Advanced Operating Systems (CS 202)

Scheduling (2)
Lottery Scheduling
Problems with Traditional schedulers

- Priority systems are ad hoc: highest priority always wins
- Try to support fair share by adjusting priorities with a feedback loop
  - Works over long term
  - highest priority still wins all the time, but now the Unix priorities are always changing
- Priority inversion: high-priority jobs can be blocked behind low-priority jobs
- Schedulers are complex and difficult to control
Lottery scheduling

- Elegant way to implement proportional share scheduling
- Priority determined by the number of tickets each thread has:
  - Priority is the relative percentage of all of the tickets whose owners compete for the resource
- Scheduler picks winning ticket randomly, gives owner the resource
- Tickets can be used for a variety of resources
Example

- Three threads
  - A has 5 tickets
  - B has 3 tickets
  - C has 2 tickets

- If all compete for the resource
  - B has 30% chance of being selected

- If only B and C compete
  - B has 60% chance of being selected
Its fair

- Lottery scheduling is *probabilistically fair*
- If a thread has a $t$ tickets out of $T$
  - Its probability of winning a lottery is $p = t/T$
  - Its expected number of wins over $n$ drawings is $np$
    - Binomial distribution
    - Variance $\sigma^2 = np(1 - p)$
Fairness (II)

- Coefficient of variation of number of wins
  \[ \frac{\sigma}{np} = \sqrt{\frac{(1-p)}{np}} \]
  - Decreases with \( \sqrt{n} \)

- Number of tries before winning the lottery follows a **geometric distribution**

- As time passes, each thread ends receiving its share of the resource
Ticket transfers

- **How to deal with dependencies?**
  - Explicit transfers of tickets from one client to another

- **Transfers can be used whenever a client blocks due to some dependency**
  - When a client waits for a reply from a server, it can temporarily transfer its tickets to the server
    - Server has no tickets of its own
  - Server priority is sum of priorities of its active clients
    - Can use lottery scheduling to give service to the clients

- **Similar to priority inheritance**
  - Can solve priority inversion
Ticket inflation

› Lets users create new tickets
  › Like printing their own money
  › Counterpart is *ticket deflation*
  › Lets mutually trusting clients adjust their priorities dynamically without explicit communication

› Currencies: set up an exchange rate
  › Enables inflation within a group
  › Simplifies mini-lotteries (e.g., for mutexes)
Example (I)

- A process manages three threads
  - A has 5 tickets
  - B has 3 tickets
  - C has 2 tickets

- It creates 10 extra tickets and assigns them to process C
  - Why?
  - Process now has 20 tickets
Example (II)

- These 20 tickets are in a new currency whose exchange rate with the base currency is 10/20

- The total value of the processes tickets expressed in the base currency is still equal to 10
Compensation tickets (I)

- I/O-bound threads are likely get less than their fair share of the CPU because they often block before their CPU quantum expires.

- Compensation tickets address this imbalance.
Compensation tickets (II)

- A client that consumes only a fraction \( f \) of its CPU quantum *can* be granted a compensation ticket
  - Ticket inflates the value by \( 1/f \) until the client starts getting the CPU
Example

- CPU quantum is 100 ms
- Client A releases the CPU after 20 ms
  - \( f = 0.2 \) or \( 1/5 \)
- Value of all tickets owned by A will be multiplied by 5 until A gets the CPU
- Is this fair?
  - What if A alternates between 1/5 and full quantum?
Compensation tickets (III)

- Compensation tickets
  - Favor I/O-bound—and interactive—threads
  - Helps them getting their fair share of the CPU
IMPLEMENTATION

- On a MIPS-based DECstation running Mach 3 microkernel
  - Time slice is 100ms
    - Fairly large as scheme does not allow preemption
- Requires
  - A fast RNG
  - A fast way to pick lottery winner
Example

- Three threads
  - A has 5 tickets
  - B has 3 tickets
  - C has 2 tickets

- List contains
  - A (0-4)
  - B (5-7)
  - C (8-9)

Search time is $O(n)$ where $n$ is list length
Optimization – use tree

RB Tree used in Linux
Completely fair scheduler (CFS)
--not lottery based
Long-term fairness (I)
Short term fluctuations

For 2:1 ticket alloc. ratio
Stride scheduling

» Deterministic version of lottery scheduling
» Mark time virtually (counting passes)
  » Each process has a stride: number of passes between being scheduled
  » Stride inversely proportional to number of tickets
  » Regular, predictable schedule
» Can also use compensation tickets
» Similar to weighted fair queuing
  » Linux CFS is similar
Stride Scheduling – Basic Algorithm

Client Variables:

- **Tickets**
  - Relative resource allocation

- **Strides (**
  - Interval between selection

- **Pass (**
  - Virtual index of next selection

- minimum ticket allocation
Stride Scheduling – Basic Algorithm

3:2:1 Allocation

- A (stride = 2)
- B (stride = 3)
- C (stride = 6)

Time 1:

Time 2:

Pass Value

Time (quanta)
Stride Scheduling – Basic Algorithm

3:2:1 Allocation

- A (stride = 2)
- B (stride = 3)
- C (stride = 6)

Time 1:
- A: 2
- B: 3
- C: 6

Time 2:
- A: 4
- B: 3
- C: 6

Time 3:
- A: 4
- B: 6
- C: 6

Graph showing pass values over time with quanta on the x-axis and pass values on the y-axis.
Stride Scheduling – Basic Algorithm

- A (stride = 2)
- B (stride = 3)
- C (stride = 6)

| Time 1: | 2  | 3  | 6  |
| Time 2: | 4  | 3  | 6  |
| Time 3: | 4  | 6  | 6  |
| Time 4: | 6  | 6  | 6  |

△ - A (stride = 2)
○ - B (stride = 3)
□ - C (stride = 6)
Stride Scheduling – Basic Algorithm

**3:2:1 Allocation**
- △ - A (stride = 2)
- ○ - B (stride = 3)
- □ - C (stride = 6)

<table>
<thead>
<tr>
<th>Time 1</th>
<th>Time 2</th>
<th>Time 3</th>
<th>Time 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>+2</td>
<td>+3</td>
<td>+2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>
Throughput Error Comparison

Error is independent of the allocation time in stride scheduling.

Hierarchical stride scheduling has more balance distribution of error between clients.
Accuracy of Prototype Implementation

- Lottery and Stride Scheduler implemented on real-system.
- Stride scheduler stayed within 1% of ideal ratio.
- Low system overhead relative to standard Linux scheduler.
Linux scheduler

- Went through several iterations
- Currently CFS
  - Fair scheduler, like stride scheduling
  - Supersedes O(1) scheduler: emphasis on constant time scheduling – why?
  - CFS is O(log(N)) because of red-black tree
  - Is it really fair?
- What to do with multi-core scheduling?