Operating Systems

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

Unix/Linux History

- UNICS (Uniplexed Information and Computing Service, 1st version of UNIX)，1969，assembly language

Keb Thompson（left）and Dennis M.Ritchie（right）

Unix/Linux History

- PDP-11UNIX (a popular computer, PDP-11),1973, K. Thompson & D. M. Ritchie

- Portable UNIX (with a portable compiler written in C, Bell Lab, Steve Johnson), 1979.

- Berkeley UNIX (1~4BSD)

  University of California Berkeley, UCB (K. Thompson's Once studied here), 1979-1983

Unix/Linux History

- **Standard UNIX**
  - ✓Portable Operating System Interface of UNIX, POSIX, IEEE, version 1003.1, produced in 1990 (modified in 1995)

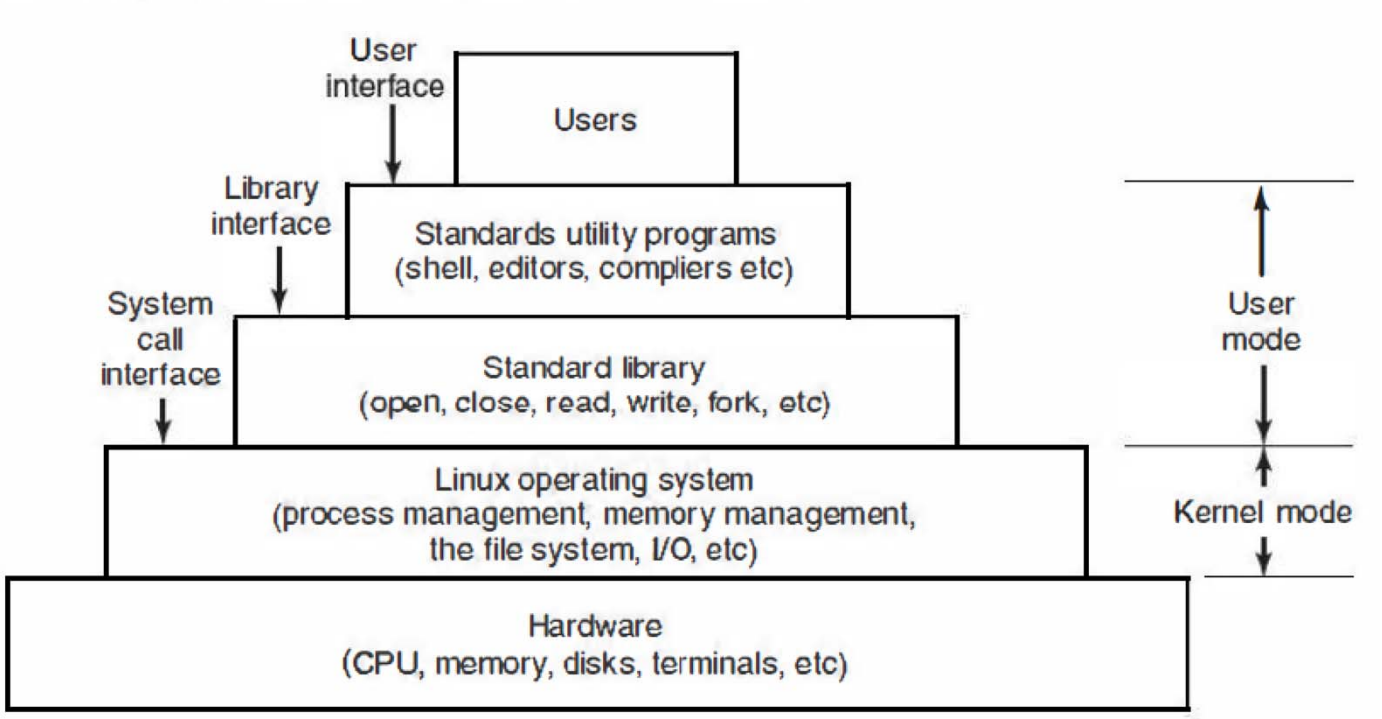
- **Minix:**
  - ✓Microkernel design, C and assembler. Dutch Scientist A. Tanenbaum, for educational purpose, 1987

- **Linux:**
  - ✓Full-blown production system, a Finnish student, Linus Torvalds, 1991

The Development of Linux

The Layers in a Linux System

- The OS controls hardware and provides System calls for user programs； Linux also provides many utility programs required by POSIX.



The layers in a Linux system.

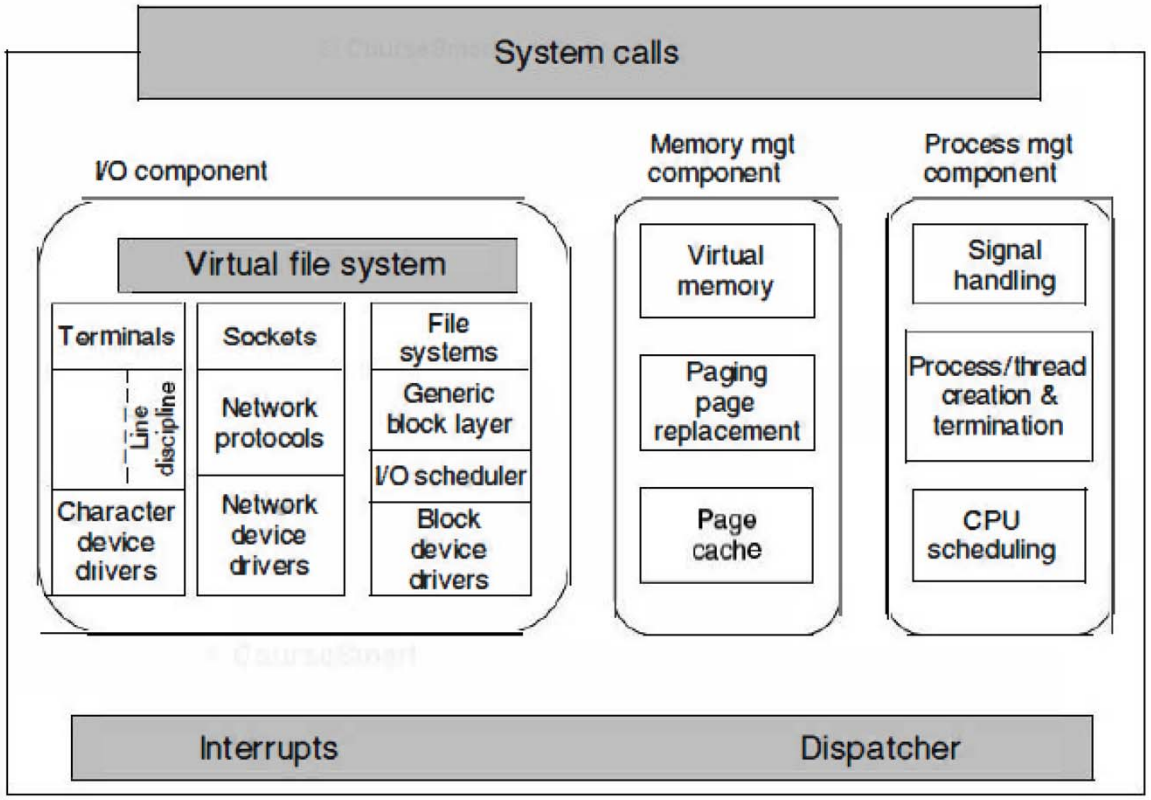
Linux Utility Programs

Program	Typical use
cat	Concatenate multiple files to standard output
chmod	Change file protection mode
cp	Copy one or more files
cut	Cut columns of text from a file
grep	Search a file for some pattern
head	Extract the first lines of a file
ls	List directory
make	Compile files to build a binary
mkdir	Make a directory
od	Octal dump a file
paste	Paste columns of text into a file
pr	Format a file for printing
rm	Remove one or more files
rmdir	Remove a directory
sort	Sort a file of lines alphabetically
tail	Extract the last lines of a file
tr	Translate between character sets

Example utility programs required by POSIX

Linux Kernel

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



Structure of the Linux kernel

Concepts Related to Linux Processes

- Daemon: a kind of special processes running on the background;
- Parent process: the forking process;
- Child process: the new process created by the fork System call;
- PID: Process Identifier, nonzero;
- Process group: consists of its parent, further ancestors, siblings, children, and further descendants;
- Signal: information, arguments, etc. that a process sends to another process.

Process Implementation

●Process table

a table maintained by the operating system to implement the process model. (One entry per process, which is called the process control block)

Process management	Memory management	File management
Registers	Pointer to text segment info	Root directory
Program counter	Pointer to data segment info	Working directory
Program status word	Pointer to stack segment info	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

Example fields in a typical process control block (PCB)

System Calls Related to Processes

System call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, opts)	Wait for a child to terminate
s = execve(name, argv, envp)	Replace a process' core image
exit(status)	Terminate process execution and return status
s = sigaction(sig, &act, &oldact)	Define action to take on signals
s = sigreturn(&context)	Return from a signal
s = sigprocmask(how, &set, &old)	Examine or change the signal mask
s = sigpending(set)	Get the set of blocked signals
s = sigsuspend(sigmask)	Replace the signal mask and suspend the process
s = kill(pid, sig)	Send a signal to a process
residual = alarm(seconds)	Set the alarm clock
s = pause( )	Suspend the caller until the next signal

s: error code; **pid:** process ID;

residual: the remaining time in the previous alarm

Process Creation in Linux

- The Fork system call creates an exact duplicate of the original process, the file descriptors, registers, and everything else are of the same in the parent and child processes; The PID is used to distinguish processes.

```
pid = fork( );                  /* if the fork succeeds, pid > 0 in the parent */
if (pid < 0) {
      handle_error( );          /* fork failed (e.g., memory or some table is full) */
} else if (pid > 0) {
                                /* parent code goes here. /*/
} else {
                                /* child code goes here. /*/
}
```

Use fork to create new process

Process Creation in Linux: Fork()

- The calling process traps to the kernel and creates a task structure and few other accompanying data structures;
- Linux then looks for an available PID, and updates the PID hash table entry to point to the new task structure;
- Memory is allocated for the child's data and stack segments, and exact copies of the parent's segments are made;
- The child process starts running.

POSIX Shell

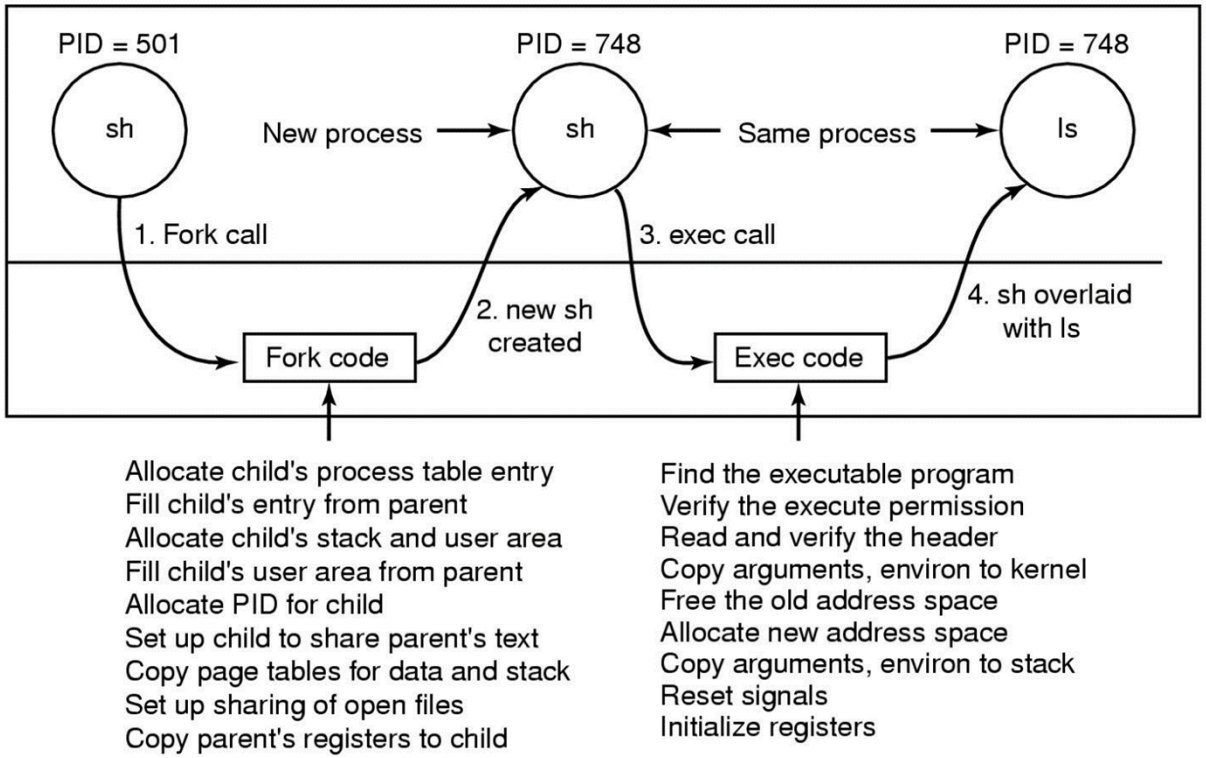
- An implementation of shell based on the system calls fork, waitpid and exec

```
while (TRUE) {                              /* repeat forever /*/
      type_prompt( );                       /* display prompt on the screen */
      read_command(command, params);        /* read input line from keyboard */

      pid = fork( );                        /* fork off a child process */
      if (pid < 0) {
            printf("Unable to fork0);       /* error condition */
            continue;                       /* repeat the loop */
      }

      if (pid != 0) {
            waitpid (–1, &status, 0);       /* parent waits for child */
      } else {
            execve(command, params, 0);     /* child does the work */
      }
}
```

A simplified shell

ls Commands

- The shell creates a new process by forking off a clone of itself. The new shell then calls exec to overlay its memory with the contents of the executable file *ls*



The steps in executing the command *ls* typed to the shell

Thread

- Question: What is thread? What are the differences between processes and threads?

✓Threads are parts of processes;
✓Threads are the smallest units that CPU schedules;
✓A process contains one thread or multiple threads;
✓Threads in the process share resources and data
✓The cost of thread creation/switching is low

POSIX Threads

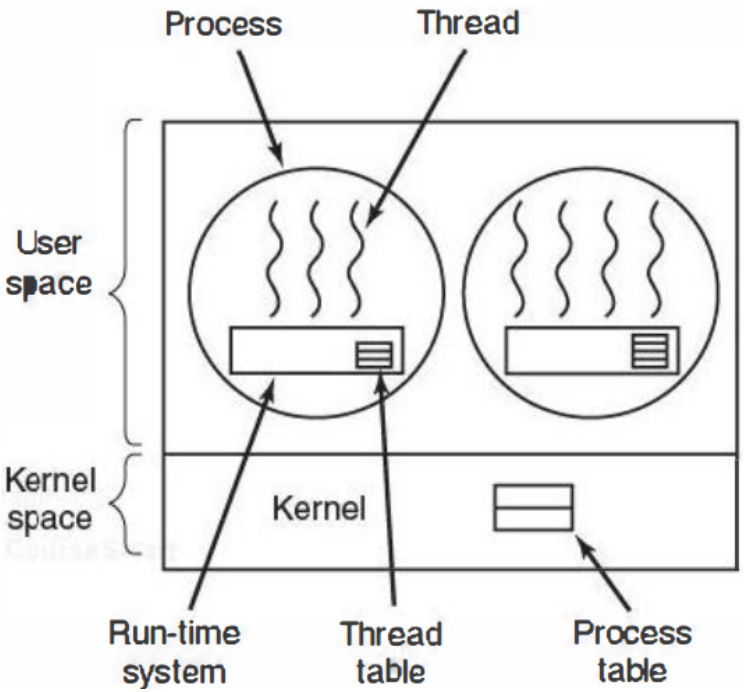
Thread call	Description
pthread_create	Create a new thread in the caller's address space
pthread_exit	Terminate the calling thread
pthread_join	Wait for a thread to terminate
pthread_mutex_init	Create a new mutex
pthread_mutex_destroy	Destroy a mutex
pthread_mutex_lock	Lock a mutex
pthread_mutex_unlock	Unlock a mutex
pthread_cond_init	Create a condition variable
pthread_cond_destroy	Destroy a condition variable
pthread_cond_wait	Wait on a condition variable
pthread_cond_signal	Release one thread waiting on a condition variable

Calling interfaces of POSIX threads

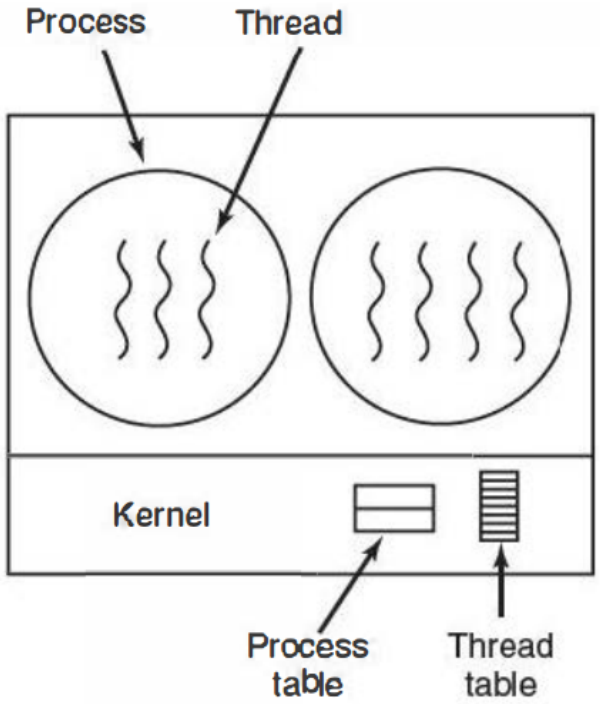
Linux Thread Creation

```
#include<stdio.h>
#include<pthread.h>
#include<stdlib.h>
void *thfun(void *arg)
{
    printf("new  thread! %s\n", (char*)arg);
    return ((void *)0);
}
int main(int argc ,char *argv[])
{   pthread_t pthid;
    int ret=pthread_create(&pthid,NULL,thfun, (void *)"hello");
    if(ret!=0){
        perror("create thread failed");
        exit(EXIT_FAILURE);
    }
    printf("main thread!\n");
    sleep(1);
    return 0;
}
```

Thread Implementation



User-Lever Thread



Kernel-Level Thread

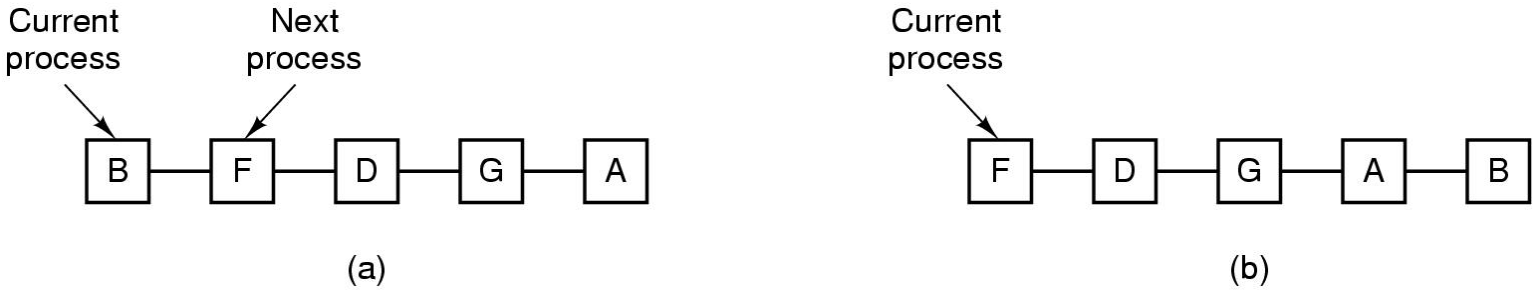
What are the advantages and disadvantages of user-level threads and kernel-level threads respectively?

Thread Scheduling

- Linux threads are kernel threads, so scheduling is based on threads, not processes.
- The common process/thread scheduling algorithms:
  - ✓First come first served
  - ✓Shortest job first
  - ✓Round Robin Scheduling
  - ✓Priority Scheduling
  - ✓…

Round Robin (RR) Scheduling

- Each process gets a small unit of CPU time (*time quantum*). After this time has elapsed, the process is preempted and added to the end of the ready queue.

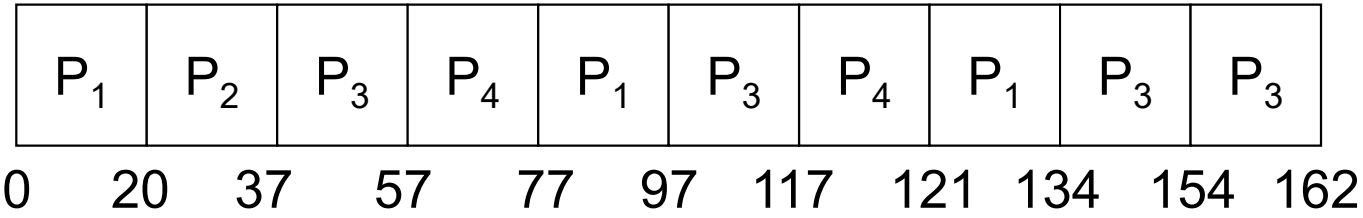


- If there are n processes in the ready queue and the time quantum is q, then each process gets $1/n$ of the CPU time. No process waits more than $(n\text{-}1)q$ time units.
- Performance
  - q large $\Rightarrow$ FIFO
  - q small $\Rightarrow$ q must be large with respect to context switch, otherwise overhead is too high.

Example of RR with Time Quantum = 20

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

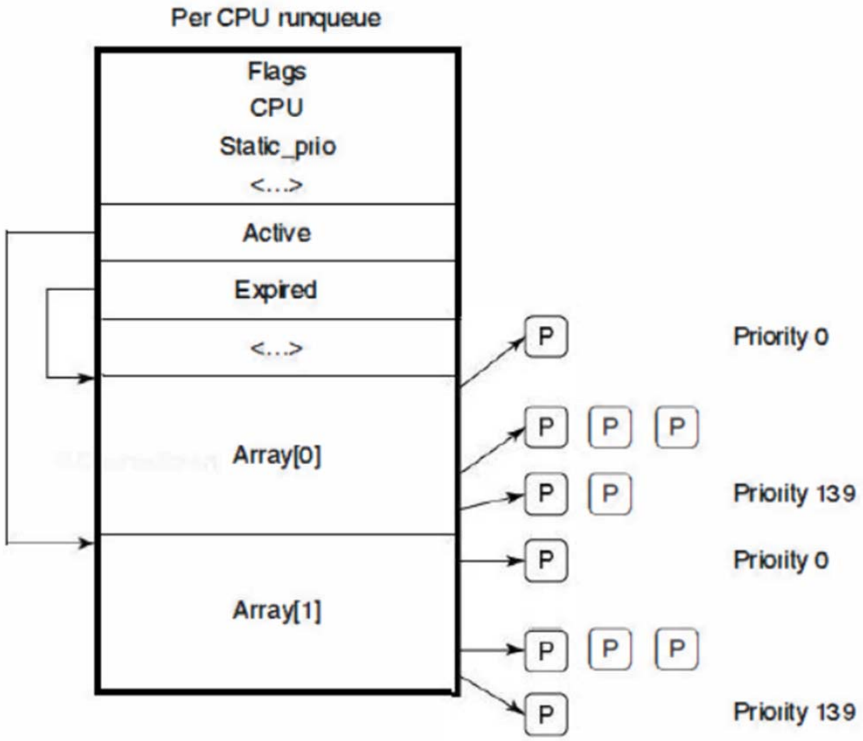
- The Gantt chart is:



- Typically, higher average turnaround than SJF, but better *response*.

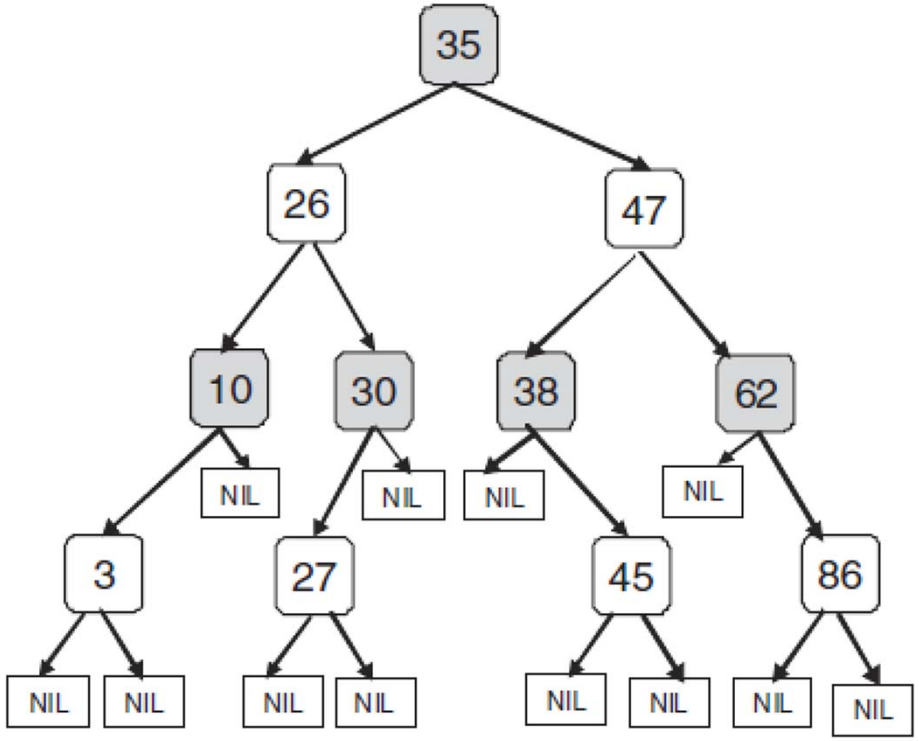
Scheduling in Linux: O(1) Scheduler

- Two kinds of queues: Active, Expired
- O(1) time complexity: select a task from the active queue with the highest priority. If the task's time quantum expires, it is moved to the expired list.
- Higher priority task has larger time quantum.



Completely Fair Scheduler (CFS) Scheduling

- Use a red-black tree as the runqueue data structure.



Red Black Tree

① Red black tree is a self balanced binary search tree, each node of which has a black or red color attribute.

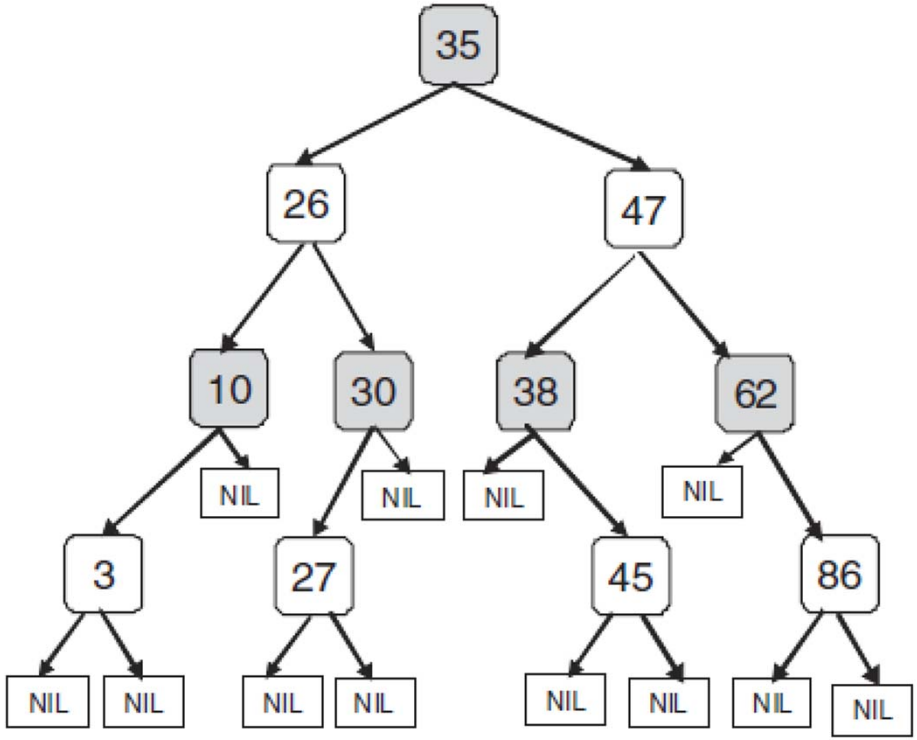
② Because it is a binary search tree, its search, insertion and deletion operations are based on the corresponding operations of binary search trees;

③ However, since the red black tree itself needs to ensure balance, it needs to make additional adjustments after each insertion and deletion to restore its own balance.

④ The time complexity of red black tree search, insertion and deletion can be guaranteed to be O(logn) in the worst case, where n is the number of elements in the tree.

Completely Fair Scheduler (CFS) Scheduling

- Use a red-black tree as the runqueue data structure;
- Always select the task with the smallest vruntime;
- Periodically increase the tasks' vruntime;
- Time complexity: O(log n); n is the number of tasks



Booting Linux

Step 1: BIOS performs Power-On-Self-Test

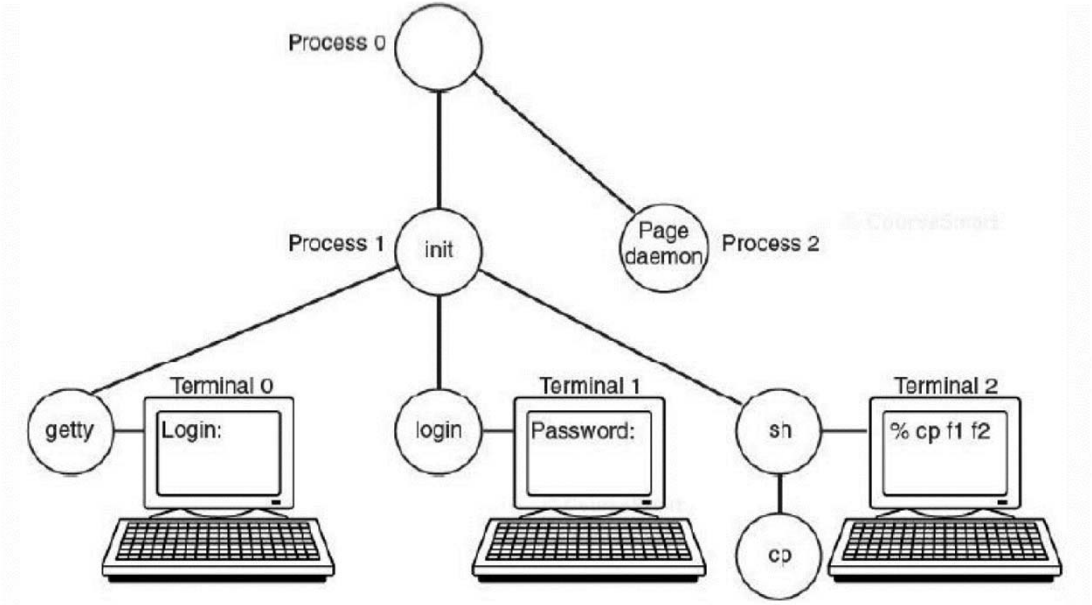
Step 2: The first sector of the disk (MBR, boot) is loaded to memory and is excuted.

Step 3: The boot load the operating system kernel and jumps to it.

Step 4: The kernel is running, start-up code (assembly language), do some initialization tasks.

Booting Linux

Step 1: Process 0 is created once all the hardware has been configuired;
Step 2: Process 0 does some initialization: e.g., mouting the root file system, create kernel thread init and page daemon;
Step 3: thread init creates the init Process to continue the initialization;
Step 4: Once the initialization is completed, the init Process calls “fork” to create a getty process;
Step 5: getty calls “exec” to run the login program, if success, then calls “exec” to create a shell.



The sequence of processes used to boot some Linux systems.

Check Points

- When did Unix fist appear?
- What are the relationships between Unix, Minix, and Linux?
- Two tasks A and B need to perform the same amount of work. However, task A has higher priority and needs to be given more CPU time. Explain how will this be achieved in O(1) Scheduler and CFS scheduler.
- When booting Linux, why not let the bootstrap loader in sector 0 just load the operating system directly?