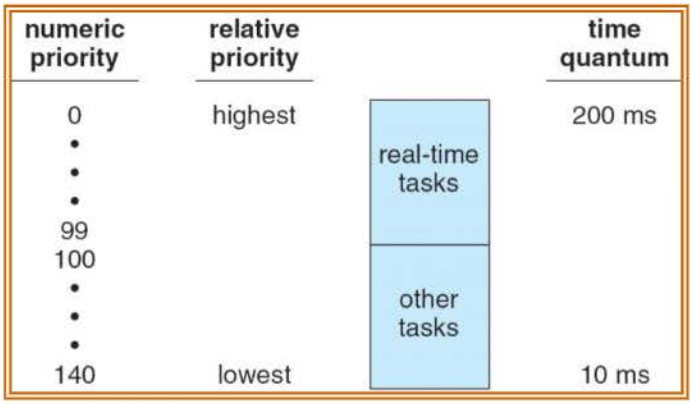
Example: Linux Scheduling

- Two algorithms
  - Time-sharing
    - Prioritized credit-based – process with most credits is scheduled next
    - Credit subtracted when timer interrupt occurs
    - When credit = 0, another process chosen
    - When all processes have credit = 0, re-crediting occurs based on factors including priority and history
  - (Soft) Real-time
    - Posix.1b compliant – two classes: FCFS and RR
    - Highest priority process always runs first

Relationship Between Priorities and Time-Slice Length



LECTURE P5: DEADLOCKS

Outline

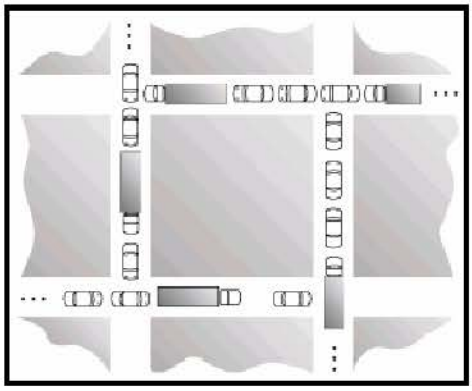
- The deadlock problem
- System model
- Deadlock characterization
- Methods for handling deadlocks
- Deadlock prevention, avoidance and detection/ recovery
- Summary

Objectives

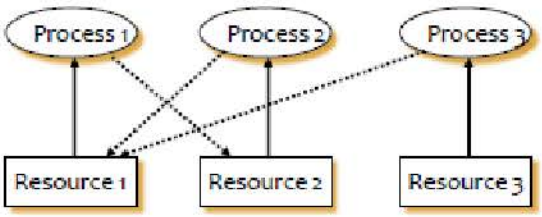
- To develop a description of deadlocks, which prevent sets of concurrent processes from completing their tasks
- To present a number of different methods for preventing deadlocks in a computer system.

Deadlock

- Permanent blocking of a set of processes that either compete for system resources or communicate with each other
- No efficient solution
- Involve conflicting needs for resources by two or more processes



General Problem: A Deadly Embrace



New Problem: Dining Philosophers

- Philosophers think and ignore food
- When they want to eat, need two forks
- Must acquire forks sequentially to match computing model
    - Pick up one and then another
- How do we prevent two philosophers holding same fork at same time?

First Try Solution: Deadlock

```
philosopher(int i) {
  while(TRUE) {
    // Think
    // Eat
    P(spork[i]);
      P(spork[(i+1) mod 5]);
        eat();
      V(spork[(i+1) mod 5]);
    V(spork[i]);
  }
}
semaphore spork[5] = (1,1,1,1,1);
fork(philosopher, 1, 0);
fork(philosopher, 1, 1);
fork(philosopher, 1, 2);
fork(philosopher, 1, 3);
fork(philosopher, 1, 4);
```

Nesting Semaphore Operations

```
pr P Operation      ps P Operation
Order               Order

P(mutex1);          P(mutex2);
  P(mutex2);          P(mutex1);
    <access R1>;        <access R1>;
    <access R2>;        <access R2>;
  V(mutex2);          V(mutex1);
V(mutex1);          V(mutex2);

        (a)                 (b)
```

One Solution

```
philosopher(int i) {
  while(TRUE) {
    // Think
    // Eat
    j = i mod 2;
    P(spork[(i+j) mod 5]);
      P(spork[(i+1-j) mod 5]);
        eat();
      V(spork[(i+1-j) mod 5]);
    V(spork[[(i+j) mod 5]);
  }
}
semaphore spork[5] = (1,1,1,1,1);
fork(philosopher, 1, 0);
fork(philosopher, 1, 1);
fork(philosopher, 1, 2);
fork(philosopher, 1, 3);
fork(philosopher, 1, 4);
```

Addressing Deadlock

- Deadlock is global condition!
- Need to analyze all processes that need all resources
- Can't make local decision based on needs of one process
- Four deadlock approaches:
    - Prevention: never allow deadlock to occur
    - Avoidance: system makes decision to head off future deadlock state
    - Detection and recovery: check for deadlock (periodically or sporadically), then recover
    - Manual intervention: operator reboot if system seems too slow

Prevention: A First Look

- Design the system so that deadlock is impossible
- Deadlock only occurs if all following TRUE!!
    - Mutual exclusion: allocated resources are exclusive property of process
    - Hold and wait: process can hold resource while waiting for another resource
    - Circular waiting: e.g. dining philosophers
    - No preemption: only process can release resources or withdraw resource request
- All four necessary for deadlock to exist
- Prevention requires resource manager to violate at least one condition at all times!

Avoidance

- Construct a formal model of system states
- Via model, choose a strategy that will not allow the system to go to a deadlock state
- Predictive approach: requires processes to declare intent regarding resources in advance
    - Represents "maximum claim" on resources
    - Process won't proceed until all resources available
    - May require "long" waits
- Amenable to formal solution/algorithm

Detection and Recovery

- Dominant commercial solution; when deadlock occurs, can we detect and recover?
- Two phases to algorithm
    - Detection: is there deadlock?
    - Recovery: preempt resources from processes
- Detection algorithm
    - When is it executed?
        - Too often - wastes resources
        - Too infrequent - blocked processes don't do enough work
    - What is its overhead?

Manual Intervention

- Commercial systems think deadlock happens too infrequently to design the O.S. to address it
- Up to users and operator to detect and cope

Prevention

Prevention

- Necessary conditions for deadlock
    - Mutual exclusion
        - Must hold at all times
        - Some resources (tape drive) must be exclusively held by a process
    - Hold and wait
    - Circular waiting
    - No preemption
- Ensure that at least one of the necessary conditions is false at all times

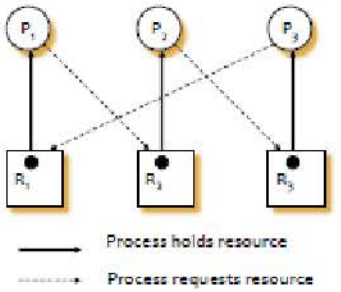
Hold and Wait

- Invalidate hold and wait: process cannot hold resource while waiting for another resource
- Approach 1: targeted to batch systems
  - Process must request all resources it needs
  - Process competes for all resources even if needs only one resource at time
  - Holds resources "done" with
- Approach 2: targeted to timesharing
  - For process to acquire a resource
    - Must release all held resources
    - Reacquire all (released and new) resources needed
  - Overhead to reacquire held resources
- Low resource utilization
- Either could encourage starvation

A Formal Model of System States

- To proceed further must construct a better mathematical model
- Assume
  - P = {p1, p2, …, pn} is a set of processes
  - R = {R1, R2, …, Rm} is a set of resources
  - CPU cycles, memory space, I/O devices, etc.
  - cj = number of units of Rj in the system
  - S = {S0, S1, …} is a set of states representing the assignment of Rj to pi
- State changes whenever processes take action; this allows us to identify
- Situations where processes are blocked (e.g. require another process to do something)
- Deadlock situation in the operating system
- Formal model based on state-transition diagram

One Simple Process-Resource Model State

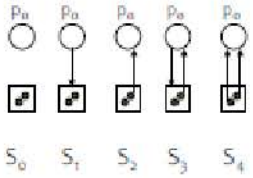


State Transitions

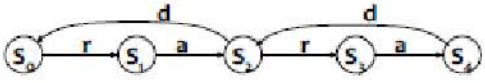
- System changes state due to action of process pi
- There are three pertinent actions:
  - Request:
    - Request one or more units of a resource
    - Transition between states labeled "ri"
  - Allocation (use):
    - All outstanding process requests for a given resource are satisfied (transition "ai")
  - De-allocation (release):
    - Process releases units of a resource (transition "di")

Example

- One process, two units of one resource
- Five possible states

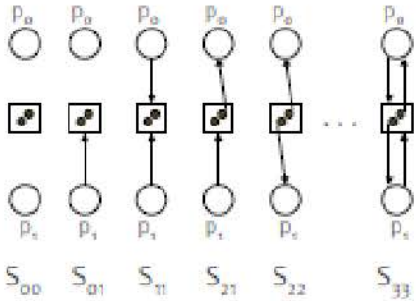


- Can request one unit at a time
- r for request, a for allocated, d for de-allocate
- Two alternatives
- Process can request, allocate, de-allocate twice
- Process can request, allocate, request, allocate, de-allocate twice
- Eventually returns to state S0

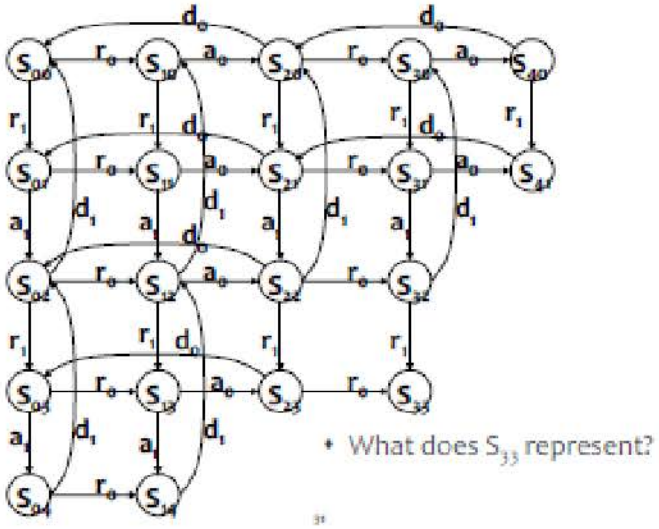


Extension of Example

- Consider two processes competing for two units of single resource
- Process limited to requesting one unit at time
- Process can't ask for more than two units
- How is prior state diagram restructured?
    - Let Sij refer to P0 in Si and P1 in Sj

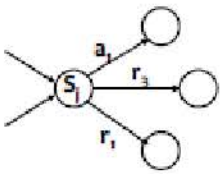


    - All states not feasible
        - S24, S34, S42, S43, S44
        - E.g., S24 - P0 has 1 unit, P1 has 2
- What does resulting state diagram resemble?



Properties of States

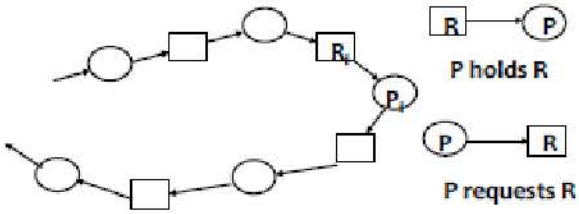
- Define deadlock in terms of patterns of transitions
- Define: pi is blocked in Sj if pi cannot cause a transition out of Sj
- If pi is blocked in Sj, and will also be blocked in every Sk reachable from Sj, then pi is deadlocked
    - Sj is called a deadlock state for pi



- Why is p2 blocked in Sj ?
    - p2 can't leave on own; requires action by either p1 or p3
    - Can't control destiny!

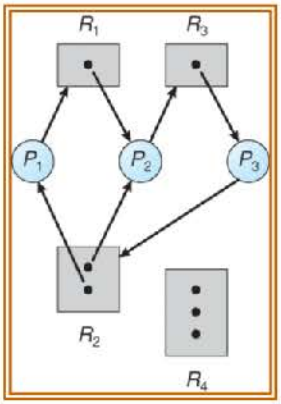
Circular Wait

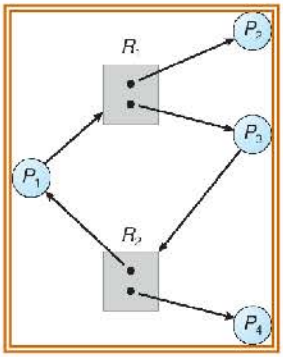
- Classic problem:
  - K processes holding units of K resources
  - Each process wants held resource from other processes
- Search for cycles in resource allocation graph; deadlock identifiable whenever there is a cycle in the graph of processes and resources
  - No cycle no deadlock; otherwise
    - If only one instance per resource type, deadlock.
    - If multiple instances possibly a deadlock
- What resource request strategy guarantees no cycles (i.e., invalidates circular wait)?
  - Total order of all system resources
- Formally, a process can only ask for Rj if Ri < Rj for all Ri the process is currently holding
  - No need to request all at once
  - However, can't go back to get one "missed"



Deadlocked resource allocation graph

Resource allocation graph with cycle but no deadlock

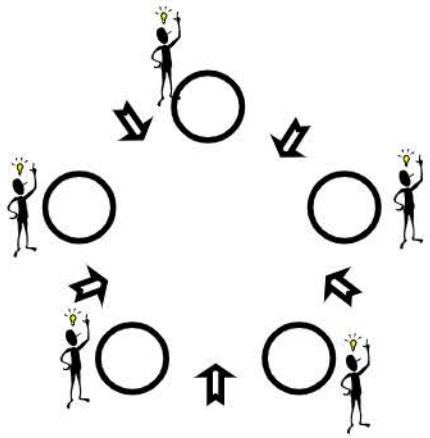




Revisiting Dining Philosophers Problem

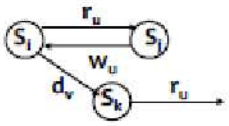
```
philosopher(int i) {
  while(TRUE) {
    // Think then Eat
    P(spork[i]);
      P(spork[(i+1) mod 5]);
        eat();
      V(spork[(i+1) mod 5]);
    V(spork[i]);
  }
}
semaphore spork[5]=(1,1,1,1,1);
fork(philosopher, 1, 0);
fork(philosopher, 1, 1);
fork(philosopher, 1, 2);
fork(philosopher, 1, 3);
fork(philosopher4, 0);
```

```
philosopher4() {
  while(TRUE) {
    // Thinking ...
    P(spork[0]); P(spork[4]);
      eat();
    V(spork[4]); V(spork[0]);
  }
}
```



Allowing Preemption

- Invalidate preemption: allow a process to time-out on a blocked request
- Thus, process withdraws the request if it fails
- Consider state transition diagram below



- Technique prevents deadlock via withdrawal
- Allows other processes that might need ru to go
- May result in livelock where processes are requesting and withdrawing without useful work
- Another possibility
    - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released.
    - Preempted resources are added to the list of resources for which the process is waiting.
    - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

Avoidance and Detection/Recovery

Avoidance

- Requires a multi-phase approach
    - Construct a model of system states
    - Choose a strategy that guarantees that the system will not go to a deadlock state
    - Service processes in some order, not necessarily order received
- Requires extra information for each process
    - Maximum Claim - maximum number of units of every resource process will ever request
- Resource manager sees the worst case and can allow transitions based on that knowledge
- Goal: to maintain "safe" state
    - If a system is in a safe state no deadlock; if in an unsafe state, possibly a deadlock

Safe vs. Unsafe States

- Defining a safe state:
    - Guarantee to be a sequence of transitions that leads back to the initial state
    - Even if all process exercise their maximum claim, there is an allocation strategy by which all claims can be met
- Defining an unsafe state:
    - System cannot guarantee there is such a sequence
    - Unsafe state can lead to a deadlock state if too many processes exercise their maximum claim at once

Banker's Algorithm

- Multiple resource instances
- Each process must a priori claim maximum use
- When a process requests a resource it may have to wait
- When a process gets all its resources it must return them in a finite amount of time

Banker's Algorithm Definitions

- Let
  - maxc[i, j] be a 2-D array that contains the maximum claim for resource Rj by process pi
  - alloc[i, j] be a 2-D array that contains the number of units of resource Rj held by process pi
  - C[j] be units of resource Rj
- Given maxc and alloc, we can always compute:

  $$avail[j] = C[j] - \sum_{0<=i<n} alloc[i,j]$$

  - Represents available units of Rj
  - System resource units available = total units - units held
- Given "available units", is there a process that can execute (i.e., is this a safe state)?
- If so, execute process, de-allocate resources, adjust arrays, recalculate and iterate

Banker's Algorithm

1. Copy the alloc[i,j] table to alloc'[i,j]
2. Given C, maxc, and alloc', compute avail vector
3. Find pi: maxc[i,j] – alloc'[i,j] <= avail[j] for 0 <= j < m and 0 <= i < n
   - If no such pi exists, the state is unsafe - halt
   - If alloc'[i,j] is 0 for all i and j, the state is safe - halt: algorithm done - all processes executed
4. Set alloc'[i,j] to 0; de-allocate all resources held by pi; go to Step 2

Example

- Start with max claim, allocation and C
  - C = < 8, 5, 9, 7 >
- Compute total held
- Find available units (C – alloc)
  - Avail = < 8 – 7, 5 – 3, 9 – 7, 7 – 5 >
    = < 1, 2, 2, 2 >
- Will want everybody's future need (max claim minus allocation)
- Can p0's max claim be met?
  - Process p0 fails on avail[3]
- Can p1's max claim be met?
  - Process p1 fails on avail[2]
- Can p2's max claim be met?
  - Yes! Redo avail and update alloc
  - avail = < 5, 2, 2, 5 >
- Can p0's max claim be met?
  - Yes! Redo avail and update alloc
- Can anyone's max claim be met?
  - Yes, anyone except p3! Choose p1
- After this can anyone's max claim be met?
  - Yes, any of them can!
  - Choose p3 then p4
- Allocations now all zero
- Safe state with all processes completed
- Halt algorithm with success
- Process order of p2, p0, p1, p3, p4
  - Safe execution
  - All maximum claims satisfied in some order
- No deadlock or unsafe state

Maximum Claim

Proc	R_0	R_1	R_2	R_3
P_0	3	2	1	4
P_1	0	2	5	2
P_2	5	1	0	5
P_3	1	5	3	0
P_4	3	0	3	3

Alloc. Resources

R_0	R_1	R_2	R_3
2	0	1	1
0	1	2	1
4	0	0	3
0	2	1	0
1	0	3	0
7	3	7	5

Need

R_0	R_1	R_2	R_3
1	2	0	3
0	1	3	1
1	1	0	2
1	3	2	0
2	0	0	3

Allocated Resources

Proc	R_0	R_1	R_2	R_3
P_0	2	0	1	1
P_1	0	1	2	1
✔P_2	0	0	0	0
P_3	0	2	1	0
P_4	1	0	3	0
Sum	3	3	7	2

Allocated Resources

Proc	R_0	R_1	R_2	R_3
✔P_0	0	0	0	0
P_1	0	1	2	1
✔P_2	0	0	0	0
P_3	0	2	1	0
P_4	1	0	3	0
Sum	1	3	6	1

Second Example

- C = < 10, 5, 7 >, avail = < 3, 3, 2 >
- Safe since the sequence p1, p3, p4, p2, p0 satisfies safety criteria
- Can request for <1, 0, 2> by p1 be granted?
  - p1's alloc now <3, 0, 2>, avail becomes <2, 3, 0>
  - Executing algorithm shows that sequence p1, p3, p4, p0, p2 satisfies safety requirement

Maximum Claim

Proc	R_0	R_1	R_2
P_0	7	5	3
P_1	3	2	2
P_2	9	0	2
P_3	2	2	2
P_4	4	3	3

Alloc. Resources

Proc	R_0	R_1	R_2
P_0	0	1	0
P_1	2	0	0
P_2	3	0	2
P_3	2	1	1
P_4	0	0	2
Sum	7	2	5

Third Example

- C = < 10, 5, 7 >, avail = < 3, 3, 2 >
- Safe since the sequence p3, p1, p4, p2, p0 satisfies safety criteria
- Can request for <1, 0, 2> by p1 be granted?
  - p1's allocation becomes <3, 0, 2>
  - New avail is <2, 3, 0>
  - Nobody can get maximum claim now
  - Therefore request not granted

Maximum Claim

Proc	R_0	R_1	R_2
P_0	7	5	3
P_1	3	2	3
P_2	9	0	3
P_3	2	2	3
P_4	4	3	3

Alloc. Resources

Proc	R_0	R_1	R_2
P_0	0	1	0
P_1	2	0	0
P_2	3	0	2
P_3	2	1	1
P_4	0	0	2
Sum	7	2	5

Detection and Recovery

- Check for deadlock (periodically or sporadically), then recover
- Can be far more aggressive with allocation
- No maximum claim, no safe/unsafe states

Detection Based on Banker's Algorithm

- Let req[i, j] be a 2-D array that contains the outstanding request for resource Rj by process pi
- As before
  - let alloc[i, j] be a 2-D array that contains the number of units of resource Rj held by process pi
  - let C[j] = units of resource Rj
  - let avail[j] = C[j] - $\Sigma_{0<=i<n}$ alloc[i,j]
- Given "available units", is there a process that can execute?
  - If so, execute process, de-allocate resources, adjust arrays, recalculate and iterate

Detection Algorithm

1. Copy the alloc[i,j] table to alloc'[i,j]
2. Finalize any process with no resources allocated to it
3. Given C and alloc', compute avail vector
4. Find pi: req[i,j] <= avail[j] for 0 <= j < m and 0 <= i < n
   - If no such pi exists, the system and any unfinalized processes deadlocked
   - if alloc'[i,j] is 0 for all i and j, halt: algorithm done
     - There is no deadlock - all processes executed
5. Set alloc'[i,j] to 0; de-allocate all resources held by pi; finalize pi; go to step 3

Example

Outstanding Requests

Proc	R_0	R_1	R_2
P_0	0	0	3
✔P_1	1	0	0
✔P_2	1	1	1
P_3	1	2	0

Alloc. Resources

Proc	R_0	R_1	R_2
P_0	0	4	0
✔P_1	0	0	0
✔P_2	1	0	1
P_3	1	1	1
Sum	2	5	2

- C = < 3, 6, 3 >
- p1 holds nothing so finalize
- Compute total held; avail = < 1, 1, 1 >
- p2 can have its request fulfilled; finalize
  - Avail now < 2, 1, 2 >
- Nobody else can get what it wants
- p0 and p3 deadlocked

Recovery

- No magic here
  - Choose a blocked process
  - Preempt it (releasing its resources)
  - Rerun the detection algorithm
  - Iterate until the state is not a deadlock state

--- END PROCESS MANGEMENT UNIT

--- MEMORY MANGEMENT UNIT

LECTURE M1: MAIN MEMORY

Outline

- Background
- Swapping
- Contiguous Memory Allocation

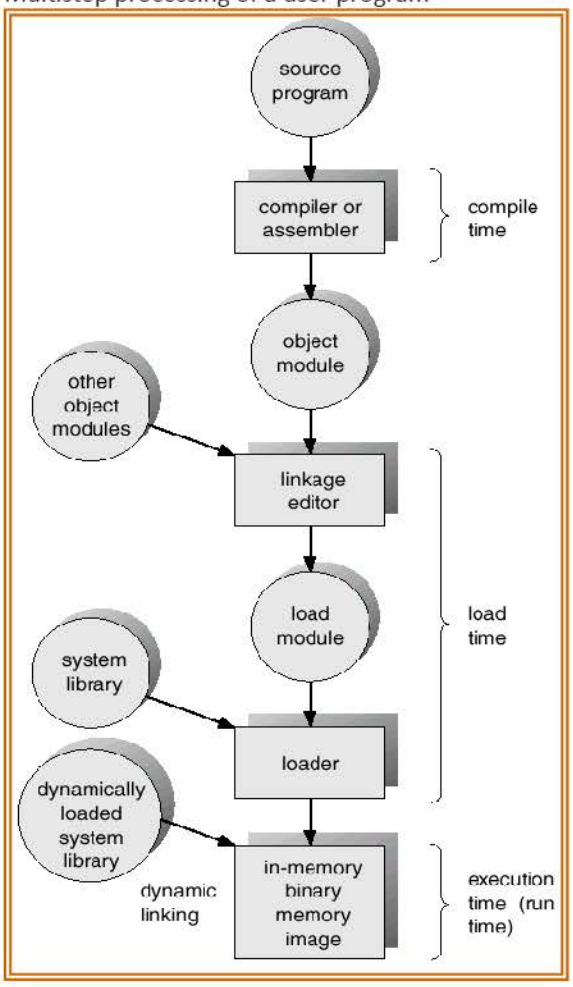
Introduction

- Memory manager requirements
  - Minimize primary memory access time: registers vs. cache vs. primary memory
  - Maximize primary memory size: virtual memory "appearance" of larger memory
  - Primary memory must be cost-effective: doubling of capacity has had greatest impact
- Today's memory manager:
  - Allocates primary memory to processes and map their address spaces
  - Minimizes access time using cost-effective memory configuration

Recall Key Compilation Steps

- Compile: produce relocatable object module
  - Static variables allocated at compile time
  - Non-static variables via run time stack
  - Dynamic variables via heap
  - Recall activation record in C
- Link time: produce absolute (or load) module
  - Collect and combine individual relocatable modules
  - Create absolute module with all addresses in process' address space with respect to base location at address 0000
- Loader: produce executable image for primary memory
  - Readjust addresses for execution
  - Offset from location 0000 must be changed
  - Once partition for process execution has been chosen bind addresses to physical memory locations
    - Referencing both instructions and data
- Dynamic memory for data structures
  - Each address space of process contains "unused" memory
  - User programs utilize for dynamic memory allocation (e.g., malloc and new)

Multistep processing of a user program



Binding of Instructions and Data to Memory

- Address binding of instructions and data to memory addresses can happen at three different stages
  - Compile time: If memory location known a priori, absolute code can be generated
    - Must recompile code if starting location changes
  - Load time: Must generate relocatable code if memory location is not known at compile time
  - Execution time: Binding delayed until run time if the process can be moved during its execution from one memory segment to another.
    - Need hardware support for address maps (e.g., base and limit registers).

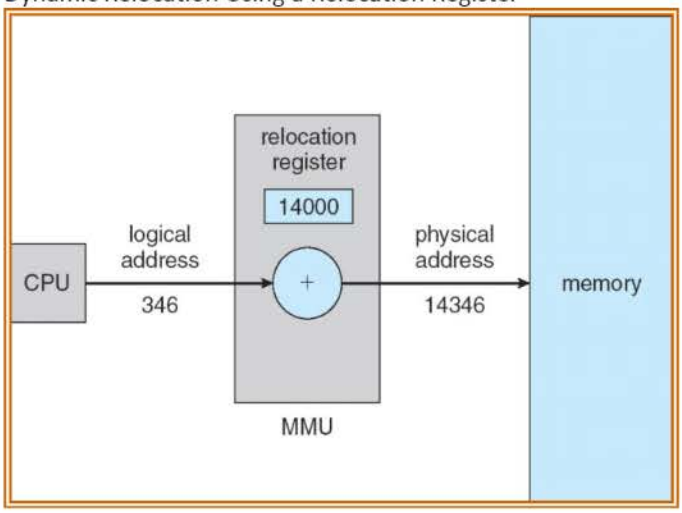
Logical vs. Physical Address Space

- The concept of a logical address space that is bound to a separate physical address space is central to proper memory management
  - Logical address – generated by the CPU; also referred to as virtual address
  - Physical address – address seen by the memory unit
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; they differ in execution-time address-binding scheme

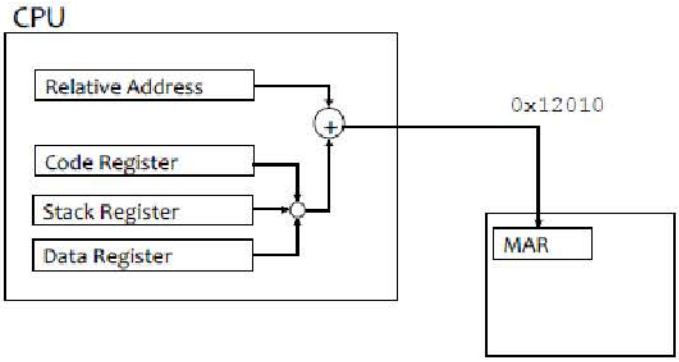
Memory Management Unit (MMU)

- Hardware device that maps virtual to physical address during run-time binding
- In MMU scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
- The user program deals with logical addresses; it never sees the real physical addresses

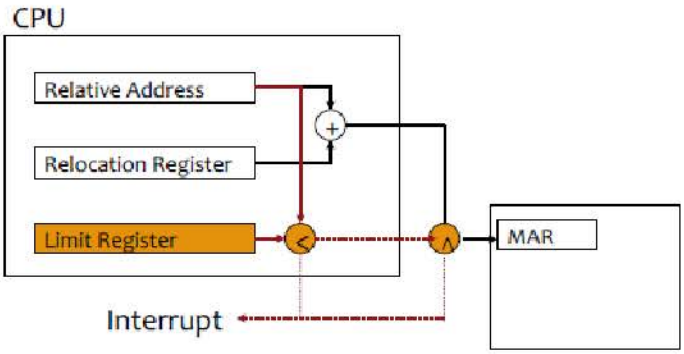
Dynamic Relocation Using a Relocation Register



Multiple Segment Allocation Registers



Also…Runtime Bound Checking



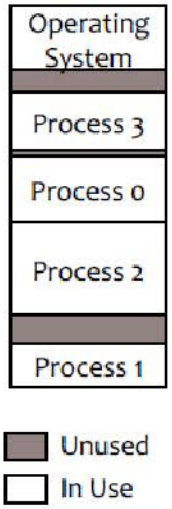
Bound checking is inexpensive to add and provides excellent memory protection

Memory Allocation

- Subdividing memory to accommodate multiple processes
- Memory needs to be allocated efficiently to pack as many processes into memory as possible.
- Two types of schemes:
  - Contiguous allocation: each process stored as a whole in a contiguous range of memory addresses
  - Non-contiguous allocation – later

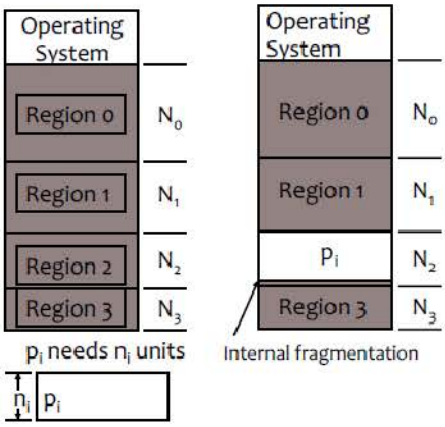
Contiguous Memory Allocation

- Main memory usually into two partitions:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory
- Relocation-register scheme used to protect user partitions from each other, and from changing operating-system code and data
  - Relocation register contains value of smallest physical address
  - Limit register contains range of logical addresses – each logical address must be less than the limit register
- Divide high memory into N fixed regions for user processes
- Memory manager must know maximum memory required by each process
- It then employs space-multiplexed sharing
- For example the figure at right employs four-way multiprogramming
  - Divides memory into at most four blocks + O.S. block
  - Results in unused space
- Need a mechanism/policy for loading process' address space into primary memory
- Key problem regardless of memory allocation strategy: fragmentation!



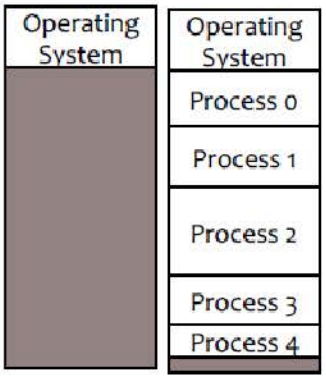
Fixed-Partition Memory – Best-Fit

- N fixed sized regions of varying sizes
- Provide
  - Small regions for small processes
  - Larger regions for larger processes
- Process loaded into any empty region with enough space
- Region larger than process
  - Internal fragmentation
- Assume each region maintains queue of processes competing for partition
  - "Large", "Small", "Medium", etc... process queues
- Loader prepares process for execution, allocator ...
  - Puts process in queue for region
  - Minimizes (region size minus process size)
- What are potential problems?
  - Popular sized queues can overfill
  - Lack of fairness
    - Large, Medium "popular"
    - Small job queue "empty"
    - Later small jobs execute before earlier medium or large jobs
- Not suited to timesharing systems

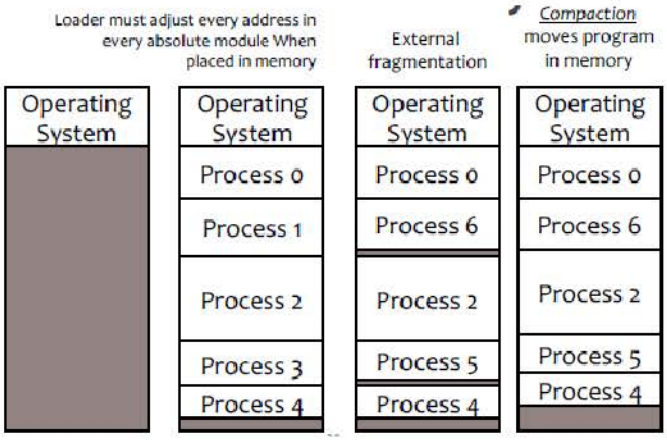


Variable-Partition Memory Strategies

- O.S. flexible in placing processes in memory
- Allocate memory in multiword blocks (1K, 2K, etc.)
    - Minimal internal fragmentation
- Live with external fragmentation at "end" of memory



Snapshort of Variable Partition Memory



Variable-Partition Memory Allocation

- Allocate primary memory "holes" from free list according to
    - Best fit
    - Worst fit
    - First fit
    - Next fit
- First-fit and best-fit better than worst-fit in terms of speed and storage utilization
- Allocation strategies differ in addressing fragmentation

Variable-Partition Memory Allocation – Best Fit

- Same as fixed approach; allocator places process in smallest block of available memory
- Suppose
    - 12K process
    - 6K, 14K, 19K, 11K, 13K blocks
- Which block would allocator choose?

Variable-Partition Memory Allocation – Worst Fit

- Choose largest memory block available for pi
- Create largest hole which is then available for allocation
- Suppose
    - 12K process
    - 6K, 14K, 19K, 11K, 13K blocks
- Which block would allocator choose?
- What is result of choice?
- Split region to create new region for process to execute

Variable-Partition Memory Allocation – First Fit

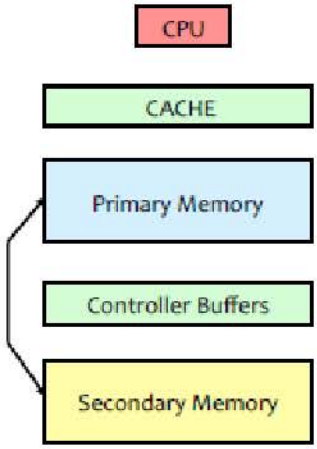
- Avoid traversal of free list by choosing first available block
- Free list could be long in practice; reduce search choice
- Suppose
    - 12K process
    - 6K, 14K, 19K, 11K, 13K blocks
- Which block would allocator choose?
- Fragment blocks at front end of list – end of list unused

Variable-Partition Memory Allocation – Next Fit

- Variation on first fit
- Convert free list into circular list
- Begin next search where previous one left off
- Suppose
  - 12K Process
  - 6K, 14K, 19K, 11K, 13K Blocks
- Which block would allocator choose?
- Where is pointer now?

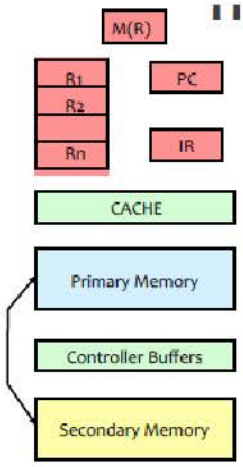
Swapping

- Suppose there is high demand for executable memory
  - Many processes need to execute
  - Assume time sharing/time quantum
  - What if all processes won't fit into main memory?
- Equitable policy might be to time-multiplex processes into the space-multiplexed memory
- Means that process can have its address space unloaded even if it still needs memory
  - Can happen when it is blocked (i.e., waiting for I/O to finish)
  - Also - if user idle on Unix, could swap out Firefox, mail, editor, etc.
- Swapping requires the ability to remove and then restart executable program at same point
- Remove means, for process Pi
  - Swap Pi from primary memory to secondary
  - Allow process Pi to be idle
  - Prior to Pi's quantum, swap Pi from secondary memory to primary
  - Key: keep CPU busy - no waiting for either swap!
- What is available to assist in this activity?



Swapping and Memory Hierarchy

- What necessitates swapping?
  - Primary memory is full
  - New process Pn Starts
  - One or more processes must be moved for Pn to execute
- Move current (active) process
  - Why? Quantum just finished so longest wait for next quantum
    - Assumes equal priorities
  - Swap process/state to secondary memory



LECTURE M2: VIRTUAL MEMORY

Outline

- Background
- Paging and Demand Paging
- Structure of the Page Table
- Page Replacement
- Allocation of Frames
- Thrashing
- Operating System Example

Objectives

- To describe the benefits of a virtual memory system
- To explain the concepts of demand paging, page-replacement algorithms, and allocation of page frames
- To discuss the principle of the working-set model

Background

- Recall: Two types of memory allocation schemes:
  - Contiguous allocation -- just saw
  - Non-contiguous allocation: divide process into pieces, each of which is stored in a different area of memory
- Virtual memory – separation of user logical memory from physical memory.
  - Only part of the program needs to be in memory for execution.
  - Logical address space can therefore be much larger than physical address space.
- Virtual memory – separation of user logical memory from physical memory.
  - Allows address spaces to be shared by several processes and for more efficient process creation.
- Works because every process has code and data locality
  - Code tends to execute in a few fragments at one time (spatial locality)
  - Tend to reference same set of data structures (temporal locality)
- Dynamically load/unload currently-used address space fragments as the process executes
- Uses dynamic address relocation/binding
  - Generalization of base-limit registers
  - Physical address corresponding to a compile-time address is not bound until run time
- Virtual memory can be implemented via demand paging or demand segmentation

Virtual Memory: Address Binding

- Use a dynamic virtual address map, Yt
- For element i (page i), examine Yt(i)
  - If result is null address Ω then generate interrupt since element in secondary memory; load address, set Yt(i) = k
  - Else, map to physical address Yt(i) = k

Size of Blocks of Memory

- Virtual memory system transfers "blocks" of the address space to/from primary memory
- Fixed size blocks: System-defined pages are moved back and forth between primary and secondary memory
- Variable size blocks: Programmer-defined segments – corresponding to logical fragments – are the unit of movement
- Paging is the commercially dominant form of virtual memory today

Paging

- Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
- Paging reduces external fragmentation
- Divide physical memory into fixed-sized blocks called frames (size is power of 2, between 512 bytes and 8192 bytes)
- Divide logical memory into blocks of same size called pages.
- When a virtual address in page i is referenced by the CPU
  - If page i is loaded at page frame j, the virtual address is relocated to page frame j
  - If page i is not loaded, the O.S. interrupts the process and loads the page into a page frame (page fault)
- To run a program of size n pages, need to find n free frames and load program
  - Thus MMU must keep track of all free frames
- Set up a page table to translate logical to physical addresses

Page Tables

- Each process has its own page table
  - Each entry contains the frame number of the corresponding page in main memory
  - A valid-invalid bit is needed to indicate whether the page is in main memory or not
  - Another modify bit is needed to indicate if the page has been altered since it was last loaded into main memory
    - If no change has been made, the page does not have to be written to the disk when it needs to be swapped out

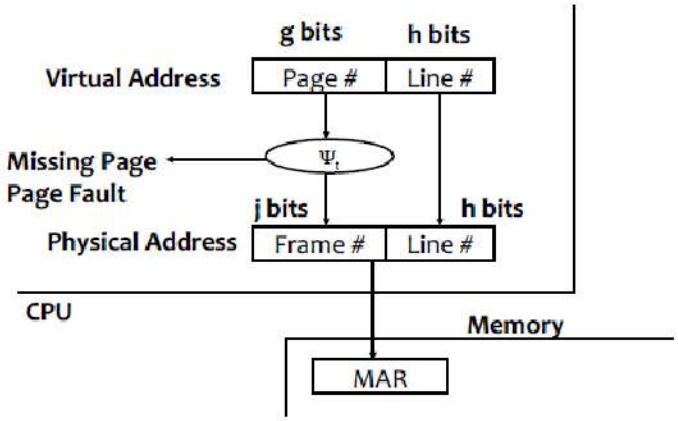
Capabilities of a Paging System

- Translate virtual address in Yt(i) to physical address <pageNumber, offset>
  - Page number (p) – function Yt(i) used as an index into a page table which contains base address of each frame in physical memory
  - Page offset (d) – combined with base address to define the physical memory address that is sent to the memory unit
- Dynamically bind pages (virtual) to page frames (real) as part of address translation and page loading process

Addresses

- Suppose there are $G = 2^g x 2^h = 2^{g+h}$ virtual addresses and $H = 2^{j+h}$ physical addresses assigned to a process
- Each page/page frame is 2^h addresses
- There are 2^g pages in the virtual address space
- 2^j page frames are allocated to the process
- Rather than map individual addresses
  - Yt maps the 2^g pages to the 2^j page frames
  - That is, $page\_frame_j = Y_t(page_i)$
  - Address k in $page_i$ corresponds to address k in $page\_frame_j$

Address Translation



Page-Based Address Translation

- Let N = {d0, d1, ... dn-1} be the pages ($n=2^g$).
- Let M = {b0, b1, ..., bm-1} be the page frames ($m=2^j$).
- Virtual address i satisfies $0 <= i < 2^{g+h}$
- Physical address k = $Yt(U) \times 2^h + V$ ($0 <= V < 2^h$)
  - U is page number, V is the line number within the page
  - Since every page is size $c=2^h$
    - page number = U = floor(i/c), line number = V = i mod c
  - Yt:[0, n-1] -> [0, m-1] U {Ω}

Example of Address Translation

- Let 0x3B9 be the virtual address and c=0x100 the page size
- Page number = U = 3
- Line number = V = 0xB9
- Obtain (in this example) Yt(3) = 4
- If Yt(3) = Ω then load page from secondary storage and continue with address translation
- Physical address is Yt(floor(i/c)) x c + i mod c
- Yt(3) x 0x100 + 0xB9 = 0x400 + 0xB9 = 0x4B9 physical address

Ψ_t
2
5
0
4

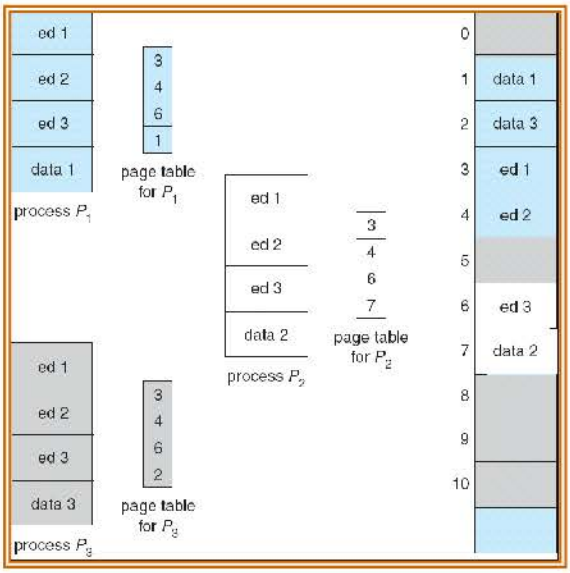
Handling Page Faults

- O.S. algorithm as follows:
  - Interrupt process containing missing page
  - Memory manager locates missing page in secondary storage
  - Load missing page into primary memory thereby (possibly) displacing (unloading) another page
  - Update page table to reflect both new and removed pages
  - CPU re-executes instruction during next time slice of process

Shared Pages

- Shared code
  - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
  - Shared code must appear in same location in the logical address space of all processes
- Private code and data
  - Each process keeps a separate copy of the code and data
  - The pages for the private code and data can appear anywhere in the logical address space

Shared Pages Example



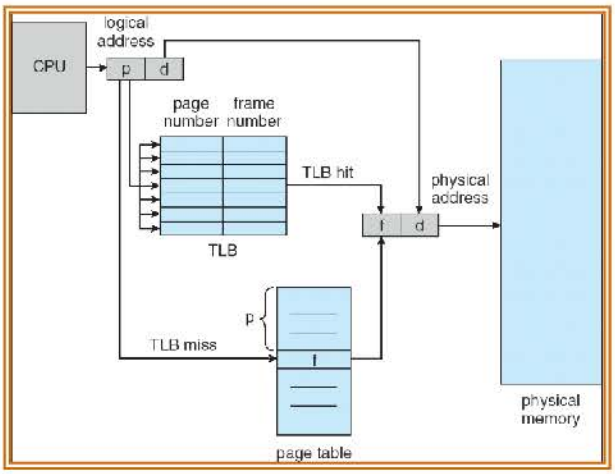
Implementation of Page Tables: Access Speed

- Ideally the page table is kept in main memory
- Each virtual memory reference can cause two physical memory accesses
  - One to fetch the page table, one to fetch the data
- To overcome this problem a high-speed cache is set up for page table entries
  - Called associative registers or the translation look-aside buffer (TLB)
- Contains page table entries that have been most recently used

Translation Look-Aside Buffer

- Functions same way as a memory cache.
  - Given a virtual address processor examines the TLB via parallel search
  - If a page table entry is present, the frame number is retrieved and the real address formed
  - If a page table entry is not found, the page number is used to index the process page table in memory
    - First check to see if page already in main memory; if not a page fault is issued.
    - The TLB is updated to include the new page entry.

Paging Hardware with TLB



Implementation of Page Tables: Size

- Ideally the page table is kept in main memory
- However the entire page table may take up too much main memory.
- One solution: page tables are also stored in virtual memory.
  - When a process is running, only part of its page table is in main memory in this case.
- Another is to creatively structure the page table in main memory.

Page Table Structure

- Hierarchical Paging
  - Break up the logical address space into multiple page tables
  - A simple technique is a two-level page table
- Hashed Page Tables
- Inverted Page Tables

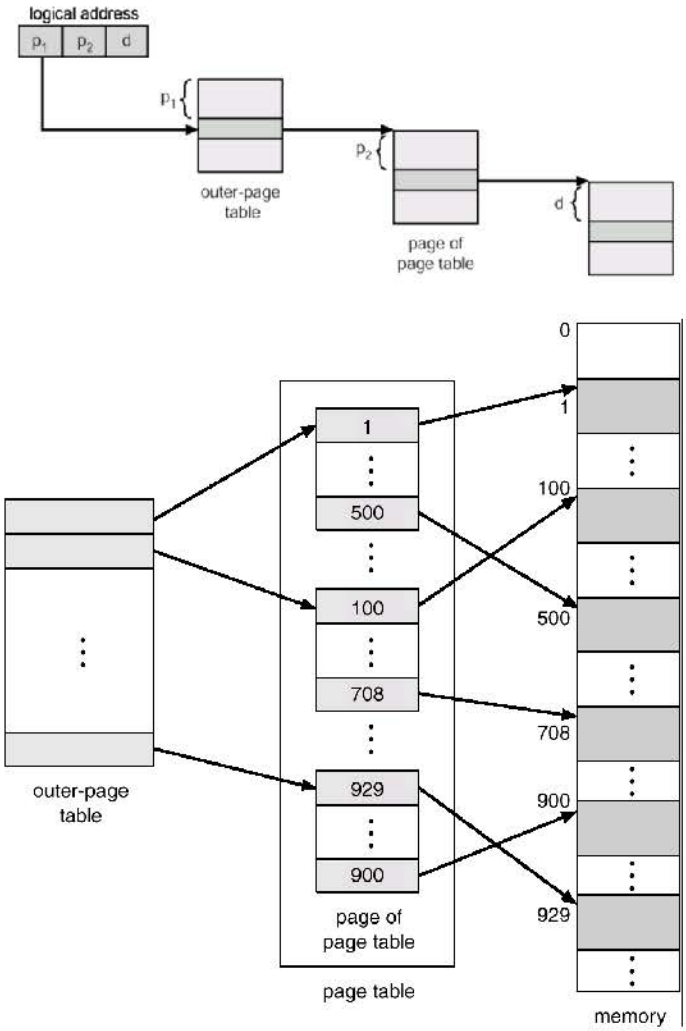
Ex: Two-Level Page Table Scheme

- Assume a 32-bit machine with 4K page size.
- A logical address is divided into a 20-bit page number and a 12-bit page offset
- Since the page table is paged, the page number is further divided into a 10-bit page number and a 10-bit page offset
- Each level stored as a separate table in memory
  - May take four memory accesses to convert a logical address to a physical one.
- Thus a logical address is as shown below.

page number		page offset
p_i	p_2	d
10	10	12

  - p1 is an index into the outer page table
  - p2 is the displacement within the page of the outer page table

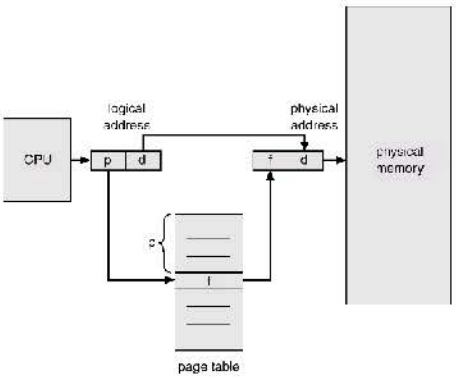
Address Translation Scheme



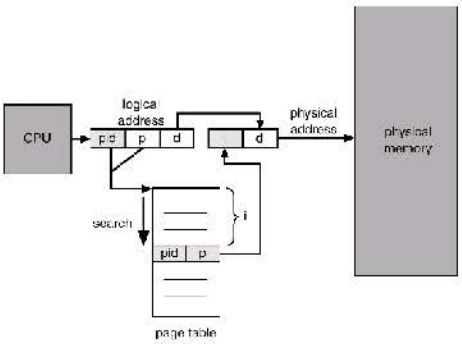
Inverted Page Table

- One entry for each real page of memory
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
- Use hash table to limit the search to one – or at most a few – page-table entries

Address Translation
Traditional Page Table



Address Translation
Inverted Page Table



Paging Algorithms and Policies

- Static paging: each process allocated a fixed number of physical pages when started
- Dynamic paging: total pages can change during execution
- Three basic policies:
  - Fetch: which page should be loaded into primary memory?
  - Replacement: which page removed if all page frames are full?
  - Placement: where is fetched page loaded in primary memory?
- Static algorithms focus on replacement
  - Placement not relevant for static paging
  - Fetch is determined by process execution

Modeling Page Behavior

- Let w = r1, r2, r3, …, ri, … be a page reference stream
  - ri is the ith page number referenced by the process
  - Subscript is the virtual time for the process
- Given a page frame allocation of m, the memory state at time t, $S_t(m)$, is set of pages loaded
  - $S_t(m) = S_{t-1}(m) \cup X_t - Y_t$
    - X_t is the set of fetched pages at time t
    - Y_t is the set of replaced pages at time t

More on Demand Paging

- Demand fetch policy
  - If r_t was loaded at time t-1, $S_t(m) = S_{t-1}(m)$
  - If r_t was not loaded at time t-1 and there were empty page frames, $S_t(m) = S_{t-1}(m) \cup \{r_t\}$
    - No need to remove a page
  - If r_t was not loaded at time t-1 and there were no empty page frames, $S_t(m) = S_{t-1}(m) \cup \{r_t\} - \{y\}$
    - Must remove some page y
- Alternative is prefetching which attempts to predict which pages can be staged into main memory
  - This is difficult to achieve in practice!

Assumptions for Static Allocation with Demand Paging

- Replacement Policy
  - Since $S_t(m) = S_{t-1}(m) \cup \{r_t\} - \{y\}$, the replacement policy must choose y
  - This uniquely identifies the paging policy

- We'll explore following replacement policies
  - Belady's optimal algorithm, LRU, and FIFO
  - Also random and LF(requently)U

Belady's Optimal Algorithm

- Replace page with maximal forward distance: $y_t = \max_{x \in S_{t-1}(m)} FWD_t(x)$
- Optimality requires complete knowledge of w
- Look ahead in reference for page that won't be needed for longest time and replace that page

Let page reference stream, ω = 2031203120316457

Frame	2	0	3	1	2	0	3	1	2	0	3	1	6	4	5	7
0	2	2	2	2	2	2	2	2	2	0	0	0	0	4	4	4
1		0	0	0	0	0	3	3	3	3	3	3	6	6	6	7
2			3	1	1	1	1	1	1	1	1	1	1	5	5	5

10 page faults

- Perfect knowledge of w -> perfect performance
- Impossible to implement
- Provides baseline for comparison

First In First Out (FIFO)

- Replace page that has been in memory the longest: $y_t = \max_{x \in S_{t-1}(m)} AGE_t(x)$
- May replace just used page if there longest!

Let page reference stream, ω = 2031203120316457

Frame	2	0	3	1	2	0	3	1	2	0	3	1	6	4	5	7
0	2	2	2	1	1	1	3	3	3	0	0	0	6	6	6	7
1		0	0	0	2	2	2	1	1	1	3	3	3	4	4	4
2			3	3	3	0	0	0	2	2	2	1	1	1	5	5

Complete to yield **16 page faults**

- Track replacement w.r.t future pages – is it unfair?

Least Recently Used (LRU)

- Replace page With maximal backward distance: $y_t = \max_{x \in S_{t-1}(m)} BKWD_t(x)$
- Which page hasn't been used "recently"?

Let page reference stream, ω = 2031203120316457

Frame	2	0	3	1	2	0	3	1	2	0	3	1	6	4	5	7
0	2	2	2	1	1	1	3	3	3	0	0	0	6	6	6	7
1		0	0	0	2	2	2	1	1	1	3	3	3	4	4	4
2			3	3	3	0	0	0	2	2	2	1	1	1	5	5

16 page faults

Example of Thrashing

- Swapping out a piece of a process just before that piece is needed
- If a process' working set is not in memory, it will produce a page fault every few instructions and is said to be thrashing
  - The processor spends most of its time swapping pieces rather than executing instructions
- Solution: the system must increase the frame allocation for the process
- If all processes have the same problem then the system is thrashing
  - CPU utilization gets to zero

E.g. Windows on a PC with 8 MB memory

Previous Example

3 frames, 16 faults

Frame	2	0	3	1	2	0	3	1	2	0	3	1	6	4	5	7
0	2	2	2	1	1	1	3	3	3	0	0	0	6	6	6	7
1		0	0	0	2	2	2	1	1	1	3	3	3	4	4	4
2			3	3	3	0	0	0	2	2	2	1	1	1	5	5

4 frames, 8 faults

Frame	2	0	3	1	2	0	3	1	2	0	3	1	6	4	5	7
0	2	2	2	2	2	2	2	2	2	2	2	2	6	6	6	6
1		0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
2			3	3	3	3	3	3	3	3	3	3	5	5	5	5
3				1	1	1	1	1	1	1	1	1	1	7	7	7

Implementation of LRU

- LRU has become preferred algorithm of choice
    - Good predictor of program behavior
    - Works well on different page reference streams
- Implementation considerations
    - Maintain page table Yt that contains entries for only "active" pages of process
    - Accounting on each page frame must be maintained
    - Must record when each page was referenced
        - When was last virtual time referenced?
        - Costly to implement in practice
    - Difficult to do in hardware
- Just saw counter implementation
    - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
    - When a page needs to be changed, look at the counters to determine which are to change
- Stack implementation – keep a stack of page numbers in a double link form:
    - When a page is referenced move it to the top
        - requires some pointers to be changed
    - No search for replacement

Approximate LRU with a Reference Bit

- Maintained for each entry of page table
    - Initially set to zero for all entries
    - Periodically reset to zero by hardware
    - Set for a page when it is referenced
- When page fault occurs
    - Algorithm inspects reference bit for all pages
    - For example suppose pages 0, 9, and 19 least referenced
    - Algorithm randomly chooses among them
- Cost of page fault
    - Writing primary page to secondary memory
    - Employ "dirty bit" for each page frame
    - Only write "dirty" pages to secondary store

Summary: Static Paging Algorithms

- Pros:
    - Fairness across processes
    - Allocated memory same regardless of process needs
    - Increases number of processes that can execute in fixed size primary memory
- Cons:
    - Potential for thrashing
    - Unfairness for processes that need more page frames than possible with size of m
    - Potential for performance degradation with poorly chosen m

Dynamic Paging Algorithms

- Amount of physical memory allocated varies as the process executes
    - Over time, page frames allocated can increase and decrease based on demand
        - Will change according to the phase of process
    - Fault rate must be "tolerable"
- Need to define placement and replacement policies
- Must consider the "locality" of process's behavior over time

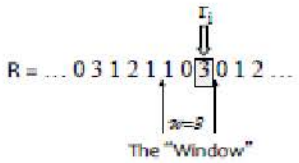
Working Set

- In dynamic paging algorithms locality a function of time
  - Process alternates from very few to very many page frames over its lifetime
- Most contemporary models based on working set to handle dynamic locality requirements
- Intuitively, the working set is the set of pages in the process's locality
  - Somewhat imprecise and time varying
- Intuitively, the working set is the set of pages in the process's locality
  - Given k processes in memory, let mi(t) be the number of page frames allocated to p_i at time t
    - $m_i(0) = 0$
    - $\Sigma_{i=1,\ldots,k}\ m_i(t) <= |\text{primary memory}|$
    - Also have $S_t(m_i(t)) = S_t(m_i(t-1)) \cup X_t - Y_t$
    - Or, more simply $S(m_i(t)) = S(m_i(t-1)) \cup X_t - Y_t$
- How is primary memory at virtual time t for process p_i allocated to adjust for locality?

Placement/Replacement of Pages

- Recall that $S(m_i(t)) = S(m_i(t-1)) \cup X_t - Y_t$
  - X_t set of pages placed in memory at time t
  - Y_t set of pages removed from memory at time t
- For the missing page r_t allocate a new page frame
  - $X_t = \{r_t\}$ in the new page frame
- How should Y_t be defined?
- Consider a parameter, τ, called the window size that bounds the previous references in stream
  - Determine $BKWD_t(y)$ for every y in $S(m_i(t-1))$
  - If $BKWD_t(y) >= \tau$, unload y and deallocate frame
  - If $BKWD_t(y) < \tau$ do not disturb y
- τ is how far back to look to replace pages

The Working Set Window



At virtual time i-1: working set = {0, 1}
At virtual time i: working set = {0, 1, 3}

Working Set Principle

- Principle states
  - Process pi should only be loaded and active if it can be allocated enough page frames to hold its entire working set
  - Otherwise, process is pi blocked
- Size of the working set is estimated using τ
  - Unfortunately, a "good" value of τ depends on the size of the locality
  - Empirically this works with a fixed τ

Examples

$\tau = 3$

Frame	0	1	2	3	0	1	2	3	0	1	2	3	4	5	6	7
0	0	0	0	3	3	3	2	2	2	1	1	1	4	4	4	7
1		1	1	1	0	0	0	3	3	3	2	2	2	5	5	5
2			2	2	2	1	1	1	0	0	0	3	3	3	6	6
Alloc	1	2	3	3	3	3	3	3	3	3	3	3	3	3	3	3

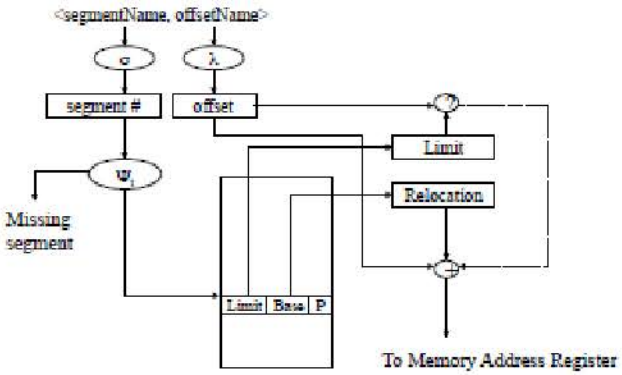
$\tau = 4$

Frame	0	1	2	3	0	1	2	3	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
1		1	1	1	1	1	1	1	1	1	1	1	1	5	5	5
2			2	2	2	2	2	2	2	2	2	2	2	2	6	6
3				3	3	3	3	3	3	3	3	3	3	3	3	7
Alloc	1	2	3	4	4	4	4	4	4	4	4	4	4	4	4	4

Segmentation

- Unit of memory movement is:
  - Variably sized
  - Defined by the programmer
- Two component addresses, <Seg#, offset>
- Address translation is more complex than paging
  - Yt: segments x offsets -> Physical Address U {Ω}
  - Yt(i, j) = k

Address Translation



Implementation

- Segmentation requires special hardware
  - Segment descriptor support
  - Segment base registers (segment, code, stack)
  - Translation hardware
- Some of translation can be static
  - No dynamic offset name binding
  - Limited protection

Example: Windows XP

- Uses demand paging with clustering. Clustering brings in pages surrounding the faulting page.
- Processes are assigned working set minimum and working set maximum
- Working set minimum is the minimum number of pages the process is guaranteed to have in memory
- A process may be assigned as many pages up to its working set maximum
- When the amount of free memory in the system falls below a threshold, automatic working set trimming is performed to restore the amount of free memory
- Working set trimming removes pages from processes that have pages in excess of their working set minimum

--- END MEMORY MANGEMENT UNIT

--- DEVICE MANGEMENT UNIT

LECTURE D1: DEVICE MANAGEMENT

Topics

- I/O Hardware
- Application I/O Interface
- Kernel I/O Subsystem
- Transforming I/O Requests to Hardware Operations
- Performance

Objectives

- Explore the structure of an operating system's I/O subsystem
- Provide details of the performance aspects of I/O hardware and software

Overview

- Management of I/O devices is a very important part of the operating system …
- … so important and so varied that entire I/O subsystems are devoted to its operation.
- I/O subsystems must contend with two (conflicting?) trends:
    - The gravitation towards standard interfaces for a wide range of devices, making it easier to add newly developed devices to existing systems, and
    - the development of entirely new types of devices, for which the existing standard interfaces are not always easy to apply.

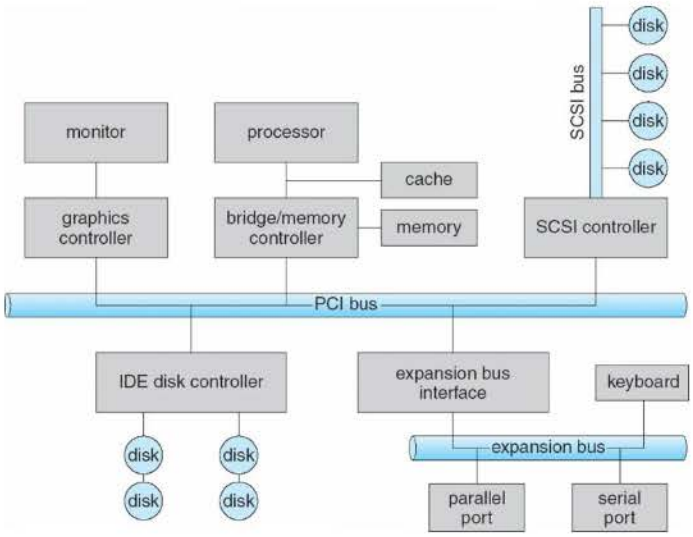
I/O Hardware

- Incredible variety of I/O devices: storage, transmission, human-interface
- Common concepts
    - signals from I/O devices interface with computer
    - port – connection point for device
    - bus – common set of wires connecting multiple devices
    - controller – electronics that operate port, bus, device

Buses

- Buses include rigid protocols for the types of messages that can be sent across the bus and the procedures for resolving contention issues.
- Four bus types commonly found in a modern PC:
    - The PCI bus is common in PCs and servers and connects high-speed high-bandwidth devices to the memory subsystem (and the CPU.) (Variation: PCI Express (PCIe))
    - The expansion bus connects slower low-bandwidth devices, which typically deliver data one character at a time (with buffering.)
    - The SCSI bus connects a number of SCSI devices to a common SCSI controller.
    - A daisy-chain bus is when a string of devices is connected to each other like beads on a chain, with only one of the devices directly connected to the host.

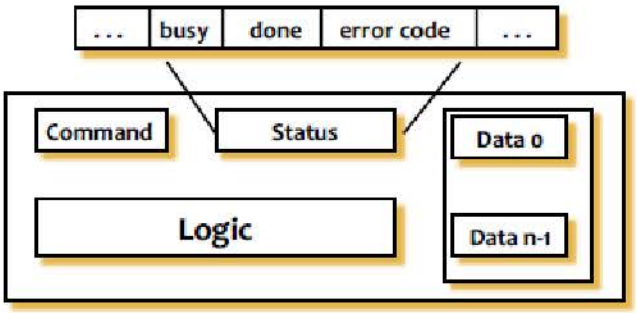
A Typical PC Bus Structure



Device Controllers

- Controller – electronics that operate port, bus, device
  - Sometimes integrated, sometimes separate circuit board (host adapter)
  - Contains processor, microcode, private memory, bus controller, etc.
    - Some talk to per-device controller with bus controller, microcode, memory, etc.
- I/O instructions control devices which have addresses
- Two ways for a processor to communicate with a controller to accomplish an I/O transfer
  - Direct I/O instructions
    - Must specify both the device and the command
  - Memory-mapped I/O
    - Device data and command registers mapped to processor address space
    - Especially for large address spaces (graphics)

Device Controller Interface

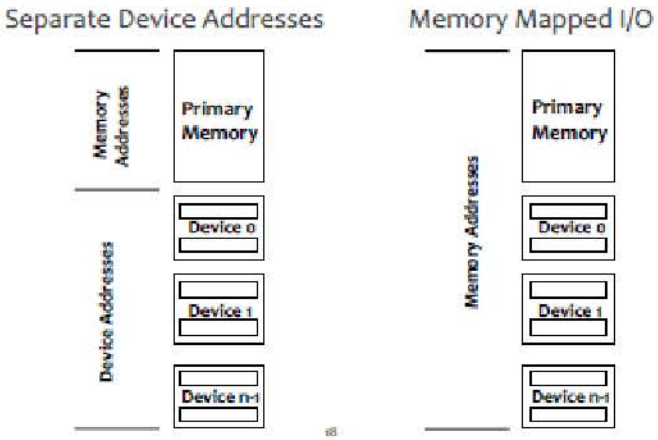


busy	done	
0	0	idle
0	1	finished
1	0	working
1	1	(undefined)

Direct I/O

- Devices must have memory via which information/status can be exchanged for I/Os
- Devices usually have registers where device driver places commands, addresses, and data to write, or read data from registers after command execution
  - Data-in register, data-out register, status register, control register
  - Typically 1-4 bytes, or FIFO buffer
- Direct approach employs dedicated memory for device addresses
  - Addressing and access more complex
  - Requires dedicated instruction set
  - Example: **copy_in R3, 0x012, 4**

Direct vs. Memory-Mapped I/O



Memory-Mapped I/O

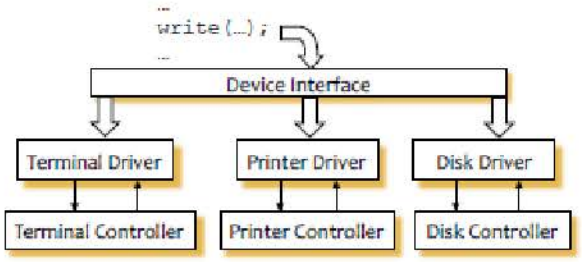
- Memory-mapped approach utilizes dedicated portion of primary memory
  - Portions of the high-order memory address space are assigned to each I/O device
  - Read and writes to those memory addresses are interpreted as commands to the I/O devices
  - Load/stores to the I/O address space can only be done by the O.S.
  - Addressing and access simplified
  - Exploits existing instruction set so reduces overall instruction set
  - Example: load R3, 0xFFFF0124
- Memory-mapped I/O is suitable for devices which must move large quantities of data quickly, such as graphics cards.
- Memory-mapped I/O can be used either instead of or more often in combination with traditional registers.
  - For example, graphics cards still use registers for control information such as setting the video mode.

Common Device I/O Port Locations on PCs (partial)

I/O address range (hex)	Device
000 – 00F	DMA controller
020 – 021	Interrupt controller
040 – 043	Timer
200 – 20F	Game controller
2F8 – 2FF	Serial port (secondary)
320 – 32F	Hard disk controller
378 – 37F	Parallel port
3D0 – 3DF	Graphics controller
3F0 – 3F7	Diskette drive controller
3F8 – 3FF	Serial port (primary)

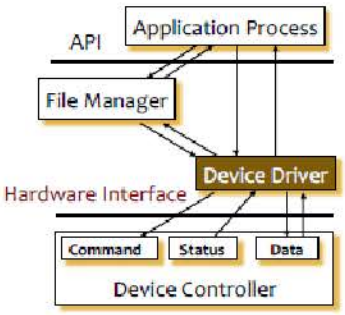
I/O Software: Device Drivers

- Software interface between OS and device controller interface



Device Management Organization

- Device driver API same across all similar devices
  - Files stored on disks, tapes, DVDs, etc.
- Device driver contains device specific implementation
  - Code of APIs is character vs. block oriented

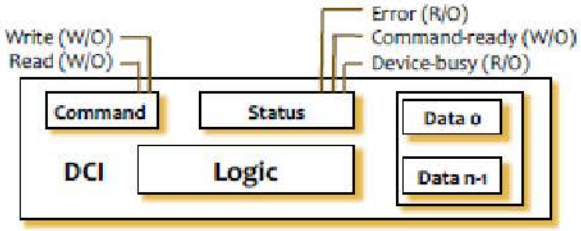


Communication of I/O Devices and Processor

- Two ways for the processor to physically communicate with the controller; now what strategy is used to do so?
- Two ways that I/O devices communicate with the processor
  - Direct I/O with Polling (Programmed I/O)
  - Interrupt-driven I/O
- Alternately can also use direct memory access (DMA) independent of the CPU

Polling: For each byte of I/O

1. Host repeatedly checks device-busy bit from DCI status register until 0
2. Host sets read or write bit and if write copies data into data-out register
3. Host sets command-ready bit to notify device of pending command
4. Controller sets its device-busy bit, executes transfer
5. Controller clears its device-busy bit, error bit, and command-ready bit when transfer done



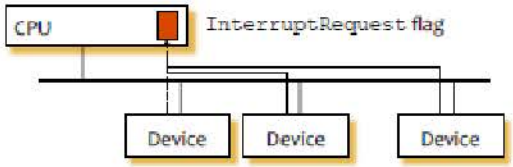
- Step 1 is busy-wait cycle to wait for I/O from device
  - Reasonable if device is fast but inefficient if device slow
  - CPU switches to other tasks?
    - But if miss a cycle data overwritten / lost

Interrupts

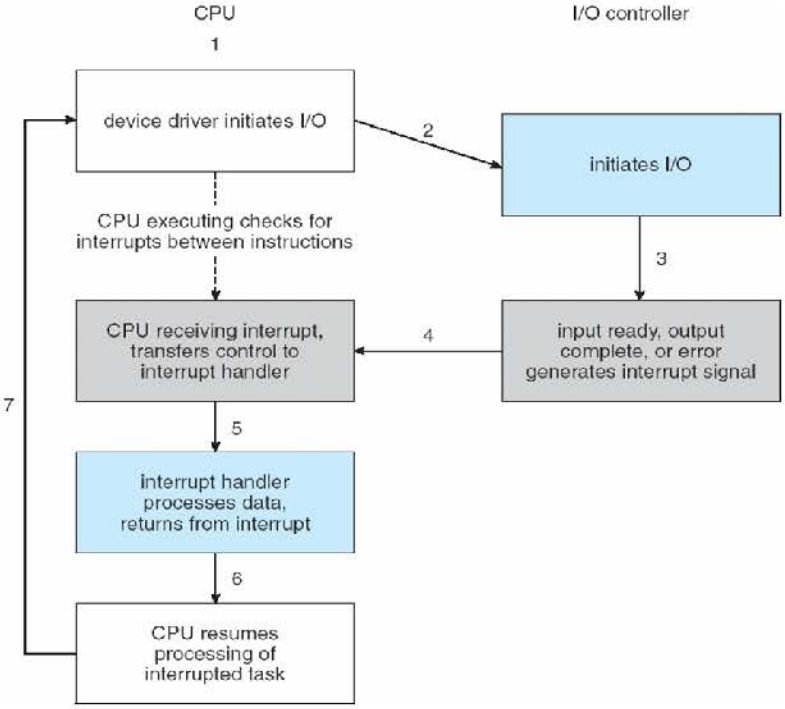
- Polling can happen in 3 instruction cycles
  - Read status, logical-and to extract status bit, branch if not zero
  - How to be more efficient if non-zero infrequently?
- Interrupts allow devices to notify the CPU when they have data to transfer or when an operation is complete, allowing the CPU to perform other duties when no I/O transfers need its immediate attention.
- CPU interrupt-request line triggered by I/O device
  - Checked by processor after each instruction

Control Unit with Interrupt

- CPU incorporates InterruptRequest flag
  - When device is not busy set this flag
  - Hardware "tells" OS that the interrupt has occurred
  - Interrupt handler part of the OS that receives interrupts and makes process ready to run



Interrupt-Driven I/O Cycle



Interrupts (cont.)

- Interrupt vector to dispatch interrupt to correct handler
  - Context switch at start and end
  - Based on priority
  - Some maskable to ignore or delay some interrupts, some nonmaskable
  - Interrupt chaining if more than one device at same interrupt number
- Interrupt mechanism also used for *exceptions*
  - Terminate process, crash system due to hardware error
- Page fault executes when memory access error
- System call executes via *trap* (i.e. software interrupt) to trigger kernel to execute request
- Multi-CPU systems can process interrupts concurrently if O.S. designed to handle it
- Used for time-sensitive processing, frequent, must be fast

Intel Pentium Processor Event-Vector Table

Vector #	Description
0	Divide error
1	Debug exception
2	Null interrupt
3	Breakpoint
4	INTO-detected overflow
5	Bound range exception
6	Invalid opcode
7	Device not available
8	Double fault
9	Coproc. segment overrun
10	Invalid task state segment

Vector #	Description
11	Segment not present
12	Stack fault
13	General protection
14	Page fault
15	(Intel reserved)
16	Floating-point error
17	Alignment check
18	Machine check
19 – 31	(Intel reserved)
32 – 255	Maskable interrupts

Summary: I/O Methods Involving CPU

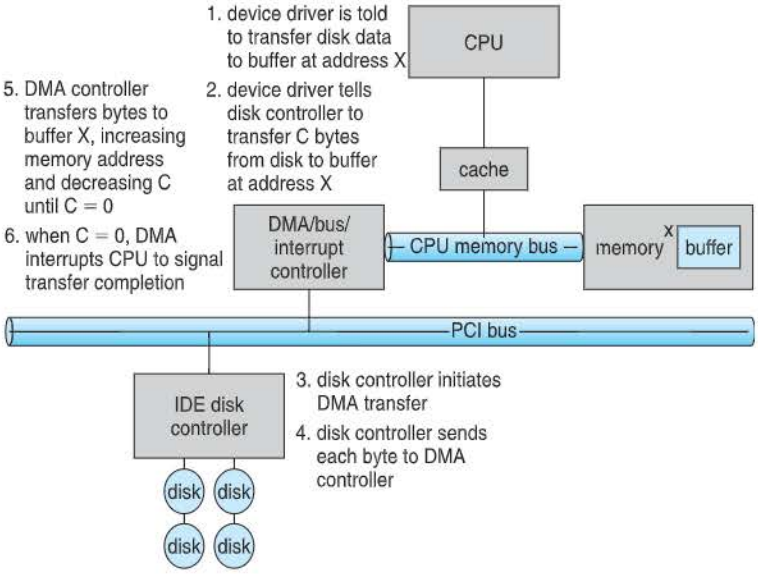
- Direct I/O with Polling (Programmed I/O)
  - I/O module performs the action, not the processor
  - Sets appropriate bits in the I/O status register
  - No interrupts occur
  - Processor checks status until operation is complete.

- Interrupt-Driven I/O
  - Processor is interrupted when I/O module is ready to exchange data
  - Processor is free to do other work; no needless waiting

Direct Memory Access

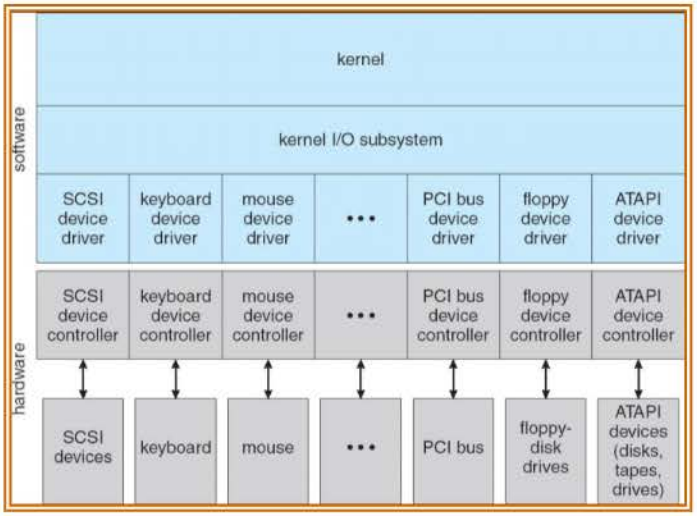
- For high bandwidth devices (like disks) used to avoid programmed I/O (one byte at a time) for large data movement
- Requires DMA controller
  - Separate processor with its own local memory; a computer in its own right
- OS writes DMA command block into memory
  - Source and destination addresses
  - Read or write mode
  - Count of bytes
  - Writes location of command block to DMA controller
- DMA controller takes control from the CPU to transfer data to and from memory over the system bus.
  - Cycle stealing used to transfer data
    - The instruction cycle is suspended so data can be transferred
    - CPU pauses one bus cycle so executes more slowly
    - No interrupts occur so context not saved
- Bypasses CPU to transfer data directly between I/O device and memory
- Only one interrupt is generated per block, rather than one interrupt per byte.
  - Interrupt is sent when the task is complete.
  - The processor is only involved at the beginning and end of the transfer.
- Number of required busy cycles can be cut by integrating the DMA and I/O functions
  - Means a path between DMA and I/O modules that does not include the system bus
- Version that is aware of virtual addresses can be even more efficient - DVMA

Six Step Process to Perform DMA Transfer



Application I/O Interface

- User application access to a wide variety of different devices is accomplished through layering
- I/O system calls encapsulate device behaviors in generic classes and isolate all of the device-specific code into device drivers
    - Device-driver layer hides differences among I/O controllers from kernel
- Application layers are presented with a common interface for all (or at least large general categories of) devices.



I/O Devices

- Devices vary in many dimensions
- Most devices can be characterized as
    - block I/O or character I/O,
    - memory mapped file access, or
    - network sockets.
- A few devices are special, such as time-of-day clock and the system timer.

Block and Character Devices

- Block devices include disk drives
    - Commands include read, write, seek
    - Raw I/O (accessing blocks directly) or file-system access
    - Memory-mapped file access possible
- Character devices include keyboards, mice, serial ports
    - Commands include get, put
    - Libraries layered on top allow line editing

Characteristics of I/O Devices

aspect	variation	example
data-transfer mode	character block	terminal disk
access method	sequential random	modem CD-ROM
transfer schedule	synchronous asynchronous	tape keyboard
sharing	dedicated sharable	tape keyboard
device speed	latency seek time transfer rate delay between operations	
I/O direction	read only write only read–write	CD-ROM graphics controller disk

Network Devices

- Varying enough from block and character to have own interface
- Unix and Windows NT/9x/2000 include socket interface
    - Separates network protocol from network operation
    - Includes select functionality
- Approaches vary widely (pipes, FIFOs, streams, queues, mailboxes)

Clocks and Timers

- Provide current time, elapsed time, timer
- Programmable interval timer used for timings, periodic interrupts
- Most O.S.s also have an escape, or back door, which allows applications to send commands directly to device drivers if needed.
    - ioctl (on UNIX) covers odd aspects of I/O such as clocks and timers

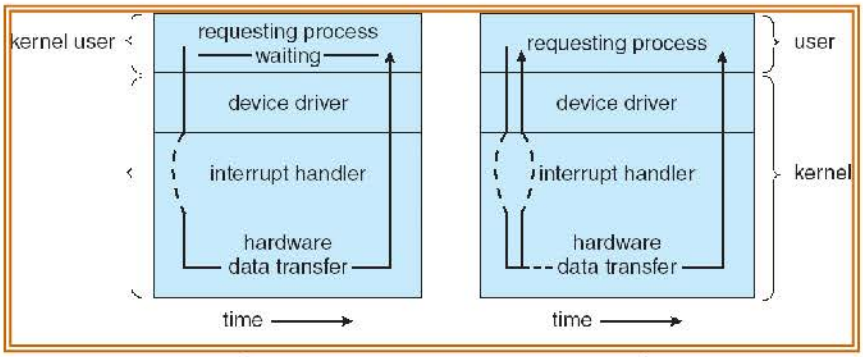
Blocking and Nonblocking I/O

- Blocking - process suspended until I/O completed
    - Easy to use and understand
    - Insufficient for some needs

- Nonblocking - I/O call returns immediately with as much data as available
  - User interface, data copy (buffered I/O)
  - Implemented via multi-threading
  - Returns quickly with count of bytes read or written
    - Does not complete I/O operation
  - Subtle variation: asynchronous I/O

Asynchronous I/O

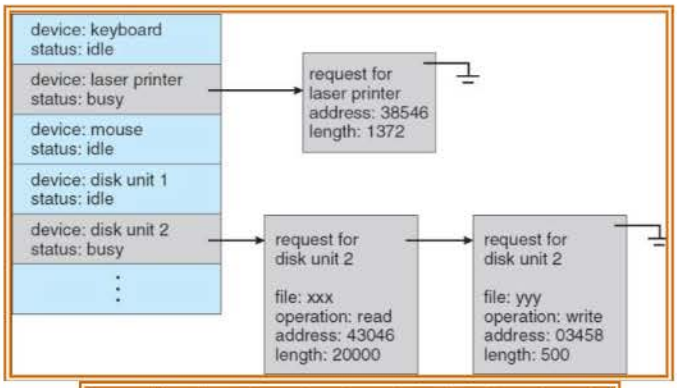
- Process runs while I/O executes
  - Difficult to use
  - I/O subsystem signals process when I/O completed



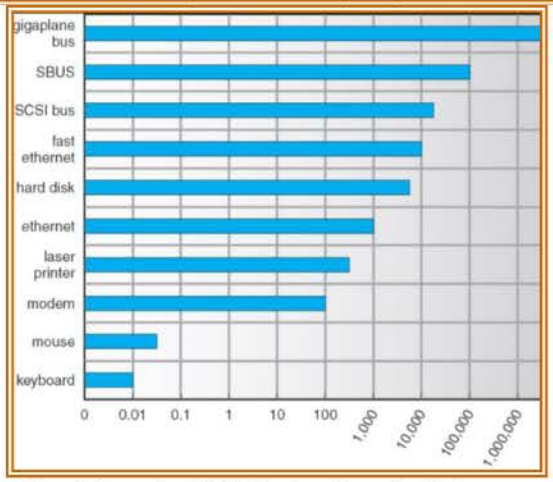
Synchronous (Non-blocking) Asynchronous

Kernel I/O Subsystem

- Scheduling
  - Some I/O request ordering via per-device queue
  - Some OSs try fairness



- Buffering - store data in memory while transferring between devices
  - To cope with device speed mismatch
  - To cope with device transfer size mismatch
  - To maintain "copy semantics"
- Caching - fast memory holding copy of data
  - Always just a copy; key to performance
- Spooling - hold output for a device
  - ...if device can serve only one request at a time (i.e., printing)
- Device reservation - provides exclusive access to a device
  - System calls for allocation and de-allocation
  - Watch out for deadlock



Sun Enterprise 6000 Device-Transfer Rates

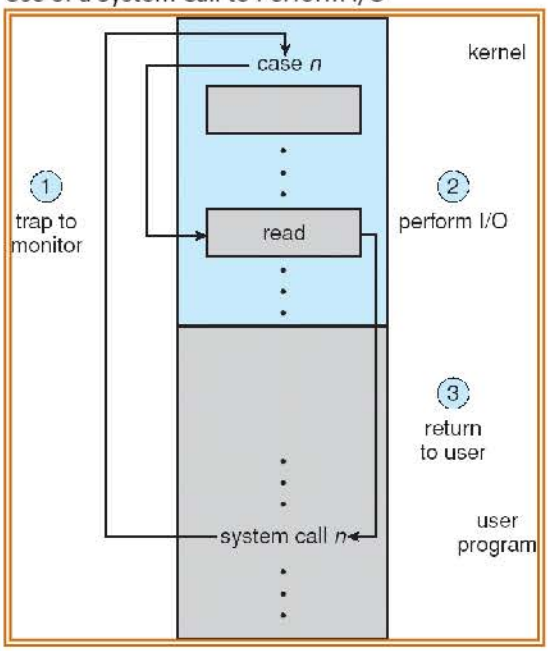
Error Handling

- OS can recover from disk read, device unavailable, transient write failures
- Most return an error number or code when I/O request fails
- System error logs hold problem reports

I/O Protection

- User process may accidentally or deliberately attempt to disrupt normal operation via illegal I/O instructions
  - All I/O instructions defined to be privileged
  - I/O must be performed via system calls
    - Memory-mapped and I/O port memory locations must be protected too
    - ... but access to these areas cannot be totally denied to user programs
      - Instead memory protection system restricts access so that only one process at a time can access particular parts of memory, such as the portion of the screen memory corresponding to a particular window.

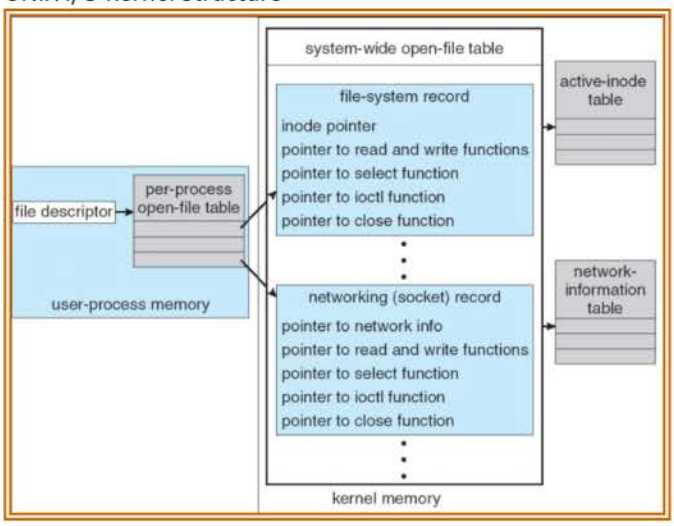
Use of a System Call to Perform I/O



Kernel Data Structures

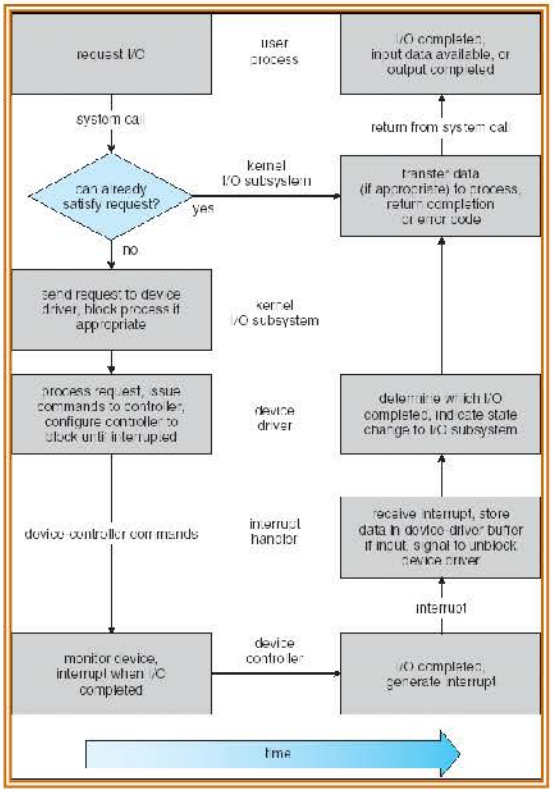
- Kernel keeps state info for I/O components, including open file tables, network connections, character device state
- Many, many complex data structures to track buffers, memory allocation, "dirty" blocks
- Some use object-oriented methods and message passing to implement I/O

UNIX I/O Kernel Structure



I/O Requests to Hardware Operations

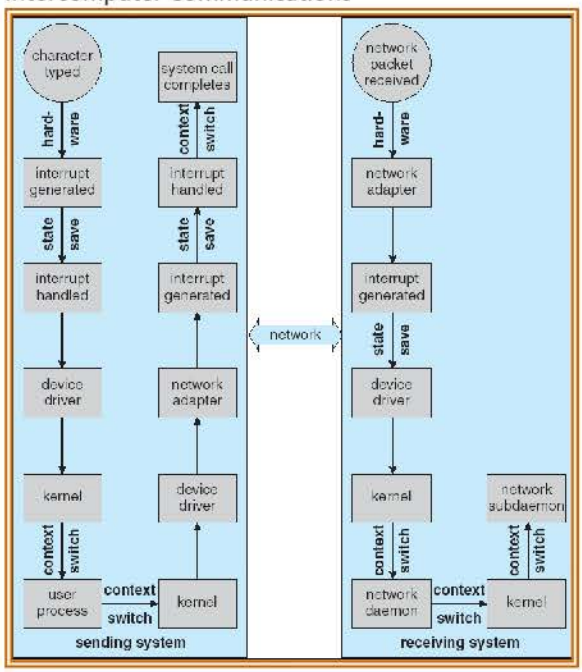
- Users request data using file names, which must ultimately be mapped to specific blocks of data from a specific device managed by a specific device driver.
- A series of lookup tables and mappings makes the access of different devices flexible, and somewhat transparent to users.
- Consider reading a file from disk for a process:
    - Determine device holding file
    - Translate name to device representation
    - Physically read data from disk into buffer
    - Make data available to requesting process
    - Return control to process



Performance

- I/O a major factor in system performance:
    - Demands on CPU to execute device driver, kernel I/O code
    - Context switches due to interrupts
        - … which causes programmed I/O to be faster than interrupt-driven I/O when the time spent busy waiting is not excessive
    - Data copying
    - Network traffic especially stressful as shown next

Intercomputer Communications

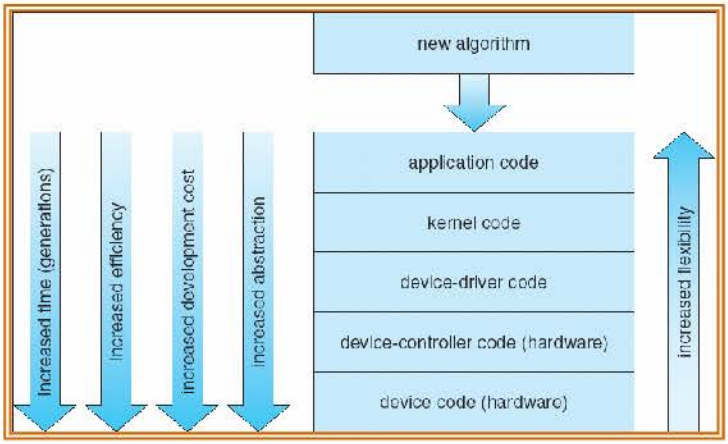


Improving Performance

- Some systems use front-end processors to off-load some of the work of I/O processing from the CPU.
    - For example a terminal concentrator can multiplex with hundreds of terminals on a single port on a large computer.
- Several principles can be employed to increase the overall efficiency of I/O processing:
    - Reduce data copying
    - Reduce interrupts by using large transfers, smart controllers, polling
    - Use DMA
    - Balance CPU, memory, bus, and I/O performance for highest throughput

Device-Functionality Progression

- The development of new I/O algorithms often follows a progression from application level code to on-board hardware implementation
- The development of new I/O algorithms often follows a progression from application level code to on-board hardware implementation
    - Lower-level implementations faster and more efficient, but higher-level ones more flexible and easier to modify.
    - Hardware-level functionality may also be harder for higher-level authorities (e.g. the kernel) to control.



Summary

- I/O management is a major component of operating system design and operation
    - Important aspect of computer operation
    - I/O devices vary greatly with new types of devices frequent
    - Various methods to control them
    - Performance management
- Ports, busses, device controllers connect to various devices
- Device drivers encapsulate device details
    - Present uniform device-access interface to I/O subsystem

LECTURE D2: MASS STORAGE STRUCTURE

Topics

- Overview of Mass Storage Structure
- Disk Structure
- Disk Scheduling
- Summary

Objectives

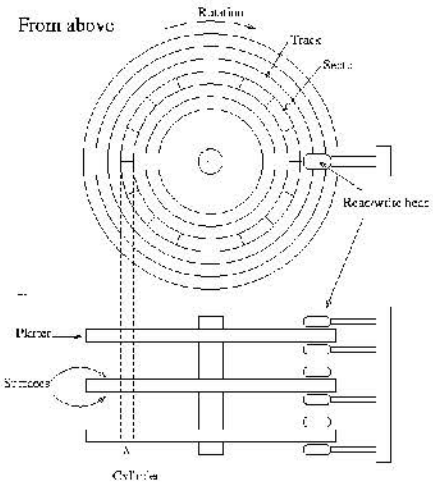
- Describe the physical structure of secondary storage devices and the resulting effects on the uses of the devices
- Explain the performance characteristics of mass-storage devices

Overview of Mass Storage Structure

- Main memory
    - Only large storage media that the CPU can access directly.
- Secondary storage
    - Extension of main memory that provides large nonvolatile storage capacity.
- Magnetic/optical disks
    - Provide bulk of secondary storage of modern computers
    - Can be removable
- Magnetic tape: early secondary-storage medium
    - Relatively permanent and holds large quantities of data
    - Access time slow
    - Random access ~1000 times slower than disk
    - Mainly used for backup, storage of infrequently-used data, transfer medium between systems
    - Kept in spool and wound or rewound past read-write head
    - Once data under head, transfer rates comparable to disk
    - 20-200GB typical storage
    - Common technologies are 4mm, 8mm, 19mm, LTO-2 and SDLT

- Magnetic/optical disks
  - Popular media (hard drives, CDs, DVDs, etc.)
  - Randomly accessed storage devices: access to information in any order
    - Sequential access not typically supported or needed, since "files" not stored sequentially
    - Recall, disk defragmentation on PC platform
  - Drive attached to computer via I/O bus
    - Busses vary, including EIDE, ATA, SATA, USB, Fibre Channel, SCSI, SAS, Firewire
    - The disk controller determines the logical interaction between the device and the computer.
      - Host controller in computer uses bus to talk to disk controller built into drive or storage array
  - Physically rigid metal or glass platters covered with magnetic recording material
    - Disk is logically divided into surfaces, which are subdivided into tracks.
    - Other key concepts: sector, cylinder, read/write heads
  - Disks typically DMA devices
    - To read or write, the disk head must be positioned at the desired track and at the beginning of the desired sector
    - Head crash results from disk head making contact with the disk surface (that's bad)

Recall: Rotating Storage Structure



Note: Parallel read/write drives activate all heads simultaneously

Latest: Solid-State Disks

- Nonvolatile memory used like a hard drive; many technology variations
- Advantages
  - Can be more reliable than HDDs
  - Much faster
    - No moving parts, so no seek time or rotational latency
- Disadvantages
  - Less capacity and more expensive per MB
  - May have shorter life span
  - Busses can be too slow (connect directly to PCI for example)

Disk Structure

- Disk drives are addressed as large 1-dimensional arrays of logical blocks, where the logical block is the smallest unit of transfer.
- The 1-dimensional array of logical blocks is mapped into the sectors of the disk sequentially.
  - Sector 0 is the first sector of the first track on the outermost cylinder.
  - Mapping proceeds in order through that track, then the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost.

Disk Characteristics and Access

- Transfer time: time to copy bits from disk surface to primary memory
  - Transfer occurs as sector moves under head
- Positioning time: time to move disk arm to desired cylinder (disk seek time), plus time for desired sector to rotate under the disk head (disk latency time)
  - Drives rotate at 60 to 200 times per second
- Disk (rotational) latency time:
  - Rotational delay waiting for proper sector to rotate under R/W head
    - i.e. time it takes for the beginning of the sector to reach the head
  - Rotate to next sector to process next request

- Disk seek time:
  - Delay while R/W head moves to the destination track/cylinder
    - i.e. time it takes to position the head at the desired track
  - Move head in/out to seek next track/cylinder
  - The reason for differences in performance
- Access time = seek (in/out) + latency (around) + transfer (bytes)
  - Time it takes to read or write
- The operating system is responsible for using hardware efficiently; for the disk drives, this means having a fast access time and high disk bandwidth.
  - Access time has two major components: seek time and rotational latency
  - Disk bandwidth is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer.

Overview: Disk Scheduling Policies

- Multiprogramming on I/O-bound programs
  - Set of processes waiting for disk
  - Each process has one or more I/O requests
- So for a single disk there will be a number of I/O requests
  - If requests are selected randomly we will get the worst possible performance
- Seek time dominates access time
  - What order should requests be processed?
  - Goal: minimize seek time (or seek distance) across the set

Disk Scheduling Policies

First come first served
- Process requests sequentially
- Fair to all processes
- Approaches random scheduling in performance if there are many processes

Shortest seek time first
- Select the disk I/O request that requires the least movement of the disk arm from its current position
- Always chose the minimum seek time

Scan
- Arm moves in one direction only, satisfying all outstanding requests until it reaches the last track in that direction
- Direction is reversed

Circular scan (C-SCAN)
- Restricts scanning to one direction only
- When the last track has been visited in one direction, the arm is returned to the opposite end of the disk and the scan begins again

Optimizing Seek Time: Strategies for Disk Access
- Assumptions (simplify to tracks only):
  - Tracks 0:99
  - One head at track 75
  - Requests for 23, 87, 36, 93, 66
- First come first served (FCFS)
  - 52+ 64 + 51 + 57 + 27 = 251 steps
- Shortest seek time first (SSTF)
  - Reorder to (75), 66, 87, 93, 36, 23
  - 9 + 21 + 6 + 57 + 13 = 106 steps
- Scan algorithm (start at one end and move in one direction, reordering as needed):
  - (75), 87, 93, 99, 66, 36, 23
  - 12 + 6 + 6 + 33 + 30 + 13 = 100 steps
- Circular scan (goes from current position to 99 and then resets at 0 - adds reset time):
  - (75), 87, 93, 99, 0, 23, 36, 66
  - 12 + 6 + 6 + home + 23 + 13 + 30 = 90 + home
- Look algorithm (scan but stop at highest track of current request):
  - (75), 87, 93, 66, 36, 23
  - 12 + 6 + 27 + 30 + 13 = 88 steps
- Circular look (goes from current position to highest and then resets at 0 - adds reset time):
  - (75), 87, 93, 0, 23, 36, 66
  - 12 + 6 + home + 23 + 13 + 30 = 84 + home

Other Policies

- Priority
  - Goal is not to optimize disk use but to meet other objectives
    - Ex: short batch jobs may have higher priority
    - Ex: provide good interactive response time
- Last-in-first-out
  - Good for transaction processing systems
    - The device is given the most recent user so there should be little arm movement
  - Possibility of starvation since a job may never regain the head of the line
- N-step-scan
  - Segments the disk request queue into subqueues of length N
  - Subqueues are processed one at a time, using Scan
  - New requests added to other queue when queue is processed
- Fscan
  - Two queues
  - One queue is empty for new requests

Selecting a Disk Scheduling Policy

- SSTF is common and has a natural appeal
- SCAN and C-SCAN perform better for systems that place a heavy load on the disk.
- Performance depends on the number and types of requests.
- Requests for disk service can be influenced by the file-allocation method.
- The disk-scheduling algorithm should be written as a separate module of the operating system, allowing it to be replaced with a different algorithm if necessary.
- Either SSTF or LOOK is a reasonable choice for the default algorithm.

--- END DEVICE MANGEMENT UNIT

--- FILE MANGEMENT UNIT

LECTURE F1: FILE-SYSTEM INTERFACE

Topics
- File concept
- Access methods
- Directory structure

Objectives
- To explain the function of file systems
- To describe the interfaces to file systems
- To discuss file-system design tradeoffs

Introduction

- User perspective: a disk is a collection of files and directories that can be manipulated using commands.
- System perspective: a disk is a collection of data blocks that can be manipulated via a cylinder/head/sector/track.
- It's the job of the o.s. to bridge the gap between these two perspectives.

File Concept to the User

- Two basic types of files:
  - Unstructured - sequence of words, bytes
  - Structured
    - Simple record structure: fixed or variable length lines
    - Complex structures: formatted document, relocatable load file
- Can simulate last two with first method by inserting appropriate control characters
- Who decides: operating system or program

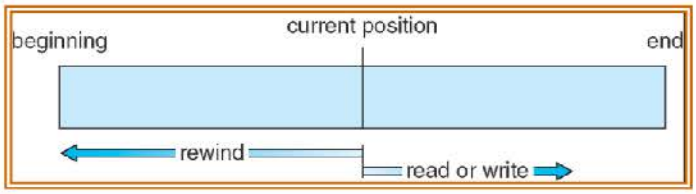
File Access Methods

Sequential access
- read next, write next
- reset
- no read after last write (rewrite)

Direct access
- read n, write n // n = relative block number
- position to n
- read next, write next
- rewrite n

Sequential-Access File

Overview



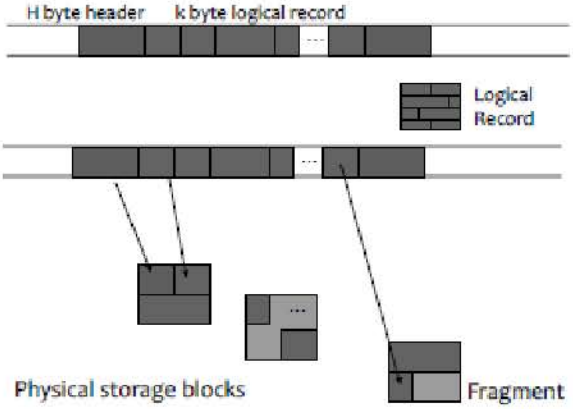
Simulation of sequential access on a direct-access file

Sequential access	Implementation for direct access
Reset	cp = 0;
Read next	read cp; cp++;
Write next	write cp; cp++;

Record-Oriented Sequential Files

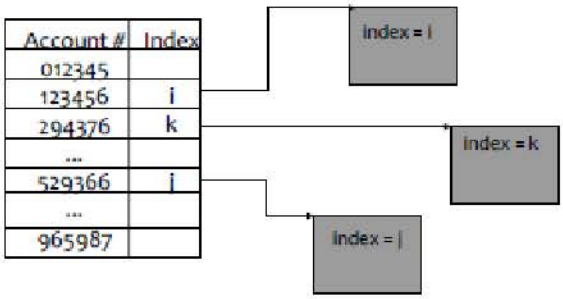
- Manage and store set of records in list
- For example, mail messages are records with
  - Header, sender, subject, receiver, body
  - Manipulated by editor, mailer (sender and receiver), browser, etc.
- Structured sequential file is a named sequence of logical records indexed by non-negative numbers
- Operations include:
  - fileID = open(fileName) and close(fileID)
  - getRecord/putrecord(fileID, record)
  - seek(fileID, position)

- Every logical record must be mapped to a physical byte stream
  - H-byte header for record descriptor
  - k bytes for record fields (sum of individual fields)
  - Padding used to even field boundaries to 2, 4, or 8 byte increments
- Block translation to/from record to/from stream
- Stream must then be mapped to physical storage
- Mapping to physical storage requires individual and sets of fields to be organized into secondary storage blocks
- Possibility of fragmentation

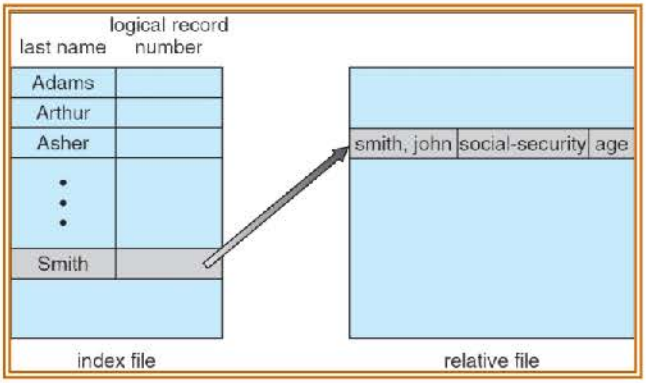


Indexed Sequential Files

- Suppose we want to directly access records
- For example, in ATM, access individual accounts without searching through all accounts
- Indexed sequential file adds an index to the file
- Data structure allows translation from index to specific record
- Application must maintain table to map key value (bank account numbers) to record indices
- Operations include:
  - fileID = open(fileName)and close(fileID)
  - getRecord(fileID, index)
  - index = putRecord(fileID, record)
  - deleteRecord(fileID, index)



Example of Index and Relative Files



File Concept to the O.S.

- The operating system must provide the data structures and algorithms necessary to implement each of the system calls.
- Among other things needed
  - File descriptors (in memory for open files, on disk for all files)
  - File locks (to protect critical sections and prevent race conditions related to file access)
  - Directory structures
  - Protection / permissions

File Descriptors

- File descriptors: maintaining detailed information on each file
- External name
  - Character string for the file name

    - Symbolic name (I.D.) associated with file
- Current State
    - Archived (stored on tape or tertiary media)
    - Closed, open for read, write, execute, etc.
- Sharable: read, write, execution sharable
- Owner: w.r.t. process and/or user – Protection
- User: process(es) accessing the open file
- Locks
    - Read lock: exclusivity of read access
    - Write lock: exclusivity of write access
- Length: number of bytes
- Time of creation, last modification, last access
    - When file created, last written to, last accessed
- Reference count
    - Number of directories referencing file
    - When count is zero, remove file; else, remove reference to file
- Storage device details (type)
    - Access strategy for blocks that comprise file
- Location - pointer to file location on device

Open File Locking

- Provided by some operating systems and file systems
- Mediates access to a file
- Mandatory or advisory:
    - Mandatory – access is denied depending on locks held and requested
    - Advisory – processes can find status of locks and decide what to do

Implementing Low Level Files

- Secondary storage device contains:
    - Volume directory (sometimes a root directory for a file system)
    - External file descriptor for each file
    - Contents of files
- Manages blocks
    - Assigns blocks to files (descriptor keeps track)
    - Keeps track of available blocks
- Maps file to/from byte stream

Directories

- Information about files are kept in the directory structure, which is maintained on the disk
- Directory: a collection of nodes containing information about all files
- A set of logically associated files and subdirectories
    - Both the directory structure and the files reside on disk
    - Backups of these two structures are kept on tapes

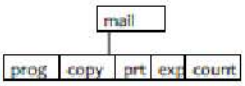
Directory Structure

- Organize the directory (logically) to obtain
    - Efficiency – locating a file quickly
    - Naming – convenient to users
        - Two users can have same name for different files
        - The same file can have several different names
    - Grouping – logical grouping of files by properties
        - e.g., all Java programs, all games, …

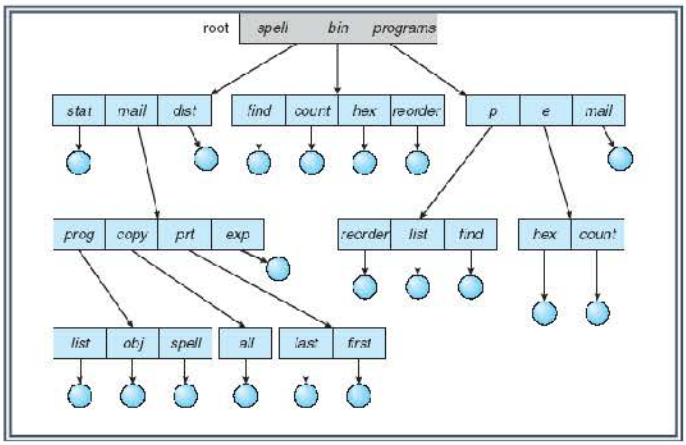
- How should files be organized within directory?
  - Flat name space:
    - All files appear in a single directory
  - Hierarchical name space
    - Directory contains files and subdirectories
    - Each file/directory appears as an entry in exactly one other directory – a tree
    - Popular variant: all directories form a tree, but a file can have multiple parents
    - Logical links to files/directories

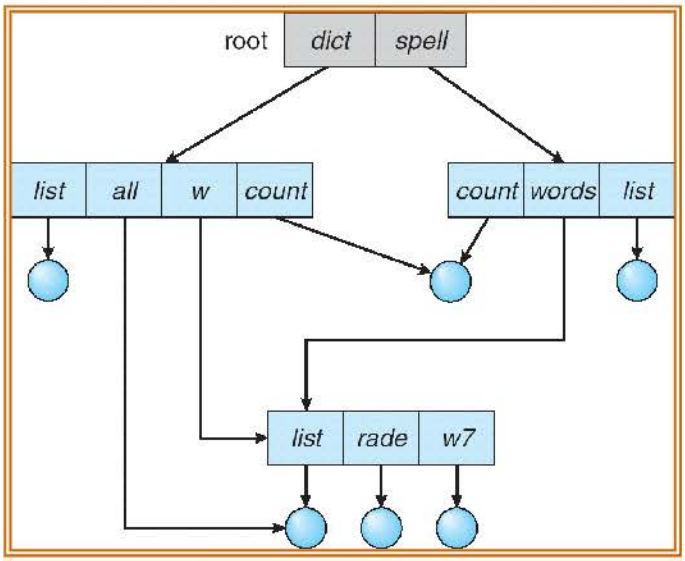
Hierarchical Name Space

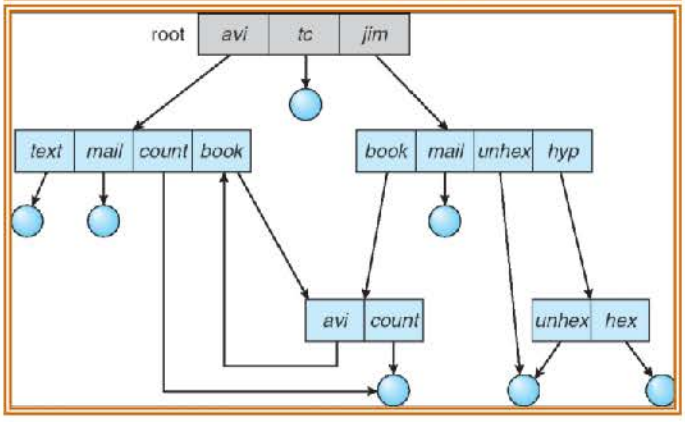
- Tree structured directories
  - Efficient searching
  - Grouping capability
  - Current directory (working directory)
  - Absolute or relative path name
    - cd /spell/mail/prog
    - type list
  - Creating a new file is done in current directory
  - Delete a file: rm <file-name>
  - Creating a new subdirectory is done in current directory: mkdir <dir-name>
    - Example: if in current directory /mail, mkdir count



    - deleting "mail" ⇒ deleting entire subtree rooted by "mail"
- Acyclic-graph directories: have shared subdirectories and files
  - Two different names (aliasing)
  - If dict deletes list -> dangling pointer; solutions:
    - Backpointers, so we can delete all pointers
      Variable size records a problem
    - Backpointers using a daisy chain organization
    - Entry-hold-count solution
  - New directory entry type
    - Link – another name (pointer) to an existing file
    - Resolve the link – follow pointer to locate the file
- General graph directory
  - How do we guarantee no cycles?
    - Allow only links to file not subdirectories
    - Garbage collection
    - Every time a new link is added use a cycle detection algorithm to determine whether it is OK







Protection

- File owner/creator should be able to control what can be done and by whom
- Types of access: Read, Write, Execute, Append, Delete, List

Access Lists and Groups

- Mode of access: read, write, execute
- Three classes of users on Unix / Linux

			RWX
▪ a) owner access	7	->	1 1 1
▪ b) group access	6	->	RWX 1 1 0
▪ c) public access	1	->	RWX 0 0 1

- Ask manager to create a group (unique name), say G, and add some users to the group.
- For a particular file (say game) or subdirectory, define an appropriate access.

- Attach a group to a file: chgrp G game

LECTURE F2: FILE-SYSTEM IMPLEMENTATION

Topics

- File-System Structure and Implementation
- Directory Implementation
- Allocation Methods
- Free-Space Management

Objectives

- To describe the details of implementing local file systems and directory structures
- To discuss block allocation and free-block algorithms and trade-offs.

File-System Structure

- File structure
  - Logical storage unit
  - Collection of related information
- File system resides on secondary storage (disks)

File-System Implementation Overview

1. Boot control block – information needed to boot
2. Volume control block – information about volume/partitions (# blocks, size of blocks, free block count, free block pointers)
3. Directory structure (inode)
4. Per file control blocks

Implementing Low Level Files

- Secondary storage device contains:
  - Volume directory (sometimes a root directory for a file system)
  - External file descriptor for each file
  - Contents of files
- Manages blocks
  - Assigns blocks to files (descriptor keeps track)
  - Keeps track of available blocks
- Maps file to/from byte stream

Directory Implementation

- Directories hold information about files
- Linear list of file names with pointer to the data blocks.
  - simple to program
  - time-consuming to execute
- Hash Table – linear list with hash data structure.
  - decreases directory search time
  - collisions – situations where two file names hash to the same location
  - fixed size

Block Management

- An allocation method refers to how fixed sized, k, storage blocks are assigned to the file
- File of length m requires N = ceil(m/k) Blocks
- Byte bi is stored in block floor(i/k)
- File manager has three basic strategies for allocating and managing blocks:
  - Contiguous allocation
  - Linked lists
  - Indexed allocation

Contiguous Allocation

- Maps the N logical blocks of file into N contiguous blocks on secondary storage device
- Simple – only starting location (block #) and length (number of blocks) are required

File descriptor

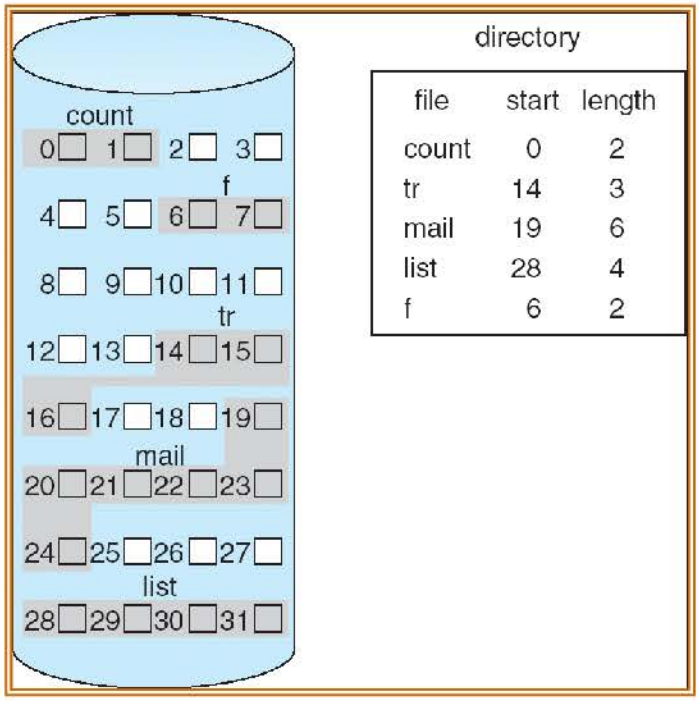
Head position	**237**
...	
First block	**785**
No. of blocks	**25**

- Supports random access
- Wasteful of space (dynamic storage-allocation problem)
- Difficult to support dynamic file sizes
- If file grows, rewrite file to secondary store
  - Utilizes best-fit, first-fit, and worst-fit
  - Fragments physical disk space
  - Resulting contiguous blocks too small to hold files
- Mapping from logical address to physical disk location

Q (quotient)
logical address/512 -------<
R (remainder)

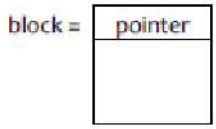
  - Block to be accessed = Q + starting block number
  - Displacement into block = R

Contiguous Allocation of Disk Space

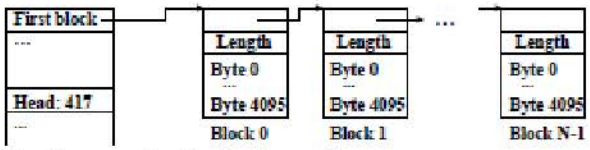


Linked Allocation

- Each file is a linked list of disk blocks which may be scattered anywhere on the disk.



- Each block contains a header with
  - Number of bytes in the block
    - allows storage of variable length blocks
- Pointer to next block



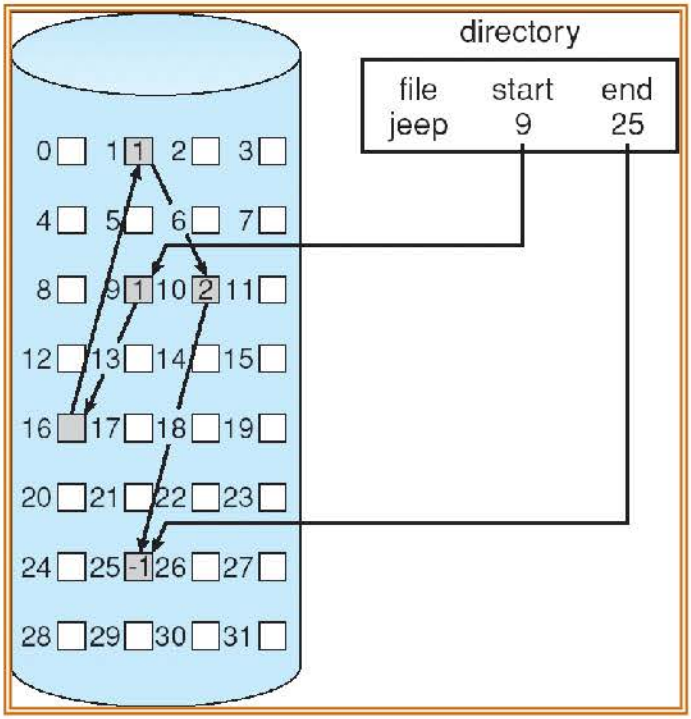
- Simple – need only starting address
- Free-space management system – no waste of space
- Files can expand and contract
- However, seeks can be slow
- No random access
- Mapping

Q

logical address /512-------<

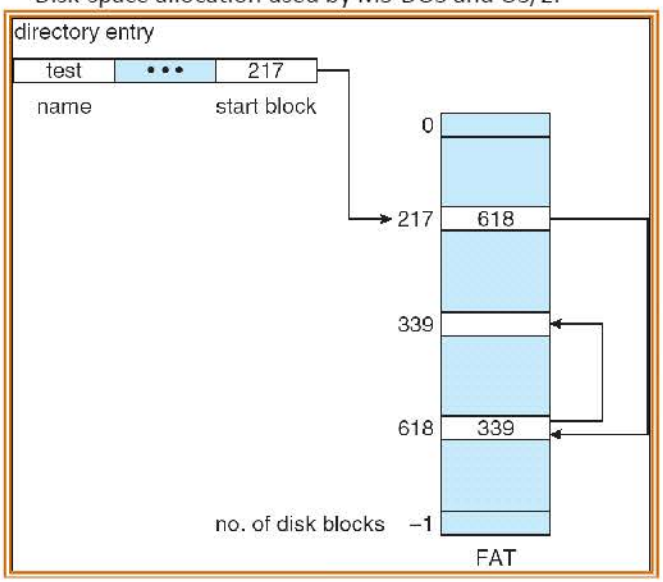
R

- Block to be accessed is the Qth block in the linked chain of blocks representing the file.
- Displacement into block = R + 1

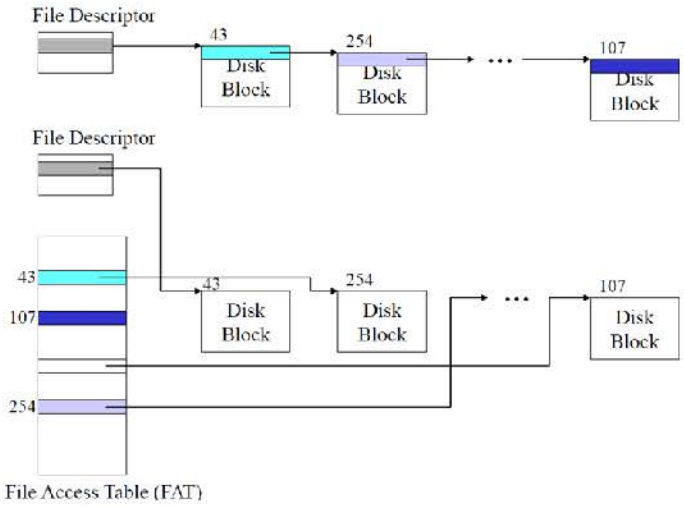


File Allocation Table (FAT)

- Disk-space allocation used by MS-DOS and OS/2.

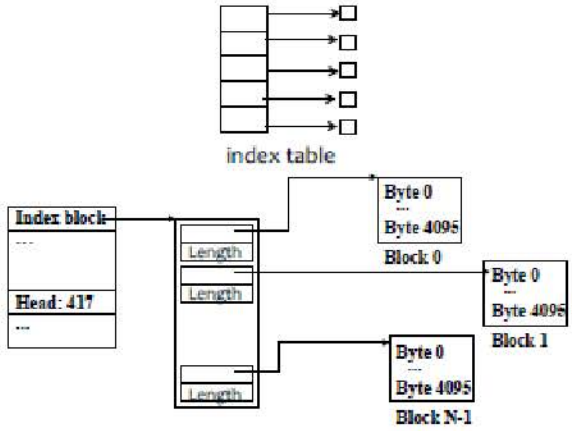


Linked List vs. DOS FAT Files



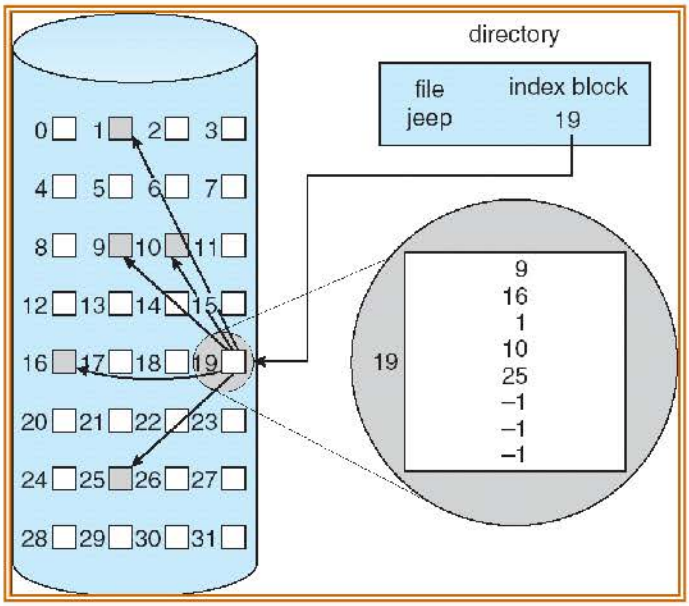
Indexed Allocation

- Brings all pointers together into the index block.
- Logical view.



- Simplify seeks
- May link indices together (for large files)
- Need index table
- Random access
- Dynamic access without external fragmentation, but have overhead of index block.
- Mapping from logical to physical in a file of maximum size of 256K words and block size of 512 words. We need only 1 block for index table.

Example of Indexed Allocation

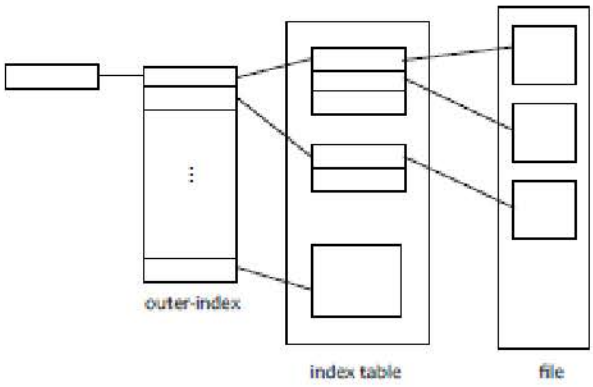


Indexed Allocation – Mapping

- Mapping from logical address to physical block in a file of unbounded length (block size of 512 words).

$$\text{logical address} / 512 \text{------} < \begin{matrix} Q \\ R \end{matrix}$$

  - Q = displacement into index table
  - R = displacement into block
- Two schemes
  - Linked scheme – Link blocks of index table (no limit on size).
  - Two-level index (maximum file size is 512^3)



Indexed Allocation – Linked Scheme

$$\text{logical address} / (512 \times 511) \text{------} < \begin{matrix} Q1 \\ R1 \end{matrix}$$

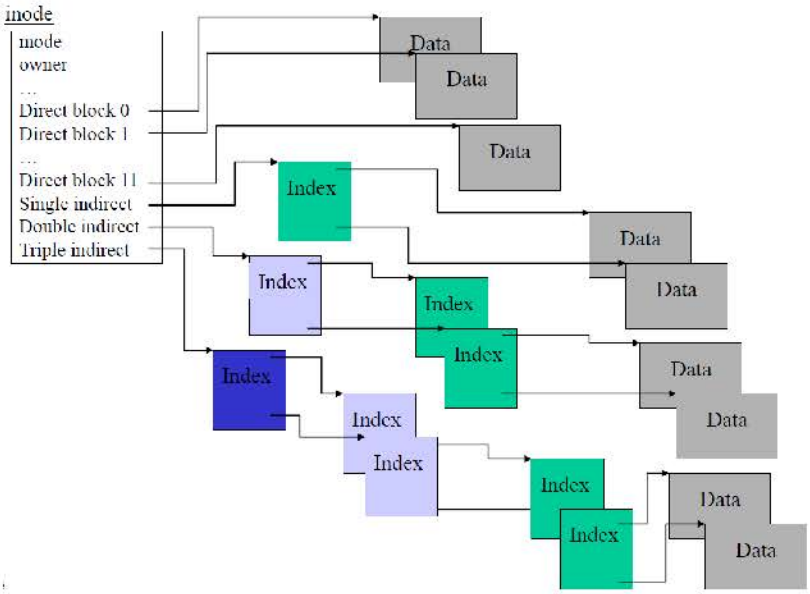
- Q1 = block of index table
- R1 used as follows: $R1 / 512$ --------< Q2 / R2
  - Q2 = displacement into block of index table
  - R2 displacement into block of file

Indexed Allocation – Linked Scheme

logical address /(512×512)--------< Q1 / R1

- Q1 = displacement into outer-index
- R1 used as follows: R1 / 512--< Q2 / R2
    - Q2 = displacement into block of index table
    - R2 displacement into block of file

Combined Scheme: UNIX Files (4K bytes per block)



Free-Space Management

- How should unallocated blocks be managed?
- Need a data structure to keep track of them
    - Linked list
        - Very large
        - Hard to manage spatial locality
    - Block status map ("disk map")
        - Bit per block
        - Easy to identify nearby free blocks
        - Useful for disk recovery

Option 1: Disk Map

- Bit vector (n blocks)

    - bit[i] = < 0 if block[i] free / 1 if block[i] occupied

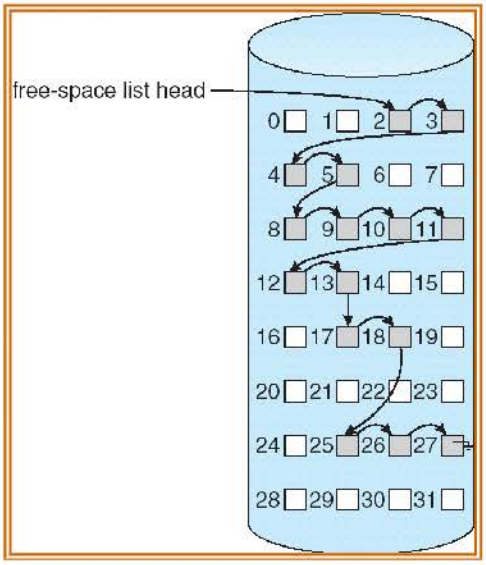
- Block number calculation: (number of bits per word) * (number of 0-value words) + (offset of first 0 bit)

- Bit map requires extra space
  - Example:
    - block size = 2^{12} bytes
    - disk size = 2^{30} bytes (1 gigabyte)
  - n = $2^{30}/2^{12} = 2^{18}$ bits (or 32K bytes)
- Easy to get contiguous files

Option 2: Linked List

- Linked list (free list)
  - Cannot get contiguous space easily
  - No waste of space
- Grouping
- Counting

Linked Free Space List on Disk



Free-Space Management

- Need to protect:
  - Pointer to free list
  - Bit map
    - Must be kept on disk
    - Copy in memory and disk may differ
    - Cannot allow for block[i] to have a situation where bit[i] = 1 in memory and bit[i] = 0 on disk
  - Solution:
    - Set bit[i] = 1 in disk
    - Allocate block[i]
    - Set bit[i] = 1 in memory

Efficiency and Performance

- Efficiency dependent on:
  - disk allocation and directory algorithms
  - types of data kept in file's directory entry
- Performance
  - disk cache – separate section of main memory for frequently used blocks
  - free-behind and read-ahead – techniques to optimize sequential access
  - improve PC performance by dedicating section of memory as virtual disk, or RAM disk

--- END FILE MANGEMENT UNIT

--- PROTECTION AND SECURITY

Outline

- Protection
  - Goals, Principles and Domain of Protection
  - Access Matrices and Their Implementation
  - Revocation of Access Rights
  - Capability-Based Systems
  - Language-Based Protection
- Security
  - The Security Problem
  - Program, System and Network Threats
  - Computer-Security Classifications
  - An Example: Windows XP

Goals

- Discuss the goals and principles of protection in a modern computer system
- Explain how protection domains combined with an access matrix are used to specify the resources a process may access
- Examine capability and language-based protection systems
- To discuss security threats and attacks

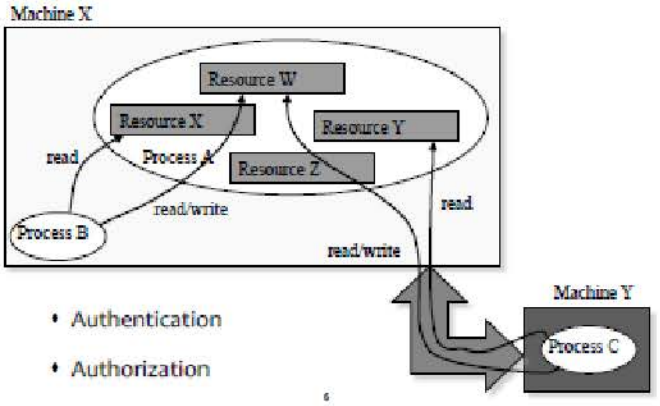
Protection Goals and Principles

- Operating system consists of a collection of objects, hardware or software
- Each object has a unique name and can be accessed through a well-defined set of operations.
- Protection problem – ensure that each object is accessed correctly and only by those processes that are allowed to do so.
- Guiding principle – principle of least privilege
  - Programs, users and systems should be given just enough privileges to perform their tasks

Policy and Mechanism

- Protection mechanisms are tools used to implement security policies
  - Authentication, authorization, cryptography
- A security policy reflects an organization's strategy for authorizing access to the computer's resources only to authenticated parties
  - Accountants have access to payroll files
  - O.S. processes have access to the page table
  - Client process has access to information provided by a server

Security Goals



Authentication

- User/process authentication
  - Is this user/process who it claims to be?
    - Passwords
    - More sophisticated mechanisms
- Authentication in networks
  - Is this computer who it claims to be?
    - File downloading
    - Obtaining network services
    - The Java promise

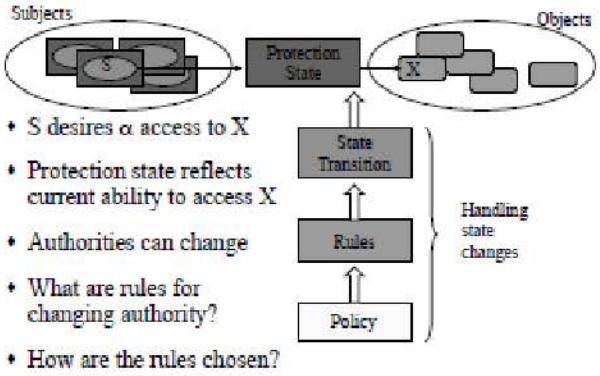
Authorization

- Is this user/process allowed to access the resource under the current policy?
- What type of access is allowable?
  - Read
  - Write
  - Execute
  - Append

Lampson's Protection Model

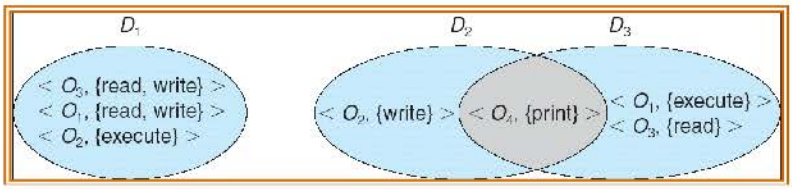
- Active parts (e.g., processes)
  - Operate in different domains
  - Subject is a process in a domain
- Passive parts are called objects
- Want mechanism to implement different security policies for subjects to access objects
  - Many different policies must be possible
  - Policy may change over time

A Protection System



Domain Structure

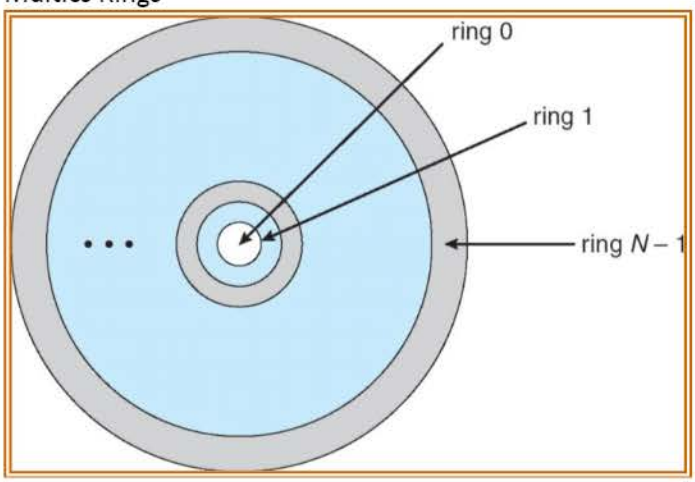
Access-right = <object-name, rights-set>
where rights-set is a subset of all valid operations that can be performed on the object.



Protection Domains

- Lampson model uses processes and domains -- how is a domain implemented?
- In UNIX system consists of two domains: user and supervisor
  - Supervisor/user hardware mode bit
  - Domain = user-id, switch accomplished via file system.
    - Each file has associated with it a domain bit (setuid bit).
    - When file is executed and setuid = on, then user-id is set to owner of the file being executed. When execution completes user-id is reset.
- Other systems employ software extensions -- rings
- Inner rings have higher authority
  - Ring 0 corresponds to supervisor mode
  - Rings 1 to S have decreasing protection, and are used to implement the O.S.
  - Rings S+1 to N-1 have decreasing protection, and are used to implement applications

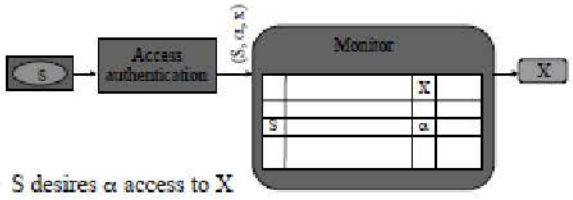
Multics Rings



Let Di and Dj be any two domain rings. If $j < i \Rightarrow Di \subseteq Dj$.

- Ring crossing is a domain change
- Inner ring crossing $\Rightarrow$rights amplification
  - Specific gates for crossing
  - Protected by an authentication mechanism
- Outer ring crossing uses less-protected objects
  - No authentication
  - Need a return path
  - Used in Multics and Intel 80386 (& above) hardware

Protection System Example



- Captures the protection state
- Generates an un-forgeable ID
- Checks the access against the protection state

Access Matrix

- View protection as a matrix (access matrix)
- Rows represent domains
- Columns represent objects
- Access(i, j) is the set of operations that a process executing in $Domain_i$ can invoke on $Object_j$

Example

domain \ object	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

Uses of Access Matrix

Implementing Access Matrix

- If a process in Domain Di tries to do "op" on object Oj, then "op" must be in the access matrix.
- Can be expanded to dynamic protection.
  - Operations to add, delete access rights.
  - Special access rights:
    - owner of Oi
    - copy op from Oi to Oj
      - control – Di can modify Dj access rights
      - transfer – switch from domain Di to Dj
- Access matrix design separates mechanism from policy.
  - Mechanism
    - Operating system provides access-matrix + rules.
    - It ensures that the matrix is only manipulated by authorized agents and that rules are strictly enforced.
  - Policy
    - User dictates policy.
    - Who can access what object and in what mode

- Usually a sparse matrix
  - Too expensive to implement as a table
  - Implement as a list of table entries
- Column oriented list is called an access control list (ACL)
  - Defines who can perform what operation on an object
  - List kept at the object
  - UNIX file protection bits are one example
- Row oriented list is a called a capability list (like a key)
  - For each domain, what operations allowed on what objects
  - List kept with the subject (i.e., process)
  - Kerberos ticket is a capability
  - Mach mailboxes protected with capabilities

More on Capabilities

- Provides an address to object from a very large address space
- Possession of a capability represents authorization for access
- Implied properties:
  - Capabilities must be very difficult to guess
  - Capabilities must be unique and not reused
  - Capabilities must be distinguishable from randomly generated bit patterns

Access Matrix, Domains as Objects

domain \ object	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

domain \ object	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch
D_3		read	execute					
D_4	read write		read write		switch			

Access Matrix with Copy and Owner Rights

Modified Access Matrix

object / domain	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute		

(a)

object / domain	F_1	F_2	F_3
D_1	owner execute		write
D_2		read* owner	read* owner write
D_3	execute		

object / domain	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute	read	

(b)

object / domain	F_1	F_2	F_3
D_1	owner execute		write
D_2		owner read* write*	read* owner write
D_3		write	write

object / domain	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch control
D_3		read	execute					
D_4	write		write		switch			

Revocation of Access Rights

- Access List – Delete access rights from access list.
  - Simple
  - Immediate
- Capability List – Scheme required to locate capability in the system before capability can be revoked.
  - Reacquisition
  - Back-pointers
  - Indirection
  - Keys

Capability-Based Systems

- Hydra
  - Fixed set of access rights known to and interpreted by the system.
  - Interpretation of user-defined rights performed solely by user's program; system provides access protection for use of these rights.
- Cambridge CAP System
  - Data capability – provides standard read, write, execute of individual storage segments associated with object.
  - Software capability – interpretation left to the subsystem, through its protected procedures.

Language-Based Protection

- Specification of protection in a programming language allows the high-level description of policies for the allocation and use of resources.
- Language implementation can provide software for protection enforcement when automatic hardware-supported checking is unavailable.
- Interpret protection specifications to generate calls on whatever protection system is provided by the hardware and the operating system.

Protection in Java

- Protection is handled by the Java Virtual Machine (JVM)
- A class is assigned a protection domain when it is loaded by the JVM.
- The protection domain indicates what operations the class can (and cannot) perform.
- If a library method is invoked that performs a privileged operation, the stack is inspected to ensure the operation can be performed by the library.

Stack Inspection

protection domain:	untrusted applet	URL loader	networking
socket permission:	none	*.lucent.com:80, connect	any

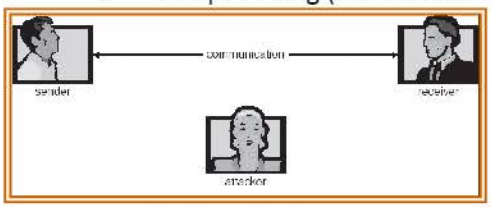
class:	gui:
	...
	get(url);
	open(addr);
	...
		get(URL u):
		...
		doPrivileged {
		open('proxy.lucent.com:80');
		}
		<request u from proxy>
		...
			open(Addr a):
			...
			checkPermission (a, connect);
			connect (a);
			...

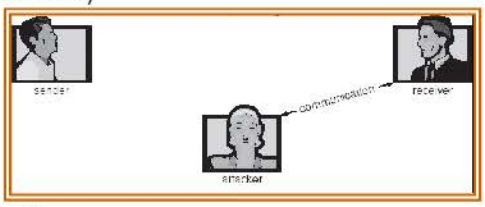
The Security Problem

Security Violations

- Security must consider external environment of the system, and protect the system resources
- Intruders (crackers) attempt to breach security
- Threat is potential security violation
- Attack is attempt to breach security
- Attack can be accidental or malicious
- Easier to protect against accidental than malicious misuse

- Categories
  - Breach of confidentiality
  - Breach of integrity
  - Breach of availability
  - Theft of service
  - Denial of service
- Methods
  - Masquerading (breach authentication)





  - Replay attack: message modification
  - Man-in-the-middle attack

  - Session hijacking

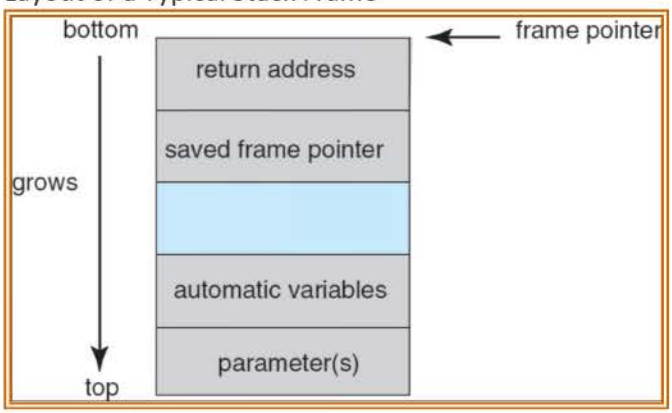
Security Measure Levels

- Security must occur at four levels to be effective:
  - Physical
  - Human
    - Avoid social engineering, phishing, dumpster diving
  - Operating System
  - Network
- Security is as week as the weakest chain

- Trojan Horse
  - Code segment that misuses its environment
  - Exploits mechanisms for allowing programs written by users to be executed by other users
  - Spyware, pop-up browser windows, covert channels
- Trap Door
  - Specific user identifier or password that circumvents normal security procedures
  - Could be included in a compiler
- Logic Bomb
  - Program that initiates a security incident under certain circumstances
- Stack and Buffer Overflow
  - Exploits a bug in a program (overflow either the stack or memory buffers)
    -

```
#include <stdio.h>
#define BUFFER SIZE 256
int main(int argc, char *argv[])
{
        char buffer[BUFFER SIZE];
        if (argc < 2) return -1;
        else {
                strcpy(buffer,argv[1]);
                return 0;
        }
}
```

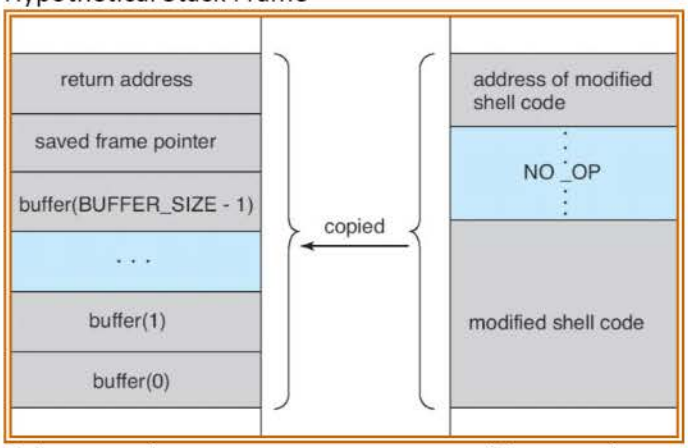
Layout of a Typical Stack Frame



Modified Shell Code

```
#include <stdio.h>
int main(int argc, char *argv[])
{
        execvp(''\bin\sh'',''\bin \sh'', NULL);
        return 0;
}
```

Hypothetical Stack Frame



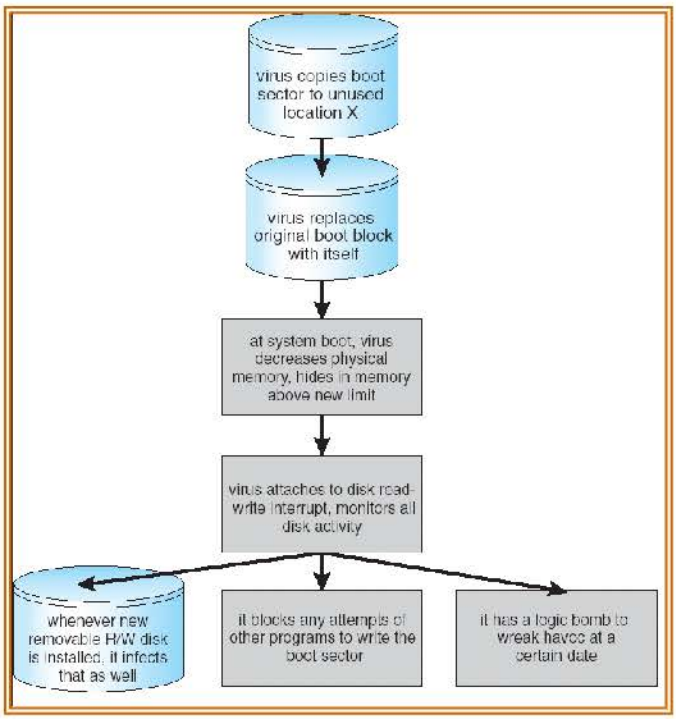
Before attack After attack

Program Threats A Boot-Sector Computer Virus

- Viruses
  - Code fragment embedded in legitimate program
  - Very specific to CPU architecture, operating system, applications
  - Usually borne via email or as a macro
  - Visual Basic Macro to reformat hard drive

```
Sub AutoOpen()
Dim oFS
  Set oFS =
CreateObject(''Scripting.FileSystemObject'')
  vs = Shell(''c:command.com /k format
c:'',vbHide)
End Sub
```

  - Virus dropper inserts virus onto the system
  - Many categories of viruses, literally many thousands of viruses
    - File
    - Boot
  - Many categories of viruses, literally many thousands of viruses
    - Macro
    - Source code
    - Polymorphic
    - Encrypted
    - Stealth
    - Tunneling
    - Multipartite
    - Armored

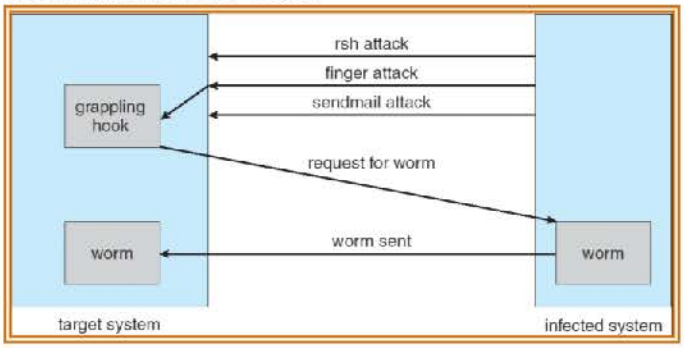


System and Network Threats

Program Threats

- Worms – use spawn mechanism; standalone program
- Internet worm
  - Exploited UNIX networking features (remote access) and bugs in finger and sendmail programs
  - Grappling hook program uploaded main worm program
- Port scanning
  - Automated attempt to connect to a range of ports on one or a range of IP addresses
- Denial of Service
  - Overload the targeted computer preventing it from doing any useful work
  - Distributed denial-of-service (DDOS) come from multiple sites at once

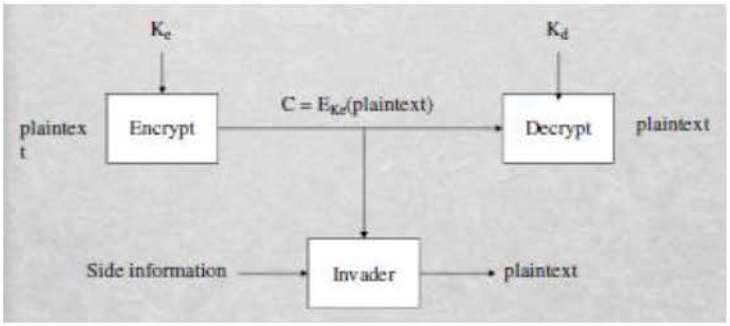
The Morris Internet Worm



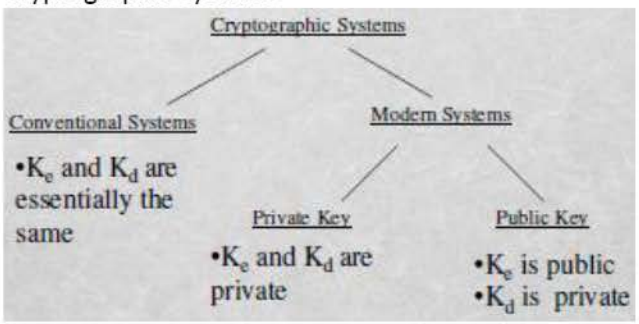
Cryptography

More on Cryptography

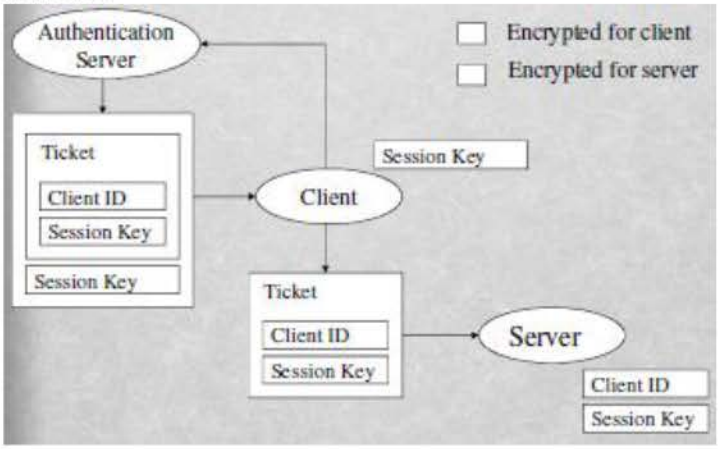
- Information can be encoded using a key when it is written (or transferred) -- encryption
- It is then decoded using a key when it is read (or received) -- decryption
- Very widely used for secure network transmission



Cryptographic Systems



Kerberos



Computer Security Classifications

- U.S. Department of Defense outlines four divisions of computer security: A, B, C, and D.
  - D – Minimal security.
  - C – Provides discretionary protection through auditing.
    - Divided into C1 and C2.
    - C1 identifies cooperating users with the same level of protection.
    - C2 allows user-level access control.
  - B – All the properties of C, however each object may have unique sensitivity labels.
    - Divided into B1, B2, and B3.
  - A – Uses formal design and verification techniques to ensure security.

Example: Windows XP

- Security is based on user accounts
- Each user has unique security ID
- Login to ID creates security access token
  - Includes security ID for user, for user's groups, and special privileges
  - Every process gets copy of token
  - System checks token to determine if access allowed or denied
- Uses a subject model to ensure access security. A subject tracks and manages permissions for each program that a user runs
- Each object in Windows XP has a security attribute defined by a security descriptor
  - For example, a file has a security descriptor that indicates the access permissions for all users

--- END PROTECTION AND SECURITY