Real Time Operating Systems

Kunal, Mayank, Mohit
Nilay, Rahul

What is a Real Time System ?

- Capable of guaranteeing timing requirements of the processes under its control
- Fast – low latency
- Predictable – able to determine task's completion time with certainty
- Both time-critical and non time-critical tasks to coexist

Hard vs Soft

- A **hard real-time system** guarantees that real-time tasks be completed within their required deadlines.
- Requires formal verification/guarantees of being to always meet its hard deadlines (except for fatal errors).
- Examples: air traffic control , vehicle subsystems control, medical systems.
- A **soft real-time system** provides priority of real-time tasks over non real-time tasks. Also known as "**best effort**" systems. Example – multimedia streaming, computer games.

Classification

- The type of system can be either a uni-processor, mp or distributed system.
- Moving to an mp or distributed real time systems adds lots of difficulty by requiring real time communication and synchronization mechanisms between the processors.

Classification

- There are two different execution models:
- In a **preemptive** model of execution a task may be interrupted (preempted) during its execution and another task run in its place.
- In a **non-preemptive** model of execution after a task that starts executing no other task may execute until this task concludes or yields the CPU.

Classification

- A **static** system is a system where all tasks are knows at design time including their release times (full apriory knowledge).
- A **dynamic** systems is a system that can dynamically create and destroy tasks at runtime (No full apriory knowledge).

Classification

- In many real time systems the tasks have different priorities.

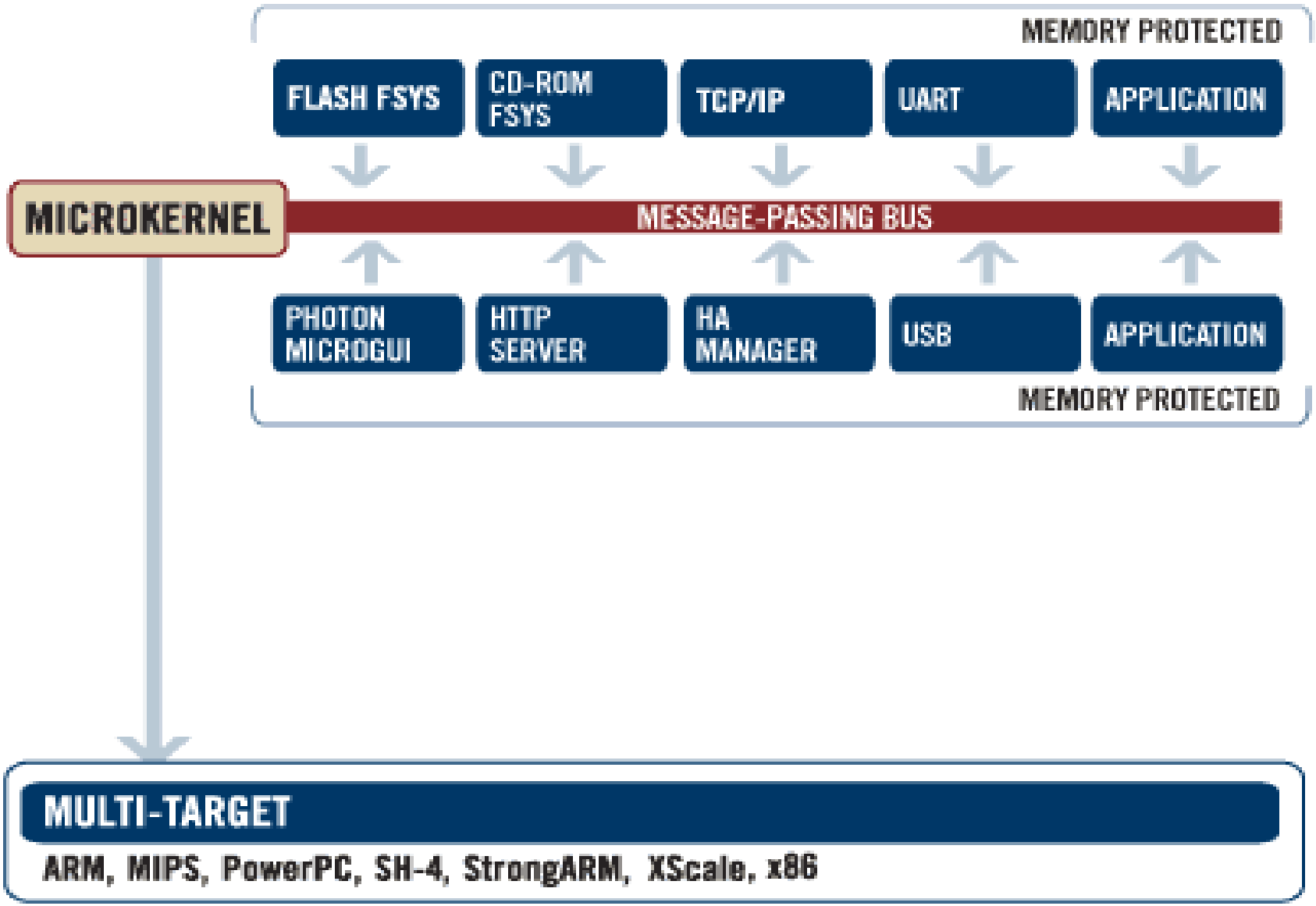
There are two possible models :

- **static-priorities :** the priorities of tasks don’t change during execution
- **dynamic-priorities** the priority of tasks may change during execution

Characteristics

- Single purpose
- Small size
- Inexpensively mass-produced
- Specific timing requirements

QNX


MEMORY PROTECTED
FLASH FSYS
CD-ROM FSYS
TCP/IP
UART
APPLICATION
MICROKERNEL
MESSAGE-PASSING BUS
PHOTON MICROGUI
HTTP SERVER
HA MANAGER
USB
APPLICATION
MEMORY PROTECTED
MULTI-TARGET
ARM, MIPS, PowerPC, SH-4, StrongARM, XScale, x86

Features Missing in RTOS

- Support for variety of peripheral devices.
- Protection and Security mechanisms
- Multiple Users
- Multiple Modes
- Dynamic Allocation of memory

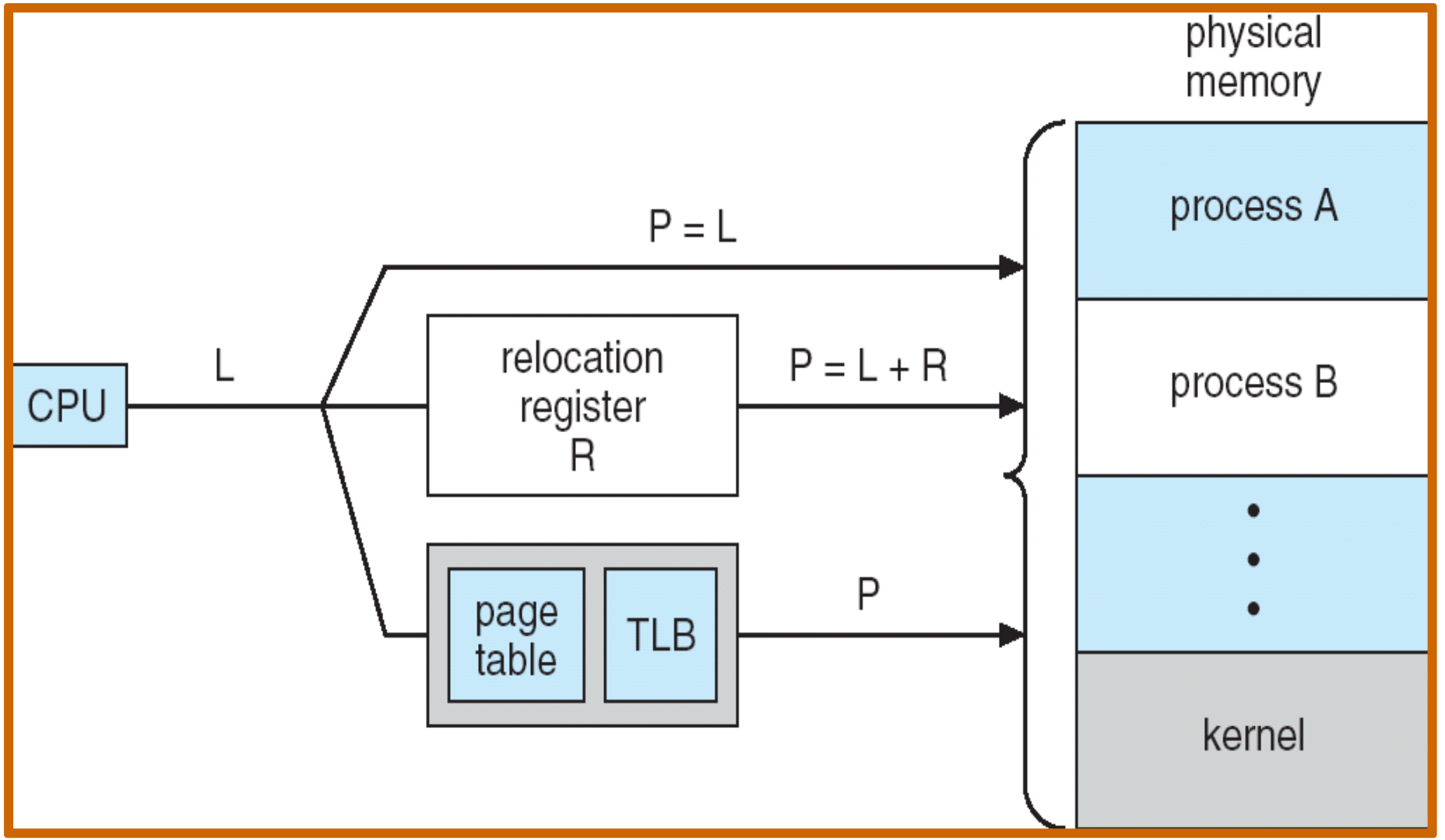
Reasons

- Real-time systems are typically single-purpose.
- Real-time systems often do not require interfacing with a user.
- Features found in a desktop PC require more substantial hardware that what is typically available in a real-time system.
- High overhead required for protected memory and for switching modes.
- Memory paging increases context switch time.
- Creates fragmentation adding to timing unpredictability.

Memory Management

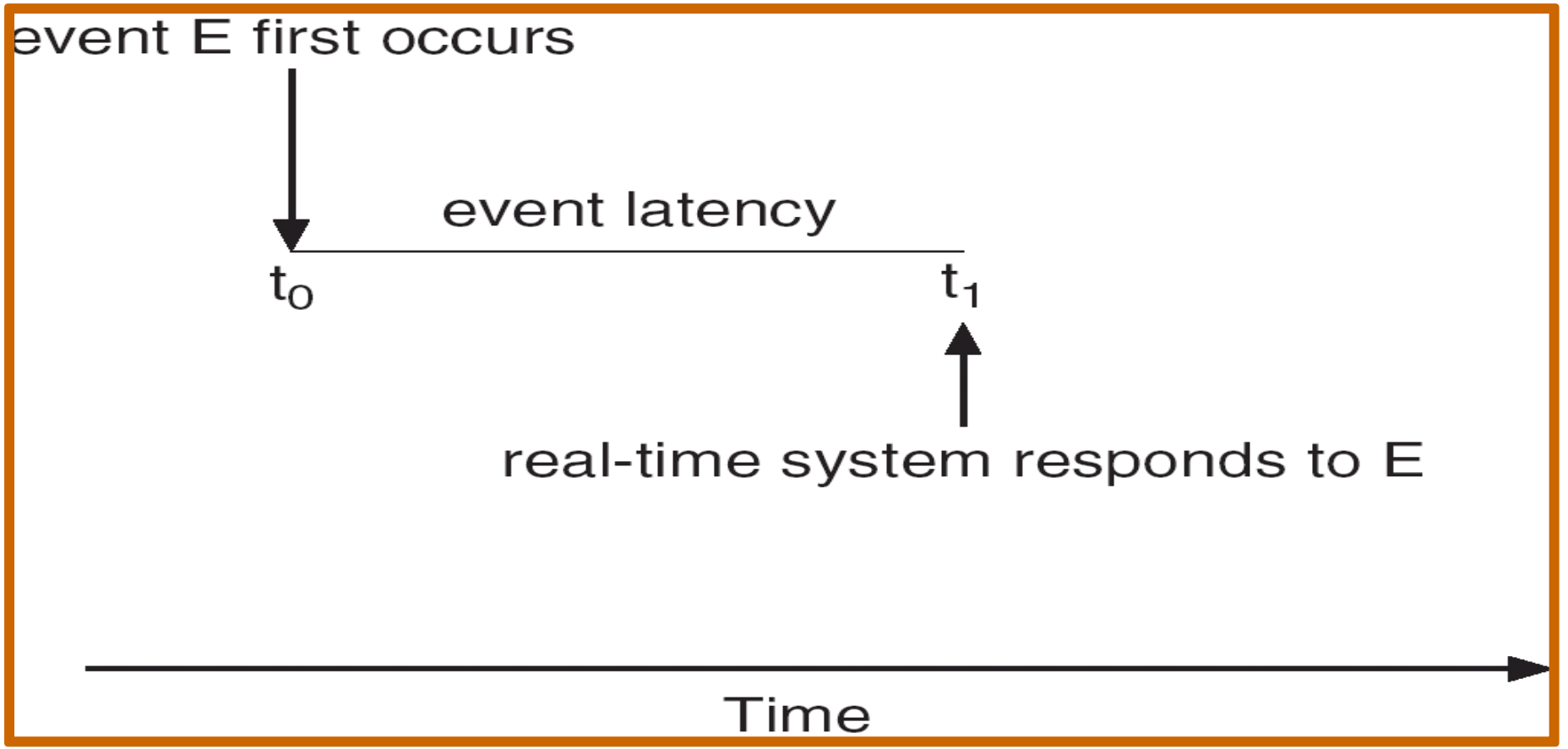
- Address translation may occur via --
  - **Real-addressing mode** where programs generate actual addresses.
  - **Relocation** register mode.
  - Implementing full **virtual memory**. Example LynxOS and OnCore Systems.

Address Translation

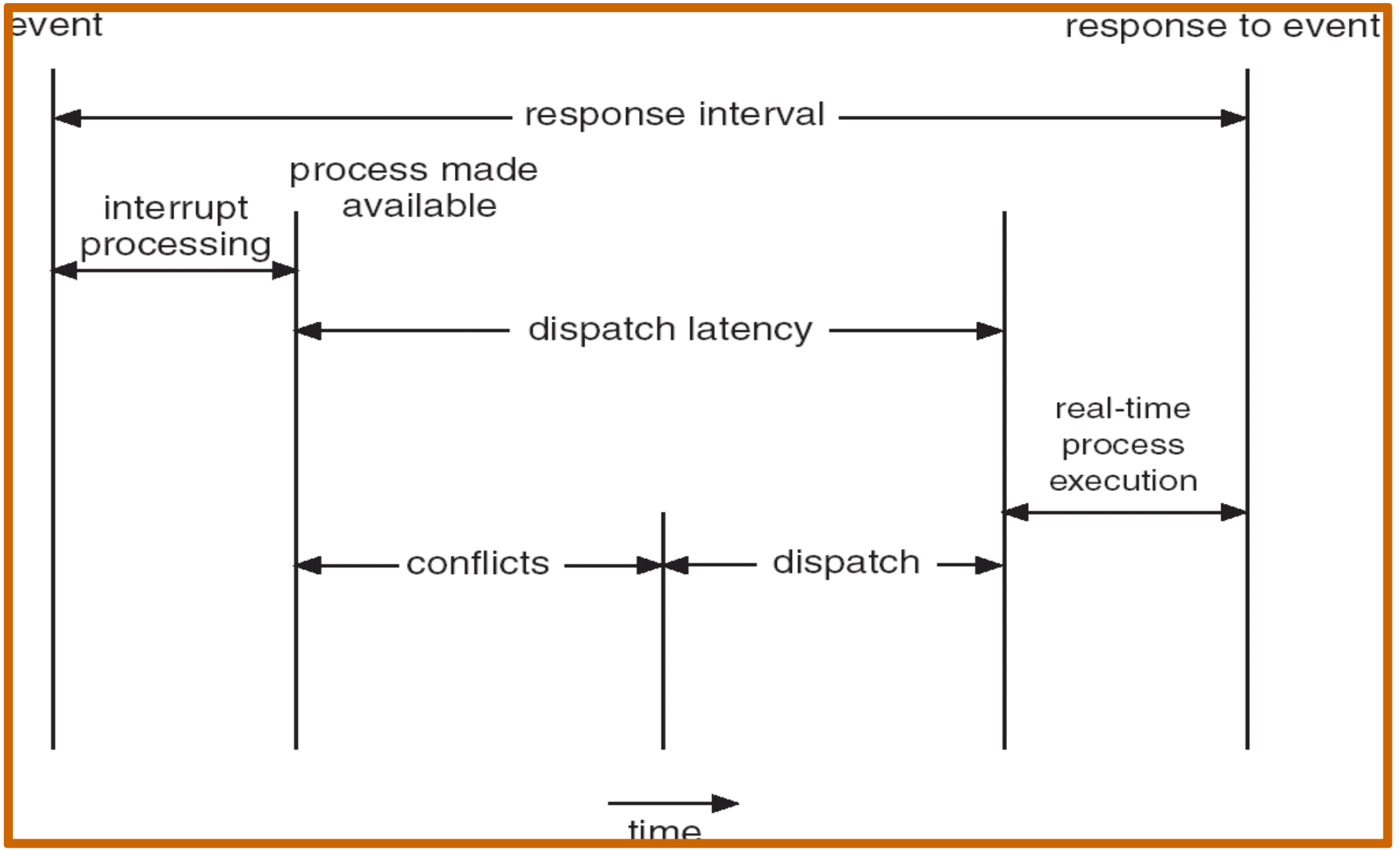


Event Latency

- Amount of time from when an event occurs to when it is serviced.

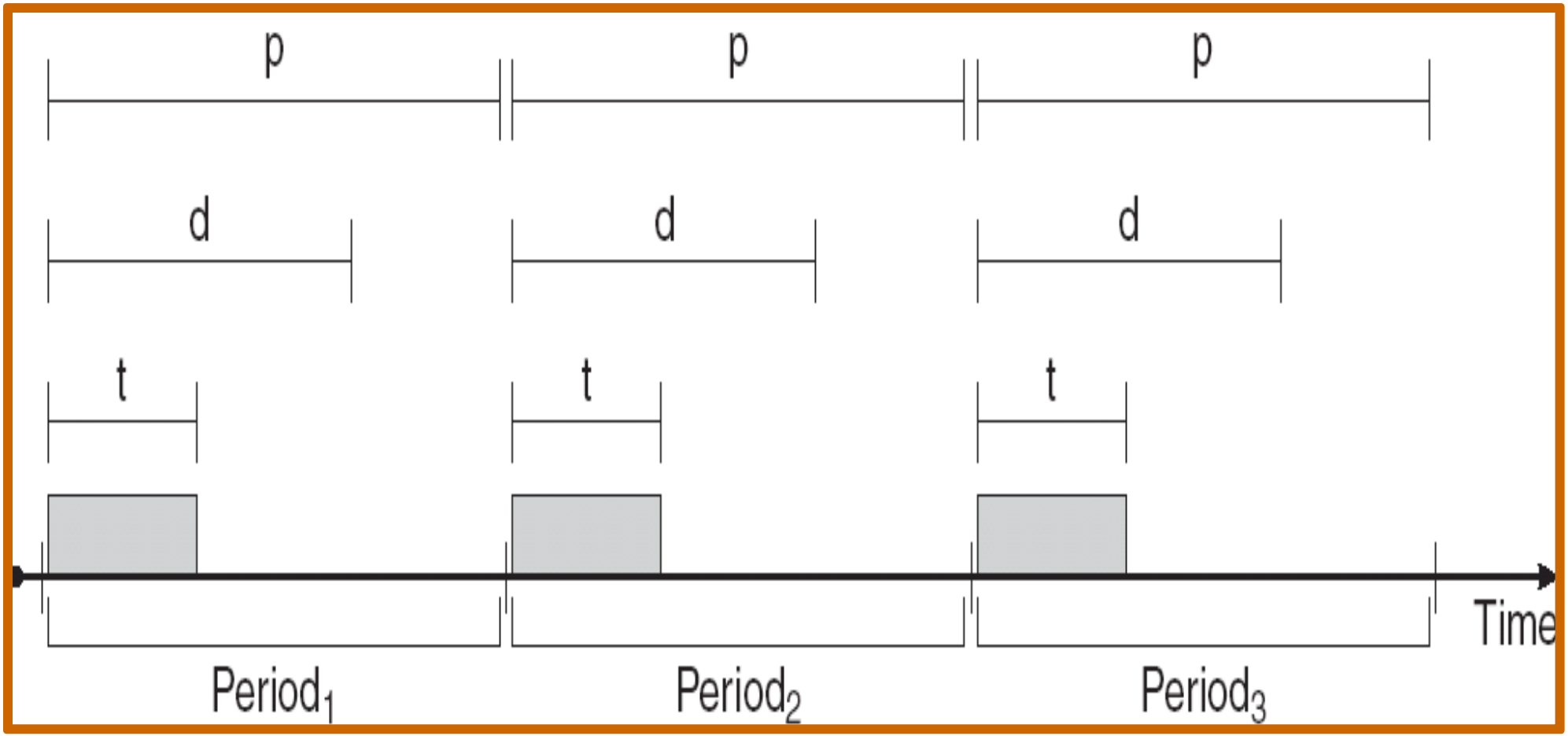


- **Dispatch latency** is the amount of time required for the scheduler to stop one process and start another.

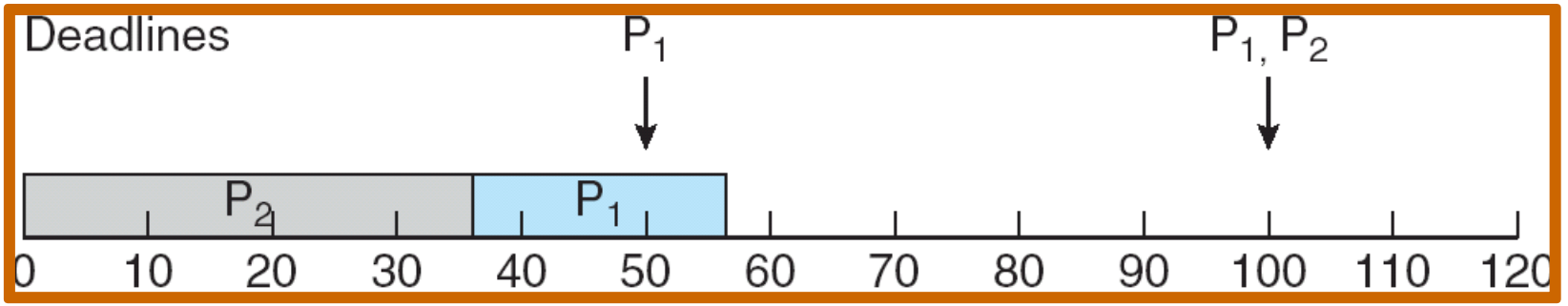


Real Time CPU Scheduling

- Periodic processes require the CPU at specified intervals (periods)
- ***p*** is the duration of the period
- ***d*** is the deadline by when the process must be serviced
- ***t*** is the processing time


p
p
p
d
d
d
t
t
t
Time
Period1
Period2
Period3

Scheduling of tasks when P_2 has a higher priority than P_1



Rate Monotonic Scheduling

- Assumption -- a periodic task model with preemption, no resource sharing, deterministic deadlines, static priorities.
- Priority is assigned based on the inverse of its period
- Shorter periods = higher priority;
- Longer periods = lower priority

- Algorithm is optimal in a way: if there is a feasible schedule for a periodic task load where $P_i <= D_i$ on a single processor with fixed priorities then RM will find a feasible schedule.
- Liu and Leyland show that the schedulable utilization of RM is bounded by :

$$U = \sum_{i=1}^{N} \frac{execution}{period_i} < N(2^{1/N} - 1)$$

where N is the number of real time tasks.

- RMS can meet all the deadlines if CPU utilization is less than 69.3%.

$$\lim_{n \to \infty}(2^{1/N} - 1) = \sim 0.69$$

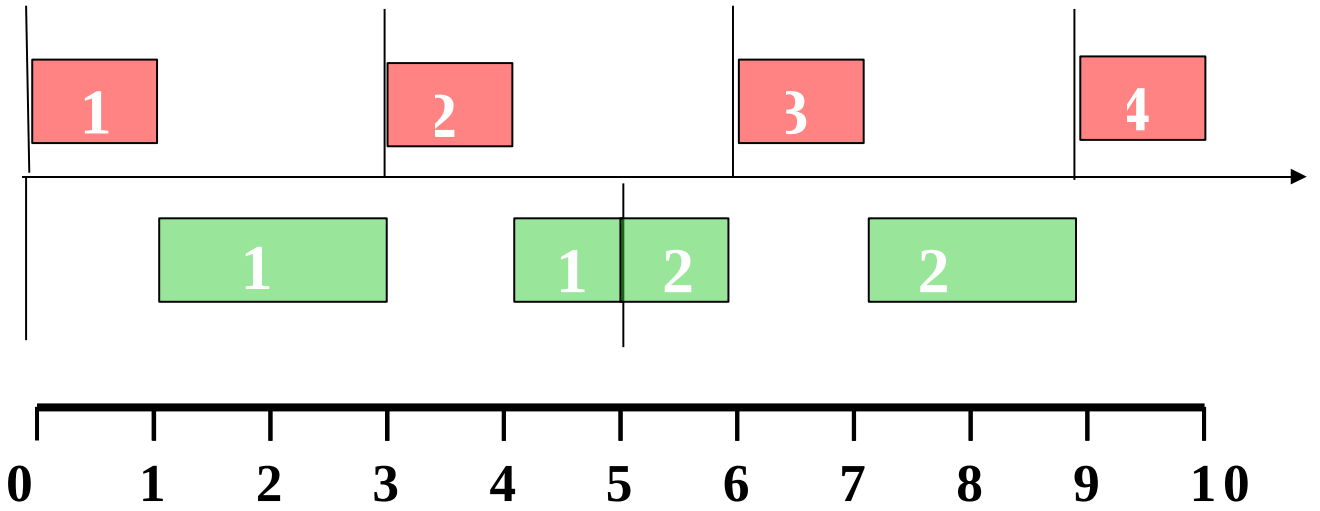
example

- If we have the following set of tasks:

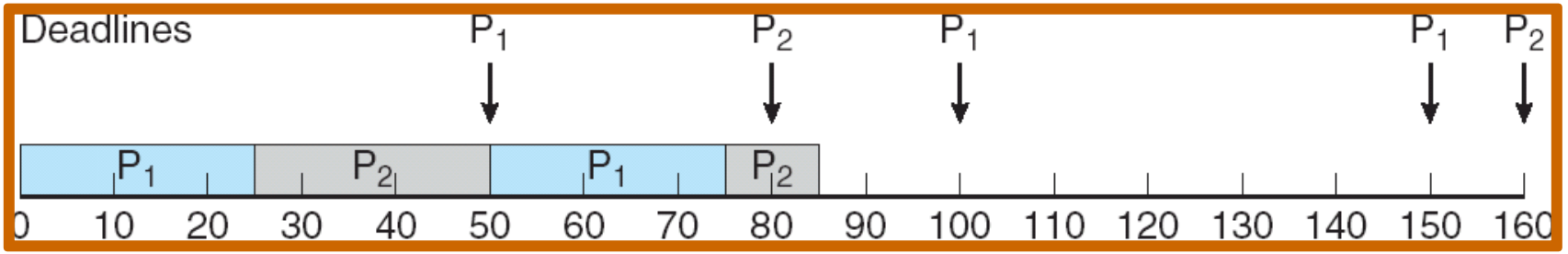
task1: period=deadline=3 execution time=1

task2: period=deadline=5 execution time=3

Then the schedule will look like this:

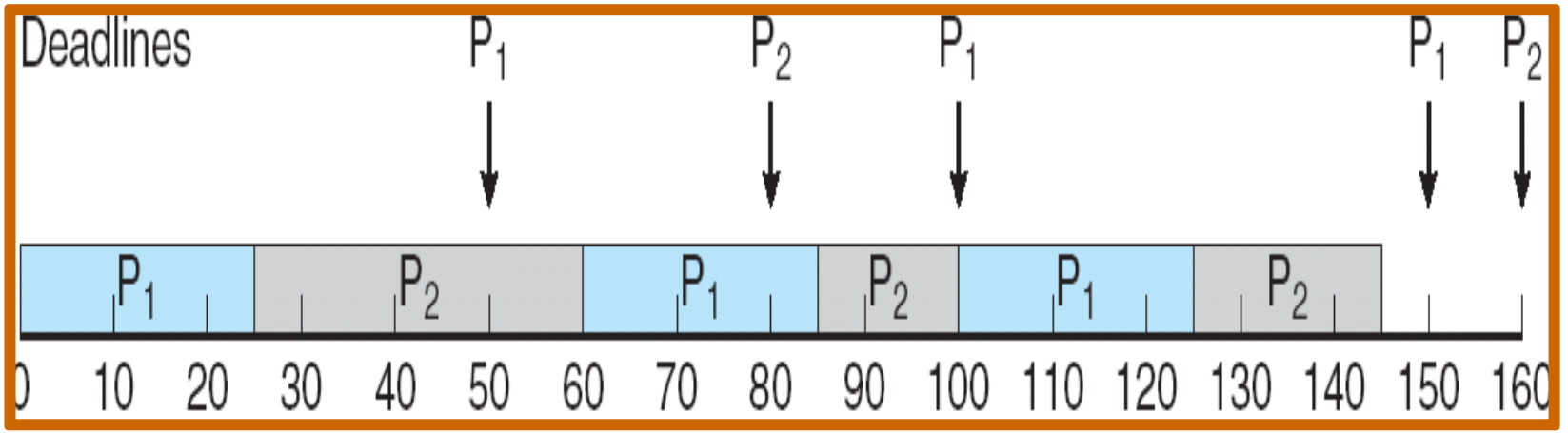


Missed Deadlines with Rate Monotonic Scheduling



Earliest Deadline First Scheduling

- Priorities are assigned according to deadlines: the earlier the deadline, the higher the priority; the later the deadline, the lower the priority.



Proportional Share Scheduling

- ***T*** shares are allocated among all processes in the system.

- An application receives ***N*** shares where ***N*** < ***T***.

- This ensures each application will receive ***N*** / ***T*** of the total processor time.

REAL TIME APPLICATION INTERFACE (RTAI)

- Strictly speaking, not a real time operating system like VxWorks or QNX
- Linux lacks real time support
- RTAI offers services of the Linux kernel core, adding features of a real time operating system
- RTAI is module oriented.

Complementary Layers in RTAI

1. HAL (Hardware Abstraction Layer) -- provides an interface to the hardware, on top of which both Linux and the hard real-time core can run.
2. Linux compatibility layer -- provides an interface to Linux, with which RTAI tasks can be integrated into the Linux task management, without Linux noticing anything.
3. RTOS core -- offers the hard real-time functionality for task scheduling, interrupt processing, and locking.
4. LX/RT (Linux Real-Time) -- makes soft and hard real-time features available to user space tasks in Linux. Puts a strong emphasis on offering a symmetric real-time API
5. Extended functionality packages.

Task Management

- A task is created with the following function:
  int rt_task_init (RT_TASK *task, void (*rt_thread)(int), int data, int stack_size, int priority, int uses_fpu, void(*signal)(void))

Another difference between both versions is that a POSIX thread initialization makes the task active immediately, while the task created by a rt_task_init() is suspended when created, and must be activated explicitly.

Via the signal parameter of rt_task_init(), the application programmer can register a function that will be executed whenever the task it belongs to will be scheduled, and before that task is scheduled. The ASR is run with interrupts disabled

Scheduling

- Tasks can configure periodic scheduling and one-shot scheduling
  RTAI has static priority-based scheduling ("SCHED_FIFO") as its default hard real-time scheduler
- But if offers also Round Robin time-sliced scheduling ("SCHED_RR"), Rate Monotonic Scheduling, and Earliest Deadline First.
  When multiple scheduler schemes are used, RTAI has made the (arbitrary) choice to give EDF tasks a higher priority than tasks scheduled with other policies.
  rt_set_sched_policy(RT_TASK *task, int policy, int rr_quantum_ns // time slice in nanoseconds, lying between // 0 (= default Linux value) and // 0x0FFFFFFF (= 1/4th of a second))

- RMS: the RMS scheduler is (re)initialized by the function void rt_spv_RMS(int cpuid), to be called after the operating system knows the timing information of all your tasks.
- EDF: this scheduler must know the start and termination times of all your tasks, so a task must call the function void rt_task_set_resume_end(RTIME resume_time, RTIME end_time); at the end of every run of one cycle of the task.

Interrupt Handling

- An interrupt handler must be registered with the operating system via a call of the following function:
- int rt_request_global_irq (unsigned int irq, void (*handler)(void));
- Installed handler takes care of properly activating any Linux handler using the same irq number, by calling the void rt_pend_linux_irq (unsigned int irq) function, which “pends” the interrupt to Linux (in software!).
- Linux processes interrupts as soon as it gets control back from RTAI. Note that, at that time, hardware interrupts are again enabled for RTAI.

- From an RTAI task, one can also register an interrupt handler with Linux, via
- int rt_request_linux_irq (unsigned int irq, void (*handler)(int irq, void *dev_id, struct pt_regs *regs), char *linux_handler_id, void *dev_id);
- This forces Linux to share the interrupt. The handler is appended to any already existing Linux handler for the same irq and run as a Linux irq handler.The handler appears in /proc/interrupts, under the name given in the parameter linux_handler_id.
- Floating point register saving is on by default in RTAI interrupt handlers. One can also select which CPU must receive and handle a particular IRQ, via the rt_assign_irq_to_cpu(int irq, int cpu) function. rt_reset_irq_to_sym_mode(int irq) resets this choice, back to the symmetric “don’t care” behaviour.

Semaphores

- RTAI has counting semaphores, binary semaphores and recursive semaphores.RTAI semaphores have priority inheritance. and (adaptive) priority ceiling.
- 1)Priority inheritance. A low-priority task that holds the lock requested by a high-priority task temporarily “inherits” the priority of that high-priority task, from the moment the high-priority task does the request.
  2)Priority ceiling. Every lock gets a priority level corresponding to the priority of the highest-priority task that can use the lock. This level is called the ceiling priority.
- priority ceiling is in the POSIX standard (POSIX_PRIO_PROTECT), the Real-Time Specification for Java (RTSJ), OSEK, and the Ada 95 real-time specifications. Priority inheritance is also part of standards such as POSIX (POSIX_PRIO_INHERIT), and the RTSJ.

Real Time FIFO

- Used for deterministic transfer of data between threads.
- Designed so that the real-time task will never be blocked when it reads or writes data
- API
  - rt_create(unsigned int fifo, int sise)
  - rt_destroy(unsigned int fifo)
  - rt_fifo_put(fifo, char *buf, int count)
  - rt_fifo_get(fifo, char *buf, count)

Timers

- How do we get correct time from a PC ?
- 8254 -- programmable interval timer/counter, that can be treated as an array of four I/O ports in the system software (from 0x40 to 0x43).
- Three are independent 16-bit counters and the fourth is a control register for mode programming.
- Programmer configures the 8254 to selects the mode and programs one of the counters for the desired delay. After this delay, the 8254 will interrupt the CPU.
- Counters are fully independent, so each counter may operate in a different mode.

- Linux programs the timer with mode 2 (rate generator, periodic pace) and loads the counter0 with the macro HZ.
- RTAI provides both a periodic (mode 2 of the 8254) and a oneshot timer (mode 0).
- Oneshot mode -- clock is reprogrammed after every interrupt. More flexible because it allows to trigger off some external event.
- Periodic mode -- programmed only at beginning and then generates interrupts periodically. Much more efficient for tasks that take regular samples.
- In oneshot mode time is measured on the base of the CPU time stamp clock (TSC) and not on the 8254 chip, which is used only to generate oneshot interrupts. This allows to reprogram the counter with only 2 I/O instructions, i.e. approximately 3 us.

HAL

- The RTHAL (Real-Time Hardware Abstraction Layer), is, not surprisingly, very platform-dependent. Its code typically contains lots of assembler code that builds the low-level infrastructure, not only for the HAL, but also for the Linux compatibility layer , the core and for LX/RT
- rthal is the central data structure of RTAI's HAL: it collects the variables and function calls that Linux uses for interrupts and task switching.
  1)ret_from_intr:it's not Linux but RTAI that decides what will be done next, after an interrupt routine has finished
  2)_switch_to:Pointer to the function that does a task switch.
  3)void (*lxrt_global_cli)(void):This is used in LX/RT scheduling;

1) RTAI replaces Linux functions that work with the interrupt hardware with pointers to functions.

2) RTAI introduces the rthal data structure to store all these pointers.

3) RTAI can now switch these pointers to pointers to its own functions whenever it wants.

- HAL can be used for other purposes than serving as a stub for the RTAI core. That is, another kind of operating system could be implemented on top of the RTHAL.
- At boot time, Linux runs as if nothing has happened, except for a small loss in performance,due to the extra level of indirection introduced by replacing function calls by pointers to function calls in the rthal structure.
- When RTAI gets activated it switches the pointers to functions in the rthal data structure from their Linux version to their RTAI version. From that moment on, Linux is under control of the RTAI kernel, because Linux works with what it thinks is the “real” hardware through the replacement functions that RTAI has installed. But these functions give a virtual hardware to Linux, while RTAI manages the real hardware.

LXRT

- The first generation used the idea to let a user space task run a companion task in the kernel, i.e., the so-called “buddy” in RTAI language. This companion task executes kernel space functions on behalf of the user space task.
- A user space task is made into an LX/RT user space task by using only a couple of LX/RT calls. The task calls rt_make_hard_real_time() (in include/rtai_lxrt.h) at the moment it wants to switch to real-time, and rt_make_soft_real_time() to switch back.
- So, the clue of the LX/RT procedure is to make the user space task launch a trap handler that executes a real-time service for the user space task; and all this is done through just one single trap handler, by encoding the desired service. Hence, a special LX/RT version must be made for all RTAI functions that one wants to be available to user space tasks, and a unique code must be given to each function.