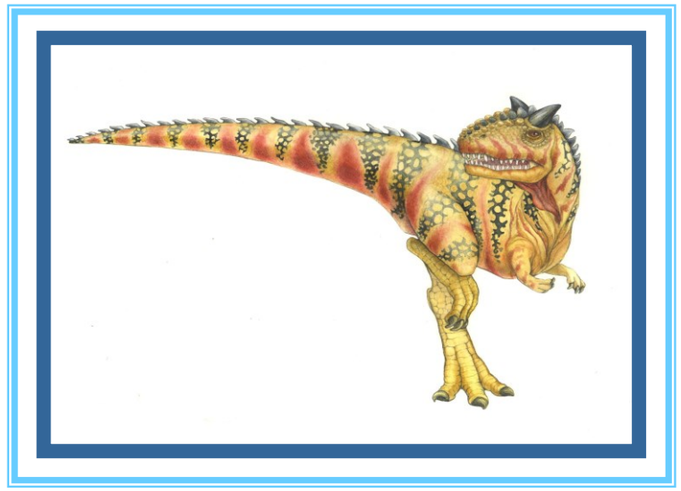
Chapter 5
Process Scheduling

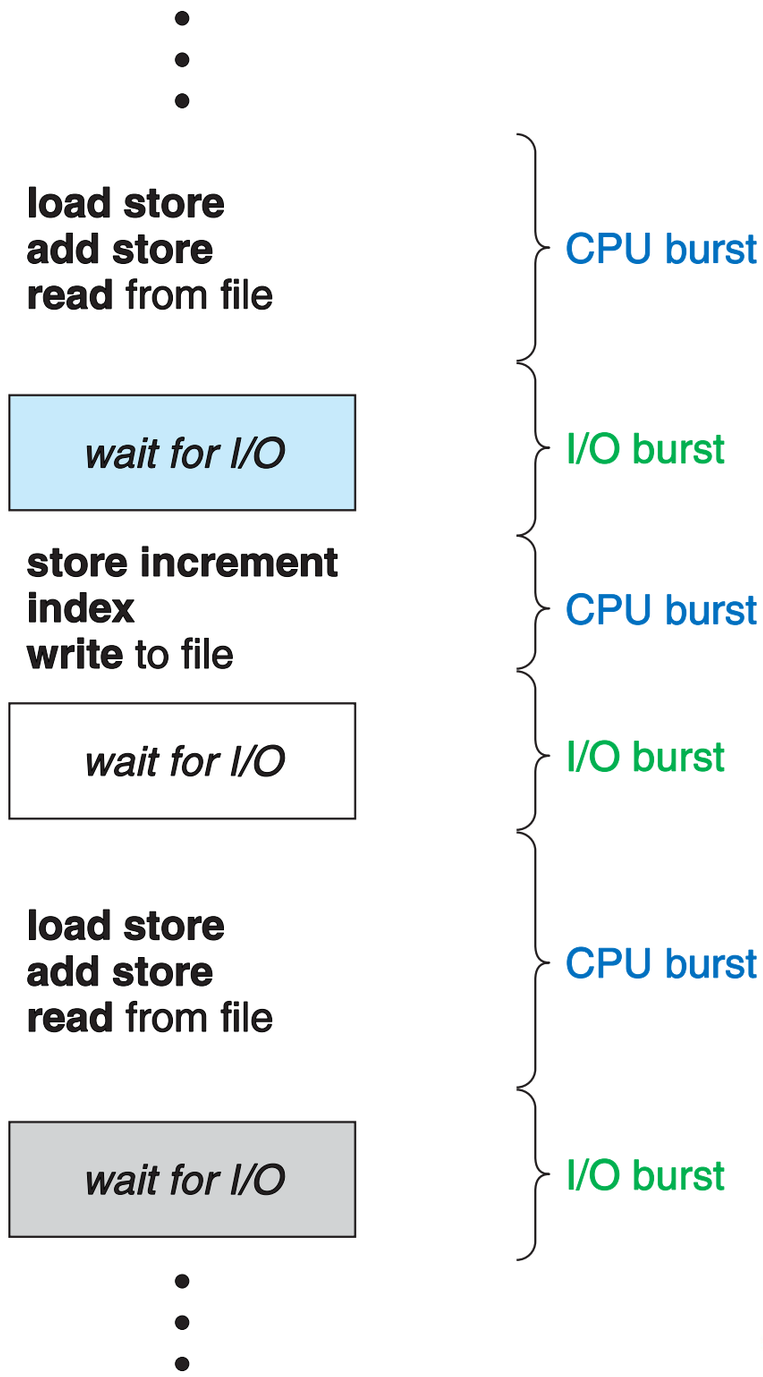


Objectives

- To introduce CPU scheduling, which is the basis for multiprogrammed operating systems
- To describe various CPU-scheduling algorithms
- To discuss evaluation criteria for selecting a CPU-scheduling algorithm for a particular system
- To examine the scheduling algorithms of operating systems

Basic Concepts

- In a single processor, only one process can run at a time.
- CPU – I/O Burst Cycle
  - Process execution consists of a cycle of CPU execution and I/O wait
  - **CPU burst** followed by **I/O burst**
- Multiprogramming
  - While a process has to wait for I/O, CPU can be **rescheduled** to run another process.
  - The aim is to **maximize CPU utilization.**
- Scheduling is a fundamental OS function.



- To demonstrate multiple processes sharing a processor
- Linux '**htop**' command
- Select a process and type '**a**' to set "**processor affinity**" to a processor (always schedule process to this CPU)

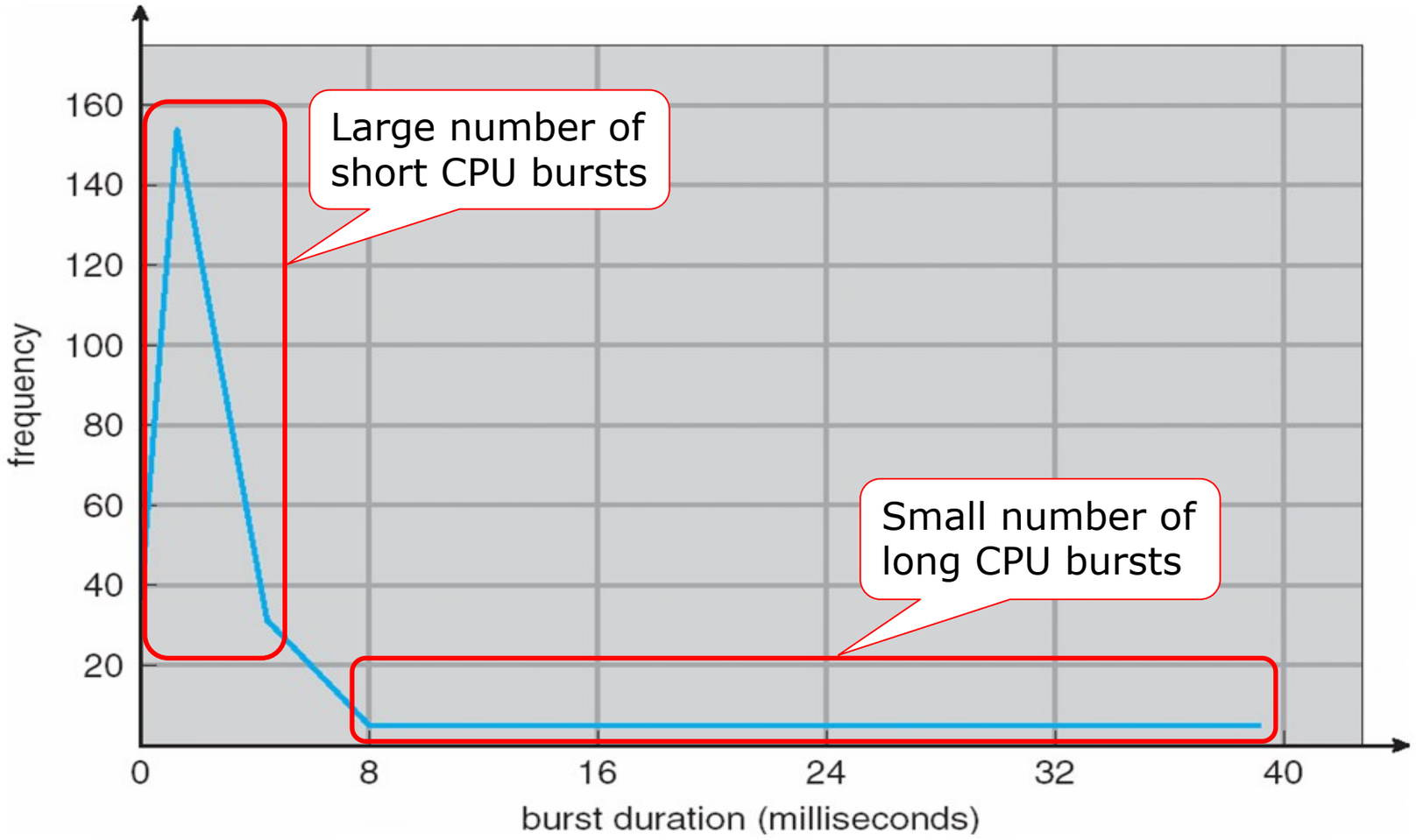
Use up-down arrows to select CPU and 'space' to toggle.

```
  0[                     0.0%]   4[                     0.0%]     8[|                    0.7%]  12[|                    4.1%]
  1[||||||||||||||||100.0%]   5[|                    0.7%]     9[                     0.0%]  13[                     0.0%]
  2[|                    1.3%]   6[                     0.0%]    10[                     0.0%]  14[|                    4.8%]
  3[                     0.0%]   7[                     0.0%]    11[                     0.0%]  15[                     0.0%]
Mem[||||||                                          385M/6.62G]   Tasks: 11, 2 thr; 2 running
Swp[                                                  0K/2.00G]   Load average: 1.13 0.96 0.49
                                                                  Uptime: 03:21:20
```

```
Use CPUs:
[ ] CPU 0
[x] CPU 1
[ ] CPU 2
[ ] CPU 3
[ ] CPU 4
[ ] CPU 5
[ ] CPU 6
[ ] CPU 7
[ ] CPU 8
[ ] CPU 9
[ ] CPU 10
[ ] CPU 11
[ ] CPU 12
[ ] CPU 13
[ ] CPU 14
[ ] CPU 15
```

```
  PID△USER       PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
    1 root        20   0  2448  1612  1500 S  0.0  0.0  0:00.01 /init
    4 root        20   0  2472   196   196 S  0.0  0.0  0:00.01 ├─ plan9 --control-socket 5 --log-level 4 --s
    5 root        20   0  2472   196   196 S  0.0  0.0  0:00.00 │  └─ plan9 --control-socket 5 --log-level 4
    6 root        20   0  2448  1612  1500 S  0.0  0.0  0:00.00 ├─ /init
  427 root        20   0  2452   112     0 S  0.0  0.0  0:00.00 ├─ /init
  428 root        20   0  2468   116     0 S  0.0  0.0  0:00.04 │  └─ /init
  429 veera       20   0  6208  5096  3308 S  0.0  0.1  0:00.01 │     └─ -bash
  440 veera       20   0  5672  3844  3036 R  0.0  0.1  0:00.40 │        └─ htop
  443 root        20   0  2452   112     0 S  0.0  0.0  0:00.00 └─ /init
  444 root        20   0  2468   116     0 S  1.3  0.0  0:05.67    └─ /init
  445 veera       20   0  6724  5752  3416 S  0.0  0.1  0:00.04       └─ -bash
  471 veera       20   0  2640   936   844 R 79.5  0.0  6:49.31          ├─ ./infinite
  472 veera       20   0  2772   932   840 S 21.4  0.0  1:00.30          └─ ./printloop
```

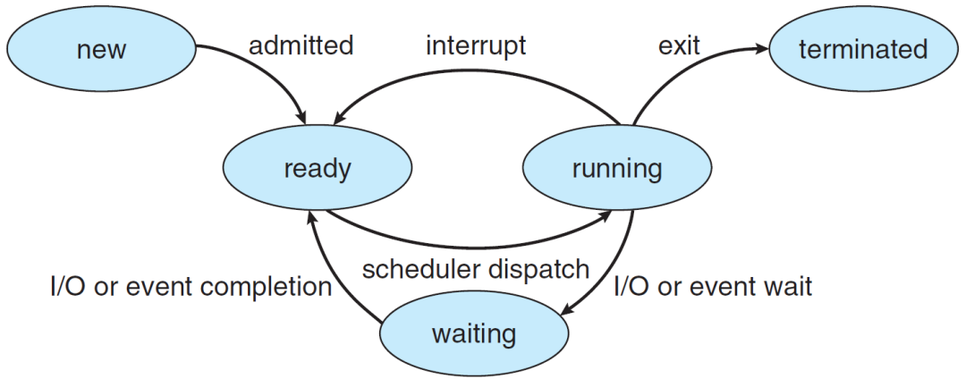
Histogram of CPU-burst Times



- Process with a lot of long CPU bursts → CPU-bound process
- Process with a lot of short CPU bursts → I/O-bound process

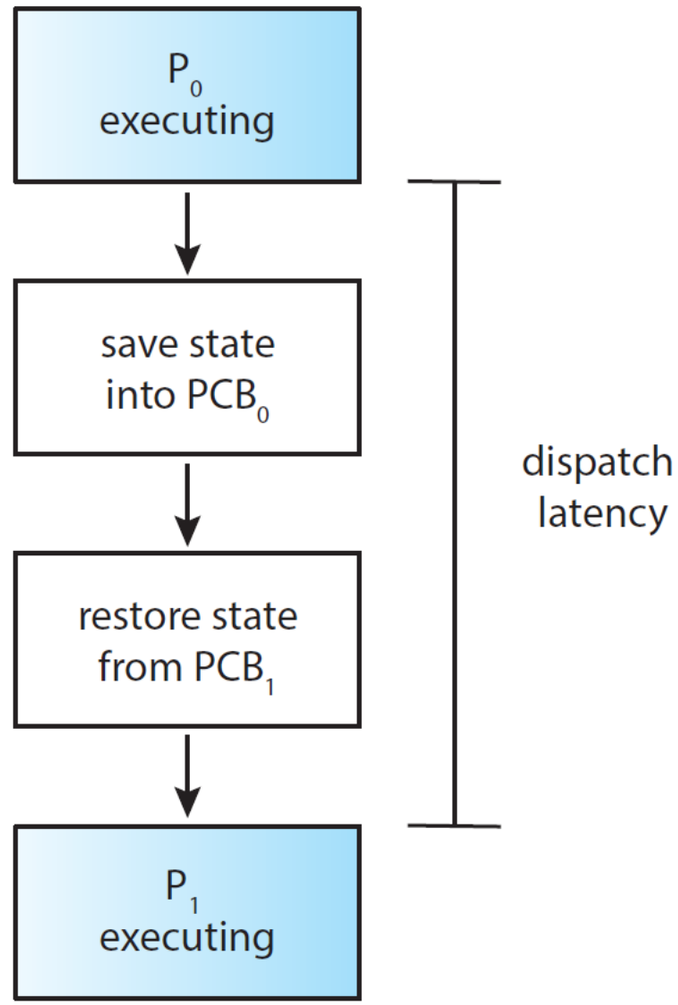
CPU Scheduler

- When CPU become idle, OS uses the **short-term scheduler** to selects one of the processes in the **ready queue** for execution.
  - Queue may be ordered in various ways
- CPU scheduling may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates
- Scheduling under 1 and 4 is **nonpreemptive** (the running process leaves the CPU by itself)
- Scheduling under 2 and 3 is **preemptive** (the OS forces the running process to leave the CPU)
- Most operating systems support preemptive scheduling. They must consider:
  - Access to shared data → race condition (see *process synchronization* lecture)
  - Preemption during crucial OS activities → kernel data structures must be managed carefully



Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running



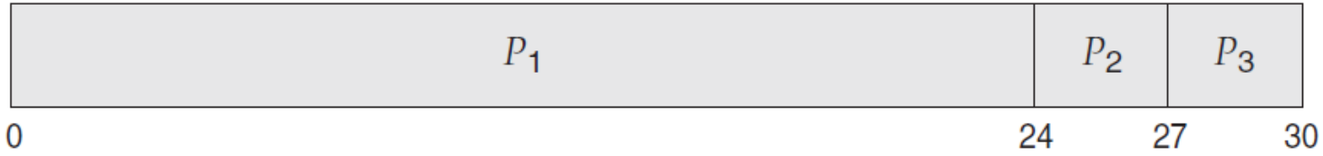
Scheduling Criteria

Criteria	Definition	Goal
CPU utilization	The % of time the CPU is executing user level process code.	Maximize
Throughput	Number of processes that complete their execution per time unit	Maximize
Turnaround time	Amount of time to execute a particular process	Minimize
Waiting time	Amount of time a process has been waiting in the ready queue	Minimize
Response time	Amount of time it takes from when a request was submitted until the first response is produced	Minimize

First-Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3
  The Gantt Chart for the schedule is:



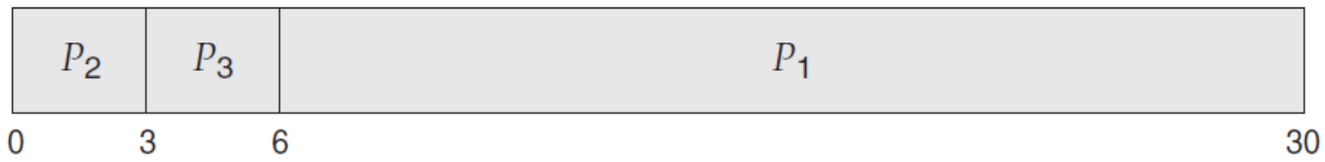
- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order:

$$P_2 , P_3 , P_1$$

- The Gantt chart for the schedule is:



- Waiting time for $P_1 = 6; P_2 = 0; P_3 = 3$
- Average waiting time: (6 + 0 + 3)/3 = 3
- Much better than previous case

First-Come, First-Served (FCFS) Scheduling

- **Convoy Effect**
  - Many I/O-bound processes with short CPU bursts waiting behind a long CPU-bound process

When I/O-bound processes with multiple short CPU bursts arrive before a CPU bound process.

P_2	P_3	P_1

The I/O-bound processes finish their CPU bursts quickly go back to wait behind the CPU-bound process.

P_1	P_2	P_3

Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst
- Schedule the process with the shortest CPU burst

- SJF is **optimal** – gives minimum average waiting time for a given set of processes

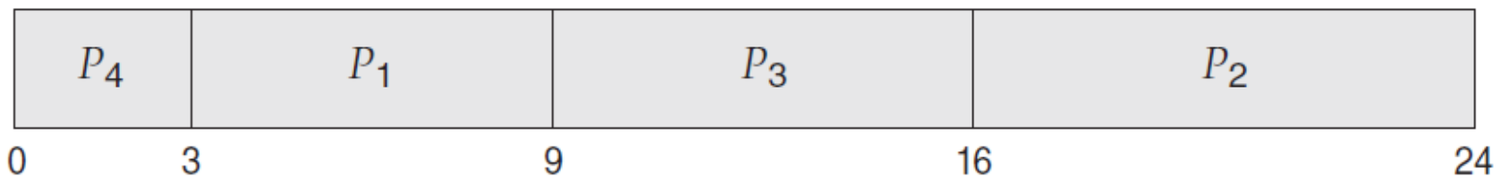
- SJF could lead to starving processes
  - Process with long CPU burst waits for processes with shorter bursts

- SJF is hard to implement
  - Problem: how to know the length of the next CPU request
  - Solution: predict the next CPU burst based on previous ones

Example of SJF

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart



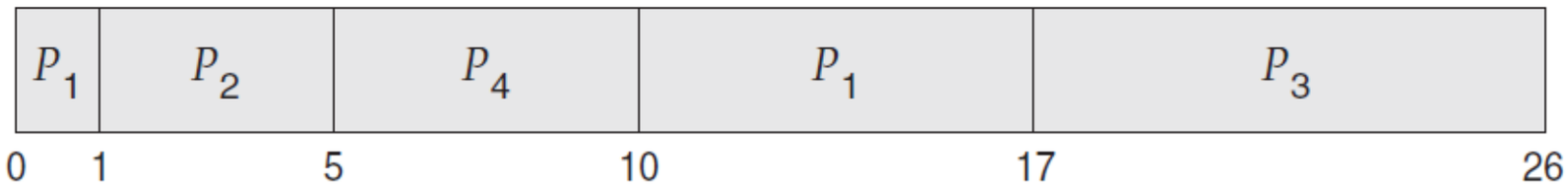
- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

Example of Shortest-remaining-time-first

- Now we add the concepts of varying arrival times and **preemption** to the analysis

Process	*Arrival* Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- *Preemptive* SJF Gantt Chart



- Average waiting time = [(10-1)+(1-1)+(17-2)+(5-3)]/4 = 26/4 = 6.5 msec

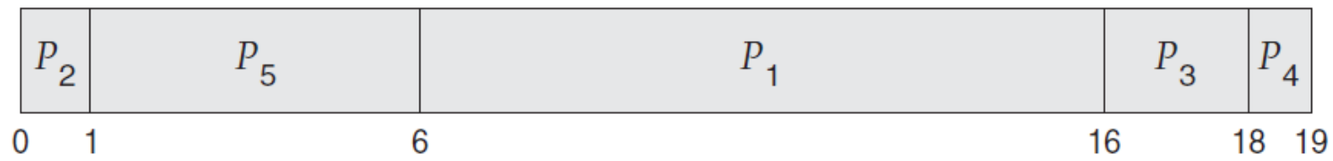
Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - Preemptive
  - Nonpreemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem: **Starvation** – low priority processes may never execute
- Solution: **Aging** – as time progresses increase the priority of the process

Example of Priority Scheduling

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

- Priority scheduling Gantt Chart



- Average waiting time = 8.2 msec

Round Robin (RR)

- Each process gets a small unit of CPU time (**time quantum** *q*), usually 10-100 milliseconds.
- After this time has elapsed, the process is preempted and added to the end of the ready queue.

- Timer interrupts every quantum to schedule next process.
- 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 in chunks of at most *q* time units at once. No process waits more than (*n*-1)*q* time units.

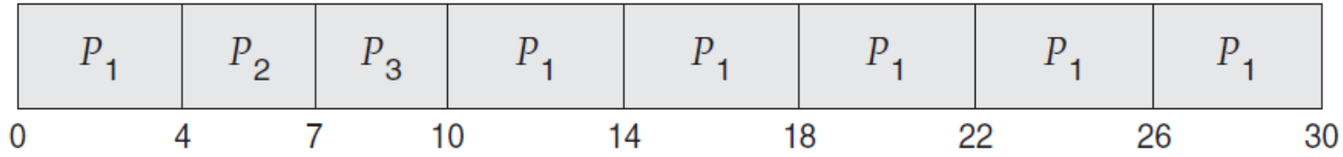
- RR is designed for time sharing systems.
- RR can prevent starvation.

Note: The term "round robin" actually came from "round ribbon".

Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

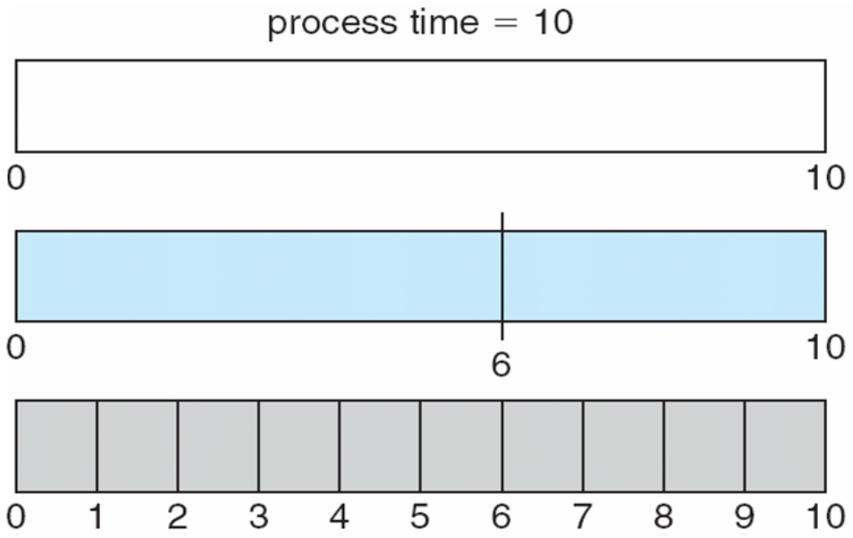
- The Gantt chart is:



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

Length of Time Quantum

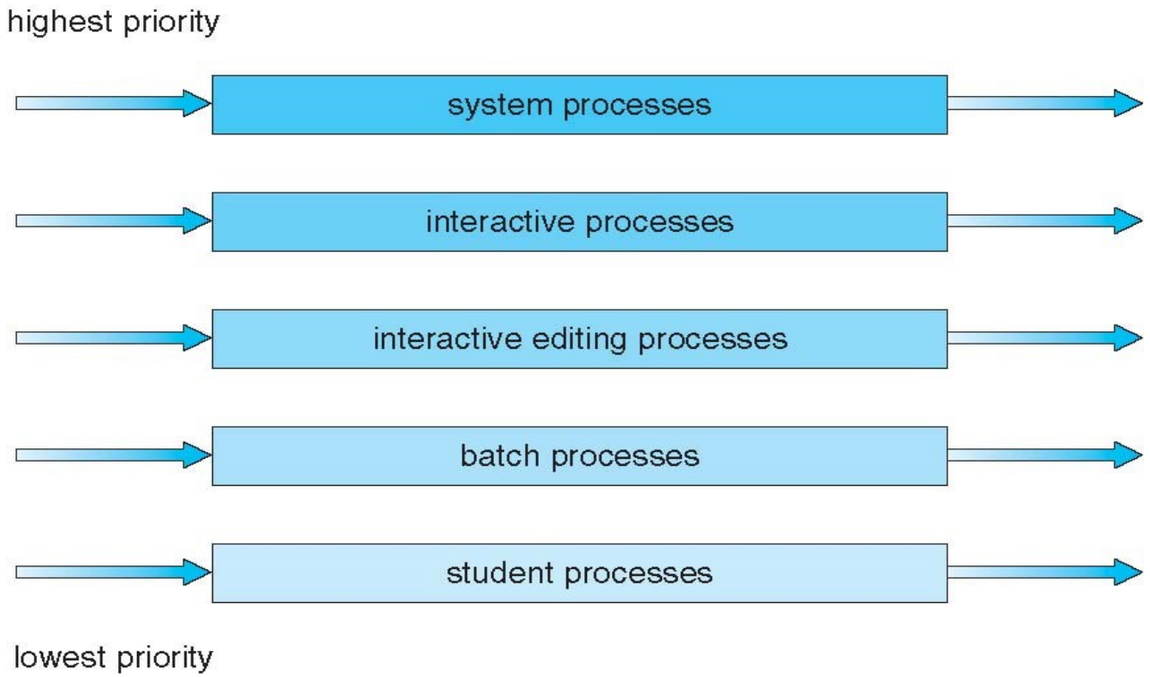
- If *q* is extremely large, RR becomes FIFO.
- Time quantum *wrt.* context switch
  - Small quantum results in a large number of context switches.
  - *q* must be large with respect to context switch time, otherwise overhead is too high.
- Time quantum *wrt.* CPU burst time
  - Average turn around time improves if most processes finishes next CPU burst in a single quantum.
  - *q* should be larger than 80% of CPU bursts.



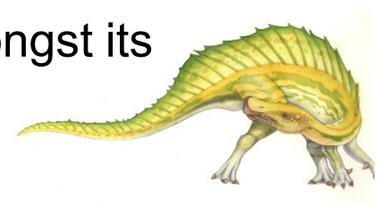
quantum	context switches
12	0
6	1
1	9

Multilevel Queue

- Ready queue is partitioned into separate queues, eg:
  - **foreground** (interactive)
  - **background** (batch)
- Processes are permanently assigned to one queue, based on some property (e.g. priority, process type, memory size)
- Each queue has its own scheduling algorithm:
  - foreground – RR
  - background – FCFS



- Scheduling must be done between the queues:
  - Fixed priority scheduling; (i.e., serve all from foreground then from background).
  - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR, 20% to background in FCFS



Thread Scheduling

- **Process-contention scope (PCS)** – scheduling among threads belonging to the same process
- **System-contention scope (SCS)** – scheduling among all threads in the system
- Systems using one-to-one model (e.g. Windows, Linux) uses only SCS.

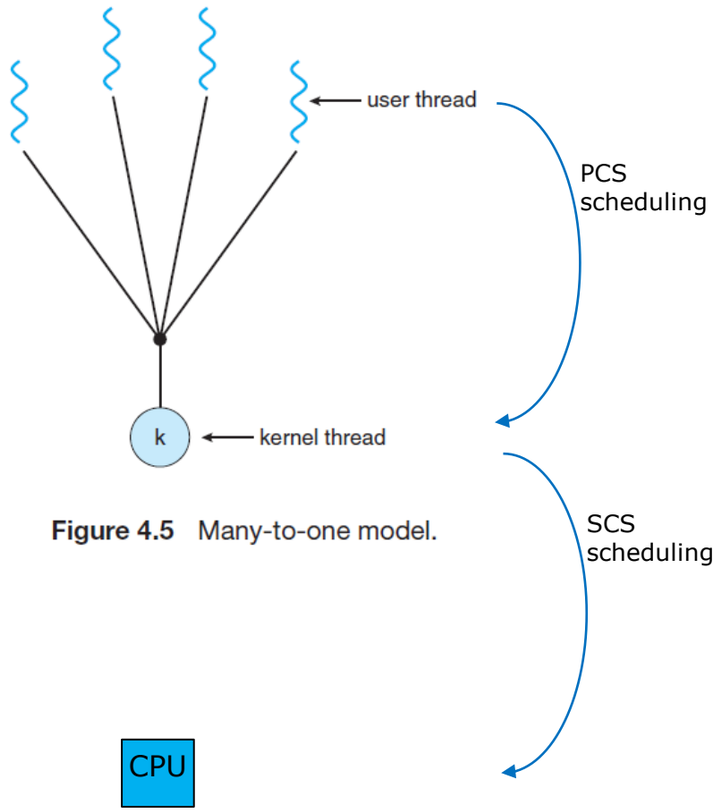


Figure 4.5 Many-to-one model.

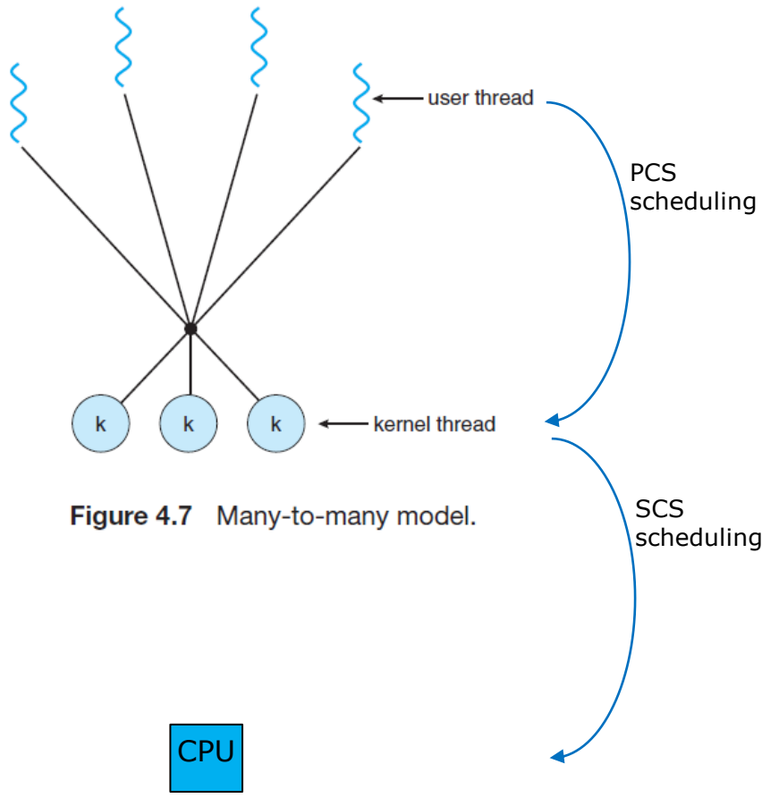


Figure 4.7 Many-to-many model.

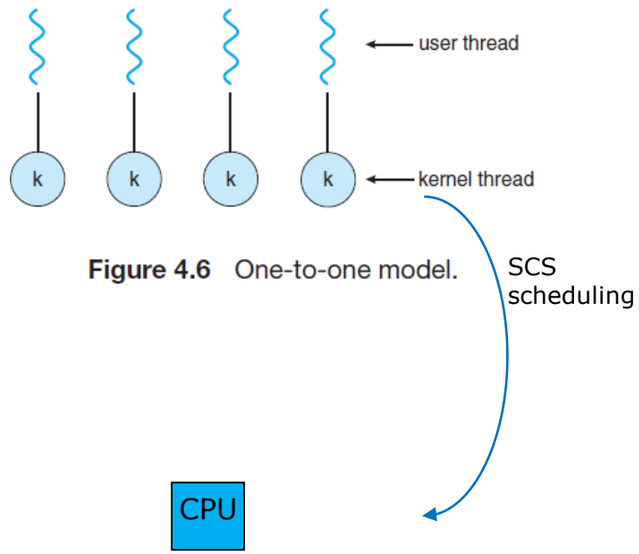
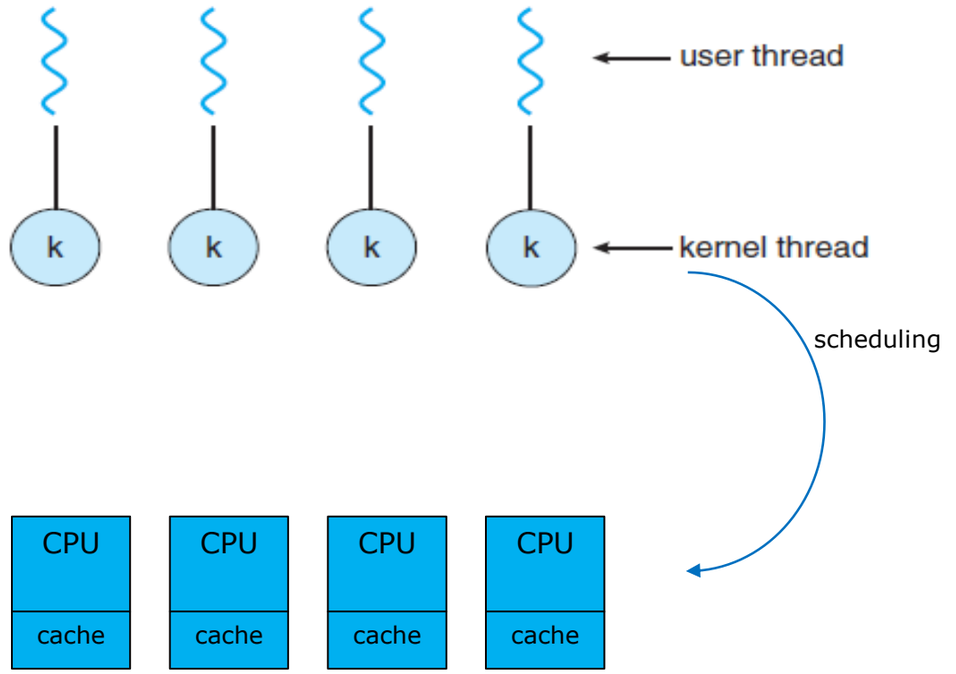


Figure 4.6 One-to-one model.

Multiple-Processor Scheduling

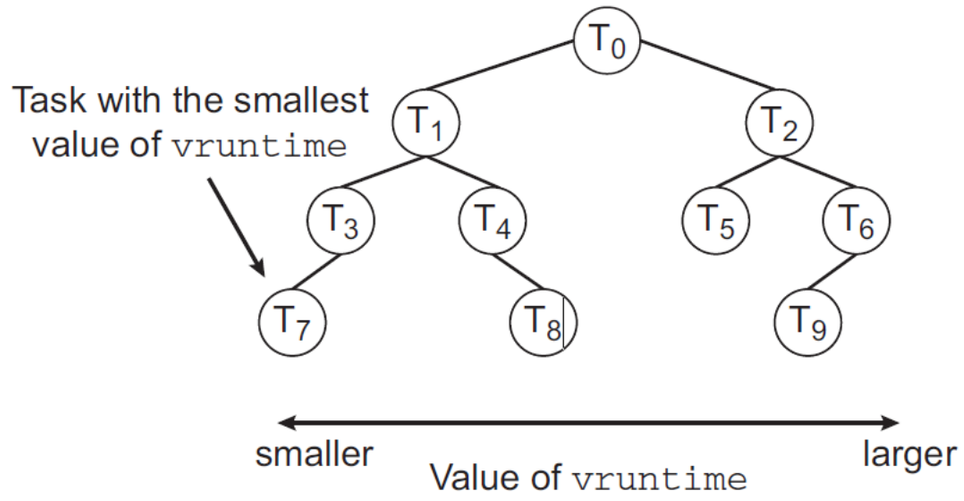
- With multi-core/multi-processor, parallel computing and load sharing become possible.
- Each processor has its own private queue of ready processes.
- Processor Affinity
  - Keep each process/thread running on the same processor.
  - To benefit from cache memory.



- Load balancing
  - Keep workload evenly distributed across all processors.
  - To benefit from multiple-processor.
  - Migrate processes from overloaded processors to less-busy processors.
- Load balancing often counteracts the benefits of process affinity.

Linux Scheduler

- Completely Fair Scheduler (CFS)
- CFS uses process priority and variable timeslices to schedule a process.
- **Scheduling classes**. Each class is assigned a specific priority. Different scheduling algorithms on different classes. To decide which task to run next, the scheduler selects the highest-priority task belonging to the highest-priority scheduling class.
- Higher priority process receives a higher time slice.
- CFS records how long each task has run (“virtual run time” or “vruntime”), which is calculated from actual runtime and a factor based on task priority. So, vruntime increases slowly for higher priority processes compared with lower priority processes.
- To decide which task to run next, the scheduler simply selects the task that has the smallest vruntime value.
- A higher-priority task can preempt a lower-priority task.
- Each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of vruntime.



DEMONSTRATION

Linux command: *ps*

```
To see every process on the system using standard syntax:
   ps -e
   ps -ef
   ps -eF
   ps -ely

To see every process on the system using BSD syntax:
   ps ax
   ps axu

To print a process tree:
   ps -ejH
   ps axjf

To get info about threads:
   ps -eLf
   ps axms
```

```
pom@X280:/mnt/c/Users/Pom$ ps axu
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.0  0.0   1744   432 ?        Sl   Feb17   0:00 /init
root         111  0.0  0.0   1764     0 ?        Ss   Feb17   0:00 /init
root         112  0.0  0.0   1764   116 ?        S    Feb17   0:00 /init
root         113  0.0  0.5 1088556 10428 pts/0   Ssl+ Feb17   0:21 /mnt/wsl/docker-desktop/docker-desktop-proxy --distro-name Ubuntu-20.04
root         119  0.0  0.0   1764     4 ?        S    Feb17   0:00 /init
pom          120  0.0  0.7 763732 14736 pts/1    Ssl+ Feb17   0:36 docker serve --address unix:///home/pom/.docker/run/docker-cli-api.sock
root         146  0.0  0.0   1764    40 ?        Ss   17:24   0:00 /init
root         147  0.0  0.0   1764    64 ?        S    17:24   0:00 /init
pom          148  0.0  0.2  10188  5300 pts/2    Ss   17:24   0:00 -bash
pom          389  0.0  0.1  10604  3300 pts/2    R+   18:11   0:00 ps axu
```

```
pom@X280:/mnt/c/Users/Pom$ ps axjf
 PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
    0     1     0     0 ?           -1 Sl       0   0:00 /init
    1   111   111   111 ?           -1 Ss       0   0:00 /init
  111   112   111   111 ?           -1 S        0   0:00  \_ /init
  112   113   113   113 pts/0      113 Ssl+     0   0:21  |   \_ /mnt/wsl/docker-desktop/docker-desktop-proxy --distro-name Ubuntu-20.04
  111   119   111   111 ?           -1 S        0   0:00  \_ /init
  119   120   120   120 pts/1      120 Ssl+  1000   0:36      \_ docker serve --address unix:///home/pom/.docker/run/docker-cli-api.sock
    1   146   146   146 ?           -1 Ss       0   0:00 /init
  146   147   146   146 ?           -1 S        0   0:00  \_ /init
  147   148   148   148 pts/2      397 Ss    1000   0:00      \_ -bash
  148   397   397   148 pts/2      397 R+    1000   0:00          \_ ps axjf
```

- Information about process is kept in ***/proc/<pid>***

```
pom@X280:/mnt/c/Users/Pom$ ls /proc
1    146        bus         crypto       filesystems  kallsyms     kpagecount  misc          partitions   swaps        uptime
111  147        cgroups     devices      fs           kcore        kpageflags  modules       sched_debug  sys          version
112  148        cmdline     diskstats    interrupts   key-users    loadavg     mounts        schedstat    sysvipc      vmallocinfo
113  401        config.gz   dma          iomem        keys         locks       mtrr          self         thread-self  vmstat
119  acpi       consoles    driver       ioports      kmsg         mdstat      net           softirqs     timer_list   zoneinfo
120  buddyinfo  cpuinfo     execdomains  irq          kpagecgroup  meminfo     pagetypeinfo  stat         tty
```

```
pom@X280:/mnt/c/Users/Pom$ ls /proc/148
arch_status  cmdline          environ  io          mountinfo   oom_adj        projid_map  smaps          status         uid_map
attr         comm             exe      limits      mounts      oom_score      root        smaps_rollup   syscall        wchan
auxv         coredump_filter  fd       map_files   mountstats  oom_score_adj  sched       stack          task
cgroup       cpuset           fdinfo   maps        net         pagemap        schedstat   stat           timers
clear_refs   cwd              gid_map  mem         ns          personality    setgroups   statm          timerslack_ns
```

Context Switches

Run “top” command in one console.

```
pom@X280:/mnt/c/Users/Pom$ top -d 1
```

Update every 1 second

```
top - 18:50:33 up 4 days, 12:32,  0 users,  load average: 0.24, 0.28, 0.31
Tasks:  13 total,   1 running,  12 sleeping,   0 stopped,   0 zombie
%Cpu(s):  1.0 us,  2.0 sy,  0.0 ni, 96.5 id,  0.0 wa,  0.0 hi,  0.5 si,  0.0 st
MiB Mem :   1916.6 total,    104.7 free,   1230.5 used,    581.4 buff/cache
MiB Swap:   1024.0 total,    229.8 free,    794.2 used.    366.1 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
    1 root      20   0    1744    432    396 S   0.0   0.0   0:00.02 init
  111 root      20   0    1764      0      0 S   0.0   0.0   0:00.00 init
  112 root      20   0    1764    116    116 S   0.0   0.0   0:00.02 init
  113 root      20   0 1088556  11988   3644 S   0.0   0.6   0:21.43 docker-desktop-
  119 root      20   0    1764      4      0 S   0.0   0.0   0:00.00 init
  120 pom       20   0  763732  14736      0 S   0.0   0.8   0:36.52 docker
  146 root      20   0    1764     40      0 S   0.0   0.0   0:00.00 init
  147 root      20   0    1764     64      0 S   0.0   0.0   0:00.39 init
  148 pom       20   0   10188   5300   3464 S   0.0   0.3   0:00.57 bash
  437 root      20   0    1764     40      0 S   0.0   0.0   0:00.00 init
  438 root      20   0    1764     64      0 S   0.0   0.0   0:00.02 init
  439 pom       20   0   10056   5148   3408 S   0.0   0.3   0:00.11 bash
  485 pom       20   0   10860   3720   3216 R   0.0   0.2   0:00.10 top
```

Run these commands in another console.

```
pom@X280:/mnt/c/Users/Pom$ date; grep ctxt /proc/485/status
Mon Feb 21 18:49:48 +07 2022
voluntary_ctxt_switches:        90
nonvoluntary_ctxt_switches:     1
pom@X280:/mnt/c/Users/Pom$ date; grep ctxt /proc/485/status
Mon Feb 21 18:49:54 +07 2022
voluntary_ctxt_switches:        97
nonvoluntary_ctxt_switches:     1
pom@X280:/mnt/c/Users/Pom$ date; grep ctxt /proc/485/status
Mon Feb 21 18:50:02 +07 2022
voluntary_ctxt_switches:        104
nonvoluntary_ctxt_switches:     1
```

This output shows the number of context switches over the lifetime of the process. Notice the distinction between *voluntary* and *nonvoluntary* context switches. A voluntary context switch occurs when a process has given up control of the CPU because it requires a resource that is currently unavailable (such as blocking for I/O.) A nonvoluntary context switch occurs when the CPU has been taken away from a process, such as when its time slice has expired or it has been preempted by a higher-priority process.

```
pom@X280:/mnt/c/Users/Pom$ top -d 0.01
```

Update every 10 msec

```
pom@X280:/mnt/c/Users/Pom$ date; grep ctxt /proc/656/status
Mon Feb 21 19:40:54 +07 2022
voluntary_ctxt_switches:        3784
nonvoluntary_ctxt_switches:     742
pom@X280:/mnt/c/Users/Pom$ date; grep ctxt /proc/656/status
Mon Feb 21 19:41:03 +07 2022
voluntary_ctxt_switches:        4681
nonvoluntary_ctxt_switches:     849
pom@X280:/mnt/c/Users/Pom$
```

Nonvoluntary context switch about every 100 msec.

```
pom@X280:~/OS$ cat infinite.c
void main()
{
        for (;;)
            ;
}
```

```
pom@X280:~/OS$ ./infinite
```

```
pom@X280:/mnt/c/Users/Pom$ date; grep ctxt /proc/730/status
Mon Feb 21 20:34:18 +07 2022
voluntary_ctxt_switches:        0
nonvoluntary_ctxt_switches:     1585
pom@X280:/mnt/c/Users/Pom$ date; grep ctxt /proc/730/status
Mon Feb 21 20:34:28 +07 2022
voluntary_ctxt_switches:        0
nonvoluntary_ctxt_switches:     1926
```

Nonvoluntary context switch about every 30 msec.

Nice (set priority of process)

```
NICE(1)                              User Commands                              NICE(1)

NAME
       nice - run a program with modified scheduling priority

SYNOPSIS
       nice [OPTION] [COMMAND [ARG]...]

DESCRIPTION
       Run  COMMAND with an adjusted niceness, which affects process scheduling.  With no COMMAND, print the current niceness.
       Niceness values range from -20 (most favorable to the process) to 19 (least favorable to the process).

       Mandatory arguments to long options are mandatory for short options too.

       -n, --adjustment=N
              add integer N to the niceness (default 10)
```

```
pom@X280:~/OS$ cat multiply.c
int main(void) {
    int total = 1;
      for (int j =1; j <=50000 ; j++)
        for (int k =1; k <=50000 ; k++)
          total *= j*k;

}
```

```
pom@X280:~/OS$ cat multiply.sh
time ./multiply &
time nice -n 1 ./multiply &
time nice -n 2 ./multiply &
time nice -n 3 ./multiply &
```

```
pom@X280:~/OS$ ./multiply.sh
pom@X280:~/OS$
real	0m14.801s
user	0m8.918s
sys	0m0.041s

real	0m19.142s
user	0m9.243s
sys	0m0.067s

real	0m21.373s
user	0m9.237s
sys	0m0.059s

real	0m22.535s
user	0m8.989s
sys	0m0.086s
```

top –p `pgrep –d ',' "multiply"`

```
top - 23:29:09 up 4 days, 17:11,  0 users,  load average: 0.92, 0.53, 0.47
Tasks:  21 total,   5 running,  16 sleeping,   0 stopped,   0 zombie
%Cpu(s):  97.8/2.2   100[|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
MiB Mem :   1916.6 total,     73.2 free,   1250.4 used,    593.0 buff/cache
MiB Swap:   1024.0 total,    290.5 free,    733.5 used.    407.4 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
  860 pom       20   0    2360    580    516 R  62.5   0.0   0:05.47 multiply
  864 pom       21   1    2360    576    512 R  49.2   0.0   0:04.28 multiply
  863 pom       22   2    2360    584    520 R  38.9   0.0   0:03.52 multiply
  865 pom       23   3    2360    516    452 R  30.6   0.0   0:02.76 multiply
```

Nicer processes finish later.