Linux Programming
6. Directories and File System

Sunwoo Lee

sunwool@inha.ac.kr

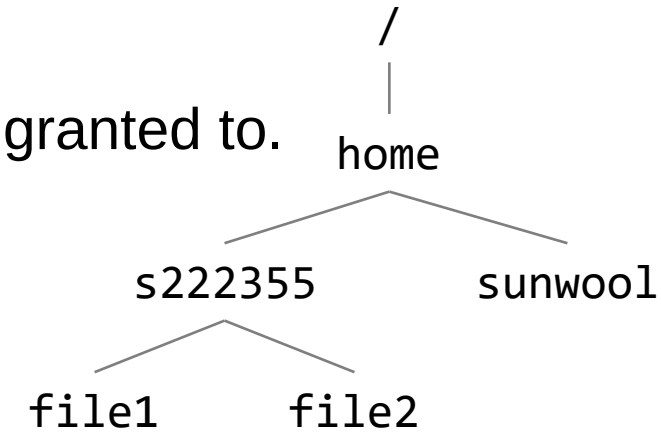
INHA UNIVERSITY

Today, we will learn…

- Directory is another type of file!
- We focus on:
  - How to manage ***directories*** in Linux environments
  - How to use ***file system*** features
  - What kind of ***special files*** are supported in Linux

Terminologies

- File system
  - A hierarchical arrangement of directories and files.
  - Everything starts in the directory called ‘root’ whose name is a single character ‘/’.
- Working directory
  - Every process has a working directory (current working directory).
  - The directory from which all relative pathnames are interpreted.
- Home directory
  - Every user has a directory where the access permission is granted to.
- Pathname
  - Absolute path: `/home/sunwool/file1`
  - Relative path: `./file1`



- ***Directory***
  - ***Implementation of a directory***
  - Programming with directory
- File System
- Device Files

Directory (1/2)

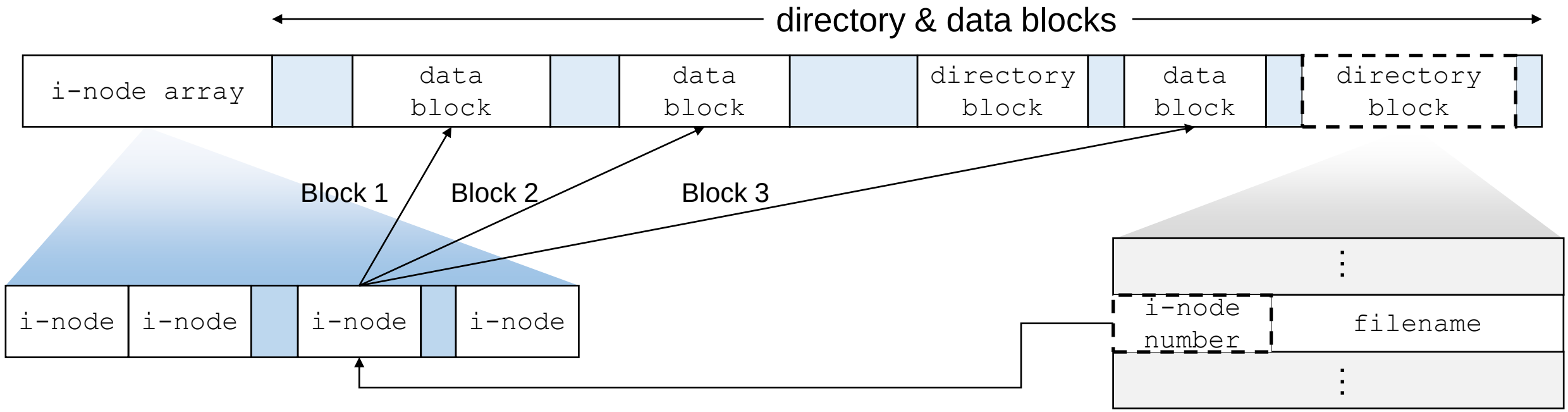
- A file containing **multiple directory entries**
- Most of the system calls for handling regular files could be used to manipulate directories.
  - Directories may not be created using `creat()` or `open()`.
  - E.g., if `O_WRONLY` or `O_RDWR` mode is used, `open()` does not work (errno = `EISDIR`).
  - Only the kernel can directly write into a directory.

Directory (2/2)

- Directory consists of a series of directory entries, one for each file or subdirectory.
- Directory entry
  - i-node number
  - Character fields (name)

120	f	r	e	d	\0				
207	b	o	o	k	m	a	r	k	\0
235	a	b	c	\0					

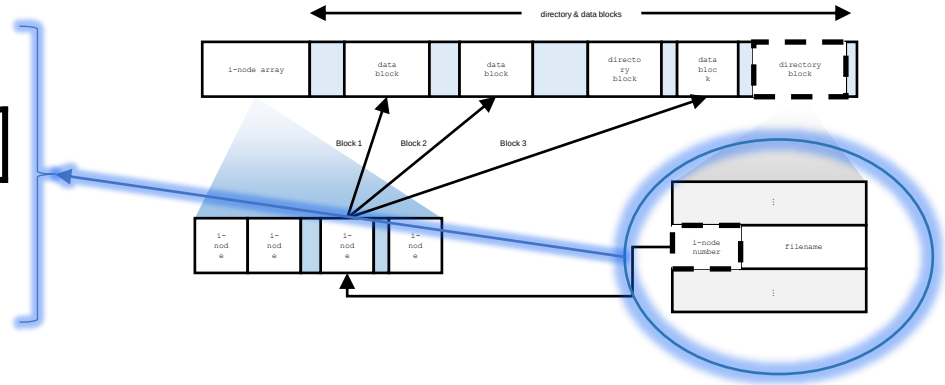
Example: Directory Structure



Revisiting link() and unlink()

- A new link simply results in a new directory entry with the same i-node number as the original.
  - `link("abc", "xyz");`

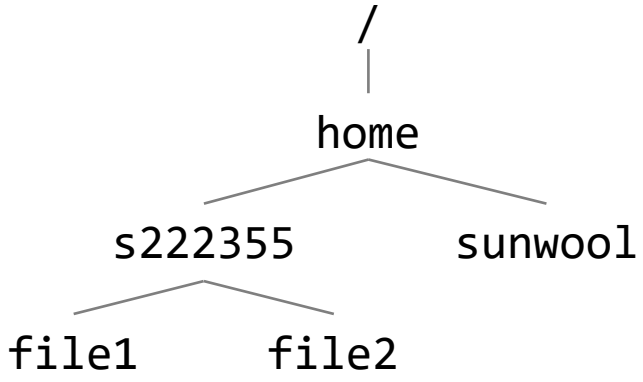
120	f	r	e	d	\0			
207	b	o	o	k	m	a	r	k	\0
235	a	b	c	\0				
235	x	y	z	\0				



- When a link is removed using the `unlink()` system call:
  - If the removed entry was the last link, the corresponding i-node structure is cleared.

Dot and Double-Dot

- '.': current working directory
- '..': the parent directory



home

123	.	\0						
2	.	.	\0					
260	s	2	2	2	3	5	5	\0
401	s	u	n	w	o	o	\0	

s222355

260	.	\0				
123	.	.	\0			
475	f	i	l	e	1	\0
476	f	i	l	e	2	\0

sunwool

401	.	\0	
123	.	.	\0

Directory Permissions (1/3)

- Directory permissions are organized in exactly the same way as regular file permissions.
- However, they are interpreted rather differently.
  - Read permission
    - Allows to list the name of files & subdirectories
  - Write permission
    - Allows to create new files or remove existing files
  - Execute permission
    - Allows to get into the directory
    - Allows to call `chdir()` system call within a program

Directory Permissions (2/3)

- E.g., to open the file /usr/include/stdio.h:
  - You need the execute permission on ‘/’, ‘/usr’, ‘/usr/include’.
- ***Note***: read permission and execute permission are different!
  - Read permission: allows to list up the contents
  - Execute permission: allows to pass through
    - The execute permission bit is also called ‘***search bit***’.

Directory Permissions (3/3)

- save-text-image (sticky bit): S_ISVTX
  - If set to 1, the files in the directory can be removed or renamed by the owner of the files, the owner of the directory, and the superuser.

Directory Permission Flag	Description
S_IRUSR	User read
S_IWUSR	User write
S_IXUSR	User execute
S_IRGRP	Group read
S_IWGRP	Group write
S_IXGRP	Group execute
S_IROTH	Others read
S_IWOTH	Others write
S_IXOTH	Others execute
S_ISVTX	Sticky bit

- ***Directory***
  - Implementation of a directory
  - ***Programming with directory***
- File System
- Device Files

Manipulating Directories

- A special family of calls to do with directories
  - `mkdir()`
  - `rmdir()`
  - `opendir()`
  - `closedir()`
  - `readdir()`
- System calls related to ‘how to manipulate directories in C programs’

Structure: dirent

```
# include <dirent.h>

struct dirent {
        ino_t  d_ino;                   /* i-node number */
        char   d_name[NAME_MAX + 1];    /* NULL-terminated filename */
}
```

- A special structure type every directory entry has.
- `d_ino` of zero denotes an empty slot in the directory.

System Call: mkdir()

```
# include <sys/stat.h>

int mkdir(const char *pathname, mode_t mode);
```

- `mkdir()` creates a new directory in the current working directory.
  - `mode` is the access permission that can be modified using `umask()`.
- `mkdir()` automatically creates two links in the new directory, '.' and '..'.
- At least one of execute bit (owner, group, or others) should be enabled to move into the directory and access its contents.

System Call: rmdir()

```
# include <unistd.h>

int rmdir(const char *pathname);
```

- rmdir() deletes an **empty** directory.
  - An empty directory indicates the one that only has '.' and '..' links.

System Call: opendir() / closedir()

```
# include <dirent.h>

DIR *opendir(const char *dirname);
```

```
# include <dirent.h>

int closedir(DIR *dirptr);
```

- `opendir()` opens the directory and returns a descriptor of type `DIR`.
  - `DIR` works in a similar way to `FILE` type in the standard I/O library.
  - It returns `NULL` on error.
- `closedir()` closes the directory referred to by `dirptr`.

System Call: readdir()

```
# include <dirent.h>

struct dirent *readdir(DIR *dirptr);
```

- `readdir()` traverses over all the entry in the directory one after another.
- The information of each directory entry is stored into `dirent` and returned.
- When `readdir()` succeeds, `dirptr` is moved to the next directory entry.

System Call: rewinddir()

```
# include <dirent.h>

void rewinddir(DIR *dirptr);
```

- `rewinddir()` moves the `dirptr` on to the first entry of the directory.
- Similar to `lseek(…, SEEK_SET)`.

Example 1 (double ls)

```
#include <dirent.h>

int my_double_ls (const char *name) {
  struct dirent *d;
  DIR *dp;

  /* open directory */
  if ((dp=opendir(name))==NULL)
    return (-1);
  /* find the valid i-node and print the directory */
  while (d=readdir(dp)) {
    if (d->d_ino != 0)
      printf("%s\n", d->d_name);
  }
  /* rewind dirptr */
  rewinddir(dp);
  /* print the directory again */
  while (d=readdir(dp)) {
    if (d->d_ino != 0)
      printf("%s\n", d->d_name);
  closedir(dp);
  return 0;
}
```

Example 2 (find entry)

```c
#include <stdio.h>              /* define NULL */
#include <dirent.h>
#include <string.h>

int match (const char *, const char*);
char *find_entry(char *dirname, char *suffix, int cont) {
  static DIR *dp=NULL;
  struct dirent *d;
  if (dp == NULL || cont == 0) {
    if (dp != NULL)
      closedir (dp);
    if ((dp = opendir(dirname)) == NULL)
      return (NULL);
  }
  while (d = readir(dp)) {
    if (d->d_ino == 0)
      continue;
    if (match(d->d_name, suffix))
      return (d->d_name);
  }
  closedir(dp);
  dp = NULL;
  return (NULL);
}
```

```c
int match (const char *s1, const char *s2) {
  int diff = strlen(s1) - strlen(s2);

  if (stlen(s1) > strlen(s2))
    return (strcmp(&s1[diff], s2) == 0);
  else
    return 0;
}
```

0	1	2	3	4	5	6	7	8
t	e	s	t	f	i	l	e	0

0	1	2	3	4	5	6	7	8
f	i	l	e	0				

The Current Working Directory

- Each Unix (LINUX) process has its own current working directory.
- The current working directory is the directory associated with the shell process that interprets his or her commands.
- Any new processes are given with the parent process' current working directory.

System Call: chdir()

```
# include <unistd.h>

int chdir(const char *path);
```

- `chdir()` changes the current working directory of the process.
- Error cases:
  - `path` is not a valid directory.
  - Execute permission does not exist at every component directory on the path.
- When the program accesses files in a specific directory, changing the current directory and then accessing them is much faster than directory accessing them using absolute paths.

```
fd1 = open("/usr/ben/abc", O_RDONLY);
fd2 = open("/usr/ben/xyz", O_RDWR);
```

```
chdir("/usr/ben");
fd1 = open("abc", O_RDONLY);
fd2 = open("xyz", O_RDWR);
```

System Call: getcwd()

```
# include <unistd.h>

char *getcwd(char *name, size_t size);
```

- `getcwd()` returns a pointer to the current directory pathname.
- The current directory name is stored into `name`.
- `size` should be at least 1 greater than the directory name.
  - If size is equal to or smaller than the directory name, it returns errno of ERANGE.

Example: my_pwd

```
/* my_pwd – print the current working directory. */

#include <stdio.h>
#include <unistd.h>
#define VERYBIG 200

void my_pwd (void);

main()
{
   my_pwd();
}

void my_pwd (void)
{
   char dirname[VERYBIG];

   if ( getcwd(dirname, VERYBIG) == NULL)
         perror("getcwd error");
   else
         printf("%s\n", dirname);
}
```

System Call: ftw() (1/2)

```
# include <unistd.h>

int ftw(const char *path, int(*func)(), int depth);
```

- `ftw()` performs a ‘directory tree walk’ starting at any given directory.
- When it finds an entry, it calls a user-defined routine `func` for every directory entry.
- Termination condition:
  - The bottom of the tree is reached, or any errors encountered.
  - `func` returns a non-zero value.

System Call: ftw() (2/2)

```
int func(const char *name, const struct stat *sptr, int type) {
  /* body of function */
}
```

- `func` should follow the above definition.
  - `name`: object name
  - `sptr`: a pointer to the stat structure about the object
  - `type`:
    - FTW_F: the object is a file
    - FTW_D: the object is a directory
    - FTW_DNR: the object is a directory that cannot be read.
    - FTW_SL: the object is a symbolic link
    - FTW_NS: object is not a symbolic link, but `stat()` could not be executed successfully

Example: permission print

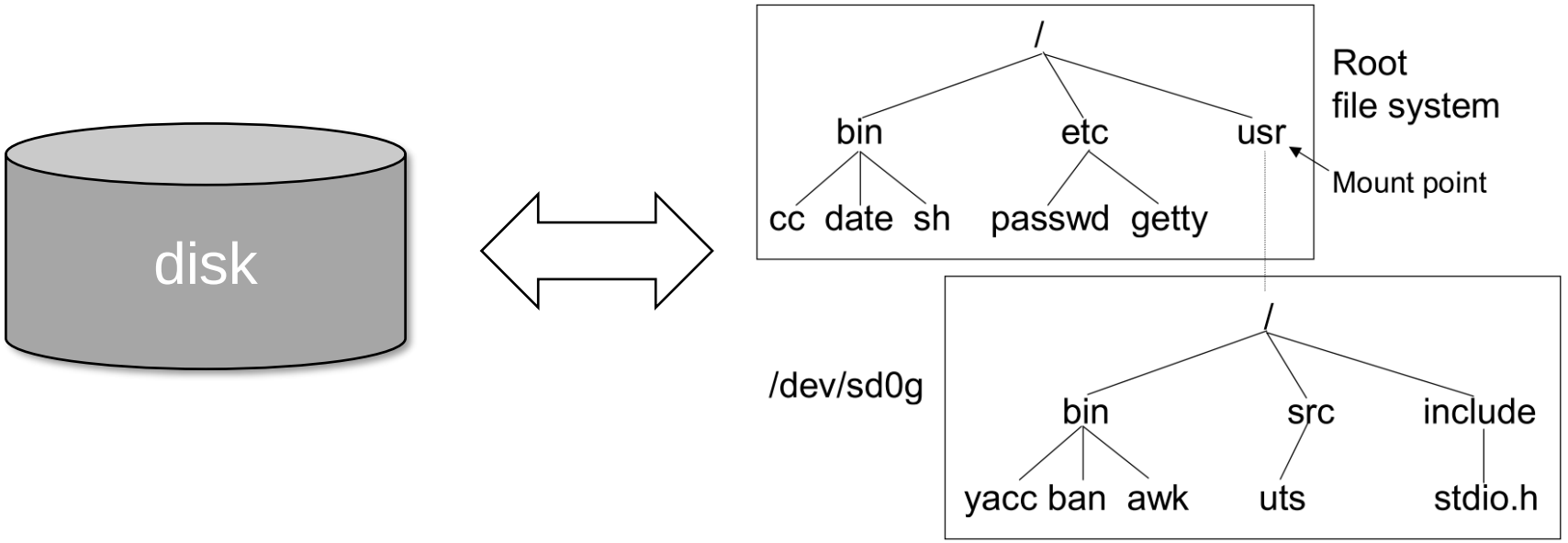
```
#include <sys/stat.h>
#include <ftw.h>

int list(const char *name, const struct stat *status, int type) {
   if (type == FTW_NS) /* return if stat fails */
      return 0;
   /* print name and permission of object. if object is not a file, add '*' between name and permission */
   if(type == FTW_F)
      printf("%-30s\t0%3o\n", name, status->st_mode&0777);
   else
      printf("%-30s*\t0%3o\n", name, status->st_mode&0777);

   return 0;
}

main (int argc, char **argv) {
   int list(const char *, const struct stat *, int);

   if (argc == 1)
      ftw (".", list, 1);
   else
      ftw (argv[1], list, 1);
   exit (0);
}
```

```
$ list
.                              * 0755
./list                           0755
./file1                          0644
./subdir                       * 0777
./subdir/another                 0644
./subdir/subdir2               * 0755
./subdir/yetanother   0644
```

- Directory
  - Implementation of a directory
  - Programming with directory
- ***File System***
- Device Files

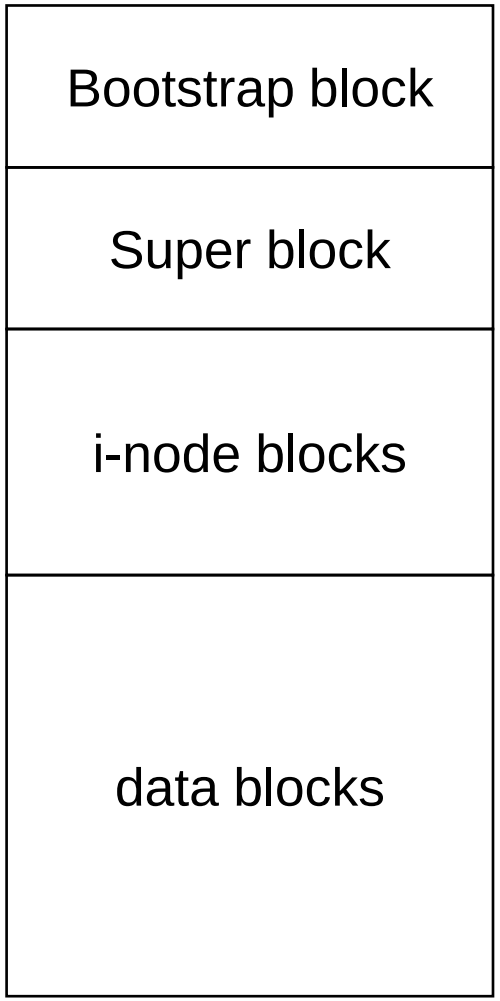
Unix (LINUX) File Systems

- A file system implements a logical structure for storing files on storage devices (e.g., HDDs or SSDs)
  - Usually, it builds up a tree structure whose top node is root, '/'.



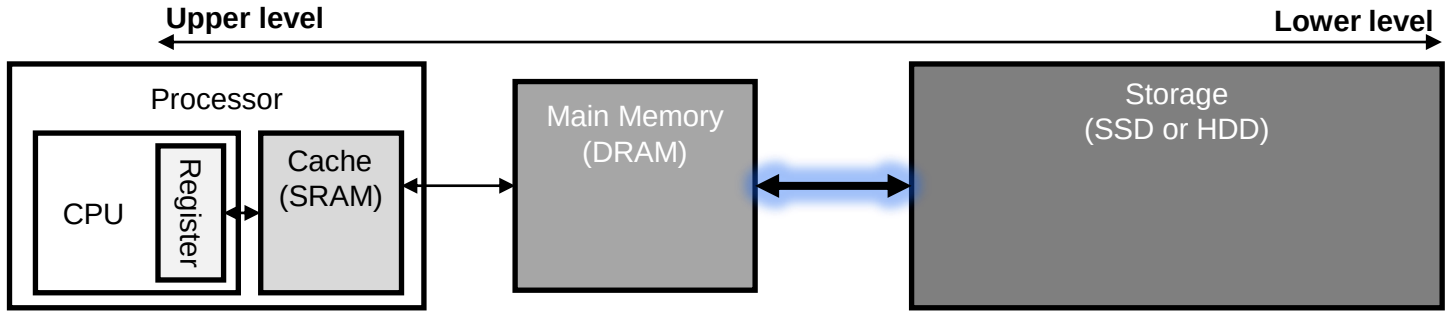
File System Structure

- Bootstrap block
  - Linux boot code is stored (read-only).
- Super block
  - The total number of blocks in the file system
  - The number of free i-nodes
  - The size of block in bytes
  - The number of free blocks
  - The number of used blocks
- i-node block
  - All i-node associated with the files on the disk
- Data block
  - Actual data in the files



Caching

- Super block is cached in the memory space so as to access the metadata fast.
- All transfers from memory to disk (i.e., writes) are typically cached in the operating system's memory space before the I/O.
  - Any given moment, data on disk may be out of date as compared to the cache
- Unix (LINUX) provides two system calls regarding data caching
  - `sync()`
  - `fsync()`



System Call: sync() and fsync()

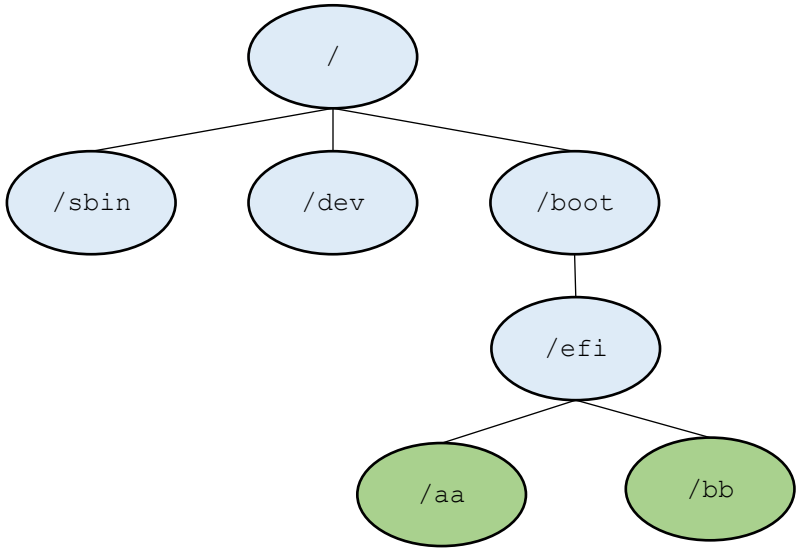
```
# include <unistd.h>

void sync(void);
int fsync(int filedes);
```

- `sync()` flushes out all the cached data and metadata in the main memory to the disk.
- `fsync()` flushes out all the cached data and metadata associated with a specific file.
- Main difference
  - `sync()` does not wait until all the writes are completed while `fsync()` waits for the writes to be finished.
- Unix (LINUX) periodically calls `sync()` to maintain consistency.

Mounting File Systems

- Mount: Attaching a file system to the current file system.
  - Block devices can be mounted to a file system.
  - Different file systems can be connected under the same tree structure.

```
sunwoo@lambda-server1:~$ df -T
Filesystem     Type      1K-blocks       Used  Available Use% Mounted on
udev           devtmpfs  263961040          0  263961040   0% /dev
tmpfs          tmpfs      52801500       3916   52797584   1% /run
/dev/sda2      ext4     3690298248 2464344156 1038427080  71% /
tmpfs          tmpfs     264007484       5008  264002476   1% /dev/shm
tmpfs          tmpfs          5120          4       5116   1% /run/lock
tmpfs          tmpfs     264007484          0  264007484   0% /sys/fs/cgroup
/dev/sda1      vfat         523248       6196     517052   2% /boot/efi
```



System Call: mount() and umount()

```
# include <sys/mount.h>

int mount(const char *source, const char *target, ...);
int umount(const char *target);
```

- `mount()` attaches the file system specified by `source` to the directory specified by `target`.
  - `source` is usually a device name but can be a directory name.
- `umount()` removes the attachment specified by `target`.
- From Linux kernel 2.4., a single file system can be mounted at multiple points.

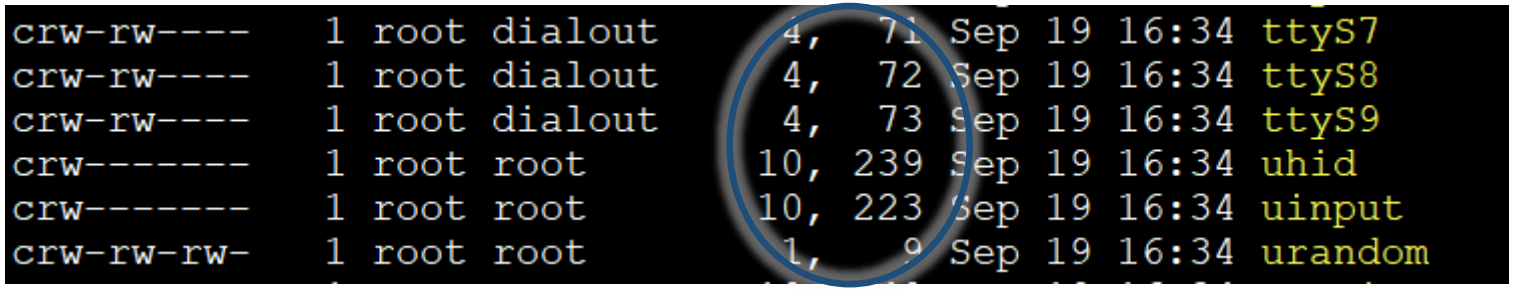
- Directory
  - Implementation of a directory
  - Programming with directory
- File System
- ***Device Files***

Data Transfer between Computer and Device

- Remember that Linux considers everything as a file.
- That is, any devices connected to the computer are also considered as files!
- Linux communicates with the devices through the appropriate files.
  - Data is transferred using regular file access system calls.
  - However, the actual underlying behaviors are different depending on the device drivers.

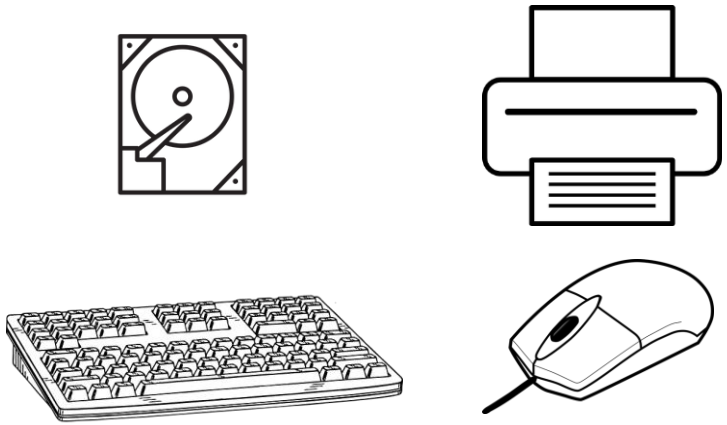
Devices in Unix (LINUX)

- Each device is accessed with an allocated unique device number.
  - Device number consists of major and minor numbers.
  - Major number: the type of device
  - Minor number: the instance of a specific type of devices
- Users do not need to know the device numbers to access the actual device → Each number is mapped to a file.



Devices in Unix (LINUX)

- The peripherals
  - Accessed through file names in the file system.
  - Disks, terminals, printers, etc.
- Reads and writes to these device files transfer data directly between the system and the corresponding peripheral devices.
- Located in ‘/dev’
  - /dev/tty00
  - /dev/lp
  - /dev/pts/as



Character and Block Device Files

- **Character** Device Files
  - E.g., printer, terminal, and network devices
  - Byte-wise random access may not be allowed.
  - The data is transferred byte sequences of an arbitrary length.
- **Block** Device Files
  - E.g., storage such as HDD or SSD
  - The data is always transferred by blocks
  - Random access is allowed.
  - File systems are supported.

Data Transfer (1/3)

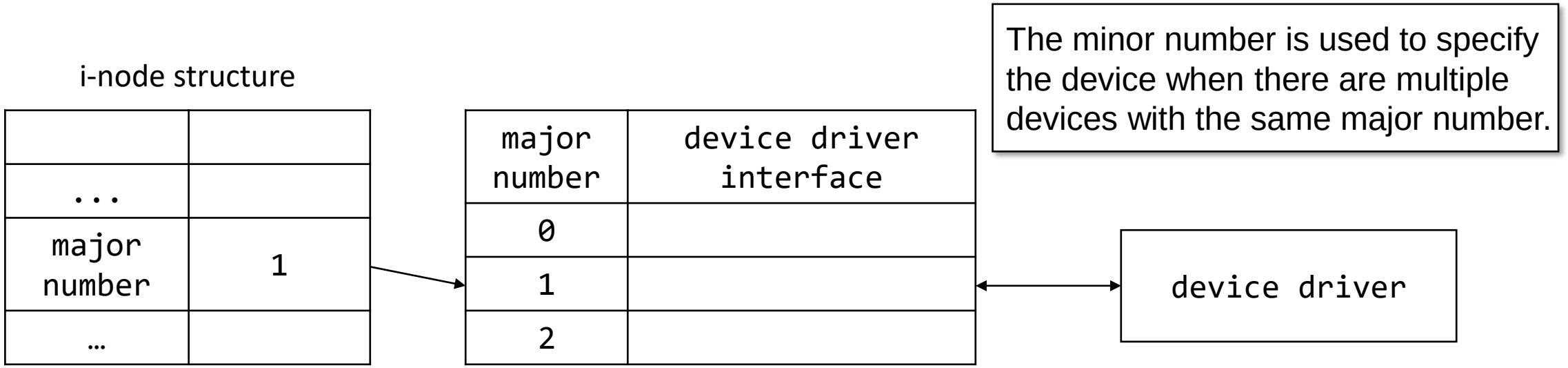
- Two device configuration tables for interacting with peripherals
  - Block device switch table
  - Character device switch table
- The tables are indexed using '***major device number***' which is stored in the device files' i-node.

Device Configuration Table

major number	device driver interface
0	
1	
2	

Data Transfer (2/3)

1. `read()` or `write()` system call finds the i-node of the target device file.
2. It reads the device type (char vs. block) and its major and minor device numbers from the i-node.
3. The appropriate device configuration table is found using the major number first, and then the connected device driver transfers the data.



Revisiting `stat` Structure

```
struct stat {
       mode_t    st_mode;    /* file type & mode (permissions) */
       ino_t     st_ino;     /* i-node number (serial number) */
       dev_t     st_dev;     /* device number (file system) */
       dev_t     st_rdev;    /* device number for device files */
       nlink_t   st_nlink;   /* number of links */
       ...
};
```

st_mode

type	special			Permission								
4 bit	u	g	s	r	w	x	r	w	x	r	w	x

binary 0001 → FIFO
binary 0010 → character dev
binary 0100 → directory
binary 0110 → block dev
binary 1000 → regular
binary 1010 → symbolic link

- `st_mode`: 060000 (S_IFBLK) for block device files and 020000 (S_IFCHR) for character device files
- `st_rdev`: major and minor numbers for device files

```
$ ls -l /dev/tty3
  crw--w--w-      1  eojin  other  8,3  Aug 16 17:19 /dev/tty3
```

character device file major, minor number

File System Information (1/2)

```
# include <sys/statvfs.h>

int statvfs(const char *path, struct statvfs *buf);
int fstatvfs(int fd, struct statvfs *buf);
```

- `statvfs()` and `fstatvfs()` obtain basic file system information such as the total number of free disk blocks or the number of free i-nodes.

```
struct statvfs {
  unsigned long f_bsize         File system block size
  unsigned long f_frsize        Fundamental file system block size
  fsblkcnt_t    f_blocks        Total number of blocks on file system in unis of f_frsize
  fsblkcnt_t    f_bfree         Total number of free blocks
  fsblkcnt_t    f_bavail        The number of free blocks available to non-privileged process
  fsfilcnt_t    f_files         Total number of i-nodes
  fsfilcnt_t    f_ffree         Total number of free i-nodes
  fsfilcnt_t    f_favail        The number of i-nodes available to non-privileged process
  unsinged long f_fsid          File system ID
  unsigned long f_flag          Bit mask of f_flag values
  unsigned long f_namemax       Maximum file name length
};
```

File System Information (2/2)

f_flag	Description
ST_RDONLY	The file system is mounted for read-only access.
ST_NOSUID	The file system does not support setuid/setgid semantics.
ST_CHOWN_RESTRICTED	The file system restricts the changing of the owner or primary group to a process that has t he appropriate privileges.
ST_THREAD_SAFE	The file system is thread-safe. Thread-safe APIs may operate on objects in this file system in a thread-safe manner.
ST_DYNAMIC_MOUNT	The file system allows itself to be dynamically mounted and unmounted.
ST_NO_MOUNT_OVER	The file system does not allow any part of it to be mounted over.
ST_NO_EXPORTS	The file system does not allow any of its objects to be exported to the Network File System (NFS) Server.
ST_SYNCHRONOUS	The file system supports the "synchronous write" semantic of NFS Version 2.
ST_CASE_SENSITIVE	The file system is case sensitive.

Example: File System Information

```
/* fsys – print file system information */
#include <sys/statvfs.h>
#include <stdlib.h>
#include <stdio.h>

main (int argc, char **argv) {
   struct statvfs buf;

   if (argc != 2)
   {
      fprintf (stderr, "usage: fsys filename\n");
      exit (1);
   }
   if (statvfs (argv[1], &buf) !=0)
   {
      fprintf (stderr, "statvfs error\n");
      exit (2);
   }

   printf ("%s:\tfree blocks %d\tfree inodes %d\n",
                        argv[1], buf.f_bfree, buf.f_ffree);
   exit (0);
}
```

System Call: `pathconf()` and `fpathconf()`

```
# include <unistd.h>

long pathconf(const char *pathname, int name);
long fpathconf(int filedes, int name);
```

- Both functions get the limitation information about files.
- Arguments
  - _PC_LINK_MAX: the maximum file link count
  - _PC_NAME_MAX: the maximum number of bytes in a file name
  - _PC_PATH_MAX: the maximum number of bytes in the pathname

Example: `pathconf()`

```
#include <unistd.h>
#include <stdio.h>

typedef struct {
  int val;
  char *name;
} Table;

main() {
  Table *tb;
  static Table options[] = {
    { _PC_LINK_MAX, "Maximum number of links"},
    { _PC_NAME_MAX, "Maximum length of a filename"},
    { _PC_PATH_MAX, "Maximum length of pathname"},
    {-1, NULL}
  };

  for (tb=options; tb->name != NULL; tb++)
    printf ("%-28.28s\t%ld\n", tb->name, pathconf("/tmp", tb->val));
}
```

```
Maximum number of links        32767
Maximum length of a filename   256
Maximum length of a pathname   1024
```

Wrap-Up

- A directory block contains entries per file.
  - Each entry has i-node number and the file name.
  - The linked i-nodes are connected to the data blocks.
- Unix (LINUX) system calls enable programs to manipulate the directories.
  - `mkdir()` / `rmdir()` / `opendir()` / `closedir()` / `readdir()`
- File systems organize the files using a tree structure
  - Many different file systems can be mounted to form a single tree.
- Device files are managed by device configuration tables
  - Device file i-node → device configuration table → device driver → actual I/O

Any Questions?