Real-Time Operating Systems

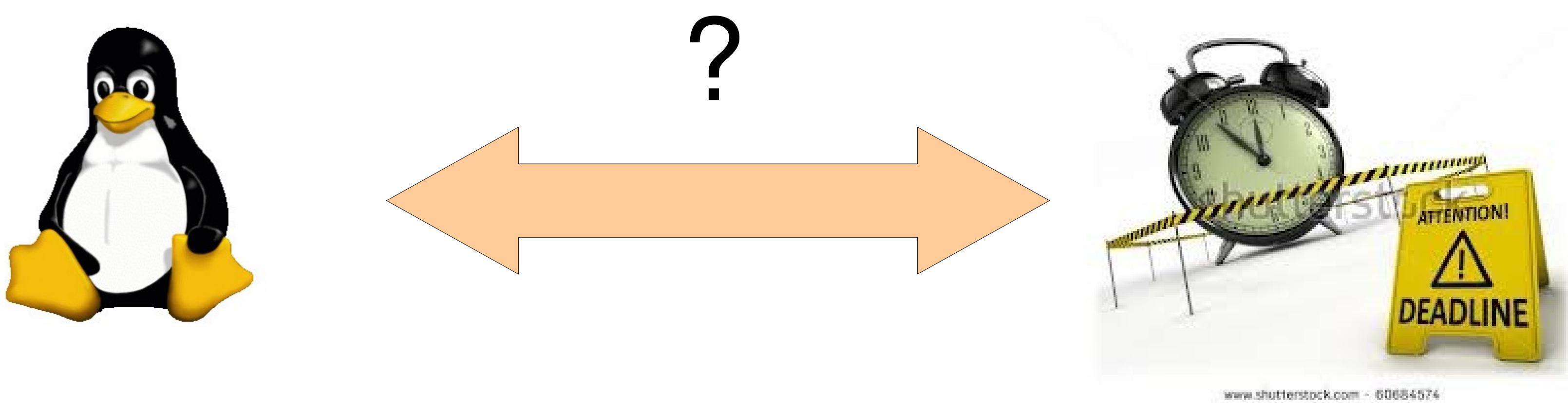
Paulo Pedreiras
DETI/UA/IT
Oct/2021

Agenda

- Linux and real-time
- Improving the real-time performance of Linux
- Short overview of latency sources
- The PREEMPT_RT project
- Real-Time application development
- Further reading

Embedded Linux and real-time

- Linux and other open-source software are increasingly used for developing Embedded Systems
- Many of these applications have **real-time requirements**
- Ideally we would like to:
    - Have the advantages of Linux: HW support, low cost, modularity, openess, ...
    - And meet *deadlines*!



Embedded Linux and real-time

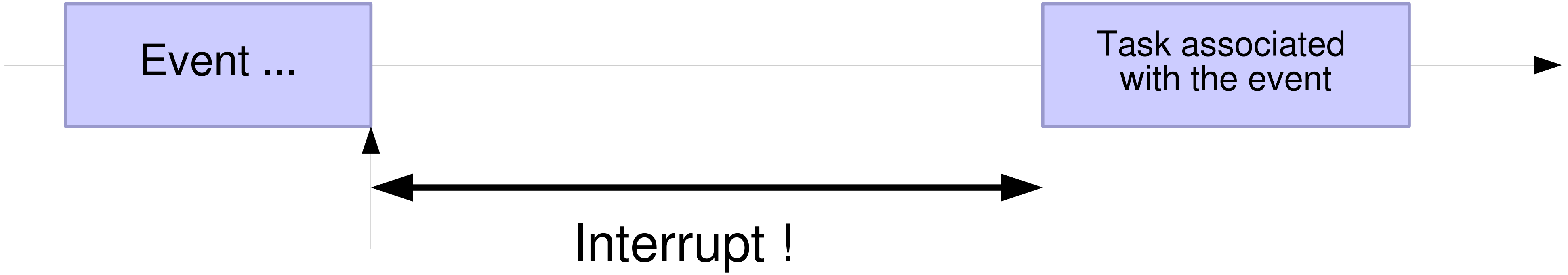
- Linux was developed as **generic desktop and server *time-sharing OS***
  - Objectives are: optimize *throughput, improve the average utilization of resources (*CPU, memory, I/O), ...
  - Timeliness in not a main issue
- Conversely, dedicated Real-Time Operative Systems (RTOS) are engineered for **having temporal determinism, often at expenses of throughput**
- In fact, ***throughput* and temporal determinism are often conflicting requirements**
  - Optimizing for one impacts negatively on the other

Improving the real-time performance of Linux

- Two ways to handle this conflict:
- **Approach 1**
  - Modify the Linux kernel to improve its temporal behavior
    - Bound the latency of syscalls, introduce fine-grain preemption, improve timer resolution, create specific services for real-time, proper scheduling, etc.
  - **Many of these features have been added to the mainline Linux kernel (from the PREEMPT_RT project).**
- **Approach 2**
  - Add a RTOS “under” linux. Linux becomes a *background task of the RTOS*
    - Approach followed e.g. by RTAI, Xenomai, ...

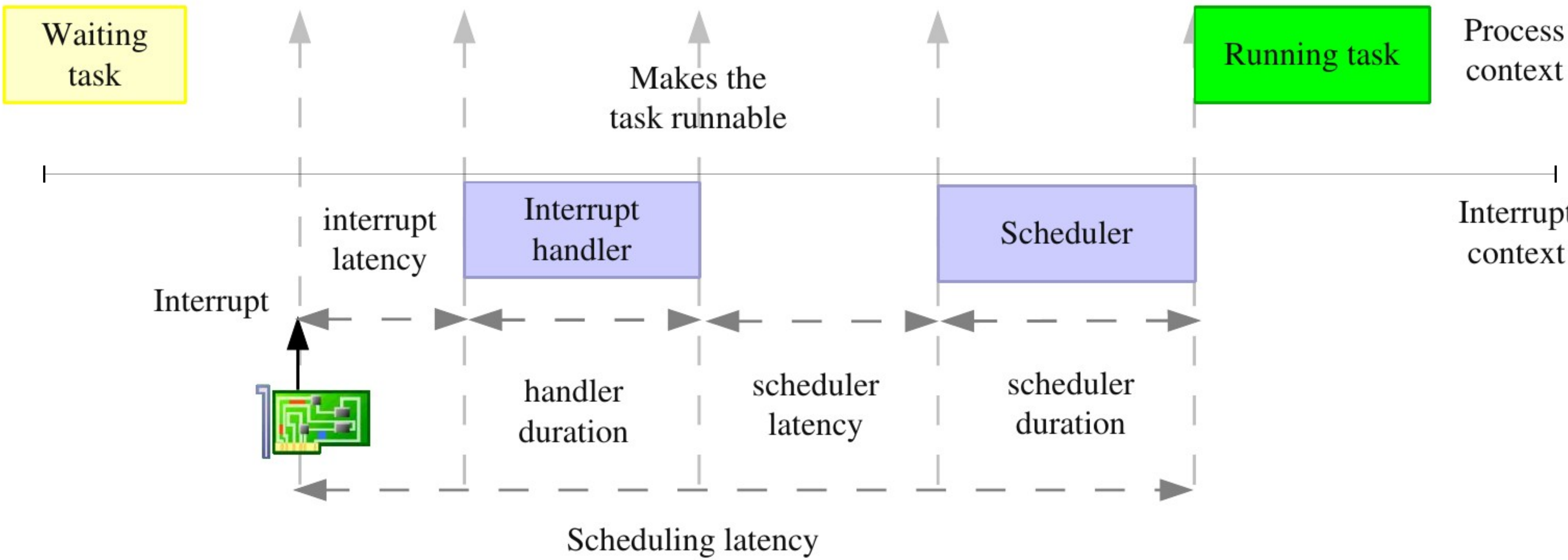
Hints on Linux kernel latency sources

- A **typical event sequence** in a real-time application is:
  - An event causes a CPU interrupt
  - The corresponding ISR is executed, activating a user-space task associated with the event
  - Eventually this task is executed and completes, thus closing the reaction to the event.
    - Time elapsed between an event and the corresponding task activation is the latency
- **The objective is reduce, as much as possible, the latency!**



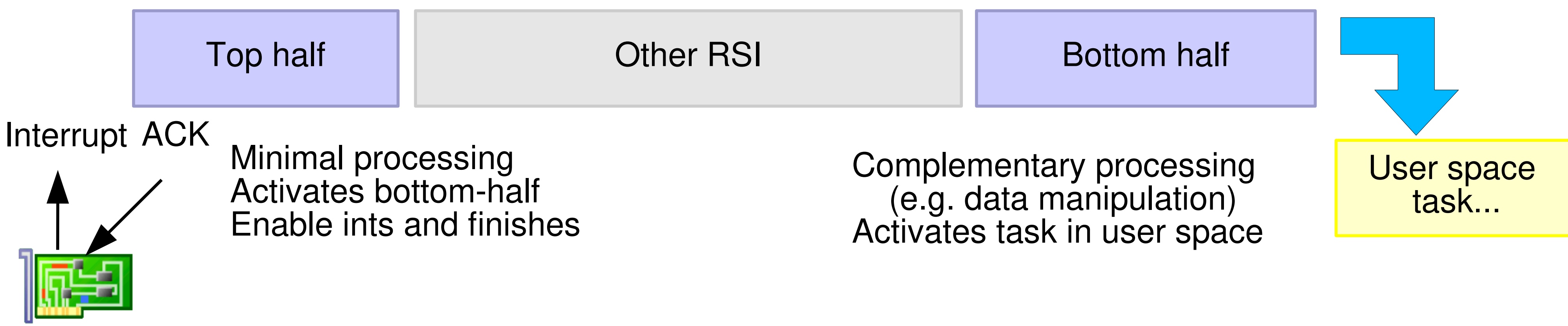
Hints on Linux kernel latency sources

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



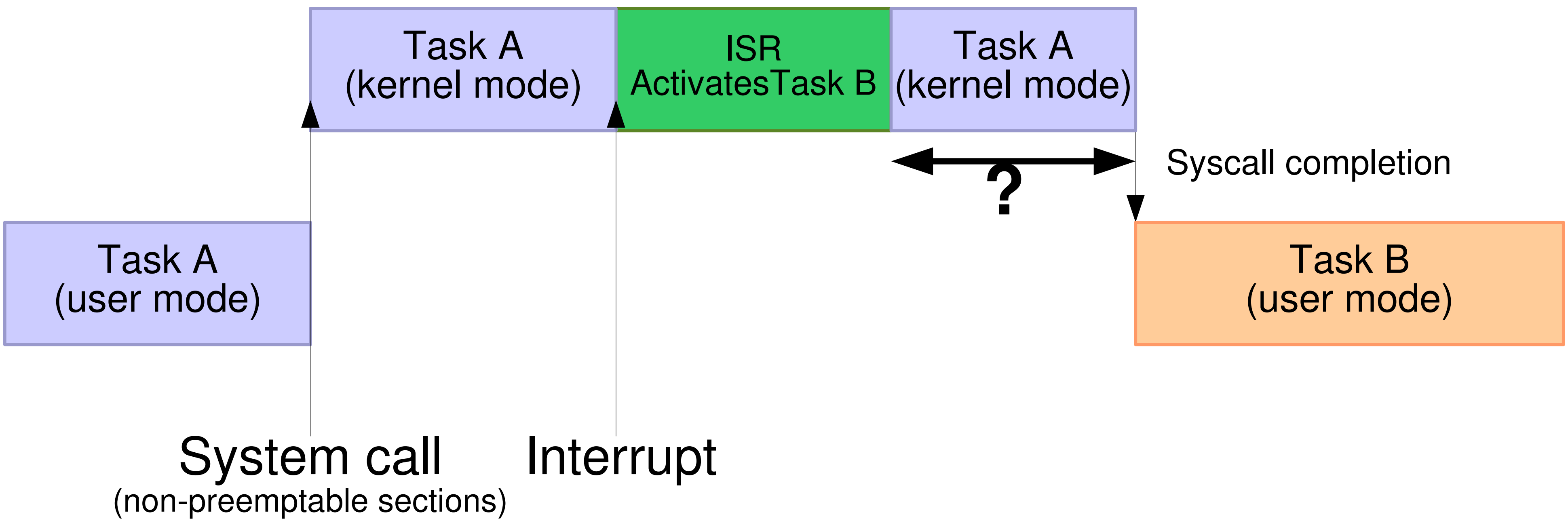
Hints on Linux kernel latency sources

- Interrupt latency sources:
  - Linux kernel (including device-drivers) disables interrupts for certain operations
  - Interrupts can interrupt other interrupts (nesting)
  - …
- Interrupt handlers are split in two parts (top/bottom; fast/slow)

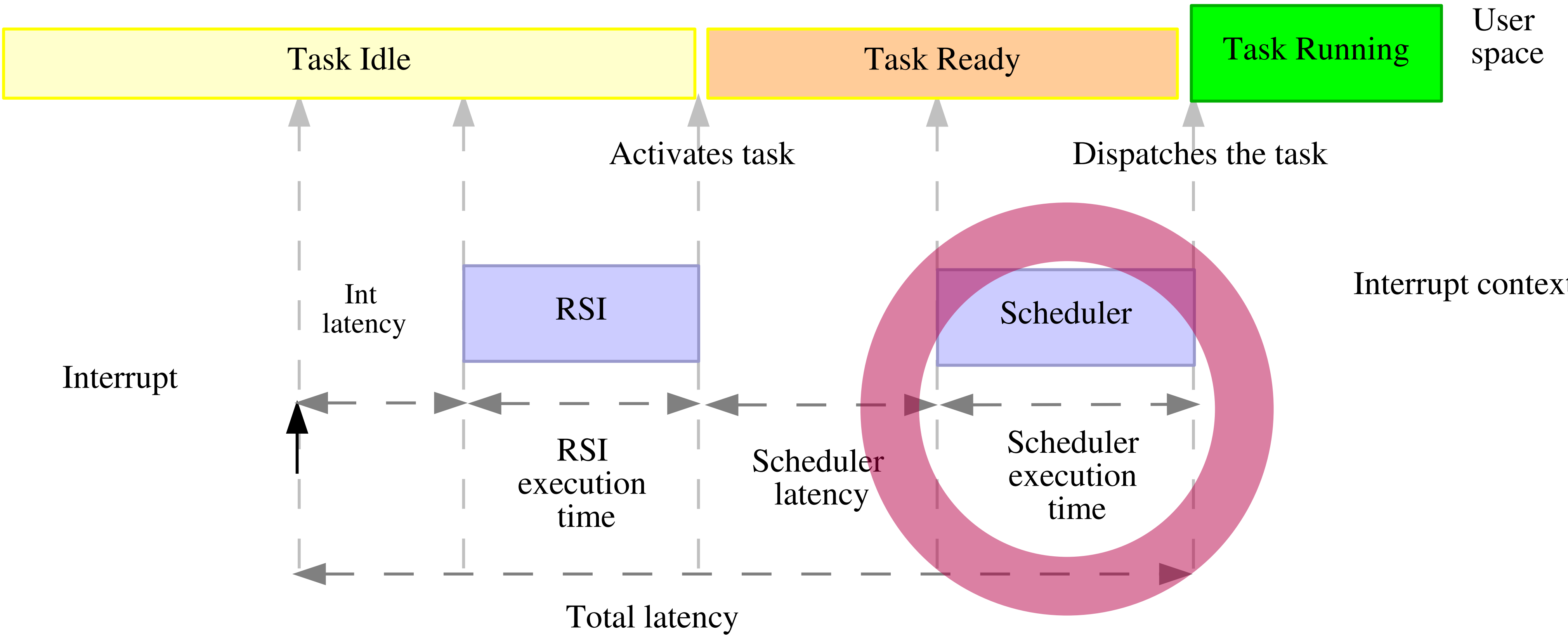


Hints on Linux kernel latency sources

- Linux kernel is preemptive
  - But not fully! Syscalls have variable duration and limited preemption



Hints on Linux kernel latency sources

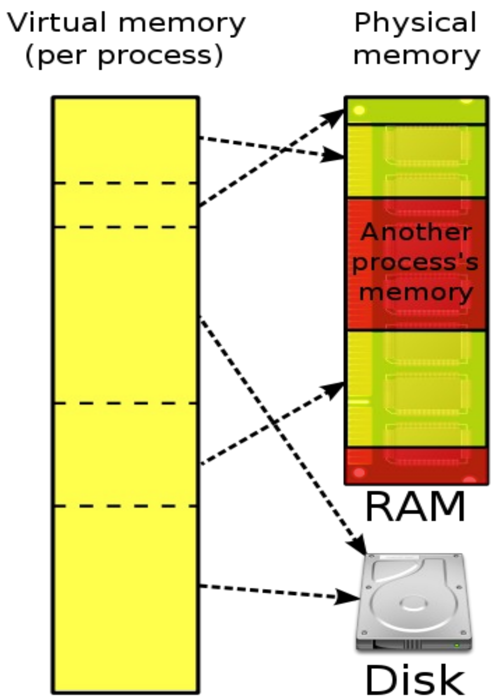


kernel latency = interrupt latency + handler duration + scheduler latency + scheduler duration

Scheduler execution time: depends on the scheduler. It is bounded.

Hints on Linux kernel latency sources

- In addition, there are many other sources of latency and jitter that affect real-time tasks in Linux:
  - Linux makes an intensive use of virtual memory. Access to data that is on disk takes much longer than access to data that is in RAM
  - Same applies to cache
  - Many C library functions have not been designed for deterministic execution
  - Priority inversion
    - Due to shared resources
  - Interrupt prioritization
  - ...



PREEMPT_RT Project

- Developed by Ingo Molnar, Thomas Gleixner e Steven Rostedt (https://rt.wiki.kernel.org)
- Wiki currently maintained in:
  - https://wiki.linuxfoundation.org/realtime/start
- Very good documentation, with technical aspects, examples, etc.
  - https://wiki.linuxfoundation.org/realtime/documentation/start
- Objective: **gradually improve the real-time behavior of the Linux kernel and bring those improvements to the mainline kernel**
  - Most of those improvements are already integrated on the mainline Linux kernel

PREEMPT_RT Project

Some of the improvements already integrated on the mainline kernel

- O(1) scheduler
- Fixed-priority scheduler
- Kernel preemption
- Improvements to the POSIX real-time API support
- Mutexes with “Priority inheritance” support
- High resolution timers
- Threaded interrupts
- sched_deadline, EDF scheduling with CBS
- …

Example: preemption control

It may require compiling the Linux kernel with the right options.

E.g. Linux supports several preemption modes:

Preemption Model	
○ No Forced Preemption (Server)	PREEMPT_NONE
◉ Voluntary Kernel Preemption (Desktop)	PREEMPT_VOLUNTARY
○ Preemptible Kernel (Low-Latency Desktop)	PREEMPT

Example: preemption control

`CONFIG_PREEMPT_NONE`
Kernel code (interrupts, exceptions, system calls) never are preempted

- Common default configuration in many distributions
- Better performance for systems that carry intensive processing, if the objective is maximize throughput
  - Minimizes context switches and associated overheads

Example: preemption control

`CONFIG_PREEMPT_VOLUNTARY`
Kernel code is preemptable at specific points

- Useful on *desktop environments*, as it increases the reactivity (as perceived by an human user)
- Rescheduling points are explicitly added to the kernel code
- Low impact on *throughput*.

Example: preemption control

`CONFIG_PREEMPT`
Most of the kernel becomes preemptable

- In most cases, when a process is dispatched it can start execution before the completion of pending syscalls (issued by other processes)
- However there are non-preemptible critical sections protected by *spinlocks*
  - Better option for Embedded Systems with RT requirements
  - Low/moderate impact on *throughput*.

For better RT performance this option must be active!

Using the Linux real-time services

Project development

- No special tools are required
  - POSIX* real-time extensions are part of the standard C library
- Link code with -lrt.
  - E.g. `gcc -o myprog myprog.c -lrt`
- API documentation
  - `man functioname`

POSIX* - Portable Operating System Interface for Computing Environments

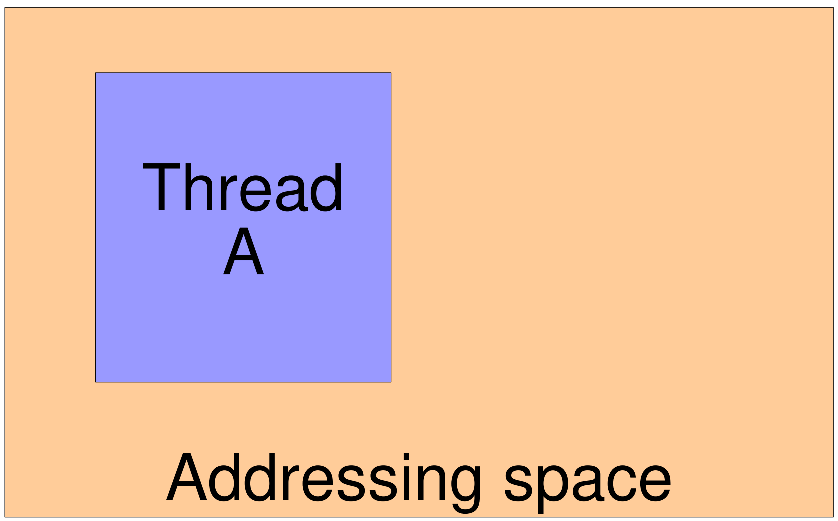
Process, thread, task ?

There is some confusion between “process”, “thread” and “task”

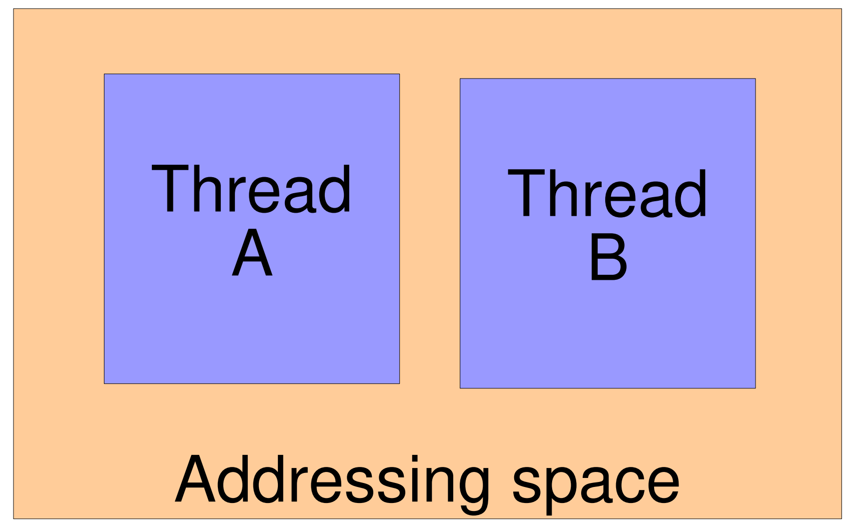
- In Unix/Linux processes are created by the `fork()` primitive and are are composed of:
  - An addressing space (for code, data, stack ...)
  - A thread, that starts the execution of the main() function
    - After creation, a process has a single thread
    - Additional threads can be created by means of the syscall `pthread_create()`
      - These threads share the same addressing space as the initial thread
      - And execute a function given as argument to the `pthread_create()` `syscall`
  - Often the term task is used interchangeably with process

Processes and threads at the kernel level

- Kernel represents threads by a structure of type "task_struct"
- From the scheduling point of view there are no differences between the initial thread and the additional threads crated via pthread_create()



Process after `fork()`



Same process after `pthread_create()`

Threads creation

- Linux supports the POSIX API
- To create a new thread
  - ```
    pthread_create(pthread_t *thread,
                   pthread_attr_t *attr,
                   void *(*routine)(*void*),
                   void *arg);
    ```
  - The new thread is created on the same addressing space, but scheduled as an independent entity
- Terminating a thread
  - ```
    pthread_exit(void *value_ptr);
    ```
- Waiting for the termination of a thread
  - ```
    pthread_join(pthread_t *thread, void **value_ptr);
    ```
    ```

Scheduling classes

Linux kernel supports several scheduling classes

- The default class is **time-sharing**
  - All processes receive some CPU time independently of its priority
  - The CPU share of each process is dynamic
    - Affected by the “nice” value, that varies between -20 (bigger) and 19 (lower)
    - Depends on the type of operations carried out
      - CPU-bound, I/O-bound
    - Can be changed by commands nice and renice
  - Non deterministic
  - **Extremely poor real-time behavior**

Scheduling classes

- There are two fixed-priority real-time scheduling classes:

  SCHED_FIFO and SCHED_RR

  - The ready task with higher priority gets the CPU
  - Priority is statically defined
    - Varies from 0 (lower) to 99 (higher)
  - SCHED_RR vs SCHED_FIFO
    - SCHED_RR: round-robin applied to tasks that share the same priority
    - SCHED_FIFO: applies a FIFO policy to tasks that share the same priority
  - Significantly improved real-time behavior
    - Depends on the platform but tens to a few hundreds of us of jitter are feasible.

Scheduling classes

- A process can be assigned with a real-time class via syscall **chrt**
  - Example: `chrt -f 99 ./myprog`
    - `-f FIFO ; 99 priority`
  - `Scheduling attributes of a process can be retrieved with chrt`
    - `chrt -p PID`

Scheduling classes

- The Linux kernel also dynamic priorities via the “sched_deadline” scheduling class
  - sched_deadline has the highest priority that can be defined by the user (specifically, higher than SCHED_FIFO and SCHED_RR)
- Task parameterization:
    - **Runtime**: Maximum execution time per period
    - **Deadline**: Time window, starting from the period beginning, during which “Runtime” must be served
    - **Period**: Periodicity of activation of the server
  - **Make sure that RUNTIME <= DEADLINE <= PERIOD**
- Example:

```
# chrt -d --sched-runtime 1000000 --sched-deadline 5000000 --
sched-period 50000000 ./dummyTask &
```

Scheduling classes

- Check with chrt -p

```
# chrt -d --sched-runtime 1000000 --sched-deadline 5000000 --sched-
period 50000000 0 ./dummyTask &
[1] 8521
# chrt -p 8521
pid 8521's current scheduling policy: SCHED_DEADLINE
pid 8521's current scheduling priority: 0
pid 8521's current runtime/deadline/period parameters:
1000000/5000000/5000000
```

- Linux ensures that the **utilization** of EDF jobs **does not exceed 95%** of the available computing time.
  - This setting can be changed by using the proc files:
    - /proc/sys/kernel/sched_rt_period_us
    - /proc/sys/kernel/sched_rt_runtime_us

Scheduling classes

- Syscall `sched_setscheduler()` allows defining the scheduling class programatically.
  - `int sched_setscheduler(pid_t pid, int policy, const struct sched_param *param);`
    - `policy`: `SCHED_OTHER, SCHED_FIFO, SCHED_RR`, etc.
    - `param`: structure that includes priority

Scheduling classes

- The **individual** priority of each thread can be defined at its creation:

```
struct sched_param parm;
pthread_attr_t attr;

pthread_attr_init(&attr);
pthread_attr_setinheritsched(&attr,
                             PTHREAD_EXPLICIT_SCHED);
pthread_attr_setschedpolicy(&attr, SCHED_FIFO);
parm.sched_priority = 42;
pthread_attr_setschedparam(&attr, &parm);
```

- The thread is created using `pthread_create()`, with argument “`attr`” `structure`
- Other options can be set (e.g. stack size)

Scheduling classes

- Using SCHED_DEADLINE programatically
  - Unfortunately the implementation of SCHED_DEADLINE is not standard
  - See “man sched_setattr”, section “CONFORMING TO”
  - struct sched_attr and methods sched_getattr() and sched_setattr() are still missing from sched.h!
  - Linux distribution dependent
  - Working example can be found in: https://www.kernel.org/doc/Documentation/scheduler/sched-deadline.txt

Memory blocking

- To avoid the **indeterminism** that results from the use of virtual memory, it is possible to lock the memory
  - The memory used by the process addressing space is always kept in RAM

    `mlockall(MCL_CURRENT | MCL_FUTURE);`

  - Locks of memory pages used by the process (current and future, if `MCL_FUTURE`)
    - Heap, stack, shared memory, ...
- Other related syscalls:
  - `munlockall`, `mlock`, `munlock`.

Mutexes

- **Mutext**: allow implementing mutual exclusion between threads of the same process
  - Creation and elimination

```
pthread_mutex_init(pthread_mutex_t *mutex,
         const pthread_mutexattr_t *mutexattr);
pthread_mutex_destroy(pthread_mutex_t *mutex);
```

  - Lock/unlock

```
pthread_mutex_lock(pthread_mutex_t *mutex);
pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- For using priority inheritance:

```
pthread_mutexattr_t attr;
pthread_mutexattr_init (&attr);
pthread_mutexattr_getprotocol
      (&attr, PTHREAD_PRIO_INHERIT);
```

Timers

- Create a timer.

```
timer_create(clockid_t clockid,struct sigevent *evp,
                timer_t *timerid)
```

  - **clockid** usually CLOCK_MONOTONIC or CLOCK_BOOTIME
  - **sigevent** defines the action to be executed when the timer expires
  - **timerid** returns the timer id
- Set a timer for a give time instant

```
timer_settime(timer_t timerid, int flags,
                struct itimerspec *newvalue,
                struct itimerspec *oldvalue)
```

- Related syscalls
  - timer_delete(), clock_getres(); timer_getoverrun(), timer_gettime().

Making a thread periodic

- ***clock_nanosleep()***
  - Allows the calling thread to sleep for an interval specified with nanosecond precision
  - int clock_nanosleep(clockid_t clock_id, int flags,
    const struct timespec *request,
    struct timespec *remain);
  - If flags have “TIMER_ABSTIME” set, sleeps until the time instant specified in request

```
clock_gettime(CLOCK_MONOTONIC, &ts);
while(1) {

   ADD_PERIOD_TO_TS;
   clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME,&ts,&tr);
   PROCESSING;

}
```

Signals

- **Signals**: mechanisms for asynchronous notifications
  - A notification may be issued:
    - By the activation of a signal handler (few limitations)
    - Unlocking by means of a primitive **sigwait**(), **sigtimedwait**() or **sigwaitinfo**().
      - Preferred method!!
  - The signal behaviour can be defined by means of syscall **sigaction**()
  - Signal masking can be carried out with **pthread_sigmask**()
  - Sending a signal can be made via **pthread_kill**() or **tgkill**()
  - Can be used signals between **SIGRTMIN** and **SIGRTMAX**

Interprocess communication

- Semaphores
  - Can be used between different processes (*named semaphores*)

    `sem_open()`, `sem_close()`, `sem_unlink()`, `sem_init()`, `sem_destroy()`, `sem_wait()`, `sem_post()`, etc.
- Message queues
  - Allow data exchanges in the form of messages.

    `mq_open()`, `mq_close()`, `mq_unlink()`, `mq_send()`, `mq_receive()`, etc.
- Shared memory
  - Allow data exchanges via a sahred memory region

    `shm_open()`, `ftruncate()`, `mmap()`, `munmap()`, `close()`, `shm_unlink()`

Debugging kernel latency

Ftrace – tool that can be used for debug as well as for latency analysis

- Developed by Steven Rostedt and part of kernel form version 2.6.27.
- Very well documented (`Documentation/ftrace.txt)`
- *Very small Overhead* when not active
- Can be used for tracing the execution of any kernel function

To learn more ...

- The Real-Time Linux Wiki at The Linux Foundation: https://wiki.linuxfoundation.org/realtime/start
- Federico Reghenzani, Giuseppe Massari, William Fornaciari. “The Real-Time Linux Kernel: A Survey on PREEMPT_RT”, ACM Computing Surveys, Volume 52, Issue 1February 2019. https://doi.org/10.1145/3297714
- http://www.osadl.org
  - Open Source Automation Development Lab (OSADL)
    Among other activities, develops and integartes RT preempt patches on the mainline kernel (HOWTOs, live CD, patches).
    https://www.osadl.org/Realtime-Linux.projects-realtime-linux.0.html

Slides adapted from “Real-time in embedded Linux systems”, by M. Opdenacker, T. Petazzoni e G. Chanteperdrix

To probe further ...

- http://www.realtimelinuxfoundation.org/
  Real-Time Linux community portal
  Organizes an annual workshop
-