"I think there is a world market for about five computers."

Thomas J.Watson (1945)

Windows 95 /Win' dz/: n., 32-bit extensions and a graphical shell for a 16-bit patch to an 8-bit operating system originally coded for a 4-bit microprocessor, written by a 2-bit company, that can't stand 1 bit of competition.

What is an OS?

- Tool to make programmer's job easy
- Resource allocator
  - Must be fair; not partial to any process, specially for process in the same class
  - Must discriminate between different class of jobs with different service requirements
  - Do the above efficiently
    - Within the constraints of fairness and eficiency, an OS should attempt to maximize throughput, minimize response time, and accomodate as many users as possible
- Control program
- Tool to facilitate efficient operation of computer system
- Virtual machine that is easier to understand and program
- Layered architecture

<table>
<tr><td>Banking system</td><td>Airline reservation</td><td>Adventure games</td></tr>
<tr><td>Compilers</td><td>Editors</td><td>Command Interpreter</td></tr>
<tr><td colspan="3">Operating System</td></tr>
<tr><td colspan="3">Machine Language</td></tr>
<tr><td colspan="3">Microprogramming</td></tr>
<tr><td colspan="3">Physical Devices</td></tr>
</table>

- UNIX structure
  - Application environment – shell, mail, text processing package, SCCS
  - Operating system – support programs for applications

Early Systems

- 1945 – 1955
- Bare machines – vacuum tubes and plugboards
- No operating system
- Black box concept – human operators
- No protection
- ENIAC – Electronic Numerical Integrator And Computer

Second Generation Systems

- 1956 – 1965

- Transistors and batch systems
- Clear distinction between designers, builders, operators, programmers, and maintenance personnel
- I/O channel
- Read ahead / spooling
- Interrupts / exceptions
- Minimal protection
- Libraries / JCL

Third Generation Systems

- 1965 – 1980
- ICs and Multiprogramming
- System 360 and S/370 family of computers
- Spooling (simultaneous peripheral operation on-line)
- Time sharing
- On-line storage for
  - System programs
  - User programs and data
  - Program libraries
- Virtual memory
- Multiprocessor configurations
- MULTICS – Multiplexed Information and Computing Service
  - Design started in 1965 and completed in 1972
  - Collaborative effort between General Electric, Bell Telephone Labs, and Project MAC of MIT
  - Aimed at providing
    * simultaneous computer access to large community of users
    * ample computation power and data storage
    * easy data sharing between users, if desired

Fourth Generation and beyond

- Personal computers and workstations
- MS-DOS and Unix
- Massively parallel systems
  - Pipelining
  - Array processing / SIMD
  - General multiprocessing / MIMD
  - Symmetric multiprocessing / SMD
    * Any process and any thread can run on any available processor

- Computer networks (communication aspect) – network operating systems
- Distributed computing – distributed operating systems

Operating System Concepts

- Program
  - Collection of instructions and data kept in ordinary file on disk
  - The file is marked as executable in the i-node
  - File contents are arranged according to rules established by the kernel
  - Source program, or text file
  - Machine language translation of the source program, or object file
  - Executable program, complete code output by linker/loader, with input from libraries
- Processes
  - Created by kernel as an environment in which a program executes
  - Program in execution
  - May be stopped and later restarted by the OS
  - Core image
    1. Instruction segment
    2. User data segment
    3. System data segment
       - Includes attributes such as current directory, open file descriptors, and accumulated CPU times
       - Information stays outside of the process address space
  - Program initializes the first two segments
  - Process may modify both instructions (rarely) and data
  - Process table – records information about each process
    Program code + data + stack + PC + SP + registers
  - Process may acquire resources (more memory, open files) not present in the program
  - Child and parent processes
  - Communication between processes through messages
  - uid and gid
  - Process id

    0 swapper
    1 /sbin/init
    2 pagedaemon
- Threads
  - Stream of instruction execution
  - A dispatchable unit of work to provide intraprocess concurrency in newer operating systems
  - A process may have multiple threads of execution in parallel, each thread executing sequentially
- Files
  - Services of file management system to hide disk/tape specifics

  - System calls for file management
  - Directory to group files together
  - Organized as a hierarchical tree
  - Root directory
  - Path name
  - Path name separator
  - Working directory
  - Protection of files (9-bit code in Unix – rwx bits)
  - File descriptor or handle – small integer to identify a file in subsequent operations, error code to indicate access denied
    * 0 – standard input
    * 1 – standard output
    * 2 – standard error
  - I/O device treated as a special file
    * block special file
    * character special file
  - Pipe – pseudo file to connect two processes

- System calls
  - Interface between user program and operating system
  - Set of extended instructions provided by the operating system
  - Applied to various software objects like processes and files
  - Invoked by user programs to communicate with the kernel and request services
  - Access routines in the kernel that do the work
  - Library procedure corresponding to each system call
    * Machine registers to hold parameters of system call
    * Trap instruction (protected procedure call) to start OS
    * Hide details of trap and make system call look like ordinary procedure call
    * Return from trap instruction
  - `count = read(file, buffer, nbytes);`
  - Actual system call READ invoked by `read`
  - number of bytes actually read returned in `count`
  - In case of error, `count` is set to -1

- Shell
  - Unix command interpreter
    * Interprets the first word of a command line as a command name
  - Is a user program and not part of the kernel
  - Prompt
  - Redirection of input and output
  - Background jobs
  - For most commands, the shell `fork`s and the child `exec`s the command associated with the name, treating the remaining words on the command line as parameters to the command
  - Allows for three types of commands:

1. Executable files
2. Shellscripts
3. Built-in shell commands

- Kernel
  - Permanently resides in the main memory
  - Controls the execution of processes by allowing their creation, termination or suspension, and communication
  - Schedules processes fairly for execution on the CPU
    - Processes share the CPU in a time-shared manner
      - CPU executes a process
      - Kernel suspends it when its time quantum elapses
      - Kernel schedules another process to execute
      - Kernel later reschedules the suspended process
  - Allocates main memory for an executing process
    - Allows processes to share portions of their address space under certain conditions, but protects the private address space of a process from outside tampering
    - If the system runs low on free memory, the kernel frees memory by writing a process temporarily to secondary memory, or *swap* device
    - If the kernel writes entire processes to a swap device, the implementation of the Unix system is called a *swapping* system; if it writes pages of memory to a swap device, it is called a *paging* system.
    - Coordinates with the machine hardware to set up a virtual to physical address that maps the compiler-generated addresses to their physical addresses
  - File system maintenance
    - Allocates secondary memory for efficient storage and retrieval of user data
    - Allocates secondary storage for user files
    - Reclaims unused storage
    - Structures the file system in a well understood manner
    - Protects user files from illegal access
  - Allows processes controlled access to peripheral devices such as terminals, tape drives, disk drives, and network devices.
  - Services provided by kernel transparently
    - Recognizes that a given file is a regular file or a device but hides the distinction from user processes
    - Formats data in a file for internal storage but hides the internal format from user processes, returning an unformatted byte stream
    - Allows shell to read terminal input, to spawn processes dynamically, to synchronize process execution, to create pipes, and to redirect I/O
  - Kernel in UNIX
    - Traditionally, the operating system itself
    - Isolated from users and applications
    - At the top level, user programs invoke OS services using system calls or library functions
    - At the lowest level, kernel primitives directly interface with the hardware
    - Kernel itself is logically divided into two parts:
      1. *File subsystem* to transfer data between memory and external devices
      2. *Process control subsystem* to control interprocess communication, process scheduling, and memory management
  - Kernel in Windows NT

    * Known as the *executive*; Microsoft calls it a *modified microkernel architecture*
        · Unlike pure microkernel, many of the systems functions outside the microkernel run in kernel mode for performance reasons
    * Manages thread scheduling, process switching, exception and interrupt handling, and multiprocessor synchronization
        · Microkernel's own code does not run in threads
    * As with UNIX, it is isolated from user programs, with user programs and applications allowed to access one of the protected subsystems
        · Each system function is managed by only one component of the OS
        · Rest of the OS and all applications access the function through the responsible component using a standardized interface
        · Key system data can be accessed only through the appropriate function
        · In principle, any module can be removed, upgraded, or replaced without rewriting the entire system or its standard application programming interface (API)
    * Two programming interfaces provided by a subsystem
        1. *Win32 interface* – for traditional Windows users and programmers
        2. *POSIX interface* – to make porting of UNIX applications easier
    * Subsystems and services access the executive using system services
    * Executive contains object manager, security reference monitor, process manager, local procedure call facility, memory manager, and an I/O manager

- Memory
    - Memory hierarchy based on storage capacity, speed, and cost
    - Higher the storage capacity, lesser the speed, and lesser the cost
    - Different memory levels, in decreasing cost per byte of storage

Registers	Few bytes	Almost CPU speed
Cache memory	Few kilobytes	Nanoseconds
Main memory	Megabytes	Microseconds
Magnetic disk	Gigabytes	Milliseconds
Magnetic tape/Optical disk	No limit	Offline storage

    - Use hierarchical memory to transfer data from lower memory to higher memory to be executed
    - Locality of reference
        * Most of the references in the memory are clustered and move from one cluster to the next
    - Volatility
    - Cache memory
        * Use of very fast memory (a few kilobytes) designated to contain data for fast access by the CPU
    - Virtual memory or extension of main memory
    - Disk cache
        * Designating a portion of main memory for disk read/write

- Memory management
    - Memory management is one of the most important services provided by the operating system
    - An operating system has five major responsibilities for managing memory:
        1. Process isolation
            * Should prevent the independent processes from interfering with the data and code segments of each other

2. Automatic allocation and management
   * Programs should be dynamically allocated across the memory depending on the availability (may or may not be contiguous)
   * Programmer should not be able to perceive this allocation
3. Modular programming support
   * Programmers should be able to define program modules
   * Programmers should be able to dynamically create/destroy/alter the size of modules
4. Protection and access control
   * Different programs should be able to co-operate in sharing some memory space
   * Contrast this with the first responsibility
   * Make sure that such sharing is controlled and processes should not be able to indiscriminately access the memory allocated to other processes
5. Long-term storage
   * Users and applications may require means for storing information for extended periods of time
   * Generally implemented with a file system

– OS may separate the memory into two distinct views: physical and logical; this division forms the basis for virtual memory

Process execution modes in Unix

- Two modes of process execution

1. User mode
   – Normal mode of execution for a process
   – Execution of a system call changes the mode to kernel mode
   – Processes can access their own instructions and data but not kernel instructions and data
   – Cannot execute certain privileged machine instructions
2. Kernel mode
   – Processes can access both kernel as well as user instructions and data
   – No limit to which instructions can be executed
   – Runs on behalf of a user process and is a part of the user process

Operating System Structure

- Minimal OS
  – CP/M or DOS
  – Initial Program Loading (Bootstrapping)
  – File system
- Monolithic Structure
  – Most primitive form of operating systems
  – No structure
  – Collection of procedures that can call any other procedure
  – Well-defined interface for procedures
  – No information hiding

  - Services provided by putting parameters in well-defined places and executing a *supervisor call*
    * Switch machine from *user mode* to *kernel mode*
  - Basic structure
    * Main program that invokes requested service procedures
    * Set of service procedures to carry out system calls
    * Set of utility procedures to help the service procedures
  - User program executes until
    * program terminates
    * time-out signal
    * service request
    * interrupt
  - Difficult to maintain
  - Difficult to take care of concurrency due to multiple users/jobs

- Layered Systems
  - Hierarchy of layers – one above the other
  - THE system (1968), MULTICS
  - Six layers
    1. Allocation of processor, switching between processes
    2. Memory and drum management
    3. Operator-process communication – process and operator console
    4. I/O management
    5. User programs
    6. Operator
  - MULTICS
    * organized as a series of concentric rings
    * inner rings more privileged

- Virtual machines
  - Basis for developing the OS
  - Provides a minimal set of operations
  - Creates a virtual CPU for every process
  - IBM System 370 – CMS, VM
  - *Virtual Machine Monitor*
  - Performs functions associated with CPU management and allocation
  - Provides synchronization and/or communication primitives for process communication

- Process Hierarchy
  - Structured as a multilevel hierarchy

    Lowest level. Virtualize CPU for all processes

    Virtual memory. Virtualize memory for all processes
      * Single virtual memory shared by all processes
      * Separate virtual memory for each process

    Virtual I/O devices.

- Client-Server Model
  - Remove as much as possible from the OS leaving a minimal kernel
  - User process (client) sends a request to server process
  - Kernel handles communications between client and server
  - Split OS into parts – file service, process service, terminal service, memory service
  - Servers run in user mode – small and manageable

I/O communication

- Programmed I/O
  - Simplest and least expensive scheme
  - CPU retains control of the device controller and takes responsibility to transfer every bit to/from the I/O devices
  - Bus
    - Address bus: To select a memory location or I/O device
    - Data bus: To transfer data
  - Hardware buffer
  - Handshaking protocol
  - Disadvantages:
    - Poor resource utilization
    - Only one device active at a time
    - Gross mismatch between the speeds of CPU and I/O devices
- Interrupt-driven I/O
  - CPU still retains control of the I/O process
  - Sends an I/O command to the I/O module and goes on to do something else
  - I/O module interrupts the CPU when it is ready to transfer more data
- Direct memory access
  - CPU trusts the DMA module to read from/write into a designated portion of the memory
  - DMA module (also called I/O channel) acts as a slave to the CPU to execute those transfers
  - DMA module takes control of the bus, and that may slow down the CPU if the CPU needs to use the bus

CPU and I/O overlap

- Hardware flag
  CPU is blocked if device is busy
- Polling by test-device-flag
- Memory-mapped I/O
  - Uses memory address register (MAR) and memory buffer register (MBR) to interact with I/O devices
- I/O-mapped I/O
  - Uses I/O address register and I/O buffer register to communicate with the I/O devices

Multiprogramming

- CPU-bound system
- I/O-bound system
- Maintain more than one independent program in the main memory
- Sharing of time and space

Multiprogramming OS

- Requires addition of new hardware components
  - DMA Hardware
  - Priority Interrupt Mechanism
  - Timer
  - Storage and Instruction Protection
  - Dynamic Address Relocation
- Complexity of operating system
- Must hide the sharing of resources between different users
- Must hide details of storage and I/O devices
- Complex file system for secondary storage

Tasks of a Multiprogramming OS

- Bridge the gap between the machine and the user level
- Manage the available resources needed by different users
- Enforce protection policies
- Provide facilities for synchronization and communication

Operating Systems as Virtual Machines

- Allows each user to perceive himself as the only user of the machine
- Fair share of available resources
- Time sharing (for CPU time)
- Abstraction
  - Availability of higher level operations as primitive operations
  - Virtual command language as the machine language of virtual machine
  - Virtual memory

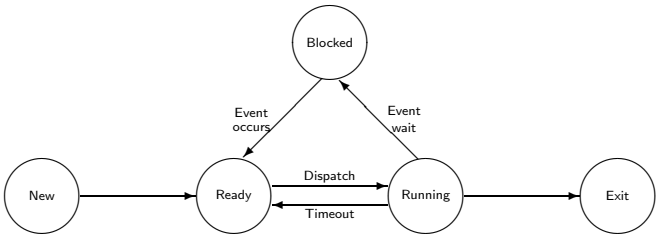
"Only a brain-damaged operating system would support task switching and not make the simple next step of supporting multitasking."

– Calvin Keegan

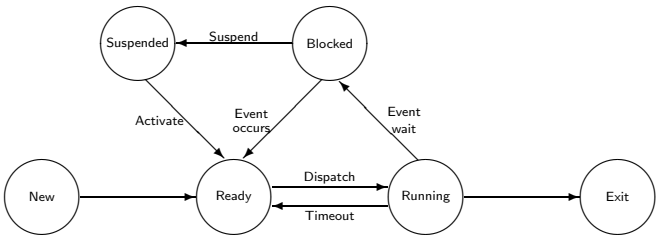
Processes

- Abstraction of a running program
- Unit of work in the system
- Pseudoparallelism
- A process is *traced* by listing the sequence of instructions that execute for that process
- The process model
  - Sequential Process / Task
    - A program in execution
    - Program code
    - Current activity
    - Process stack
      - subroutine parameters
      - return addresses
      - temporary variables
    - Data section
      - Global variables
- Concurrent Processes
  - Multiprogramming
  - Interleaving of traces of different processes characterizes the behavior of the CPU
  - Physical resource sharing
    - Required due to limited hardware resources
  - Logical resource sharing
    - Concurrent access to the same resource like files
  - Computation speedup
    - Break each task into subtasks
    - Execute each subtask on separate processing element
  - Modularity
    - Division of system functions into separate modules
  - Convenience
    - Perform a number of tasks in parallel
  - Real-time requirements for I/O
- Process Hierarchies
  - Parent-child relationship
  - `fork(2)` call in Unix
  - In MS-DOS, parent suspends itself and lets the child execute
- Process states
  - Running

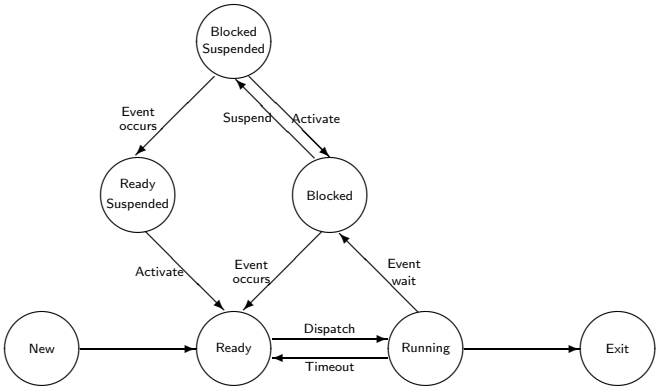
- Ready (Not running, waiting for the CPU)
- Blocked / Wait on an event (other than CPU) (Not running)
- Two other states complete the five-state model – New and Exit
  * A process being created can be said to be in state New; it will be in state Ready after it has been created
  * A process being terminated can be said to be in state Exit



- Above model suffices for most of the discussion on process management in operating systems; however, it is limited in the sense that the system screeches to a halt (even in the model) if all the processes are resident in memory and they all are waiting for some event to happen
- Create a new state Suspend to keep track of blocked processes that have been temporarily kicked out of memory to make room for new processes to come in
- The state transition diagram in the revised model is



- Which process to grant the CPU when the current process is swapped out?
  * Preference for a previously suspended process over a new process to avoid increasing the total load on the system
  * Suspended processes are actually blocked at the time of suspension and making them ready will just change their state back to blocked
  * Decide whether the process is blocked on an event (suspended or not) or whether the process has been swapped out (suspended or not)
- The new state transition diagram is



- Implementation of processes

  - Process table
    * One entry for each process
    * program counter
    * stack pointer
    * memory allocation
    * open files
    * accounting and scheduling information
  - *Interrupt vector*
    * Contains address of *interrupt service procedure*
      · saves all registers in the process table entry
      · services the interrupt

- Process creation
  - Build the data structures that are needed to manage the process
  - When is a process created? – job submission, login, application such as printing
  - Static or dynamic process creation
  - Allocation of resources (CPU time, memory, files)
    * Subprocess obtains resources directly from the OS
    * Subprocess constrained to share resources from a subset of the parent process
  - Initialization data (input)
  - Process execution
    * Parent continues to execute concurrently with its children
    * Parent waits until all its children have terminated

- Processes in Unix
  - Identified by a unique integer – *process identifier*
  - Created by the `fork(2)` system call
    * Copy the three segments (instructions, user-data, and system-data) without initialization from a program
    * New process is the copy of the address space of the original process to allow easy communication of the parent process with its child
    * Both processes continue execution at the instruction after the `fork`
    * Return code for the `fork` is
      · zero for the child process
      · process id of the child for the parent process
  - Use `exec(2)` system call after `fork` to replace the child process's memory space with a new program (binary file)
    * Overlay the image of a program onto the running process
    * Reinitialize a process from a designated program
    * Program changes while the process remains
  - `exit(2)` system call
    * Finish executing a process
  - `wait(2)` system call
    * Wait for child process to stop or terminate
    * Synchronize process execution with the `exit` of a previously `forked` process
  - `brk(2)` system call

        * Change the amount of space allocated for the calling process's data segment
        * Control the size of memory allocated to a process
    - `signal(3)` library function
        * Control process response to extraordinary events
        * The complete family of `signal` functions (see man page) provides for simplified signal management for application processes

- MS-DOS Processes
    - Created by a system call to load a specified binary file into memory and execute it
    - Parent is suspended and waits for child to finish execution

- Process Termination
    - Normal termination
        * Process terminates when it executes its last statement
        * Upon termination, the OS deletes the process
        * Process may return data (output) to its parent
    - Termination by another process
        * Termination by the system call abort
        * Usually terminated only by the parent of the process because
            · child may exceed the usage of its allocated resources
            · task assigned to the child is no longer required
    - Cascading termination
        * Upon termination of parent process
        * Initiated by the OS

- `cobegin/coend`
    - Also known as `parbegin/parend`
    - Explicitly specify a set of program segments to be executed concurrently

```
cobegin
    p_1;
    p_2;
    ...
    p_n;
coend;
```

$$(a + b) \times (c + d) - (e/f)$$

```
cobegin
    t_1 = a + b;
    t_2 = c + d;
    t_3 = e / f;
coend
t_4 = t_1 * t_2;
t_5 = t_4 - t_3;
```

- `fork`, `join`, and `quit` Primitives
    - More general than `cobegin/coend`
    - `fork x`

    - Creates a new process `q` when executed by process `p`
    - Starts execution of process `q` at instruction labeled `x`
    - Process `p` executes at the instruction following the `fork`
  - `quit`
    - Terminates the process that executes this command
  - `join t, y`
    - Provides an indivisible instruction
    - Provides the equivalent of test-and-set instruction in a concurrent language

```
if ( ! --t ) goto y;
```

  - Program segment with new primitives

```
      m = 3;
      fork p2;
      fork p3;
p1 :  t1 = a + b; join m, p4; quit;
p2 :  t2 = c + d; join m, p4; quit;
p3 :  t3 = e / f; join m, p4; quit;
p4 :  t4 = t1 × t2;
      t5 = t4 - t3;
```

Process Control Subsystem in Unix

- Significant part of the Unix kernel (along with the file subsystem)
- Contains three modules
  - Interprocess communication
  - Scheduler
  - Memory management

Interprocess Communication

- Race conditions
  - A race condition occurs when two processes (or threads) access the same variable/resource without doing any synchronization
  - One process is doing a coordinated update of several variables
  - The second process observing one or more of those variables will see inconsistent results
  - Final outcome dependent on the precise timing of two processes
  - Example
    - One process is changing the balance in a bank account while another is simultaneously observing the account balance and the last activity date
    - Now, consider the scenario where the process changing the balance gets interrupted after updating the last activity date but before updating the balance
    - If the other process reads the data at this point, it does not get accurate information (either in the current or past time)

Critical Section Problem

- Section of code that modifies some memory/file/table while assuming its exclusive control

- Mutually exclusive execution in time
- Template for each process that involves critical section

```
do
{
    ...                     /* Entry section;                */
    critical_section();     /* Assumed to be present         */
    ...                     /* Exit section                  */
    remainder_section();    /* Assumed to be present         */
}
while ( 1 );
```

  You are to fill in the gaps specified by `...` for entry and exit sections in this template and test the resulting program for compliance with the protocol specified next

- Design of a protocol to be used by the processes to cooperate with following constraints
  - Mutual Exclusion – If process p_i is executing in its critical section, then no other processes can be executing in their critical sections.
  - Progress – If no process is executing in its critical section, the selection of a process that will be allowed to enter its critical section cannot be postponed indefinitely.
  - Bounded Waiting – There must exist a bound on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.
- Assumptions
  - No assumption about the hardware instructions
  - No assumption about the number of processors supported
  - Basic machine language instructions executed atomically
- Disabling interrupts
  - Brute-force approach
  - Not proper to give users the power to disable interrupts
    - User may not enable interrupts after being done
    - Multiple CPU configuration
- Lock variables
  - Share a variable that is set when a process is in its critical section
- Strict alternation

```
extern int turn;    /* Shared variable between both processes */

do
{
    while ( turn != i ) /* do nothing */ ;
    critical_section();
    turn = j;
    remainder_section();
}
while ( 1 );
```

  - Does not satisfy progress requirement
  - Does not keep sufficient information about the state of each process

- Use of a flag

```
extern int flag[2];          /* Shared variable; one for each process */

do
    flag[i] = 1;             /* true */
    while ( flag[j] );
    critical_section();
    flag[i] = 0;             /* false */
    remainder_section();
while ( 1 );
```

  - Satisfies the mutual exclusion requirement
  - Does not satisfy the progress requirement

    Time T_0 p_0 sets flag[0] to true
    Time T_1 p_1 sets flag[1] to true

    Processes p_0 and p_1 loop forever in their respective while statements
  - Critically dependent on the exact timing of two processes
  - Switch the order of instructions in entry section
    * No mutual exclusion

- Peterson's solution
  - Combines the key ideas from the two earlier solutions

```
/* Code for process 0; similar code exists for process 1 */

extern int flag[2];           /* Shared variables */
extern int turn;              /* Shared variable  */

process_0( void )
{
    do
        /* Entry section */
        flag[0] = true;         /* Raise my flag */
        turn = 1;               /* Cede turn to other process */
        while ( flag[1] && turn == 1 ) ;

        critical_section();

        /* Exit section */
        flag[0] = false;

        remainder_section();

    while ( 1 );
}
```

- Multiple Process Solution – Solution 4
  - The array `flag` can take one of the three values `(idle, want-in, in-cs)`

```
enum state { idle, want_in, in_cs };
extern int turn;
extern state flag[n];   // Flag corresponding to each process (in shared memory)

// Code for process i

int   j;             // Local to each process

do
    do
        flag[i] = want_in;     // Raise my flag
        j = turn;              // Set local variable
        while ( j != i )
            j = ( flag[j] != idle ) ? turn : ( j + 1 ) % n;

        // Declare intention to enter critical section

        flag[i] = in_cs;

        // Check that no one else is in critical section

        for ( j = 0; j < n; j++ )
            if ( ( j != i ) && ( flag[j] == in_cs ) )
                break;

    while ( j < n ) || ( turn != i && flag[turn] != idle );

    // Assign turn to self and enter critical section

    turn = i;
    critical_section();

    // Exit section

    j = (turn + 1) % n;
    while (flag[j] == idle) do
       j = (j + 1) % n;

    // Assign turn to the next waiting process and change own flag to idle

    turn = j;
    flag[i] = idle;

    remainder_section();
while ( 1 );
```

- p_i enters the critical section only if `flag[j]` $\neq$ `in-cs` for all `j` $\neq$ `i`.
- `turn` can be modified only upon entry to and exit from the critical section. The first contending process enters its critical section.
- Upon exit, the successor process is designated to be the one following the current process.
- Mutual Exclusion
  - p_i enters the critical section only if `flag[j]` $\neq$ `in_cs` for all `j` $\neq$ `i`.
  - Only p_i can set `flag[i] = in_cs`.

    * p_i inspects `flag[j]` only while `flag[i] = in_cs`.
  - Progress
    * `turn` can be modified only upon entry to and exit from the critical section.
    * No process is executing or leaving its critical section $\Rightarrow$ `turn` remains constant.
    * First contending process in the cyclic ordering `(turn, turn+1, ..., n-1, 0, ..., turn-1)` enters its critical section.
  - Bounded Wait
    * Upon exit from the critical section, a process must designate its unique successor the first contending process in the cyclic ordering `turn+1, ..., n-1, 0, ..., turn-1, turn`.
    * Any process waiting to enter its critical section will do so in at most `n-1` turns.

- Bakery Algorithm
  - Each process has a unique id
  - Process id is assigned in a completely ordered manner

```
extern bool choosing[n];     /* Shared Boolean array                     */
extern int  number[n];       /* Shared integer array to hold turn number */

void process_i ( int i )     /* ith Process                              */
{
    do
        choosing[i] = true;
        number[i] = 1 + max(number[0], ..., number[n-1]);
        choosing[i] = false;
        for ( int j = 0; j < n; j++ )
        {
            while ( choosing[j] );     /* Wait while someone else is choosing */
            while ( ( number[j] ) && (number[j],j) < (number[i],i) );
        }

        critical_section();

        number[i] = 0;

        remainder_section();
    while ( 1 );
}
```

  - If p_i is in its critical section and p_k $(k \neq i)$ has already chosen its number[k] $\neq$ 0, then (number[i],i) $<$ (number[k],k).

Synchronization Hardware

- `test_and_set` instruction

```
int test_and_set (int& target )
{
    int tmp;
    tmp = target;
    target = 1;  /* True */
    return ( tmp );
}
```

- Implementing Mutual Exclusion with `test_and_set`

```
do
   while test_and_set ( lock );
   critical_section();
   lock = false;
   remainder_section();
while ( 1 );
```

Semaphores

- Producer-consumer Problem
  - Shared buffer between producer and consumer
  - Number of items kept in the variable `count`
  - Printer spooler
  - The | operator
  - Race conditions
- An integer variable that can only be accessed through two standard atomic operations – wait (P) and signal (V)

Operation	Semaphore	Dutch	Meaning
Wait	P	*proberen*	test
Signal	V	*verhogen*	increment

- The classical definitions for *wait* and *signal* are

```
wait ( S ):     while ( S <= 0 );
                S--;

signal ( S ):   S++;
```

- Mutual exclusion implementation with semaphores

```
do
    wait (mutex);
    critical_section();
    signal (mutex);
    remainder_section();
while ( 1 );
```

- Synchronization of processes with semaphores

p_1	S_1; `signal (synch);`
p_2	`wait (synch);` S_2;

- Implementing Semaphore Operations
  - Binary semaphores using `test_and_set`
    - Check out the instruction definition as previously given
  - Implementation with a *busy-wait*

```
class bin_semaphore
{
    private:
        bool       s;      /* Binary semaphore    */
    public:
        bin_semaphore ( void )    // Default constructor
        {
            s = false;
        }

        void P ( void )           // Wait on semaphore
        {
            while ( test_and_set ( s ) );
        }

        void V ( void )           // Signal the semaphore
        {
            s = false;
        }
}
```

– General semaphore

```
class semaphore
{
    private:
        bin_semaphore    mutex;
        bin_semaphore    delay;
        int              count;

    public:
        void semaphore ( void )        // Default constructor
        {
            count = 1;
            delay.P();
        }

        void semaphore ( int num )     // Parameterized constructor
        {
            count = num;
            delay.P();
        }

        void P ( void )
        {
            mutex.P();
            if ( --count < 0 )
            {
                mutex.V();
                delay.P();
            }
            mutex.V();
        }

        void V ( void )
```

```
        {
            mutex.P();
            if ( ++count <= 0 )
                delay.V();
            else
                mutex.V();
        }
}
```

– Busy-wait Problem – Processes waste CPU cycles while waiting to enter their critical sections
  * Modify `wait` operation into the `block` operation. The process can block itself rather than busy-waiting.
  * Place the process into a wait queue associated with the critical section
  * Modify `signal` operation into the `wakeup` operation.
  * Change the state of the process from *wait* to *ready*.

– Block-Wakeup Protocol

```
// Semaphore with block wakeup protocol

class sem_int
{
    private:
        int       value;     // Number of resources
        queue     l;         // List of processes

    public:
        void sem_int ( void )     // Default constructor
        {
            value = 1;
            l = create_queue();
        }

        void sem_int ( int n )    // Constructor function
        {
            value = n;
            l = create_queue();
        }

        void P ( void )
        {
            if ( --value < 0 )
            {
                enqueue ( l, p );   // Enqueue the invoking process
                block();
            }
        }

        void V ( void )
        {
            if ( ++value <= 0 )
            {
                process p = dequeue ( l );
                wakeup ( p );
            }
        }
```

```
    };
```

Producer-Consumer problem with semaphores

```
void producer ( void )
{
    do
        produce ( item );
        wait ( empty );       // empty is semaphore
        wait ( mutex );       // mutex is semaphore
        put ( item );
        signal ( mutex );
        signal ( full );
    while ( 1 );
}

void consumer ( void )
{
    do
        wait ( full );
        wait ( mutex );
        remove ( item );
        signal ( mutex );
        signal ( empty );
        consume ( item );
    while ( 1 );
}
```

Problem: What if order of `wait` is reversed in `producer`

Event Counters

- Solve the producer-consumer problem without requiring mutual exclusion
- Special kind of variable with three operations
  1. `E.read()`: Return the current value of `E`
  2. `E.advance()`: Atomically increment `E` by 1
  3. `E.await(v)`: Wait until `E` has a value of `v` or more
- Event counters always start at 0 and always increase

```
class event_counter
{
    int     ec;    // Event counter

    public:
        event_counter ( void )            // Default constructor
        {
            ec = 0;
        }
        int read ( void ) const { return ( ec ); }
        void advance ( void ) { ec++; }
        void await ( const int v ) { while ( ec < v ); }
};
```

```
extern event_counter    in, out;         // Shared event counters
void producer ( void )
{
    int sequence ( 0 );
    do
        produce ( item );
        sequence++;
        out.await ( sequence - num_buffers );
        put ( item );
        in.advance();
    while ( 1 );
}
void consumer ( void )
{
    int sequence ( 0 );
    do
        sequence++;
        in.await ( sequence );
        remove ( item );
        out.advance();
        consume ( item );
    while ( 1 );
}
```

Higher-Level Synchronization Methods

- P and V operations do not permit a segment of code to be designated explicitly as a critical section.
- Two parts of a semaphore operation
  - Block-wakeup of processes
  - Counting of semaphore
- Possibility of a *deadlock* – Omission or unintentional execution of a V operation.
- Monitors
  - Implemented as a class with private and public functions
  - Collection of data [resources] and private functions to manipulate this data
  - A monitor must guarantee the following:
    - Access to the resource is possible only via one of the monitor procedures.
    - Procedures are mutually exclusive in time. Only one process at a time can be active within the monitor.
  - Additional mechanism for synchronization or communication – the `condition` construct

    ```
    condition x;
    ```

    - `condition` variables are accessed by only two operations – `wait` and `signal`
    - `x.wait()` suspends the process that invokes this operation until another process invokes `x.signal()`
    - `x.signal()` resumes exactly one suspended process; it has no effect if no process is suspended
  - Selection of a process to execute within monitor after `signal`
    - `x.signal()` executed by process `P` allowing the suspended process `Q` to resume execution
      1. `P` waits until `Q` leaves the monitor, or waits for another condition
      2. `Q` waits until `P` leaves the monitor, or waits for another condition

Choice 1 advocated by Hoare

- The Dining Philosophers Problem – Solution by Monitors

```
enum state_type { thinking, hungry, eating };

class dining_philosophers
{
    private:
        state_type state[5];      // State of five philosophers
        condition  self[5];       // Condition object for synchronization

        void test ( int i )
        {
            if ( ( state[ ( i + 4 ) % 5 ] != eating )  &&
                 ( state[ i ] == hungry )            &&
                 ( state[ ( i + 1 ) % 5 ] != eating ) )
            {
                state[ i ] = eating;
                self[i].signal();
            }
        }

    public:
        void dining_philosophers ( void )   // Constructor
        {
            for ( int i = 0; i < 5; state[i++] = thinking );
        }

        void pickup ( int i )     // i corresponds to the philosopher
        {
            state[i] = hungry;
            test ( i );
            if ( state[i] != eating )
                self[i].wait();
        }

        void putdown ( int i )    // i corresponds to the philosopher
        {
            state[i] = thinking;
            test ( ( i + 4 ) % 5 );
            test ( ( i + 1 ) % 5 );
        }
}
```

  - Philosopher i must invoke the operations pickup and putdown on an instance dp of the dining_philosophers monitor

```
dining_philosophers dp;

dp.pickup(i);       // Philosopher i picks up the chopsticks
   ...
dp.eat(i);          // Philosopher i eats (for random amount of time)
   ...
dp.putdown(i);      // Philosopher i puts down the chopsticks
```

  - No two neighbors eating simultaneously – no deadlocks
  - Possible for a philosopher to starve to death

- Implementation of a Monitor
  - Execution of procedures must be mutually exclusive
  - A `wait` must block the current process on the corresponding `condition`
  - If no process in running in the monitor and some process is waiting, it must be selected. If more than one waiting process, some criterion for selecting one must be deployed.
  - Implementation using semaphores
    * Semaphore `mutex` corresponding to the monitor initialized to 1
      · Before entry, execute `wait(mutex)`
      · Upon exit, execute `signal(mutex)`
    * Semaphore `next` to suspend the processes unable to enter the monitor initialized to 0
    * Integer variable `next_count` to count the number of processes waiting to enter the monitor

```
mutex.wait();
    ...
void P ( void ) { ... }    // Body of P()
    ...
if ( next_count > 0 )
    next.signal();
else
    mutex.signal();
```

    * Semaphore `x_sem` for condition `x`, initialized to 0
    * Integer variable `x_count`

```
class condition
{
    int          num_waiting_procs;
    semaphore    sem;
    static int   next_count;
    static semaphore next;
    static semaphore mutex;

    public:
        condition ( void )         // Default constructor
        : sem ( 0 )
        {
            num_waiting_procs = 0;
        }

        void wait ( void )
        {
            num_waiting_procs++;
            if ( next_count > 0 )
                next.signal();
            else
                mutex.signal();
            sem.wait();
            num_waiting_procs--;
        }
```

```
        void signal ( void )
        {
            if ( num_waiting_procs <= 0 )
                return;
            num_waiting_procs++;
            sem.signal();
            next.wait();
            next_count--;
        }
};
```

- Conditional Critical Regions (CCRs)
  - Designed by Hoare and Brinch-Hansen to overcome the deficiencies of semaphores
  - Explicitly designate a portion of code to be critical section
  - Specify the variables (resource) to be protected by the critical section

    `resource r :: v_1, v_2, ..., v_n`
  - Specify the conditions under which the critical section may be entered to access the elements that form the resource

    `region r when B do S`
    - `B` is a condition to guard entry into critical section `S`
    - At any time, only one process is permitted to enter the code segment associated with resource `r`
  - The statement `region r when B do S` is implemented by

```
semaphore mutex ( 1 ), delay ( 0 );
int       delay_cnt ( 0 );

mutex.P();
del_cnt++;
while ( !B )
{
    mutex.V();
    delay.P();
    mutex.P();
}
del_cnt--;
S;            // Critical section code
for ( int i ( 0 ); i < del_cnt; i++ )
    delay.V();
mutex.V();
```

Message-Based Synchronization Schemes

- Communication between processes is achieved by:
  - Shared memory (semaphores, CCRs, monitors)
  - Message systems
    - Desirable to prevent sharing, possibly for security reasons or no shared memory availability due to different physical hardware
- Communication by Passing Messages
  - Processes communicate without any need for shared variables

  - Two basic communication primitives
    * `send` message
    * `receive` message

`send(P, message)`	Send a message to process P
`receive(Q, message)`	Receive a message from process Q

  - Messages passed through a *communication link*

- Producer/Consumer Problem

```
void producer ( void )
{
    while ( 1 )
    {
        produce ( data );
        send ( consumer, data );
    }
}
```

```
void consumer ( void )
{
    while ( 1 )
    {
        receive ( producer, data );
        consume ( data );
    }
}
```

- Issues to be resolved in message communication
  - *Synchronous v/s Asynchronous Communication*
    * Upon `send`, does the sending process continue (asynchronous or nonblocking communication), or does it wait for the message to be accepted by the receiving process (synchronous or blocking communication)?
    * What happens when a `receive` is issued and there is no message waiting (blocking or nonblocking)?
  - *Implicit v/s Explicit Naming*
    * Does the sender specify exactly one receiver (explicit naming) or does it transmit the message to all the other processes (implicit naming)?

`send (p, message)`	Send a `message` to process `p`
`send (A, message)`	Send a `message` to mailbox `A`

    * Does the receiver accept from a certain sender (explicit naming) or can it accept from any sender (implicit naming)?

`receive (p, message)`	Receive a `message` from process `p`
`receive (id, message)`	Receive a `message` from any process; `id` is the process id
`receive (A, message)`	Receive a `message` from mailbox `A`

Ports and Mailboxes

- Achieve synchronization of asynchronous process by embedding a busy-wait loop, with a non-blocking `receive` to simulate the effect of implicit naming
  - Inefficient solution
- Indirect communication avoids the inefficiency of busy-wait
  - Make the queues holding messages between senders and receivers visible to the processes, in the form of mailboxes
  - Messages are sent to and received from mailboxes
  - Most general communication facility between n senders and m receivers

  - Unique identification for each mailbox
  - A process may communicate with another process by a number of different mailboxes
  - Two processes may communicate only if they have a shared mailbox
- Properties of a communication link
  - A link is established between a pair of processes only if they have a shared mailbox
  - A link may be associated with more than two processes
  - Between each pair of communicating processes, there may be a number of different links, each corresponding to one mailbox
  - A link may be either unidirectional or bidirectional
- Ports
  - In a distributed environment, the `receive` referring to same mailbox may reside on different machines
  - Port is a limited form of mailbox associated with only one receiver
  - All messages originating with different processes but addressed to the same port are sent to one central place associated with the receiver

Remote Procedure Calls

- High-level concept for process communication
- Transfers control to another process, possibly on a different computer, while suspending the calling process
- Called procedure resides in separate address space and no global variables are shared
- Communication strictly by parameters

```
send (RP_guard, parameters);
receive (RP_guard, results);
```

- The remote procedure guard is implemented by

```
void RP_guard ( void )
{
    do
        receive (caller, parameters);
        ...
        send (caller, results);
    while ( 1 );
}
```

- Static versus dynamic creation of remote procedures
- `rendezvous` mechanism in Ada

Process Scheduling

The Operating System Kernel

- Basic set of primitive operations and processes
  - *Primitive:*
    - Like a subroutine call or macro expansion
    - Part of the calling process
    - Critical section for the process
  - *Process*
    - Synchronous execution with respect to the calling process
    - Can block itself or continuously poll for work
    - More complicated than primitives and more time and space
- Provides a way to provide protected system services, like *supervisor call instruction*
  - Protects the OS and key OS data structures (like process control blocks) from interference by user programs
  - The fact that a process is executing in kernel mode is indicated by a bit in program status word (PSW)
- Set of kernel operations
  - Process Management: Process creation, destruction, and interprocess communication; scheduling and dispatching; process switching; management of process control blocks
  - Resource Management: Memory (allocation of address space; swapping; page and segment management), secondary storage, I/O devices, and files
  - Input/Output: Transfer of data between memory and I/O devices; buffer management; allocation of I/O channels and devices to processes
  - Interrupt handling: Process termination, I/O completion, service requests, software errors, hardware malfunction
- Kernel in Unix
  - Controls the execution of processes by allowing their creation, termination, suspension, and communication
  - Schedules processes *fairly* for execution on CPU
    - CPU executes a process
    - Kernel suspends process when its time quantum elapses
    - Kernel schedules another process to execute
    - Kernel later reschedules the suspended process
  - Allocates main memory for an executing process

    - Swapping system: Writes entire process to the swap device
    - Paging system: Writes pages of memory to the swap device
  - Allocates secondary memory for efficient storage and retrieval of user data
  - Allows controlled peripheral device access to processes
- Highest Level of User Processes: The *shell* in Unix
  - Created for each user (login request)
  - Initiates, monitors, and controls the progress for user
  - Maintains global accounting and resource data structures for the user
  - Keeps static information about user, like identification, time requirements, I/O requirements, priority, type of processes, resource needs
  - May create child processes (progenies)
- Process image
  - Collection of programs, data, stack, and attributes that form the process
  - User data
    - Modifiable part of the user space
    - Program data, user stack area, and modifiable code
  - User program
    - Executable code
  - System stack
    - Used to store parameters and calling addresses for procedure and system calls
  - Process control block
    - Data needed by the OS to control the process

Data Structures for Processes and Resources

- Process control block
  - Most important data structure in an OS
  - Read and modified by almost every subsystem in the OS, including scheduler, resource allocator, and performance monitor
  - Constructed at process creation time
    - Physical manifestation of the process
    - Set of data locations for local and global variables and any defined constants
  - Contains specific information associated with a specific process
    - The information can be broadly classified as process identification, processor state information, and process control information

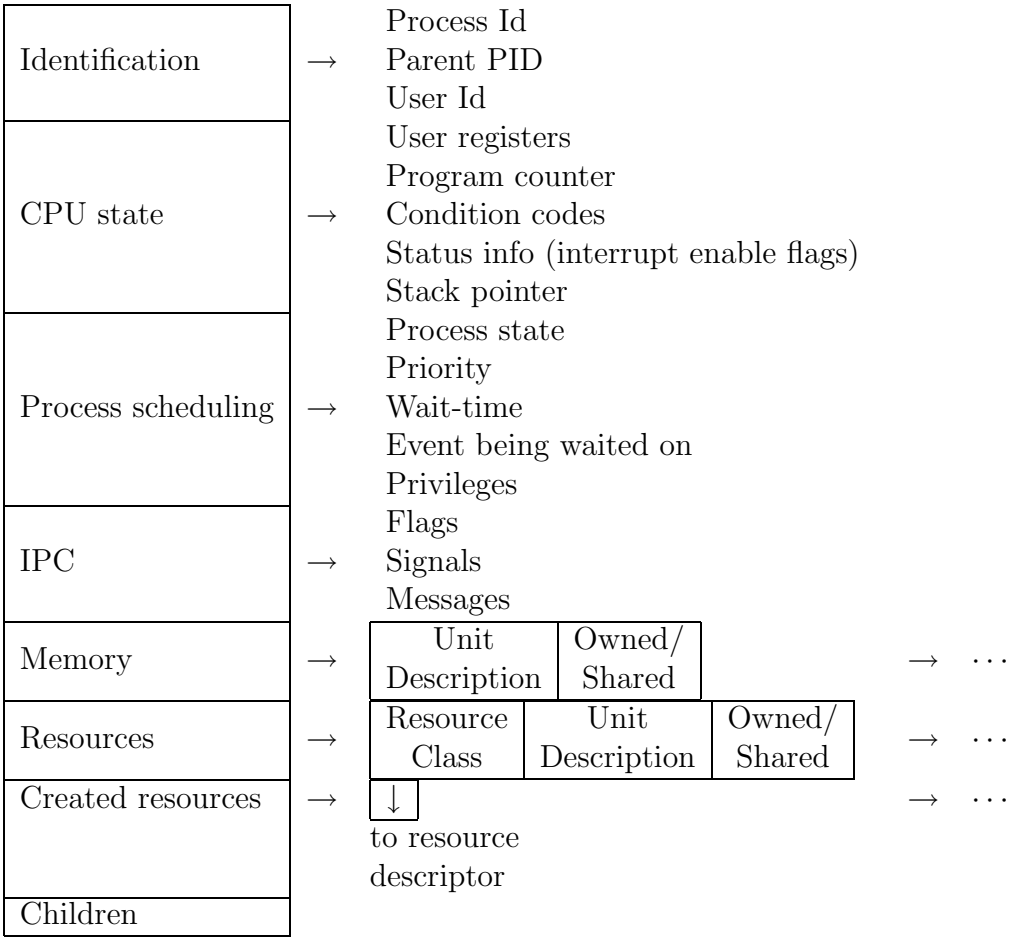


Figure 1: Process Control Block

* Can be described by Figure 1
* *Identification*. Provided by a pointer p^ to the PCB
  - Always a unique integer in Unix, providing an index into the primary process table
  - Used by all subsystems in the OS to determine information about a process
  - Parent of the process
* *CPU state*.
  - Provides snapshot of the process
  - While the process is running, the user registers contain a certain value that is to be saved if the process is interrupted
  - Typically described by the registers that form the processor status word (PSW) or *state vector*
  - Exemplified by EFLAGS register on Pentium that is used by any OS running on Pentium, including Unix and Windows NT
* *Process State*. Current activity of the process
  - Running. Executing instructions.
  - Ready. Waiting to be assigned to a processor.

            - Blocked. Waiting for some event to occur.
        - Allocated address space (Memory map)
        - Resources associated with the process (open files, I/O devices)
        - *Other Information.*
            - Progenies/offsprings of the process
            - Priority of the process
            - Accounting information. Amount of CPU and real time used, time limits, account numbers, etc.
            - I/O status information. Outstanding I/O requests, I/O devices allocated, list of open files, etc.

- Resource Descriptors.
    - Resource.
        - Reusable, relatively stable, and often scarce commodity
        - Successively requested, used, and released by processes
        - Hardware (physical) or software (logical) components
    - Resource class.
        - *Inventory.* Number and identification of available units
        - *Waiting list.* Blocked processes with unsatisfied requests
        - *Allocator.* Selection criterion for honoring the requests
    - Contains specific information associated with a certain resource
        - `p^.status_data` points to the waiting list associated with the resource.
        - Dynamic and static resource descriptors, with static descriptors being more common
        - Identification, type, and origin
            - Resource class identified by a pointer to its descriptor
            - Descriptor pointer maintained by the creator in the process control block
            - External resource name $\Rightarrow$ unique resource id
            - Serially reusable or consumable?
        - Inventory List
            - Associated with each resource class
            - Shows the availability of resources of that class
            - Description of the units in the class
        - Waiting Process List
            - List of processes blocked on the resource class
            - Details and size of the resource request
            - Details of resources allocated to the process
        - Allocator
            - Matches available resources to the requests from blocked processes

        - Uses the inventory list and the waiting process list
      - Additional Information
        - Measurement of resource demand and allocation
        - Total current allocation and availability of elements of the resource class

- Context of a process
  - Information to be saved that may be altered when an interrupt is serviced by the interrupt handler
  - Includes items such as program counter, registers, and stack pointer

Basic Operations on Processes and Resources

- Implemented by kernel primitives
- Maintain the *state* of the operating system
- Indivisible primitives protected by "busy-wait" type of locks
- *Process Control Primitives*
  - `create`. Establish a new process
    - Assign a new unique process identifier (PID) to the new process
    - Allocate memory to the process for all elements of process image, including private user address space and stack; the values can possibly come from the parent process; set up any linkages, and then, allocate space for process control block
    - Create a new process control block corresponding to the above PID and add it to the process table; initialize different values in there such as parent PID, list of children (initialized to `null`), program counter (set to program entry point), system stack pointer (set to define the process stack boundaries)
    - Initial CPU state, typically initialized to *Ready* or *Ready, suspend*
    - Add the process id of new process to the list of children of the creating (parent) process
    - $\mathtt{r}_0$. Initial allocation of resources
    - $\mathtt{k}_0$. Initial priority of the process
    - Accounting information and limits
    - Add the process to the *ready list*
    - Initial allocation of memory and resources must be a subset of parent's and be assigned as shared
    - Initial priority of the process can be greater than the parent's
  - `suspend`. Change process state to suspended
    - A process may suspend only its descendants
    - May include cascaded suspension

    - Stop the process if the process is in *running state* and save the state of the processor in the process control block
    - If process is already in *blocked state*, then leave it blocked, else change its state to *ready state*
    - If need be, call the `scheduler` to schedule the processor to some other process
  - `activate`. Change process state to active
    - Change one of the descendant processes to *ready state*
    - Add the process to the *ready list*
  - `destroy`. Remove one or more processes
    - Cascaded destruction
    - Only descendant processes may be destroyed
    - If the process to be "killed" is running, stop its execution
    - Free all the resources currently allocated to the process
    - Remove the process control block associated with the killed process
  - `change_priority`. Set a new priority for the process
    - Change the priority in the process control block
    - Move the process to a different queue to reflect the new priority

- Resource Primitives
  - `create_resource_class`. Create the descriptor for a new resource class
    - Dynamically establish the descriptor for a new resource class
    - Initialize and define inventory and waiting lists
    - Criterion for allocation of the resources
    - Specification for insertion and removal of resources
  - `destroy_resource_class`. Destroy the descriptor for a resource class
    - Dynamically remove the descriptor for an existing resource class
    - Resource class can only be destroyed by its creator or an ancestor of the creator
    - If any processes are waiting for the resource, their state is changed to *ready*
  - `request`. Request some units of a resource class
    - Includes the details of request – number of resources, absolute minimum required, urgency of request
    - Request details and calling process-id are added to the waiting queue
    - Allocation details are returned to the calling process
    - If the request cannot be immediately satisfied, the process is blocked
    - Allocator gives the resources to waiting processes and modifies the allocation details for the process and its inventory
    - Allocator also modifies the resource ownership in the process control block of the process
  - `release`. Release some units of a resource class

  - Return unwanted and serially reusable resources to the resource inventory
  - Inform the allocator about the return

Organization of Process Schedulers

- Objective of Multiprogramming: Maximize CPU utilization and increase *throughput*
- Two processes P_0 and P_1

P_0	t_0	i_0	t_1	i_1	$\cdots$	i_{n-1}	t_n

P_1	t'_0	i'_0	t'_1	i'_1	$\cdots$	i'_{m-1}	t'_m

- Two processes P_0 and P_1 without multiprogramming

P_0	t_0	i_0	t_1	i_1	$\cdots$	i_{n-1}	t_n	P_0 terminated						
P_1	P_1 waiting							t'_0	i'_0	t'_1	i'_1	$\cdots$	i'_{m-1}	t'_m

- Processes P_0 and P_1 with multiprogramming

P_0	t_0		t_1		$\cdots$	t_n	
P_1		t'_0		t'_1	$\cdots$		t'_m

- Each entering process goes into *job queue*. Processes in job queue
  - reside on mass storage
  - await allocation of main memory
- Processes residing in main memory and awaiting CPU time are kept in *ready queue*
- Processes waiting for allocation of a certain I/O device reside in *device queue*
- *Scheduler*
  - Concerned with deciding a policy about which process to be dispatched
  - After selection, loads the process state or dispatches
  - Process selection based on a scheduling algorithm
- Short-term v/s Long-term schedulers
  - Long-term scheduler
    - Selects processes from job queue
    - Loads the selected processes into memory for execution
    - Updates the ready queue
    - Controls the *degree of multiprogramming* (the number of processes in the main memory)

    - Not executed as frequently as the short-term scheduler
    - Should generate a good mix of CPU-bound and I/O-bound proesses
    - May not be present in some systems (like time sharing systems)
  - Short-term scheduler
    - Selects processes from ready queue
    - Allocates CPU to the selected process
    - Dispatches the process
    - Executed frequently (every few milliseconds, like 10 msec)
    - Must make a decision quickly $\Rightarrow$ must be extremely fast

Process or CPU Scheduling

- *Scheduler* decides the process to run first by using a *scheduling algorithm*
- Desirable features of a scheduling algorithm
  - Fairness: Make sure each process gets its fair share of the CPU
  - Efficiency: Keep the CPU busy 100% of the time
  - Response time: Minimize response time for interactive users
  - Turnaround: Minimize the time batch users must wait for output
  - Throughput: Maximize the number of jobs processed per hour
- Types of scheduling
  - Preemptive
    - Temporarily suspend the logically runnable processes
    - More expensive in terms of CPU time (to save the processor state)
    - Can be caused by

      Interrupt. Not dependent on the execution of current instruction but a reaction to an external asynchronous event

      Trap. Happens as a result of execution of the current instruction; used for handling error or exceptional condition

      Supervisor call. Explicit request to perform some function by the kernel
  - Nonpreemptive
    - Run a process to completion
- The Universal Scheduler: specified in terms of the following concepts
  1. Decision Mode
     - Select the process to be assigned to the CPU
  2. Priority function

  - Applied to all processes in the ready queue to determine the $current\ priority$
3. Arbitration rule
  - Applied to select a process in case two processes are found with the same current priority

- The Decision Mode
  * Time (decision epoch) to select a process for execution
  * Preemptive and nonpreemptive decision
  * Selection of a process occurs
    1. when a new process arrives
    2. when an existing process terminates
    3. when a waiting process changes state to ready
    4. when a running process changes state to waiting (I/O request)
    5. when a running process changes state to ready (interrupt)
    6. every q seconds (quantum-oriented)
    7. when priority of a ready process exceeds the priority of a running process
  * Selective preemption: Uses a bit pair (u_p, v_p)
    u_p set if p may preempt another process
    v_p set if p may be preempted by another process
- The Priority Function
  * Defines the priority of a ready process using some parameters associated with the process
  * Memory requirements
    Important due to swapping overhead
    Smaller memory size $\Rightarrow$ Less swapping overhead
    Smaller memory size $\Rightarrow$ More processes can be serviced
  * Attained service time
    Total time when the process is in the running state
  * Real time in system
    Total actual time the process spends in the system since its arrival
  * Total service time
    Total CPU time consumed by the process during its lifetime
    Equals attained service time when the process terminates
    Higher priority for shorter processes
    Preferential treatment of shorter processes reduces the average time a process spends in the system
  * External priorities
    Differentiate between classes of user and system processes
    Interactive processes $\Rightarrow$ Higher priority
    Batch processes $\Rightarrow$ Lower priority
    Accounting for the resource utilization

    - Timeliness
      Dependent upon the urgency of a task
      Deadlines
    - System load
      Maintain good response time during heavy load
      Reduce swapping overhead by larger quanta of time
  - The Arbitration Rule
    - Random choice
    - Round robin (cyclic ordering)
    - chronological ordering (FIFO)

- Time-Based Scheduling Algorithms
  - May be independent of required service time
- First-in/First-out Scheduling
  - Simplest CPU-scheduling algorithm
  - Nonpreemptive decision mode
  - Upon process creation, link its PCB to rear of the FIFO queue
  - Scheduler allocates the CPU to the process at the front of the FIFO queue
  - Average waiting time can be long

Process	Burst time
P_1	24
P_2	3
P_3	3

    Let the processes arrive in the following order:

$$P_1, P_2, P_3$$

    Then, the average waiting time is calculated from:

P_1		P_2		P_3	
1	24	25	27	28	30

    Average waiting time $= \frac{0+24+27}{3} = 17$ units
  - Priority function $P(r) = r$
- Last-in/First-out Scheduling
  - Similar to FIFO scheduling
  - Average waiting time is calculated from:

P_3		P_2		P_1	
1	3	4	6	7	30

Average waiting time $= \frac{0+3+6}{3} = 3$ units
Substantial saving but what if the order of arrival is reversed.

- Priority function $P(r) = -r$

- Shortest Job Next Scheduling
  - Associate the length of the next CPU burst with each process
  - Assign the process with shortest CPU burst requirement to the CPU
  - Nonpreemptive scheduling
  - Specially suitable to batch processing (long term scheduling)
  - Ties broken by FIFO scheduling
  - Consider the following set of processes

Process	Burst time
P_1	6
P_2	8
P_3	7
P_4	3

Scheduling is done as:

P_4	P_1	P_3	P_2
1 3	4 9	10 16	17 24

Average waiting time $= \frac{3+16+9+0}{4} = 7$ units
  - Using FIFO scheduling, the average waiting time is given by $\frac{0+6+14+21}{4} = 10.25$ units
  - Priority function $P(t) = -t$
  - Provably optimal scheduling – Least average waiting time
    - Moving a short job before a long one decreases the waiting time for short job more than it increase the waiting time for the longer process
  - Problem: To determine the length of the CPU burst for the jobs

- Longest Job First Scheduling – Homework exercise

- Shortest Remaining Time Scheduling
  - Preemptive version of shortest job next scheduling
  - Preemptive in nature (only at arrival time)
  - Highest priority to process that need least time to complete
  - Priority function $P(\tau) = \frac{1}{\tau}$
  - Consider the following processes

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

  - Schedule for execution

P_1	P_2	P_4	P_1	P_3
1 2	5 6	10 11	17 18	26

  - Average waiting time calculations

- Round-Robin Scheduling
  - Preemptive in nature
  - Preemption based on time slices or time quanta
  - Time quantum between 10 and 100 milliseconds
  - All user processes treated to be at the same priority
  - Ready queue treated as a circular queue
    - New processes added to the rear of the ready queue
    - Preempted processes added to the rear of the ready queue
    - Scheduler picks up a process from the head of the queue and dispatches it with a timer interrupt set after the time quantum
  - CPU burst $<$ 1 quantum $\Rightarrow$ process releases CPU voluntarily
  - Timer interrupt results in context switch and the process is put at the rear of the ready queue
  - No process is allocated CPU for more than 1 quantum in a row
  - Consider the following processes

Process	Burst time
P_1	24
P_2	3
P_3	3

    Time quantum $=$ 4 milliseconds

P_1	P_2	P_3	P_1	P_1	P_1	P_1	P_1
1 4	5 7	8 10	11 14	15 18	19 22	23 26	27 30

    Average waiting time $= \frac{6+4+7}{3} = 5.66$ milliseconds
  - Let there be
    n processes in the ready queue, and q be the time quantum, then
    each process gets $\frac{1}{n}$ of CPU time in chunks of at most q time units
    Hence, each process must wait no longer than $(n-1) \times q$ time units for its next quantum
  - Performance depends heavily on the size of time quantum

    - Large time quantum ⇒ FIFO scheduling
    - Small time quantum ⇒ Large context switching overhead
    - Rule of thumb: 80% of the CPU bursts should be shorter than the time quantum

- Multilevel Feedback Scheduling
  - Most general CPU scheduling algorithm
  - Background
    - Make a distinction between foreground (interactive) and background (batch) processes
    - Different response time requirements for the two types of processes and hence, different scheduling needs
    - Separate queue for different types of processes, with the process priority being defined by the queue
  - Separate processes with different CPU burst requirements
  - Too much CPU time ⇒ lower priority
  - I/O-bound and interactive process ⇒ higher priority
  - $Aging$ to prevent starvation
  - n different priority levels – $\Pi_1 \cdots \Pi_n$
  - Each process may not receive more than T_Π time units at priority level Π
  - When a process receives time T_Π, decrease its priority to $\Pi - 1$
  - Process may remain at the lowest priority level for infinite time

- Policy Driven CPU Scheduling
  - Based on a $policy\ function$
  - Policy function gives the correlation between actual and desired resource utilization
  - Attempt to strike a balance between actual and desired resource utilization

- Comparison of scheduling methods
  - Average waiting time
  - Average turnaround time

The Deadlock Problem

Law passed by the Kansas Legislature in early 20th century:
"When two trains approach each other at a crossing, both shall come to a full stop and neither shall start upon again until the other has gone."

Neil Groundwater has the following to say about working with Unix at Bell Labs in 1972:

> ... the terminals on the development machine were in a common room ... when one wanted to use the line printer. There was no spooling or lockout. `pr myfile > /dev/lp` was how you sent your listing to the printer. If two users sent output to the printer at the same time, their outputs were interspersed. Whoever shouted. "line printer!" first owned the queue.[1]

- Permanent blocking of a set of processes that either compete for system resources or communicate with each other
  - Several processes may compete for a finite set of resources
  - Processes request resources and if a resource is not available, enter a wait state
  - Requested resources may be held by other waiting processes
  - Require divine intervention to get out of this problem
- A significant problem in real systems
- Little attention paid to the study of the problem because
  - Most multiprogramming systems limit parallelism to some system processes only, and only on a limited basis
  - Systems allocate resources to processes statically
- Deadlock problem becoming more important because of increasing use of multiprocessing systems (like real-time, life support, vehicle monitoring)
- Important in answering the question about the completion of a process
- Deadlocks can occur with
  - Serially reusable resources – printer, tape drive, memory
  - Serially consumable resources – messages

Examples of Deadlocks in Computer Systems

- File Sharing
  - Consider two processes p_1 and p_2
  - They update a file F and require a scratch tape during the updating
  - Only one tape drive T available
  - T and F are serially reusable resources, and can be used only by $exclusive\ access$
  - p_2 needs T immediately prior to updating
  - $request$ operation
    - Blocks the process requesting the resource
    - Puts the process on the wait queue
    - The process is to remain blocked until the requested resource is available
    - If the resource is available, the process is granted an exclusive access to it.

[1] Peter H. Salus. *A Quarter Century of UNIX*. Addison Wesley, Reading, MA. 1994

  - *release* operation
    - Returns the resource being released to the system
    - Wakes up the process waiting for the resource, if any
  - p_1 and p_2 may run as follows

p_1:	⋮	p_2:	⋮
	`request(F);`		`request(T);`
r_1:	`request(T);`		⋮
	⋮	r_2:	`request(F);`
	⋮		⋮
	`release(T);`		`release(F);`
	`release(F);`		`release(T);`
	⋮		⋮

  - p_1 can block on T holding F while p_2 can block on F holding T

- Single Resource Sharing
  - Deadlock due to no memory being available and existing processes requesting more memory
  - Fairly common cause of deadlock
- Locking in Database Systems
  - Locking required to preserve the *consistency* of databases
  - Problem when two records to be updated by two different processes are locked
- Self-deadlock
  - Attempt to obtain a "lock" by a process that is already owned by it
- Deadlocking by nefarious users
  - Given by R. C. Holt

```
void deadlock ( task )
{
    wait (event);
} /* deadlock */
```

- Effective Deadlocks
  - Exemplified by Shortest Job Next Scheduling
- Deadlocks problem characterization
  - Deadlock Detection
    - Process resource graphs
  - Deadlock Recovery
    - "Best" ways of recovering from a deadlock
  - Deadlock Prevention
    - Not allowing a deadlock to happen

A Systems Model

- Finite number of resources in the system to be distributed among a number of competing processes
- Partition the resources into several classes
- Identical resources assigned to the same class (CPU cycles, memory space, files, tape drives, printers)
- Allocation of any instance of resource from a class will satisfy the request
- State of the OS – allocation status of various resources
- Process actions
  - request a resource
    - request a device
    - open a file
    - allocate memory
  - acquire/use a resource
    - read from/write to a device
    - read/write a file
    - use the memory
  - release a resource
    - release a device
    - close a file
    - free memory
- Resources acquired and used only through system calls
- Allocation record to be maintained as a $system\ table$
- State of the OS changed only by the process actions
- Processes to be modeled as nondeterministic entities
- Deadlock when every process is waiting for an event that can be caused by only one of the waiting processes
- System $< \sigma, \pi >$
  - σ – $\{S, T, U, V, \ldots\}$ – system states
  - π – $\{p_1, p_2, \ldots\}$ – processes
- Process p_i – a partial function from system states into nonempty subsets of system states

$$p_i : \sigma \rightarrow \{\sigma\}$$

  $S \xrightarrow{*} W$ implies
  - $S = W$
  - $S \xrightarrow{i} W$ for some p_i
  - $S \xrightarrow{i} T$ for some p_i and T, and $T \xrightarrow{*} W$
- Process blocked if it cannot change state of the system

$$\nexists T | S \xrightarrow{i} T$$

- Process deadlocked in S if
  - Process is blocked in S

  - No operations can make the process to be *unblocked*

  p_i is deadlocked in S if

$$\forall T | S \xrightarrow{*} T$$

  p_i is blocked in T

- Deadlock state S if $\exists p_i$ deadlocked in S
- *Safe state* S if

$$\forall T | S \xrightarrow{*} T$$

  T is not a deadlock state

Deadlock Characterization

- Necessary conditions for deadlocks – Four conditions to hold simultaneously
  - Mutual exclusion – At least one resource must be held in a non-sharable mode
  - Hold and wait – Existence of a process holding at least one resource and waiting to acquire additional resources currently held by other processes
  - No preemption – Resources cannot be preempted by the system
  - Circular wait – Processes waiting for resources held by other waiting processes

Deadlock with Serially Reusable Resources

- *Serially reusable resource* – A finite set of identical units
  - The number of units is constant
  - Each unit is allocated to one and only one process
  - A process may release a unit only if it has previously acquired it

Deadlocks in Unix

- Possible deadlock condition that cannot be detected
- Number of processes limited by the number of available entries in the process table
- If process table is full, the `fork` system call fails
- Process can wait for a random amount of time before `fork`ing again
- Examples:
  - 10 processes creating 12 children each
  - 100 entries in the process table
  - Each process has already created 9 children
  - No more space in the process table $\Rightarrow$ deadlock
  - Deadlocks due to open files, swap space
- Another cause of deadlock can be due to the inode table becoming full in the filesystem

Resource Allocation Graph

- Directed graph to describe deadlocks

- Set of vertices V consisting of
  - $P = P_1, P_2, \ldots$ – Set of processes
  - Represent process nodes as circles
  - $R = R_1, R_2, \ldots$ – Set of resource types
  - Represent resource nodes as squares with a dot ($\cdot$) representing each instance of the resource
- Set of edges E
  - Directed edge from P_i to R_j
    - *request edge*
    - denoted by $P_i \rightarrow R_j$
    - P_i has requested for an instance of R_j and is currently waiting for that resource
  - Directed edge from R_j to P_i
    - *assignment edge*
    - denoted by $R_j \rightarrow P_i$
    - an instance of R_j has been allocated to P_i
- No cycles in the graph $\Rightarrow$ no deadlock
- Cycle in the graph $\Rightarrow$ deadlock
- Each process involved in a cycle is deadlocked
- Cycle in the resource graph is necessary and sufficient condition for the existence of a deadlock
- If a graph contains several instances of a resource type, a cycle is not a sufficient condition for a deadlock but it still is a necessary condition

Deadlock Detection

- Simulate the *most favored execution* of each unblocked process
  - An unblocked process may acquire all the needed resources
  - Run and then release *all* the acquired resources
  - Remain dormant thereafter
  - Released resources may wake up some previously blocked process
  - Continue the above steps as long as possible
  - If any blocked processes remain, they are deadlocked
- Reduction of resource graphs
  - Process blocked if it cannot progress by either of the following operations
    - Request
    - Acquisition
    - Release
  - Reduction of resource graph
    - Reduced by a process p_i
      - by removing all edges to and from p_i
      - p_i is neither blocked nor isolated node

      p_i becomes an isolated node
    - Irreducible if the graph cannot be reduced by any process
    - Completely reducible if a sequence of reductions deletes *all* the edges in the graph

  - **Lemma 1**. *All reduction sequences of a given resource graph lead to the same irreducible graph.*

- Algorithms for Deadlock Detection with SR Resources
  - **The Deadlock Theorem**. *S is a deadlock state if and only if the resource graph of S is not completely reducible.*
  - Representation of resource graph
    - Matrix representation – Two $n \times m$ matrices
      - Allocation matrix A – processes as rows and resources as columns
        $A_{ij}, i = 1, \ldots, n, j = 1, \ldots, m$ gives the number of units of resource R_j allocated to process p_i
      - Request matrix B – Similar to A
        B_{ij} gives the number of units of resource R_j requested by process p_i
    - Linked list structure – Four lists
      - Resources allocated to processes

        $$p_i \rightarrow (R_x, a_x) \rightarrow (R_y, a_y) \rightarrow \cdots \rightarrow (R_z, a_z)$$

      - Resources requested by processes
      - Allocation list of processes with respect to a resource
      - Request list of processes with respect to a resource
    - Available units vector $(r_1, \ldots r_m)$
  - Deadlocks detected by looping through the process request lists, making reductions where possible
  - Worst case execution time – mn^2
  - Algorithm `deadlock`

```
// Check if the request for process pnum is less than or equal to available
// vector

bool req_lt_avail ( const int * req, const int * avail, const int pnum, \
                    const int num_res )
{
    int i ( 0 );
    for ( ; i < num_res; i++ )
        if ( req[pnum*num_res+i] > avail[i] )
            break;
    return ( i == num_res );
}

bool deadlock ( const int * available, const int m, const int n, \
                const int * request, const int * allocated )
{
    int  work[m];           // m resources
    bool finish[n];         // n processes

    for ( int i ( 0 ); i < m; work[i] = available[i++] );
    for ( int i ( 0 ); i < n; finish[i++] = false );

    int p ( 0 );
    for ( ; p < n; p++ )   // For each process
    {
        if ( finish[p] ) continue;
        if ( req_lt_avail ( request, work, p, m ) )
        {
```

```
                finish[p] = true;
                for ( int i ( 0 ); i < m; i++ )
                    work[i] += allocated[p*m+i];
                p = 0;
            }
        }

        for ( p = 0; p < n; p++ )
            if ( ! finish[p] )
                break;

        return ( p != n );
    }
```

– Example

	Allocation			Request			Available		
	A	B	C	A	B	C	A	B	C
p_0	0	1	0	0	0	0	0	0	0
p_1	2	0	0	2	0	2			
p_2	3	0	3	0	0	0			
p_3	2	1	1	1	0	0			
p_4	0	0	2	0	0	2			

No deadlock with the sequence $< p_0, p_2, p_3, p_1, p_4 >$

– Consider that p_2 makes an additional request for an instance of type C

	Allocation			Request			Available		
	A	B	C	A	B	C	A	B	C
p_0	0	1	0	0	0	0	0	0	0
p_1	2	0	0	2	0	2			
p_2	3	0	3	0	0	1			
p_3	2	1	1	1	0	0			
p_4	0	0	2	0	0	2			

deadlock with processes $< p_1, p_2, p_3, p_4 >$

- `reach(a)` – Set of nodes in the graph reachable from `a`.
- **Theorem 2. The Cycle Theorem.** *A cycle in a resource graph is a necessary condition for deadlock.*
- **Theorem 3.** *If S is not a deadlock state and $S \xrightarrow{i} T$, then T is a deadlock state if and only if the operation by p_i is a request and p_i is deadlocked in T.*
- Special Cases of Resource Graphs
  - **Knot:** A knot in a directed graph $\langle N, E \rangle$ is a subset of nodes $M \subseteq N$ such that $\forall a \in M$, $\texttt{reach}(a) = M$
  - Immediate Allocation
    - Expedient States – All processes having requests are blocked
    - Expedient state $\Rightarrow$ A knot in the corresponding resource graph is a sufficient condition for deadlock
  - Single-Unit Resources – Cycle is sufficient and necessary condition for deadlock
- Recovery from Deadlock
  - Recovery by process termination
    - Terminate deadlocked processes in a systematic way
    - When enough processes terminated to recover from deadlock, stop terminations

    - Problems with the approach
        - If the process is in the midst of updating a file, its termination may leave the file in an incorrect state
        - If the process is in the midst of printing, the printer must be reset
    - Processes should be terminated based on some criterion/policy
        - Priority of a process
        - CPU time used and expected usage before completion
        - Number and type of resources being used (can they be preempted easily?)
        - Number of resources needed for completion
        - Number of processes needed to be terminated
        - Are the processes interactive or batch?
    - Minimum cost recovery
    - Cost of recovery
        - Cost of destroying a process
        - Cost of recovery from the next process state
  - Recovery by resource preemption
    - Enough resources to be preempted from processes and made available to deadlocked processes to resolve the deadlock
    - Selecting a victim
    - Rollback
    - Prevention of starvation – Ensure that the resources are not always preempted from the same process

- Deadlock Prevention
  - Each process must request and acquire all the needed resources at the same time
  - Deny one of the required conditions for a deadlock
    - Mutual Exclusion
        - Cannot be done for non-sharable resources (like printers)
        - Sharable resources (read-only files) do not require mutually exclusive access $\Rightarrow$ cannot be involved in deadlock
        - Cannot deny mutual exclusion as some resources are inherently non-sharable
    - Hold and Wait
        - Processes can request and acquire all the resources at one time
        - Request resources only if the process is holding none
          If the process is holding any resources, they must be released before requests can be granted
        - Disadvantages
          1. Low resource utilization – resources may get allocated but not used for a long time
          2. Possibility of starvation – on popular resources
    - No Preemption
        - If a process holding resources requests for another resource that cannot be immediately allocated, all currently held resources are preempted
        - Process restarted only when it regains all the resources
        - Suitable for resources whose state can be easily saved – CPU registers, memory
    - Circular Wait
        - Impose a total ordering on all resource types
        - Each process requests resources in an increasing order of enumeration
        - If several instances of a resource required, a single request must be issued for all of them

- Deadlock Prevention based on Maximum Claims

- Also called Deadlock Avoidance
- A priori knowledge of maximum possible claims for each process
- Dynamically examine the resource allocation status to ensure that no circular wait condition can exist
- Resource allocation state
  - Defined by the number of available and allocated resources, and the maximum demands of the processes
  - Safe, if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock
- System in safe state only if there exists a *safe sequence*
- All unsafe states are not deadlock states
- An unsafe state may lead to a deadlock
- Example
  - System with 12 magnetic tape drives

Process	Max needs	Allocation	Current needs
p_0	10	5	5
p_1	4	2	2
p_2	9	2	7

Current availability : 3

Safe sequence: $\langle p_1, p_0, p_2 \rangle$

  - Possible to go from a safe state to an unsafe state
    Let the state after allocating two tapes to process p_1 be

System with 12 magnetic tape drives

Process	Max needs	Allocation	Current needs
p_0	10	5	5
p_1	4	4	0
p_2	9	2	7

Current availability : 1

    Let p_2 request and acquire the last remaining tape drive
  - Mistake in allocating one more tape drive to p_2
- Problem: To detect the possibility of unsafe state and deny requests even if resources are still available
- Banker's Algorithm
  - Based on banking system that never allocates its available cash such that it can no longer satisfy the needs of all its customers

- Deadlock Avoidance
  - Requires a process to declare the maximum instances of each resource type needed
  - Upon request, the system must determine whether the allocation will leave the system in a safe state
  - Number of processes in the system – n
  - Number of resource classes – m
  - Data structures
    - `available`
      - A vector of length m
      - Number of available resources of each type
      - `available[j] = k` $\Rightarrow$ k instances of resource class R_j are available
    - `maximum`
      - An $n \times m$ matrix
      - Defines maximum demand for each process
      - `maximum[i,j] = k` $\Rightarrow$ process p_i may request at most k instances of resource class R_j
    - `allocation`

        - An $n \times m$ matrix
        - Defines the number of resources of each type currently allocated to each process
        - `allocation[i,j] = k` ⇒ process p_i is currently allocated k instances of resource class R_j
    * `need`
        - An $n \times m$ matrix
        - Indicates the remaining resource need of each process
        - `need[i,j] =k` ⇒ process p_i may need k more instances of resource type R_j in order to complete its task
        - `need[i,j] = maximum[i,j] - allocation[i,j]`
- Banker's Algorithm
    * $\texttt{request}_i$
        - Request vector for process p_i
        - $\texttt{request}_i\texttt{[j] = k}$ ⇒ process p_i wants k instances of resource class R_j
    * Upon request for resources, the following actions are taken

```
if requesti > needi then
   raise error condition
else
   if requesti ≤ available then
   {  available -= requesti
      allocationi += requesti
      needi -= requesti
   }
   else
      wait (pi)
```

    * If resulting resource-allocation state is safe, transaction is completed and process p_i is allocated its resources
    * If the new state is unsafe, p_i must wait for $\texttt{request}_i$ and the old allocation state is restored
- Safety Algorithm
    * Finds out whether or not a system is in a safe state

```
var
   work :  integer vector [1..m]
   finish :  boolean vector [1..n]

   work = available
   for ( i = 1; i < n; i++ )
      finish[i] = false;
x :find an i such that
      {  finish[i] == false
         needi ≤ work
      }
   if there is no such i then
      {  if finish[i] == true for all i then
            system is in a safe state
      }
   else
      {  work += allocationi
         finish[i] = true
         go to x
      }
```

- Example
    * System with five processes

	Allocation			Maximum			Available		
	A	B	C	A	B	C	A	B	C
p_0	0	1	0	7	5	3	3	3	2
p_1	2	0	0	3	2	2			
p_2	3	0	2	9	0	2			
p_3	2	1	1	2	2	2			
p_4	0	0	2	4	3	3			

* Matrix `need`

	Need		
	A	B	C
p_0	7	4	3
p_1	1	2	2
p_2	6	0	0
p_3	0	1	1
p_4	4	3	1

* Sequence $\langle p_1, p_3, p_4, p_2, p_0 \rangle$ satisfies the safety criterion
* Let process p_1 request one additional instance of resource class A and two additional instances of resource class C

$$\text{request}_1 = (1, 0, 2)$$

* $\text{request}_1 \leq$ available is true
* New state

	Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C
p_0	0	1	0	7	4	3	2	3	0
p_1	3	0	2	0	2	0			
p_2	3	0	2	6	0	0			
p_3	2	1	1	0	1	1			
p_4	0	0	2	4	3	1			

* Sequence $\langle p_1, p_3, p_4, p_0, p_2 \rangle$ satisfies the safety criterion
* Request for (3, 3, 0) by p_4 cannot be granted

Memory Management

"Programs expand to fill the memory that holds them."

Preparing a program for execution

- Development of programs
  - Source program
  - Compilation/Assembly
  - Object program
  - Linking / Linkage editors
  - Loading
  - Memory
    - Large array of words (or bytes)
    - Unique address of each word
    - CPU fetches from and stores into memory addresses
  - Instruction execution cycle
    - Fetch an instruction (opcode) from memory
    - Decode instruction
    - Fetch operands from memory, if needed
    - Execute instruction
    - Store results into memory, if necessary
  - Memory unit sees only the addresses, and not how they are generated (instruction counter, indexing, direct)
- Address Binding
  - Binding – Mapping from one address space to another
  - Program must be loaded into memory before execution
  - Before being loaded in memory, object module resides on disk
  - Input queue
  - Loading of processes may result in relocation of addresses
    - Link external references to entry points as needed
  - User process may reside in any part of the memory
  - Symbolic addresses in source programs (like `i`)
  - Compiler $binds$ symbolic addresses to relocatable addresses
  - Linkage editor or loader $binds$ relocatable addresses to absolute addresses
  - Types of binding
    - Compile time binding
      - Binding of absolute addresses by compiler
      - Possible only if compiler knows the memory locations to be used
      - MS-DOS .com format programs
    - Load time binding
      - Based on relocatable code generated by the compiler
      - Final binding delayed until load time
      - If change in starting address, reload the user code to incorporate address changes
    - Execution time binding

            - Process may be moved from one address to another during execution
            - Binding must be delayed until run time
            - Requires special hardware

- Relocation
    - Compiler may work with *assumed* logical address space when creating an object module
    - Relocation – Adjustment of operand and branch addresses within the program
    - Static Relocation
        - Similar to compile time binding
        - Internal references
            - References to locations within the same program address space
            - Earliest possible moment to perform binding at the time of compilation
            - If bound at compile time, compiler must have the actual starting address of object module
            - Early binding is restrictive and rarely used
        - External references
            - References to locations within the address space of other programs
            - More practical
            - All modules to be linked together must be known to resolve references
            - Linking loader
        - Separate linkage editor and loader
            - More flexible
            - Starting address need not be known at linking time
            - Relocatable physical addresses bound by relocating the complete module
    - Dynamic Relocation
        - All object modules kept on disk in relocatable load format
        - Relocation at runtime precedes *each* storage reference
        - Invisible to all users (except for system programmers)
        - Also called *virtual memory*
        - Permits efficient use of main storage
        - Binding of physical addresses can be delayed to the last possible moment
        - When a routine needs to call another routine, the calling routine first checks to see whether the other routine has been loaded into memory
        - Relocatable linking loader can load the new routine if needed
        - Unused routine is never loaded
        - Useful to handle large amount of infrequently used code (like error handlers)

- Linking
    - Allows independent development and translation of modules
    - Compiler generates *external symbol table*
    - Resolution of external references
        - Chain method
            - Using chain of pointers in the module
            - Resolution by linking at the end through external symbol table
            - External symbol table not a part of the final code
        - Indirect addressing
            - External symbol table a permanent part of the program

            - Transfer vector
            - External symbol table reflects the actual address of the reference when known
    - *Static Linking*
        - All external references are resolved before program execution
    - *Dynamic Linking*
        - External references resolved during execution
        - *Dynamically linked libraries*
            - Particularly useful for system libraries
            - Programs need to have a copy of the language library in the executable image, if no dynamic linking
            - Include a *stub* in the image for each library-routine reference
            - Stub

              Small piece of code

              Indicates the procedures to locate the appropriate memory resident library routine

              Upon execution, stub replaces itself with the routine and executes it

              Repeated execution executes the library routine directly

              Useful in library updates or bug fixes

              A new version of the library does not require the programs to be relinked

              Library version numbers to ensure compatibility

Implementation of memory management

- Done through *memory tables* to keep track of both real (main) as well as virtual (secondary) memory
- Memory management unit (MMU)
    - Identifies a memory location in ROM, RAM, or I/O memory given a physical address
    - Does *not* translate physical address
    - Physical address is provided by an address bus to initiate the movement of code or data from one platform device to another
    - Physical address is generated by devices that act as *bus masters* on the address buses (such as CPU)
    - Frame buffers and simple serial ports are *slave devices* that respond to the addresses

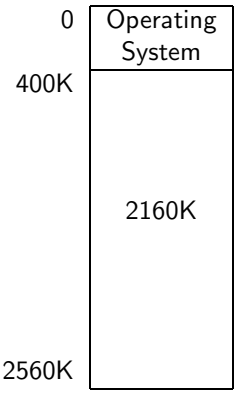
Simple Memory Management Schemes

- Shortage of main memory due to
    - Size of many applications
    - Several active processes may need to share memory at the same time
- Fixed Partition Memory Management
    - Simplest memory management scheme for multiprogrammed systems
    - Divide memory into fixed size *partitions*
    - Partitions fixed at system initialization time and may not be changed during system operation
    - Single-Partition Allocation
        - User is provided with a bare machine
        - User has full control of entire memory space
        - Advantages
            - Maximum flexibility to the user

        - User controls the use of memory as per his own desire
        - Maximum possible simplicity
        - Minimum cost
        - No need for special hardware
        - No need for operating system software
      * Disadvantages
        - No services
        - OS has no control over interrupts
        - No mechanism to process system calls and errors
        - No space to provide multiprogramming
  - Two-Partition Allocation
    * Memory divided into two partitions
      - Resident operating system
      - User memory area
    * OS placed in low memory or high memory depending upon the location of interrupt vector
    * Need to protect OS code and data from changes by user processes
      - Protection must be provided by hardware
      - Can be implemented by using base-register and limit-register
    * Loading of user processes
      - First address of user space must be beyond the base register
      - Any change in base address requires recompilation of code
      - Could be avoided by having relocatable code from the compiler
      - Base value must be *static* during program execution
      - OS size cannot change during program execution
        Change in buffer space for device drivers
        Loading code for rarely used system calls
        Transient OS code
    * Handling transient code
      - Load user processes into high memory down to base register
        Allows the use of all available memory
      - Delay address binding until execution time
        Base register known as the *relocation register*
        Value in base register added to every address reference
        User program never sees the real physical addresses
        User program deals only with logical addresses

- Multiple-Partition Allocation
  - Necessary for multiprogrammed systems
  - Allocate memory to various processes in the wait queue to be brought into memory
  - Simplest scheme
    * Divide memory into a large number of fixed-size partitions
    * One process to each partition
    * Degree of multiprogramming bound by the number of partitions
    * Partitions allocated to processes and released upon process termination
    * Originally used by IBM OS/360 (MFT)
    * Primarily useful in a batch environment

- Variable size partitions – Basic implementation
  - Keep a table indicating the availability of various memory partitions
  - Any large block of available memory is called a *hole*
  - Initially the entire memory is identified as a large hole
  - When a process arrives, the allocation table is searched for a large enough hole and if available, the hole is allocated to the process
  - Example
    - Total memory available – 2560K
    - Resident OS – 400K
    - User memory – 2160K



Job Queue

Process	Memory	Time
p_1	600K	10
p_2	1000K	5
p_3	300K	20
p_4	700K	8
p_5	500K	15

- Set of holes of various sizes scattered throughout the memory
- Holes can grow when jobs in adjacent holes are terminated
- Holes can also diminish in size if many small jobs are present
- Problem of *fragmentation*
  - Division of main memory into small holes not usable by any process
  - Enough total memory space exists to satisfy a request but is fragmented into a number of small holes
  - Possibility of starvation for large jobs
- Used by IBM OS/MVT (multiprogramming with variable number of tasks, 1969)
- Dynamic storage allocation problem
  - Selects a hole to which a process should be allocated
  - First-fit strategy
    - Allocate first hole that is big enough
    - Stop searching as soon as first hole large enough to hold the process is found
  - Best-fit strategy
    - Allocate the smallest hole that is big enough
    - Entire list of holes is to be searched
    - Search of entire list can be avoided by keeping the list of holes sorted by size
  - Worst-fit strategy
    - Allocate the largest available hole
    - Similar problems as the best-fit approach
- Memory Compaction
  - Shuffle the memory contents to place all free memory into one large hole
  - Possible only if the system supports dynamic relocation at execution time
  - Total compaction
  - Partial compaction

    - Dynamic memory allocation in C
        - C heap manager is fairly primitive
        - The `malloc` family of functions allocates memory and the heap manager takes it back when it is `freed`
        - There is no facility for heap compaction to provide for bigger chunks of memory
        - The problem of fragmentation is for real in C because movement of data by a heap compactor can leave incorrect address information in pointers
        - Microsoft Windows has heap compaction built in but it requires you to use special memory handles instead of pointers
        - The handles can be temporarily converted to pointers, after locking the memory so the heap compactor cannot move it

- Overlays
    - Size of process is limited to size of available memory
    - Technique of *overlaying* employed to execute programs that cannot be fit into available memory
    - Keep in memory only those instructions and data that are needed at any given time
    - When other instructions are needed, they are loaded into space previously occupied by instructions that are not needed
    - A 2-pass assembler

Pass 1	70K	Generate symbol table
Pass 2	80K	Generate object code
Symbol table	20K	
Common routines	30K	

Total memory requirement – 200K
Available memory – 150K

    - Divide the task as into overlay segments

Overlay 1	Overlay 2
Pass 1 code	Pass 2 code
Symbol table	Symbol table
Common routines	Common routines
Overlay driver (10K)	

    - Code for each overlay kept on disk as absolute memory images
    - Requires special relocation and linking algorithms
    - No special support required from the OS
    - Slow to execute due to additional I/O to load memory images for different parts of the program
    - Programmer completely responsible to define overlays

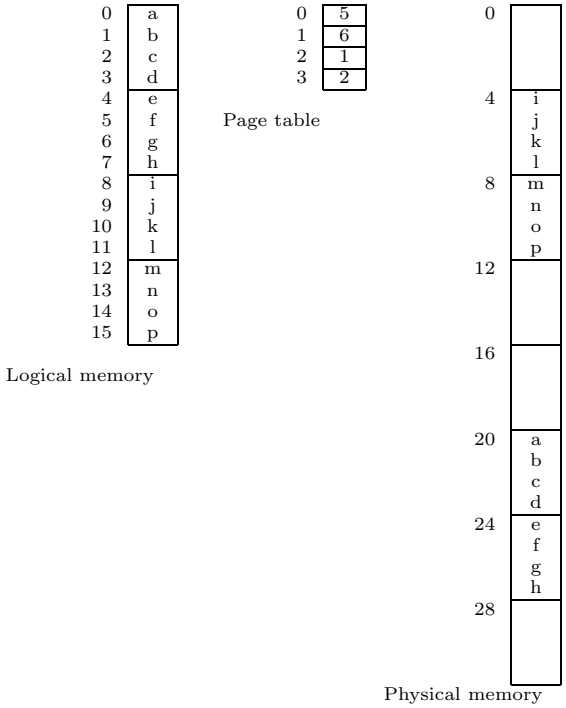
Principles of Virtual Memory

- Hide the real memory from the user
- Swapping
    - Remove a process temporarily from main memory to a *backing store* and later, bring it back into memory for continued execution
    - Swap-in and Swap-out
    - Suitable for round-robin scheduling by swapping processes in and out of main memory
        - Quantum should be large enough such that swapping time is negligible
        - Reasonable amount of computation should be done between swaps

  - *Roll-out, Roll-in Swapping*
    * Allows to preempt a lower priority process in favor of a higher priority process
    * After the higher priority process is done, the preempted process is continued
  - Process may or may not be swapped into the same memory space depending upon the availability of execution time binding
  - Backing store
    * Preferably a fast disk
    * Must be large enough to accommodate copies of all memory images of all processes
    * Must provide direct access to each memory image
    * Maintain a ready queue of all processes on the backing store
    * Dispatcher brings the process into memory if needed
  - Calculation of swap time
    * User process of size 100K
    * Backing store – Standard head disk with a transfer rate of 1 MB/sec
    * Actual transfer time – 100 msec
    * Add latency (8 msec) – 108 msec
    * Swap-in + Swap-out – 216 msec
  - Total transfer time directly proportional to the amount of memory swapped
  - Swap only completely idle processes (not with pending I/O)

- Multiple Base Registers
  - Provides a solution to the fragmentation problem
  - Break the memory needed by a process into several parts
  - One base register corresponding to each part with a mechanism to translate from logical to physical address

- Paging
  - All segments of the same size
  - Permits a process's memory to be noncontiguous
  - Avoids the problem of fitting varying-sized memory segments into backing store
  - Hardware requirements
    * Physical memory broken into fixed size blocks called $frames$
    * Logical memory broken into blocks of same size called $pages$
    * To execute, pages of process are loaded into frames from backing store
    * Backing store divided into fixed size blocks of the same size as page or frame
    * Every address generated by the CPU divided into two parts
      · Page number p
      · Page offset d
    * Page number used as index into a $page\ table$
      · Page table contains the base address of each page in memory
    * Page offset defines the address of the location within the page
    * Page size defined by hardware (typically between 2^9 to 2^{11})
    * Page size 2^n bytes
      · Low order n bits in the address indicate the page offset
      · Remaining high order bits designate the page number
    * Example – Page size of four words and physical memory of eight pages



- Scheduling processes in a paged system
  - Each page an instance of memory resource
  - Size of a process can be expressed in pages
  - Available memory known from the list of unallocated frames
  - If the process's memory requirement can be fulfilled, allocate memory to the process
  - Pages loaded into memory from the list of available frames
  - Example

free-frame list			free-frame list		
14	13	unused	15	13	page 1
13	14	unused		14	page 0
18	15	unused		15	unused
20	16			16	
15	17			17	
	18	unused		18	page 2
new process	19		new process	19	
page 0	20	unused	page 0	20	page 3
page 1	21		page 1	21	
page 2			page 2		
page 3			page 3		

new process page table

0	14
1	13
2	18
3	20

  - No external fragmentation possible with paging
  - Internal fragmentation possible – an average of half a page per process
  - Page size considerations

    - * Small page size ⇒ More overhead in page table plus more swapping overhead
    - * Large page size ⇒ More internal fragmentation
  - Implementation of page table
    - * Simplest implementation through a set of dedicated registers
    - * Registers reloaded by the CPU dispatcher
    - * Registers need to be extremely fast
    - * Good strategy if the page table is very small (< 256 entries)
    - * Not satisfactory if the page table size is large (like a million entries)
    - * *Page-table Base Register*
      - · Suitable for large page tables
      - · Pointer to the page table in memory
      - · Achieves reduction in context switching time
      - · Increase in time to access memory locations
    - * *Associative registers*
      - · Also called *translation look-aside buffers*
      - · Two parts to each register
        1. key
        2. value
      - · All keys compared simultaneously with the key being searched for
      - · Expensive hardware
      - · Limited part of page table kept in associative memory
      - · *Hit ratio* – Percentage of time a page number is found in the associative registers
      - · Effective memory access time

- Shared pages
  - Possible to share common code with paging
  - Shared code must be reentrant (pure code)
    - * Reentrant code allows itself to be shared by multiple users concurrently
    - * The code cannot modify itself and the local data for each user is kept in separate space
    - * The code has two parts
      1. Permanent part is the instructions that make up the code
      2. Temporary part contains memory for local variables for use by the code
    - * Each execution of the permanent part creates a temporary part, known as the *activation record* for the code
  - Separate data pages for each process
  - Code to be shared – text editor, windowing system, compilers

- Memory protection in paging systems
  - Accomplished by protection bits associated with each frame
  - Protection bits kept in page table
  - Define the page to be read only or read and write
  - Protection checked at the time of page table reference to find the physical page number
  - Hardware trap or memory protection violation
  - *Page table length register*
    - * Indicates the size of the page table
    - * Value checked against every logical address to validate the address

        * Failure results in trap to the OS

- Logical memory vs Physical memory
  - Logical memory
    * Provides user's view of the memory
    * Memory treated as one contiguous space, containing only one program
  - Physical memory
    * User program $scattered$ throughout the memory
    * Also holds other programs
  - Mapping from logical addresses to physical addresses hidden from the user
  - System could use more memory than any individual user
  - Allocation of frames kept in $frame\ table$

- Segmentation
  - User prefers to view memory as a collection of variable-sized segments, like arrays, functions, procedures, and main program
  - No necessary order in the segments
  - Length of each segment is defined by its purpose in the program
  - Elements within a segment defined by their offset from the beginning of segment
    * First statement of the procedure
    * Seventeenth entry in the symbol table
    * Fifth instruction of the `sqrt` function
  - Logical address space considered to be collection of segments
  - A name and length for each segment
  - Address – Segment name and offset within the segment
    * Segment name to be explicitly specified unlike paging
  - The only memory management scheme available on Intel 8086
    * Memory divided into $code$, $data$, and $stack$ segments
  - Hardware for segmentation
    * Mapping between logical and physical addresses achieved through a $segment\ table$
    * Each entry in segment table is made up of
      · Segment $base$
      · Segment $limit$
    * Segment table can be abstracted as an array of $base\text{-}limit\ register\ pairs$
    * Two parts in a logical address
      1. Segment name/number s
         · Used as an index into the segment table
      2. Segment offset d
         · Added to the segment base to produce the physical address
         · Must be between 0 and the segment limit
         · Attempt to address beyond the segment limit results in a trap to the OS
  - Implementation of segment tables
    * Kept either in registers or in memory
    * Segment table base register
    * Segment table length register

        * Associative registers to improve memory access time
    - Protection and sharing
        * Segments represent a semantically defined portion of a program
        * Protection and sharing like paging
        * Possible to share parts of a program
            · Share the `sqrt` function segment between two independent programs
    - Fragmentation
        * Memory allocation becomes a dynamic storage allocation problem
        * Possibility of external fragmentation
            · All blocks of memory are too small to accommodate a segment
        * Compaction can be used whenever needed (because segmentation is based on dynamic relocation)
        * External fragmentation problem is also dependent on average size of segments

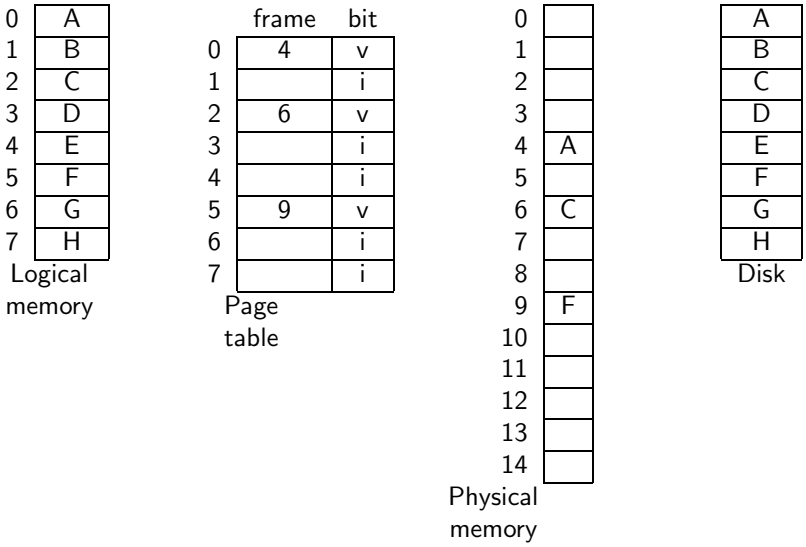
- Paged Segmentation
    - Used in the MULTICS system
    - Page the segments
    - Separate page table for each segment
    - Segment table entry contains the base address of a page table for the segment
    - Segment offset is broken into page number and page offset
    - Page number indexes into page table to give the frame number
    - Frame number is combined with page offset to give physical address
    - MULTICS had 18 bit segment number and 16 bit offset
    - Segment offset contained 6-bit page number and 10-bit page offset
    - Each segment limited in length by its segment-table entry
      Therefore page table need not be full-sized
      It requires only as many entries as needed
    - On an average, half a page of internal fragmentation per segment
    - Eliminated external fragmentation but introduced internal fragmentation and increased table space overhead

Implementation of Virtual Memory

- Allows the execution of processes that may not be completely in main memory
- Programs can be larger than the available physical memory
- Motivation
    - The entire program may not need to be in the memory for execution
    - Code to handle unusual conditions may never be executed
    - Complex data structures are generally allocated more memory than needed (like symbol table)
    - Certain options and features of a program may be used rarely, like text-editor command to change case of all letters in a file
- Benefits of virtual memory
    - Programs not constrained by the amount of physical memory available
    - User does not have to worry about complex techniques like overlays
    - Increase in CPU utilization and throughput (more programs can fit in available memory)

  - Less I/O needed to load or swap user programs

- *Demand Paging*
  - Similar to a paging system with swapping
  - Rather than swapping the entire process into memory, use a "lazy swapper"
  - Never swap a page into memory unless needed
  - Distinction between *swapper* and *pager*
    * Swapper swaps an entire process into memory
    * Pager brings in individual pages into memory
  - In the beginning, pager guesses the pages to be used by the process
  - Pages not needed are not brought into memory
  - Hardware support for demand paging
    * An extra bit attached to each entry in the page table – *valid-invalid bit*
    * This bit indicates whether the page is in memory or not
    * Example



  - For *memory-resident pages*, execution proceeds normally
  - If page not in memory, a *page fault trap* occurs
  - Upon page fault, the required page brought into memory
    * Check an internal table to determine whether the reference was valid or invalid memory access
    * If invalid, terminate the process. If valid, page in the required page
    * Find a free frame (from the free frame list)
    * Schedule the disk to read the required page into the newly allocated frame
    * Modify the internal table to indicate that the page is in memory
    * Restart the instruction interrupted by page fault
  - *Pure demand paging* – Don't bring even a single page into memory
  - *Locality of reference*

- Performance of demand paging
  - Effective access time for demand-paged memory
    * Usual memory access time (m) – 10 to 200 nsec

    - No page faults ⇒ Effective access time same as memory access time
    - Probability of page fault $= p$
    - p expected to be very close to zero so that there are few page faults
    - Effective access time $= (1 - p) \times m + p \times$ page fault time
    - Need to know the time required to service page fault

- What happens at page fault?
  - Trap to the OS
  - Save the user registers and process state
  - Determine that the interrupt was a page fault
  - Check that the page reference was legal and determine the location of page on the disk
  - Issue a read from the disk to a free frame
    - Wait in a queue for the device until the request is serviced
    - Wait for the device seek and/or latency time
    - Begin the transfer of the page to a free frame
  - While waiting, allocate CPU to some other process
  - Interrupt from the disk (I/O complete)
  - Save registers and process state for the other user
  - Determine that the interrupt was from the disk
  - Correct the page table and other tables to show that the desired page is now in memory
  - Wait for the CPU to be allocated to the process again
  - Restore user registers, process state, and new page table, then resume the interrupted instruction
- Computation of effective access time
  - Bottleneck in read from disk
    - Latency time – 8 msec
    - Seek time – 15 msec
    - Transfer time – 1 msec
    - Total page read time – 24 msec
  - About 1 msec for other things (page switch)
  - Average page-fault service time – 25 msec
  - Memory access time – 100 nanosec
  - Effective access time
    $$\begin{aligned} &= (1 - p) \times 100 + p \times 25,000,000 \\ &= 100 + 24,999,900 \times p \\ &\approx 25 \times p\text{msec} \end{aligned}$$
  - Assume 1 access out of 1000 to cause a page fault
    - Effective access time – 25 μsec
    - Degradation due to demand paging – 250%
  - For 10% degradation
    $$\begin{aligned} 110 &> 100 + 25,000,000 \times p \\ 10 &> 25,000,000 \times p \\ p &< 0.0000004 \end{aligned}$$
    Reasonable performance possible through less than 1 memory access out of 2,500,000 causing a page fault

Page Replacement

- Limited number of pages available in memory
- Need to optimize swapping (placement and replacement)
- Increase in multiprogramming through replacement optimization
- Increase in degree of multiprogramming ⇒ overallocation of memory
- Assume no free frames available
  - Option to terminate the process
    - Against the philosophy behind virtual memory
    - User should not be aware of the underlying memory management
  - Option to swap out a process
    - No guarantee that the process will get the CPU back pretty fast
  - Option to replace a page in memory
- Modified page-fault service routine
  - Find the location of the desired page on the disk
  - Find a free frame
    - If there is a free frame, use it
    - Otherwise, use a *page replacement algorithm* to find a *victim* frame
    - Write the victim page to the disk; change the page and frame tables accordingly
  - Read the desired page into the (newly) free frame; change the page and frame tables
  - Restart the user process
- No free frames ⇒ two page transfers
- Increase in effective access time
- *Dirty bit*
  - Also known as *modify bit*
  - Each frame has a dirty bit associated with it in hardware
  - Dirty bit is set if the page has been modified or written into
  - If the page is selected for replacement
    - Check the dirty bit associated with the frame
    - If the bit is set write it back into its place on disk
    - Otherwise, the page in disk is same as the current one
- Page replacement algorithms
  - Aim – to minimize the page fault rate
  - Evaluate an algorithm by running it on a particular string of memory references and compute the number of page faults
  - Memory references string called a *reference string*
  - Consider only the page number and not the entire address
  - Address sequence

0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101,
0611, 0102, 0103, 0104, 0101, 0610, 0102, 0103,
0104, 0101, 0609, 0102, 0105

- 100 byte to a page
- Reference string

  1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1

- Second factor in page faults – Number of pages available
- More the number of pages, less the page faults
- FIFO Replacement Algorithm
  - Associate with each page the time when that page was brought in memory
  - The victim is the oldest page in the memory
  - Example reference string

    $$7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1$$

  - With three pages, page faults as follows:

7	7	7	2	2	2	4	4	4	0	0	0	7	7	7
	0	0	0	3	3	3	2	2	2	1	1	1	0	0
		1	1	1	0	0	0	3	3	3	2	2	2	1

  - May reduce a page that contains a heavily used variable that was initialized a while back
  - Bad replacement choice ⇒ Increase in page fault rate
  - Consider another reference string

    $$1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5$$

  - With three pages, the page faults are:

1	1	1	4	4	4	5	5	5
	2	2	2	1	1	1	3	3
		3	3	3	2	2	2	4

  - With four pages, the page faults are:

1	1	1	1	5	5	5	5	4	4
	2	2	2	2	1	1	1	1	5
		3	3	3	3	2	2	2	2
			4	4	4	4	3	3	3

  - *Belady's Anomaly* – For some page replacement algorithms, the page fault rate may increase as the number of allocated frames increases
- Optimal Page Replacement Algorithm
  - Also called OPT or MIN
  - Has the lowest page-fault rate of all algorithms
  - Never suffers from Belady's Anomaly
  - "Replace the page that will not be used for the longest period of time"
  - Example reference string

    $$7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1$$

  - With three pages, page faults as follows:

7	7	7	2	2	2	2	2	7
	0	0	0	0	4	0	0	0
		1	1	3	3	3	1	1

  - Guarantees the lowest possible page-fault rate of all algorithms
  - Requires future knowledge of page references
  - Mainly useful for comparative studies with other algorithms

  - LRU Page Replacement Algorithm
    * Approximation to the optimal page replacement algorithm
    * Replace the page that has not been used for the longest period of time
    * Example reference string

$$7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1$$

    * With three pages, page faults as follows:

7	7	7	2	2	4	4	4	0	1	1	1
	0	0	0	0	0	0	3	3	3	0	0
		1	1	3	3	2	2	2	2	2	7

    * Implementation may require substantial hardware assistance
    * Problem to determine an order for the frame defined by the time of last use
    * Implementations based on $counters$
      · With each page-table entry, associate a time-of-use register
      · Add a logical clock or counter to the CPU
      · Increment the clock for every memory reference
      · Copy the contents of logical counter to the time-of-use register at every page reference
      · Replace the page with the smallest time value
      · Times must be maintained when page tables are changed
      · Overflow of the clock must be considered
    * Implementations based on $stack$
      · Keep a stack of page numbers
      · Upon reference, remove the page from stack and put it on top of stack
      · Best implemented by a doubly linked list
      · Each update expensive but no cost for search (for replacement)
      · Particularly appropriate for software or microcode implementations

- Stack algorithms
  - Class of page replacement algorithms that never exhibit Belady's anomaly
  - Set of pages in memory for n frames is always a $subset$ of the set of pages in memory with $n + 1$ frames
- LRU Approximation Algorithms
  - Associate a $reference\ bit$ with each entry in the page table
  - Reference bit set whenever the page is referenced (read or write)
  - Initially, all bits are reset
  - After some time, determine the usage of pages by examining the reference bit
  - No knowledge of order of use
  - Provides basis for many page-replacement algorithms that approximate replacement
  - Additional-Reference-Bits Algorithm
    * Keep an 8-bit byte for each page in a table in memory
    * At regular intervals (100 msec), shift the reference bits by 1 bit to the right
    * 8-bit shift-registers contain the history of page reference for the last eight time periods
    * Page with lowest number is the LRU page
    * 11000100 used more recently than 01110111
    * Numbers not guaranteed to be unique
  - Second-Chance Algorithm
    * Basically a FIFO replacement algorithm

    - When a page is selected, examine its reference bit
    - If reference bit is 0, replace the page
    - If reference bit is 1, give the page a second chance and proceed to select the next FIFO page
    - To give second chance, reset the reference bit and set the page arrival time to current time
    - A frequently used page is always kept in memory
    - Commonly implemented by a circular queue
    - Worst case when all bits are set (degenerates to FIFO replacement)
  - LFU Algorithm
    - Keep counter of number of references made to each page
    - Replace the page with the smallest count
    - Motivation – Actively used page has a large reference count
    - What if a page is heavily used initially but never used again
    - Solution – Shift the counts right at regular intervals forming a decaying average usage count
  - MFU Algorithm
    - Page with smallest count was probably just brought into memory and is yet to be used
    - Implementation of LFU and MFU fairly expensive, and they do not approximate OPT very well
  - Using $used\ bit$ and $dirty\ bit$ in tandem
    - Four cases
      - (0,0) neither used nor modified
      - (0,1) not used (recently) but modified
      - (1,0) used but clean
      - (1,1) used and modified
    - Replace a page within the lowest class

Allocation of Frames

- No problem with the single user virtual memory systems
- Problem when demand paging combined with multiprogramming
- Minimum number of frames
  - Cannot allocate more than the number of available frames (unless page sharing is allowed)
  - As the number of frames allocated to a process decreases, page fault rate increases, slowing process execution
  - Minimum number of frames to be allocated defined by instruction set architecture
    - Must have enough frames to hold all the different pages that any single instruction can reference
    - The instruction itself may go into two separate pages
  - Maximum number of frames defined by amount of available physical memory
- Allocation algorithms
  - Equal allocation
    - m frames and n processes
    - Allocate m/n frames to each process
    - Any leftover frames given to $free\ frame\ buffer\ pool$
  - Proportional allocation
    - Allocate memory to each process according to its size
    - If size of virtual memory for process p_i is s_i, the total memory is given by $S = \sum s_i$

    - Total number of available frames – m
    - Allocate a_i frames to process p_i where
    $$a_i = \frac{s_i}{S} \times m$$
    - a_i must be adjusted to an integer, greater than the minimum number of frames required by the instruction set architecture, with a sum not exceeding m
    - Split 62 frames between two processes – one of 10 pages and the other of 127 pages
    - First process gets 4 frames and the second gets 57 frames
  - Allocation dependent on degree of multiprogramming
  - No consideration for the priority of the processes

Thrashing

- Number of frames allocated to a process falls below the minimum level ⇒ Suspend the process's execution
- Technically possible to reduce the allocated frames to a minimum for a process
- Practically, the process may require a higher number of frames than minimum for effective execution
- If process does not get enough frames, it page-faults quickly and frequently
- Need to replace a page that may be in active use
- Thrashing – High paging activity
- Process thrashing if it spends more time in paging activity than in actual execution
- Cause of thrashing
  - OS monitors CPU utilization
  - Low CPU utilization ⇒ increase the degree of multiprogramming by introducing new process
  - Use a *global page replacement algorithm*
  - May result in increase in paging activity and thrashing
  - Processes waiting for pages to arrive leave the ready queue and join the wait queue
  - CPU utilization drops
  - CPU scheduler sees the decrease in CPU utilization and increases the degree of multiprogramming
  - Decrease in system throughput
  - At a certain point, to increase the CPU utilization and stop thrashing, we must decrease the degree of multiprogramming
  - Effect of thrashing can be reduced by *local replacement algorithms*
    - A thrashing process cannot steal frames from another process
    - Thrashing processes will be in queue for paging device for more time
    - Average service time for a page fault will increase
    - Effective access time will increase even for processes that are not thrashing
  - Locality model of process execution
    - Technique to guess the number of frames *needed* by a process
    - As a process executes, it moves from locality to locality
    - Locality – Set of pages that are generally used together
  - Allocate enough frames to a process to accommodate its current locality
- Working-Set Model

  - Based on the presumption of locality
  - Uses a parameter Δ to define *working-set window*
  - Set of pages in the most recent Δ page reference is the working set
  - Storage management strategy
    * At each reference, the current working set is determined and only those pages belonging to the working set are retained
    * A program may run if and only if its entire current working set is in memory
  - Actively used page $\in$ working set
  - Page not in use drops out after Δ time units of non-reference
  - Example
    * Memory reference string

      ...
      2 6 1 5 7 7 7 7 5 1 Δ_{t_1}
      6 2 3 4 1 2
      3 4 4 4 3 4 4 4 4 Δ_{t_2}
      1 3 2 3 4 4 4 3 4 4 4
      ...

    * If $\Delta = 10$ memory references, the working set at time t_1 is $\{1, 2, 5, 6, 7\}$ and at time t_2, it is $\{3, 4\}$
  - Accuracy of working set dependent upon the size of Δ
  - Let WSS_i be the working set size for process p_i
  - Then, the total demand for frames D is given by $\sum \text{WSS}_i$
  - Thrashing occurs if $D > m$
  - OS monitors the working set of each process
    * Allocate to each process enough frames to accommodate its working set
    * Enough extra frames $\Rightarrow$ initiate another process
    * $D > m \Rightarrow$ select a process to suspend
  - Working set strategy prevents thrashing while keeping the degree of multiprogramming high
  - Optimizes CPU utilization
  - Problem in keeping track of the working set
    * Can be solved by using a timer interrupt and a reference bit
    * Upon timer interrupt, copy and clear the reference bit value for each page

- Page-fault frequency
  - More direct approach than working set model
  - Measures the time interval between successive page faults
  - Page fault rate for a process too high $\Rightarrow$ the process needs more pages
  - Page fault rate too low $\Rightarrow$ process may have too many frames
  - Establish upper and lower-level bounds on desired page fault rate
  - If the time interval between the current and the previous page fault exceeds a pre-specified value, all the pages not referenced during this time are removed from the memory
  - PFF guarantees that the resident set grows when page faults are frequent, and shrinks when the page fault rate decreases
  - The resident set is adjusted only at the time of a page fault (compare with the working set model)

- Prepaging

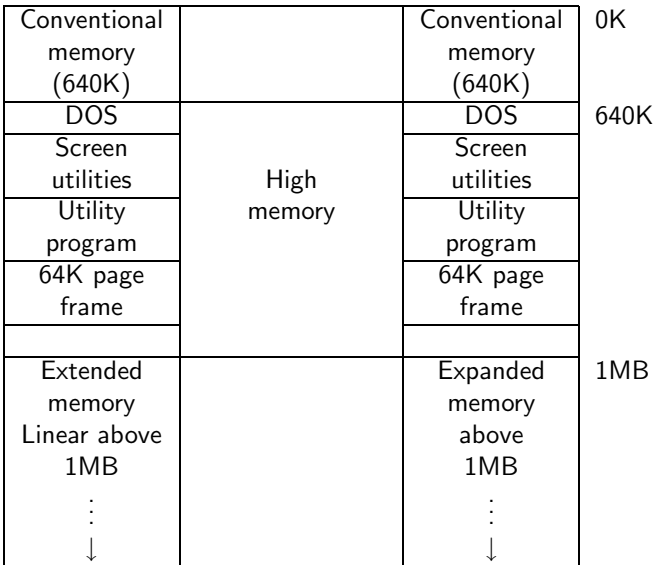
  - Bring into memory at one time all pages that will be needed
  - Relevant during suspension of a process
  - Keep the working set of the process
  - Rarely used for newly created processes
- Page size
  - Size of page table
    - Smaller the page size, larger the page table
    - Each process must have its own copy of the page table
  - Smaller page size, less internal fragmentation
  - Time required to read/write a page in disk
  - Smaller page size, better *resolution* to identify working set
  - Trend toward larger page size
    - Intel 80386 – 4K page
    - Motorola 68030 – variable page size from 256 bytes to 32K
- Program structure
  - Careful selection of data structures
  - Locality of reference

Memory management in MS-DOS

- DOS uses different memory types – Dynamic RAM, Static RAM, Video RAM, and Flash RAM
  - Dynamic RAM (DRAM)
    - Used for the bulk of immediate access storage requirements
    - Data can only be accessed during refresh cycles, making DRAM slower than other memory types
    - Linked to the CPU by local bus to provide faster data transfer than standard bus
    - Chips range in capacity from 32 and 64 Kb to 16Mb
  - Static RAM (SRAM)
    - Does not require constant electrical refreshing to maintain its contents
- Memory allocation in MS-DOS
  - Determined by the amount of physical memory available and the processor
  - 386-based machines can address up to 4 GB of memory but were once limited by MS-DOS to 1 MB
  - Memory is allocated into the following components
    1. **Conventional memory**
       - Ranges from 0 to 640K
       - Used primarily for user programs
    2. **High memory**
       - Ranges from 640K to 1M
       - Used to map ROMs, video, keyboards, and disk buffers
    3. **Extended memory**
       - Memory directly above the 1M limit
       - Most programs cannot directly use this memory without a major rewrite to enable them to address it
         - Problem can be overcome by using extended memory managers, such as `HIMEM.SYS` from Microsoft

- Such managers allow the programs to run unmodified in extended memory to the limits supported by the processor (4G on 386)

4. **Expanded memory**
   - Not a preferred way for using additional memory
   - Slower than extended memory
   - Use a 64K "page frame" in high memory to access memory outside DOS's 1M limit
   - Program chunks of 16K are swapped into this page frame as needed, managed by expanded memory manager
   - Emulated in extended memory by using a special program such as Microsoft's `EMM386`



File Systems

- Result of the integration of storage resources under a single hierarchy
- File
  - A collection of related information defined by its creator
  - The abstraction used by the kernel to represent and organize the system's non-volatile storage resources, including hard disks, floppy disks, CD-ROMs, and optical disks.
- Storage capacity of a system restricted to size of available virtual memory
- May not be enough for applications involving large data (face expt.)
- Virtual memory is volatile
- May not be good for long term storage
- Information need not be dependent upon process
- `passwd` file may need to be modified by different processes
- Essential requirements of long-term information storage
  - Store very large amount of information
  - Information must survive termination of processes (be *persistent*)
  - Multiple processes must be able to access information concurrently
- Store information in *files*
- File system – Part of the OS that deals with file management

Files

- Most visible aspect of an OS
- Mechanism to store and retrieve information from the disk
- Represent programs (both source and object) and data
- Data files may be numeric, alphanumeric, alphabetic, or binary
- May be free form or formatted rigidly
- Accessed by a *name*
- Created by a process and continues to exist after the process has terminated
- Information in the file defined by creator
- Naming conventions
  - Set of a fixed number of characters (letters, digits, special characters)
  - Case sensitivity
  - File type should be known to OS
    - to avoid common problems like printing binary files
    - to automatically recompile a program if source modified (TOPS 20)
  - File extension in DOS

  - *Magic number* in `Unix`
    * Identification number for the type of file
    * The `file(1)` command identifies the type of a file using, among other tests, a test for whether the file begins with a certain *magic number*
    * Magic number is specified in the file `/etc/magic` using four fields
      · Offset: A number specifying the offset, in bytes, into the file of data which is to be tested
      · Type: Type of data to be tested – byte, short (2-byte), long (4-byte), or string
      · Value: Expected value for file type
      · Message: Message to be printed if comparison succeeds
    * Used by the `C` compiler to distinguish between source, object, and assembly file formats
    * Developing magic numbers
      · Start with first four letters of program name (e.g., list)
      · Convert them to hex: 0x6c607374
      · Add 0x80808080 to the number
      · The resulting magic number is: 0xECE0F3F4
      · High bit is set on each byte to make the byte non-ASCII and avoid confusion between ASCII and binary files

- File structure
  - Byte sequence
    * Unix and MS-DOS use byte sequence
    * Meaning on the bytes is imposed by user programs
    * Provides maximum flexibility but minimal support
    * Advantages to users who want to define their own semantics on files
  - Record sequence
    * Each file of a fixed length record
    * Card readers and line printer based records
    * Used in CP/M with a 128-character record
  - Tree
    * Useful for searches
    * Used in some mainframes

- File types
  - Regular files
    * Most common types of files
    * May contain ASCII characters, binary data, executable program binaries, program input or output
    * no kernel level support to structure the contents of these files
    * Both sequential and random access are supported
  - Directories
    * Binary file containing a list of files contained in it (including other directories)
    * May contain any kind of files, in any combination
    * `.` and `..` refer to directory itself and its parent directory
    * Created by `mkdir` and deleted by `rmdir`, if empty
    * Non-empty directories can be deleted by `rm -r`
    * Each entry made up of a file-inode pair
      · Used to associate inodes and directory locations

        - Data on disk has no knowledge of its logical location within the filesystem
- Character-special files and Block-special files
    - Allow Unix applications to communicate with the hardware and peripherals
    - Reside in the `/dev` directory
    - Kernel keeps the links for device drivers (installed during configuration)
    - Device drivers
        - Present a standard communications interface
        - Take the request from the kernel and act upon it
    - Character-special files
        - Allow the device drivers to perform their own I/O buffering
        - Used for unbuffered data transfer to and from a device
        - Generally have names beginning with `r` (for *raw*), such as `/dev/rsd0a`
    - Block-special devices
        - Expect the kernel to perform buffering for them
        - Used for devices that handle I/O in large chunks, known as blocks
        - Generally have names without the `r`, such as `/dev/sd0a`
    - Possible to have more than one instance of each type of device
        - Device files characterized by major and minor device number
        - Major device number
          Tells the kernel the driver corresponding to the file
        - Minor device number
          Tells the kernel about the specific instance of the device
          Tape drivers may use the minor device number to select the density for writing the tape
    - Created with `mknod` command and removed with `rm`
- Hard links
    - Additional name (alias) for a file
    - Associates two or more filenames with the same inode
    - Indistinguishable from the file it is linked to
    - Share the same disk data blocks while functioning as independent directory entries
    - May not span disk partitions as inode numbers are only unique within a given device
    - Unix maintains a count of the number of links that point to the same file and does not release the data blocks until the last link has been deleted
    - Created with `ln` and removed with `rm`
- Symbolic links
    - Also known as *soft* link
    - Pointer files that name another file elsewhere on the file system
    - Reference by name; distinct from the file being pointed to
    - Points to a Unix pathname, not to an actual disk location
    - May even refer to non-existent files, or form a loop
    - May contain absolute or relative path
    - Created with `ln -s` and removeed with `rm`
    - Problem of using `..` in the symbolic link
- FIFO special file, or "named pipe" (ATT)
    - Characterized by transient data
    - Allow communications between two unrelated processes running on the same host
    - Once data is read from a pipe, it cannot be read again

        * Data is read in the order in which it was written to the pipe, and the system allows no deviation from that order
        * Created with `mknod` command and removed with `rm`
    - Unix domain sockets (BSD)
        * Connections between processes for communications
        * Part of the TCP/IP networking functionality
        * Communication end point, tied to a particular port, to which processes may attach
        * Socket `/dev/printer` is used to send messages to the line printer spooling daemon `lpd`
        * Visible to other processes as directory entries but cannot be read from or written into by processes not involved in the connection
        * Created with the `socket` system call, and removed with `rm` or `unlink` command (if not in use)
- Regular files
    - Text files (ASCII)
        * Lines of text
        * Lines may be terminated by carriage return
        * File itself has an end-of-file character
        * Useful for interprocess communication via pipes in `Unix`
    - Binary files
        * Not readily readable
        * Has internal structure depending upon the type of file (executable or archive)
        * Executable file (`a.out` format)
            · Header: Magic number, Text size, Data size, BSS size, Symbol table size, Entry point, Flags
            · Text
            · Data
            · Relocation bits
            · Symbol table
        * BSS or Block Started by Symbol
            · Uninitialized data for which kernel should allocate space
            · Used by an obsolete IBM assembler, BSS was an assembler pseudo-opcode that filled an area of memory with zeros
        * Library archive: compiled but not linked modules
            · Header: Module name, Date, Owner, Protection, Size
            · Object module
- File access
    - Sequential access
        * Bytes or records can be read only sequentially
        * Like magnetic tape
    - Random access
        * Bytes or records can be read out of order
        * Access based on key rather than position
        * Like magnetic disk
    - Distinction more apparent in older OSs
- File attributes
    - Data about files

  - Protection, Password, Creator, Owner, Read-only flag, Hidden file flag, System file flag, Archive flag, ASCII/binary flag, Random access flag, Temporary flag, Lock flag, Record length, Key position, Key length, Creation time, Time of last access, Time of last change, Current size, Maximum size

- File operations: File treated as abstract data type
  - `create`.
    - Create a new file of size zero (no data)
    - `Unix` commands: `creat`(2), `open`(2), `mknod`(8)
    - The attributes are set by the environment in which the file is created, *e.g.* `umask`(1)
  - `delete`.
    - Delete an existing file
    - Some systems may automatically delete a file that has not been used in n days
  - `open`.
    - Establish a logical connection between process and file
    - Fetch the attributes and list of disk addresses into main memory for rapid access during subsequent calls
    - I/O devices `attached` instead of `opened`
    - System call `open`(2)
  - `close`.
    - Disconnects `file` from the current process
    - `file` not accessible to the process after `close`
    - I/O devices `detached` instead of `closed`
  - `read`.
    - Transfer the logical record starting at current position in file to memory starting at `buf`
    - `buf` known as input buffer
    - Older systems have a `read_seq` to achieve the same effect and `read_direct` for random access files
  - `write`.
    - Transfer the memory starting at `buf` to logical record starting at current position in file
    - If current position is the end of file, file size may increase
    - If current position is in the middle of file, some records may be overwritten and lost
    - Older systems have a `write_seq` to achieve the same effect and `write_direct` for random access files
  - `append`.
    - Restrictive form of `write`
  - `seek`.
    - Primarily used for random access files
    - Repositions the pointer to a specific place in file
    - `Unix` system call `lseek`(2)
  - `get_attributes`.
    - Get information about the file
    - In `Unix`, system calls `stat`(2), `fstat`(2), `lstat`(2)
      - Device file resides on
      - File serial number, i-node number
      - File mode
      - Number of hard links to the file
      - uid/gid of owner
      - Device identifier (for special files)

            - Size of file
            - Last time of access, modification, status change
            - Preferred block size of file system
            - Actual number of blocks allocated
    - `set_attributes`.
        - Set the attributes of a file, *e.g.*, protection, access time
        - `Unix` system calls: `utimes`(2), `chmod`(2)
    - `rename`.
        - Change the name of a file

- Memory-mapped files
    - Map a file into the address space of a running process
    - First introduced in MULTICS
    - Given a file name and virtual address, map the file name to the virtual address
    - System internal tables are changed to make the actual file serve as the backing store in virtual memory for the mapped pages
    - Works best in a system with segmentation
        - Each file can be mapped to its own segment
    - `Unix` system calls `mmap`(2) and `munmap`(2)
        - `mmap` establishes a mapping between the process's address space at an address `pa` for `len` bytes to the memory object represented by `fd` at `off` for `len` bytes
        - `munmap` removes the mappings for pages in the specified address range
        - `mprotect`(2) sets/changes the protection of memory mapping
    - Problems with memory-mapped files
        - System cannot tell the size of the file when it is ready to `unmap`
            - File (or data segment) size in `Unix` is increased by system calls `brk`(2), `sbrk`(2)
            - System page size can be determined by `getpagesize`(2)
        - What if a memory-mapped file is opened for conventional reading by another process
        - File may be larger than a segment, or entire virtual address space

Directories

- Provided by the file system to keep track of files
- Hierarchical directory systems
    - Directory contains a number of entries – one for each file
    - Directory may keep the attributes of a file within itself, like a table, or may keep them elsewhere and access them through a pointer
    - In opening a file, the OS puts all the attributes in main memory for subsequent usage
    - Single directory for all the users
        - Used in most primitive systems
        - Can cause conflicts and confusion
        - May not be appropriate for any large multiuser system
    - One directory per user
        - Eliminates name confusion across users

    - May not be satisfactory if users have many files
- Hierarchical directories
    - Tree-like structure
    - Root directory sits at the top of the tree; all directories spring out of the root
    - Allows logical grouping of files
    - Every process can have its own working directory to avoid affecting other processes
    - Directories `.` and `..`
    - Used in most of the modern systems
- Information in directory entry
    - File name
      Symbolic file name
      Only information kept in human readable form
    - File type
      Needed for those systems that support different file types
    - Location
      Pointer to the device and location of file on that device
    - Size
      Current size of the file
      May also include the maximum possible size for the file
    - Current position
      Pointer to the current read/write position in the file
    - Protection
      Access control information
    - Usage count
      Number of processes currently using the file
    - Time, date, and process identification
      May be kept for creation, last modification, and last use
      Useful for protection and usage monitoring
- Directory entry may use from 16 to over 1000 bytes for each file
- Size of directory may itself become very large
- Directory can be brought piecemeal into memory as needed
- Data structures for a directory
    - Linked list
        - Requires linear search to find an entry
        - Simple to program but time consuming in execution
        - To create a new file, must search entire directory to avoid name duplication
        - To delete a file, search the directory and release the space allocated to it
          Entry can be marked as unused or attached to a list of free entries
    - Sorted list
        - Allows binary search and decrease average search time
        - Search algorithm is more complicated to program
        - May complicate creation and deletion as large amount of directory information may be moved to keep list sorted
    - Hash table
        - Greatly improves directory search time
        - Insertion and deletion straightforward
        - Problem because of fixed size of hash tables

- Path names
  - Convention to specify a file in the tree-like hierarchy of directories
  - Hierarchy starts at the directory /, known as the root directory
  - Made up of a list of directories crossed to reach the file, followed by the file name itself
  - Absolute path name
    - Path from root directory to the file
    - Always start at the root directory and is unique
    - `/usr/bin/X11/xdvi`
  - Relative path name
    - Used in conjunction with the concept of "current working directory"
    - Specified relative to the current directory
    - More convenient than the absolute form and achieves the same effect
- Unix allows arbitrary depth for the file tree
  - The name of each directory must be less than 256 characters
  - No pathname can be longer than 1023 characters
  - Files with pathname longer than 1023 characters can be accessed by cding to an intermediate directory
- Directory operations
  - `create_directory`.
    - Create a directory and put entries for `.` and `..` in there
    - `mkdir` command in `Unix` and MS-DOS
    - `mkdir`(2) system call in `Unix`
      - `int mkdir(path, mode)`
      - Creates a directory with the name `path`
      - Mode mask of the new directory is initialized from `mode`
      - Set-GID bit of `mode` is ignored and is inherited from the parent directory
      - Directory owner-id is the process's effective user-id
      - Group-id is the GID of the directory in which the new directory is created
  - `delete_directory`.
    - Delete an existing directory
    - Only empty directory can be deleted
    - Directory containing just `.` and `..` is considered empty
    - `rmdir` command in `Unix` and MS-DOS
      - In `Unix`, it is forbidden to remove the file `..`
    - `rmdir`(2) system call in `Unix`
      - `int rmdir(path)`
      - Can remove a directory only if its link count is zero and no process has the directory open
      - If one or more processes have the directory open when the last link is removed, the `.` and `..` entries, if present, are removed before `rmdir()` returns and no new entries may be created in the directory, but the directory is not removed until all references to the directory have been closed
      - `rmdir()` updates the time fields of the parent directory
  - `open_directory`.
    - Open a directory to read/write
    - Used by many applications, *e.g.* `ls`

    - C library function opendir(3)
        - `DIR *opendir (dirname)`
          `char *dirname`
        - Open the directory named by `dirname` and associate a `directory stream` with it
        - Returns a pointer to identify the directory stream in subsequent operations
        - If directory cannot be accessed, a `null` pointer is returned
- `close_directory`.
    - Close the directory that was opened for reading
    - C library function `closedir`(3)
        - `int closedir (dirp)`
          `DIR *dirp`
        - Closes the named directory stream and frees the structure associated with the `DIR` pointer
- `read_directory`.
    - Returns the next entry in an open directory
    - C library function `readdir`(3)
        - Returns a pointer to the next directory entry
        - Returns `null` upon reaching the end of the directory or detecting an invalid `seekdir`(3) operation
- `rename`.
    - Operates the same way as renaming a file
- `link`.
    - Allows creation of aliases (links) for the files/directories
    - Same file can appear in multiple directories
    - User command `ln`(1)
        - Creates hard or symbolic links to files
        - A file may have any number of links
        - Links do not affect other attributes of a file
        - Hard links
          Can only be made to existing files
          Cannot be made across file systems (disk partitions, mounted file systems)
          To remove a file, all hard links to it must be removed
        - Symbolic links
          Points to another named file
          Can span file systems and point to directories
          Removing the original file does not affect or alter the symbolic link itself
          cd to a symbolic link puts you in the pointed-to location within the file system; changing to the parent of the symbolic link puts you in the parent of the original directory
          Problem can be solved in `C-shell` by `pushd` and `popd`
    - System call `link`(2)
        - Used to make a hard link to a file
        - Increments the link count of the file by one
    - System call `symlink`(2)
        - Used to make a symbolic link to a file
- `unlink`.
    - Remove a directory entry
    - If the file being removed is present in one directory, it is removed from the file system
    - If the file being removed is present in multiple directories, only the path name specified is removed; others remain

    * User commands `rm`(1) and `rmdir`(1)
    * System call `unlink`(2)
        · Removes the directory entry
        · Decrements link count of the file referred to by that entry
        · If the entry is the last link to the file, and no process has the file open, all resources associated with the file are reclaimed
        · If the file is open in any process, actual resource reclamation is delayed until it is closed, even though the directory entry has disappeared

File system implementation

- File system manages files, allocating files space, administering free space, controlling access to files, and retrieving data for users
- Processes interact with the file system using a set of system calls
- File data is accessed using a buffering mechanism that regulates data flow between the kernel and secondary storage devices
    - Buffering mechanism interacts with the block I/O device drivers to initiate data transfer to and from kernel
    - Device drivers are kernel modules that control the operation of peripheral devices
    - It is the device driver that makes the kernel treat a device as a block-special device (or random access storage device)
        * Yes, the Unix kernel can allow a tape drive to be treated as a random access storage device
    - File system also interacts directly with the character-special devices
- Implementing files
    - Support of primitives for manipulating files and directories
    - Make an association between disk blocks and files
    - Contiguous allocation
        * Simplest allocation technique
        * Simple to implement; files can be accessed by knowing the first block of the file on the disk
        * Improves performance as the entire file can be read in one operation
        * Problem 1 – File size may not be known in advance
        * Problem 2 – Disk fragmentation; can be partially solved by compaction
    - Linked list allocation
        * Files kept as a linked list of disk blocks
        * First word of each block points to the next block
        * Only the address of the first block appears in the directory entry
        * No disk fragmentation
        * Random access is extremely slow
        * Data in a block is not a power of 2 (to accommodate the link to next block)
    - Linked list allocation using an index

Unix model of ownership

- Processes and files are owned by someone

- Files have both a user owner and a group owner, with the two ownerships decoupled
- Allows file protections and permissions to be organized as per local needs
- File ownership
  - Owner of a file has its primary control and can be superseded only by root
  - Every file has a primary owner and one or more group owners
  - Primary owner can modify the access privileges on the file for everyone (except root) by using the `chmod` command
  - Groups are defined in `/etc/group`
  - Owner of the file decides for group privileges to allow shared access among members of the group
  - The user owner of a file need not be a member of the group that owns it
  - Ownership of file can be changed by `chown` command
  - Group ownership of the file can be changed by `chgrp` command
  - File permissions can be checked by `ls -lg` under BSD and `ls -l` under AT&T
  - Owner and group are tracked by uid and gid, which are mapped to user names and group names by using the files `/etc/passwd` and `/etc/group`, respectively
- Ownership of new file
  - User ownership with the user who creates it
  - Group ownership dependent on BSD or SYS V
  - Group ownership in BSD
    - Same as the group ownership of the directory in which the file is created
    - Default on DEC Ultrix
  - Group ownership in SYS V
    - Current group of the user who creates the file
    - Default on most Unix systems
  - The default attributes can be changed by setting the SGID bit (set group-id bit) on the directory
- Changing file ownership
  - Possible through `chown` and `chgrp` commands
  - `chown new-owner file(s)`
    - In SYS V
      - Root and user owner of the file can change the owner of a file
      - Once a user changes ownership of his file, he cannot get it back by himself
    - In BSD
      - Only the root may change the ownership of a file
  - `chown -R` recursively changes the ownership of all files in a directory, including the directory itself
  - `chgrp new-group file(s)`
    - Changes group ownership of a file to `new-group`
    - User must own the file and be a member of the `new-group`
  - `chown new-owner.new-group file(s)` changes both user and group ownership in one operation

Deconstructing the filesystem

- File tree
  - Composed of chunks called filesystems
    - Filesystem consists of one directory and its subdirectories
    - Attached to the file tree with the `mount` command
      - `mount` maps a directory within the tree at mount point
      - Previous contents of mount point are not accessible as long as a filesystem is mounted there
      - Preferable to declare mount points as empty directories
      - Example
        `mount /dev/sd0a /users`
        installs the filesystem on device `/dev/sd0a` at the mount point `/users`
    - Detached from a file tree with the `umount` command
      - A busy filesystem cannot be detached
      - Filesystem is busy if it contains open files, has a process entered into it using the `cd` command, or contains executables that are running
      - In some flavors (OSF/1 and BSDI), busy filesystems can be detached by using `umount -f` command
    - `lsof` program
      - List of open files
      - Catalogs open file descriptors by process and filenames
- Root filesystem
  - Contains the root directory and a minimal set of subdirectories
  - The `kernel` resides in the root filesystem
  - `/bin`
    - Contains binary executables for different user commands
    - Commands needed for minimum system operability
    - `/bin` is sometimes a link to `/usr/bin`
    - Other directories containing the user commands are `/usr/bin` and `/usr/ucb`
  - `/dev`
    - Contains device entries for terminals, disks, modems, etc.
    - Device types indicated by the name of the file
      - `dsk` – Disk accessed in block mode
      - `rdsk` – Disk accessed in raw mode
      - `mt` – Magnetic tape accessed in block mode
      - `rmt` – Magnetic tape accessed in raw mode
      - `term` – Terminal on serial line
      - `pts` or `ptc` – Pseudo terminal
  - `/etc` and `/sbin`
    - System configuration files and executables, administrative files, boot scripts
    - Executable binaries for most system administration commands
    - `/etc/default`, if it exists, may contain default parameter values for various commands
  - `/home`
    - Users' home directories
    - `/u` or `/users` in some systems
  - `/lost+found`
    - Directory for lost files

    - Files may be lost due to disk error or improper system shutdown
        - Refer to disk locations marked as used but not listed in any directory
        - A non-empty inode not listed in any directory
    - The program `fsck`, normally run at boot time, finds these files
    - Every disk partition has a `lost+found` directory
- `/mnt`
    - Mount directory for temporary mounts
- `/proc`
    - Images of all running processes
    - Allows processes to be manipulable using Unix file access system calls
    - Files correspond to active processes (entries in the kernel process table)
    - There may be additional files (on Linux) containing information on system configuration: use of interrupts and I/O ports, and allocation of DMA channel and CPU
- `/tcb`
    - Trusted Computer Base
    - Directory tree for security-related database files on some systems offering enhanced security features
    - Configuration files related to the TCB are stored under `/etc/auth`
- `/tmp`
    - Temporary files that disappear between reboots
    - Available to all users as a scratch directory
- `/usr`
    - Contains subdirectories for locally generated programs, executables for user and administrative commands, shared libraries, and other parts of Unix OS
    - Also may contain application programs
    - `/usr/adm`
        - Administrative directory
        - Accounting files, records of resource usage
        - Recent versions of Unix have this directory changed to and linked to `/var/adm`
    - `/usr/bin`
        - Executable files, including shellscripts
        - Executables for X window system are stored in `/usr/bin/X11`
    - `/usr/games`
        - Games and diversions (old collection; not fun any more)
        - Some sites may not even have this one
    - `/usr/include`
        - Header files for C programs
        - Useful to define the program's interface to standard system libraries
        - Directory `/usr/inlcude/sys` contains include files for operating system
    - `/usr/ucb`
        - Berkeley utilities and programs
    - `/usr/lib`
        - Support files for standard Unix applications
        - Standard C libraries for math and I/O
        - Names of the form `libx.a` where `x` is one or more characters related to the library's contents
        - Also may contain configuration files for some Unix services
    - `/usr/local`

        - Local software
        - Subdivided into another hierarchy
        - `/usr/local/adm`
        - `/usr/local/bin`
        - `/usr/local/etc`
        - `/usr/local/lib`
        - `/usr/local/sbin`
        - `/usr/local/src`
    * `/usr/man`
        - On-line manual pages
        - Divided into subdirectories for the various sections of the manual
        - Contains several `manx` and `catx` directories where `x` denotes the number 1 through 8, or the letters l or n
        - `catx` directories may be eliminated to save space
        - Significance of the numbers is given by the following table

1	User commands
2	System calls
3	Subroutines
4	Devices (Special files and hardware)
5	File formats and configuration files
6	Games and demos
7	Miscellaneous: characters sets, filesystem types, etc
8	System administration and maintenance
l	Local
n	New

    * `/usr/share`
        - Shared data
        - Static data files, such as manual pages, font directories, files for `spell`
        - `/usr/share/man`
          Shared manual pages
- `/var`
    * Varying data, including spooling and other volatile directories
    * `/var/spool`
        - Spooling directories for printers, mail, UUCP
        - `cron` utility also keeps the files here
    * `/var/tmp`
        - Temporary space where the files do not disappear between reboots

File access in UNIX

- Controlled by a set of nine permission bits
- Three sets of three bits each (RWX) corresponding to user, group, and other
- An additional three bits affect the operation of execution of a program
- The twelve mode bits are stored together with four bits of file type information in one 16-bit word
- File type bits
    - Set at the time of file creation

  - Cannot be changed

- Mode bits
  - Control three types of file access – read, write, and execute
  - Meaning for file and directory

Access	File	Directory
read	View contents	Search contents (using `ls`)
write	Change contents	Change contents (add or delete files)
execute	Run the executable	Allowed to get into the directory

  - A shellscript can be run only if the user has both read and executable permission on it
  - Can be changed by the file owner or root using the `chmod` command
- The setuid and setgid bits
  - Bits with octal values 4000 and 2000
  - Allow programs to access files and processes that may be otherwise off limits
  - On SunOS, setgid bit on a directory controls the group ownership of the newly created files within the directory
- Sticky bit
  - Bit with octal value 1000
  - Not important in current systems
  - If the sticky bit is set on a directory, you cannot delete or rename a files unless you are the owner of the directory, file, or executable
  - Makes directories like `/tmp` somewhat private
- Summary of the three bits

Code	Name	Meaning
t	Sticky bit	Keep executable in memory after exit
s	SUID	Set process user-id on execution
s	SGID	Set process group id on execution
l	File locking	Set mandatory locking in reads/writes

  Under SunOS, turning on sgid access on a file and removing execute permission results in mandatory file locking
- If the setuid or sticky bit is set but the corresponding execute permission bit is not set, the s or t in the filename listing appear in the uppercase
- Assigning default permissions
  - Set by the `umask` command
  - Complement of the permissions allowed by default

Inodes

- Index node
- Structure to keep information about each file, including its attributes and location
- Contain about 40 separate pieces of information useful to the kernel
  - UID and GID

  - File type
  - File creation, access, and modification time
  - Inode modification time
  - Number of links to the file
  - Size of the file
  - Disk addresses, specifying or leading to the actual disk locations for disk blocks that make up the file
- Inode table
  - Created at the time of creation of filesystem (disk partition)
  - Always in the same position on the filesystem
  - Inode table size determines the maximum number of files, including directories, special files, and links, that can be stored into the filesystem
  - Typically, one inode for every 2 to 8 K of file storage
- Opening a file
  - Process refers to the file by name
  - Kernel parses the filename one component at a time, checking that the process has permission to search the directories in the path
  - Kernel eventually retrieves the inode for the file
  - Upon creation of a new file, kernel assigns it an unused inode
  - Inodes stored in the file system but kernel reads them into an in-core inode table when manipulating files

File system structure

- Four components
  1. Boot block
     - Occupies the beginning of the file system, typically the first sector
     - May contain the bootstrap code
     - Only one boot block needed for initialization but every filesystem has a possibly empty boot block
  2. Super block
     - Describes the state of the file system
       * Size of the file system
       * Maximum number of files it can store
       * Location of the free space
  3. Inode list
     - List of inodes following the superblock
     - Kernel references inodes by indexing into the inode list
     - Root inode
       * Inode by which the directory structure of the file system is accessible after execution of the `mount` system call
  4. Data blocks
     - Start at the end of the inode list
     - Contain file data and administrative data
     - An allocated data block can belong to one and only one file in the file system

Distributed file systems

- Allows file systems to be distributed on different physical machines while still accessible from any one of those machines
- Advantages include easier backups and management
- Exemplified by NFS (network file system, developed by Sun) and RFS (developed by AT&T) on UNIX and NTFS on Windows NT
- Must provide features for performance, security, and fault tolerance
- May provide support for cross-OS semantics
  - Almost impossible for a Unix system to understand other systems' protection mechanism
  - DOS has no protection system to speak of
  - VAX/VMS and Windows NT have access control lists which cannot be mapped easily onto every Unix file system
  - Other issues include mapping of user IDs, file names, record formats, and so on
- RFS
  - Supplied as a part of SYS V
  - Designed to provide Unix file system semantics across a network
  - Non-Unix files systems cannot be easily integrated into RFS
  - Called a *stateful* system because it maintains the state of the open files at the file server where files reside
  - Efficient but can cause problems when a remote file server crashes causing a loss of state
- NFS
  - Stateless; no state is maintained for open files at the remote server
  - Less efficient as remote file server may have to reopen a file for each network I/O transaction
    - Can be improved by caching
  - Recovery from a crash on remote server is transparent to client since the remote server has no state to lose
    - Some Unix semantics are impossible to support if remote server does not save state
    - Set of Unix semantics supported is relatively primitive enough that most non-Unix file systems can be accessed under NFS

Windows NT file system

- Windows NT provides the option of two different filesystems
  1. NTFS
     - Provides a modern namespace and supports large-sized files and volumes
     - Has a more comprehensive security mechanism compared to UNIX filesystem permission bits
     - NTFS files and directories have access control lists (ACLs) to control access at the user level (available in Solaris; `getfacl`, `setfacl`, and other ACL commands)
     - NTFS also associates specific rights with users and groups; the rights match actions that users may wish to perform such as backing up files and directories
       - This enables an admin to distribute permissions as needed, without giving away the entire system
  2. FAT

  - File allocation table
  - Allows compatibility with previous Windows/DOS operating systems

Active Directory in Windows 2000

- Implementation closely resembles Netscape Directory Server
- Supported on NTFS-mounted file systems (not on FAT or FAT32)
- Hierarchically structured
- Employs a namespace closely resembling the internet's DNS (Domain Name System) namespace
- Directory service provider locations are stored in a DNS server and can be located by clients and other services using Lightweight Directory Access Protocol (LDAP) queries
- Service-provider location updates can be applied automatically with Dynamic DNS
- Updates to DNS can also be done manually independent of the Active Directory
- Hierarchy of *Organizational Units* and other objects within domains; also hierarchy of domains
  - Domain tree
    - Hierarchy of domains constitutes a contiguous namespace
    - Example: `hoare.cs.umsl.edu` and `laplace.cs.umsl.edu` are linked to `cs.umsl.edu`
  - Domain forest
    - Domains do not constitute a contiguous namespace
    - Example: `cs.umsl.edu` and `cnn.com`
    - Active Directory can create a *virtual root* to cope with such a scenario
- Global catalog
  - Part of domain tree and domain forest hierarchy
  - Keeps track of all the objects in multiple domains without storing every attribute
  - Indexes the limited number of attributes to facilitate fast searches, avoiding the search in entire domain
- Security issues
  - Security and administration privileges can be assigned to users in a highly granular fashion within a domain (domain tree)
  - The process is known as *delegation* (Microsoft term)
  - Rights can be inherited
    - A person with certain privileges in domain `cs.umsl.edu` will get the same privileges on all machines in the domain tree, for example on `hoare.cs.umsl.edu`
  - Trust between domains
    - In Windows NT, trust between domains is managed manually and is one of the problematic issues for multiple domains
    - Active directory automatically establishes bidirectional trust relationships between domain and supports transitive trust relationships
- Integration with Internet
  - Active directory namespace is almost identical to Internet namespace, using a host.com structure
  - Windows 2000 can use DNS as service locator for the directory

- Exception to the rule
  - Internet permits the existence of duplicate names within a domain, for example, `laplace.cs.umsl.edu` and `laplace.phys.umsl.edu`
  - W2K would not permit the reuse of name `laplace` within a domain because an account named `sanjiv@umsl.edu` can be assigned to only one of these names

I/O Management

- OS controls all I/O devices
- Preferable to have the same interface for all I/O devices (*device independence*)

Secondary Storage Management

- Secondary storage – An extension of primary storage
  - Must hold vast amount of data permanently
  - Main memory is too small to store all needed programs and data permanently
  - Main memory is volatile storage device
  - Magnetic tape
    - Quite slow in comparison to main memory
    - Limited to sequential access
    - Unsuitable to provide random access needed for virtual memory
  - Magnetic disks, CDROMs, Optical disks
    - The storage capacity is much larger
    - The price per bit is much lower
    - Information is not lost when power is turned off
- Disk hardware
  - Physical structure
    - Disk surface divided into tracks
    - A read/write head positioned just above the disk surface
    - Information stored by magnetic recording on the track under read/write head
    - Fixed head disk
    - Moving head disk
    - Designed for large amount of storage
    - Primary design consideration cost, size, and speed
    - Head crash
  - Hardware for disk system
    - Disk drive
      Device motor
      Read/write head
      Associated logic
    - Disk controller
      Determines the logical interaction with the computer
      Can service more than one drive (*overlapped seeks*)
    - Cylinder
      The same numbered tracks on all the disk surfaces
      Each track contains between 8 to 32 sectors
    - Sector
      Smallest unit of information that can be read from/written into disk
      Range from 32 bytes to 4096 bytes
    - Data accessed by specifying surface, track, and sector
    - View the disk as three dimensional array of sectors

    * OS treats the disk as one dimensional array of disk blocks
      s – Number of sectors per track
      t – Number of tracks per cylinder
      Disk address b of cylinder i, surface j, sector k

      $$b = k + s \times (j + i \times t)$$

    * Seek time
      Time required by read/write head to move to requested track
      Farther apart the tracks, more the seek time
    * Latency time
      Time required for the requested sector to come under the read/write head

- Device directory
  - Contains identification of files on the disk
    * Name of file
    * Address on the disk
    * Length, type, owner
    * Time of creation
    * Time of last use
    * Protections
  - Often stored in a fixed address

Free-Space Management

- Free-space list – All disk blocks that are free
- Bit vector
  - Each block represented by a bit
  - Relatively simple approach
  - Efficient to find n consecutive free blocks on the disk
  - Uses bit manipulation instructions (Intel 80386, Motorolla 68020/30)
  - Used by Apple Macintosh
  - Inefficient unless the entire vector kept in main memory for most accesses and occasionally written to disk for recovery
  - May not be feasible to keep the bitmap in memory for large disks
- Linked list
  - Link all free disk blocks together
  - Not efficient – to traverse the list, must read each block requiring substantial I/O time
- Grouping
  - Store the addresses of n free blocks in first free block
  - nth block contains the address of another n free blocks
- Counting
  - Several contiguous blocks may be allocated or freed en masse
  - Keep the address of first free block and the number n of free contiguous blocks that follow

Allocation Methods

- Problem – Allocate space to files so that
  - disk space is utilized effectively
  - files can be accessed quickly
- Assume a file to be a sequence of blocks
- Contiguous allocation
  - Each file occupies a set of contiguous addresses on disk
  - Number of disk seeks required to access contiguously allocated files is minimal
  - Seek time, if needed, is minimal
  - Defined by the disk address and number of blocks
  - Straightforward file access
    - Sequential access – Remember the last block referenced and when necessary, read the next block
    - Direct access – To access block i of a file starting at b, access block $b + i$
  - Problem in finding space for a new file
    - Equivalent to general dynamic storage allocation problem
    - Solution by first-fit, best-fit, and worst-fit strategies
    - External fragmentation
    - Must repack or compact files occasionally
    - Determining the size of file being created
    - A file growing slowly (over a period of a few months) must be allocated enough space for its final size
- Linked allocation
  - Each file a linked list of disk blocks
  - Disk blocks may be scattered anywhere on the disk
  - Directory contains a pointer to first and last block of file
  - Easy to fix the problems in contiguous allocation
  - No external fragmentation
  - No need to declare the size of a file
  - No need to compact disk space
  - Problems
    - Effective only for sequentially accessed files
    - Wasted space to keep pointers (2 words out of 512 $\Rightarrow$ 0.39% wastage)
    - Reliability – A bug might overwrite or lose a pointer
      Might be solved by doubly linked lists (more waste of space)
  - File Allocation Table (FAT)
    - Create a table on disk, indexed by block number
    - One entry for each disk block
    - Used as a linked list
    - Unused blocks indicated by a zero-valued entry
    - Used by MS-DOS and OS/2
- Indexed allocation

  - Bring all pointers into one block called *index block*
  - Index block for each file – disk-block addresses
  - ith entry in index block $\equiv$ ith block of file
  - Supports direct access without suffering from external fragmentation
  - Pointer overhead generally higher than that for linked allocation
  - More space wasted for small files
  - Size of index block
    * Preferred to be small
    * Linked scheme
      Normally taken as one disk block
      Larger files can be accomodated by linking together several index blocks
    * Multilevel index
      Separate index block to point to index blocks which point to file blocks
      Assume 256 pointers to one index block
      65,536 pointers to two levels of index
      1K per block
      64M file
    * Combined scheme
      BSD `Unix`
      First 15 pointers of the index block into device directory
      First 12 pointers point to *direct blocks*
      Data for small files do not need separate index block
      Block size of 4K $\Rightarrow$ 48K of data accessed directly
      Next three pointers point to *indirect blocks*
      First indirect block pointer $\equiv$ address of single indirect block
      Index block containing addresses of blocks that contain data
      Second indirect block pointer $\equiv$ *double indirect block pointer*
      Contains address of a block that contains addresses of blocks that contain data
      Third indirect block pointer $\equiv$ *triple indirect block pointer*

Disk Scheduling

- Disk service for any request must be as fast as possible
- Scheduling meant to improve the average disk service time
- Speed of service depends on
  - Seek time, most dominating in most disks
  - Latency time, or rotational delay
  - Data transfer time
- Each disk drive has a queue of pending requests
- Each request made up of
  - Whether input or output
  - Disk address (disk, cylinder, surface, sector)
  - Memory address
  - Amount of information to be transferred – (byte count)
- FCFS Scheduling

  - First Come First Serve scheduling
  - Simplest form of disk scheduling
  - May not provide the best possible service
  - Ordered disk queue with requests on tracks

$$98, 183, 37, 122, 14, 124, 65, 67$$

  - Read/write head initially at track 53
  - Total head movement = 640 tracks
  - Wild swing from 122 to 14 and back to 124
  - Wild swings occur because the requests do not always come from the same process; they are interleaved with requests from other processes

- SSTF Scheduling
  - Shortest Seek Time First scheduling
  - Service all requests close to the current head position before moving the head far away
  - Move the head to the closest track in the service queue
  - Example service queue can be serviced as

$$53, 65, 67, 37, 14, 98, 122, 124, 183$$

  - Total head movement of 263 tracks
  - May cause starvation of some requests
  - Not optimal
    * Consider the service schedule as

$$53, 37, 14, 65, 67, 98, 122, 124, 183$$

    * Total head movement of 208 tracks

- SCAN Scheduling
  - Also called $elevator\ algorithm$ because of similarity with building elevators
  - Head continuously scans the disk from end to end
  - Read/write head starts at one end of the disk
  - It moves towards the other end, servicing all requests as it reaches each track
  - At other end, direction of head movement is reversed and servicing continues
  - Assume head moving towards 0 on the example queue

$$53, 37, 14, 0, 65, 67, 98, 122, 124, 183$$

  - Total head movement of 236 tracks
  - Upper time bound of twice the number of cyliners on any request
  - Few requests as the head reverses direction
  - Heaviest density of requests at the other end

- C-SCAN Scheduling
  - Circular SCAN
  - Variation of SCAN scheduling

  - Move the head from one end to the other
  - Upon reaching the other end, immediately come back to the first end without servicing any requests on the way
- LOOK Scheduling
  - Move the head only as far as the last request in that direction
  - No more requests in the current direction, reverse the head movement
  - *Look* for a request before moving in that direction
  - LOOK and C-LOOK scheduling

Selecting a Disk-Scheduling Algorithm

- Natural appeal in SSTF scheduling
- SCAN and C-SCAN more appropriate for systems that place heavy load on disk
- Performance heavily dependent on number and types of requests
- Requests greatly influenced by file allocation method
  - Contiguously allocated file generates several requests close together on the disk
  - Linked allocation might include blocks that are scattered far apart
- Location of directories and index blocks
  - Directory accessed upon the first reference to each file
  - Placing directories halfway between the inner and outer track of disk reduced head movement

File Systems

- Data elements in file grouped together for the purpose of access control, retrieval, and modification
- Logical records packed into blocks
- File system in Unix
  - Significant part of the Unix kernel
  - Accesses file data using a buffering mechanism to control data flow between kernel and I/O devices
- Directory Structure
  - Files represented by entries in a *device directory*
  - Information in the device directory
    - Name of file
    - Location of file
    - Size of file
    - Type of file
  - Device directory may be sufficient for single user system with limited storage
  - With increase in number of users and amount of storage, a directory *structure* is required
  - Directory structure
    - Provides a mechanism to organize many files in a file system
    - May span device boundaries and may include several different disk units
    - May even span disks on different computers

  - User concerned only with logical file structure
  - Systems may have two separate directory structures
    * Device directory
      Describes the physical properties of each file – location, size, allocation method, etc.
    * File directory
      Describes the logical organization of files on all devices
      Logical properties of the file – name, type, owner, accounting information, protection, etc.
      May simply point to the device directory to provide information on physical properties

Hierarchical Model of the File and I/O Subsystems

- Average user needs to be concerned only with logical files and devices
- Average user should not know machine level details
- Unified view of file system and I/O
- Hierarchical organization of file system and I/O
  - File system functions closer to the user
  - I/O details closer to the hardware
- Functional levels
  - Directory retrieval
    * Map from symbolic file names to precise location of the file, its descriptor, or a table containing this information
    * Directory is searched for entry to the referenced file
  - Basic file system
    * Activate and deactivate files by opening and closing routines
    * Verifies the access rights of user, if necessary
    * Retrieves the descriptor when file is opened
  - Physical organization methods
    * Translation from original logical file address into physical secondary storage request
    * Allocation of secondary storage and main storage buffers
  - Device I/O techniques
    * Requested operations and physical records are converted into appropriate sequences of I/O instructions, channel commands, and controller orders
  - I/O scheduling and control
    * Actual queuing, scheduling, initiating, and controlling of all I/O requests
    * Direct communication with I/O hardware
    * Basic I/O servicing and status reporting

Consistency Semantics

- Important criterion for evaluation of file systems that allows file sharing
- Specifies the semantics of multiple users accessing a file simultaneously
- Specifies when modifications of data by one user are observable by others
- File session
  - Series of accesses between an open and close operation by the same user on the same file

- Unix Semantics
  - Writes to an open file by a user are visible immediately to other users that have this file open at the same time
  - There is a mode of sharing where users share the pointer of current location into the file. Advancing of pointer by one user affects all sharing users. A file has single image that interleaves all accesses, regardless of their origin

File Protection

- Save the file from
  - Physical damage – Reliability
    - Damage possible because of
      Hardware problems – error in read/write
      Power surge or failure
      Head crash
      Dirt and temperature
      Vandalism
      Accidental deletion
      Bugs in file system software
    - Duplicate copies of files
    - File backup at regular intervals
  - Improper access – Protection
    - Physical removal of floppies and locking them up
    - Problem in large system due to need to provide shared access to the files
    - Extremes
      Provide complete access by prohibiting access
      Provide free access with no protection
    - Controlled access
      Access by limiting the types of possible file accesses
      Read access
      Write access
      Execute access
      Append access
      Delete access
      Rename
      Copy
      Edit
    - Protection for directories
      Create a file in the directory
      Delete a file in the directory
      Determine the existence of a file in the directory
- Protection associated with
  - File by itself
  - Path used to access the file
  - With numerous path names, a user may have different access rights to a file dependent upon the path used
  - Protection based on names
    - If a user cannot name a file, he cannot operate on it
  - Protection based on passwords
    - Associate a password with each file

    - Access to each file controlled by password
    - Distinct password with each file – too many passwords to remember
    - Same password for all files – once password broken, all files accessible
    - Associate password with subdirectories (TOPS 20)
    - Multiple level passwords
- Protection based on access lists
    - Associate access list with each file containing names of users and types of accesses allowed
    - Problems with access lists
      Constructing access lists is tedious
      List of users to be allowed certain access may not be known in advance
      Space management problem in directory entry to account for variable list size
- Protection based on access groups
    - Classify users into groups
      Owner
      Group
      Universe
    - Requires strict control of group membership
    - Unix allows groups to be created and managed only by root
    - Only three fields needed to provide protection – rwx