

The PREEMPT_RT Approach To Real Time

Wu Zhangjin / Falcon
wuzhangjin@gmail.com

泰晓科技| TinyLab.org
http://tinylab.org

March 21, 2014



Overview

1. Introduction
2. What is ‘Real Time’
3. Latency Components
4. PREEMPT_RT approach
5. PREEMPT_RT setup
6. PREEMPT_RT Porting
7. Latency Testing
8. Latency Monitoring
9. Latency Debugging
10. Realtime application development
11. References



PREEMPT_RT Project

- From: 2006
- Lanuched by: Ingo Molnar
- Latest versions : v2.6.33.9-rt31, v3.0.1-rt11, 3.12.13-rt21
- Current maintainer: Thomas Gleixner, Peter Zijlstra
- Main Contributors: Thomas Gleixner(Hrtimers), Ingo Molnar(CFS, Threaded interrupt, Perf, Sleeping spinlock), Steven Rostedt(Ftrace, PI-mutexes, Lockdep), Paul E. McKenney(RCU), Peter Zijlstra(BKL)
- Web Site: http://rt.wiki.kernel.org
- Download: http://www.kernel.org/pub/linux/kernel/projects/rt/
- Mailing lists: linux-kernel <linux-kernel@vger.kernel.org>, linux-rt-users <linux-rt-users@vger.kernel.org>
- Online QA Farm: https://www.osadl.org/QA-Farm-Realtime.qa-farm-about.0.html



Real Time System

A real-time system is one in which

- The correctness of the computations
  - not only depends upon the logical correctness of the computation
  - but also upon the time at which the result is produced
- If the timing constraints of the system are not met, system failure is said to have occurred.



Application 1 of Real Time System

Figure: Airbag of Car



Application 2 of Real Time System

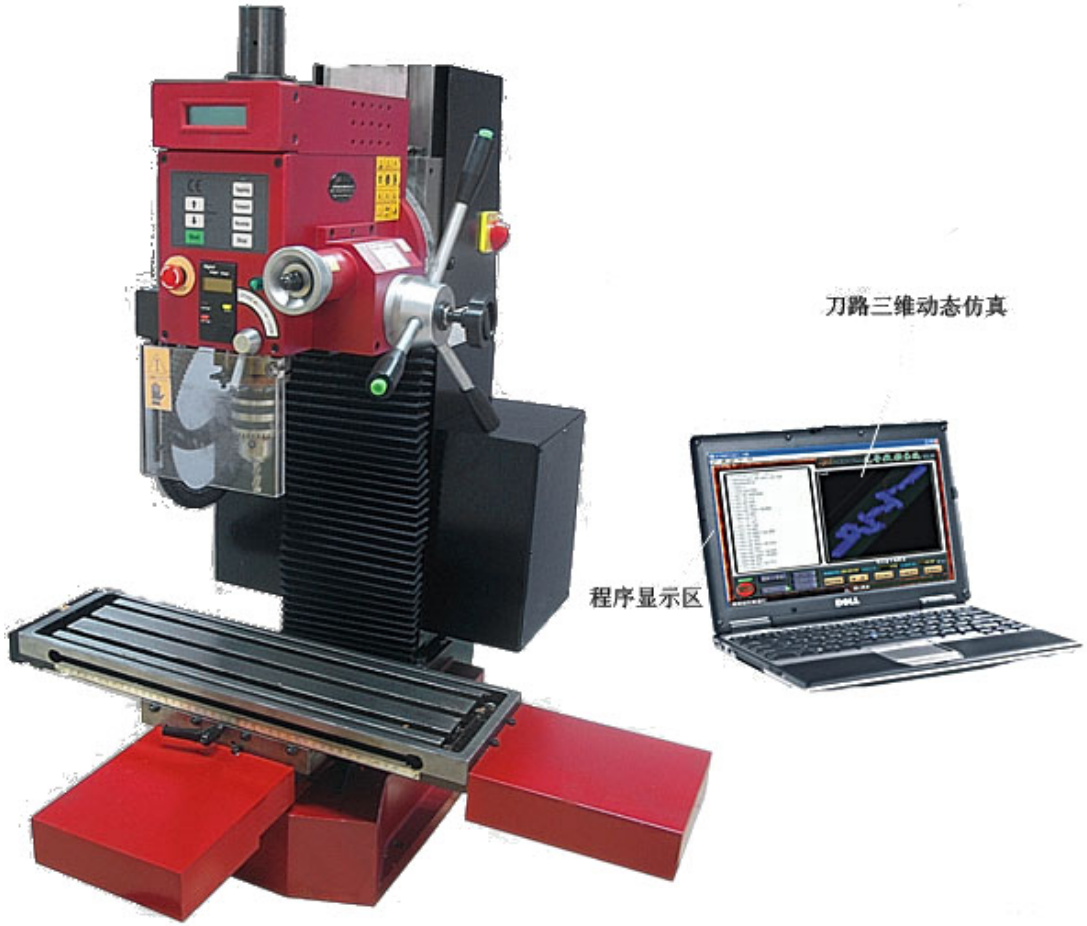

Figure: Digital Machine



Real Time Operating System

- Realtime in operating systems
  - 'The ability of the operating system to provide a required level of service in a bounded response time'
- Response time/Latency
  - 'The time that elapses between a stimulus and the response to it'
- Bounded
  - 'real-time is not real-fast'
  - real-time is about 'time determinism'
- Jitter
  - Jitter is the amount of variation in latency/response time, It reflects the 'Bounded' range
- Worst-Case Latency
  - 'Realtime is about providing guaranteed worst case latencies for this reaction time'



Application 1 of Real Time Operating System

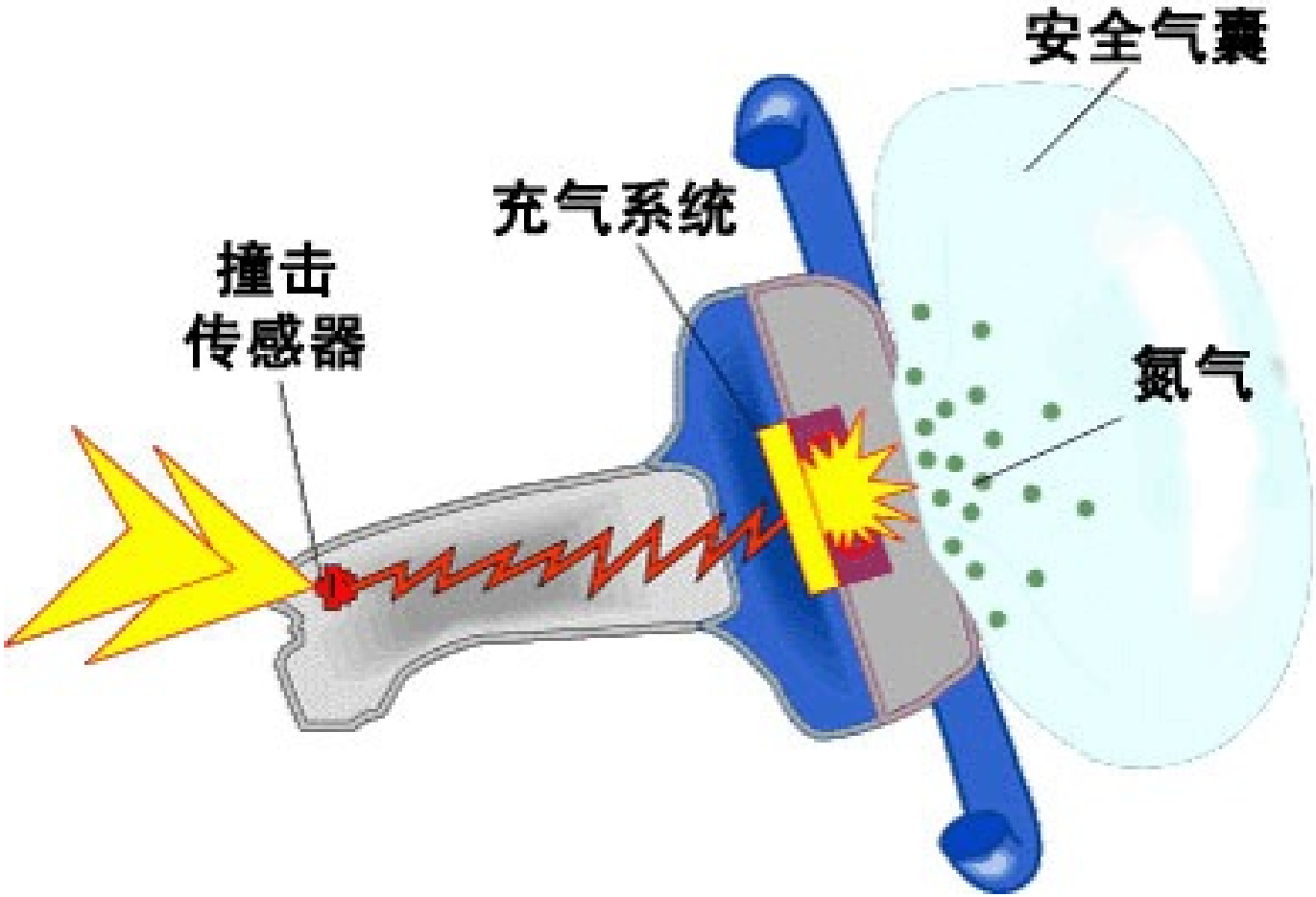

Figure: Control System for Airbag of Car



Application 2 of Real Time Operating System

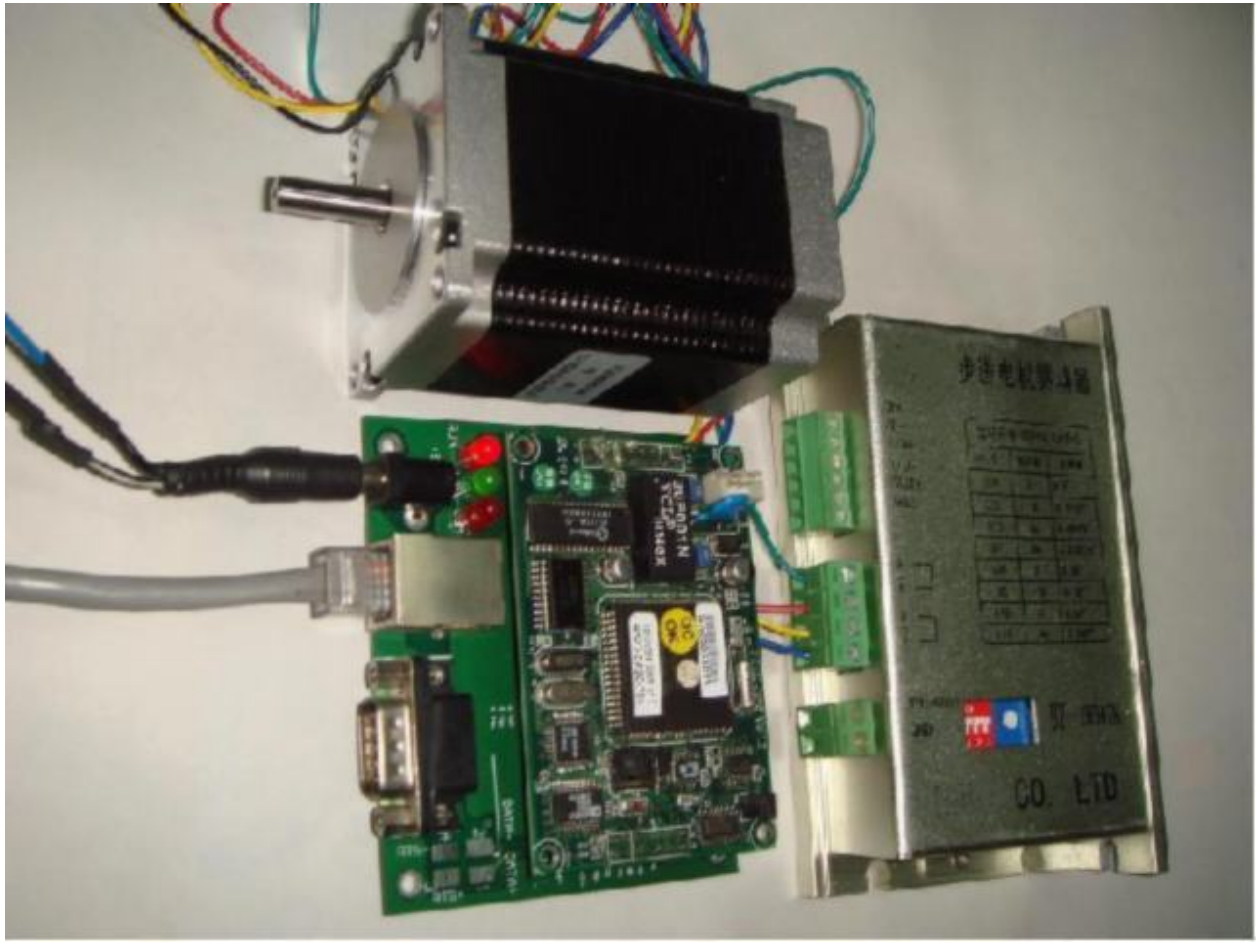

Figure: Control System for Digital Machine



Asynchronous Events

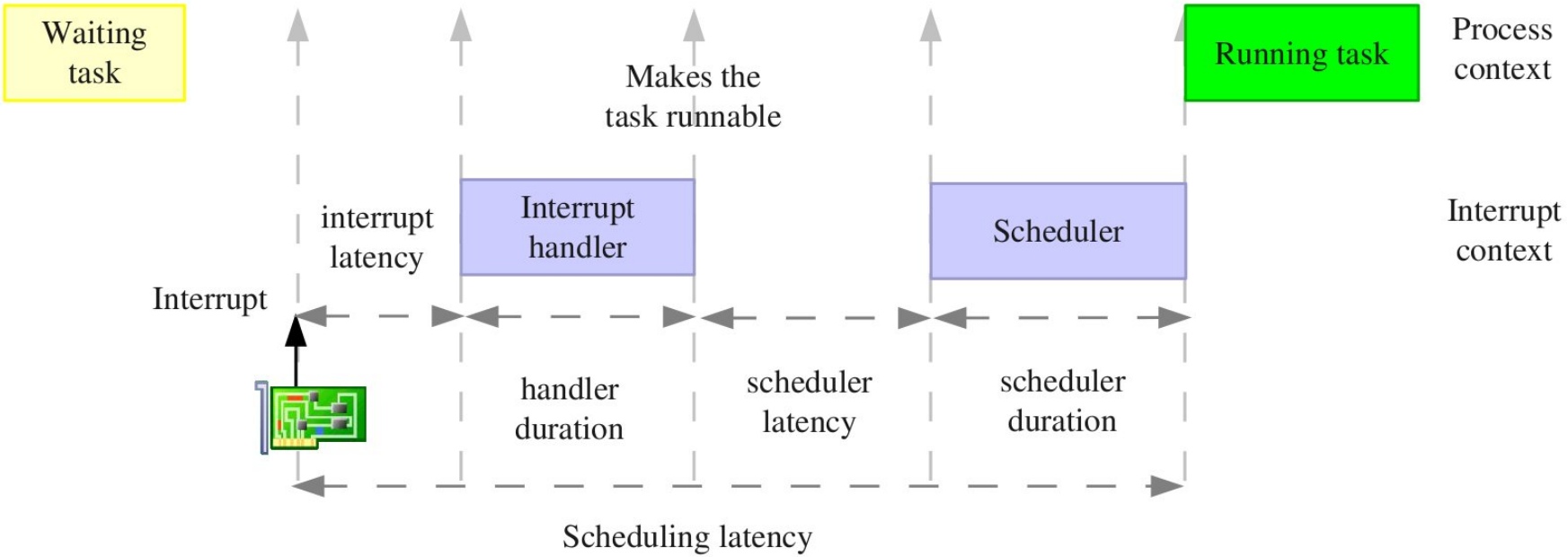

Figure: Kernel Latency of Asynchronous Events

kernel latency = interrupt latency + handler duration + scheduler latency + scheduler duration



Interrupt Latency

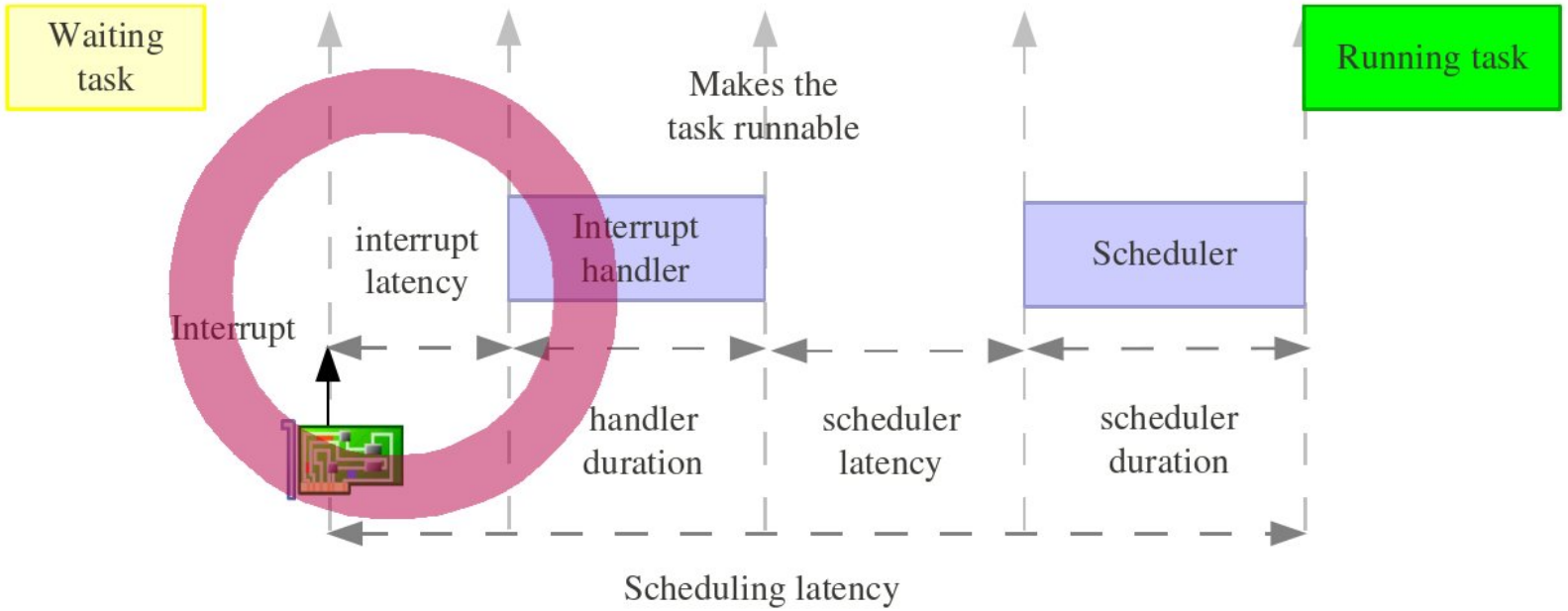

Figure: Interrupt Latency

Time elapsed before executing the interrupt handler.

Source of Interrupt Latency

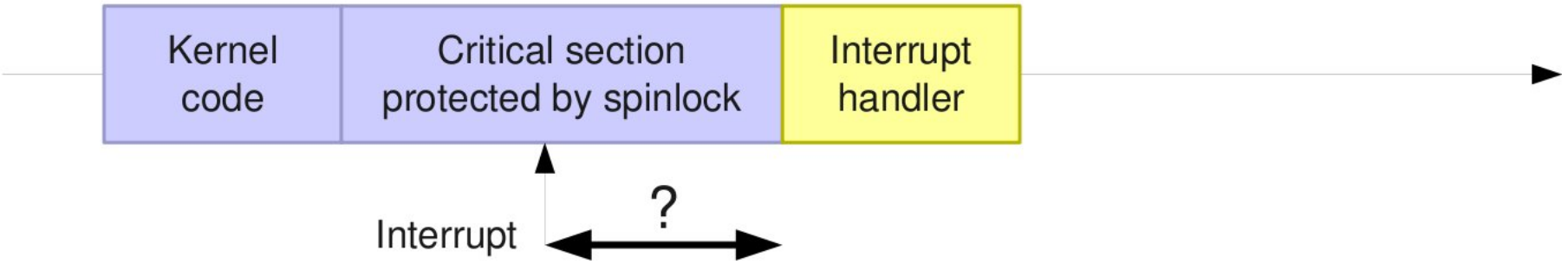

Figure: Critical Sections may disable interrupt

- Any sections(e.g. critial sections protected by spinlocks to prevent concurrency between process context and interrupt context) disables interrupt may delay the beginning of interrupt handler.
- Interrupts shared among different priority tasks may delay the begining of the high priority task's interrupt handler



Interrupt Handling

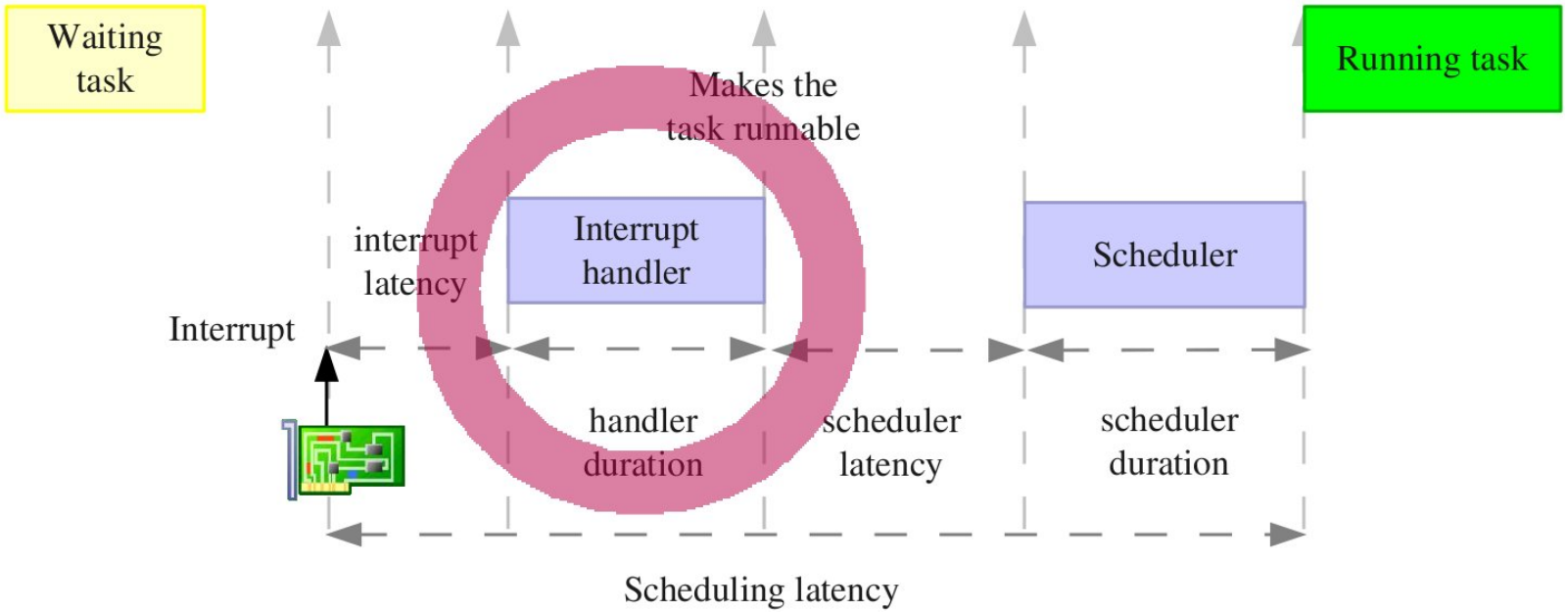

Figure: Interrupt duration

Time taken to execute the interrupt handler.

Interrupt Handler Implementation

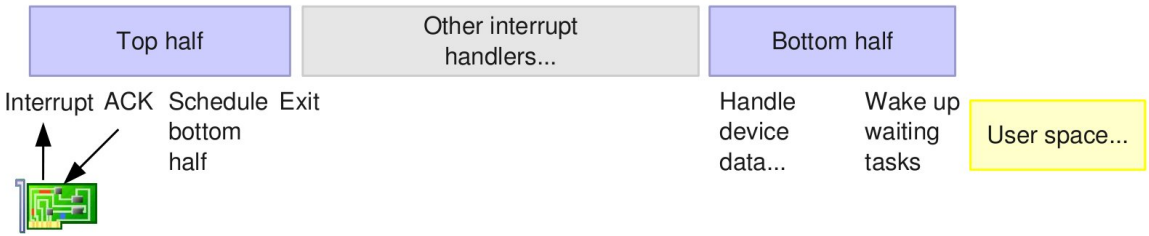

Figure: Interrupt handler = top half + bottom half

- In Linux, many interrupt handlers are split in two parts
  - A top-half, started by the CPU as soon as interrupt are enabled. It runs with the interrupt line disabled and is supposed to complete as quickly as possible.
  - A bottom-half, scheduled by the top-half, which starts after all pending top-half have completed their execution.
- Therefore, for real-time critical interrupts, bottom-half shouldn't be used: their execution is delayed by all other interrupts in the system.

Interrupt Handler Inversion

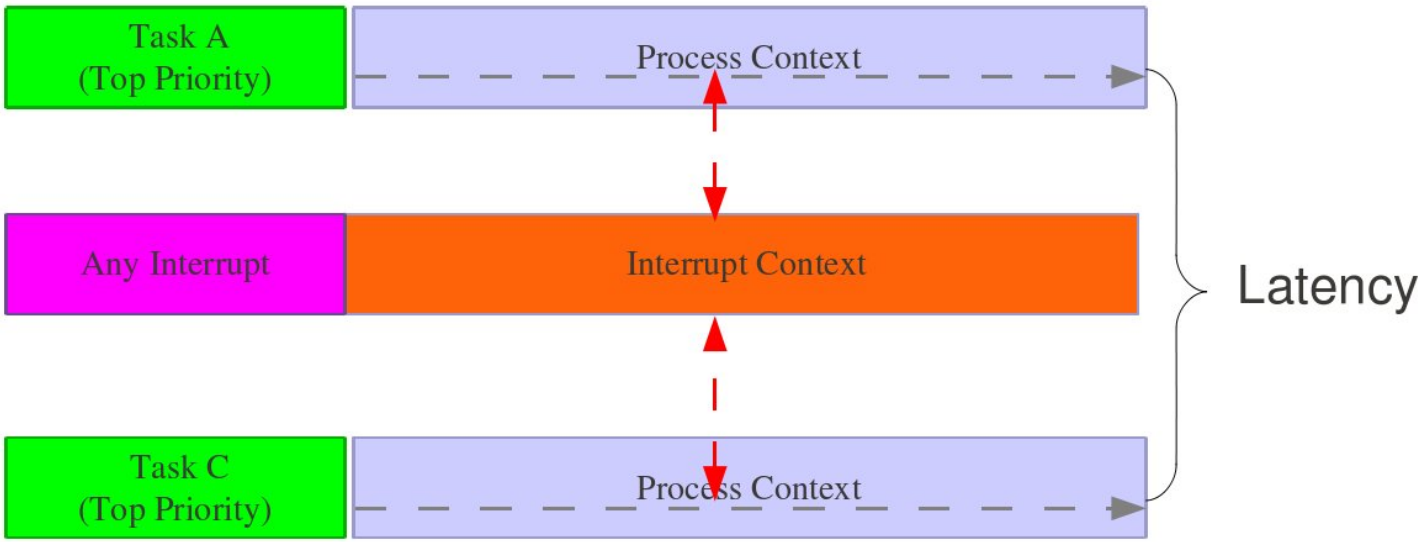

Figure: Kernel Latency of Interrupt Handler Inversion

In Linux, interrupt handlers are executed directly by the CPU interrupt mechanisms, and not under control of the Linux scheduler. Therefore, all interrupt handlers have an higher priority than all tasks running on the system.



Scheduler Latency

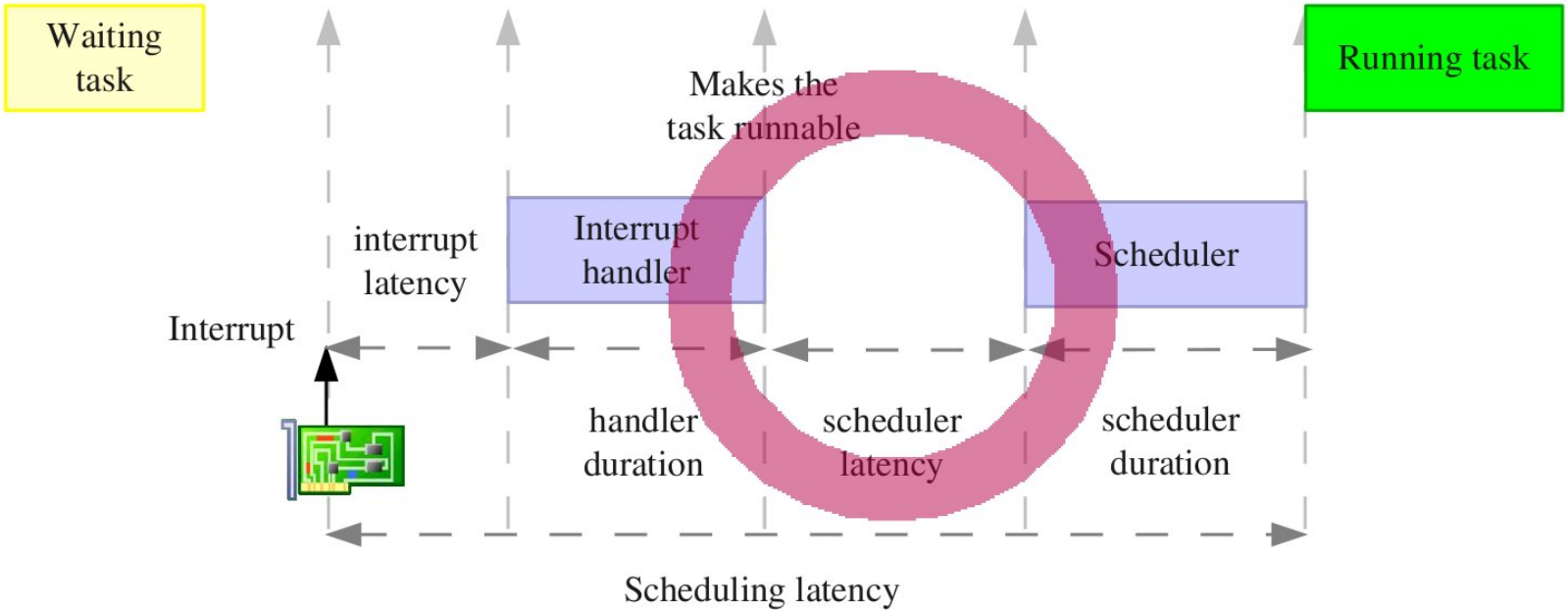

Figure: Scheduler Latency or Wakeup Latency

Time elapsed before executing the scheduler.



Kernel Preemption

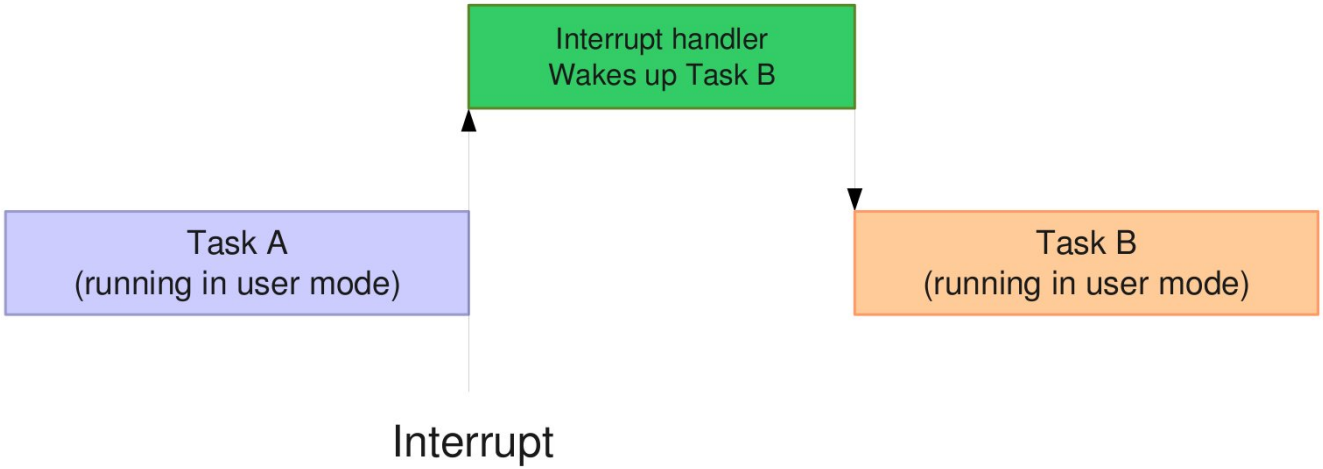

Figure: Preemption

- The Linux kernel is a preemptive operating system
- When a task runs in userspace mode and gets interrupted by an interruption, if the interrupt handler wakes up another task, this task can be scheduled as soon as we return from the interrupt handler.

No Kernel Preemption

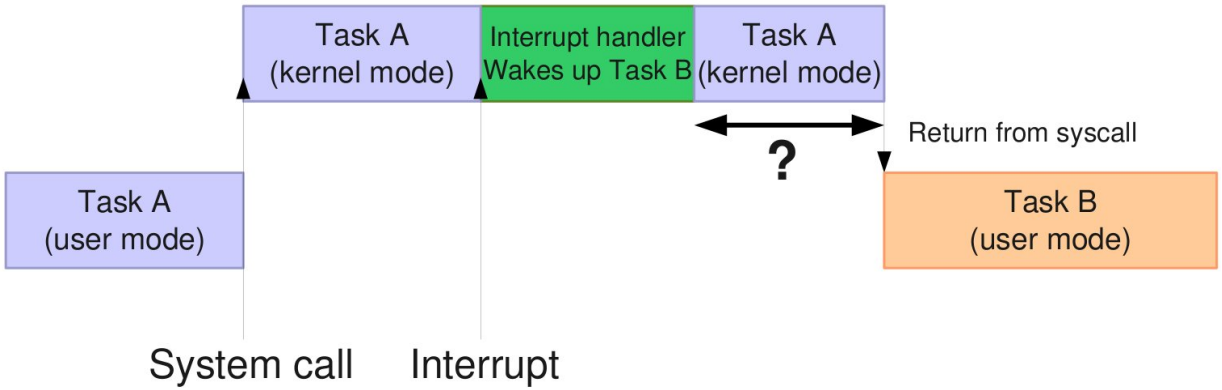

Figure: No Kernel Preemption

- However, when the interrupt comes while the task is executing a system call, this system call has to finish before another task can be scheduled.
- By default, the Linux kernel does not do kernel preemption.
- This means that the time before which the scheduler will be called to schedule another task is unbounded.

Source of Scheduler Latency

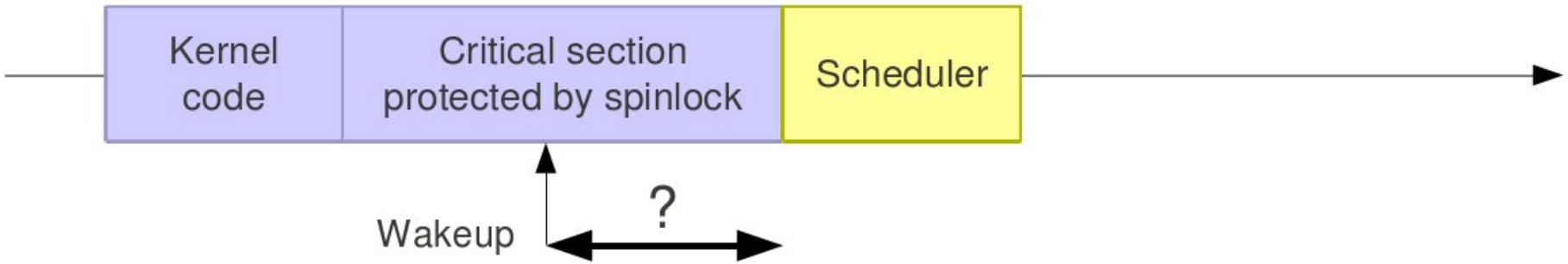

Figure: Critical sections may disable preemption

- Some critical sections(e.g. protected by spinlocks) disable preemption.
- Big source code section(e.g. while, for loops) which doesn't call schedule() obviously.



Scheduler Duration

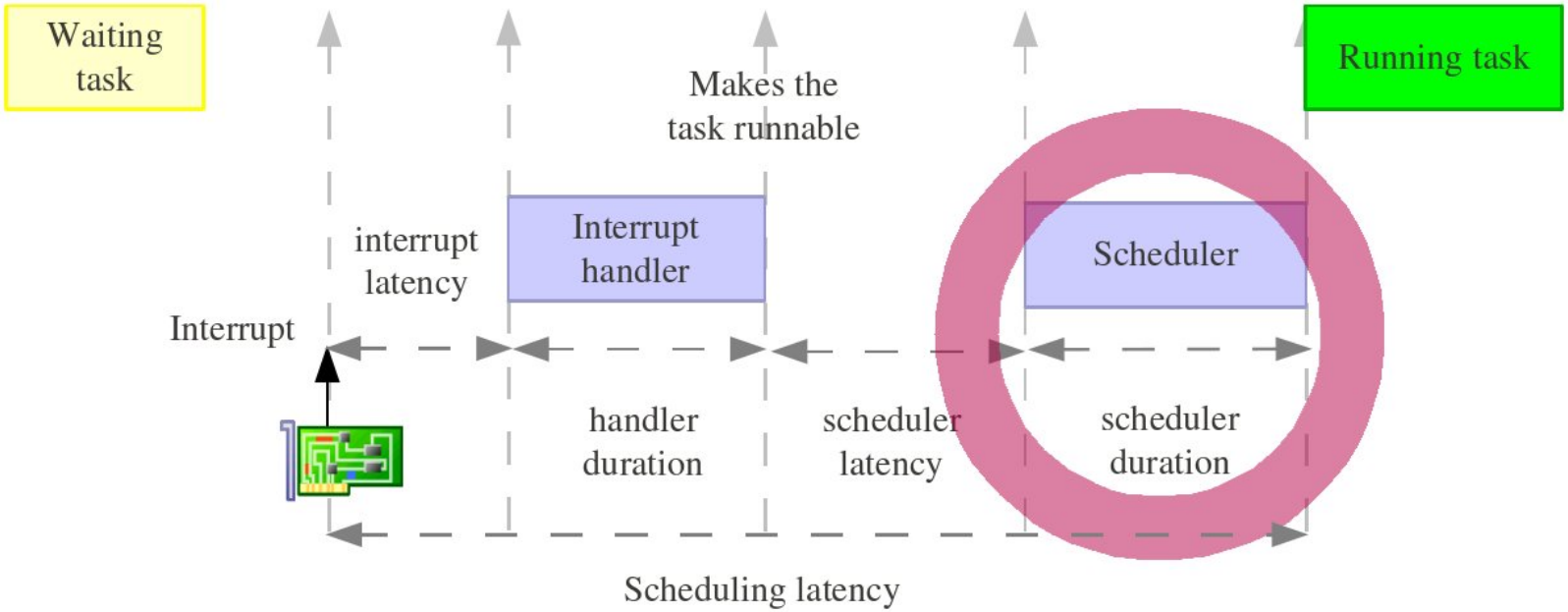

Figure: Scheduler Duration

Time taken to execute the scheduler and switch to the new task.

Time costed by Scheduler Duration

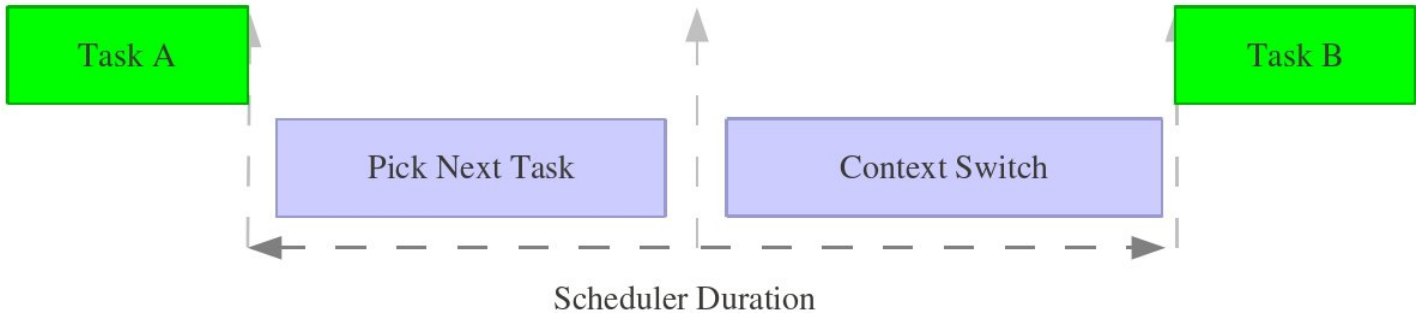

Figure: Scheduler Duration

- Pick next task: depends on scheduling policy
  - Time-sharing scheduling or priority based scheduling
- Switch to next task
  - The time spends on context switch, may include mm switch and cpu registers/stack switch.



Process Running

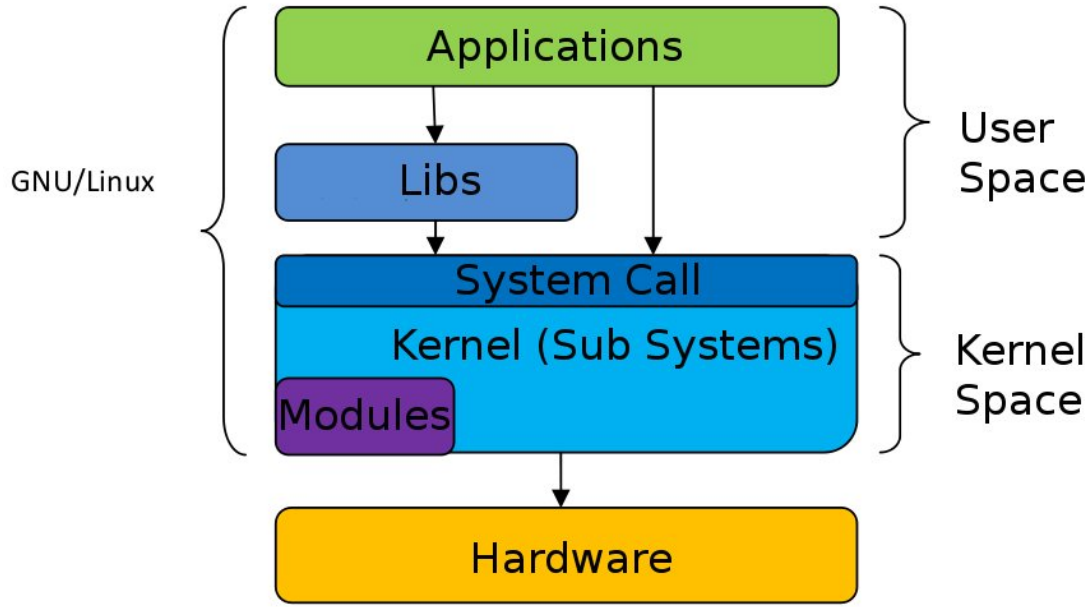

Figure: GNU/Linux System Architecture

The execution determinism is related to lots of parts, especially Page Fault, Branch Miss, Cache Miss and TLB miss.



Periodic Task

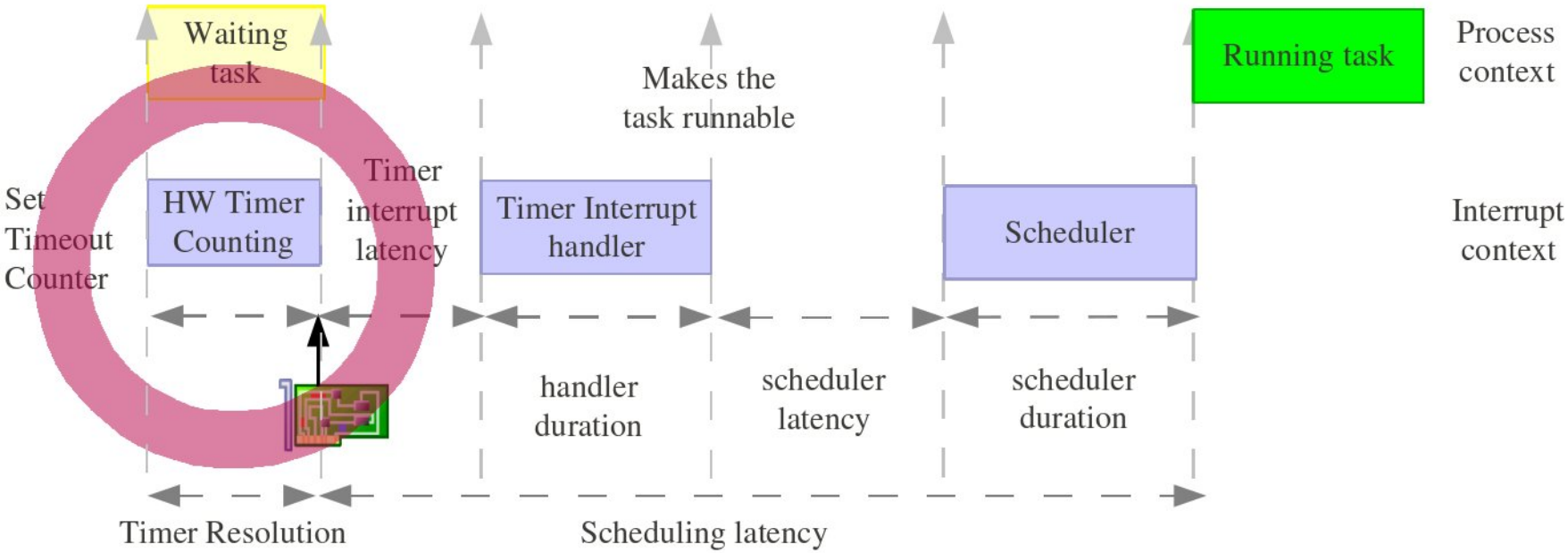

Figure: Kernel Latency of Periodic Task

The kernel latency of periodic task is also related to the resolution of timer system.



Multitask Processing

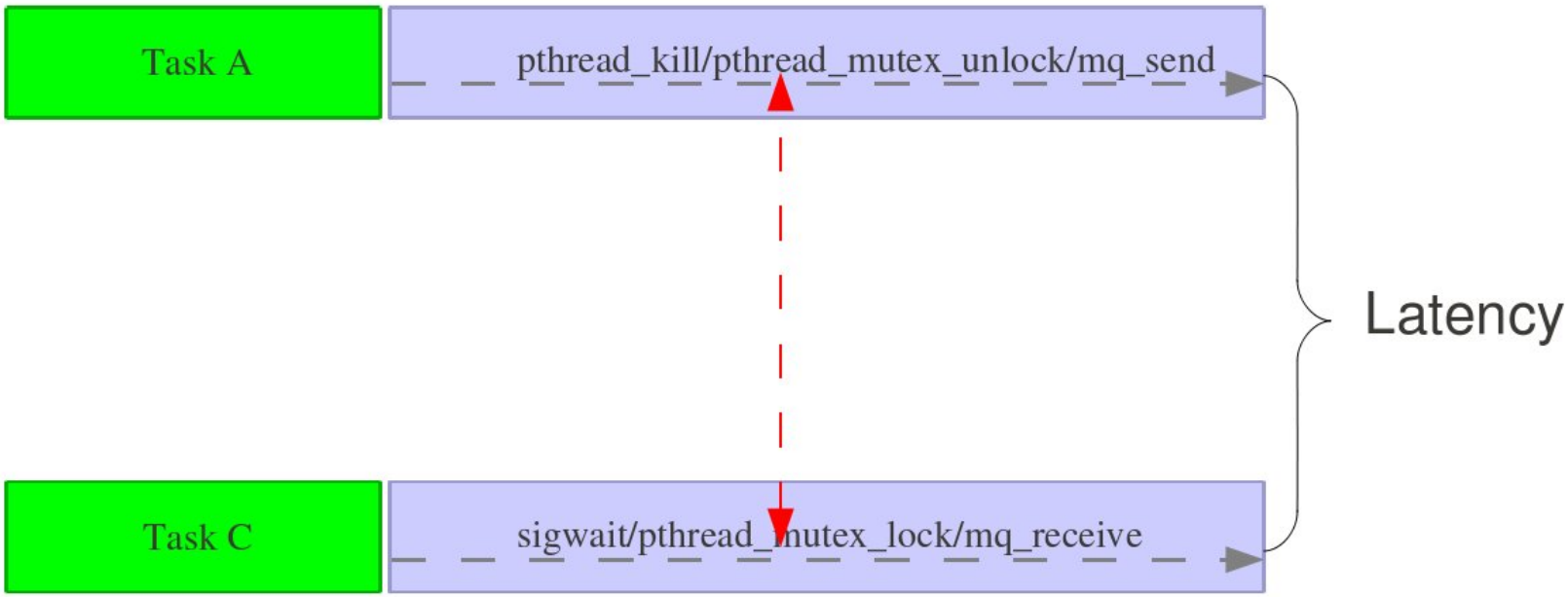

Figure: Kernel Latency of Sync, Mutex and Communication

The latency of multitasking is more complicated, need to consider synchronization, mutex and communication among multi-tasks, their latency should be also determinable.



Priority Inversion

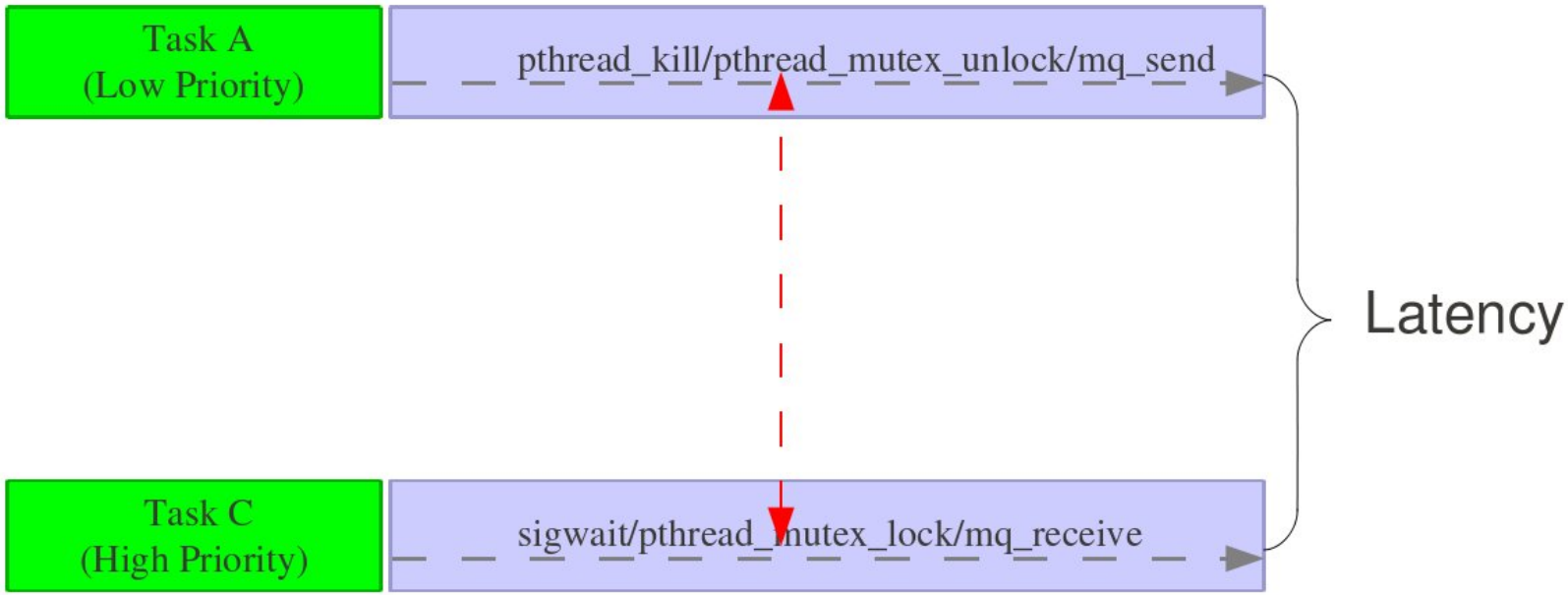

Figure: Kernel Latency of Priority Inversion

Priority Inversion is a high priority task need wait for a resource owned by a low priority task.



Non-determinism Priority Inversion

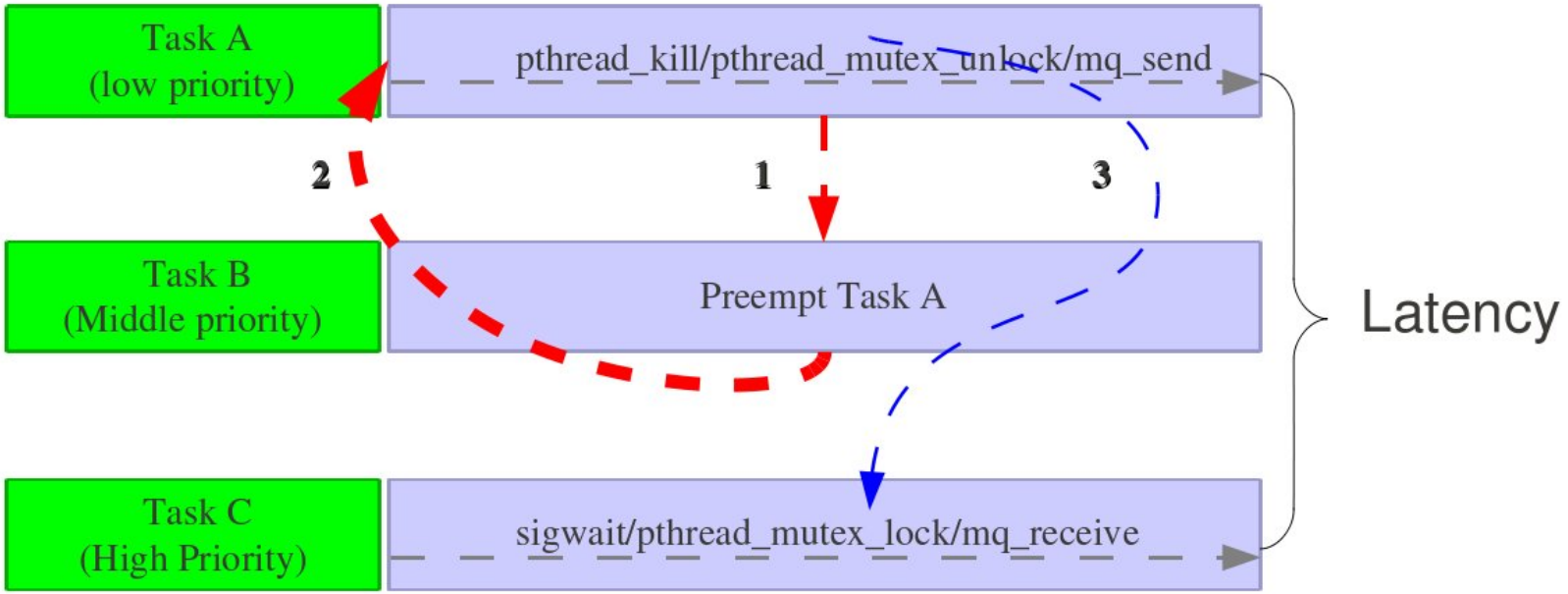

Figure: Kernel Latency of Non-determinism Priority Inversion

Non-determinism Priority Inversion is a middle priority task preempt the low priority task above and as a result, the latency of the high priority task is non-determinable.



Threaded Interrupt

- To solve the interrupt inversion problem, PREEMPT_RT has introduced the concept of threaded interrupts
- The interrupt handlers run in normal kernel threads, so that, the priorities of the different interrupt handlers can be configured
- The real interrupt handler, as executed by the CPU, is only in charge of masking the interrupt and waking-up the corresponding thread
- The idea of threaded interrupts also allows to use sleeping spinlocks (see later)



Threaded Interrupt(cont.)

- Merged since 2.6.30, the conversion of interrupt handlers to threaded interrupts is not automatic: drivers must be modified
  - setup_irq(),request_irq() —-> request_threaded_irq()

```
rc = request_threaded_irq(irq, handler, thread_fn, irqflags, devname,
                          dev_id);
```

  - http://www.kernel.org/doc/htmldocs/genericirq.html
- In PREEMPT_RT, ‘all interrupt handlers’(include top half and bottem half) are switched to threaded interrupts
  - SoftIRQ
    - Create: spawn_ksoftirqd() –> kthread_create() –> run_ksoftirqd()
    - Wakeup: irq_exit() –> do_softirqd() –> wakeup_softirqd()
  - HardIRQ:
    - Create: __setup_irq() –> kthread_create() –> irq_thread()
    - Wakeup: do_IRQ() –> handle_IRQ_event() –> wake_up_process()



Threaded Interrupt(cont.)

Some IRQs can/should not be threaded

- Timers: ‘driver’ of the scheduler should have top priority

```
struct irqaction c0_compare_irqaction = {
        .handler = c0_compare_interrupt,
        .flags = IRQF_DISABLED | IRQF_PERCPU | IRQF_TIMER,
        .name = "timer",
};
```

- Unthread short handlers to reduce context-switch

```
struct irqaction cascade_irqaction = {
        .handler = no_action,
        .name = "cascade",
        .flags = IRQF_NODELAY,
};
```

- ‘Emergent’ IRQ should be handled immediately

```
static struct irqaction busirq = {
        .name = "bus error",
        .flags = IRQF_DISABLED | IRQF_NODELAY,
};
```



Threaded Interrupt(cont.)

Unthreaded IRQ handlers must use ‘raw’ spinlocks

- Sleeping locks in interrupt context will block the system
- Raw spinlocks are prefixed by raw_
- Including definition, declaration and helper functions
- Example: i8253 Timer interrupt

```
static struct irqaction irq0  = {
        .handler = timer_interrupt,
        .flags = IRQF_DISABLED | IRQF_NOBALANCING | IRQF_TIMER,
        .name = "timer"
};
...
-extern spinlock_t i8253_lock;
+extern raw_spinlock_t i8253_lock;
...
-DEFINE_SPINLOCK(i8253_lock);
+DEFINE_RAW_SPINLOCK(i8253_lock);
...
-       spin_lock(&i8253_lock);
+       raw_spin_lock(&i8253_lock);
...
-       spin_unlock(&i8253_lock);
+       raw_spin_unlock(&i8253_lock);
```



No Forced Preemption(Server)

- Kernel code (interrupts, exceptions, system calls) never preempted.
- Best for systems making intense computations, on which overall throughput is key.
- Best to reduce task switching to maximize CPU and cache usage (by reducing context switching).
- Still benefits from some Linux 2.6 improvements, CFS: O(log n), increased multiprocessor safety (work on RT preemption was useful to identify hard to find SMP bugs).
- Can also benefit from a lower timer frequency (100 Hz instead of 250 or 1000).



Voluntary Kernel Preemption (Desktop)

- Adds explicit rescheduling points throughout kernel code

```
drivers/char/tty_io.c: do_tty_write():
                                for (;;) {
                                        ...
                                        cond_resched();
                                }
```

- May need to break spinlocks

```
drivers/md/raid5.c: raid5d ():
                        spin_lock_irq(...);
                        while (1) {
                                ...
                                spin_unlock_irq(...);
                                ...
                                cond_resched();
                                spin_lock_irq(...);
                        }
                        ...
                        spin_unlock_irq(...);
```

- Minor impact on throughput.



Preemptible Kernel (Low-Latency Desktop)

- Most kernel code can be involuntarily(forcely) preempted at any time.
- When a process becomes runnable, no more need to wait for kernel code to return before running the scheduler.
- A rescheduling point occurs when exiting the outer critical section.
- Typically for desktop or embedded systems with latency requirements in the milliseconds range.
- Still a relatively minor impact on throughput.



Preemptible Kernel (Low-Latency Desktop)(cont.)

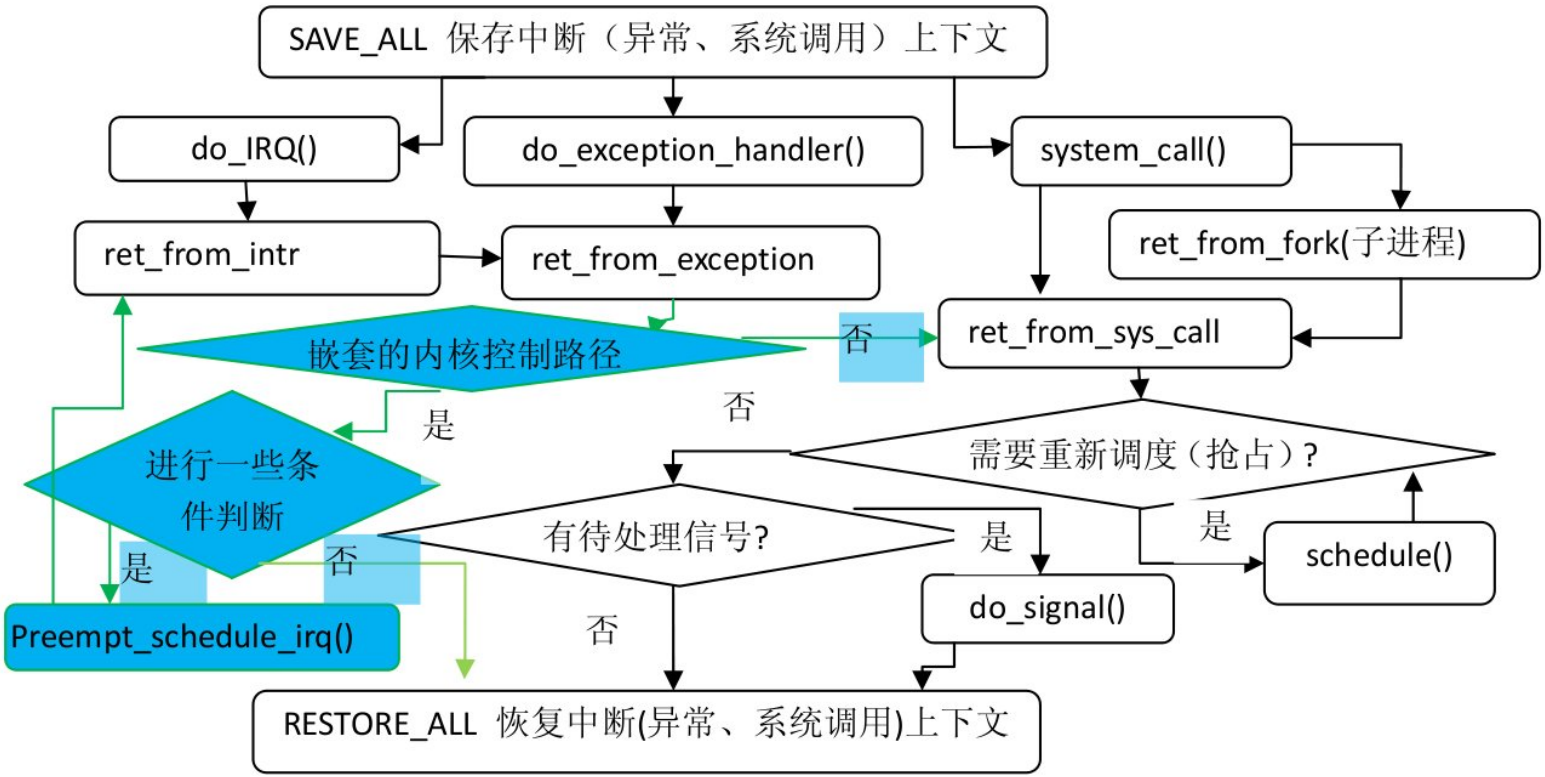

Figure: Preempt irq



Complete Preemption (Real-Time)

- Replaces all kernel spinlocks by mutexes (or so-called sleeping spinlocks)
- Instead of providing mutual exclusion by disabling interrupts and preemption, they are just normal locks : when contention happens, the process is blocked and another one is selected by the scheduler
- Works well with threaded interrupts, since threads can block, while usual interrupt handlers could not
- Some core, carefully controlled, kernel spinlocks remain as normal spinlocks
- With CONFIG_PREEMPT_RT, virtually all kernel code becomes preemptible: An interrupt can occur at any time, when returning from the interrupt handler, the woken up process can start immediately



Real Time Scheduling Policy

- Linux scheduler support different scheduling classes
- The default class, in which processes are started by default is a time-sharing class
  - All processes, regardless of priority, get some CPU time
  - Their CPU time proportion is dynamic and affected by the nice value, which ranges from -20 (highest) to 19 (lowest). Can be set using the nice or renice commands
- The real-time classes SCHED_FIFO and SCHED_RR
  - The highest priority process gets all the CPU time, until it blocks.
  - In SCHED_RR, round-robin scheduling between the processes of the same priority.
  - Priorities ranging from 0 (lowest) to 99 (highest)
- Do we need another real-time class? please see Documentation/scheduler/ and SCHED_DEADLINE: http://www.evidence.eu.com/content/view/313/390/



High Resolution Timers

- The resolution of the timers used to be bound to the resolution of the regular system tick
  - Usually 100 Hz or 250 Hz, depending on the architecture and the configuration
  - A resolution of only 10 ms or 4 ms.
  - Increasing the regular system tick frequency is not an option as it would consume too much resources
- The high-resolution timers infrastructure, merged in 2.6.21, allows to use the available hardware timers to program interrupts at the right moment.
  - Hardware timers are multiplexed, so that a single hardware timer is sufficient to handle a large number of software-programmed timers.
  - Usable directly from user-space using the usual timer APIs
- Please get more from Documentation/timers/hrtimers.txt



High Resolution Timers(cont.)

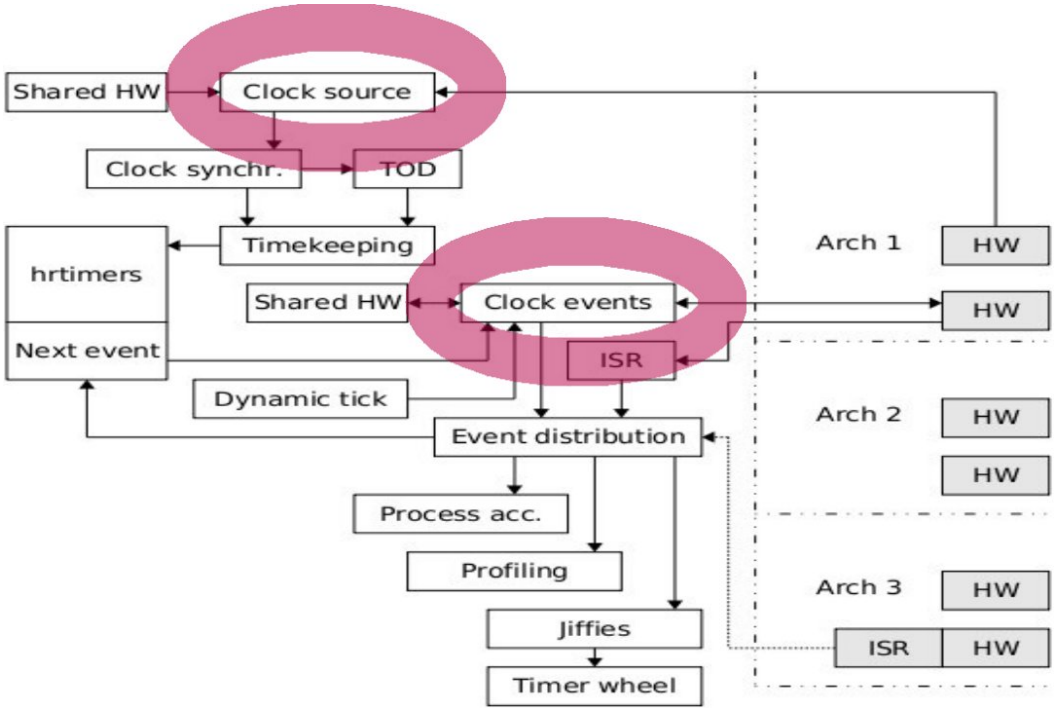

Figure: Linux Time System With hrtimers

Please see details from “Hrtimers and Beyond”:

http://www.kernel.org/doc/ols/2006/ols2006v1-pages-333-346.pdf



High Resolution Timers(cont.): Clock Source

- Get timestamp from hardware count registers

```
arch/mips/kernel/csrc-r4k.c:
static cycle_t c0_hpt_read(struct clocksource *cs)
{
        return read_c0_count();
}
static struct clocksource clocksource_mips = {
        .name           = "MIPS",
        .read           = c0_hpt_read,
        .mask           = CLOCKSOURCE_MASK(32),
        .flags          = CLOCK_SOURCE_IS_CONTINUOUS,
};
init_r4k_clocksource(): clocksource_set_clock(&clocksource_mips, mips_hpt_frequency);
```

- Convert timestamp from cycles to nanoseconds

```
kernel/time/timekeeping.c: timekeeping_get_ns_raw():
        /* read clocksource: */
        clock = timekeeper.clock;
        cycle_now = clock->read(clock);

        /* calculate the delta since the last update_wall_time: */
        cycle_delta = (cycle_now - clock->cycle_last) & clock->mask;
        /* return delta convert to nanoseconds using ntp adjusted mult. */
        return clocksource_cyc2ns(cycle_delta, clock->mult, clock->shift);
```



High Resolution Timers(cont.): Clock Events

- Convert delta from nanoseconds to cycles

```
kernel/time/clockevents.c: clockevents_program_event():
        delta = ktime_to_ns(ktime_sub(expires, now));
        clc = delta * dev->mult;
        clc >>= dev->shift;
        return dev->set_next_event((unsigned long) clc, dev);
```

- Set timer interrupt for the next event

```
arch/mips/kernel/cevt-r4k.c:
mips_next_event():
        unsigned int cnt;
        int res;

        cnt = read_c0_count();
        cnt += delta;
        write_c0_compare(cnt);
        res = ((int)(read_c0_count() - cnt) > 0) ? -ETIME : 0;
        return res;
r4k_clockevent_init():
        struct clock_event_device *cd;
        cd->name                = "MIPS";
        cd->features            = CLOCK_EVT_FEAT_ONESHOT;
        clockevent_set_clock(cd, mips_hpt_frequency);
        cd->set_next_event      = mips_next_event;
```



Priority inheritance

- One classical solution to the priority inversion problem is called priority inheritance
  - The idea is that when a task of a low priority holds a lock requested by an higher priority task, the priority of the first task gets temporarly raised to the priority of the second task : it has inherited its priority.
- In Linux, since 2.6.18, mutexes support priority inheritance
- In userspace, priority inheritance must be explicitly enabled on a per-mutex basis.
- Please get more from Documentation/{rt-mutex-design.txt, rt-mutex.txt, pi-futex.txt}



Setting up PREEMPT_RT Kernel

- Check the available -rt patches in

  http://www.kernel.org/pub/linux/kernel/projects/rt/
- Download and extract mainline Linux kernel
- Download the corresponding -rt patches
- Apply it to the mainline kernel tree
- Configure CONFIG_PREEMPT_RT and High-resolution timers
- Compile your kernel, and boot
- Setting appropriate priorities to the interrupt threads and tasks



Porting PREEMPT_RT patches

- Check the available -rt patches
- Choose one version is very near the kernel version supports your board
  - best to choose the same version
  - better to choose one support the ARCH of your board
- Analyze the board related parts
- Port them one part by one part
  - Threaded interrupts: unthreaded and spinlocks
  - Preemption: Add scheduling points
  - If require SCHED_DEADLINE, port it
  - High Resolution Timers: clocksource and clockevents
- Please refer to “Porting RT-preempt to Loongson2F”



Testing PREEMPT_RT Kernel

- The most popular test tool: cyclictest
  - https://rt.wiki.kernel.org/index.php/Cyclictest
  - cyclictest -l10000 -m p99 -i200 -q
- Build worst case test scenario
  - https://rt.wiki.kernel.org/index.php/Worstcase_Latency_Test_Scenario
- Plot the testing result
  - gnuplot
- Test every latency component
  - Please read “Research and Practice on PreemptRT Patch of Linux” and

    http://dev.lemote.com/cgit/rt4ls.git/tree/tools/rt/interrupt_latency/latency_tracer.c?h=rt/2.6.33/loongson
- More test tools
  - The realtime testcases under ltp-full testsuite
  - http://elinux.org/Realtime_Testing_Best_Practices



Monitoring PREEMPT_RT Kernel

- Online monitoring and comparing
  - Use the QA Farm: https://www.osadl.org/?id=864
- Local monitoring
  - Use cyclictest itself without -q
  - Use cyclictest + Oscilloscope
  - Oscilloscope is available from

    http://docs.redhat.com/docs/en-US/Red_Hat_Enterprise_MRG/1.3/html/Tuna_User_Guide/chap-Tuna_User_Guide-Using_Testing_Tools_with_TUNA.html



Latency debugging with Ftrace

- Maintainer: Steven Rostedt, http://people.redhat.com/srostedt
- Implementation
  - Latency Tracers: Add tracing points manually
  - Function Tracers: Add tracing points with -pg of gcc
- Tracers
  - preemptirqsoff: Trace sections disables interrupts and preemption
  - wakeup: Trace the scheduler/wakeup latency
  - function: Get more details about kernel actions
- Tools
  - trace_cmd, http://people.redhat.com/srostedt/trace-cmd-linuxcon-2010.odp
  - kernelshark, http://people.redhat.com/srostedt/kernelshark/HTML
- Documentations
  - Documentation/trace/
  - http://www.omappedia.org/wiki/Installing_and_Using_Ftrace
  - “Finding Origins of Latencies Using Ftrace”
  - “Debugging the kernel using Ftrace”



Latency debugging with Perf

- Maintainer: Ingo Molnar
- Predecessor: Oprofile
  - only cope with hardware performance counters
- Perf is a peformance debugging infrastructure
  - Trace system performance bottleneck with the hardware performance counters(Cache Miss, Branch Miss, TLB Miss) and software performance counters(Page Fault)
  - A new system call is added for it: sys_perf_event_open()
- tools/perf/Documentation/



Developing Real Time Application

- No special library is needed, the POSIX realtime API is part of the standard C library
- The glibc or eglibc C libraries are recommended, as the support of some real-time features is not available yet in uClibc
  - Priority inheritance mutexes or NPTL on some architectures, for example
- Compile a program
  - ARCH-linux-gcc -o myprog myprog.c -lrt
- To get the documentation of the POSIX API
  - Install the manpages-posix-dev package and Run man functioname



Developing Real Time Application(cont.)

- Schedule a task with a specific scheduling class and a specific priority
  - chrt -f 99 ./myprog
  - Use the sched_setscheduler() or pthread_attr_setschedpolicy() API for process and thread respectively
- Memory locking
  - In order to solve the non-determinism introduced by virtual memory, memory can be locked to Guarantee that the system will keep it allocated and Guarantee that the system has pre-loaded everything into memory
  - mlockall(MCL_CURRENT | MCL_FUTURE); Locks all the memory of the current address space, for currently mapped pages and pages mapped in the future



Developing Real Time Application(cont.)

- PI-Mutexes
  - Priority inheritance must explictly be activated: pthread_mutexattr_getprotocol(&attr, PTHREAD_PRIO_INHERIT);
- Clock and Timers
  - clock_getres(), clock_nanosleep(), clock_gettime()
- Synchronization, mutex and communication
  - Not all existing polices guarantee real-time, please test them before using
  - pthread_kill/sigwait, pthread_mutex_{unlock,lock} and semop() are validated in “Research and Practice on PreemptRT Patch of Linux”
- Real Time Application Example
  - https://rt.wiki.kernel.org/index.php/HOWTO:_Build_an_RT-application
- Read more from “Posix.4 Programmers Guide: Programming for the Real World”



References

- Internals of the RT Patch, Steven Rostedt
  - http://www.kernel.org/doc/ols/2007/ols2007v2_pages_161_172.pdf
- Real-Time Linux Wiki
  - http://rt.wiki.kernel.org
- Download Page
  - http://www.kernel.org/pub/linux/kernel/projects/rt/
- Git Repository
  - http://git.kernel.org/?p=linux/kernel/git/tip/linux-2.6-tip.git;a=shortlog;h=rt/2.6.33
- POSIX Standard 1003.1
  - http://pubs.opengroup.org/onlinepubs/007908799/xsh/realtime.html
- Real time in embedded Linux systems
  - http://free-electrons.com/docs/realtime/
- http://www.faqs.org/faqs/realtime-computing/faq/
- http://dictionary.reference.com/browse/latency
- Real Time vs. Real Fast: How to Choose?