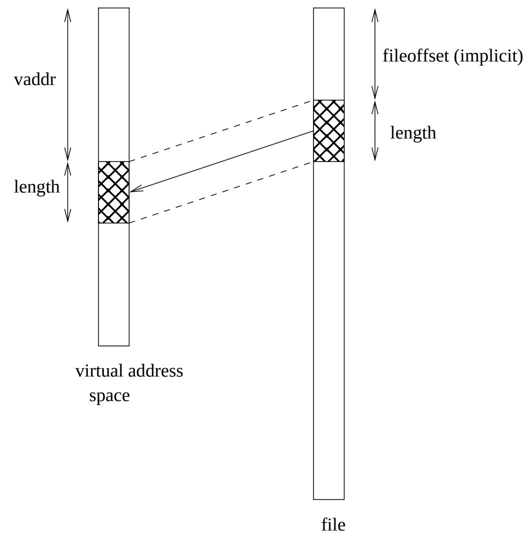
Files and File Systems

- files: persistent, named data objects
  - data consists of a sequence of numbered bytes
  - file may change size over time
  - file has associated meta-data
    * examples: owner, access controls, file type, creation and access timestamps
- file system: a collection of files which share a common name space
  - allows files to be created, destroyed, renamed, ...

File Interface

- open, close
  - open returns a file identifier (or handle or descriptor), which is used in subsequent operations to identify the file. (Why is this done?)
- read, write, seek
  - read copies data from a file into a virtual address space
  - write copies data from a virtual address space into a file
  - seek enables non-sequential reading/writing
- get/set file meta-data, e.g., Unix `fstat, chmod`

File Read



`read(fileID, vaddr, length)`

File Position

- each file descriptor (open file) has an associated file position
- read and write operations
  - start from the current file position
  - update the current file position
- this makes sequential file I/O easy for an application to request
- for non-sequential (random) file I/O, use:
  - a seek operation (`lseek`) to adjust file position before reading or writing
  - a positioned read or write operation, e.g., Unix `pread, pwrite`:
    `pread(fileId,vaddr,length,filePosition)`

Sequential File Reading Example (Unix)

```
char buf[512];
int i;
int f = open("myfile",O_RDONLY);
for(i=0; i<100; i++) {
  read(f,(void *)buf,512);
}
close(f);
```

Read the first $100 * 512$ bytes of a file, 512 bytes at a time.

File Reading Example Using Seek (Unix)

```
char buf[512];
int i;
int f = open("myfile",O_RDONLY);
for(i=1; i<=100; i++) {
  lseek(f,(100-i)*512,SEEK_SET);
  read(f,(void *)buf,512);
}
close(f);
```

Read the first $100 * 512$ bytes of a file, 512 bytes at a time, in reverse order.

File Reading Example Using Positioned Read

```
char buf[512];
int i;
int f = open("myfile",O_RDONLY);
for(i=0; i<100; i+=2) {
  pread(f,(void *)buf,512,i*512);
}
close(f);
```

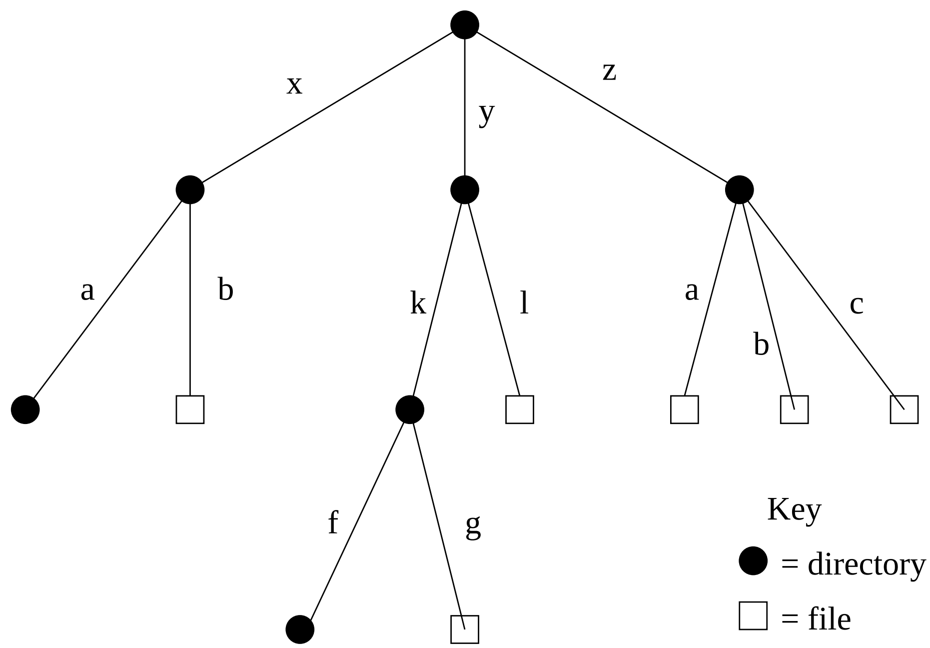
Read every second 512 byte chunk of a file, until 50 have been read.

Directories and File Names

- A directory maps *file names* (strings) to *i-numbers*
  - an i-number is a unique (within a file system) identifier for a file or directory
  - given an i-number, the file system can find the data and meta-data for the file
- Directories provide a way for applications to group related files
- Since directories can be nested, a filesystem's directories can be viewed as a tree, with a single *root* directory.
- In a directory tree, files are leaves
- Files may be identified by *pathnames*, which describe a path through the directory tree from the root directory to the file, e.g.:

  `/home/user/courses/cs350/notes/filesys.pdf`
- Directories also have pathnames
- Applications refer to files using pathnames, not i-numbers

Hierarchical Namespace Example



Hard Links

- a *hard link* is an association between a name (string) and an i-number
  - each entry in a directory is a hard link
- when a file is created, so is a hard link to that file
  - `open(/a/b/c,O_CREAT|O_TRUNC)`
  - this creates a new file if a file called `/a/b/c` does not already exist
  - it also creates a hard link to the file in the directory `/a/b`
- Once a file is created, *additional* hard links can be made to it.
  - example: `link(/x/b,/y/k/h)` creates a new hard link `h` in directory `/y/k`. The link refers to the i-number of file `/x/b`, which must exist.
- linking to an existing file creates a new pathname for that file
  - each file has a unique i-number, but may have multiple pathnames
- Not possible to `link` to a directory (to avoid cycles)

Unlinking and Referential Integrity

- hard links can be removed:
  - `unlink(/x/b)`
- the file system ensures that hard links have *referential integrity*, which means that if the link exists, the file that it refers to also exists.
  - When a hard link is created, it refers to an existing file.
  - There is no system call to delete a file. Instead, a file is deleted when its last hard link is removed.

Symbolic Links

- a *symbolic link*, or *soft link*, is an association between a name (string) and a pathname.
  - `symlink(/z/a,/y/k/m)` creates a symbolic link `m` in directory `/y/k`. The symbolic link refers to the pathname `/z/a`.
- If an application attempts to open `/y/k/m`, the file system will
  1. recognize `/y/k/m` as a symbolic link, and
  2. attempt to open `/z/a` instead
- referential integrity is *not* preserved for symbolic links
  - in the example above, `/z/a` need not exist!

UNIX/Linux Link Example (1 of 3)

```
% cat > file1
This is file1.
<cntl-d>
% ls -li
685844 -rw------- 1 user group 15 2008-08-20 file1
% ln file1 link1
% ln -s file1 sym1
% ln not-here link2
ln: not-here: No such file or directory
% ln -s not-here sym2
```

Files, hard links, and soft/symbolic links.

UNIX/Linux Link Example (2 of 3)

```
% ls -li
685844 -rw------- 2 user group 15 2008-08-20 file1
685844 -rw------- 2 user group 15 2008-08-20 link1
685845 lrwxrwxrwx 1 user group  5 2008-08-20 sym1 -> file1
685846 lrwxrwxrwx 1 user group  8 2008-08-20 sym2 -> not-here
% cat file1
This is file1.
% cat link1
This is file1.
% cat sym1
This is file1.
% cat sym2
cat: sym2: No such file or directory
% /bin/rm file1
```

Accessing and manipulating files, hard links, and soft/symbolic links.

UNIX/Linux Link Example (3 of 3)

```
% ls -li
685844 -rw------- 1 user group 15 2008-08-20 link1
685845 lrwxrwxrwx 1 user group  5 2008-08-20 sym1 -> file1
685846 lrwxrwxrwx 1 user group  8 2008-08-20 sym2 -> not-here
% cat link1
This is file1.
% cat sym1
cat: sym1: No such file or directory
% cat > file1
This is a brand new file1.
<cntl-d>
% ls -li
685847 -rw------- 1 user group 27 2008-08-20 file1
685844 -rw------- 1 user group 15 2008-08-20 link1
685845 lrwxrwxrwx 1 user group  5 2008-08-20 sym1 -> file1
685846 lrwxrwxrwx 1 user group  8 2008-08-20 sym2 -> not-here
% cat link1
This is file1.
% cat sym1
This is a brand new file1.
```

Different behaviour for hard links and soft/symbolic links.

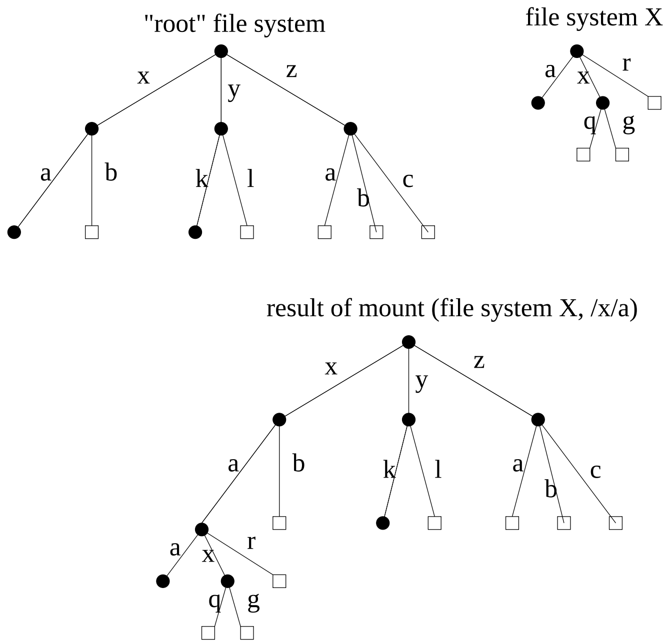
Multiple File Systems

- it is not uncommon for a system to have multiple file systems
- some kind of global file namespace is required
- two examples:

  DOS/Windows: use two-part file names: file system name, pathname within file system
  - example: `C:\user\cs350\schedule.txt`

  Unix: create single hierarchical namespace that combines the namespaces of two file systems
  - Unix `mount` system call does this
- mounting does *not* make two file systems into one file system
  - it merely creates a single, hierarchical namespace that combines the namespaces of two file systems
  - the new namespace is temporary - it exists only until the file system is unmounted

Unix `mount` Example



Links and Multiple File Systems

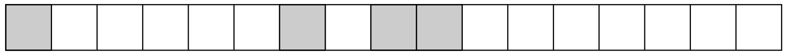
- hard links cannot cross file system boundaries
  - each hard link maps a name to an i-number, which is unique only *within* a file system
- for example, even after the mount operation illustrated on the previous slide, `link(/x/a/x/g,/z/d)` would result in an error, because the new link, which is in the root file system refers to an object in file system X
- soft links do not have this limitation
- for example, after the mount operation illustrated on the previous slide:
  - `symlink(/x/a/x/g,/z/d)` would succeed
  - `open(/z/d)` would succeed, with the effect of opening `/z/a/x/g`.
- even if the `symlink` operation were to occur *before* the `mount` command, it would succeed

File System Implementation

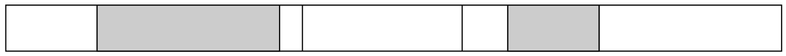
- what needs to be stored persistently?
  - file data
  - file meta-data
  - directories and links
  - file system meta-data
- non-persistent information
  - open files per process
  - file position for each open file
  - *cached* copies of persistent data

Space Allocation and Layout

- space on secondary storage may be allocated in fixed-size chunks or in chunks of varying size
- fixed-size chunks: *blocks*
  - simple space management
  - internal fragmentation (unused space in allocated blocks)
- variable-size chunks: *extents*
  - more complex space management
  - external fragmentation (wasted unallocated space)



fixed−size allocation



variable−size allocation

Layout matters on secondary storage! Try to lay a file out sequentially, or in large sequential extents that can be read and written efficiently.

File Indexing

- where is the data for a given file?
- common solution: per-file indexing
  - for each file, an index with pointers to data blocks or extents
    - in extent-based systems, need pointer and length for each extent
- how big should the index be?
  - need to accommodate both small files and very large files
  - approach: allow different index sizes for different files

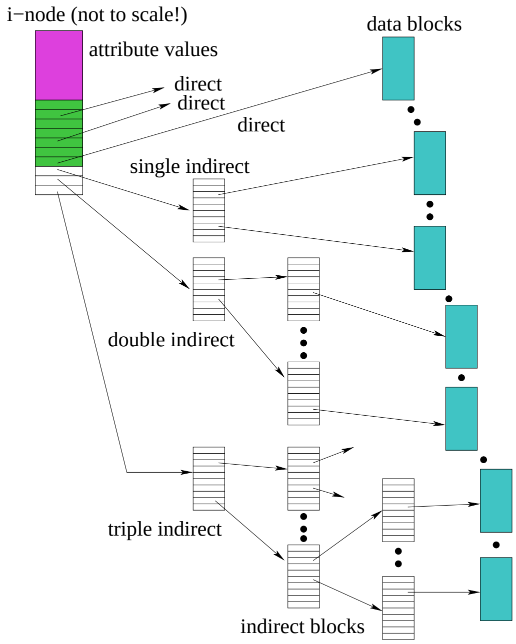
i-nodes

- per file index structure, *fixed size*
- holds file meta-data, and small number of pointers to data blocks
  - for small files, pointers in the i-node are sufficient to point to all data blocks
  - for larger files, allocate additional *indirect* blocks, which hold pointers to additional data blocks
- i-node table holds i-nodes for all files in a file system
  - in persistent storage
  - given i-number, can directly determine location of corresponding i-node in the i-node table

Example: Linux ext3 i-nodes

- i-node fields
  - file type
  - file permissions
  - file length
  - number of file blocks
  - time of last file access
  - time of last i-node update, last file update
  - number of hard links to this file
  - 12 *direct* data block pointers
  - one single, one double, one triple *indirect* data block pointer
- i-node size: 128 bytes
- i-node table: broken into smaller tables, each in a known location on the secondary storage device (disk)

i-node Diagram



Directories

- Implemented as a special type of file.
- Directory file contains directory entries, each consisting of
  - a file name (component of a path name)
  - the corresponding i-number
- Directory files can be read by application programs (e.g., `ls`)
- Directory files are only updated by the kernel, in response to file system operations, e.g, create file, create link
- Application programs cannot write directly to directory files. (Why not?)

Implementing Hard Links

- hard links are simply directory entries
- for example, consider:

  `link(/y/k/g,/z/m)`
- to implement this:
  1. find out the internal file identifier for `/y/k/g`
  2. create a new entry in directory `/z`
     - file name in new entry is `m`
     - file identifier (i-number) in the new entry is the one discovered in step 1

Implementing Soft Links

- soft links can be implemented as a special type of file
- for example, consider:

  `symlink(/y/k/g,/z/m)`
- to implement this:
  - create a new *symlink* file
  - add a new entry in directory `/z`
    - ∗ file name in new entry is `m`
    - ∗ i-number in the new entry is the i-number of the new symlink file
  - store the pathname string "/y/k/g" as the contents of the new symlink file

Pathname Translation

- input: a file pathname
- output: the i-number of the file the pathname refers to
- common to many file system calls, e.g., open
- basic idea (without error checking):

```
i = i-number of root directory
while (n = next component of pathname) {
   if i is not a directory then return ERROR
   i = lookup n in directory i
   if (i is a symbolic link file) {
      i = translate(link)
   }
}
return i
```


In-Memory (Non-Persistent) Structures

- per process
  - descriptor table
    - which file descriptors does this process have open?
    - to which file does each open descriptor refer?
    - what is the current file position for each descriptor?
- system wide
  - open file table
    - which files are currently open (by any process)?
  - i-node cache
    - in-memory copies of recently-used i-nodes
  - block cache
    - in-memory copies of data blocks and indirect blocks

Problems Caused by Failures

- a single logical file system operation may require several disk I/O operations
- example: deleting a file
  - remove entry from directory
  - remove file index (i-node) from i-node table
  - mark file's data blocks free in free space index
- what if, because of a failure, some but not all of these changes are reflected on the disk?

- system failure will destroy in-memory file system structures
- persistent structures should be *crash consistent*, i.e., should be consistent when system restarts after a failure

---

Fault Tolerance

- special-purpose consistency checkers (e.g., Unix `fsck` in Berkeley FFS, Linux ext2)
  - runs after a crash, before normal operations resume
  - find and attempt to repair inconsistent file system data structures, e.g.:
    - file with no directory entry
    - free space that is not marked as free
- journaling (e.g., Veritas, NTFS, Linux ext3)
  - record file system meta-data changes in a journal (log), so that sequences of changes can be written to disk in a single operation
  - *after* changes have been journaled, update the disk data structures (*write-ahead logging*)
  - after a failure, redo journaled updates in case they were not done before the failure