MASTERING REAL-TIME LINUX

ROAD TO PREEMPT_RT AND XENOMAI 3

JEAN-FRANÇOIS DEVERGE (AUG 2016)

CREDITS AND LINKS

- I. Puaut « Real-time system lecture » at http://www.irisa.fr/alf/downloads/puaut/STR/RTSlecture.pdf
- J. Huang "Making Linux do Hard Real-time" at http://www.slideshare.net/jserv/realtime-linux
- P. Gerum « Xenomai Training » Internal 2012
- J. Kizka « Xenomai 3 – An Overview of the Real-Time Framework for Linux » at http://events.linuxfoundation.org/sites/events/files/slides/ELC-2016-Xenomai_0.pdf
- M. Vandal et al « LITMUS-RT: A Hands-On Primer » http://www.litmus-rt.org/tutorial/tutorial-slides.pdf
- H. Takada "Introducing a new temporal partitioning scheme to AUTOSAR OS" https://www.autosar.org/fileadmin/files/events/2015-10-29-8th-autosar-open/Introducing_a_new_temporal_partitioning_scheme_to_AUTOSAR_OS_Takada.pdf
- J. Lelli "SCHED_DEADLINE a status update" at http://events.linuxfoundation.org/sites/events/files/slides/SCHED_DEADLINE-20160404.pdf
- J.H. Brown "How fast is fast enough? Choosing between Xenomai and Linux for real-time applications" https://www.osadl.org/fileadmin/dam/rtlws/12/Brown.pdf
- L. Henriques "Threaded IRQs on Linux PREEMPT –RT" http://www.artist-embedded.org/docs/Events/2009/OSPERT/OSPERT09-Henriques.pdf

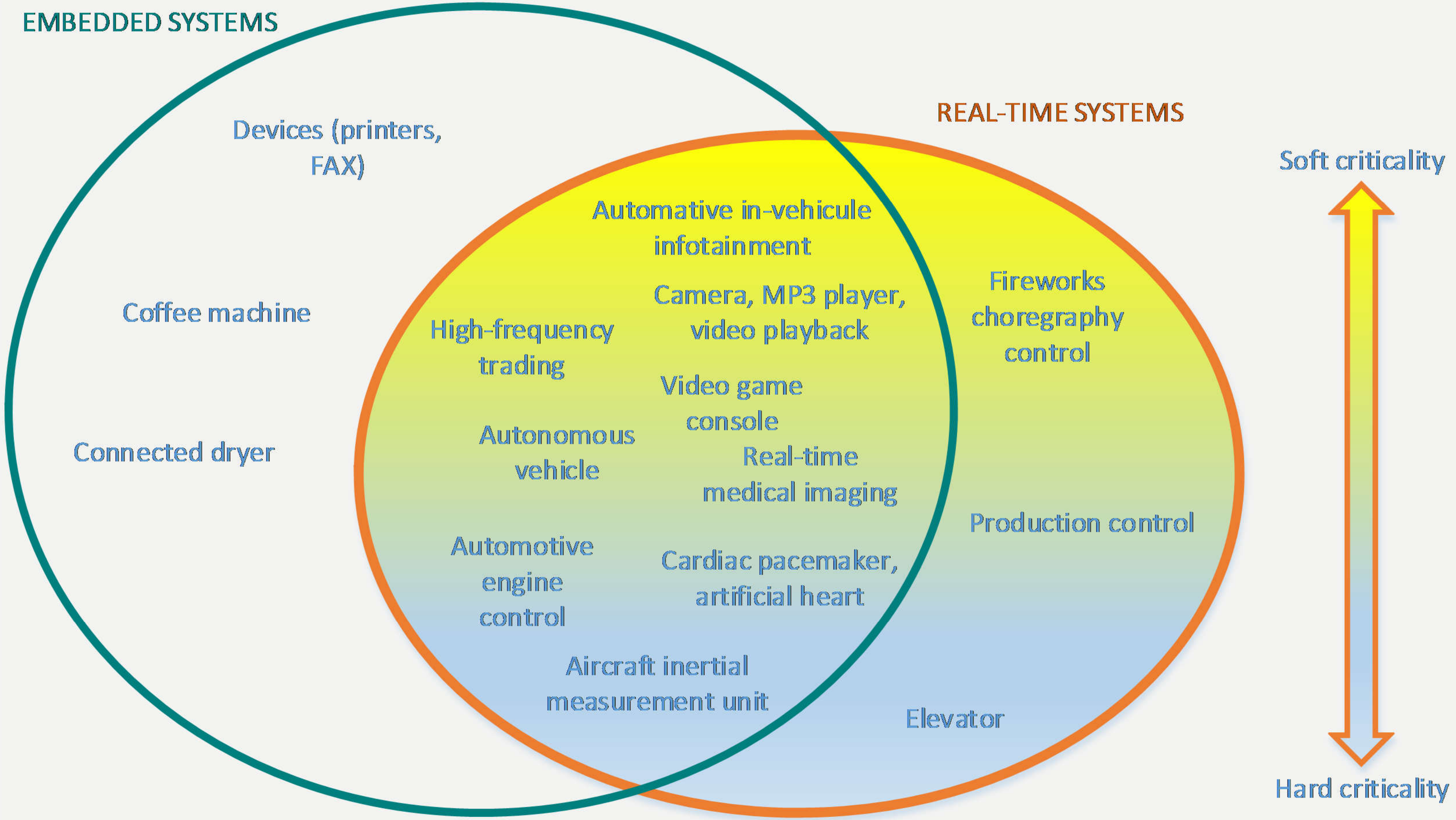
INTRO

WHAT IS REAL-TIME AND WHY LINUX FOR REAL-TIME SYSTEMS

PROPOSED DEFINITION

- Real-time
  - Time: correctness depends not only on the result but also to the time of the resulting action
  - Real: physical or external time
- Timing constraint: specified delay between two events
  - Specified in terms of real (physical, external) time
  - Deadline: maximum delay between task arrival and termination

HARD VS SOFT REAL-TIME SYSTEMS



TWO MAJOR TYPES OF REAL-TIME

- Hard real time: violating a timing constraint results in catastrophic consequences (loss of human lives, ecological or financial disaster)
  - Need of strict guarantees and strong determinism
- Soft real time: meeting timing constraints is (highly) desirable and missing a timing constraint does not jeopardize the system correctness, but degrades the quality
  - Need good-enough guarantees and best effort determinism
- Batch or interactive: reaction time invisible to the user

  However, “pure” real-time systems only exists in smallest embedded devices (e.g. one application = one processor). A complex application can run concurrently multiple types of real-time tasks and batch/interactive tasks on the same device, also called « mixed criticality real-time systems »
  - This talk covers primarily a single or multi-core processor running a mix of batch/interactive tasks and real-time tasks

WHY LINUX

- Lots of available commercial or free Real-Time Operating Systems (RTOS)
  - VxWorks, QNX, ThreadX, VRTX, uC/OS, FreeRTOS, RIOT, ChibiOS/RT, RTEMS…
  - Most RTOS targets minimal features for smallest memory footprint but VxWorks and QNX are notably near-complete operating system (filesystem, Ethernet/IP stack support, memory protection)
- Industry looks for standard hardware with transparent RTOS support
  - Single board computer (SBC), e.g. Raspberry PI does provide Linux and Windows BSP, but no BSP for VxWorks (community does partial port to FreeRTOS for Raspberry PI)
  - COM Express or QSeven modules always provide Linux BSP, sometimes for Windows and rarely for VxWorks and QNX
- Boards manufacturers considers Linux as a standard for board BSP and does not deliver RTOS-specific BSP on product launch. Industrial clients are encouraged to use Linux instead of a proper RTOS solution due to development cost and time to market.

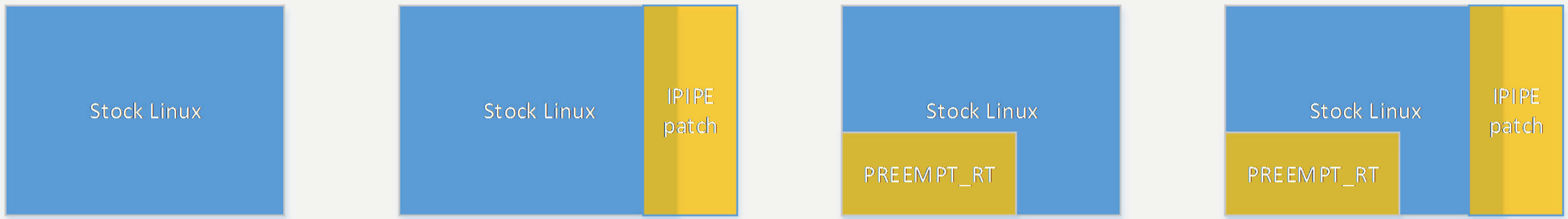
http://www.orbitmicro.com/company/blog/wp-content/uploads/2009/05/qseven-hand.jpg

WHY STANDARD LINUX IS NOT A GOOD RTOS

- Definition: standard Linux is a stock Linux kernel from kernel.org, or customized Linux with a vendor specific BSP (e.g. board specific drivers etc.)
- Pros: Linux is an excellent development environment
  - Standard toolchains, cross compiler, automatic distribution generation, profilers and debuggers, standard POSIX programming API
  - Completely customizable kernel, available libraries and middleware for most application domains
- Cons: Linux is not natively designed to guarantee timing constraints
  - Inadequate scheduling policies, paging mechanisms, interrupt management

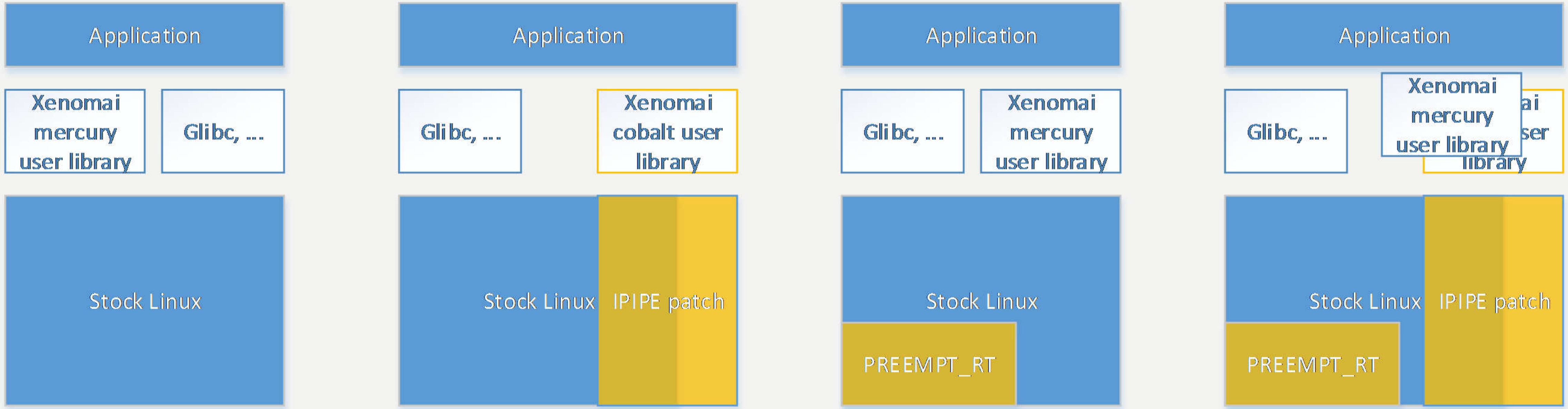
SOLUTIONS: PREEMPT_RT AND IPIPE

- PREEMPT_RT and Xenomai 3 (aka « IPIPE ») provides two independent kernel sources patches to make Linux kernel a proper real-time system runtime
  - These two patches implement two separate approaches to ensure timing constraints
  - IPIPE patch and PREEMPT_RT patch can be applied separately or in complement



XENOMAI 3 USER LIBRARY

- Xenomai 3 also provides an user library to emulate common RTOS API (VxWorks, PSOS, POSIX RT extension).
  - The Xenomai 3's software stack can be used with or without application of (IPIPE or PREEMPT_RT) patches onto Linux kernel. The deterministic behavior of RTOS API will depend on underlying Linux real-time behavior
  - 'mercury' and 'cobalt' are the codenames of the different Xenomai 3 user libraries



XENOMAI USER LIBRARY CONFIGURATION

- Xenomai 3 source provides an user library for both IPIPE or PREEMPT_RT or Stock Linux
  - PREEMPT_RT or Stock Linux mode:
    1. tar xjvf xenomai-3.0.2.tar.bz2
    2. ./configure --with-core=**mercury**
    3. make && make install
  - IPIPE mode:
    1. tar xjvf xenomai-3.0.2.tar.bz2
    2. ./configure --with-core=**cobalt**
    3. make && make install
- Whatever, application can be built/linked to proper (cobalt or mercury ?) Xenomai's user library with a smart 'xeno-config' script

  g++ –o my_rt_application source.cpp **$(xeno-config --cflags --ldflags --skin=vxworks)**

HANDS-ON: WHAT IS THE REAL-TIME BEHAVIOR OF YOUR SYSTEM ?

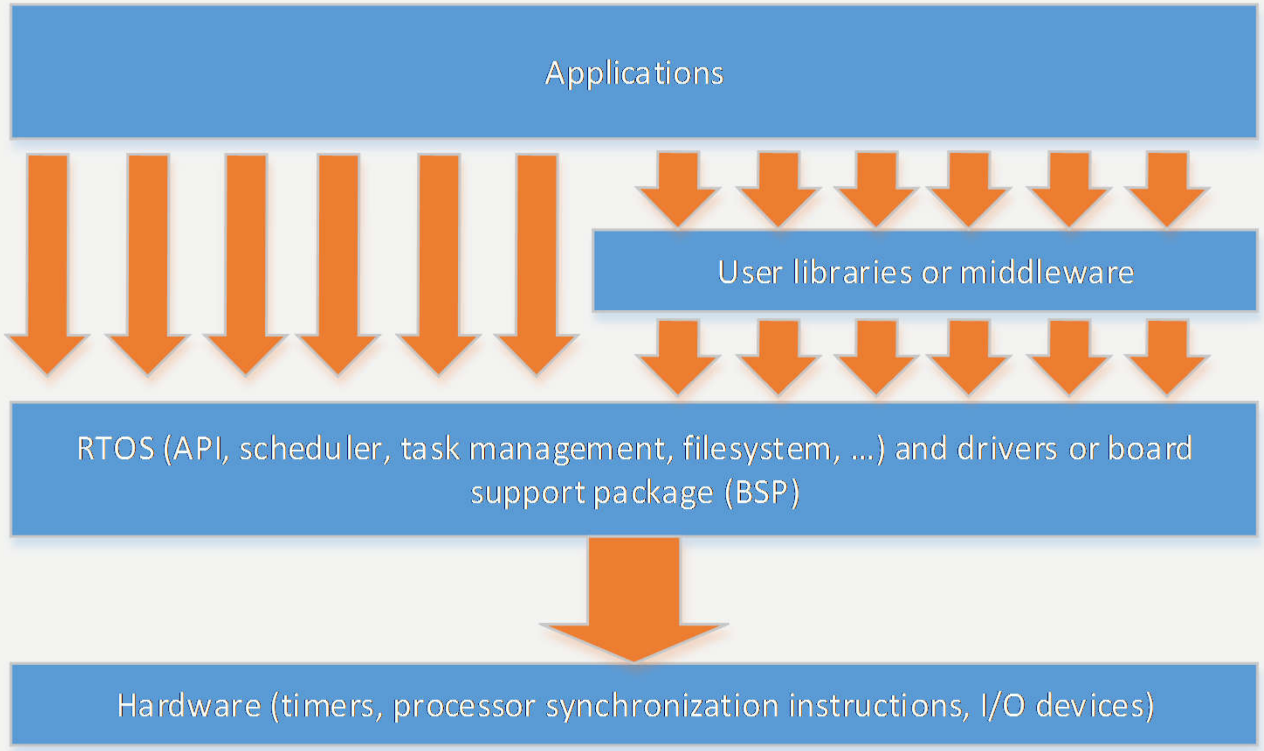
- Build Xenomai 3 for your target machine (e.g. Desktop, Laptop, Raspberry PI, Galileo) for a **mercury** flavor
- Launch benchmark « `latency –h –g log.txt` » (depends on your installation; this could be in /usr/bin/latency)
- During benchmark execution, activates different features of the target machine (buttons, video, sound, network, plug a USB mouse/keyboard)
- Stop the latency measurement with CTRL-C
  - In microseconds, what is the worst-case latency of your system ?
  - Plot the histogram stored in text file log.txt.

RTOS BASIC CONCEPTS

SCHEDULING ALGORITHM AND INTERPROCESS COMMUNICATION

PRINCIPLES OF A RTOS

- Share the processor(s) between concurrent computations
  - RTOS implements synchronization primitives from processor instruction set
- Why an application would need a RTOS ?
  1. Intrinsic parallelism of applications
  2. Some applications meet their deadlines only when using specific scheduling (see next slides)
  3. Portability considerations: common API and abstract from the target hardware (ARM, PPC, X86, …)
  4. Better exploitation of processor during I/O operations (rescheduling during I/O operations)



RTOS SERVICES AND CLIENT API

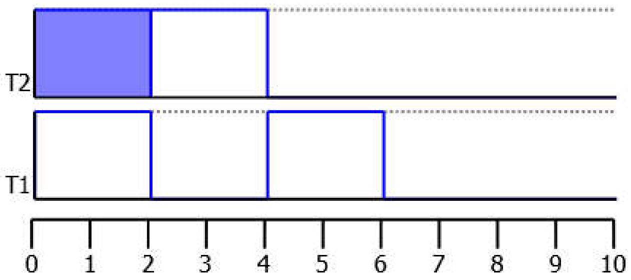
RTOS

- Task management: spawn, deletion, activation, stop, priority set/get
- Time management: periodic task activation, configurable alarm, timeout on blocking API
- Interprocess communication (IPC): semaphore, signals, message queue, memory allocation
- Device driver management: interrupt handling, driver infrastructure

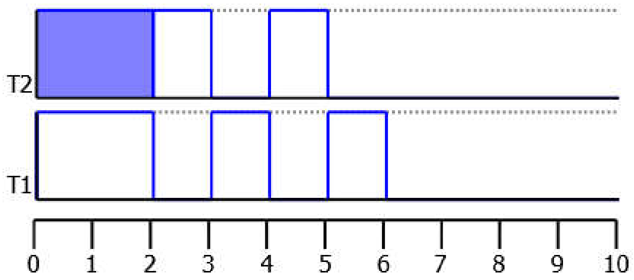
1. VxWorks taskLib.h: taskSpawn, taskInit, taskActivate, exit, taskDelete; taskSuspend, taskResume, taskRestart, taskPrioritySet, taskPriorityGet, taskDelay, taskLock, taskUnlock, taskIdSelf
2. tickLib.h and wdLib.h: tickGet, tickSet, wdCreate, wdDelete, wdStart, wdCancel
3. semLib.h and msgQLib.h: semBCreate, semMCreate, semGive, semTake, semFlush, semDelete, msgQCreate, msgQDelete, msgQSend, msgQReceive, msgQNumMsgs, new, delete, malloc, free
4. intLib.h and ioLib.h: intLock, intUnlock, intConnect, iosDrvInstall, open, read, write, close, ioctl, chdir, ioGlobalStdSet, ioGlobalStdGet

REAL-TIME SCHEDULING

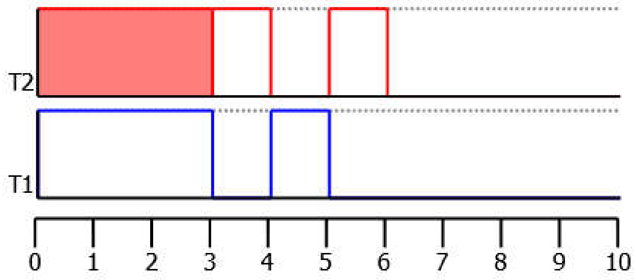
- Scheduling algorithm implements the execution order of tasks on processor(s)
- Task scheduling has an impact on timing constraints
- Example for single processor :
    - Task T1 : arrival at time 0, duration 4, absolute deadline 7
    - Task T2 : arrival at time 2, duration 2, absolute deadline 5
    - Scheduler S1: preemptive round-robin every tick of 1
    - Scheduler S2: preemptive, priority-based, T2 higher priority than T1
- Deadlines are always met with S2, but not necessarily with S1



S2 scheduling success



S1 scheduling success



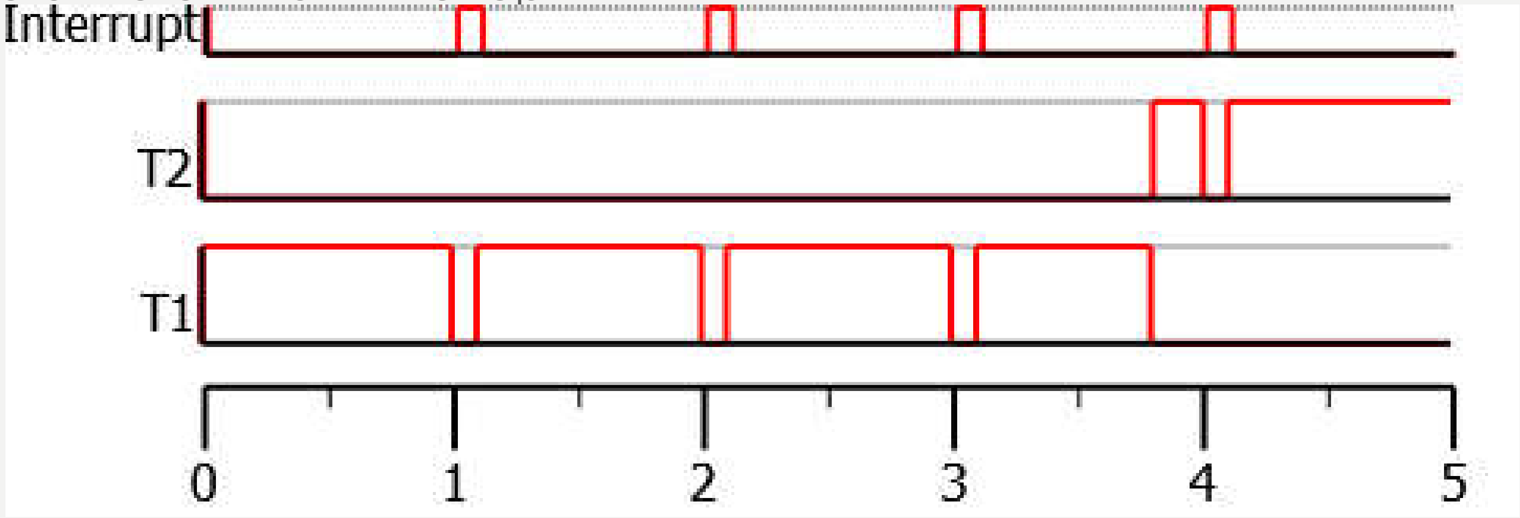
S1 scheduling failure

OFFLINE AND ONLINE SCHEDULING

- Offline scheduling precomputes sequence of task execution at design time
  - Schedule table is fixed and stored in memory. A simple dispatcher is executing the linear schedule at runtime
  - Pros: implementation is simple, CPU overloads can be detected on schedule points
  - Cons: must need exact task arrival need and bounded execution time.
- Online scheduling requires dispatcher to take scheduling decisions at run time
  - Schedule points are task arrival, task termination, return from interrupt, interprocess communication (IPC)
  - Process table may contain priority, quota or group attributes for scheduling policy
  - Cons: CPU overloads are harder to detect, dispatcher is more complex

NON-PREEMPTIVE SCHEDULING

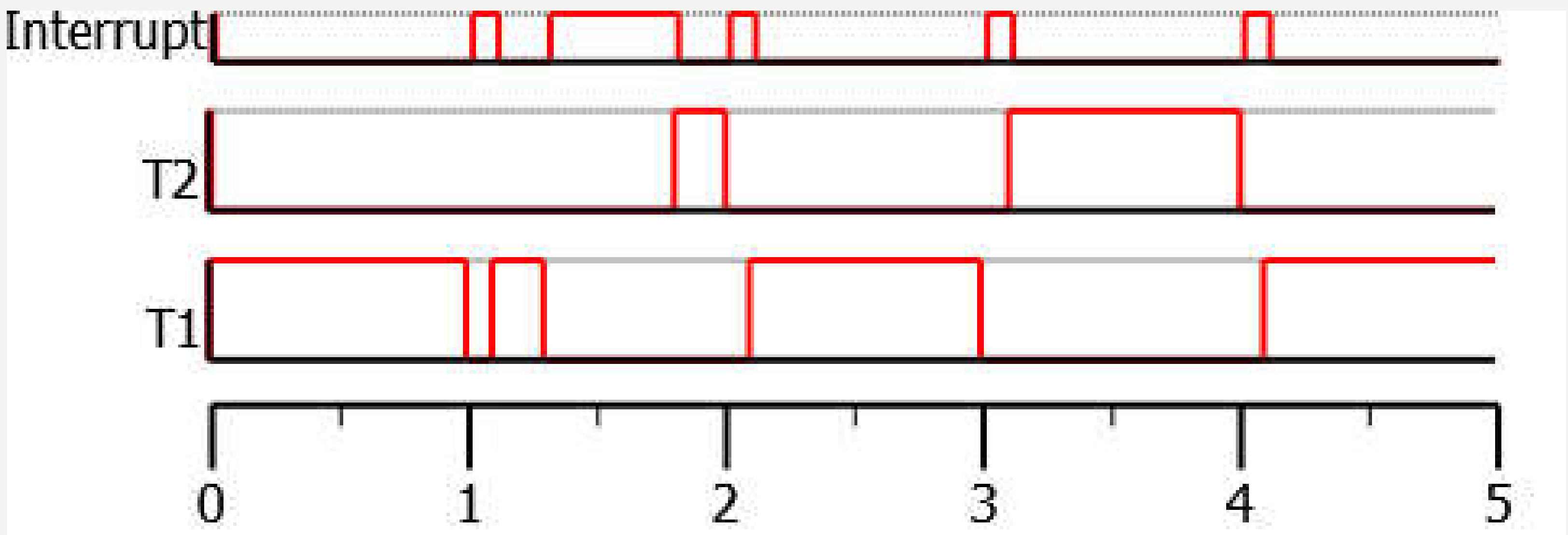
- Non-preemptive scheduling : a task is executed until completed since the task is started
  - An (ideally short) hardware interrupt can cause task suspension, but the same task will continue on return from interrupt



  - The task can provoke a volunteer reschedule to next task with sched_yield(), task_delay() primitives or on blocking IPC (semaphore, message queue, etc.)

PREEMPTIVE SCHEDULING

– Preemptive scheduling : return from interrupt can assign the processor to another ready task
  - Timer ticks and I/O device interrupts are possible sources of tasks preemptions
  - Pros: better reactivity to external events
  - Ex: T2 is blocked on device I/O event since time 0. At time 1.3, device interrupt is received and T2 is unblocked ; T2 is elected by scheduler on return from interrupt. The task T1 has been preempted by T2.



QUICK QUESTION: WHAT ARE FEASIBLE TASKS

- On a real-time system with three tasks T1, T2 and T3.
  - Priority(T3) > Priority(T2) > Priority(T1).
  - T1, T2, T3 arrival times are 0, 2, 4 respectively.
  - T1, T2, T3 execution times are 4, 2, 3 respectively
  - Three keyboard interrupts occurs at time 1.2, 2.6, 3.1.
  - Execution time of an interrupt handling is 0.2.
  - RTOS system tick is every 1. Scheduling policy is non-preemptive **priority-based**
- T1, T2, T3 deadlines are 5, 8, 10
  - Quick question: what are feasible tasks ? (Hint: draw a schedule diagram)
  - Propose a modification to this RTOS to make this task set feasible

TASKS: PROCESS VS THREADS

A. **Process are** tasks managed with an individual memory space
   - Operating systems typically implement memory protection of processes with Memory protection unit (MPU), memory management unit (MMU) or segmentation
   - Two processes can (sometimes) share a segment or a page ; this feature is the shared memory (SHM) IPC
   - Quick question : what are two processes sharing _all_ their pages of their memory space ?

B. **Threads are** tasks with shared memory space
   - The operating system creates first an individual memory space and allocate the process: a process is the first thread in an individual memory space

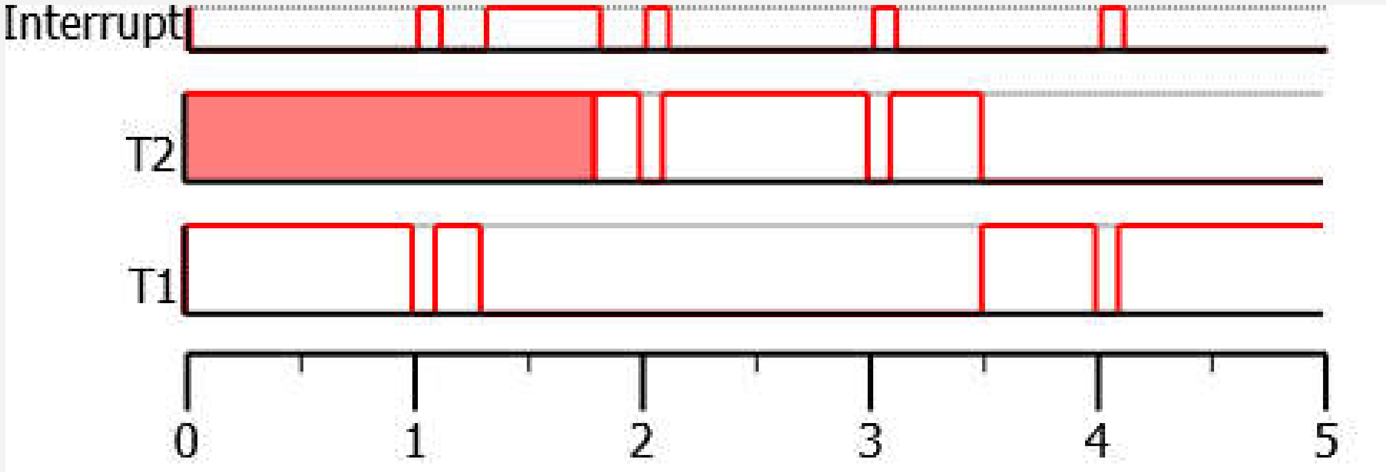
- Schedulers of modern operating systems do not make fundamental difference between processes and threads, they all look the same to the scheduler
  - (For Linux, process = thread)

TASKS: THREADS VS FIBERS

- **Fibers are** threads : threads and fibers shares the same address space
- … But fibers are non-preemptible
  - Fiber is executed until completed since the fiber is started
  - Pros: no need for synchronization for resources only accessed by fibers; no need to reschedule from return from interrupt if fiber is running
- Global task dispatcher schedules all fibers as a unique entry in task table
  - Fibers have the highest global priority to ensure fibers are not preempted by (preemptible) threads ; but interrupt handler execution can « interrupt » fibers execution
  - Fibers can implement their own non-preemptive scheduler policy (priority-based or round-robin based or quota based) in a local task dispatcher
- « In general, fibers do not provide advantages over a well-designed multithreaded application. However, using fibers can make it easier to port applications that were designed to schedule their own threads. » MSDN

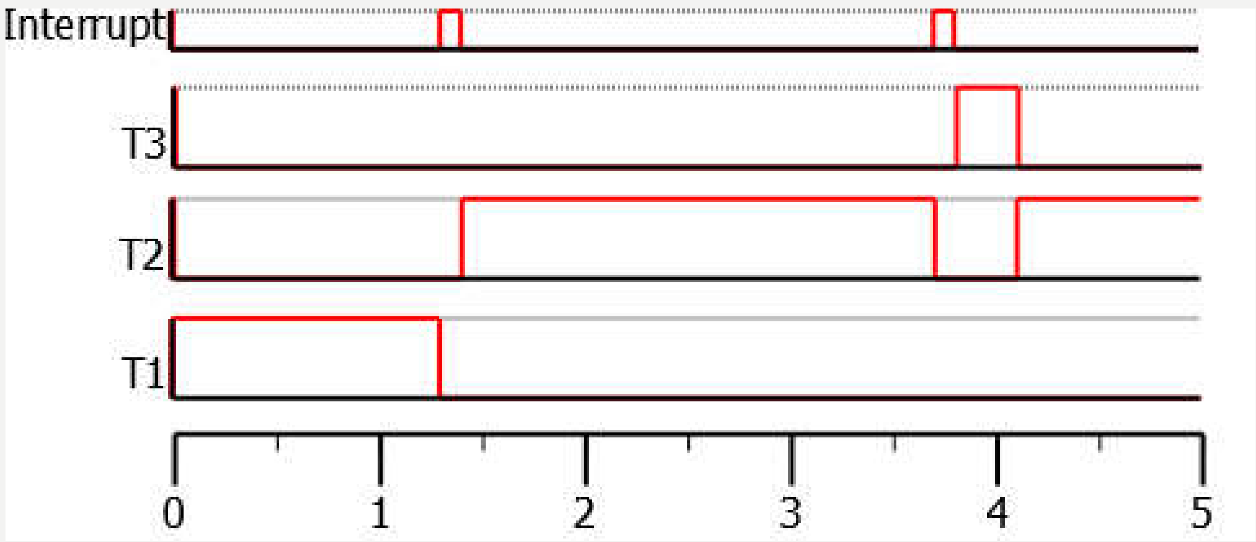
ROUND-ROBIN (RR) AND BATCH SCHEDULING

- General purpose operating system (GPOS) can implement multi level scheduler: round robin scheduling for interactive tasks and batch scheduling for I/O tasks
- Round robin scheduling :
  - A task is executed until next tick expiry (or specified time slice of multiple ticks)
  - Pros: fair scheduling, mainly present for desktop systems (with reactive display)
  - Cons: system throughput is reduced by reschedule penalty at every tick
- Batch scheduling :
  - A task is executed until completed or IO device interrupts unblocks a resource (IPC)
  - Pros: increase system throughput and reduce globally tasks execution times
  - Cons: CPU starvation for non-I/O tasks



PRIORITY-BASED SCHEDULING

- At any time, the running task is the active task with highest priority
  - Fixed priority scheduler: task priority are specified at design (build) time
  - Dynamic priority scheduler: task priority may vary over time
  - Example:
    - T3 and T2 are initially (time=0) blocked on resources R10 and R14 respectively
    - Fixed Priority(T3) > Priority(T2) > Priority(T1) ; operating system is tickless
    - First interrupt releases resource R10 at time 1,3 and second interrupt releases R14 at time 3,7

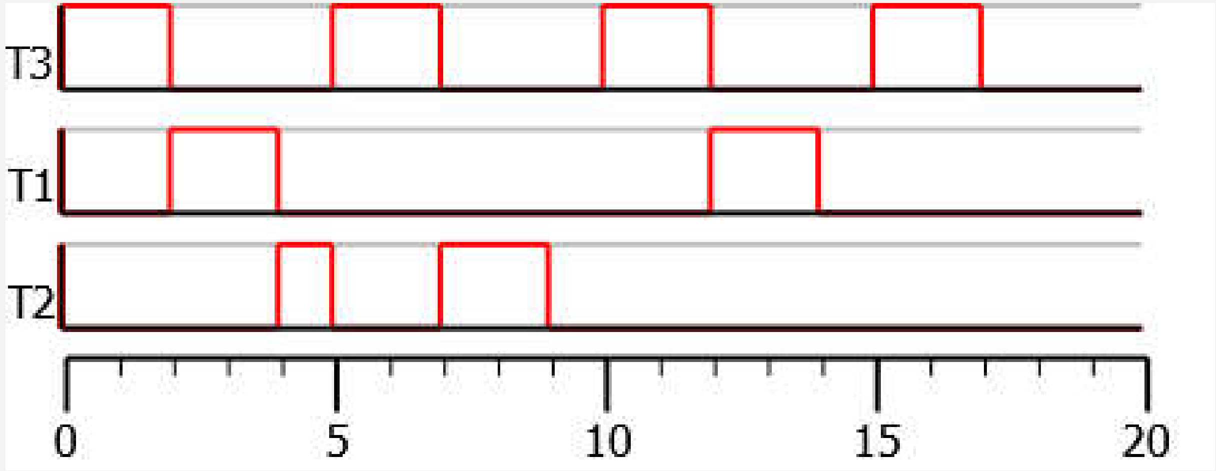


RATE MONOTONIC (RM) SCHEDULING

- RM: tasks with shorter periods will have higher priorities
  - Fixed priority scheduling of periodic tasks
    - Pros: require simple priority-based scheduler, the industry practice for critical systems
  - Aperiodic tasks are serviced in background
  - Interrupts load can be considered like
    - (1) constant noise (e.g.: tick is pure periodic load)
    - (2) or non negligible I/O handling must be managed in a polling task instead of interrupt-based driver

SCHEDULE TABLE HYPERPERIOD

- Remainder: RM ; tasks with shorter periods will have higher priorities
- Consider this schedule table for this task set for RM
  - T1 period=10ms, execution time=2ms
  - T2 period=20ms, execution time=3ms
  - T3 period=5ms, execution time=2ms



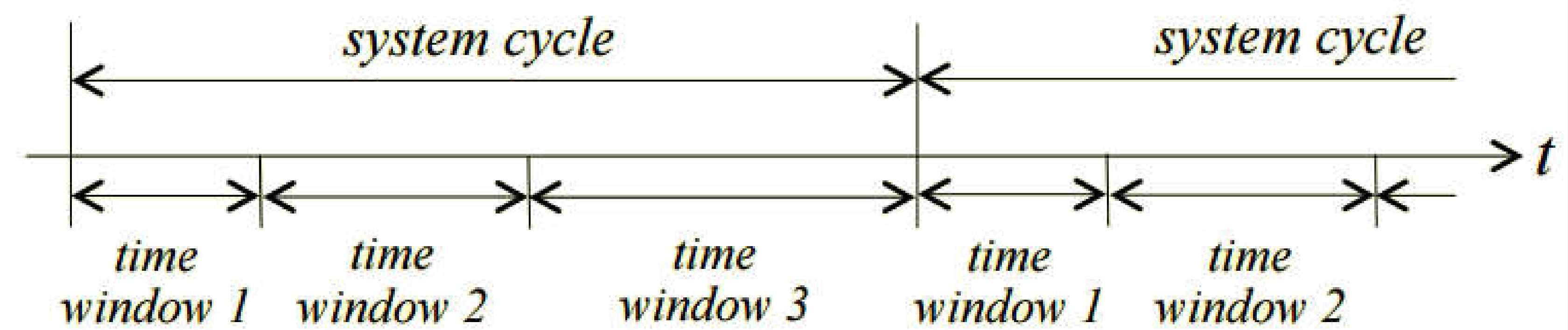
- Quick question: what is the hyperperiod value of this task set ? Why hyperperiod is important for verification of the schedule ?

EARLIEST DEADLINE FIRST (EDF) SCHEDULING

- EDF: at each instant, tasks with earlier deadlines will be executed at higher priorities
  - Dynamic priority scheduling to both periodic and non-periodic tasks
  - Cons: difficult to debug in overload situations
  - Better CPU exploitation than RM : better management of dynamic uses cases (e.g. multiple concurrent media streams with different timing requirements)
- Quick question: draw a schedule table for this task set
  - T1 period=10ms, execution time=2ms
  - T2 period=20ms, execution time=3ms
  - T3 period=5ms, execution time=2ms

TEMPORAL PARTITIONING (TP) SCHEDULER

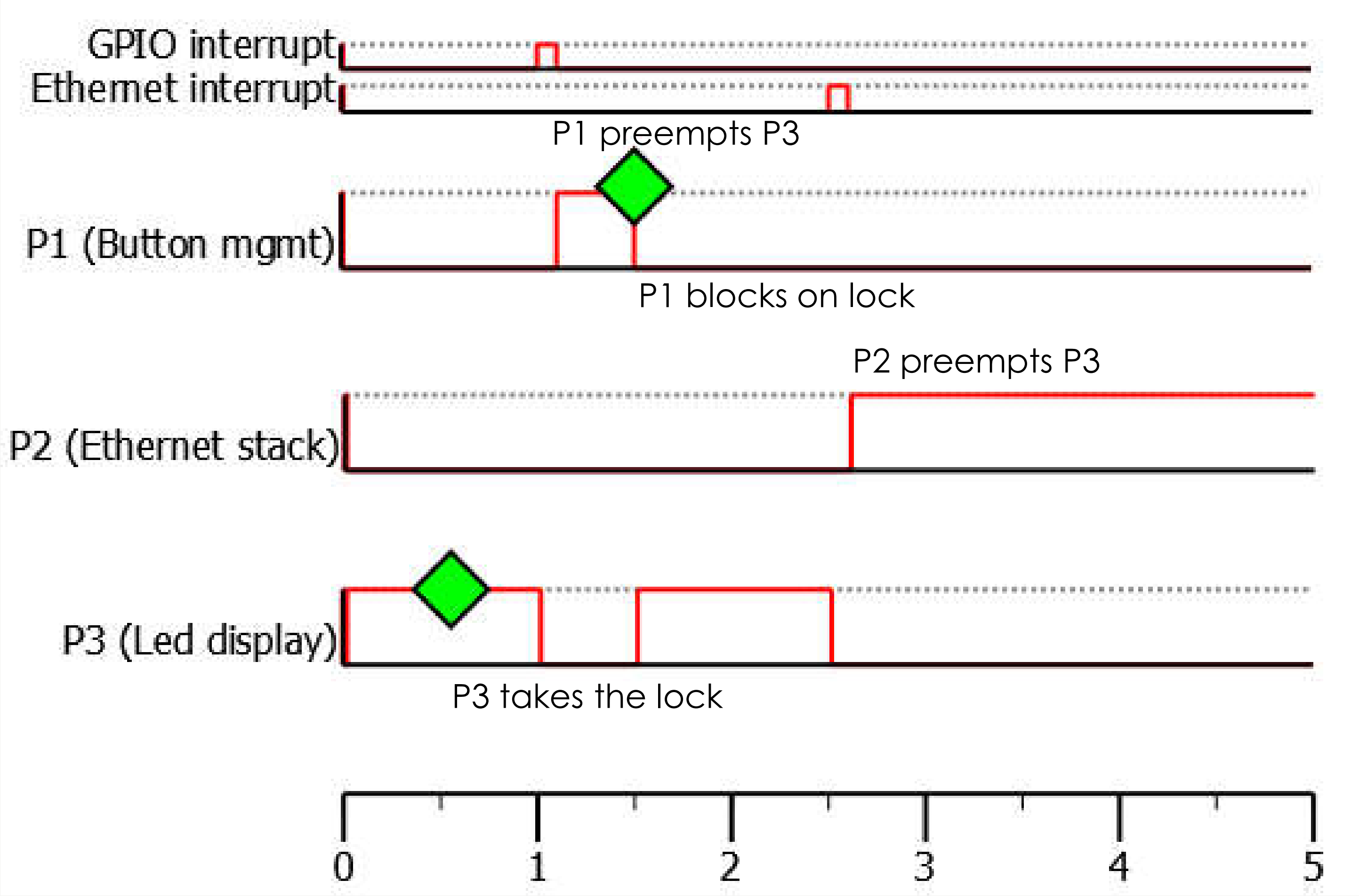
- The system cycle is divided into several time windows. Each time window is assigned to a partition. The partition is executed within the assigned time windows



- Within a time window, tasks belonging to the partition are executed with its own scheduling policy (Round-robin, RM, EDF, …)
- An I/O device's interrupt is unmasked only within the time window assigned to the partition it belongs.
- Pros: minimize temporal influence between partitions

SCHEDULER PRIORITY INVERSION

- **Use-case:** three task P1, P2, P3 manages respectively buttons input (high priority), Ethernet communication (medium priority) and LED user interface (low priority). Buttons and LEDs are managed by a common GPIO device ; concurrent accesses to GPIO device are protected by a lock



- Observations
  1. P1 is blocked on lock until P3 releases it
  2. But P3 waits for P2 termination
  3. Consequently, P1 waits for P2 termination

- P1 waits for P2 while PRIORITY(P1) > PRIORITY(P2)
  - The top priority process could wait for an indefinite duration
  - This is the priority inversion problem

SOLUTIONS TO PRIORITY INVERSION

1. Solution « priority inheritance protocol (PIP)»: the process (P1) gives its priority to the process (P2) holding the lock
   - Scheduler implementation impacts
     - The priority of P2 must be recomputed after lock release
     - Need to support transitive priority inheritance: if a process A blocks B and process B blocks C, then both processes B and A must inherit from C
2. Solution « priority ceiling protocol (PCP) »: lock is associated with the maximum priority of possible process owners (P1 and P2)
   - Scheduler implementation impacts
     - On lock access, the process priority is raised to maximum lock priority
     - Need to know the list of process that may access to each lock at design time

- Note: PCP and PIP are possible for RM, EDF… but these protocols are limited to each partition for temporal partition scheduling
- Many RTOS implements PIP, sometimes PCP is preferred.

MULTI-CORE SCHEDULING

- Soft real-time industry practice:
  - Multicore-aware scheduling (e.g. EDF variants) can provoke task migration
    - Task can migrate from one processor to free processor at runtime
    - Linux patch for multicore-aware real-time scheduler policy : http://www.litmus-rt.org/
  - Pros: load balancing helps to meet deadlines
  - Cons: migration overhead may mitigate benefit (processor's cache pollution); debugging overload scenario are complex to handle
- Hard real-time industry practice:
  - Static priority-based scheduling (e.g. original RM) per core
  - Forbid migration, fixed allocation of tasks per core

LINUX SCHEDULING POLICIES

- Linux enables per-thread scheduling policy with two APIs:
  1. `int sched_setattr(pid_t pid, const struct sched_attr *attr, unsigned int flags);`
  2. `int sched_getattr(pid_t pid, const struct sched_attr *attr, unsigned int size, unsigned int flags);`
- SCHED_DEADLINE implements EDF policy
- SCHED_FIFO is a static priority-based scheduler to implement RM policy
- SCHED_RR is a static priority-based scheduler (too), but a time slice is applied to round robin for tasks with same priority
- SCHED_NORMAL, SCHED_BATCH and SCHED_IDLE are non-RT scheduling policies

```
struct sched_attr {
    u32 size;
    u32 sched_policy;
    u64 sched_flags;

    /* SCHED_NORMAL, SCHED_BATCH */
    s32 sched_nice;

    /* SCHED_FIFO, SCHED_RR */
    u32 sched_priority;

    /* SCHED_DEADLINE */
    u64 sched_runtime;
    u64 sched_deadline;
    u64 sched_period;
};
```

XENOMAI/IPIPE SCHEDULING POLICIES

- In addition to SCHED_FIFO and SCHED_RR, Xenomai/IPIPE introduces additional scheduling policy with POSIX APIs sched_setattr/sched_getattr
  - SCHED_TP implements the temporal partitioning scheduling policy for groups of threads (a group can be one or more threads)
  - SCHED_SPORADIC implements a task server scheduler used to run sporadic activities with quota to avoid periodic (under SCHED_RR or SCHED_FIFO) tasks perturbation
  - SCHED_QUOTA implements a budget-based scheduling policy. The group of threads is suspended since the budget exceeded. The budget is refilled every quota interval
- Note: PREEMPT_RT patch does not provide additional scheduling policies. With PREEMPT_RT, only Linux scheduling policies can apply.

SCHEDULER MATRIX

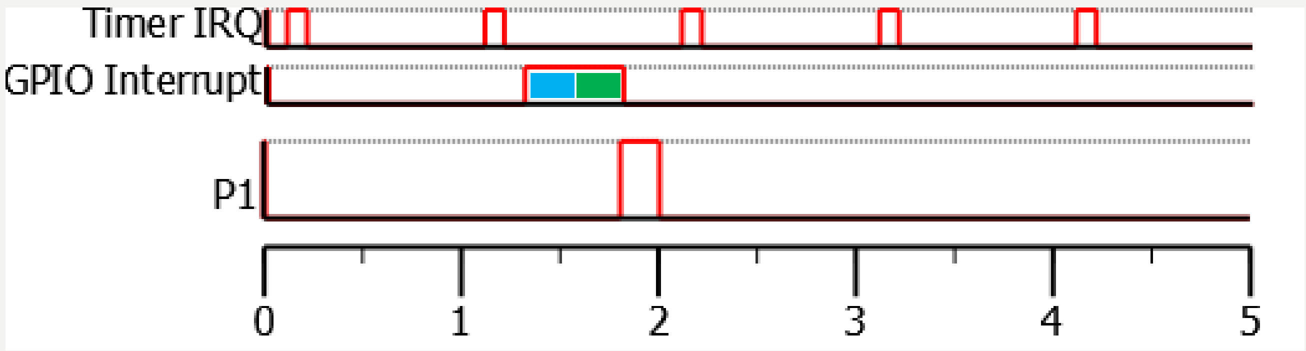
	'stock' Linux	Linux + PREEMPT_RT	Linux + IPIPE	Linux + IPIPE + PREEMPT_RT
SCHED_NORMAL, SCHED_BATCH, SCHED_IDLE	**A**	**A**	**A**	**A**
SCHED_FIFO, SCHED_RR	**A**	**A**	**A+B**	**A+B**
SCHED_DEADLINE	**A**	**A**	**A**	**A**
SCHED_SPORADIC, SCHED_QUOTA, SCHED_TP	-	-	**B**	**B**

A: available to Linux or PREEMPT_RT threads
B: available to IPIPE threads

(LINUX) SOURCES OF LATENCY

RTOS LATENCY

- RTOS shall provide bounded execution on external/internal events
  - **Interrupt latency**: time between an external hardware event happens and the software interrupt handler is called. This latency includes device handler's execution time
  - **Scheduler latency**: time between a software event and the process execution. This latency includes scheduler's execution time
- Typical scenario: a task is waiting for the GPIO line status change
  - A possible implementation: GPIO interrupt handler should unlock a semaphore and a process is waiting on the GPIO device's semaphore



SOURCES OF INTERRUPT LATENCY

1. **Interrupts are temporally masked by kernel code to implement critical section**
2. Concurrent interrupts are managed per priority/in cascade by kernel
   – « Interrupt storm » is the phenomenon when system is busy due to concurrent interrupts for a long time (>1ms)
3. Shared interrupt provokes the execution of all handlers attached to this line leading to spurious handler execution
   – Good practice: avoid shared interrupt line for time-critical devices

- `cat /proc/interrupts` provides Linux kernel interrupt statististics
  – `cat /proc/xenomai/irq` provides interrupts statistics handled at Xenomai/IPIPE level

```
           CPU0       CPU1       CPU2       CPU3
  0:         44          0          0          0   IO-APIC-edge      timer
  1:         34         45         15        111   IO-APIC-edge      i8042
  5:          1          0          0          0   IO-APIC-edge      parport0
  8:          0          1          0          0   IO-APIC-edge      rtc0
  9:        121        127        127        139   IO-APIC-fasteoi   acpi
 12:       2943       2175       1889       3422   IO-APIC-edge      i8042
 16:        406        357        176        183   IO-APIC  16-fasteoi   ehci_hcd:usb3, snd_hda_int
 17:          3          1          5          4   IO-APIC  17-fasteoi
 20:         11          8          9          7   IO-APIC  20-fasteoi   ehci_hcd:usb4, firewire_oh
 22:         32         25         20         39   IO-APIC  22-fasteoi   mmc0, r592, yenta, r852,
 24:          0          0          1          0   PCI-MSI-edge      xhci_hcd
 25:          0          0          0          0   PCI-MSI-edge      xhci_hcd
 26:          0          0          0          0   PCI-MSI-edge      xhci_hcd
 27:          0          0          0          0   PCI-MSI-edge      xhci_hcd
 28:          0          0          0          0   PCI-MSI-edge      xhci_hcd
 29:         42         46         48   52716122   PCI-MSI-edge      eth0
 30:  118664542  113090410  121994540  125872284   PCI-MSI-edge      0000:00:1f.2
 31:          6          5          8          5   PCI-MSI-edge      mei_me
 33:        259        256   65900723        250   PCI-MSI-edge      iwlwifi
 34:        111        114        112        118   PCI-MSI-edge      snd_hda_intel
 35:   23200076    6144585    3374980   25278853   PCI-MSI-edge      nouveau
NMI:     296372     304240     304983     305661   Non-maskable interrupts
LOC: 2363180159   64001718   64767045   63221366   Local timer interrupts
SPU:          0          0          0          0   Spurious interrupts
PMI:     296372     304240     304983     305661   Performance monitoring interrupts
IWI:          0          0          1          1   IRQ work interrupts
RTR:          0          0          0          0   APIC ICR read retries
RES:   30428454   36871536   40378362   44958253   Rescheduling interrupts
CAL:    1264363    1046028    1216575    1111716   Function call interrupts
TLB:      43609     524393     740618     742043   TLB shootdowns
TRM:          0          0          0          0   Thermal event interrupts
THR:          0          0          0          0   Threshold APIC interrupts
MCE:          0          0          0          0   Machine check exceptions
MCP:       2805       2805       2805       2805   Machine check polls
ERR:          0
MIS:          0
~
```

HOW TO ISOLATE NON-CRITICAL INTERRUPTS FROM REAL-TIME TASKS

- Solution 1: on multicore architectures, the interrupt controller driver can redirect each interrupt source to a specific core
  - e.g. the pseudo file /proc/irq/**9**/smp_affinity contains the interrupt allocation mask for interrupt **9**
  - Default value=0x0F per default on 4-core processor = (0x01 | 0x02 | 0x04 | 0x08)
  - Good practices:
    - Allocate any non-critical interrupt sources to processor cores with no real-time threads
    - Fix each individual critical interrupt sources to specific core to reduce cache pollution of interrupt handlers
- Solution 2: on single core processor, the RTOS can mask non-critical interrupts during real-time processing and RTOS can unmask this interrupt subset during idle time.
  - The Temporal Partitioning scheduler is a generalization of this approach
  - In next chapter, the Xenomai/IPIPE implements this mechanism for critical/non-critical interrupt segregation

SOURCES OF SCHEDULER LATENCY

- Sources of scheduler latency
  1. The **delay** for other software activities before scheduler code is really invoked (see issue in red)
  2. Time to execute the schedule policy, it may depends on number of tasks and data structures (linked-list, bitmap)
  3. Constant time to execute the context switch (registers swap)
- Ideally, the scheduler is ran immediately after any events (hardware interrupts or software signals)
  - However, GPOS may reschedule only in limited conditions
    - E.g. Stock Linux is partially preemptive. Stock Linux allows to reschedule/preempt if the current task was running in userspace. Otherwise, if the running process is executing a system call, the process is in kernel space. Consequently, kernel may have to wait for the end of the current system call to reschedule.

DEVICE DRIVER LATENCY

- To guarantee the **delay** prior scheduler execution, all the kernel code (system calls and interrupt handlers) must have bounded response time
  - System calls handlers or interrupt handlers usually not implemented with care for real-time requirements
  - Expect uncertainties unless you implemented all kernel support code by yourself
- Solution: disable any non-required Linux kernel drivers and fixes the bugs

VIRTUAL MEMORY LATENCY

- **Problem**: process memory is allocated on demand by Linux kernel
  - On first execution, application accesses code or data for the first time, it is loaded on demand, which can creates huge delays
  - Linux virtual memory allocator is **fast**, but does not guarantee a **maximum execution time**
- **Solution**: Lock the whole process address space in RAM at process startup in main() with

```
#include <sys/mman.h>
…
mlockall( MCL_CURRENT | MCL_FUTURE );
```

PROCESSOR PERTURBATIONS

- Modern processors microarchitecture implements different types of caches (L1, L2, prefetch buffer, translation lookaside buffer, branch predictor) and pipeline design
  - The first execution of a task suffers from cold cache effect
  - Interlaced task execution suffers from cache pollution (due to preemption on a single core or due to concurrent execution on multicore)
- External devices suffers from variable execution time
  - DRAM internal refresh mechanism can delay memory access
  - Flash memory access time depends on sequence/unordered memory pattern (row buffer selection)
  - Network links generally suffer from unbounded transmission delay with retry
  - Etc.…
- A practical solution is to consider processor+hardware with no timing variabilities (consider Motorola 68010 1MHz and UART-based communications only)
- An alternate solution is to manage the sources of timing variabilities of your PC ATOM 1.8GHz processor with an estimated overhead
  - The industrial practice considers a **safety margin x2** to the real-time design
    1. Measure the execution time samples of your real-time tasks during a acceptable session’s length and run the schedule analysis for RM/EDF with tasks execution time x2
    2. Measure CPU load and optimize tasks until global CPU load < 50%

REAL-TIME HELLO WORLD

```c
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <sched.h>
#include <sys/mman.h>
#include <string.h>

#define MY_PRIORITY (49) /* we use 49 as the PRREMPT_RT use 50
                            as the priority of kernel tasklets
                            and interrupt handler by default */

#define MAX_SAFE_STACK (8*1024) /* The maximum stack size which is
                                   guaranteed safe to access without
                                   faulting */

#define NSEC_PER_SEC    (1000000000) /* The number of nsecs per sec. */

void stack_prefault(void) {

        unsigned char dummy[MAX_SAFE_STACK];

        memset(dummy, 0, MAX_SAFE_STACK);
        return;
}
```

- Hands-on: use Xenomai user libraries (VxWorks, PSOS, Alchemy or any non-POSIX skins) to reimplement this sample program (sample code from https://rt.wiki.kernel.org/index.php/RT_PREEMPT_HOWTO)

```c
int main(int argc, char* argv[])
{
        struct timespec t;
        struct sched_param param;
        int interval = 50000; /* 50us*/

        /* Declare ourself as a real time task */

        param.sched_priority = MY_PRIORITY;
        if(sched_setscheduler(0, SCHED_FIFO, &param) == -1) {
                perror("sched_setscheduler failed");
                exit(-1);
        }

        /* Lock memory */

        if(mlockall(MCL_CURRENT|MCL_FUTURE) == -1) {
                perror("mlockall failed");
                exit(-2);
        }

        /* Pre-fault our stack */

        stack_prefault();

        clock_gettime(CLOCK_MONOTONIC ,&t);
        /* start after one second */
        t.tv_sec++;

        while(1) {
                /* wait until next shot */
                clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &t, NULL);

                /* do the stuff */

                /* calculate next shot */
                t.tv_nsec += interval;

                while (t.tv_nsec >= NSEC_PER_SEC) {
                       t.tv_nsec -= NSEC_PER_SEC;
                        t.tv_sec++;
                }
   }
}
```

MOTIVATION: STOCK LINUX CONFIG OPTIONS

- Stock Linux contains three preemption strategies
  1. CONFIG_PREEMPT_NONE disables any preemption from kernel code and system calls ; context switching are minimized to enhance system throughput. This is the historic Linux design.
  2. CONFIG_PREEMPT_VOLUNTARY adds explicit rescheduling point into the kernel code and return from system calls. This strategy makes a desktop system reactive enough with minor system throughput penalty.
  3. CONFIG_PREEMPT enables preemption on return from interrupt. User process code and most kernel code can be preempted at any time (except critical sections of kernel code).
- Problem: none of Stock Linux could give <100ms timing guarantee in practice
- Quick question: what is the default preemption strategy of your target machine ?
  - The command « grep PREEMPT /boot/config-* » displays kernel options

PREEMPT_RT DESIGN

PREEMPT_RT FEATURE

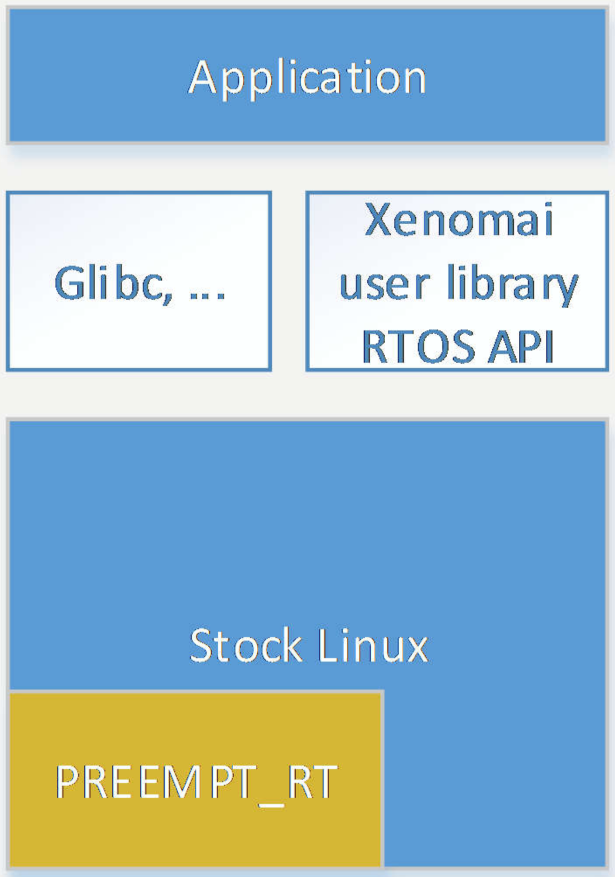
- Historically, « PREEMPT » (**no _RT**) patch from Montavista 1999
  - Remove the big kernel lock and make the system call preemptible
  - Integrated to the Linux kernel mainline nowadays
    - See table (source https://www.osadl.org/Realtime-Linux.projects-realtime-linux.0.html)
- PREEMPT_**RT** modifies Linux kernel code
  - Kernel critical section and kernel-lock primitives are made preemptible by interrupts ; kernel mutex implements priority inheritance protocol
  - Interrupt handler are partly executed as preemptible prioritized kernel threads => driver model is slightly impacted
  - High resolution timers allows fine-grained timeouts to system calls instead of system tick resolution
- Stock Linux kernel is biased toward desktop/server development. Politically, PREEMPT_RT is not scheduled for full integration to the Linux kernel mainline.

Architecture / Feature	x86	x86/64	powerpc	arm	mips	68knommu
Deterministic Scheduler	🟢	🟢	🟢	🟢	🟢	🟢
Preemption Support	🟢	🟢	🟢	🟢	🟢	🟢
PI Mutexes	🟢	🟢	🟢	🟢	🟢	🟡
High-Resolution Timer	🟢	🟢	🟢	🟢	🟢	🟢
Preemptive Read-Copy Update	🟢	🟢	🟢	🟢	🟢	🟢
IRQ Threads	🟢	🟢	🟢	🟢	🟢	🟡
Raw Spinlock Annotation	🟢	🟢	🟢	🟢	🟢	🟢
Forced IRQ Threads	🟢	🟢	🟢	🟢	🟢	🟢
R/W Semaphore Cleanup	🟢	🟢	🟢	🟢	🟢	🟢
Full Realtime Preemption Support	🟡	🟡	🟡	🟡	🟡	🟡

🟢 Available in mainline Linux
🟡 Available when Realtime Preempt patches applied

PREEMPT_RT DESIGN

- PREEMPT_RT is a patch (gzipped 200kB) applied to Linux kernel
  - The PREEMPT_RT Linux kernel is binary compatible, underlying modifications are transparent to applicative and user libraries
  - PREEMPT_RT does not extend Linux nor adds system calls for real-time applications
- Pros: real-time Linux behavior comes (almost) for free to Linux application
- Cons: Bad (evil) drivers or services (like filesystem) can continue to break real-time behavior of the system
- The Linux kernel included by third party vendor (Ubuntu, etc.) in desktop/server distribution is NEVER applicable to hard real-time systems
- Solution: a real-time system designer shall
  1. Limit the number of drivers and services included to Linux kernel build
  2. Rewrite buggy-but-desirable device drivers or kernel services



PREEMPT_RT USAGE

- Historically, PREEMPT_RT is
  - x86-based effort but ARM is now a mature target
  - Popular for high-frequency trading community
- PREEMPT_RT can provide 1ms hard real-time constraints
  - Performance depends on architecture performance and setup (=disable dynamic frequency scaling)
  - Determinism depends on drivers set included in kernel build (=pay attention to GPU driver)
- Pros:
  - Linux default toolchain, monitoring/profiler tools are compatible with Linux/PREEMPT_RT
  - Regular driver model
- Cons:
  - Complex piece of engineering
  - Single real-time API (POSIX), require to port existing real-time application to POSIX API
- Today, PREEMPT_RT is the platform of choice for media streaming, robotics, … since timing constraints is of 1ms scale

PREEMPT_RT INSTALLATION

- Installation procedure example for Ubuntu 14.04
  1. `sudo apt-get install libncurses5-dev kernel-package`
  2. `sudo apt-get build-dep --no-install-recommends linux`
  3. `wget https://www.kernel.org/pub/linux/kernel/v3.x/linux-3.18.36.tar.gz`
  4. `tar xzvf linux-3.18.20.tar.gz && cd linux-3.18.20`
  5. `wget https://www.kernel.org/pub/linux/kernel/projects/rt/3.18/older/patch-3.18.20-rt18.patch.gz`
  6. `gzip -cd patch-3.18.20-rt18.patch.gz |patch -p1`
  7. `make oldconfig   #Activate « 5. Fully Preemptible Kernel (RT) (PREEMPT_RT_FULL) (NEW)”`
  8. `make menuconfig #Disable «  Hacking – Torture » and undesirable feature if possible`
  9. `sudo CONCURRENCY_LEVEL=5 CLEAN_SOURCE=no fakeroot make-kpkg --initrd --append-to-version -preempt-rt18 --revision 1.0 kernel_image kernel_headers`
  10. `sudo dpkg –i ../*.deb`

HANDS-ON: WHAT IS THE REAL-TIME BEHAVIOR OF YOUR PREMPT_RT SYSTEM ?

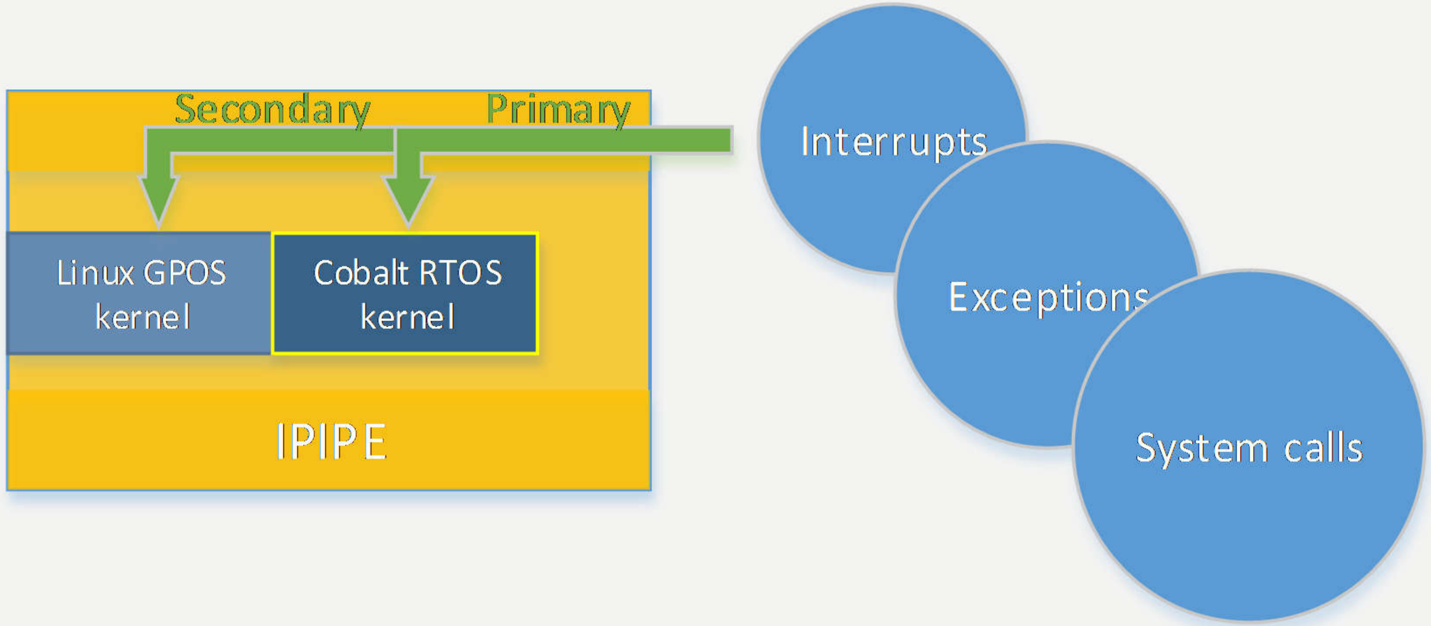
- Build a Linux kernel with PREEMPT_RT patch for a target machine (Desktop, Laptop, Raspberry PI, Galileo)
- Build Xenomai 3 for a **mercury** flavor (this should have already be done since Slides part 1 ; no need to rebuild)
- Launch benchmark « `latency -h -g log.txt` » (depends on your installation; this could be in /usr/bin/latency)
- During benchmark execution, activates different features of the target machine (buttons, video, sound, network)
- Stop the latency measurement with CTRL-C
  - In microseconds, what is the worst-case latency of your system ?
  - Plot the histogram stored in text file log.txt.

IPIPE DESIGN

IPIPE = INTERRUPT PIPELINE

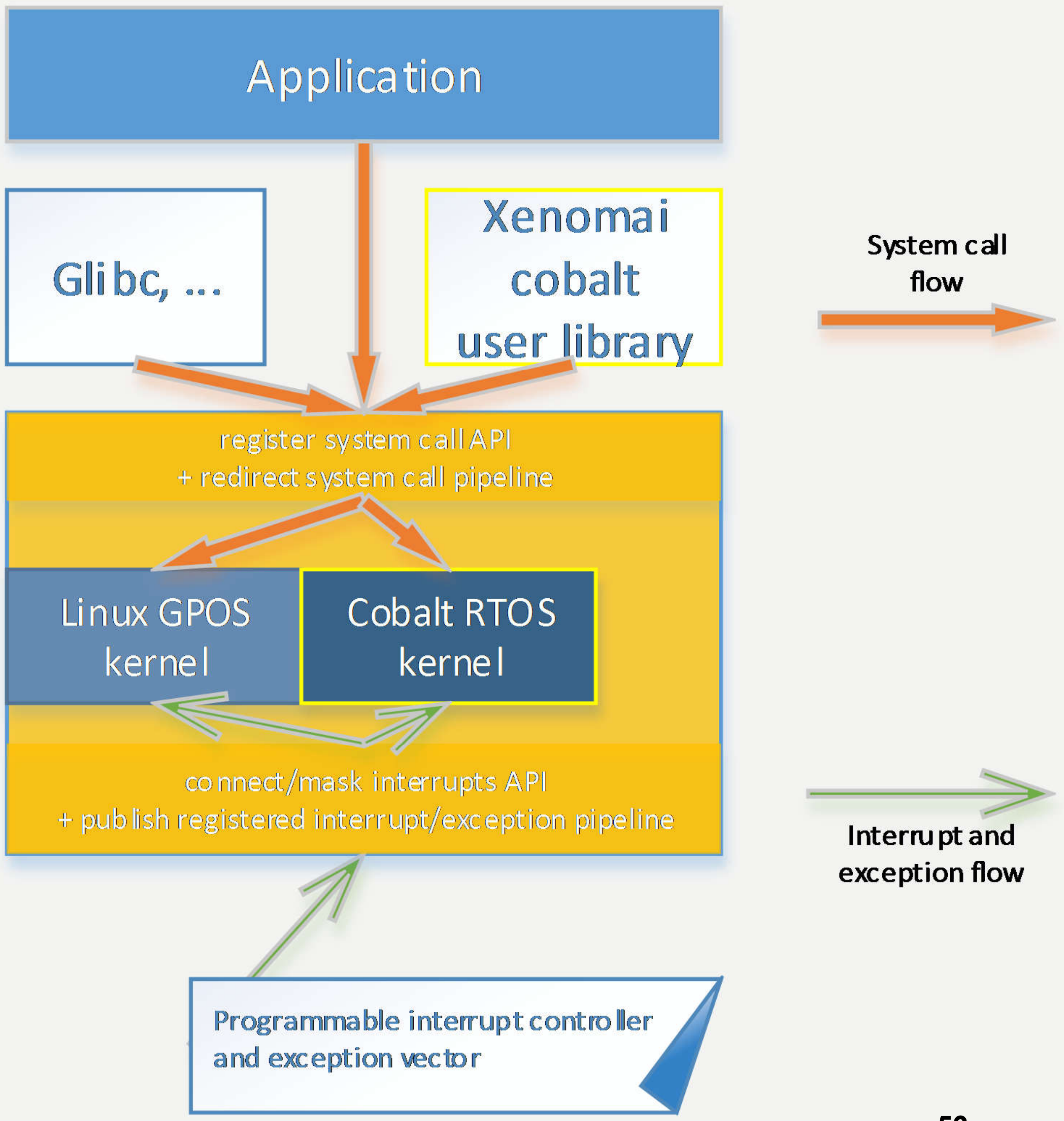
IPIPE FEATURE

- IPIPE (for Interrupt PIPEline) organize the system into prioritized domains
  - Xenomai/Cobalt RTOS is the highest priority domain
  - Linux GPOS is the [ROOT] domain
- IPIPE dispatches events to each domain
  - Events are hardware interrupts, system calls and exceptions
  - Domains have to register for receiving specific events, IPIPE is in charge to dispatch events in priority order
- IPIPE provides an architectural neutral API-based BSP
  - Support for Powerpc32/64, x86_32/64, ARM, Blackfin, nios2



IPIPE DESIGN

- IPIPE is a patch (gzipped 100kB) applied to Linux kernel
  - IPIPE fixes Linux critical section to preemptible lock (similarly to PREEMPT_RT)
  - IPIPE allows to plug a new set of system calls for real-time applications
  - IPIPE allow to dispatch incoming interrupts or exception to target RTOS/GPOS kernel
- Cobalt is a complete RTOS (gzipped 240kB) that implements specific RTOS services (e.g. scheduler policies)
  - Cobalt is the codename for Xenomai/IPIPE RTOS kernel software
  - Cobalt provides Real-Time Driver Model API (RTDM) to implement time-critical device drivers



IPIPE USAGE

- Historically, Microsoft Windows platforms supports dual kernel approaches to mix real-time constraints (from RTOS kernel) and User interface (from Windows kernel)
  - Since ‘90: Intel iRMX/Windows, INTime derivatives, Ardence RTX
  - On Linux, RTLinux (1997-2005), (RTAI, since 1999), (Xenomai, since 2001)
- Xenomai/IPIPE provides 100us hard real-time constraints
  - Performance depends on architecture performance and setup, but IPIPE patch warns you at build or boot time for hardware related timing critical issues
    - Option dynamic frequency scaling is warned in « make menuconfig » display
    - Software workarounds are provided to disable dangerous features (SMI on Intel chipsets)
  - Excellent determinism, you can put any Linux device drivers you need (GPU drivers etc.) except evil code disabling interrupts manually in a third party closed binary module
- Pros:
  - Real-time tasks are generally immune from Linux kernel mainline regression
  - Cobalt RTOS kernel has similar code complexity to conventional RTOS (e.g. VxWorks) but lower complexity than a Linux with PREEMPT_RT
- Today, Xenomai is the platform of choice for robotics, production control… since timing constraints is of 100us scale

IPIPE INSTALLATION

- Installation procedure example for Ubuntu 14.04

  1. sudo apt-get install devscripts debhelper dh-kpatches checkinstall # **debian packaging tools**
  2. wget http://xenomai.org/downloads/xenomai/stable/latest/xenomai-3.0.2.tar.bz2
  3. tar xjvf xenomai-3.0.2.tar.bz2
  4. cd xenomai-3.0.2 && DEBEMAIL="your@email" DEBFULLNAME="Your Name" debchange -v 3.0.2 Release 3.0.2 && debuild -uc –us
  5. cd .. && sudo dpkg -i *deb
  6. wget https://www.kernel.org/pub/linux/kernel/v3.x/linuµx-3.18.20.tar.gz
  7. tar xzvf linux-3.18.20.tar.gz && cd linux-3.18.20
  8. /usr/src/xenomai-kernel-source/scripts/prepare-kernel.sh --arch=x86 --ipipe=/usr/src/xenomai-kernel-source/kernel/cobalt/arch/x86/patches/ipipe-core-3.18.20-x86-6.patch
  9. make oldconfig # **reply with default choices**
  10. make menuconfig # **disable undesirable feature (dynamic clock frequency…)**
  11. sudo CONCURRENCY_LEVEL=5 CLEAN_SOURCE=no fakeroot make-kpkg --initrd --append-to-version -xenomai-3.0.2 --revision 1.0 kernel_image kernel_headers
  12. cd .. && sudo dpkg -i *deb

```
# Disable these stuffs in make menu
- Processor type and features
 [ ] Allow for memory compaction
 [ ] Transparent Hugepage Support
 [ ] Contiguous Memory Allocator
 [ ]   Page migration
- Power management and ACPI option
  [ ] Cpu Frequency scaling
  [ ] Cpuidle Driver for Intel Proc
- ACPI (Advanced Configuration and
    < > Processor
- Device Drivers
  [ ] Staging drivers
```

HANDS-ON: WHAT IS THE REAL-TIME BEHAVIOR OF YOUR IPIPE SYSTEM ?

- Build a Linux kernel with IPIPE patch and Xenomai/IPIPE for a target machine (Desktop, Laptop, Raspberry PI, Galileo)
- Build Xenomai 3’s **cobalt** user library
- Launch benchmark « `latency -h -g log.txt` » (depends on your installation; this could be in **/usr/lib/xenomai/testsuite/latency**)
- During benchmark execution, activates different features of the target machine (buttons, video, sound, network)
- Stop the latency measurement with CTRL-C
  - In microseconds, what is the worst-case latency of your system ?
  - Plot the histogram stored in text file log.txt.

XENOMAI USER LIBRARIES

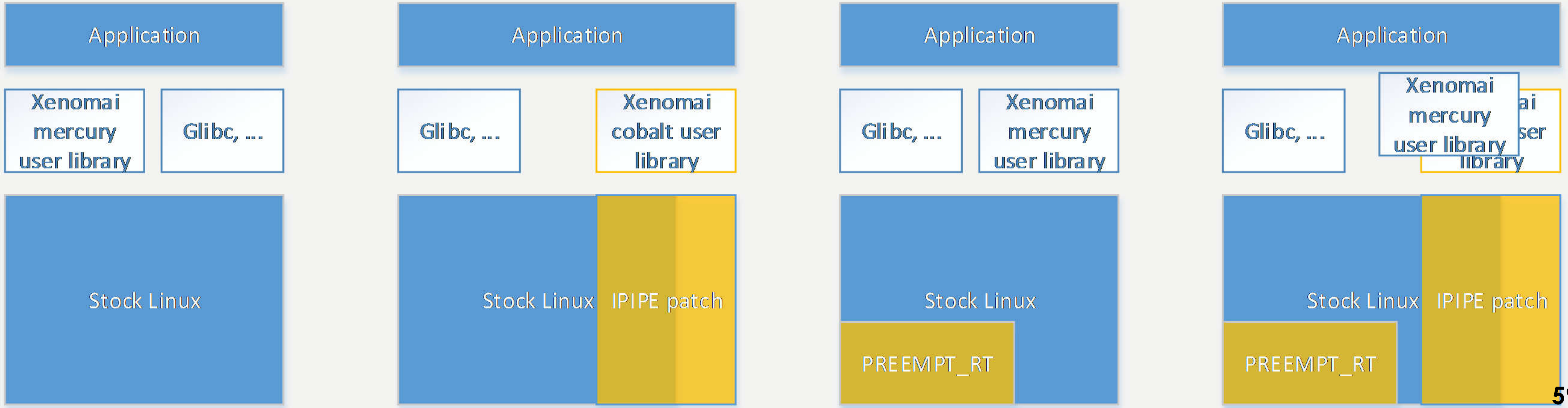
MERCURY AND COBALT

XENOMAI FEATURES

- VxWorks, QNX, PSOS, UITRON, VRTX have installed software codebase for the last 20 years on hardware boards…. of 20 years-old
- Problem: port of existing RTOS code to stock Linux on new hardware is complex due to
  1. Requirement to rewrite calls to RTOS-specific API to Linux API invocation
  2. Lack of real-time determinism (lock blocking time and interrupt latency) or compatible scheduling policies
  3. Process memory protection of user process in Linux, while most RTOS have flat address space
     - Need to clean up the code from inline assembly code
     - Need to write « real » device drivers instead of direct registers updates from application code
- Xenomai provides
  1. RTOS API are emulated through user libraries (**Xenomai/Mercury and Xenomai/Cobalt**) ; application can mix various API together (e.g. POSIX and VxWorks)
  2. Linux patch (**IPIPE**) and its own RTOS kernel code (**Cobalt**) to reach desirable determinism
  3. Real-Time Driver Model (**RTDM**) to implement time-critical device driver in Linux kernel

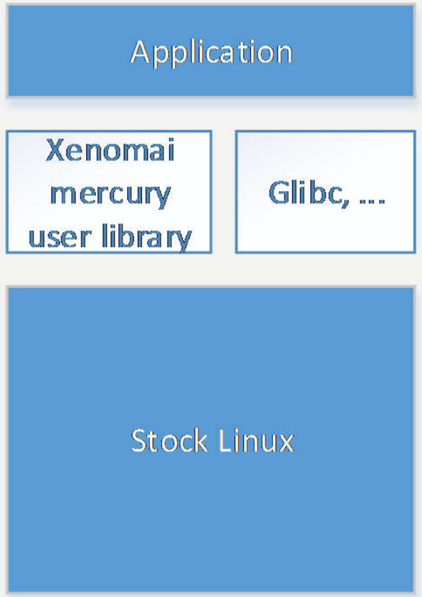
XENOMAI DESIGN

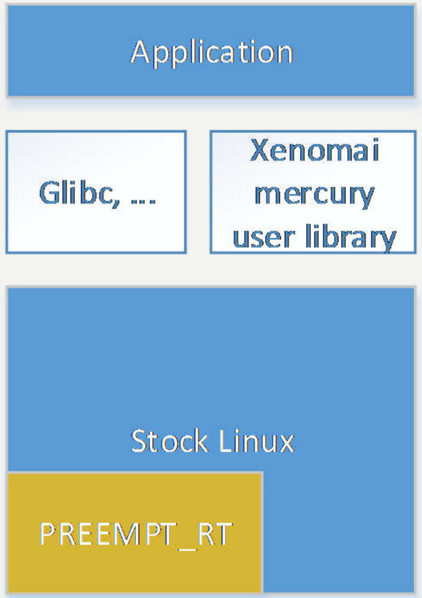
- Xenomai 3 provides a set of user libraries to emulate common RTOS API (VxWorks, PSOS, POSIX RT extension)
  - Application is built against Xenomai library with script $(xeno-config –cflags –ldflags –vxworks)
  - Application invokes POSIX, VxWorks API… from linked Xenomai user library
    - Under the hood: Xenomai/Mercury user library is connected to Linux system calls
      - Xenomai/Cobalt is connected to Linux/IPIPE system calls to emulate RTOS API behavior



XENOMAI/MERCURY PROCESS MODEL

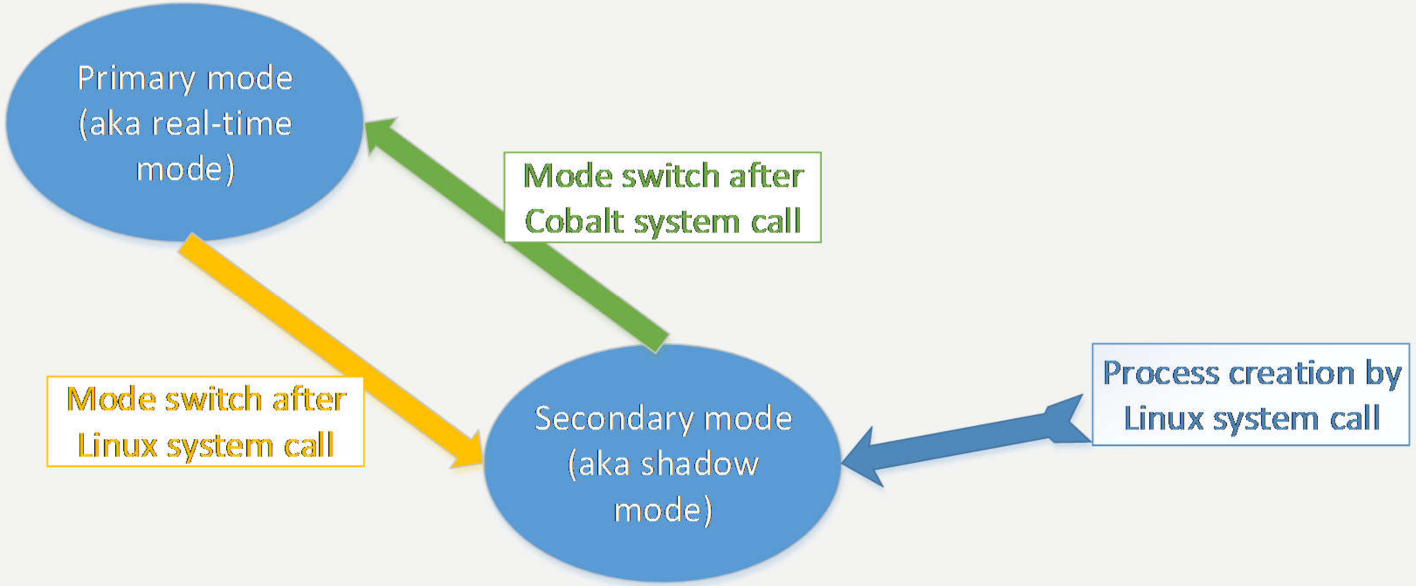
- (Note: in Linux, thread = process)
- Application is Linux process linked to the Xenomai « mercury » user library
  - Application can call any Linux GPOS services and any Linux device drivers
  - Application can call RTOS API directly emulated from Linux system calls
- The real-time performance of Xenomai/Mercury process depends on Stock Linux configuration or PREEMPT_RT patch





XENOMAI/COBALT PROCESS MODEL

- Application is linked to the Xenomai « libcobalt » user library
  - Xenomai/Cobalt process is running over a dual kernel Linux GPOS+Cobalt RTOS
- The Xenomai/Cobalt process can switch between these two kernels anytime
  - Secondary mode: all Linux GPOS services and Linux device drivers are accessible
  - Primary mode: all Xenomai RTOS services and **RTDM** device drivers are accessible
- The Xenomai/Cobalt process transparently switches between Primary or Secondary mode after a system call to Linux or Xenomai system call respectively



XENOMAI/COBALT SCHEDULING

- In Primary mode, the process is scheduled by Cobalt RTOS scheduler
  - The Cobalt RTOS scheduling policy and priority apply to the process
    - Process in Primary mode can only be preempted by interrupts managed by the Cobalt kernel
    - Interrupts of non time-critical devices (WLAN, SATA) managed by Linux kernel are masked in Primary mode
- In Secondary mode, the process is scheduled by Linux scheduler
  - The Linux GPOS scheduling policy and priority apply to the process. The process can be preempted by any interrupts and processes in Primary mode
  - Linux kernel is the idle task of Cobalt kernel, i.e. processes in Secondary mode can run if Cobalt kernel is idle

PROCESS MODE SWITCHING

- Mode switching is an automatic way for a Xenomai/Cobalt process to access to non real-time resources managed by Linux kernel (filesystem, network, USB etc.)
  - At startup, real-time tasks typically read configuration files and set up internal data in Secondary mode prior real-time activities in Primary mode
- The mode switch is partly realized by the **gatekeeper** process. This gatekeeper runs in Secondary mode and the execution time is > 50ms per mode switch
  - Register set is preserved across mode switch
  - **A real-time task shall avoid whenever possible any mode switch at runtime !**
    - Steps for Primary to Secondary mode switch
      1. Cobalt kernel suspends the process from RTOS scheduling table
      2. A virtual interrupt is sent through IPIPE to Linux kernel
      3. Since Cobalt kernel runs in idle mode, Linux kernel is awaken
      4. Linux kernel receive the (virtual) interrupt and IRQ handler calls the procedure wake_up_handler() to reschedule the « shadow » process into Linux schedule table

HOW TO DETECT UWANTED MODE SWITCH

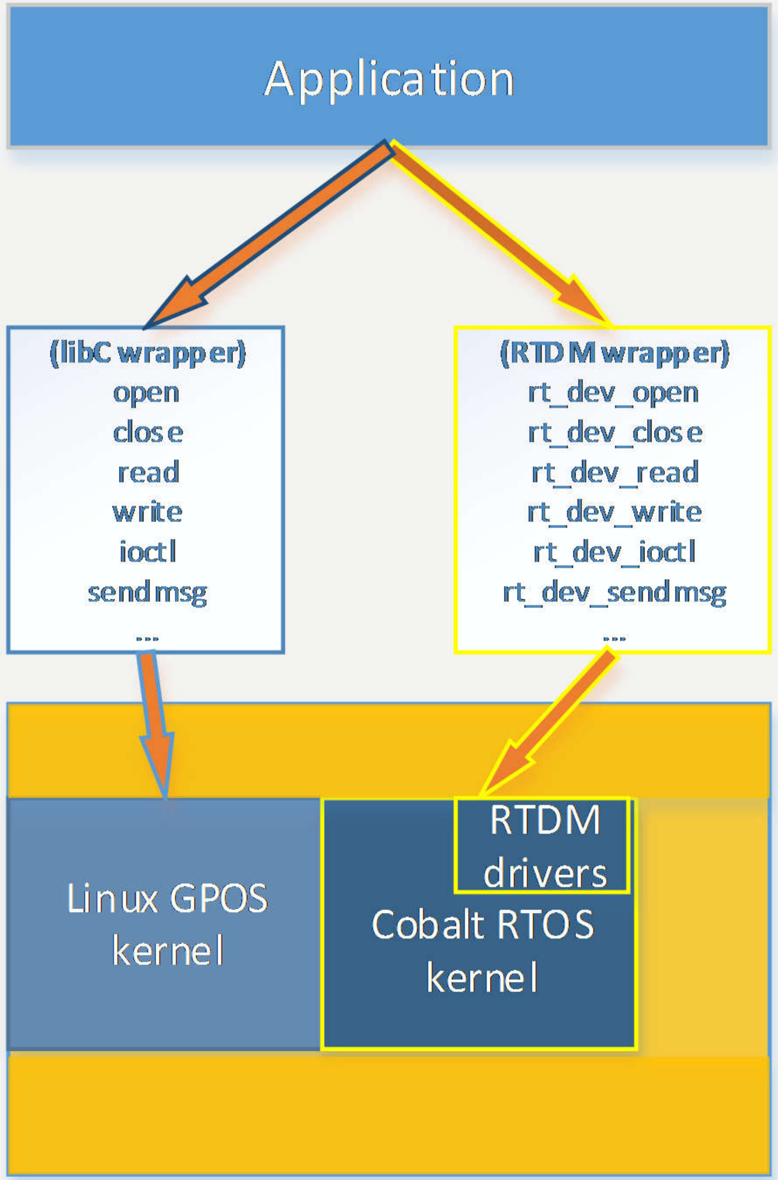
1. User program can automatically detect Primary to Secondary mode switch by catching the SIGXCPU signal
   - Register a signal hook: `signal(SIGXCPU, warn_upon_switch )`
   - The user supplied `warn_upon_switch` can display the stack backtrace
2. Xenomai's **sbin/slackspot** can monitor any mode switches and display a stack backtrace
   - See complete tutorial https://xenomai.org/2014/06/finding-spurious-relaxes
3. The number of mode switch (MSW) per process is displayed in file /proc/xenomai/sched/stat
   - $ cat /proc/xenomai/sched/stat

CPU	PID	**MSW**	**CSW**	XSC	PF	STAT	%CPU	NAME
0	0	0	2304565812	0	0	00018000	100.0	[ROOT/0]
1	0	0	460889	0	0	00018000	100.0	[ROOT/1]
2	0	0	0	0	0	00018000	100.0	[ROOT/2]
3	0	0	28	0	0	00018000	100.0	[ROOT/3]
3	28818	**1**	**1**	5	0	000600c0	0.0	latency
3	28820	**7**	**14**	11	0	00060042	0.0	display-28818
0	28821	**2**	**70447**	70457	0	0004c042	0.0	sampling-28818
1	0	0	72998268	0	0	00000000	0.0	[IRQ16641: [timer]]
2	0	0	72270038	0	0	00000000	0.0	[IRQ16641: [timer]]
3	0	0	74427478	0	0	00000000	0.0	[IRQ16641: [timer]]

- **Quick question:** what does mean **MSW** and **CSW** for the real-time process ? Is it good for MSW and CSW to vary over time (or not) ?

REAL-TIME DRIVER MODEL (RTDM)

- Historically, RTDM is an API to implement portable drivers between RTAI and Xenomai
- RTDM is a kernel infrastructure to implement time-critical device driver in Linux kernel
  - RTDM supplements the Linux driver API to improve drivers determinism
- RTDM drivers are regular kernel modules (.ko)
  - Internally, RTDM drivers must use Cobalt RTOS kernel services for thread spawn, alarms and synchronization services to obtain timing guarantees
- Support for UARTs, GPIOs, analog/digital converters, CAN, Ethernet devices



RTDM SUPPORT LIBRARY

- Build RTDM device driver requires Linux/IPIPE and Cobalt kernel runtime environment.
  - Xenomai/Cobalt application have native access to RTDM devices
- In the future, RTDM drivers would be portable to Linux/PREEMPT_RT environments
  – Today, Xenomai/Mercury provides an user library (libcopperplate) to enable Linux access to RTDM devices

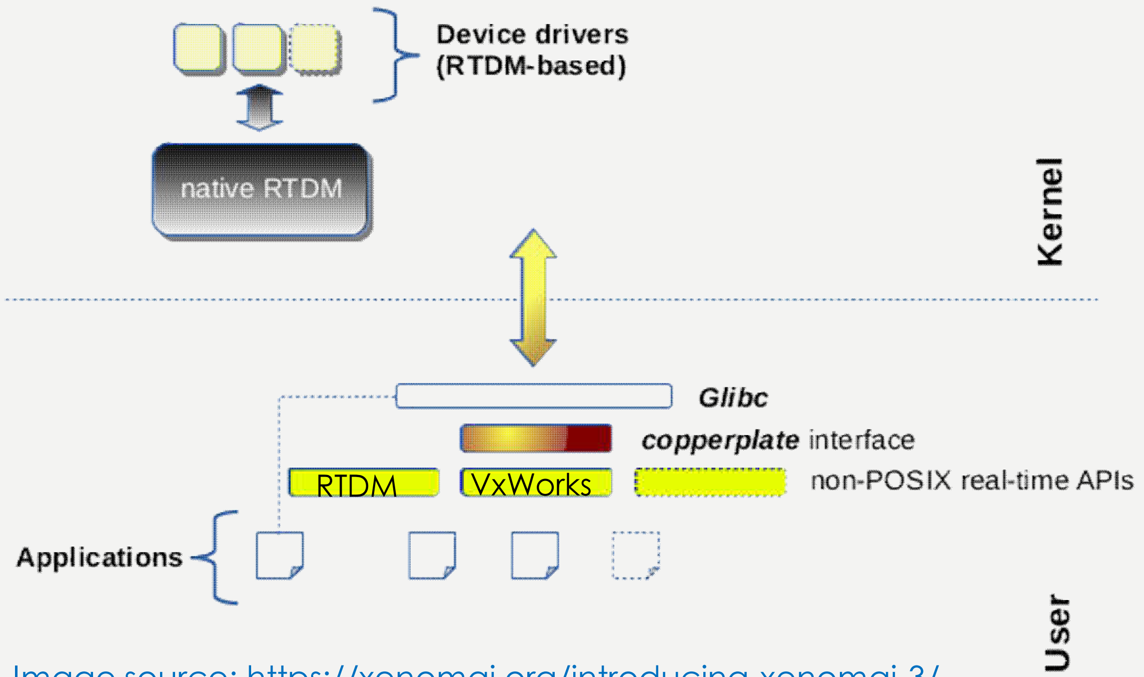


Image source: https://xenomai.org/introducing-xenomai-3/

HELLO WORLD FOR XENOMAI/COBALT

- Hands-on: rebuild your program for Xenomai/Cobalt and launch the program. Find the file in directory /proc/xenomai/ with CPU usage of your program.
- Quick question: in file « stat », does MSW/CSW values change over time ?

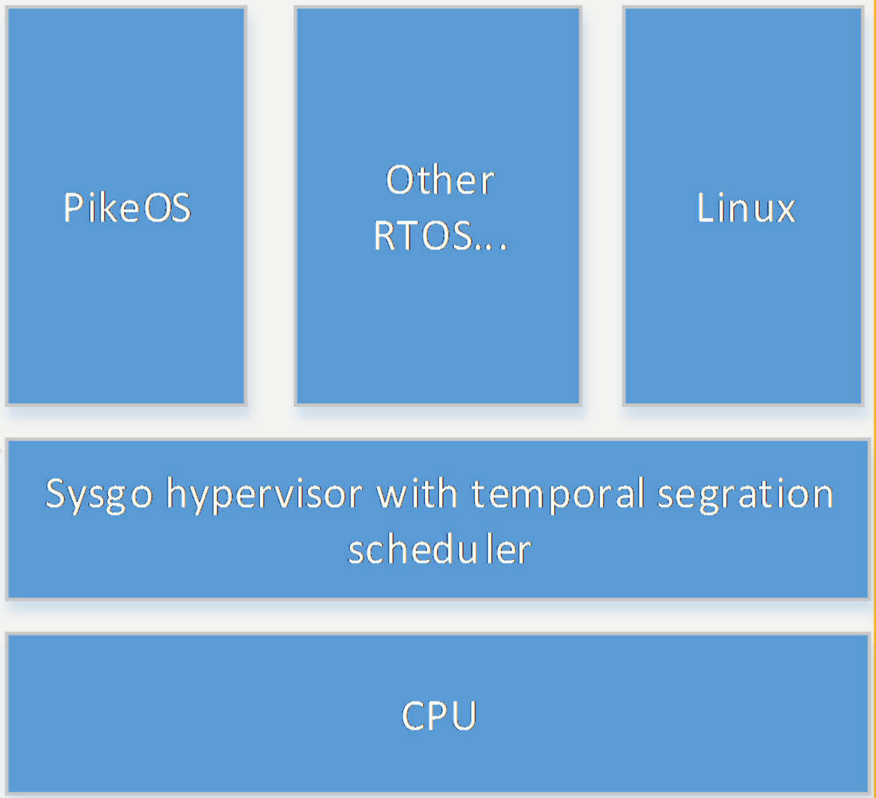
```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <sched.h>
#include <sys/mman.h>
#include <string.h>

#define MY_PRIORITY (49) /* we use 49 as the PRREMPT_RT use 50
                            as the priority of kernel tasklets
                            and interrupt handler by default */

#define MAX_SAFE_STACK (8*1024) /* The maximum stack size which is
                                   guaranteed safe to access without
                                   faulting */

#define NSEC_PER_SEC    (1000000000) /* The number of nsecs per sec. */

void stack_prefault(void) {

        unsigned char dummy[MAX_SAFE_STACK];

        memset(dummy, 0, MAX_SAFE_STACK);
        return;
}
int main(int argc, char* argv[])
{
        struct timespec t;
        struct sched_param param;
        int interval = 50000; /* 50us*/

        /* Declare ourself as a real time task */

        param.sched_priority = MY_PRIORITY;
        if(sched_setscheduler(0, SCHED_FIFO, &param) == -1) {
                perror("sched_setscheduler failed");
                exit(-1);
        }

        /* Lock memory */

        if(mlockall(MCL_CURRENT|MCL_FUTURE) == -1) {
                perror("mlockall failed");
                exit(-2);
        }
```

RELATED COMPETITORS

IN ONLY TWO SLIDES

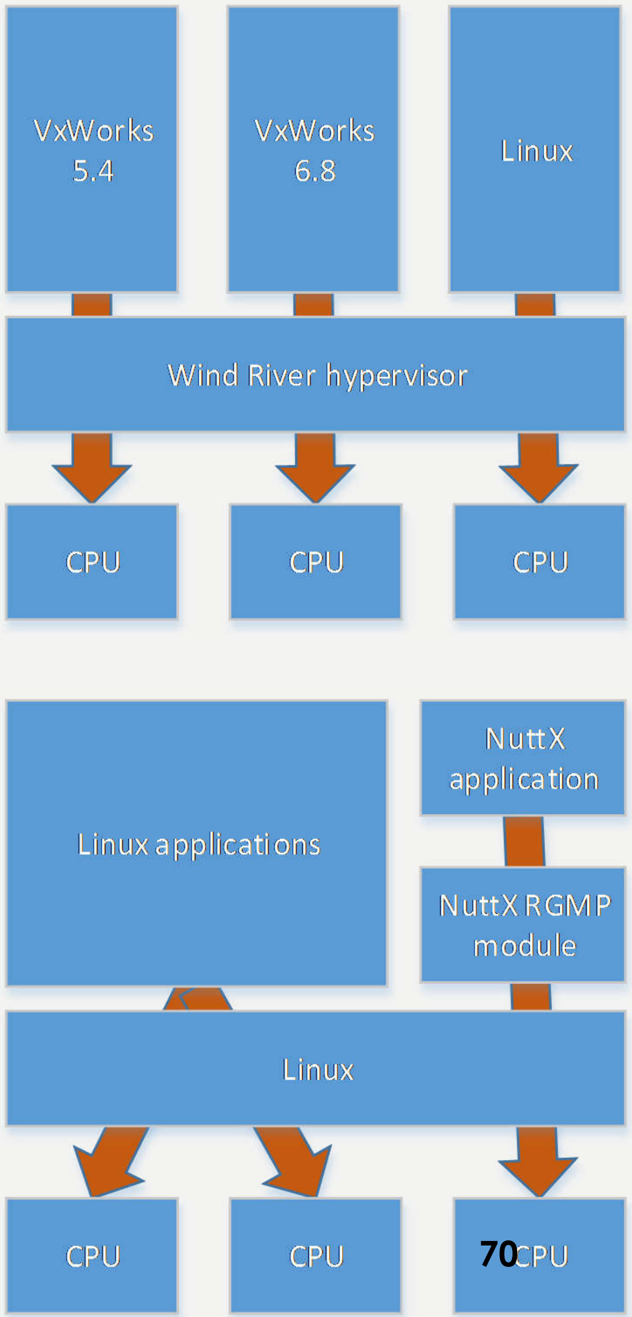
VIRTUALIZATION

- Hypervisor boots the system and allocates resources to virtualized environments
  - Ex: Sysgo hypervisor approach
    - On a given CPU core, RTOS (PikeOS) runs in a predefined temporal slot; Linux VM (or other RTOS) instances are allocated to remaining slot. CPU underload from one slot is given dynamically to others slots.
    - Communication between different RTOS/GPOS is possible through Hypervisor's IPC (socket-like)
    - Pros: safe CPU sharing between multiple RTOS or GPOS ; isolation of RTOS kernel memory space and Linux guest
    - Cons: hypervisor needs a complete BSP; need specific toolchain for each environment
  - Note: Hypervisor has design similarities to Xenomai's IPIPE patch: interrupt are first handled by hypervisor and next reported to guest OS.



ASYMMETRIC MULTIPROCESSING (AMP)

- Multicore is logically partitioned between RTOS and GPOS
  - Ex 1: Wind River hypervisor approach
    - Similar to Sysgo Hypervisor except each core runs only one RTOS/GPOS instance
    - Pros: isolation of RTOS kernel memory space and Linux guest
    - Cons: hypervisor needs a complete BSP and different toolchains for each RTOS/GPOS
  - Ex 2: “RTOS and GPOS on Multi-Processors” project (RGMP)
    - The system boots on normal Linux installation, but user shall reserve cores and memory space from Linux bootline. RGMP is a normal Linux module with a framework to port a RTOS
    - Pros: no need for specific RTOS BSP, the system boots on Linux system with a unique toolchain for all environments.
    - Cons: RTOS has to be ported to RGMP framework



CONCLUSIONS

QUICK GUIDE TO PORT/DEVELOP REAL-TIME SOFTWARE TO LINUX

HOW TO PORT AN EXISTING REAL-TIME SOFTWARE TO LINUX

- The application is based on VxWorks, pSOS… API ?
    - Xenomai user library user library is a big help for source code adaptation to Linux
    - Timing requirements are **over 100ms** ? Xenomai on Stock Linux is proper
    - Timing requirements are **over 1ms** ? Xenomai and Linux/PREEMPT_RT are required
    - Timing requirements are **over 100us** ? Xenomai and Linux/IPIPE provides the required performance
- The application is based on POSIX RT API ?
    - Application can be ported directly to Linux if timing requirements are **over 1ms** (Linux/PREEMPT_RT) or **over 100ms** (Stock Linux)
    - Timing requirements are **over 100us** ? Xenomai and Linux/IPIPE provides required performance
- Time-critical device drivers must be ported to Xenomai/RTDM if these devices are involved into time-critical processing with timing requirements **under 1ms.**

HOW TO DEVELOP A NEW REAL-TIME APPLICATION TO LINUX

- While POSIX RT could be largely improved (complexity), this API is complete
  - I recommend to develop new software over POSIX RT API; main RTOS competitors supports POSIX RT API, it could help to share software codebase
- (Again) Development of real-time software to Linux requires good knowledge of its timing requirements
  1. Timing requirements are **over 100ms** ? Stock Linux is proper
  2. Timing requirements are **over 1ms** ? Linux with PREEMPT_RT patch. Third party binary drivers shall be avoided due to possible interrupt locking by driver code. **Need careful selection of Linux kernel drivers.**
  3. Timing requirements are **over 100us** ? Linux with IPIPE patch. Third party binary drivers shall be avoided due to possible direct interrupt locking by driver code
- (Again) Time-critical device drivers must be developed to Xenomai/RTDM if these devices are involved into time-critical processing with timing requirements **under 1ms.**