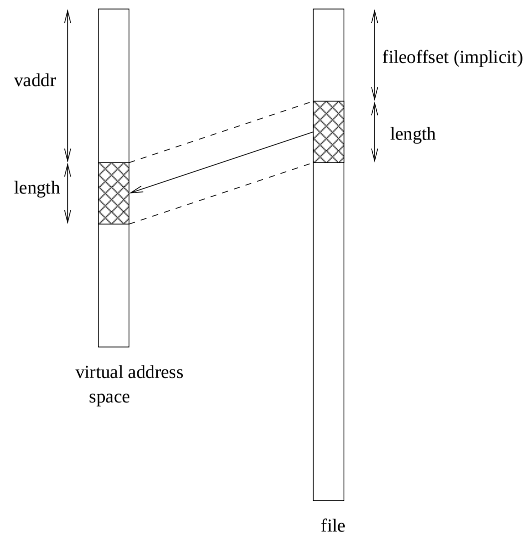
Files and File Systems

- files: persistent, named data objects
  - data consists of a sequence of numbered bytes
  - alternatively, a file may have some internal structure, e.g., a file may consist of sequence of numbered records
  - file may change size over time
  - file has associated meta-data (attributes), in addition to the file name
    - ∗ 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
  - must specify which file to read, which part of the file to read, and where to put the data that has been read (similar for write).
  - often, file position is implicit (why?)
- seek
- get/set file attributes, e.g., Unix `fstat, chmod`

File Read



`read(fileID, vaddr, length)`

File Position

- may be associated with the file, with a process, or with a file descriptor (Unix style)
- 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:
  - `seek`, 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);
lseek(f,99*512,SEEK_SET);
for(i=0; i<100; i++) {
  read(f,(void *)buf,512);
  lseek(f,-1024,SEEK_CUR);
}
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.

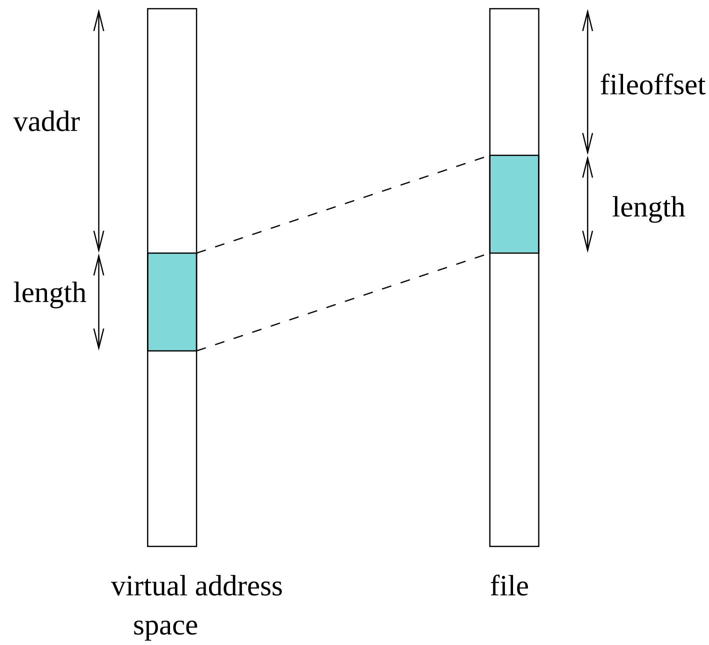
Memory-Mapped Files

- generic interface:

  `vaddr ← mmap(file descriptor,fileoffset,length)`
  `munmap(vaddr,length)`
- `mmap` call returns the virtual address to which the file is mapped
- `munmap` call unmaps mapped files within the specified virtual address range

Memory-mapping is an alternative to the read/write file interface.

Memory Mapping Illustration



Memory Mapping Update Semantics

- what should happen if the virtual memory to which a file has been mapped is updated?
- some options:
  - prohibit updates (read-only mapping)
  - eager propagation of the update to the file (too slow!)
  - lazy propagation of the update to the file
    - user may be able to request propagation (e.g., Posix `msync()`
    - propagation may be guaranteed by `munmap()`
  - allow updates, but do not propagate them to the file

Memory Mapping Concurrency Semantics

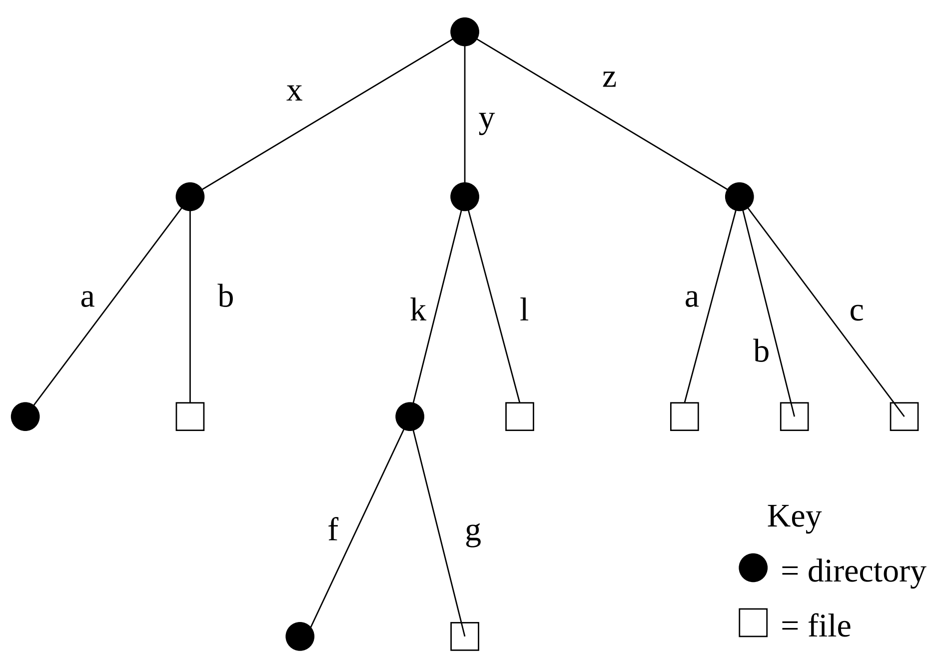
- what should happen if a memory mapped file is updated?
  - by a process that has mmapped the same file
  - by a process that is updating the file using a `write()` system call
- options are similar to those on the previous slide. Typically:
  - propagate lazily: processes that have mapped the file *may* eventually see the changes
  - propagate eagerly: other processes will see the changes
    - ∗ typically implemented by invalidating other process's page table entries

File Names

- application-visible objects (e.g., files, directories) are given names
- the file system is responsible for associating names with objects
- the namespace is typically structured, often as a tree or a DAG
- namespace structure provides a way for users and applications to organize and manage information
- in a structured namespace, objects may be identified by *pathnames*, which describe a path from a root object to the object being identified, e.g.:

  `/home/kmsalem/courses/cs350/notes/filesys.ps`

Hierarchical Namespace Example

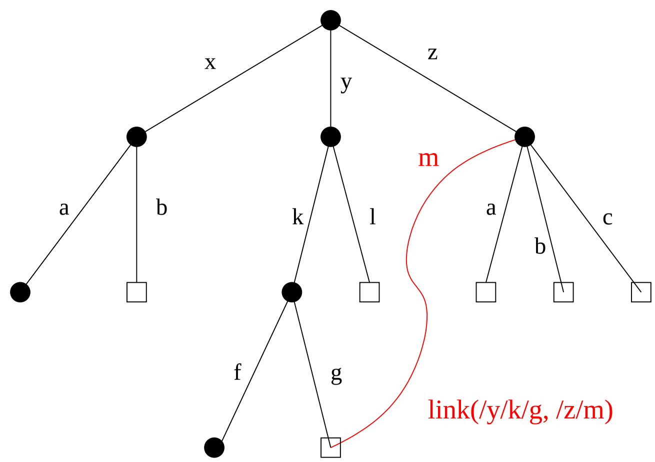


Hard Links

- a *hard link* is an association between a name and an underlying file (or directory)
- typically, when a file is created, a single link is created to the that file as well (else the file would be difficult to use!)
  - POSIX example: `creat(pathname,mode)` creates both a new empty file object and a link to that object (using `pathname`)
- some file systems allow additional hard links to be made to existing files. This allows more than one name from the file system's namespace to refer the *same underlying object*.
  - POSIX example: `link(oldpath,newpath)` creates a new hard link, using `newpath`, to the underlying object identified by `oldpath`

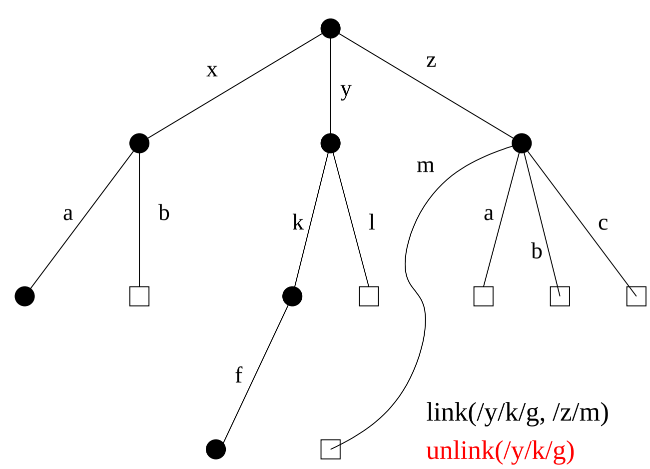
File systems ensure *referential integrity* for hard links. A hard link refers to the object it was created for until the link is explicitly destroyed. (What are the implications of this?)

Hard Link Illustration



Hard links are a way to create *non-hierarchical structure* in the namespace. Hard link creation may be restricted to restrict the kinds of structure that applications can create. Example: no hard links to directories.

Unlink Example

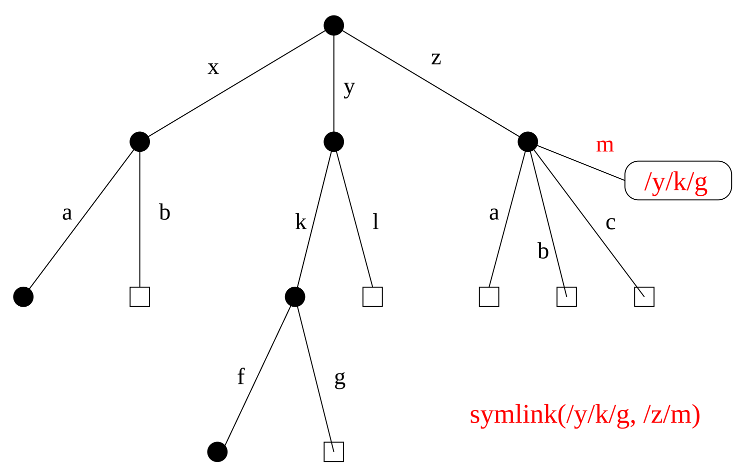


Removing the *last* link to a file causes the file itself to be deleted. Deleting a file that has a link would destroy the referential integrity of the link.

Symbolic Links

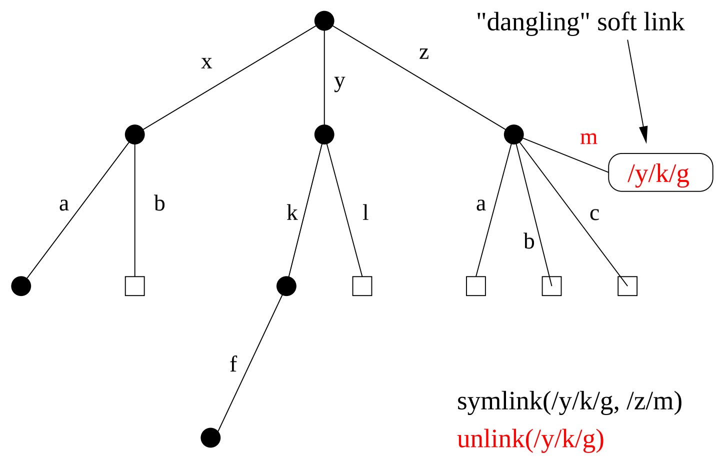
- a *Symbolic link*, or *soft link*, is an association between two names in the file namespace. Think of it is a way of defining a synonym for a filename.
  - `symlink(oldpath,newpath)` creates a symbolic link from `newpath` to `oldpath`, i.e., `newpath` becomes a synonym for `oldpath`.
- symbolic links relate filenames to filenames, while hard links relate filenames to underlying file objects!
- referential integrity is *not* preserved for symbolic links, e.g., the system call above can succeed even if there is no object named `oldpath`

Soft Link Example



`/y/k/g` still has only one hard link after the `symlink` call. A new symlink object records the association between `/z/m` and `/y/k/g`. `open(/z/m)` will now have the same effect as `open(/y/k/g)`.

Soft Link Example with Unlink



A file is deleted by this `unlink` call. An attempt to `open(/z/m)` after the `unlink` will result in an error. If a *new* file called `/y/k/g` is created, a subsequent `open(/z/m)` will open the new file.

Linux Link Example (1 of 2)

```
% cat > file1
This is file1.
% ls -li
685844 -rw------- 1 kmsalem kmsalem 15 2008-08-20 file1
% ln file1 link1
% ln -s file1 sym1
% ls -li
685844 -rw------- 2 kmsalem kmsalem 15 2008-08-20 file1
685844 -rw------- 2 kmsalem kmsalem 15 2008-08-20 link1
685845 lrwxrwxrwx 1 kmsalem kmsalem  5 2008-08-20 sym1 -> file1
% cat file1
This is file1.
% cat link1
This is file1.
% cat sym1
This is file1.
```

A file, a hard link, a soft link.

Linux Link Example (2 of 2)

```
% /bin/rm file1
% ls -li
685844 -rw------- 1 kmsalem kmsalem 15 2008-08-20 link1
685845 lrwxrwxrwx 1 kmsalem kmsalem  5 2008-08-20 sym1 -> file1
% cat link1
This is file1.
% cat sym1
cat: sym1: No such file or directory
% cat > file1
This is a brand new file1.
% ls -li
685846 -rw------- 1 kmsalem kmsalem 27 2008-08-20 file1
685844 -rw------- 1 kmsalem kmsalem 15 2008-08-20 link1
685845 lrwxrwxrwx 1 kmsalem kmsalem  5 2008-08-20 sym1 -> file1
% cat link1
This is file1.
% cat sym1
This is a brand new file1.
```

Different behaviour for hard links and soft links.

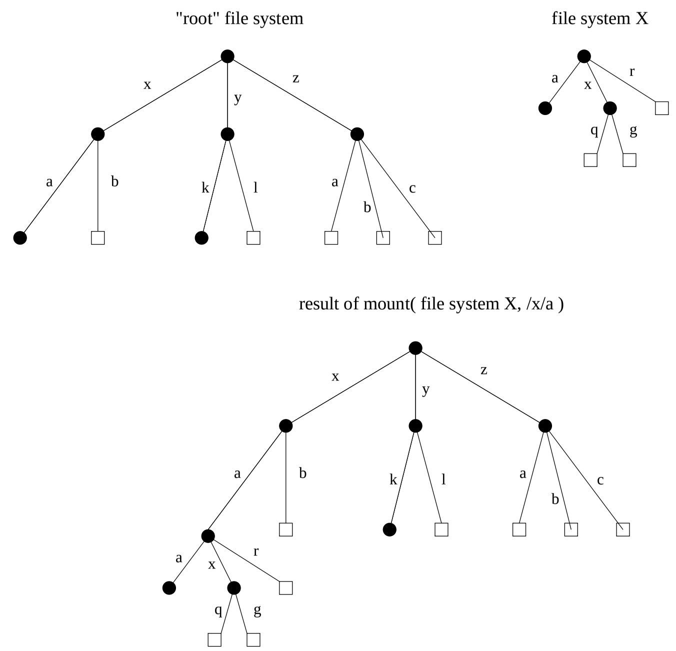
Multiple File Systems

- 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
  - example: `C:\kmsalem\cs350\schedule.txt`

  Unix: merge file graphs into a single graph
  - Unix `mount` system call does this

Unix `mount` Example



Links and Multiple File Systems

- a hard link associates a name in the file system namespace with a file in that file system
- typically, hard links cannot cross file system boundaries
- 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

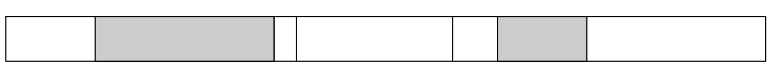
File System Implementation

- space management
- file indexing (how to locate file data and meta-data)
- directories
- links
- buffering, in-memory data structures
- persistence

Space Allocation and Layout

- space may be allocated in fixed-size chunks, or in chunks of varying size
- fixed-size chunks: simple space management, but internal fragmentation
- variable-size chunks: external fragmentation

fixed−size allocation



variable−size allocation

- *layout* matters! Try to lay a file out sequentially, or in large sequential extents that can be read and written efficiently.

File Indexing

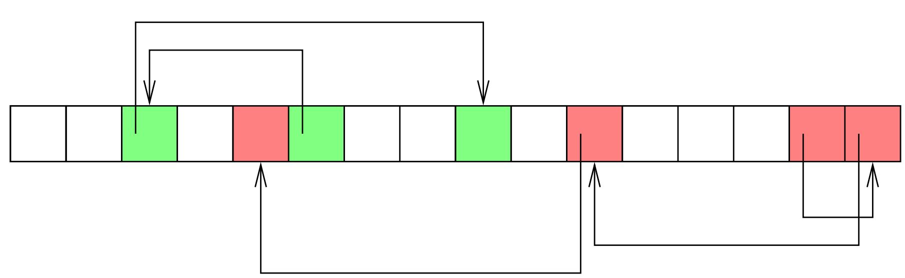
- in general, a file will require more than one chunk of allocated space
- this is especially true because files can grow
- how to find all of a file's data?

  chaining:
  - each chunk includes a pointer to the next chunk
  - OK for sequential access, poor for random access

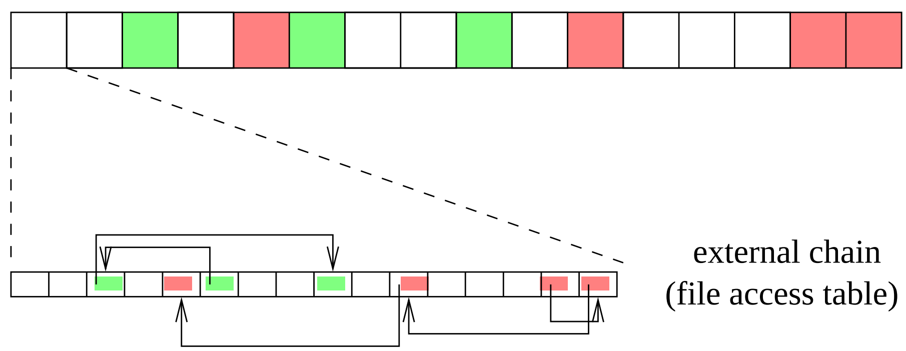
  external chaining: DOS file allocation table (FAT), for example
  - like chaining, but the chain is kept in an external structure

  per-file index: Unix i-node, for example
  - for each file, maintain a table of pointers to the file's blocks or extents

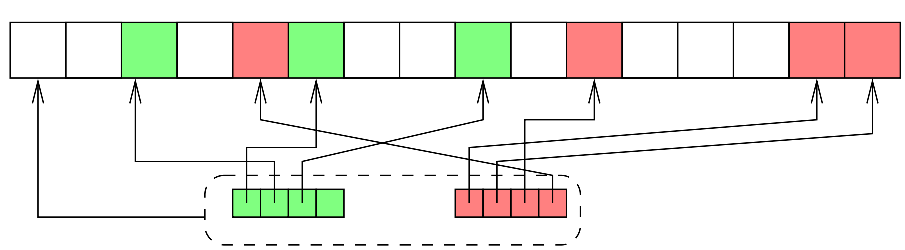
Chaining



External Chaining (File Access Table)



Per-File Indexing



Internal File Identifiers

- typically, a file system will assign a unique internal identifier to each file, directory or other object
- given an identifer, the file system can *directly* locate a record containing key information about the file, such as:
  - the per-file index to the file data (if per-file indexing is used), or the location of the file's first data block (if chaining is used)
  - file meta-data (or a reference to the meta-data), such as
    - file owner
    - file access permissions
    - file acesss timestamps
    - file type
- for example, a file identifier might be a number which indexes an on-disk array of file records

Example: Unix i-nodes

- an i-node is a record describing a file
- each i-node is uniquely identified by an i-number, which determines its physical location on the disk
- an i-node is a *fixed size* record containing:

  file attribute values
  - file type
  - file owner and group
  - access controls
  - creation, reference and update timestamps
  - file size

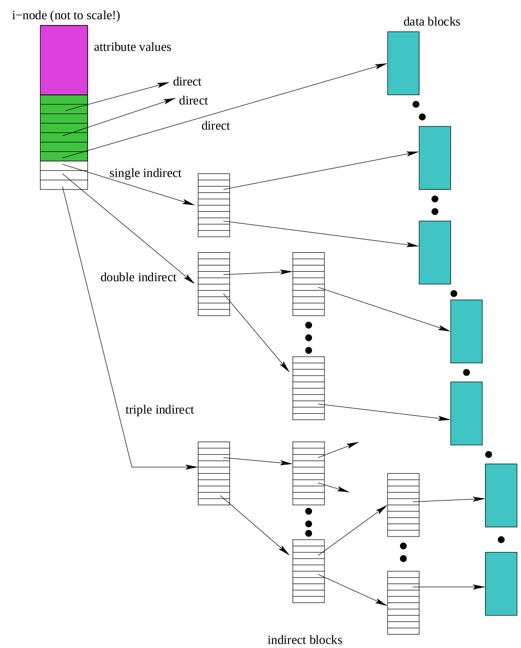
  direct block pointers: approximately 10 of these

  single indirect block pointer

  double indirect block pointer

  triple indirect block pointer

i-node Diagram



Directories

- A directory consists of a set of entries, where each entry is a record that includes:
  - a file name (component of a path name)
  - the internal file identifier (e.g., i-number) of the file
- A directory can be implemented as a special type of file. The directory entries are the contents of the file.
- The file system should not allow directory files to be directly written by application programs. Instead, the directory is updated by the file system as files are created and destroyed

Implementing Hard Links

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

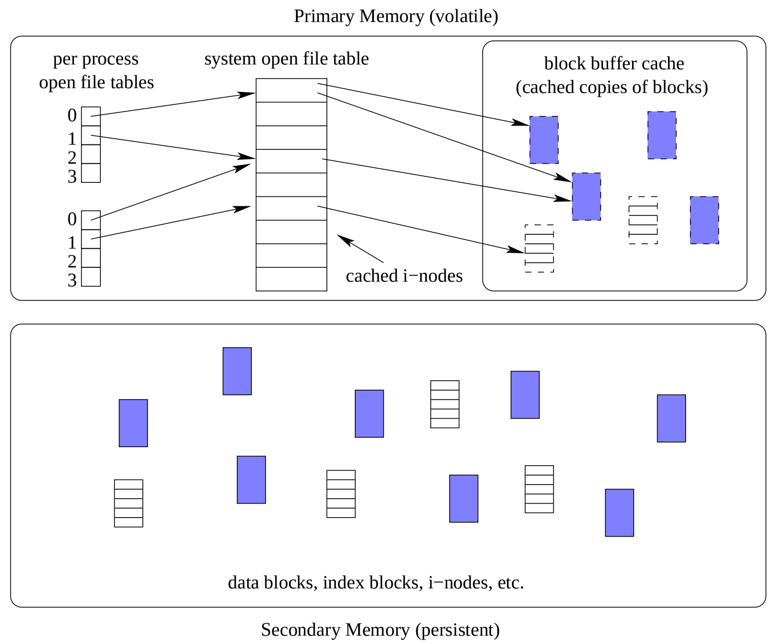
  `link(/y/k/g,/z/m)`
- to implement this:
  1. find out the internal file identifier for `/y/k/g`
  2. create a new entry in directory `/z`
     - file name in new entry is `m`
     - file identifier (i-number) in the new entry is the one discovered in step 1

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
- change the behaviour of the `open` system call so that when the symlink file is encountered during `open(/z/m)`, the file `/y/k/g` will be opened instead.

Main Memory Data Structures



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 a failure, some but not all of these changes are reflected on the disk?

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