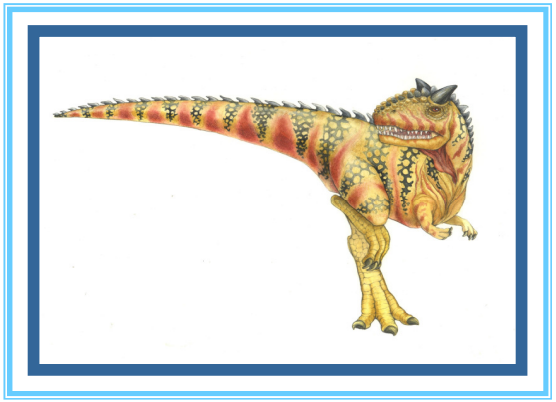
Chapter 5
Process Scheduling



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.

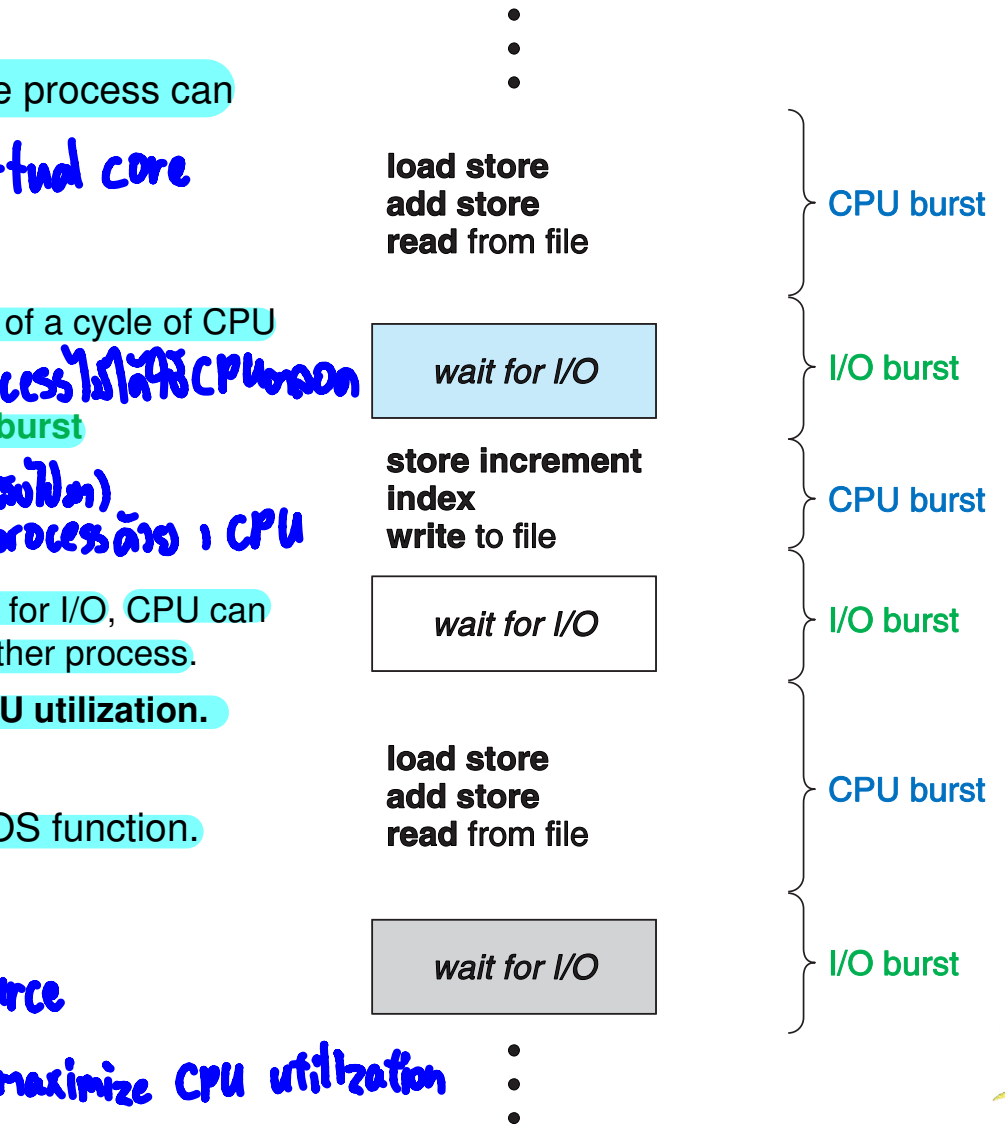
CPU – I/O Burst Cycle

- Process execution consists of a cycle of CPU execution and I/O wait
- CPU burst followed by I/O burst

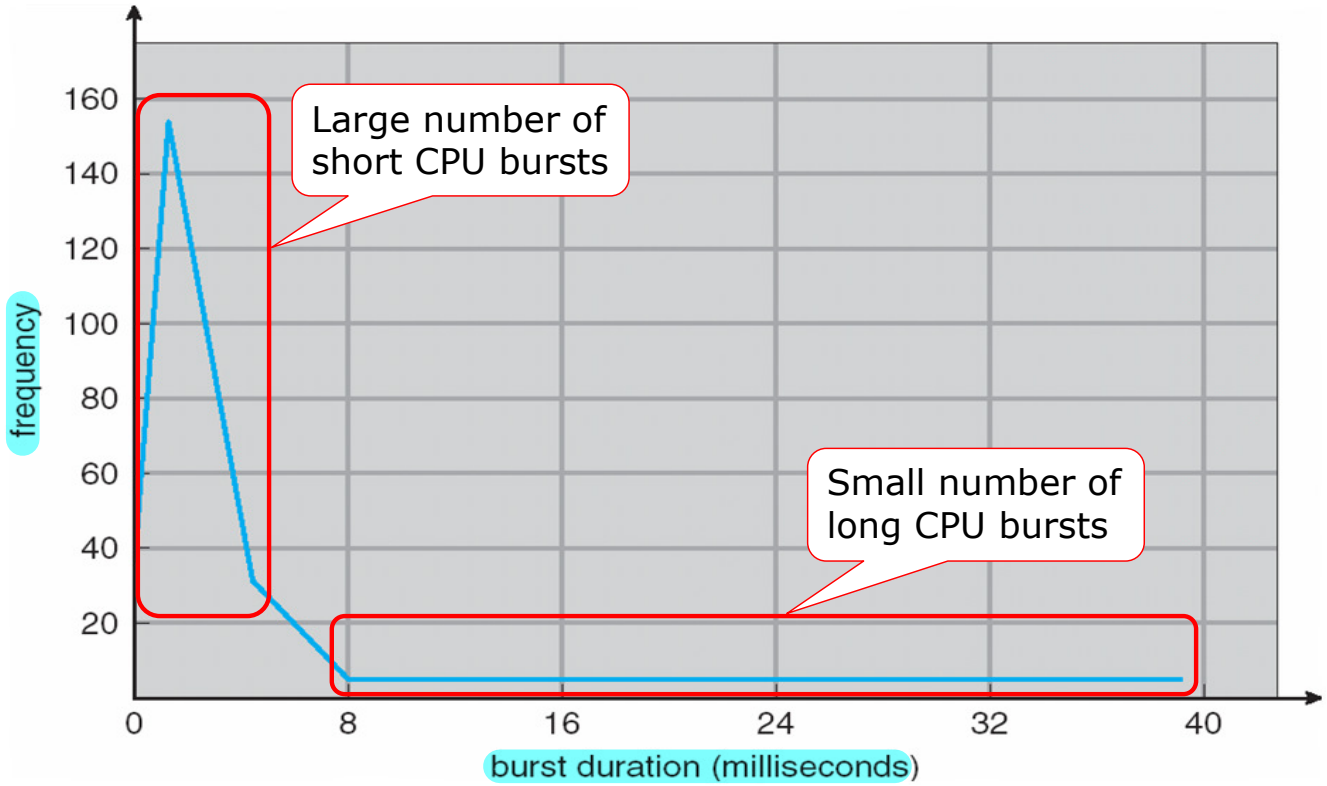
Multiprogramming

- 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.



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-bound vs. I/O-bound programs

CPU-bound program

infinite.c

```
void main()
{
    int i;
    for (;;)
        i++;
}
```

I/O-bound program

printloop.c

```
#include <stdio.h>

void main()
{
    int i;
    for (;;)
        printf("%d\n", i++);
}
```

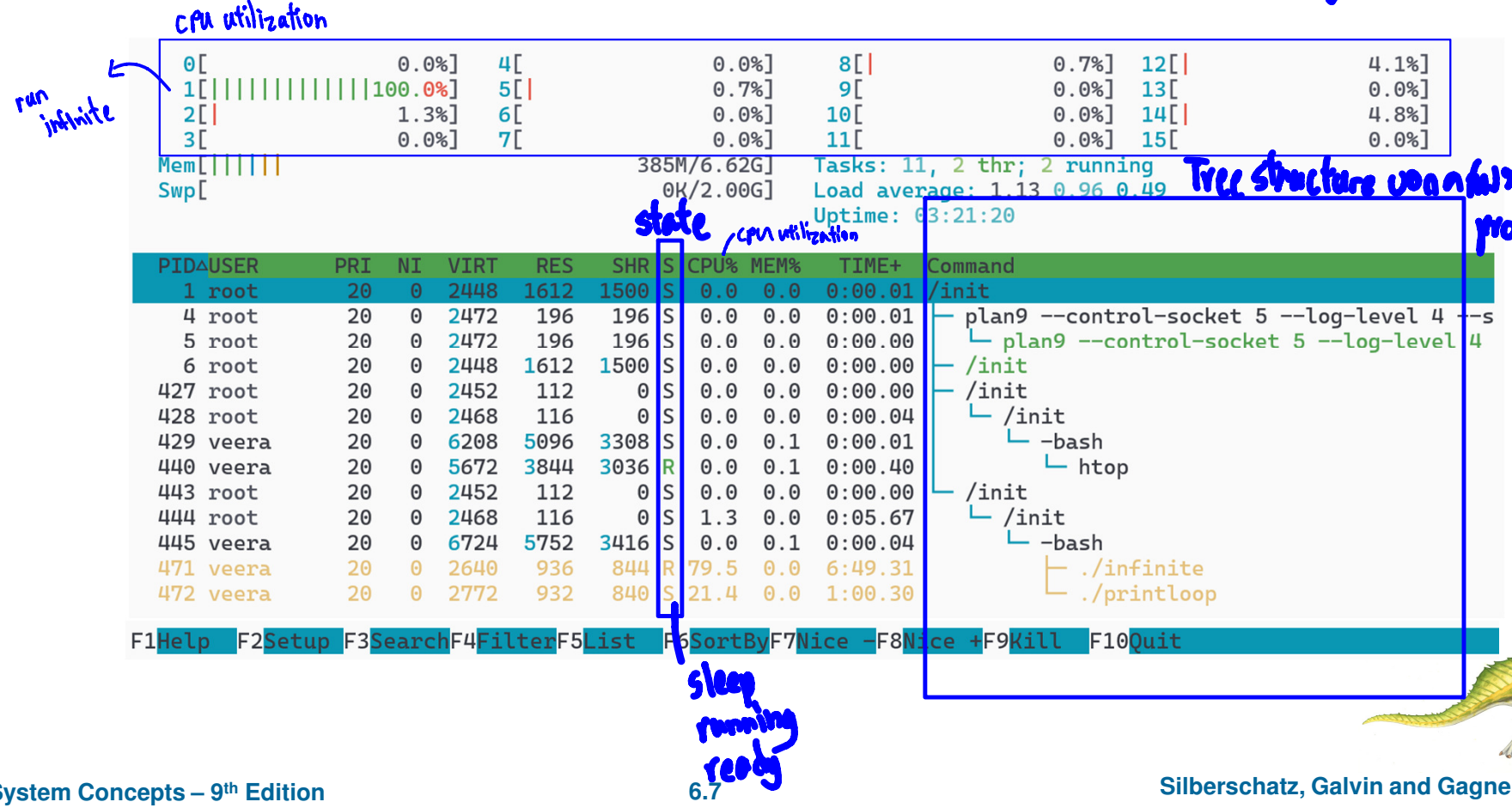
readfileloop.c

```
#include <stdio.h>
#include <unistd.h>

int main() {
    FILE *fp;
    char buffer[1000000];

    while(1) {
        fp = fopen("/usr/bin/bash", "rb");
        while (fread(buffer, 1, sizeof(buffer),
fp))
            ; // Just read, no processing
        fclose(fp);
        sync(); // Flush filesystem cache
    }
    return 0;
}
```

Linux 'htop' command

Interactive system monitor process viewer and process manager.

An alternative to 'top' command.



Linux ‘htop’ command

set “**processor affinity**” (binding a process to a set of CPUs)

- Use up-down arrows to select a process and type ‘**a**’ → a list of CPUs will appear
- Use up-down arrows to select CPU and ‘space’ to toggle (x means “Use”)

เลือก CPU

set “**niceness**” to a process (PRIority = 20 + NIce value)

- Use up-down arrows to select a process and type ‘**a**’ → a list of CPUs will appear

PRI ยิ่งน้อยยิ่งได้ใช้

```
  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

F1Help  F2Setup F3SearchF4FilterF5List  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```

Linux ‘htop’ command

set “**niceness**” to a process

- PRI is the priority level. The lower the PRI, the higher the priority of the process will be (more likely to use CPU).
- NI is “nice value”, ranges between -20 to +19. A nicer process lets other processes run before.
- PRI = 20 + NI
- Use F8 to increase Nice value and F7 to decrease Nice value (Use ‘sudo htop’ to decrease nice value)

```
  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

  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

F1Help  F2Setup F3SearchF4FilterF5List  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```

Experiments

- Run htop on one terminal and other programs in another terminal as background process using &.

Experiment 1:

- Run *infinite* program. Observe state (S) and CPU%.
- Run *infinite* many times. Observe how *infinite* processes are assigned to CPUs.
- Run *printloop*. Observe state (S) and CPU%. Also observe how the process is assigned to a CPU.
- Run *readfileloop*. Compare CPU% to *printloop*.

Experiment 2:

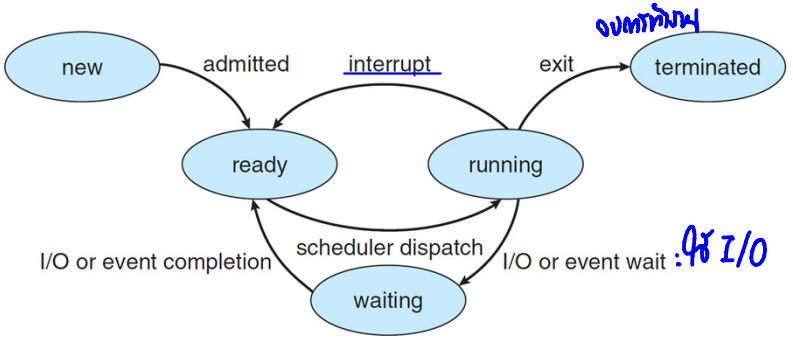
- Set process affinity of *infinite* processes to the same CPU. Observe their CPU%.

Experiment 3:

- Set nice value to *infinite* processes that use the same CPU. Observe their CPU%

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

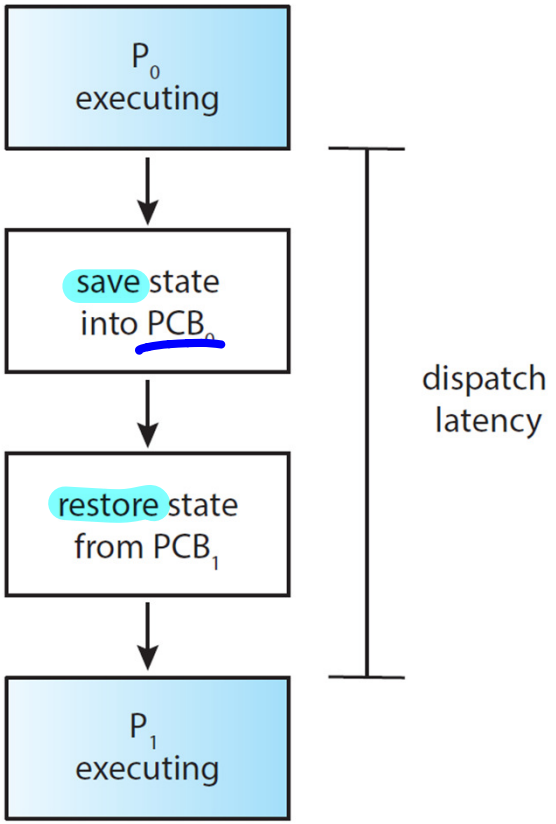
Dispatcher

save state

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	Total amount of time a process spent while 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

main goal

↳ in interactive program

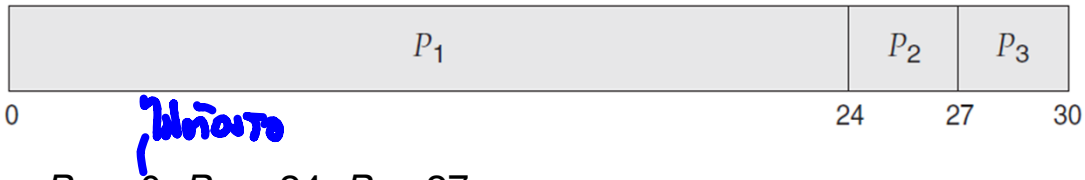
Burst time : Total amount of time a process uses the CPU.
Turnaround time = Burst time + Waiting time

Scheduling Algorithm

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

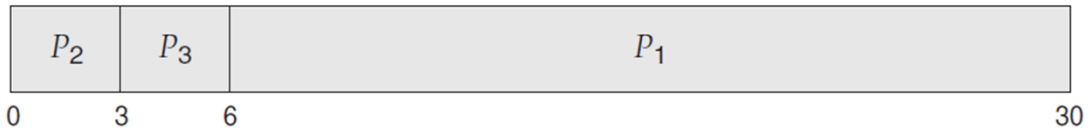
Average waiting time: (0 + 24 + 27)/3 = 17

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.

Finally

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

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

→ อันนี้จะได้ที่ช้ามาก, เพราะโดนแซงเรื่อยๆ, ไม่ได้ที่สักที

SJF could lead to starving processes

- Process with long CPU burst waits for processes with shorter bursts

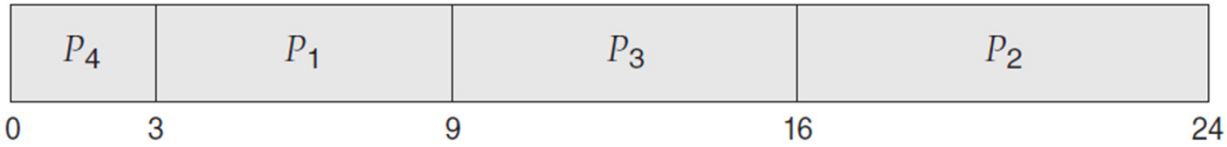
SJF is hard to implement

- 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



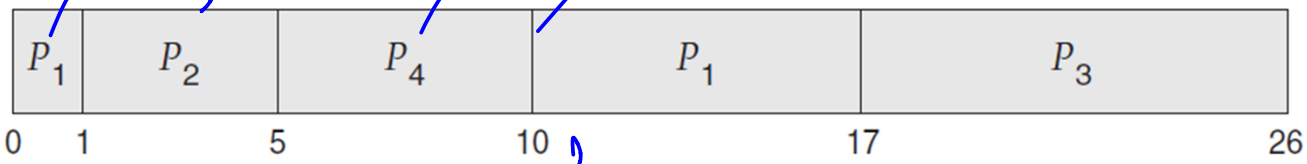
Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

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

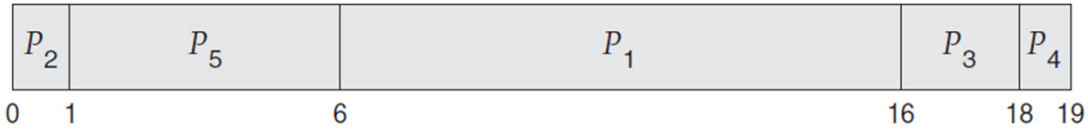
Problem: **Starvation** – low priority processes may never execute

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.

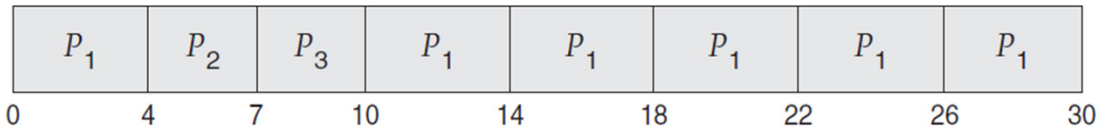
Note: The term "round robin" actually came from "round ribbon".

Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

time quantum = 4

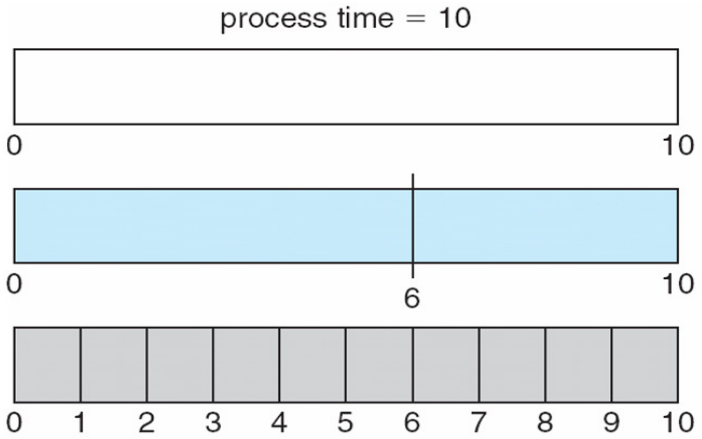
- The Gantt chart is:



- 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

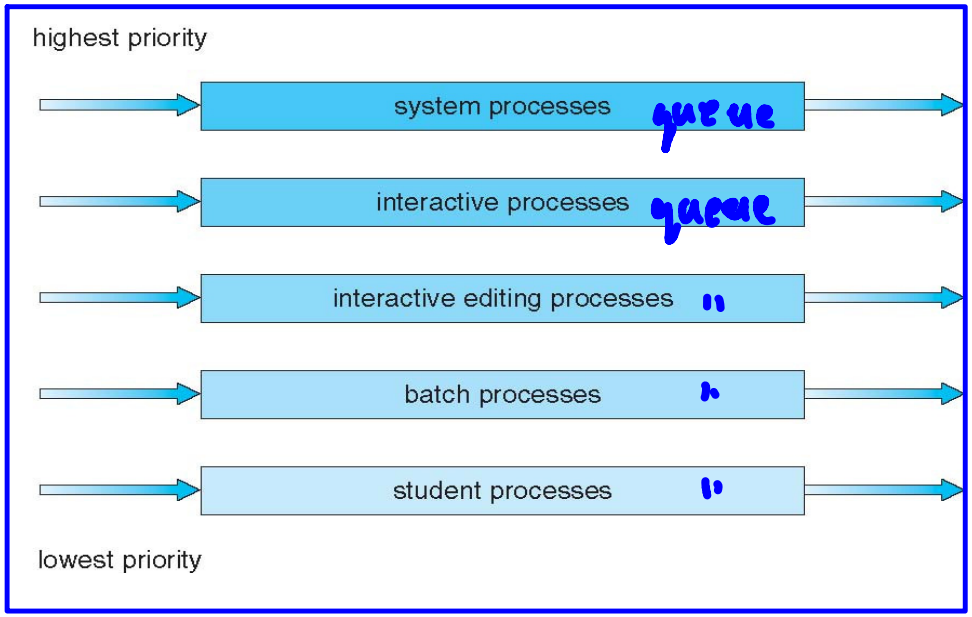
Ready queue is partitioned into separate queues, eg:

- **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 : 4ü responsive
- 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.

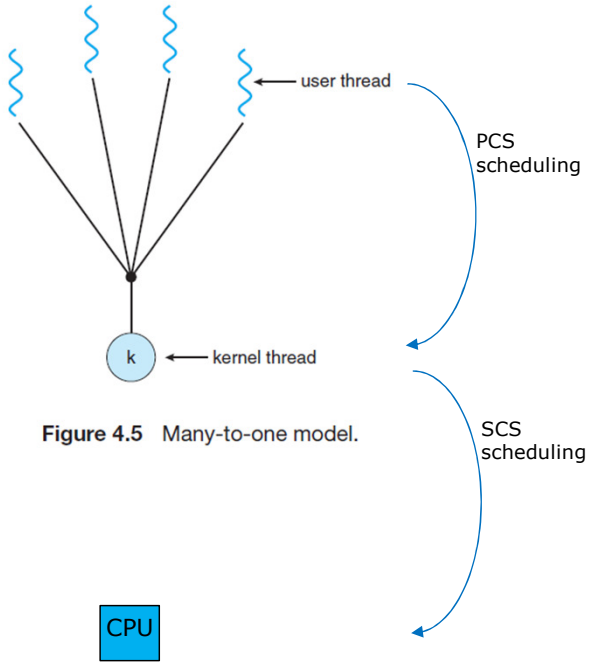


Figure 4.5 Many-to-one model.

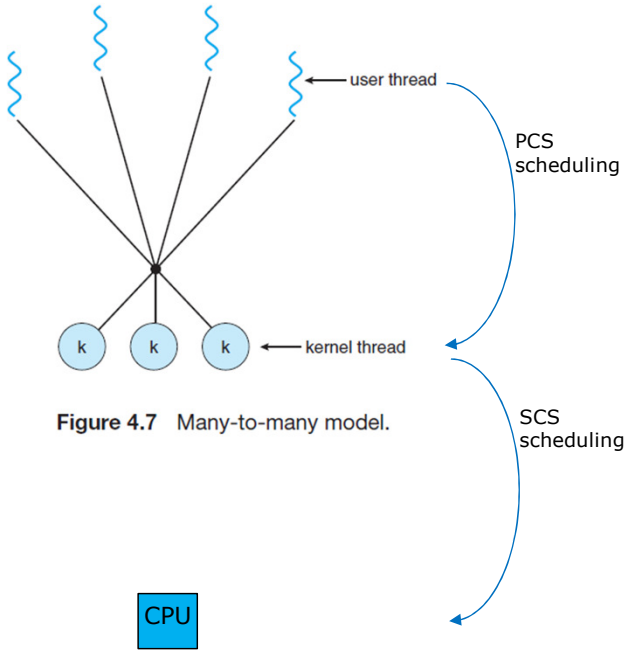


Figure 4.7 Many-to-many model.

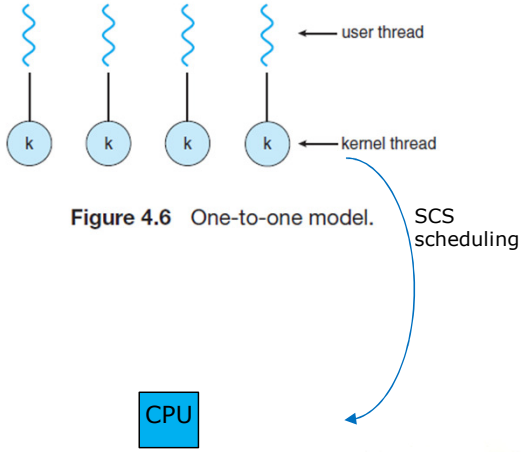


Figure 4.6 One-to-one model.

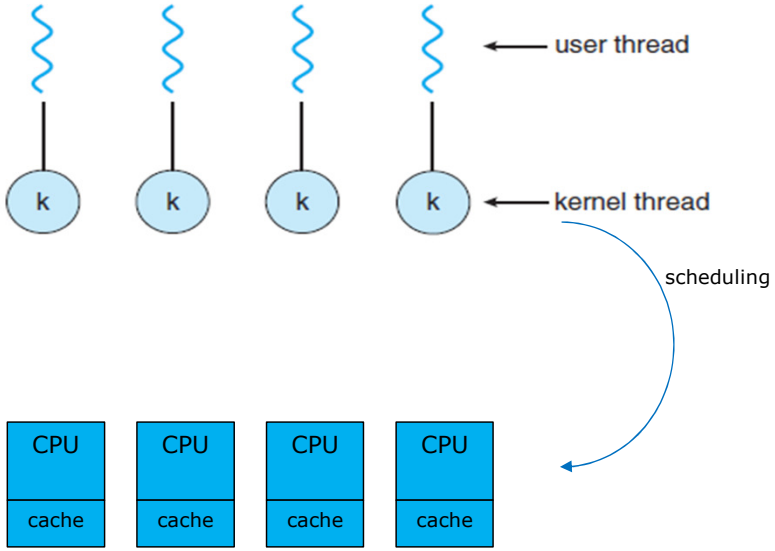
Multiple-Processor Scheduling

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.