Operating Systems

Scheduling (Virtual CPU)

Hongliang Liang, BUPT

Sep. 2023

Topics covered in this lecture

Scheduling has two aspects: 1) how to switch from one process to another and 2) what process should run next

- Mechanism: context switch (how to switch)
- Mechanism: preemption (keeping control)
- Policy: scheduling (where to switch to)

Scheduling

How does the kernel switch from one process to another?

- System has several `ready` processes
- For simplicity, we assume one CPU

How does the kernel stay in control?

Scheduling

How does the kernel switch from one process to another?

- System has several `ready` processes
- For simplicity, we assume one CPU

How does the kernel stay in control?

- Processes may `yield()` or execute I/O
- Configurable timer interrupts let OS take control

Scheduling

How does the kernel switch from one process to another?

- System has several `ready` processes
- For simplicity, we assume one CPU

How does the kernel stay in control?

- Processes may `yield()` or execute I/O
- Configurable timer interrupts let OS take control

How does the kernel switch from one process to another?

- Context switch saves running process' state in kernel structure
- Context switch restores state of next process
- Context switch transfers control to next process and "returns"

Scheduling

How does the kernel switch from one process to another?

- System has several `ready` processes
- For simplicity, we assume one CPU

How does the kernel stay in control?

- Processes may `yield()` or execute I/O
- Configurable timer interrupts let OS take control

How does the kernel switch from one process to another?

- Context switch saves running process' state in kernel structure
- Context switch restores state of next process
- Context switch transfers control to next process and "returns"

Note: a context switch is ***transparent*** to the processes

Mechanism: context switch

A context switch is a mechanism that allows the OS to store the current process state and switch to another, previously stored context.

Reasons for a context switch:

- The process completes/exits
- The process executes a slow HW operation (e.g., loading from disk) and the OS switches to another **ready** task
- The hardware requires OS help and issues an interrupt
- The OS decides to preempt the task and switch to another task (i.e., the process has used up its time slice)

Mechanism: context switch (pseudo code)

- A function call that returns asynchronously: process A starts the execution of the context switch but process B continues execution after the return of the function.
  - The function saves all registers in a scratch area (on the process' kernel stack or in a predefined area of the `task struct`).
  - The OS switches address spaces.
  - The function restores all registers from the scratch area.
  - The OS returns to process B.

Mechanism: context switch (example, one address space)

```
# void ctx_swtch(struct context *old, struct context *new)
# Save old registers
movl 4(%esp), %eax  # load ptr to old into eax
popl 0(%eax)        # store old IP to old
movl %esp, 4(%eax)  # store stack pointer
movl %ebx, 8(%eax)  # store other registers
...
movl %ebp, 28(%eax)

# Load new registers
movl 4(%esp), %eax  # load ptr to new into eax
movl 28(%eax), %ebp # restore other registers
...
movl 8(%eax), %ebx
movl 4(%eax), %esp  # stack switch (from now on new stack)
pushl 0(%eax)       # store return addr
```

Mechanism: preemption

If a task never gives up control (`yield()`), exits, or performs I/O then it could run forever and the OS could not gain control.

Mechanism: preemption

If a task never gives up control (`yield()`), exits, or performs I/O then it could run forever and the OS could not gain control.

The OS therefore sets a **timer** before scheduling a process. If the timer expires, the hardware interrupts the execution of the process and switches to the kernel. The kernel then decides if the process may continue.

What is a scheduling policy?

The context switch *mechanism* takes care of ***how*** the kernel switches from one process to another, namely by storing its context and restoring the context of the other process.

The scheduling *policy* determines ***which*** process should run next. If there is only one "ready" process then the answer is easy. If there are more processes then the policy decides in which order processes execute.

Figure 1: Scheduling

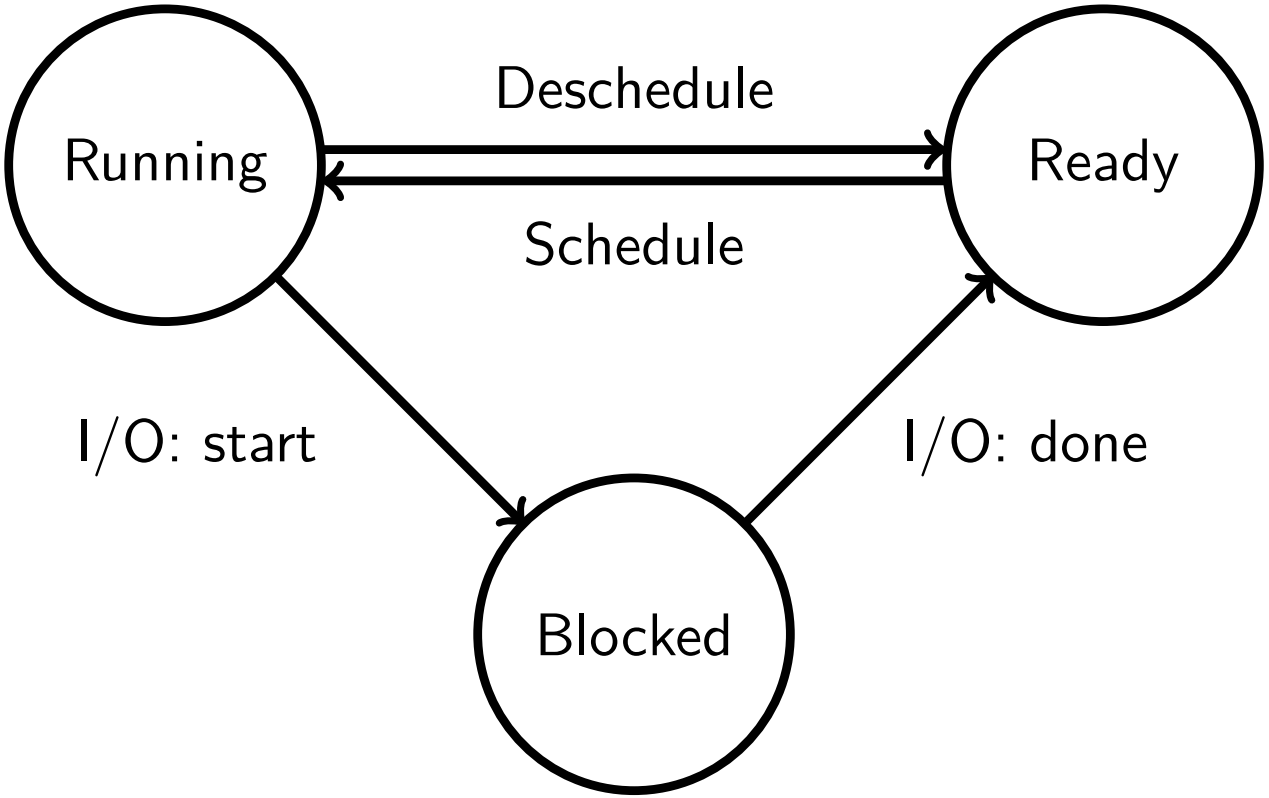
Scheduler metrics

Scheduler metrics

When analyzing scheduler policies, we use the following terms:

- **Utilization:** what fraction of time is the CPU executing a program (goal: maximize)
- **Turnaround time:** total time needed to complete a job, $T_{completion} - T_{arrival}$ (goal: minimize)
- **Response time:** time from when the job arrives to the first time it is scheduled, $T_{firstrun} - T_{arrival}$ (goal: minimize)
- **Fairness:** all processes get same amount of CPU over time (goal: no starvation)
- **Progress:** allow processes to make forward progress (goal: minimize kernel interrupts)

Reminder: process states



Scheduler implementation

Simplest form: **each state has an associated queue of tasks**.

```
task_struct_t *get_next_task() {
  // consult task queues to find next runnable task
}

void enqueue_task(task_struct_t *task) {
  // set task to ready

  // update ready queue so that it can run at its turn
}
```

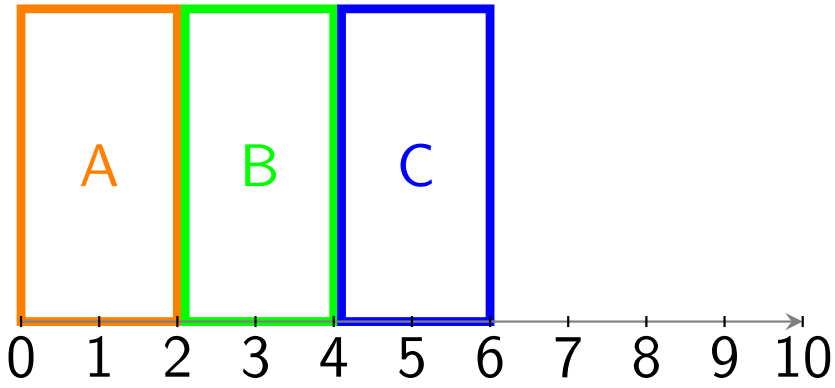
Scheduling assumptions

Let's understand scheduler policies step by step. We start with some simplifying assumptions

- Each job runs for the same amount of time
- All jobs arrive at the same time
- Once started, each job runs to completion
- All jobs only use the CPU (no I/O)
- Run-time of jobs is known
- For now, we assume a single CPU

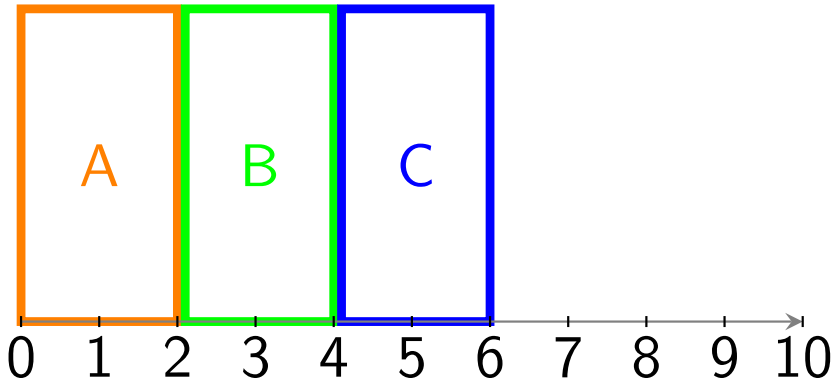
First Come(In), First Served(Out) —-FCFS(FIFO)

- Tasks A, B, C of `len=2` arrive at `T=0`. (0,2)
- Average turnaround
  - `(2+4+6)/3 = 4`
- Average response
  - `(0+2+4)/3 = 2`



First Come(In), First Served(Out) —-FCFS(FIFO)

- Tasks A, B, C of `len=2` arrive at `T=0`. (0,2)
- Average turnaround
  - `(2+4+6)/3 = 4`
- Average response
  - `(0+2+4)/3 = 2`



Finding: easy, simple, straight forward.

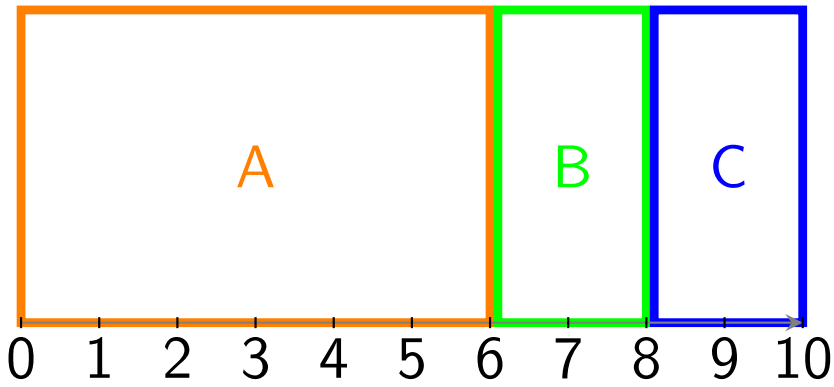
What are drawbacks?

Scheduling assumptions

- ~~***Each job runs for the same amount of time***~~
- All jobs arrive at the same time
- Once started, each job runs to completion
- All jobs only use the CPU (no I/O)
- Run-time of jobs is known

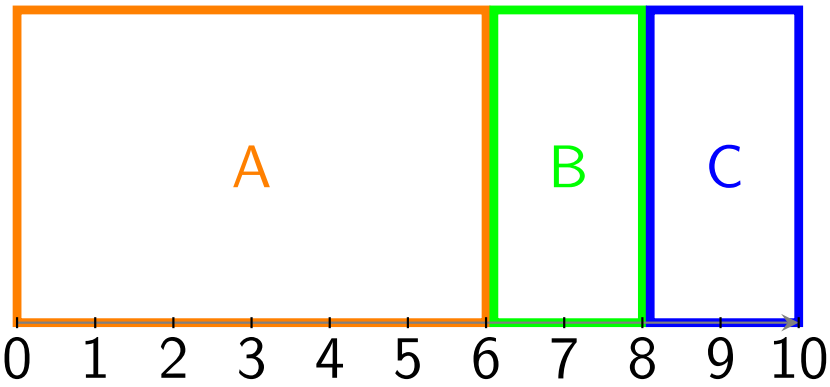
FCFS(FIFO) challenge: long running task

- Task A is now of `len=6`
- Average turnaround
  - `(6+8+10)/3 = 8`
- Average response
  - `(0+6+8)/3 = 4.7`



FCFS(FIFO) challenge: long running task

- Task A is now of `len=6`
- Average turnaround
  - `(6+8+10)/3 = 8`
- Average response
  - `(0+6+8)/3 = 4.7`

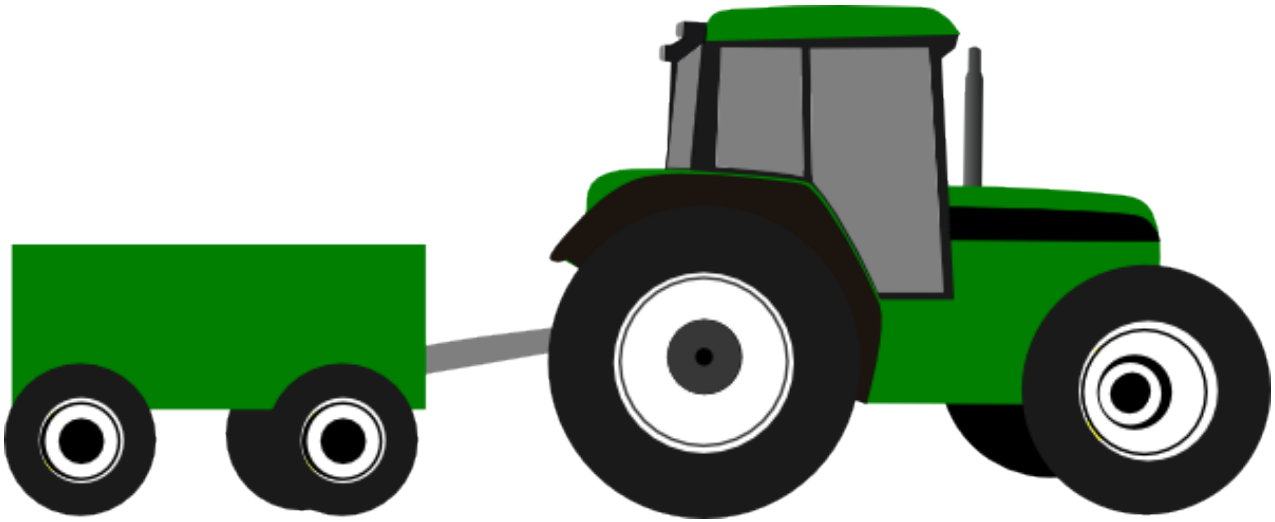


Finding: long jobs delay short jobs, turnaround/response time suffer!

SJF: Shortest Job First

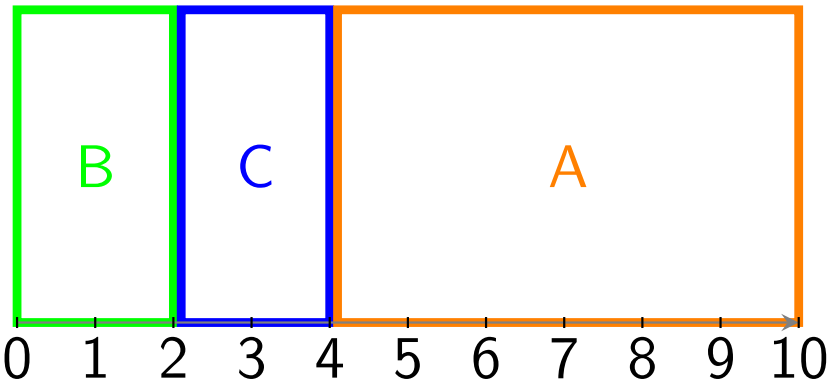
- Long running tasks delay other tasks (**convoy effect**: one long running task delays many short running tasks, like a truck followed by many cars)
- Short jobs must wait for completion of long task

New scheduler: choose ready job with shortest runtime!



SJF: turnaround

- Task A is now of `len=6`
- Average turnaround
  - `(2+4+10)/3 = 5.3`
- Average response
  - `(0+2+4)/3 = 2`

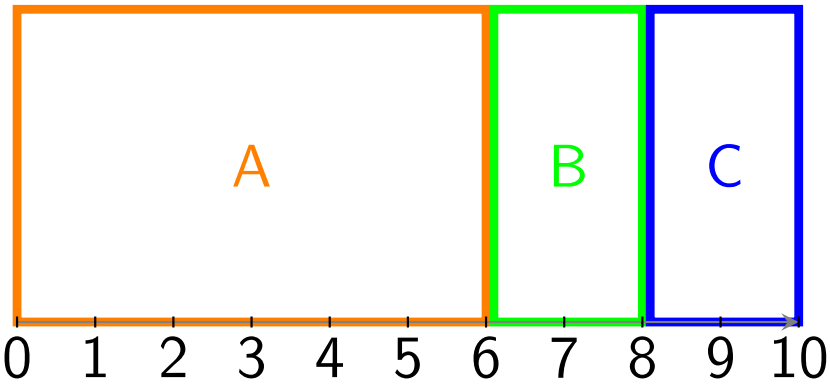


Scheduling assumptions

- ~~Each job runs for the same amount of time~~
- ~~***All jobs arrive at the same time***~~
- Once started, each job runs to completion
- All jobs only use the CPU (no I/O)
- Run-time of jobs is known

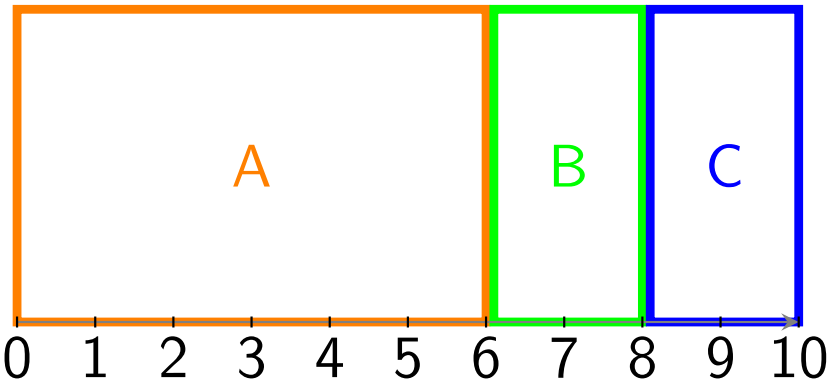
SJF: another convoy!

- Tasks B, C now arrive at 1
- Average turnaround
  - `(6+7+9)/3 = 7.3`
- Average response
  - `(0+5+7)/3 = 4`



SJF: another convoy!

- Tasks B, C now arrive at 1
- Average turnaround
  - `(6+7+9)/3 = 7.3`
- Average response
  - `(0+5+7)/3 = 4`



Finding: long running jobs cannot be interrupted, delay short jobs

Preemptive scheduling

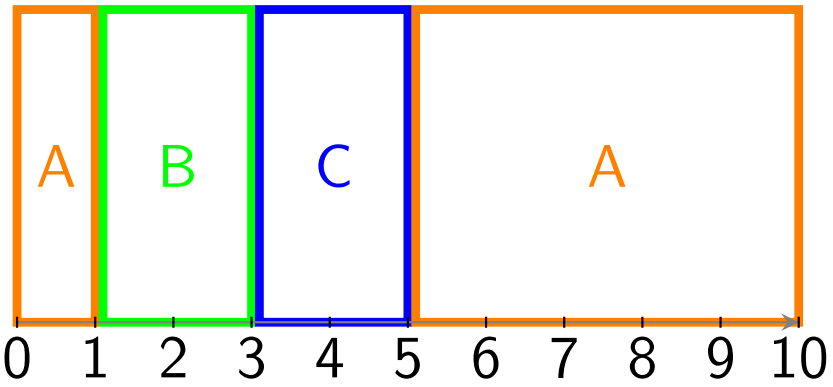
- Previous schedulers (FIFO, SJF) are non-preemptive.
  Non-preemptive schedulers only switch to another process if the current process gives up the CPU **voluntarily**.
- **Preemptive** schedulers may take CPU control at any time, switching to another process according to the scheduling policy.

Preemptive scheduling

- Previous schedulers (FIFO, SJF) are non-preemptive. Non-preemptive schedulers only switch to another process if the current process gives up the CPU **voluntarily**.
- **Preemptive** schedulers may take CPU control at any time, switching to another process according to the scheduling policy.
- New scheduler: shortest time to completion first (STCF), always run the job that will complete the fastest.

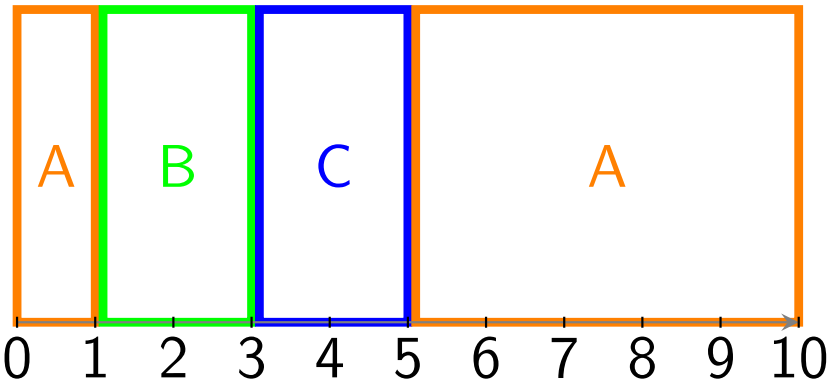
Preemptive scheduling: STCF

- Tasks B, C now arrive at 1
- Average turnaround
  - `(2+4+10)/3 = 5.3`
- "First" response
  - `(0+0+2)/3 = 0.7`
  - Task A takes a break!



Preemptive scheduling: STCF

- Tasks B, C now arrive at 1
- Average turnaround
  - `(2+4+10)/3 = 5.3`
- "First" response
  - `(0+0+2)/3 = 0.7`
  - Task A takes a break!



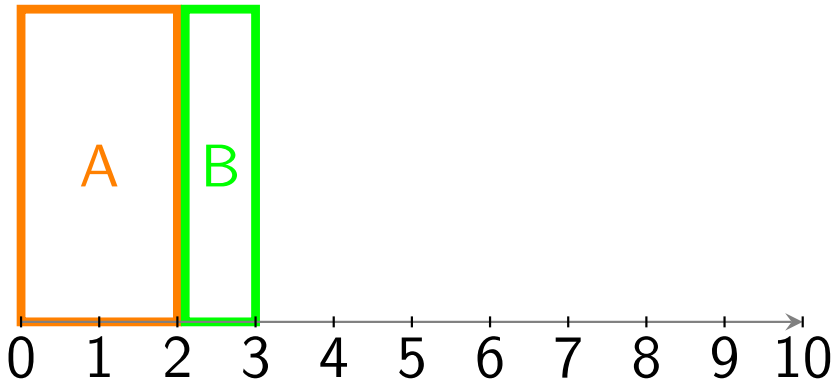
Finding: reschedule **whenever** new jobs arrive, **prioritize** short jobs

Next metric: response time

- So far, we have optimized for turnaround time (i.e., completing the tasks as fast as possible).
- On an interactive system, response time is equally important, i.e., how long it takes until a task is scheduled.

Turnaround versus response time

- Tasks A (0,2) and B (1, 1)
- B turnaround: 2
- B response time: 1



Scheduling assumptions

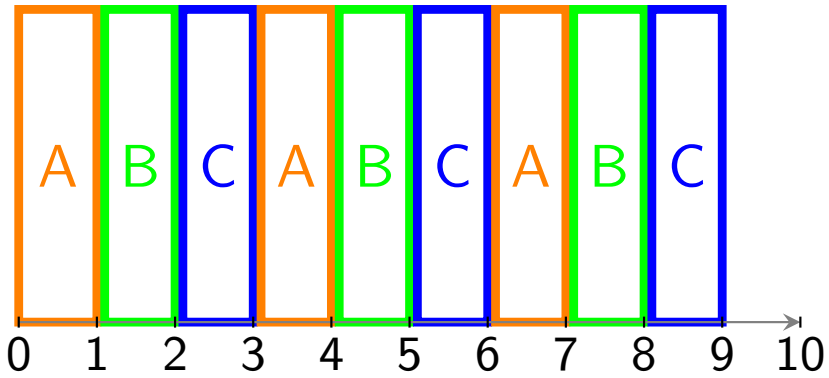
- ~~Each job runs for the same amount of time~~
- ~~All jobs arrive at the same time~~
- ~~***Once started, each job runs to completion***~~
- All jobs only use the CPU (no I/O)
- Run-time of jobs is known

Round robin (RR)

- Previous schedulers optimize for turnaround.
- Optimize response time: alternate ready processes **every fixed-length time slice**.

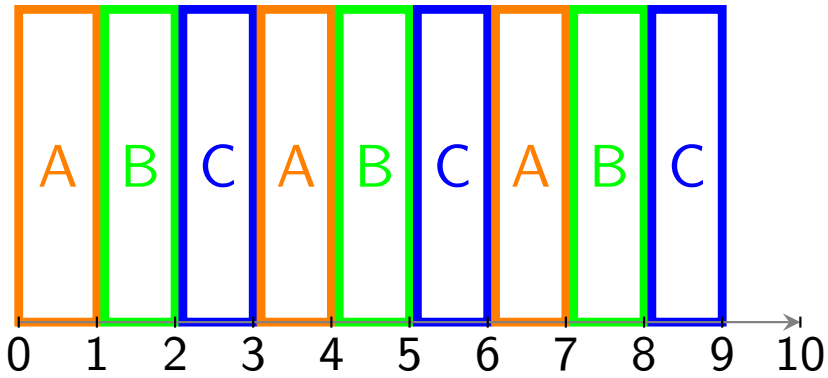
Round robin

- Tasks A, B, C (0, 3)
- Average response time
  - (0+1+2)/3 = 1
- Compare to FIFO where average response time is 3
- Turnaround increases
  - (7+8+9)/3 = 8 for RR
    (3+6+9)/3 = 6 for SJF



Round robin

- Tasks A, B, C (0, 3)
- Average response time
  - (0+1+2)/3 = 1
- Compare to FIFO where average response time is 3
- Turnaround increases
  - (7+8+9)/3 = 8 for RR
    (3+6+9)/3 = 6 for SJF



Finding: responsiveness increases turnaround (for equally long tasks)

Effectiveness of Round-Robin Scheduling

- The number of jobs
  More jobs -> slower response times
- The length of the time slice (scheduling quantum)
  - Longer time slice -> slower response times
  - Shorter time slice -> more overhead
  - 10 ms to 100 ms is often a reasonable compromise

Scheduling assumptions

- ~~Each job runs for the same amount of time~~
- ~~All jobs arrive at the same time~~
- ~~Once started, each job runs to completion~~
- ~~***All jobs only use the CPU (no I/O)***~~
- Run-time of jobs is known

I/O awareness

- So far, the scheduler only considers preemptive events (i.e., the timer runs out) or process termination/creation to reschedule.
- I/O is usually **incredibly slow** and can be carried out asynchronously

I/O awareness

- So far, the scheduler only considers preemptive events (i.e., the timer runs out) or process termination/creation to reschedule.
- I/O is usually **incredibly slow** and can be carried out asynchronously

Finding: scheduler must consider I/O, unused time used by others

Scheduling assumptions

- ~~Each job runs for the same amount of time~~
- ~~All jobs arrive at the same time~~
- ~~All jobs only use the CPU (no I/O)~~
- ~~***Run-time of jobs is known***~~

How can we design a scheduler that minimizes *response* time for interactive jobs while also minimizing *turnaround* time **without knowledge of job run-time**?

Advanced scheduling: multi-level feedback queue (MLFQ)

Goal: general purpose scheduling

Challenge: The scheduler must support both long running background tasks (batch processes) and low latency foreground tasks (interactive processes).

Advanced scheduling: multi-level feedback queue (MLFQ)

Goal: general purpose scheduling

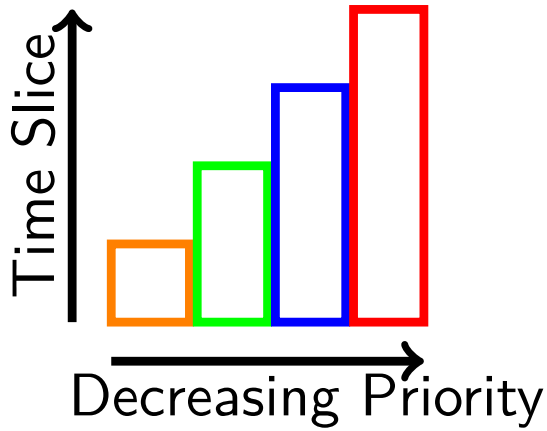
Challenge: The scheduler must support both long running background tasks (batch processes) and low latency foreground tasks (interactive processes).

- Batch process: response time not important, cares for long run times (reduce the cost of context switches, cares for lots of CPU, not when)
- Interactive process: response time critical, short bursts (context switching cost not important, not much CPU needed but frequently)
- First proposed by Corbato et al. in 1962 — who later won the ACM Turing Award

MLFQ: basics

Approach: multiple levels of round robin

- Each level with higher priority preempts all lower levels
- Process at higher level will always be scheduled first
- **High levels have short time slices, lower levels run for longer**



Set of rules **adjusts priorities dynamically**.

- Rule 1: if `prio(A) > prio(B)` then A runs.
- Rule 2: if `prio(A) == prio(B)` then A, B run in RR

MLFQ: priority adjustments

Goal: use past behavior as predictor for future behavior.

- Rule 3: processes start at top priority
- Rule 4: if a process uses up full time slice, **lower** its priority

MLFQ challenges: starvation

Low priority tasks may never run on a busy system.

- Rule 5: **periodically** move all jobs to the topmost queue

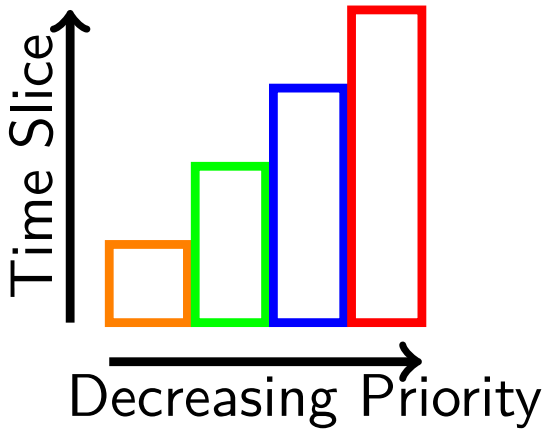
MLFQ challenges: gaming the scheduler

High priority process could yield before its time slice is up, remaining at high priority.

- Rule 4': account for total time at priority level (and not just time of the last time slice)

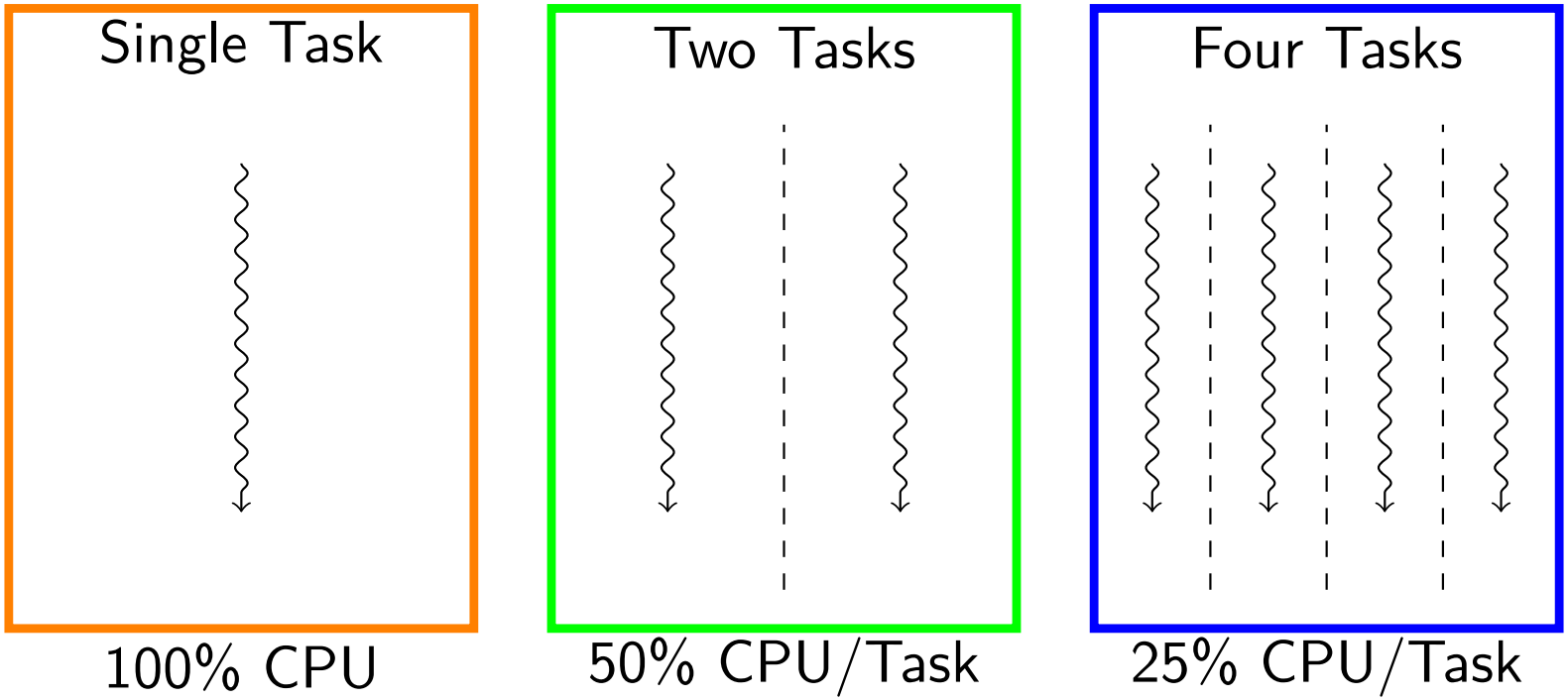
MLFQ summary

- Rule 1: if `prio(A) > prio(B)` then A runs.
- Rule 2: if `prio(A) == prio(B)` A, B run in RR
- Rule 3: new processes start with top priority
- Rule 4: lower process' priority when whole time slice is used
- Rule 5: periodically move all jobs to the topmost queue



CFS: Completely Fair Scheduler

- Idea: each task runs in parallel and consumes equal CPU share
- Approach: calculate time process receives on ideal processor
- Example: assume 4 processes are ready, so each would receive 1/4 of the CPU (add this time to the book keeping)



CFS: virtual runtime

On real hardware, we can run only a single task at once, so we have to introduce the concept of "virtual runtime." The virtual runtime of a task specifies when its next timeslice would start execution on the ideal multi-tasking CPU described above. In practice, the vruntime of a task is its actual runtime normalized to the total number of running tasks.

- CFS keeps track of how long each process should have executed on an ideal processor.
- For each time slice, it calculates the fraction each process would have received and keeps these balances in a tree.
- The process with the highest balance is then scheduled
- Linux used an O(1) scheduler based on multi-level feedback queues but switched to a completely fair scheduler in 2007

CFS: implementation

- Implementation: keep all processes in a **red-black tree**, sorted by maximum execution time (keep track of their positive balance)
- Scheduling
  - Schedule leftmost process (the one with the highest balance)
  - If the process exits, remove it from the scheduling tree
  - On interrupt (end of time slice or I/O), reinsert the process into the tree at its new position
  - Repeat

Summary

- Context switch and preemption are fundamental mechanisms that allow the OS to remain in control and to implement higher level scheduling policies.
- Schedulers need to optimize for different metrics: utilization, turnaround, response time, fairness and forward progress
  - FIFO: simple, non-preemptive scheduler
  - SJF: non-preemptive, prevents process jams
  - STCF: preemptive, prevents jams of late processes
  - RR: preemptive, great response time, bad turnaround
  - MLFQ: preemptive, most realistic
  - CFS: fair scheduler by virtualizing time
- Insight: past behavior is good predictor for future behavior