Scheduling: Proportional Share
Proportional Share Scheduler

• **Fair-share scheduler**
  • Guarantee that each job obtain *a certain percentage* of CPU time
  • Not optimized for turnaround or response time
Basic Concept

• Tickets
  • Represent the share of a resource that a process should receive
  • The percent of tickets represents its share of the system resource in question

• Example
  • There are two processes, A and B
    • Process A has 75 tickets → receive 75% of the CPU
    • Process B has 25 tickets → receive 25% of the CPU
Lottery scheduling

• The scheduler picks a winning ticket
  • Switch to the winning process and run it

• Example
  • There are 100 tickets
    • Process A has 75 tickets: 0 to 74
    • Process B has 25 tickets: 75 to 99

  | Scheduler’s winning tickets: 63 85 70 39 76 17 29 41 36 39 10 99 68 83 63 |
  | Resulting scheduler:       A   B   A   A   B   A   A   A   A   A   B   A   B   A |

  A runs 11/15 = 73.3%, B runs 4/15 = 26.7%

The longer these two jobs compete,
The more likely they are to achieve the desired percentages
Ticket Mechanisms

• Ticket currency
  • A user allocates tickets among their own jobs in whatever currency they would like
  • The system converts the currency into the correct global value
  • Example
    • There are 200 tickets (Global currency)
    • Process A has 100 tickets
    • Process B has 100 tickets

User A  \[ \rightarrow 500 \text{ (A's currency)} \text{ to } A1 \rightarrow 50 \text{ (global currency)} \]
\[ \rightarrow 500 \text{ (A's currency)} \text{ to } A2 \rightarrow 50 \text{ (global currency)} \]

User B  \[ \rightarrow 10 \text{ (B's currency)} \text{ to } B1 \rightarrow 100 \text{ (global currency)} \]
Ticket Mechanisms (Cont.)

• Ticket transfer
  • A process can temporarily hand off its tickets to another process

• Ticket inflation
  • A process can temporarily raise or lower the number of tickets it owns
  • If any one process needs more CPU time, it can boost its tickets
  • Assumes processes cooperate and are friendly with each other
Implementation

• Example: There are three processes, A, B, and C

  • Keep the processes in a list:

    ```
    head
    Job:A
      Tix:100
    Job:B
      Tix:50
    Job:C
      Tix:250
    NULL
    ```

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

3 // winner: use some call to a random number generator to
4 // get a value, between 0 and the total # of tickets
5 int winner = getrandom(0, totaltickets);

7 // current: use this to walk through the list of jobs
8 node_t *current = head;

9 // loop until the sum of ticket values is > the winner
10 while (current) {
11    counter = counter + current->tickets;
12    if (counter > winner)
13      break; // found the winner
14    current = current->next;
15  }

18 // 'current' is the winner: schedule it...
```
Implementation (Cont.)

• $U$: unfairness metric
  • The time the first job completes divided by the time that the second job completes

• Example:
  • There are two jobs, each jobs has runtime 10
    • First job finishes at time 10
    • Second job finishes at time 20
  • $U = \frac{10}{20} = 0.5$
  • $U$ will be close to 1 when both jobs finish at nearly the same time
Lottery Fairness Study

• There are two jobs
  • Each job has the same number of tickets (100)

When the job length is not very long, average unfairness can be quite severe
Lottery Discussion

• Simplicity of implementation
  • Random number generator
  • List of processes
  • Total number of tickets

• How do you assign tickets?
  • Tough problem
  • System behavior depends on how tickets are allocated

• Let the users decide how to allocate tickets?
Stride Scheduling

• Random is easy to implement, but may not deliver the exact right proportions

• **Stride** of each process
  • (A large number) / (the number of tickets of the process)
  • Example: A large number = 10,000
    • Process A has 100 tickets → stride of A is 100
    • Process B has 50 tickets → stride of B is 200

• A process runs, increment a counter (=pass value) for it by its stride.
  • Pick the process to run that has the **lowest pass value**

```c
current = remove_min(queue); // pick client with minimum pass
schedule(current); // use resource for quantum
current->pass += current->stride; // compute next pass using stride
insert(queue, current); // put back into the queue
```

A pseudo code implementation
Stride Scheduling Example

<table>
<thead>
<tr>
<th>Pass(A) (stride=100)</th>
<th>Pass(B) (stride=200)</th>
<th>Pass(C) (stride=40)</th>
<th>Who Runs?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>A</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>0</td>
<td>B</td>
</tr>
<tr>
<td>100</td>
<td>200</td>
<td>0</td>
<td>C</td>
</tr>
<tr>
<td>100</td>
<td>200</td>
<td>40</td>
<td>C</td>
</tr>
<tr>
<td>100</td>
<td>200</td>
<td>80</td>
<td>C</td>
</tr>
<tr>
<td>100</td>
<td>200</td>
<td>120</td>
<td>A</td>
</tr>
<tr>
<td>200</td>
<td>200</td>
<td>120</td>
<td>C</td>
</tr>
<tr>
<td>200</td>
<td>200</td>
<td>160</td>
<td>C</td>
</tr>
<tr>
<td>200</td>
<td>200</td>
<td>200</td>
<td>...</td>
</tr>
</tbody>
</table>

If new job enters with pass value 0, It will *monopolize* the CPU!