Scheduling

Mendel Rosenblum

Scheduling

Optional readings:

Operating Systems: Principles and Practice:

Chapter 7 up through Section 7.2

What is CPU Scheduling?

- Given a dispatcher (a **mechanism**) that can switch between threads:
  - Some threads that are ready to run
  - A number of cores
- CPU Scheduling: A **policy**
  - Which thread should run on each core and for how long?

Approach: Single core first, then generalize to multiple cores

First-in-first-out (FIFO) scheduling

- Also called **non-preemptive scheduling**
- Approach:
  - Keep ready threads in a single list: the ready queue
  - When a thread becomes ready, add it to the back of the ready queue
  - Run the first thread on the queue until it exits or blocks.
- Problem with FIFO approach?

First-in-first-out (FIFO) scheduling problems

- Can cause starvation
- High response time

Add preemption: **time slices**

- Do preemptive scheduling
  - Limit time thread can run without a context switch
  - Called a ***time slice***
- Implement with a timer hardware
  - Run a thread but have a timer interrupt after some amount of time
  - Example: Linux 4ms time slice

Round robin scheduling

- Approach: Loop doing:
  - Run thread at front of ready queue for one time slice
  - Return to back of ready queue
  - Each thread gets equal share of the cores.

Tradeoffs on length of time slice

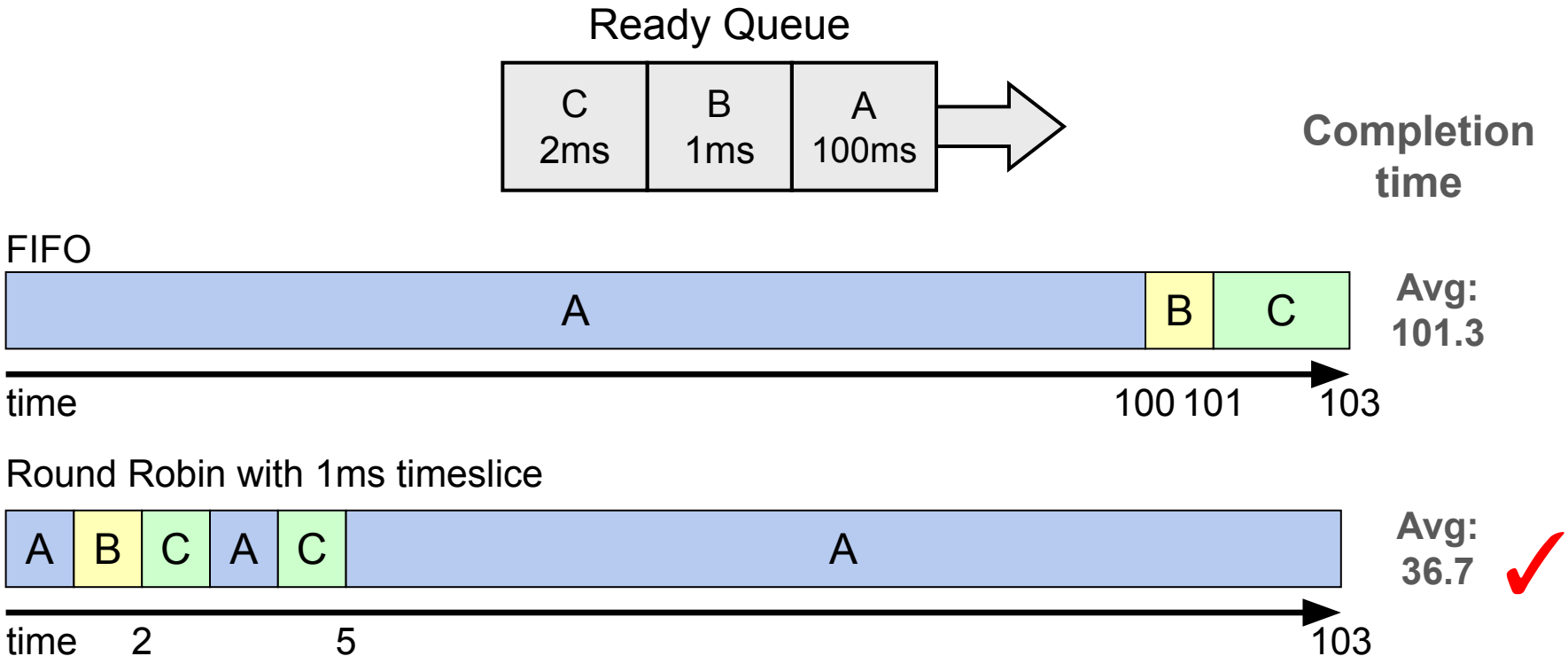
- What are the tradeoffs in setting the length of the time slice
  - Too long?
  - Too short?

- Too long: Becomes FIFO and its problems
- Too short: Too much switching overhead

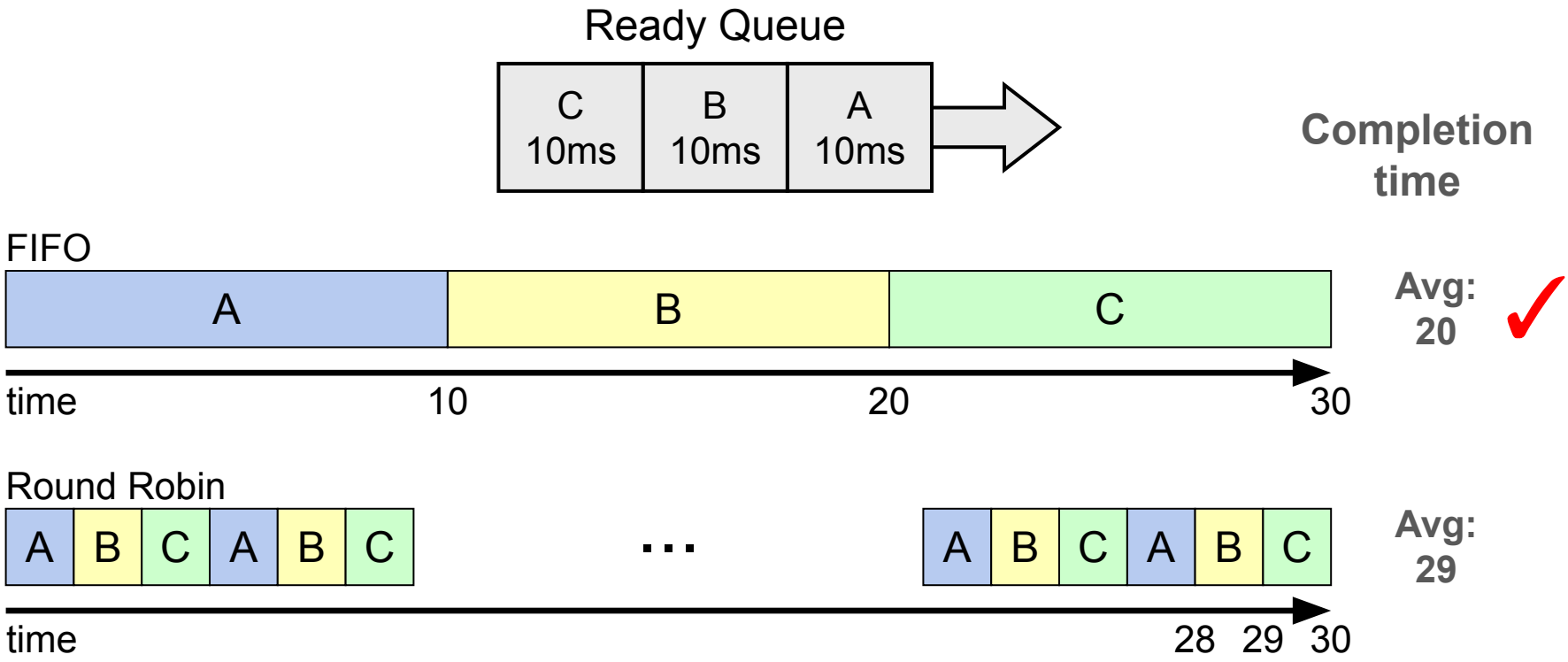
Scheduling Goals

- How do we decide whether a scheduling algorithm is good?
- Possible metrics:
  - Minimize response time
    - Keep user busy
  - Use resources efficiently:
    - Full utilization: keep both cores and disks busy
    - Low overhead: minimize context switch overheads
  - Fairness (distribute CPU cycles equitably)
    - Avoid starvation

Scenario #1: Compare FIFO & Round Robin



Scenario #2: Compare FIFO & Round Robin



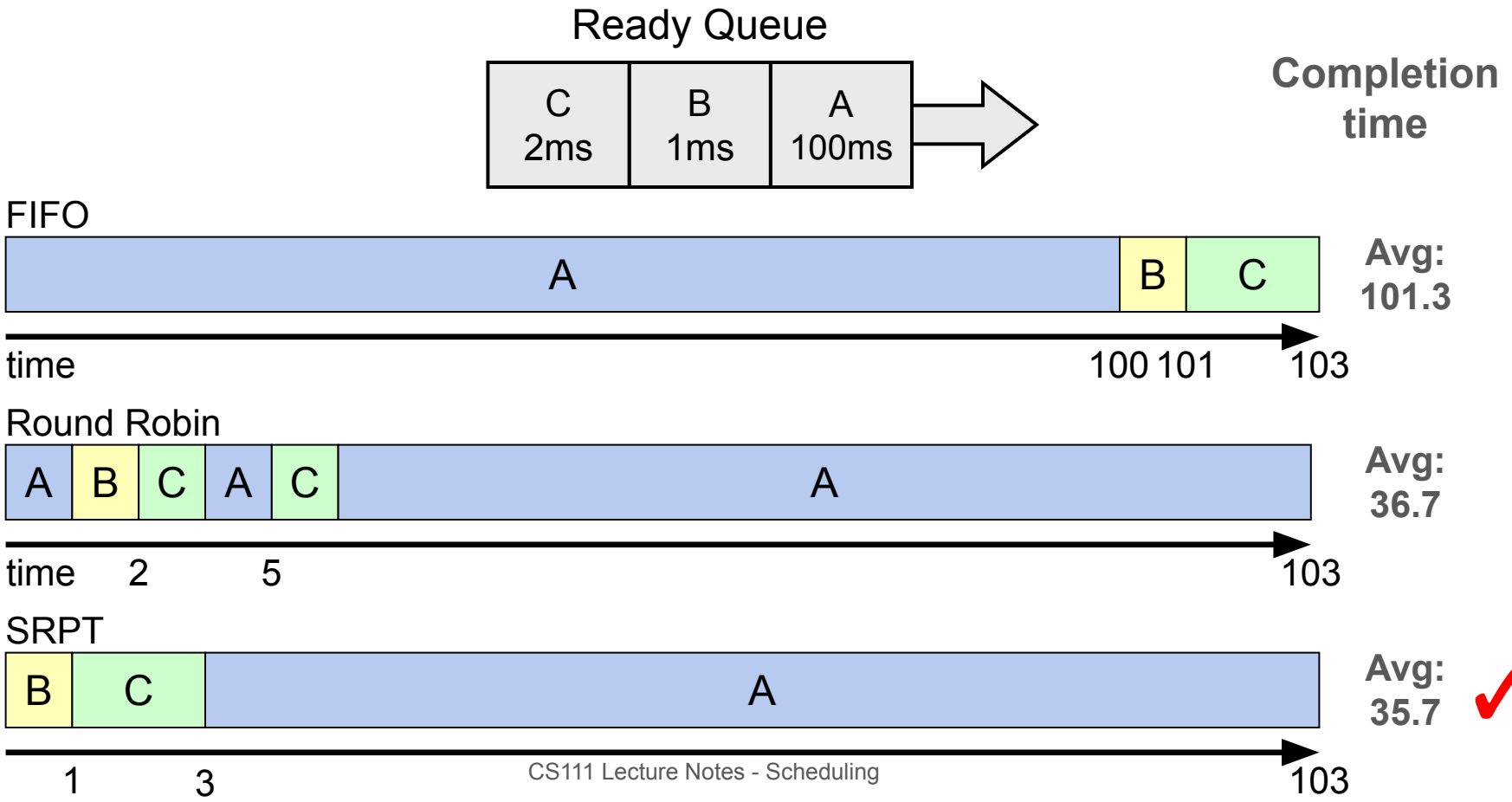
Fairness vs average response time

- Is it better to be fair than lower average response time?
  - Interesting societal question
- What would reducing time slice help the situtation?
- Can we go for optimal response time?

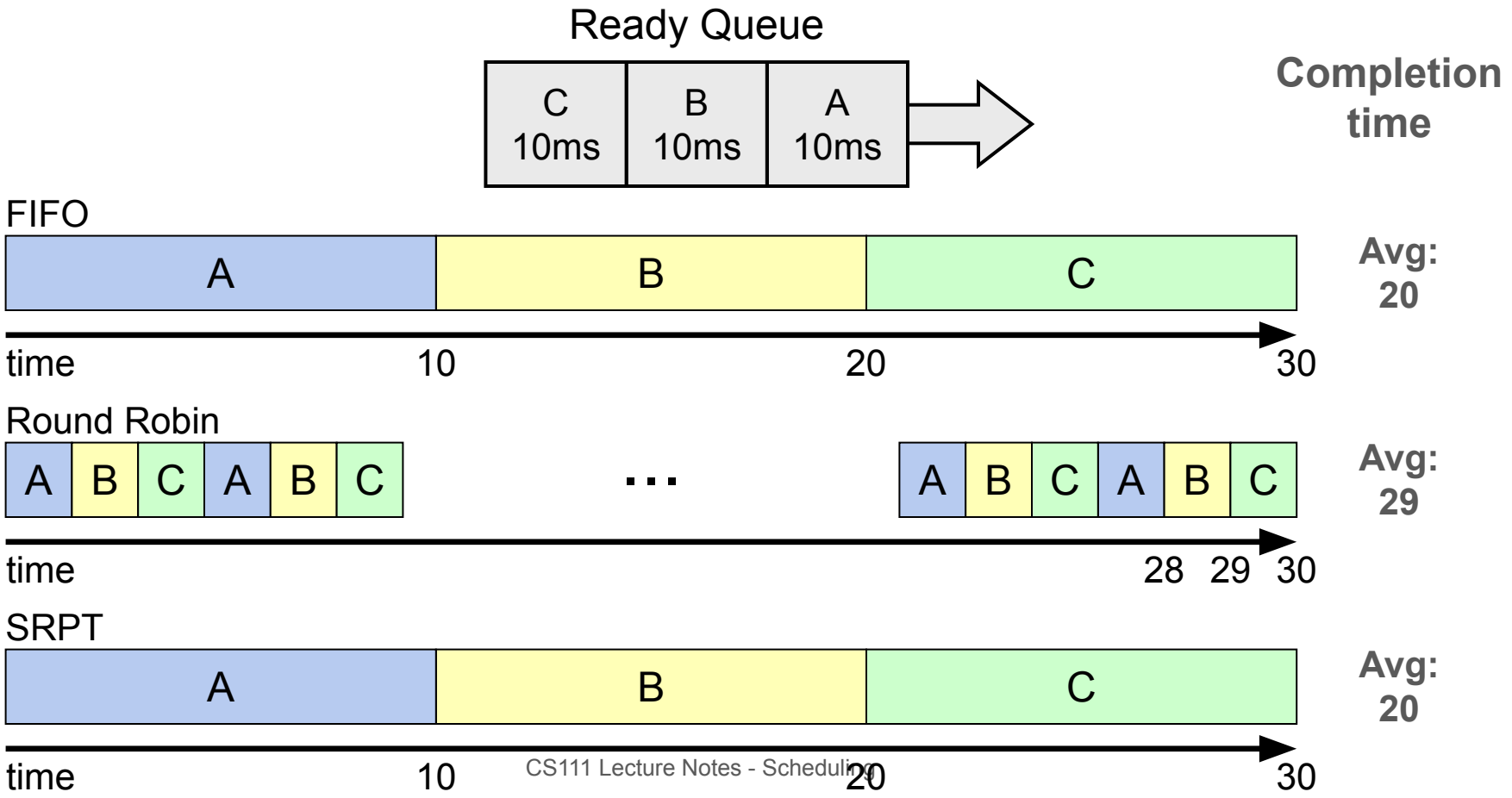
SRPT (Shortest Remaining Processing Time)

- Provably optimal response time scheduling algorithm
- Approach:
  - Run the thread that will finish most quickly
  - Run without interruptions ("run to completion")

Scenario #1: FIFO, Round Robin, SRPT



Scenario #2: FIFO, Round Robin, SRPT



SRPT advantages and disadvantages

- Advantages:
  - Best or equal response time
  - Improves overall resource utilization
    - Consider:
      - Thread #A: CPU-bound
      - Thread #B: I/O-bound.
    - SRPT will give priority to I/O-bound jobs
      - Keeps the disks/network as busy as possible.
  - Give preference to those who need the least
- Disadvantages:
  - Impossible to implement: Require knowing the future
  - Starvation possible

SRPT does the right thing much of the time

Consider three jobs:

- Copy of a big file
  - Loop: Disk read (5ms), short CPU burst (1 ms), disk write (5 ms)
- Interactive editor
  - Loop: Read character (100 ms), short CPU burst (.1 ms)
- Number crunching job
  - Loop: long CPU burst (hours)

Can we appropriate SRPT?

- Key idea: use past performance to predict future
- Past thread behavior tends to predict future behavior
  - Mostly works.
- Thread behavior tends to be consistent
  - A thead has been executing for a long time without blocking, it's likely to continue executing.

Dispatcher and scheduler

- Dispatcher - context switches between threads
- Scheduler - decide which thread should run on a core next
- Incorrect mental model: Dispatcher calls scheduler for next thread.
- Dispatcher and scheduler share a data structure
    - Dispatcher: Fast get next thread to run
    - Scheduler: Update the data structure to control next thread to run

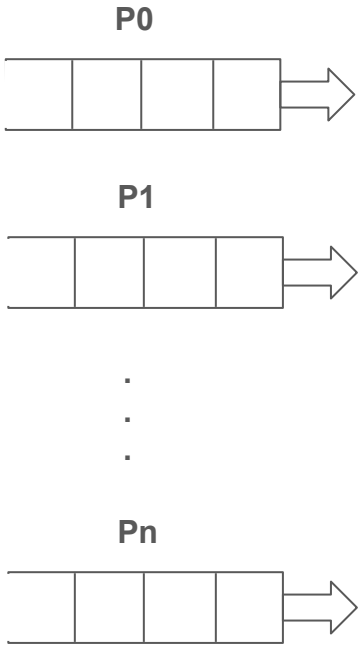
Priority-Based Scheduling

- Priorities: most real schedulers support a priority for each thread
  - Dispatcher always run the thread with highest priority
  - Find thread with highest priority - easy data structure for fast dispatcher
  - In case of priority tie, use round-robin among highest priority threads
- Use priorities to implement various scheduling policies
  - Example: approximate SRPT
  - Can expose priorities to user to allow user control of scheduling

Dispatcher data structure: Priority Queues

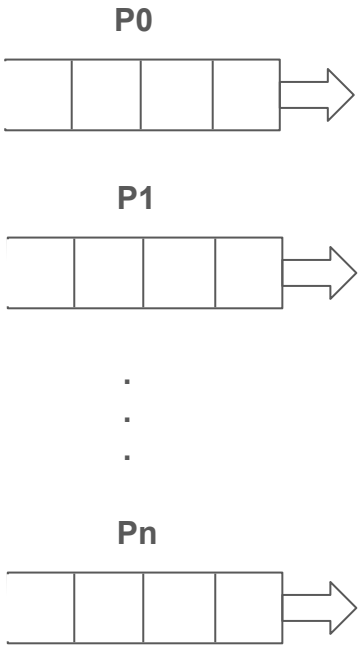
- Multiple queues: One ready queue for each priority level

Ready queues



Priority Queues Overall Idea

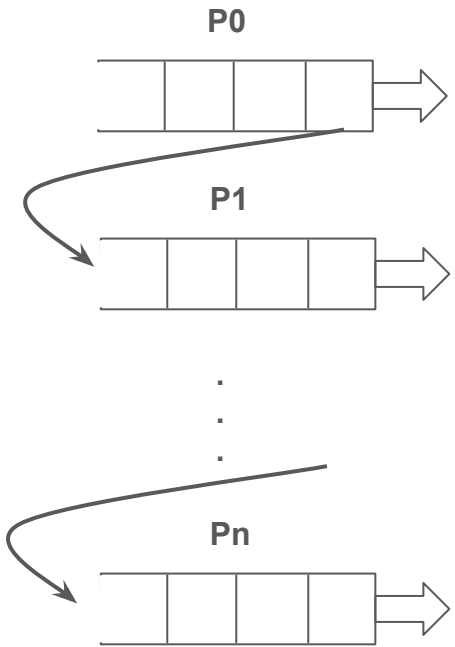
Ready queues



- Threads that aren't using much CPU time stay in the higher-priority queues
  - Interactive (waiting on user)
  - I/O bound (waiting on I/O device)
- Threads that are CPU-bound migrate to lower-priority queues
  - Compute bound

Priority Queues One Possible Approach

Ready queues



- Ready thread starts in highest priority queue
- A thread hits end of its time slice without blocking, move to the next lower queue
- Threads move up when they block

I/O-bound stay in the highest-priority queues

CPU-bound migrate to lower-priority queues

What's the problem?

CPU-bound can starve

Announcements

4.4 BSD Scheduler - Early 1990s Unix

- Keep information about **recent** CPU usage for each thread
  - Dispatcher records start and stop time of threads
- Give highest priority to thread that has used the least CPU time recently
- Interactive and I/O-bound threads will use little CPU time and remain at high priority
- CPU-bound threads will eventually get lower priority as they accumulate CPU time.

Can a CPU-bound thread gets starved by interactive threads and stops getting CPU time?

No - priority goes up as it waits to run - eventually will be highest priority

What happens if the system gets so overloaded that no thread is getting much CPU time?

Devolve to round-robin in the highest priority queue

User input into scheduling: Unix `nice` level

- Users can override the default CPU scheduling behavior

Nice value can emphasize or de-emphasize the CPU usage of a thread

- 0 default
- +19 emphasize CPU usage so gives low priority (most nice)
- -20 de-emphasize CPU usage so gives highest priority (least nice)

```
nice -n 19 ./background_script.sh
```

```
nice -n -20 ./run_with_highest_priority.sh
```

Multicore Scheduling: Simple (initial) approach

- Shared across cores:
  - Scheduling data structures (`ready queues, lock`)
- Private per core:
  - Dispatcher reading shared ready queue
  - Timer interrupts for time slices
- Run the k highest-priority threads on the k cores
- When a thread becomes ready, see if its priority is higher than the lowest-priority thread currently running. If so, preempt that thread
  - Mechanism: IPI (inter-processor interrupt)

Problems/issues for simple multicore approach

- Lock contention on the ready queue
  - System will bottleneck on the central ready queue
  - Solution:
    - Separate ready queue per core
    - Balance queues over time (***work stealing***).
- Thread state accumulates on a core when it runs
  - Thread running on a core builds state making it is expensive to move
    - Example: Memory caches will have to be reloaded
  - **Core affinity**: Try to keep a thread on the same core as much as possible

Desirability property: Work-conserving

- **Work-conserving** CPU scheduler
  - Never leaves a core idle when there is runnable work available
- Hard to implement because of contention and core affinity
  - Overall system could run slower to achieve work conserving
- Good in theory 😀

What is the scheduler and when does it run?

- What: code that runs in response to some events
  - Note: not a thread
- When: certain events
  - Thread unblocks
  - Timer interrupt
  - IPI from another core

Scheduling keeps changing in importance

- During timesharing error, CPU scheduling super important
- With single-user personal computers, not so important
  - Let user decide priorities.
- With multi-core machines, CPU time isn't particularly scarce
  - Scheduling algorithm makes less difference.
- Scheduling interesting for datacenters:
  - Scheduling large applications across hundreds or thousands of servers
  - Balancing CPU-heavy background tasks with latency-critical tasks responding to users
    - Consider cloud tenant web server sharing a machine with ML training job

Conclusion

Scheduling algorithms should not affect the results produced by the system
- Algorithms do impact the system's efficiency and response time

The best schemes are adaptive
- Scheduler algorithm adjust based on the workload
- Stange constants in the code that have a big effect

To be optimal, we'd have to predict the future
- Can use the past to predict the future