Outline for Today

• Objectives:
  – Linux scheduler
  – Lottery scheduling

Linux Scheduling Policy

• Runnable process with highest priority and timeslice remaining runs (SCHED_OTHER policy)
  – Dynamically calculated priority
    • Starts with nice value
    • Bonus or penalty reflecting whether I/O or compute bound by tracking sleep time vs. runnable time:
      – sleep_avg – accumulated during sleep up to MAX_SLEEP_AVG (10 ms default)
      – decremented by timer tick while running
Linux Scheduling Policy

– Dynamically calculated timeslice
  • The higher the dynamic priority, the longer the timeslice:

    Low priority less interactive   |
    | 10ms  | 150ms  | 300ms |
    High priority more interactive

– Recalculated every round when “expired” and “active” swap
– Exceptions for expired interactive
  • Go back on active unless there are starving expired tasks

Runqueue for O(1) Scheduler

active

expired

priority array

priority queue

Higher priority more I/O 300ms

lower priority more CPU 10ms
Runqueue for O(1) Scheduler
Linux Real-time

• No guarantees
• SCHED_FIFO
  – Static priority, effectively higher than SCHED_OTHER processes*
  – No timeslice – it runs until it blocks or yields voluntarily
  – RR within same priority level
• SCHED_RR
  – As above but with a timeslice.

* Although their priority number ranges overlap

Diversion: Synchronization

• Disable Interrupts
• Busywaiting solutions - spinlocks
  – execute a tight loop if critical section is busy
  – benefits from specialized atomic (read-mod-write) instructions
• Blocking synchronization
  – sleep (enqueued on wait queue) while critical section is busy.
Support for SMP

- Every processor has its own private runqueue
- Locking – spinlock protects runqueue
- Load balancing – pulls tasks from busiest runqueue into mine.
- Affinity – cpus_allowed bitmask constrains a process to particular set of processors

Symmetric mp

- `load_balance` runs from `schedule()` when runqueue is empty or periodically esp. during idle.
- Prefers to pull processes from expired, not cache-hot, high priority, allowed by affinity

Lottery Scheduling
Waldspurger and Weihl (OSDI 94)
Claims

• Goal: responsive control over the relative rates of computation
• Claims:
  – Support for modular resource management
  – Generalizable to diverse resources
  – Efficient implementation of proportional-share resource management: consumption rates of resources by active computations are proportional to relative shares allocated

Basic Idea

• Resource rights are represented by lottery tickets
  – abstract, relative (vary dynamically wrt contention), uniform (handle heterogeneity)
  – responsiveness: adjusting relative # tickets gets immediately reflected in next lottery
• At allocation time: hold a lottery; Resource goes to the computation holding the winning ticket.
Fairness

- Expected allocation is proportional to # tickets held - actual allocation becomes closer over time.
- Number of lotteries won by client
  \[ E[w] = n \cdot p \]
  where \( p = \frac{t}{T} \)
- Response time (# lotteries to wait for first win)
  \[ E[n] = \frac{1}{p} \]

| \( w \) | # wins |
| \( t \) | # tickets |
| \( T \) | total # tickets |
| \( n \) | # lotteries |

Example List-based Lottery

\( T = 20 \)

| 10 | 2 | 5 | 1 | 2 |

Summing:

- 10
- 12
- 17

Random(0, 19) = 15
Bells and Whistles

- **Ticket transfers** - objects that can be explicitly passed in messages
  - Can be used to solve priority inversions
- **Ticket inflation**
  - Create more - used among mutually trusting clients to dynamically adjust ticket allocations
- **Currencies** - “local” control, exchange rates
- **Compensation tickets** - to maintain share
  - use only $f$ of quantum, ticket inflated by $1/f$ in next

Kernel Objects

- **ticket**
  - 1000 base amount
  - currency

- **Currency**
  - C_name
  - 300 Active amount

- **Issued tickets**
- **Backing tickets**

15

16
1 alice = 5 base

1 bob = 20 base

1 task2 = .4 alice = 2 base

1 task3 = .4 alice = 1.33 base
Example List-based Lottery

\[ T = 3000 \text{ base} \]

\begin{center}
\begin{tabular}{|c|c|c|c|}
\hline
10 task2 & 2bob & 5 task3 & 1 & 2bob \\
\hline
\end{tabular}
\end{center}

Random(0, 2999) = 1500

Compensation

- A holds 400 base, B holds 400 base
- A runs full 100msec quantum, B yields at 20msec
- B uses 1/5 allotted time
  - Gets 400/(1/5) = 2000 base at each subsequent lottery for the rest of this quantum
    - a compensation ticket valued at 2000 - 400
**Ticket Transfer**

- Synchronous RPC between client and server
- Create ticket in client’s currency and send to server to fund it’s currency
- On reply, the transfer ticket is destroyed

**Control Scenarios**

- Dynamic Control
  Conditionally and dynamically grant tickets
- Adaptability
- Resource abstraction barriers supported by currencies. Insulate tasks.
UI

- mktkt, rmktkt, mkcur, rmcur
- fund, unfund
- lstkt, lscur, fundx (shell)

Relative Rate Accuracy
Fairness Over Time

Client-Server Query Processing Rates
Controlling Video Rates

Insulation
Other Kinds of Resources

- Claim: can be used for any resource where queuing is used
- Control relative waiting times for mutex locks.
  - Mutex currency funded out of currencies of waiting threads
  - Holder gets inheritance ticket in addition to its own funding, passed on to next holder (resulting from lottery) on release.
- Space sharing - inverse lottery, loser is victim (e.g. in page replacement decision, processor node preemption in MP partitioning)

Lock Funding

![Diagram of Lock Funding]

- Waiting thread
- Holding thread
- Mutex currency funded out of currencies of waiting threads
- Holder gets inheritance ticket in addition to its own funding, passed on to next holder (resulting from lottery) on release.
Lock Funding

- New holding thread: 1
- Waiting thread: 1
- Old holding thread: 1
- Mutex Waiting Times

 Mutex Waiting Times

Group A

Group B