CS330: Operating Systems

Process scheduling policies

Scheduling mechanism (recap)

- How is the list of ready processes managed?
- Each process is associated with three primary states: Running, Ready and Waiting. A process can moved to waiting state from running state, if needed.
- What if there are no processes in ready queue? Can that happen?
- There is always an idle process which executes HLT
- Can we classify the schedulers based on how they are invoked?
- Non-preemptive: triggered by the process, Preemptive: OS interjections
- What is a good scheduling strategy?
- Metrics: turn-around time, waiting time and response time

Scheduling mechanism (recap)

- How is the list of ready processes managed?
- Each process is associated with three primary states: Running, Ready and Waiting. A process can moved to waiting state from running state, if needed.
- What if there are no processes in ready queue? Can that happen?
- There is always an idle process which executes HLT
- Can we classify the schedulers based on how they are invoked?
- Non-preemptive: triggered by the process, Preemptive: OS interjections
- What is a good scheduling strategy?
- Metrics: turn-around time, waiting time and response time

Agenda: Process scheduling policies (OSTEP Ch7, Ch8)

First Come First Served (FCFS)

- FIFO queue based non-preemptive scheduling
- Example

First Come First Served (FCFS)

- FIFO queue based non-preemptive scheduling
- Example
- Advantages
  - Easy to implement
- Issues with FCFS
  - Convoy effect
  - Not suitable for interactive applications

Shortest Job First (SJF)

- Select the process with shortest CPU burst
- Pick the next process only when the current process is finished (non-preemptive)
- Example

Shortest Job First (SJF)

- Select the process with shortest CPU burst
- Pick the next process only when the current process is finished (non-preemptive)
- Example
- Optimal on waiting time and turnaround time
- Not realistic (how can we know the execution time?)

Shortest Time to Completion First (STCF)

- Pick the process with shortest remaining time when a new process arrives in the ready queue (SRTF)
- Example
- Improves the efficiency of SJF at the cost of more context switches

Round-robin scheduling

- Preemptive scheduling with time slicing
- Ready queue is maintained as a circular queue
- At end of the time quantum, If there are other processes in the queue
  - Current process goes to the TAIL of the queue
  - Next process is picked up from the HEAD of the queue
- New processes are added to the TAIL of the queue
- Design choice: size of time quantum

Priority scheduling

- Select the process with highest priority
- Can be preemptive and non-preemptive
- SJF: priority defined by job length
- Advantages: practical (no assumptions)
- Disadvantages: Starvation

Problem formulation with I/O bursts

Process	Arrival Time	CPU bursts	I/O bursts
P1	0	0-3, 7-9, 14-15	3-7,9-14
P2	2	2-10, 12-15	10-12
P3	3	3-4, 10-11	4-10

- Most processes goes through a series of CPU and I/O bursts
- Looks complicated for analysis, can it be simplified?

Problem formulation with I/O bursts

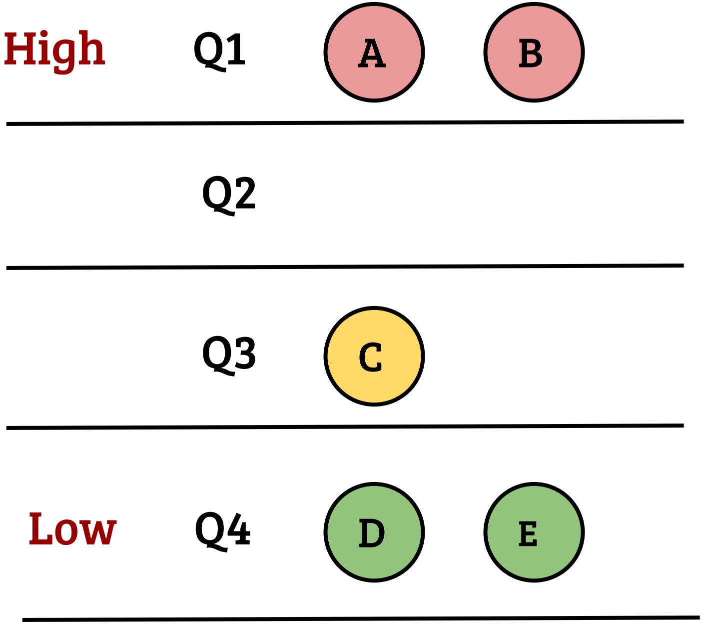
Process	Arrival Time	CPU bursts	I/O bursts
P1	0	0-3, 7-9, 14-15	3-7,9-14
P2	2	2-10, 12-15	10-12
P3	3	3-4, 10-11	4-10

- Most processes require a series of CPU and I/O bursts
- Looks complicated for analysis, can it be simplified?
- Every CPU burst is treated as a new process where the CPU burst start is the process arrival time and burst length is the execution time

Basic scheduling policies (recap)

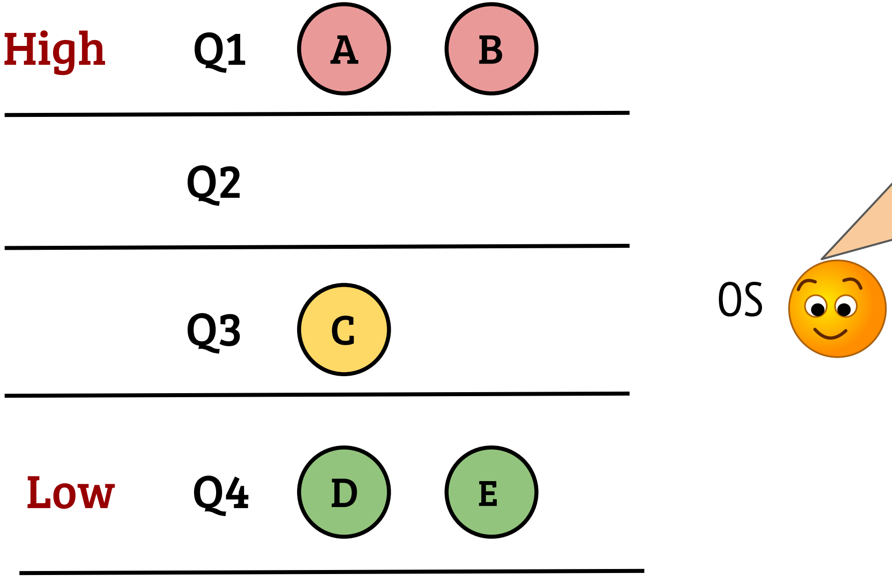
- Scheduling metrics: turnaround time, waiting time, response time
- Fast come first serve (FCFS)
  - Simple but inefficient (convoy effect)
- Shortest job first (SJF) and Shortest time to completion first (STCF)
  - Optimal and efficient. Issues: unrealistic, starvation
- Round robin (RR)
  - Good response time, Issues: scheduling overheads
- Priority scheduling
  - Starvation

Static priority based scheduling



- Processes are assigned to different queues based on their priority
- Process from the non-empty highest priority queue is always picked
- Different queues may implement different schemes within a queue
- Main concern: Starvation
  - Ex: High priority processes hug the CPU

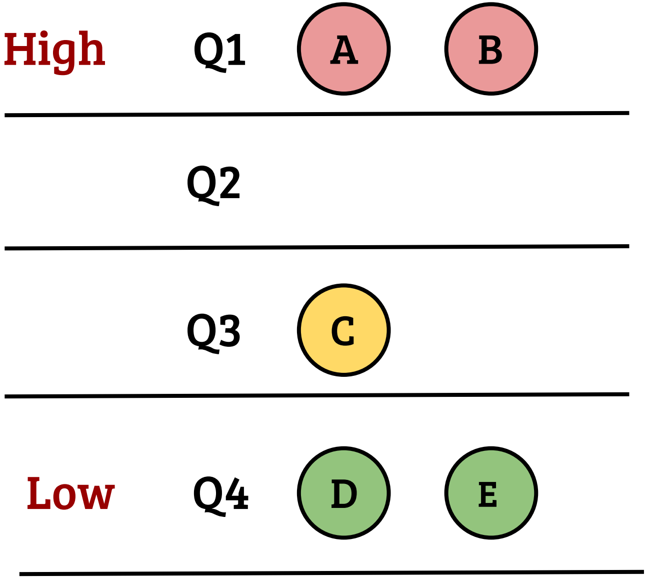
Multilevel feedback queue



Dynamically adjust priorities such that

1. Interactive applications are responsive
2. Short jobs do not suffer
3. No starvation
4. No user can trick the scheduler

Multilevel feedback queue



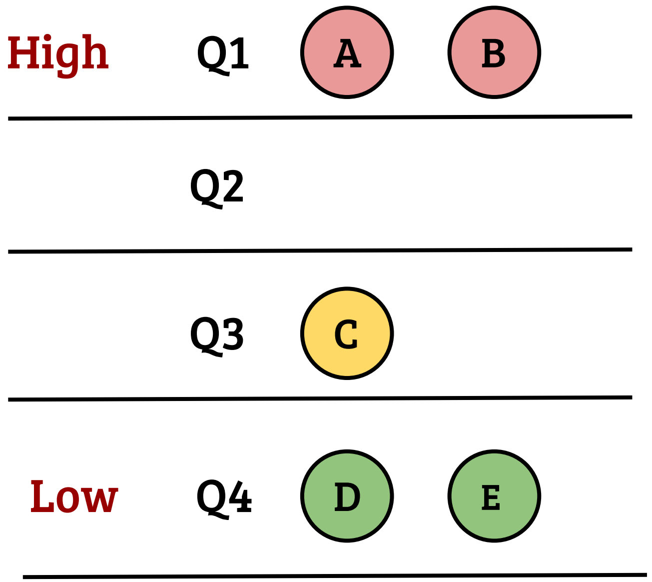
OS 

Dynamically adjust priorities such that

1. Interactive applications are responsive
2. Short jobs do not suffer
3. No starvation
4. No user can trick the scheduler

- Basic multi-level strategy
    - Pick a process from highest priority queue
    - Within a queue, apply RR

Multilevel feedback queue: Dynamic priorities



- A process is assigned the highest priority when it is created
- If the process consumes the slice (scheduler invoked because of timer), its priority is reduced
- If the process relinquishes the CPU (I/O wait etc.), its priority remain the same

Multilevel feedback queue: Dynamic priorities

- A process is assigned the highest priority when

- How does this strategy work for short jobs?
- How does the strategy work for interactive jobs?
- Does it avoid starvation?
- Can a user trick the scheduler?

etc.), its priority remain the same

MLFQ: Approximation of SJF

- MLFQ can approximate SJF because
  - Long running jobs are moved to low priority queues
  - New jobs are added to highest priority queue
- A shorter job may not get a chance to execute for a small duration. What is the upper bound?

MLFQ: Approximation of SJF

- MLFQ can approximate SJF because
  - Long running jobs are moved to low priority queues
  - New jobs are added to highest priority queue
- A shorter job may not get a chance to execute for a small duration. What is the upper bound?
- (# of jobs in the highest priority queue + 1) X (time quantum)

Multilevel feedback queue: Dynamic priorities

High Q1 A B

- A process is assigned the highest priority when

- How does this strategy work for short jobs?
- Works nicely, approximates SJF
- How does the strategy work for interactive jobs?
- Does it avoid starvation?
- Can a user trick the scheduler?

etc.), its priority remain the same

MLFQ: Interactive jobs

- MLFQ favors interactive jobs because
  - Interactive jobs maintain the highest priority as they relinquish the CPU before quantum expires
  - Long running jobs are moved to low priority queues

MLFQ: Interactive jobs

- MLFQ favors interactive jobs because
  - Interactive jobs maintain the highest priority as they relinquish the CPU before quantum expires
  - Long running jobs are moved to low priority queues
- Conclusion: In a steady state, interactive jobs compete with short and other interactive jobs

Multilevel feedback queue: Dynamic priorities

- How does this strategy work for short jobs?
- Works nicely, approximates SJF
- How does the strategy work for interactive jobs?
- Works pretty well as interactive jobs retain priority
- Does it avoid starvation?
- Can a user trick the scheduler?

MLFQ: Starvation and other issues

- Long running processes may starve with the proposed scheme
- Additionally, permanent demotion of priority hurts processes which change their behavior
    - Example: A process performing a lot of computation only at start gets pushed to a low priority queue permanently
- How to avoid the above issues?

MLFQ: Starvation and other issues

- Long running processes may starve with the proposed scheme
- Additionally, permanent demotion of priority hurts processes which change their behavior
    - Example: A process performing a lot of computation only at start gets pushed to a low priority queue permanently
- How to avoid the above issues?
    - Periodic priority boost: all processes moved to high priority queue
    - Priority boost with aging: recalculate the priority based on scheduling history of a process

Multilevel feedback queue: Dynamic priorities

- How does this strategy work for short jobs?
- Works nicely, approximates SJF
- How does the strategy work for interactive jobs?
- Works pretty well as interactive jobs retain priority
- Does it avoid starvation?
- No. Requires additional mechanism like priority boost.
- Can a user trick the scheduler?

MLFQ: The tricky user

- A smart user can maintain highest priority for long running processes by exploiting the scheduling strategy. How?

MLFQ: The tricky user

- A smart user can maintain highest priority for long running processes by exploiting the scheduling strategy. How?
- Assumption: user knows the time quantum

MLFQ: The tricky user

- A smart user can maintain highest priority for long running processes by exploiting the scheduling strategy. How?
- Assumption: user knows the time quantum
- Strategy: Voluntarily release the CPU before time quantum expires
- Result: Batch process competes with other interactive processes!

MLFQ: The tricky user

- A smart user can maintain highest priority for long running processes by exploiting the scheduling strategy. How?
- Assumption: user knows the time quantum
- Strategy: Voluntarily release the CPU before time quantum expires
- Result: Batch process competes with other interactive processes!
- Core of the issue: binary history regarding a process

MLFQ: The tricky user

- A smart user can maintain highest priority for long running processes by exploiting the scheduling strategy. How?
- Assumption: user knows the time quantum
- Strategy: Voluntarily release the CPU before time quantum expires
- Result: Batch process competes with other interactive processes!
- Core of the issue: binary history regarding a process
    - MLFQ: Process consumed or not consumed the quantum
    - Advanced MLFQ: Better accounting, variable quantums

Multilevel feedback queue: Dynamic priorities

- How does this strategy work for short jobs?
- Works nicely, approximates SJF
- How does the strategy work for interactive jobs?
- Works pretty well as interactive jobs retain priority
- Does it avoid starvation?
- No. Requires additional mechanism like priority boost.
- Can a user trick the scheduler?
- Yes. Additional history regarding execution is required to be maintained

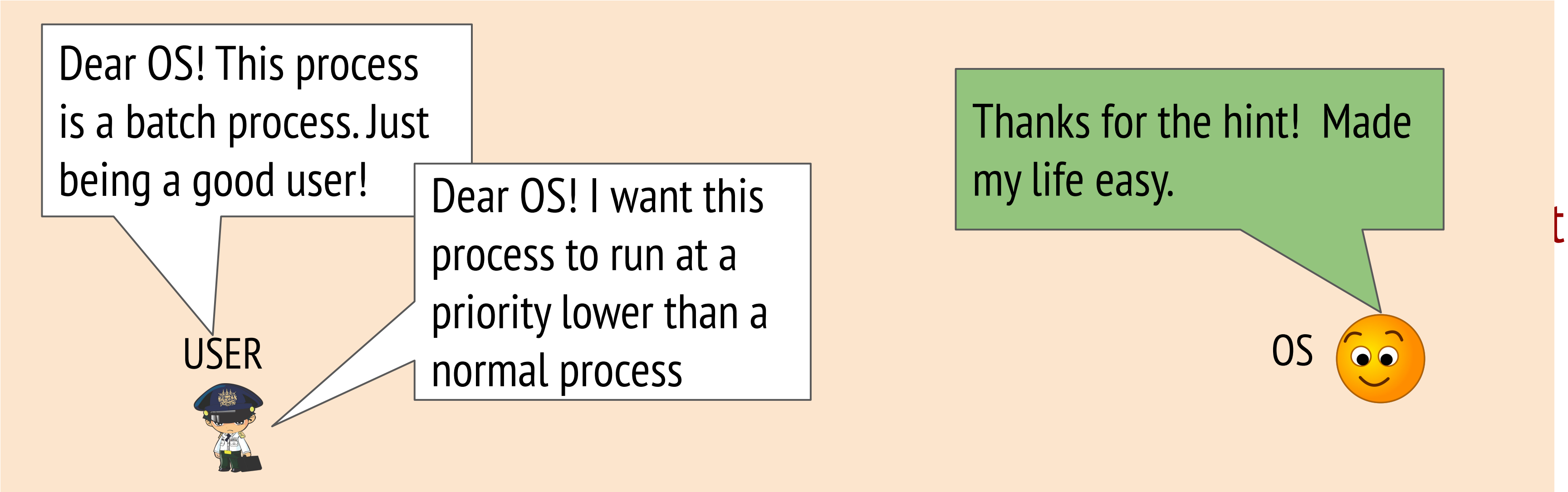
Scheduling is much more complex in a real OS!

- Scheduling requirement of different processes in the system are different
  - Real-time processes: Should meet strict deadlines
  - Interactive processes: Responsive scheduling
  - Batch processes: Starvation free scheduling

Scheduling is much more complex in a real OS!

- Scheduling requirement of different processes in the system are different
  - Real-time processes: Should meet strict deadlines
  - Interactive processes: Responsive scheduling
  - Batch processes: Starvation free scheduling
- Well intentioned users should be able to influence the scheduling policy in a positive manner

Scheduling is much more complex in a real OS!



- Well intentioned users should be able to influence the scheduling policy in a positive manner

Scheduling is much more complex in a real OS!

- Scheduling requirement of different processes in the system are different
  - Real-time processes: Should meet strict deadlines
  - Interactive processes: Responsive scheduling
  - Batch processes: Starvation free scheduling
- Well intentioned users should be able to influence the scheduling policy in a positive manner
- Greed of greedy users should be controlled by the OS

Scheduling is much more complex in a real OS!

Dear OS! This process requires higher priority than other normal processes. You know what, it is very interactive.
Not really! Just trying to fool you.

Buddy! You can fool me for a little while. I will catch you eventually.

USER

OS

- Greed of greedy users should be controlled by the OS

Scheduling is much more complex in a real OS!

- Scheduling requirement of different processes in the system are different
  - Real-time processes: Should meet strict deadlines
  - Interactive processes: Responsive scheduling
  - Batch processes: Starvation free scheduling
- Well intentioned users should be able to influence the scheduling policy in a positive manner
- Greed of greedy users should be controlled by the OS
- Conclusion: OS scheduling should provide flexibility while being auto-tuning in nature

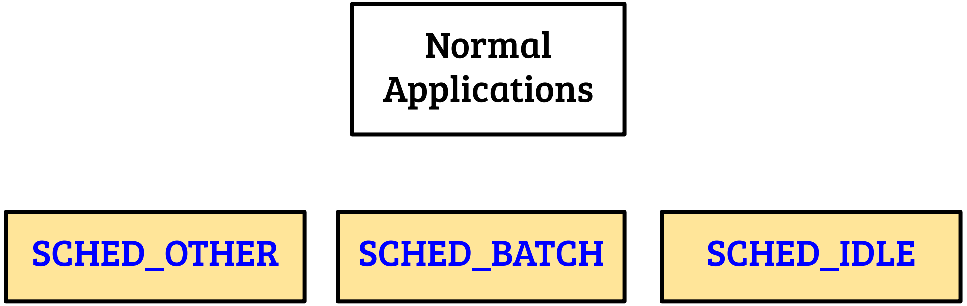
Linux scheduling classes: Real time applications

Real time applications

SCHED_FIFO **SCHED_RR**

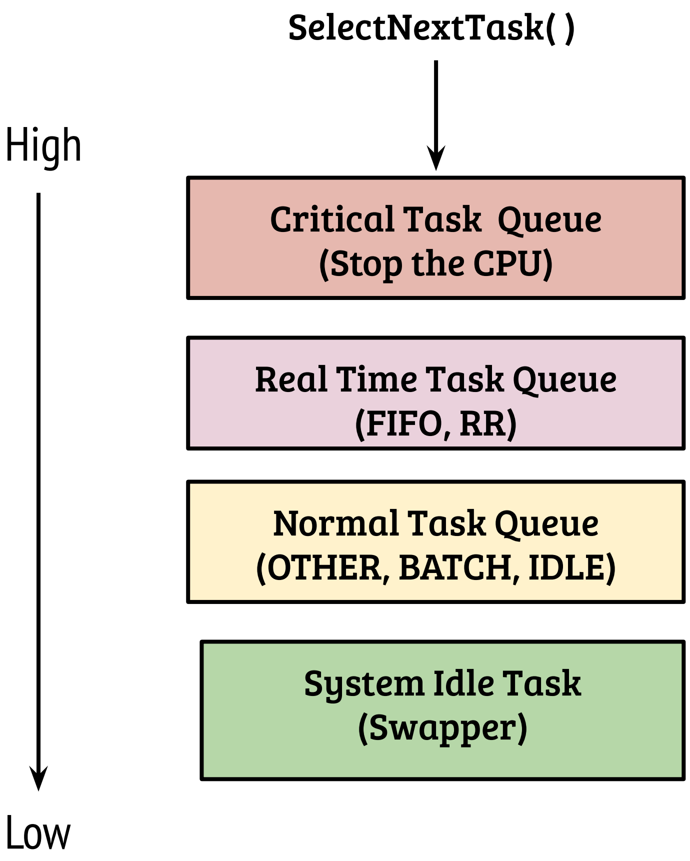
- Real time applications are always higher priority than normal processes
- Priority value: 1 to 99 (In Linux, lower value ⇒ higher priority)
- FIFO: Run to completion
- RR: Round robin within a given priority-level
- *sched_setscheduler* system call to define scheduling class and priorities

Linux scheduling classes: normal applications



- SCHED_OTHER: Default policy, OS dynamic priorities and variable time slicing comes into picture
- SCHED_BATCH: Assume CPU bound while calculating dynamic priorities
- SCHED_IDLE: Very low priority jobs

Selecting the next task



- A task is picked from the non-empty highest priority queue
- Critical task queue contains tasks which require immediate attention: hardware events, restart etc.
- Normal task queue (a.k.a fair scheduling class) implements the heuristics to self-adjust
- If all the queues are empty, swapper task is scheduled (HLT the CPU)

Normal (fair) scheduling class

- 40 priority levels (100 to 139)
- Every process starts with a default priority of 120
- Linux provides *nice* system call to adjust the static priority
    - *nice(int x)*, where x is between 19 to -20
    - nice(19) ⇒ Move the process to lowest priority queue i.e., 139
    - nice(-20) ⇒ Move the process to highest priority queue i.e., 100

Normal (fair) scheduling class

- 40 priority levels (100 to 139)
- Every process starts with a default priority of 120
- Linux provides *nice* system call to adjust the static priority
  - *nice(int x)*, where x is between 19 to -20
  - nice(19) ⇒ Move the process to lowest priority queue i.e., 139
  - nice(-20) ⇒ Move the process to highest priority queue i.e., 100
- Dynamic priority is calculated by the Linux kernel considering the interactiveness of the process
  - More interactive processes move towards the priority level 100