Solving Real-World Real-Time Scheduling Problems With RT_PREEMPT and Deadline-Based Scheduler

Xi Wang

Broadcom Corporation

Questions, Comments:
xiwang@broadcom.com
peknap@yahoo.com

Introduction

- **Higher degree of processor sharing in embedded systems**
  - Before: Several task specific processors with dedicated tasks, often running without OS
  - Now: Multiple tasks running on powerful application processors, including some DSP tasks
    - E.g. making a video call from a mobile phone
    - Common tasks include media tasks for audio/video streams, network traffic, wireless and security, etc.
- **Expectations of real-time scheduling have become higher**
  - Multiple low latency tasks with significant CPU time usage is more challenging and more interesting
  - Maintaining low latency and reaching high CPU utilization are often conflicting goals for traditional strict priority or rate monotonic scheduling

Introduction

- **This talk**
  - Part 1: Meeting real-time requirements of both VoIP and packet forwarding tasks by modifying Linux priority model with RT_PREEMPT
  - Part 2: Take it further, how to make everyone happy in a three class scheduling problem (not yet a solution, but sharing thoughts)

Introduction

- **System in background**
  - Home gateway device with three major classes of tasks: Packet forwarding, VoIP and Applications such as UI
    - VoIP: Real-time
    - Packet forwarding: Even more so – low latency tolerance due to limited Rx buffers in the embedded device
- **Warning: UI application can be hard read-time**
  - User's patience runs out → computers destroyed
    - If in doubt, search Internet

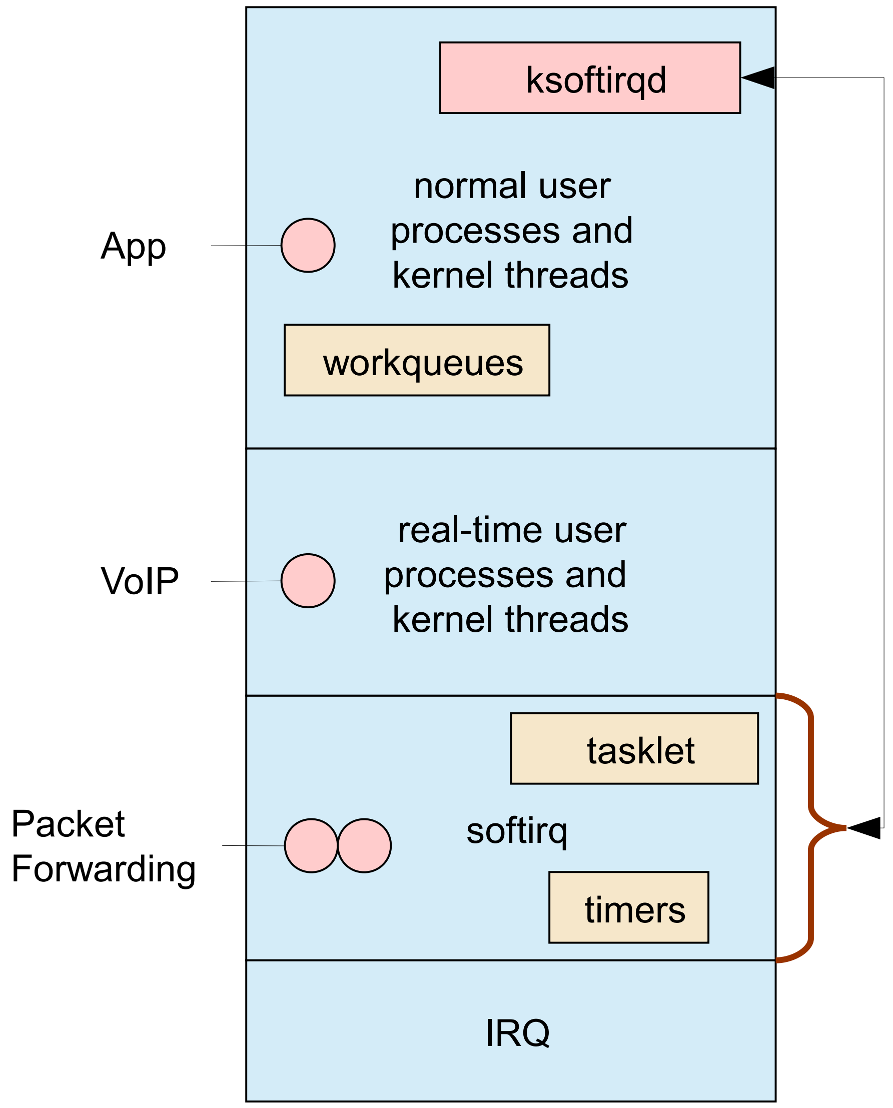
VoIP and Packet Forwarding

- **Problem for Part 1**
  - When system is loaded with heavy network traffic,VoIP tasks cannot get enough CPU cycles

- **Incomplete solution**
  - Make voice threads run at real-time class
    - Despite that they can preempt other normal priority kernel threads at will, network traffic still has higher priority

- **Complete understanding the behavior of softirq is the key to solve the problem**

Native Linux Priority Model


ksoftirqd
App
normal user processes and kernel threads
workqueues
VoIP
real-time user processes and kernel threads
tasklet
Packet Forwarding
softirq
timers
IRQ

- **Variable priority of softirq**
  - Very high when lightly loaded – preempt everything except hard irq
  - Low, running in ksoftirqd kernel thread when oversubscribed
- **The goal is to provide low latency to softirq tasks without starving other tasks**
  - The downside is that there is no consistent priority model

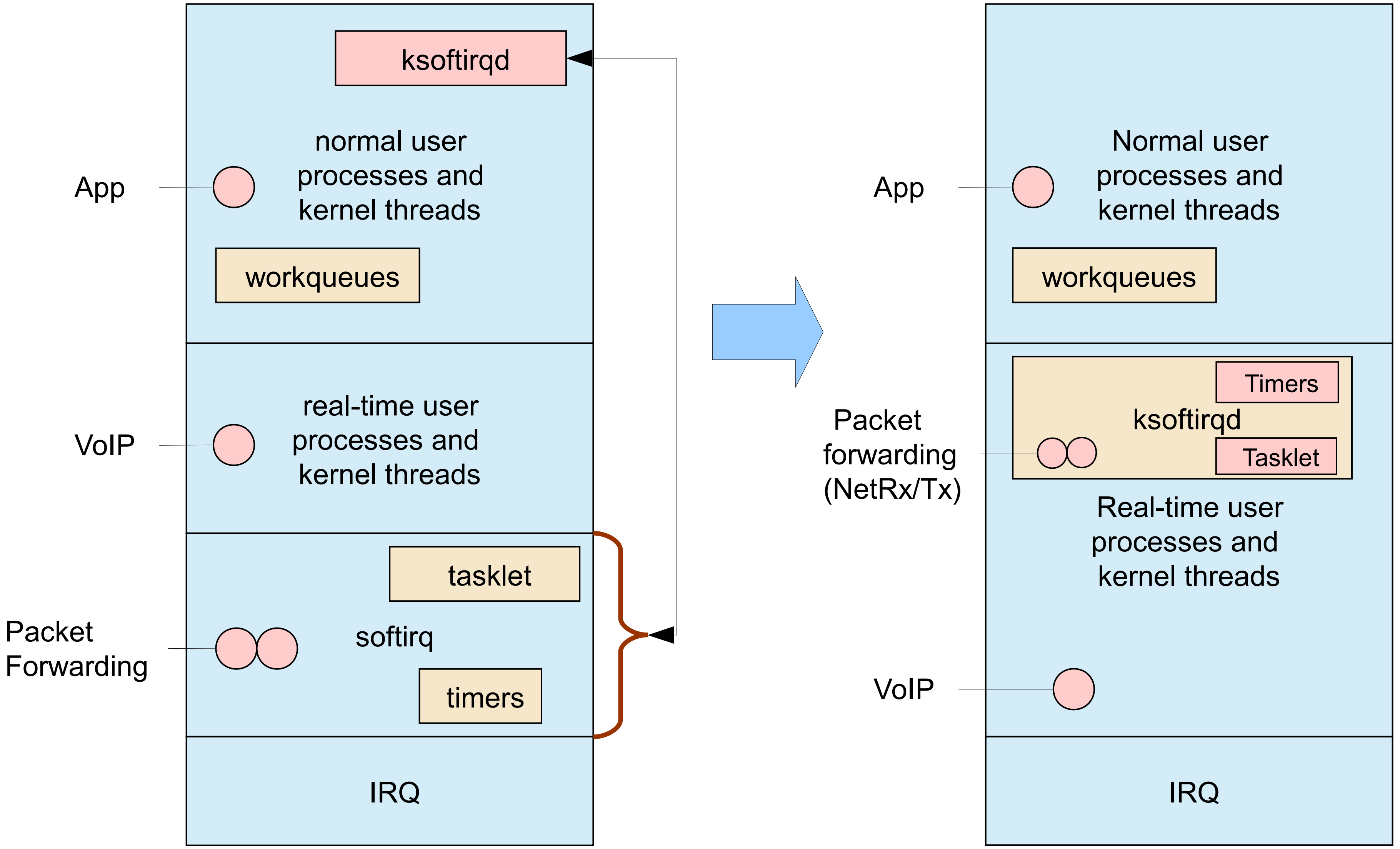
Undesirable Default Priority Model

- **In default Linux priority model, no kernel thread/user process can have higher strict priority over network packet processing**
  - softirq is thread like but not under the control of process scheduler
  - Moving voice processing to softirq is not practical, nor does it give voice tasks the highest priority
    - It could run in ksoftirqd too
  - Moving to hard irq might work as an ugly solution, but nobody would like the side effects
  - Layered OS is possible, but overkill
- **The problem cannot be solved without changing this model**

Changing the Priority Model

- **The solution: Always run softirq in kernel thread context**
  - Now a feature in RT_PREEMPT
    - Called “preempt softirq” in Kconfig, but better described as running softirq in thread context
  - This code is less-known and small, but important
    - First developed by Ingo Monlar's as an independent patch. Later included in RT_PREEMPT. Some of the RT_PREEMPT features are mainlined, but not this one (as of 2.6.38)
    - With this change, traditional softirq no longer exists– softirq tasks always run in ksoftirqd

Changing The Priority Model


ksoftirqd
App
normal user processes and kernel threads
workqueues
VoIP
real-time user processes and kernel threads
tasklet
Packet Forwarding
softirq
timers
IRQ
App
Normal user processes and kernel threads
workqueues
Timers
Packet forwarding (NetRx/Tx)
ksoftirqd
Tasklet
Real-time user processes and kernel threads
VoIP
IRQ

Under The New Priority Model

- **Moving softirq to a process context and using real-time scheduling classes made a working system**
  - VoIP tasks are always happy
    - Higher strict priority over everything except hard irq
  - Packet forwarding is mostly happy
    - Lower priority than VoIP but won't be affected by anything else

On softirq

- **My observations on softirq**
  - Original purpose is trying to do everything in a non-preemptive kernel
    - Defer hard irq processing
    - Enable high priority low-latency tasks, and avoid starving other tasks
    - Provide preemption when kernel preemption is not enabled/supported
  - Kernel process scheduling has included most of these features over the years
  - Cost of softirq
    - Priority jumps up and down
    - Having both Interrupt context and non-interrupt context can create problems
      - Inflexible binding of spin lock ↔ softirq, mutex ↔ process
  - Feature creep, complex protocols in net rx/tx are not necessarily irq related
- **Phase out softirq?**
  - This gets the author's vote, but Linux platforms are diverse, and it may be bad for other systems
    - At least packet forwarding is not worse, reporting that running Net Rx/Tx in process context resulted in similar network packet throughput
  - Keep hard irq, everything else in the process context under the control of process scheduler
    - Some special mechanisms and optimizations for ultra low latency tasks

Next Step

- **Make it even better – problem for Part 2**
- **No compromise, make everyone completely happy**
  - Meet strict performance requirements of both VoIP and packet forwarding tasks without one having the priority over the other
  - Provide latency guarantees to UI applications too, so users can enjoy a responsive UI

Next Step

- **Disclaimer: I would like to share some thoughts for which I haven't experimented**
  - Me and process scheduler code
    - Tried an early version of SCHED_DEADLINE, but didn't give it enough time to see results
    - Tried to write my own deadline based scheduler
      - Not necessarily to make it a final product, but to better understand the problem. Didn't find enough time to complete
    - Relation between this talk to SCHED_DEADLINE
      - Overlapping parts can be equivalent, need to do more investigations to find out all the similarities and differences
- **In next pages**
  - A bucket model from me
  - A proposed scheduling mechanism
  - Characteristics, problems and more features to add

Understanding Requirements

- **Real-time requirements cannot be described with simple priorities**
  - Max latency
    - Packet forwarding: 1 ms
      - Limited Rx buffers + wirespeed forwarding
    - VoIP: 10 ms class
      - Protocols, DMA
    - Application: 500 ms class
  - CPU time asked
    - Packet forwarding: 0%~100%
    - VoIP: 0%~50%
    - Application: 0%~100%
  - CPU time “priority”
    - Packet forwarding: Lower than VoIP, higher than application
    - VoIP: Gets as much as needed
    - Application: Shouldn't be starved (guarantee > 5%)

Making It Better

- **QoS process scheduling?**
  - Detailed requirements on multiple aspects are similar to network QoS
  - Both network QoS and process scheduling are related to resource allocation and queuing
    - CFS ~ Fair Queuing

QoS Scheduling

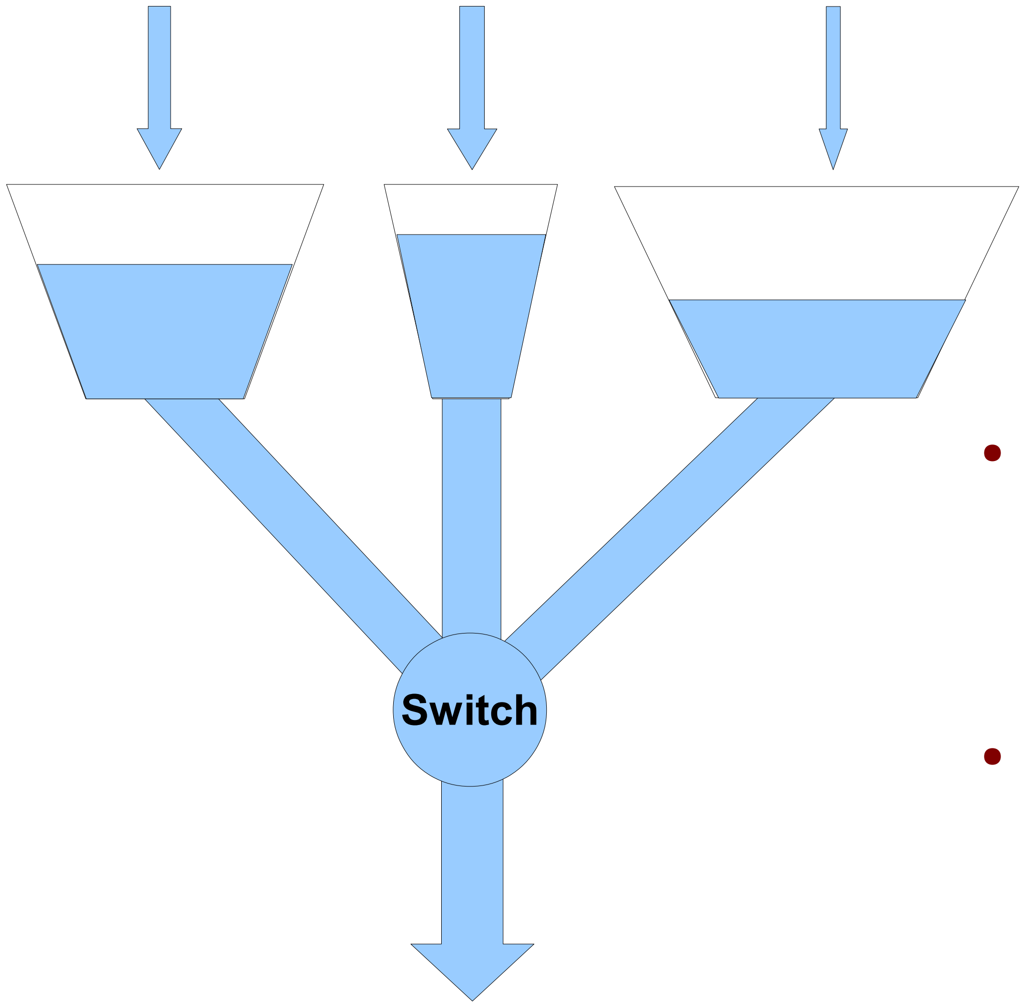
- **Review of existing mechanisms – making everyone happy is difficult**
  - Strict priority can starve tasks
    - Another side effect is that priority inversion leads to deadlocks
  - CFS provide good CPU time allocation, but no latency guarantee
  - Real-time group scheduling can be a solution, but with some drawbacks
    - Global scheduling period needs to be aligned with the most latency sensitive task, very frequent task switching possible
    - Fixed CPU time allocation
- **To solve the three-class QoS scheduling problem**
  - Deadline-based scheduling as the main component, complimented by some additional mechanisms

QoS Scheduling

- **Background on deadline-based scheduling**
  - For batch processing, clear model for Earliest Deadline First (EDF) scheduling
    - N tasks, each with known deadline
    - Always pick the task with earliest deadline when rescheduling
    - Optimal – guarantee that every job meets its deadline if schedulable
  - Extended to dynamic scheduling with recurrence
    - Regenerate each task at a certain period, and run the classical EDF in each period
      - Alternative methods may exist
  - See SCHED_DEADLINE and related papers
    - http://gitorious.org/sched_deadline/pages/Home
- **In this talk we use a continuous bucket model instead**
  - Turns out things can work without scheduling period

Bucket Model

Each task can be described with inflow rate and bucket size



CPU can drain any bucket at anytime, but only one at a given moment

- **It's simple**
  - Drain the right bucket at the right time, do not let any bucket to overflow
  - Modeled after producer → buffer → consumer problems
    - Can be reversed to prevent underflow for media applications
- **Continuous model**
  - No scheduling period or task recurrence period, tasks as incoming streams
- **Background**
  - Borrow concept from network QoS
  - Generic model, not limited to deadline scheduling

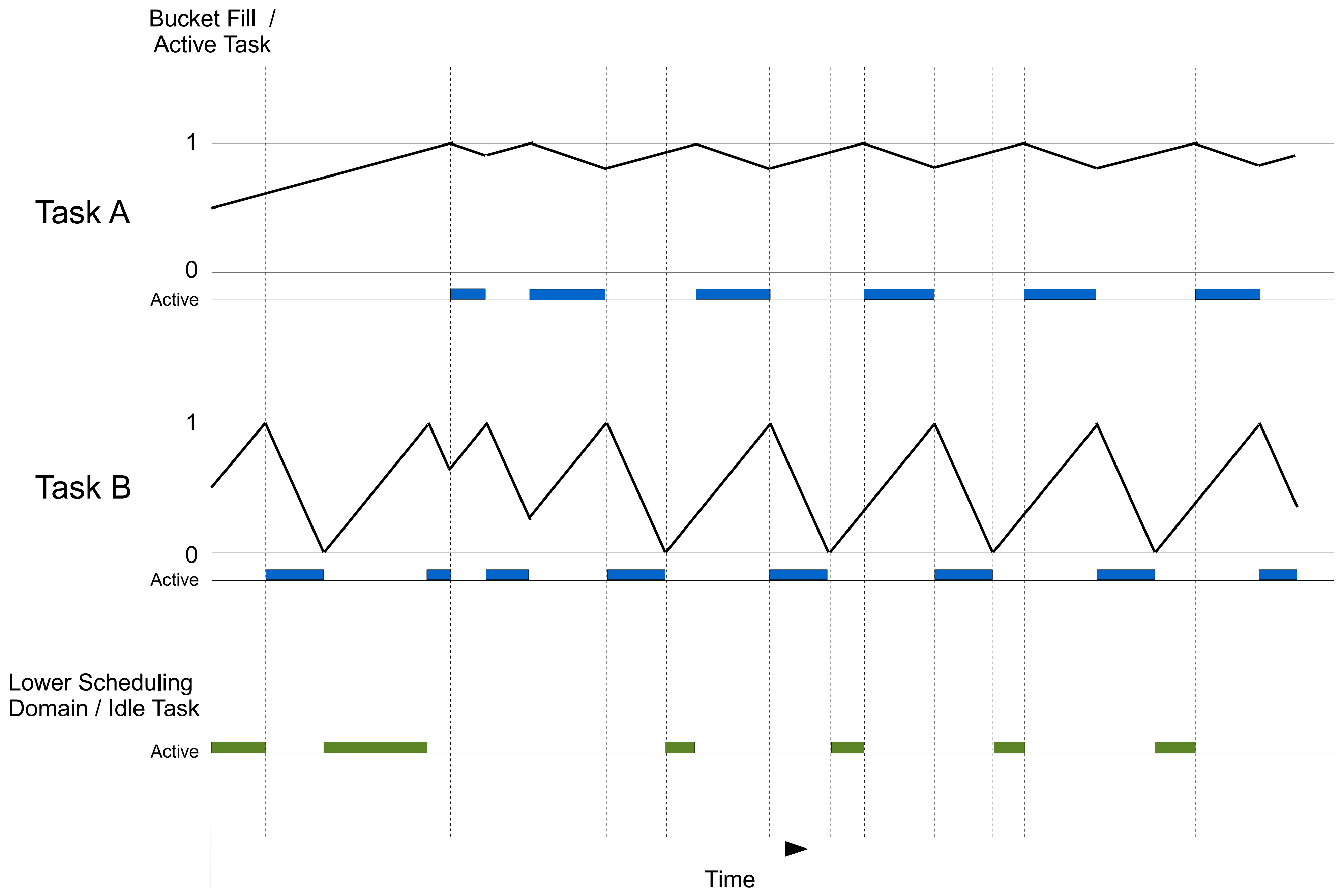
QoS Scheduling

- **Mapping tasks into bucket model**
  - The first parameter, deadline or buffer overflow time often already exists in real world
    - Packet forwarding: ~1 ms
    - VoIP: ~10 ms
    - UI Application: ~500 ms (patience runs out = buffer overflow)
  - Two independent parameters
    - Assuming constant inflow rate and bucket size
    - Bucket Size, Inflow Rate → Deadline
    - Inflow Rate, Drain Rate → CPU Time Allocation
    - Only two independent parameters in this model, e.g. bucket size can be normalized out. I use
      - Bucket fill time (latency)
      - Bucket drain time (bandwidth, determines CPU time allocation when combined with bucket fill time)
- **Classful task scheduling**
  - No need to map deadline parameters to every task, deadline scheduling should happen at task group level

Deadline Based Scheduling Algorithm

- **Have bucket model, will schedule**
- **When to reschedule is important**
  - Multiple valid solutions to meet the goal
  - Cannot always run earliest deadline first in a continuous model – results in infinitely high task switching frequency
- **A straightforward first step design with lazy switching**
  - Mechanism A
    - Triggered: When one of the inactive task's bucket is full – deadline reached
    - Action: Switch to that task
  - Mechanism B
    - Triggered: When current task's bucket is empty
      - Indicated by current finishes/blocks
      - As calculated by bucket model, but the task hasn't finished – timeslice overrun
    - Action: Switch to lower scheduling domains or idle task

Lazy Switching Bucket Scheduler Example



Discussion

- **Characteristics, good or bad**
  - Adaptive time slice
    - If anyone familiar with image processing, this should be comparable to dithering, half-tone pattern or error diffusion
  - Each task is protected with latency and CPU time guarantee, won't be affected by other tasks
  - Bad mode / pattern possible
    - When two classes with close or equal parameters and both of their bucket is about to overflow, there will be very frequent task switching / thrashing
    - Need to be prevented / mitigated with additional mechanisms
    - One possible solution is to look ahead to one or more future deadlines and switch before bucket is full
  - Overheads are higher than most schedulers – but we probably only need a handful of QoS scheduling classes in a given system

Discussion

- **Features to add**
  - SMP ignored in this talk, should be SMP aware
  - Soft real-time features
    - What was discussed before made fixed CPU time reservations, sometimes we want to consider both worst case and average case, where tasks do not use up all their CPU time allocations
      - Two sets of bucket parameters, one for worst case guarantee, one for more optimistic situation
      - Allow each task to have two taps to receive CPU time, a primary tap at deadline scheduling domain and a second tap at lower scheduling domain, e.g. CFS

Conclusion

- **A good time to experiment with process schedulers**
  - Task scheduling can cause a surprising number of problems in modern embedded systems
  - Don't be afraid to work on it. Task scheduling may be less up to date as you think – in actual OS or academia
    - Process scheduler is deeply buried in OS core and rarely gives trouble in desktop systems
    - Not very easy to experiment and do research on
      - Many ideas should have been discussed, but not necessarily connected to real OS
  - With pluggable scheduler design in newer Linux kernels, it is now relatively easy to experiment

Conclusion

- **Do we really need more scheduling mechanisms?**
  - More challenging for embedded systems
    - More real-time requirements, less powerful processors, one-size-fits-all may not be enough
  - Less challenging for embedded systems
    - Scalability is not a must
    - OK to be mission specific
      - Useful even if not mainlined
  - Answer is probably yes