CS 3113
File Systems

Data Storage Challenges

For any storage system, we have to answer questions such as:

- How will new data be stored? How do we select its location?
- When we want to retrieve data, how do we find this data and access it?

What matters:

- Efficiency in storage and access
- Integrity
- Volume of data
- Ease of access, even when faced with many different physical implementations

The Type of Application Matters

Different applications have different requirements for storage:

- Data collection: quickly storing data when it arrives in big bursts
- Databases: often highly-structured data
  - Rapid look-up by key (or multiple keys)
- Many other apps: semi-structured

File Systems are About Abstraction

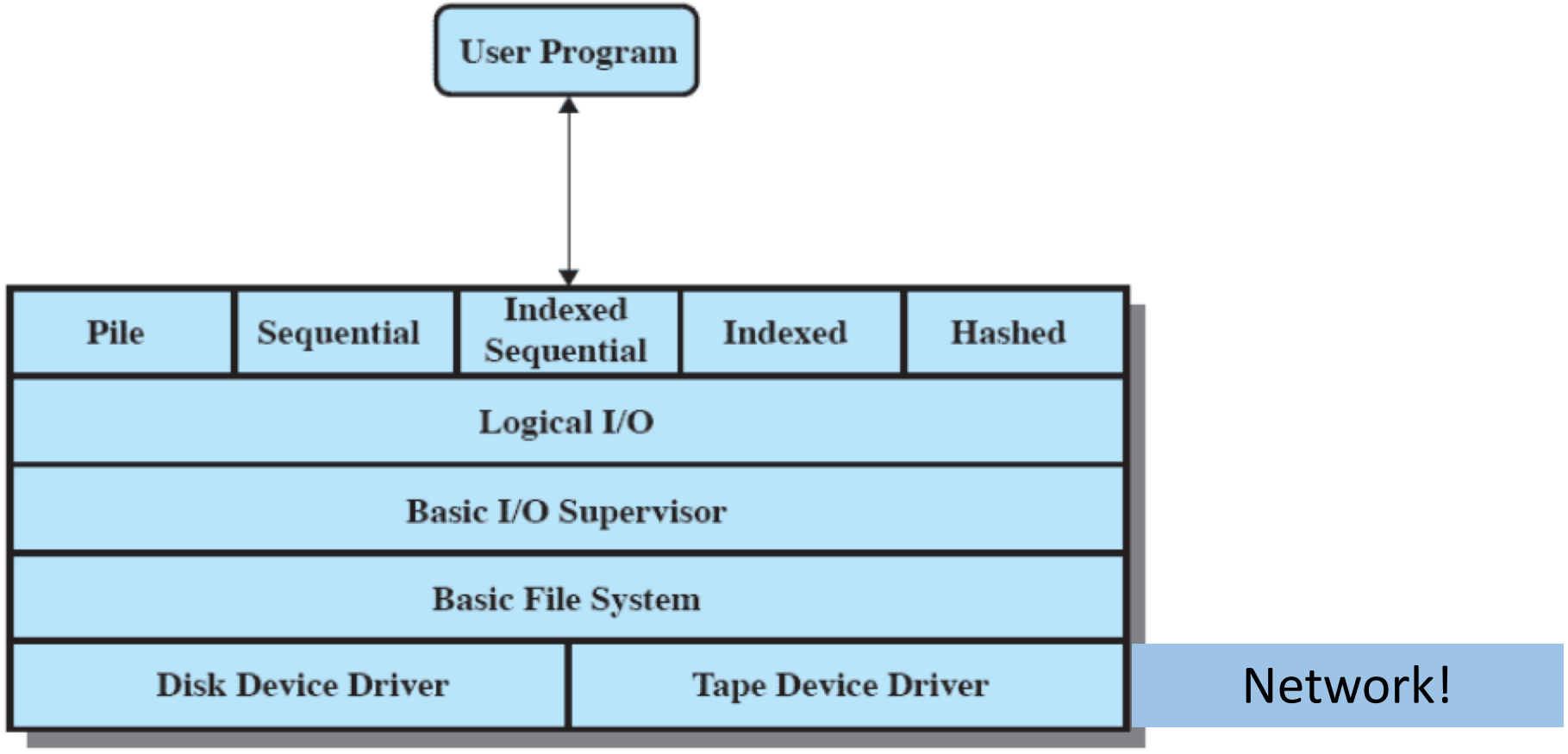
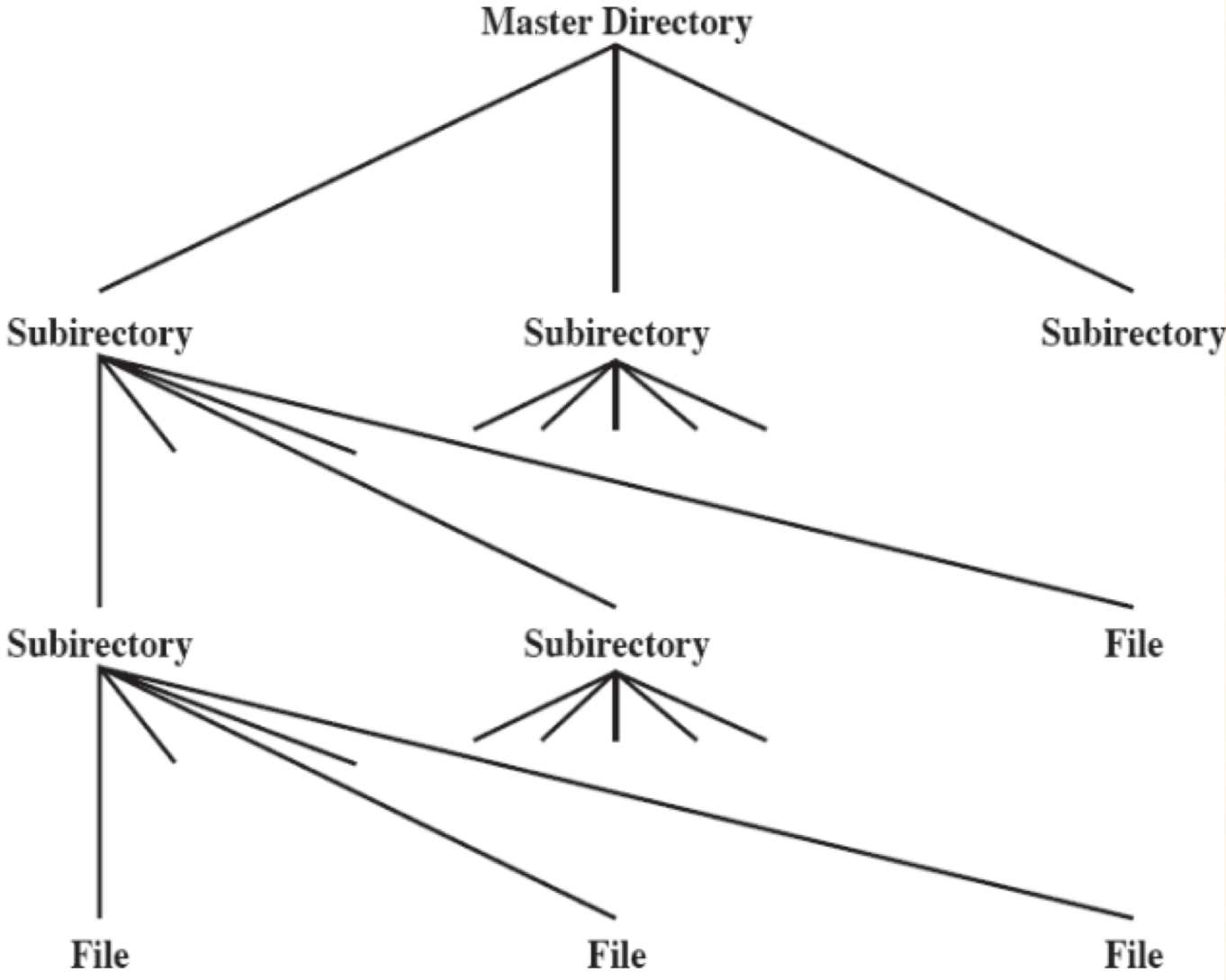


Figure 12.1 File System Software Architecture

Directory Hierarchy



File/Directory Permissions

- Shell: `ls -l`

- Nominal permissions: read (r), write (w), execute (x)
  - x for a directory means that one can access the details of the directory
- Three different permission sets:
  - User (owner) of the file/directory. A user ID is associated with object
  - Group ownership of the file/directory. A group ID is also associated with the object
  - Other ownership (any user).

Disks

- Block-type device: data are read/written in fixed-sized groups of bytes (blocks).
- Not uncommon to have blocks of 512, 1024 or 4096 bytes

Disks

Figure 14-1 shows the relationship between disk partitions and file systems, and shows the parts of a (generic) file system.

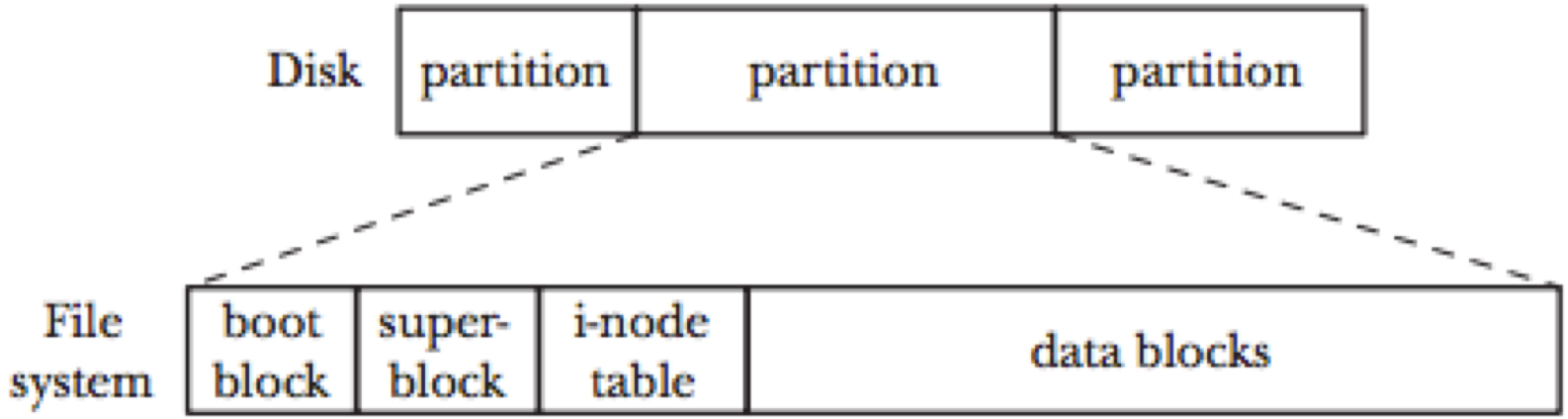


Figure 14-1: Layout of disk partitions and a file system

Partition Layout

- Boot block: can contain the first information that a computer needs to boot into an OS
- Superblock: data about the partition: size of the individual blocks, size of the i-node table and size of the file system
- I-node table: one entry for each directory or file represented in the file system
- Data blocks: data for files and directories

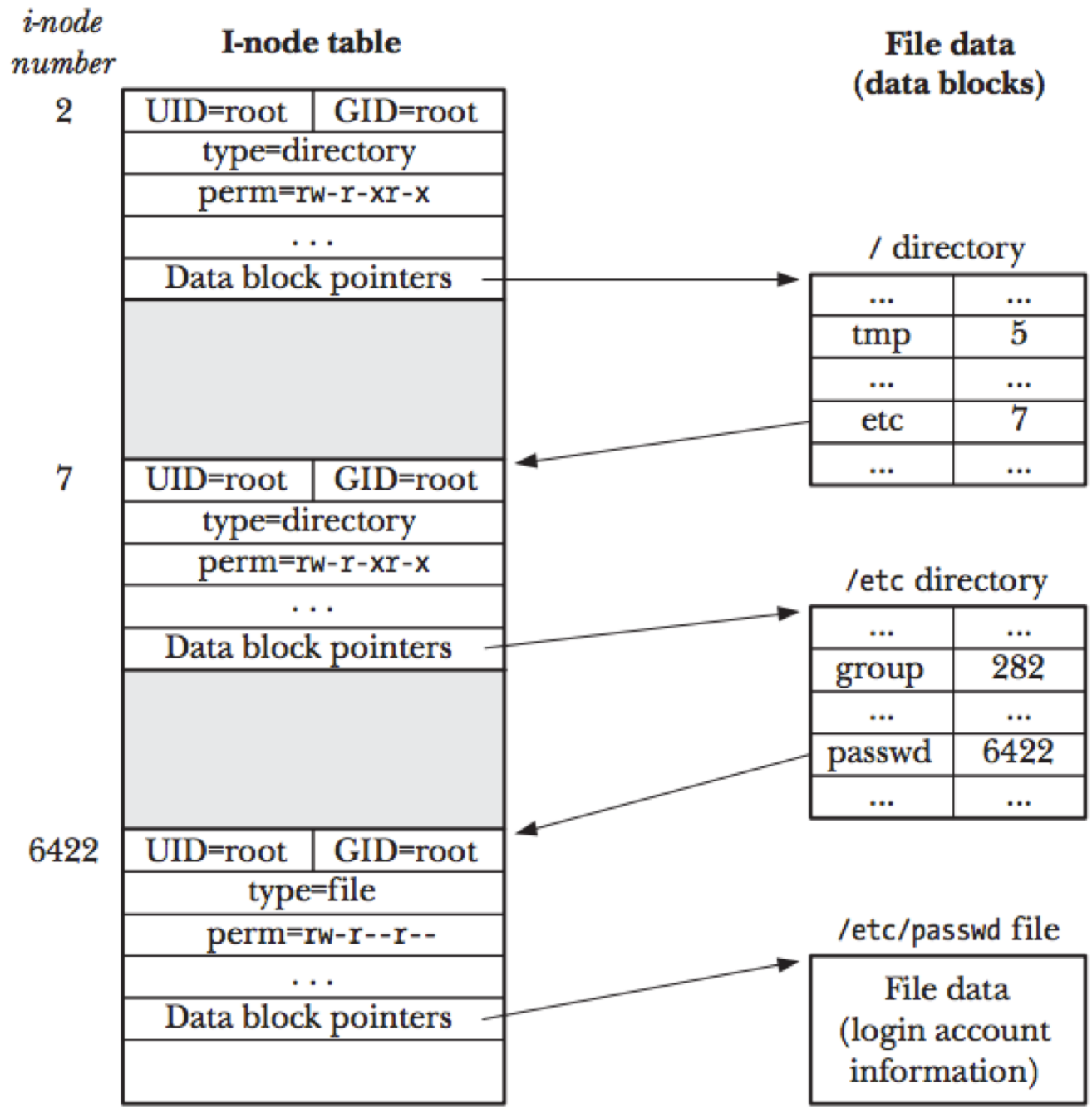
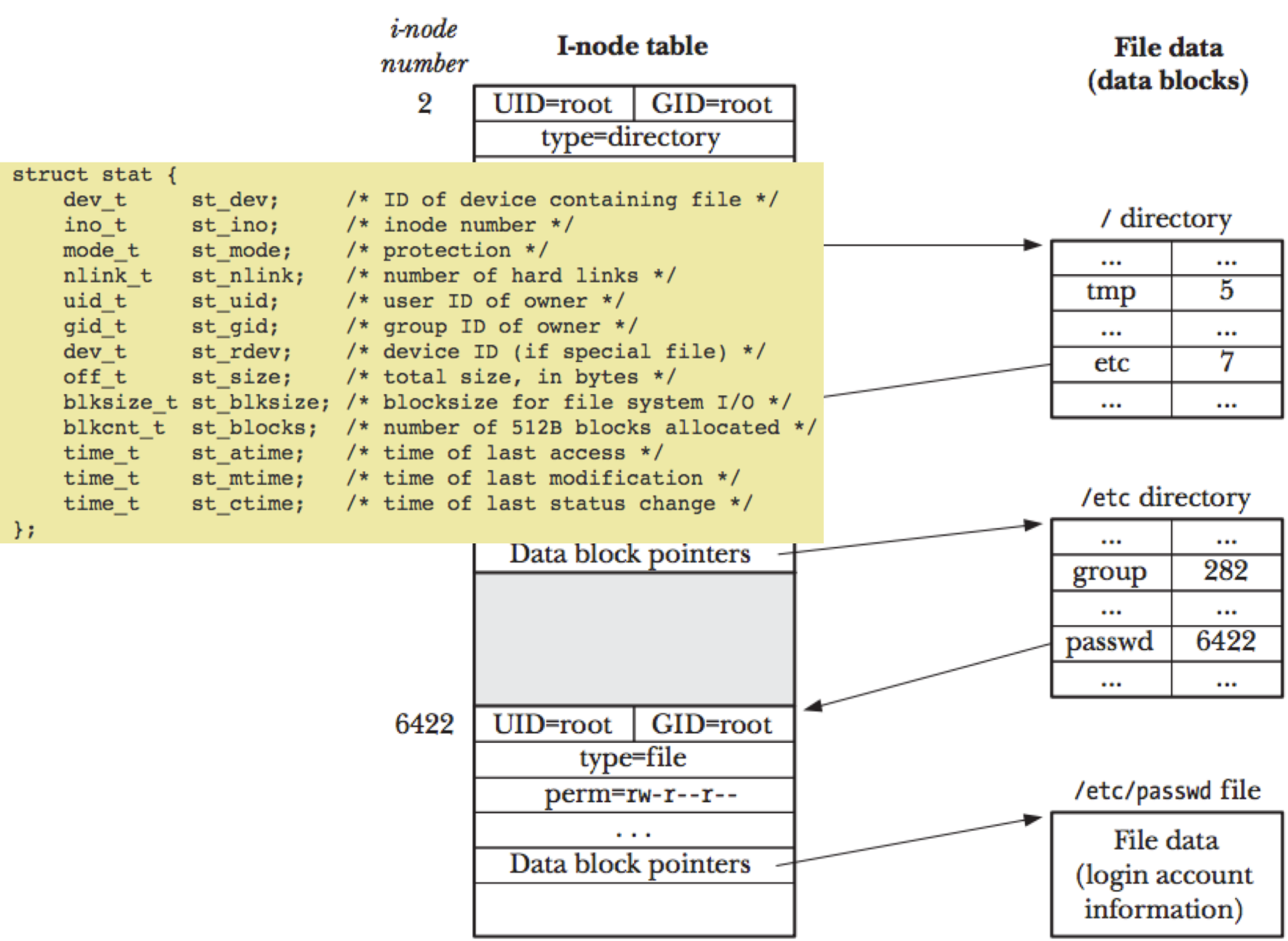


Figure 18-1: Relationship between i-node and directory structures for the file `/etc/passwd`




```
struct stat {
    dev_t     st_dev;     /* ID of device containing file */
    ino_t     st_ino;     /* inode number */
    mode_t    st_mode;    /* protection */
    nlink_t   st_nlink;   /* number of hard links */
    uid_t     st_uid;     /* user ID of owner */
    gid_t     st_gid;     /* group ID of owner */
    dev_t     st_rdev;    /* device ID (if special file) */
    off_t     st_size;    /* total size, in bytes */
    blksize_t st_blksize; /* blocksize for file system I/O */
    blkcnt_t  st_blocks;  /* number of 512B blocks allocated */
    time_t    st_atime;   /* time of last access */
    time_t    st_mtime;   /* time of last modification */
    time_t    st_ctime;   /* time of last status change */
};
```

Figure 18-1: Relationship between i-node and directory structures for the file `/etc/passwd`

Hard vs Symbolic Links

- Hard link: a directory entry that references the I-Node for a child file/directory
  - Multiple hard links to the same I-Node are possible. Hence, the child I-Node will keep a count of how many references are made to it
  - When we “remove” a file/directory, the count is decremented. If the count drops to zero, then the contents are actually removed from the file system
  - Hard links cannot cross file systems
- Symbolic (or soft) link: symbolic representation of the path to a file/directory
  - When the symbolic link is accessed, the file system will follow this path
  - Can cross file systems

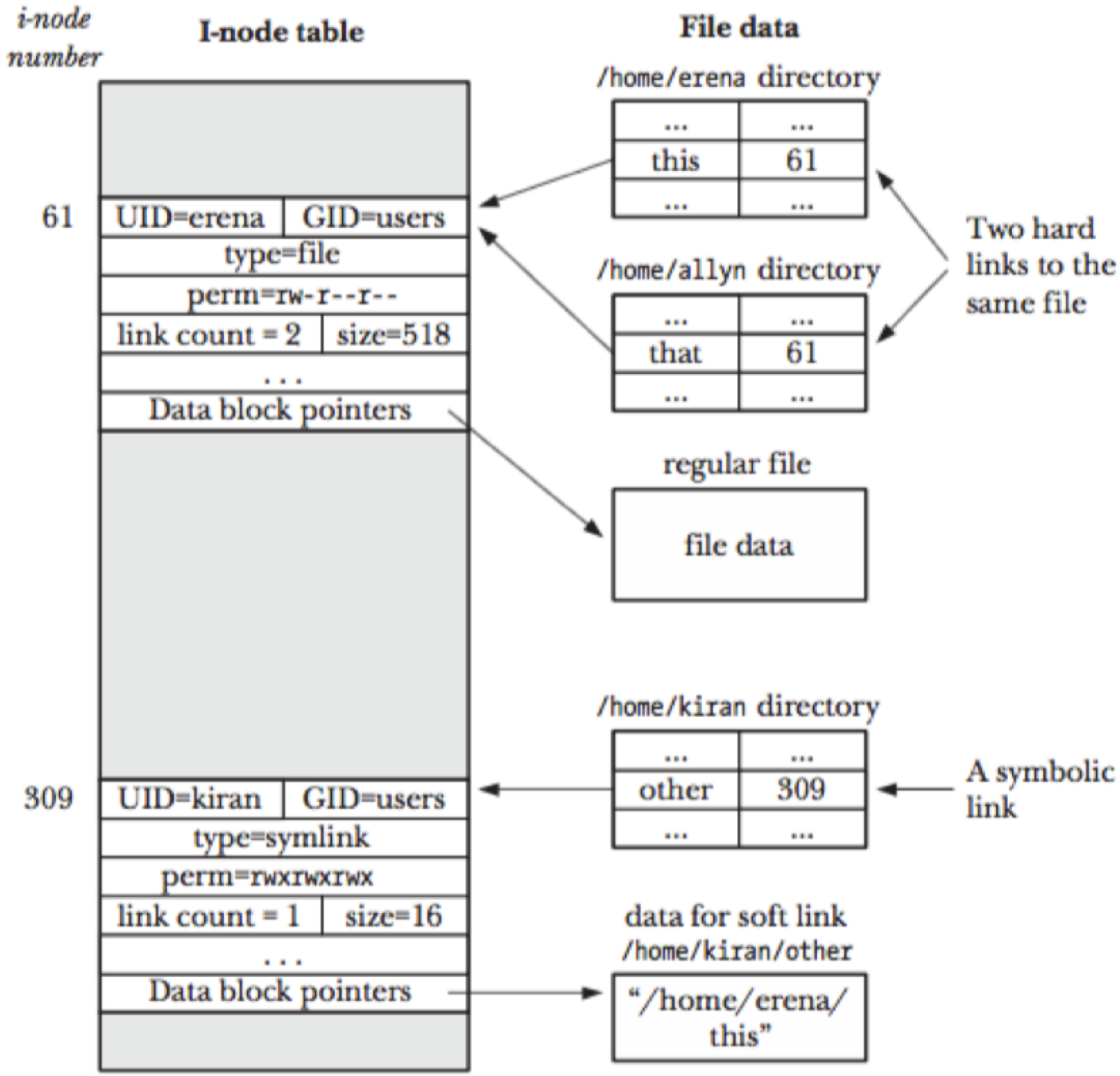


Figure 18-2: Representation of hard and symbolic links

ext2 file system

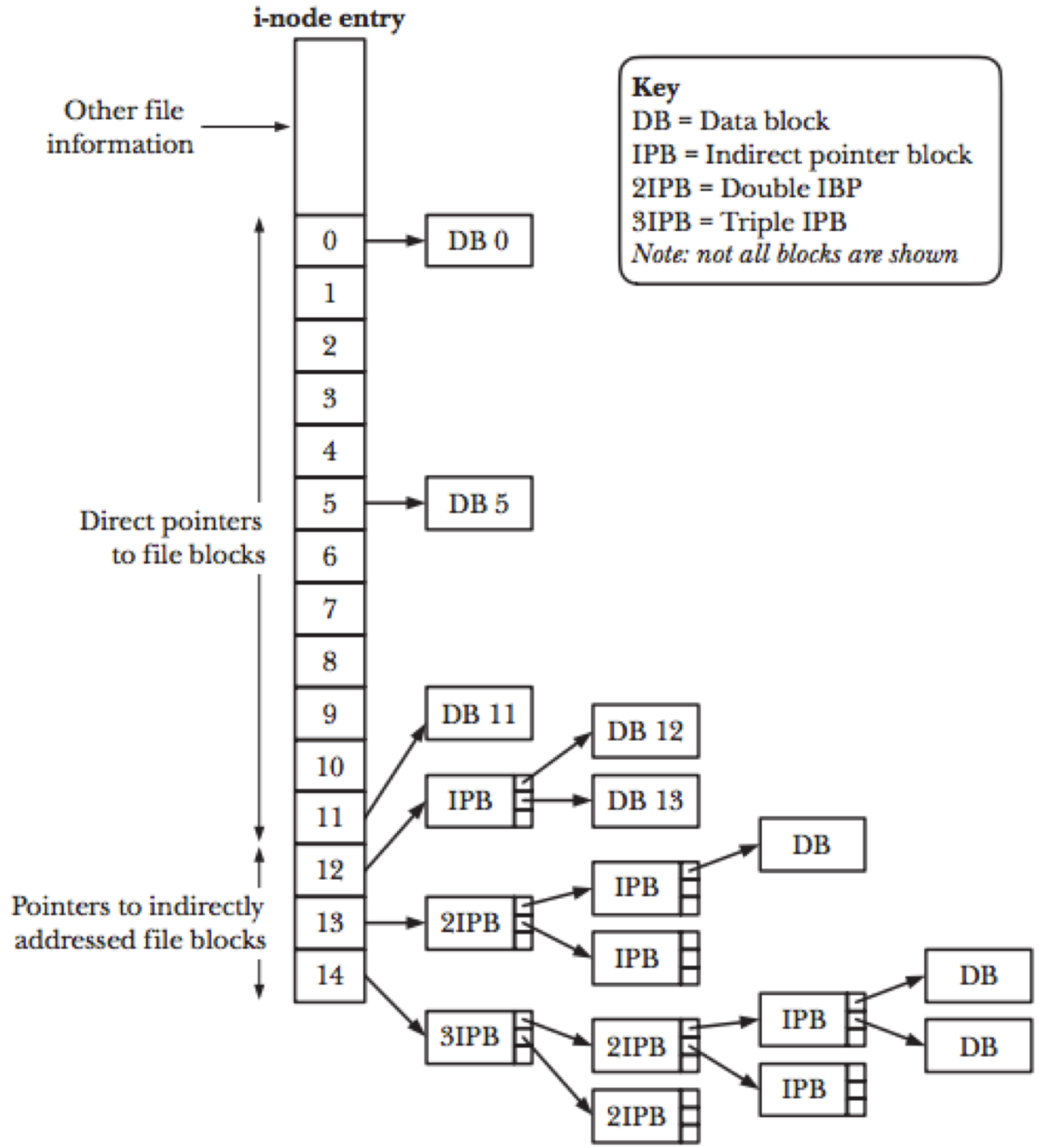


Figure 14-2: Structure of file blocks for a file in an *ext2* file system

The Many File System Types

- Different OSes have made different decisions about file system structure
- In addition: file system structure within Linux has evolved: ext2, ext3, ext4
- We would like to be able to access any of these file systems from within Linux
- Solution: Linux provides a layer of abstraction called the **Virtual File System** (VFS)
  - Provides a standard set of file/directory manipulation operations
  - However, the user program level may need to take steps to deal with features not supported by the underlying file system

Mount Points

- We would like to provide a file system abstraction that makes it appear as though all of the storage resources live within one common directory tree (starting from /)
- Linux solution: provide a way to virtually make a file system appear as though it is a directory with the root directory

To the instance…

View mounted file systems:

`df`

View mounted file systems:

- /proc/mounts
- /etc/fstab

To the instance…

Create a new file system in a file:

```
dd if=/dev/zero of=~/myfile bs=512 count=4096
mkfs.ext3 ~/myfile
sudo mkdir /myfs
sudo mount ~fagg/myfile /myfs
```

Unmount the new file system:

```
sudo umount /myfs
```

- Note: not allowed if the fs is being accessed at that instant

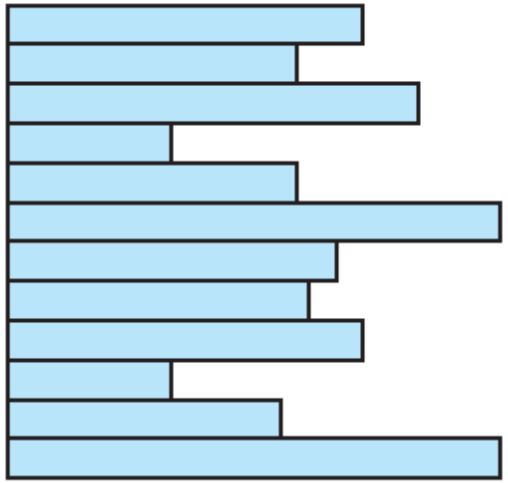
Other Notes

- Be careful about mounting file systems from other people
  - In particular, a useful mount option is “nosetuid”: this disables programs on that file system from being executable as the admin
  - Also the “noexec” option turns off execution of any file on the mounted file system
- Mounts can also come from the network!

Organizing Data within a File: Application Dependent

Pile

- Store records as data arrive
- Record structure may vary from one to the next
- Records/fields should describe themselves in some way
- Rapid storage: append to the end of the file
- Slow sequential access due to variable length records
- Slow access: must search through the file for the data of interest

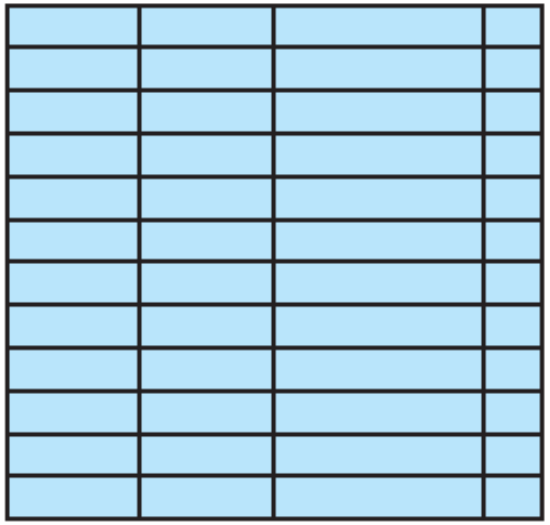


Variable-length records
Variable set of fields
Chronological order

(a) Pile File

Sequential File

- All records have the same structure (we know this ahead of time)
  - Records DO NOT have to be self-describing
- Key field: unique description of the record (e.g., a record ID)
- Rapid storage: append to the end of the file
- Fast sequential access
- Slow random access: must search through the file for the key of interest
  - But easy to know where the records start in the file



Fixed-length records
Fixed set of fields in fixed order
Sequential order based on key field

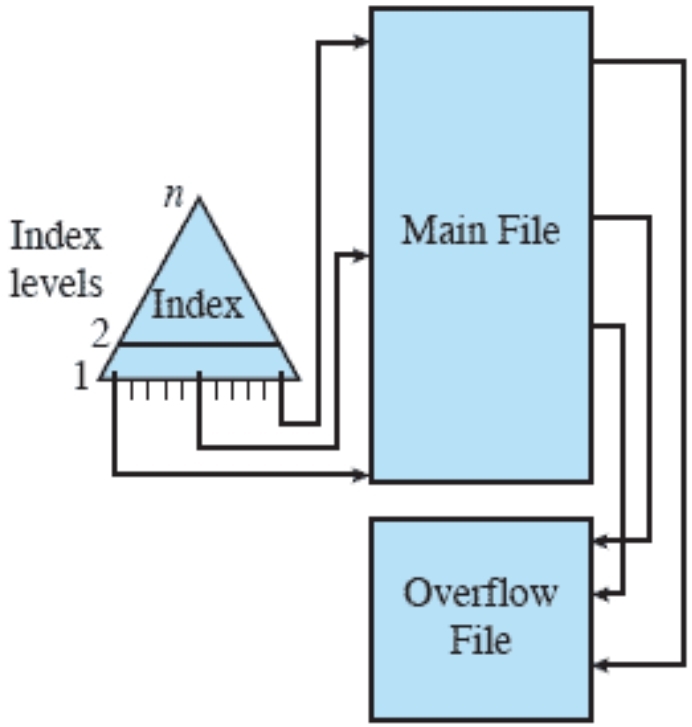
(b) Sequential File

Indexed Sequential File

Simple case:

- Index consists of <key, address> pairs
  - Given key, quickly find address (or small range of addresses)
- Main file: sequential file containing the records
- Overflow file: quick storage of new records
  - These records will be incorporated into the sequential file as time allows
- Rapid storage: append to overflow
- Fast sequential access
- Fast random access

Extensions: can have multiple levels of indices

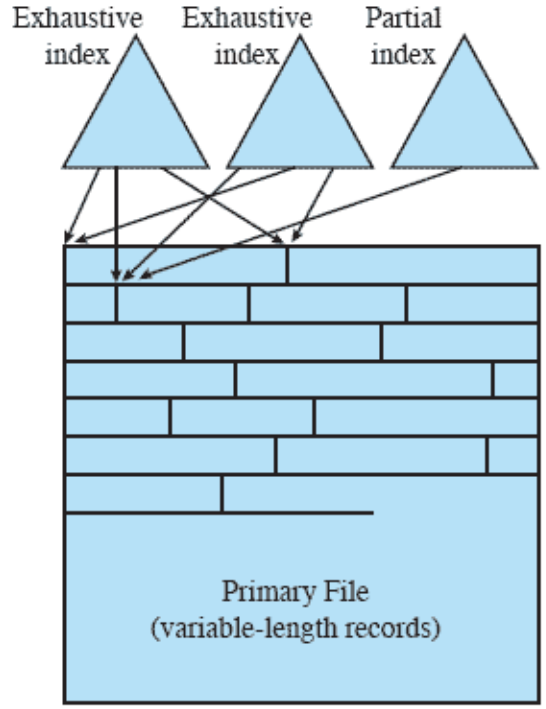


(c) Indexed Sequential File

Indexed File

Goal: want to be able to search using different fields of the records

- Multiple indices:
  - Exhaustive: each record is represented
  - Partial: not every record is represented
- General case: records can be of variable length
  - But common to have fixed-length records
- Rapid storage: append to file
- Slow sequential access
- Fast random access



(d) Indexed File

Hashed File

- Fixed-length records
- Fixed-length file
  - Some records may not contain information
- Use hash function from key to address
- Rapid storage: place at address (beware of collisions!)
- Slow sequential access
- Very fast random access

Sequential Indexing with B-Trees