

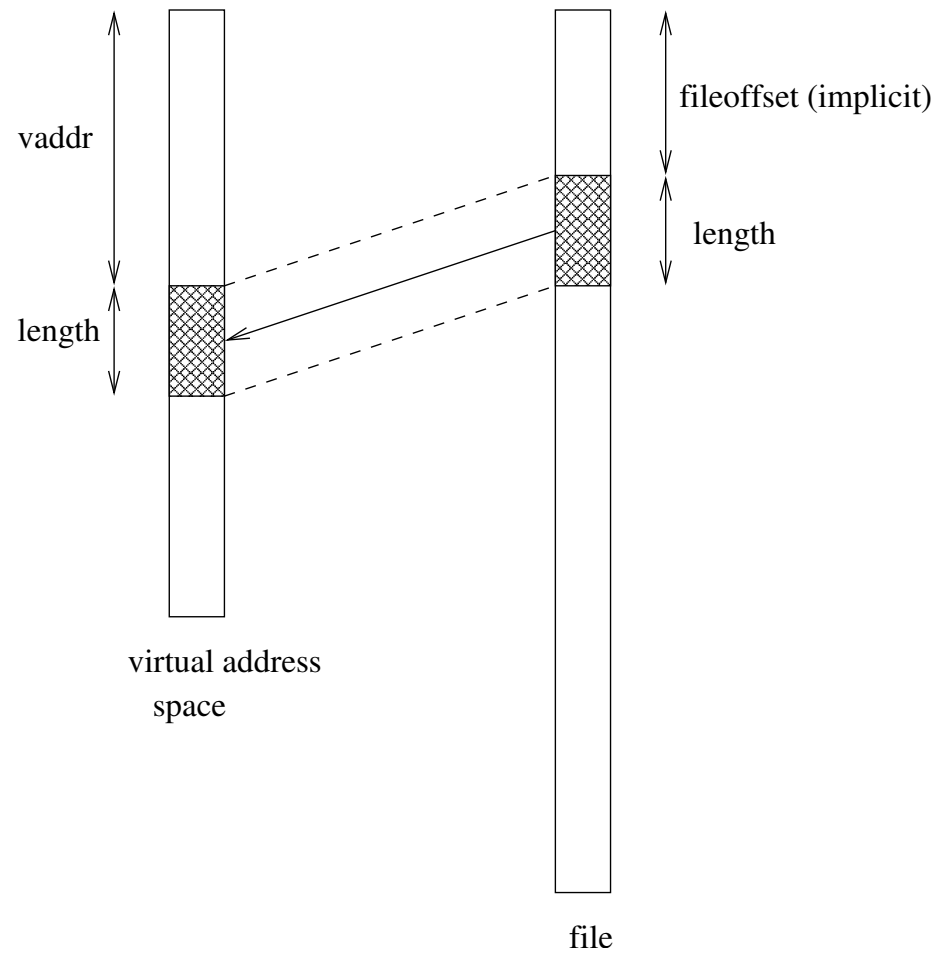
## Files and File Systems

- files: persistent, named data objects
  - data consists of a sequence of numbered bytes
  - alternatively, a file may have some internal structure, e.g., a data 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

---

---

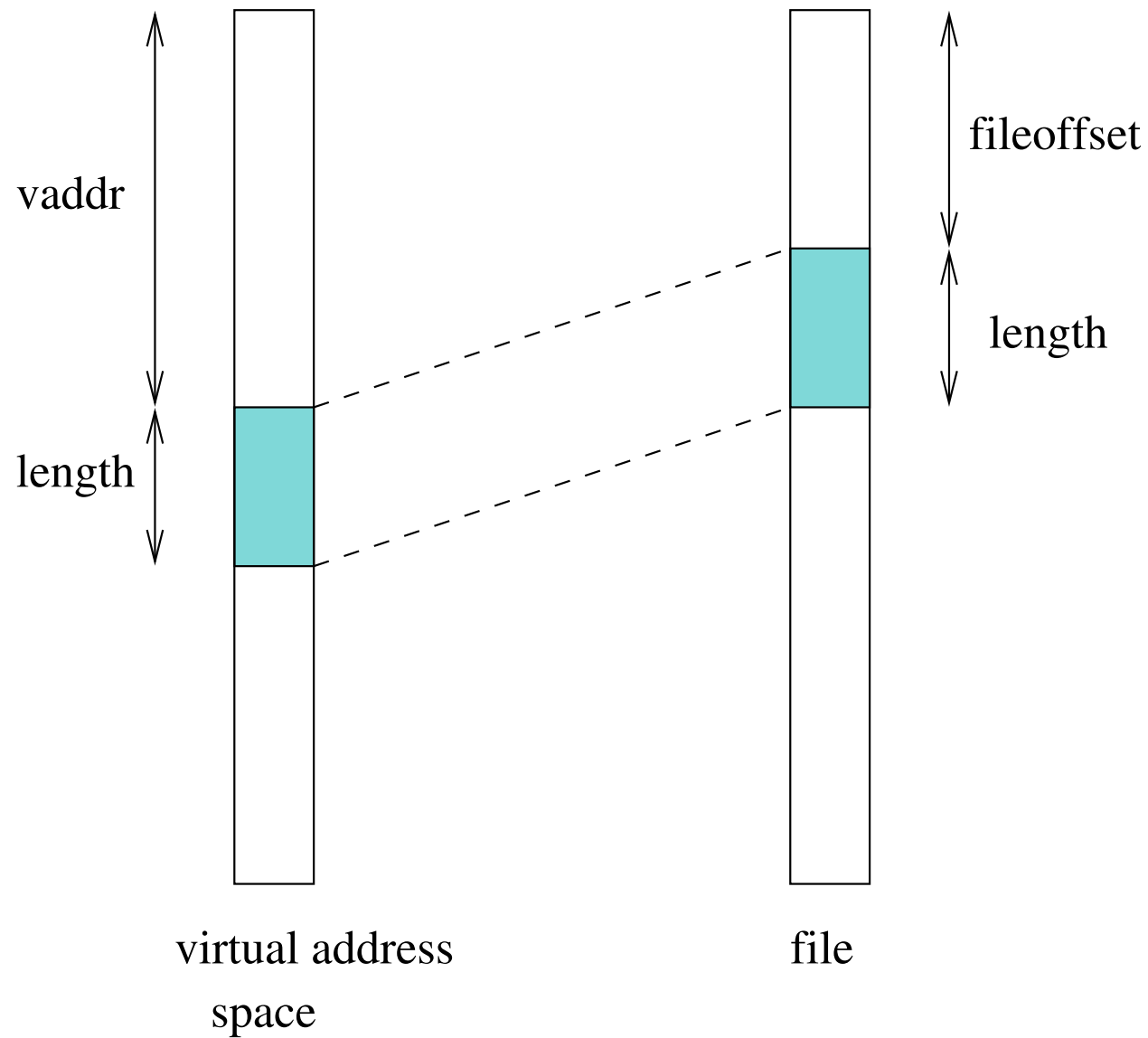
Memory-mapping is an alternative to the read/write file interface.

---

---



## 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

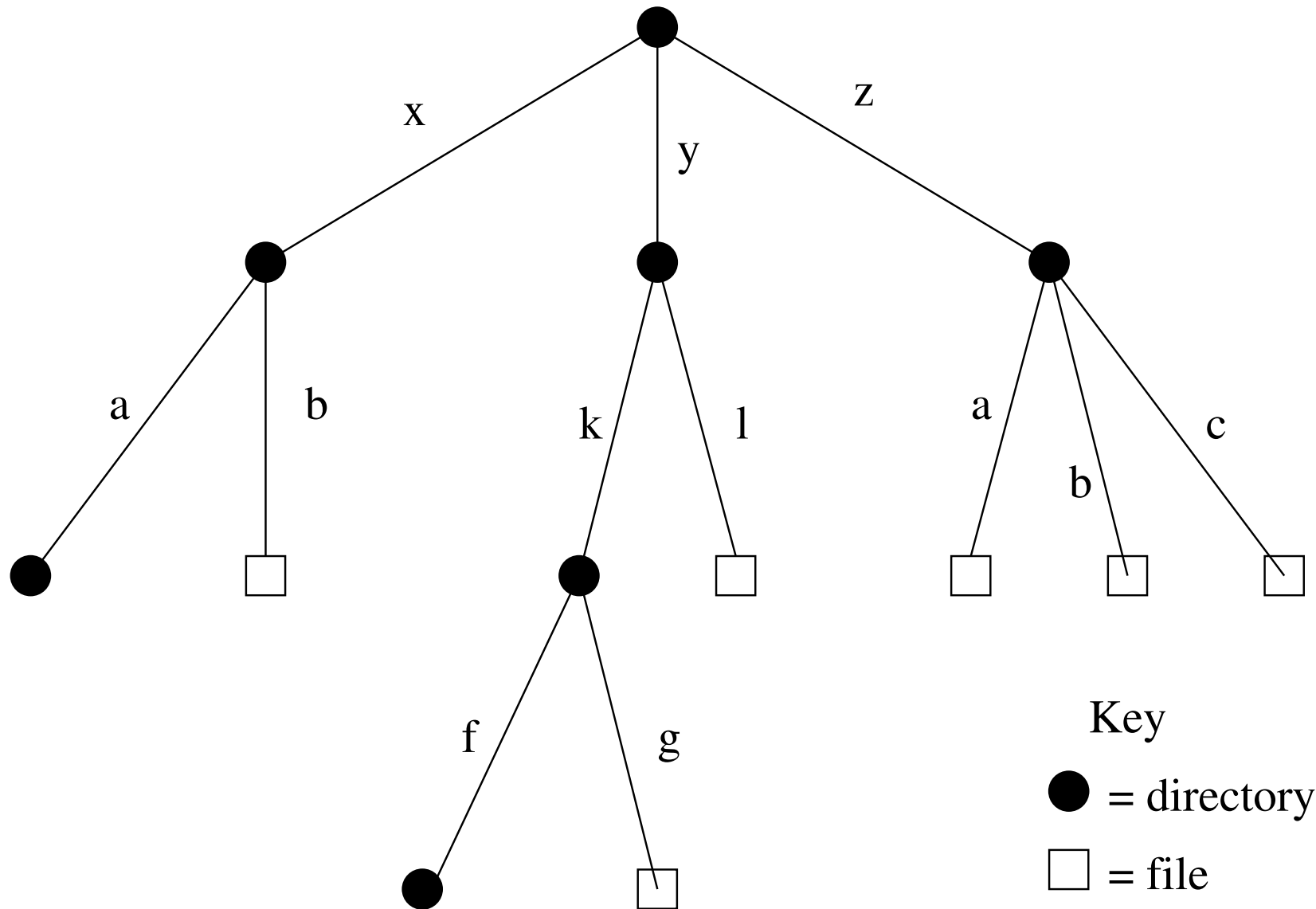
## Memory Mapping Concurrency Semantics

- what should happen if a memory mapped file is updated?
  - by a process that has `mmap`d 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

- flat namespace
    - file names are simple strings
  - hierarchical namespace
    - directories (folders) can be used to organize files and/or other directories
    - directory inclusion graph is a tree
    - pathname: file or directory is identified by a *path* in the tree
- Unix:** `/home/kmsalem/courses/cs350/notes/filesys.ps`
- Windows:** `c:\kmsalem\cs350\schedule.txt`

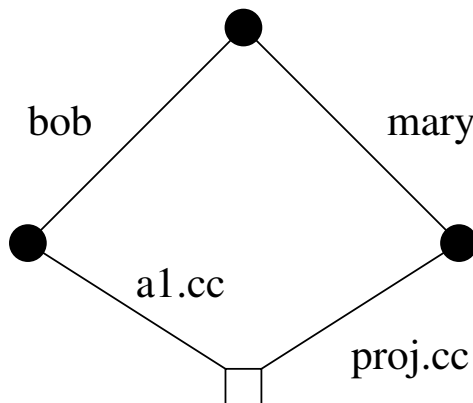
## Hierarchical Namespace Example



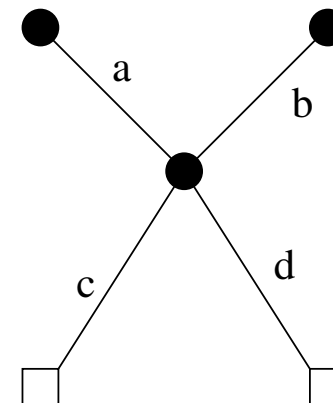
## Acyclic File Namespaces

- directory inclusion graph can be a (rooted) DAG
- allows files/directories to have more than one pathname
  - increased flexibility for file sharing and file organization
  - file removal and some other file system operations are more complicated
- examples:

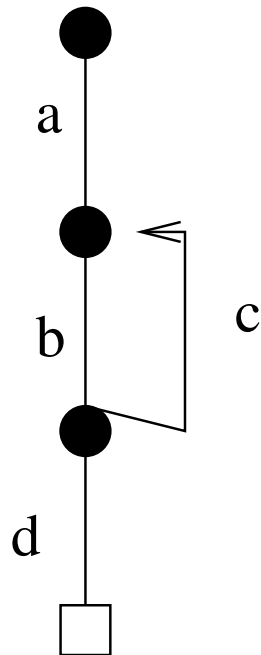
Rooted Acyclic Namespace



An Unrooted DAG



## General File Namespaces



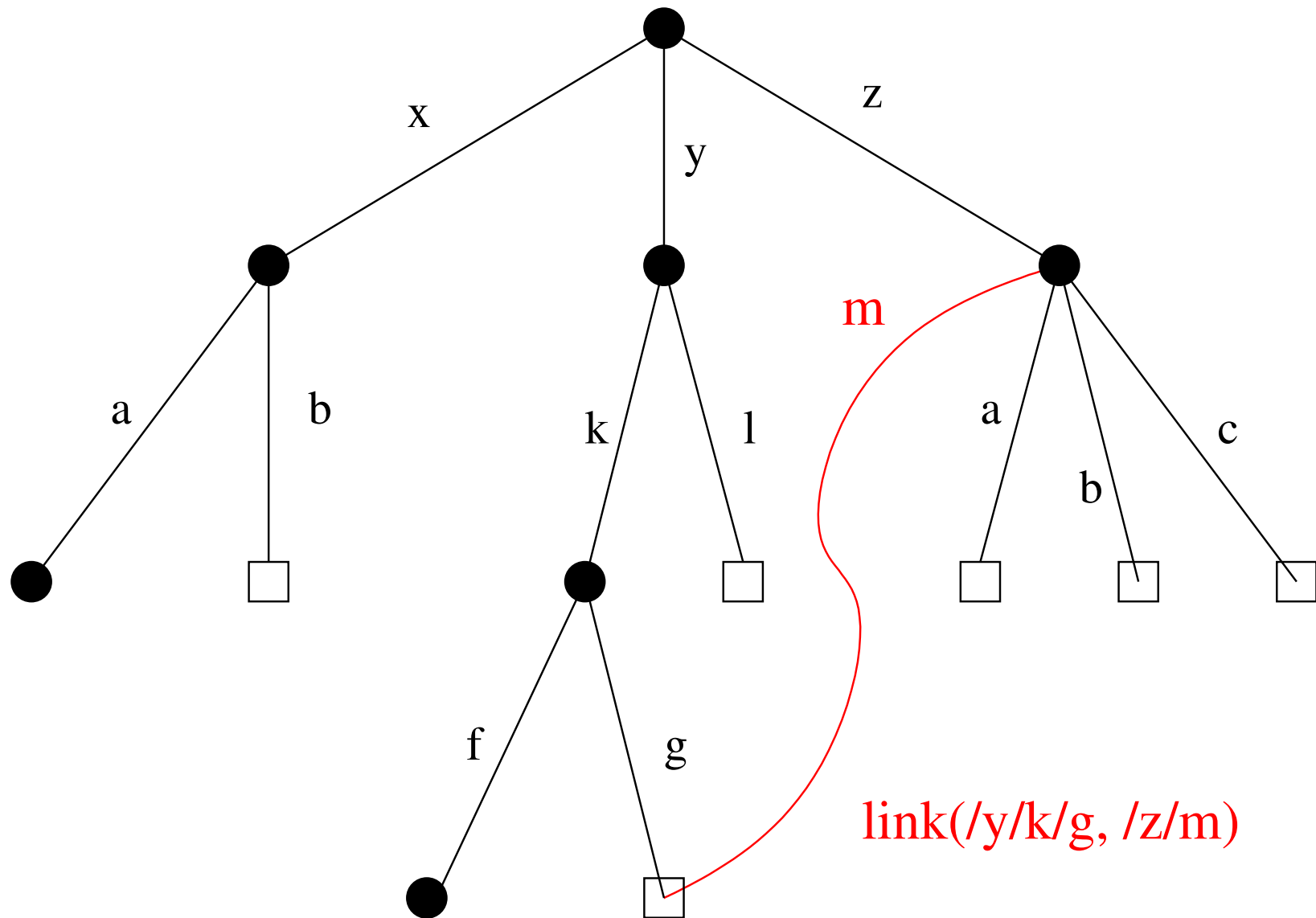
- no restriction on inclusion graph (except perhaps that it should have a designated root node)
- maximum flexibility
- additional complications, e.g.:
  - reference counts are no longer sufficient for implementing file deletion
  - pathnames can have an infinite number of components

## File Links

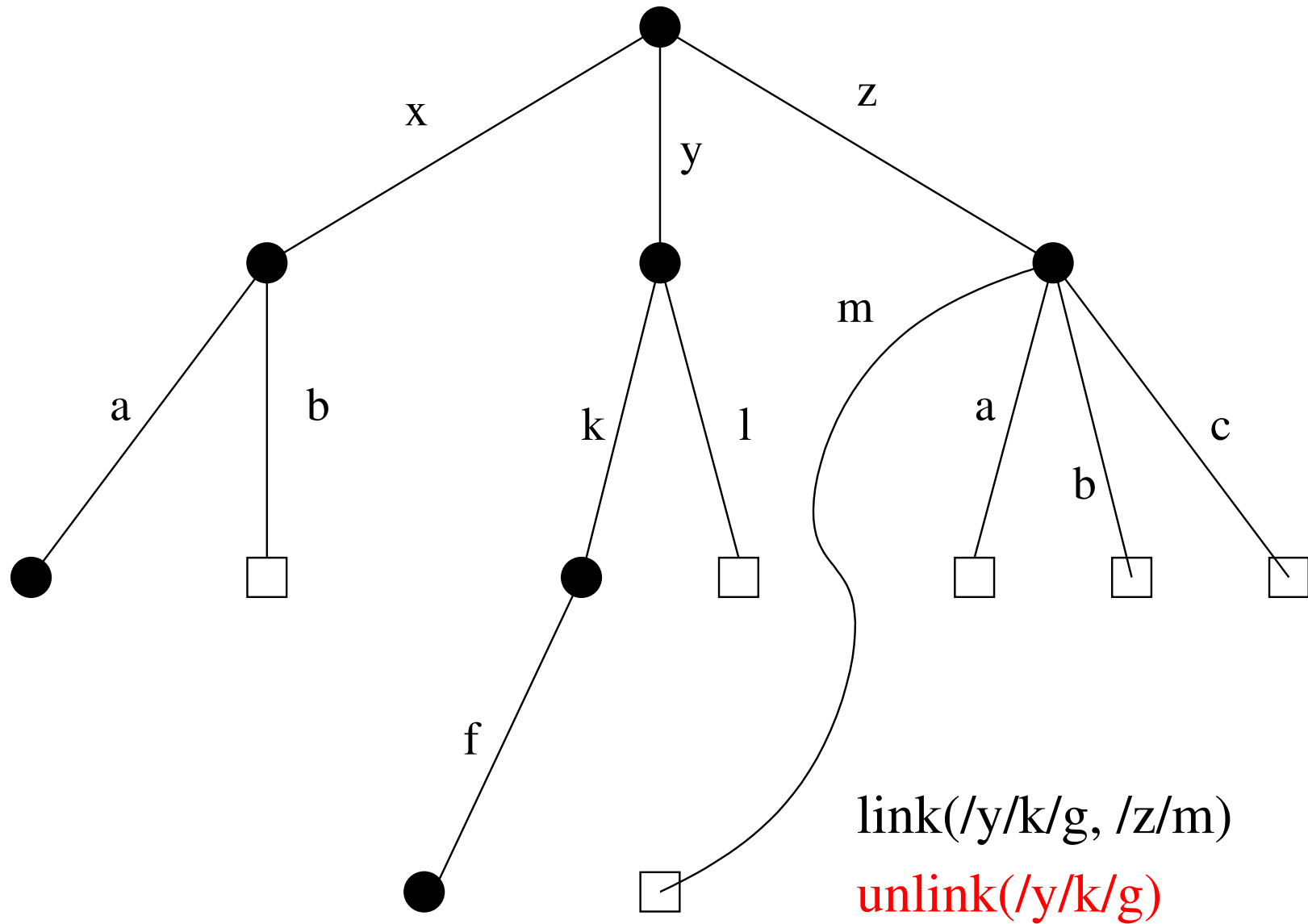
- typically, a new file or directory is linked to a single “parent” directory when it is created. This gives a hierarchical namespace.
- another mechanism can then be used to create additional links to existing files or directories, introducing non-hierarchical structure in the namespace.
- hard links
  - “first class” links, like the original link to a file
  - *referential integrity* is maintained (no “dangling” hard links)
  - scope usually restricted to a single file system
  - Unix: hard links can be made to files, but not to directories. This restriction is sufficient to avoid cycles. (Why?)
- soft links (a.k.a. “symbolic links”, “shortcuts”)
  - referential integrity is *not* maintained
  - flexible: may be allowed to span file systems, may link to directories and (possibly) create cycles



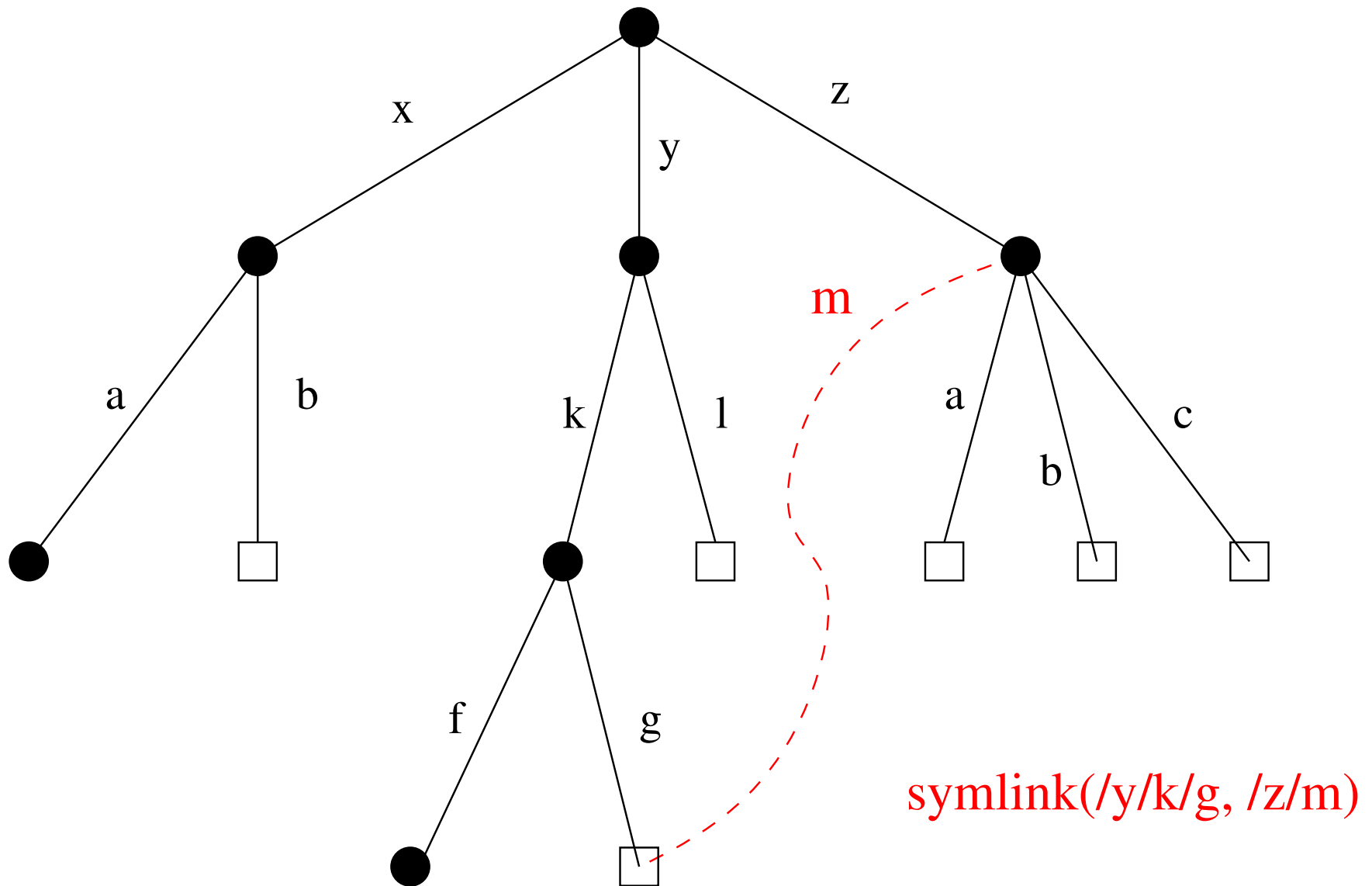
## Hard Link Example (Part 1)



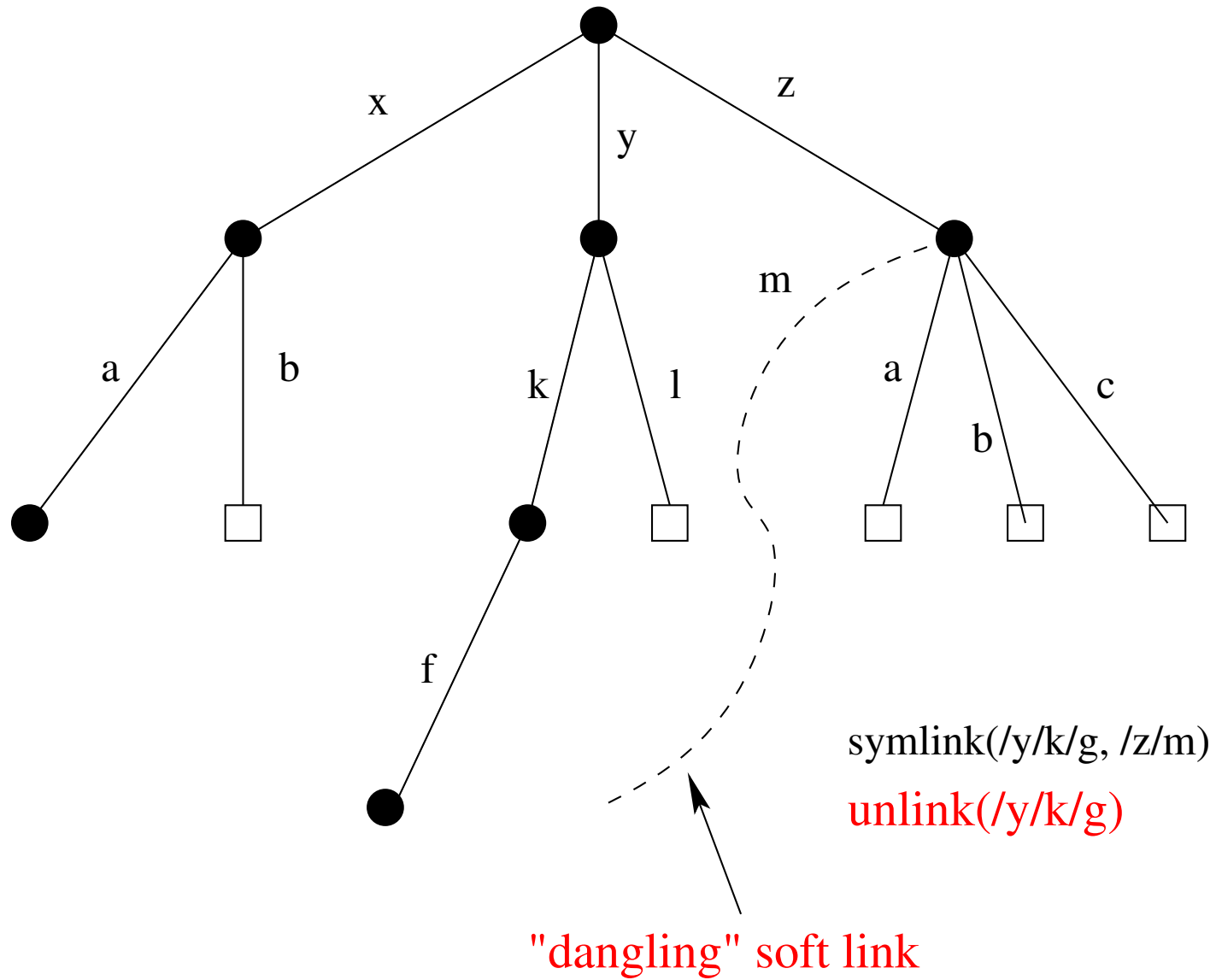
## Hard Link Example (Part 2)



## Soft Link Example (Part 1)



## Soft Link Example (Part 2)



## 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:

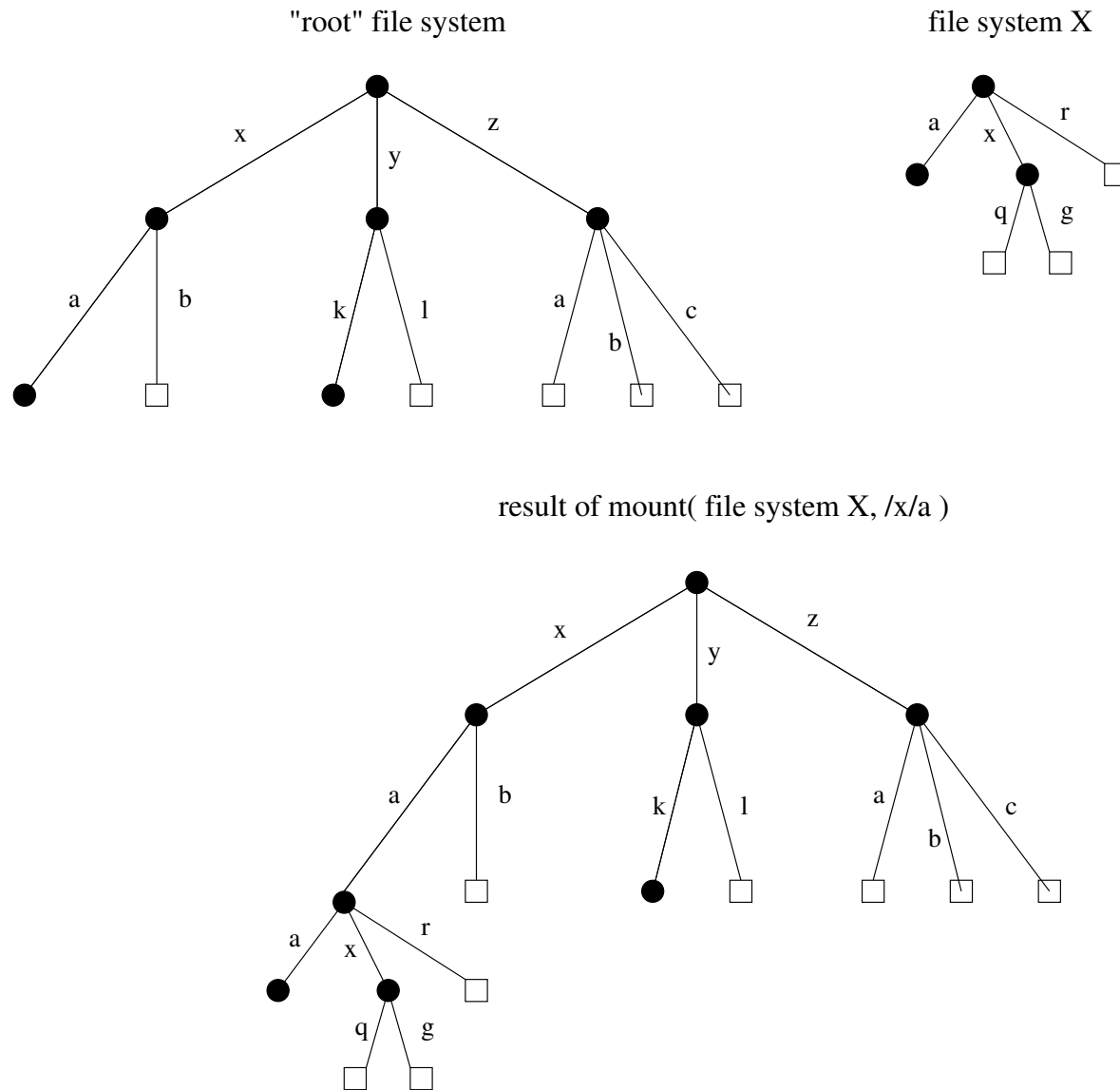
**DOS:** use two-part file names: file system name,pathname

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

**Unix:** merge file graphs into a single graph

– Unix `mount` system call does this

## Unix mount Example



## File System Implementation

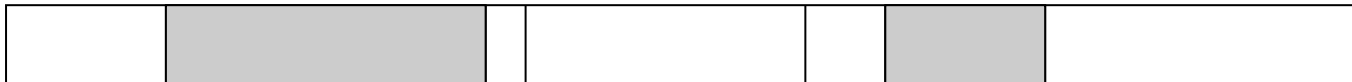
- space management
- file indexing (how to locate file data and meta-data)
- directories
- links
- buffering, in-memory data structures
- persistence

## Space Allocation

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



fixed-size allocation



variable-size allocation



## Space Allocation (continued)

- differences between primary and secondary memory
  - larger transfers are cheaper (per byte) than smaller transfers
  - sequential I/O is faster than random I/O
- both of these suggest that space should be allocated to files in large chunks, sometimes called *extents*

## File Indexing

- in general, a file will require more than one chunk of allocated space (extent)
- 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

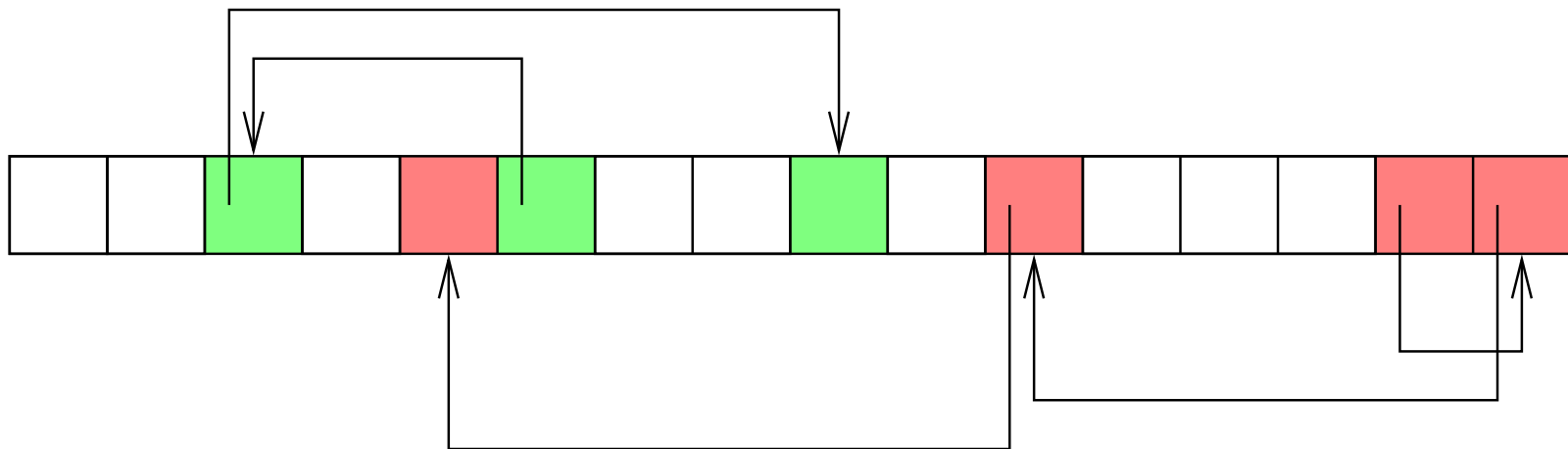
**external chaining:** DOS file allocation table (FAT), for example

- like chaining, but the chain is kept in an external structure

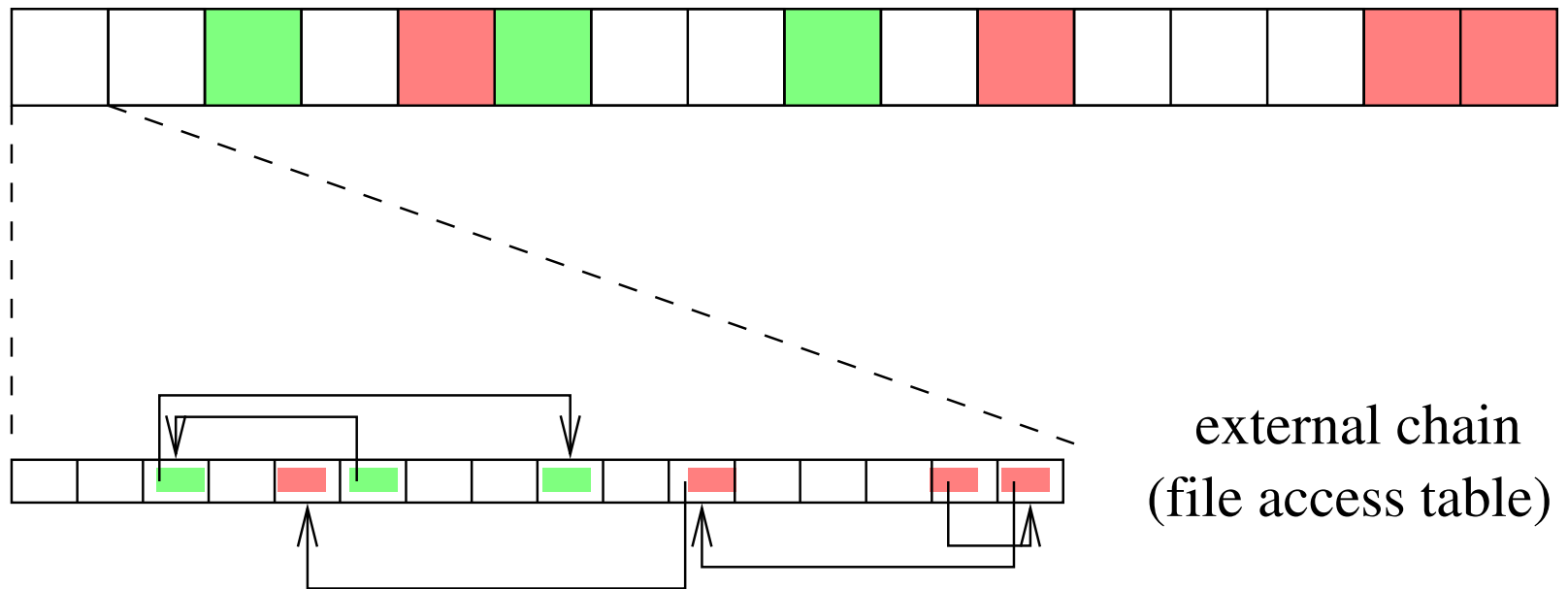
**per-file index:** Unix i-node and NachOS FileHeader, for example

- for each file, maintain a table of pointers to the file's blocks or extents

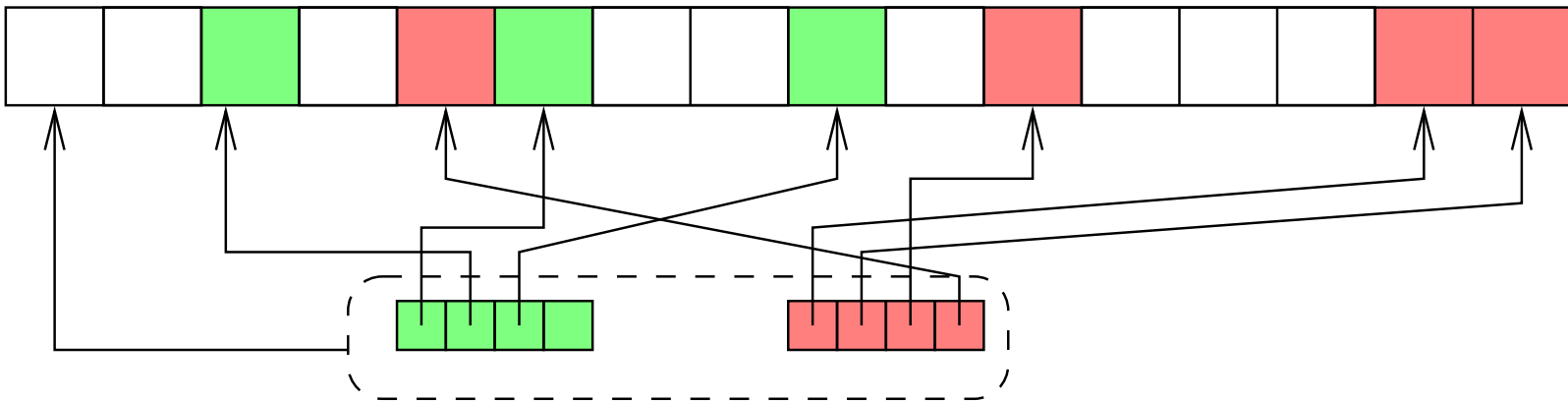
# Chaining



## External Chaining (File Access Table)



## Per-File Indexing



## File Meta-Data and Other Information

- where to store file meta-data?
  - immediately preceding the file data
  - with the file index (if per-file indexing is being used)
  - with the directory entry for the file
    - \* this is a problem if a file can have multiple names, and thus multiple directory entries

## Unix i-nodes

- an i-node is a particular implementation of a per-file index
- 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

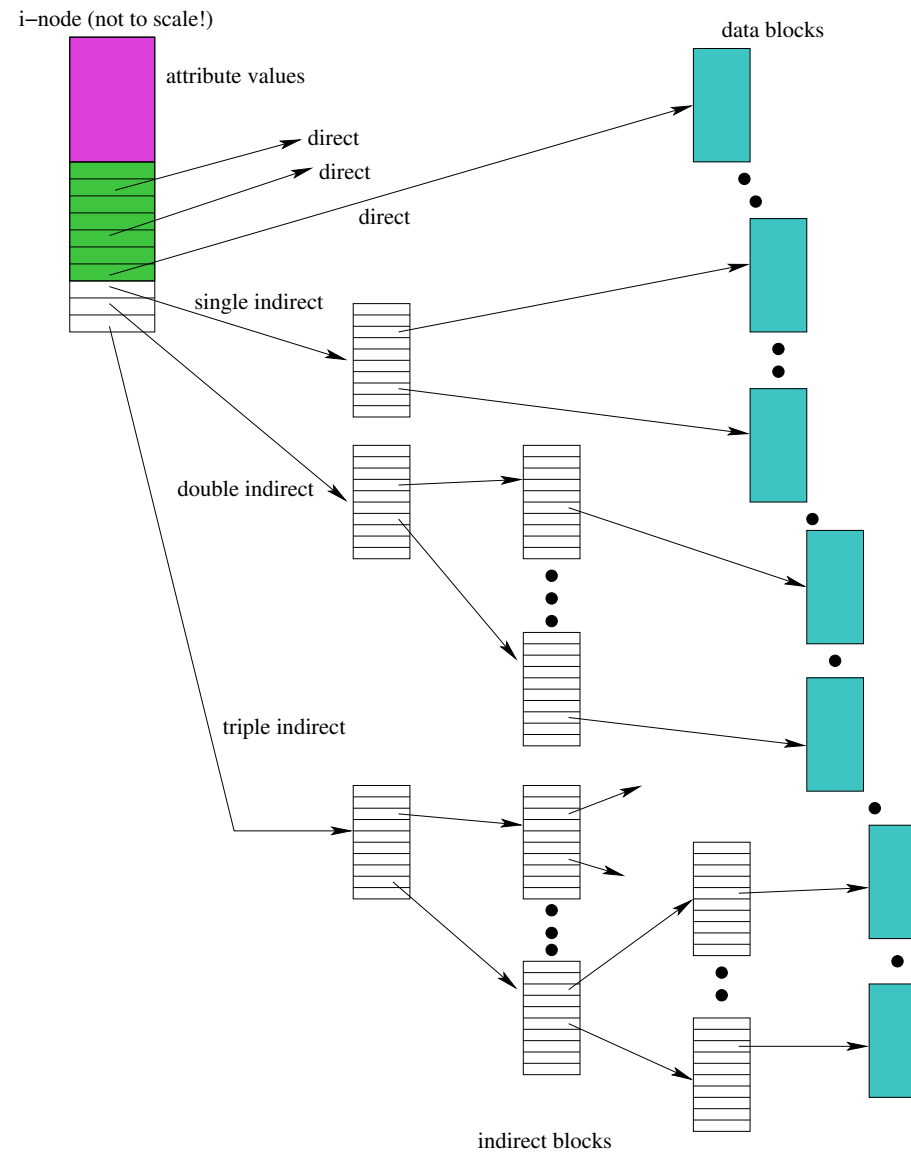
**direct block pointers:** approximately 10 of these

**single indirect block pointer**

**double indirect block pointer**

**triple indirect block pointer**

## i-node Diagram





## NachOS FileHeader

```
#define NumDirect ((SectorSize-2*sizeof(int))/sizeof(int))
class FileHeader {
    public:
        // methods here
    private:
        int numBytes;        // file size in bytes
        int numSectors;      // file size in sectors
        int dataSectors[NumDirect]; // direct pointers
}
```

## 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)
  - a file “locator”
    - \* location of the first block of the file, if chaining or external chaining is used
    - \* location of the file index, if per-file indexing is being used
- A directory can be implemented like any other file, except:
  - interface should allow reading of records (can be provided by a special system call or an library)
  - file should not be writable directly by application programs
  - directory records are updated by the kernel as files are created and destroyed

## Implementing Hard Links (Unix)

- hard links are simply directory entries

- for example, consider:

```
link (/y/k/g, /z/m)
```

- to implement this:
  - create a new entry in directory /z
    - \* file name in new entry is m
    - \* file locator (i-number) in the new entry is the same as the i-number for entry g in directory /y/k

## Implementing Soft Links (Unix)

- soft links are implemented as a special type of file

- for example, consider:

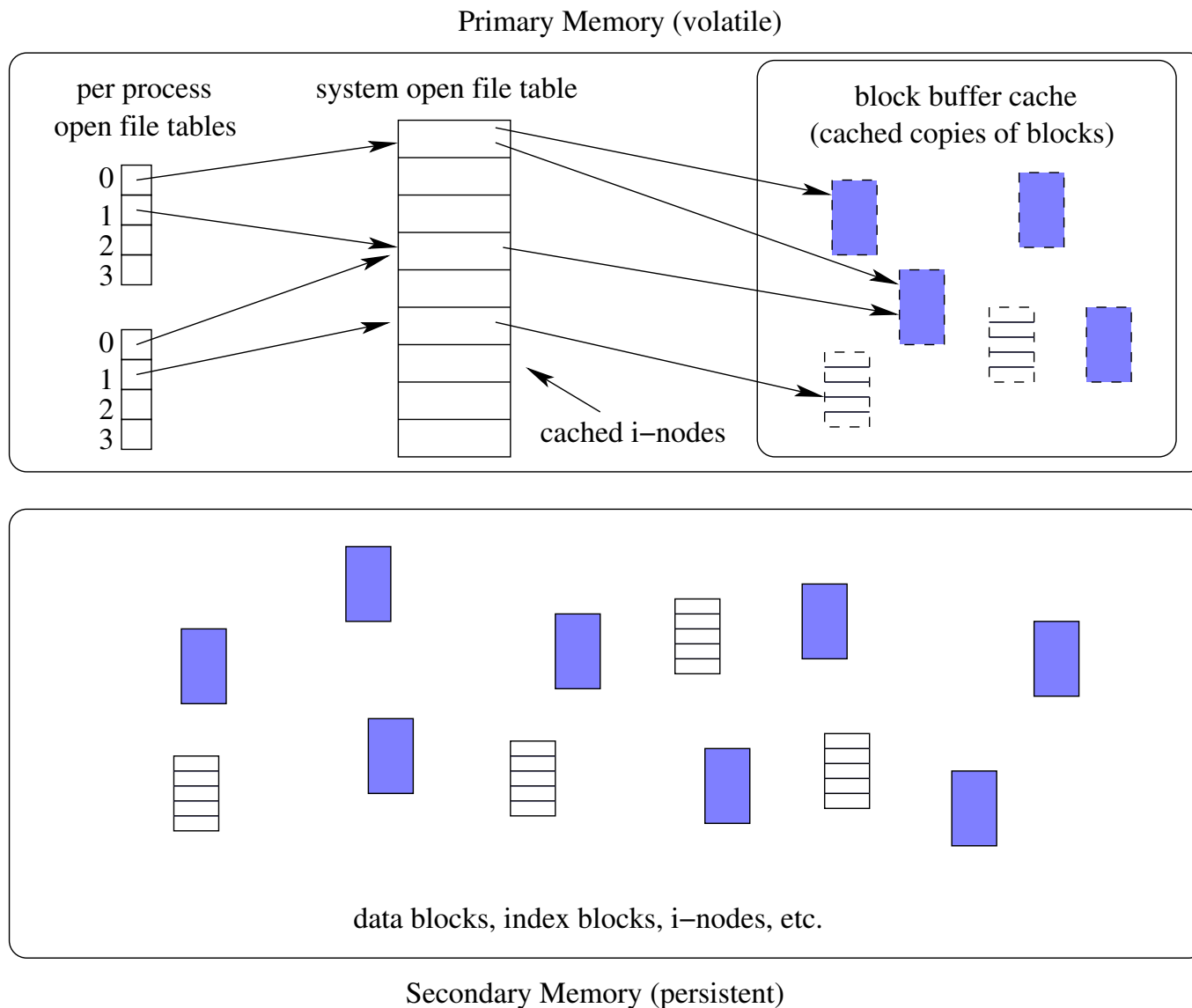
```
symlink (/y/k/g, /z/m)
```

- 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.

## File System Meta-Data

- file system must record:
  - location of file indexes or file allocation table
  - location of free list(s) or free space index
  - file system parameters, e.g., block size
  - file system identifier and other attributes
- example: Unix *superblock*
  - located at fixed, predefined location(s) on the disk
- example: NachOS free space bitmap and directory files
  - headers for these files are located in disk sectors 0 and 1

## Main Memory Data Structures



## A Simple Exercise

- Walk through the steps that the file system must take to implement `Open`.
  - which data structures (from the previous slide) are updated?
  - how much disk I/O is involved?

## 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