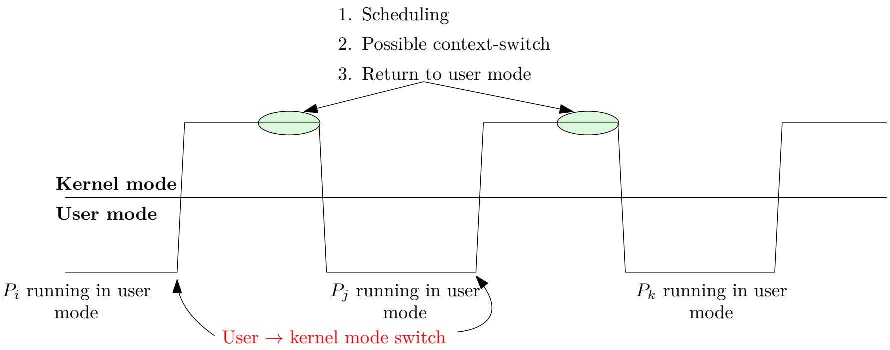
Operating Systems: Scheduling

Indian Statistical Institute

https://www.isical.ac.in/~mandar/courses.html#os

Background: mode switch, process switch



Scheduling chooses P_j, P_k, etc. from all ready-to-run processes.

Preemptive vs. non-preemptive scheduling

- Process switch can occur when a process
  1. needs to wait for some resource / puts itself to sleep (via *sleep*())
  2. exits
     (conclusion of exit system call invokes context switch code)
  3. returns from kernel mode to user mode but is *not the most eligible process to run*
- **Non-preemptive scheduling**: scheduling takes place only in 1 and 2
  - when a process gets the CPU, it keeps it until it sleeps/exits
- **Preemptive scheduling**: case 3 is also permissible

THUMB RULE: Preemptive scheduler runs *whenever* a process is added to the Ready Queue.

Terminology / recap

Reference: Section 3.2, 6.1

Process states

- RUNNING, READY, WAITING, etc.
- typically alternates between *CPU bursts* and *I/O bursts*
- time-sharing / multiprogramming: to maximize CPU utilization
  - multiple processes are kept in memory simultaneously
  - when one process is waiting, another process executes

CPU bound process: spends more time doing computations, generates I/O requests infrequently

I/O bound process: spends more time doing I/O than computing

Job queue: contains all processes in the system

Ready queue: contains all processes that reside in main memory and are ready to run

Device queue: contains all procs. waiting for a particular device

First-come first-served

Reference: Section 6.3.1

Method:

1. Maintain a FIFO queue.
2. When a process enters the ready queue, it is placed at the end of the queue.
3. When the CPU is free, it is allocated to the process at the head of the queue.

Properties:

- Non-preemptive
- Unsuitable for time-sharing systems ($\because$ each user should get a share of the CPU at regular intervals)
- Average waiting time is not minimal
- Convoy effect: many processes may have to wait for one long process to finish
  Example: 1 CPU-bound proc. + many I/O bound procs.

First-come first-served

Example:

Ready processes	Burst time
P_1	24
P_2	3
P_3	3

Processes arrive in the order P_1, P_2, P_3

Gantt chart:

P_1	P_2	P_3

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

Shortest job first

Reference: Section 6.3.2

Method:

1. When the CPU is available, assign it to the process with the shortest next CPU burst.
2. Break ties on a FCFS basis.

Properties:

- Optimal in terms of average waiting time
- Suitable for job scheduling in a batch system
  (use time limit specified by user at time of submission)
- *Length of the next CPU request is generally not known*

Pre-emptive SJF: (shortest remaining time first)

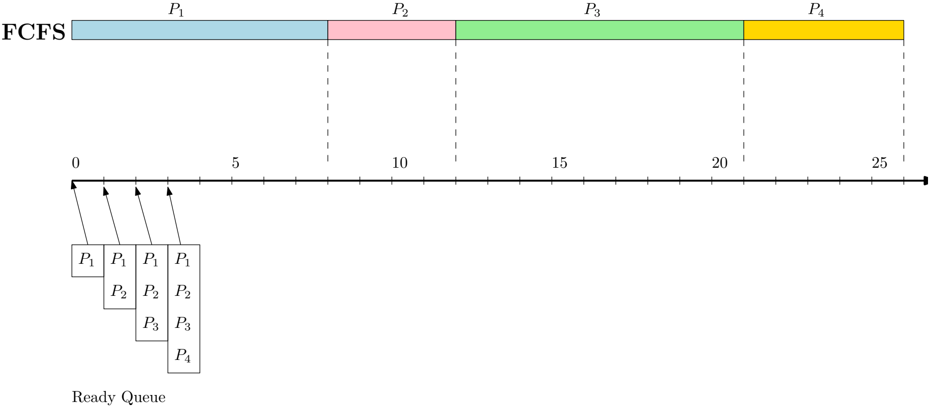
1. When a new process arrives at the ready queue, compare its CPU burst with remaining time for current process.
2. If new process has shorter burst, preempt current process.

Shortest job first

Example:

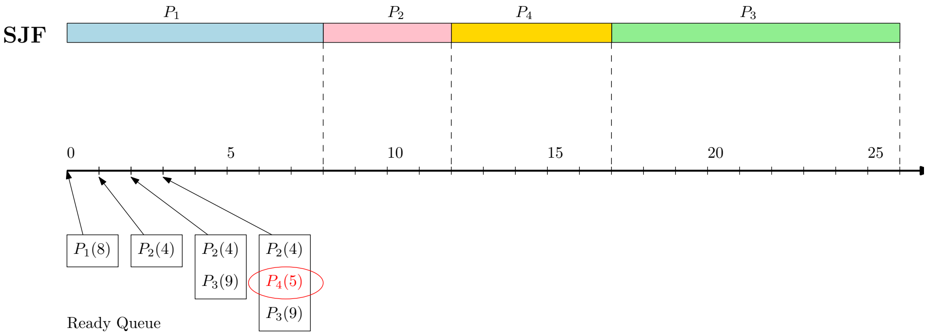
Ready processes	Arrival time	Burst time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

FCFS



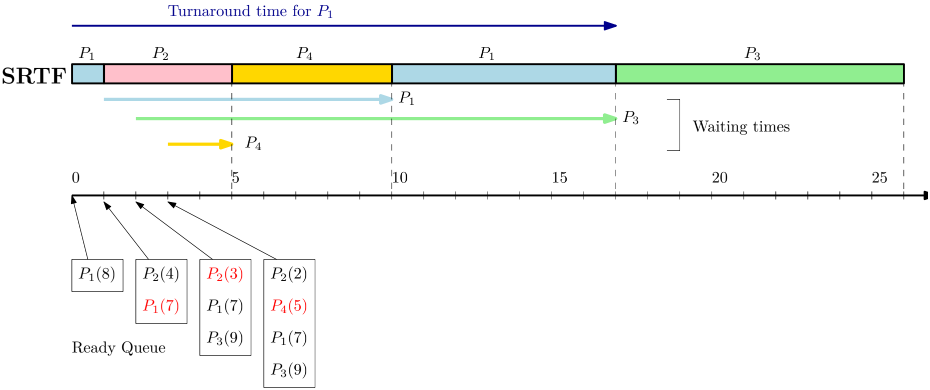
Average waiting time: $(0 + 7 + 10 + 18)/4 = 8.75$ms

Shortest job first



Average waiting time: $(0 + 7 + 15 + 9)/4 = 7.75$ms

Shortest remaining time first



Average waiting time: $(9 + 0 + 15 + 2)/4 = 6.5$ms

Turnaround times: $17, 4, 24, 7$ resp.

Priority scheduling

Reference: Section 6.3.3

Method:

1. Compute a priority for each process.
   - Internal priorities: computed using time limits, memory requirements, ratio of avg. I/O burst to avg. CPU burst, etc.
   - External priorities: computed on the basis of external political / administrative factors
2. Allocate CPU to process with highest priority.
3. Break ties on a FCFS basis.

Properties:

- Can be preemptive or non-preemptive (cf. SJF)

Priority scheduling

Example: (low numbers $\Rightarrow$ high priority)

Processes	**Burst time**	**Priority**
P_1	10	3
P_2	1	1
P_3	2	3
P_4	1	4
P_5	5	2

Priority scheduling

Example: (low numbers $\Rightarrow$ high priority)

Processes	Burst time	Priority	Scheduled	Waiting time
P_1	10	3	6–16	6
P_2	1	1	0–1	0
P_3	2	3	16–18	16
P_4	1	4	18–19	18
P_5	5	2	1–6	1

Average waiting time: $41/5 = 8.2$ ms

Priority scheduling

Properties:

- **Starvation / indefinite blocking:** if high priority processes keep arriving, low priority process may have to wait indefinitely for CPU.
- Priority scheduling with *aging*: priority may be increased in proportion to waiting time to prevent starvation

Round robin

Reference: Section 6.3.4

Time quantum (or time slice): maximum interval of time between two invocations of the scheduler

- a process can be allocated the CPU for one quantum at one time
- usually between 10–100ms

Method:

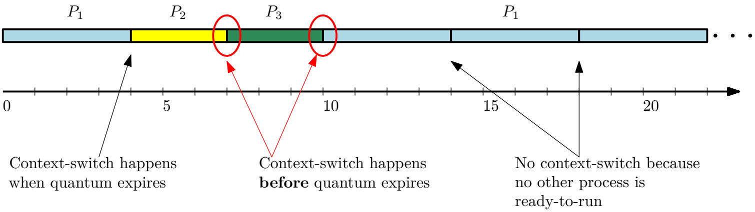
1. Maintain a FIFO queue of ready processes.
2. Allocate CPU to first process from queue; set timer for 1 time quantum.
3. If running process releases CPU, or timer expires:
   preempt current process and switch context to the next process in the ready queue;
   add previously running process to tail of ready queue.

Round robin

Example:

Ready processes	Burst time
P_1	24
P_2	3
P_3	3

Time quantum = 4ms



Average waiting time: 5.66ms

Round robin

Properties:

- Suitable for time-sharing systems ($\because$ every process gets the CPU for q time units after waiting for $(n-1)q$ time units)
- Duration of time quantum:
  - large time quantum $\Rightarrow$ RR $\rightarrow$ FCFS
  - small time quantum $\Rightarrow$ context-switching overhead $\uparrow$

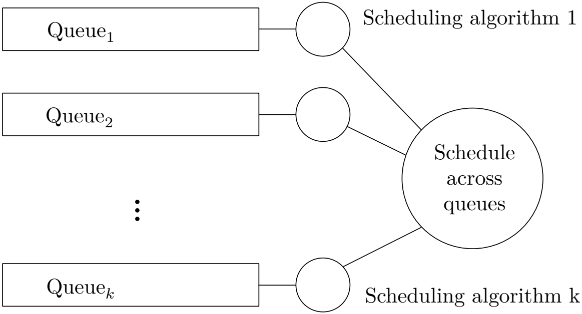
Multilevel queue

Reference: Section 6.3.5

Method:

1. Partition the queue into several separate queues; assign a fixed priority value to each.
2. Assign each process to some fixed queue, based on its properties. Example: system procs. / interactive procs. / interactive editing procs. / batch procs. / student procs.
3. Select a queue based on:
   - fixed priority (*starvation possible*), OR
   - priority-based proportional time slicing, e.g.,
     - 50% of time servicing Q_0,
     - 30% of time servicing Q_1,
     - 20% of time servicing Q_2.
4. Select a job from the queue using a suitable scheduling algorithm (e.g. FCFS, RR).

Multilevel queue



Properties:

- Preemptive

Multilevel feedback queue

Reference: Section 6.3.6

- Processes may be moved between scheduling queues
- Parameters:
  - # of queues
  - scheduling algorithm / time slice for each queue
  - initial queue selection policy
  - promotion/demotion policies

Example:

- 3 queues, Q_0, Q_1, Q_2
- scheduling policies:
  Q_0 = RR (quantum = 8ms) Q_1 = RR (quantum = 16ms) Q_2 = FCFS
- on entry to ready queue, processes assigned to Q_0
- on exit from Q_0, process is placed at tail of Q_1
  on exit from Q_1, process is placed at tail of Q_2
  OPTIONAL: if process waits too long in Q_2, promote it to Q_1

Scheduling criteria I

Reference: Section 6.2

- CPU utilization: proportion of time that CPU does “useful work”
  - CPU time spent executing in user mode
  - CPU time spent by kernel servicing a user request
  - **does not include** CPU time spent by kernel doing system work e.g., scheduling, context switching
- Throughput: number of processes that are completed per unit time
  - long processes $\Rightarrow$ throughput $\downarrow$
    short processes $\Rightarrow$ throughput $\uparrow$
- Turnaround time: interval from the time of submission of a process to the time of completion
- Waiting time: total amount of time spent by a process in the ready queue

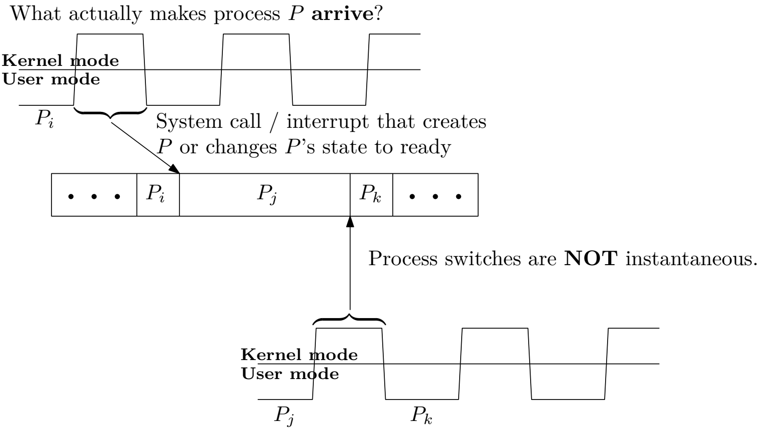
Scheduling criteria II

- Response time: time from the submission of a request until the first response is produced
  (amount of time taken to *start* responding, not including the time taken to complete the output)

NOTE: maximum (minimum)/average/variance may be suitable for evaluation

Commonly ignored / simplified details

- When / how do processes “arrive”?
- Process switching time in Gantt charts



Real-time scheduling

Real-time scheduling

Reference: Section 6.6

Hard real-time systems:

- Critical tasks must be completed within a guaranteed amount of time
- Resource reservation:
  - processes are submitted with deadlines
  - scheduler may admit the process and guarantee completion, or reject
- Duration of operating system functions must be predictable and bounded
- Consists of special-purpose software running on dedicated hardware

Soft real-time systems:

- Critical processes receive priority over “ordinary” processes
- May be implemented as a general-purpose system

Real-time scheduling

Preempttible vs. non-preemptible kernels:

- Non-preemptible kernels
  - context switch can happen only at restricted points
    - completion of system call/interrupt
    - `sleep()`
    - specially inserted *preemption points*
  - delays may be unpredictable
  - easier to implement
- Preemptible kernels
  - suitable for soft real-time systems
  - harder to implement

Priority inversion

- High priority process may have to wait for resource held by a low priority process
- **Priority inheritance:** processes that are accessing resources required by high priority process inherit the high priority until they release the resource