Operating System (OS)
CS232

Scheduling Algorithm: Proportional Share Scheduling

Dr. Muhammad Mobeen Movania

Outlines

- Why Proportional Share Scheduler
- Types of Proportional Share Schedulers
  - Lottery Scheduling/Fair Share Scheduler
  - Stride Scheduling/Deterministic Fair Share Scheduler
  - Completely Fair Scheduler (CFS)
- Weighing Parameter
- Implementation Details
- Summary

Why Proportional Share Schedulers?

- Key Idea
  - Instead of optimizing for turnaround or response time, a scheduler might instead try to guarantee that each job obtain a certain percentage of CPU time
- Key Questions
  - How to share the CPU proportionally?
    - What are the key mechanisms?
    - How effective are they?

Types of Proportional Share Schedulers

- Three types on the mechanism through which the proportional share of CPU is given
  - Lottery Scheduling (Hold a lottery randomly to see which process runs next)
  - Stride Scheduling (Deterministic lottery scheduling based on a stride value which is inversely proportional to the no. of tickets)
  - CFS Scheduler (Fairly divide CPU among all processes through accumulating virtual runtime)

Lottery Scheduling

- Each process is awarded a certain number of ***tickets***.
- Key Idea:
  - The ***percentage of tickets*** a process has represents its ***share of the resource*** in question
- Mechanism:
  - The Lottery scheduler holds a lottery randomly to pick a winning ticket
  - The process to which the winning ticket belongs to runs next

Example - Lottery Scheduling

- Total Tickets: 100 (A 75% share and B 25% share)
  - Process A (Tickets: 0-74), Process B (Tickets:75-99)
- Example of Lottery scheduler’s winning tickets

  63 85 70 39 76 17 29 41 36 39 10 99 68 83 63 62 43 0 49

- Resulting schedule

  A B A A B A A A A A A B A B A A A A A

- Summary
  - No guarantees due to randomness
  - We get 16/20 (80% A) and 4/20 (20% B)
  - More chances and we are likely to get the desired percentages

Ticket Currency

- Mechanism to manipulate tickets in lottery scheduling
- Ticket Currency
  - allows a user with a set of tickets to allocate tickets among their own jobs in whatever currency they would like
  - the system then automatically converts said currency into the correct global value

Example - Ticket Currency

- Two Users A and B each with 100 tickets
  - User A
    - Runs two jobs A1 and A2 gives each 500 tickets in A’s currency (equal to 50 tickets in global currency)
  - User B
    - Runs a single job and gives it 10 tickets (equal to 50 tickets in global currency)
- Lottery is held on global ticket currency to determine which job runs
- Why use currency
  - Allows ticket transfer: a process can hand off its tickets to another process (useful in client/server setting)
  - Allows ticket inflation: a process can temporarily raise or lower the number of tickets it owns

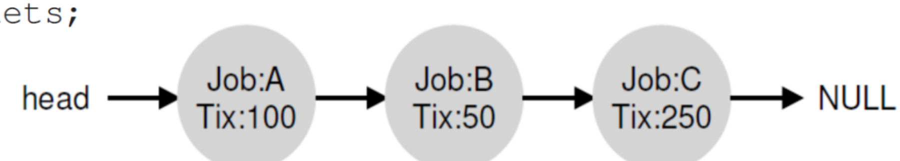
Lottery - Implementation

- Easy to implement as we just need
  - A good random number generator
  - A data structure to keep track of processes
  - Total number of tickets

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

// winner: use some call to a random number generator to
//         get a value, between 0 and the total # of tic
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...
```



Issues with Lottery Scheduling

- 1) Unfairness
  - Two processes A and B with the same number of tickets (100), each job should finish at the same time but this is generally not the case
  - Unfairness Metric (U) = $\frac{completion\_time_A}{completion\_time_B}$
  - U approaches 1 only as the jobs run for a significant number of time slices
- 2) How to assign tickets?
  - Open problem
  - We assume the user knows which ticket should be assigned to which job
- 3) Non-deterministic
  - Based on randomness which approximates to correct answer when outcomes are more

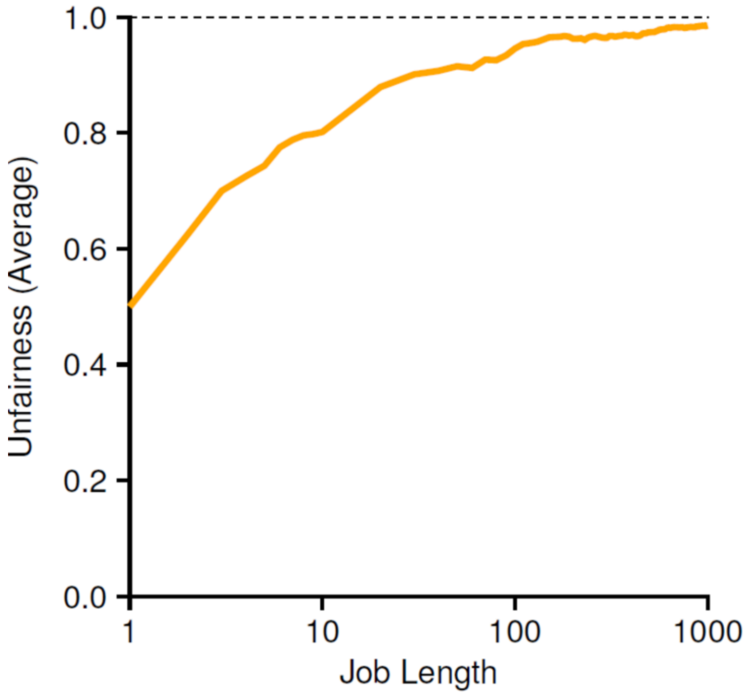


Figure 9.2: **Lottery Fairness Study**

Stride Scheduling – Deterministic Proportional Share Scheduling

- Stride Scheduling
  - $stride \propto \frac{1}{Ticket\_share}$
- Each process has a counter (*pass* value)
- At any given time scheduler will:
  - Select the process with minimum pass value
  - Increment its pass value by its stride
  - Schedule it

Example – Stride Scheduler

- Three processes, A(100), B(50), C(250) tickets
- Total tickets = 10000
  - $stride_A = \frac{10000}{100} = 100,$
  - $stride_B = \frac{10000}{50} = 200,$
  - $stride_C = \frac{10000}{250} = 40$
- Each process starts with a pass value of 0

Example – Stride Scheduler

Pass(A) (stride=100)	Pass(B) (stride=200)	Pass(C) (stride=40)	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	...

Figure 9.3: **Stride Scheduling: A Trace**

Issues - Stride Scheduler

- Example Summary
  - C ran five times, A twice, and B just once, exactly in proportion to their ticket values of 250, 100, and 50
- Lottery scheduling achieves the proportions probabilistically over time but stride scheduling gets them exactly right at the end of each scheduling cycle.
- Lottery scheduling has no global state so incorporating new processes is a breeze
  - In stride scheduling, its difficult to assign a correct pass value to the new job
  - If the new job is given a pass value of 0, it monopolizes the CPU

Completely Fair Scheduler (CFS)

- Linux Scheduler
- Uses Fair Share Scheduling
- Doesn’t have a fixed time slice
- Keeps a counter vruntime for each process and adds a time slice to it for each process
- The value of time slice is dynamically adjusted for the next run based on the available workload
- When a scheduling decision occurs, it’d select the process with lowest vruntime to run
- Run for how long?
  - Time slice too small → good fairness but scheduling overhead
  - Time slice too big → low scheduling overhead but costs fairness in short term

CFS

- Maintains various parameters
  - sched_latency
  - $time\_slice = \dfrac{sched\_latency}{num\_processes}$
  - sched_latency = 48, num_processes=4
    - time_slice = 12

Time slice adjusted dynamically

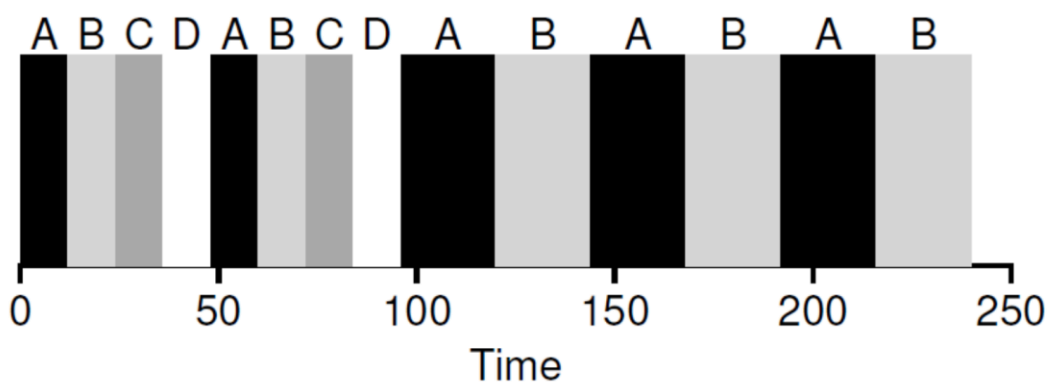


Figure 9.4: **CFS Simple Example**

CFS

- What if too many processes running?
  - min_granularity
  - time_slice = max(min_granularity, calculated_time_slice_value)
- Timer interrupt goes off every 1 ms
  - Everytime a process leaves CPU, its vruntime gets updated
  - This way eventually fairness is ensured

CFS: Niceness

- Processes can be nice to other processes (or not)
- A higher nice value would give them a lower weight → smaller slice
- nice_A = -5, nice_B = 0 (default value)

$$\mathtt{time\_slice}_k = \frac{\mathtt{weight}_k}{\sum_{i=0}^{n-1} \mathtt{weight}_i} \cdot \mathtt{sched\_latency} \tag{9.1}$$

$$\mathtt{vruntime}_i = \mathtt{vruntime}_i + \frac{\mathtt{weight}_0}{\mathtt{weight}_i} \cdot \mathtt{runtime}_i \tag{9.2}$$

- process vruntime will also increase in inverse proportion to its weight

Data structure: RB Tree

- Self balancing and Sorted
  - Removing a process is trivial
  - Insertion is log(n)
- Only has ready processes inserted on their vruntime values
- I/O, sleeping processes assigned min vruntime when they join.

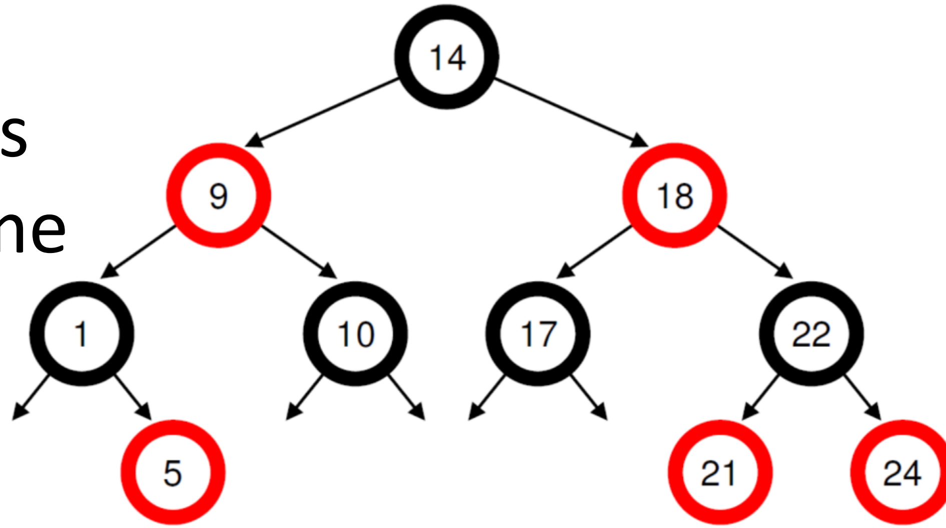


Figure 9.5: **CFS Red-Black Tree**

Summary

- We introduced proportional-share scheduling and briefly discussed three approaches
  - lottery scheduling, stride scheduling, and the Completely Fair Scheduler (CFS)
- Lottery uses randomness to achieve proportional share, is non-deterministic but scales well
- Stride uses stride to calculate exact proportions hence is deterministic but does not scale well
- CFS is like a weighted round robin scheduler with dynamic time slices, scales and performs well