Round-Robin Scheduling (RR)

- Designed especially for time-sharing systems.
- Similar to FCFS scheduling, but preemption is added to switch between processes.
- A **time slice** is defined, usually between 10 and 100 milliseconds. The ready queue is treated as a circular queue, the scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time slice
- To implement the RR scheduling, we keep the ready queue as a FIFO queue. New processes are added at the tail. The scheduler picks the first process from the queue, sets a timer to interrupt after one time slice, and dispatches the process.
- One of two things will happen:
  1. The process has a CPU burst of less than 1 time slice. Then, the process itself will release the CPU voluntarily.
  2. The timer will go off, and cause an interrupt. A context switch will be executed, and the process will be put at the tail of the ready queue.

- If the time slice $= \infty$, then RR=FCFS.
- If the time slice is very small (1 microsecond), then the RR approach is called **Processor Sharing**, and "appears" as if each of n processes had its own processor, running at the speed of $1/n$ of the real processor (in theory, because we have to take into account context switches—we want time slices to be bigger than contex-switch time).

Linux

Linux provides two separate process-scheduling algorithms:

1. A time-sharing algorithm for fair preemptive scheduling among multiple processes.
2. An algorithm for real-time tasks where absolute priorities are more important than fairness.

Linux allows only processes running in user mode to be preempted. A process may not be preempted while it is running in kernel mode, even if a real-time process with a higher priority is available to run.

In Linux, part of every PCB is a scheduling class, that defines which of the two algorithms to apply to the process.

For time-shared processes, Linux uses a prioritized, credit-based algorithm. Each process possesses a certain number of scheduling credits.

When a new task must be chosen to run, the process with the most credits is selected. Every time that a time interrupt occurs, the currently running process looses one credit. When the number of credits reaches zero, it is suspended and another process is chosen.

If no runnable processes have any credits, then Linux performs a recrediting operation, adding credits to every process in the system (rather than just to the runable ones), according to the following rule:

$$\text{credits} = \frac{\text{credits}}{2} + \text{priority}$$

This algorithm mixes two factors:

1. history
2. priority

One-half of the credits that a process still holds since the previous recrediting operation will be retained after the algorithm has been applied. This way, it retains some history of recent behaviour.

Background batch jobs can be given a low priority. Interactive, or I/O bound processes, for which a rapid response time is important, can be given a higher priority.

Exercises: 6.1, 6.2, 6.3, 6.4, 6.5, 6.7.

6.1 Given n processes to be scheduled on one processor, how many different schedules are possible?

$n!$

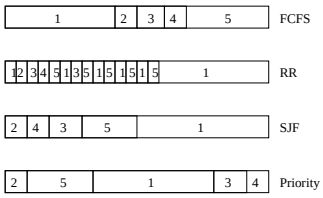
6.2 Define the difference between preemptive and nonpreemptive scheduling.

Preemptive scheduling allows a process to be interrupted in the midst of its execution, taking the CPU away and allocating it to another process. Nonpreemptive scheduling ensures that a process relinquishes control of the CPU only when it finishes with its current CPU burst.

6.3(a) Consider the following set of processes:

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	3
P_4	1	4
P_5	5	2

Draw four Gantt charts illustrating the execution of these processes using FCFS, SJF, a nonpreemptive priority (a smaller priority number implies a higher priority), and RR (time slice = 1) scheduling.



The UNIX File System

- Everything in the UNIX system is a file.
- When the first version of the system was being designed, before it even had a name, the discussion focused on the structure of a file system that would be clean and easy to use.
- The file system is central to the success and convenience of UNIX. It is a good example of the "keep it simple" philosophy, and the power achieved by careful implementation of a few well chosen ideas.
- Details of using the file system:
  –what files are
  –how they are represented
  –directories and the file system hierarchy
  –permissions
  –inodes (system's internal record of files)
  –device files

- A file is a sequence of bytes. No structure is imposed on the file by the system, and no meaning is attached to its contents—the meaning of the bytes depends solely on the program that interprets the file.
- This is true not just of disc files, but of peripheral devices as well. Magnetic tapes, mail messages, characters typed on the keyboard, line printer output, data flowing in pipes—each of these files is just a sequence of bytes as far as the system and the programs in it are concerned.

Create a file:

```
~$ ed
a
now is the time
for all good people
.
w junk
36
q
~$ ls -l junk
-rw-rw-r--    1 you   you    36 Mar  9 07:56 junk
~$
```

junk is a file with 36 bytes—the 36 characters typed while appending. To see the file:

```
~$ cat junk
now is the time
for all good people
~$
```

The command od (octal dump) prints a visible representation of all the bytes of a file:

```
~$ od -c junk
0000000   n   o   w       i   s       t   h   e
0000020   f   o   r       a   l   l       g   o
0000040   p   l   e  \n
0000044
~$
```

- The 7-digit numbers down the left side are positions in the file, that is, the ordinal numbers of the next character shown, in octal. (The emphasis on octal numbers is a holdover from the PDP-11, for which octal was a preferred notation.)

- Each line ends with a character with octal value 012. This is the ASCII newline character; it is what the system places in the input when you press the RETURN key. By a convention borrowed from C, the character representation of a newline is `\n`. But this is just a convention to make it easy to read—the value stored in the file is the single byte 012.

- Other special characters include backspace (octal value 010, printed as `\b`), tab (011, `\t`), etc.

- It is important to distinguish between how a character is stored in a file and how it is interpreted in various situations. For example, when you type a backspace on your keyboard, the kernel intrprets it to mean that you want to discard whatever character you typed previously. Both that character and the backspace disappear, but the backspace is echoed to your terminal, where it makes the cursor move one position backward. (You can insert a backspace character in a file by typing `\←` .)

- The UNIX system is unusual in its approach to representing control information, particularly its use of newlines to terminate lines.

- Many systems provide instead "records," one per line, each of which contains not only your data but also a count of the number of characters in the line (and no newline).

- Other systems terminate each line with a carriage return and a newline, because that sequence is necessary for output on most terminals. (The word "linefeed" is a synonym for newline, so this sequence is often called "CRLF.")

- The UNIX system does neither—there are no records, no record counts, and no bytes in any file that you or your programs did not put there. A newline is expanded into a carriage return and a newline when sent to a terminal, but programs need only deal witht he single newline character, because that is all they see.

- The kernel keeps track of file lengths, so there is no need to add a special end-of-file character at the end; a program encounters end-of-file when it has processed all the bytes in the file.

- Programs retrieve the data in a file by a system call (a subroutine in the kernel) called

  `read`

  Each time `read` is called, it returns the next part of a file. It also says how many bytes of the file were returned, so end of file is assumed when 0 bytes are returned.

- This scheme makes a lot of sense: all files must end. Since all files must be accessed through `read`, returning zero is an interpretation independent way to represent the end of file without introducing a new special character.

- **What's in a file?** The format of a file is determined by the program that uses it; it is not determined by the file system. Thus, the kernel cannot tell you the type of a file, because it does not know it.

- The command `file` makes an educated guess:

```
~$ file /bin /bin/ed junk
/bin:    directory
/bin/ed: ELF 32-bit LSB executable, Intel 80386, version 1,
dynamically linked (uses shared libs), stripped
junk:    ASCII English text
~$
```


- To determine the file types, `file` didn't pay any attention to the file names (although it could have), because naming conventions are just conventions.

- Instead, `file` reads the first few hundred bytes of a file and looks for clues to the file type. Sometimes the clues are obvious: a runnable program is marked by a binary "magic number" at its beginning. `od` with no options dumps the file in 16-bit (2-byte) words and makes the magic number visible:

```
~$ od /bin/ed
0000000 042577 043114 000401 000001 000000 000000 000000 000000
0000020 000002 000003 000001 000000 110600 004004 000064 000000
0000040 014464 000001 000000 000000 000064 000040 000006 000050
...
```

  The octal value 42577 marks an executable. Specific magic numbers are system dependent.

- For text files, the clues may be deeper in the file, so `file` looks for words line `#include` to identify C source, or line beginning with a period to identify `nroff` or `troff` input.

Directories and Filenames

- All files have unambigous names. If you have a file `/home/you/junk` and you type `ls junk`, then you obtain: `junk`, not `/home/you/junk`, why?
- Because each process has a **current directory**, and all file names are implicitly assumed to start with the name of that directory, unless they begin directly with a slash. Your login shell and `ls`, therefore have a curent directory. The command `pwd` (print working directory) identifies the current directory:

```
$ pwd
/home/you
$
```

- The current directory is an attribute of a process, not a person or a program—people have login directories, processes have current directories. If a process creates a child process, the child inherits the current directory of its parent. If the child changes to a new directory, the parent is unaffected.
- The notion of current directory is a notational convenience, but its real purpose is organizational.

Permissions

- Every file has a set of **permissions** associated with it, which determine who can do what with the file. However, there is a special user, the **super-user**, who can read and modify any file in the system, (login name `root`.)
- You can, however, encrypt files in your directory (using PGP, for example). It is believed that in real life, most security breaches are due to passwords that are given away or easily guessed.
- When you log in, you type a name and a password, The system actually recognizes you by your user-id, **uid**. Besides a uid you are assigned a **group-id**, which places you in a class of users. The file system, and the UNIX system in general, determines what you can do by the permissions granted to your uid and group-id.

- The file `/etc/passwd` is the password file; it contains all the login information about each user. You can discover your uid and group-id, as does the system, by looking up your name in `/etc/passwd`:

```
$ grep you /etc/passwd
you:gkmbCTrJ04COM:500:500:You:/home/you:/bin/bash
$
```

  The fields in the password file are separated by colons and are laid out like this:

  login-id:encrypted-password:uid:group-id:miscellany:login-directory:shell

- The file is ordinary text, but the field definitions and separator are a convention agreed upon by the programs that use the information in this file.
- Note that your password appears here in an encrypted form, since anyone can read this file. When you give your password at login, it is encrypted and compared with the results in `/etc/passwd`.
-
```
~$ ls -l /etc/passwd
-rw-r--r-- 1 root root 1271 Feb 12 12:03 /etc/passwd
~$
```


- Anybody can open `/etc/passwd` and make changes, but only root can write the changes back. The program to change passwords is called `passwd` and:

```
~$ ls -l /usr/bin/passwd
-r-s--x--x 1 root root 13476 Aug 7 2001 /usr/bin/passwd
~$
```

- The `s` instead of `x` in the execute field for the file owner states that, when the command is run, it is to be given the permissions corresponding to the file owner, in this case root. Because `passwd` is "set-uid" to root, any user can run the `passwd` command to edit the password file.
- The set-uid bit is a simple and elegant idea (patented by Dennis Ritchie) that solves a number of security problems. For example, the author of a game program can make the program set-uid to the owner, so that it can update a score file that is otherwise protected form other users' access.

- But the set-uid concept is potentially dangerous. `passwd` has to be correct; if it were not, it could destroy system information under root's auspices. If, for example, it had the permissions:

  ```
  -rwsrwxrwx
  ```

  it could be overwritten by any user, who could therefore replace the file with a program that does anything.

- The set-uid bit is powerful, but used primarily for a few system programs such as `passwd`. A more ordinary file:

  ```
  ~$ ls -l /usr/bin/who
  -rwxr-xr-x 1 root root 14364 Jul 23 2001 /usr/bin/who
  ~$
  ```

  is executable by everybody, and writable by root and the group root. When you type `who` to the shell, the shell "finds" the file `who`, and checks if it has execute permission. If it does, the shell calls the kernel to run it.

- Note that a program is just a file with execute permission.

- Directory permissions operate a little differently, but the basic idea is the same:

  ```
  ~/somedir$ ls -ld .
  drwxrwx--x 46 you you 4096 Jan 31 11:48 .
  ~/somedir$
  ```

  - The leading `d` signifies that `.` is indeed a directory.
  - An `r` means that you can read the directory, with the `ls` command, for example.
  - A `w` means that you can create and delete files in this directory, because that requires modifying and therefore writing the directory file. (There are system calls that create and remove files, and only through them is it possible to change the contents of a directory—the `w` tells you who can use these system calls.)
  - Permission to remove a file is independent of the file itself. If you have write permission in a directory, you may remove files there, even files that are protected against writing.
  - The `x` field in the permissions on a directory does not mean execution; it means "search." The mode `--x` for other users means that users may access any file that they know about in that directory, but may not run `ls` on it or read it (with `od`) to see what files are there.

- The `chmod` command changes permissions on files (the permission descriptions can be provided by symbolic descriptions, or by octal numbers). Only the owner of the file (and the su) may change permissions on a file.

- If a directory is writable, people can remove files in it regardless of the permissions on the files themselves. Remove write permission from directory to make sure no one can delete files from it.

Inodes

- A file has several components: a name, contents, adm info, etc. The adm info is store in the inode (originally, "i-node"), along with essential system data such as how long it is, where on the disc the contents of the file are stored, and so on.

- There are three times in the inode:
  - time when the contents of the file were last modified
  - time when the file was last used
  - time the inode itself was last changes (e.g., to reset permissions)

- The system's internal name for a file is its i-number: the number of the inode holding the file's information. `ls -i` reports the i-numbers in decimal.

- The `ln` command makes a link to an existing file, with the syntax: `ln old-file new-file`. The purpose of a link is to give two names to the same file, often so that it can appear in two different directories. Two links to a file point to the same inode, and hence have the same i-number:

  ```
  $ ls -li junk
  1531917 -rw-rw-r-- 1 you you 8 Mar 12 08:30 junk
  $ ln junk trash
  $ ls -li
  1531917 -rw-rw-r-- 2 you you 8 Mar 12 08:30 junk
  1531917 -rw-rw-r-- 2 you you 8 Mar 12 08:30 trash
  ```

  Each link is equally important, and there is no difference between the first link, and subsequent ones (the "first link" is junk!).

- On the other hand:

  ```
  $ ls -li
  1531917 -rw-rw-r-- 1 you you 8 Mar 12 08:30 junk
  $ cp junk trash
  $ ls -li
  1531917 -rw-rw-r-- 1 you you 8 Mar 12 08:30 junk
  1531916 -rw-rw-r-- 1 you you 8 Mar 12 08:36 trash
  ```

Devices

- One of the nicest ideas in the UNIX system is the way it deals with peripherals—discs, tape drives, printers, terminals, etc. Rather than having special routines to, for example, read magnetic tape, there is a file called `/dev/mt0`. Inside the kernel, references to that file are converted into hardware commands to access the tape, so if a program reads `/dev/mt0`, the contents of a tape mounted on the drive are returned.

- For example,

  ```
  $ cp /dev/mt0 junk
  ```

  copies the contents of the tape to a file called `junk`. `cp` has no idea there is anything special about `/dev/mt0`; it is just a file—a sequence of bytes.


```
69445 crw-rw---- 1 root uucp  4, 66 Aug 30  2001 ttyS2
70258 lrwxrwxrwx 1 root root      8 Nov 15 06:36 cdrom -> /dev/hdc
65683 brw------- 1 root disk 22,  0 Aug 30  2001 hdc
65649 brw-rw---- 1 root disk  3,  0 Aug 30  2001 hda
66109 crw-r----- 1 root kmem  1,  2 Aug 30  2001 kmem
65242 crw-r----- 1 root kmem  1,  1 Aug 30  2001 mem
69064 crw--w---- 1 root root  4,  0 Mar 12 07:17 tty0
69065 crw------- 1 root root  4,  1 Mar 12 07:18 tty1
```

For a device file, the inode contains the internal name for the device, which consists of:

- Its type: character (c) or block (b). For example, disks and tapes are block devices, and terminals and modems are character devices.

- A pair of numbers, called the major and minor device numbers. The major number encodes the type of device, while the minor number distinguishes different instances of the device.

- All files accessible in a UNIX system are arranged in one big tree, the file hierarchy, rooted at `/`. These files can be spread out over several devices. The `mount` command serves to attach the file system found on some device to the big file tree. Conversely, the `umount` command will detach it again.

- For example:

  ```
  mount -t iso9660 /dev/cdrom /mnt/cdrom
  ```