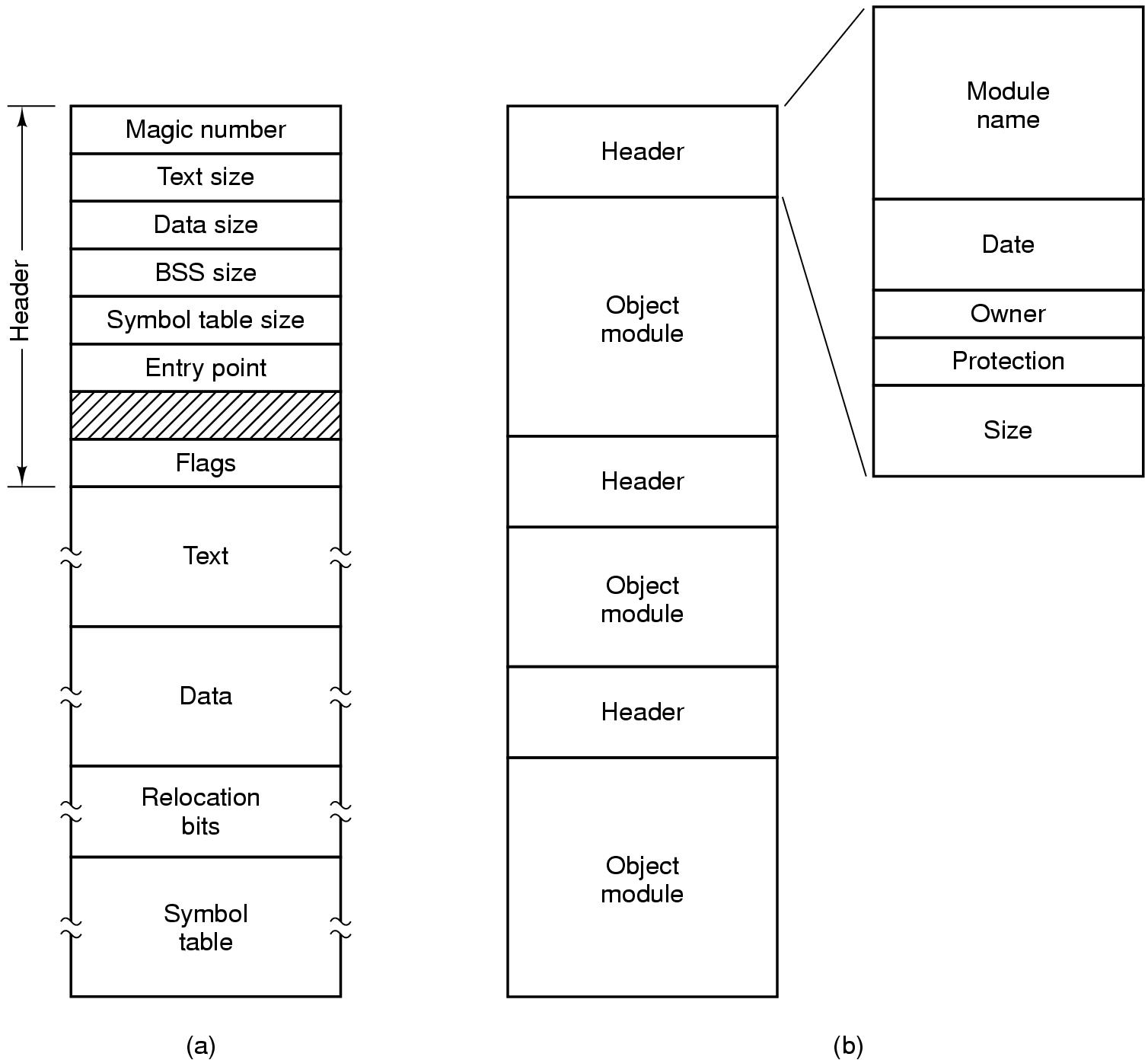
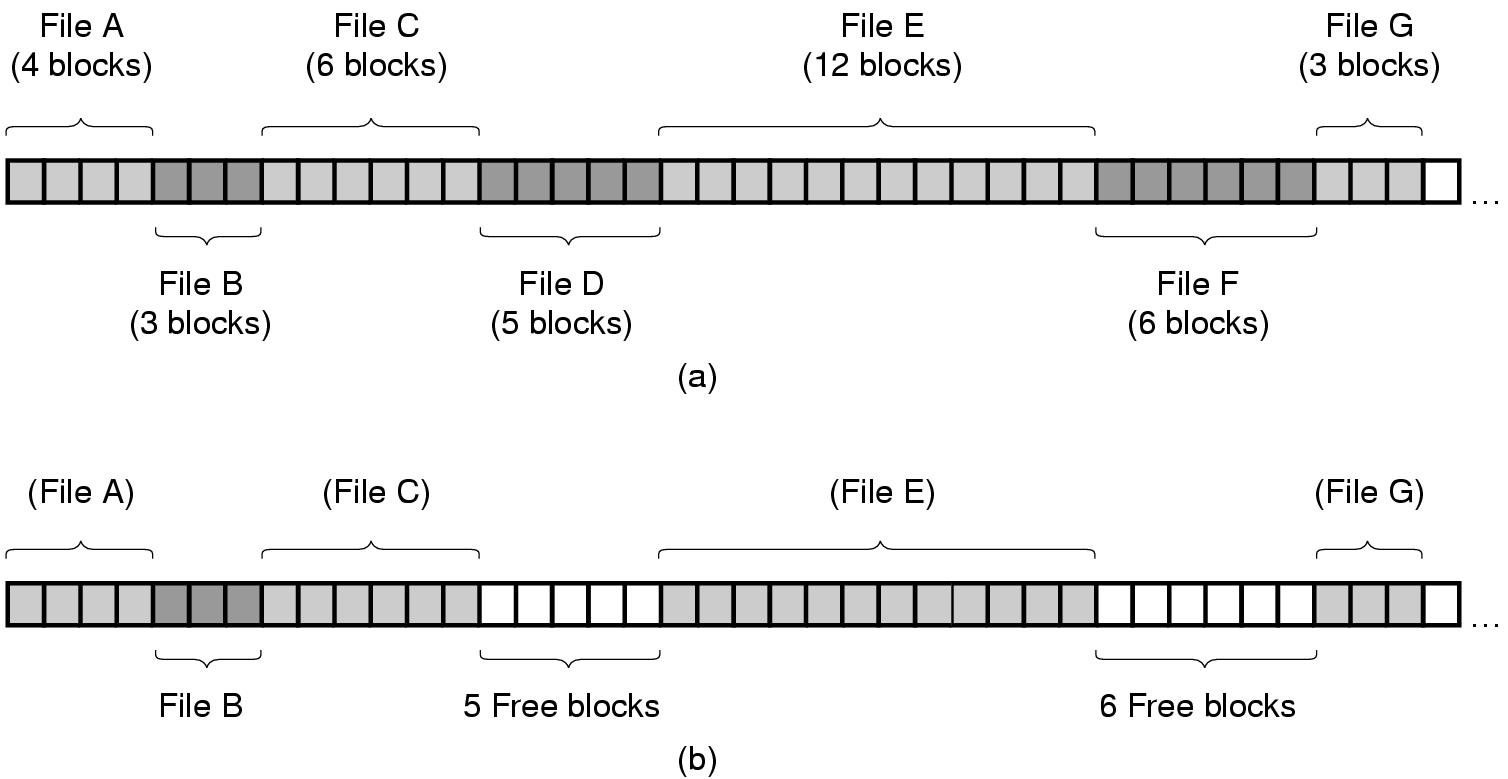
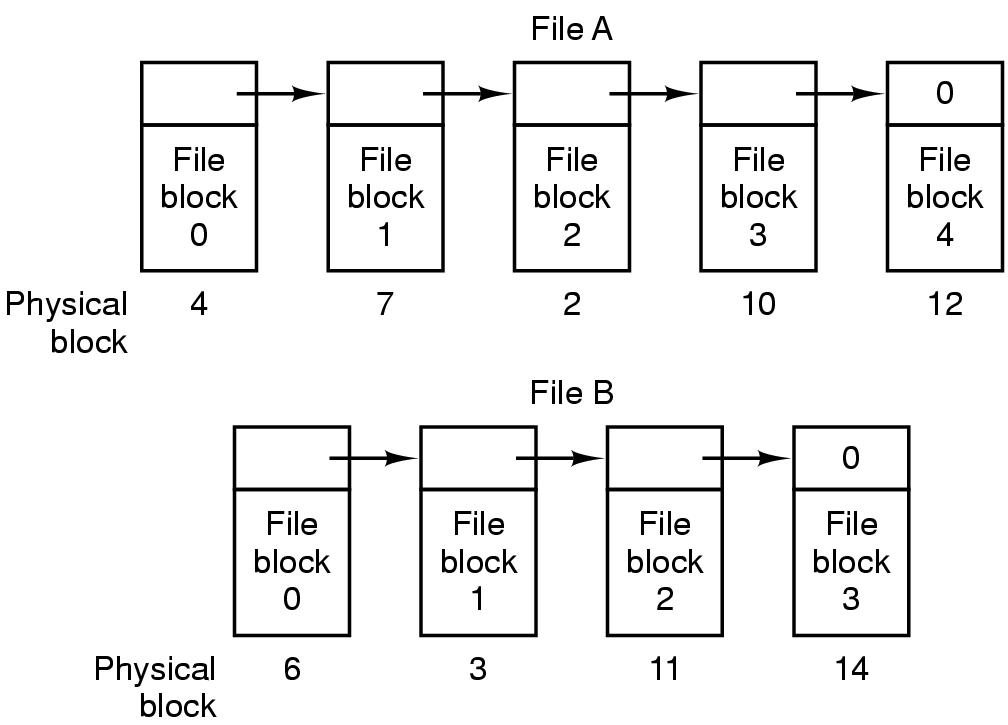
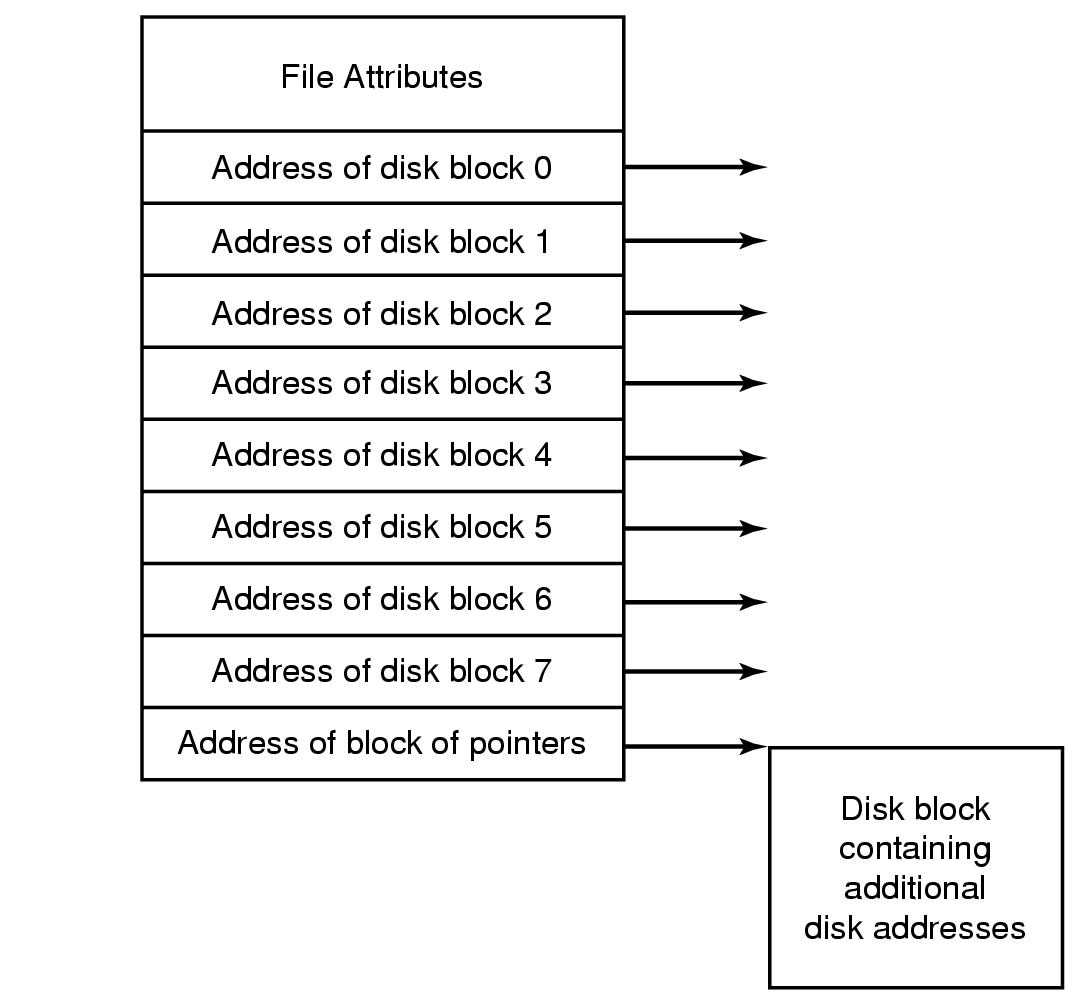
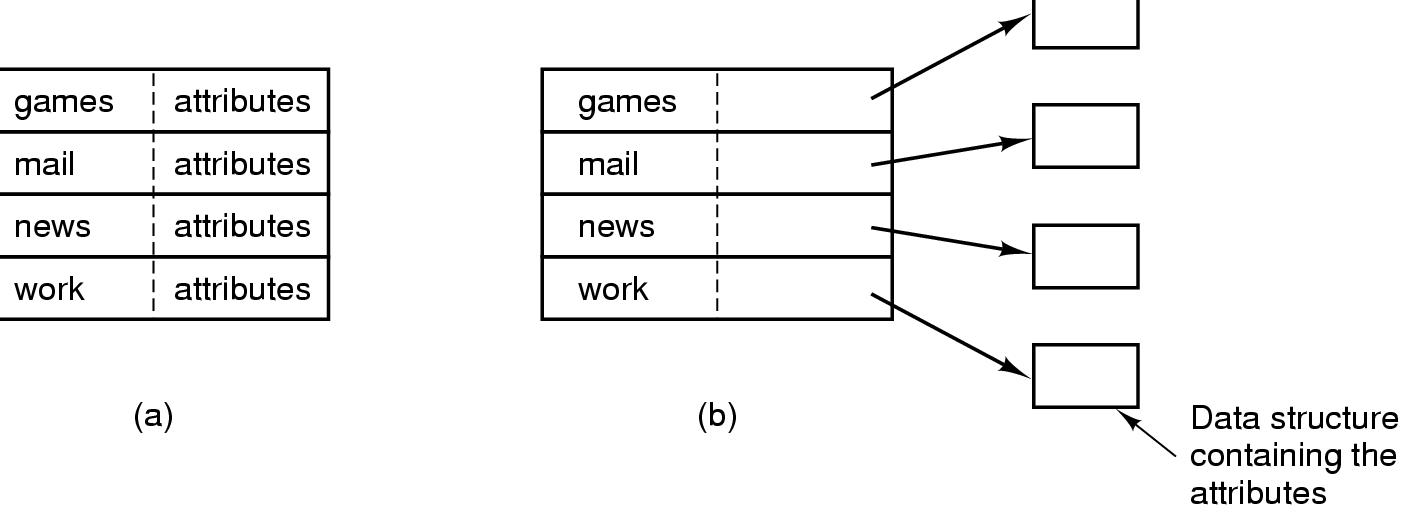
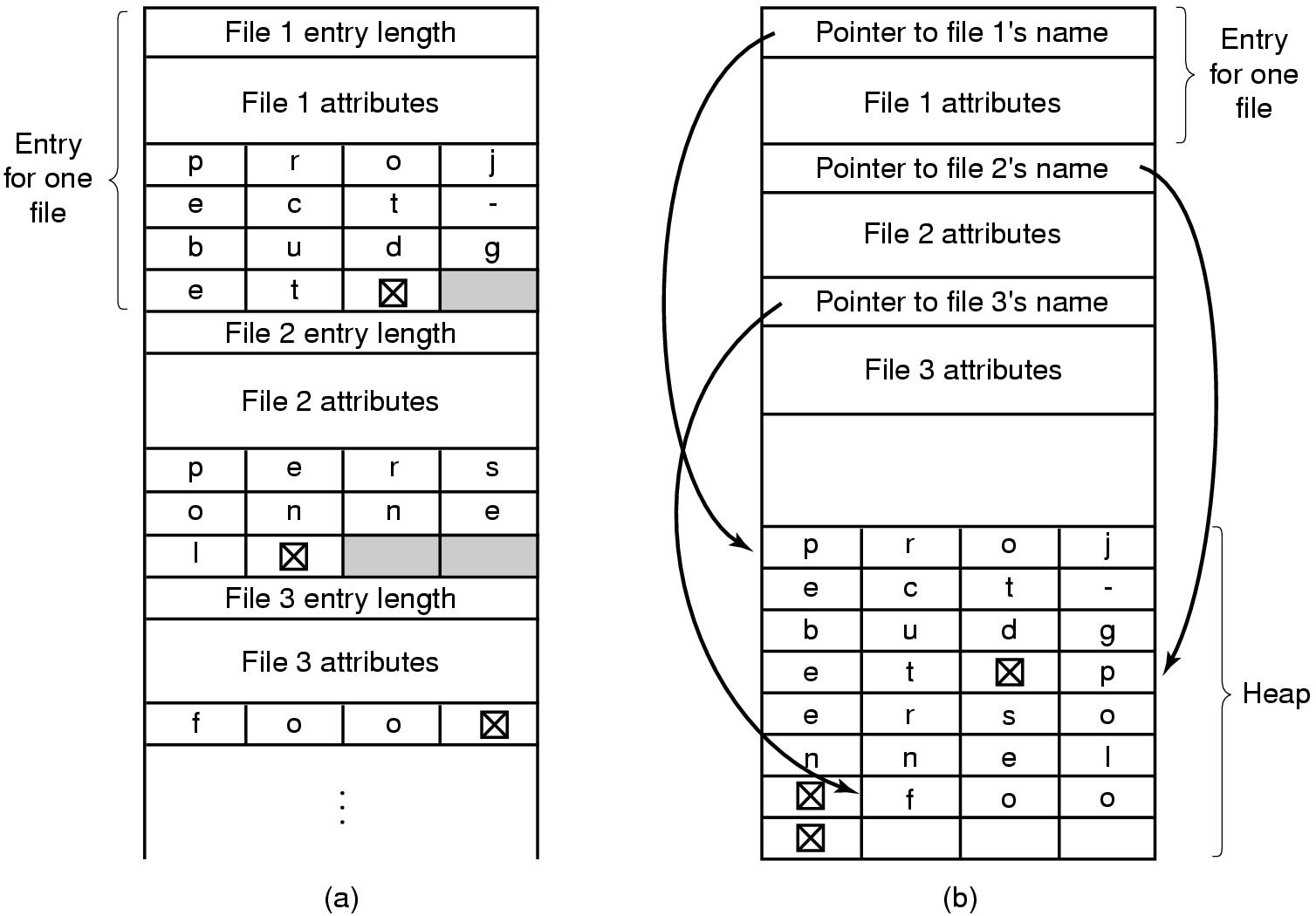
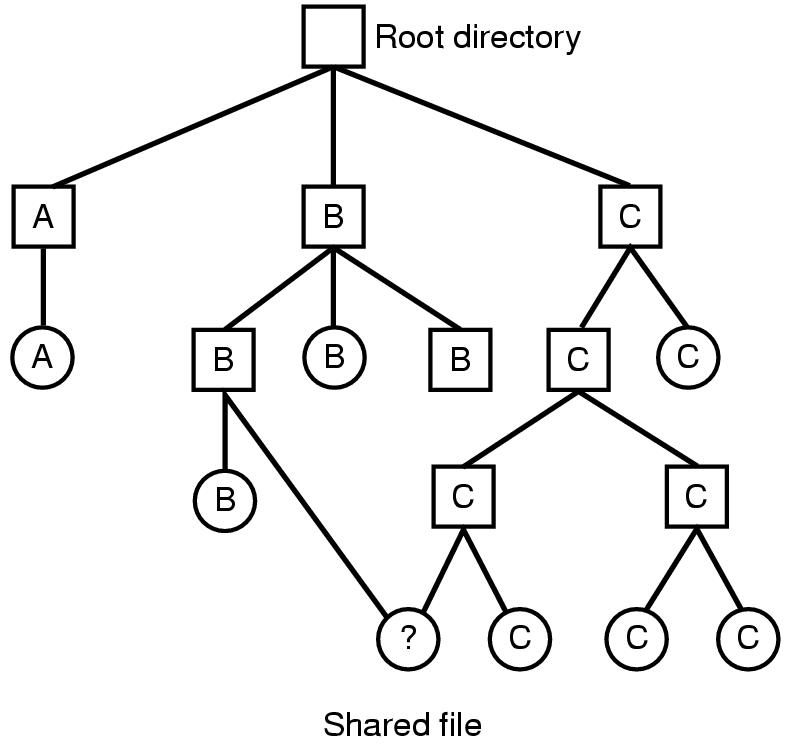
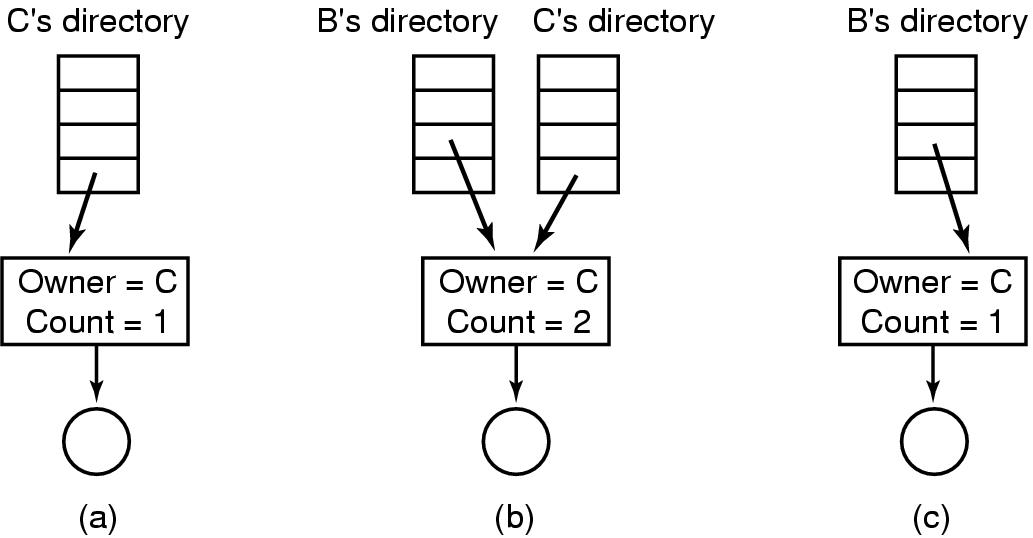
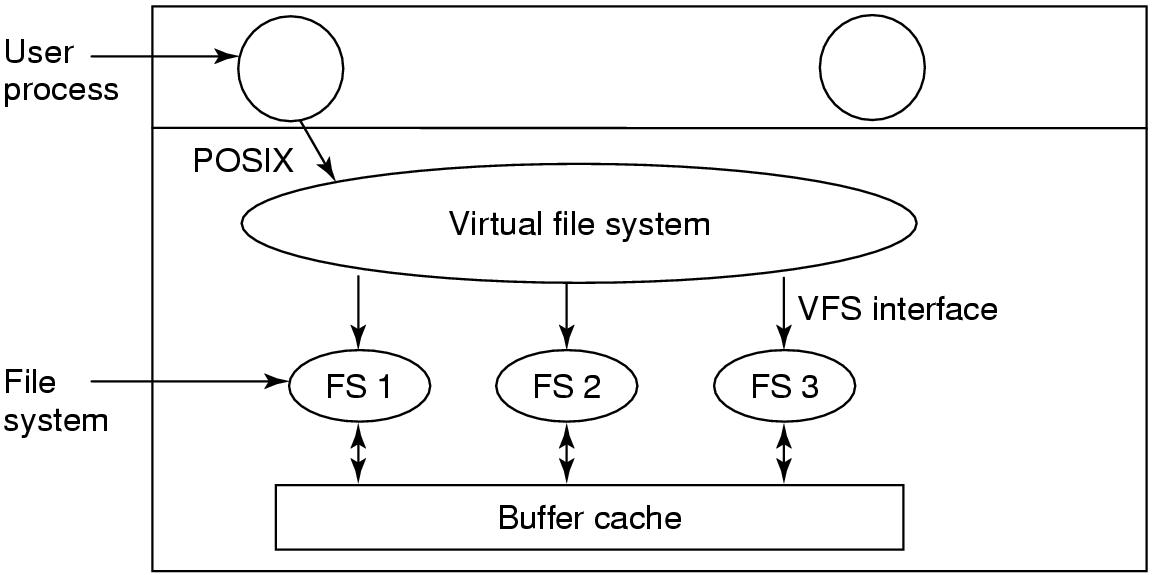
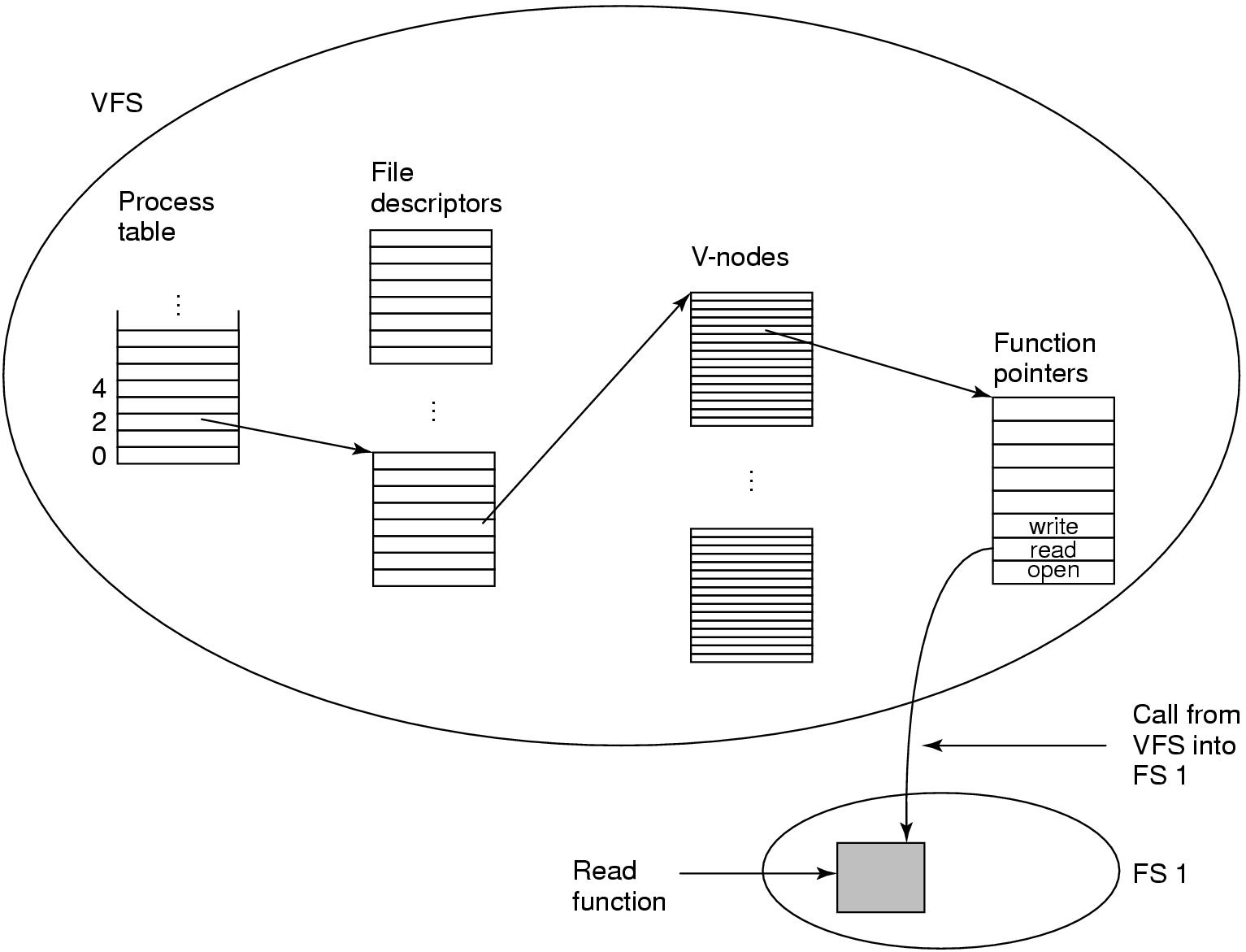
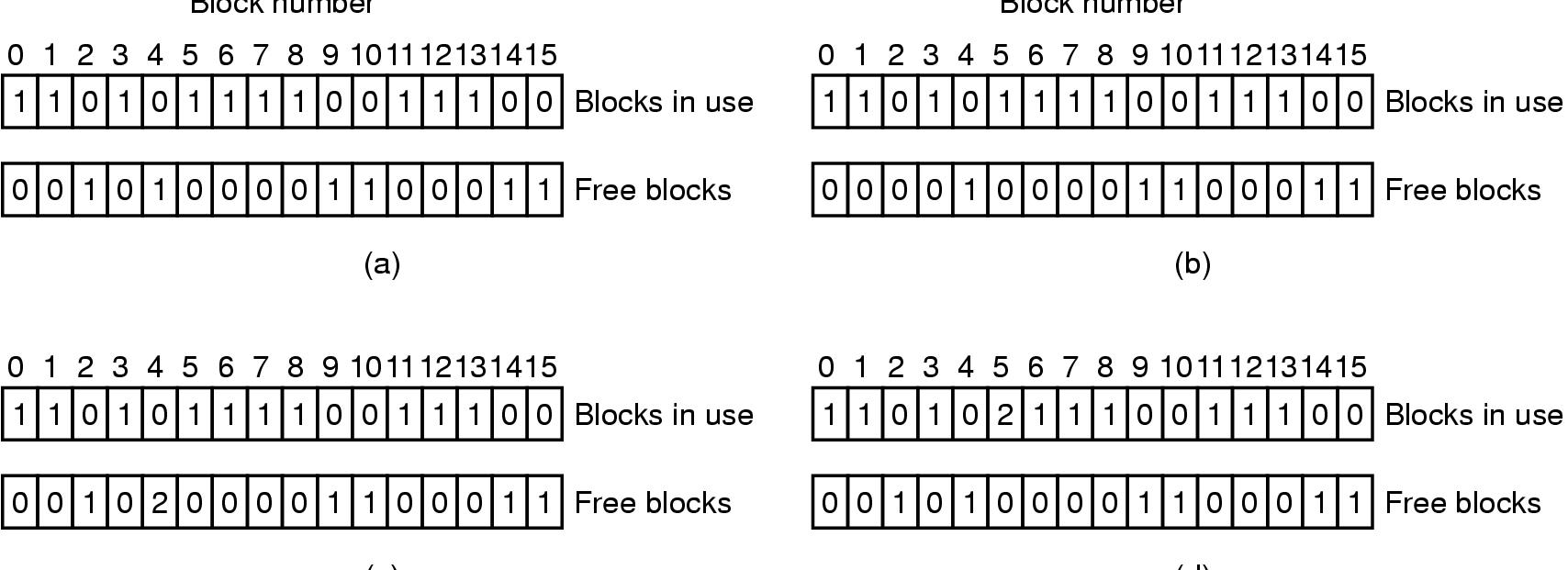
# Chapter 4...File systems

* Requirements
  + It must be possible to store a very large amount of information.
  + The information must survive the termination of the process using it.
  + Multiple processes must be able to access the information concurrently.
* Files is the OS way to abstract disks
* Linux is case sensitive regarding file names , Windows is not
* File structure
  + Byte sequence ( Linux and Windows ): Most flexible , apps determine meaning of file
  + Record sequence(fixed length records) : Old systems , read /write one record
  + Sorted Tree records (not the same length): Used to locate a record with a certain key , used in Data processing
* File Types
  + Regular , Directories
  + Character based : Model serial I/O devices like printers
  + Block based : to Model disks (CD)
* Executable file structure
  + Header contains magic number to tell OS a file is executable
  + Text (executable image) size , data size , symbol table (used for debugging),addresses and flags , and relocation bits

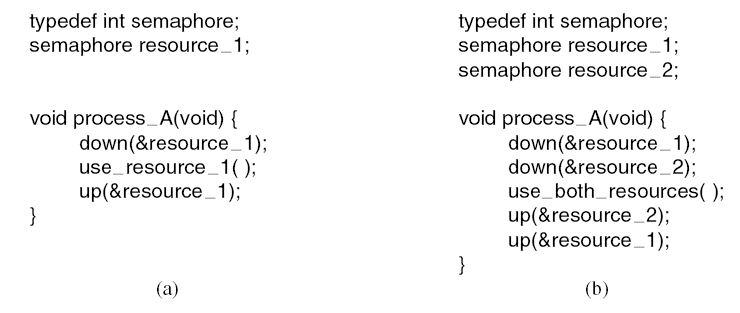
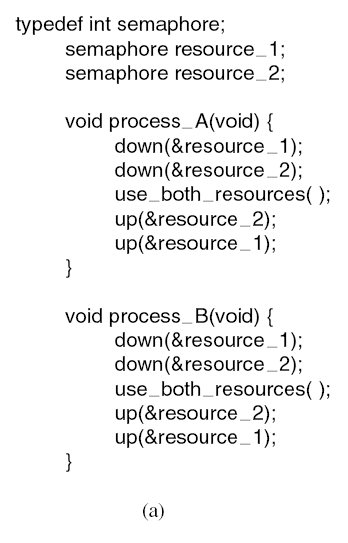
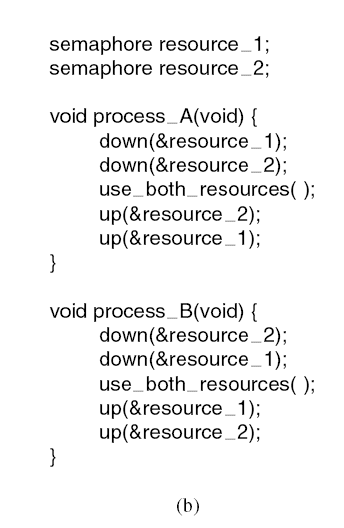
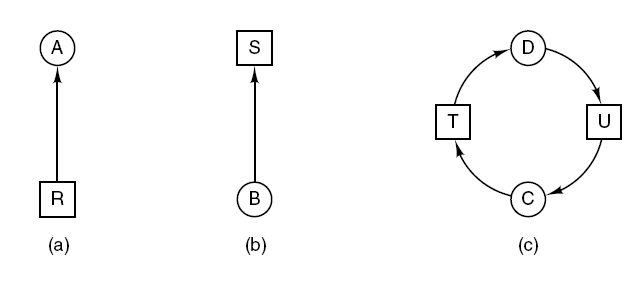
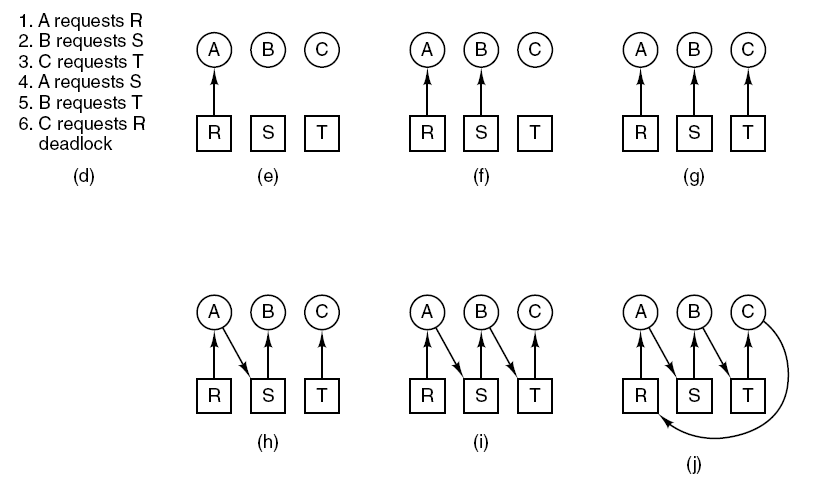
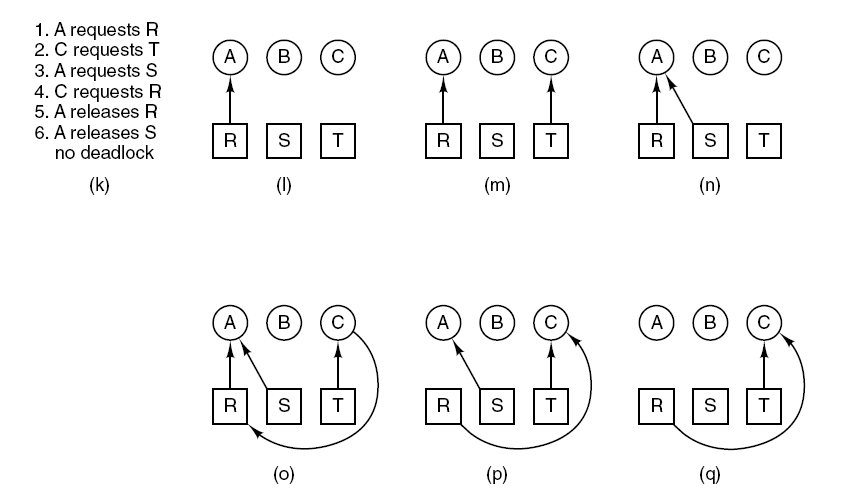
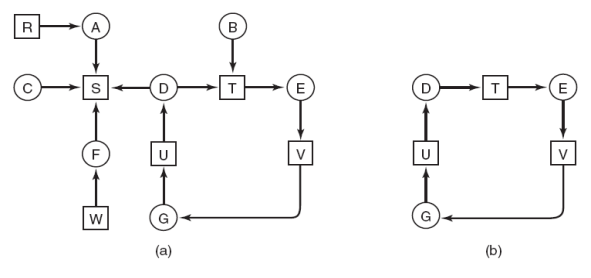
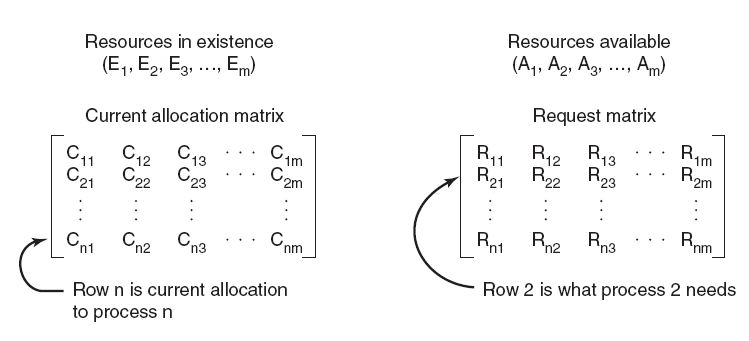
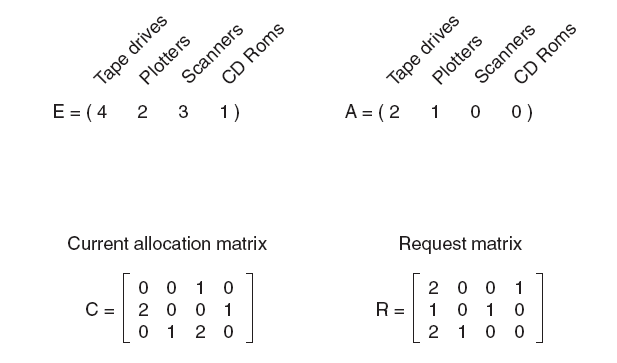
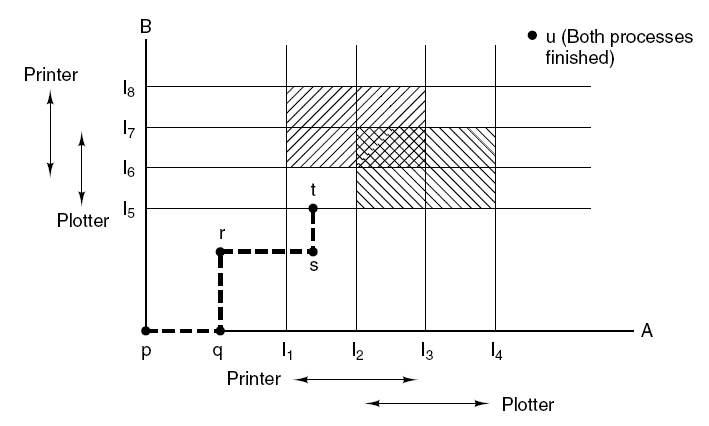
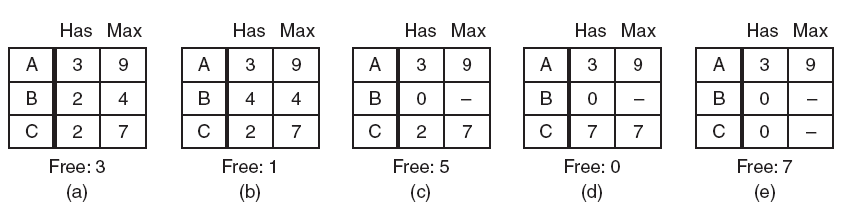
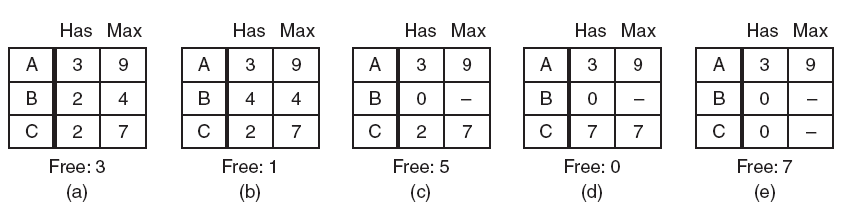
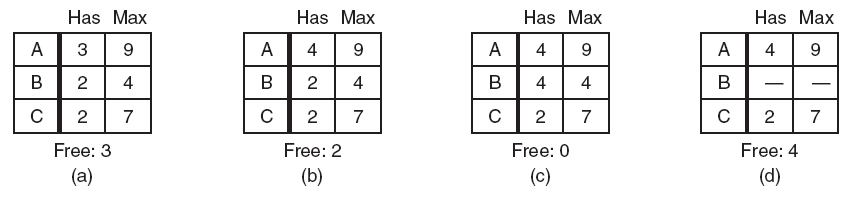
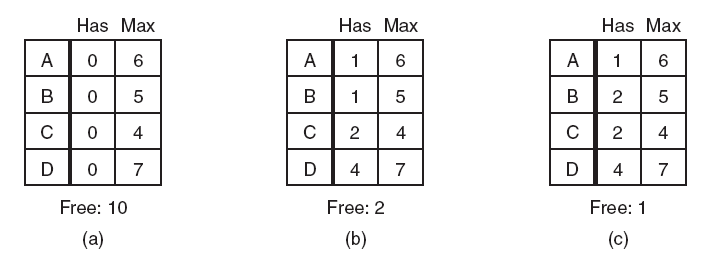
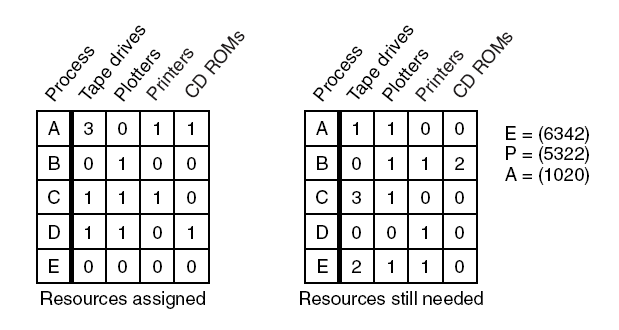
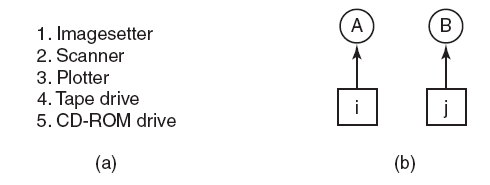
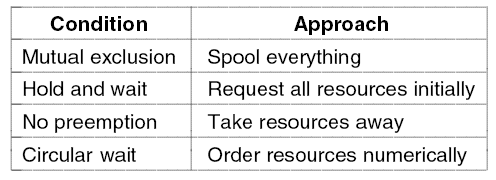
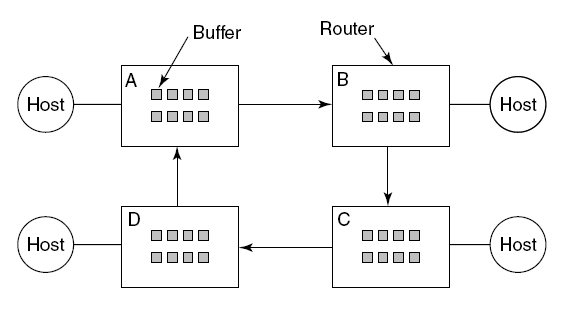
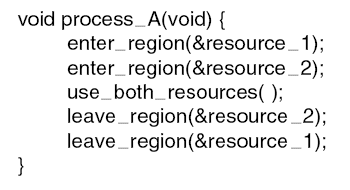
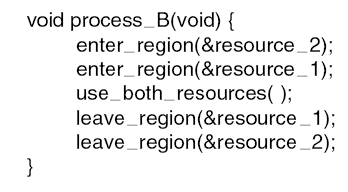


* File access : Sequential or random (via seek )
* File attributes: size, access/protection flags, owner, creation/modification date, password …
* File Operations : read ,append ,seek ,write ,open (return file descriptor, and load file attributes and address in memory), close (to clear memory ), rename , remove, get/set attributes
* Directories organizes into hierarchy , operations create , delete ,un/link
* Path : absolute and relative , set working directory ( not used in libraries as they are shared by multiple apps, it has to revert the working directory)
* File system layout
  + Disk is divided into multiple partitions , each one has its own file system
  + Sector 0 has master boot record , then partition table each entry has start and end address of partions, one of them is marked as active . Bios loads MBR , then active partition which contains (boot block ) where it loads OS
  + 
  + Super block: has magic number which identifies the file system, number of free blocks, root dir ,inodes array , the rest is the actual files data
* Contagious allocation : write file as a set of contagious blocks , 50 K file will be 50 block of size 1 K
  + Pros : simple to implement , to keep track of file store 2 numbers first block address, number of blocks
    - Read performance : entire file could be read with one seek operation
    - Suitable if all files sizes are known upfront e.g CD/DVD -ROMs
  + Cons :
    - fragmentation as over time gaps exists , to create a file it’s size need to be known upfront to find a suitable hole for it , not possible . also compaction is too costly
    - as file grows it needs to be moved
  + DVD (universal disk format) has limit on file size so with a large file can’t fit as one physical file -> **extends** : Physical parts of a big large file e.g a movie
  + 
* Linked list allocation: each block has pointer to next block address , directory stores address of the first block for each file .



* + Pros : No space is lost due to fragmentation
  + Cons : random access is extremely slow , OS has to read all blocks sequentially (list)
    - Each block is not a power of 2 , as part of it is used for pointer , causes extra overhead to load power of 2 data as need an extra read operation and concatenation
  + Linked list as table in memory : solves above problems by storing pointers in a table in memory (File allocation table )FAT
    - Pros : entire block is now available for data
      * Reading sequential addresses in memory is much faster than disk seeks
    - Cons : Entire table has to be in memory , not possible for large disks as entries are per block
    - 
* I-nodes (index nodes) : associated with each file contains block addresses and file attributes
  + Pros : given inode we can find all blocks , only inode resides in memory when file is opened , for total k opened files memory usage will be k\*n ( where n is number of entries for each inode), n is propostional to file size not disk size -> scales well
  + For large files : keep last entry for another block containing extra file blocks addresses
  + 
* Implementing directories: maps file name(path) to actual file blocks (data)
  + Directory as a list of fixed entries per file : each one contains file name , attributes , address(WINDOWS)
  + Directory as a list each entry is just a file name and refers to inode (via inode number) (LINUX)
  + Directory entries : fixed size =>File names fixed e.g. limit to 255 , wastes space in fixed size entries
  + 
    - Variable length file names:
      * starts with fixed length header contains file name size and other attributes , each entry is started at word boundy by filling file name to an integral number of words
        + Problem : when removing files it’s file name leaves a gaps with different sizes-> compacting is feasible as this list is in memory
      * Keep entries as fixed length ,but keeps file names in the heap at the end
        + Pros : no gaps when removing files , no need for filling chars at word boundaries
    - 
  + Solving problem of linear search in large directory entries list
    - * Hash tables to represent directories , Pros : faster file lookups
        + Cons : needs extra administrations , makes sense when OS know directory has thousands of files
      * Via Caching
* Shared files : a file is appears accessible by two different directories via links
  + Problem if copied disk addresses into new directory entries, when append to file only one directory will know about the new blocks
    - Via Inode : directories don’t store addresses instead refers to inodes (LINUX) ( hard links), it increases the reference count in inodes
      * Cons : when removing file by owner, the count is decreased but inode is still owned by original owner and referenced by other user , owner still pays for it from his user quota
    - Via Symbolic linking :Create a new file of type link which contains only the path to original file linked , OS will read the file determine it’s a link thus follows the path to the file ( soft links)
      * Pros : links could store links to remote files
      * Cons : extra overhead as the Link file has to be parsed first to fetch actual file
        + Extra inode is needed for each link , thus extra disk block
  + For all linking solutions : file can be repeated , thus an app that dumps all directories to a tape will dump a file twice as multiple paths locates the same file
  +  
* Virtual file systems : integrating/mounting different file systems and they all look like one , by abstracting code common between all file systems
  + 
  + POSIX API : interface common to all FSs ,used by programs
  + VFS API : interface implemented by FSs designers to integrate their file systems , Possible to retrieve files even remotely via NFS (network file system)
  + During booting/mounting FS will register itself to VFS by sending list of function addresses VFS is expected to call , later VFS will check a file path to determine it’s magic number ->FS create v-node ,and call actual FS copying the inode into v-node along with function pointers ,and then create and entry file descriptor referring to v-node then it can read /write
  + 
* Optimizing Block size : large wastes space as small files will hold one large block
  + - Small : wastes time as file will need large number of blocks , thus more seeks and rotation delay
* Consistency : FS read/modify blocks in memory then write them , problem is if it crashes before that specially for inodes or directory entries blocks , a tool like fsck checks for this
  + Block check : construct 2 tables one is for used blocks other is free blocks , reads all used blocks from inodes increment each block by 1 in first table , then for free blocks increment each in second table , FS is consistent IFF counter is 1 in only one table for each block
    - Missing block: marked 0 in both tables , harmless but wastes space -> add it to free blocks list
    - Duplicate free block : counter =2 in free blocks table -> rebuild free list
    - Duplicate data block: 2 files are using same block , can’t remove single file or it’ll be marked in both tables , can’t remove both files as it’ll be marked as free twice -> allocate a new block , copy contents to it , and mark it to be used with only one file ,report that error to user
  + File check : starts at root and recursively , for each inode create a counter of how many times the file/hard links are encountered , compare it with counter in inode . FS is consistent IFF they match
    - Inode link count is higher : means if all files are removed the inode will still be there , it wastes space -> fix the link count with the calculated counter value
    - Inode link count is lower : catastrophic , means if 2 directory entries are referring to a file and one is removed , count will be 0 and block marked to be free or even used for other file , the first file now refer to non existent block -> fix as above ,the link count to calculated value
  + 
* Performance : block/buffer cache ,keep blocks in memory use modified LRU, use hash table for addresses, and double linked list to store collisions , double list allows for easy move to front of list
  + Critical blocks which affect consistency of FS , can’t stay in memory for long time when modified (e.g. inode), thus immediately write them to disk
  + If block is not frequently used move it to front of list to allow it to be written to disk sooner
* Linux periodically writes buffers , windows flush cache on every write (write through cache)

# Deadlocks

* Two processes asks for exclusive access to resources already held by each other, both block forever
* Could happen between machines if resources can be reserved remotely like a shared printer network
* Preemptable resource : a resource that could be taken away from the process with no effects , e.g. memory
* Non preemptable resource : operation will fail if held resource is given to other process , e.g CD recorder, printer
* Process request resource, use it, release it. if request failed process is blocked ( put to sleep)
  + 
* Request order can cause deadlocks ( case b below)
*  
* DeadLock : *A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause , as all is waiting thus all will block forever*
* Conditions for Resource Deadlocks ( all 4 must present to have a deadlock)
  + Mutual exclusion condition : each resource is assigned to exactly one process
  + Hold and wait condition : process can hold a resource and request/wait for more resources
  + No preemption condition: resource can’t be taken away from process before it releases it
  + Circular wait condition : chain of waiting processes for resources held by each other
* Deadlock modeling (DAG): resource R is assigned to process A , process B is requesting/waiting for resource S
* Cycle in DAG means deadlock
* Cycle -> Deadlock!!
* Deadlock avoided by blocking B Request to S
* Strategies for dealing with deadlocks:
  + Just ignore the problem, if it doesn’t happen frequently enough
  + Detection and recovery. Let deadlocks occur, detect them, take action.
  + Dynamic avoidance by careful resource allocation.
  + Prevention, by structurally negating one of the four required conditions.
* Deadlock Detection:
  + One resource of each type , DAG has a cycle
    - 
    - Algorithm to detect a cycle ( DFS like )
      * For each node, N in the graph, perform the following five steps with N as the starting node.
      * Initialize L to the empty list, designate all arcs as unmarked.
      * Add current node to end of L, check to see if node now appears in L two times. If it does, graph contains a cycle (listed in L), algorithm terminates.
      * From given node, see if any unmarked outgoing arcs. If so, go to step 5; if not, go to step 6.
      * Pick an unmarked outgoing arc at random and mark it. Then follow it to the new current node and go to step 3.
      * If this is initial node, graph does not contain any cycles, algorithm terminates. Otherwise, dead end. Remove it, go back to previous node, make that one current node, go to step 3.
      * In above DAG , start at B and list L will be [B,T,E,V,G,U,D], now add T again and acycle!
  + Multiple resource of each type
    - Ei , resources exists of type i
    - Deadlock detection algorithm ( compare vectors A<B if all As less than or equal Bs
      * Look for an unmarked process, *Pi* , for which the i-th row of *R* is less than or equal to *A*.
      * If such a process is found, add the *i-th* row of *C* to *A*, mark the process, and go back to step 1
      * If no such process exists, the algorithm terminates, all unmarked processes are deadlocked
    - It’s looking for process that runs to completion , all it’s requests can be provided by available resources , so simulates as if it’s completed , add it’s allocated resources {C} to available and check others , if a process can’t be marked thus can’t complete hence it’s deadlocked
    - 
      * R1 <=A no ,R2 no , R3 <=A :yes , so add C -> A{2,2,2,0}, mark p3 , now R2 <=A :yes mark p2 and add C to A -> A{4,2,2,1}, no R1 yes; mark p1-> A{4,2,3,1}==E (all resources ) , no process is unmarked so no deadlock exists
    - Run deadlock detection every resource request ( slow )-> run each K minutes or when CPU utilization drops , the reason if there is a deadlock processes won’t be working and CPU drops
* Deadlock recovery:
  + Preemption : take a resource from a process block it and give that to other one -> not a solution as some resources can’t be allocated this way e.g. printers after printing some pages
  + Rollback: checkpoint process periodically (write state to file so it can be restarted later) , include resource status , new checkpoints don’t override file instead has new files .
    - At deadlock: rollback process A before it acquired the resource, start old checkpoint again , and allocate resource to other process B, if A tries to acquire it again while B holds it then it waits
  + Killing process: pick deadlocked process kill it give it’s resources to other , or kill non deadlocked one and give it’s resources to deadlocked ones .Choose process which could be rerun without problems (e.g compiler can be rerun ok , but DB can have side effects if killed while in the progress )
* Deadlock Avoidance:
  + Problem is R ( all needed resources) is not known at advance , process asks for resources as it’s running , to prevent deadlock OS checks if granting request will lead to a deadlock or it’s safe to grant it
*  A moves horizontally , B vertically ( one process at a time ) , double hashed area is unsafe, at t it’s unsafe so run A until it finishes then B -> u
* Safe state : if there is some scheduling order where all process can run to completion even if all of them suddenly asked for all their maximum number of resources immediately
*  
* {{a}} A needs 6, B needs 2 , C needs 5 , free = 3, fullfil B by 2 , now B can run and return allocated resources(4), now we have free=5 -> {{c}} . fulfill C {{d}}, now fulfill A {{e}} , hence it’s safe state
* Unsafe state : can’t schedule to fulfill all and stuck . it doesn’t mean deadlock state , but it means no guarantee it could be made to run all processes to completion
* start from (b) unsafe state
* Banker’s algorithm (single resource)
  + It checks each request and check as if it was granted , if lead to unsafe state then it’s blocked
  + a)safe , b) safe c)unsafe
* Banker’s (multiple resources)
* 
* Look for row, R, whose unmet resource needs all ≤ A. If no such row exists, system will eventually deadlock since no process can run to completion
* Assume process of row chosen requests all resources it needs and finishes. Mark process as terminated, add all its resources to the A vector.
* Repeat steps 1 and 2 until either all processes marked terminated (initial state was safe) or no process left whose resource needs can be met (there is a deadlock).
* Cons : in reality knowing requests in advance is impossible
* In figure above : D can run (R<=A) , then E or A then B or C ( then it’s safe)
* Now if B asks for printer and then E , Then A={1000}, then it’s unsafe and E request is deferred
* Deadlock prevention :
  + Attacking mutual exclusion: if no resource is allowed to be assigned exclusively to a signle process deadlocks can never happen. By spooling resource ( printer), all processes can write output at same time , only one process (Daemon) has only access to printer .
    - Daemons only print when output file is complete , this might lead to other type of deadlock , if two process writes only half of output and filled the spool directory space -> Disk deadlock
* Attacking hold and wait : prevent processes holding a resource to wait for more resources .
  + Ask all process to acquire all their resources at once before executing , else wait . Problems : process don’t know in advance all their resources , inefficient if a process asks for two resources and only use the other after hours of processing ( like tape to write output) .
  + Ask programmer to list all resources ( in batch job systems). Wastes resources , programmer burden .
  + When a process needs more resource , temporarily release resources it currently has , try again to get all resources at once or wait-> no deadlock
* Attacking no preemption: if it’s possible to forcibly take a resource from a process while it’s running and assign it to another one. problem: might be impossible for some resources . -> virtualized resources like spooling printer. cons: not all resources can be virtualized e.g. records in DB must be locked to be used
* Attacking circular wait : naïve soln : process only has one resource , if needs more it has to release what it has (not acceptable option e.g. process reads huge file from tape and write to printer)
  + Global numbering : process can ask for resources as long request are made in numerical order (e.g printer first then tape not the reverse ) -> resource allocation DAG will have no cycles
  + if i>J A can’t ask for j , if j>i B can’t ask for i , thus no cycles can ever happen
  + Variation: if process has 9 ,10 and released them then it can start fresh asking for lower numbers(1,2,..)
  + Cons : finding ordering schema that can fit everyone is not easy
* 
* Two phase locking : specialized in locking DB records . first phase: process tries to lock all records it needs one at a time .if success start second phase where it process and release locks .Else , it releases all its current locks and try phase 1 again .
  + Cons: Not applicable in general e.g. real time systems can’t simply terminate a process and start allover again , similarly network where it reads packets , or any thing that can’t be repeated , only certain apps .
* Communication deadlock: More abstract resources e.g messages in network sender waits for ACK ,or message is lost both process will block forever .
  + Use timeouts : if timer goes off before getting reply resend message -> communication protocol
  + Resource deadlock in network :
  + if buffers in routers are full , no packets could be moved (deadlock)
* Live lock :When using busy waiting to acquire locks , 2 process are consuming CPU in busy waiting forever ( not blocked but are working live with no progress) , -> busy waiting leading to a live lock
  +  
  + Ex: max number or processes is determined by number of entries in process table ,if 10 process are forking 12 subprocess and if failed to find entry try again , we may have full process table (100) and all of them are busy waiting to add more new sub processes -> live lock
  + Ex: max number of opened files , swap space on disk , in fact every table in OS is a finite resource -> solving by adding restricting rules but it’s too costly and inconvenient .
  + Difference between deadlock( blocked process) vs livelock : process in livelock is still running so it has option to do something else if request fails n times , blocked process in deadlock has no such option .
* Starvation: if multiple processes are asking for resources some allocation policies might lead to a process never gets a chance to acquire its resources although it’s not deadlocked.
  + Ex: allocate printer to smallest file first->a process with huge file will wait forever for other shorter files
  + Soln: First come first served policy , at some time that process will be waiting enough to be processed