CS 552
File Systems

Richard West

(Based on my notes from CS410)

File Types

- **Regular files**, which contain data (either text or binary form)
- **Directory files**, which contain names of other files and pointers to information about them
- **Character files** – used for character devices (such as terminal devices e.g., /dev/tty1 for 1st console device)
- **Block special files** – these refer to block devices such as disks e.g., /dev/hda on a Linux system
- **FIFOs** – a.k.a. named pipes (used for IPC)
- **Sockets** – these file types are used for communication between processes that may reside on different machines, connected via a network
- **Symbolic links** – these are aliases (alternatives names) that link to a specific file accessible via another name

File Access Permissions

- The **st_mode** member of the **stat** structure, that is filled in by a call to **stat**, contains the access permission bits for a file
  - Bits are defined in **<sys/stat.h>**
  - **S_IRUSR, …, S_IWGRP,…, S_IXOTH**
  - **S_ISUID** – set-user-ID bit
  - **S_ISGID** – set-group-ID bit
  - **S_ISVTX** – saved text (a.k.a. “sticky bit”)
    - More on the latter “sticky bit” later

File Access Permissions

- **chmod** can be used to change/set file permissions
- Whenever we want to open a file we must have execute permissions in each directory mentioned in the pathname passed to **open**
  - Implicitly, we must have execute permissions on the current directory, even if it is not specified
  - e.g., to open **/var/log/messages** we need execute permissions on **/, /var**, and **/var/log**
  - Then, we need appropriate permissions on **messages** depending on how we're opening the file

Directory Access Permissions

- NOTE: read-permission on a directory lets us read its entries i.e., access all filenames contained within

- Execute permission lets us pass through a directory when it is a component of a pathname we're trying to access
- Question? What directory permissions do we need to create a new file?
- Answer: We need write and execute permissions in the directory, since **`creat`** is an instance of **open**

Filesystems

- Every **file** in a UNIX-based filesystem has a corresponding **i-node**
  - The i-node contains (attribute) information about a file
    - Type
    - Access permission bits
    - Size
    - Pointers to data blocks (if storage is allocated)
    - Etc.
- A **directory entry** consists of a filename & i-node number for the file

Filesystems

- Every i-node has a link count that contains the number of directories which point to the i-node
  - When the link count reaches 0 the file can be deleted from the filesystem
  - Data blocks that hold the contents of the file can be freed

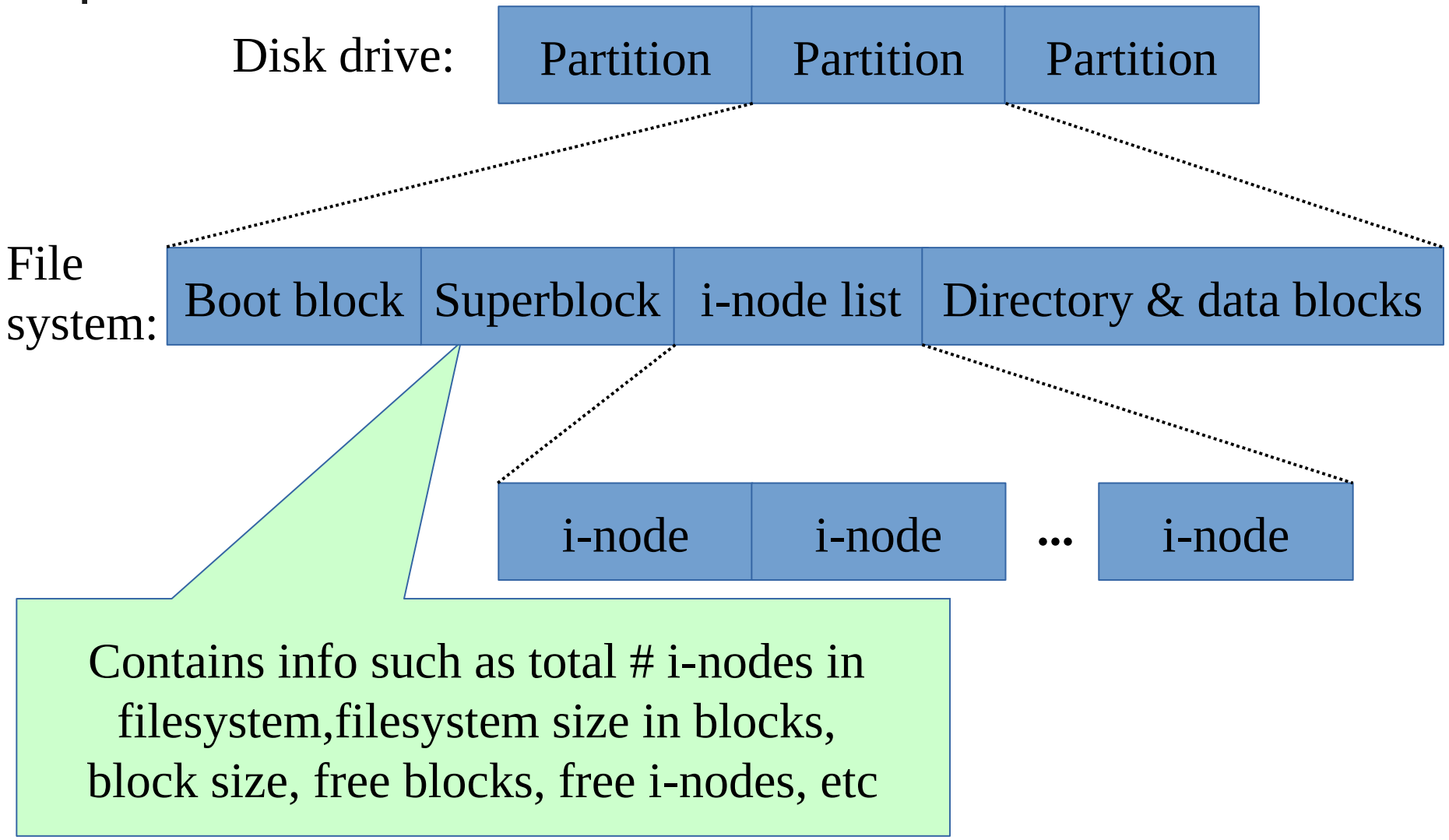
Linking to a File

- A link to a file is created using the **`link()`** function:

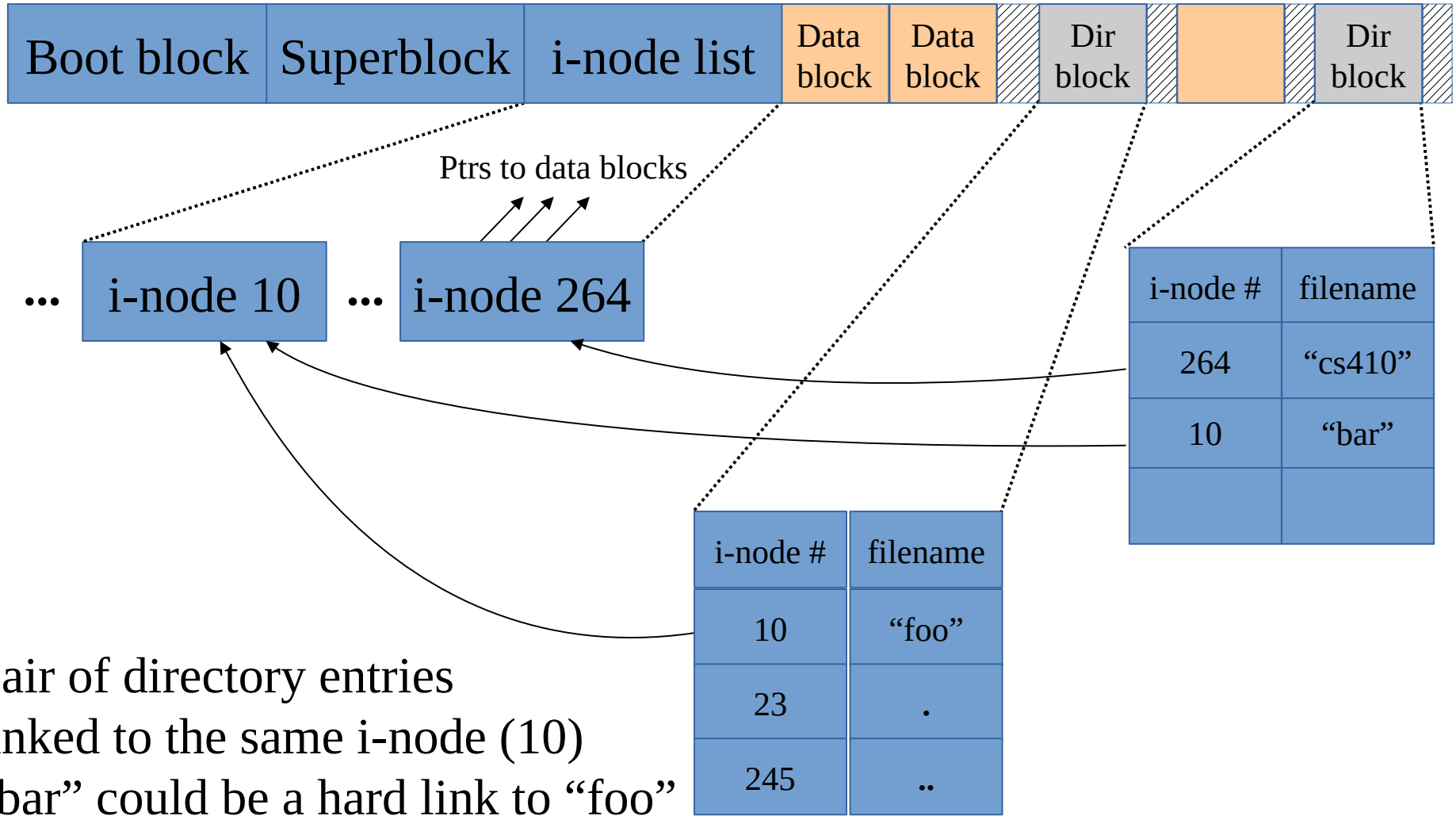
```
int link(const char *current_path, const char
  *new_path);
```

- **`link()`** creates a new directory entry (new_path) that references an existing file (defined by current_path)
- The creation of the new directory entry & the increment of the link count must be atomic
  - Most UNIX-based systems require both pathname arguments to **`link()`** to refer to files in the same filesystem

Filesystem Layout



Filesystem Partition



Pair of directory entries
linked to the same i-node (10)
“bar” could be a hard link to “foo”

Linking to a File

- A link to a file is removed (and, hence, the corresponding directory entry is removed) using **`unlink()`**
  - **`int unlink(const char *pathname);`**
- **`unlink()`** decrements a file's link count (stored in its i-node)
- When a file is closed, the kernel checks the number of processes that have the file open
  - If this count reaches 0 then the kernel checks the link count
    - If the link count = 0, the file's contents are deleted from the filesystem

Linking to a File

- Question: what happens when we do the following:
  - **open** file;
  - **unlink** file;
  - **sleep**(n seconds);
  - **exit** process;

- Answer: the file is not deleted until the process terminates (assuming no other processes have the file open)
  - The process has access to the (open) file until it terminates even though the file is no longer visible to others in the filesystem after the call to **unlink** brings the link count to 0.

Hard Links

- A link to a file can be created by the shell command `**ln**`
  - e.g., **$ln foo bar** // makes bar a link to foo
  - This is a hard link
  - If we examine the contents of foo and bar

    e.g., **$cat foo; cat bar**

    they should be the same

Hard Links

- e.g., **`$cat > foo`**

hello

// Enter text into file **`foo`** and hit newline + ctrl-D (EOF)

`$ln foo bar`

// **`bar`** now links to **`foo`**

`$cat >> foo`

world

// Append text to **`foo`** and hit newline + ctrl-D

`$cat foo` now reads:

hello

world

`$cat bar` also reads:

hello

world

- **`$rm foo`** removes the directory entry for file **`foo`** but **`bar`** remains
  - i.e., the file still exists because the i-node link count is non-zero

Symbolic Links

- In contrast to a hard link, we can establish a symbolic link
- Symbolic links are used to move a file or directory hierarchy to some other location in a filesystem
  - e.g., at the shell prompt:

```
$ln -s /path/to/filename symbolic_filename
```

- If the filename exists we can look at the contents via the symbolic_filename
  - e.g., **`$cat symbolic_filename`**
- If we remove filename the symbolic link still exists but the symbolic_filename can no longer access the contents of filename

Symbolic Links

- The function **`symlink()`** is available to programmers and operates similar to **`link()`**

- Creating a symbolic link to an existing file does not increase the link count (in the i-node) to the existing file

Reading Directories

- Writing to a directory file is reserved for the kernel
- Users can create new files within a directory, as well as remove them, assuming they have valid permissions

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir(const char *pathname);
        // returns ptr if OK, NULL on error
struct dirent *readdir(DIR *dp);
        // returns ptr if OK, NULL @ end of directory or error
```

Reading Directories

- A directory entry structure is system dependent but contains at least the filename and i-node number

- Repeated calls to **`readdir()`** read success entries in the directory, until the end of the directory is reaches (marked by a **`NULL`** returned to **`readdir()`**)

Forcing File Data to (Disk) Storage

- Some applications, such as databases, require updates to files on disk to always see the latest values
  - If the system crashed, we need to be sure the filesystem contents are in a consistent state
- **`read()`** and **`write()`** operations force data to be read/written from/to kernel buffers before being transferred from/to the I/O device such as a disk
- The kernel uses a buffer cache to store data copied by **`write()`** system calls from user-level to kernel-level
- A delayed write occurs in the kernel, at some convenient time, to write blocks from the buffer cache to the I/O device

Forcing File Data to (Disk) Storage

- **`sync()`** and **`fsync()`** force consistency of the filesystem with the contents of the buffer cache

`void sync(void);` // queue all modified blocks in the buffer cache for writing to the I/O device

`int fsync(int fd);` // force blocks for a given file to the I/O device. Returns 0 on success, -1 otherwise

File IO

- Five basic functions for file IO are:
  - **`open, read, write, lseek, close`**
- These functions are unbuffered IO routines
  - They are syscalls that have “immediate” effect on the file (which may be a device file) that they are associated with
    - i.e., there is no buffering done in the user process, although the kernel may introduce buffering of the data

File Descriptors

- The kernel refers to all open files by file descriptors
  - File descriptors are non-negative integers that are returned to a process when a file is created or opened for use
- By convention the UNIX shell associates descriptors 0, 1 and 2 with standard input, output and error, respectively
- POSIX.1 standard defines the constants **`STDIN_FILENO, STDOUT_FILENO and STDERR_FILENO`** (in **`<unistd.h>`**)
- **`OPEN_MAX`** is the max number of open files a process can have (set in **`<limits.h>`**)

Opening a File

- A file is opened or created using the **open** function

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open (const char *pathname,
  int oflag, ... /* mode_t mode */);
(RETURNS: file descriptor if OK, -1 on error)
```

- The 3rd argument "..." indicates a variable argument list (where the # and type of arguments may vary)

Opening a File

- The 3rd argument is only used when a new file is created and indicates the mode bits for the file (shown in comments)
  - e.g., **S_IRWXU** → read, write, execute permissions by owner/user
  - **S_IRUSR** → user has read permission
  - **S_IWUSR** → user has write permission
  - **(S_IXOTH, S_IWGRP, etc)**

- The mode (i.e., permission) bits restrict the capabilities or access rights on a file, thereby affecting future operations on a file, irrespective of the **oflag** value

Opening a File

- With open:
  - **`pathname`** = name of file to open/create
  - **`oflag`** = bitwise logical OR of a set of option flags, defined in <**`fcntl.h`**>
    - Must include one of: **`O_RDONLY`**, **`O_RDWR`**, **`O_WRONLY`** (reading only, reading/writing, writing only)
    - Plus any of: **`O_APPEND`**, **`O_CREAT`**, **`O_NONBLOCK`**, etc

File Creation

- Can also be done with the **creat** syscall

```
int creat(const char *pathname,mode_t mode);
RETURNS: fd opened for write-only if OK, -1 on error
```

  - **creat** is equivalent to:

```
open(pathname, O_WRONLY|O_CREAT|O_TRUNC, mode)
```

  - Problem is that file is only opened for writing
  - Can use **open** to create a file with alternative access rights:

e.g.,

```
open(pathname, O_RDWR|O_CREAT|O_TRUNC, mode);
```

Closing a File

- An open file is closed by:

```
#include <unistd.h>
int close (int filedes);
RETURNS: 0 if OK, -1 on error
```

- Termination of a process causes the kernel to automatically close any open files associated with the process, so no explicit call to **close** is required

Accessing Data within a File

- Every file has a current file position, measured as an integer offset (in bytes) from the beginning of the file
- The offset is initially 0 unless the **`O_APPEND`** flag is used when a file is opened
- Read/write ops adjust the current file position (based on the number of bytes read/written)
  - NB: files are read/written from the current file position

Seeking a File

- We can read/write data from a given position using:

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int filedes, off_t offset, int
  whence);
RETURNS: new file offset/position if OK, -1 on
  error
```

whence can be:

SEEK_SET – current file position = **offset** (2nd arg)

SEEK_CUR – current file position += **offset**

SEEK_END – current file position = end of file + **offset**

Seeking a File

- **`lseek`** records the **`current file position`** within the kernel (for a given process)
  - No IO is performed

- **`lseek`** can only be used on regular files that store data (as opposed to pipes, FIFOs, etc – more later!)

File Holes

- Question: what happens if we create a file, fill it with 10 bytes of data and then seek to an offset of 40 bytes to write 10 extra bytes?

- Answer: we get a HOLE of 30 NULL bytes (‘\0’) in the middle of the file
  - File size = 50 bytes

Reading a File

- We can read an open file as follows:

```
#include <unistd.h>
ssize_t read(int filedes, void *buf, size_t
  nbytes);
RETURNS: # of bytes read, 0 if EOF, -1 on
  error
```

  - We may return less than **nbytes** if **EOF** is reached or if we read from a device file (e.g., for a network device) if the device buffers data

Writing a File

```
#include <unistd.h>
ssize_t write(int filedes, const void *buf,
  size_t nbytes);
RETURNS: #bytes written if OK, -1 on error
```

File System Data Structures

- UNIX supports the sharing of open files between different processes
- 3 data structures are used by the kernel to affect file sharing

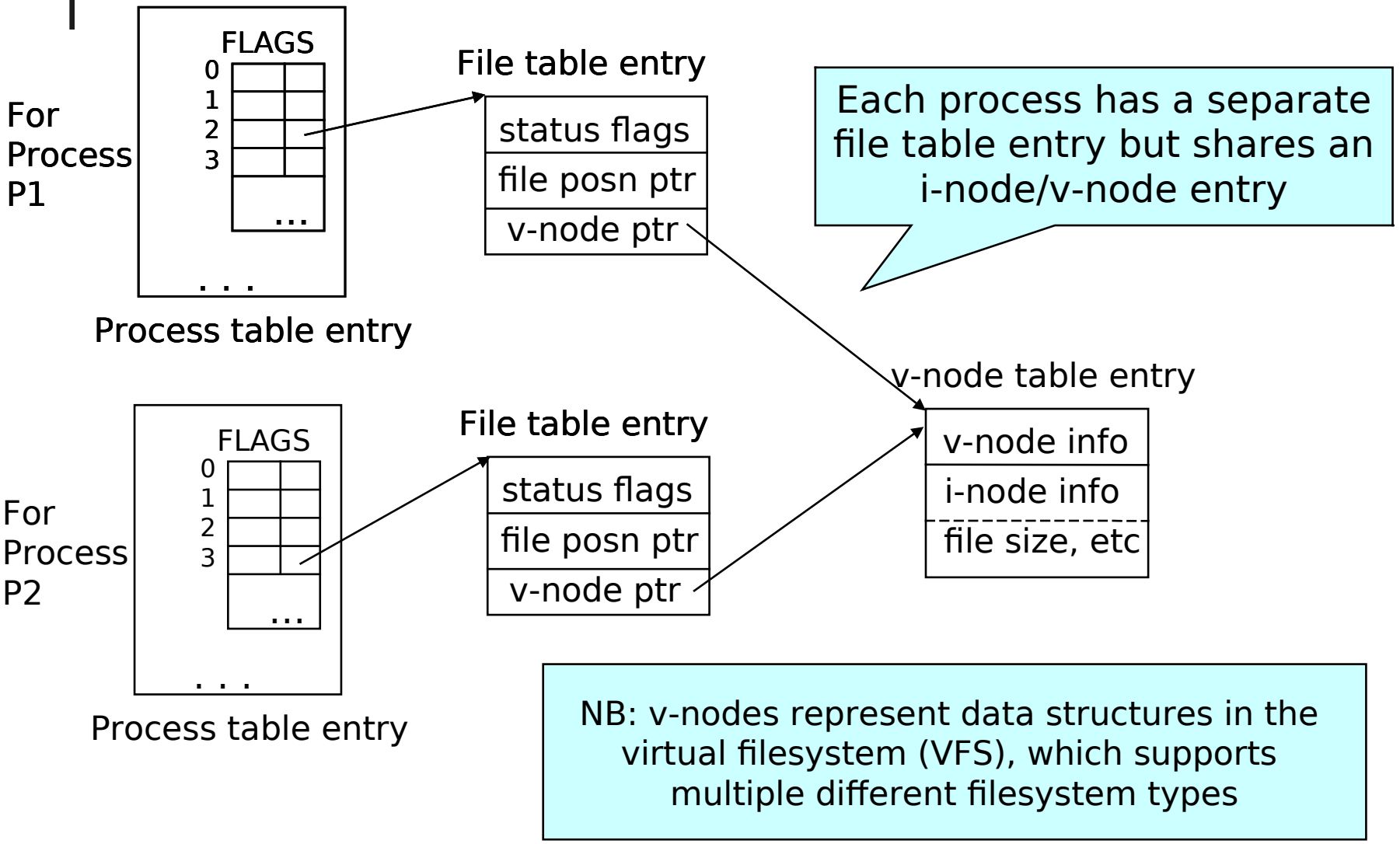
1. Every process has an entry in the process table
   - Each process table entry has a table (or vector) of open file descriptors
   - Each file descriptor has a corresponding set of flags and a pointer to a file table entry
   - NB: Most systems only have the **`FD_CLOEXEC`** flag per fd, defined in **`<fcntl.h>`**. **`FD_CLOEXEC`** flag means close a file descriptor on **`exec`** of a child process. Default is not to close fd in a child - more later!

File System Data Structures

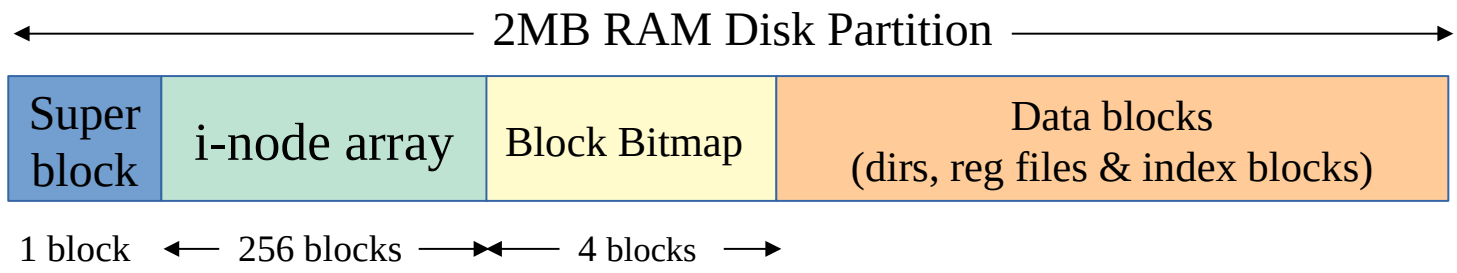
1. The kernel maintains a file table for all open files
   - Each file table entry contains
     - Status flags for the file (when 1st opened: **`O_RDWR`** etc)
     - Current file offset/position
     - A v-node pointer

2. Each open file (or device) has a v-node, containing info about the type of file & pointers to functions that operate on the file
     - The v-node also contains i-node info for the file (more later!)

File Sharing Data Structures



Filesystem Data structures (for DISCOS assignment)

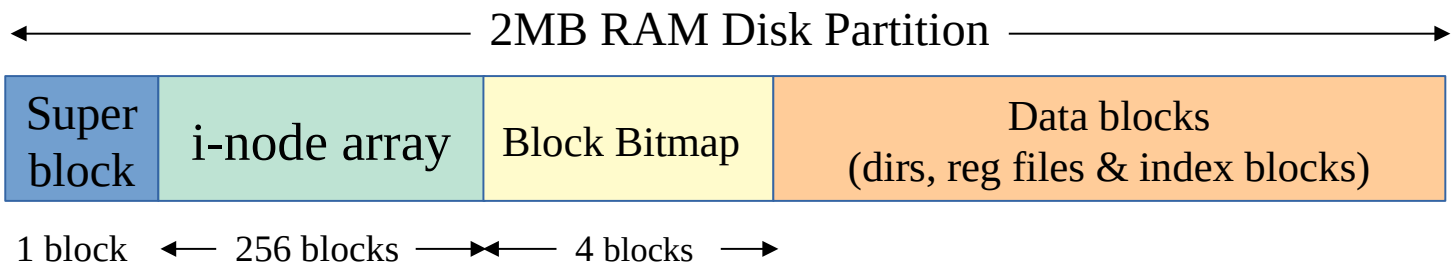


- Ramdisk block – 256 bytes
- Index node (a.k.a. i-node) – 64 bytes
- Block bitmap – 1 bit per data block (e.g., 0 = free, 1 = allocated)

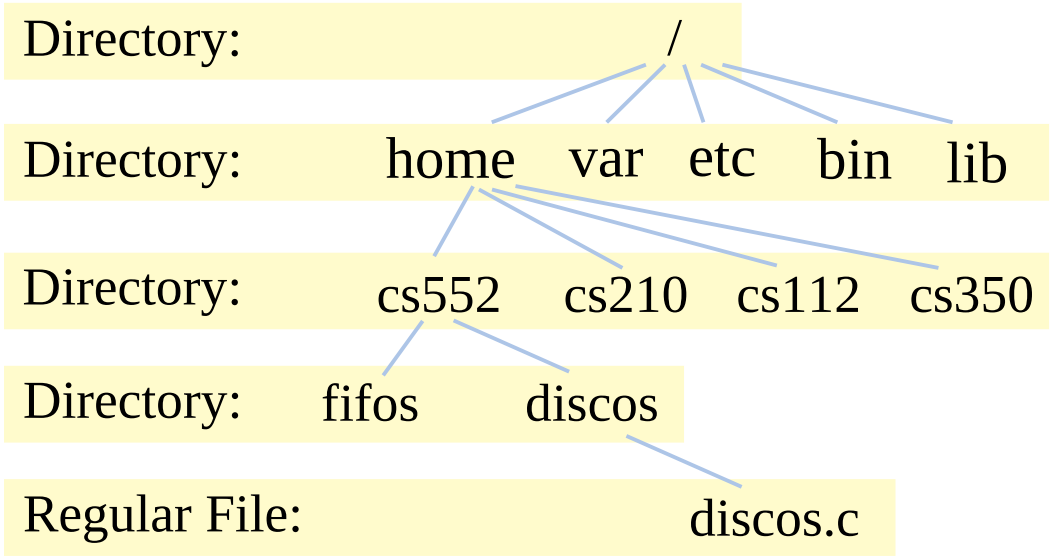
- Regular file – holds arbitrary data (text or binary)
- Directory file – holds directory *entries* for its position in *tree*

- Directory entry – 16-bytes: [filename, i-node #]
- Filename – null-terminated string padded to 14-bytes
- Index node # – 2-byte index into the i-node array

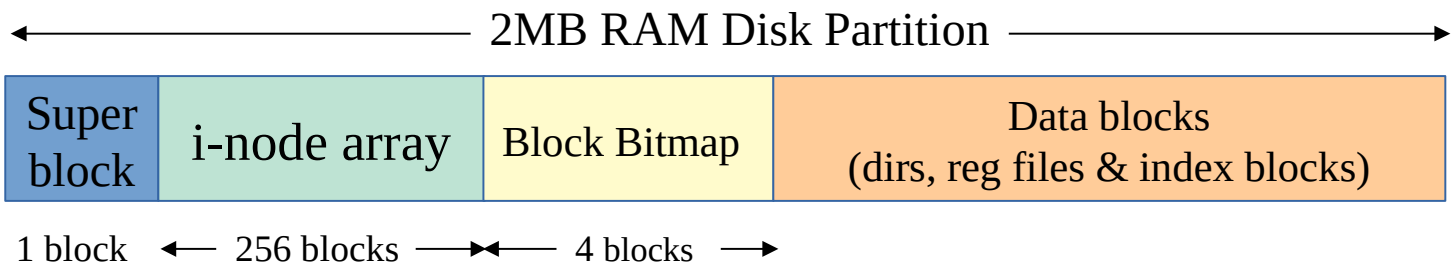
Filesystem Data structures (for DISCOS assignment)



- Example tree:
  - *Pathname – /home*/cs552/discos/discos.c



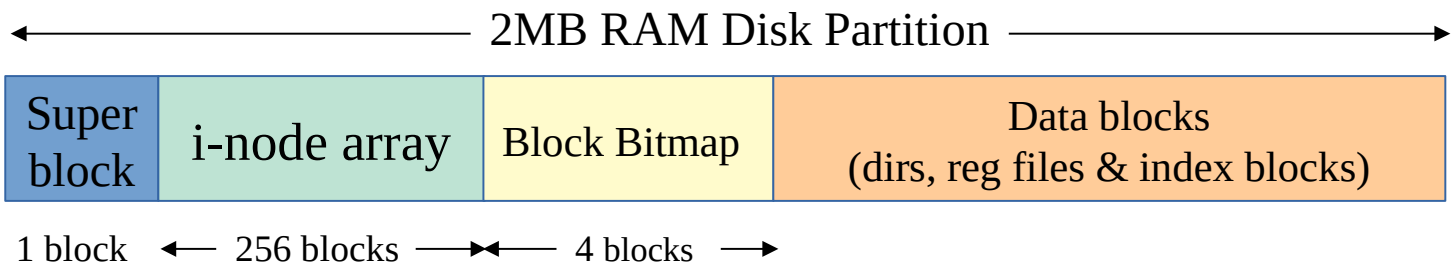
Filesystem Data structures (for DISCOS assignment)



- index node – contains information about a file (one node per file, 64-bytes)
  - For project:
    - *type* – either “dir” or “reg” (4-bytes)
    - *size* – current file size in bytes (4-bytes)
    - *location* – identifies block storing file contents (40-bytes)
    - *access rights* – default=*read-write*, optional=*read-only*, *write-only*

- Superblock – contains meta-information about partition
  - # free blocks
  - # free index nodes
  - Other information of your choosing (e.g., location of 1st data block)

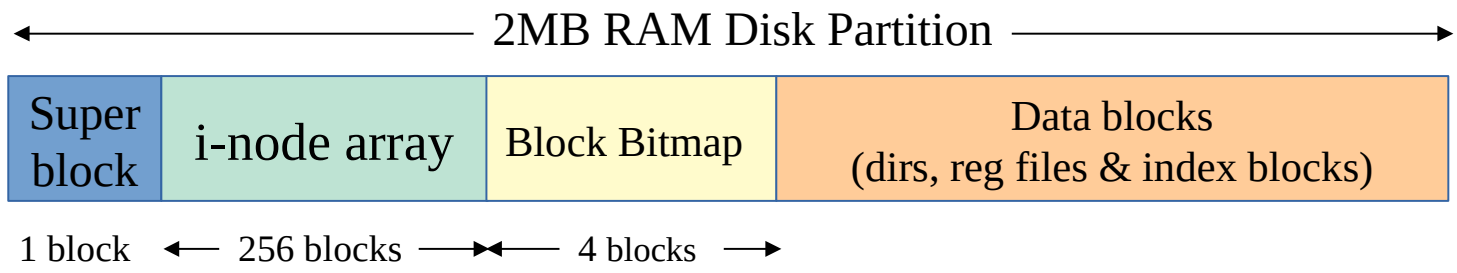
Filesystem Data structures (for DISCOS assignment)



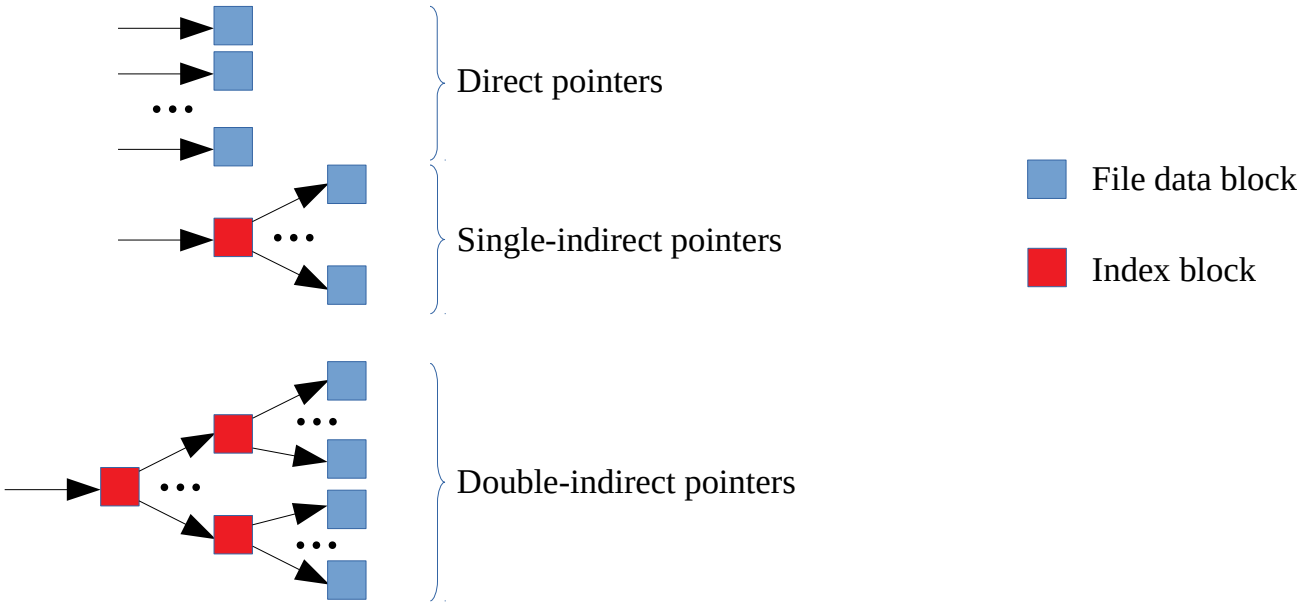
- Filesystem is implicitly a tree rooted with directory named "/" having i-node 0
  - "/" is 1st (zeroth) entry in i-node array
  - Each file is allocated an integer number of disk blocks (no block sharing)
- *location* attribute in i-node:
  - 10 block pointers, 4-bytes per pointer (40-bytes total)
  - 1st 8 pointers – direct block pointers
  - 9th pointer – single-indirect
  - 10th pointer – double-indirect

- Max file size: $256*8 + 256*64 + 256*64^2 = 1067008$ bytes
  - can fit in 2MB partition
- What about maximum number of files?
  - Total i-nodes: 256 blocks*256/64 = 1024 (1023 files discounting "/")

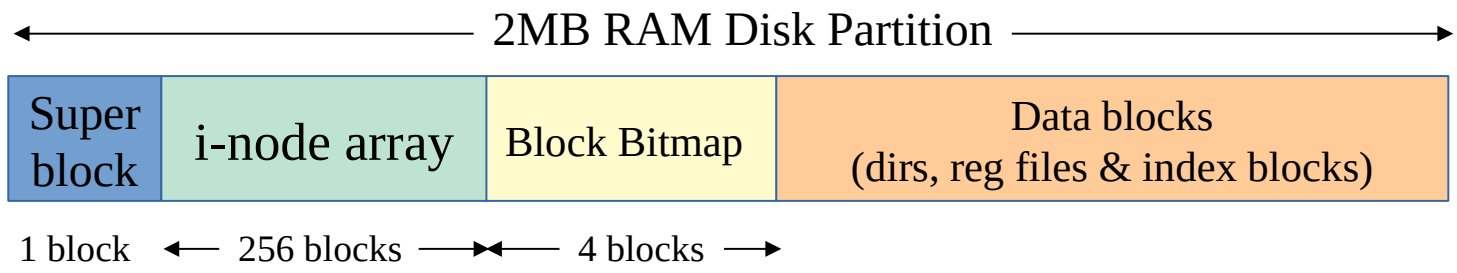
Filesystem Data structures (for DISCOS assignment)



- I-node *location* attribute – Direct, single-indirect, double-indirect block pointers:



Filesystem Data structures (for DISCOS assignment)



- Open files – need a table of open file descriptors
  - Table per process (Linux)
  - Table per thread (if using FIFOS)

- Need to implement file operations:
  - *rd_creat, rd_mkdir, rd_open, rd_close, rd_read, rd_write, rd_lseek, rd_unlink, rd_chmod*

Filesystem Data structures (for DISCOS assignment)