Outline for Today

• Objectives:
  – Interrupts (continued)
  – Lottery Scheduling

• Announcements
Role of Interrupts in I/O

So, the program needs to access an I/O device…

• Start an I/O operation (special instructions or memory-mapped I/O)
• Device controller performs the operation asynchronously (in parallel with) CPU processing (between controller's buffer & device).
• If DMA, data transferred between controller's buffer and memory without CPU involvement.
• Interrupt signals I/O completion when device is done.
CPU handles Interrupt

- Device raises interrupt line, CPU detects this, CPU stops current operation, disables interrupts, enters kernel mode, saves current program counter, jumps to predefined location, saves other processor state needed to continue at interrupted instruction.

- For each interrupt number, jumps to address of appropriate interrupt service routine. [do_IRQ()]
  - Interrupt context
  - Kernel stack of whatever was interrupted
  - Can not sleep

- Handlers on this line do what needs to be done. [handle_IRQ_event()]
  - Unless SA_INTERRUPT specified when registered, re-enable interrupts during handler execution
  - If line is shared, loop through all handlers

- Restores saved state at interrupted instruction [ret_from_intr()], returns to user mode.
Interrupt Control

- `local_irq_disable()` and `local_irq_enable()` -- affecting all interrupts for this processor
- `local_irq_save(…)` and `local_irq_restore(…)` – save and disable interrupts on this processor and restore previous interrupt state
- `disable_irq(irq)`, `disable_irq_nosynch(irq)`, `enable_irq(irq)` – affecting particular interrupt line
- Informational:
  - `irqs_disabled()` – local interrupts disabled?
  - `in_interrupt()` – in interrupt context or process context?
  - `In_irq()` – executing an interrupt handler?
Bottom Half Processing

- Deferring work that is too heavyweight for interrupt handling
- Mechanisms:
  - Softirqs
    - "soft interrupts", statically defined (32 max.) action functions that can run concurrently on SMP pending when bit set in 32-bitmask (usually set in associated interrupt handler [raise_softirq()])
    - run with interrupts enabled, proper locking required
  - Tasklets
    - dynamically created functions that are built upon softirqs, two of the same type can not run concurrently
    - lists of tasklet_struct hooked to 2 of the softirqs
  - Workqueue
    - implemented as kernel-based "worker" threads with process context of their own -- thus allowed to sleep
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 resource allocation is proportional to # tickets held - actual allocation becomes closer over time.

• Throughput – Expected number of lotteries won by client
  \[ E[w] = n \times p \text{ where } p = \frac{t}{T} \]

• Response time -- # lotteries to wait for first win
  \[ E[n] = \frac{1}{p} \]

• No starvation

\[
\begin{array}{|c|c|}
\hline
w & \text{# wins} \\
\hline
t & \text{# tickets} \\
T & \text{total # tickets} \\
n & \text{# lotteries} \\
\hline
\end{array}
\]
Example List-based Lottery

\[ T = 20 \]

\[
\begin{array}{cccc}
10 & 2 & 5 & 1 & 2
\end{array}
\]

Summing: \[ 10 + 12 + 17 = 49 \]

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 with I/O
  - use only $f$ of quantum, ticket inflated by $\frac{1}{f}$ in next
Kernel Objects

1000 base

amount

currency

ticket

300

C_name

Issued tickets

Backing tickets

Currency name

Active amount
Example List-based Lottery

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

\[
\begin{array}{cccccc}
10 \text{ task2} & 2\text{bob} & 5 \text{ task3} & 1 & 2\text{bob} \\
\end{array}
\]

base

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, rmtkt, mkcur, rmcur
- fund, unfund
- lstkt, lscur, fundx (shell)
Relative Rate Accuracy
Fairness Over Time

![Graph showing average iterations over time.](image)
Client-Server Query Processing Rates
Controlling Video Rates

![Graph showing cumulative frames over time for different rates A, B, and C.](image-url)
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

Waiting thread 1

lock 1

holding thread 1

Waiting thread 1

connection arrows:
- t to lock
- bt to holding thread
Lock Funding
Mutex Waiting Times

Group B

Group A

Mutex Acquisitions

Waiting Time (sec)