Universidade Estadual de Campinas
Instituto de Computação

MC504 Sistemas Operacionais

T06 CPU Virtualization: Scheduling
Proportional Share

Referência principal
Ch.9 of Operating Systems: Three Easy Pieces by Remzi and Andrea Arpaci-Dusseau (pages.cs.wisc.edu/~remzi/OSTEP/)

Discutido em classe em 20 de agosto de 2018

Arthur João Catto, PhD

2º semestre de 2018

Proportional Share Scheduler

- Also referred to as a *fair-share scheduler*
- Tries to guarantee that each job obtains a certain percentage of CPU time.
- Is not optimized for turnaround or response time.
- We will discuss two implementations
  - Lottery scheduling
  - Stride scheduling

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** are the basic concept underlying proportional-share scheduling:
  - Indicate the relative amount of a resource that a process should receive.
  - The percent of tickets that a process holds represents its share of the system resource in question.
- Example
  - There are two processes, A and B, in a system with 40 tickets.
    - Process A has 30 tickets → it is supposed to receive 75% of the CPU
    - Process B has 10 tickets → it is supposed to receive 25% of the CPU
- Lottery scheduling achieves this probabilistically (*but not deterministically*) by holding a lottery every so often (say, every time slice).

Lottery scheduling

- The scheduler knows how many tickets are in the system.
- The scheduler picks a winning ticket, loads the state of that *winning process* and runs it.
- Example
  - There are 100 tickets
  - Process A has 75 tickets ($0 \longrightarrow 74$) and Process B has 25 tickets ($75 \longrightarrow 99$).

Winning tickets	63	85	70	39	76	17	29	41	36	39	10	99	68	83	63	62	43	0	49	49
Resulting schedule	A		A	A		A	A	A	A	A	A		A		A	A	A	A	A	A
		B			B							B		B						

- Randomness leads to probabilistic correctness but no guarantee.
  - In the example, B got 20% of CPU time instead of desired 25%.

Ticket Mechanisms

- Ticket currency
  - Users allocate tickets among their own jobs in whatever currency they would like.
  - The system converts user's currency into a meaningful global value.
  - Example
    - The system will share 200 tickets equally among users A and B.
    - User A runs two jobs A1 and A2 and has given 30 tickets to A1 and 20 to A2 (A's currency).
    - User B is running only one job B1 and has given it 10 tickets (B's currency).

User	Global Tkts
A	100
B	100

User	Job	Local Tkts	Local %	Global Tkts
A	A1	30	60%	60
A	A2	20	40%	40
B	B1	10	100%	100

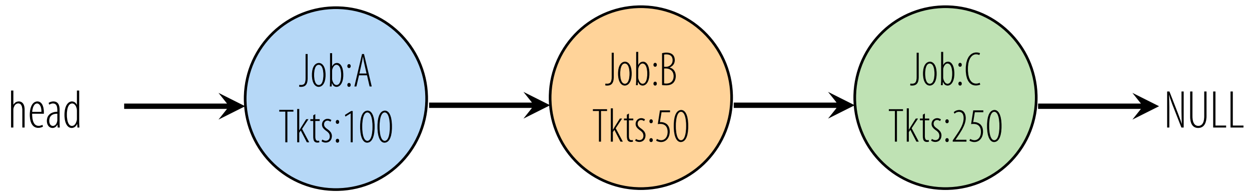
Ticket Mechanisms (Cont.)

- Ticket transfer
  - A process can temporarily hand off its tickets to another process.
  - Especially useful in a client/server setting, where a client can transfer its tickets to a server to increase its share of CPU time while it does some work on the client's behalf.
- Ticket inflation
  - This makes sense only in a cooperative scenario, where processes trust one another.
  - 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 and get its share increased without any communication with other processes.

Implementation

- Requirements
  - Good random number generator to pick the winning tickets
  - A data structure to track the processes in the system
  - The total number of tickets

- Example:
  - There are three processes, A, B, and C, kept in a list.



  - To make a decision, the scheduler picks a random number (the winner) from the total number of tickets (400, in this example).
  - Then, it traverses the list, using a simple counter to find the winner, as shown by the code in the next slide.

Lottery scheduling decision code

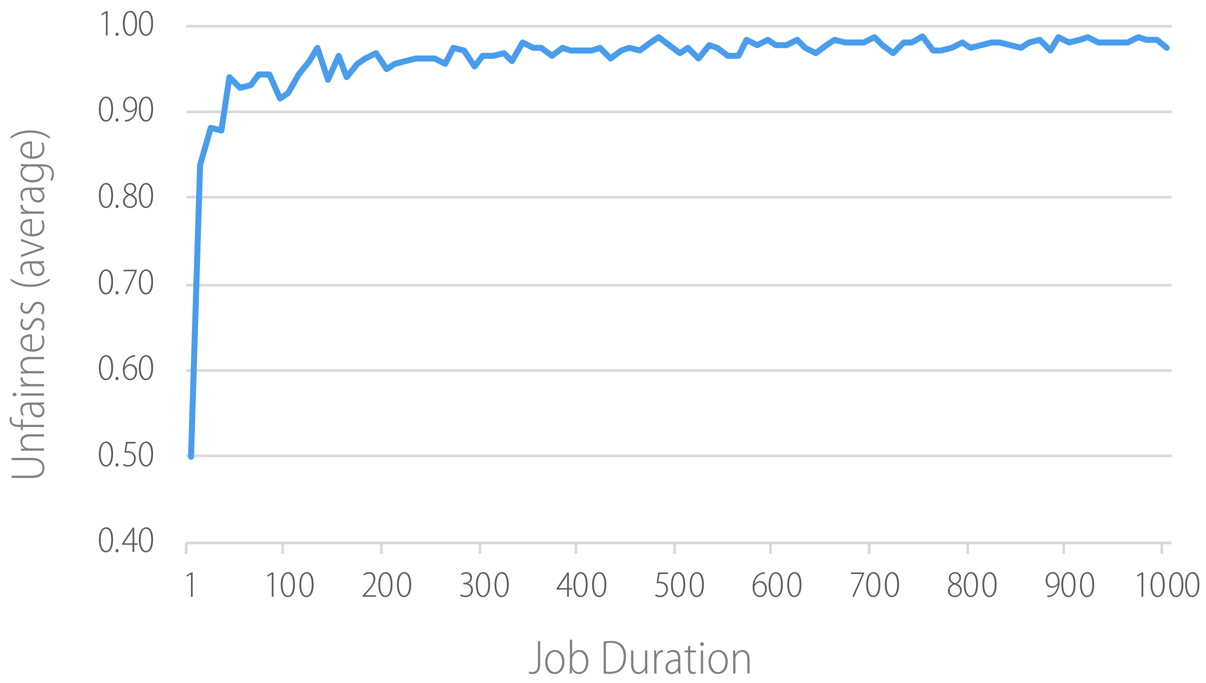
```
// counter: used to track if we've found the winner yet
int counter = 0;

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

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

// loop until the sum of ticket values is > the winner
while (current) {
    counter = counter + current->tickets;
    if (counter > winner)
        break;          // found the winner
    current = current->next;
}
// 'current' is the winner: schedule it...
```

Brief study of lottery fairness

- Consider two jobs competing against each other with the same number of tickets and same run time.
- Let us define an unfairness metric $\boldsymbol{U}$ as the ratio between the times to completion of the first and the second job to finish.
  - $\boldsymbol{U}$ will be close to 1 when both jobs finish at nearly the same time.
  - Since both jobs have the same number of tickets and run time, the closer $\boldsymbol{U}$ is to 1, the fairer the scheduler.
- Example:
  - First job finishes at time 10 and the second job finishes at time 20
    - $\boldsymbol{U = \frac{10}{20} = 0.5}$

Brief study of lottery fairness

- Two jobs with the same number of tickets
- Job duration varying from 1 to 1000 with step = 10
- 10 simulation runs for each job duration



- For short job durations, unfairness can be quite severe.

Stride Scheduling

- The behavior of lottery scheduling is non-deterministic and only probabilistically correct.
- Stride scheduling is an also straightforward deterministic fair-share method.
- Each job in a system has a *stride* which is inversely proportional to its number of tickets.
  - The more tickets a job has, the smaller its *stride*.
- Each job also has a *pass*, which is an indicator of its progress.
- Every time a job runs, its *pass* is incremented by its *stride*.
- Every time the scheduler needs a process, it picks the one with the lowest *pass* value.

An example of stride scheduling

- Consider processes A, B and C with *stride* values of 100, 200 and 40, all with *pass* values initially at 0, and the following scheduling algorithm

```
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
```

- The behavior of the scheduler over time could be as shown in the table.

Time slice	Pass(A)	Pass(B)	Pass(C)	Who runs?
0	0	0	0	

- Can you demonstrate that the allocation was precise?

Summary

- Although lottery and stride scheduling are conceptually interesting and straightforward to implement, they have not achieved widespread adoption as CPU schedulers for a number of reasons, e.g.
  - They do not address I/O
  - They leave open the hard problem of ticket assignment.
- General-purpose schedulers such as MLFQ do better and thus are more widely adopted.
- On the other hand, proportional-share schedulers are more useful in virtualized scenarios (e.g. VMWare) where fair-sharing of resources is important and some of these problems are relatively less impacting.