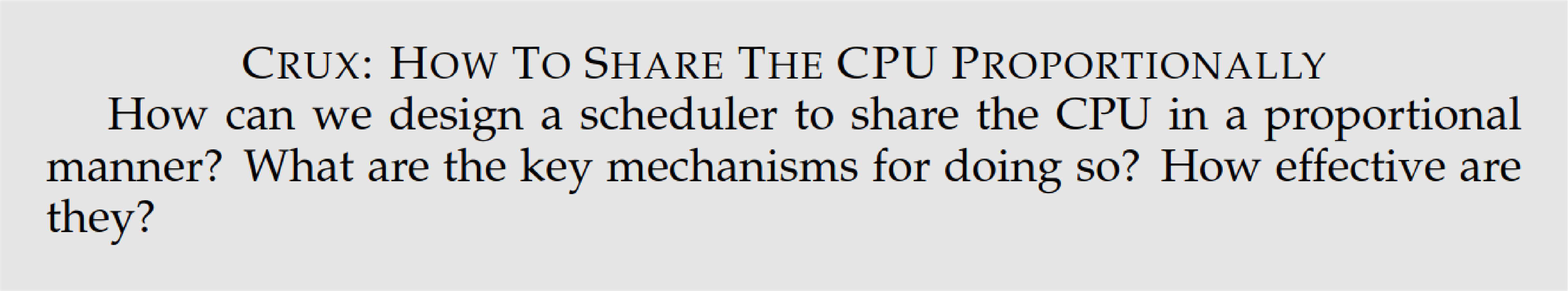
Chapter 9: Proportional Share Scheduling

Adam Disney

THE UNIVERSITY OF
TENNESSEE
KNOXVILLE

A new metric

- Instead of focusing on turnaround or response time, let's guarantee each job gets a certain percentage of CPU time.
- We'll do this with a **proportional-share** or **fair-share** scheduler.
  - One example is known as **lottery scheduling.**



Crux: How To Share The CPU Proportionally

How can we design a scheduler to share the CPU in a proportional manner? What are the key mechanisms for doing so? How effective are they?

Basic Concept: Tickets Represent Your Share

- **Tickets** represent the share of a resource that an entity should receive.
- The percent of the total tickets an entity holds represents the percentage of the resource it should receive.
- For example, if process A has 75 tickets and process B has 25 tickets, then A should receive 75% of the CPU time and B should receive 25% of the CPU time.

Basic Concept: Tickets Represent Your Share

- Lottery scheduling achieves this probabilistically by holding a lottery.
- The idea is to draw a ticket and schedule the process that holds that ticket.

Ticket Mechanisms: Ticket Currency

- **Ticket currency** allows a user to create their own currency.
  - There's a global currency and then whatever currency each user creates.
- For example, user A and user B both are given 100 global tickets.
  - User A runs two jobs A1 and A2
    - A1 and A2 are given 500 "A bucks" each
  - User B runs one job B1
    - B1 is given 10 "B bucks"
- A1 and A2 each have 50% of the "A bucks" so they each have 50 global tickets.
- B1 has 100% of the "B bucks" so it has 100 global tickets.

Ticket Mechanisms: Ticket Transfer

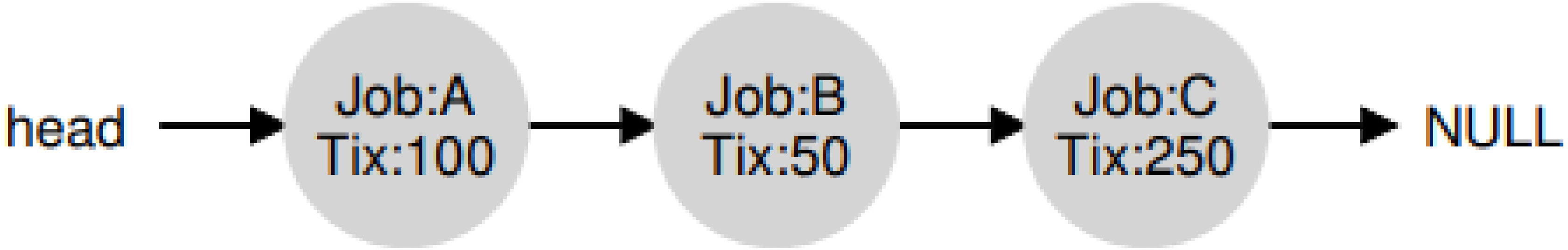
- **Ticket transfer** allows a process to lend tickets to another process.
- Useful in a cooperative setting like client/server running on the same machine.
  - The client sends a request to the server.
  - Since it must wait on the server to finish the request, the client can lend its tickets to the server to give it a higher chance of being scheduled.
  - When done, the server gives the tickets back to the client.

Ticket Mechanisms: Ticket Inflation

- **Ticket inflation** allows a process to change the number of tickets it owns.
  - Really only helpful in a cooperative environment.
- If a process knows it's going to need more CPU time, it can simply boost its tickets without coordinating with other processes.

Implementation

- **Lottery scheduling** is quite simple. It only needs:
  - a good random number generator
  - a data structure to track processes (e.g., a list)
  - a total number of tickets.
- Generate a number, N
- Traverse list of processes adding up their ticket values
- Winner once the total is greater than N



Implementation

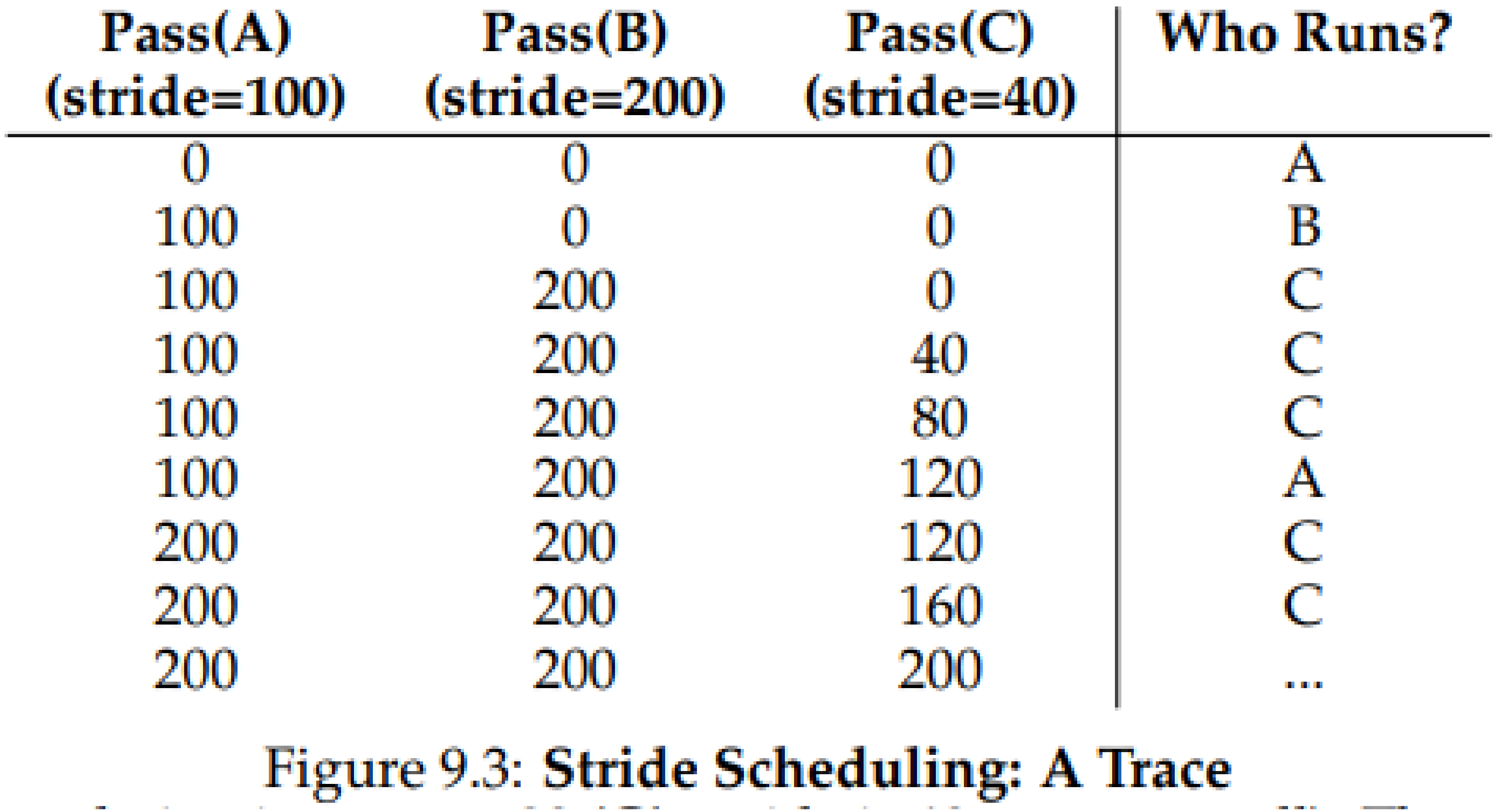
- If we run two jobs of the same length, how fair is this scheduler?
- Randomness affects short jobs
- Fairness = job_finish_1 / job_finish_2
- We want fairness to be 1

Figure 9.2: **Lottery Fairness Study**

Stride Scheduling

- Why use randomness at all if sometimes it isn't fair?
- **Stride Scheduling** is a deterministic fair-share scheduler.
- Assign each job a **stride** which is the inverse in proportion to the number of tickets it has**.**
  - Stride = (some large number) / tickets
- Each process has a running **pass** value starting at 0.
- When a process is scheduled, increment its **pass** by **stride.**
- Always schedule the process with the lowest **pass** breaking ties arbitrarily.

Stride Scheduling



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**

Stride Scheduling

- Each process ran exactly in proportion to its tickets so why use lottery scheduling at all?
  - No global state
  - If a new process comes along, what should its **pass** be?

The Linux Completely Fair Scheduler (CFS)

- Highly efficient and scalable fair-share scheduler.
- Aims to spend very little time making decisions.
- This is important to not waste resources.
  - Google datacenter even after aggressive optimization used 5% of the CPU time scheduling!
- Reducing overhead is a key goal in modern schedulers.
- Goal is to divide the CPU evenly among all competing processes.
- It does so with a **virtual runtime (vruntime)**

CFS: Basic Operation

- Each time a process runs, it accumulates vruntime.
- CFS always picks the process with the lowest vruntime.
- CFS varies the time slice size with sched_latency.
  - sched_latency represents the largest time slice size possible
  - Time slice size is determined simply by sched_latency divided by the number of processes running.
- CFS uses min_granularity to prevent the time slice from becoming too small.

CFS: Basic Operation

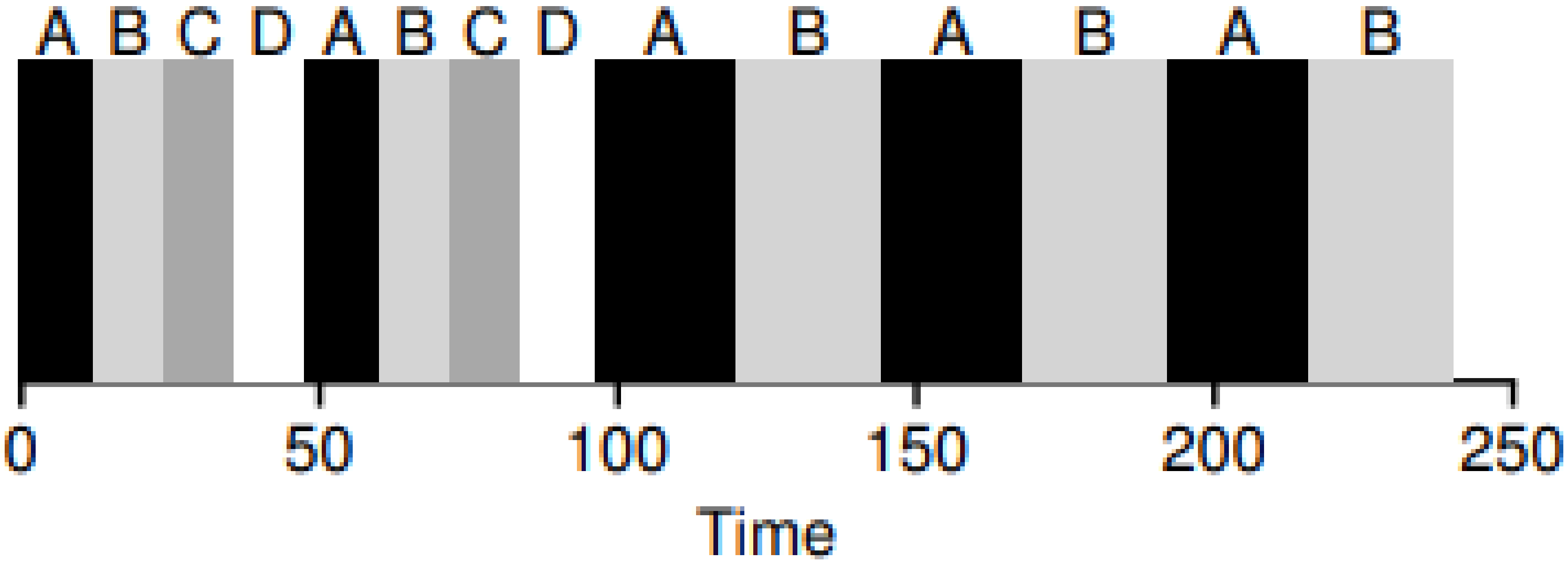
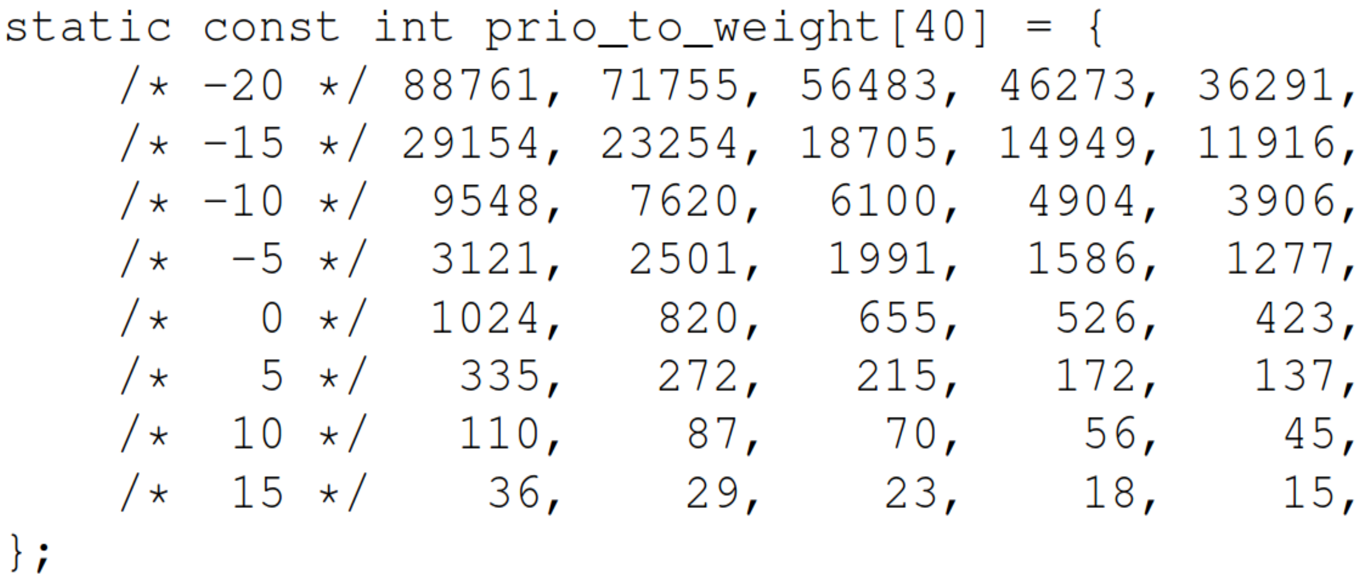


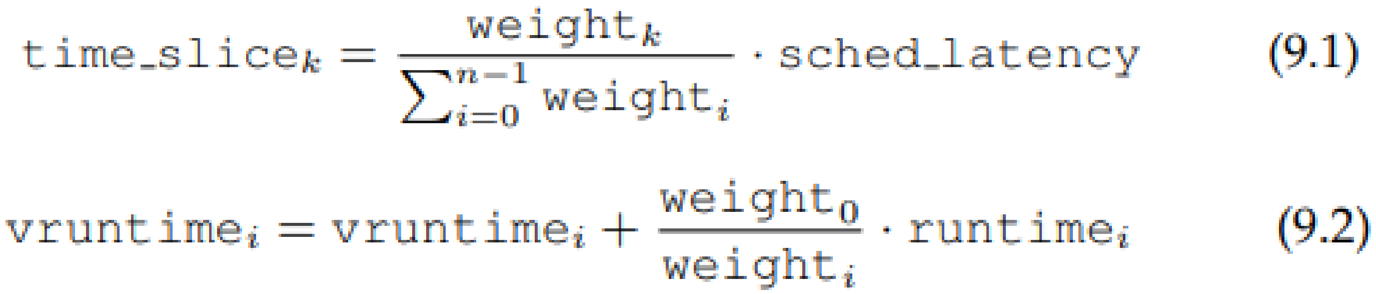
Figure 9.4: **CFS Simple Example**

CFS: Niceness

- Niceness adds weighting to the time slice calculation.
- Time slice = portion of weight all processes running * max time slice
- Vruntime = previous vruntime + time just ran * weighting based on niceness



```
static const int prio_to_weight[40] = {
    /* -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,
};
```



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

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

CFS: Using Red-Black Trees

- A list is inefficient when looking for the lowest vruntime so we use a red-black tree keyed on vruntime.
- Why not a heap?

CFS: Dealing With I/O And Sleeping Processes

- When a job wakes up from sleeping or becomes unblocked, it's vruntime is set to the maximum of its own vruntime and the minimum in the tree.
- This avoids starvation at the cost of not being fair to frequently sleeping processes.

CFS: Other Fun

- Of course, there are many other features to tune this scheduler to deal with
  - Cache performance
  - Multiple CPUs
  - Large groups of processes