Process-aware Interrupt Scheduling and Accounting

Yuting Zhang and Richard West

Boston University
Boston, MA
{danazh,richwest}@cs.bu.edu

Computer Science

Introduction

- While many RT operating systems exist, aim of this work is to empower off-the-shelf systems with predictable service management
  - Leverage widely-deployed systems having low development and maintenance costs
  - Add safe, predictable and efficient app-specific services to commodity OSes for real-time use
  - Focus of this talk specifically on improving predictability and accountability of interrupt processing

Commodity OSes for Real-Time

- Many variants based on systems such as Linux:
  - Linux/RK, QLinux, RED-Linux, RTAI, KURT Linux, and RT Linux
  - e.g., RTLinux Free provides predictable execution of kernel-level real-time tasks
    - Bounds are enforced on interrupt processing overheads by deferring non-RT tasks when RT tasks require service
    - NOTE: Many commodity systems suffer unpredictability (unbounded delays) due to interrupt-disabling, e.g., in critical sections of poorly-written device drivers

The Problem of Interrupts

- Asynchronous events e.g., from hardware completing I/O requests and timer interrupts…
  - Affect process/thread scheduling decisions
  - Typically invoke interrupt handlers at priorities above those of processes/threads
    - i.e., interrupt scheduling disparate from process/thread scheduling
- Time spent handling interrupts impacts the timeliness of RT tasks and their ability to meet deadlines
- Overhead of handling an interrupt is charged to the process that is running when the interrupt occurs
  - Not necessarily the process associated (if any) with the interrupt

Goals

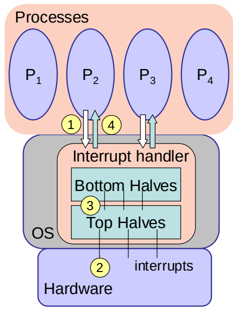
- How to properly account for interrupt processing and correctly charge CPU time overheads to correct process, where possible
- How to schedule deferrable interrupt handling so that predictable task execution is guaranteed

Interrupt Handling

- Interrupt service routines are often split into “top” and “bottom” halves
  - Idea is to avoid lengthy periods of time in “interrupt context”
  - Top half executed at time of interrupt but bottom half may be deferred (e.g., to a schedulable thread)

Process-Independent Interrupt Service

- Traditional approach:
  - (1) I/O service request via kernel
  - (2) OS sends request to device via driver code;
  - Hardware device responds w/ an interrupt, handled by a "top half"
  - (3) Deferrable "bottom half" completes service for prior interrupt and wakes waiting process(es) – Usually runs w/ interrupts enabled
  - (4) A woken process can then be scheduled to resume after blocking I/O request



Example: Linux

- Avoid undue impact of interrupt handling on CPU time for a running process
  - Execute a finite # of pending deferrable fns after top half execution (in "interrupt context")
    - Linux deferrable fns: softirqs and tasklets (bottom halves now deprecated)
    - Iterate through softirq handling a fixed number of times to avoid undue delay to processes but good responsiveness for interrupts (e.g., via network)
  - Defer subsequent bottom halves to threads
    - Awaken "ksoftirqd_CPUn" kernel thread

Linux Problems

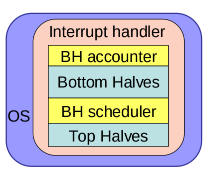
- A real-time or high-priority blocked process waiting on I/O may be unduly delayed by a deferred bottom half
  - Mismatch between bottom half priority and process
- Interrupt handling takes place in context of an arbitrary process
  - May lead to incorrect CPU time accounting
- Why not schedule bottom halves in accordance with priorities of processes affected by their execution?
- For fairness and predictability: charge CPU time of interrupt handling to affected process(es), where possible

Process-Aware Interrupt Handling

- Not all interrupts associated with specific processes
  - e.g., timer interrupt to update system clock tick, IPIs…
  - Not necessarily a problem if we can account for such costs in execution time of tasks e.g., during scheduling
- I/O requests via syscalls (e.g., read/write) associate a process with a device that may generate an interrupt
  - For this class of interrupts we assign process priorities to bottom half (deferrable) interrupt handling
  - Allow top halves to run with immediate effect but consider dependency between bottom halves and processes

Bottom Half Scheduling / Accounting

- Modify Linux kernel to include interrupt accounting
  - TSC measurements on bottom halves
  - Determine target process for interrupt processing and update system time accordingly
- BH/interrupt scheduler immediately between *do_irq()* and *do_softirq()*
  - Predict target process associated with interrupt and set BH priority accordingly



Interrupt Accounting Algorithm

- Measure the average execution time of a bottom half (BH) across multiple BH executions
  - On x86 use rdtsc since time granularity typically < 1 clock tick
- Measure total interrupts processed and # processed for each process in 1 clock tick
- Adjust system CPU time for processes due to mischarged interrupt costs
- For simplicity, focus on interrupts for one device type (e.g., NIC) but idea applies to all I/O devices

System CPU Time Compensation (1/2)

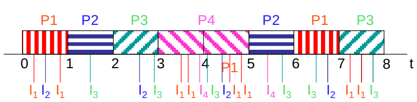
- **N(t)** - integer # interrupts whose total BH execution time = 1 clock tick (or *jiffy*)
  - Actually use an Exponentially-Weighted Moving Avg for N(t), N'(t)
  - $N'(t) = (1-\gamma)N'(t-1) + \gamma N(t) \mid 0 < \gamma < 1$
- **m(t)** - # interrupts processed in last clock tick
- $\mathbf{x_k(t)}$ - # unaccounted interrupts for process $\mathbf{P_k}$
- Let $P_i(t)$ be active at time t
  - $m(t) - x_i(t)$ (if +ve) is # interrupts overcharged to P_i

System CPU Time Compensation (2/2)

- At each clock tick (do_timer) update accounting info as follows:

```
x_i(t) = x_i(t) – m(t);     // current # under-charged if +ve
sign = sign of (x_i(t));
while (abs(x_i(t)) >= N(t)) // update integer # of jiffies
    ▪ system_time(P_i) += 1*sign;
    ▪ timeslice(P_i) -= 1*sign;
    ▪ x_i(t) = x_i(t) – N(t);
m(t) = 0;
```

Example: System CPU Time Compensation



$x_1(1)$: -3 + 2 = -1, $x_2(2)$: -1 + 1= 0,
$x_3(3)$: -2 + 2 = 0, $x_4(4)$: -3 + 1 =-2,
$x_4(5)$: -2 + -4+ 0= -6, $x_2(6)$: 0 + -2 + 2 = 0,
$x_1(7)$: -1 + -2+ 4= 1, $x_3(8)$: 0 + -3 + 4 = 1,

Interrupt Scheduling Algorithm

- (1) Find candidates associated with interrupt on device, D
  - In top half can determine D
  - A blocked process waiting on D may be associated with the interrupt
  - We require I/O requests to register process ID and priorities with corresponding device
- (2) Predicting process associated with interrupt on D
  - At end of top half select highest priority ($\rho_{max(D)}$) from processes waiting on D
  - Use a heap structure for waiting processes
- (3) Compare priority of BH with running process
  - If ($\rho_{max(D)} = \rho_{BH}$) > $\rho_{current}$ run BH else process

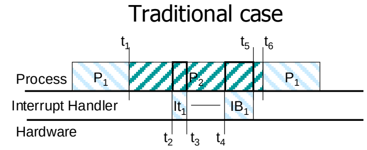
Interrupt Scheduling Observations

- No need for ksoftirqd_CPUn
  - Run interrupt scheduler at time of process scheduling
  - If pending BH highest prio run in context of current process, else do switch to highest prio process
- Setting prio of BH (ρ_{BH}) to highest process prio ($\rho_{max(D)}$) for device D
  - Rationale: no worse than current approach of always preferring BH (at least for finite occurrences) over process
    - Simple priority scheme can provide better predictability for more important processes

Example: Interrupt Scheduling (1/3)

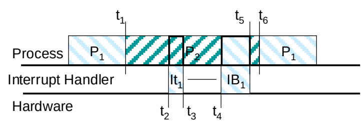
- t_1: P_1 issues I/O request and blocks, allowing P_2 to run
- t_2: top half interrupt processing for P_1 in P_2's context
- t_3: top half completes
- t_4-t_5: bottom half runs
- t_6: P_1 wakes up and runs

Traditional case



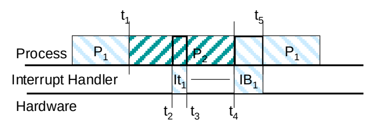
Example: Interrupt Scheduling (2/3)

- Previous case: top and bottom half processing charged to P_2
- Our approach: correctly charge bottom half processing to P_1



Example: Interrupt Scheduling (3/3)

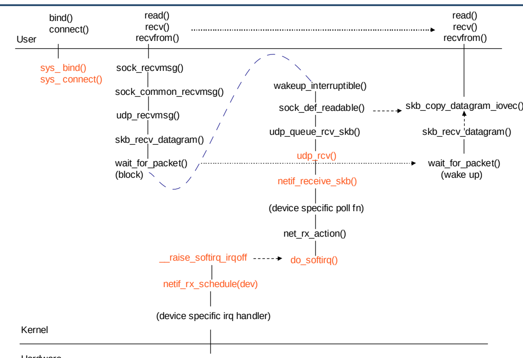
- If P_2 is higher priority than P_1, let P_2 finish and defer the BH for P_1



System Implementation

- Implemented scheduling & accounting framework on top of existing Linux bottom half (specifically, softirq) mechanism
- Focus on network packet reception (NET_RX_SOFTIRQ)
- Read TSC for each net_rx_action call as part of softirq
- Determine # pkts received in one clock tick
- udp_rcv() identifies proper socket/process for arriving pkt(s)
- Modify account_system_time() to compensate processes
- Interrupt scheduling code implemented in do_softirq()
  - Before call to softirq handler (e.g., net_rx_action())

Linux Control Path for UDP Packet Reception



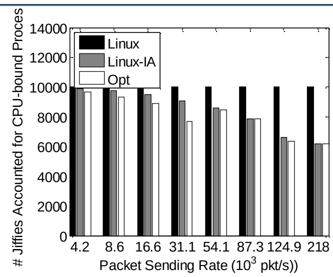
Experiments

- UDP server receives pkts on designated port
  - CPU-bound process also active on server to observe effect of interrupt handling due to pkt processing
- UDP client sends pkts to server at adjustable rates
- Machines have 2.4GHz Pentium IV uniprocessors and 1.2GB RAM each
- Gigabit Ethernet connectivity
- Linux 2.6.14 with 100Hz timer resolution
- Compare base 2.6.14 kernel w/ our patched kernel running accounting (Linux-IA) and scheduling (Linux-ISA) code

Accounting Accuracy

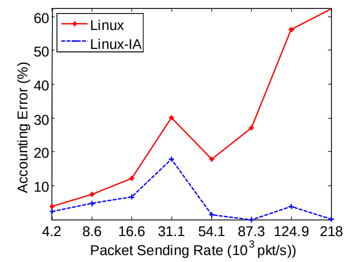
- CPU-bound process set to real-time priority 50 in SCHED_FIFO class
  - Repeatedly runs for 100 secs & then sleeps 10 secs
- UDP server process non-real-time
- UDP client sends 512 byte pkts to server at constant rate
- Read /proc/pid/stat to measure user/system time

Accounting Accuracy Results



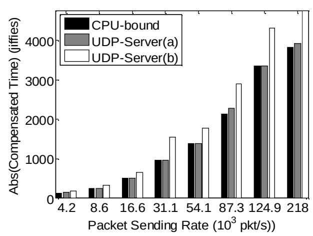
- Optimal case (Opt) is total user/system-level CPU time that should be charged to CPU-bound process discounting unrelated interrupt processing
- Linux-IA close to optimal but original Linux miss-charges all interrupt processing

Ratio of Accounting Error to Optimal



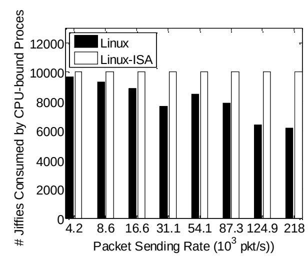
- Error as high as 60% in Linux
- Less than 20% and more often less than 5% using Linux-IA

Absolute Compensated Time



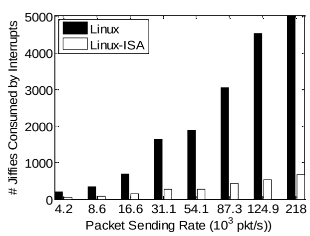
- UDP-Server(a) – charged time for interrupts over 100s of each 110s period of CPU bound process
- UDP-Server(b) – charged time over full 110s period
- CPU-bound – system service time deducted from CPU-bound process

Bottom Half Scheduling Effects



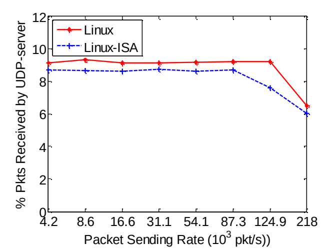
- Linux – CPU-bound process affected by interrupts
- Linux-ISA – defer bottom-half interrupt processing until (higher priority) real-time CPU-bound process sleeps

Time Consumed by Interrupts (every 110s)



- Time consumed by CPU-server every 110s handling interrupts
- Linux-ISA – bottom half handling deferred to interval [100-110s]
- Linux – bottom half processing not deferred

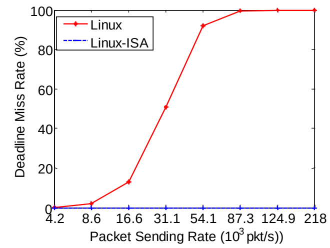
UDP-Server Packet Reception Rate



Bursty Packet Transmission Experiments

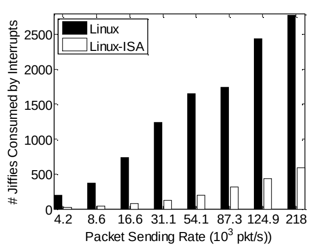
- UDP-client sends bursts of pkts w/ avg geometric sizes of 5000 pkts
  - Different avg exponential burst inter-arrival times
- CPU-bound process is periodic w/ C=0.95s and T=1.0s
  - Runs for 100s as before
- Deadline at end of each 1s period

Deadline Miss Rate

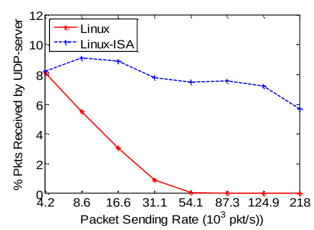


- Linux-ISA – no missed deadlines for CPU-bound process
- Bottom half interrupt handling deferred until CPU-bound process completes each period

Interrupt Overheads (100s interval)



Performance of UDP-server



- CPU-bound process cannot finish executing in 1s period when interrupt overheads are high
  - Always competes for CPU cycles, starving lower priority UDP-server
- Linux-ISA guarantees “slack” time usage for UDP-server

Conclusions and Future Work

- Explore dependency between processes and interrupts
- Focus on bottom half scheduling and accounting
  - Compensate processes for time spent in bottom halves
  - Charge correct processes benefiting from interrupts
- Unify the scheduling of bottom half interrupt handlers w/ processes
  - Improve predictability of real-time tasks while avoiding undue interrupt-handling overheads
  - Consequently, benefit non-real-time tasks also!
- Future? Better predictors of process(es) associated w/ interrupts for scheduling purposes
- Interrupt management on multi-processors/cores