## Fair Share Scheduling

- MLFQ aims to optimize performance (response time, turnaround) by prioritizing short I/O bound processes over long CPU bound ones
- an alternative is *fair-share* (or *proportional share*)
  - each job is guaranteed a certain percentage of CPU time
- three strategies
  - lottery scheduling
  - stride scheduling
  - weighted fair queuing
- a real scheduler
  - Completely Fair Scheduler (CFS)



## Lottery Scheduling

- idea
  - each process holds a number of tickets representing the share of CPU the process should receive
  - scheduler periodically picks a ticket, runs the owning process



## Ticket Mechanisms

Mechanisms are the underlying tools than enable the implementation of policies.

- multiple *ticket currencies* allow separation of user and system ticket allocation policies
  - users can decide how to split tickets amongst their processes independent of the number of tickets the system allocates them
  - user tickets are converted to global currency for the lottery
    - process 1's global tickets = ( # tickets user A allocates to process 1 ) / ( total # tickets in A's currency ) * ( # tickets A is allocated by system )
- *ticket transfer* allows processes to loan tickets to other processes
  - e.g. client can loan tickets to server to boost its likelihood of be scheduled while doing work on behalf of the client
- *ticket inflation* allows processes to temporarily boost their own ticket count
  - appropriate in an environment where processes are trusted



## Implementation

- system maintains –
  - a list of processes with their ticket counts
    - for efficiency, store in order of decreasing ticket count
  - the total number of tickets


- pick a ticket number
- go through the list of processes, counting up the number of tickets held so far
  - the holder of the winning ticket is the process whose tickets first make the running total exceed the winning ticket number

```
// counter: used to track if we've found the winner yet
int counter = 0;

// winner: call some random number generator to
//         get a value >= 0 and <= (totaltickets - 1)
int winner = getrandom(0, totaltickets);

// current: use this to walk through the list of jobs
node_t *current = head;
while (current) {
    counter = counter + current->tickets;
    if (counter > winner)
        break; // found the winner
    current = current->next;
}
// 'current' is the winner: schedule it...
```

# Lottery Scheduling

- advantages
  - fast and simple to implement
- drawbacks
  - does not guarantee actual allocations will match desired allocations – only probabilistic correctness
    - can have greater differences in the short term
- challenges
  - how to allocate tickets?

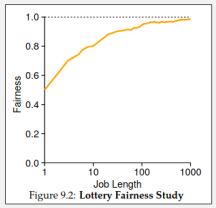

Figure 9.2: **Lottery Fairness Study**

comparison of two equal jobs (same runtime, same number of tickets)

1 = both jobs finish at the same time

0.5 = jobs run in sequence – one runs to completion, then the other runs to completion



# Stride Scheduling

- idea
  - each process has associated *stride* and *pass* values
    - stride is inversely proportional to the number of tickets the process has – it measures how much of a time step is credited to a process when it runs
    - pass is a counter reflecting how much time the process has gotten – when the process runs, its pass is incremented by its stride
- algorithm
  - for each time slice, the scheduler picks the process with the lowest pass value to run
    - pass is incremented by stride

```
curr = remove_min(queue);   // pick client with min pass
schedule(curr);             // run for quantum
curr->pass += curr->stride; // update pass using stride
insert(queue, curr);        // return curr to queue
```



# Example

- scenario
  - ticket counts
    - A has 100 tickets
    - B has 50 tickets
    - C has 250 tickets
  - stride
    - pick a large value to divide by the ticket counts
    - A: 10000/100 = 100
    - B: 10000/50 = 200
    - C: 10000/250 = 40

why 10000? there's nothing special about it as the ratios of the stride values will be the same for any number picked, but a large enough number means that the stride values can be integers without (much) loss of accuracy

  - pass
    - all pass values start at 0

| 100 | 200 | 40 | | 100 | 50 | 250 |
|---|---|---|---|---|---|---|
| pass | | | who runs? | share of CPU time | | |
| A | B | C | | A | B | C |
| | | | | 0 | 0 | 0 |
| 0 | 0 | 0 | A | 1 | 0 | 0 |
| 100 | 0 | 0 | B | 1 | 1 | 0 |
| 100 | 200 | 0 | C | 1 | 1 | 1 |
| 100 | 200 | 40 | C | 1 | 1 | 2 |
| 100 | 200 | 80 | C | 1 | 1 | 3 |
| 100 | 200 | 120 | A | 2 | 1 | 3 |
| 200 | 200 | 120 | C | 2 | 1 | 4 |
| 200 | 200 | 160 | C | 2 | 1 | 5 |
| 200 | 200 | 200 | A | 3 | 1 | 5 |
| 300 | 200 | 200 | B | 3 | 2 | 5 |
| 300 | 400 | 200 | C | 3 | 2 | 6 |
| 300 | 400 | 240 | C | 3 | 2 | 7 |
| 300 | 400 | 280 | C | 3 | 2 | 8 |
| 300 | 400 | 320 | A | 4 | 2 | 8 |
| 400 | 400 | 320 | C | 4 | 2 | 9 |
| 400 | 400 | 360 | C | 4 | 2 | 10 |
| 400 | 400 | 400 | A | 5 | 2 | 10 |
| 500 | 400 | 400 | B | 5 | 3 | 10 |
| 500 | 600 | 400 | C | 5 | 3 | 11 |
| 500 | 600 | 440 | C | 5 | 3 | 12 |
| 500 | 600 | 480 | C | 5 | 3 | 13 |
| 500 | 600 | 520 | A | 6 | 3 | 13 |
| 600 | 600 | 520 | C | 6 | 3 | 14 |
| 600 | 600 | 560 | C | 6 | 3 | 15 |
| 600 | 600 | 600 | | 6 | 3 | 15 |

each time all the pass values match, every process has gotten its exact share according to its ticket count



# Stride Scheduling

- advantages
  - deterministic
  - guaranteed to have allocated exactly the right proportion of CPU time for each process every complete scheduling cycle
- challenges
  - what should pass be initialized to for new processes?
    - if 0, the new process will monopolize the CPU until it catches up to processes that have been around longer

## Weighted Fair Queueing (WFQ)

- RR with different priorities for processes
  - priorities can be decided by scheduler or set by users
  - time slice allocated is proportional to the priority
    - higher priority gets a longer time slice
- wrinkle: it may not be possible to exactly enforce the time slice
  - timer interrupts may not align with slice
  - process may block before the end of its slice
- solution: keep track of the actual running time of each process and schedule the process that has used the least fraction of its fair share (as determined by its priority)
  - adjust future slice sizes to compensate for difference between time used and fair share



## Challenges of Fair Share Scheduling

- jobs that perform I/O occasionally may not get a fair share of CPU time
- how to assign tickets or priorities?
  - MLFQ handles this automatically
- but there are also many applications where fair share schedulers are effective
  - e.g. virtualized data center or cloud