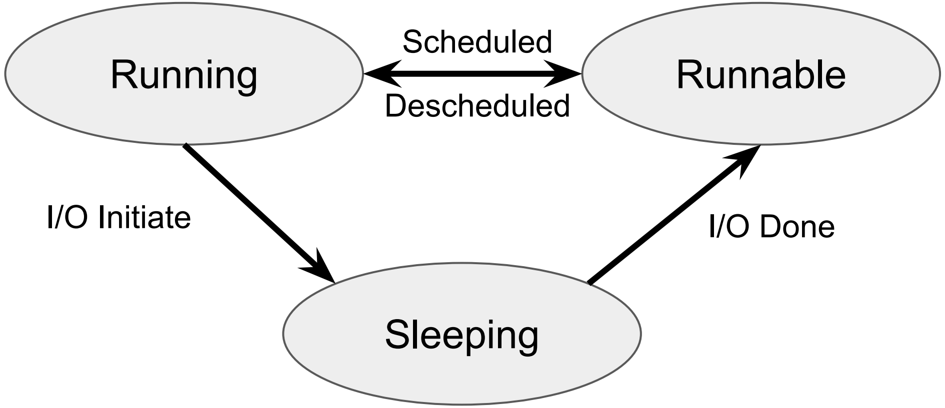
Scheduling Fair

Fair schedulers are “fair” to processes, running them proportionally to an assigned priority.

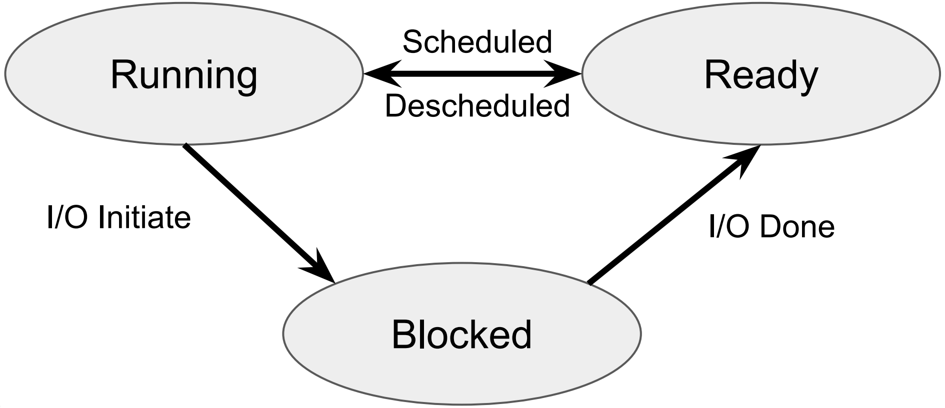
- User sets priority - Linux nice
- OS sets priority based on history of execution - MLFQ

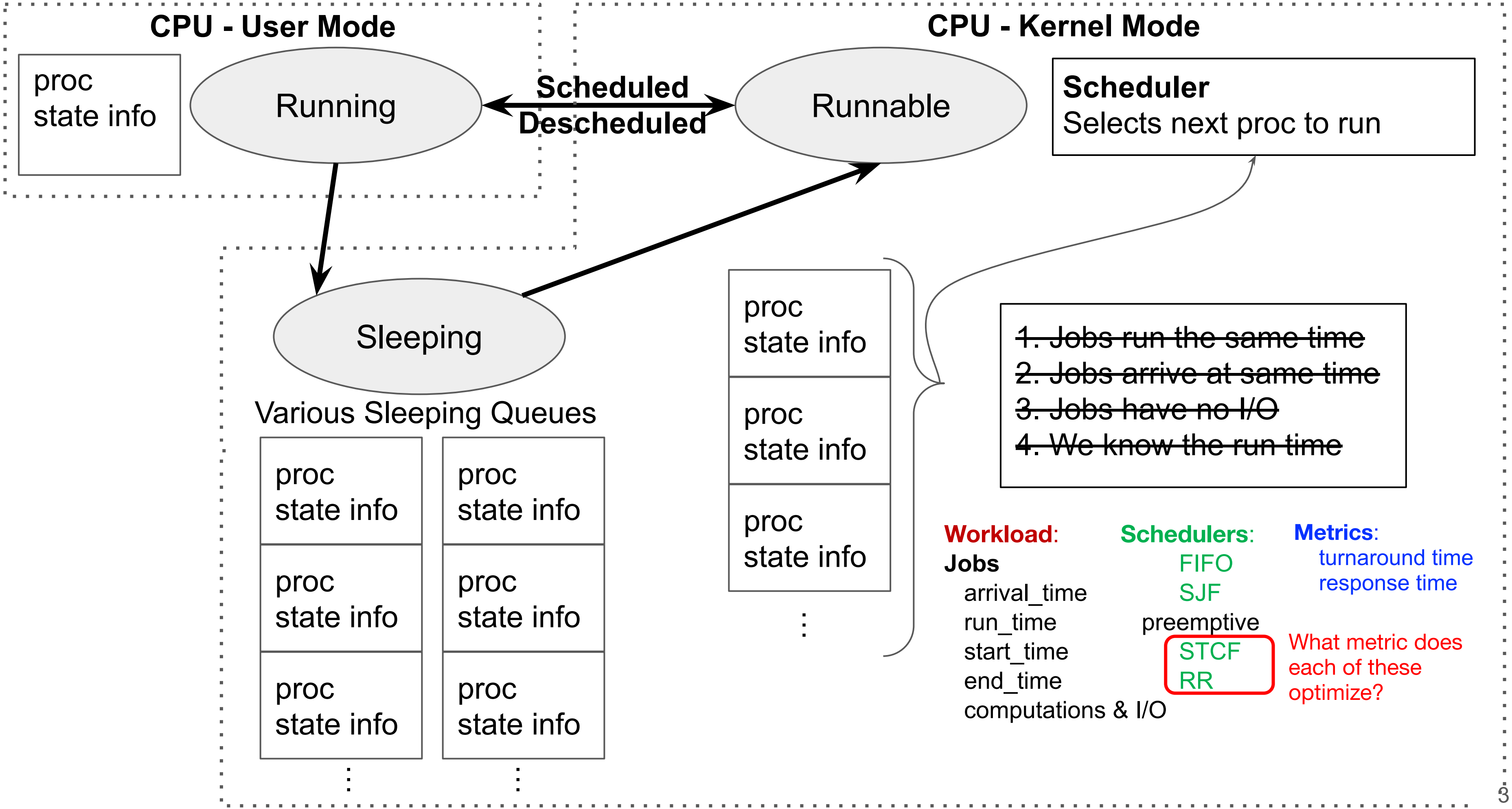
Process State Transitions

Xv6

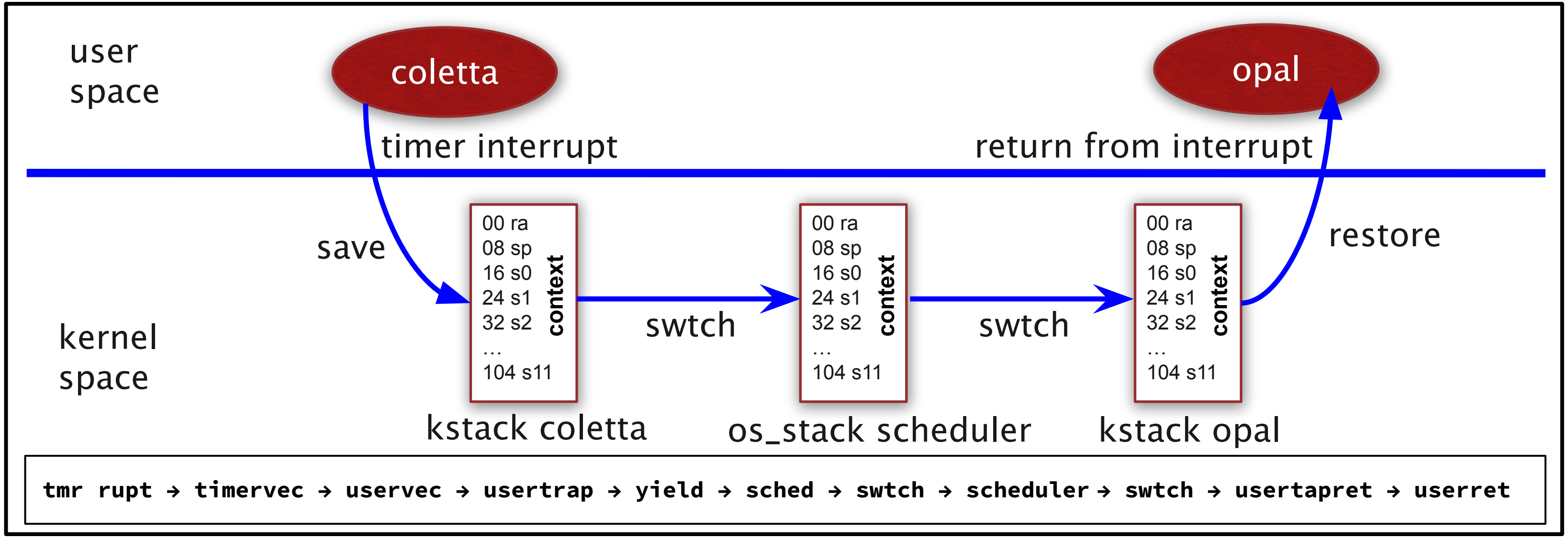


OSTEP




CPU - User Mode
proc
state info
Running
Scheduled
Descheduled
CPU - Kernel Mode
Runnable
Scheduler
Selects next proc to run
Sleeping
Various Sleeping Queues
proc
state info
proc
state info
proc
state info
proc
state info
proc
state info
proc
state info
proc
state info
proc
state info
proc
state info
~~1. Jobs run the same time~~
~~2. Jobs arrive at same time~~
~~3. Jobs have no I/O~~
~~4. We know the run time~~
Workload:
Jobs
arrival_time
run_time
start_time
end_time
computations & I/O
Schedulers:
FIFO
SJF
preemptive
STCF
RR
Metrics:
turnaround time
response time
What metric does each of these optimize?

Context Switch



MLFQ - Multilevel Feedback Queue

- A general-purpose scheduler that supports workloads with different goals.
  - **interactive** programs care about **response time**
  - **batch** programs care about **turnaround time**
- Create multiple priority levels of round-robin. The higher priority levels are serviced first.
- Multics is an OS from 1960's created by MIT, GE, and Bell Labs.
- Multics used a variation of the MLFQ.
- Multics is a precursor to Unix is a precursor to Linux.
- Link to the Multics scheduler and dispatcher - https://multicians.org/pxss.html
- Most good ideas are easy to understand conceptually

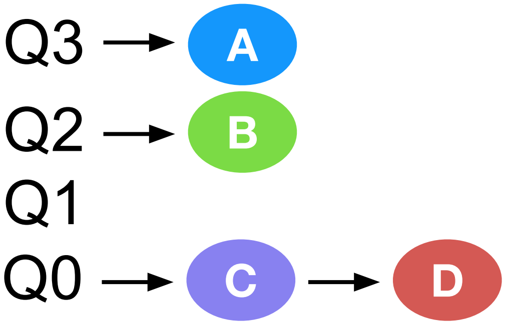
MLFQ

- Place procs on multiple queues
- Each queue has a priority
- Rule 1: If priority(A) > Priority(B), A runs
- Rule 2: If priority(A) == Priority(B) A & B run in RR
- Rule 3: Processes start at top priority
- Rule 4: If process uses the entire time slice, demote priority

  Interactive stay high.

  Compute get demoted.

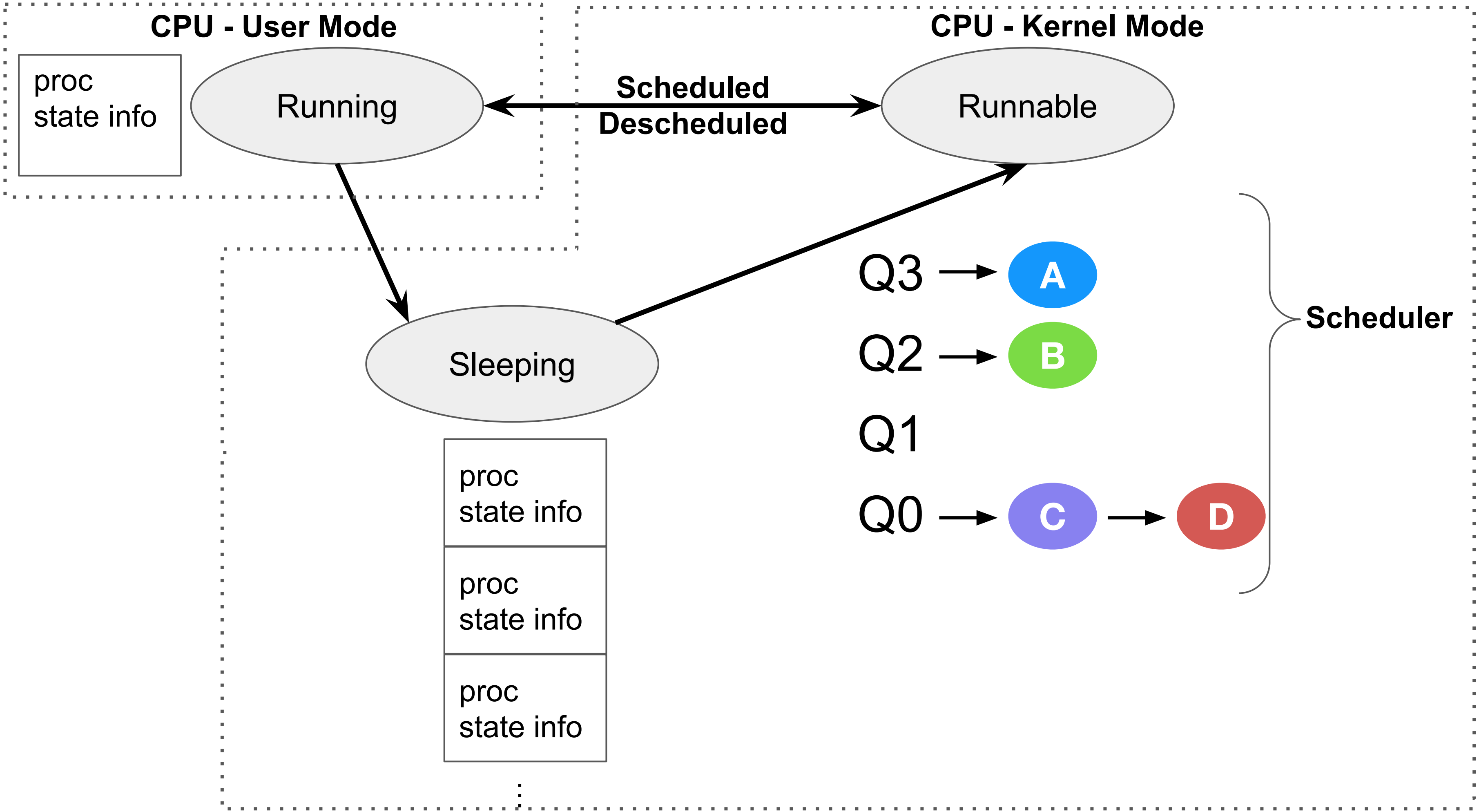
~~1. Jobs run the same time~~
~~2. Jobs arrive at same time~~
~~3. Jobs have no I/O~~
~~4. We know the run time~~



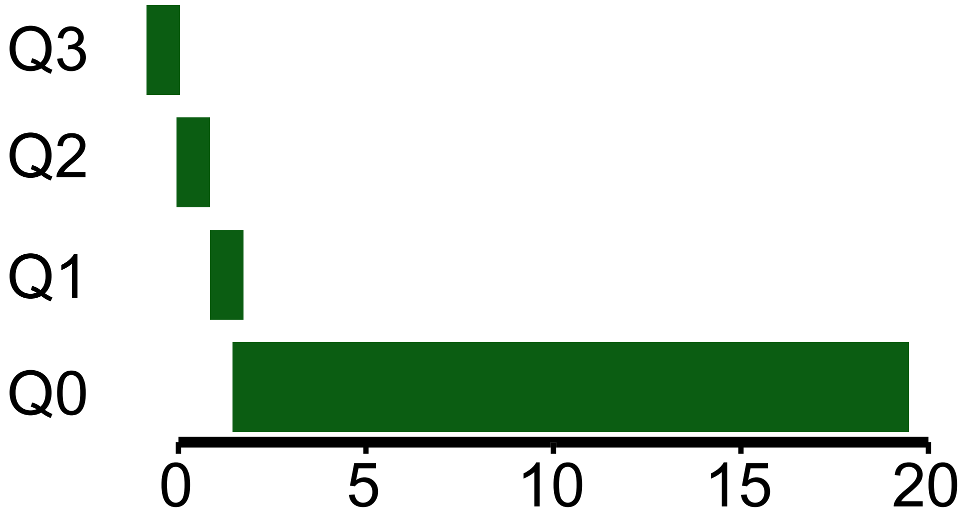
To set priority of procs.

- User sets priority - Linux `nice`
- OS sets priority based on history of execution → used by MLFQ

Number of queues depends on implem.

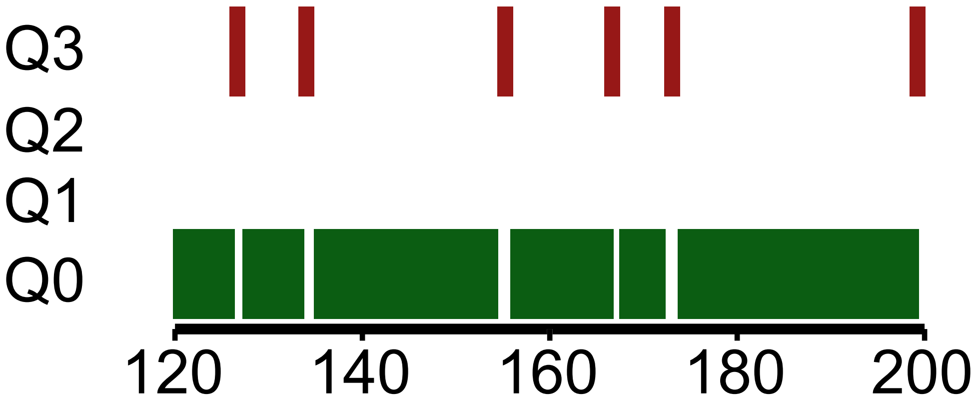

CPU - User Mode
proc
state info
Running
Scheduled
Descheduled
CPU - Kernel Mode
Runnable
Sleeping
proc
state info
proc
state info
proc
state info
Q3
A
Q2
B
Q1
Q0
C
D
Scheduler

MLFQ - One Long Job (Example)



- Job begins at highest Q and is demoted to the lowest, where it remains

MLFQ - Interactive and Compute



- Interactive process never uses entire time slice, so never demoted
- Starvation - lower priority queues never get to run
  - Periodically boost priority of jobs (or jobs that haven't been scheduled)
- Gaming the system - someone issues a wait just prior to consuming their quantum
  - Count for job's total run time at priority level (instead of just time slice); downgrade when exceed threshold

Fair Share Scheduling

- Fair share or proportional share scheduling
  - Guarantee each job obtains a certain percentage of CPU time
  - Not trying to optimize for turnaround or response time
- Lottery Scheduler
- Stride Scheduler
- Linux Completely Fair Scheduler

To set priority of procs.

- User sets priority - Linux `nice` → used by Lottery, Stride, and LCFS
- OS sets priority based on history of execution → used by MLFQ

Lottery Scheduling

- Goal: proportional (fair) share
- Approach:
  - Give processes lottery tickets
  - Higher priority processes are given more tickets
  - Random drawing - whoever wins runs
- Amazingly simple to implement

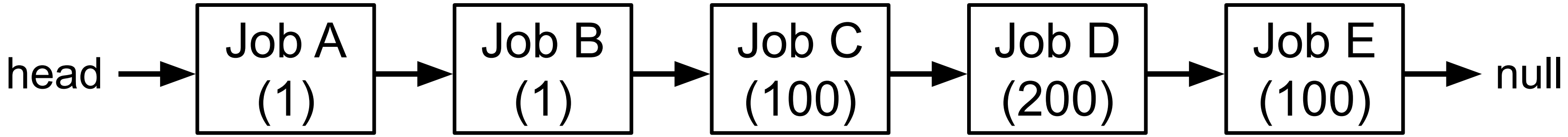
Lottery Scheduler

```
int counter = 0;
int winner = getrandom(0,totaltickets);
struct proc *current = kernel_proc;
while(current) {
      counter += current->tickets;
      if (counter > winner) break;
      current = current->next;
}
// current gets to run
```

Who runs if **winner** is:

50
350
0

402 Tickets



Stride Scheduler

A big number

- Allocate tickets to processes
- **Stride value** for each process uses number of tickets
  - `stride_value = 10000 / num_tickets`
- **Pass value** is a counter for each process. Each time a process runs, increment `pass_value` by `stride_value`
  - `pass_value += stride_value`
- Stride scheduler runs the process with the smallest pass value
  - `current = remove_min_pass_value(queue);`
  - `dispatch(current)`
  - `current->pass_value += current->stride_value`
  - `enqueue(queue, current)`

Stride Scheduler Example

Pass(A) (stride=100) (tics=100)	Pass(B) (stride=200) (tics=50)	Pass(C) (stride=40) (tics=250)	Who Runs?
0	0	0	A
100	0	0	B
100	200	0	C
100	200	40	C
100	200	80	C
100	200	120	A
200	200	120	C
200	200	160	C
200	200	200	A
300	200	200	B
300	400	200	C
300	400	240	C
300	400	280	C

Big Number Tickets Stride

```
10000 / 100 is 100
10000 / 50  is 200
10000 / 250 is 40
```

Stride Scheduler Example

- Lottery achieves fairness this over time.
    - Lottery easily handles new processes entering the mix
- Stride achieves fairness on each scheduling cycle.
    - Stride must handle new processes entering the mix
    - Big number - 10,000
    - Three processes with tickets and stride
        - Tickets: PA 100, PB 50, PC 250
        - Stride: PA 100, PB 200, PC 40
    - ProcD enters with 500 tickets

Linux Completely Fair Scheduling

- `vruntime` - incremented as processes run
  - Select the process with lowest vruntime
  - First - you can think of vruntime as runtime
- `niceness` - weighting priorities
  - Used to compute `time_slice` and `vruntime`
  - Procs with smaller `nice` (higher priority) get a larger time slice.
- Time slice - smaller time slice increases near term fairness
  - Balance fairness, CPU throughput, and context switch overhead
  - `schedule_latency` - a value that is divided by number of processes to compute time slice. E.g., 48ms
    - 4 running processes, each gets a 12ms time slice
    - 100 running processes, each gets a 480us time slice - too small
  - `min_granularity` - smallest time slice allowed, e.g., 6ms

Linux CFS

```
// The weight is roughly equivalent to 1024/(1.25)**(nice)
static const int prior_to_weight[40] = { // negative - higher prior
 /* -20 */     88761,     71755,     56483,     46273,     36291,
 /* -15 */     29154,     23254,     18705,     14949,     11916,
 /* -10 */      9548,      7620,      6100,      4904,      3906,
 /*  -5 */      3121,      2501,      1991,      1586,      1277,
 /*   0 */      1024,       820,       655,       526,       423,
 /*   5 */       335,       272,       215,       172,       137,
 /*  10 */       110,        87,        70,        56,        45,
 /*  15 */        36,        29,        23,        18,        15,
};
int weight = 0;
for procs ready to run
   weight += weight_of_proc
time_slice = weight_of_proc_to_run / weight * sched_latency;
time_slice = time_slice < min_gran ? min_gran : time_slice;

vruntime = vruntime + weight[0]/weight_of_proc * runtime
```

Example Time Slice, vruntime

- Two runnable processes
    - Proc A - nice of -5, which is weight of 3121
    - Proc B - nice of 0, which is weight of 1024

- Time slices for 100ms schedule latency
    - weight = 3121 + 1024, which is 4145
    - ProcA time slice 3121/4145 * 100ms = 75.3ms
    - ProcB time slice 1024/4145 * 100ms = 24.7ms

- Virtual Runtime - increases equally
    - ProcA vruntime = vruntime + 1024/3121*runtime
        - 0 + 24.7
    - ProcB vruntime = vruntime + 1024/1024*runtime
        - 0 + 24.7

```
int weight = 0;
for all runnable procs
   weight += weight_of_proc
time_slice = weight_of_proc_to_run /
weight*sched_latency;
time_slice = time_slice<min_gran ? min_gran :
time_slice;

vruntime = vruntime + weight[0]/weight_of_proc * runtime
```

- Select Proc with lowest virtual runtime
- RR where ProcA gets 75ms timeslices and ProcB gets 25ms timeslices.

Lottery and LCFS Schedulers - Both Fair

- Lottery achieves fairness by giving higher priority processes more time slices
- LCFS achieve fairness by giving higher priority processes longer time slices

Xv6 Scheduler

- Xv6 has a round-robin scheduler
- You will implement a priority scheduler
  - Processes assigned priorities (e.g., 10, 22, 55)
  - Highest runnable priority runs
    - If (procA : priority 55) and (procB : priority 10) are runnable, procA runs.