Operating System Concepts

Lecture 31: File System Implementation

Omid Ardakanian
oardakan@ualberta.ca
University of Alberta

MWF 12:00-12:50 VVC 2 215

Mounting a file system

- directory structure may be built out of several volumes which must be mounted before the data stored on them become available within the file system namespace

Mounting a file system

- directory structure may be built out of several volumes which must be mounted before the data stored on them become available within the file system namespace
- mounting a file system is to verify that the specified device contains a valid file system, and make it available within the file-system namespace by attaching it to a certain location in the directory structure (i.e. the **mount point**)
  - the mount point is a special directory which has a flag set in its FCB
  - FCB of this directory also contains a pointer to the **mount table** which indicates which device is mounted there

Mounting a file system

- directory structure may be built out of several volumes which must be mounted before the data stored on them become available within the file system namespace
- mounting a file system is to verify that the specified device contains a valid file system, and make it available within the file-system namespace by attaching it to a certain location in the directory structure (i.e. the **mount point**)
  - the mount point is a special directory which has a flag set in its FCB
  - FCB of this directory also contains a pointer to the **mount table** which indicates which device is mounted there
- users switch among file systems as they traverse the directory structure
  - this is handled transparently

Mounting a file system

- directory structure may be built out of several volumes which must be mounted before the data stored on them become available within the file system namespace
- mounting a file system is to verify that the specified device contains a valid file system, and make it available within the file-system namespace by attaching it to a certain location in the directory structure (i.e. the **mount point**)
  - the mount point is a special directory which has a flag set in its FCB
  - FCB of this directory also contains a pointer to the **mount table** which indicates which device is mounted there
- users switch among file systems as they traverse the directory structure
  - this is handled transparently
- the directory structure within the mounted file system can be accessed by preceding the directory names by the mount point path name
  - so if `/dev/sdb` mounted over `/home` contains a directory called `userx` its full path name will be `/home/userx`

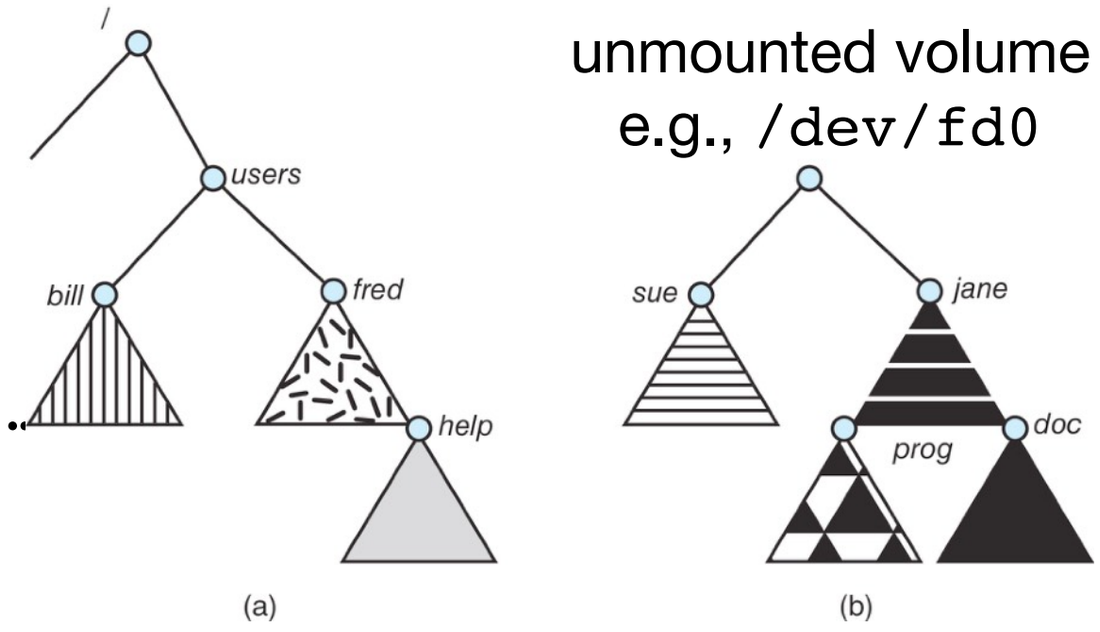
Mounting a file system

- directory structure may be built out of several volumes which must be mounted before the data stored on them become available within the file system namespace
- mounting a file system is to verify that the specified device contains a valid file system, and make it available within the file-system namespace by attaching it to a certain location in the directory structure (i.e. the **mount point**)
  - the mount point is a special directory which has a flag set in its FCB
  - FCB of this directory also contains a pointer to the **mount table** which indicates which device is mounted there
- users switch among file systems as they traverse the directory structure
  - this is handled transparently
- the directory structure within the mounted file system can be accessed by preceding the directory names by the mount point path name
  - so if `/dev/sdb` mounted over `/home` contains a directory called `userx` its full path name will be `/home/userx`
- mount point is usually an empty directory
  - but what if it is not empty?
    - disallow the mount
    - obscure the directory's existing files until the file system is unmounted

Mounting over a nonempty directory

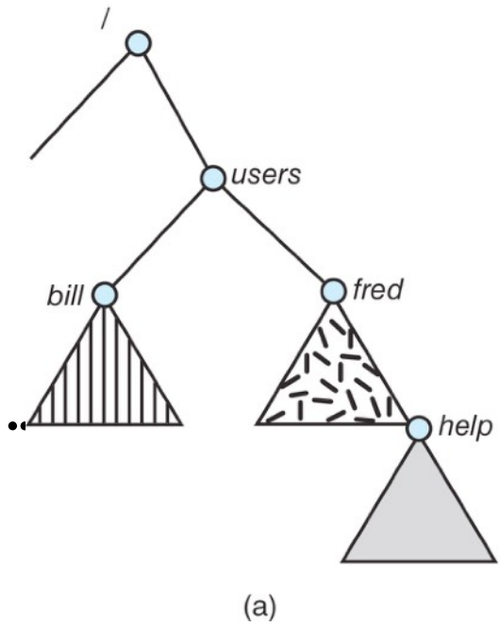
the existing file system, e.g.,
root partition containing the OS

unmounted volume
e.g., `/dev/fd0`



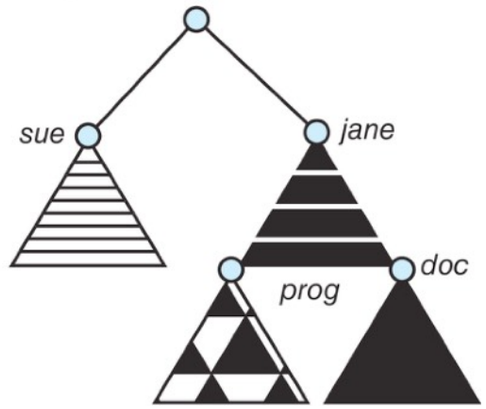
Mounting over a nonempty directory

the existing file system, e.g.,
root partition containing the OS



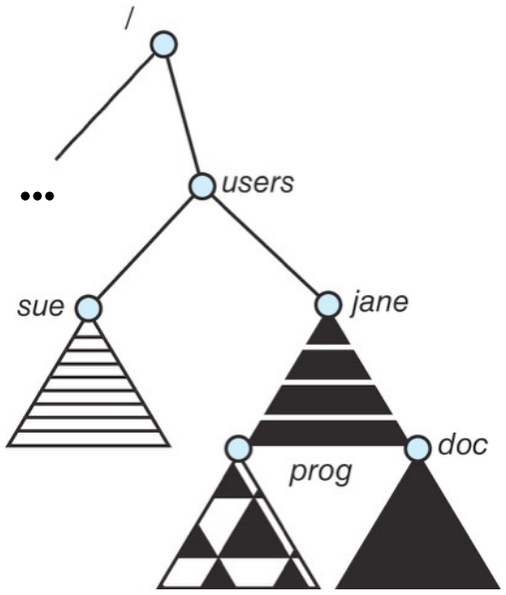
(a)

unmounted volume
e.g., `/dev/fd0`



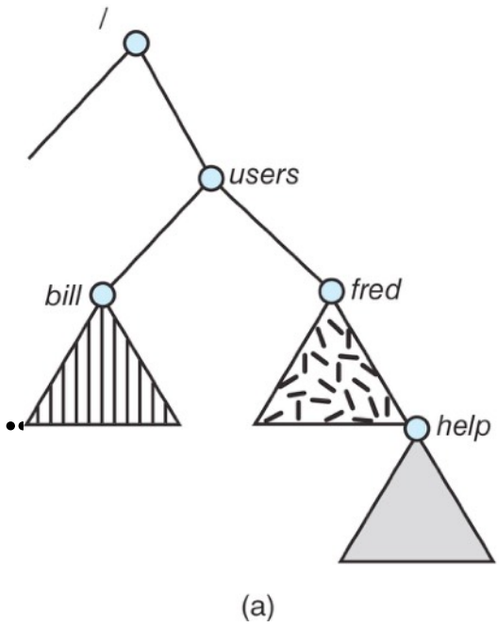
(b)

mounting over `/users`



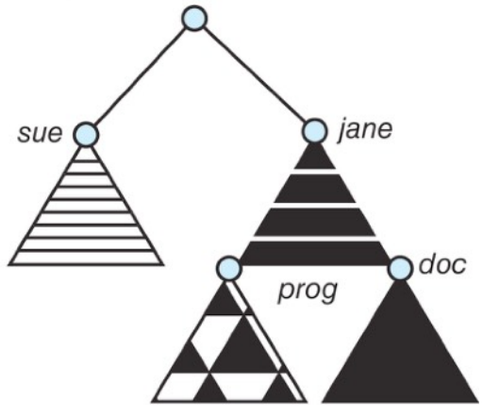
Mounting over a nonempty directory

the existing file system, e.g.,
root partition containing the OS



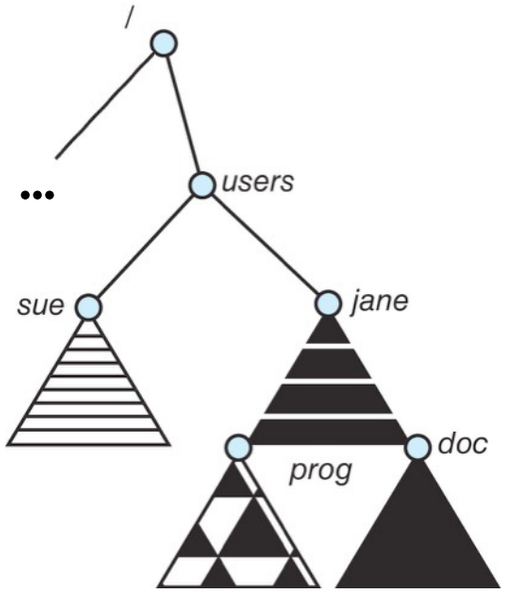
(a)

unmounted volume
e.g., `/dev/fd0`

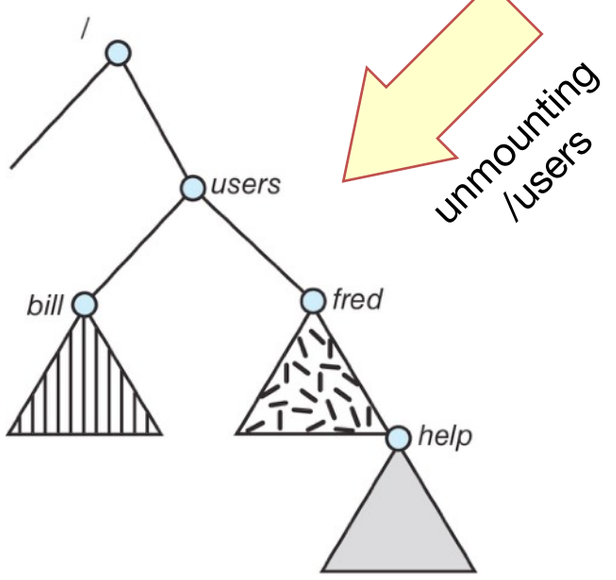


(b)

mounting over `/users`



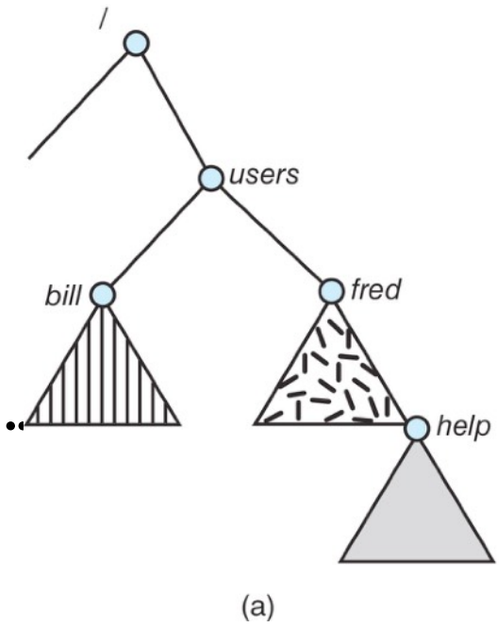
unmounting /users



(a)

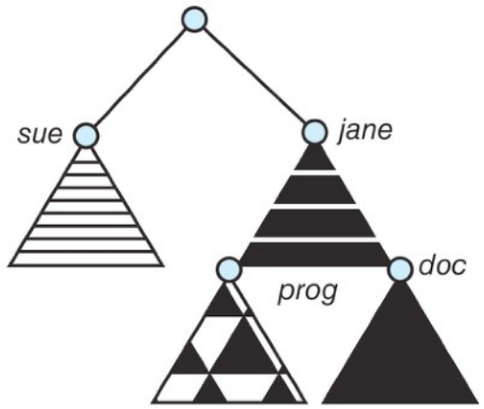
Mounting over a nonempty directory

the existing file system, e.g.,
root partition containing the OS



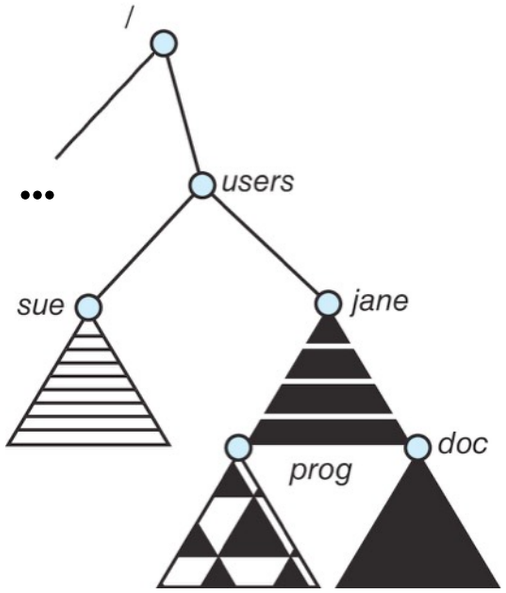
(a)

unmounted volume
e.g., `/dev/fd0`

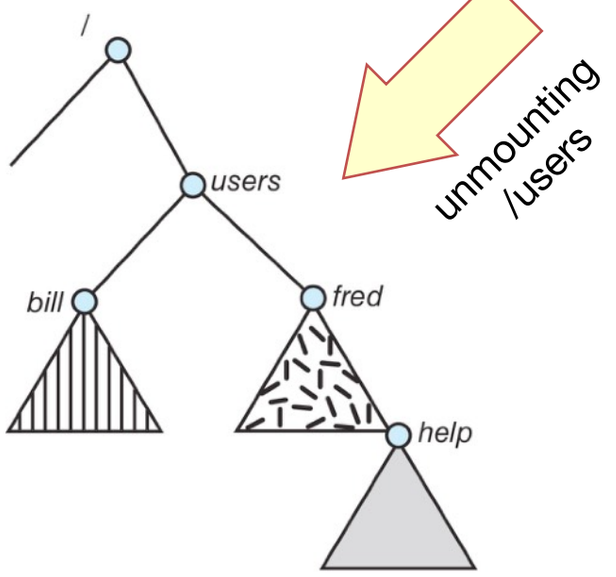


(b)

mounting over `/users`



unmounting /users



(a)

see `man mount` in Linux

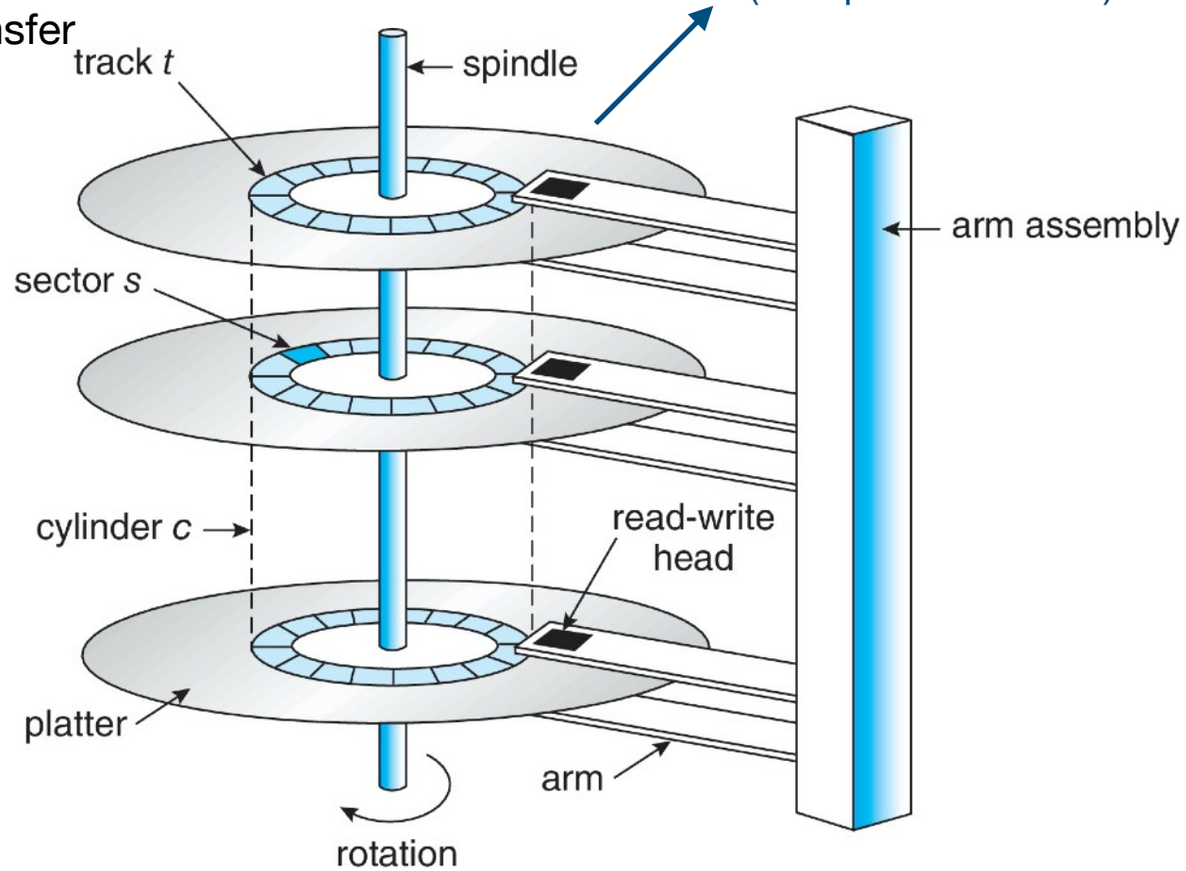
Today's class

- How file system requests disk I/O and how it is handled?
- How disk blocks are allocated to files?
  - contiguous allocation
  - linked allocation
  - indexed allocation
  - multilevel indexed allocation

Disk structure

- disks come organized in disk pack consisting of a stack of circular platters
  - each platter has two sides where each side has its own read-write head (above and below the platter)
- tracks are concentric rings on disk platter (bits are laid out serially on tracks)
  - the first track is the outer-most track on each platter
- a cylinder is a set of vertically-aligned tracks on all platters
- each track is divided into a number of sectors
  - a sector/block defines the minimum unit of data transfer
  - a sector has a specific physical address, commonly expressed as a CHS (cylinder-head-sector) tuple
    e.g., cylinder 45, head 6, sector 10

both sides of platters are used (except on the ends)



File system and disk I/O

- hard disk drives support direct and sequential access to data
  - this happens by moving read-write heads and waiting for disk to rotate

File system and disk I/O

- hard disk drives support direct and sequential access to data
  - this happens by moving read-write heads and waiting for disk to rotate
- data transfer between memory and disk is performed in units of blocks (sectors)
  - sector size can be from 32 bytes to 4KB (the usual size is 512 bytes)

File system and disk I/O

- hard disk drives support direct and sequential access to data
  - this happens by moving read-write heads and waiting for disk to rotate
- data transfer between memory and disk is performed in units of blocks (sectors)
  - sector size can be from 32 bytes to 4KB (the usual size is 512 bytes)
- file system knows logical addresses of a file's blocks
  - logical addresses must be translated to physical addresses
  - logical addresses are sequential, physical addresses are not

File system and disk I/O

- hard disk drives support direct and sequential access to data
  - this happens by moving read-write heads and waiting for disk to rotate
- data transfer between memory and disk is performed in units of blocks (sectors)
  - sector size can be from 32 bytes to 4KB (the usual size is 512 bytes)
- file system knows logical addresses of a file's blocks
  - logical addresses must be translated to physical addresses
  - logical addresses are sequential, physical addresses are not
- no head movement is needed when reading/writing sectors of one track (equidistant from the centre)
  - no seek time

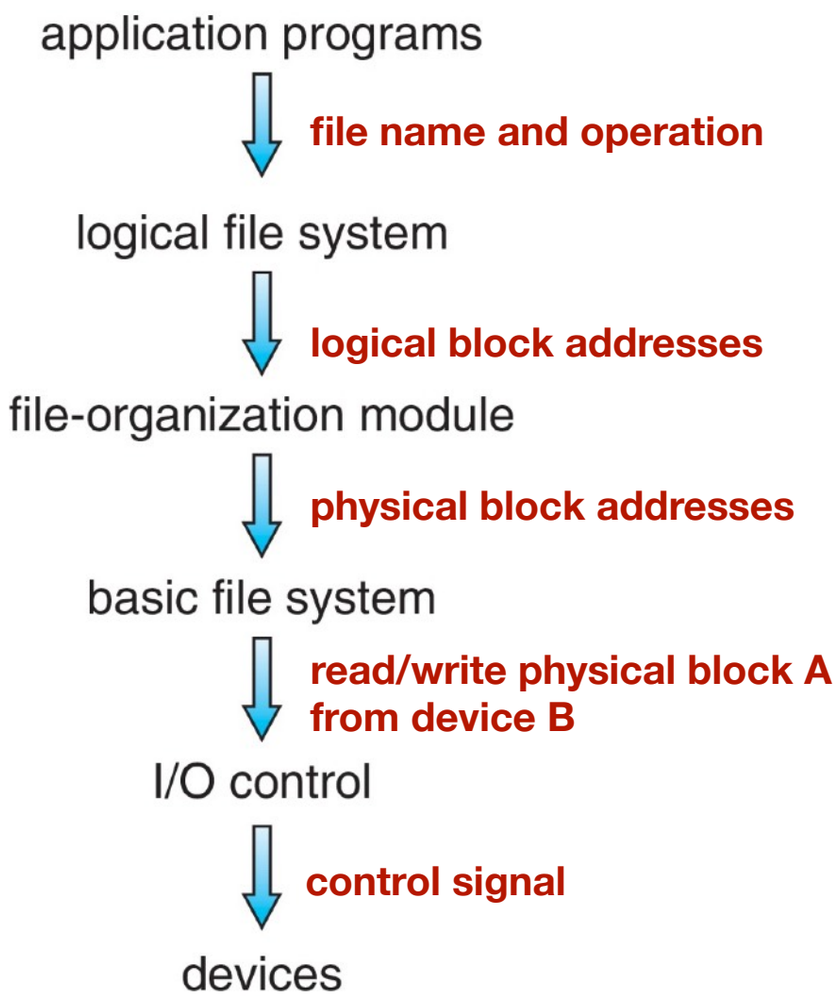
File system and disk I/O

- hard disk drives support direct and sequential access to data
  - this happens by moving read-write heads and waiting for disk to rotate
- data transfer between memory and disk is performed in units of blocks (sectors)
  - sector size can be from 32 bytes to 4KB (the usual size is 512 bytes)
- file system knows logical addresses of a file's blocks
  - logical addresses must be translated to physical addresses
  - logical addresses are sequential, physical addresses are not
- no head movement is needed when reading/writing sectors of one track (equidistant from the centre)
  - no seek time
- a block in the file system memory buffer is allocated before the transfer of a disk block starts

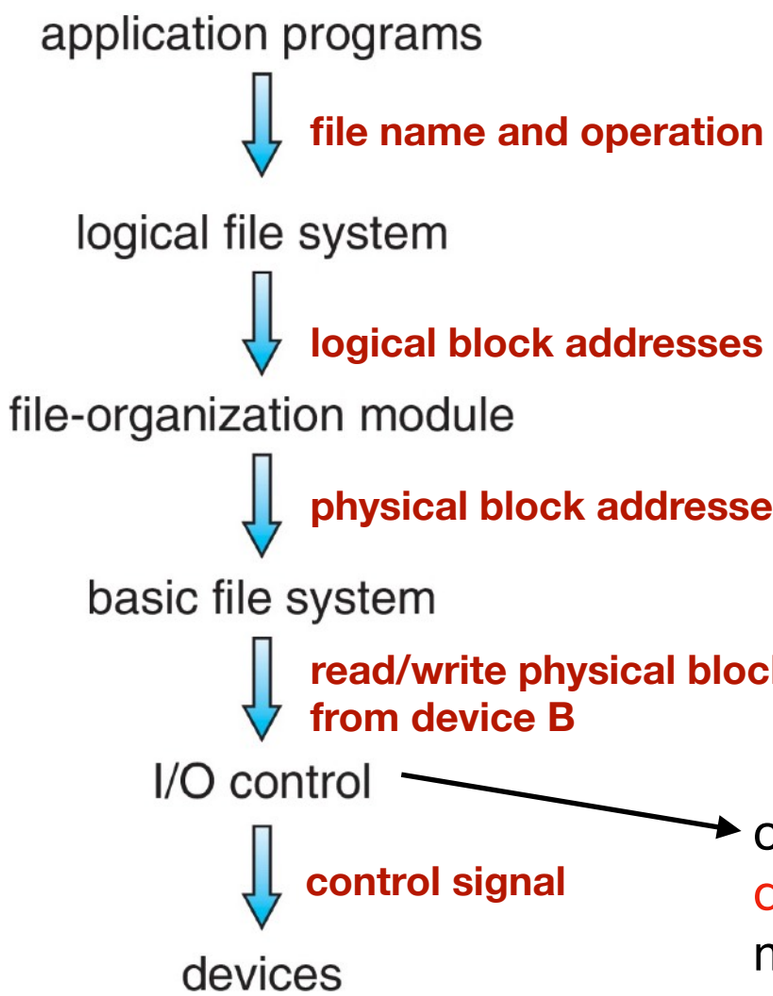
Disk latency and bandwidth

- I/O latency: time to initiate a disk transfer of 1 byte to memory
  - seek time: time to position the head over the correct cylinder
  - rotational latency: the time for the correct sector to rotate under the head
- bandwidth: the rate of I/O transfer once it starts

Layered file system

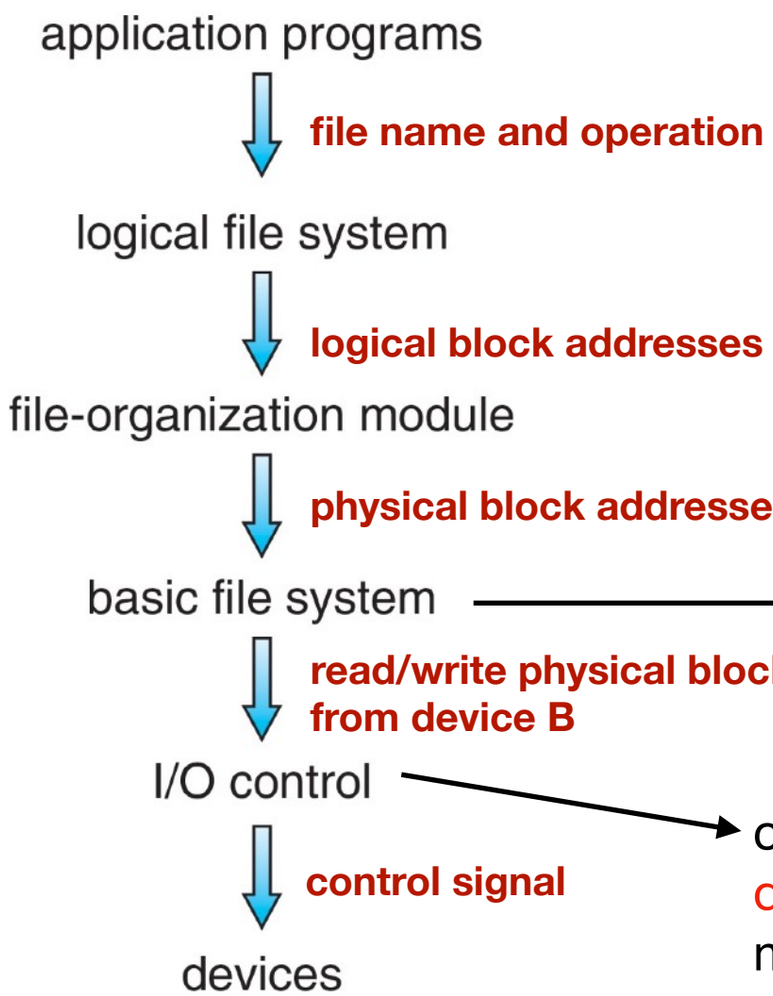


Layered file system



consists of device drivers and interrupt handlers;
device driver writes a specific bit pattern to I/O controller's memory to specify the device location (physical address) and the action that must be performed on

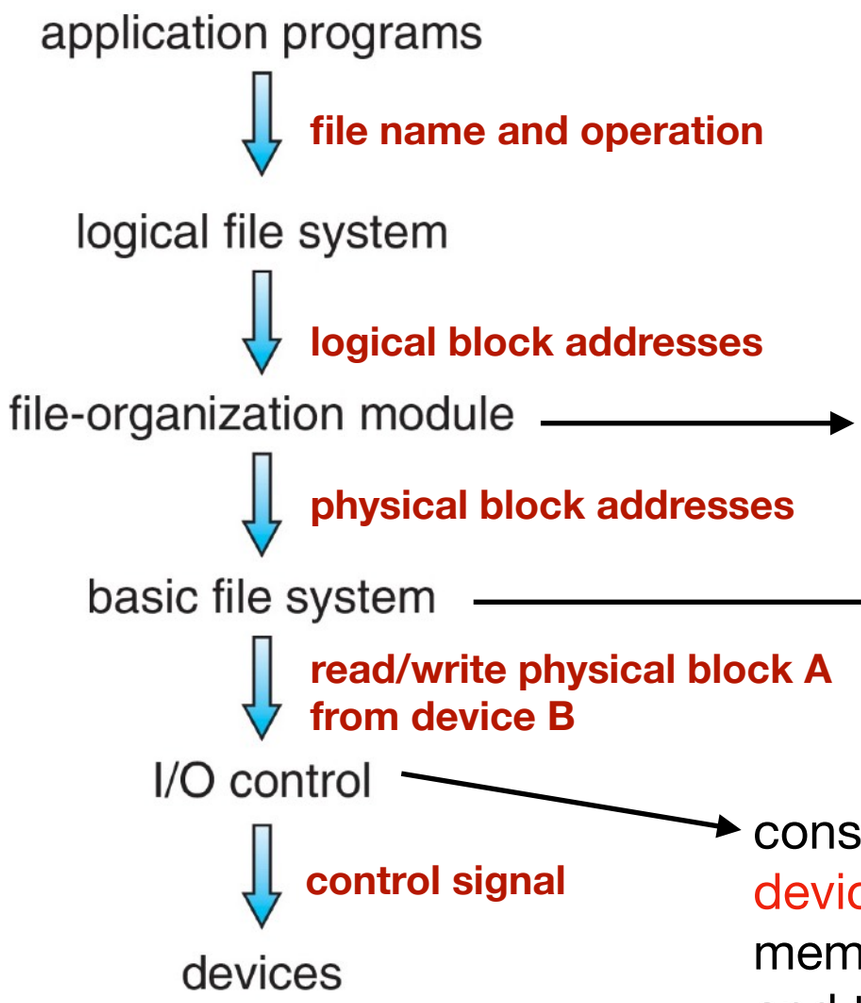
Layered file system



manages memory buffers and caches needed for data transfer between memory and disk

consists of device drivers and interrupt handlers;
device driver writes a specific bit pattern to I/O controller's memory to specify the device location (physical address) and the action that must be performed on

Layered file system

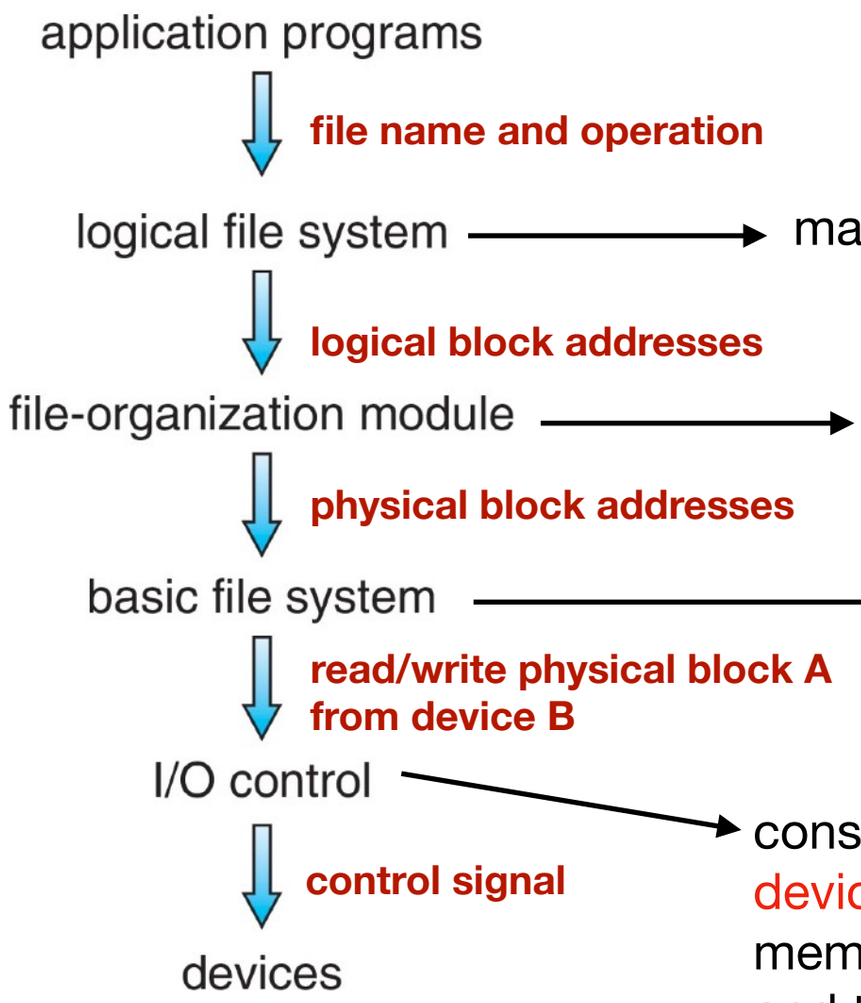


translates logical block address to physical block address provided the file allocation algorithm used, and tracks unallocated blocks (via the free-space list)

manages memory buffers and caches needed for data transfer between memory and disk

consists of device drivers and interrupt handlers;
device driver writes a specific bit pattern to I/O controller's memory to specify the device location (physical address) and the action that must be performed on

Layered file system



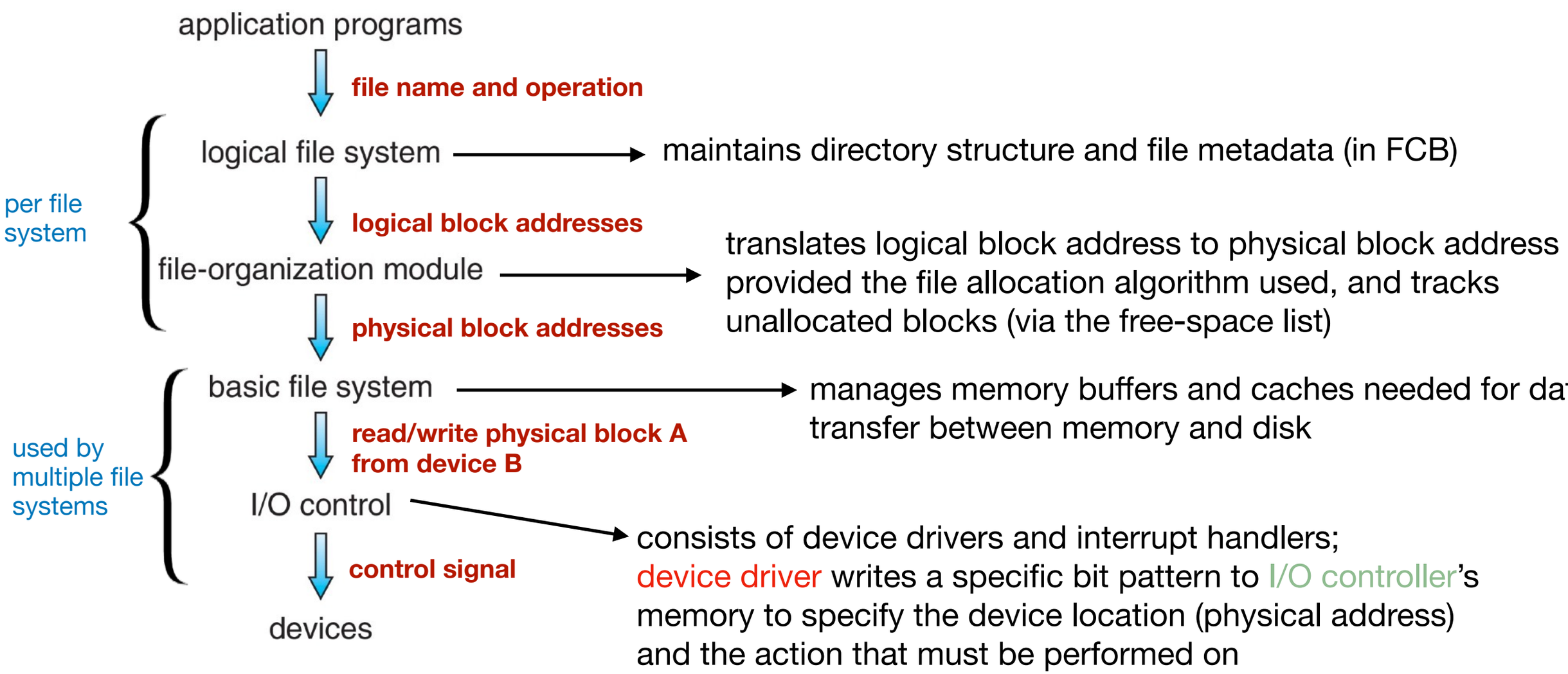
maintains directory structure and file metadata (in FCB)

translates logical block address to physical block address provided the file allocation algorithm used, and tracks unallocated blocks (via the free-space list)

manages memory buffers and caches needed for data transfer between memory and disk

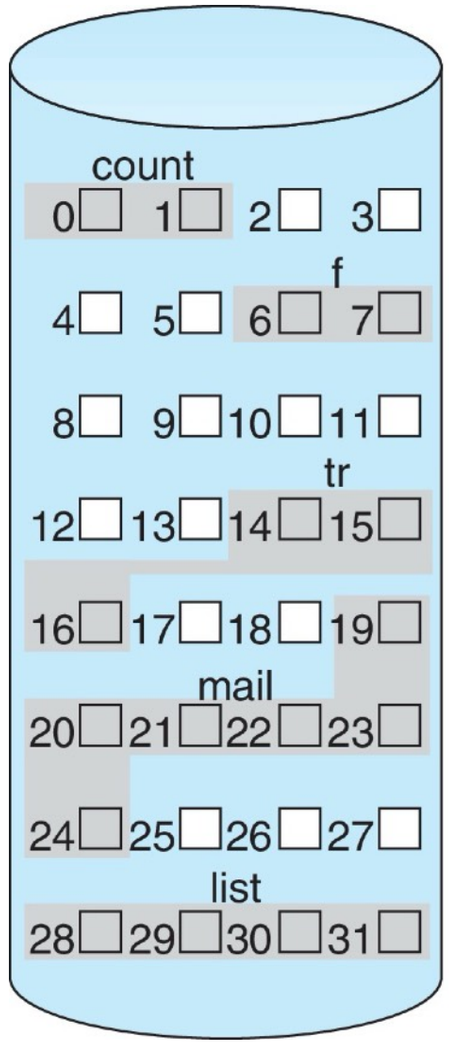
consists of device drivers and interrupt handlers;
device driver writes a specific bit pattern to I/O controller's memory to specify the device location (physical address) and the action that must be performed on

Layered file system



Contiguous allocation

- each file occupies a number of contiguous blocks

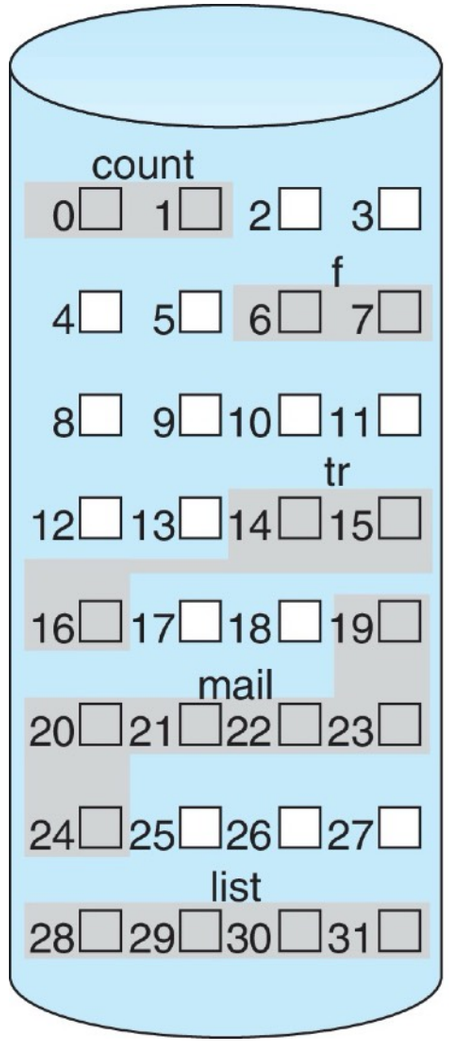


directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Contiguous allocation

- each file occupies a number of contiguous blocks
- advantages:
  - in most cases contiguous blocks are written on sectors of the same cylinder, hence the number of disk seeks required to access data sequentially is minimum
  - direct access is easy because address translation is straightforward
  - simplicity as only starting location (block number) and length must be stored

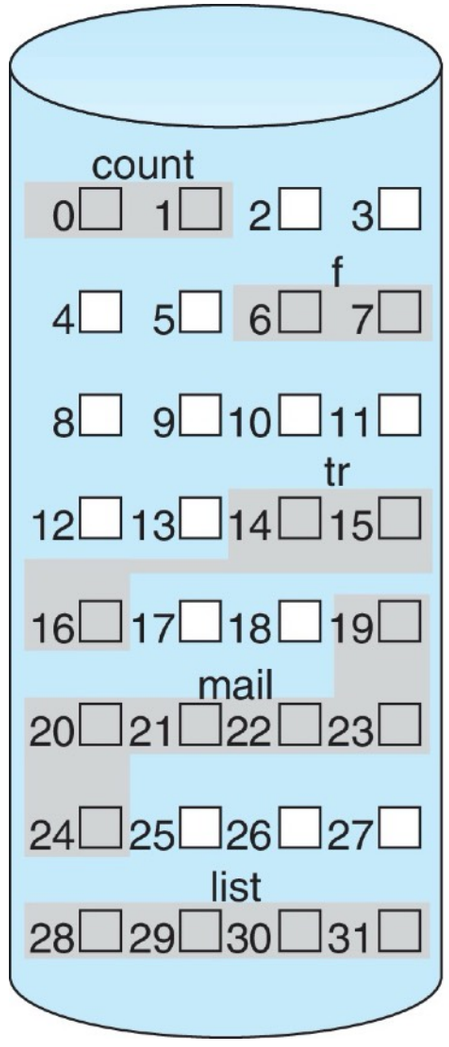


directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Contiguous allocation

- each file occupies a number of contiguous blocks
- advantages:
  - in most cases contiguous blocks are written on sectors of the same cylinder, hence the number of disk seeks required to access data sequentially is minimum
  - direct access is easy because address translation is straightforward
  - simplicity as only starting location (block number) and length must be stored
- disadvantages:
  - hard to find space for files and to determine how much space they will eventually need
  - incremental growth of file introduces complexity
  - external fragmentation; compaction/defragmentation is needed from time to time

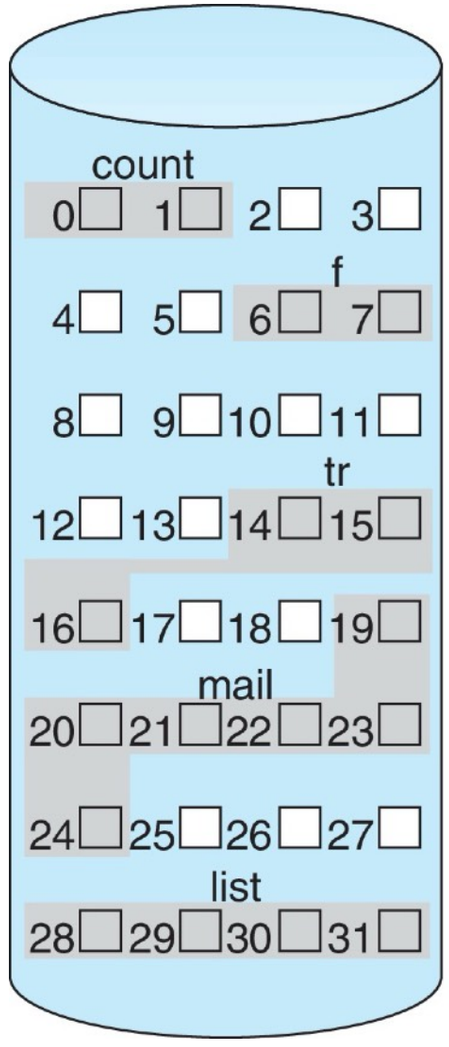


directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Contiguous allocation

- each file occupies a number of contiguous blocks
- advantages:
  - in most cases contiguous blocks are written on sectors of the same cylinder, hence the number of disk seeks required to access data sequentially is minimum
  - direct access is easy because address translation is straightforward
  - simplicity as only starting location (block number) and length must be stored
- disadvantages:
  - hard to find space for files and to determine how much space they will eventually need
  - incremental growth of file introduces complexity
  - external fragmentation; compaction/defragmentation is needed from time to time
- ideal for write-once read-many (WORM) devices, like CD-R and DVD-R (**why?**)

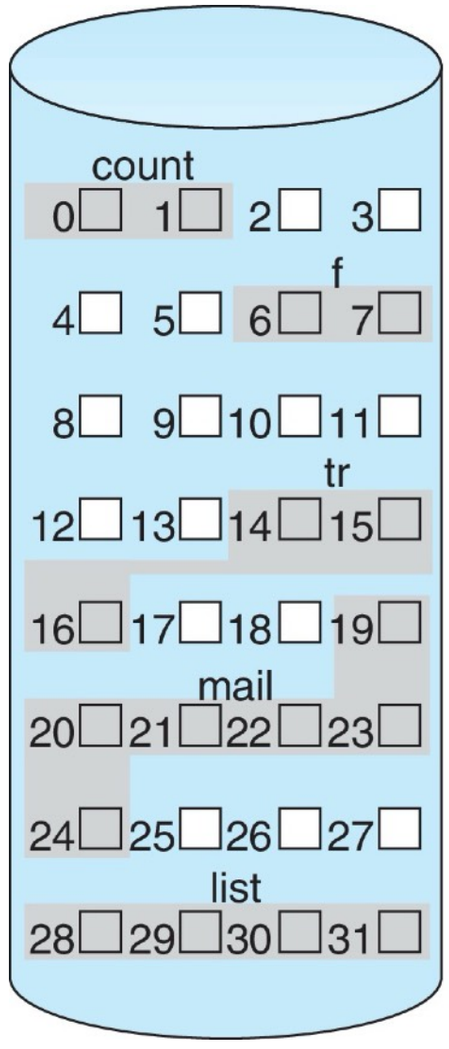


directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Contiguous allocation

- address translation
  (**file address x → logical disk block j, offset x**)
  - divide the logical file address by block size; let the quotient be `i` and the remainder be `r`
  - request for logical file block `i` is mapped to physical block `s + i` where `s` is the first block allocated to this file
  - `r` is the offset in the physical block `s + i`

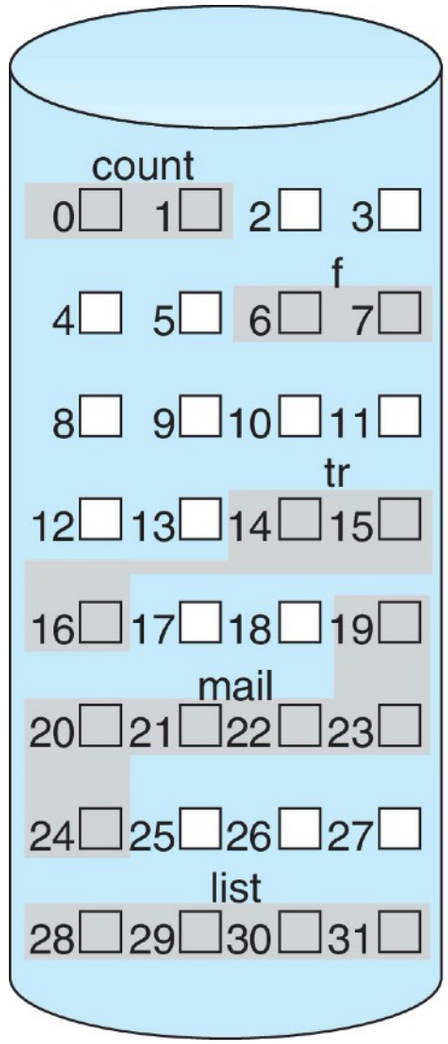


directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

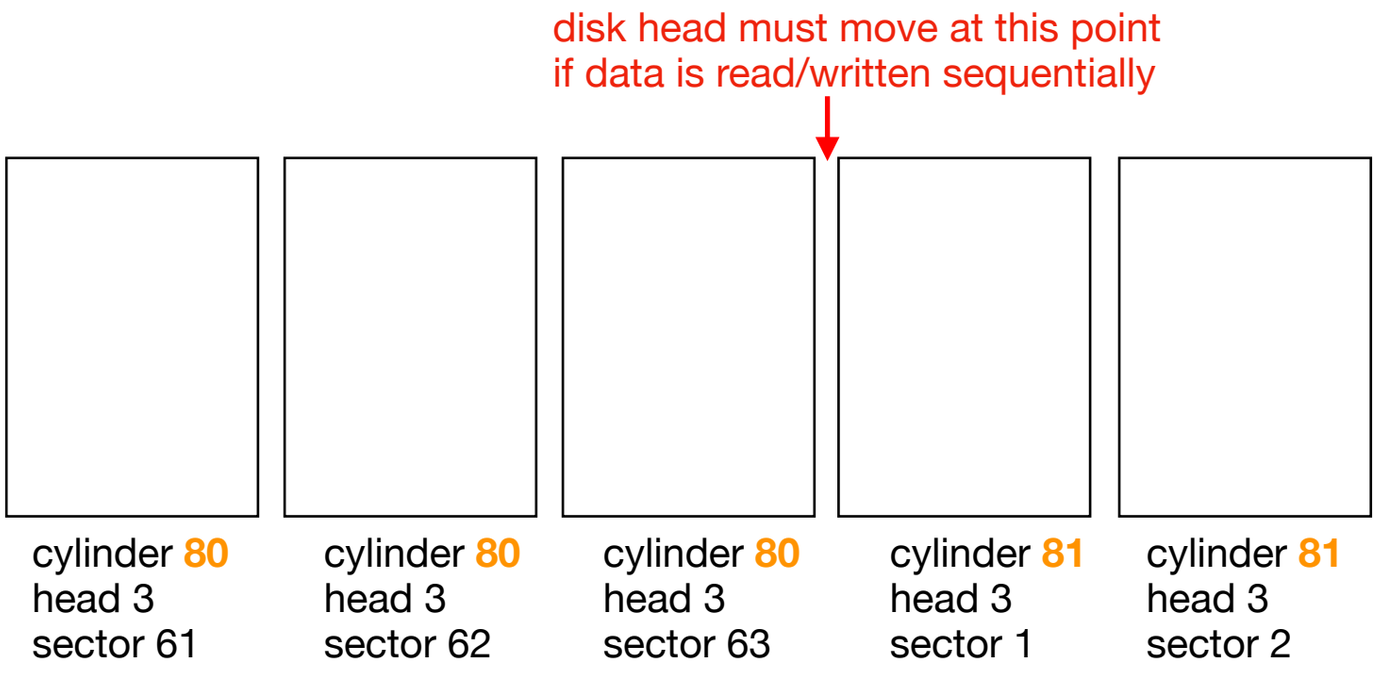
Contiguous allocation

- address translation
  (**file address x → logical disk block j, offset x**)
  - divide the logical file address by block size; let the quotient be `i` and the remainder be `r`
  - request for logical file block `i` is mapped to physical block `s + i` where `s` is the first block allocated to this file
  - `r` is the offset in the physical block `s + i`



directory

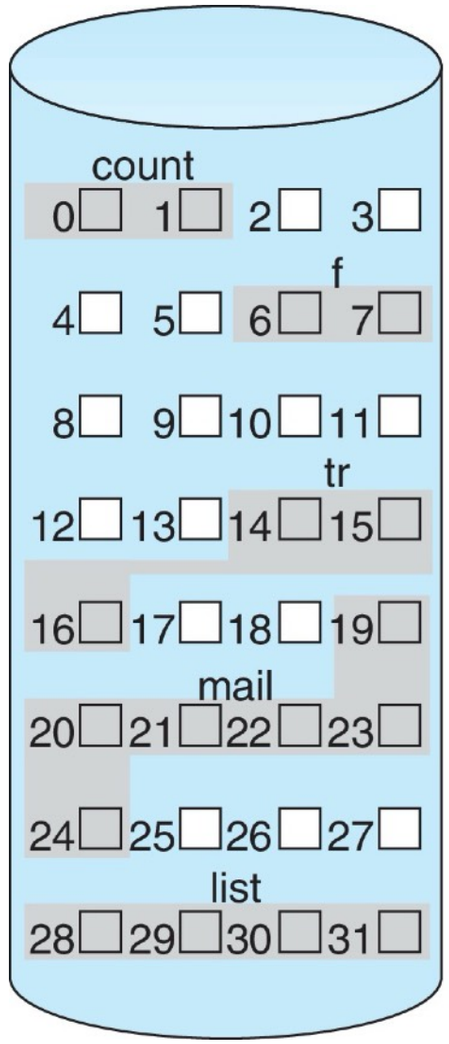
file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2



suppose each track is divided to 63 sectors and sector number starts at 1

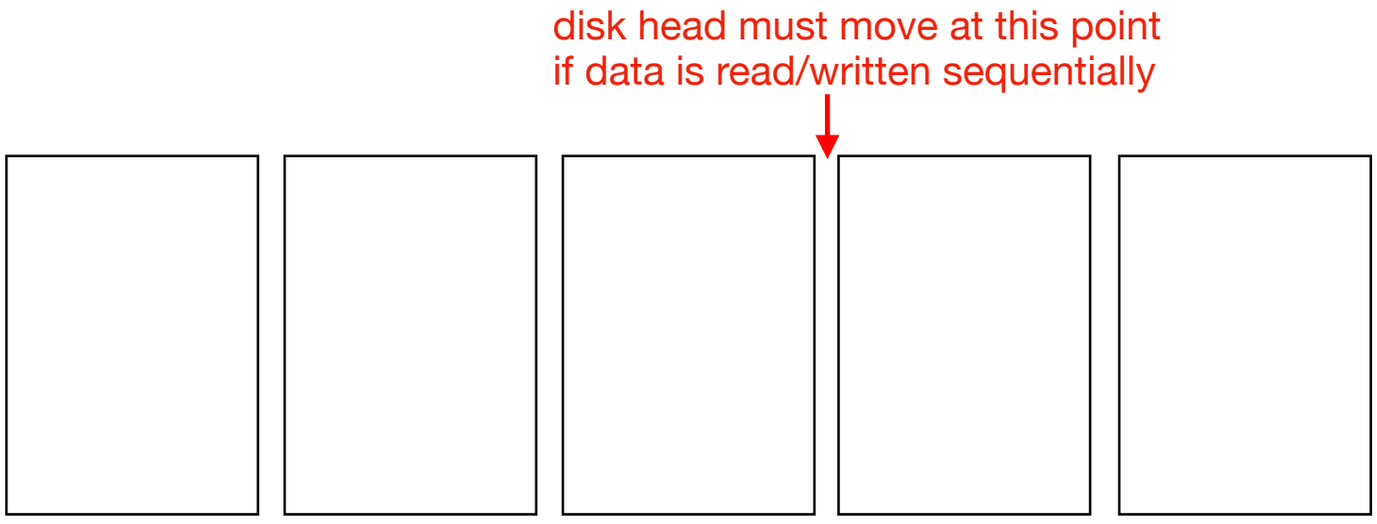
Contiguous allocation

- address translation
  (**file address x → logical disk block j, offset x**)
  - divide the logical file address by block size; let the quotient be `i` and the remainder be `r`
  - request for logical file block `i` is mapped to physical block `s + i` where `s` is the first block allocated to this file
  - `r` is the offset in the physical block `s + i`



directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2



suppose each track is divided to 63 sectors and sector number starts at 1

How to do this translation?

logical disk block n → cylinder i, head j, sector k

Modified contiguous allocation

- **basic idea**: allocate a contiguous chunk of space initially (known as an **extent**); if more space is needed allocate a new extent
  - so a file consists of multiple extents, each being a contiguous chunk of space

Modified contiguous allocation

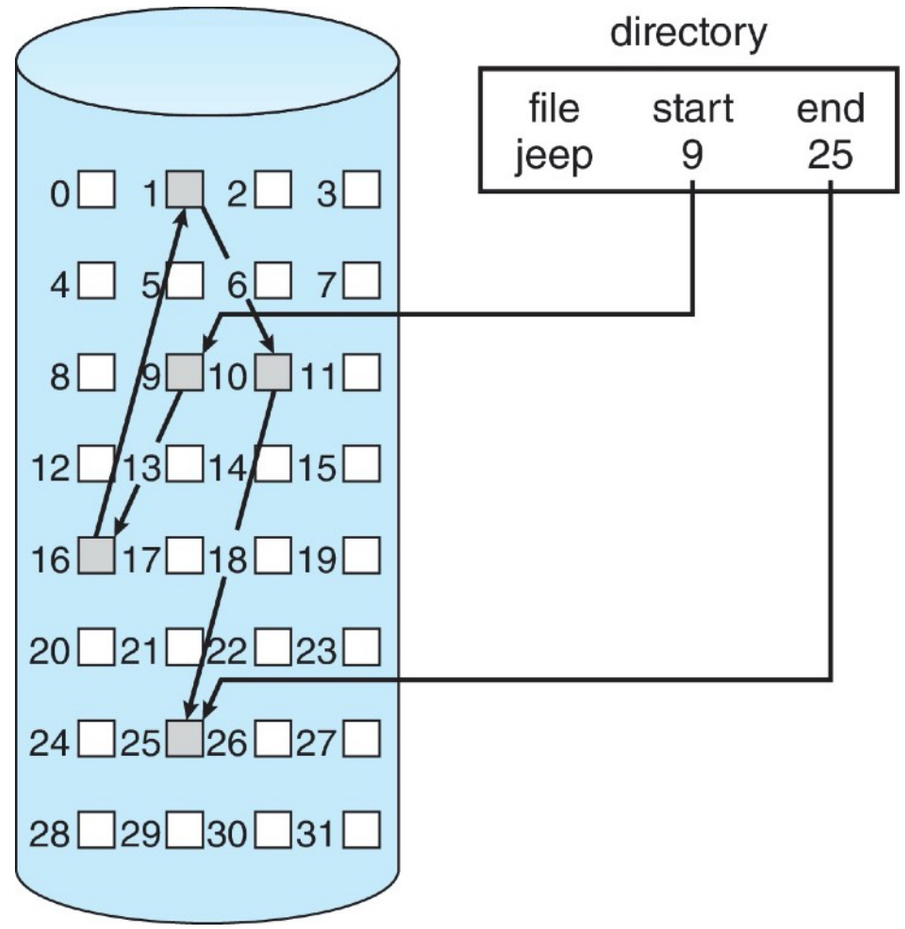
- **basic idea**: allocate a contiguous chunk of space initially (known as an **extent**); if more space is needed allocate a new extent
  - so a file consists of multiple extents, each being a contiguous chunk of space
- keep a link to the first block of the next extent allocated to this file in addition to recording the start block and length of each extent

Modified contiguous allocation

- **basic idea**: allocate a contiguous chunk of space initially (known as an **extent**); if more space is needed allocate a new extent
  - so a file consists of multiple extents, each being a contiguous chunk of space
- keep a link to the first block of the next extent allocated to this file in addition to recording the start block and length of each extent
- how to set the extent size?
  - large extent sizes cause internal fragmentation
  - using variable-size extents contributes to external fragmentation

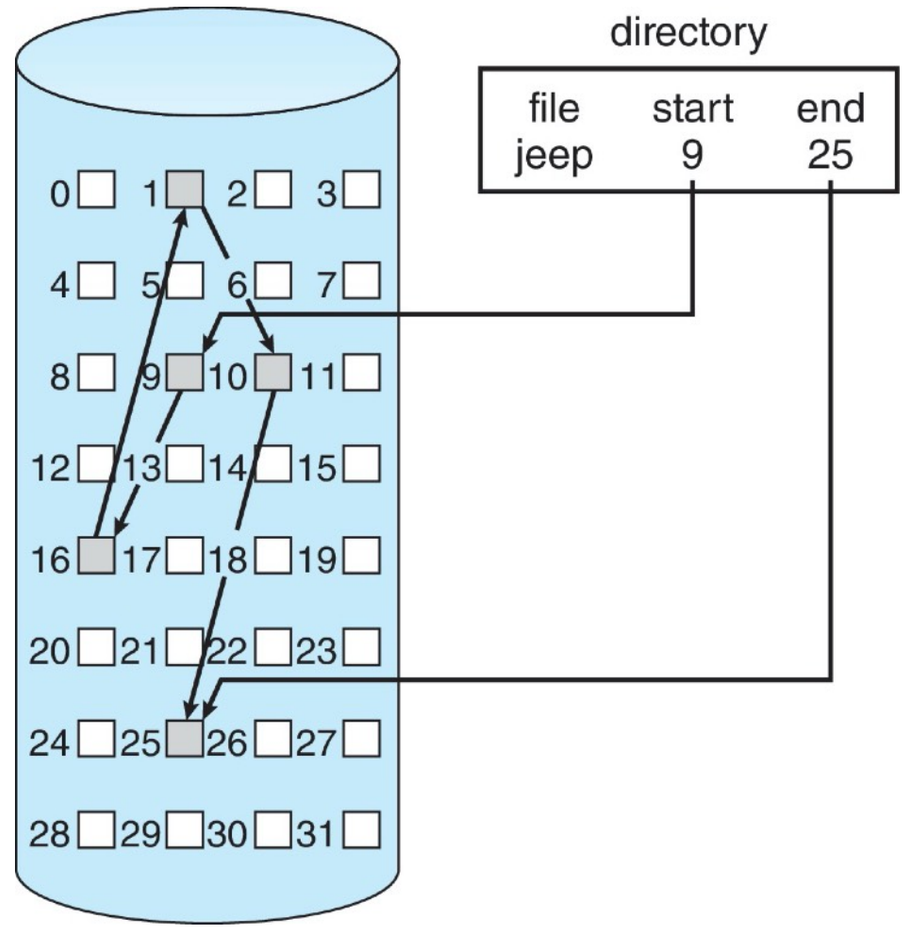
Linked allocation

- each file a linked list of blocks scattered on disk
  - each block contains pointer to next block;
    last block contains a null pointer



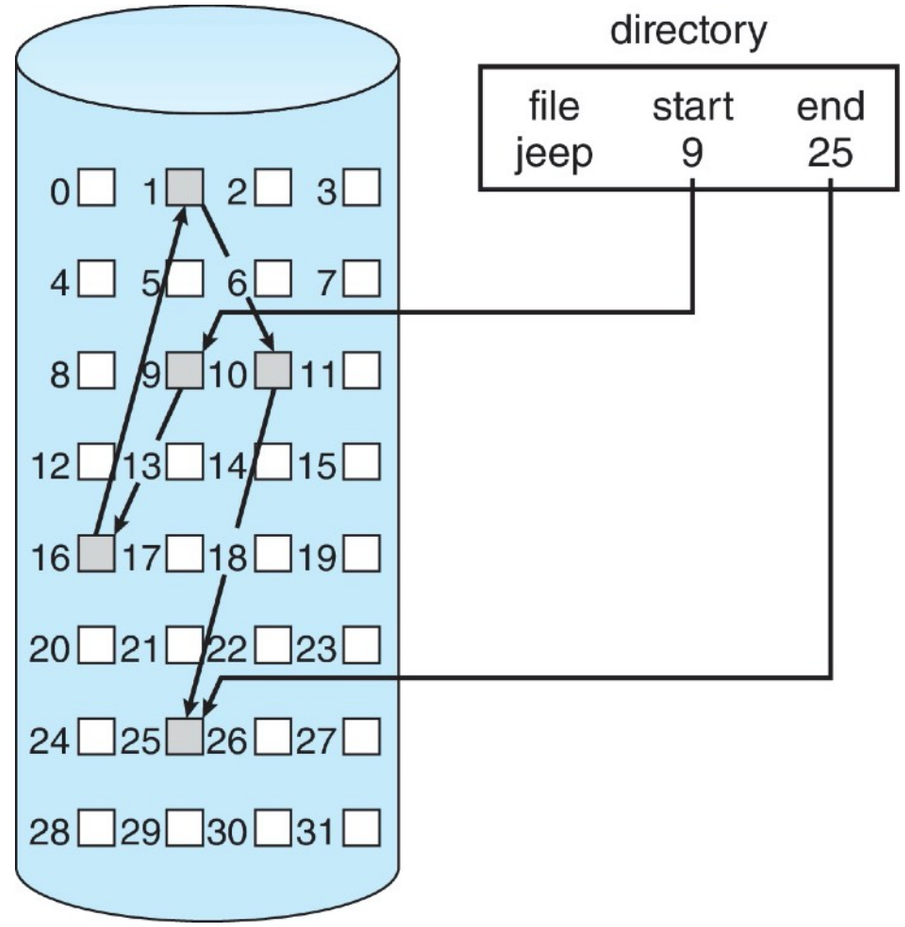
Linked allocation

- each file a linked list of blocks scattered on disk
  - each block contains pointer to next block;
    last block contains a null pointer
- advantages:
  - no external fragmentation; no need for compacting disk



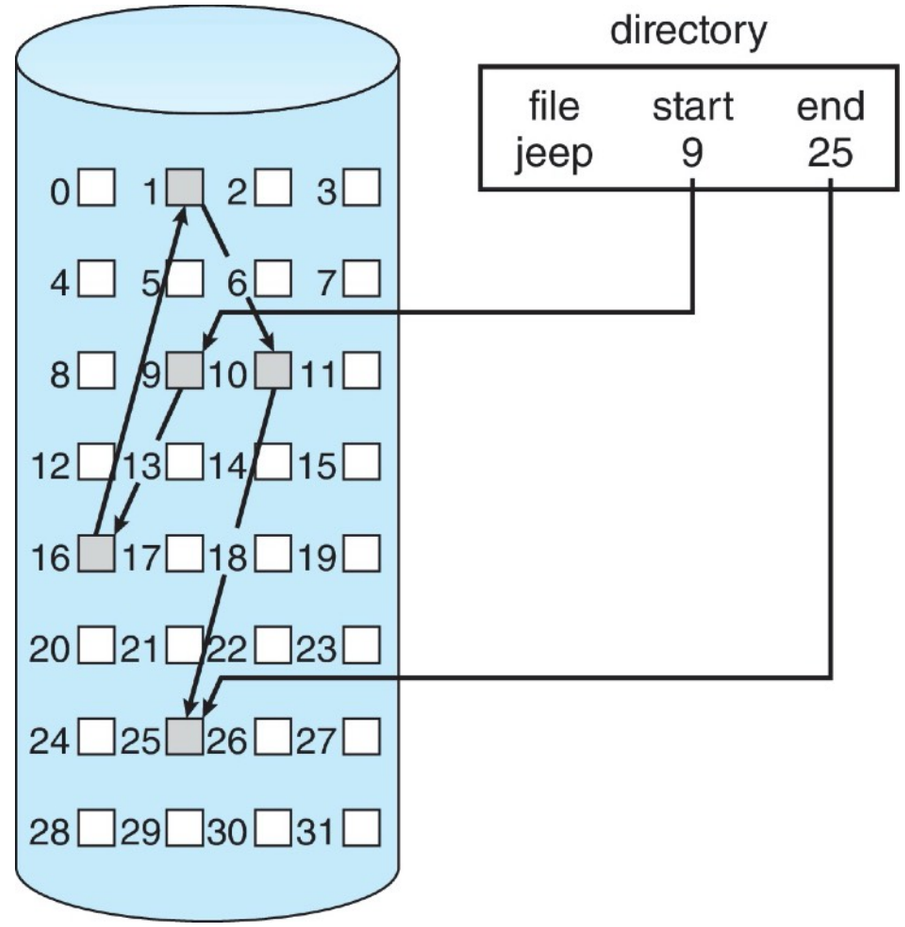
Linked allocation

- each file a linked list of blocks scattered on disk
  - each block contains pointer to next block;
    last block contains a null pointer
- advantages:
  - no external fragmentation; no need for compacting disk
- disadvantages:
  - random access is inefficient (locating a block may require many disk reads and seeks)



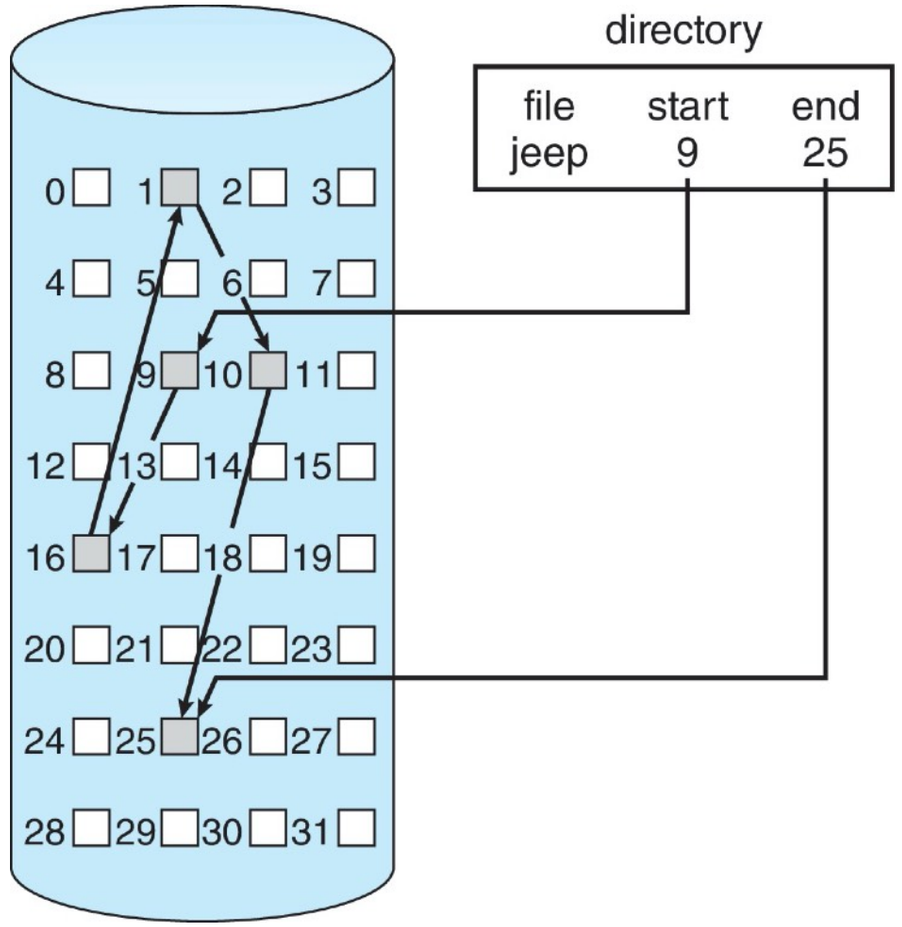
Linked allocation

- each file a linked list of blocks scattered on disk
  - each block contains pointer to next block;
    last block contains a null pointer
- advantages:
  - no external fragmentation; no need for compacting disk
- disadvantages:
  - random access is inefficient (locating a block may require many disk reads and seeks)
  - if each block is 512 bytes and a pointer to the next block needs 4 bytes (assuming 32-bit address space), only 508 bytes from each block are available to user; so 0.78% of disk space is wasted



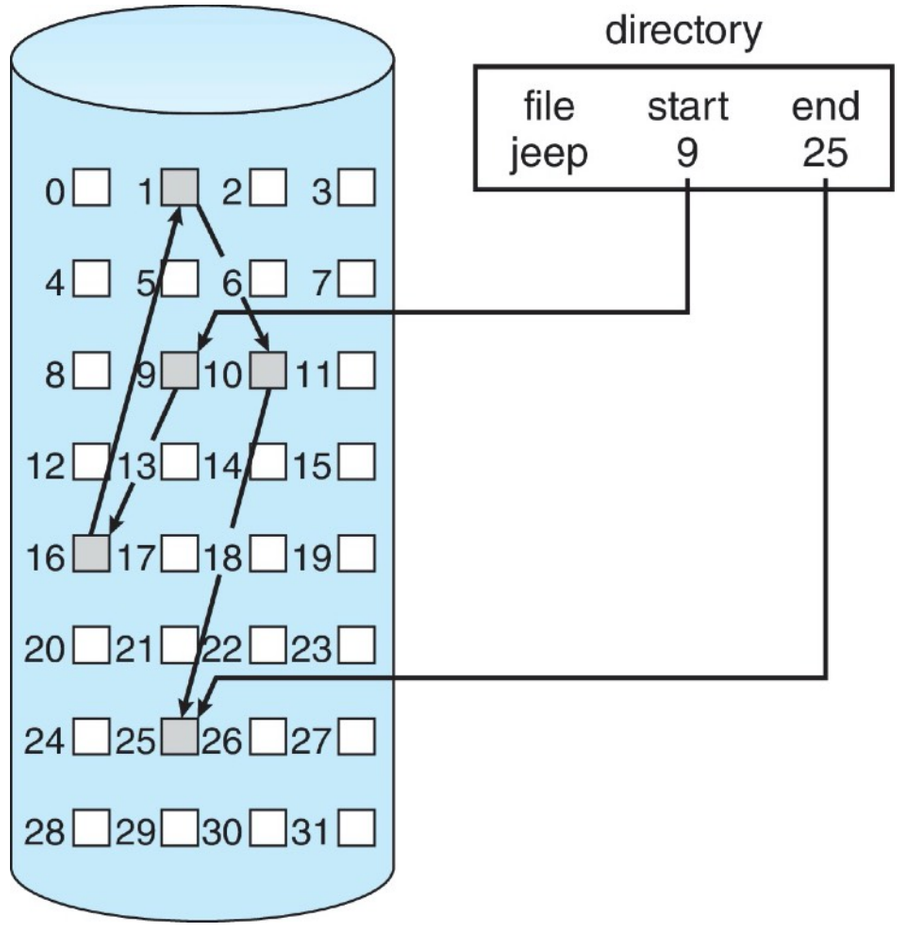
Linked allocation

- each file a linked list of blocks scattered on disk
  - each block contains pointer to next block;
    last block contains a null pointer
- advantages:
  - no external fragmentation; no need for compacting disk
- disadvantages:
  - random access is inefficient (locating a block may require many disk reads and seeks)
  - if each block is 512 bytes and a pointer to the next block needs 4 bytes (assuming 32-bit address space), only 508 bytes from each block are available to user; so 0.78% of disk space is wasted
    - **can improve efficiency by collecting multiple blocks into clusters and using a pointer per cluster at the cost of increasing internal fragmentation**



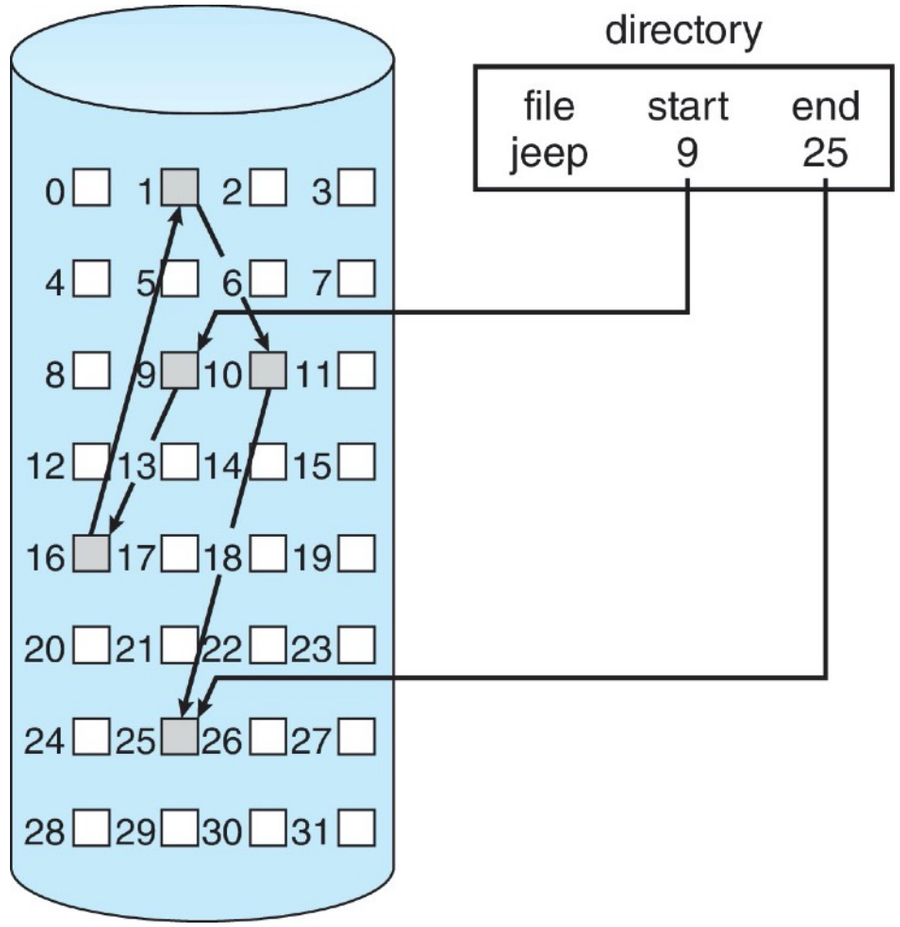
Linked allocation

- each file a linked list of blocks scattered on disk
  - each block contains pointer to next block;
    last block contains a null pointer
- advantages:
  - no external fragmentation; no need for compacting disk
- disadvantages:
  - random access is inefficient (locating a block may require many disk reads and seeks)
  - if each block is 512 bytes and a pointer to the next block needs 4 bytes (assuming 32-bit address space), only 508 bytes from each block are available to user; so 0.78% of disk space is wasted
    - **can improve efficiency by collecting multiple blocks into clusters and using a pointer per cluster at the cost of increasing internal fragmentation**
  - reliability: what if a next-block pointer is damaged?



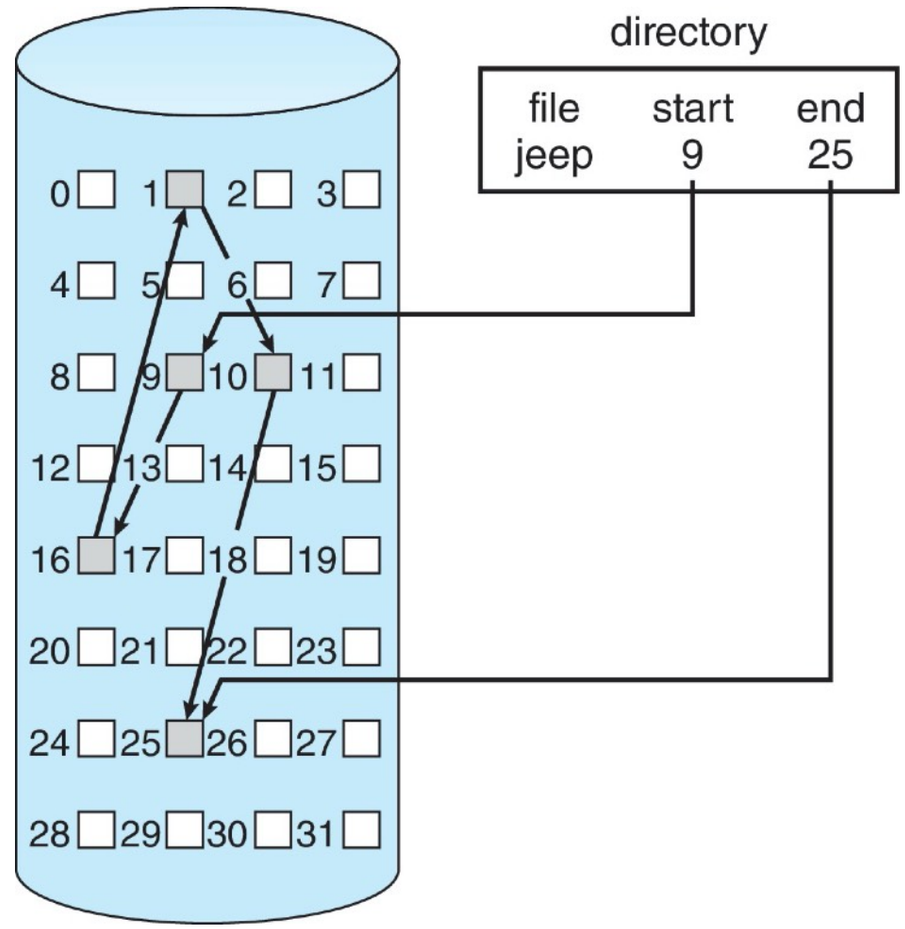
Linked allocation

- each file a linked list of blocks scattered on disk
  - each block contains pointer to next block;
    last block contains a null pointer
- advantages:
  - no external fragmentation; no need for compacting disk
- disadvantages:
  - random access is inefficient (locating a block may require many disk reads and seeks)
  - if each block is 512 bytes and a pointer to the next block needs 4 bytes (assuming 32-bit address space), only 508 bytes from each block are available to user; so 0.78% of disk space is wasted
    - **can improve efficiency by collecting multiple blocks into clusters and using a pointer per cluster at the cost of increasing internal fragmentation**
  - reliability: what if a next-block pointer is damaged?
    - **can maintain a doubly linked list but this increases the wasted space**



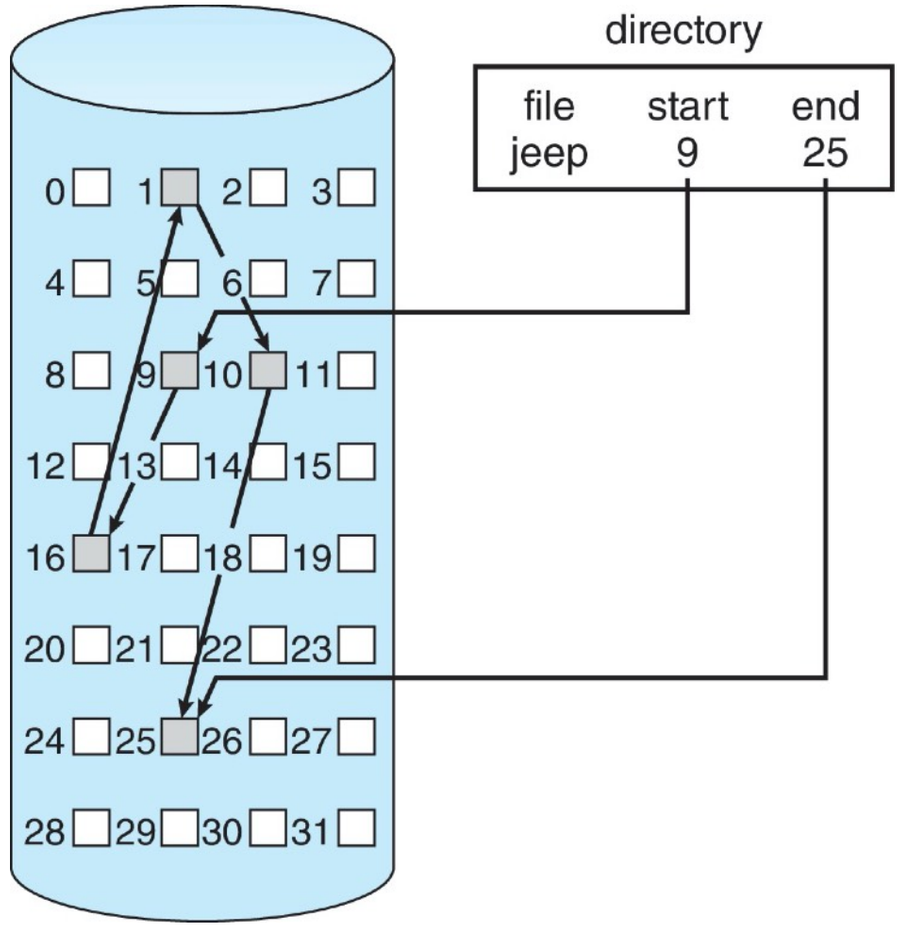
Linked allocation

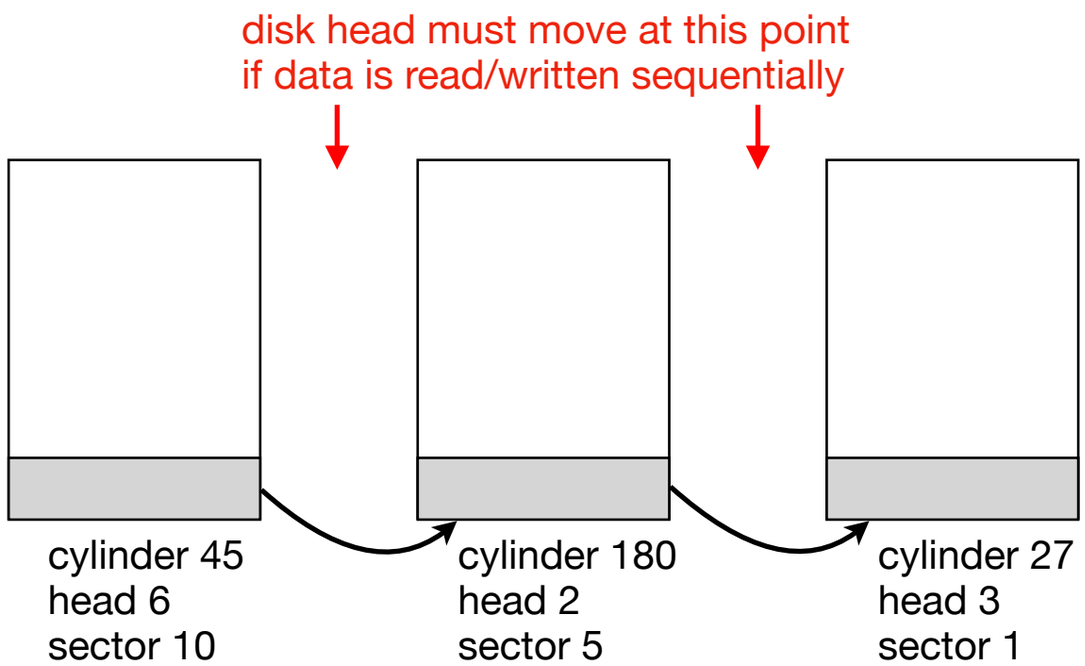
- address translation
  - divide the logical file address by block size - p (where p is the size of the next-block pointer); let the quotient be `i` and the remainder be `r`
  - request for logical file block `i` is translated to the `i`th physical block in the linked chain of blocks representing the file
  - `r+1` is the offset in this physical block



Linked allocation

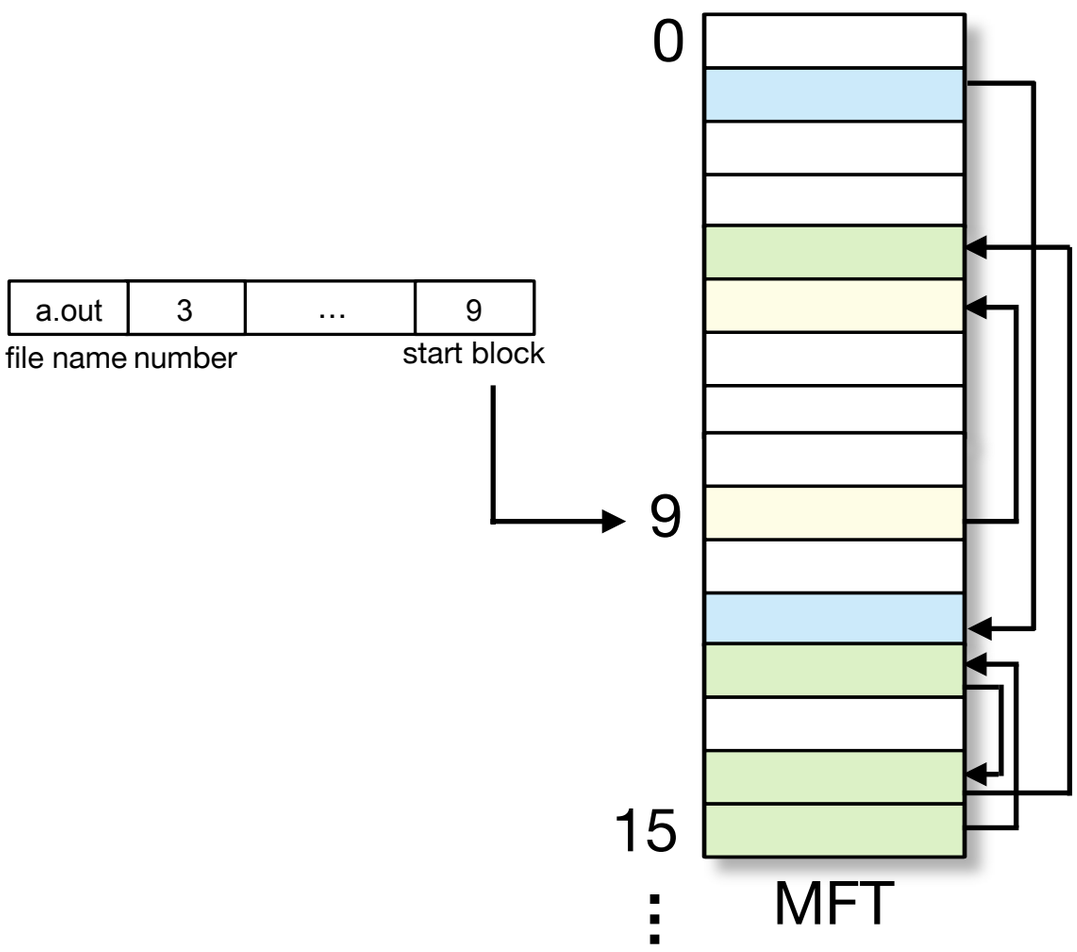
- address translation
  - divide the logical file address by block size - p (where p is the size of the next-block pointer); let the quotient be `i` and the remainder be `r`
  - request for logical file block `i` is translated to the `i`th physical block in the linked chain of blocks representing the file
  - `r+1` is the offset in this physical block





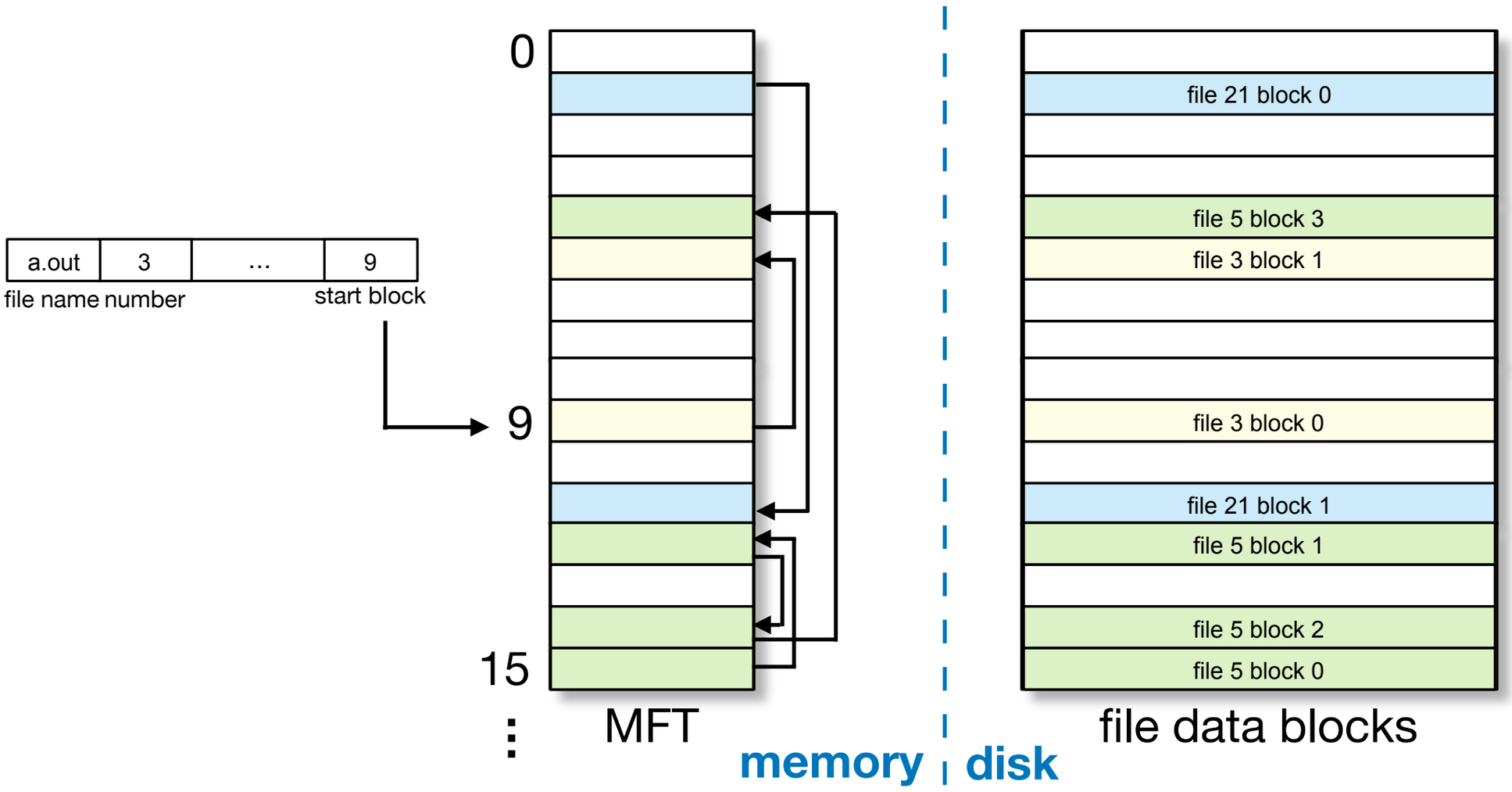
Variation on linked allocation

- File Allocation Table (FAT) was used in MS-DOS
  - master file table (MFT) contains an entry for each data block; this entry holds the pointer to the entry containing the next data block of the file



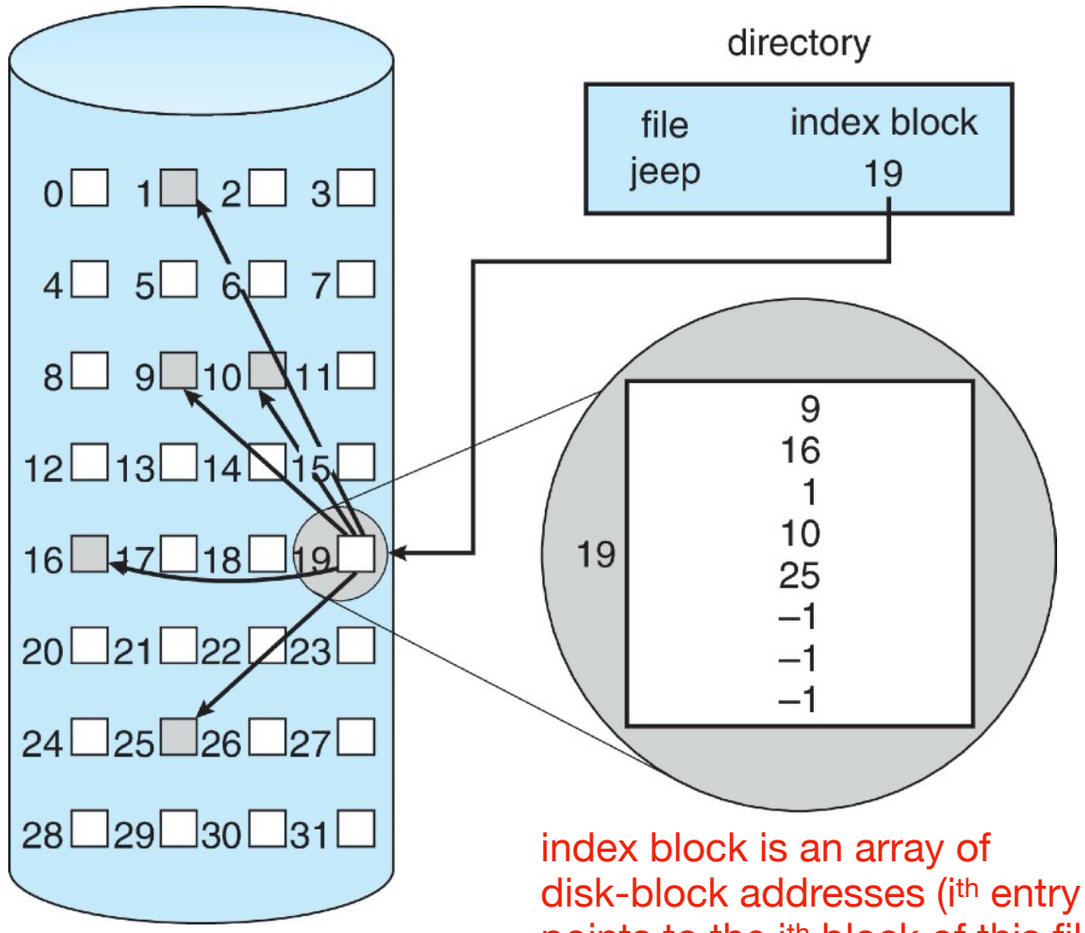
Variation on linked allocation

- File Allocation Table (FAT) was used in MS-DOS
  - master file table (MFT) contains an entry for each data block; this entry holds the pointer to the entry containing the next data block of the file
  - MFT is cached in memory so it can accessed quickly



Indexed allocation

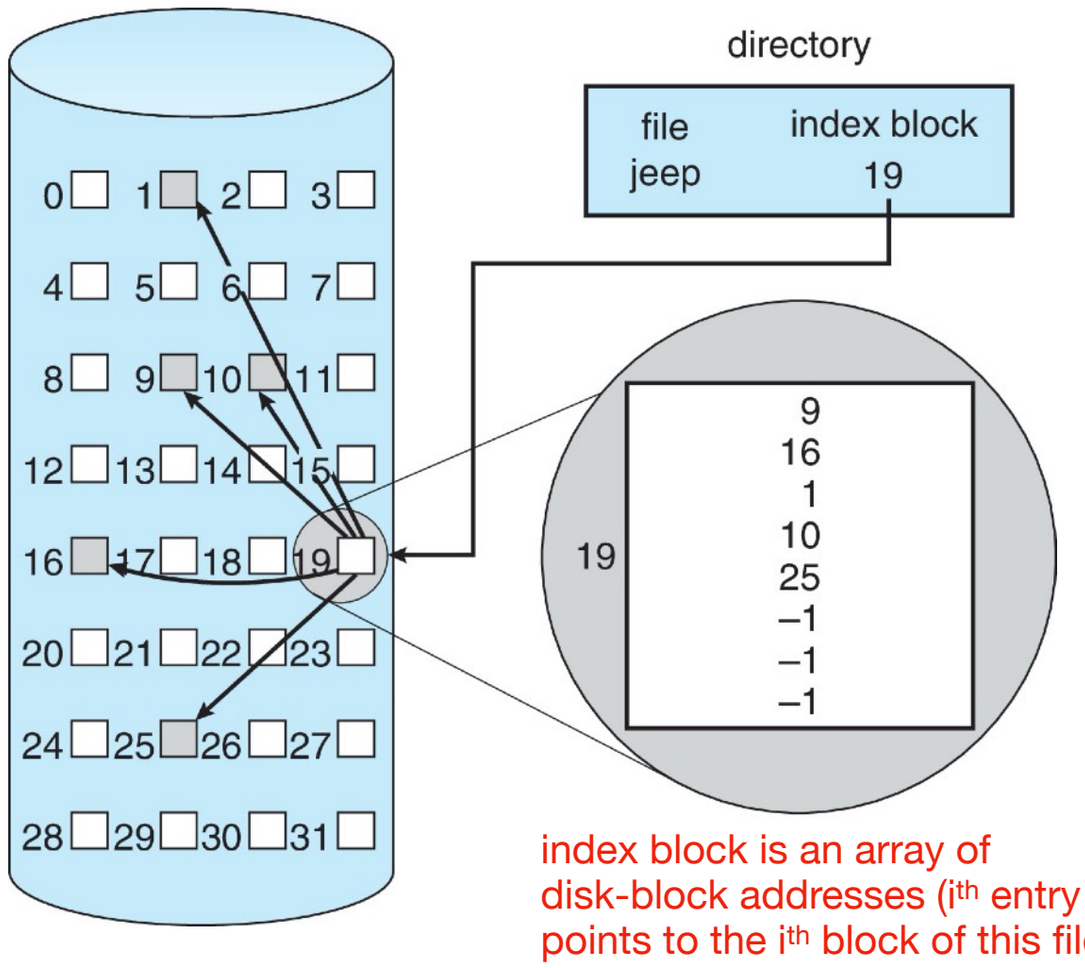
- let's bring all block pointers together into one location (i.e., the index block) for each file
  - the index block can be brought in memory for faster access



index block is an array of disk-block addresses (i^{th} entry points to the i^{th} block of this file)

Indexed allocation

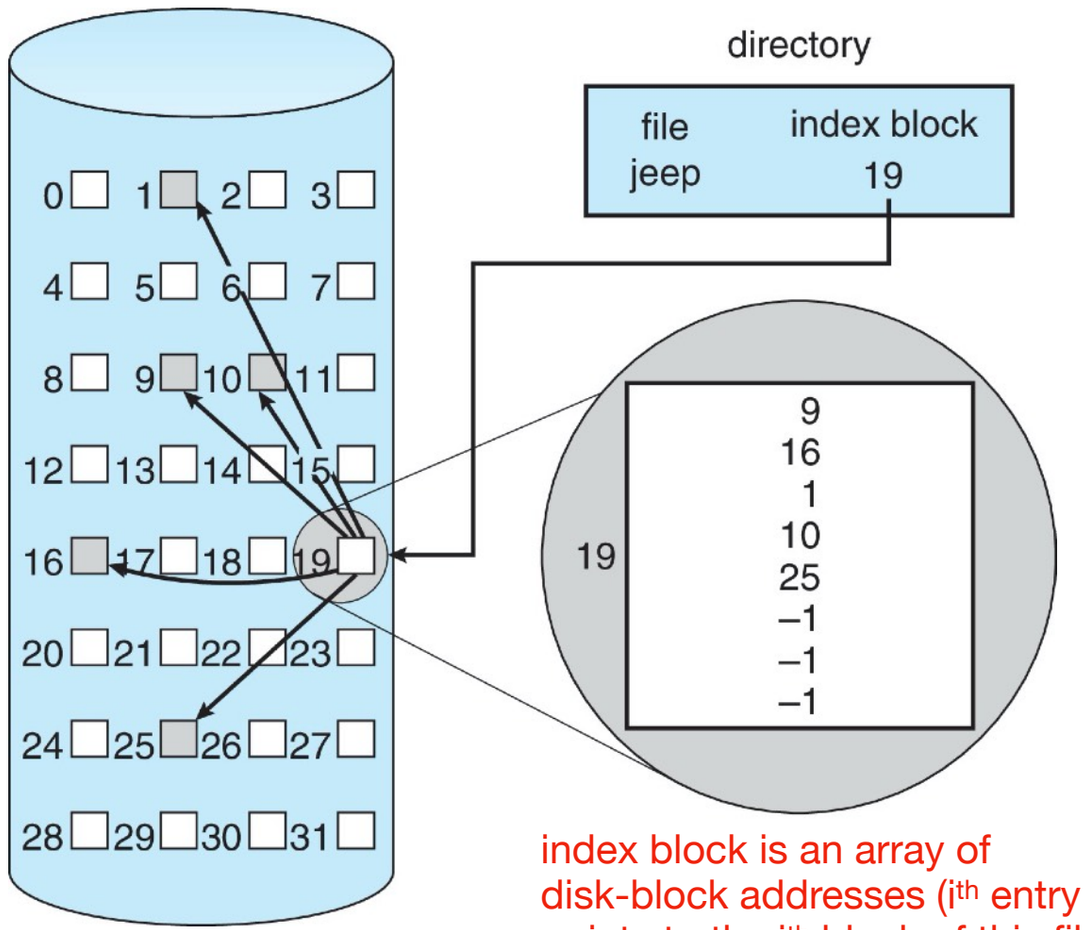
- let's bring all block pointers together into one location (i.e., the index block) for each file
  - the index block can be brought in memory for faster access
- advantages:
  - both sequential and random access are efficient
  - no external fragmentation; no need for compacting disk



index block is an array of disk-block addresses (i^{th} entry points to the i^{th} block of this file)

Indexed allocation

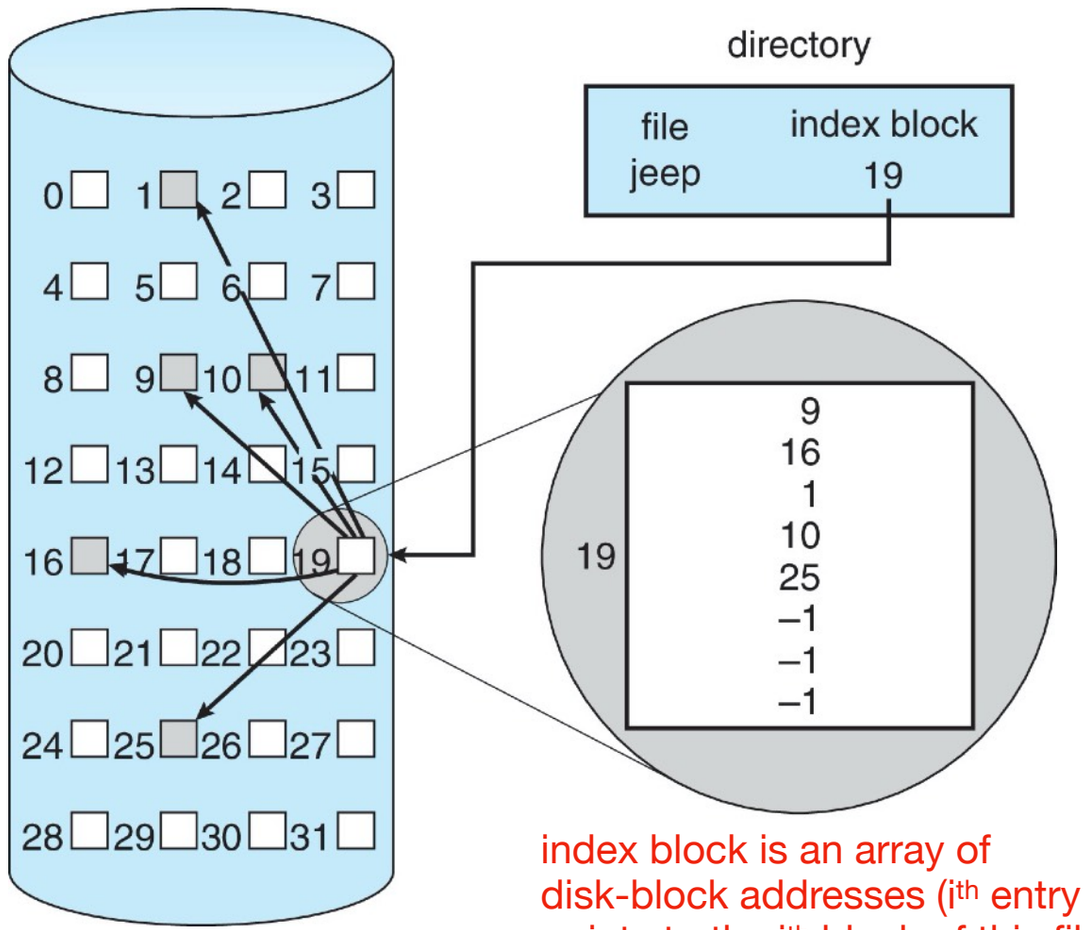
- let's bring all block pointers together into one location (i.e., the index block) for each file
  - the index block can be brought in memory for faster access
- advantages:
  - both sequential and random access are efficient
  - no external fragmentation; no need for compacting disk
- disadvantages:
  - for small files, the pointer overhead is greater than linked allocation
  - the maximum file size is fixed



index block is an array of disk-block addresses (i^{th} entry points to the i^{th} block of this file)

Indexed allocation

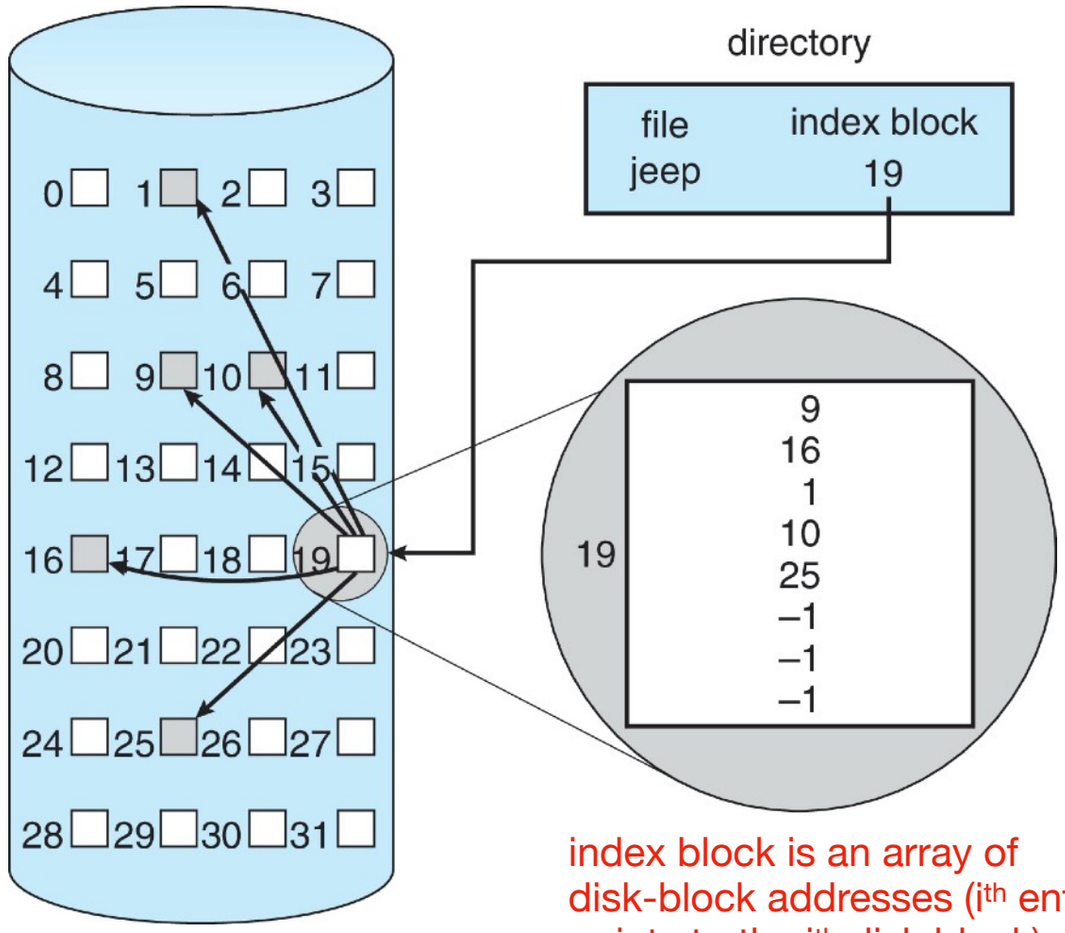
- let's bring all block pointers together into one location (i.e., the index block) for each file
  - the index block can be brought in memory for faster access
- advantages:
  - both sequential and random access are efficient
  - no external fragmentation; no need for compacting disk
- disadvantages:
  - for small files, the pointer overhead is greater than linked allocation
  - the maximum file size is fixed
- how to relax the constraint on the maximum file size?
  - index blocks can be chained if needed, i.e., a pointer to the next index block is stored in each index block
  - index block can be indexed itself



index block is an array of disk-block addresses (i^{th} entry points to the i^{th} block of this file)

Indexed allocation

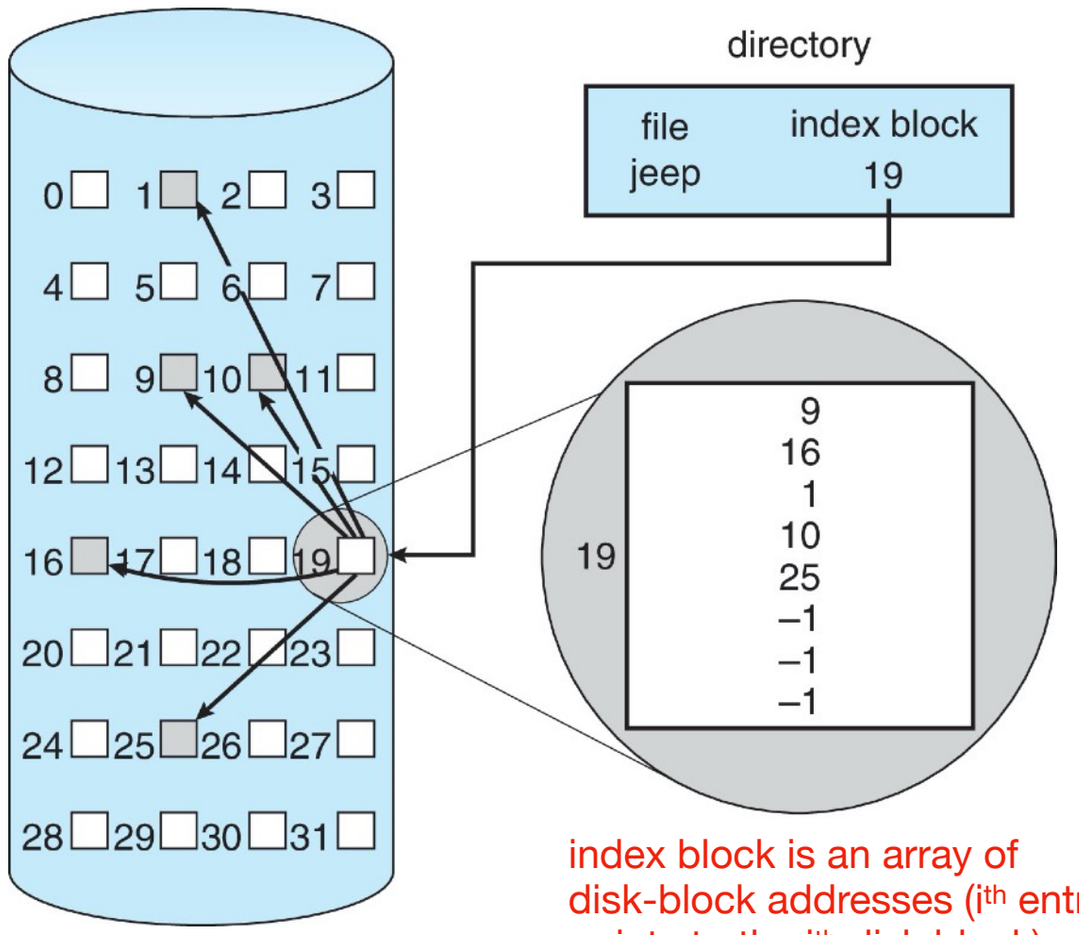
- address translation
  - depends on how many blocks are needed to store the index table
  - divide the logical address by (block size)x(block size - p) to find the index block that contains the index of the block
  - then create the physical address

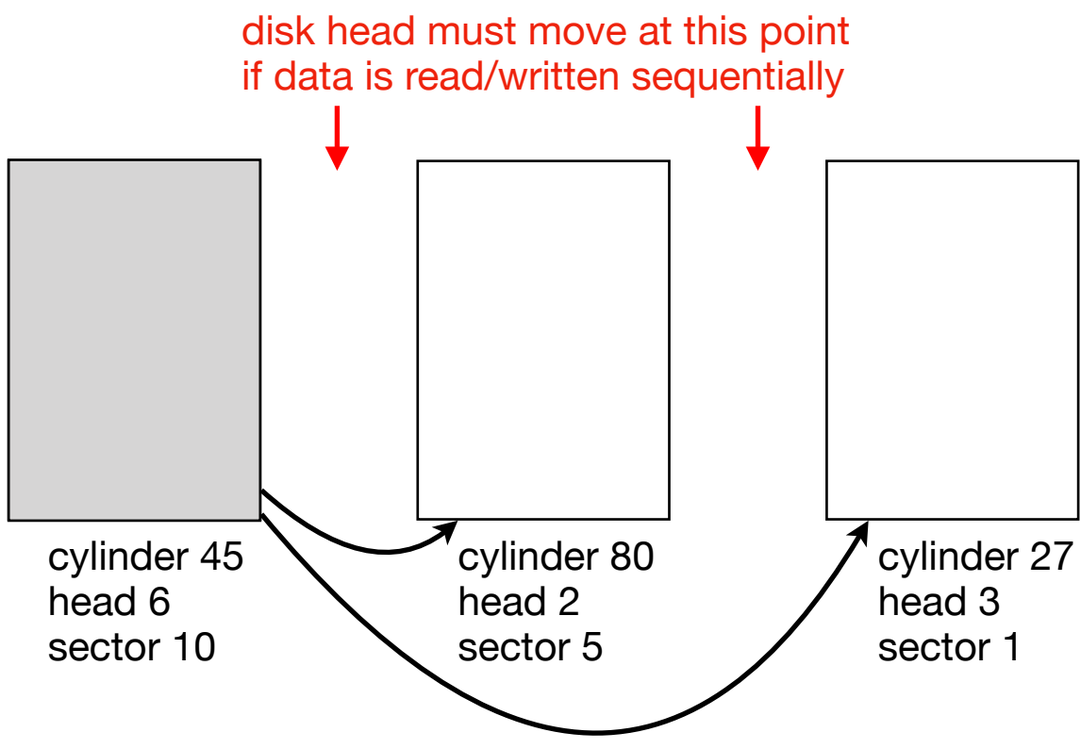


index block is an array of disk-block addresses (i^{th} entry points to the i^{th} disk block)

Indexed allocation

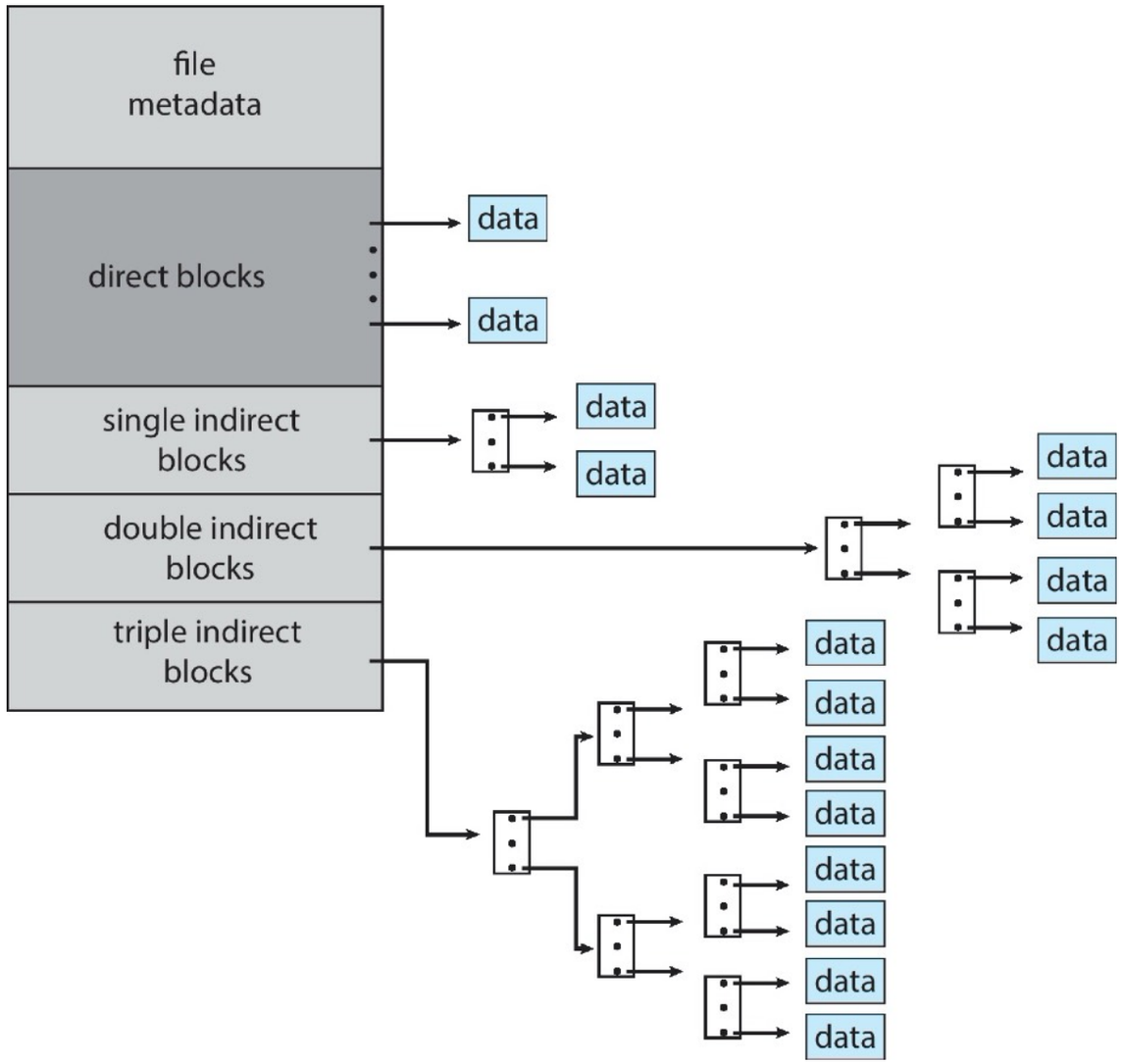
- address translation
  - depends on how many blocks are needed to store the index table
  - divide the logical address by (block size)x(block size - p) to find the index block that contains the index of the block
  - then create the physical address





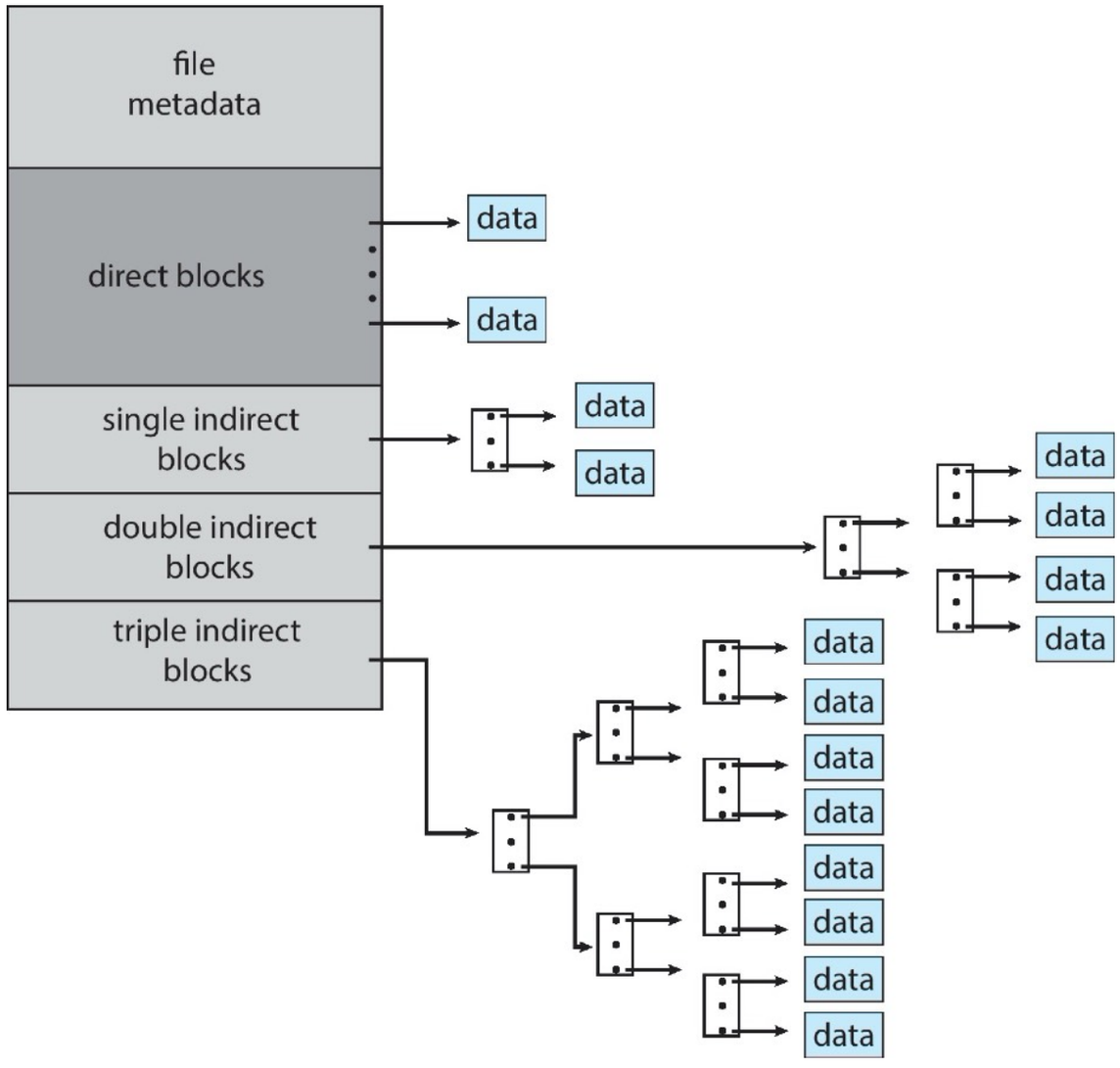
Hybrid multi-level approach

- used in UNIX-based file systems



Hybrid multi-level approach

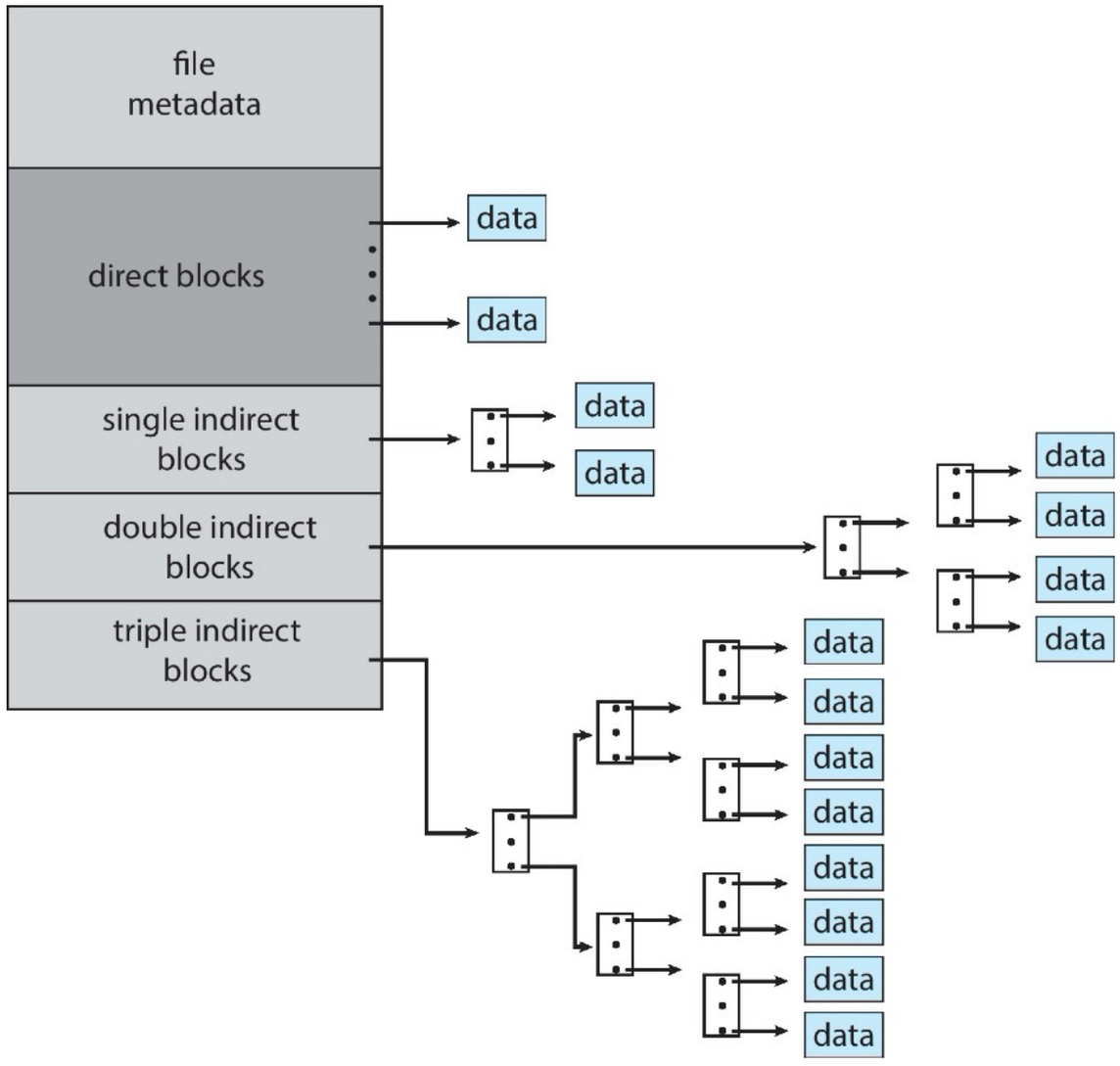
- used in UNIX-based file systems
- each inode contains 13 block pointers
  - the first 10 pointers point to 4KB data blocks (direct pointers)
  - the 11th pointer points to a block of 1024 pointers to 4KB data blocks (1-level indirection)
  - the 12th pointer points to a block of pointers to indirect blocks (2-level indirection)
  - the 13th pointer points to a block of pointers to indirect blocks (3-level indirection)



more index blocks than can be addressed with 32-bit file pointer

Hybrid multi-level approach

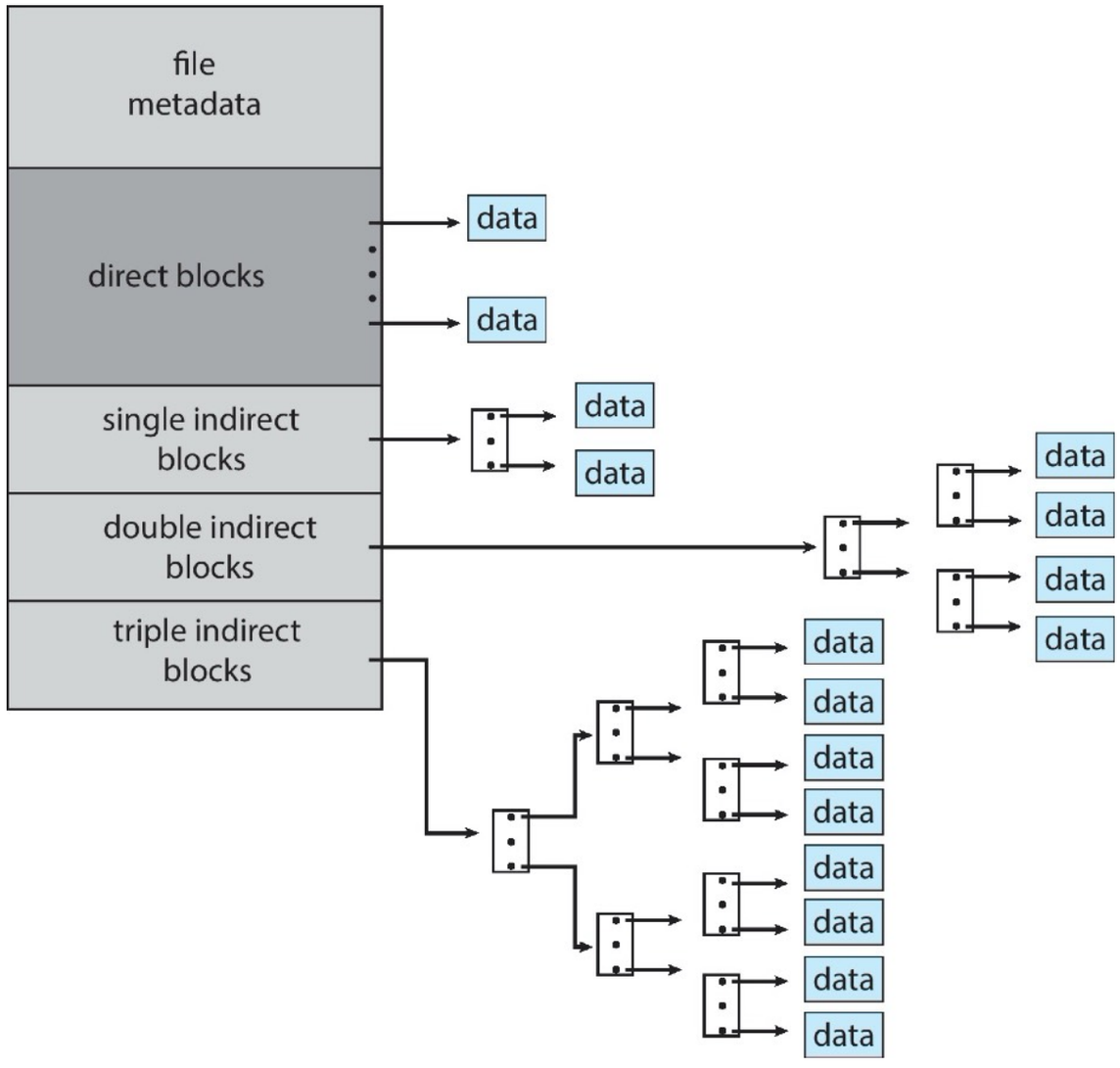
- used in UNIX-based file systems
- each inode contains 13 block pointers
  - the first 10 pointers point to 4KB data blocks (direct pointers)
  - the 11th pointer points to a block of 1024 pointers to 4KB data blocks (1-level indirection)
  - the 12th pointer points to a block of pointers to indirect blocks (2-level indirection)
  - the 13th pointer points to a block of pointers to indirect blocks (3-level indirection)
- advantages
  - supports incremental file growth, accessing small files is efficient (no indirection), max size limit is a couple of GB



more index blocks than can be addressed with 32-bit file pointer

Hybrid multi-level approach

- used in UNIX-based file systems
- each inode contains 13 block pointers
  - the first 10 pointers point to 4KB data blocks (direct pointers)
  - the 11th pointer points to a block of 1024 pointers to 4KB data blocks (1-level indirection)
  - the 12th pointer points to a block of pointers to indirect blocks (2-level indirection)
  - the 13th pointer points to a block of pointers to indirect blocks (3-level indirection)
- advantages
  - supports incremental file growth, accessing small files is efficient (no indirection), max size limit is a couple of GB
- disadvantages
  - several levels of indirection is inefficient for random access to very large files, accessing file data needs many disk seeks (especially for large files)



more index blocks than can be addressed with 32-bit file pointer

Performance

- contiguous allocation works well for sequential and random accesses

Performance

- contiguous allocation works well for sequential and random accesses
- linked allocation is good for sequential access, but not random access

Performance

- contiguous allocation works well for sequential and random accesses
- linked allocation is good for sequential access, but not random access
- if access type is declared at creation, OS can select either contiguous or linked allocation scheme for that file

Performance

- contiguous allocation works well for sequential and random accesses
- linked allocation is good for sequential access, but not random access
- if access type is declared at creation, OS can select either contiguous or linked allocation scheme for that file
- indexed allocation is more complex
  - single block access could require 2 index block reads followed by a data block read

Performance

- contiguous allocation works well for sequential and random accesses
- linked allocation is good for sequential access, but not random access
- if access type is declared at creation, OS can select either contiguous or linked allocation scheme for that file
- indexed allocation is more complex
  - single block access could require 2 index block reads followed by a data block read
- how about a hybrid approach?
  - small files (up to 3 or 4 blocks) are contiguously allocated; indexed allocation is used when file grows large
  - works well since most files are small!