
256KB	1.1%	1.1%

=>> The Replacement policy used becomes less important for larger caches.
If there are more blocks to choose from, the choice is less critical.
Probability of replacing the block that's needed next is relatively low.

- **Handling Memory Writes with Cache:**
  - Writes generally take longer than reads.
  - Fortunately, reads dominate cache accesses.
    - All instructions are reads, and most instructions don't write to memory.
    - Typically, read to write ratio is 10 to 1 (Writes occur only 10% of the time).
  - Where should the data written out by the CPU be stored ?
  - Three mutually exclusive ways the system can proceed:
    - 1) Write-Through or Store-Through
      - Writes updated data to both the cache and main memory simultaneously.
      - Makes action upon replacement simple:
        - When cache block is replaced, block is just flushed.
      - Ensures that the cache and MM entries are always the same.
      - Can result in unnecessary Write operations being performed to the MM.
        - Esp. if same cache word is updated many times during its residency.
      - Consecutive writes will result in T avg. approaching the slower MM rates.
    - 2) Write-Back, Write-out, Copy back, or Store-to-cache approach.
      - Update the cache location only and mark it as being "dirty" (modified).
        - Requires an extra bit to be associated with each cache block.
      - If the block is currently not in cache, then move it there before writing.
      - When a block needs to be removed from cache, it can be:
        - 1) Discarded if it is still "clean".
        - 2) Written back to MM if it is marked as being "dirty".
      - More universally popular than Write-Through Method.
    - 3) Store-Wherever approach
      - If copy resides in cache, then only that copy is updated.
        - Requires "dirty" bit to identify the block for write-back when replaced.
      - If no copy resides in cache, only the main memory is updated.
      - Element is not moved into the cache on a write.
        - Ignores locality with writes.

- **Other Parameter that affect Performance of Cache memory:**
  - - Size of the Block transferred from MM to cache:
    - Large blocks result in less overhead.
      - Easier to fetch one 8-word block than fetch eight 1-word blocks.
      - Also Maximizes benefits of spatial locality.
    - Small blocks only move in the data actually needed; results in less waste.
  - - Size of the Cache:
    - Larger cache will generally increase hit ratio asymptotically.
    - Larger cache will enable larger sized blocks to be used.

=> Using a cache is more economical than using fast memory devices for the entire MM.

- Primary goal of Cache Memory is to increase Speed
  Requires memory hierarchy management between cache and MM RAM.

- Primary goal of Virtual Memory is to increase Space
  Requires memory hierarchy management between MM RAM and disk.

- Virtual Memory:
  - Parallels many of the ideas of cache memory.
    e.g.) General problem of deciding which block is to be removed from a
    full MM is similar to deciding which block to remove from cache.
    Concepts like LRU replacement algorithm can be applied to page replacement.
    Cache management is usually handled by simple algorithms (must be very fast).
    Virtual memory management can be more sophisticated (lower in hierarchy).

  - Addresses two key issues:

    1) Computer users often want more MM than is physically present in machine.
    Second generation machines required manual memory overlays.
    Programmer had to identify mutually exclusive memory reference areas.
    Programmer could bring only one area into MM at any one time.
    Labor intensive: 50% of development time was spent implementing overlays.

    2) Time Sharing systems need automatic dynamic memory relocation capability.
    System serves multiple users "concurrently".
    Not every user's program or data need be in MM at the same time.
    Slow I/O response times (e.g. human) enabled processes to be swapped.
    A swapped out program is "suspended" in mid-execution and moved out of MM.
    On swap-in, it may be better to place program in a different memory location.
    Also, Parts of the same program can be in different memory locations.
    Block allocation can be done without regard to maintaining contiguity.
    Reduces need to rearrange memory when process population changes.
    Results in a noncontiguous and nonsequential dispersion of program(s) in MM.
    Portions of a program that appear to be contiguous may not be physically.
    Could be scattered throughout the memory in different block locations.
    Enables the OS to make maximum use of memory space.
    A small physical location can be used for a large virtual address space.
    Reduces occurrence of fragmentation.

  - Fragmentation:
    A phenomenon which can cause wasted memory at any level of hierarchy.
    When a large percentage of memory becomes small slots (fragments).
    The slots are not contiguous and therefore, are too small to be useful.
    A time-consuming "garbage collection" process can be used to
    amalgamate the many small slots into a larger, more useful area.

Fragmentation can result from:

- Roundoff in block allocation.
- Growing and contracting data areas.
- Programs entering and leaving the multiprogramming mix.
- Repeatedly deleting and adding various sized files (disk frag.).

• Underlying concept of Virtual Memory:

Program and data are assigned addresses independent of:

- The amount of physical MM storage actually available.
  Addresses Space issue (1) above.
- The location from which the program will actually be executed.
  Addresses Dynamic Relocation issue (2) above.

Program and data both reside in virtual memory space.

Memory Management unit moves data into and out of MM RAM from disk.

The units of data transfer and replacement (blocks) are called pages or segments.

• Dynamic Address Translation (DAT):

Mapping: The correspondence between virtual storage and physical storage.

A mapping table enables DAT to occur.

Virtual address is mapped to a physical address via the mapping table.

Relocates a program and its data segments in real time.

Translation is done transparently and "on the fly" (dynamically).

Does not require modifying the addresses within the program's instructions.

Requirements for the DAT mechanism:

- Fast Mapping
  Mapping function must be simple to compute in negligible time.
- Fast Update
  When block is relocated, tables must be updated quickly.
- Fast Context Switching
  When CPU switches processes, new mapping must engage quickly.
- Efficient Memory Hierarchy Management
  Hit rate should be maximized and time for miss penalty minimized.
- Efficient Usage of Physical Memory
  Achieve intensive use of memory by avoiding fragmentation.

Minimizing Overhead:

If mapping table itself were kept in RAM, then performing the DAT would cost an entire extra RAM memory cycle time. (Table + Element lookup).

Translation Lookaside Buffer (TLB) performs mapping.

Special hardware generally used to accelerate table lookup process.

Small associative memory and possibly fast adders.

- **Two Ramifications of Virtual Memory**

  1) Addressability

  Program can form addresses larger than the range of real physical memory.
  Frees programmer from concerns about fitting programs into available memory.
  Illusion is created by both the processor hardware and the virtual memory OS.
  Programmer has the conceptual ability to treat the memory as if it were:

  - Infinite in size
  - Always there

  2) Relocatability

  No direct relationship b/w addresses used by a prog and their physical locations.
  No linear ordering relation is implied by blocks in physical memory.
  Virtual address 5700 > 4500 sequentially; but not necessarily in physical mem.
  Disassociates an address from a memory location.
  System / maintenance programmers must be aware of relocatability.
  e.g.) A "different" memory location fails every time a program is run.
  Program can be scattered throughout memory.
  Core dump must be interpreted.
  Contents of memory at any one time is a collection of disjoint blocks.

- **Three block-oriented mapping schemes for virtual memory:**
  - Paging

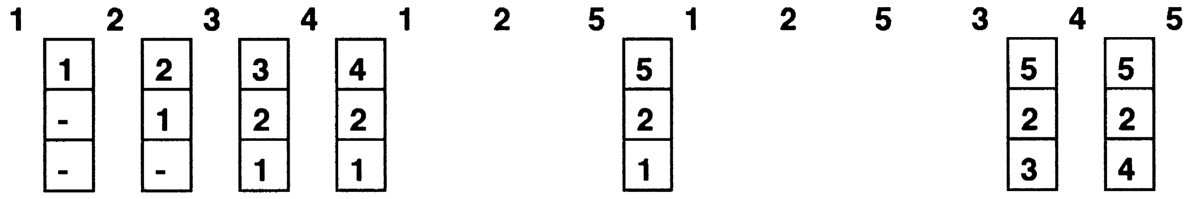
    Simplest method of virtual to physical translation.
    Page: A fixed size block of virtual address space.
    All pages are therefore the same size.
    Typical page sizes range from 512 bytes to 4K bytes.
    Page Frame (or Slot): Block of physical memory space.
    Any page can be stored in any page frame.
    An address x generated by CPU is partitioned into high and low order bits.
    High order bits are treated as a page number.
    Low order bits are treated as an offset within the page.
    Address translation is just substitution of page frame number for page number.
    - External fragmentation problem is solved.
    Each page will exactly fit in a page frame.
    - However, fixed-size pages can lead to internal fragmentation.
    e.g.) Assume page sizes of 4K bytes and a program size of 13K bytes
    Results in 4 assigned pages (16K bytes) wasting 3K in the last page.
    If too few slots are allocated to a program, thrashing occurs.
    Thrashing: Constant movement of the same blocks between two levels.
    If severe, can lead to deadlock where no process can proceed.

- **Segmentation**
  - Paging is concerned with the partitioning of <u>physical</u> storage space.
    - Pages assume a predetermined relationship to blocks of physical storage.
  - Segmentation is concerned with the allocation of <u>functional</u> space.
    - Segments are viewed as logical subdivisions of virtual storage.
    - Example segments: procedures, subroutines, matrices, data structures.
  - A segment is:
    - 1) An independent linear address space.
    - 2) <u>Variable</u> in size as opposed to fixed in size as pages.
  - Size of each block must be explicitly recorded in the mapping table.
  - Segments enable:
    - -Data structures to expand and contract.
    - -Protection.
      - Programs (read only) can be in separate segments from data (R/W).
    - -Sharing in multiprogrammed systems.
      - Multiple processes executing same program can use same segments.
      - Several users may be executing same program with different data.
        - Their data is loaded into different areas of physical storage.
        - But they can share a single copy of the program in physical mem.
          - Pointers set to same location via the mapping mechanism.
  - No internal fragmentation occurs using variable sized Segments.
  - External fragmentation can occur unless contiguous blocks are relocated.
    - Segments normally require contiguous storage locations in MM.
    - Since segments are variable size, range in sizes can be rather large.
    - Therefore, even if some empty regions exist, they may not be usable.

- **Segmentation with Paging**
  - Let each segment define a linear memory space which itself is paged.
    - Segmentation is applied to the user's virtual space.
    - Paging is applied to the real physical memory space.
    - Each segment can contain a variable number of fixed size pages.
  - Accrues the advantages of both segmentation and paging.
  - A compromise on the size of Allocated Memory:
    - Large blocks are good because it keeps the translation table small.
      - However, only a small number of blocks can be allocated.
      - Limits the number of users or applications that can be serviced.
    - Small blocks can service many users or applications concurrently.
      - But requires a large translation table.
    - => Use Large blocks (segments) partitioned into smaller blocks (pages).
  - Requires a double look-up on two tables (a segment table and a page table).

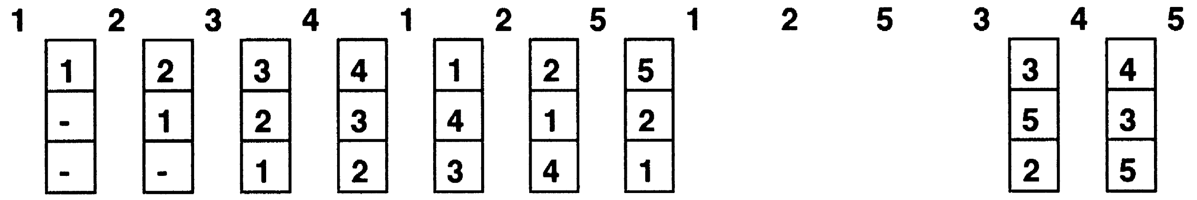
- **Replacement Algorithms for Virtual Memory:**
  Similar to Cache replacement algorithms, but can be more complex in S/W.

- **Optimal Replacement**
  Replace the page which will not be used for the longest (future) period of time.
  A theoretically 'best' page replacement algorithm for a given fixed size of VM.
  Produces the lowest possible page fault rate.
  Example: Assume the following page reference string w/ 3 frames of allocated VM.
    Faults are shown in boxes; hits are not shown.

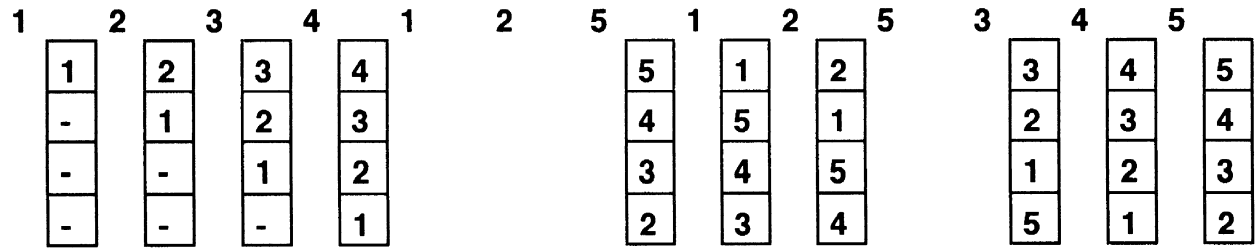


  Seven page faults occur.
  Generally impossible to implement, since it requires future knowledge of ref. string.
  Merely used to gauge the performance of real algorithms against best theoretical.

- **FIFO**
  Simplest page replacement algorithm.
  Can be implemented with a simple shift mechanism (or a head/tail queue).
    Needs no knowledge of reference string (past or future).
  When a page fault occurs, replace the one that was brought in first.
  Example: Assume the following page reference string with 3 frames of allocated VM.



  Nine page faults occur.
  Problem: Performance may not always be good.
    Can exhibit inconsistent behavior known as Belady's anomaly.
    Number of faults can increase if job is given more virtual memory.
  Example: Same reference string as above only with 4 frames instead of 3



  Ten page faults occur.

- **Least Recently Used**

  Replace the page which has not been used for the longest period of time.

  Can use either a counter or a stack mechanism.

  Example: Using a stack with 3 frames of allocated VM (keeping most recent on top). Faults shown in boxes; hits only rearrange stack.

1	2	3	4	1	2	5	1	2	5	3	4	5
$\boxed{1}$	$\boxed{2}$	$\boxed{3}$	$\boxed{4}$	$\boxed{1}$	$\boxed{2}$	$\boxed{5}$	1	2	5	$\boxed{3}$	$\boxed{4}$	
$\boxed{-}$	$\boxed{1}$	$\boxed{2}$	$\boxed{3}$	$\boxed{4}$	$\boxed{1}$	$\boxed{2}$	5	1	2	$\boxed{5}$	$\boxed{3}$	
$\boxed{-}$	$\boxed{-}$	$\boxed{1}$	$\boxed{2}$	$\boxed{3}$	$\boxed{4}$	$\boxed{1}$	2	5	1	$\boxed{2}$	$\boxed{5}$	

  Nine page faults occur.

  LRU is more expensive to implement than FIFO, but is more consistent.

  Does not exhibit Belady's anomaly, since by induction:

  For n pages, LRU keeps the n most recently referenced pages in memory.

  For n+1 pages, LRU still keeps the subset of n most recent pages in mem.

  More overhead needed since stack must be updated on each access.

  LRU generally gives better performance since temporal locality is preserved.

  Example: Same reference string as above only with 4 frames instead of 3.

1	2	3	4	1	2	5	1	2	5	3	4	5
$\boxed{1}$	$\boxed{2}$	$\boxed{3}$	$\boxed{4}$	1	2	$\boxed{5}$	1	2	5	$\boxed{3}$	$\boxed{4}$	
$\boxed{-}$	$\boxed{1}$	$\boxed{2}$	$\boxed{3}$	4	1	$\boxed{2}$	5	1	2	$\boxed{5}$	$\boxed{3}$	
$\boxed{-}$	$\boxed{-}$	$\boxed{1}$	$\boxed{2}$	3	4	$\boxed{1}$	2	5	1	$\boxed{2}$	$\boxed{5}$	
$\boxed{-}$	$\boxed{-}$	$\boxed{-}$	$\boxed{1}$	2	3	$\boxed{4}$	4	4	4	$\boxed{1}$	$\boxed{2}$	

  Seven page faults occur.

- **Fixed Partitioning vs. Dynamic Partitioning**

  Want to maximize number of jobs in a multiprogrammed, virtual memory.

  Increases probability that job scheduler can find a job not in I/O or wait state.

  More alternatives for process switching.

  Maximizes throughput and overall system performance.

  Increases total number of jobs processed per unit time.

  Minimizes total space-time product.

  Decreases amount of memory a job needs and the total time it needs it for.

  - Fixed size virtual memory allocation.

  LRU, Random, FIFO are all fixed size allocation methods.

  A job is given a fixed allocation of virtual memory while running.

  Needs to be "burned in" at design time (either as a H/W or S/W parameter).

Fixed size allocation schemes are also extremely hard to optimize.
 If space allocated to each job is too small, thrashing occurs.
 If too much memory is allocated, space is wasted.
 A wide mix of programs with wide range of memory requirements exist.
 Memory requirements of a job can change over its run time.

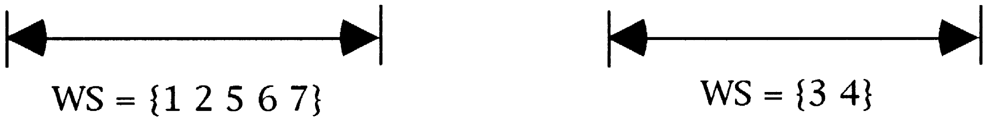
Want self-adaptive virtual memory size allocation schemes.
 More sophisticated replacement policies dynamically adjust allocated VM RAM.
 The number of page frames per process varies during execution.
 This will minimize the memory space-time product and maximize throughput.
 Two popular algorithms:
 1) Working Set (WS)
 2) Page Fault Frequency (PFF)
 If respective parameters are properly adjusted, WS and PFF are similar in perf.
 Both are generally better than fixed size LRU.

- Working Set

Denning 1968.
Working set is the minimum amount of space required for a program to perform well.
A working set is the set of distinct pages most recently referenced by a program.
Uses a working set window defined by T.
 Requires looking back at the last T references in the page reference string.
Remove page frame whenever it has not been referenced during last T references.
As a program executes, it moves from locality to locality.
 A locality is a set of pages which are actively used together.
 e.g.) Entering a subroutine brings in its locality of instructions, variables, etc.
 Upon exit, the pages associated with this locality are no longer needed.
Try to detect and preserve in RAM the dynamic locality set of a program in execution.
 If enough pages are allocated to a job's newly entered locality,
 it will fault until all of the pages of this locality are in memory,
 then it will not fault again until it changes localities again.
Performance of the working set algorithm depends upon the selection of T.
 If T is too small, it will not encompass the entire working set (thrashing).
 If T is too large, it will overlap several localities (wasting space).
Essentially a generalization of LRU.
 Any page not referenced in window size T can be replaced; not just the LRU.
Overhead is associated with keeping track of the working set.
 Working set is essentially a moving window.
 A page is in the working set if it is referenced anytime during last T references.
 Interesting note: A page is not necessarily replaced at page fault time.
 It may just move out of the 'window' upon a hit of another page.

- **Example: Using a window of T = 10.**
  Number of pages allocated would be five at one point, then two at another.

...2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3 2 3 4 4 4...



- **Example: Using a window T = 6**
  Note: Not really a stack algorithm (order on stack not significant).

1	2	3	4	1	2	5	1	2	5	3	4	5
$\boxed{1}$	$\boxed{2}$	$\boxed{3}$	$\boxed{4}$	4	4	$\boxed{5}$	5	5	5	$\boxed{3}$	$\boxed{4}$	4
	$\boxed{1}$	$\boxed{2}$	$\boxed{3}$	3	3	$\boxed{4}$	4	4	2	$\boxed{5}$	$\boxed{3}$	3
		$\boxed{1}$	$\boxed{2}$	2	2	$\boxed{3}$	3	2	1	$\boxed{2}$	$\boxed{5}$	5
			$\boxed{1}$	1	1	$\boxed{2}$	2	1		$\boxed{1}$	$\boxed{2}$	2
						$\boxed{1}$	1				$\boxed{1}$	1

Seven page faults occur.

Space-Time product (assuming time between references are equal):

Integrate (sum) memory cells used over time (in this case, let each time = 1):

1 + 2 + 3 + 4 + 4 + 4 + 5 + 5 + 4 + 3 + 4 + 5 + 5 = 49.

- **Page Fault Frequency (PFF)**

  Chu 1974.

  Since goal of dynamic allocation is to reduce thrashing, measure page fault rate.

  Thrashing is simply a high page fault rate; so want to control PFF.

  Establish an upper and lower threshold bound on the desired page fault rate.

  If fault rate is too high, the process needs more frames.

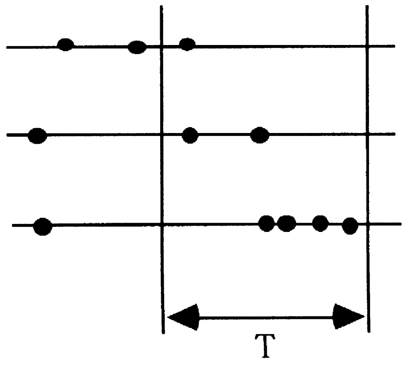
  Increase its allocation.

  If fault rate is too low, process may have too many frames allocated to it.

  Decrease its allocation.

  This will free up frames for other processes in a multiprogrammed system.

- **Example: Time line showing page faults (assume target PFF is 2 faults per T).**
  Three different jobs; three different situations.



Decrease virtual memory allocation (PFF below threshold)

Acceptable PFF w/in a T time window

Increase virtual memory allocation (PFF above threshold)

- Example:
  - Upper threshold: Add a page frame if current page is a fault and last one was a fault.
  - Lower threshold: Remove a page (LRU) if this is the third ref. in a row w/o a fault.

1	2	3	4	1	2	5	1	2	5	3	4	5
$\boxed{1}$	$\boxed{2}$	$\boxed{3}$	$\boxed{4}$	1	2	$\boxed{5}$	1	2	5	$\boxed{3}$	$\boxed{4}$	5
	$\boxed{1}$	$\boxed{2}$	$\boxed{3}$	4	1	$\boxed{2}$	5	1	2	$\boxed{5}$	$\boxed{3}$	4
		$\boxed{1}$	$\boxed{2}$	3	4	$\boxed{1}$	2	5	1	$\boxed{2}$	$\boxed{5}$	3
			$\boxed{1}$	2	3	$\boxed{4}$	4	4			$\boxed{2}$	2

Seven page faults occur.

Space-Time product: 1 + 2 + 3 + 4 + 4 + 4 + 4 + 4 + 4 + 3 + 3 + 4 + 4 = 44.

- Finding the frames to allocate.
  - Free cells are usually kept on a linked list.
  - When the empty cells vary in size, first cell may not be large enough nor optimal.
  - Many algorithms exist for performing cell allocation.
  - Need to compromise b/w the computational cost of finding a cell and its optimality.
  - Optimality depends on the rate of fragmentation.

- Three examples:
  1) Best Fit Strategy
     - Requires a search of all cells available for allocation.
     - Pick the one that is the correct size or is the smallest that is large enough.
  2) First Fit Strategy
     - Scans the list and then allocates the first cell that is large enough.
     - Requires minimal computation, but may promote fragmentation.
  3) Buddy System Strategy
     - Based on the hypothesis that the sizes of cells requested are not random.
     - Assumes if a cell of size x is needed, other cells of size x will also be needed.
     - If size x requested, the next larger power of two size cell is actually allocated.
     - A future request for a cell the same size can then be found without a search.
       - It is just the address of the neighboring cell (the other half allocated).
  - Knuth (1968) performed simulation studies.
    - Buddy system is best of three above.
    - Knuth also found: First fit algorithm is superior to the best-fit algorithm.
      - Best fit will search for closest available cell to that needed.
      - Remaining leftover space is guaranteed to be small.
        - Definition of "best".
      - Results in fragmentation.
      - Many small useless cells are eventually left.

- **Other Frame Allocation Schemes:**
  - Many "Improved" versions of Buddy Scheme exist.
    These generally try to reduce the amount of resulting internal fragmentation.
  - Next Fit: A variation of First Fit.
    Searches from where it left off, instead of at beginning of free memory list.
    Gives worse performance than First Fit.
  - Worst Fit: Opposite approach of Best Fit.
    Tries to leave the biggest hole available so that it might be useful later.
    Empirically shown to be bad.

- **Formal definitions of Swapping vs. Demand Paging**
  - Swapping:
    All pages of a program are loaded into memory blocks when the program beings
    Thus, in order to start, all of the program must be contained in memory.
  - Demand Paging:
    A program is started when some minimum space is available.
    Additional pages are brought into memory only when they are needed.
  - Most systems today are Demand Paging, but also informally called "Swapping".

- **RAM Memory Terminology**

- Cycle Time: The speed with which consecutive blocks of words may be written into or read out of memory. Defined as the minimum time between successive references to storage. Consists of the access time plus a waiting (or refresh) time.

- Access Time: The interval between arrival of the signal activating memory and the completion of write-in or read-out.

- Waiting time: The "resting period" or the time the storage device needs to settle sufficiently before the next reference to it can be performed. Especially needed for RAM (a destructive dynamic memory) which requires memory cells just read to be refreshed.

- Dynamic Storage: Uses continuous recirculation of some physical quantity.

- Volatile Storage: Loses the stored information with time or power-off.
  e.g.) Capacitive memory cell requires power to maintain charge.

- Destructive Read-Out: Process of reading information effectively destroys it.
  Data then needs to be restored or refreshed.
  e.g.) Capacitive memory cell gets discharged when it is read.

- Most semiconductor LSI memory chips are dynamic, volatile, and destructive.
  They require a waiting or refresh time as an inherent part of their cycle time.
  Good memory management reduces the perceived impact of RAM refresh times.

- Memory Interleaving: Divide memory into a number of separate banks.
    - Goal: Avoid CPU idle during wait (restore/refresh) portion of memory cycle time.
    - Various degrees of interleaving architectures are possible ("k-way" interleaving).
        - e.g.) If memory is four-way interleaved, four separate memory banks exist.
            - Each contains one-quarter of the total words in memory.
    - It is common to divide the words in the banks using a modulo function.
        - e.g.) If 4-way interleaving is used, words 1, 5, 9... will be in bank one.

    Assume the CPU can overlap requests between banks (typically so).
    - Thus, while a fetch or restore is in progress for a word in one bank,
        - a word from a different bank may be fetched.
    - CPU should issue requests for words before these words are actually needed.
    - Mainly useful for words in a straight-line program segment (i.e. instructions).
    - Operands (data) normally results in out-of-sequence requests to memory.

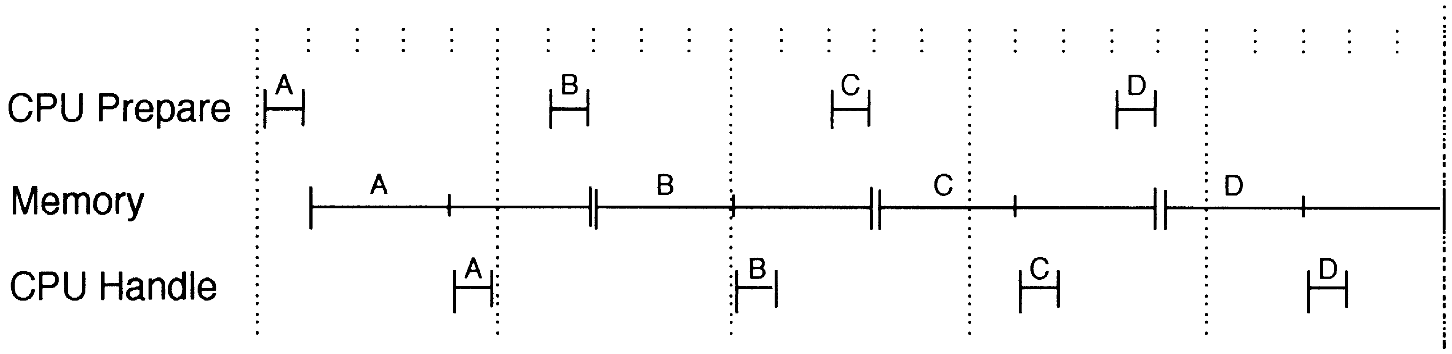
    In this way, it is possible to access consecutive words of memory rapidly.

    Interleaving can have a dramatic effect on the average effective speed of memory.
    - Consider a memory consisting of k modules.
        - Effective cycle time:
            - Decreased by a factor k if all k modules are kept busy continuously.
            - Varies depending upon the program and bank arrangement.
            - Speedup is usually in the range of SQRT(k) to k.
        - Number of modules usually range from 16 to 32.
    - e.g.) Assume a new fetch can be started when previous fetch is half completed.
        - This assumption is reasonable because cycle time = access + wait time.
        - Access Time can typically equal Wait Time.
        - Then, with interleaving, memory will seem to operate twice as fast.

    Implementation:
    - A module is selected by the low-order ($\log_2$ k) bits of the MM address.
    - The high-order bits determine the location within that module.

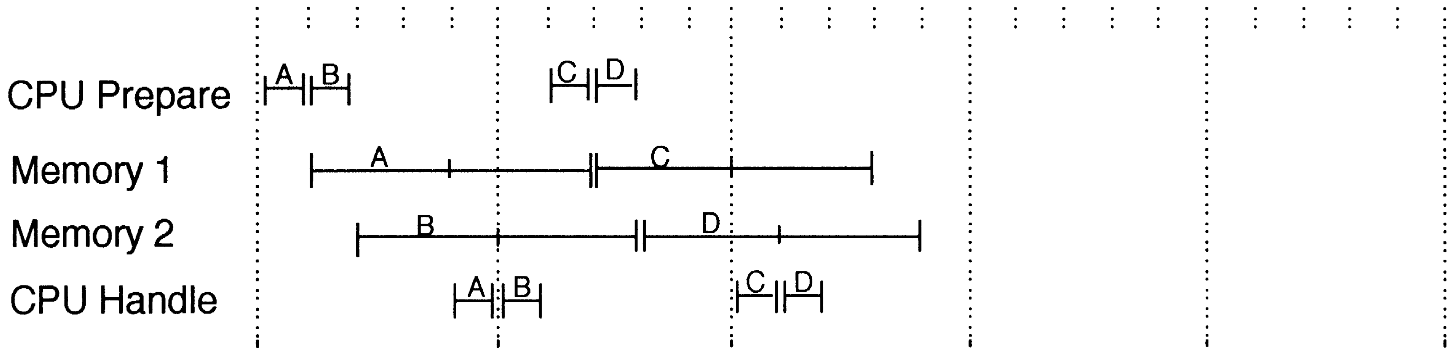
- Example:
    - Assume that Memory has a 0.6 μs Access Time and 1.2 μs Cycle Time.
    - The CPU takes 0.2 μs to prepare a memory request and 0.2 μs to process the result.
    - The CPU can either prepare a memory request or process a result in parallel with the memory. The CPU can issue a request for an operand (e.g. B) before actually receiving and processing a previously requested operand (e.g. A).
    - The total time needed by the overall system to perform an operation spans the time between the CPU's preparation of the very first memory request to the time when the CPU completes processing of the last memory request.
    - Four operands, A thru D are to be retrieved from memory and processed.

Non-Interleaved:



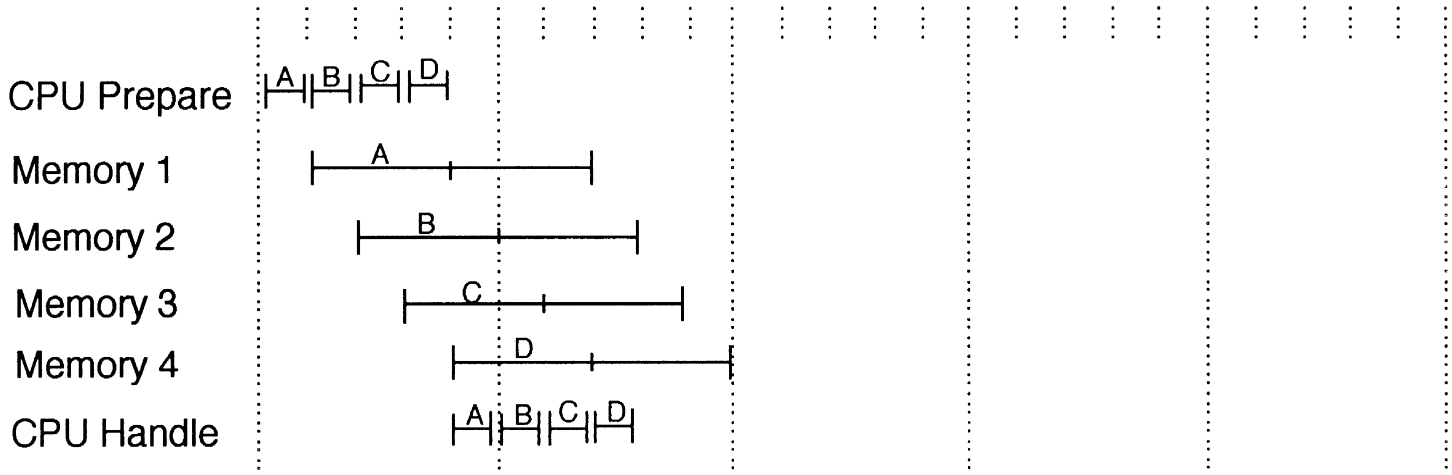
Total Time = 4.6 μs
CPU Utilization = 8 / 23 = 34%

Two-Way Interleaved:



Total Time = 2.4 μs
CPU Utilization = 8 / 12 = 66%
Speedup over Non-Interleaved = 4.6 / 2.4 = 1.9

Four-Way Interleaved:



Total Time = 1.6 μs
CPU Utilization = 8 / 8 = 100%
Speedup over Non-Interleaved = 4.6 / 1.6 = 2.8
Speedup over Two-Way Interleaved = 2.4 / 1.6 = 1.5

- **Interleaved memory makes contiguous block transfers very efficient.**
  So, transferring data in blocks from the MM to the cache can be done quite fast.

- **Data Arrangement in Interleaved Memory**
  Interleaved memory works best with sequential addresses (e.g. instructions).
  Optimization can also be performed for data accesses (e.g. matrices).
  Best arrangement varies with dimensions of matrix; we look at one example.
  Typical operations on a matrix involve access to rows, columns, and diagonals.

- Arrangement 1: "Straight" Storage Scheme across 4 Modules

M1	M2	M3	M4
A(00)	A(01)	A(02)	A(03)
A(10)	A(11)	A(12)	A(13)
A(20)	A(21)	A(22)	A(23)
A(30)	A(31)	A(32)	A(33)

Allows access to Rows and Diagonals without conflict
Conflict on Columns (e.g. A(y2) are all be read from M3 while M1, M2, M4 are idle).

- Arrangement 2: "Skewed" Storage Scheme
Barrel shift each row by an increasing amount
Row 0: No Shift; Row 1: Shift Right One; Row2: Shift Right Two ....

M1	M2	M3	M4
A(00)	A(01)	A(02)	A(03)
A(13)	A(10)	A(11)	A(12)
A(22)	A(23)	A(20)	A(21)
A(31)	A(32)	A(33)	A(30)

Allows access to Rows and Columns without conflict.
(Minor) Conflict on Diagonals.

- Arrangement 3: "Two's Skewing" Storage Scheme
Barrel shift each row by two
Insert an extra module; Skip one module per row (wasteful of memory cells)

M1	M2	M3	M4	M5
A(00)	A(01)	A(02)	A(03)	-
A(13)	-	A(10)	A(11)	A(12)
A(21)	A(22)	A(23)	-	A(20)
-	A(30)	A(31)	A(32)	A(33)

Allows access to Rows, Columns, and Diagonals without conflict.

- **Memory Summary:**

  Future may bring larger and faster memories.

  CPUs might possibly have speed matched memories at comparable costs.

  This would reduce the need for memory hierarchies and cache memory.

  Might also eliminate the need for virtual memory (giga-byte RAM chips).

  Processors without a memory hierarchy would be simpler since:

  - All Memory accesses would always take the same amount of time and
  - No memory management overhead would be necessary

- **I/O Device Management**

  Disk Drive and Tape Drive Units generally consist of two components:

  1) Mechanical

     The actual physical device itself (e.g. the disk drive).

  2) Electronic

     The device controller or adapter (e.g. a circuit card).

- Operating System always deals with the controller; not the actual device.

  Each controller has registers used for communicating with the CPU.

  OS performs I/O by writing commands into the controller's registers.

  After command is issued, CPU can swap to another process.

  Controller issues interrupt to CPU when I/O is completed.

- An important function of the controller is internal buffering.

  Controller converts serial bit stream from device into block of bytes.

  When system bus is ready, the block is (DMA) transferred to main memory.

  Mechanical device (e.g. disk) usually stays in motion.

  Controllers cannot perform input and output at same time.

  Cannot fill its buffer and write its buffer to main memory concurrently.

  Thus, data cannot be read from physical device while writing to main mem.

  Controller can only read every other block on the disk.

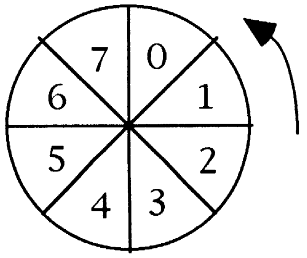
  (Assuming disk read time = main memory write time).

  Therefore, disk formats must be interleaved to allow controller to write to MM.

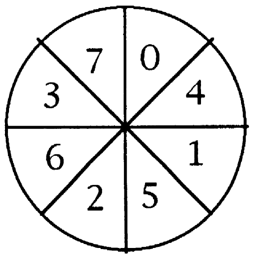
  If main memory write time > disk read time, larger interleave factors required.

  Allows OS to read consecutively numbered blocks at max mechanical device speed.

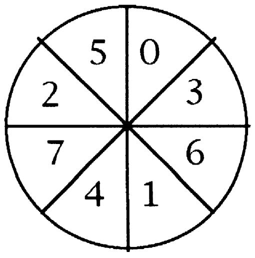
  Examples of various degrees of Interleaving:



Non-Interleaved



Single Interleaved



Double Interleaved

- **Disk Arm Scheduling Algorithms**

  Time needed to read or write a disk block is determined by three factors:
  1) Seek Time
     Time to move arm to proper cylinder (track).
  2) Rotational Delay (Latency Time)
     Time for the proper sector to rotate under the head so reading can begin.
     Disks usually rotate at about 3600 RPM (one rev per 16.7 msec).
     On average, rotational delay is 1/2 a revolution (8.3 msec).
  3) Transfer Time
     Time to read and transfer the data of interest.
     Transfer time = Rotation Time * [(Bytes Transferred) / (Bytes per Track)]

  Seek Time is the most significant of all three above.
  - Seek Time typically greater than Rotational Latency Time.
  - Read arm must be accelerated and decelerated accurately to proper cylinder.

  Scheduling algorithms concentrate on reducing Seek Times.
  - Several disk read requests can arrive during the span of just one seek time.
  - Also, assume a multiprogrammed system.
    - Many different processes may be generating requests to the same disk.
      - Processes may do accesses to disk at different frequencies.
        - Disk does not see Temporal Locality
      - Several different localities of references are getting mixed.
        - Disk does not see Spatial Locality
  - Requests are effectively random.
  - Maintain a linked list entry for all pending cylinder number read requests.

  Goal of Disk Arm Scheduler is to Analyze pending read requests and
  perform some kind of optimization in terms of which request to handle next.

  We examine three disk algorithms as a prelude to processor scheduling algorithms.

  Desirable Characteristics of a Scheduling Policy:
  - Fairness
  - Throughput
  - Mean Response Time
  - Variance in Response Time (i.e. Predictability)

- First-Come, First-Served (FCFS)

  Disk Driver accepts requests one-at-a-time and handles them one-at-a-time.
  Requests are processed in the order received (no reordering of the queue).

  - Example:

Order Received:	11		1		40		16		34		9		12	
Order Processed:	11		1		40		16		34		9		12	
Cylinders Traveled:		10	+	39	+	24	+	18	+	25	+	3	=	119

Advantage: Simple to implement.

Fair to all requests.

Performs acceptably if load is light.

Disadvantage: No optimization.

Can exhibit a random seek pattern.

Successive requests can cause seeks from innermost to outermost cylinders.

Under heavy load, FCFS can result in very long waiting times.

- Shortest Seek First (SSF)

  Scheduler examines the positional relationships among the pending requests.

  Handle the closest request next to minimize arm travel distance (and seek time).

  - Example:

Order Received:	11	1	40	16	34	9	12
Order Processed:	11	12	9	16	1	34	40
Cylinders Traveled:	1 +	3 +	7 +	15 +	33 +	6 =	65

Advantage: Significantly reduces arm motion over FCFS.

Higher throughput and better mean response time than FCFS.

Disadvantage: Under heavy load conditions, arm statistically migrates to center.

Cylinder requests far from middle of disk may get poor (almost no) service.

High variance of response times makes it unacceptable for interactive systems.

Minimal response time goal conflicts with fairness goal.

- Elevator Algorithm (SCAN Scheduling)

  Denning, 1967.

  Similar to elevator control in tall buildings.

  Move in one direction handling all requests along the way.

  Continue movement in one direction until no more requests in that direction.

  Then, switch directions, and repeat process.

  Basis of most disk scheduling strategies actually implemented.

  - Example: (Assume direction is initially "up", i.e. towards higher cyl numbers)

Order Received:	11	1	40	16	34	9	12
Order Processed:	11	12	16	34	40	9	1
Cylinders Traveled:	1 +	4 +	18 +	6 +	31 +	8 =	68

Advantage: Reconciles conflicting goals of efficiency and fairness.

Reduces variance in response times over SSF.

Given any collection of requests, upper bound of total motion is fixed.

Upper bound = 2 x Number of Cylinders

Disadvantage: Requires an extra bit for current direction (either "up" or "down").

Performance usually worse than SSF (in terms of cyls traveled since fairer).

- **RAM Disks**
  - A RAM Disk is a disk device simulated in conventional RAM.
  - Uses a preallocated portion of the main memory for storing disk blocks.
  - Each block is the same size as the block size used on the real disk.
  - Essentially serves as a cache buffer between RAM and Disk.
  - Much faster than conventional disks.
    - Completely eliminates mechanical delays suffered by physically moving device.
    - No seek or rotational delays make it good for frequently accessed data.
    - Reference times are uniform rather than widely variable as with physical disks.
  - Particularly useful in high-performance applications.
  - Disadvantages:
    - More expensive than regular disk.
    - Volatile.

- **Optical Disks**
  - Current CD-ROM technology is slower than conventional hard disk.
    - Current trend is to reduce rotational delays (2X, 4X, 6X speed drives).
  - Generally more durable than a magnetic disk platter.
  - Usually Write-Once-Read-Many (WORM) in nature.
  - Potential storage capacity is tremendous.
    - Some researchers claim 10**21 bits on single disk is possible.
    - 10**21 bits = All documents, pictures, movies, sounds <u>ever</u> recorded.
  - DVD format due late 1996 will be dual layered, dual sided (an entire movie).

- **PARITY**
  - Physical devices are inherently inaccurate; must have built-in error correction.
  - Parity is the most common error detection system used.
  - Obtained by including extra check digit with the information bits.
    - Check digit is set such that decimal sum of 1's in data is either odd or even.
  - These codes can detect single errors, but not necessarily multiple errors.
  - Thus, undetected errors are always possible (but can be made highly improbable).
  - Multiple, successive bit failures can be especially high in magnetic tape dropouts.
    - We can use a double-parity check scheme.
    - Consider the set of numbers as a bit matrix (rows and columns).
    - Parity is computed for both the rows and the columns.
  - This enables double error detection; single error correction.
    - e.g.) A double error in the first row will result in a good row parity.
      - However, 2 columns will then be incorrect thereby detecting double error.
      - But correction cannot be accomplished since location of fault not known.
    - e.g.) Any single error will produce a parity error on the row and column for it.
      - This provides the row and column coordinates of the incorrect bit.

- **Introduction to Scheduling: Process States**

  A Process moves through various states between being submitted and completed:

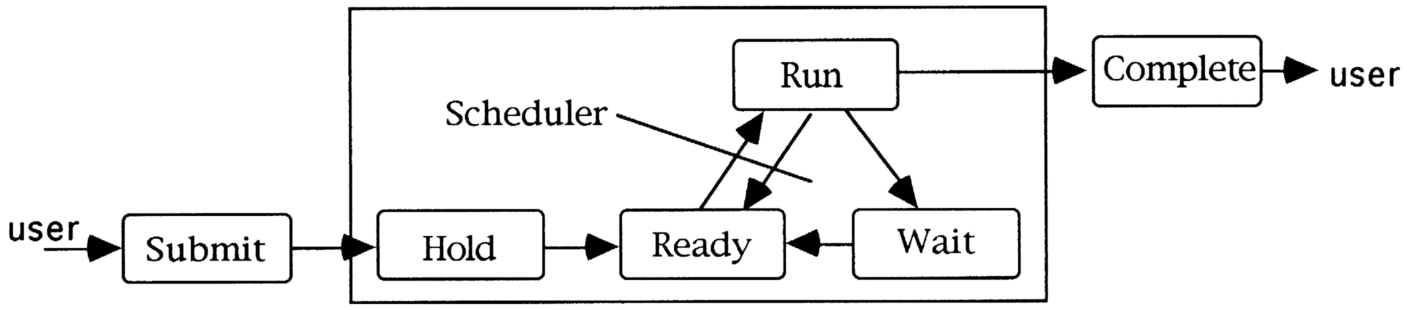
  1) Hold State: A user's job has been Submitted and Spooled onto disk.
     No resources have been allocated to it yet to move it to the ready state.
     It has simply been "read" into the system's high-speed storage area.
     There can be many jobs (from many users) in the Hold State.

  2) Ready: Process is "ready to run", but must wait for its turn on the processor.
     The resources (e.g. memory) it needs are available & can be allocated to it.
     Process must wait for the Scheduler to give it time on the CPU.
     There can be many processes in the Ready Queue waiting for the CPU.

  3) Run: Process is currently being executed.
     Process has been allocated the CPU and any other resources it needs.
     On a uniprocessor, only one process at a time can be in the Run state.
     Process continues to possess the CPU and run until it either:
     - Is interrupted by the Scheduler (because its time slice is up), or
     - It performs I/O, thereby moving itself to the Wait State.

  4) Wait: Process is waiting for some event to happen (e.g. I/O operation).
     Process gives up CPU while waiting for a slower device to complete.
     Also called the Blocked State (i.e., process blocks itself by doing I/O).
     After I/O is complete, process returns to Ready Queue.
     There can be several processes waiting for I/O to complete for them.

  5) Complete: Process has finished all CPU execution and I/O.
     All resources are now deallocated and freed for other processes to use.



The life cycle of a process can be represented by transitions between these states.

- Typical Scenario:
  A user submits job to system.
  <u>Spooler</u> stores it onto disk in a Hold State.
  If <u>Memory</u> and <u>Device Manager</u> say OK, job is put onto the Ready Queue.
  <u>Scheduler</u> scans list of ready processes, chooses one, and runs it.
  Process will typically cycle many times through Ready-Run-Wait States.
  When process competes, all allocated resources are freed.

- **Scheduler:**
  - Chooses and controls the job that the CPU is (will be) working on (next).
  - Ensures that the right tasks from the Ready List execute at the right time.
    - Highly integrated and linked with resource allocation issues.
      - Assumes that the CPU is a valuable resource of prime importance.
        - Every second can represent millions of instruction executions.
  - Shares the processor(s) between the processes that are able to run.
    - Generally, there will be many active processes and just one processor.
  - Enforces scheduling and priority rules.
  - Scheduler takes action whenever:
    1) The current process stops executing (due to it performing I/O), or
    2) It believes that the CPU can be better utilized on a different process.

- Scheduler should guarantee reliable and efficient system performance.
  - "Efficient" generally means: Try to keep the CPU busy 100 percent of the time.
  - "Performance" is a function of three factors:
    1) Throughput:
       - Number of jobs completed by the computer in a given period of time.
       - Function of both hardware speed and software facilities.
    2) Response Time:
       - Time interval between a request being made and a reply received by user.
       - Synonymous with "Turnaround Time" for batch jobs.
    3) Availability:
       - "Uptime", i.e., The system's ability to handle requests for computation.
       - A function of reliability and dependability of both the H/W and S/W (OS).
  - Definition of "Most Efficient Performance" may vary from system to system.
    - For Batch System, we may want to maximize total throughput.
    - For Interactive System, we may want to minimize average response time.
    - For Critical Application (e.g. Space Shuttle), we want reliability and availability.
  - => System Performance is highly dependent upon the Scheduler.

- Scheduler should strive to achieve the following Goals:
  - Be Fair: Treat all processes the same. Ensure each one gets a fair share of CPU.
  - Maximize Throughput: Service the largest number of processes per unit time.
  - Minimize Response Times: While serving a Maximum number of Interactive users.
  - Minimize Turnaround Time: For batch users to receive output from job.
  - Avoid Indefinite Postponement (which can be as severe as deadlock).
    - No job within the system should have to wait forever (also called Starvation).
    - Best accomplished by aging:
      - As a process waits for a resource, its priority should grow.
      - Eventually, priority becomes high enough for process to get resource & run.

- Achieve a Balance between Response and Utilization.
  - Minimize worst-case response times *within specified constraints.*
  - Best way to guarantee good response time is to always have resources avail.
    - This strategy is used in real time systems where fast response is essential.
    - However, the cost will be reduced resource utilization.
  - Economics often makes effective resource utilization imperative.
- Equalize Resource Utilization.
  - Keep system balanced and all resources busy.
  - Avoid having a bottleneck on one resource when other resources are idle.
  - Give preference to tasks likely to use underutilized resources.
- Give preference to processes holding key resources.
  - Even if low-priority process is holding a key resource, the resource may be in demand by a high-priority process. Scheduler should give the process holding resource better treatment than it would ordinarily receive so that the process will release the key resource sooner.
- Give higher priority to processes exhibiting good behavior (e.g. low fault rate). *[?]*
  - *Some people think* that this will ensure that:
    - The current state of operation is efficient (no resource bottleneck).
    - The badly performing jobs are postponed until the near future.
      - Enables current jobs to complete and free up resources, thereby, increasing odds that bad job will perform better in future.
- Be Predictable: A given job should run in about the same amount of time and space (Space-Time Product) cost regardless of the load on the system.
- Enforce Priorities: If environment is one where processes are given priorities, then the scheduler should favor the higher-priority processes.
- Degrade Gracefully under heavy loads.
  - A scheduling mechanism should not collapse under heavy system loads.
  - Under heavy load, system should either:
    - Prevent excessive loading by disallowing new incoming processes, or
    - Continue working at a reduced performance level for all processes.
- Minimize overhead:
  - Scheduler should not take (too much) time away from real processes.
  - Scheduler must work quickly and efficiently.
    - An intensively used part of the OS (used as much as the memory manager).
  - Global OS Theme: A *little* expended overhead should *greatly* improve perf.
  - That is, the Operating System Scheduler overhead should "pay for itself".
- Work well with all types of processes, each of which is unique and unpredictable.

Note: Many of these goals are contradictory and will require tradeoffs to be made.
e.g.) To improve the average response time, long jobs may have to wait a long time.

=> Scheduling is a complex (multi-dimensional optimization) problem.

- **Non-Preemptive vs. Preemptive Scheduling**

  - Non-Preemptive Scheduling:
    Tasks stop only when complete (i.e. Run to Completion).
    Scheduler cannot stop tasks once they are executing.
    Scheduler must try to predict task requirements to minimize turn-around time.
    Commonly used in batch systems.

  - Preemptive Scheduling:
    Allows the process to be interrupted whenever the scheduler sees fit.
    e.g.) Scheduler might want to handle a newly arriving process.
    The OS can force tasks to stop when a new task arrives with a higher priority.
    Preemption incurs some overhead due to process or context switching.
    Need to take a 'snapshot' of the current state and save it for later restore.

Note: To simplify discussion of Scheduling, we assume processes never perform I/O.
i.e., Processes move only between the Ready and Run States; never the Wait State.
Also, we assume fixed, static priorities specified by the user upon job startup.

- Priority can be assigned to processes either Statically or Dynamically.
  The OS usually reserves highest priority for itself over any user's job.
  Scheduler and Memory/Device Managers must always be in control.

  - Static Priority: Task priority is fixed, generally at the beginning of its execution.
    Once a job's priority is established, it is never changed.
    Static Characteristics may include:
    Memory size, estimated execution time, estimated amount of I/O activity.
    The best source of these estimates are from the user himself.
    In some cases, a user willing to pay a premium can state a high prio.
    e.g.) Cray allows 4X normal priority if user pays 4X the cost.
    Problems:
    Malicious user could deliberately skew estimates to gain higher priority.
    Some low priority processes could languish in queue and never run.
    Infinite supply of high priority jobs indefinitely postpones lower ones.
    Also, response / turnaround times may vary widely and unpredictably.
    Depends not only on number of jobs, but also on their priority mix.
    Difficult to handle many processes with identical priorities.
    All jobs cannot run at once and no method to rank them.

  - Dynamic Priority: OS can change priority if conditions change.
    Used in multiprogramming and real-time operating systems.
    Dynamic Priorities can be assigned by the system and can change with time.
    Requires more overhead than a static priority scheme.
    Priority needs to be recalculated several times over life of process.

Dynamic Characteristics include:

- A process's current resource allotment and resource requests.
- Amount of recent I/O operations.
- Page-fault rate.
- Recent amount of processing time.
  - Total time spent in execution vs. total time in system (wall clock time).
  - Total time spent waiting vs. total time in system.
- Projected requirement of running time for completion of the task.

• Some possible prioritization philosophies:

- In regards to a process's current resource allotment and resource request:
  - Give tasks freeing resources higher priority than tasks acquiring resources.
  - Give processes which possess large number of resources higher priority.
    - Also, grant further requests to these processes whenever possible ASAP.
    - Rationale: Try to get resource-intensive processes to complete faster.
      - This way, the resources they possess are freed faster too.
    - Problem:
      - Resource-intensive jobs may monopolize the machine.
- In regards to a process's amount of recent I/O operations:
  - Give tasks using peripheral devices high priority.
    - Ensures that peripherals are kept busy as much as possible.
      - Reduces the potential of I/O bottlenecks.
    - Tasks using faster devices should have higher priority than slower devices.
    - Example Implementation Scheme:
      - I/O bound processes might have priority set to 1/f
        - where f is fraction of the last CPU time slice used by the process.
      - Assume a process is allocated a 100ms time slice of the CPU.
      - So, a process that used only 2ms of CPU time before it generated an
        - I/O request and returned to the wait state has a priority 1/.02= 50.
      - Whereas, a process that used 100ms/100ms has a low priority of 1.

• It is interesting to note that even today, some designers have different philosophies.

- These philosophies can lead to radically different prioritization schemes.
  - Which, in turn, can lead to radically different OS Scheduling algorithms.
- An example of different philosophies which leads to a different priority scheme:
  - In regards to a process's page fault rate:
    - Argument A: Favor processes that have low fault rates because
      - they have established their working sets.
    - Argument B: Favor processes with high fault rates because
      - they use the CPU only briefly before generating an I/O request.

Another example:

- In regards to a process's recent amount of processing time:

  Argument A: Favor processes that have received little execution time in order to give them equal treatment (i.e. their fair share of processing).

  Argument B: Favor a process that has received much execution time, because it should be near completion and want it to exit ASAP.

• The number of possible scheduling algorithms is very large.
  Analysis and simulation is often restricted by the complexity of most systems.
  Only means of selecting an optimal algorithm for a specific case is by experiment.
  A Scheduling philosophy can be classified based on:
  - The occasions on when the processor can be reallocated (preempted), and
  - The algorithm used to choose the next process to be given the CPU and run.

• First-In First-Out (FIFO), or First Come First Served (FCFS), or "Run-To-Completion"
  Simplest non-preemptive algorithm.
  Next task to execute is the always the first in the queue.
  Applied when there is no reason to give preference to one job over the other.
  Advantages:
    Fairest algorithm. Task that has been waiting the longest gets to go first.
    Easy to keep track of queue.
  Disadvantage:
    Not an adequate strategy when both large and small tasks are present.
    Expected time spent in the system is same for all jobs.
      Short jobs do not get through system any faster than long jobs.

- Example FCFS: Assume 5 jobs with their specified remaining time (T_r) to completion.

Job Name	Job Arrives After Time	T_r = Time to Completion
A	0	5
B	1	2
C	2	3
D	4	1
E	7	4

	T_r	1	T_r	2	T_r	3	T_r	4	T_r	5	T_r	6	T_r	7	T_r	8	T_r	9	T_r	10	T_r	11	T_r	12	T_r	13	T_r	14	T_r	15
A	5	--		--		--		--		--	0				0						0		0							
B											2	--		--	0						0		0							
C											3				3	--		--		--	0		0							
D											1				1						1	--	0							
E															4						4		4	--		--		--		--

- **Highest Priority First (HPF)**

  Assign the processor to the process which has the highest priority.

  - **Nonpreemptive version HPF:**

    Highest priority process executes until it finishes.

    If a process arrives with higher priority than running process, it must wait.

    Example: The 5 jobs are given priorities (1 = Lowest) and varying arrival times.

    Assume once a job runs, it never does I/O (never goes to wait state).

Job Name	Job Arrives After Time	Job Priority	T_r = Time to Completion
A	0	2	5
B	0	3	2
C	1	5	3
D	3	1	1
E	3	4	4

	T_r	1	T_r	2	T_r	3	T_r	4	T_r	5	T_r	6	T_r	7	T_r	8	T_r	9	T_r	10	T_r	11	T_r	12	T_r	13	T_r	14	T_r	15
A	5				5						5								5	--		--		--		--		--	0	
B	2	--		--	0						0								0										0	
C					3	--		--		--	0								0										0	
D											1								1										1	--
E											4	--		--		--		--	0										0	

  - **Preemptive version HPF:**

    A higher priority arrival interrupts the running process's execution.

    Preempted process is returned to the ready queue.

    Example: Assume priority 1 is Lowest and jobs never perform I/O.

Job Name	Job Arrives After Time	Job Priority	T_r = Time to Completion
A	0	2	5
B	0	3	2
C	1	5	3
D	3	1	1
E	3	4	4

	T_r	1	T_r	2	T_r	3	T_r	4	T_r	5	T_r	6	T_r	7	T_r	8	T_r	9	T_r	10	T_r	11	T_r	12	T_r	13	T_r	14	T_r	15
A	5		5				5		5								5		5	--		--		--		--		--	0	
B	2	--	1				1		1								1	--	0										0	
C			3	--		--	1	--	0								0		0										0	
D							1		1								1		1										1	--
E							4		4	--		--		--		--	0		0										0	

- In general, priority schemes vary depending on:
  - Components of the priority, and the use of preemption.
  - Other priority schemes include:
  - Basing priority on I/O Activity:
    - An important goal of scheduling is to keep peripheral units busy.
    - Processes should be given high priority during periods of heavy I/O activity
    - Easily monitored, since process goes to blocked state after invoking I/O
    - Set the process priority inversely proportional to length of time since its last I/O
    - Empirical evidence shows programs tend to do I/O in concentrated bursts.
    - Those processes doing the most I/O have best chance of gaining control of CPU
    - Therefore, this strategy will keep peripheral devices busy most of the time.
  - Basing priority on estimated remaining run time:
    - One of the most common HPF strategies schedules the shortest job first (SJF).
    - Priority is inversely proportional to processing time; give high prio to short jobs.

- SJF / SRT: Process the job that will finish sooner first.
  - Non-Preemptive version called Shortest Job First (SJF):
    - Can be viewed as an improvement to non-preemptive FIFO.
      - SJF reduces average waiting time over FIFO.
        - SJF favors short jobs at the expense of larger ones.
        - Idea is to try to ensure that the next job run will leave system ASAP.
        - This tends to reduce the number of waiting jobs
      - But can have larger variance (i.e. more unpredictability) esp. for large jobs.
    - Suitable only for batch systems.
    - Always results in the minimum average turn-around time.
      - Works best in an environment where short jobs form the bulk of the load.
    - Next task selected is the shortest one (lowest predicted CPU time & Memory).
      - Clearly advantageous for a process to have low estimated execution time.
    - Since exact execution time is rarely known in advance, an estimate is used.
      - On some systems, user provides estimate of resources needed by job.
      - Need to prevent abusive user from understating requirements of a task.
        - e.g.) Abort task if actual resources >> stated predicted resources.
    - Can completely lock out large tasks given an infinite supply of small tasks.
      - Possible remedy: Aging
        - Let large job execute if waiting time (age) exceeds some threshold.
        - Current Elapsed time = Waiting time + Accumulated Service Time.
        - Use a priority scheme based on:
          - (Current Elapsed Time) / (Accumulated Service Time)
        - Allows actual times, not estimates to be used.
        - Requires more overhead in tracking times.
        - Ratio needs to be set to one until first service increment is received.

- **Example Shortest Job First (SJF):**

  Non-Preemptive: Once the shortest job is identified and started, scheduler cannot re-evaluate the Ready queue until the job completes.

Job Name	Job Arrives After Time	T_r = Time to Completion
A	0	5
B	0	2
C	1	3
D	3	1
E	3	4

	T_r	1	T_r	2	T_r	3	T_r	4	T_r	5	T_r	6	T_r	7	T_r	8	T_r	9	T_r	10	T_r	11	T_r	12	T_r	13	T_r	14	T_r	15
A	5				5						5		5								5	--		--		--		--		--
B	2	--		--	0						0		0								0									
C					3	--		--		--	0		0								0									
D											1	--	0								0									
E											4		4	--		--		--		--	0									

- **Preemptive version SJF:**

  Usually called Shortest Remaining Time (SRT).

  Process with the smallest estimated run-time to completion is run next.

  New arrivals with a shorter estimated run-time can preempt current process.

  Scheduler must re-evaluate Ready queue when a new job arrives.

  Also requires estimates of run-time, probably provided by user.

  - **Example: Shortest Remaining Time (SRT)**

Job Name	Job Arrives After Time	T_r = Time to Completion
A	0	5
B	0	2
C	1	3
D	3	1
E	3	4

	T_r	1	T_r	2	T_r	3	T_r	4	T_r	5	T_r	6	T_r	7	T_r	8	T_r	9	T_r	10	T_r	11	T_r	12	T_r	13	T_r	14	T_r	15
A	5		5		5		5		5				5								5	--		--		--		--		--
B	2	--	1	--	0		0		0				0								0									
C			3		3	--	2		2	--		--	0								0									
D							1	--	0				0								0									
E							4		4				4	--		--		--		--	0									

SRT Requires more overhead than SJF.

- Need to keep track of elapsed service time of running job.
- Must handle occasional preemptions.
- When should a running job (which is almost complete) be preempted ?
  - Suppose new job has estimated time only slightly less than current job.
  - Pure SRT will always preempt if new job has shorter remaining time.
  - But wiser strategy will factor into account overhead time to preempt:
    - Only preempt if: |Current RemTime - New RemTime | > Overhead
- *Weigh overhead of resource mgmt mechanism against anticipated benefits.*

• Recall that SJF can be thought of as HPF based on shortest estimated run time.

- That is, SJF is priority based.
- The number of different priority schemes is virtually limitless.
- Unfortunately, HPF schemes can keep low priority processes waiting for a long time.
- In some systems, this is unacceptable.
  - e.g.) Time-sharing user requires 1 second of execution every 5 seconds, rather than a 12-minute interval of execution every hour.
  - Need a more predictable, finer-grain, time-based scheduling algorithm.

• Round-Robin (RR)

- Inherently a Preemptive algorithm.
- Not based on any static or dynamic priority information.
  - Does not rely on information which needs to be supplied by user.
- Gives rapid response to short jobs when running times are not known in advance.
- Each task in Ready List is executed in order.
- Would be very similar to a <u>pre</u>emptive version of FCFS or FIFO.
  - Each task executes until it needs to wait (e.g. for I/O), or completes.
  - Scheduler then starts execution of the next task on the Ready List.
  - Task that is waiting reenters Ready List at the end of the queue.
- Round Robin includes the concept of time sharing.
  - Processor's time is divided into intervals called quantums (q).
  - Each task is given a maximum execution time limit (time slice) equal to q.
  - Top process in ready queue is allowed to run for q seconds.
  - If task has not completed by q seconds, it is interrupted by scheduler and put at the end of the Ready List.
  - Next process then gets its turn at the CPU for q seconds.
  - If there are K processes in the ready queue, then:
    - Each process gets q seconds out of every Kq seconds of processor time.
    - Each process "sees" a processor with speed 1/K that of the physical CPU.
  - Size of the ready queue is an important design parameter.
    - Determines how fast processes can make progress.

q affects how evenly the processor is distributed over short periods of time.

Typical quantum time slices are on the order of 50 to 100 milliseconds.

-If q is infinite, then RR reduces to FCFS.

Large q results in a higher chance of more random variance in wait time.

e.g.) If four new processes arrive just as a process is finishing its quantum, its next quantum will be delayed by 4q.

e.g.) A process can get lucky if all processes ahead of it get blocked.

-If q is decreased, then:

Random effects are reduced since the probability of an arrival or blockage in any single quantum becomes small.

Also, as q is decreased, service for smaller processes improves.

Small q gives all ready processes equivalent level of service.

However, q should not be made too small.

As q approaches process switching time, overhead becomes too high.

Overhead directly proportional to amount of context switching.

e.g.) If q = 20ms and context switch time = 10ms, overhead is 50%

System can spend more time context switching than executing.

For "reasonable-sized" quanta,

Overhead time to switch from one process to another is negligible.

Time slice is usually physically enforced by an interval timer or real time clock.

Gives control back to scheduler upon time slice expiration.

Ensures that no task can monopolize the processor; each one gets its "fair-share".

RR is Particularly useful on large, interactive, time-shared systems.

System should shift its attention often enough so that each interactive user perceives himself getting continuous attention from CPU.

But should also enable users to execute time consuming commands too.

Processes which require lengthy service will cycle round the queue several times.

The expected time a job spends in system is related to service time it needs.

Short jobs will get through much faster than long jobs.

Processes whose service time is less than the quantum will terminate on first cycle.

Both long and short jobs finish in time proportional to time independently needed.

Small jobs will not suffer as much from presence of long jobs as in FIFO.

RR can only be applied to a preemptive resource such as CPU.

Cannot be used by a nonpreemptive resource (e.g. line printer).

DEC PDP-8 TSS-8 used round-robin dispatching system.

Problem:

Performance can degrade suddenly if load too heavy for given quantum size.

A possible solution is to increase quantum size as number of processes rises.

This increases probability that process will terminate within the quantum.

Reduces the overhead of process switching.

- **Example Round Robin (q=1)**

Job Name	Job Arrives After Time	T_r = Time to Completion
A	0	5
B	0	2
C	1	3
D	3	1
E	3	4

	T_r	1	T_r	2	T_r	3	T_r	4	T_r	5	T_r	6	T_r	7	T_r	8	T_r	9	T_r	10	T_r	11	T_r	12	T_r	13	T_r	14	T_r	15
A	5	--	4				4	--			3		3					--					2	--				--	0	
B	2		2	--			1			--	0		0										0						0	
C			3			--	2				2		2			--						--	0						0	
D							1				1	--	0										0						0	
E							4				4		4	--						--			2			--			1	--

Implemented using a circular queue structure.

Our protocol: Append new arrivals to the end of the queue *before* returning the preempted job to the end of queue.

Time	*Queue*	*Processing*	
1	B	A	
			C Arrives
2	A C	B	
3	B A	C	
			D & E Arrive
4	C E D B	A	
5	A C E D	B	
			B Completes
6	A C E	D	
			D Completes
7	A C	E	
8	E A	C	
9	C E	A	
10	A C	E	
11	E A	C	
			C Completes
12	E	A	
13	A	E	
14	E	A	
			A Completes
15		E	
			E Completes

- **Modified Round-Robin**
  - One can argue that RR is unfair to long jobs.
    - New job is accepted and given same treatment as an old job.
    - Some argue that system should recognize obligation to committed jobs.
      - Newly arriving job should not immediately receive resources.
        - Especially if resources are needed by old jobs already running.
    - Want new job admittance condition to depend upon:
      - 1) The arrival time of the new job.
      - 2) The number of jobs currently sharing the system.
  - Very similar to RR except that a job which just received a service quantum is
    - not put at the end of the queue, but to a place somewhere in the middle.
  - Jobs competing for service in queue are divided into new and accepted jobs.
  - Moveable partition divides queue.
  - Accepted jobs are scheduled RR.
    - Job which just received quantum is put to end of the accepted portion of queue.
  - A new job does not receive any service at all until a certain condition is met.
    - Condition is usually aging priority.
      - New job accumulates priority with wait time.
      - New job runs when priority equals priority of accepted jobs.
    - When condition met, new job first in line is appended to end of accepted queue.
    - New job is accepted by moving partition one step to left.
  - Problem:
    - Very difficult to implement.
    - Priority for both new and accepted jobs must be updated every time slice.

- Example Modified Round Robin (q=1)

Job Name	Job Arrives After Time	T_r = Time to Completion
A	0	5
B	0	2
C	1	3
D	3	1
E	3	4

	T_r	1	T_r	2	T_r	3	T_r	4	T_r	5	T_r	6	T_r	7	T_r	8	T_r	9	T_r	10	T_r	11	T_r	12	T_r	13	T_r	14	T_r	15
A	5	--				--			3	--		--		--	0				0								0			
B	2			--				--	0						0				0								0			
C															3	--			2			--				--	0			
D															1			--	0								0			
E															4				4	--				--			2	--		--

Up through t=7, only admit 2 active: BA; After t=7, move partition to admit 3 active: EDC

- **Two-Level Queue or Foreground / Background Systems**
  - Variation of the Round-Robin algorithm.
    - Another attempt to overcome sudden load degradation problem noted above.
  - Considered to be a "long-range" scheduling policy.
    - Effectively keeps history of previous passes through system into account.
  - Utilize a dual scheduling strategy.
  - Processes not completed w/in a fixed number of quanta are moved to background Q.
  - Background Queue:
    - Only serviced if there are no processes in Foreground.
    - Can be treated RR with a larger time quantum, or FCFS.
  - Short jobs which depart before reaching second stage are serviced faster than when a RR scheduling discipline is uniformly applied to all jobs, irrespective of stage.
  - Often used in mixed batch and multi-access environments.
    - Allows batch jobs to move to background while servicing interactive foreground.
  - Dual queues enable system to take advantage of distribution of execution times.
    - e.g.) Suppose it is known that if a process does not complete in 3q, then it will probably not complete until receiving 100q. This property of execution time can be exploited by using two queues. Each process goes through 3q in foreground, then gets dropped to background queue.
- Algorithm can be generalized to Multi-Level Queues (also called Feedback Queues).
  - Multiple queues are chained together, each representing a different priority class.
    - Each stage is subject to an individual scheduling discipline.
    - Job enters stage S(i) if its accumulated service time exceeds threshold T(i).
  - Process starts at highest class (the first queue) and progressively moves to successively lower priority queues (classes) with each time limit expiration.
  - Lower priority classes only run if no processes are in higher priority classes.
    - Long processes get serviced less and less frequently as it moves down chain.
    - Ensures that CPU is readily available to service short, interactive processes.
  - Lower classes are given increasing larger time limits.
    - CPU-intensive processes get larger time slices at lower levels of priority and without the overhead caused by excessive context switching in high prio Q.
  - - Example: Use increasing powers of 2 for time limits of queues.
    - Q1: Processes in this highest priority queue are run for one quantum.
    - Q2: Next lower priority queue allows processes to run for two quantum.
    - Q3: Further lower priority queue has 4 quantum time limit, etc.
    - So, a CPU-intensive process needing 100q would swap only 7 times.
      - 1 + 2 + 4 + 8 + 16 + 32 + 37 = 100
    - Better than 100 swaps through single queue, Q1, if pure round robin were used.
  - - Example: DEC System-10 has three round robin queues
    - Quantas for level one, two, three are: .02 sec, .25 sec, 2 sec respectively

- Example: UNIX preemptive dynamic priority algorithm.
    - 1) All system processes have priorities greater than highest user-process priority.
    - 2) User-process priorities are:
        - Determined by the ratio of elapsed time to CPU time used over a fixed period.
        - Updated once per second with control passed to scheduler after each update.
    - Thus:
        - System processes always get preference over user processes.
        - A set of CPU-bound processes will run round-robin with a 1s time quantum.
        - The system is self-regulating since:
            - I/O-bound processes will retain high priority.
                - Ensures good interactive terminal response.
            - CPU-bound processes will have its priority lowered rapidly.
                - However, it will not be ignored entirely.
                - While it is not using CPU, its priority will rise until it gets to run again.

• Scheduling Summary

FCFS is simplest algorithm, but can cause short jobs to wait for very long jobs.

SJF provides shortest average waiting time, but difficult to implement since it requires knowledge of run times.

HPF can be considered generalized SJF with highest priority given to shortest job.
- Requires externally provided priority ranking.

RR is more appropriate for a time-shared system.

FCFS is non-preemptive.

Both HPF and SJF can be either non-preemptive or preemptive.

RR is preemptive.

Both SJF and HPF may suffer from starvation.
- Aging is a technique that can be used to prevent starvation.

RR requires selection of a time quantum.
- If q is too large, RR degenerates to FCFS
- If q is too small, the overhead from context switching becomes excessive.

Normally, job scheduling is heavily influenced by resource allocation issues.
- A job can run if scheduler approves <u>and</u> if its required resources are available.
    - Resources include memory, disk drives, tape drives, printers, etc.
- Previous discussion completely ignored resource allocation considerations.
    - In fact, we assumed no I/O requests were ever generated by processes.
- Next topic, Resource Management, is discussed independent of scheduling.

- We have see how the OS is involved in Memory and CPU Management.
  - In general, the OS is involved in all aspects of computer resource management.
  - In a multiple-resource system:
    - OS must be concerned with the interactions among processes and resources.
    - As processes run, they request and (if granted by the OS) acquire resources.
    - Resources are shared among multiple processes that are active in the system.
    - Conflicts can arise due to the sharing of finite resources.
    - The most significant resource allocation problem is that of Deadlock.

- Deadlock (Deadly Embrace, or Indefinite Postponement)
  - A system of resources can cause a system of processes to deadlock whenever any two or more processes are forced to wait (in a blocked state).
  - It is possible that the waiting processes will never again become "ready" because the resources they have requested are held by other waiting processes.
  - Deadlock can occur if a pattern of waiting processes is established as follows:
    - P2 is forced to wait for P1. P3 is forced to wait for P2. P1 is waiting for P3.
    - In effect, P1 ends up waiting for itself.
      - Since P1 is blocked, it cannot execute and remove itself from waiting.
    - That is, a *cycle* is established:
      - P1 <- P2 <- P3 <- P1
  - For example:
    - Assume a system with four tape drives and two processes.
    - If each process has 2 tape drives and needs 3, each will wait for the other.
  - Deadlock scenarios are not unique to the operating system environment.
    - Consider 2 people crossing a river from opposite sides.
    - At most, one foot can be on each stepping stone at a time.
    - To cross the river, a person must use each of the stepping stones.
    - Each person crossing river is a process and stepping stones are resources.
    - Deadlock can occur if:
      - Person starts to cross river without first finding out whether someone else is trying to cross from the other side.
      - Two people start crossing river from opposite sides and meet in the middle.
    - Avoiding deadlock requires each person crossing river to follow a protocol.
      - One such protocol might be to see if anyone is crossing from other side.
      - If so, wait until they are done. Otherwise, it is safe to cross.
      - Must also handle situation where 2 people want to cross at "same" time.
        - If both go, deadlock occurs. (Both stuck in the middle of the river)
        - If both wait, deadlock also occurs. (Both wait forever)
      - One solution is to give one side of the river higher priority to break the tie.
      - But, now given infinite supply from one side, deadlock can still occur.
      - Modified solution: Alternate the direction of crossing periodically.

- **A system has a number of resources to be distributed among competing processes.**
  The sequence of events required for a process to use a resource is:
  - **1) Request the resource.**
  - **2) Use the resource.**
  - **3) Release the resource.**

  If a resource is not available when it is requested, the requesting process must wait.
  In a deadlock, processes never finish executing and never release resources.
  This, in turn, prevents other jobs from obtaining their requested resources.
  Eventually, no process can go, all work stops, and the entire system "hangs".

- **Four conditions are necessary (but not sufficient) for deadlock (Coffman, 1971).**
  - **1) Mutual Exclusion Condition.**
    Only one process at a time can use a particular resource.
    If another process requests that same resource, it must wait until it is released.
  - **2) Hold and Wait Condition.**
    Processes holding resources granted earlier can request new resources.
    If process must wait for the new resource, it holds the ones it currently has.
  - **3) No Preemption Condition.**
    Resources previously granted cannot be forcibly taken away from a process.
    They can only be released voluntarily by the processes holding them.
  - **4) Circular Wait Condition.**
    There must be a circular chain of two or more processes.
    Each is waiting for a resource held by the next member of the chain.

- **Resource Graphs (A method for Deadlock Detection)**
  Holt in 1972 modeled the four conditions for deadlock using directed graphs.
  Square nodes are used to represent resources.
  Round nodes are used to represent processes.
  Arc from a resource to a process means resource is currently held by that process.

$$\begin{tikzcd}\boxed{R} \arrow[r] & \textcircled{A}\end{tikzcd}$$

Process A holding (using) Resource R

Arc from a process to resource means process is currently blocked for that resource.

$$\begin{tikzcd}\boxed{S} & \textcircled{B} \arrow[l]\end{tikzcd}$$

Process B requesting (but must wait for) Resource S

If any cycles occur in the graph, then a deadlock situation exists.

$$\begin{tikzcd} & \textcircled{A} \arrow[dr] & \\ \boxed{R} \arrow[ur] & & \boxed{S} \arrow[dl] \\ & \textcircled{B} \arrow[ul] & \end{tikzcd}$$

Process A holds R, waiting for S

Process B holds S, waiting for R

- There are four general classifications of the methods used in dealing with deadlock.
  Tradeoff between overhead and effectiveness depends upon designer's judgment.

- Method #1: Ignore the problem completely.
  - That is, Let the user push the reset button if and when system deadlocks.
  - May make sense from an engineering standpoint.
  - Assumption: A "minuscule" probability of a deadlock occurring is "acceptable".
  - Does not incur any penalty in performance or convenience.
    - Other methods put restrictions on processes and/or programmers.
  - UNIX, for example, essentially ignores deadlock problem.
    - Assumes users prefer occasional deadlocks to rules and restrictions.
  - Unfortunately, more complex systems have higher probabilities of deadlock.
    - Becoming increasingly dangerous to ignore the possibility of a deadlock.
  - Random experiments were used to estimate likelihood of a deadlock occurring.
    - Simulation modeled a system with six processes and four resources.
    - Conclusion: Frequency of deadlock can be as high as 10 percent.
    - => OS designers need to deal with the deadlock problem for complex sys.

- Method #2: Detection and Recovery
  - Based on the designer's judgment that deadlock occurs infrequently.
    - It would cost more to prevent or avoid it than to just test for its occurrence.
    - If and when it occurs, effect a recovery procedure to correct it.
  - Allow the system to enter all possible operating states with no restrictions.
    - Allows entry into deadlock states, but prohibits permanent residence there.
  - Detection can be performed by checking resource graph for cycles.
    - An algorithm to detect circular chains can be executed at some frequency.
    - Can also just check for processes blocked for very long times.
    - Or, can let the system operator manually perform detection (much slower).
    - If a deadlock state is detected, then execute a recovery procedure.
  - Recovery can be done by:
    - Aborting all processes in the system and restarting the entire OS.
      - Simple method that always works, but not all jobs may be deadlocked.
    - Aborting all deadlocked processes involved in the cycle.
      - Must determine those processes and resources involved in deadlock.
      - Users will eventually reintroduce them, but hopefully in different order.
    - Successively aborting processes involved in the cycle until cycle cleared.
      - Detection algorithm must be run after each termination.
  - Problem:
    - Cost of detection depends upon how often detection algorithm is run.
    - Cost of recovery is the lost computation time from aborted processes.
      - Must also ensure that any modified file is returned to it's initial state.

- Method #3: Prevention by negating one of the four necessary conditions for deadlock:
  - Based upon the computer designer's judgment that deadlock is costly.
    - Therefore, it is best to expend overhead to absolutely *preclude any possibility* of deadlock arising under *any* circumstances.
    - Prohibits the existence of any unsafe states.
  - Approach: Examine each of the four necessary conditions for deadlock.
    - Try to prevent at least one of the conditions from ever happening.

- Prevention of Condition 1 (Mutual Exclusion)
  - Difficult to prevent since some resources are inherently non-sharable.
    - e.g.) Card Reader, Writeable file
  - Not practical to try to eliminate this condition.
    - Mutual exclusion limitation must remain for some resources.

- Prevention of Condition 2 (Hold and Wait)
  - Can be prevented by using Preallocation.
  - Force all processes to request all resources initially, all at once.
  - Process cannot begin execution until all resource requests are granted.
  - Advantage:
    - Easy to implement.
    - Often used (e.g., OS/360).
  - Problem:
    - Entails a substantial cost in terms of efficiency of resource utilization.
    - Each process must wait until all of its resources are available.
      - (Even if one is not needed until very late in its execution)
    - Wasteful to commit resource if likelihood it will be needed is small.
      - Resources allocated may not even be used.
      - e.g.) Printer for core dumping (May not even arise if no errors).
    - Resources used only for a short periods of time are allocated for entire job's life, and are therefore inaccessible for long periods of time.
    - Difficult to formulate and predict requirements before execution starts.

- Prevention of Condition 3 (No Preemption)
  - Require a process that is refused its request to free all of its resources.
    - That is, all the resources it has are preempted.
  - It would then need to request them again later.
  - Preemption cost of most resources would be too high and impractical.
    - (e.g. How can you take a line printer away from a job that's printing ?)
    - CPU preemption has relatively low overhead cost, So:
      - Task switching the CPU is reasonable (and done in real life).
      - But, Task switching other resources is generally not feasible.

- <u>Prevention of Condition 4 (Circular Wait)</u>
  - Cycles can be avoided by imposing an order on resource types.
  - This method is called Standard Allocation Pattern or Hierarchic Allocation.
    - Resources are assigned a level (order) in a hierarchy.
  - All resource allocation requests must be made in ascending order.
  - If a process has resource k, it can only request resources at level > k
  - Advantages:
    - Less restrictive than full preallocation.
      - Doesn't require all resources at once; just in a certain order.
    - Does not require knowledge about maximum needs in advance.
  - Problems:
    - More costly to implement than full preallocation.
    - Constraint is imposed on the natural order of resource requests.
      - Imposes a burden on the application programmers.
    - Order in which a process needs resources could differ from hierarchy.
      - e.g.) Process needs plotter early, tape drive at end of execution.
        - But tape is at lower order in hierarchy than plotter.
        - So tape is allocated early and remains idle until end.
    - Mitigated by placing commonly used resources early in ordering.
    - Hard to find an ordering that satisfies every process and everyone.
      - Especially if number of resources is large and users diverse.

- Method #4: <u>Avoidance</u> by careful resource allocation.
  - In general, Prevention is a clean solution (removes all possibility of deadlocks).
  - But, Prevention methods often result in poor resource utilization.
  - Avoidance tries to impose less stringent conditions than prevention schemes.
  - Goal is to obtain better resource utilization by running 'a little more dangerously'.
    - Does not precondition the system to remove all possibility of any deadlock.
    - Allows potential deadlock states to exist.
      - But whenever a deadlock is approached, it is carefully sidestepped.
      - Prohibits the entry into unsafe states, but does not eliminate unsafe states.
    - Ensures that deadlock, although not impossible, will not occur for the particular set of processes and requests being run at the moment.
  - Approach:
    - Utilize a method of controlled allocation.
    - Allocate a resource only if it is certain deadlock cannot occur as a result of it.
    - Relies on being able to construct a resource allocation mechanism that can predict effect of honoring individual allocation requests.
    - Requests are denied if they can lead to a situation where deadlock is possible.
      - That is, avoid moving the system into unsafe states.

All states can be partitioned into safe states and unsafe states.
- Safe state: System can allocate resources in some order to avoid deadlock
- Unsafe state: Implies that deadlock can occur as a result of the allocation.
- Note: Unsafe does not necessarily mean that system is in deadlock.
  - Nor does an unsafe state mean that deadlock is imminent.
  - Merely implies that situation is beyond control of resource allocator.
  - Implies that some unfortunate exec sequence *could* lead to deadlock.

The test of whether a state is safe or not requires a search.
- Sequences of various future process actions must be evaluated.

Requires a priori information about max resource needs for each process.
- Processes must state their "Maximum Claim".

Very conservative strategy.
- Refuses to allocate resources if there is *any* danger of deadlock arising.
- Assumes that all processes make their maximum claims at next step.
  - That is, this type of analysis is a *worst case* analysis.
- Still tends to underutilize resources (but not as bad as Prevention method).

• Banker's Algorithm

Dijkstra (1968).

Best known Safe/Unsafe determination algorithm used by the avoidance strategy.

The Analogy:
- Modeled after the lending policies often employed in banks.
- Bank has limited amount of funds (resources) to lend to borrowers (processes).
- Bank extends a line of credit (maximum amount of resources) to each borrower.
- Client can request funds incrementally (instead of all at once) up to his limit.
- Customer does not repay loan until he has requested up to his full limit.
- Bank usually gives a total of maximum credits > total resources it actually has.
- Bank does not expect all clients to request max loan amounts simultaneously.
- If, at some point, some client requests funds, can the bank lend them to him ?
  - (without incurring the risk of not having enough left to make the further loans that will permit all clients to eventually reach max limit and repay)

Banker grants requests as long as requested allocation cannot lead to deadlock.
- There must be an execution sequence that permits every process to complete.
- That is, there must be "a way out".

The algorithm used to determine if an allocation leads to a safe state:
- Assume that the loan to the requesting client is granted.
- Banker identifies the client whose current loan is closest to his limit.
- If banker can lend the difference to that client, then:
  - Assume that client accepts his limit, and repays his full amount.
  - Now repeat process for the next client who is closest to his limit.
- If <u>all</u> clients limits can be met in some sequence, state is safe: OK to allocate.

- Example: Assume 12 units of a resource are available.

State 1: Safe

Process	Used	Max
P1	1	4
P2	4	6
P3	5	12

Available: 2

State 2: Unsafe

Process	Used	Max
P1	4	5
P2	6	12
P3	1	7

Available: 1

State 1 is safe because it is still possible for all three processes to finish.
  Ten of the 12 units are currently in use (2 available).
  A possible sequence of completion would be:
    Give 2 available to P2 since it is closest to its maximum.
    P2 completes and returns 6. Now 6 available.
    Give 3 to P1 since it is closer to its max than P3. Now 3 available.
    P1 completes and returns 4. Now 7 available.
    Give 7 to P3. P3 completes and returns 12. Now 12 available.
    All jobs can complete, therefore, state is safe.
State 2 is unsafe because a deadlock situation could conceivably arise.
  Eleven of the 12 units are in use (1 available).
    Give 1 available to P1 since it is closest to its maximum.
    P1 completes and returns 5. Now 5 available.
    But now, neither P2 nor P3 can complete, even if granted the last 5.
    The completion of all user processes cannot be guaranteed.
    Therefore, the state is unsafe and deadlock could possibly occur.

- Example of a Safe to Unsafe State Transition

State 3: Safe

Process	Used	Max
P1	1	4
P2	4	6
P3	5	8

Available: 2

State 4: Unsafe

Process	Used	Max
P1	1	4
P2	4	6
P3	6	8

Available: 1

Suppose current state of system is State 3 (a safe state).
Now suppose P3 requests an additional 1 unit of resources.
Question: Should the resource allocator grant or deny P3's request ?
If the system were to grant this request, then State 4 would result.
P2 and P3 both require 2 to complete, but only 1 is available.
Completion of all processes cannot be guaranteed.
So, P3's original request for 1 additional unit should be declined (P3 must wait)

- Banker's Algorithm can be extended to multiple resources.
  - Each process must state its maximum claim for each of the resources.
    - Using currently assigned and max claim for each process, derive needed.
  - Formulate three vectors, each element corresponding to one of the resources.
    - E: The existing resources that the system is configured with.
    - P: Total resources that are currently possessed by all processes.
    - A: Available resources = E - P
  - Algorithm for determining if state is safe or unsafe:
    - Find a row whose Needed vector ≤ Available vector.
      - If no such row exists, then state is unsafe.
    - Assume process of that row requests, uses, and releases all resources it needs.
      - Mark process as terminated and add its Used vector to Available vector.
    - Repeat above until all processes terminate (safe state), or unsafe state found.

- Example: Given the following Current State in a system with four resources: R, S, T, U.

	Used				Maximum				Needed			
Process	R	S	T	U	R	S	T	U	R	S	T	U
P1	3	0	1	1	4	1	1	1	1	1	0	0
P2	0	1	0	0	0	2	1	2	0	1	1	2
P3	1	1	1	0	4	2	1	0	3	1	0	0
P4	1	1	0	1	1	1	1	1	0	0	1	0
P5	0	0	0	0	2	1	1	0	2	1	1	0

E:	6	3	4	2
P:	5	3	2	2
A:	1	0	2	0

Question: At this point, if P2 requests 1 unit of T, is it safe to do so ?
  Sequence of events to perform check would be:
  1) Grant P2 1 unit of T. Update P2's Needed & Used Vectors, and P & A Vectors

	Used				Maximum				Needed			
Process	R	S	T	U	R	S	T	U	R	S	T	U
P1	3	0	1	1	4	1	1	1	1	1	0	0
P2	0	1	1	0	0	2	1	2	0	1	0	2
P3	1	1	1	0	4	2	1	0	3	1	0	0
P4	1	1	0	1	1	1	1	1	0	0	1	0
P5	0	0	0	0	2	1	1	0	2	1	1	0

E:	6	3	4	2
P:	5	3	3	2
A:	1	0	1	0

  2) Let process P4 complete. Adding P4's Used Vector to A gives A = (2111).
  3) Let process P5 complete. Adding P5's Used Vector to A gives A = (2111).
  4) Let process P1 complete. Adding P1's Used Vector to A gives A = (5122).
  5) Let process P2 complete. Adding P2's Used Vector to A gives A = (5232).
  6) Let process P3 complete. Since all processes can complete, state is safe.

- Example: After granting P2 1 unit of T, new state is:

	Used				Maximum				Needed			
Process	R	S	T	U	R	S	T	U	R	S	T	U
P1	3	0	1	1	4	1	1	1	1	1	0	0
P2	0	1	1	0	0	2	1	2	0	1	0	2
P3	1	1	1	0	4	2	1	0	3	1	0	0
P4	1	1	0	1	1	1	1	1	0	0	1	0
P5	0	0	0	0	2	1	1	0	2	1	1	0

E:	6	3	4	2
P:	5	3	3	2
A:	1	0	1	0

Question: Now, if P5 requests the last 1 unit of T, is it safe to do so ?

Sequence of events to perform check would be:

1) Grant P5 1 unit of T. Update P5's Needed & Used Vectors, and P & A Vectors

	Used				Maximum				Needed			
Process	R	S	T	U	R	S	T	U	R	S	T	U
P1	3	0	1	1	4	1	1	1	1	1	0	0
P2	0	1	1	0	0	2	1	2	0	1	0	2
P3	1	1	1	0	4	2	1	0	3	1	0	0
P4	1	1	0	1	1	1	1	1	0	0	1	0
P5	0	0	1	0	2	1	1	0	2	1	0	0

E:	6	3	4	2
P:	5	3	4	2
A:	1	0	0	0

2) There is no process whose Needed Vector $\leq$ A = (1000).

This state is unsafe. P5's request cannot be granted.

- Banker's Algorithm Problems:

Must know maximum resource needs for all processes in advance.

But in most situations, this is not practical.

Data Structures for Multiple Resource Case vary both in size and value with time.

Number of processes is not fixed.

Users can dynamically log in and out (creating and destroying processes).

Number of resources is not fixed.

Devices can go down and become unavailable; or new equipment added.

On-line dynamic unsafe state detection must be done after every resource request.

Can be very expensive in terms of overhead.

Despite the overhead expended, resource usage can still be poor.

Algorithm is too conservative; assumes the worst possible case at each step.

Jobs may not actually use all of their maximum 'loan' amounts.

Deadlock might not have occurred even if system had entered an unsafe state.

The effect on scheduling can be substantial.

Jobs closest to their maximum resource needs are allowed to proceed first.

A stream of small-resource jobs can indefinitely postpone large-resource job.

- **Resource Trajectories**

  A planar representation of the space of possible computations.

  Statements 1 through 4 correspond to statements in P2; Statements 5 - 8 to P1.

  Vertical axis shows progress of P1; horizontal axis represents progress of P2.

  On uniprocessor, trajectory is limited to increasing horizontal and vertical directions

  Instantaneous status of a computation is represented by a point in the plane.

  e.g.) Point W shows that: P2 has started but not yet reached statement 1 and
  P1 has passed statement 6 but not yet reached 7.

  Path through plane represents execution order of statements in P1 and P2.

  e.g.) Trajectory A shows the sequence of statements 5, 6, 7, 8, 1, 2, 3, 4.
  That is, P1 is run for four statements (5 - 8), then P2 run for four (1 - 4)

  Unsafe regions for trajectory can be identified by overlapping resource needs.

  Assume: P1 requests R1 at 6 and releases it at 7.
  P2 requests R1 at 1 and releases it at 3.

  Shaded rectangle bounded by 6, 7, 1, 3 cannot be crossed in trajectory.

  P1 & P2 cannot both have R1 at same time due to mutual exclusion on R1.

  Also assume: P1 requests R2 at 5 and releases it at 8.
  P2 requests R2 at 2 and releases it at 4.

  Therefore, region bounded by 5, 8, 2, 4 also cannot be crossed.

  Trajectory B shows a safe execution order of 1, 2, 5, 3, 4, 6, 7, 8.

  When P1 requests R2 at 5 (point X), it is unavailable and P1 must wait for it.

  P1 cannot cross 5 until it acquires and can use R2.

  P2 continues to execute past statement 4, releases R2 allowing P1 to continue.

  Trajectory C shows an unsafe execution path for statement order 5, 1, 6, 2.

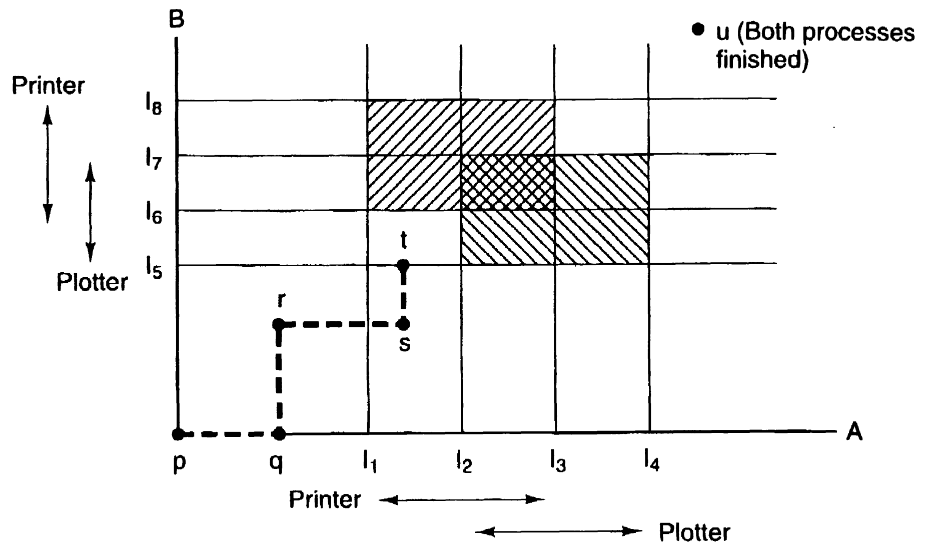
  P1 becomes blocked at 6 (point Y) since it must wait for R1 to become avail.

  P2 becomes blocked at 2 (point Z) since it must wait for R2 (held by P1 after 5).

  Both processes are now deadlocked:

  P1 is waiting for P2 to reach statement 3 and P2 waiting for P1 to finish 8.

  Deadlock becomes inevitable once computation enters rectangle U, an unsafe state.



- We have seen how the OS is involved in Memory, CPU, and Resource Management.
  - Now we focus on Processes.
  - A single job can create multiple processes (e.g., as in the Lab project using FORK).
  - In a multiprogrammed system, there are several processes "active" at once.
    - Only one process can be executing at any instant in time given a uniprocessor.
    - Each may be in a different state of execution as it swaps in and out.
    - Generally, programmers have no control over when processes are swapped.
      - Each process could potentially get interrupted after any instruction.
      - Under control of CPU Scheduler, Resource Manager, Memory Manager.
  - Processes need to communicate with each other: Interprocess Communication (IPC).

- Race Conditions.
  - A potential IPC problem when two or more processes interact via shared data.
  - Final outcome depends on the exact instruction sequence of the processes.
    - Depends on which process "wins" the "race".
  - Extremely difficult to debug because of its transient nature; Lurking bugs possible.

- Example: When process wants to print a file, it enters the filename onto the spooler list.
    - Process P is printer. It checks spooler list periodically and prints the jobs.
    - Spooler list has consecutively numbered slots; each can hold a filename.
    - There are global shared variables associated with the spooler list.
      - LIST: Large linear list where LIST(i) holds a filename in the i^{th} slot.
      - IN: Points to the next free slot where a filename can be put in spooler list.
      - OUT: Points to the next file (slot) in the spooler list to be printed out.
    - Assume that two processes, A and B, want to print at the same time.
    - Each process must perform the following instructions:

```
next_free_slot = IN                   /* read value of IN
LIST (next_free_slot) = <filename>    /* write filename
IN = next_free_slot + 1               /* update value of IN
```

    - Global Shared variables are shown in capital letters.
    - Local variables (one set exists for each process) are shown in lower case.
    - Assume LIST (4, 5, and 6) already contain filenames queued for printing.
    - So IN = 7; OUT = 4 and execution sequence as follows (time flows down):

```
A: next_free_slot_A = 7                                        /* A reads IN of 7
                              B: next_free_slot_B = 7          /* B reads IN of 7
                              B: LIST(7) = <filename_B>        /* B writes filename
                              B: IN = 8                        /* B updates IN
A: LIST(7) = <filename_A>                                      /* A writes filename
A: IN = 8                                                      /* A updates IN
```

  - Problem: Process A writes over Process B's filename in LIST(7) because
    - Process B began using a shared variable before Process A was done with it.

- **Critical Sections.**
  - First proposed by Dijkstra in 1965.
  - A Critical Section of a program is where global shared memory is being accessed.
  - Being inside a critical section is a special status accorded to a process.
    - Process has exclusive access to shared modifiable data while in critical region.
    - All other processes needing access to that shared data are kept waiting.
    - Therefore, critical sections must:
      - Execute as quickly as possible.
      - Be coded carefully to reduce any possibility of errors (e.g., infinite loops).
      - Ensure termination housekeeping if process in critical region aborts.
        - Must release mutual exclusion so other processes can enter region.
  - The majority of the time, a process performs only non-critical work on local vars.
    - Therefore critical sections of a program can be made very small.
  - Goal is to enable interacting asynchronous concurrent processes to yield correct results independent of their execution speeds and when they are swapped.
  - To avoid race conditions, mutual exclusion must be enforced within critical sections.
    - Prohibits more than one process from accessing shared memory at same time.
  - If no two processes enter their critical sections at same time, no race conditions.
  - For correct and efficient operation using shared data, a solution must ensure that:
    1) No two processes can be inside their critical sections at same time.
    2) No assumptions needed about relative process speeds or number of CPUs.
    3) No process stopped outside its critical section can block other processes.
    4) No process can be indefinitely postponed from entering its critical section.

- **Mutual Exclusion.**
  - Designing Primitive Operations for achieving mutual exclusion is a major OS issue.
  - Only needed when processes access shared modifiable data (in critical region).
    - Concurrent execution OK if two processes do not conflict with one another.
  - Several methods can be used to ensure that only one process is in a critical section.
  - We look at several as an introduction to Semaphores.

- **Attempt #1: Disabling Interrupts**
  - Simplest solution if we assume CPU can only task switch a process via an interrupt.
  - Process disables all interrupts just after entering its critical region.
  - It can then read/write shared variables without any other process intervening.
  - Process re-enables them just before leaving its critical region.
  - In theory, this approach should guarantee mutual exclusion.
  - In fact, system kernel often makes use of this technique for small pieces of code.
  - However, it is unwise to give a user process the same power.
    - User program could "forget" to re-enable interrupts and become "unstoppable".
  - Therefore, this is not an appropriate, practical solution.

- Attempt #2: Lock Variables
  - Utilize a single, global, shared LOCK (check-out) variable initialized to 0.
  - LOCK is set to 1 if any process is in (has checked-out) a critical section.
  - When process needs to enter its critical section, it first tests LOCK.
    - If LOCK = 0, process sets LOCK = 1 and enters the critical section.
      - Process must reset LOCK to 0 upon exiting critical section.
    - If LOCK = 1, process must wait until the LOCK is reset to 0 by another process.
  - This approach has the same problem as that in the print spooler list example.
    - Assume Process A reads LOCK = 0 but is interrupted before it can set LOCK = 1
    - Process B also reads LOCK = 0, sets LOCK to 1, and enters critical section.
    - Process A resumes, also sets LOCK = 1, and enters critical section too.
  - Again, processes and programmers cannot control the exact instruction sequence.
    - Cannot assume anything about relative speeds of async. concurrent processes.
    - Must examine all possible interleaved execution sequences (worst case ones).
  - Recursive problem: No way to ensure mutual exclusion on LOCK variable itself.

- Attempt #3: Strict Alternation
  - Two processes alternate in entering the critical section by taking turns.
  - Define a global shared variable TURN.
  - If TURN = 0, Process A can enter critical section; If TURN = 1, Process B can enter.
  - Process may have to "busy wait" for its turn to enter critical section.
    - Busy Wait:
      - Process continuously tests a variable in a tight loop for some value.
      - Should be avoided since it wastes CPU time doing unproductive work.
      - Can also tie up a resource needed by the process that is being waited for.
  - [See Tanenbaum, Figure 2-8]
  - Operation:
    - TURN initialized to 0.
    - Process A reads TURN = 0 and enters critical section.
    - Even if process B intervenes, it also reads TURN = 0 and must busy wait.
    - When Process A exits critical section, it sets TURN = 1, letting B enter.
  - Completely eliminates all race conditions, but has a problem.
  - Problem:
    - Suppose Process B reads TURN = 1, enters, and then exits critical section.
    - Now, TURN = 0 while both Process A and Process B are in non-critical sections.
    - Process B gets swapped out, and Process A has opportunity to run.
    - Process A reads TURN = 0, enters and exits critical section, and sets TURN = 1.
    - Process B still swapped out, so Process A finishes non-critical section.
    - Process A returns to top of loop and must now wait since TURN = 1.
      - That is, Process A cannot enter critical section twice in a row.
      - Strict alternation between Process A and B must be obeyed.

This approach Violates condition 3:

- No process stopped outside a critical section should block other processes
- Process B is too slow to keep up with Process A.

This is a poor solution if relative speeds of processes is significantly different.

If this solution were used for the print spooler example earlier, then:

Process A would not be allowed to print another file even if printer is ready and Process B was doing something completely unrelated to printing.

Not a practical solution for achieving mutual exclusion.

• Attempt #4 (This one works): Peterson's Solution

A combination of lock variables and alternation.

An improvement over Dekker's Algorithm, the first S/W-only solution for mutual excl.

Utilizes two shared variables:

- TURN: Set to the pid of the process whose turn it is to enter critical region.
  - Note: Although 'turn' is used, processes do not have to alternate turns.
- INTERESTED[i]: Set to .true. if process i is interested in entering critical region.

Can be organized as two shared subroutines along with shared variables above:

- Enter_Region: Called by each process with its pid as a parameter before entry.
  - Sets INTERESTED[pid] = .true.
  - Sets TURN = pid
  - Will cause the calling process to wait if necessary before entering CR
- Leave_Region: Called by the process after exiting critical region.
  - Sets INTERESTED[pid] = .false.

[See Tanenbaum Figure 2-9]

Note that the variables 'process' and 'other' are local variables.

INTERESTED[0] and INTERESTED[1] both initialized to .false.

Operation:

- Process A (assigned the PID = 0) calls Enter_Region
  - Indicates its interest in entering by setting INTERESTED[0] = .true.
  - Process A sets TURN = 0
  - Assume at this point that Process A gets swapped out.
- Now, suppose Process B (assigned the PID = 1) also calls Enter_Region
  - Process B sets INTERESTED[1] = .true.
  - Process B sets TURN = 1
  - Now assume worst case, and that Process B gets swapped out and A runs.
- Process A executes While zero times since TURN = 1 and process = PID A = 0.
  - So Process A enters critical region.
- Even if Process B resumes execution, it executes While and must wait since TURN = 1 and process = PID B =1; INTERESTED[0] = .true.

Key Point: It doesn't matter which process "wins" the "race" and sets TURN = its pid, because TURN is actually part of the test along with INTERESTED[other].

- Attempt #5 (This one works even better than #4): Semaphores (Dijkstra, 1965)
  - The analogy:
    - A semaphore can act like a gate into a restricted area.
    - Status of the gate is either open (raised) or closed (lowered).
    - When a process needs access, it executes a P operation.
      - If gate is open, process enters and closes gate behind it.
      - Otherwise, if gate is closed, process must wait until gate is reopened.
    - Once process has entered, used shared var, and exits critical section, it calls V
      - V opens the gate and allows (signals) one waiting process to enter.
      - If no processes waiting, gate is left open for next process needing access.
  - History: Dutch words for 'wait' and 'signal' have initial letters P and V respectively.
  - Software Implementation:
    - Let S be a semaphore.
    - A semaphore is simply an integer value.
    - Its value can only be altered via operations P(S) and V(S).
  - An important aspect of semaphores is that P and V are executed atomically.
    - That is, P and V are made indivisible, generally with OS and H/W support.
    - Executed by the OS kernel with all interrupts disabled.
    - Indivisible: Once started, they will be completed without interruption.
    - This prevents two processes from executing P or V operations at the same time.
    - No other process can access the semaphore until the operation is completed.
  - Also, inherent and built-into P and V are OS-level wait and signal capabilities.
    - Processes are put to sleep (swapped out of CPU) while waiting for a signal.
    - This eliminates the busy wait problem required with previous approaches.
  - *[Note: Algorithm for P(S) presented here is a generalization of that in Tanenbaum]*

  - P(S) acquires permission to enter a critical region.
    - Alternate names for P include Wait and Down.
  - When a process executes P(S):
    - S is *decremented* by 1, and
    - IF S *.LT.* 0, then the calling process is stopped and put on a waiting queue.
      - It remains blocked until a V(S) operation by another process releases it.
    - ELSE calling process continues

  - V(S) records the termination of a critical region.
    - V(S) must reactivate one of the waiting processes (if any).
    - Alternate names for V include Signal and Up.
  - When a process executes V(S):
    - S is *incremented* by 1, and
    - IF S *.LE.* 0, then one process from the waiting queue is removed from the queue
      - The waiting process from the queue is allowed to resume execution.
    - The calling process which invoked V(S) can also continue execution.

- Semaphores can be used in several different ways.
  - Different applications require different initial values for and numbers of Semaphores
  - A semaphore which has a maximum value of one is called a binary semaphore.
    - Binary Semaphore often referred to as MUTEX (MUTual EXclusion).
  - Two processes can implement mutual exclusion by using a binary semaphore.
    - Critical sections are bracketed by P(S) and V(S).
    - P(S) is the entry or opening bracket; V(S) is the exit or closing bracket.
  - For two processes with a binary semaphore:
    - If S = 1, then neither process is executing its critical section.
    - If S = 0, then one process is executing its critical section.
    - If S = -1, then one process is in its critical section and other is waiting to enter.

- Using Semaphores to Wait for something to be done.

Process A will wait until Process B reaches a certain point.

SEM is initialized to 0

```
Process A                          Process B
    .                                  .
Start(Process B)                   <step that A is waiting for>
P(SEM)                             V(SEM)
    .                                  .
    .                                  .
```

- Using Semaphores for Rendezvous

No matter which process runs faster, it will wait for the other to re-synchronize.

SEM1 and SEM2 are both initialized to 0

```
Process A                          Process B
    .                                  .
V(SEM1)                            V(SEM2)
P(SEM2)                            P(SEM1)
    .                                  .
    .                                  .
```

- Using Semaphores for Mutual Exclusion

```
    Process A                          Process B
Loop_A:                            Loop_B:
        <non-critical>                     <non-critical>
    P(SEM)                             P(SEM)
        <critical section>                 <critical section>
    V(SEM)                             V(SEM)
        <non-critical>                     <non-critical>
    GOTO Loop_A                        GOTO Loop_B
```

- **Example: Two Process Mutual Exclusion with SEM initialized to 1.**

Time	Process	Action	In C.R.	Waiting	SEM
					1
1	A	P(SEM)			
			A		0
2	B	P(SEM)			
				B	-1
3	A	V(SEM)			
			B		0
4	B	V(SEM)			
					1

- **Example: Four Process Mutual Exclusion with SEM initialized to 1.**

Time	Process	Action	In C.R.	Waiting	SEM
					1
1	A	P(SEM)			
			A		0
2	A	V(SEM)			
					1
3	B	P(SEM)			
			B		0
4	C	P(SEM)			
				C	-1
5	D	P(SEM)			
				C, D	-2
6	B	V(SEM)			
			C	D	-1
7	C	V(SEM)			
			D		0
8	D	V(SEM)			
					1

- **Using Semaphores for a Producer-Consumer System**
  - **Producer-Consumer (Bounded-Buffer) Problem:**
    - **A "Classic" Process Coordination Problem.**
    - **Has two processes that communicate via a common buffer.**
      - **Producer generates and places items in a shared buffer.**
      - **Consumer takes items off of the shared buffer and uses them.**
  - **This problem is not just one of mutual exclusion, it is also that of synchronization.**
    - **The producer and consumer proceed asynchronously and at different speeds.**
    - **If Producer runs faster than the Consumer, the buffer may fill up.**
      - **Producer must wait if buffer is full until the Consumer empties it.**
    - **If Consumer runs faster than the Producer, the buffer may be empty.**
      - **Consumer must wait if buffer is empty until the Producer fills it.**
    - **Synchronization needed to enable one process to wait until the other signals it.**

Three semaphores are needed to the solve producer-consumer problem:

1) EMPTY will count the number of buffer slots not yet filled (i.e., empty)
   EMPTY is initialized to N, the total number of slots in buffer, 2 in ex. below.
   Producer must wait at P(EMPTY) if there are no more empty slots.
   Consumer increments EMPTY via V(EMPTY) after consuming a slot.
2) FULL will count the number of buffer slots that have been filled.
   FULL is initialized to 0 since entire buffer is initially empty.
   Consumer must wait at P(FULL) if there are no more filled buffer slots.
   Producer increments FULL via V(FULL) after producing (filling) a slot.
3) MUTEX guarantees that only one process is operating on buffer pointers.
   MUTEX is initialized to 1 and guards *<Fill Buffer>* and *<Consume Buffer>*.

```
Producer:                                Consumer:
          <non-critical>                           <non-critical>
     P(EMPTY)                                 P(FULL)
          P(MUTEX)                                 P(MUTEX)
          <Fill Buffer>                            <Consume Buffer>
          V(MUTEX)                                 V(MUTEX)
     V(FULL)                                  V(EMPTY)
          <non-critical>                           <non-critical>
     GOTO Producer                            GOTO Consumer
```

Time	*Producer*	*Consumer*	*Empty*	*Full*
			2	0
1		P(FULL) Blocked (Buffer empty)		
			2	-1
2	P(EMPTY)			
			1	-1
3	V(FULL) Unblocks Consumer			
			1	0
4	P(EMPTY)			
			0	0
5		V(EMPTY)		
			1	0
6	V(FULL)			
			1	1
7	P(EMPTY)			
			0	1
8	V(FULL)			
			0	2
9	P(EMPTY) Blocked (Buffer full)			
			-1	2
10		P(FULL)		
			-1	1
11		V(EMPTY) Unblocks Producer		
			0	1

- Using Semaphores for The Readers and Writers Problem

  Another "Classic" IPC Problem which models access to a shared Data Base.

  Two types of processes, readers and writers, compete for access to the Data Base.

  Multiple readers can read the Data Base concurrently.

  Only a single writer can write to the Data Base at a time.

  Also, if a writer is writing to the Data Base, no readers can be allowed access to it.

  The first and last reader to enter the critical region are significant.

  RC counts the number of readers to allow us to identify the first & last reader

  Solution below uses two semaphores: [See Figure 2-19 in Tanenbaum]

  1) DB is initialized to 1 and guarantees that access for writing is exclusive.
     Used by the writer before and after writing to protect critical write section.
     Also used by the first reader to determine if a writer is currently writing.
     If so, it and all other readers behind it wait; otherwise reader enters.
     Last reader to exit from critical section must release writer from waiting.

  2) MUTEX, initialized to 1, guarantees that the shared variable RC is protected.
     Only one reader at a time can update the reader_counter variable, RC.

<u>Reader</u>:

```
    P(MUTEX)                     /* Only one reader can update RC at a time
        RC = RC + 1              /* Increment number of readers upon reader entry
        IF (RC = 1) THEN P(DB)   /* 1st reader ensures mutual exclusion with writers
    V(MUTEX)                     /* End critical section on RC update

        <read_data_base>         /* Critical only to writers; OK to have many readers

    P(MUTEX)                     /* Only one reader can update RC at a time
        RC = RC - 1              /* Decrement number of readers upon reader exit
        IF (RC = 0) THEN V(DB)   /* Last reader must release any waiting writer
    V (MUTEX)                    /* End critical section on RC update

    GOTO Reader
```

<u>Writer</u>:

```
    P(DB)                        /* Only one writer at a time and only if no readers
        <write_data_base>        /* Critical Section
    V(DB)                        /* End critical section on write_data_base.

    GOTO Writer
```

Note: Solution above implicitly gives readers priority over writers.

  Only the first reader has to compete with any writers to gain access.

  Other readers will pass directly into CR provided a reader is still in the CR

  No writers are allowed to begin writing if there are any readers reading.

  So, writers can be indefinitely postponed given an infinite supply of readers.

- The most visible part of the Operating System as seen by the user is the file system.
  - Convenience and usability of an OS is determined to a large degree by the file sys.
  - Issues include:
    - File and Directory System Interface (From a user viewpoint).
      - What constitutes a file, how it is named, protected, accessed, etc.
    - File and Directory System Structure (From a designer viewpoint).
      - Organization in memory, efficiency.
    - Reliability and Security (From both user and designer viewpoint).
      - Maintaining File and Directory System Consistency.
      - Protection mechanisms.

- Files
  - Three common file organizations:
    - 1) Simple byte sequence.
      - e.g.) UNIX files.
    - 2) Sequence of fixed-size records.
      - Allows arbitrary records to be read or written.
      - Cannot insert or delete records in the middle of a file.
      - e.g.) CP/M files.
    - 3) Tree of disk blocks.
      - Also called ISAM (Indexed Sequential Access Method).
      - Each block holds n keyed records.
      - Records can be searched by key and can be inserted anywhere in tree.
      - If block is full, split block to insert an additional record.
      - e.g.) Used on many mainframes.
  - An important OS goal is to achieve device-independence.
    - Make access the same independent of where the logical file physically resides.
      - e.g.) Uniform I/O call to Floppy file, Hard Disk file, Terminal, or Line Printer

- Directories
  - A file which helps organize and keep track of other files.
  - Typically contains one entry per file.
  - Some form of hierarchy (i.e., a tree of directories) is generally used to:
    - Give each user as many (sub)directories as he needs.
    - Provide a means to group related files together.
  - Two methods can be used to identify a specific file within the tree directory:
    - 1) Absolute Path Name.
      - A complete path starting at the root directory all the way down to the file.
    - 2) Relative Path Name.
      - Path name is relative to the user's current working directory.
      - e.g.) UNIX paths beginning with "/" are absolute; all others are relative.

- **File system design**
  - Users are interested with how files are named, how directories are organized, etc.
  - Designers are concerned with efficiency and reliability of disk space management.
  - Tradeoffs are similar to those discussed earlier with Memory Management.
  - Issue #1) Two strategies possible for storing an n byte file:
    1) Use a contiguous sequence of n bytes of disk space.
        - If the file grows (which can happen quite often), it will have to be moved.
        - Moving a file from one disk location to another is time consuming.
    2) Split the file up into a number of fixed-size blocks.
        - Allows the blocks to be stored in non-contiguous locations.

    => Choose to use approach #2: Split file into fixed-size, non adjacent blocks.
  - Issue #2) Block size is an important parameter.
    - Median file size on a UNIX system is 1K (Mullender and Tanenbaum, 1984).
    - Large block sizes significantly greater than 1K will waste space,
      - but can map very easily to a physical disk parameter (e.g., 32K cylinder).
    - Small block sizes will waste less space,
      - but will require several disk seek and rotation wait times.
    - Classic space-time tradeoff.
      - These design parameters are inherently in conflict.
      - *[See Tanenbaum figure 5.15]*
    - => Typical choice is 512 bytes, 1K, or 2K.
  - Issue #3) Two methods possible for keeping track of free blocks:
    1) Linked list of disk blocks, each holding as many free disk blocks as possible.
        - e.g.) Using 1K blocks and 16-bit disk block addresses, each block on the free block list can hold 511 free block addresses (512th is pointer).
    2) Bit map.
        - Uses one bit per block to identify it as being either free or not.
        - Requires less space (1 bit/block) vs. linked list (16 bits/block).

    [See Tanenbaum figure 5.16]

- **File Structure**
  - Since a file consists of sequence of blocks, need some way to keep track of blocks.
  - Storing blocks consecutively does not enable files to grow.
  - Store blocks of a file as a linked list; maintain pointers in RAM memory for speed.
  - Associate with each disk, a table called the File Allocation Table (FAT).
  - FAT contains one entry for each disk block.
  - The directory entry for each file gives the block number of the first block of the file.
  - That slot in the FAT contains the block number of the next block.
  - End of file is marked by EOF entry in the FAT.
  - Free blocks are identified by FREE entry in the FAT.
  - This linked list allocation scheme is used by MS-DOS.

Problem:

With large disks, FAT would require significant MM space.

e.g.) For a 64Meg disk using 1K sized blocks, FAT is 128K.

Pointers for all files on the entire disk are scattered throughout the FAT.

The entire FAT is potentially needed, even if only one file is open.

UNIX solution:

Keep the block lists for different files in different places.

Associate with each file, a small table on disk called the i-node.

[See Tanenbaum figure 5.10]

The i-node stores 10 disk block numbers and 3 indirect block numbers.

Indirect block pointers are single, double, or triple indirect.

For files up to 10 blocks long, all disk addresses are available in the i-node.

When a file grows beyond 10 disk blocks:

Acquire a free disk block and set the single indirect block pointer to it.

If disk block size is 1K and 32-bit addresses are used,

Then the single indirect block can hold 256 disk addresses.

Single indirect scheme is sufficient for files up to 266 blocks.

10 are stored directly in the i-node; 256 in the single indirect block.

Above 266 blocks, the double indirect pointer is used to point to a disk block.

As before, the disk block contains 256 pointers; but not to data blocks.

The 256 pointers are used to point to 256 single indirect blocks.

Each indirect block then points to the data block as before.

Double indirect scheme sufficient for files up to 266 + 256**2 = 65,802 blocks.

Files longer than 65,802 blocks require triple indirect pointer scheme.

File size limitation is about 16 gigabytes.

Advantages of UNIX scheme:

Access to small files is fast since it uses only direct blocks.

Yet can handle very large files too by using higher degrees of indirectness.

At most, four disk references are needed to locate any location of a file.

One to get i-node, and up to three indirect lookups.

- **Directory Structure**

Main purpose of directory system is to map pathnames onto i-nodes (or equivalent).

Generally organized as a hierarchy and conceptually very simple to implement.

- UNIX directories are just files.

Each entry of file contains filename and its i-node number.

Each directory can contain an arbitrary number of filename, i-node pairs.

Absolute path starts from root directory and follows pointers to file.

Relative path starts from current working directory.

Entry ".." has the i-node number for the parent directory.

- **File System Consistency**
  - A reliability issue, especially after a system crash when files might be left open.
    - File and directory structures are only updated at certain times.
    - System crash can leave file/directory system in an inconsistent state.
  - Most computers have a utility program that checks for file system consistency.
    - Automatically engaged as part of Mac OS upon reboot after a crash.
  - Two consistency checks are made: blocks and files.
  - Block consistency check: *[See Tanenabum figure 5.18]*
    - Build a table with two counters per block, each initialized to 0.
      - #1 counter keeps track of how many times the block is present in a file.
      - #2 counter shows how often it is present in the free list (or the bit map).
    - Program then reads all the i-nodes.
      - Starting from an i-node, can get a list of all blocks contained in a file.
      - #1 Counter for that block is incremented every time it occurs in a file.
    - Program then examines the free list or bit map of free blocks.
      - Each time block shows up in free list, increment its #2 counter.
    - If each block is accounted for (counter #1 + #2 = 1), file system is consistent.
    - If a block has a 0 in both counter #1 and #2, it is a missing block.
      - Not serious, but wastes space and reduces capacity of the disk.
      - To reclaim missing block, just add it to the free list.
    - If a block has a count > 1 in the #2 counter, it is redundant in the free list.
      - Duplicates can occur only in a list implementation; not in a bit map.
      - Solution is to rebuild the free list.
    - If a block has a count > 1 in the #1 counter, it probably means data is corrupted.
      - That is, the same block is present in two or more files (not typically so).
      - If any of these files are removed, that block will be put on the free list.
        - This will lead to same block being used and being free at same time.
      - If all files are removed, the block will be put onto the free list many times.
        - Same error as in block's count > 1 in the #2 counter.
      - Best recovery solution is to copy the contents of block into a free block.
        - Insert the copy into one of the files to separate their block overlap.
        - Also, inform the user of serious problem: file(s) are probably garbled.
  - File consistency check:
    - Similar to checking that each block is accounted for.
    - Now, perform check for files to ensure that directory information is correct.
  - Heuristic consistency checks can also be performed.
    - Gross checks to detect garbled directories.
    - Often used by third-party disk verification programs (e.g. Norton Utilities).
    - Examples: Unusually large directories or files.
      - Access modes that disallow user access but give access to others.

- **Security:**
  - File systems contain valuable information.
  - Technology trends will accentuate computer privacy and security needs.
    - e.g.) Electronic fund transfers; Medical X-ray transmission over internet, etc.
  - The OS must protect files and other objects (e.g., CPU, MM) in the computer system.
  - Two important facets of security:
    - 1) Data Loss.
      - Caused by Natural Disasters, H/W or S/W failures, Human errors.
      - Solved using geographically remote backups.
    - 2) Intruders.
      - Passive Intruders want to read files they are not authorized to read.
      - Active Intruders are more malicious; they want to make changes to data.
      - Different "enemy" will put forth various levels of effort to penetrate system.
        - Casual prying by nontechnical users motivated by curiosity.
        - Technically skilled "hackers" motivated by the challenge.
        - Criminals (e.g., bank programmer-thief) motivated by money.
        - Military spies motivated and funded by foreign countries.

- Infamous Historical Security Flaws
  - -UNIX *lpr* utility prints a file and has an option to remove the file after it has printed.
    - Early versions of UNIX allowed user to print password file, then remove it.
  - -Trojan Horse Attack.
    - Modify a program to do illegal things in addition to its usual functions.
    - Example: Modify the editor to save a user's file to the thief's account as well.
  - -TENEX Flaw implemented on DEC-10 computers using paging.
    - TENEX allowed users to monitor their program's paging behavior.
      - System could call a user function whenever a page fault occurred.
    - OS checked passwords one character at a time.
      - Stopped checking as soon as it saw that the password was wrong.
    - Intruder positions guessed password with first character at the end of one page.
      - Rest of password would be positioned at the start of the next page.
    - Second page could be forced out of memory by intruder.
      - Reference many other pages to guarantee second page replaced by MMU.
    - Now, intruder tries his guessed password.
      - If first character of password is wrong, system reports "Bad Password".
        - System stops and never references second page.
      - If first character is right, system reads second character on second page.
        - This causes a page fault which can be monitored by intruder.
    - Intruder repeats process character by character for the password.
    - Assuming 128 ASCII possibilities for each position of an n length password,
      - Intruder could reduce 128^n possible permutations to only 128n guesses.

- **Other Generic Security Attacks**

  The average OS "still leaks like a sieve".

  "Tiger" or "Penetration teams" have identified several likely weak links in systems:

  - Request MM pages, disk or tape records and just read them.
    Many systems do not erase previous user's contents before allocating them.
  - Begin logging in and press BREAK, DEL, or CNTRL-C to interrupt process.
    Some systems will abort password checking routine and give successful login.
  - Write a program that types "login:" on the screen and records the next entries.
    Unknowing user will type in login name and password for program to record.

  Unfortunately, Penetration teams prove the presence, not absence, of loopholes.

- **Design Principles for Security**

  Saltzer and Schroeder (1975) list general principles for designing secure systems.

  - System design should be public.
    Open design is seen by more eyes and exposes faults easier.
  - Default mode should be no access.
    Errors in which legitimate access is refused will be reported rather quickly.
    Errors in which unauthorized access is allowed generally will not get reported.
  - Dynamically check for current authority.
    Many systems check for permission only when file is opened, not on each read.
    File could be kept open for full access even if privileges were recently revoked.
  - Give each process the least privilege possible.
    Limits damage possible by virus or Trojan horse.
    e.g.) Give editor authority to access only the file being edited.
  - Protection scheme must be user-friendly.
    Otherwise, users will circumvent protection scheme to avoid inconvenience.

- **User Authentication**

  User authentication is the process of identifying the user with high confidence.

  Many protection schemes are based on knowing the identity of the user.
  Different users are given different access rights.

  Passwords are the most widely used form of authentication.
  Easy to defeat since users often pick passwords that can be easily guessed.
  86% of all passwords are first names, last names, birthdates, street address, etc

  Variation on password scheme is the challenge-response method.

  - Ask user various background information and check against prestored answer.
    e.g.) On what street was your elementary school ?
  - Ask user to provide an algorithm-based response that changes with query.
    e.g.) (If algorithm is n**2) System generates random 7; User responds 49.

  Physical Identification schemes are now becoming technically feasible.
  e.g.) Fingerprint, voiceprint, retina-print, dynamic signature analysis.

- **Protection Mechanisms**

  Define a domain as a set consisting of the (object, operation rights) pair.

    Each pair specifies an object and some valid operations that can be done on it.

  At every instant in time, each process executes in some protection domain.

    There is some collection of objects a process can access.

    Each object has associated with it some set of rights.

  Processes can move from domain to domain. *[See Tanenbaum figure 5.23-24]*

  Use a protection matrix to record object rights by domain.

    Rows are domains; Columns are objects.

    Contents of intersected cell shows rights that the domain has upon the object.

  Problem: Inefficient to store entire protection matrix since it is large and sparse.

    Most domains have no access at all to most objects (blank intersection cell).

- **Access Control Lists (ACL)**

  Store only the nonempty cells of the Protection Matrix organized by <u>columns</u>.

  Associate with each object a list showing all the domains that can access it and how

  - Example: MULTICS

  Assume three users with (UID, GID) = (Jan, System) (Els, Staff) (Jelle, Student).

  Let each ACL entry denote Read, Write, eXecute privileges.

  Wildcard '*' in an ID field means all UIDs or GIDs.

  Given the following files with their respective ACLs:

      File0: (Jan, *, RWX)
      File2: (Jan, *, RW-), (Els, Staff, R--)
      File4: (Jelle, *, ---), (*, Student, R--)

  File0 can be read, written, or executed by any process with UID = Jan and any GID.

  File2 can be read or written by Jan; it can also be read by UID=Els, GID=Staff.

  File4 can be read by any GID=Student *except for* the UID=Jelle from any GID.

    ACLs can block out specific UIDs or GIDs while allowing everyone else access.

  - Example: UNIX

  The domain of a UNIX process is defined by its UID and GID.

  Given any unique (UID, GID) pair for a process, one can specify:

    1) A complete list of all objects that can be accessed by the process.

    2) The level of access allowed (Reading, Writing, Executing).

  Additionally, each UNIX process has two domain halves: user part + kernel part.

    Normally, process runs in domain specified by user part.

    When process makes system call, it switches to domain specified by kernel.

      Kernel part generally has more access to more objects than user part.

  Three bits (RWX) are provided per file for owner, owner's group, and others.

    Essentially a list associated with an object showing who can access it and how.

    Simpler and cheaper implementation of ACL compressed to 9 bits.

- Capability List
  - Store only the nonempty cells of the Protection Matrix organized by rows.
  - Associate with each process a list of objects that may be accessed by it and how.
  - Individual items on the capability list are called capabilities.
  - *[See Tanenbaum figure 5.25]*
  - Each Capability has:
    - A Type field showing what kind of an object it is.
    - A bit mapped Rights field showing operations permitted on the object.
    - An Object field which is a pointer to the object itself (e.g., file's i-node number).
  - To do an operation on an object, the process must provide a pointer to its capability.
    - Simple possession of the capability means that access is allowed.
  - Capabilities are also objects in the system and must be protected.
    - Can associate with each object a tag to identify it as a capability.
      - If tag is set, do not allow the user to modify it.

- Contrasting ACL vs. Capability List
  - Access control list is analogous to a "guest list".
    - Guard needs to see the person's I.D.
    - He checks the name against the guest list (requires some work).
    - Revocation is easy.
      - Just remove the person's name from the guest list.
  - Capability list is analogous to a "ticket".
    - Guard just takes a person's ticket assuming it gives him the right to enter.
    - Easier for the guard than checking a guest list.
    - Revocation is harder.
      - Once ticket is issued to a person, hard to get it back.

- Security Summary
  - Good protection mechanisms are difficult to design.
    - A logically complete solution is not necessarily an adequate one.
    - Overhead for protection can be substantial, especially if performed dynamically.
    - The solution must also be cost effective.
  - Problem of security must be viewed with the complete system in mind.
    - Breach in security is likely to happen in the weakest link of the protection chain.
    - System is only secure as its weakest point.
    - Often, the weak link is not the most complex or technically advanced part of sys.
  - Human users represent a significant vulnerability point.
    - Must control the amount of access / knowledge possessed by any one user.
      - e.g.) Give System Operator privilege to only change (not read) passwords.
    - Common method used in banks is authorization in pairs.
      - Two people are required to perform a sensitive transaction.