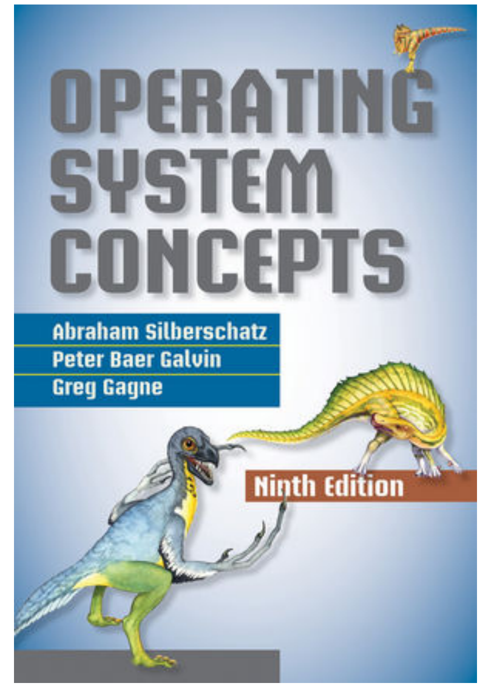
5 - CPU Scheduling

EECE 315 (101)

ECE – UBC

2013 W2



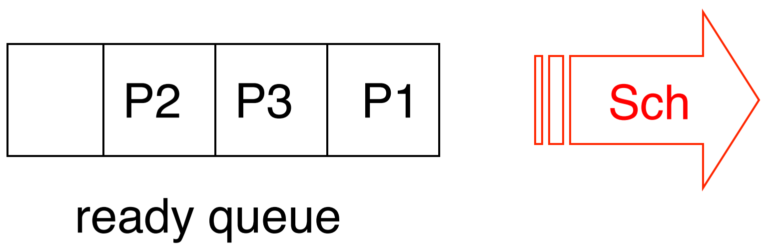
Acknowledgement: This set of slides is partly based on the PPTs provided by the Wiley's companion website (including textbook images, when not explicitly mentioned/referenced).

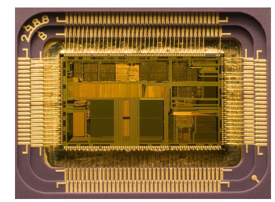
Lecture Outline

- **Basic Concepts: Scheduling Criteria**
- Scheduling Algorithms
- Thread Scheduling

CPU Scheduler

- The CPU is one of the primary computer resources and should be scheduled before use
  - To select one process from among the processes in memory that are ready to execute, and allocate the CPU to it





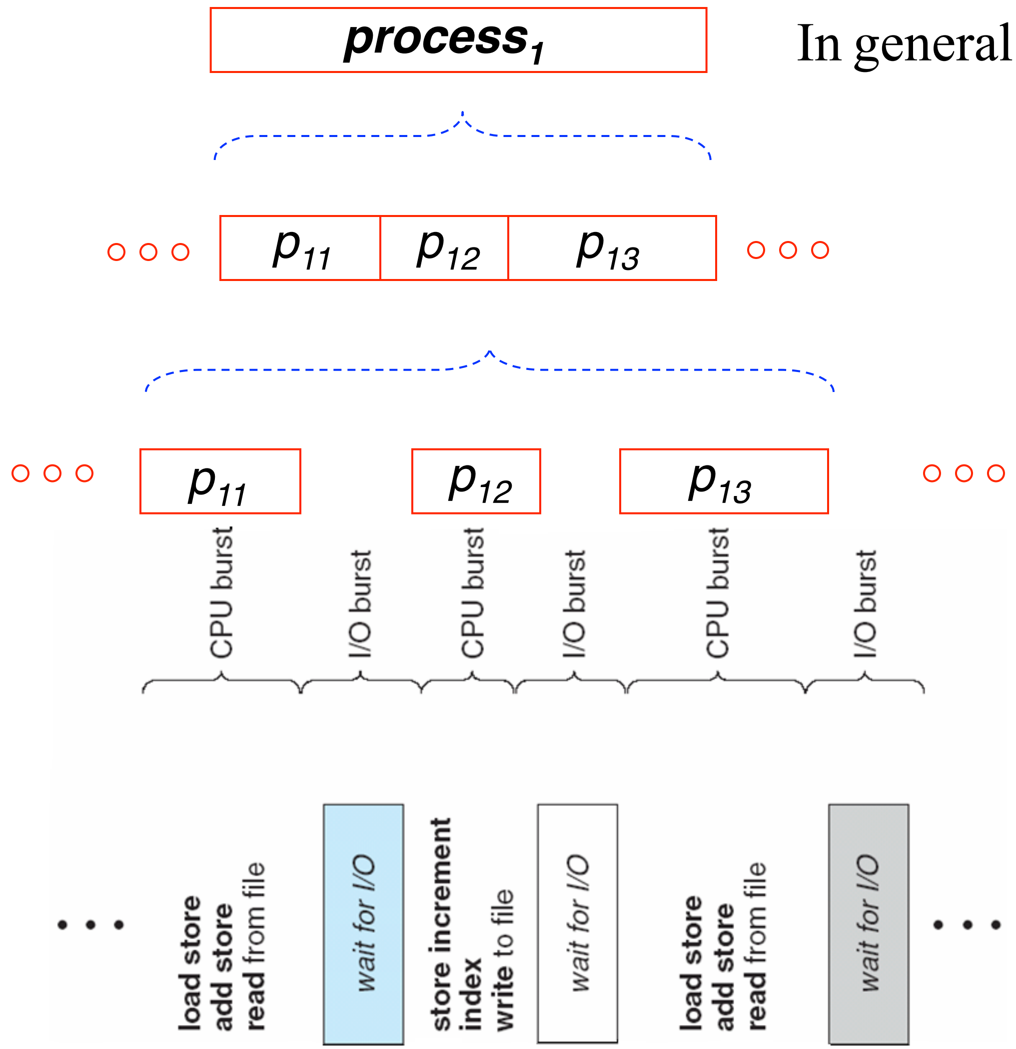
CPU

  - The success of CPU scheduling depends on the observed property of processes
  - In our daily life, we are mostly used to first-in first-out queues (FIFO).
    - The above ready queue though is not necessarily FIFO.

Process Partition into Smaller CPU Bursts

Note that each process (here ***process$_1$***) is generally divided into a number of CPU bursts.

In general: $process_i = \sum_k P_{ik}$

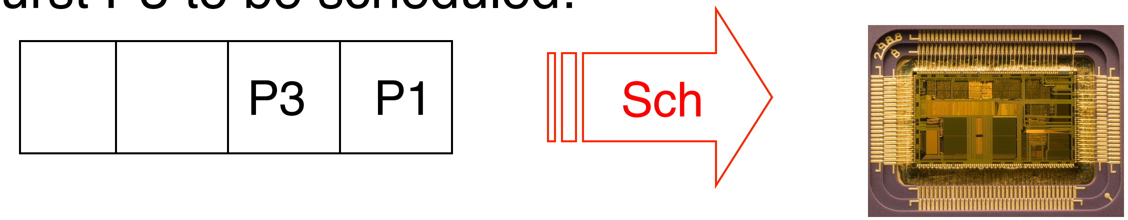


Scheduling Criteria

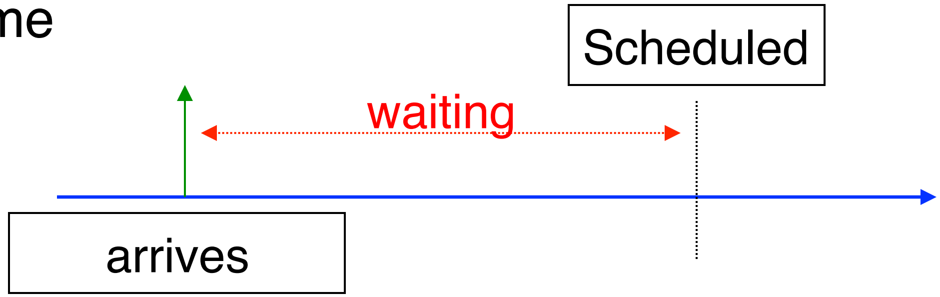
- Many criteria have been used for comparing scheduling algorithms.
  - **Waiting time** – amount of time a process has been waiting in the ready queue
  - **Turnaround time** – amount of time to execute a particular process (includes waiting time in memory and ready queue, executing in CPU and doing IO)
  - **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output
  - **CPU utilization** – keep the CPU as busy as possible
  - **Throughput** – # of processes that complete their execution per time unit

Scheduling Criteria - Example

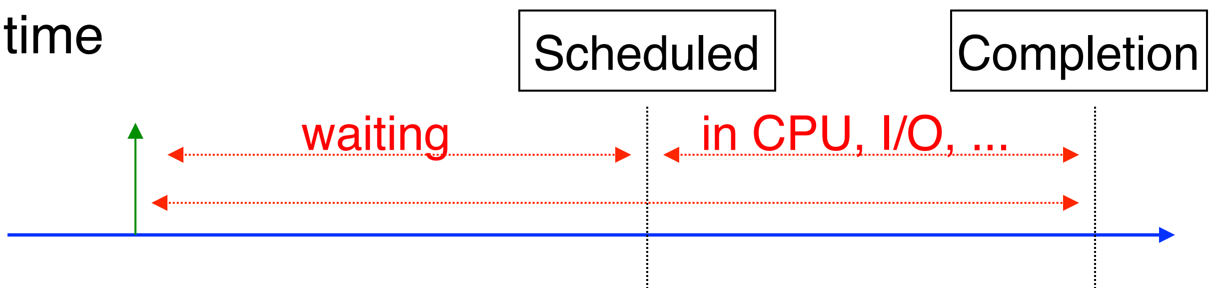
Assume a CPU burst P3 to be scheduled:



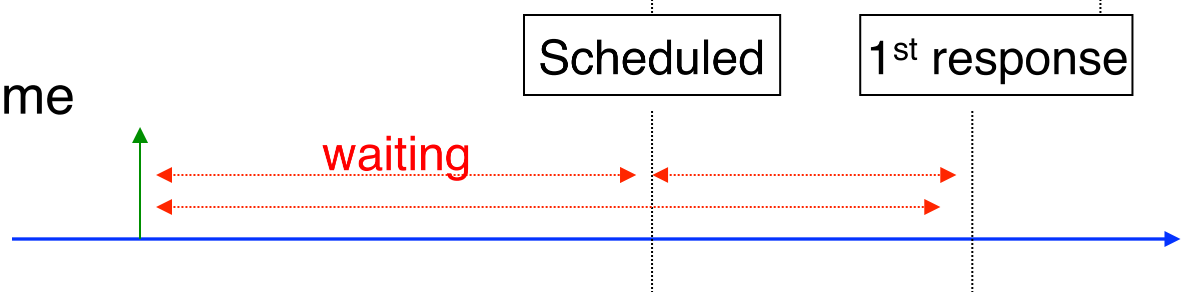
- Waiting time



- Turnaround time



- Response time



Scheduling Algorithm Optimization Criteria

- We usually aim to minimize or maximize a criterion for optimization.
  - choosing the best element from a set of available alternatives
  - aim to minimize
    - turnaround time
    - waiting time
    - response time
  - aim to maximize
    - CPU utilization
    - throughput

- In most cases, we optimize an *average* measure.
  - However, under many other circumstances, the optimization may be on other statistical measures, such as *minimum* or *maximum* values, or *variance*.

Lecture Outline

- Basic Concepts: Scheduling Criteria
- **Scheduling Algorithms**
- Thread Scheduling

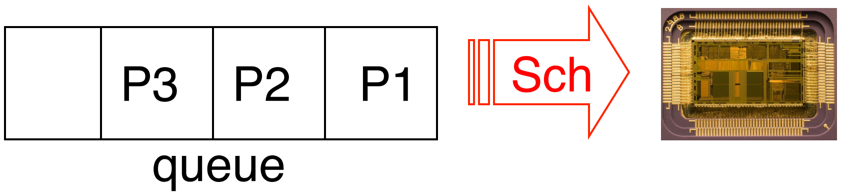
Scheduling Algorithms

- There are many different scheduling algorithms.

- In the following slides, we are going to study:
  - First-Come, First-Served (FCFS)
  - Shortest-Job-First (SJF)
  - Priority Scheduling
  - Round Robin (RR)

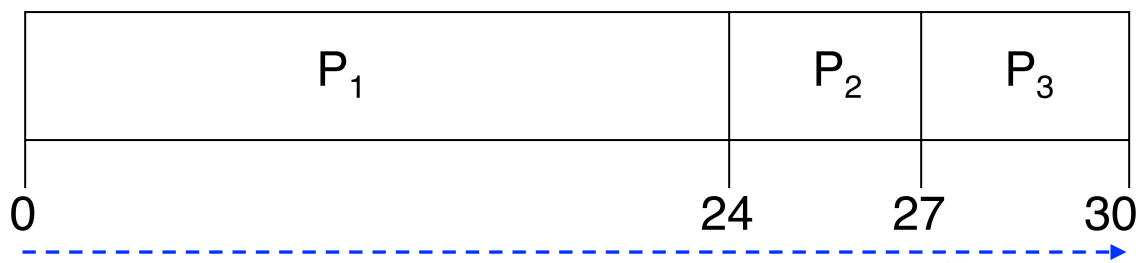
First-Come, First-Served (FCFS) Scheduling

- In this scheme, the process that requests the CPU first is allocated the CPU first.
- **Example**: Suppose that CPU bursts of processes arrive in the order: P_1, P_2, P_3

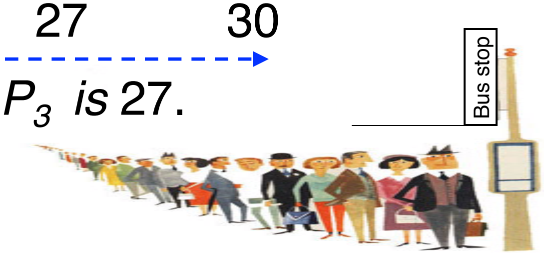


Process	Burst Time
P_1	24
P_2	3
P_3	3

- A Gantt chart is a <u>bar chart that illustrates a particular schedule</u>, including start and finish times.
  - The Gantt Chart for the above example schedule is:



- The waiting time for P_1 *is* 0, for P_2 is 24, *and for* P_3 *is* 27.
- Thus the average waiting time is equal to
  - (0 + 24 + 27)/3 = 17



FCFS Scheduling (Cont)

- Now let's consider another scenario; suppose that this time the processes arrive in the following order:

Process	Burst Time
P_1	24
P_2	3
P_3	3

$$P_2 \text{ , } P_3 \text{ , } P_1$$

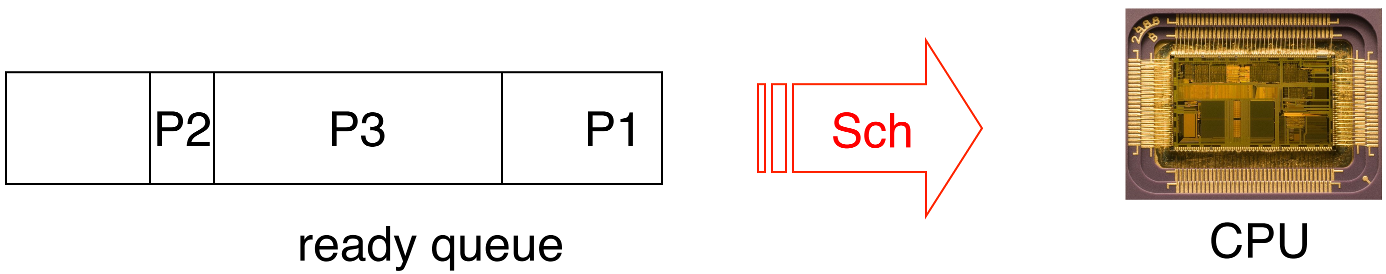
- The Gantt chart for the schedule is changed to:

- Thus the waiting time for P_1 *is* 6, for P_2 is 0*, and for* P_3 *is* 3.
  - The average waiting time is now: (6 + 0 + 3)/3 = 3
  - which is much better than the previous case

- The above two scenarios show that the average time under FCFS policy is generally not minimal and may vary substantially.
  - *Convoy effect*: all other short processes should wait for a long process to get off the CPU

Shortest-Job-First (SJF) Scheduling

- A different approach to CPU scheduling is the shortest-job-first scheduling algorithm. We start with non-preemptive SJF.
  - This algorithm associates with each process the length of its next CPU burst.
  - When the CPU is available, it is assigned to the process with the smallest next CPU burst



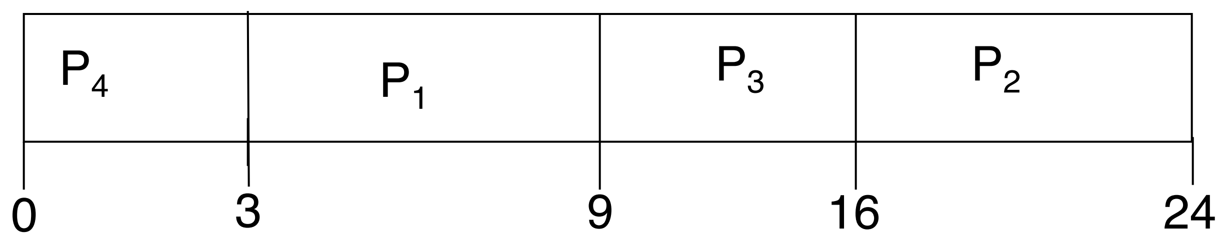
- Probably a more appropriate term for this scheduling method would be the *shortest-next-CPU-burst* algorithm.

SJF Scheduling (cont)

- Let's look at an example:

Process	**Burst Time**
P_1	6
P_2	8
P_3	7
P_4	3

- The Gantt chart for this SJF scheduling is:



- The average waiting time is now: (3 + 16 + 9 + 0) / 4 = 7

- SJF is optimal, in that it gives the minimum average waiting time for a given set of processes
  - The real <u>difficulty is knowing the length</u> of the next CPU request

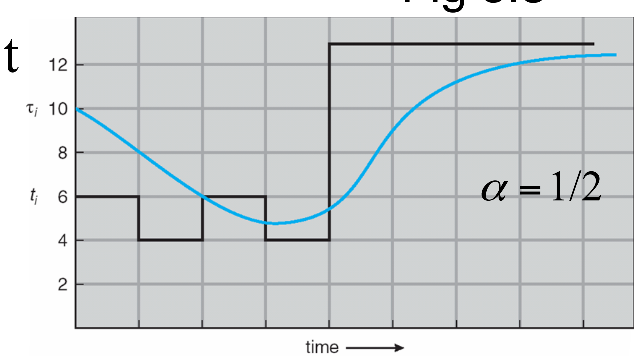
Determining Length of Next CPU Burst

- A process consists of a number of CPU bursts. We may *not know* the length of the next CPU burst (which is to be scheduled), but we may be able to *predict* its value.
  - With short-term CPU scheduling, we can try to approximate SJF scheduling
- In order to compute an approximation of the length of the next CPU burst, we use the length of the previous CPU bursts (which is known), using exponential averaging

1. t_n = actual length of n^{th} CPU burst
2. τ_{n+1} = predicted value for the next CPU burst
3. a parameter α, $0 \le \alpha \le 1$
4. Define:

$$\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n .$$

Fig 5.3



CPU burst (t_i)		6	4	6	4	13	13	13	…
"guess" (τ_i)	10	8	6	6	5	9	11	12	…

Examples of Exponential Averaging

- The parameter, α, controls the relative weight of recent and past history in our prediction.
- Let's consider the following two extremes: $(\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n)$
  - if $\alpha = 0$
    - $\tau_{n+1} = \tau_n$
    - Recent history does not count
  - if $\alpha = 1$
    - $\tau_{n+1} = t_n$
    - Only the actual last CPU burst counts
- If we expand the formula, we get:

$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\alpha\, t_{n-1} + \dots + (1 - \alpha)^j \alpha\, t_{n-j} + \dots + (1 - \alpha)^{n+1} \tau_0$$

  - Since both α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor

Preemptive SJF (SRTF)

- An SJF algorithm can be preemptive, that is, it will preempt a currently executing process. The preemptive SJF is called shortest-remaining-time-first.
- Consider the following example:

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

Note: So far in all examples the arrival time values were not considered. In this example, we are considering different arrival times.

- The Gantt chart for this SJF scheduling is:

p_1	p_2	p_4	p_1	p_3

0 1 5 10 17 26

- The average waiting time is now: [(10-1)+(1-1)+(17-2)+ (5-3)]/4 = 6.5

Priority Scheduling

- In the priority scheduling,
  - a priority number (integer) is associated with each process
    - usually: smallest integer ≡ highest priority
  - and the CPU is allocated to the process with the highest priority
- The SFJ is a special case of the general priority scheduling algorithm.
  - In SJF, the priority is the predicted next CPU burst time
- A major problem with this scheduling algorithm is the possibility of *indefinite blocking* or *Starvation*
  - This algorithm may leave some low priority processes waiting indefinitely
  - A solution to the above problem is the aging technique**:**
    - Aging is the technique of gradually increasing the priority of processes that are waiting in the system as time progresses

Priority Scheduling Example

- Consider the following example:

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

- The Gantt chart for this priority scheduling is:

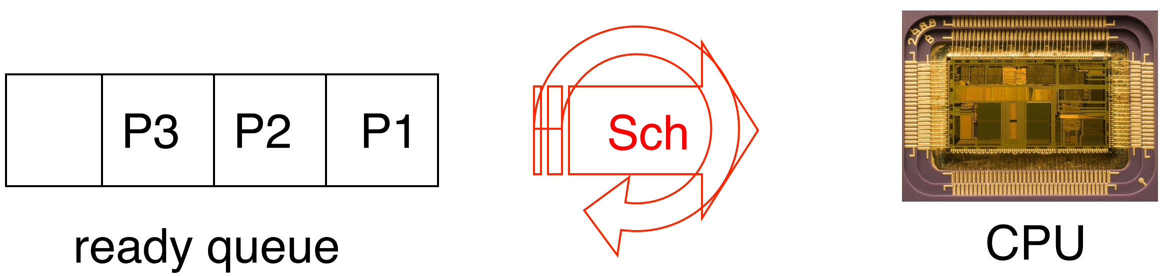
p_2	p_5	p_1	p_3	p_4

0 1 6 16 18 19

- The average waiting time is now: (6+0+16+18+1)/5 = 8.2

Round Robin (RR) Scheduling

- The Round Robin scheduling algorithm is designed especially for time-sharing systems.
- Each process gets a small unit of CPU time (called *time quantum* or *time slice*), usually with a length of 10 to 100 milliseconds.
  - After this time has elapsed, the process <u>is preempted</u> and added to the end of the ready queue.



- In RR, no process is allocated the CPU for more than 1 time quantum in a row (unless it is the only runnable 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.

Example of RR

- Let's take a look at an example:

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Assuming a time quantum of 4, the Gantt chart is:

P_1	P_2	P_3	P_1	P_1	P_1	P_1	P_1

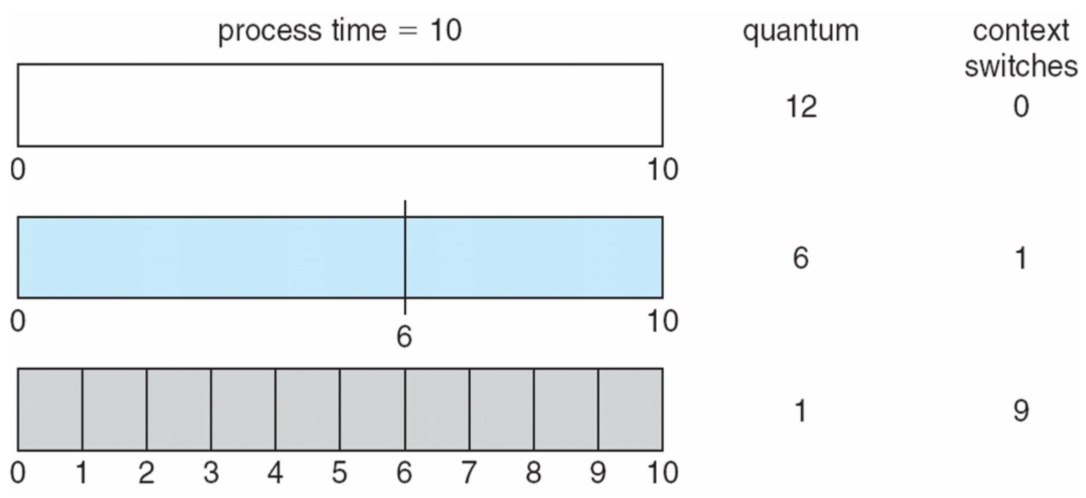
0 4 7 10 14 18 22 26 30

- The average waiting time is now:[(10-4)+4+7]/3 = 5.66

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

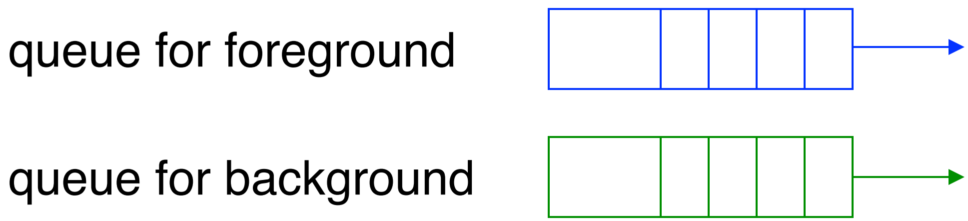
Time Quantum and Context Switch Time

- The performance of the RR algorithm depends on
  - the size of the time quantum (q)
    - *q* very large ⇒ FIFO
    - *q* small ⇒ processor sharing
      - This creates the appearance that each process is running on its own CPU at $1/n$ the speed of the real processor
    - *q* must be large with respect to context switch, otherwise overhead is too high
  - the context switching effect

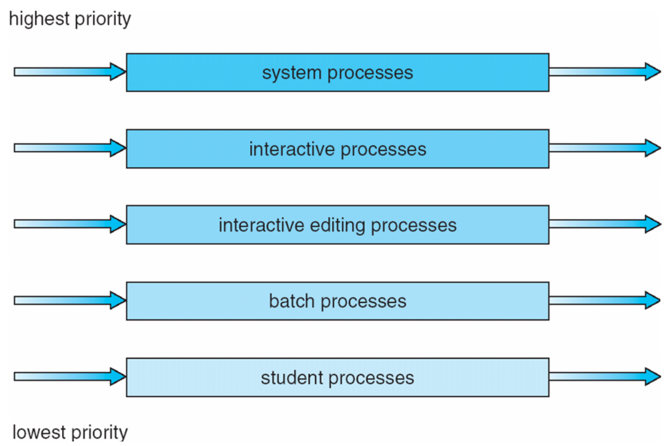


Multilevel Queue

- In many situations, we prefer to classify the processes into different groups.
  - For example, a common division is made between *foreground* and *background* processes.
  - These two types have different scheduling requirements or priorities.

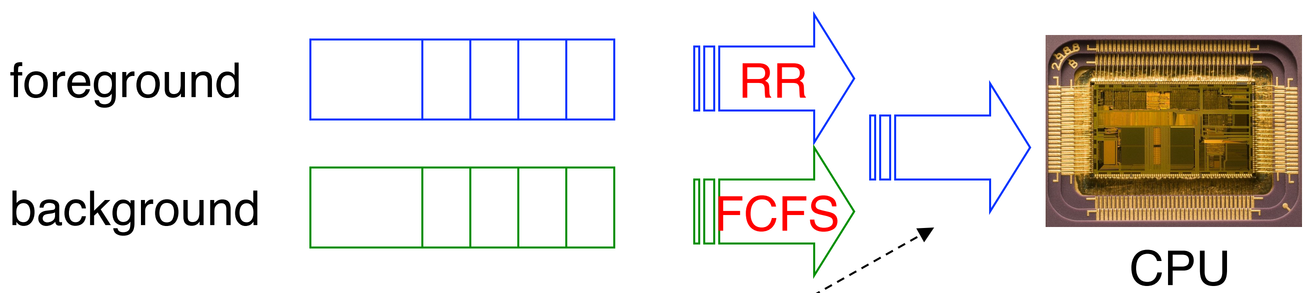


- A multi-level queue scheduling algorithm partitions the ready queue into several separate queues.



Multilevel Queue Scheduling

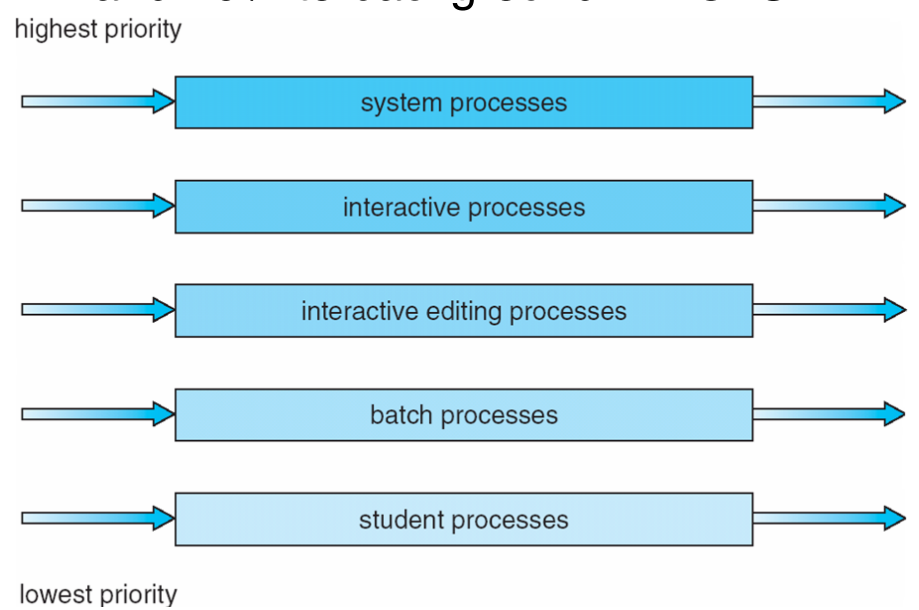
- In multi-level queue scheduling,
  - each process is assigned to some queue,
    - based on some property of the process (e.g. priority, process type, memory size, …)
  - each queue may have its <u>own scheduling algorithm</u>
    - e.g. for the two-level queue:
      - the foreground queue might be scheduled by RR
      - the background queue might be scheduled by FCFS



  - in addition, there must be <u>scheduling among</u> the queues

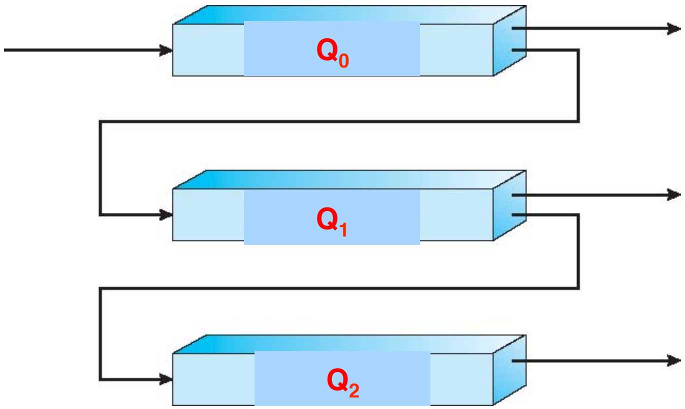
Multilevel Queue Scheduling (cont)

- One possibility is to let each queue have absolute priority over lower-priority queues.
  - fixed priority scheduling; There is the possibility of starvation.
    - e.g., serve all from foreground then from background.
- Another possibility is to time slice among the queues.
  - each queue gets a certain amount of CPU time which it can schedule amongst its processes
    - e.g. 80% to foreground in RR and 20% to background in FCFS
- Here is another example:



Multilevel Feedback Queue

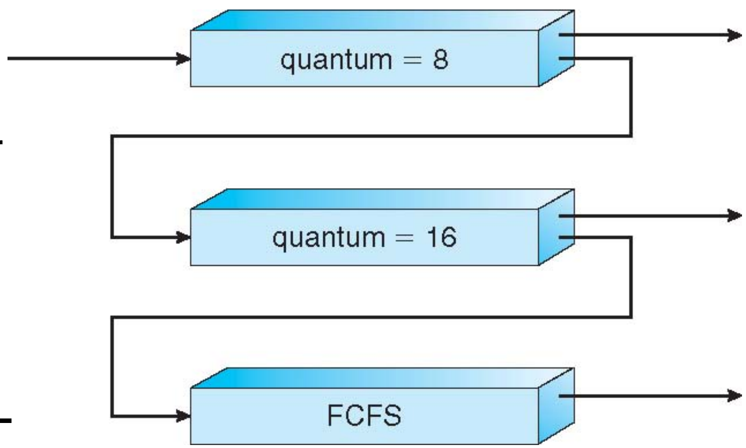
- A process can move between the various queues; aging can be implemented this way
  - Example:
    - New process enters Q_0
    - Processes move based on a rule
    - Q_1 is served only if Q_0 empty
    - Q_2 is served only if Q_1 empty



- Multilevel-feedback-queue scheduler is defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine which queue a process will enter when that process needs service
  - method used to determine
    - when to upgrade a process
    - when to demote a process

Example of Multilevel Feedback Queue

- Consider the pervious multilevel feedback queue with three queues, assume:
  - ***Q_0*** : with a time quantum of 8 ms (like RR or FCFS but with a time quantum and preemptive)
  - ***Q1*** : with a time quantum of 16 ms (like RR or FCFS but with a time quantum and preemptive)
  - ***Q2*** : FCFS
  - A process that arrives for Q_0 will preepmt a process is Q_1 or Q2. A process in Q1 will pre-empt a process in Q2.



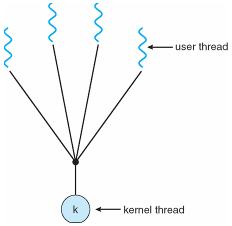
- **Scheduling**:
  - A new process is put in queue Q_0 which is served FCFS
    - When it gains CPU, job receives 8 milliseconds
    - If it does not finish in 8 milliseconds, job is preempted and moved to queue Q_1
  - At Q_1 job is again served FCFS and receives 16 additional milliseconds
    - If it still does not complete, it is moved to queue Q_2

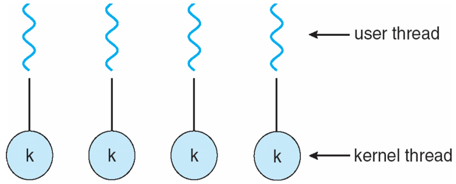
Lecture Outline

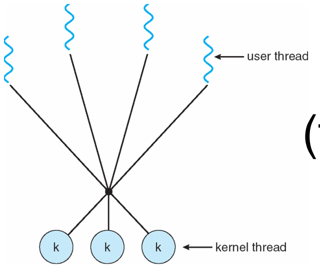
- Basic Concepts: Scheduling Criteria
- Scheduling Algorithms
- **Thread Scheduling**

Thread Scheduling

- On operating systems that support threads, it is kernel-level threads, not processes, that are being scheduled by the OS.
  - User-level threads are managed by a thread library and the kernel is not aware of them
  - To run on a CPU, user-level threads must be mapped to an associated kernel-level thread

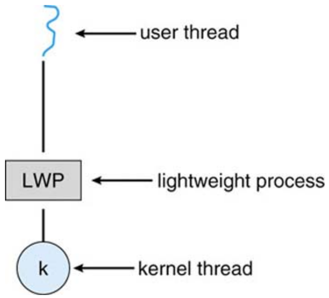






(from Ch 4)

  - The terms process scheduling and thread scheduling are sometimes used interchangeably.
- This mapping may be indirect and may use a lightweight process (LWP).



(from Ch 4)

Contention Scope

- On many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP;
  - this scheme is known as **process-contention scope (PCS)** since scheduling competition is within the same process
- To decide which kernel thread to schedule onto a CPU, the kernel uses **system-contention scope (SCS)**
  - With SCS, competition for the CPU is among all threads in the system
  - Systems using one-to-one model, use SCS only (e.g. Windows)
- Pthread API allows specifying either PCS or SCS during thread creation
  - **`PTHREAD_SCOPE_PROCESS`** schedules threads using PCS scheduling
  - **`PTHREAD_SCOPE_SYSTEM`** schedules threads using SCS scheduling.

Lecture Outline

- Basic Concepts: Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- **Multicore Scheduling**