

Processes and process management in Linux

Mohammad S. Hasan

Staffordshire University, UK

Lecture Outline

- ◆ **Process representation in Linux i.e. PCB**
- ◆ **Process creation in Linux**
- ◆ **System calls**
 - **Fork()**
 - **Exec()**
 - **Wait()**
 - **Exit()**
- ◆ **Process scheduling**

Process ID

- ◆ Every process has a unique PID which identifies it to system
- ◆ Init process has PID = 1
- ◆ All processes spawned from init or a descendent of init has a parent and hence the parent has a PID
- ◆ Each process can find out the PID of its parent
 - PPID (Parent Process ID)

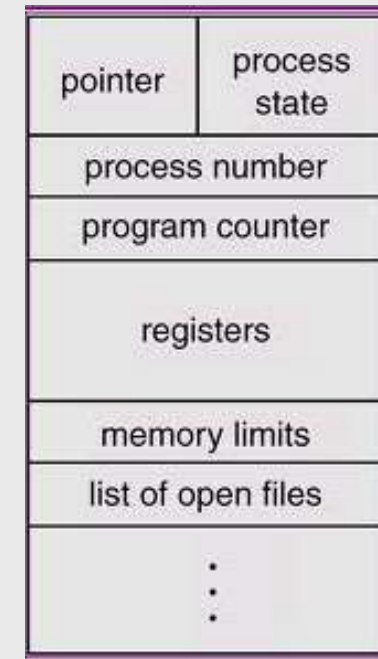
Process representation

- ◆ Processes in Linux are represented by a Process Control Block (process descriptor) like other OS
- ◆ Linux implements PCB as a “C struct” data structure
 - **task_struct**
- ◆ Each process is allocated a task_struct when the process is created
- ◆ Note – the following C struct contains only some of the more relevant fields – the real task_struct is larger

Struct task_struct

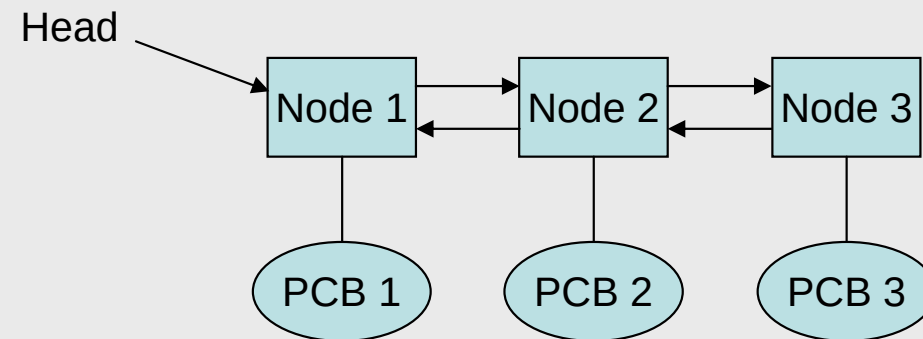
```

struct task_struct{
    volatile long
    long
    struct task_struct
    struct task_struct
    int
    int
    struct task_struct
    struct wait_queue
    struct task_struct
    unsigned long
    struct tms
    unsigned long
    unsigned short
    struct thread_struct
    struct files_struct
    struct mm_struct
    state;
    counter, priority;
    *next_task, *prev_task;
    *next_run, *prev_run;
    exit_code, exit_signal;
    pid;
    *p_opprt, *p_cptr;
    *wait_chldexit;
    *pidhash_next;
    policy;
    times;
    start_time;
    uid, gid;
    tss;
    *files;
    *mm;
}
    
```



Struct task_struct (cont.)

- ◆ The process structures are kept in a doubly linked list (pointers go to next and to previous element in list)
- ◆ We will consider these as they become relevant
 - ***next, *next_run**
 - ***prev, *prev_run**
 - ***pidhash_next**
 - **state, pid**



Struct task_struct (cont.)

- ◆ This is so that number of processes can grow dynamically without a specific upper limit
 - Easy to add and remove
- ◆ However linked lists are inefficient to search
 - Would require search sequentially from start of list
- ◆ So OS has a table pidhash
 - pidhash contains pointers to process's task_struct
 - *pidhash just points to next element in hash table

Struct task_struct (cont.)

- ◆ **State** – contains the process' state info – e.g.
 - **runnable** (in ready queue),
 - **interruptable** (waiting for an interrupt from some event e.g. I/O), etc.
- ◆ **Processes in runnable state are kept in the ready queue of processes which is formed by another linked list of the task_structs using**
 - ***next_run,**
 - ***prev_run**

Struct task_struct (cont.)

◆ Priority and counter

- used by scheduler (but will consider their use later in lecture)

◆ Pid

- process id number that uniquely identifies the process to the OS

◆ *p_opptr

- points to the original parent process of the current process

◆ *p_cptr

- points to the most recently produced child process of the process

Struct task_struct (cont.)

◆ Times

- points to a struct that holds info about the amount of time the process has run in user mode and time spent by the OS running in kernel

◆ start_time

- time process was created

◆ uid, gid

- id of user who created process and group id of user

Struct task_struct (cont.)

◆ Tss

- copy of processor registers saved when process not running

◆ *files

- points to data structure that contains information about files (but also any I/O stream) process has open

◆ *mm

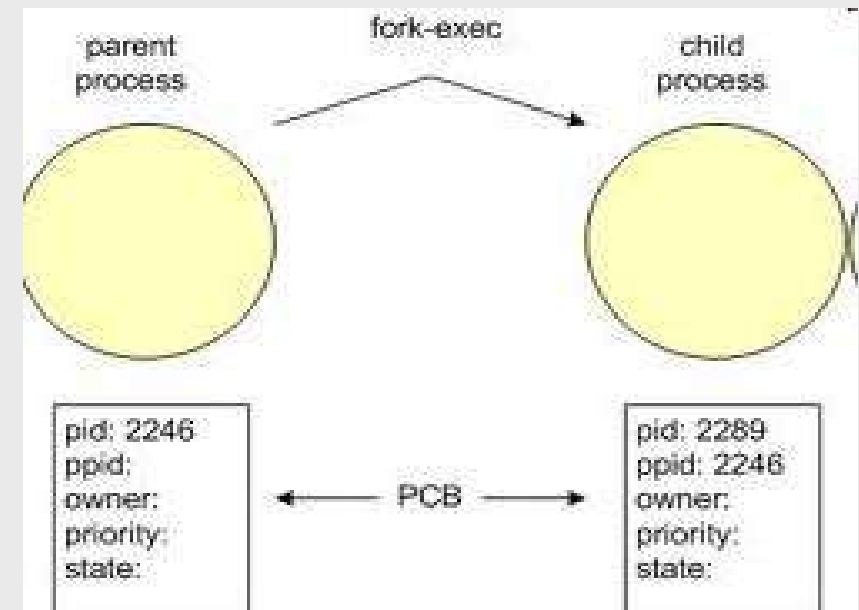
- points to data structure that has information about memory management

Process image

- ◆ **Reminder - in Linux the process image is**
 - **organisation of memory for a process**
- ◆ **Process image consists of (in a simplified form)**
 - **Text segment – machine code instructions of program**
 - **User data segment – initialised data and space for uninitialised data**
 - **System data segment – all the system data structures for the process**

Process creation

- ◆ The only way to create a new process is for an existing process to execute the *fork()* system call.
- ◆ This *fork()* call causes the calling process to be duplicated into 2 concurrent processes
 - a parent process and
 - child process.



Process creation (cont.)

- ◆ **The parent and child processes have**
 - **identical process text (program code)**
 - **Identical user data segments and**
 - **almost identical system data segments**
- ◆ **Process attributes NOT inherited by the child process from the parent include:**
 - **process ID (PID)**
 - **parent process ID (PPID)**
 - **Accumulated CPU time**

Process creation (Cont.)

- ◆ This inheritance allows attributes setup by the shell to be used by any executed commands
 - Working directory,
 - Effective User ID (EUID)
- ◆ Of particular interest is the inheritance of open file descriptors
- ◆ Because of this it is possible for several processes to be trying to take terminal input at the same time

Process creation (Cont.)

- After child process has been spawned both parent and child continue execution with the `fork()` call returning with a different value depending upon whether the process is the parent or the child process
 - PID of child process returned to parent and 0 to child

fork() example

```
classicfork()
```

```
{  
    pid_t forkval;  
  
    if (forkval = fork()) { /* nonzero child PID - parent */  
    {    /* parent code */  
    }  
    else { /* forkval == 0 - child */  
        /* child code */  
    }  
}
```

Exec()

- ◆ After a `fork()` call it is usual to run a new program in the child process slot
- ◆ The only way to run a new program is by making an `exec()` system call
- ◆ The `exec()` system call replaces text (code) and user data segments of calling process with text and data segments of program file passed as parameter to `exec()` call

Exec() (Cont.)

- ◆ In addition to a parameter that identifies program file, exec() family of calls requires as parameters the command line switches and parameters i.e. used to form the argv and argc of the program file being exec()-ed
 - e.g.
 - `execl("/bin/ls", "ls", "-l", 0);`

Waiting

- ◆ Once parent process has set a child running it has 2 possible choices
 - carry on its own execution
 - wait for child to terminate
- ◆ The latter choice uses the `wait()` system call
- ◆ If there are any child processes running, `wait()` sleeps until one of them terminates

Waiting (Cont.)

◆ **wait()**

- **returns the terminating process ID**
- **Also stores the exit status of the terminating process in the integer variable pointed to by 'status'**
- ◆ **If a parent process terminates before its children then they are adopted by process 1 (init)**

Waiting (Cont.)

- ◆ **A child that terminates before parent process executes `wait()` becomes a 'zombie' process**
 - **all resources except process data structure released back to system until parent executes `wait()` and**
 - **then exit status returned and child termination completed**

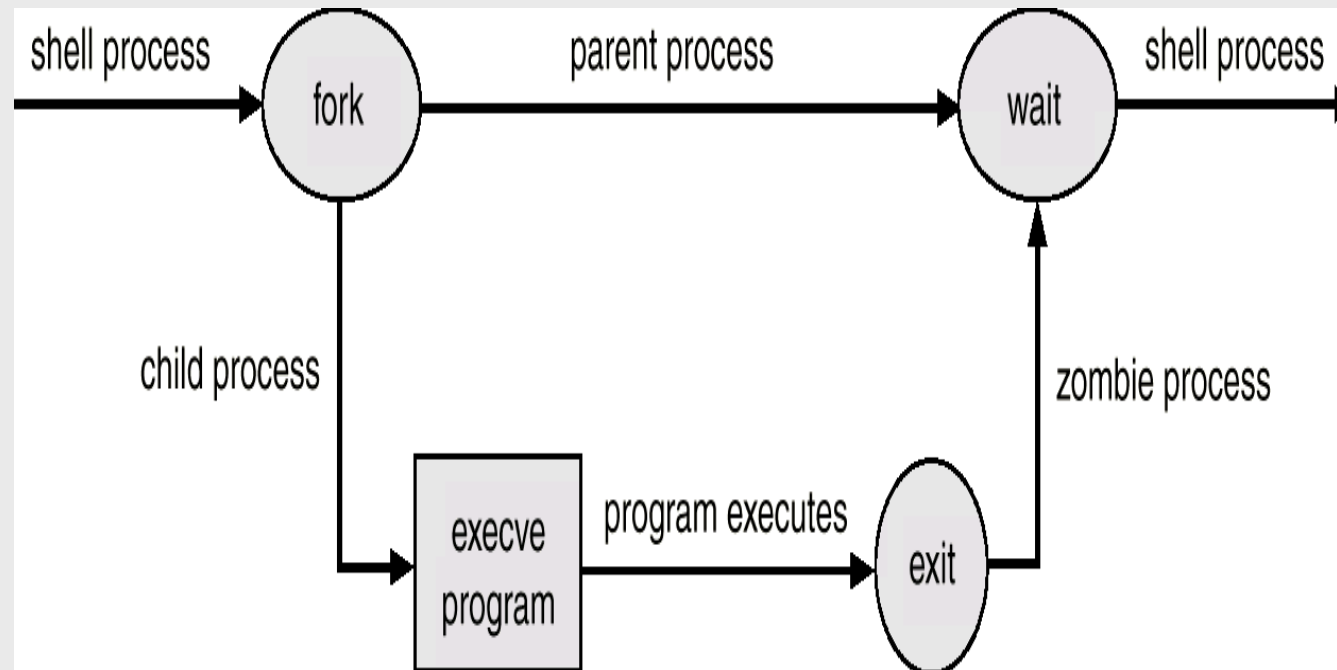
Exiting

- ◆ There are very few ways that a process can terminate.
- ◆ The most common is the `exit()` system call
 - `#include <stdlib.h>`
 - `void exit(int status) /* end process */`
- ◆ The status value is returned via the status pointer in a matching `wait()` system call
- ◆ The `exit()` system call is the only system call that **NEVER** returns

Exiting (cont.)

- ◆ In addition to terminating a process `exit()` also
 - flushes data buffers and
 - closes all open file descriptors
- ◆ An `exit()` status value is always returned even if none is specified. This would just make the result random garbage.
- ◆ Exit status value
 - Normally an exit status of zero indicates that the process has terminated without error.
 - Non zero values are user defined.

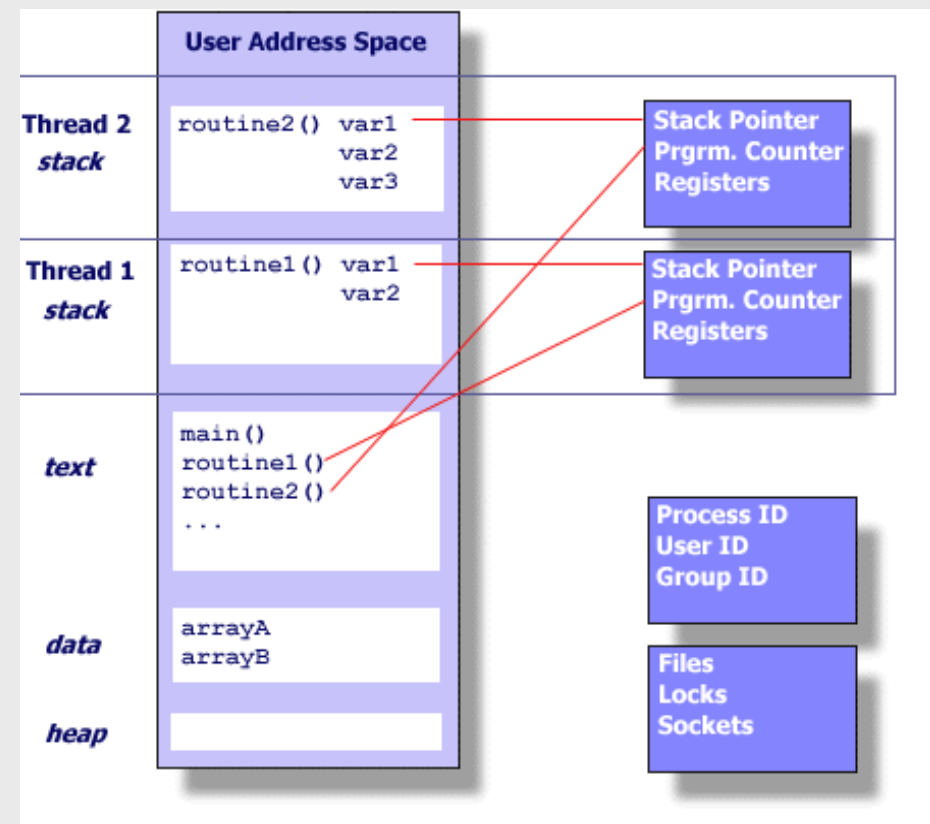
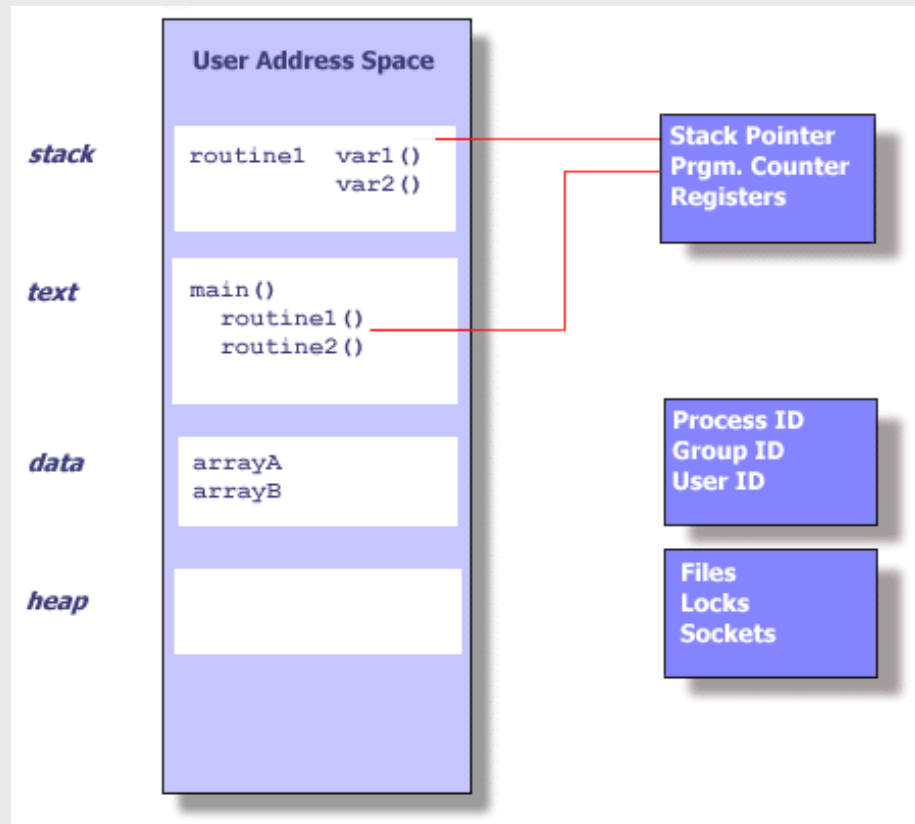
Illustration of Process Control Calls



Processes and Threads

- ◆ Linux uses the same internal representation for processes and threads;
 - A thread is simply a new process that happens to share the same address space as its parent.
- ◆ A distinction is only made when a new thread is created by the clone system call.
- ◆ Example
 - Process: Microsoft Word
 - Threads: Spell check, Auto save etc.

Processes and Threads (Cont.)



Processes and Threads (Cont.)

◆ Creating process

- **fork creates a new process with its own entirely new process context**

◆ Creating thread

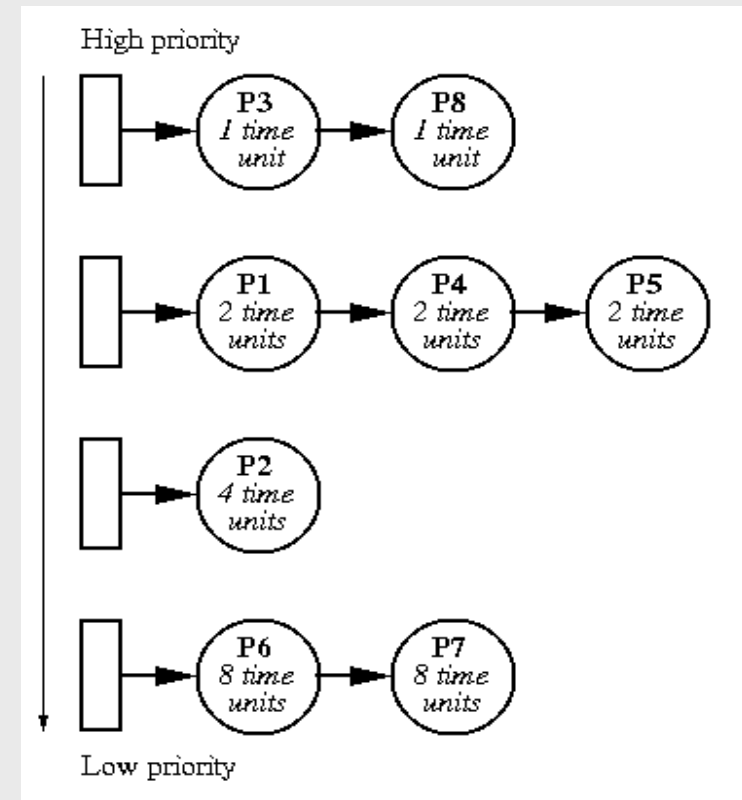
- **clone creates a new process with its own identity, but that is allowed to share the data structures of its parent**
- ◆ **Using clone gives an application fine-grained control over exactly what is shared between two threads.**

Scheduling in Linux

- ◆ While scheduling is normally thought of as managing the running of processes, in Linux, scheduling also includes the running of various kernel tasks.
- ◆ Running kernel tasks encompasses both tasks that are requested by a running process and tasks that execute internally on behalf of the process.

Scheduling in Linux (Cont.)

- ◆ Linux uses two process-scheduling algorithms:
 - A time-sharing algorithm for fair preemptive scheduling between multiple processes
 - A real-time algorithm for tasks where absolute priorities are more important than fairness
- ◆ Example
 - Microsoft Word and
 - A temperature control process



Scheduling in Linux (Cont.)

- ◆ A process's scheduling class defines which algorithm to apply.
- ◆ For time-sharing processes, Linux uses a prioritised, credit based algorithm.
- ◆ Counter is the value found in the `task_struct` described earlier
- ◆ Counter value starts with value of time quantum for the process

Scheduling in Linux (Cont.)

- ◆ The counter is decremented while
 - process runs until it reaches 0 (quantum expires) or
 - it makes a system call (when counter will hold time remaining of quantum)
- ◆ When scheduler is called it schedules the process with highest counter value (time quantum remaining)
- ◆ If all counter values are 0 then it recalculates counter value using a crediting rule

$$\text{counter} := \frac{\text{counter}}{2} + \text{priority}$$

Scheduling in Linux (Cont.)

◆ Counter value explained

- Processes that have had their time quantum expire will have a counter value of 0,
- but processes that have made a system call before the time quantum expires will have a counter value > 0 , but they will initially be blocked waiting for some event (I/O typically) i.e. not be on ready queue

Scheduling in Linux (Cont.)

- ◆ While they are waiting for I/O service, the crediting rule will still be applied to them so that their counter value will grow slowly while they wait and be larger, the longer they are waiting
- ◆ Once the I/O event has happened they will be put back on the ready queue but with a high counter value so that the scheduling algorithm will favour I/O bound jobs.

Scheduling in Linux (Cont.)

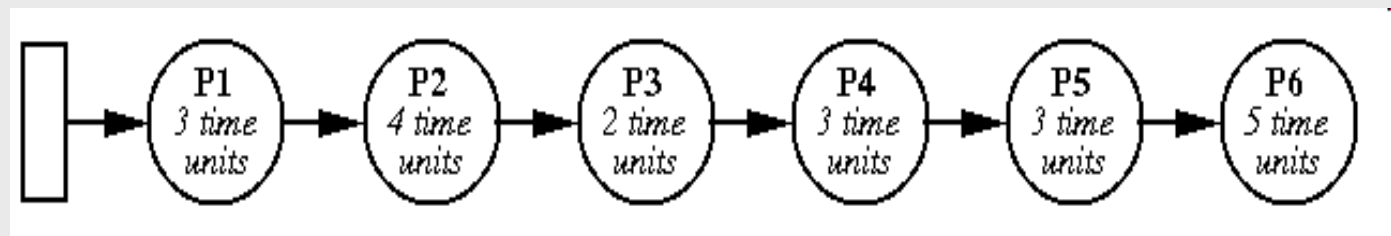
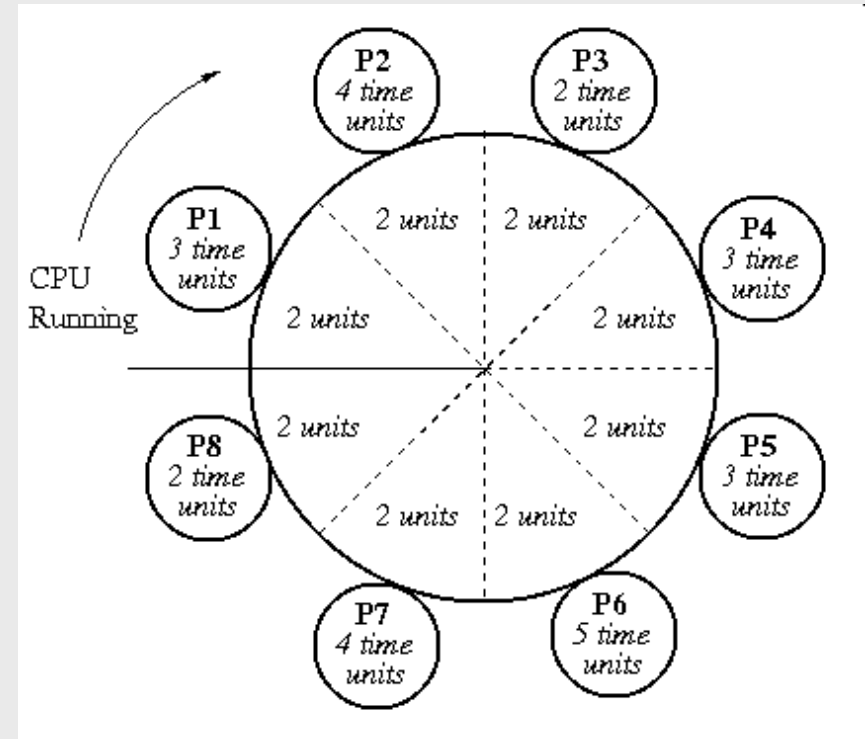
- ◆ The crediting rule factors in both the process's history and its priority.

Scheduling in Linux (Cont.)

- ◆ Linux implements the FIFO and round-robin real-time scheduling classes;
 - in both cases, each process has a fixed priority in addition to its scheduling class.
- ◆ The scheduler runs the process with the highest priority; for equal-priority processes, it runs the longest-waiting one

Scheduling in Linux (Cont.)

- ❖ **FIFO processes continue to run until they either exit or block**
- ❖ **A round-robin process will be preempted after a while and moved to the end of the scheduling queue, so that round-robin processes of equal priority automatically time-shares between themselves.**



References

➤ **Operating System Concepts. Chapter 22.**