Devices, Input, and Output

Process Interface

- Concept of file underlies interface
  - More about this next
- Enables processes to interact with devices
  - Also kernel structures such as /dev/null and /proc
- Need at least 1 special system call to handle device-specific functions

System Calls: *open*, *close*

- *open* makes file accessible to process
- Form: descriptor = *open*(file, how, . . .)
  - Now process uses descriptor to refer the file
  - If device not ready, process may block or call may return error code
  - Call also checks privileges to ensure user can open the file
- *close* disassociates file from process
- Form: *close*(descriptor)
  - Device driver does any needed clean-up

System Calls: *seek*

- *seek* positions pointer associated with descriptor as instructed
- Form: *seek*(descriptor, where)
  - Read/write pointer repositioned to where
  - Examples: go to arbitrary location in file, position on tape
- Linux: *lseek*(descriptor, offset, whence)
  - whence indicates if offset is from beginning or end of descriptor, or current position of read/write pointer
  - Returns new position on success, –1 on error; but –1 may be valid value
  - Disambiguate using *errno*

System Calls: *seek*

- Linux: *lseek* example

```
external int errno;
. . .
errno = 0;
if (lseek(desc, offset, SEEK_SET) == -1 && errno != 0){
        /* handle error */
}
else{
        /* handle success */
}
```

System Calls: *read*

- Transfers data from descriptor object to memory
- Form: nread = *read*(descriptor, memory address, amount)
  - Reads nread bytes, which is at most amount
  - Returns 0 on end of file, error code on error
- Form: nread = *readv*(descriptor, memory list, list length)
  - Like read, but reads data into multiple memory locations
  - Locations given in memory list; also number of bytes for each
  - Returns number of bytes read, or 0 on end of file, error code on error

System Calls: *write*

- Transfers data from memory to descriptor object
- Form: nbyte = *write*(descriptor, memory address, amount)
  - Outputs nbyte bytes, which is at most amount
  - Returns error code on error
- Form: nbyte = *writev*(descriptor, memory list, list length)
  - Like write, but writes data from multiple memory locations
  - Locations given in memory list; also number of bytes for each
  - Returns number of bytes written, error code on error

Blocking vs. Non-Blocking Read and Write

- Blocking transfer is synchronous
  - So when the next statement is executed, transfer has been completed
- Non-blocking transfer is asynchronous
  - So next statement executed whether or not transfer has been completed
- Two ways to determine when non-blocking transfer completes:
  - Use polling by checking an indicator
  - Use interrupts

Non-Blocking Read and Write

- Process requests interrupt from kernel when transfer completes
  - System call may arrange this; on Linux, it's SIGIO
- Process must arrange to catch interrupt and process it
  - Usually a system call like *handler*(signal, function)
- If process does need to block until transfer is complete, need a system call like *wait*(descriptor, timeout)
  - Blocks until transfer to or from descriptor completes
  - If not completed by timeout, then wake up and continue
- Never modify memory involved in transfer until transfer completes
  - Results are undefined

System Calls: *control*

- Used for device-specific actions
- Form: *control*(descriptor, action, . . .)
  - action is device specific and may require other parameters
- Linux example: make FAT file system read-only:

  attrmask = ATTR_RO;

  ioctl(desc, FAT_IOCTL_SET_ATTRIBUTES, &attrmask)
- Linux example: insert ch into (terminal) input queue:

  toinsert = ch;

  ioctl(desc, TIOCSTI, &toinsert)

Linux Examples

- Make FAT file system read-only:

  attrmask = ATTR_RO;

  ioctl(desc, FAT_IOCTL_SET_ATTRIBUTES, &attrmask)

- Insert ch into (terminal) input queue:

  toinsert = ch;

  ioctl(desc, TIOCSTI, &toinsert)

- Give up role of controlling terminal:

  ioctl(desc, TIOCNOTTY)

File Systems

File Systems

- File: a collection of data
  - *virtual*: how the user (process) sees the file
  - *physical*: how the file is represented to the hardware and operating system.
- Filename: often reflects something about the file, particularly the extension
  - TOPS-20: file names are *name*.*ext*, where *ext* is a three-character extension describing the file; “bas” for BASIC, “for” for FORTRAN, “bli” for BLISS, “obj” for object, “exe” for executable, “txt” for text, and so forth
  - Linux, FreeBSD, and MINIX: the last letter(s) may designate something; “.c” for C source files, “.cc” for C++ source files, “.py” for Python files

Directories

- Files organized into *directories* to make organizing them easier
  - “folders” for Mac, Windows
- Directory contains pairs of (name, location)
  - Location may be a physical location (disk address) or an index into an array containing those locations or any other datum used to locate files
  - Example: in Linux, location is the inode number

Organization of Directories

- Flat (one-level) directories
- Hierarchical directories
- Graph-structured directories

Flat (One-Level) Directory

- All files are in the same, single directory
- Problems:
  - No two files can have the same name
    - To keep users having to worry about collisions, the system could make the user name a component of each file name)
  - To find a file, one must search the whole directory

Hierarchical Directory

- Impose tree structure on directories
  - Typically there is a root directory, then other directories for users, system executables, and other things
- Identifying files: use path name
  - Current working directory: where in the file system the process is currently
  - Absolute path: from root directory
    - Examples: /usr/bin/tcsh, /home/tanz
  - Relative path: from some directory other than the root
    - Examples: a/b/c; ../xyzzy; ./a.out

Graph-Structured Directory

- Basically a hierarchical system, but with the ability to *alias* files across branches
  - Linux, UNIX have this (contrary to popular belief)
- *Direct alias*: one (file) location appears twice (or more) in directories, often with different names
  - In Linux terminology, a hard link
- *Indirect alias*: special type of file containing path name of another file
  - Said to be an indirect alias for the file it names
  - Operating system interpolates the name of the file being aliased on a reference to the indirect alias
  - In Linux terminology, a symbolic link or soft link

Aliasing Issues

- No such thing as a “true” name now
  - You can refer to same file with multiple names
  - For hard links, no way to tell which was the original name
- Deletion: if a file is deleted under one alias, is it inaccessible using the other aliases?
  - *Yes*: must find all other aliases and delete them; very time-consuming
  - *No*: use a *link count* to track how many aliases a file has and don’t delete file until all aliases deleted

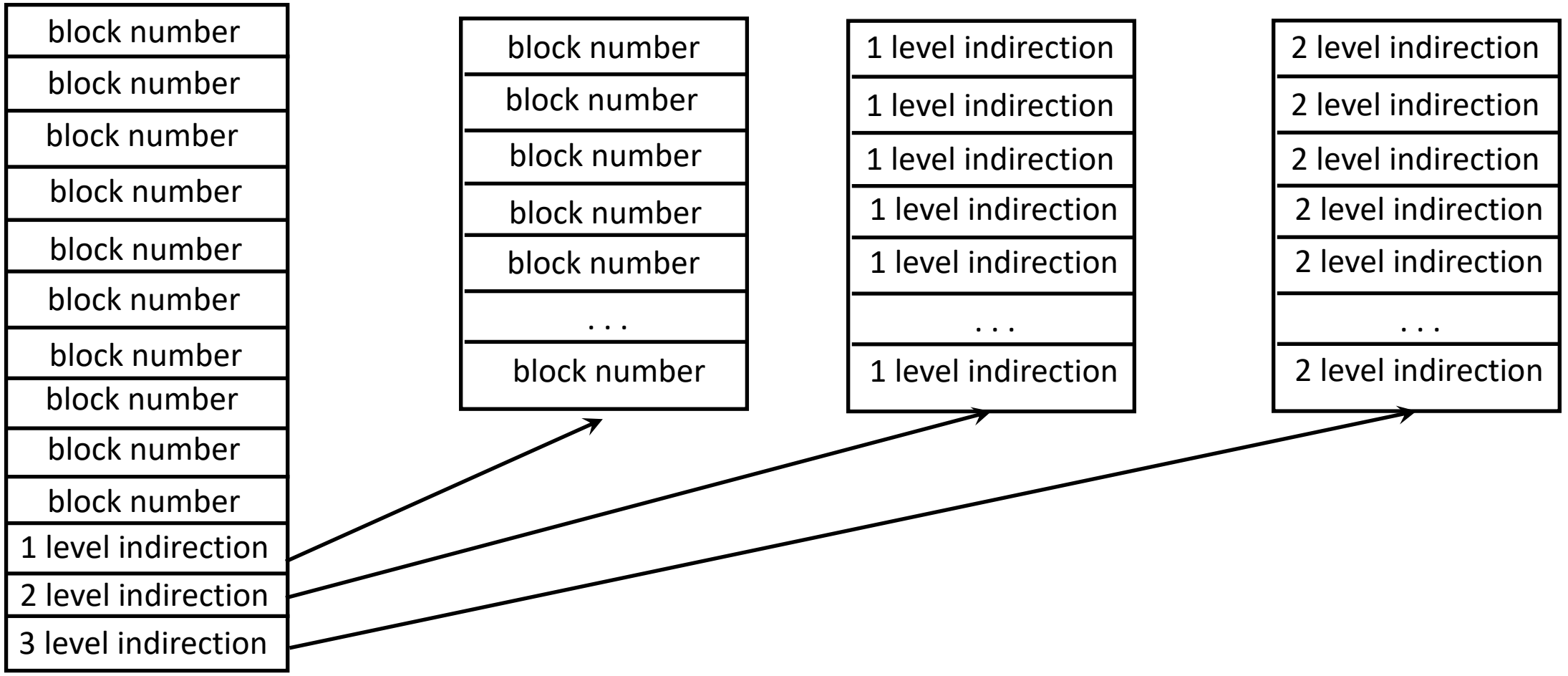
Aliasing Issues

- Accounting: on systems that charge by storage space used, the owner of a file pays for storage (and other related charges)
  - So if another user creates a direct alias to the file, the owner might no longer be able to delete all references to it!
- Solution: have each person owning a link to the file (*ie*., owning a directory containing a link to the file) pay a percentage of the cost of the file

Information About File: UNIX V7 inode

```
struct  inode {
      char    i_flag;
      char    i_count;       /* reference count */
      dev_t   i_dev;         /* device where inode resides */
      ino_t   i_number;      /* i number, 1-to-1 with device address */
      unsigned short  i_mode;
      short   i_nlink;       /* directory entries */
      short   i_uid;         /* owner */
      short   i_gid;         /* group of owner */
      off_t   i_size;        /* size of file */
      union {
            struct {
                  daddr_t i_addr[13];  /* if normal file/directory */
                  daddr_t i_lastr;       /* last logical block read (for read-ahead) */
            };
            struct  {
                  daddr_t i_rdev;              /* i_addr[0] */
                  struct  group   i_group;       /*  multiplexor group file */
            };
      } i_un;
};
```

Layout of Addresses in inode



Superblock in UNIX V7

- Holds information about the file system
- Replicated in known places on disk
  - So if one gets damaged, another can substitute for it

```
struct  filsys {
        unsigned short s_isize;         /* size in blocks of i-list */
        daddr_t s_fsize;                /* size in blocks of entire volume */
        short   s_nfree;                /* number of addresses in s_free */
        daddr_t s_free[NICFREE];        /* free block list */
        short   s_ninode;               /* number of i-nodes in s_inode */
        ino_t   s_inode[NICINOD];       /* free i-node list */
        char    s_flock;                /* lock during free list manipulation */
        char    s_ilock;                /* lock during i-list manipulation */
        char    s_fmod;                 /* super block modified flag */
        char    s_ronly;                /* mounted read-only flag */
        time_t  s_time;                 /* last super block update */
        /* remainder not maintained by this version of the system */
        daddr_t s_tfree;                /* total free blocks*/
        ino_t   s_tinode;               /* total free inodes */
        short   s_m;                    /* interleave factor */
        short   s_n;                    /* " " */
        char    s_fname[6];             /* file system name */
        char    s_fpack[6];             /* file system pack name */
};
```