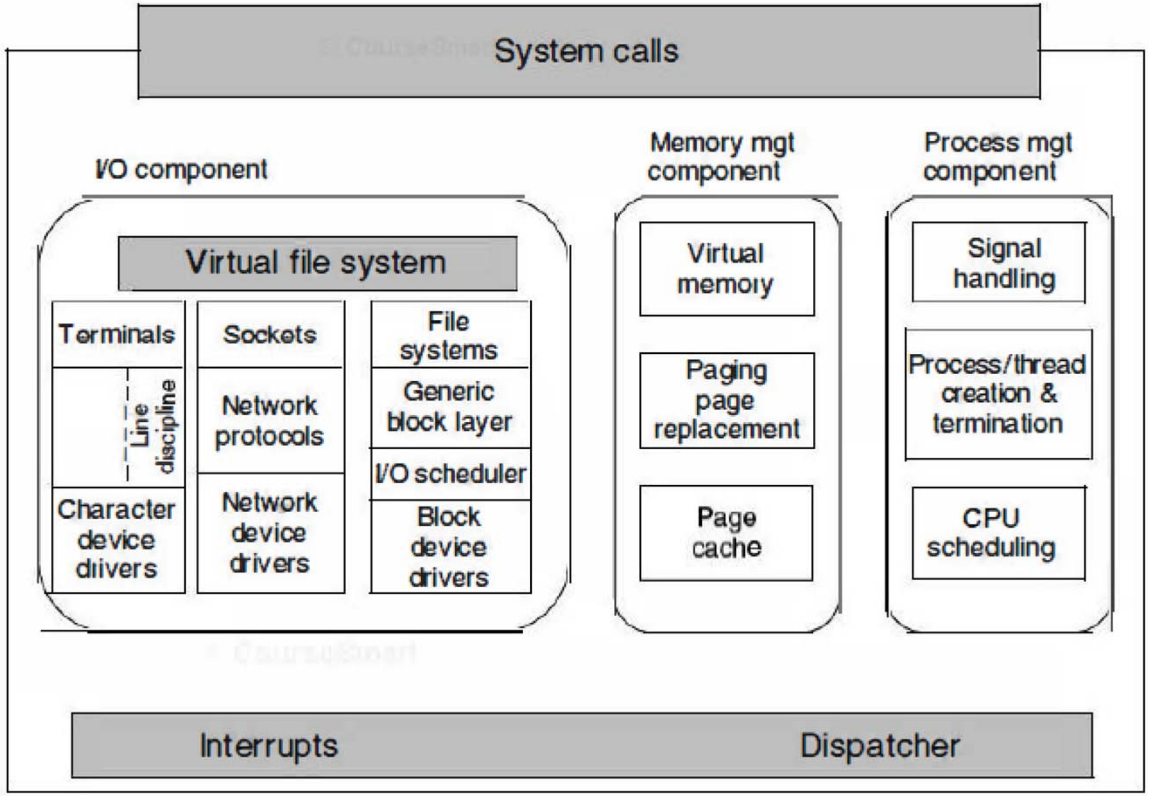
# Operating Systems

Jinghui Zhong （钟竞辉）
Office：B3-515
Email ： jinghuizhong@scut.edu.cn

# Linux Kernel

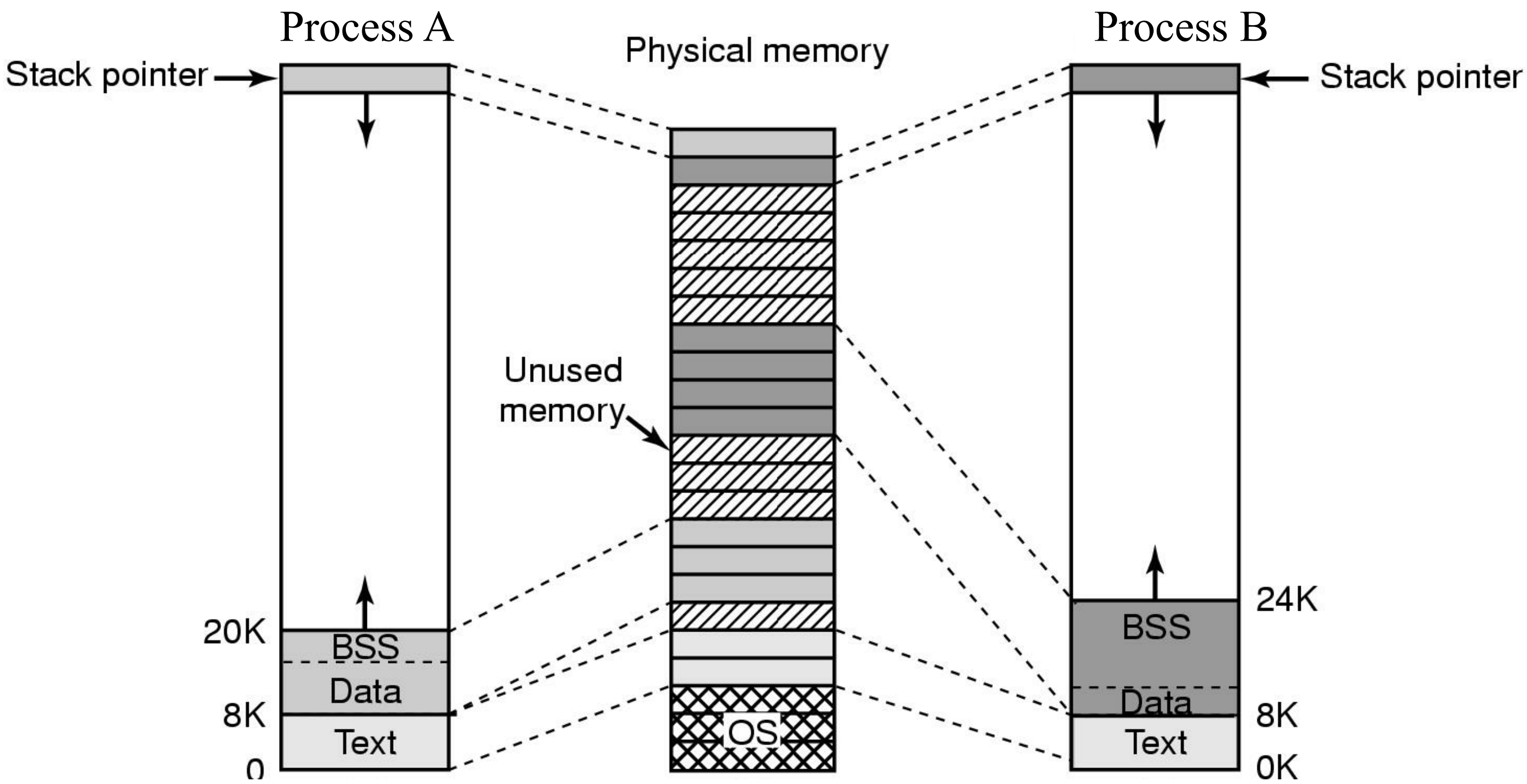
- The kernel sits directly on the hardware and consists of : **I/O Device Component; Memory Management Component, and Processes Management Component**




Structure of the Linux kernel

# Memory Management in Linux

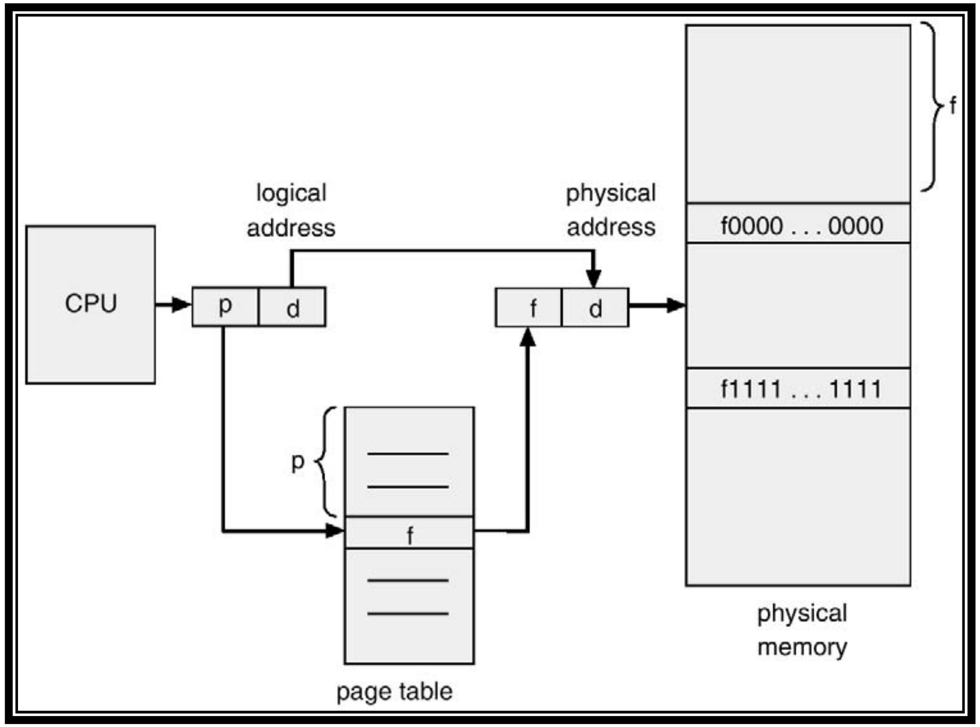
- Memory management in Linux is based on paging; Processes can share Text Segment.




A shares same code fragment with B, but not data and stack.

# Memory Management in Linux

- Virtual Memory (VM):
  - ✓Allow multiple processes share the physical memory;
  - ✓Allow the execution of big process.

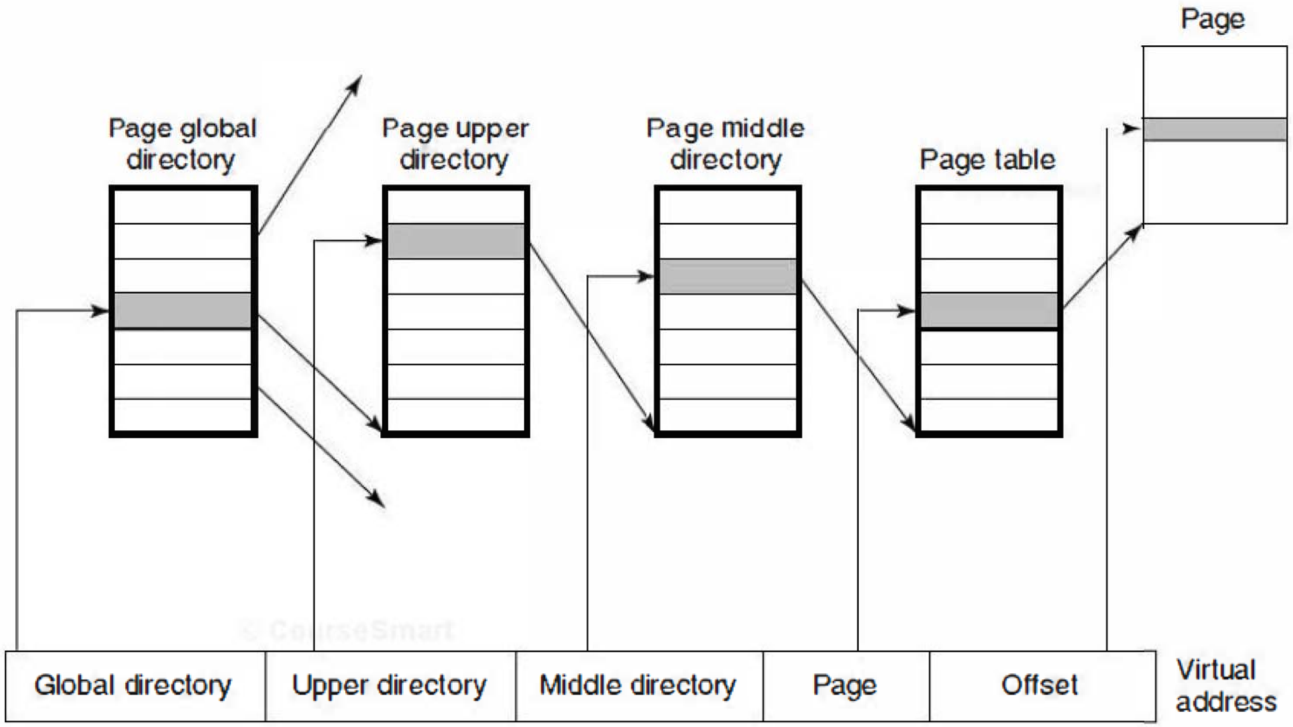



- Techniques to improve VM's efficiency:

  TLB, multi-level page table

# Multiple Level Page Table

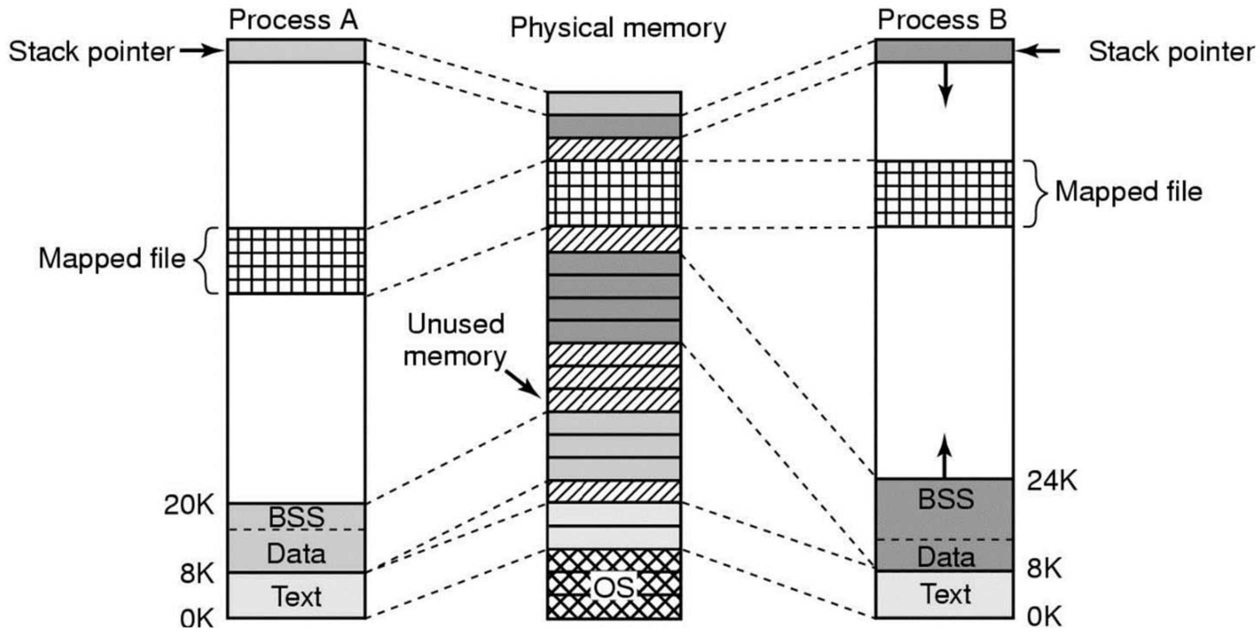
- Each virtual address is broken up into five fields. The value of each directory entry is a pointer to one of the next-level directories.




Linux uses four-level page tables

# Memory-mapped Files

- Processes in Linux can access file data through memory-mapped files so that they can map the same file simultaneously.




- System calls for memory-mapped files:

**void** * mmap( **void** ***start**, **size_t length**, **int prot** , **int flags**,**int fd**, off_t **offset**);

# Inter Process Communication via Memory-map file

```c
#include<sys/mman.h>
#include <sys/stat.h>
#include<fcntl.h>
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<error.h>
#define BUF_SIZE 100
int main(int argc, char ** argv)
{
int fd, nread, i;
struct stat sb;
char *mapped, buf[BUF_SIZE];
for(i = 0; i < BUF_SIZE; i++){
        buf[i] = '#';
}
if((fd = open(argv[1], O_RDWR)) <0 ){
        perror("open");
}
if((fstat(fd, &sb)) == -1){
        perror("fstat");
}
if((mapped = (char*)mmap(NULL, sb.st_size, PROT_READ|
        PROT_WRITE, MAP_SHARED,fd,0))==(void *)-1){
        perror("mmap");
}
close(fd);
while(1){
        printf("%s\n", mapped);
        sleep(2);
}

return 1;
}
```

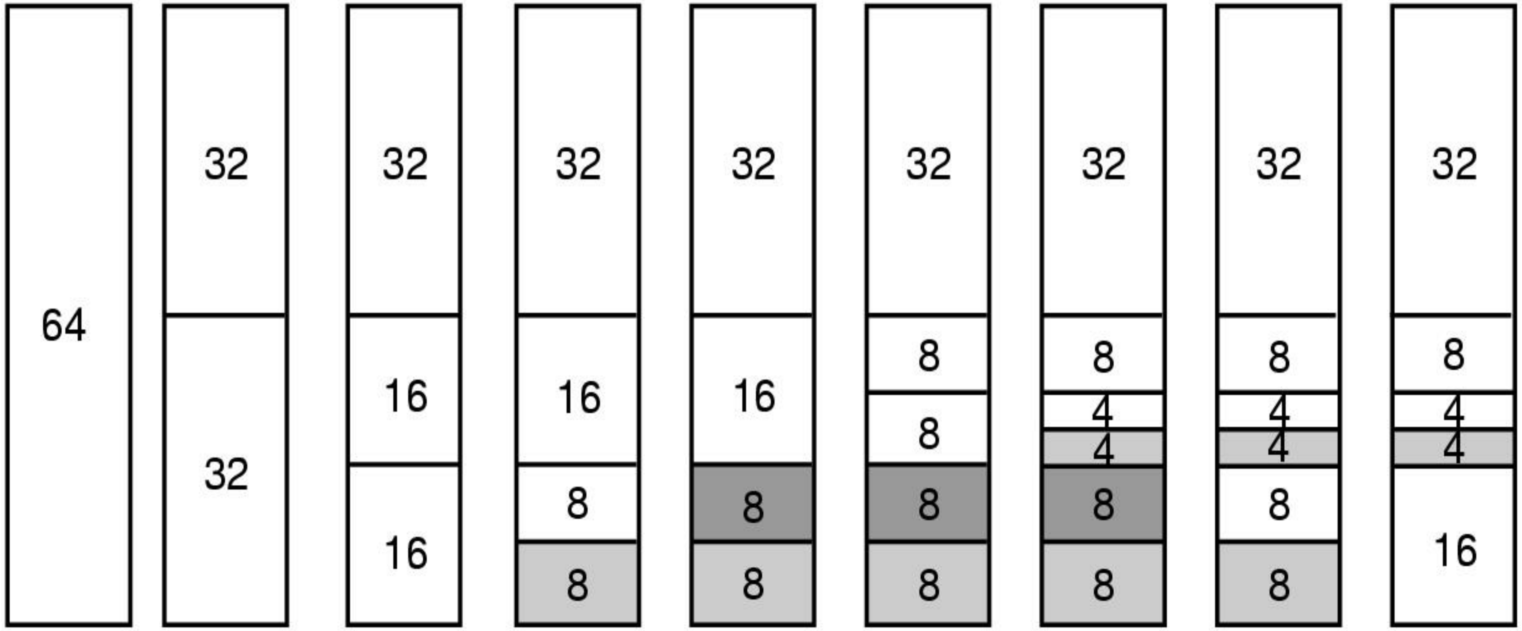
```c
#include<sys/mman.h>
#include <sys/stat.h>
#include<fcntl.h>
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<error.h>
#define BUF_SIZE 100
int main(int argc, char ** argv)
{
int fd, nread, i;
struct stat sb;
char *mapped, buf[BUF_SIZE];
for(i = 0; i < BUF_SIZE; i++){
        buf[i] = '#';
}
if((fd = open(argv[1], O_RDWR)) <0 ){
        perror("open");
}
if((fstat(fd, &sb)) == -1){
        perror("fstat");
}
if((mapped = (char*)mmap(NULL, sb.st_size, PROT_READ|
        PROT_WRITE, MAP_SHARED,fd,0))==(void *)-1){
        perror("mmap");
}
close(fd);

int v = 0;
while(1){
        v = (v + 1)%10;
        mapped[v] = '0' + v;
        sleep(5);
}

return 0;
}
```

# Memory Allocation Mechanism

- **The main mechanism for allocating new page frames of physical memory is the page allocator. The page allocator operates using the well-known buddy algorithm.**




Buddy Algorithm

# Buddy Algorithm: An Example

Memory size： 1024 K;

Events:

1: A 70K;

2: B 35K;

3: C 80K;

4: A ends;

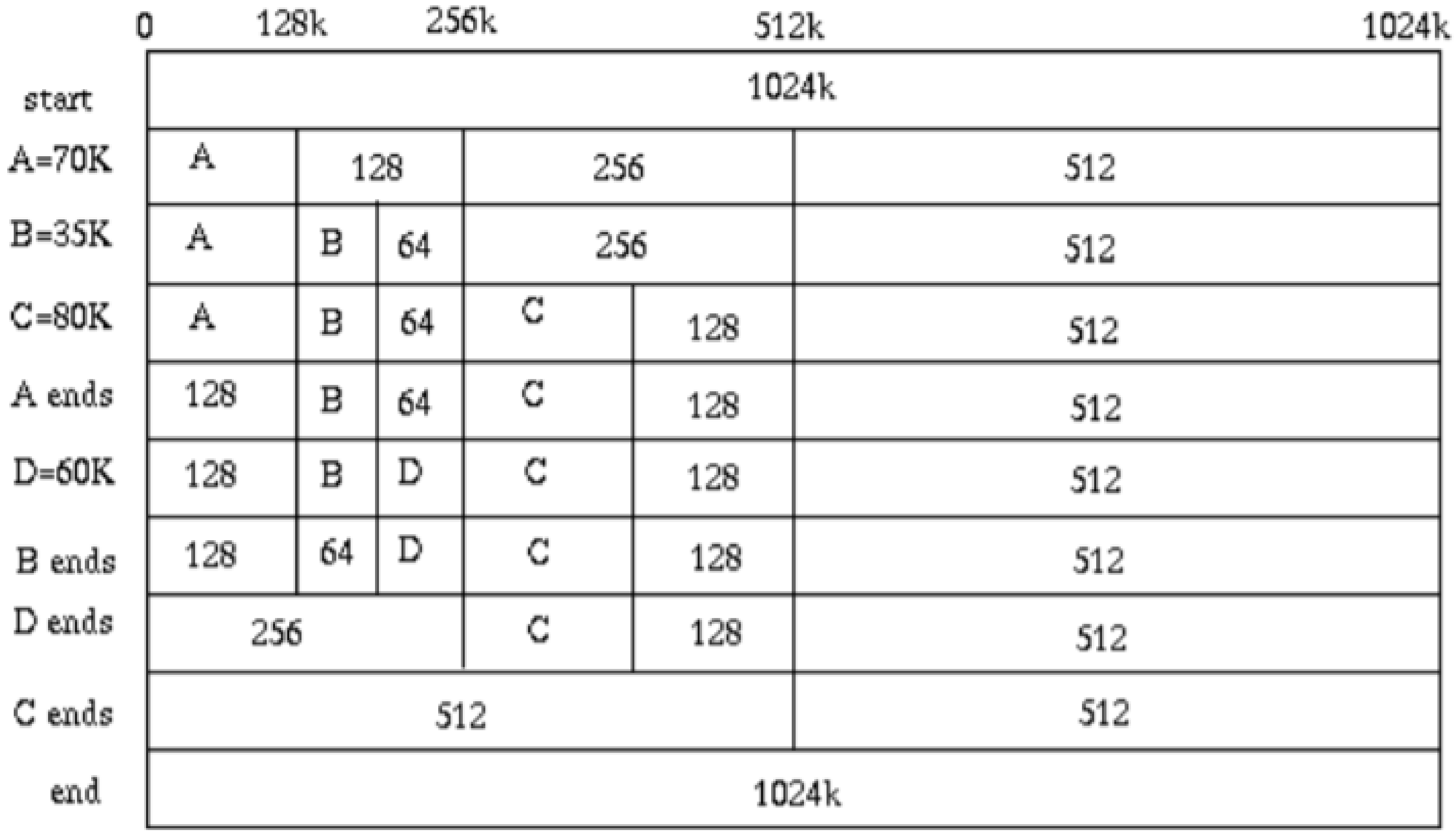
5: D 60K;

6: B ends;

7: D ends;

8: C ends;

9:



# Page Replacement Mechanism

- **A page daemon is initialized at the beginning;**
- **The page daemon checks the page usage periodically (100ms) ; it will free up more pages if the number of free pages is not enough;**
- **Four types of pages:**

① unreclaimable: e.g., locked pages, kernel mode stack;

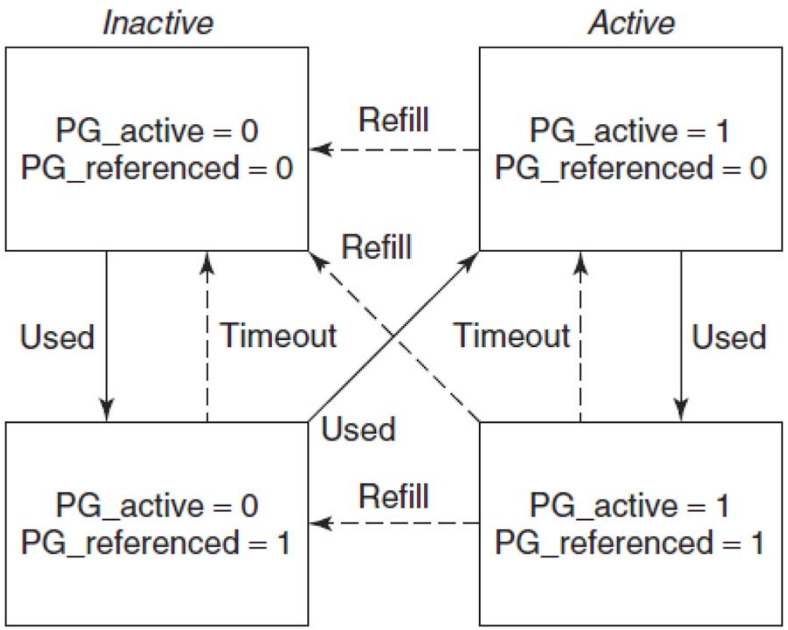
② swappable: must be written back before being reclaimed;

③ syncable: must be written back if it is dirty;

④ discardable: can be reclaimed immediately.

# Page Replacement Algorithm

- Uses an enhanced LRU algorithm, maintaining two flags per page: active/inactive, and referenced or not.
- In the first scan, the reference bits of pages are cleared. If during the second scan, the page is referenced it is moved to a state that is less likely to be reclaimed. Otherwise, the page is moved to a state that is more likely to be reclaimed.

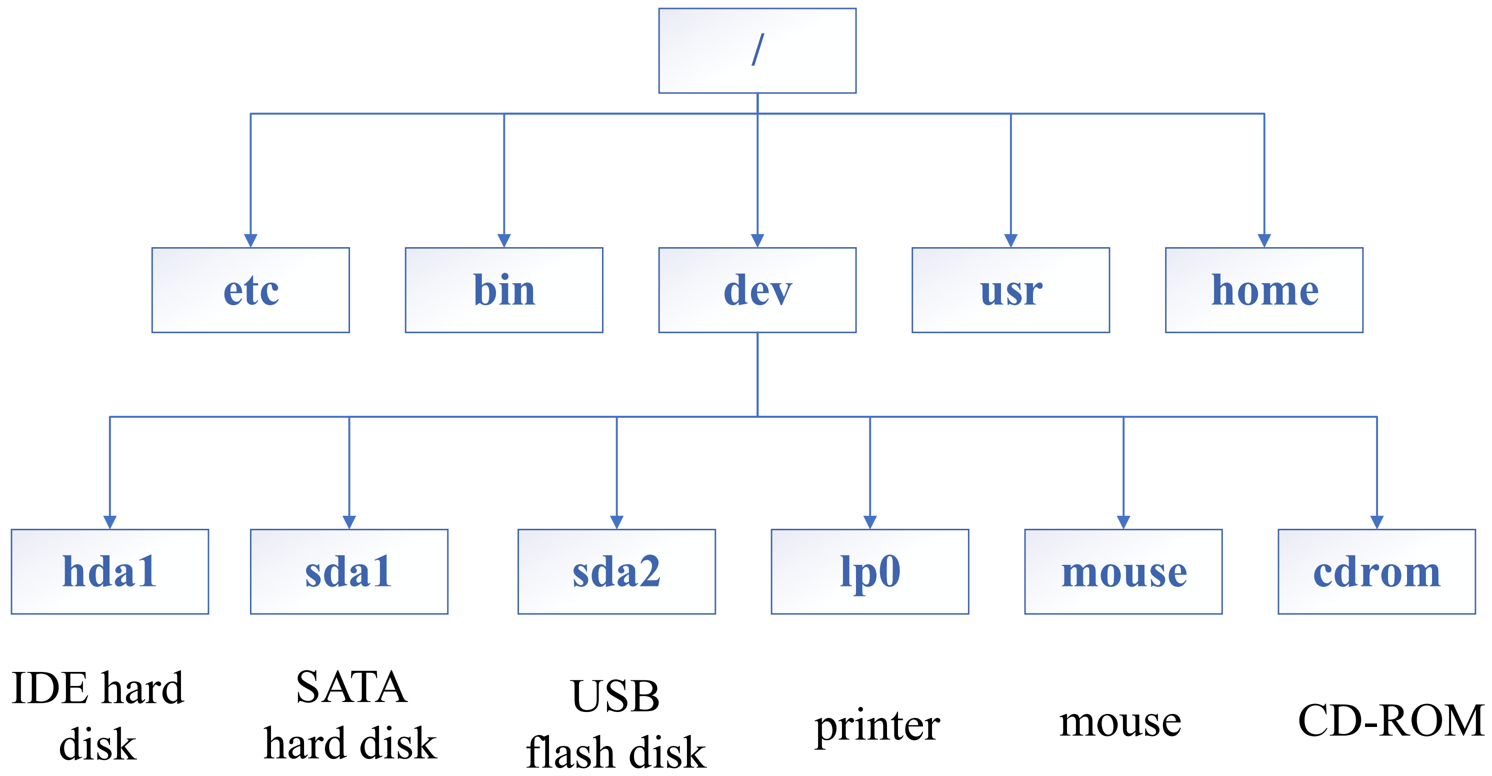


# Linux I/O

- Linux integrates devices into the file system as what are called **special files**.

✓**Block special files**—magnetic disks, etc.

✓**Character special files**—keyboards, printers, mice, etc.



# Input/Output System Calls in Linux

- Each I/O device in a Linux system generally has a special file associated with it. Most I/O can be done by just using the proper file.
- Some special files require special POSIX calls.

| Function call | Description |
|---|---|
| s = cfsetospeed(&termios, speed) | Set the output speed |
| s = cfsetispeed(&termios, speed) | Set the input speed |
| s = cfgetospeed(&termios, speed) | Get the output speed |
| s = cfgtetispeed(&termios, speed) | Get the input speed |
| s = tcsetattr(fd, opt, &termios) | Set the attributes |
| s = tcgetattr(fd, &termios) | Get the attributes |

The main POSIX calls for managing the terminal

# Implementation of Input/Output in Linux

- Each special file is associated with a device driver that handles the corresponding device.

① **Major device number**——to identify the types of devices

② **Minor device number**——to identify devices of the same type

- The system indexes into the hash table of character devices to select the proper structure, then calls the corresponding function to have the work performed.

| Device | Open | Close | Read | Write | Ioctl | Other |
|---|---|---|---|---|---|---|
| Null | null | null | null | null | null | ... |
| Memory | null | null | mem_read | mem_write | null | ... |
| Keyboard | k_open | k_close | k_read | error | k_ioctl | ... |
| Tty | tty_open | tty_close | tty_read | tty_write | tty_ioctl | ... |
| Printer | lp_open | lp_close | error | lp_write | lp_ioctl | ... |

Some of the file operations supported for typical character devices

# The Linux File System

- Files can be grouped together in directories; Directories are stored as files; Directories can contain subdirectories.
- The root directory is called /, and the / character is also used to separate directory names.

| Directory | Contents |
|---|---|
| bin | Binary (executable) programs |
| dev | Special files for I/O devices |
| etc | Miscellaneous system files |
| lib | Libraries |
| usr | User directories |

Some important directories found in most Linux systems

# Some system calls for files

| System call | Description |
|---|---|
| fd = creat(name, mode) | One way to create a new file |
| fd = open(file, how, ...) | Open a file for reading, writing or both |
| s = close(fd) | Close an open file |
| n = read(fd, buffer, nbytes) | Read data from a file into a buffer |
| n = write(fd, buffer, nbytes) | Write data from a buffer into a file |
| position = lseek(fd, offset, whence) | Move the file pointer |
| s = stat(name, &buf) | Get a file's status information |
| s = fstat(fd, &buf) | Get a file's status information |
| s = pipe(&fd[0]) | Create a pipe |
| s = fcntl(fd, cmd, ...) | File locking and other operations |

The return code s is -1 if an error has occurred; fd is a file descriptor, and position is a file offset. The parameters should be self explanatory

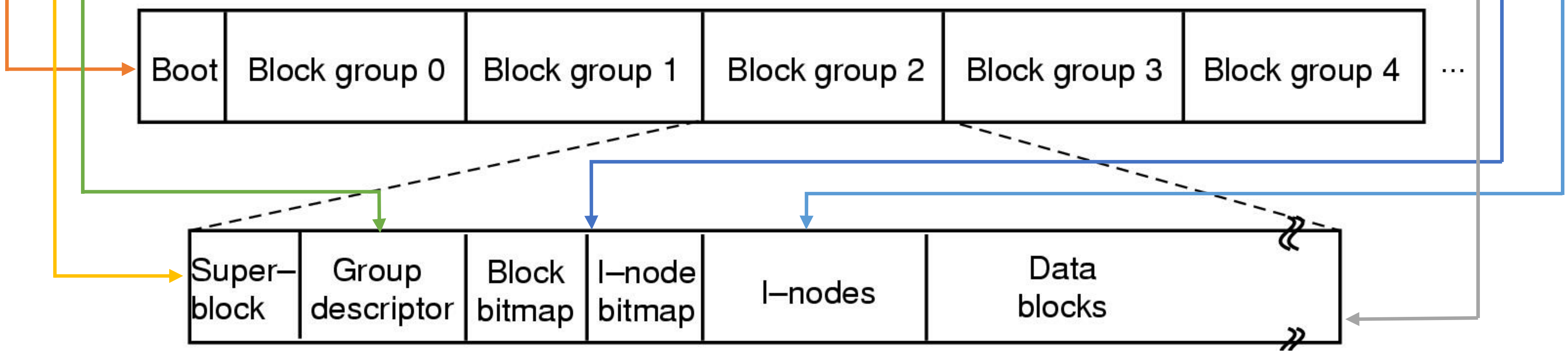
# Some System Calls for Directories

- mkdir, rmkdir： to create and to destroy directories (A directory can only be removed if it is empty)
- link： to create a link； unlink： to delete a link
- chdir： to change the working directory
- opendir， closedir， readdir， rewinddir： for reading directories

| System call | Description |
| --- | --- |
| s = mkdir(path, mode) | Create a new directory |
| s = rmdir(path) | Remove a directory |
| s = link(oldpath, newpath) | Create a link to an existing file |
| s = unlink(path) | Unlink a file |
| s = chdir(path) | Change the working directory |
| dir = opendir(path) | Open a directory for reading |
| s = closedir(dir) | Close a directory |
| dirent = readdir(dir) | Read one directory entry |
| rewinddir(dir) | Rewind a directory so it can be reread |

# The Linux Ext2 File System

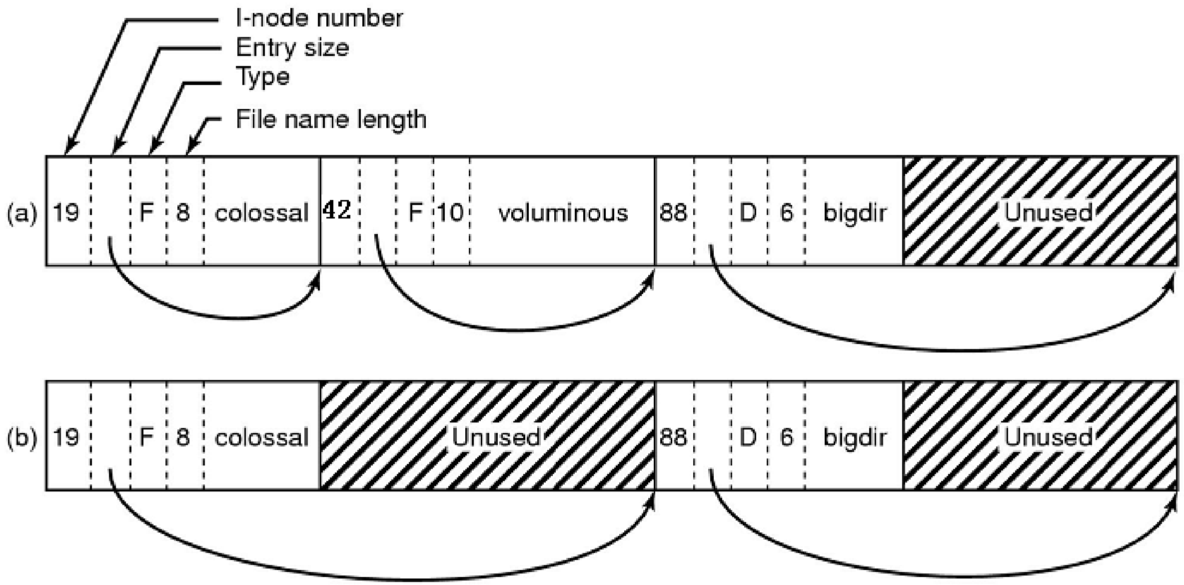
- **block 0**: contains code to boot the computer
- **superblock**: contains information about the layout of the file system(number of i-nodes, number of disk blocks, etc.)
- **group descriptor**: contains information about the location of the bitmaps, the number of free blocks and i-nodes in the group, etc.
- **two bitmaps**: keep track of the free blocks and free i-nodes, respectively
- **i-nodes**: contains accounting information as well as enough information to locate all the disk blocks that hold the file's data
- **data blocks**: areas where all files and directories store




**Disk layout of the Linux ext2 file system**

# The Linux Ext2 File System

- The directory file allows file names up to 255 characters.
- Each directory consists of some integral number of disk blocks so that directories can be written atomically to the disk.
- Entries for files and directories are in unsorted order within a directory.
- Entries may not span disk blocks.




(a). A Linux directory with three files

(b). The same directory after the file voluminous has been removed

# The Structure of I-Node

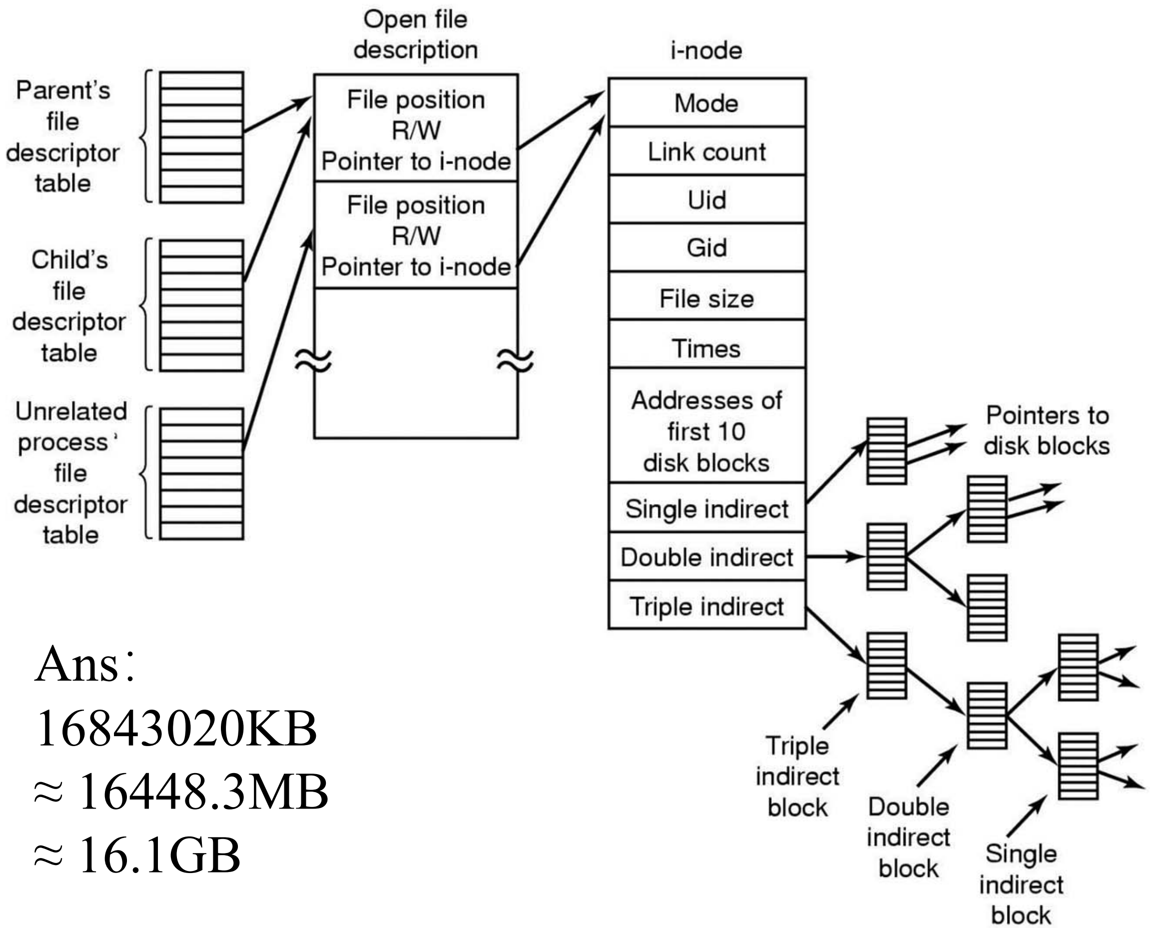
- The i-node is put in the i-node table, a kernel data structure that holds all the i-nodes for currently open files and directories.

| Field | Bytes | Description |
|---|---|---|
| Mode | 2 | File type, protection bits, setuid, setgid bits |
| Nlinks | 2 | Number of directory entries pointing to this i-node |
| Uid | 2 | UID of the file owner |
| Gid | 2 | GID of the file owner |
| Size | 4 | File size in bytes |
| Addr | 39 | Address of first 10 disk blocks, then 3 indirect blocks |
| Gen | 1 | Generation number (incremented every time i-node is reused) |
| Atime | 4 | Time the file was last accessed |
| Mtime | 4 | Time the file was last modified |
| Ctime | 4 | Time the i-node was last changed (except the other times) |

Some fields in the i-node structure in Linux

# Linux ext2 File System

- If a block sizes 1KB with an address of length 4 byte, how large a file can the following i-node indexes at most?




Ans：
16843020KB
≈ 16448.3MB
≈ 16.1GB

The relation between the file descriptor table,
the open file description table, and the i-node table

# Security in Linux

- The user community for a Linux system consists of some number of registered users, each of whom has a unique UID(User ID) which is an integer between 0 and 65535.
- Users can be organized into groups, which are also numbered with 16-bit integers called GIDs(Group IDs).
- The basic security mechanism in Linux is simple. Each process carries the UID and GID of its owner.

| | Binary | Symbolic | Allowed file accesses |
|---|---|---|---|
| read | 111000000 | rwx–––––– | Owner can read, write, and execute |
| write | 111111000 | | |
| execute | 110100000 | | |
| | 110100100 | | Please try the others |
| | 111101101 | | |
| | 000000000 | | |
| | 000000111 | | |

others

group

Owner

Some example file protection modes

# Security System Calls in Linux

- chmod： the most heavily used one, to change the protection mode
- access： to see if a particular access would be allowed using the real UID and GID
- getuid,geteuid,getgid,getegid： return the real and effective UIDs and GIDs
- chown,setuid,setgid： only allowed for the superuser, to change a file's owner, and to change the process' UID and GID

| System cal | Description |
|---|---|
| s=chmod(path,mode) | Change a file's protection mode |
| s=access(path,mode) | Check access using the real UID and GID |
| uid=getuid() | Get the real UID |
| uid=geteuid() | Get the effective UID |
| gid=getgid() | Get the real GID |
| gid=getegid() | Get the effective GID |
| s=chown(path,owner,group) | Change owner and group |
| s=setuid(uid) | Set the UID |
| s=setgid(gid) | Set the GID |

# Session 2

- **Task 2.1: Sleeping Barber Problem**

The barber shop has 5 chairs, 1 barber and 1 barber chair; 20 customers come in the barber shop randomly; If there is no customer, the barber falls asleep; If a customer come in the shop:

① If all chairs are occupied, the customer leaves the shop;

② If the barber is busy and there are free chairs, the customer sits in one of the free chairs;

③ If the barber is asleep, the customer wakes up the barber.

# Sleeping Barber Problem

| Barber | Customer |
| --- | --- |

**Barber**

```
While (True) do {
   Down(cust_ready);
   Down(mutex);
   seat_num++;
   Up(barber_ready);
   Up(mutex);
 # cut hair here
}
```

**Customer**

```
Down(mutex);
If(seat_num > 0){
  seat_num--;
   Up(cust_ready);
   Up(mutex);
   Down(barber_ready)
  }else Up(mutex);
}
```

# Session 2

## ● Task 2.2: Reader & Writer Problem

① 10 readers and 10 writers try to access data S;

② New reader come in every 1 second, and spend 1 second to read the data. New writer come in every 5 seconds and spend 6 seconds to update the data;

③ If readers are reading data, the writers have to wait until all readers finish their jobs.

④ If writers are updating data, the readers have to wait until the writers finish his job.

# Reader & Writer Problem

**Reader**

Down(mutex);
reader ++;
If(reader == 1) Down(wmutex);
Up(mutex);
Read data
Down(mutex);
reader--;
If(reader == 0) Up(wmutex);
Up(mutex);

**Writer**

Down(wmutex)
Write data
Up(wmutex)