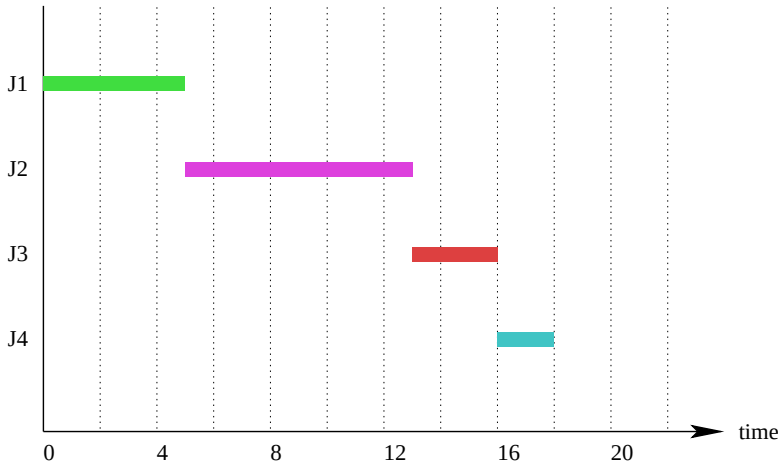
Job Scheduling Model

- problem scenario: a set of *jobs* needs to be executed using a single server, on which only one job at a time may run
- for the ith job, we have an arrival time a_i and a run time r_i
- after the ith job has run on the server for total time r_i, it finishes and leaves the system
- a job *scheduler* decides which job should be running on the server at each point in time
- let s_i ($s_i \geq a_i$) represent the time at which the ith job first runs, and let f_i represent the time at which the ith job finishes
  - the *turnaround time* of the ith job is $f_i - a_i$
  - the *response time* of the ith job is $s_i - a_i$

Basic Non-Preemptive Schedulers: FCFS and SJF

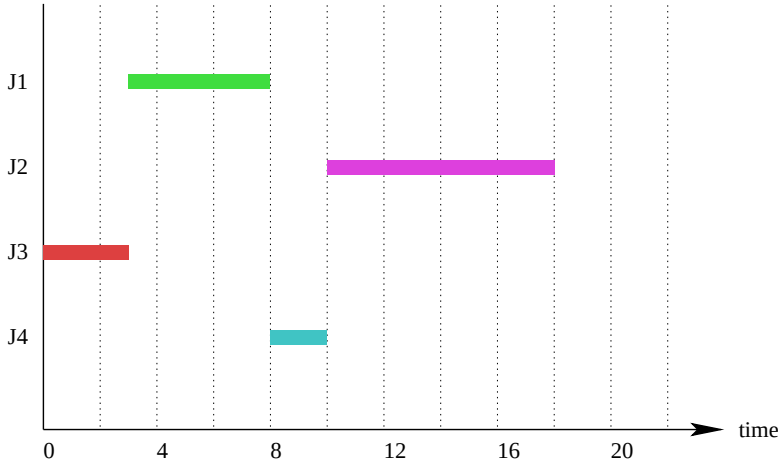
- FCFS: runs jobs in arrival time order.
  - simple, avoids starvation
  - pre-emptive variant: round-robin
- SJF: shortest job first - run jobs in increasing order of r_i
  - minimizes average *turnaround* time
  - long jobs may starve
  - pre-emptive variant: SRTF (shortest remaining time first)

FCFS Gantt Chart Example



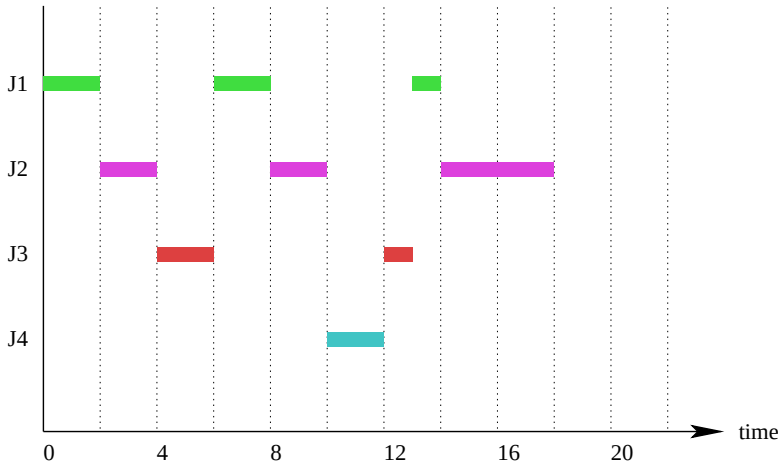
Job	J1	J2	J3	J4
arrival (a_i)	0	0	0	5
run time (r_i)	5	8	3	2

SJF Example



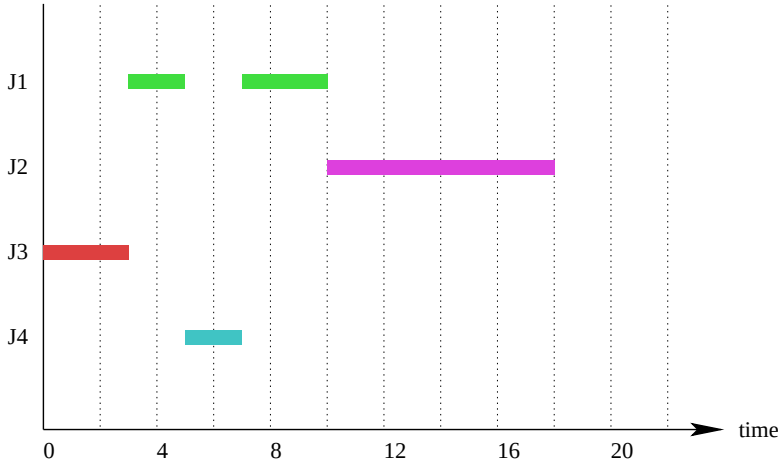
Job	J1	J2	J3	J4
arrival (a_i)	0	0	0	5
run time (r_i)	5	8	3	2

Round Robin Example



Job	J1	J2	J3	J4
arrival (a_i)	0	0	0	5
run time (r_i)	5	8	3	2

SRTF Example



Job	J1	J2	J3	J4
arrival (a_i)	0	0	0	5
run time (r_i)	5	8	3	2

CPU Scheduling

- CPU scheduling is job scheduling where:
  - the server is a CPU (or a single core of a multi-core CPU)
  - the jobs are *ready threads*
    - a thread "arrives" when it becomes ready, i.e., when it is first created, or when it wakes up from sleep
    - the run-time of the thread is the amount of time that it will run before it either finishes or blocks
  - thread run times are typically *not known* in advance by the scheduler
- typical scheduler objectives
  - responsiveness - low *response time* for some or all threads
  - "fair" sharing of the CPU
  - efficiency - there is a cost to switching

Prioritization

- CPU schedulers are often expected to consider process or thread priorities
- priorities may be
  - specified by the application (e.g., Linux `setpriority/sched_setscheduler`)
  - chosen by the scheduler
  - some combination of these
- two approaches to scheduling with priorites
  1. schedule the highest priority thread
  2. weighted fair sharing
     - let p_i be the priority of the ith thread
     - try to give each thread a "share" of the CPU in proportion to its priority:

$$\frac{p_i}{\sum_j p_j} \tag{1}$$

Multi-level Feedback Queues

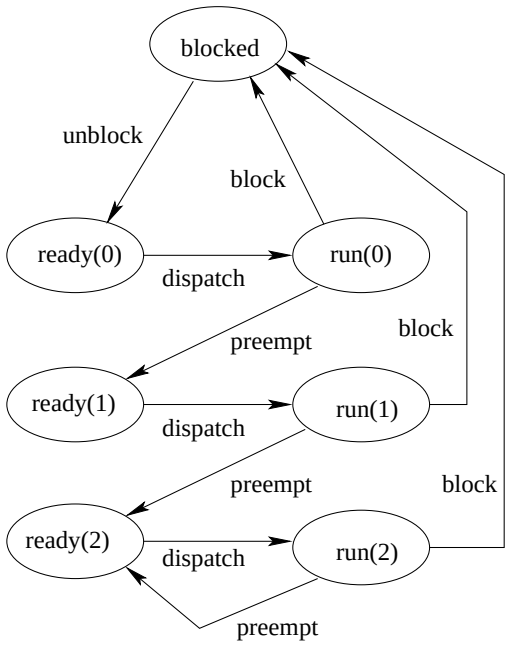
- objective: good responsiveness for *interactive* processes
  - threads of interactive processes block frequently, have short run times
- idea: gradually diminish priority of threads with long run times and infrequent blocking
  - if a thread blocks before its quantum is used up, *raise* its priority
  - if a thread uses its entire quantum, *lower* its priority

Multi-level Feedback Queues (Algorithm)

- scheduler maintains several round-robin ready queues
  - highest priority threads in queue Q_0, lower priority in Q_1, still lower in Q_2, and so on.
- scheduler always chooses thread from the lowest non-empty queue
- threads in queue Q_i use quantum q_i, and $q_i \leq q_j$ if $i < j$
- newly ready threads go into ready queue Q_0
- a level i thread that is preempted goes into queue Q_{i+1}

This basic algorithm may *starve* threads in lower queues. Various enhancements can avoid this, e.g, periodically migrate all threads into Q_0.

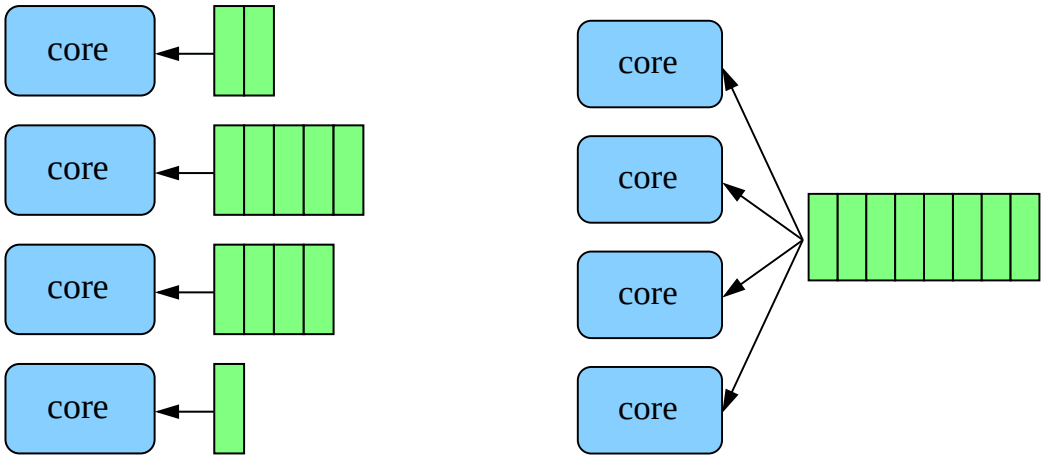
3 Level Feedback Queue State Diagram



Linux Completely Fair Scheduler (CFS) - Key Ideas

- "Completely Fair Scheduling" - a weighted fair sharing approach
- suppose that c_i is the actual amount of time that the scheduler has allowed the ith thread to run.
- on an *ideally shared* processor, we would expect $c_0 \frac{\sum_j p_j}{p_0} = c_1 \frac{\sum_j p_j}{p_1} = \cdots$
- CFS calls $c_i \frac{\sum_j p_j}{p_i}$ the *virtual runtime* of the ith thread, and tracks it for each thread
- CFS chooses the thread with the lowest virtual runtime, and runs it until some other thread's virtual runtime is lower (subject to a minimum runtime quantum)
  - virtual runtime advances more slowly for higher priority threads, so they get longer time slices
  - all ready threads run regularly, so good responsiveness

Scheduling on Multi-Core Processors



per core ready queue(s) vs. shared ready queue(s)

Scalability and Cache Affinity

- Contention and Scalability
  - access to shared ready queue is a critical section, mutual exclusion needed
  - as number of cores grows, contention for ready queue becomes a problem
  - per core design *scales* to a larger number of cores
- CPU cache affinity
  - as thread runs, data it accesses is loaded into CPU cache(s)
  - moving the thread to another core means data must be reloaded into that core's caches
  - as thread runs, it acquires an *affinity* for one core because of the cached data
  - per core design benefits from affinity by keeping threads on the same core
  - shared queue design does not

Load Balancing

- in per-core design, queues may have different lengths
- this results in *load imbalance* across the cores
  - cores may be idle while others are busy
  - threads on lightly loaded cores get more CPU time than threads on heavily loaded cores
- not an issue in shared queue design
- per-core designs typically need some mechanism for *thread migration* to address load imbalances
  - migration means moving threads from heavily loaded cores to lightly loaded cores