Today

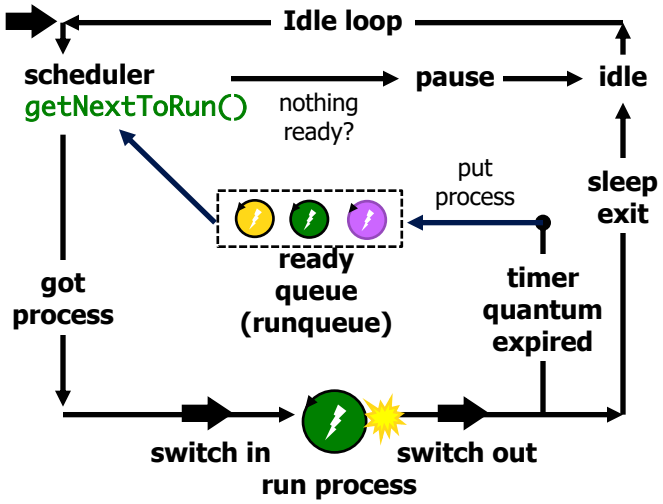
https://pollev.com/sprenkle

- Process Scheduling

Review

- What is the CPU's flow?
- When is a "new" process scheduled to run?
  - What are the mechanics of scheduling a process?
- When determining process scheduling policy, what are some considerations we should keep in mind?
  - Process properties?
  - Metrics?
  - Preemptive vs Non-preemptive?
  - ??
- What are the impacts of these considerations?

Review: CPU's Flow



Review: CPU Scheduling Policy

- In designing the CPU scheduler there are two major policy questions that must be answered:
  - Under what circumstances will the scheduler be invoked?
    - Non-preemptive vs. Preemptive scheduling
  - When the scheduler is invoked, what criterion will it use to select from the ready queue the next process to run?
    - Scheduling Algorithm

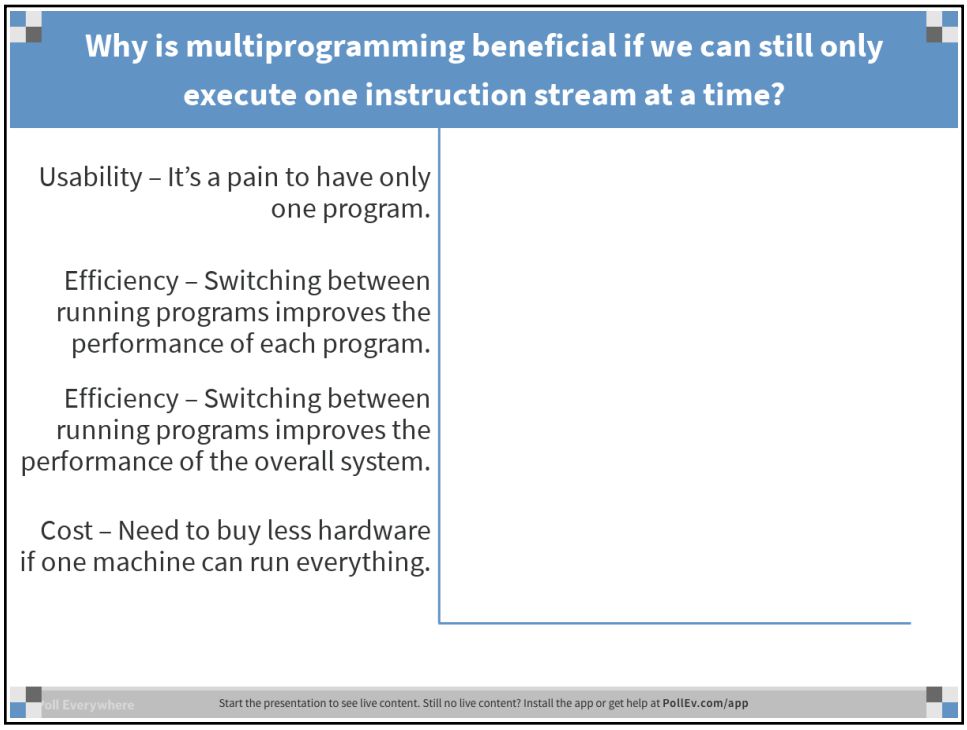
Review: Scheduling Opportunities

There are four opportunities for the CPU scheduler to select a new process to run:

1. The running process blocks (running → waiting)
2. A new process is created (new → ready)
3. The running process is interrupted (running →ready)
   - Or yields
   - A process may also unblock. (waiting →ready)
4. A process exits. (running→terminated)

Scheduling Metrics/Policy Goals

- CPU Utilization
  - percentage of time CPU is being used (not idle)
- Response (or turnaround) time or latency, responsiveness
  - How long does it take to complete a task or request? (R)
    - Typically concerned with average
  - Say a task takes D time units of work (its service demand)
    - But how long does it spend waiting for service?
- Throughput
  - How many tasks/requests complete per unit of time? (X)
- Fairness
  - how well is the CPU distributed among processes
- Meet deadlines, reduce jitter for periodic tasks
  - e.g., videos and other continuous media



Why is multiprogramming beneficial if we can still only execute one instruction stream at a time?

- **Usability – It's a pain to have only one program.**
- Efficiency – Switching between running programs improves the performance of each program.
- **Efficiency – Switching between running programs improves the performance of the overall system.**
- **Cost – Need to buy less hardware if one machine can run everything.**
- Some other reason(s).

Metrics in Practice

Which metrics are most important in each scenario?
What might a reasonable scheduling policy look like for each?

1. Super computer: large, long running jobs submitted by many users
2. Medical device: sensors for monitoring patient and actuators for doing something to them (e.g., administering medication)
3. General-purpose desktop/laptop: variety of tasks happening for one user (browser, email, music, messaging, etc.)

Metrics: CPU utilization, response time, throughput, fairness, others?

Observations

- Super computer probably has lots of CPU hungry tasks, not much I/O.
- Task *priority* probably critical to medical device.
- Humans like interactivity on desktop/laptop, even at the expense of overall runtime.

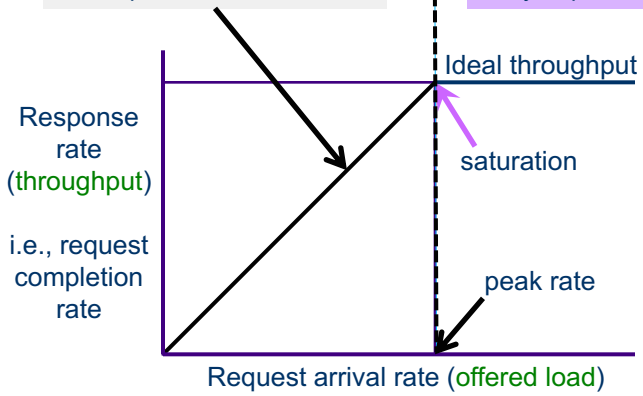
CPU Scheduling: There is no one-size-fits-all "best" policy…

- Depends on the goals of the system.
- Often have multiple (conflicting) goals or primary metrics

Ideal Throughput

throughput == arrival rate
The center is not saturated: it completes requests at the rate requests are submitted.

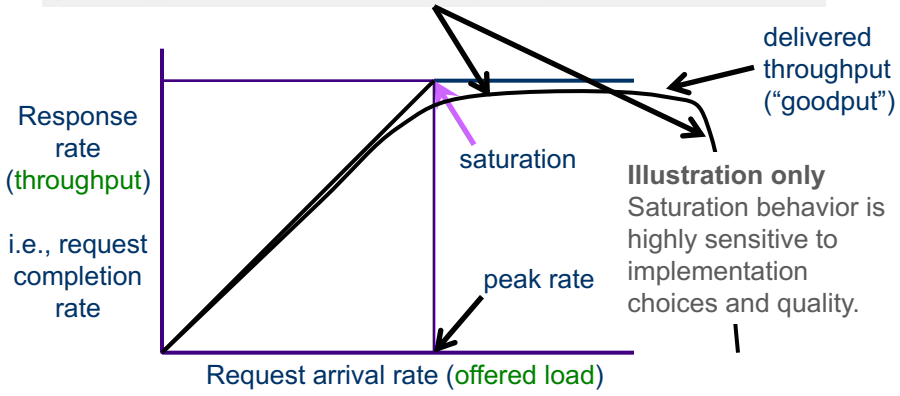
throughput == peak rate
The center is *saturated*.
It can't go any faster, no matter how many requests are submitted.



This graph shows throughput (e.g., of a server) as a function of offered load. It is idealized: your mileage may vary.

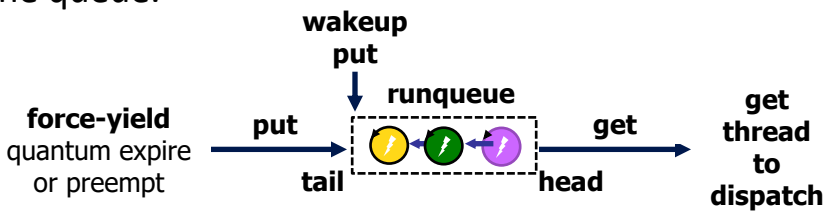
Throughput: reality

Thrashing* or *congestion collapse
Real servers/devices often have some pathological behaviors at saturation. They abort requests after investing work in them (thrashing), which wastes work, reducing throughput.



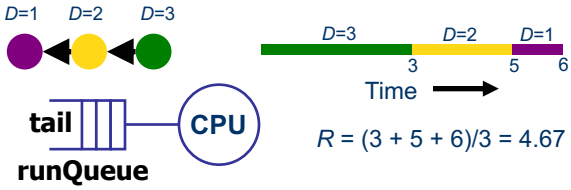
A simple policy: FCFS

- The most basic scheduling policy is first-come-first-served (FCFS), also called first-in-first-out (FIFO).
- FCFS is like the checkout line at the Kwik-e-mart
- Maintain a queue ordered by time of arrival.
- `GetNextToRun` selects from the front (head) of the queue.



Evaluating FCFS

- **Throughput.** FCFS is as good as any non-preemptive policy.
- **Fairness.** FCFS is intuitively fair…sort of.
  - "The early bird gets the worm"…and everyone is fed…eventually.
- **Response time.** Long jobs keep everyone else waiting.
  - Consider service demand (D) for a process/job/thread.

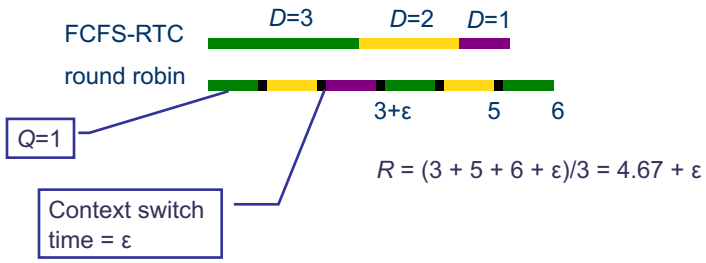


Non-Preemptive vs Preemptive

- Depending upon which scheduling opportunities are used by a scheduler, the scheduling can be:
  - **Non-Preemptive**: The scheduler will allow the running process to continue to run as long as it remains ready (i.e., doesn't block or exit).
  - **Preemptive**: The scheduler may set aside the running process in favor of another at any scheduling opportunity
    - Enables time-sharing, priority scheduling

Preemptive FCFS: Round Robin

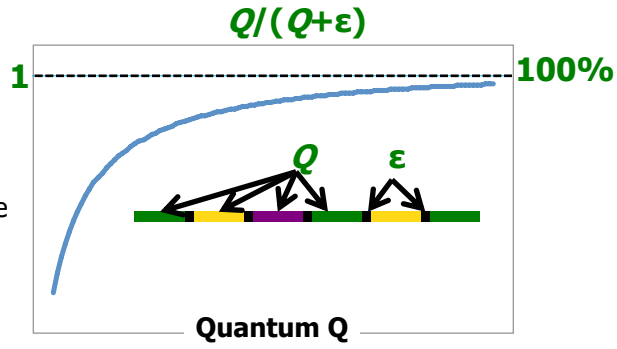
- Preemptive timeslicing is one way to improve fairness of FCFS
- If job does not block or exit, force an involuntary context switch after each quantum Q of CPU time
- FCFS without preemptive timeslicing is "run to completion" (RTC)
- FCFS with preemptive timeslicing is called *round robin*



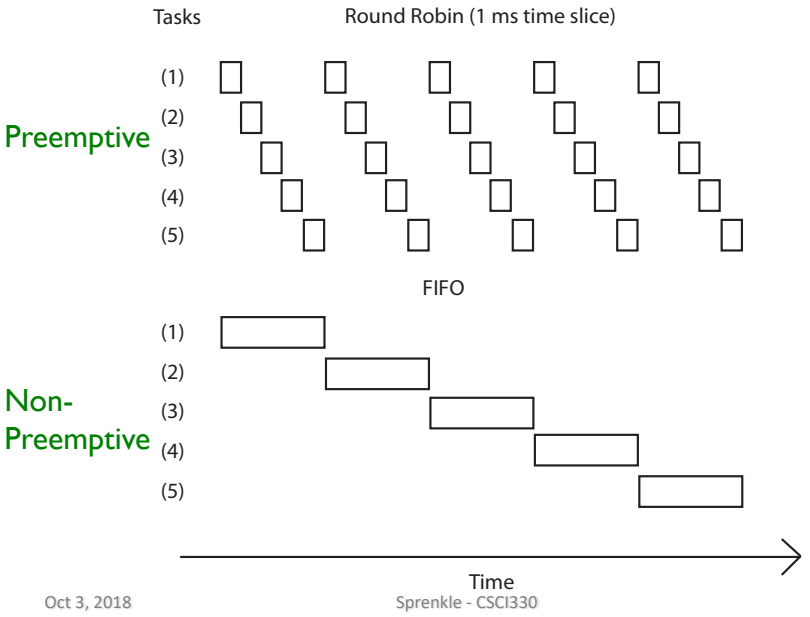
Overhead and Goodput

- **Context switching is overhead: wasted effort**
  - It is a cost that the system imposes to get the work done. It is not actually doing the work.

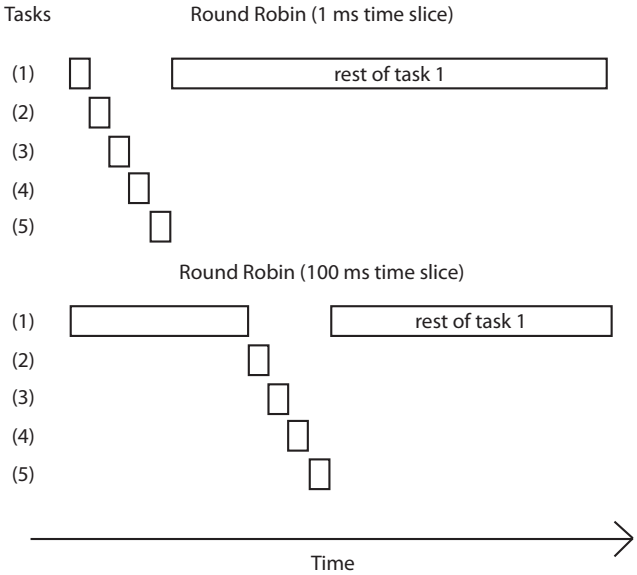
Efficiency or goodput
What percentage of the time is the busy resource doing useful work?



Round Robin vs. FIFO

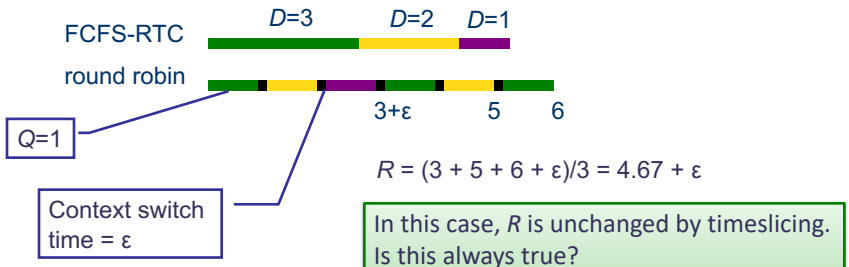


Round Robin: Slice Length



Preemptive FCFS: Round Robin

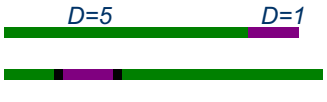
- Preemptive timeslicing is one way to improve fairness of FCFS
- If job does not block or exit, force an involuntary context switch after each quantum Q of CPU time
- FCFS without preemptive timeslicing is "run to completion" (RTC)
- FCFS with preemptive timeslicing is called *round robin*



$R = (3 + 5 + 6 + \varepsilon)/3 = 4.67 + \varepsilon$

In this case, *R* is unchanged by timeslicing.
Is this always true?
How else can you analyze RR?

Evaluating Round Robin



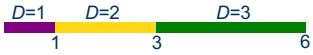
$R = (5+6)/2 = 5.5$

$R = (2+6 + \varepsilon)/2 = 4 + \varepsilon$

- **Response time.** RR reduces response time for short jobs.
  - For a given load, wait time is proportional to the job's total service demand D.
- **Fairness.** RR reduces variance in wait times.
  - But: RR makes jobs wait for jobs that arrived later.
- **Throughput.** RR imposes extra context switch overhead.
  - Degrades to FCFS-RTC with large Q.

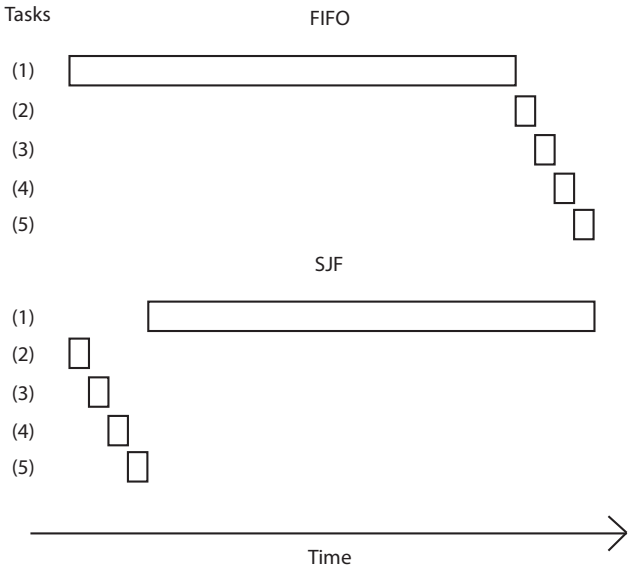
Minimizing Response Time: SJF (STCF)

- **Shortest Job First** (SJF) is provably optimal if the goal is to minimize average-case R.
  - Also called Shortest Time to Completion First (STCF) or Shortest Remaining Processing Time (SRPT)
- Idea: get short jobs out of the way quickly to minimize the number of jobs waiting while a long job runs.
  - Intuition: longest jobs do the least possible damage to the wait times of their competitors.



$R = (1 + 3 + 6)/3 = 3.33$

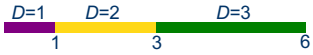
FIFO vs. SJF



Minimizing Response Time: SJF (STCF)

- **Shortest Job First** (SJF) is provably optimal if the goal is to minimize average-case R.
  - Also called Shortest Time to Completion First (STCF) or Shortest Remaining Processing Time (SRPT)
- Idea: get short jobs out of the way quickly to minimize the number of jobs waiting while a long job runs.
  - Intuition: longest jobs do the least possible damage to the wait times of their competitors.

Any limitations?



$R = (1 + 3 + 6)/3 = 3.33$

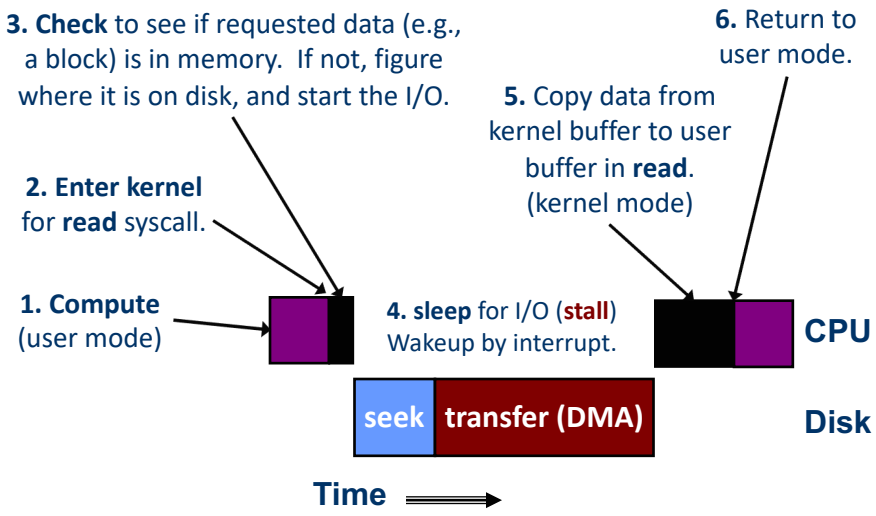
Could starve long-running processes

26

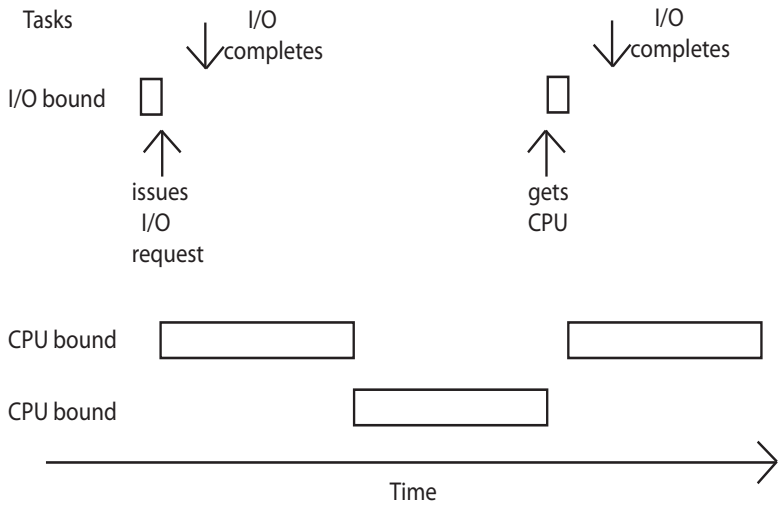
The Process Mix

- Two broad classes of processes:
  - **CPU Bound**: A process that is spending most of its time doing CPU operations.
  - **I/O Bound**: A process that is spending most of its time doing I/O operations.
- Processes can switch between being CPU Bound and being I/O Bound during their execution

Anatomy of a read



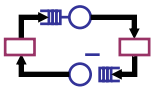
Mixed Workload



Two Schedules for CPU/Disk

1. Naive **Round Robin**

CPU busy 25/37: U = 67%
Disk busy 15/37: U = 40%



2. Add internal priority boost for I/O completion

CPU busy 25/25: U = 100%
Disk busy 15/25: U = 60%

33% improvement in utilization
When there is work to do,
U == efficiency.
More U means better throughput.

Looking Ahead

- Interprocess Communication
- Project 2 released
  - System calls
  - Much more C programming, actually using pointers
  - Due in two Mondays