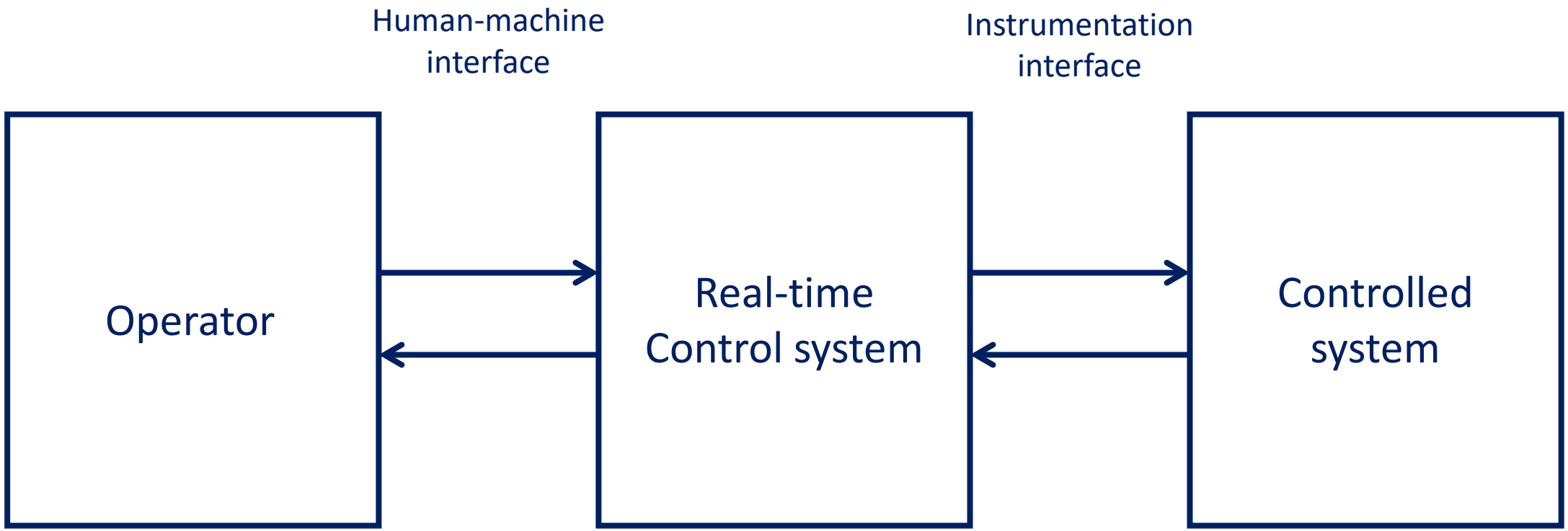
POLITECNICO
MILANO 1863

Design of computer-based real-time acquisition and control systems

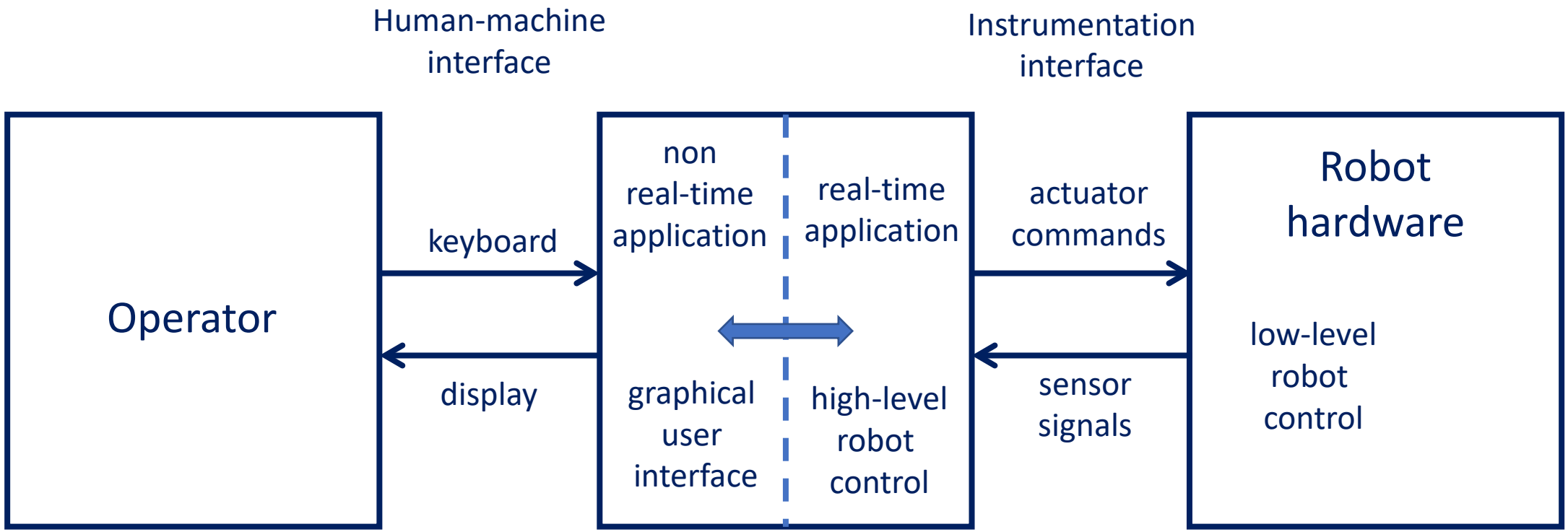
Application examples

Dr. Stefano Dalla Gasperina

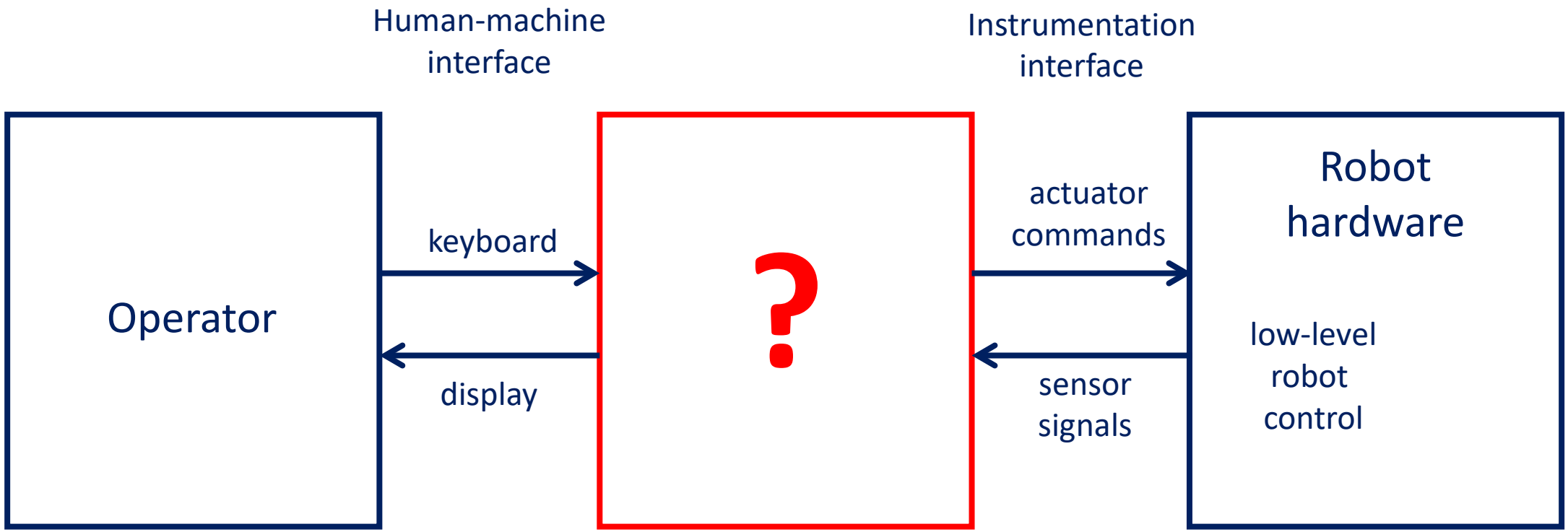
Example of a real-time control system



Example of a real-time robot control system

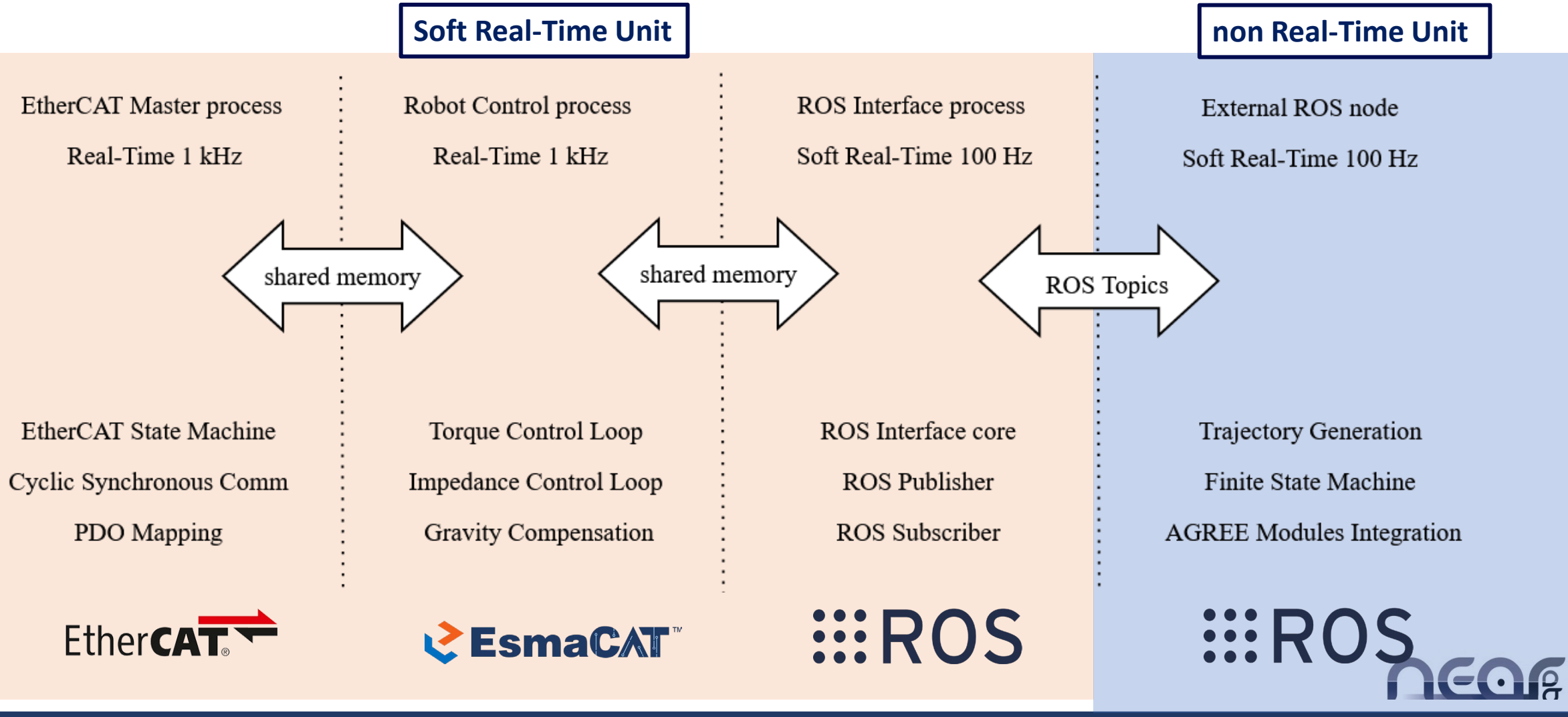


Example of a real-time robot control system



S. Dalla Gasperina, email: stefano.dallagasperina@polimi.it

Example of software architecture



S. Dalla Gasperina, email: stefano.dallagasperina@polimi.it

How to program a Linux real-time application?

First things first:

- **How to obtain a Linux real-time kernel?**
- How do we check if our system is real-time enabled?

How to obtain a Linux real-time kernel?

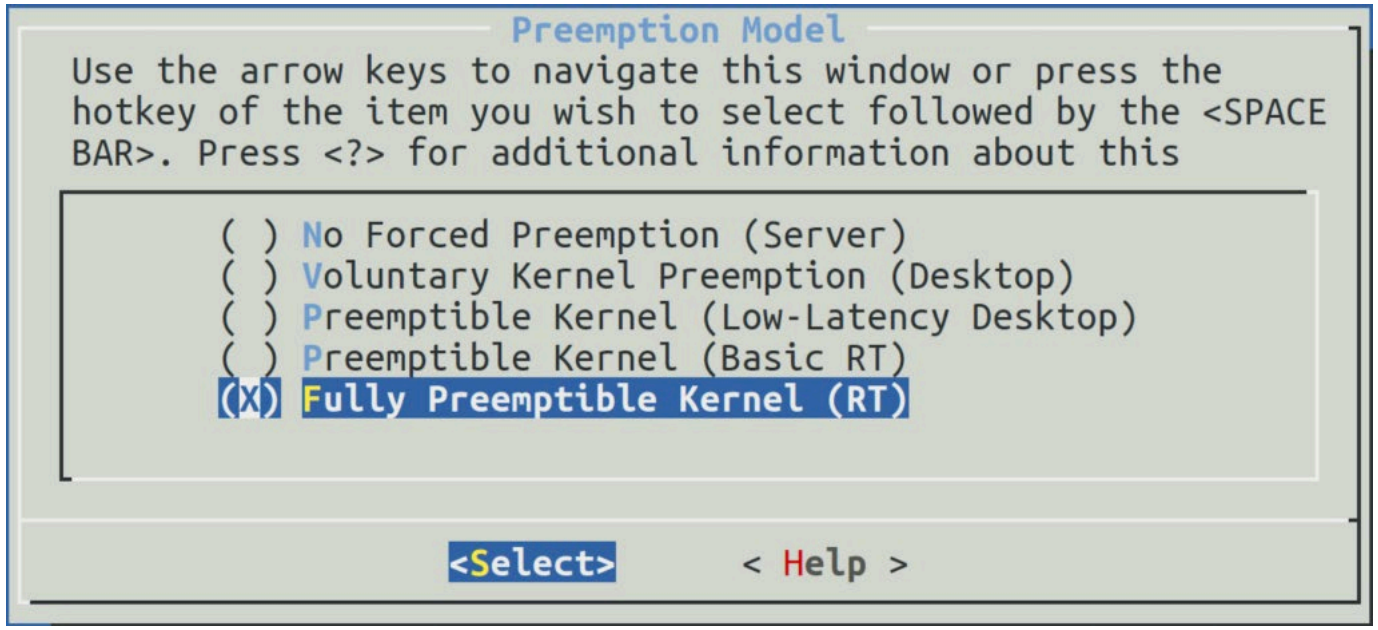
One of the simplest approaches is to make your kernel **PREEMPTIBLE!**

"The ability of the operating system to preempt or stop a currently scheduled task in favor of a higher priority task. The scheduling may be one of, but not limited to, process or I/O scheduling etc."

- Download the kernel source code
- Download the PREEMPT patch for your kernel version
- Configure your kernel
- Compile the kernel
- Reboot and select your kernel in GRUB (boot loader)!

https://wiki.linuxfoundation.org/realtime/documentation/howto/applications/preemptrt_setup

How to obtain a Linux real-time kernel?



http://youngmok.com/tutorial-how-to-make-rt-preempt-linux-with-ubuntu-18-04-02/

How to check if our system is real-time enabled?

- Check if you are using the correct kernel!

```
# uname  -a
```

- Use the **cyclictest** tool. (Part of the **rt-tests** package.)
  - measures/tracks latencies from hardware interrupt to user space
  - run at the priority level to evaluate

- Generate worst case system loads.
  - scheduling load: the **sysbench** or **hackbench** tool
  - interrupt load: flood pinging with ”ping -f”
  - serial/network load: ”top -d 0” via console and network shells

https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/rt-tests

How to check if our system is real-time enabled?

Use cyclictest and various load generation tools and methods to try to find a worst-case latency for your system.

```
# cyclictest --mlockall --smp --priority=80 --interval=1000 --distance=500
```

Monitor the CPU with:

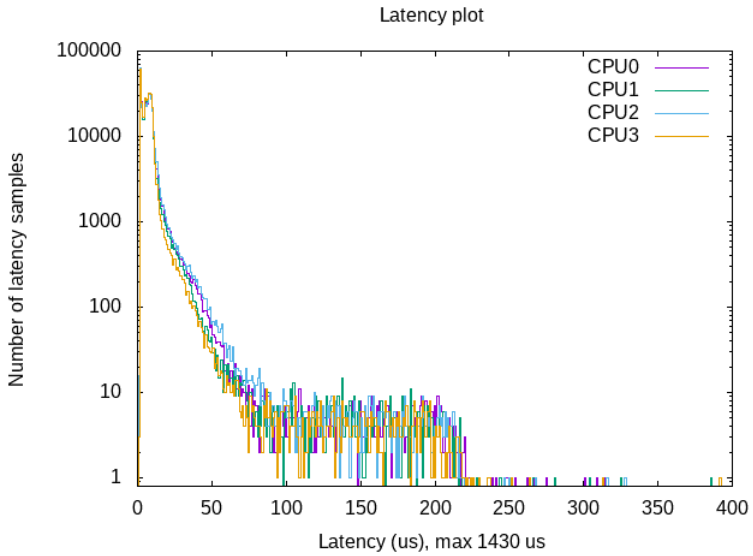
```
# htop
```

Some ideas for load generation:

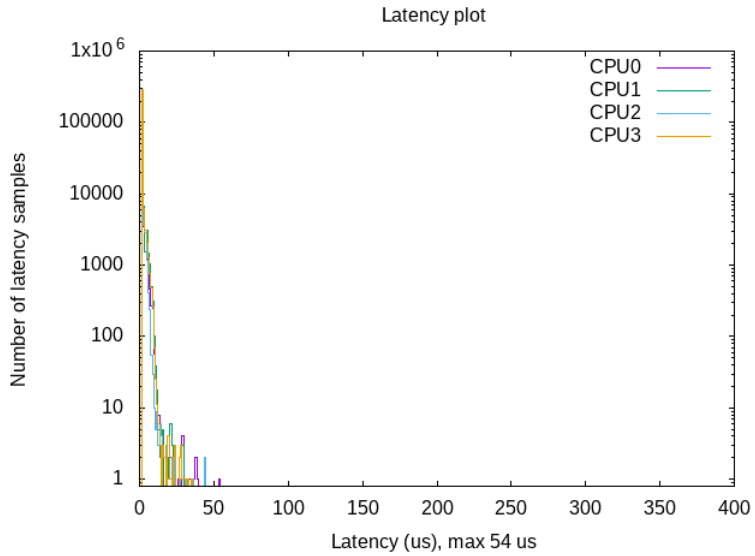
```
# sysbench cpu --threads=4 run
# while true; do hackbench; done
# top -d 0
# while true; do echo -n; done
```

https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/start

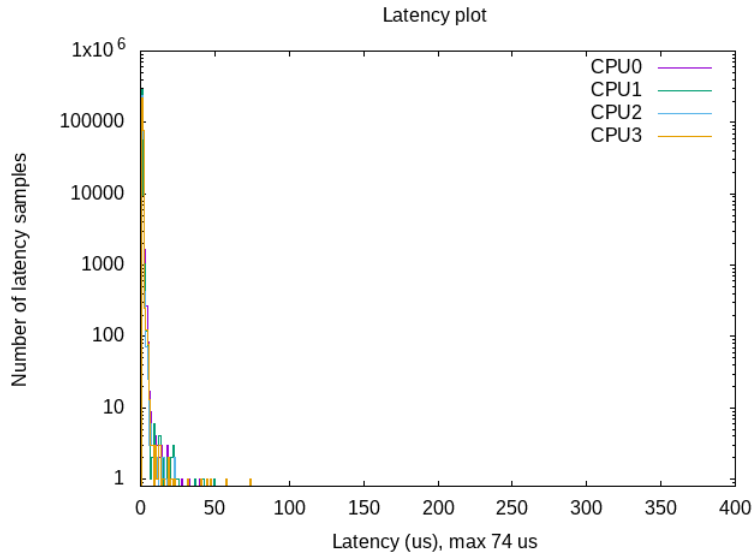
How to check if our system is real-time enabled?



Kernel 5.4.0 generic



Kernel 5.4.0 PREEMPT



Kernel 4.4.208 PREEMPT

https://www.osadl.org/uploads/media/mklatencyplot.bash

How to program a Linux real-time application?

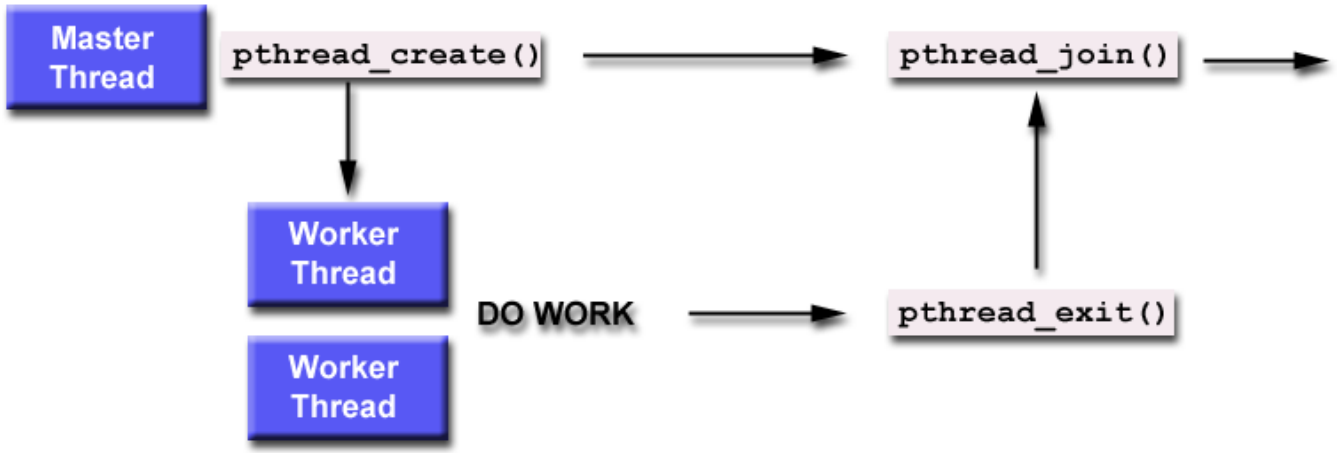
What's next:

- **How to program a real-time cyclic thread?**
- How to share variables among threads?
- How to **safely** share variables with other processes (RT and nRT)?

Code snippets can be found at: https://github.com/stex2005/realtime_ipc_exercises

How to program a real-time application?

- Use **threads!**
  - Data is implicitly shared
  - Consider data on a thread's stack private
- Set **scheduler policy** and **priority** of your worker threads
- **Create** your worker threads
- **Join** your worker threads until they end



How to program with POSIX standard?

Linux real-time features are implemented using the **POSIX standard API.**
Most developers are already comfortable with this interface.

- No exotic libraries.
- No exotic objects.
- No exotic functions.
- No exotic semantics.

POSIX: time handling

The “most standard” way to store time values for real-time processing is through the **timespec structure.**

```
#define <time.h>

struct timespec {
time_t tv_sec; // seconds (int32)
long tv_nsec; // nanoseconds
}
```

Unfortunately, the standard library does not provide any helper function to do operations with **timespec** variables. (use “**time_spec_operation.h**”)

POSIX: getting the time

To get/set the **current time**, the following functions are available:

```
#include <time.h>

int clock_gettime(clockid_t clock_id, struct timespec *tp);
int clock_settime(clockid_t clock_id, const struct timespec *tp);
```

clockid_t is a data type that represents the type of real-time clock that we want to use

POSIX: sleep functions

To suspend a thread, we can call the following function:

```
#include <time.h>

int clock_nanosleep(clockid_t clock_id, int flags,
const struct timespec *rqtp, struct timespec *rmtp);
```

This is the most flexible and complete function for suspending a thread (only available in the POSIX RT profile)

- **flags** is used to decided if we want to suspend for a relative amount of time, or until an absolute point in time. It can be **TIMER_ABSTIME** or **0** for relative intervals.

- **rqtp** is a pointer to a **timespec** value that contain either the interval of time or the absolute point in time until which the thread is suspended (depending on the flag value)

POSIX: clock types

- **Choose CLOCK_MONOTONIC.** This is a clock that cannot be set and represents monotonic time since some unspecified starting point.

- **Do not use CLOCK_REALTIME.** This is a clock that represents the ”real” time. For example, Monday 11 October 2021 15:00:00. This clock can be set by NTP, the user, etc.

- **Use absolute time values.** Calculating relative times is error prone because the calculation itself takes time.

POSIX: example of cyclic operations

```
#define PERIOD_NS 1000000

void *cyclic_thread(void *arg) {

struct timespec next;
clock_gettime(CLOCK_REALTIME, &next);

while (1) {
        timespec_add_nsec(&next, &next,PERIOD_NS);

        /* DO STUFF HERE */

        clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME,
                        &next, NULL);

return NULL;
}
```

POSIX: handling threads

`pthread_create()`
start a new thread

`pthread_join()`
wait for child thread to join

`pthread_exit()`
stop thread

`pthread_detach(thread_id)`
detached threads do not “join”

POSIX: scheduling policy

It is possible to specify the policy and the parameters by using the thread attributes before creating the thread.

```
#include <pthread.h>

int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
```

Input arguments:

- **attr** attributes
- **policy** can be **SCHED_RR**, **SCHED_FIFO** (fixed priority scheduling with or without round-robin) or **SCHED_OTHER** (standard Linux scheduler).

IMPORTANT: to use the real-time scheduling policies, the user id of the process must be root

https://wiki.linuxfoundation.org/realtime/documentation/howto/applications/application_base

POSIX: setting priority

- Set the desired priority (should range from **0** to **99** for real-time processes)

```
struct sched_param param;
param.priority = 1;

pthread_attr_setschedparam(
 pthread_attr_t *attr,
 struct sched_param *param);
```

- Don’t forget to tell the scheduler to use attributes specified by the user in **attr**

```
pthread_attr_setinheritsched(
 pthread_attr_t *attr,
 PTHREAD_EXPLICIT_SCHED);
```

POSIX: example

```
pthread_attr_t attr;
struct sched_param param;

pthread_attr_init(&attr);

pthread_attr_setschedpolicy(&attr, SCHED_FIFO);

param.sched_priority = 50; // Priority should range between 0 and 99
pthread_attr_setschedparam(&attr, &param);

pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED);

pthread_create(&pthread, &attr, cyclic_thread, nullptr);
```

Use **htop** to check process priority!

POSIX: don’t forget to

- Disable paging by locking to block in the memory some parts of the process image or the total process image

```
mlockall(MCL_CURRENT|MCL_FUTURE);
```

- Disable CPU power management with the “**set_latency_target**” trick:

if the file /dev/cpu_dma_latency exists, open it a write a zero into it. This will tell the power management system not to go to a high CPU C-state, forcing 0 us latency.

https://access.redhat.com/articles/65410

POSIX: warning!

It is important to underline that only the superuser (root) can assign real-time scheduling paramters to a thread, for security reasons.

if a thread with **SCHED_FIFO policy executes forever in a loop**, no other thread with lower priority can execute.

All other threads **will starve.**

How to program a Linux real-time application?

What's next:

- How to program a real-time cylic thread?
- **How to share variables among threads?**
- How to share variables with other processes (RT and nRT)?

Code snippets can be found at: https://github.com/stex2005/realtime_ipc_exercises

How to share variables among threads?

Use **pthread_mutex(es)!**

A mutex is a special kind of **binary semaphore**, with several restrictions:

- It can only be used for **mutual exclusion** (and not for synchronization)
- If a **thread locks** the mutex, only the **same thread** can **unlock** it!
- If the **mutex is locked**, no other threads can **work on the mutex**

Multiple locks should be used, each for a **set of shared data items** that is disjoint from another set of shared data items (no single lock for everything)

POSIX: pthread mu(tually)ex(clusive)!

```
#include <pthread.h>

pthread_mutex_t m;

int pthread_mutex_init(pthread_mutex_t *m,
const pthread_mutex_attr_t *attr);

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

- **Lock** corresponds to a **wait** on a binary semaphore;
- **Unlock** corresponds to a **post** on a binary semaphore;

POSIX: example

```
pthread_mutex_t mylock;
pthread_mutex_init(&mylock, NULL);
...
pthread_mutex_lock(&mylock);
...
... critical section ...
...
pthread_mutex_unlock(&mylock);
...
pthread_mutex_destroy(&mylock);
```

- POSIX mutex locks can be used for thread synchronization
- Threads share user space, processes do not

POSIX: example

Thread 1:

```
pthread_mutex_lock(&queue_lck);
... add element to shared queue ...
pthread_mutex_unlock(&queue_lck);
```

Thread 2:

```
pthread_mutex_lock(&queue_lck);
... remove element from shared queue ...
pthread_mutex_unlock(&queue_lck)
```

mutexes can be used to lock operations on a shared queue

What if we need synchronization of threads?

To simplify the implementation of **critical section** with mutex, it is possible to use **condition variables**

A **condition variable** is a special kind of synchronization primitive that can only be used together with a mutex

These can be associated with **pthread mutex objects** to provide synchronized notification.

POSIX: pthread condition variables

```
int pthread_cond_wait(pthread_cond_t *restrict cond,
pthread_mutex_t *restrict mutex);
```

A call to pthread_cond_wait() is equivalent to:

- release the mutex
- block on the condition
- when unblock from condition, lock the mutex again

```
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

To unblock one thread on a condition use **cond_signal**
To unblock all the threads locked on a condition use **cond_broadcast**

POSIX: example sender-receiver

```
#include <pthread.h>
pthread_mutex_t lock;
pthread_cond_t cond;
```

Code of receiver:

```
pthread_mutex_lock(&lock);
pthread_cond_wait(&cond, &lock);
/* we have been signaled */
/* do the work */
pthread_mutex_unlock(&lock);
```

Code of sender:

```
pthread_mutex_lock(&lock);
/* do the work */
pthread_cond_broadcast(&cond);
pthread_mutex_unlock(&lock);
```

N.B. The sender should notify the receiver before releasing the lock associated with the conditional variable.

POSIX: example with two conditions

Declarations

```
pthread_mutex_t mutex;
pthread_cond_t notempty, notfull;
```

Initialization

```
pthread_mutex_init(&mutex, NULL);
pthread_cond_init(&notempty, NULL);
pthread_cond_init(&notfull, NULL);
```

A consumer

```
while (1)
{ pthread_mutex_lock(&mutex);
  if (container is empty)
    pthread_cond_wait(&mutex, &notempty);
  get item from container
  pthread_cond_signal(&mutex, &notfull);
  pthread_mutex_unlock(&mutex);
}
```

A producer

```
while (1)
{ pthread_mutex_lock(&mutex);
  if (container is full)
    pthread_cond_wait(&mutex, &notfull);
  add item to container
  pthread_cond_signal(&mutex, &notempty);
  pthread_mutex_unlock(&mutex);
}
```

How to program a Linux real-time application?

What's next:

- How to program a real-time cyclic thread?
- How to share variables among threads?
- **How to share variables with other processes (RT and nRT)?**

Code snippets can be found at: https://github.com/stex2005/realtime_ipc_exercises

How to share variables with other processes?

Interprocess communication (IPC)!

- Processes do not automatically share data (do not share user space)
- Use **files** to share data? Slow, but portable
- Use **shared memory segments** to allocate shared variables between two or more processes! Two or more processes can access the common memory.

POSIX: shared memory API

Use **ipcs** to check shared memory usage

```
#define DEFAULT_KEY_ID 1234
key_t key;
key = ftok("/bin",DEFAULT_KEY_ID)
```

Creates the shared memory key.

```
struct shared_memory_packet_t{
        /* your shared variables */
        bool flag;
};
```

Builds your shared memory packet.

```
shared_memory_packet_t packet;
shmid = shmget(key,sizeof(packet),0666|IPC_CREAT)
```

Returns the shared memory identifier for a given key (key is for naming and locking)

POSIX: shared memory API

Use **ipcs** to check shared memory usage

```
data = (shared_memory_packet_t*) shmat(shmid,(void*)0,0);
```

Attaches the segment identified by a shared memory identifier and returns the address of the memory segment

```
shmdt(data)
```

Detaches the shared memory segment

```
shmctl(shmid,IPC_RMID,NULL)
```

Deletes the segment with `IPC_RMID` argument

N.B. These operations (apart from shmctl) has to be done on all processes!

POSIX: example write-read

Use **ipcs** to check shared memory usage

```
#include <iostream>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
using namespace std;

int main()
{
    // ftok to generate unique key
    key_t key = ftok("shmfile",65);

    // shmget returns an identifier in shmid
    int shmid = shmget(key,1024,0666|IPC_CREAT);

    // shmat to attach to shared memory
    char *str = (char*) shmat(shmid,(void*)0,0);

    cout<<"Write Data : ";
    gets(str);

    printf("Data written in memory: %s\n",str);

    //detach from shared memory
    shmdt(str);

    return 0;
}
```

```
#include <iostream>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
using namespace std;

int main()
{
    // ftok to generate unique key
    key_t key = ftok("shmfile",65);

    // shmget returns an identifier in shmid
    int shmid = shmget(key,1024,0666|IPC_CREAT);

    // shmat to attach to shared memory
    char *str = (char*) shmat(shmid,(void*)0,0);

    printf("Data read from memory: %s\n",str);

    //detach from shared memory
    shmdt(str);

    // destroy the shared memory
    shmctl(shmid,IPC_RMID,NULL);

    return 0;
}
```

How to program a Linux real-time application?

What's next:

- How to program a real-time cyclic thread?
- How to share variables among threads?
- **How to safely share variables with other processes (RT and nRT)?**

Code snippets can be found at: https://github.com/stex2005/realtime_ipc_exercises

How to safely share variables with other processes?

Use shared mutexes and conditional variables over shared memory!

Activate the **shared feature** if the conditional variable is accessed by **multiple processes** in shared memory.

POSIX: shared mutexes and conditional variables

```
pthread_mutexattr mutexattr;
pthread_mutexattr_init(&mutexattr);
pthread_mutexattr_setpshared(&mutexattr, PTHREAD_PROCESS_SHARED);
pthread_mutex_init(&data->lock, &mutexattr);
pthread_mutexattr_destroy(&mutexattr);

pthread_condattr condattr;
pthread_condattr_init(&condattr);
pthread_condattr_setpshared(&condattr, PTHREAD_PROCESS_SHARED);
pthread_cond_init(&data->cond, &condattr);
pthread_condattr_destroy(&condattr);
```

&data->lock and & **data->cond** are shared mutexes and conditional variables in the shared memory packet

N.B. the shared mutexes and conditional variables have to be initialized only once, in the master/main process!

POSIX: examples

Try this at home!

- One real-time process (100Hz), one non-real-time process (1kHz)
- The real-time process writes at 100 Hz in the shared memory segment (e.g. increments a shared counter)
- The non-real-time process (through locks/conds) waits for the real-time to release a condition. Then, the non-real-time process reads the shared variables and sends them back. (e.g. decrements a shared counter).
- **What should happen?**
- **Try to use all the tools!** (e.g. shm, pthread_mutex, pthread_cond, clock_nanosleep, pthread_setparam, etc…)

Useful links:

https://events19.linuxfoundation.org/wp-content/uploads/2017/12/e-ale-rt-apps_John-Ogness.pdf

https://www.cs.fsu.edu/~engelen/courses/HPC/SharedMemory1.pdf

http://retis.sssup.it/~lipari/courses/str09/10.rtprogramming-handout.pdf

Thank you for your attention!

If you have any questions please contact Stefano Dalla Gasperina: stefano.dallagasperina@polimi.it