UNIVERSITAT DE BARCELONA

Programació d'Arquitectures Encastades
Real Time Operating Systems

David Roma (droma@ub.edu)

Real Time Operating Systems

1. OS
2. Threads and context switching
3. Scheduler
4. RTOS
5. TI-RTOS
6. TI-RTOS threads
7. Scheduling example
8. Hardware Interrupts (HWI), Software Interrupts (SWI) &Tasks
9. Race conditions
10. Semaphores
11. Priority Inversion
12. Mutex gates
13. Queue
14. Memory management

Operating System (OS)

When programming, we need to take into account the extra objectives, aside from the specifications of our project:

1. *How can we lower the development time?*
2. *How can we make our code more easy to maintain?*
3. *How can we ease future integration of more features?*

Using an OS instead of a bare metal approach solves some of this questions.

1. An OS incorporates drivers, lowering the development time and **making our code portable** from one platform to another.
2. Inside an OS we will usually have some multitasking support. Therefore, we can have one **independent task** for every functionality we wan to integrate. This also eases **reusing code** in different projects.
3. Having independent task allows us to add new features without modifying the behavior of the existing ones.

Operating System vs Bare Metal

Operating System	Bare metal
Drivers: - Portability - Validated functions - Integrated runtime checks	Optimum usage of resources (depends on us)
Multitasking/multithreading	Lowest possible latency (again, if done right)

Using an operating System has some penalties. We will add an overhead to our code but for larger designs or design that may grow the advantages outweigh the disadvantages.

Threads and context switching

We define a thread as a sequence of programmed instructions (i.e. function) that has is own independent and usually permanent memory allocation. So, different threads will executed it's code without interfering with the memory of the others.

In a single processor system, threads will share between them the CPU time. Now suppose while one thread is running, another thread has priority over the currently running one and the first thread can be blocked (**preemptible**). Changing from the current thread to another one is what we call **context switching**.

A context switch also happens due to **hardware interruptions**. Hence, everything we will see here explained for threads will also be true when an external interruptions triggers an IRQ function.

Threads and context switching

Lets assume the currently running thread was just executing x = a + b, which in our system will consist of three steps:

1. Copying a form the memory to the CPU registers
2. Copying b from the memory to the CPU registers
3. Adding them and writing the result to the memory

And it hadn't finished when a context switch happens, it was just on step 2! So, context switching is more than only changing the current program counter to the new code to execute. It's also about storing the current registers so that when we go back to the current thread it will resume its operations undisturbed.

Don't worry, the responsible to do this will be the OS!

Scheduler

The scheduler is the part of the OS that is responsible to determinate when a context switching needs to happen and which of the existing threads will be the next to run.

There are different types of schedulers:

- **FIFO**: there is no preemption, all tasks run until completion. No prioritization, once one is finished, the first task to enter the queue is executed.
- **Earliest deadline first**: every task has a time deadline. The task with the nearest deadline will be the next to be executed.
- **Round-robin scheduling**: all tasks have a fixed equal cpu time. Once this time is over, the next task is executed.
- **Priority preemptive scheduling**: highest priority threads always run first (TI-RTOS).

Threads controlled by a scheduler will be in one of this three states:

- Running: executed by the CPU.
- Ready: ready to be executed but waiting for CPU time.
- Blocked: waiting for an event or resource.

Real Time Operating System

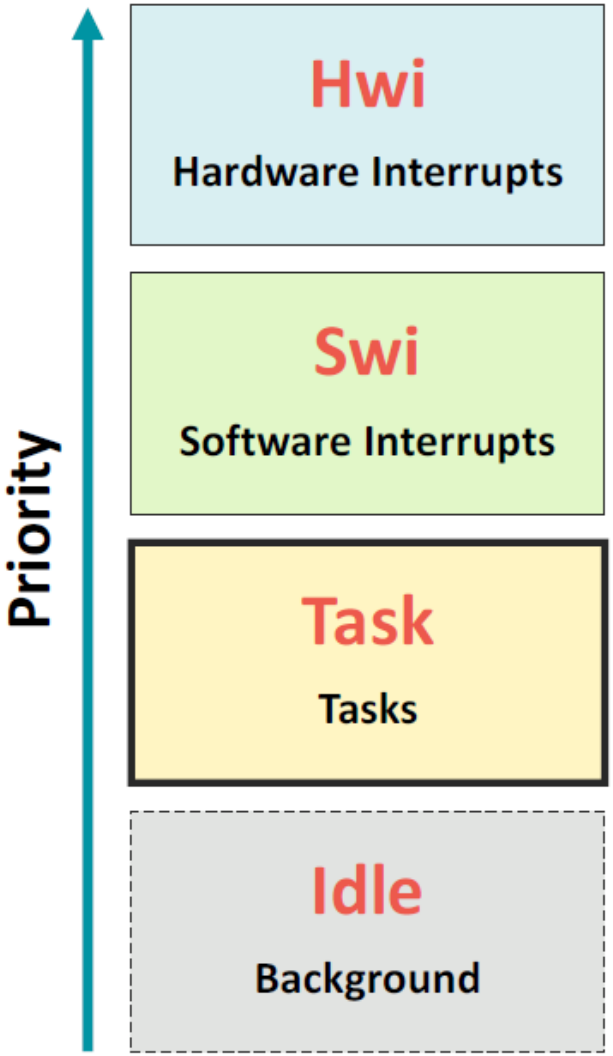
A real time OS is an OS which main focus are **minimum interrupt and thread switching latency**. So, a real time operating system should allow us to respond to an interrupt in the shortest time possible and always in a deterministic way, independently which was the current thread being executed.

We can find two types of RTOS:

- **Time-sharing**: switches tasks (if needed) on a regular clocked interrupt.
- **Event-driven**: switches task only when an event of higher interrupt needs servicing (TI-RTOS).

In RTOS, dynamic memory allocation can be an issue. It's recommended that all required memory allocation is specified statically at compile time.

TI-RTOS Thread Types



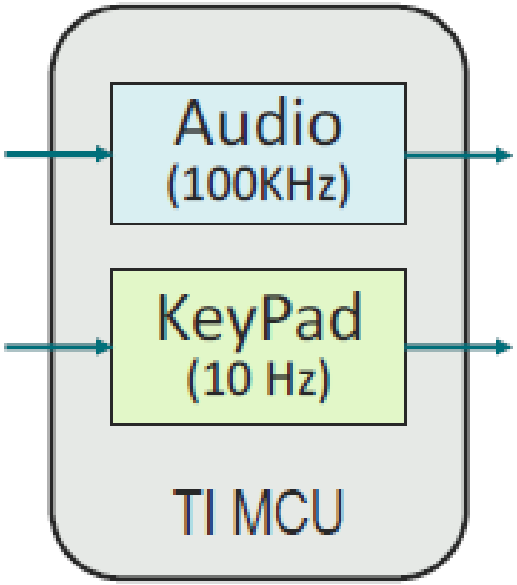
- Hardware event triggers Hwi to run
- BIOS handles context save/restore, nesting
- Hwi triggers follow-up processing
- Priorities set in silicon

- Software *posts* Swi to run
- Performs Hwi '*follow-up*' activity (process data)
- Up to 32 priority levels (16 on C28x/MSP430)

- Usually enabled to run by posting a '*semaphore*' (a task signaling mechanism)
- Designed to run concurrently – pauses when waiting for data (semaphore)
- Up to 32 priority levels (16 on C28x/MSP430)

- Runs as an infinite while(1) loop
- Users can assign multiple functions to Idle
- Single priority level

Scheduling Problem – Two Threads



Problem Definition: you have two different threads that need to be serviced independently

- Will one routine conflict with the other?
- How do you SCHEDULE each thread?
- Is one "thread" higher PRIORITY than the other?

Let's explore a few options we can use to SCHEDULE these two threads…

```
TimerA_ISR()
{
 read sample;
   Audio
}
```

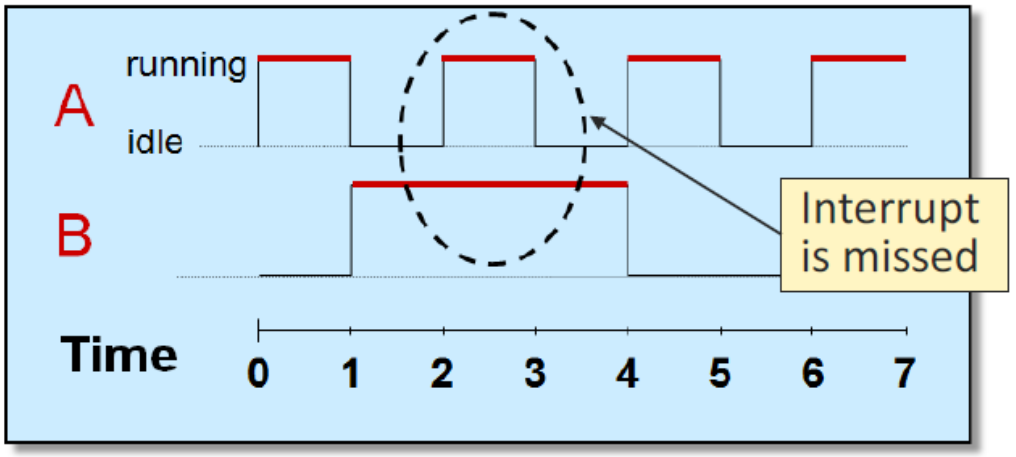
```
TimerB_ISR()
{
 read keypad;
   KeyPad
}
```

```
main()
{
  while(1);
}
```

Solution #1 – an *interrupt driven system* places each function in its own ISR

	Period	Compute	Usage
Audio	10µS	5µS	50%
Keypad	100ms	1ms	1%
			51%

- While CPU usage is fine, one interrupt may block the other (instantaneous):



UNIVERSITAT DE BARCELONA

```
Audio_ISR()
{
 read sample;
 post_Audio;
}
```

```
Keypad_ISR()
{
 read keypad;
 post_KeyPad;
}
```

```
main()
{
  init_stuff;
  while(1);
  BIOS_start();
}
```

Hwi – Hardware INT

- *Context save/restore (done by BIOS)*
- *Hard real-time "read"*
- *Post Swi for follow-up*

Swi – Software INT

- *Posted by software*
- *PROCESS data*
- *User can select priority*

Idle – Background

- *Runs multiple fxns inside of a while(1) loop*

HI

LO

If we can DECOUPLE the processing from the TRIGGER (Hwi):

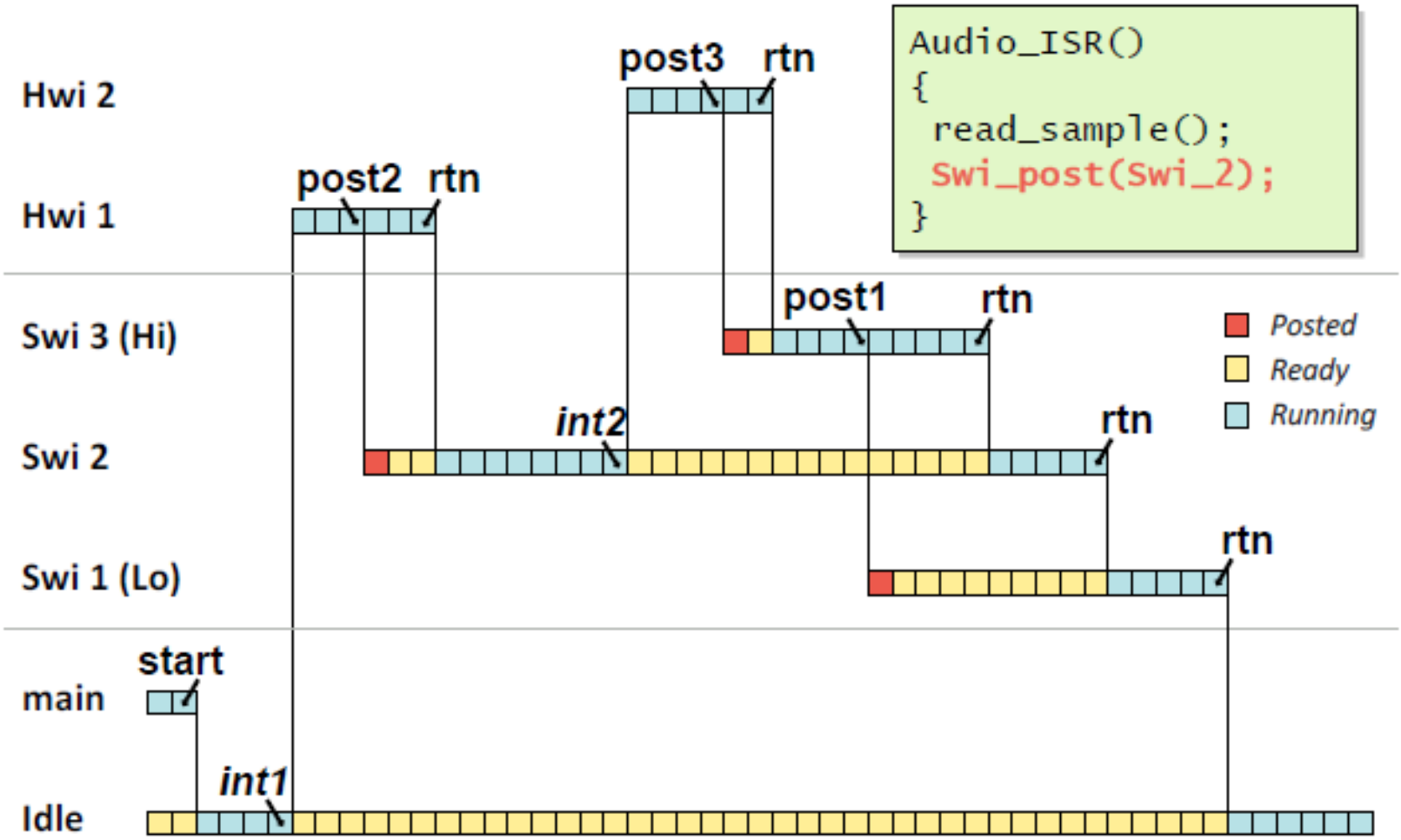
- ISRs become VERY short (no need for nesting)
- Configure PRIs of threads via software
- Add as many threads as we need (no limit)
- Touch ONE, no CODE changes to the others !

Swi can be seen as a special kind of tasks

What is the difference between a THREAD and a FUNCTION ?

UNIVERSITAT DE BARCELONA

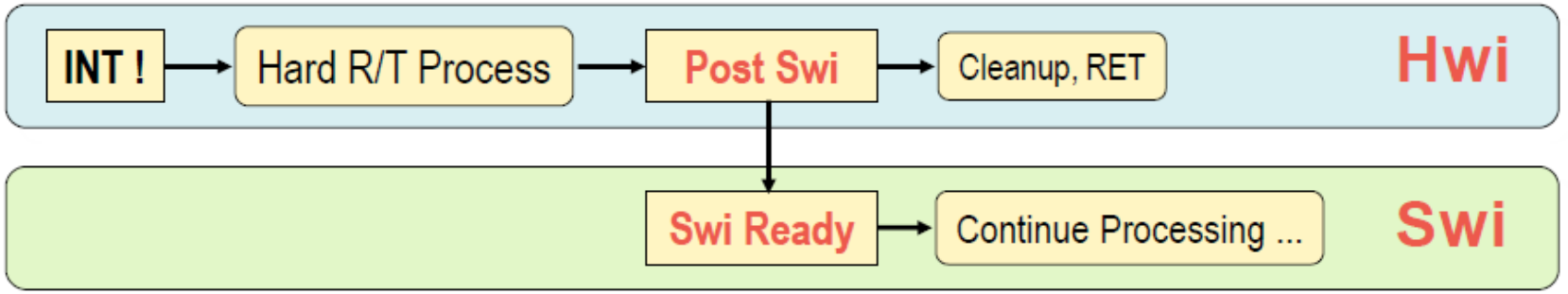
Priority Based Scheduling



User SETs the priorities, BIOS executes them

When to use a software Interrupt (swi)?

Execution flow for flexible real-time systems:



Hwi

- Fast response to INTs
- Min context switching
- High priority for CPU
- Limited # of Hwi possible

isrAudio:

```
*buf++ = *XBUF;
cnt++;
if (cnt >= BLKSZ) {
   Swi_post(swiFir);
   cnt = 0;
   pingPong ^= 1;
}
```

Swi

- Latency in response time
- Context switch
- Selectable priority levels
- Scheduler manages execution

- TI-RTOS Kernel provides for Hwi and Swi management
- TI-RTOS Kernel allows the Hwi to post a Swi to the ready queue

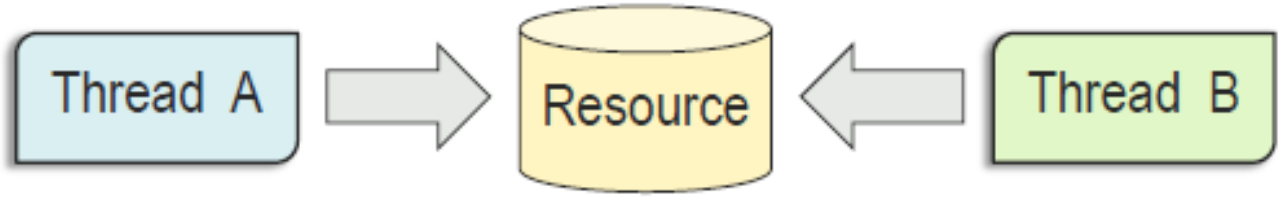
Thread memory sharing

"Producer – Consumer" Model

- Thread A ***produces*** a buffer or Msg and places it in a container.
- Thread B blocks until available, then ***consumes*** it when signaled (no contention)
- Data communication is achieved using BIOS "containers" (objects)

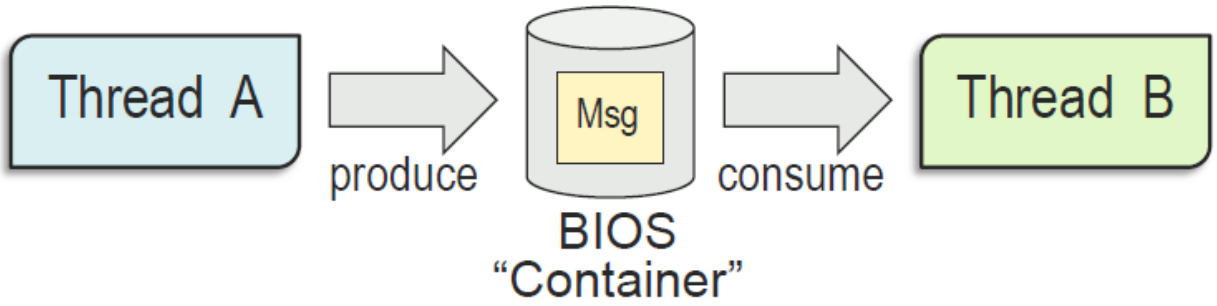
"Concurrent Access" Model

Resource = Data or Peripheral



- Any thread could access "Resource" at any time (no structured protocol/container)
- Pre-emption of one thread by another can cause contention or priority inversion

Thread memory sharing



How it Works:

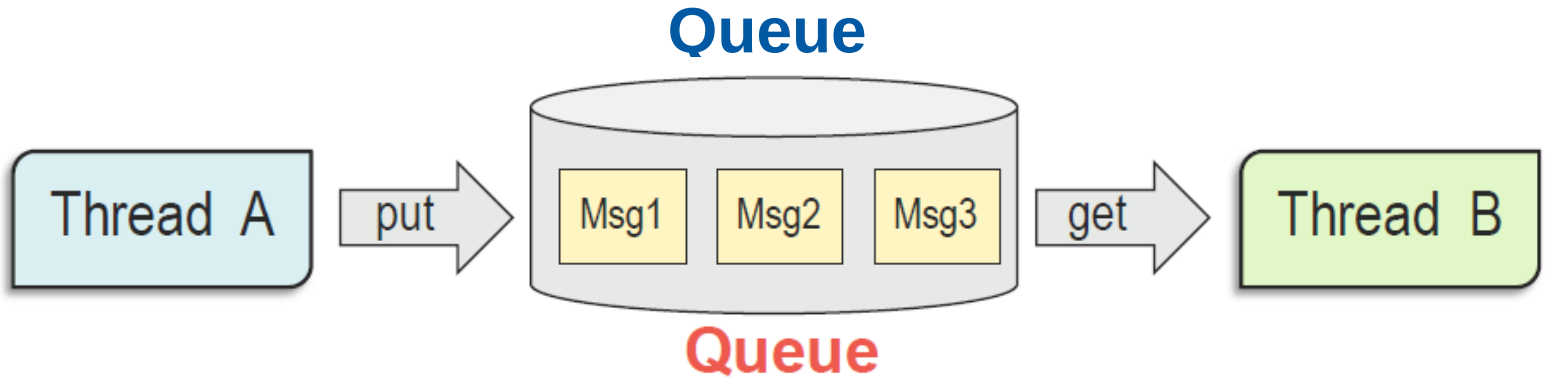
- Data is UNIDIRECTIONAL – one thread waits for the other to ***produce***
- Thread B often BLOCKS until data is ***produced***

Advantages:

- Both threads use same protocol – contention is often avoided
- Threads become more "modular" and therefore reuse improves

Examples:

- BIOS: **Queue, Mailbox**
- May have built-in synchronization or user can add signaling via *Semaphores* and *Events*



- A **Queue** is a BIOS object that can contain *anything* you like
- "Data" is called a "Msg" – simply a structure defined by the user
- Msgs are "reclaimed" on a FIFO basis
- Key APIs:

```
Queue_put();
Queue_get();
```

- *Advantages*: simple, not copy based
- *Disadvantage*: no signaling built in

Race conditions

Suppose we have to independent threads which increment by 1 a common value (i.e. `cnt++`) without any synchronization mechanism between them. Also, reading, incrementing and writing back this value is not an **atomic operation** (it is interruptible). In this case, we can have the following situations:

Thread 1	Thread 2		cnt
read		←	0
increase			0
write		→	1
	read	←	1
	increase		1
	write	→	2

Thread 1	Thread 2		cnt
read		←	0
	read	←	0
increase			0
	increase		0
write		→	1
	write	→	1

A **race condition** happens when the output is dependent on the timing of events and therefore the systems behavior becomes unpredictable. **Mutual exclusion** (MutEx) is a propriety of concurrency control to prevent race conditions.

Mutex

A Mutex allows guaranteeing that only ONE thread at a given time is allowed to executed a certain code area, i.e., it is a **locking mechanism** (mostly used for "concurrent access" model).

```
#include <pthread.h>

void concurret_access_function() {
        pthread_mutex_lock(pthread_mutex_t* mutex);
        //Only one thread may execute this code area at the same time
        cnt++;
        pthread_mutex_unlock(pthread_mutex_t* mutex);
}
```

Hence, adding a mutex lock around the *cnt*++ operation of the previous example will ensure that only one thread will be able to execute this function while the other thread will be **blocked** waiting until the first one finished.

Semaphores

A semaphore is like a **counter variable** on which the increment or decrement operations are guaranteed to be **atomic**. It's most used for **signaling** (most useful for "producer-consumer" model).

A binary semaphore will have only to possible values, 0 and 1.

We may also change the previous code so that both threads before executing the statements with the shared variable will check if the semaphore is 1 (green light). If it is, they will decrement it to zero (red light) and go on with their code. And if it is already zero, there will wait until it is one again (the thread will be blocked).

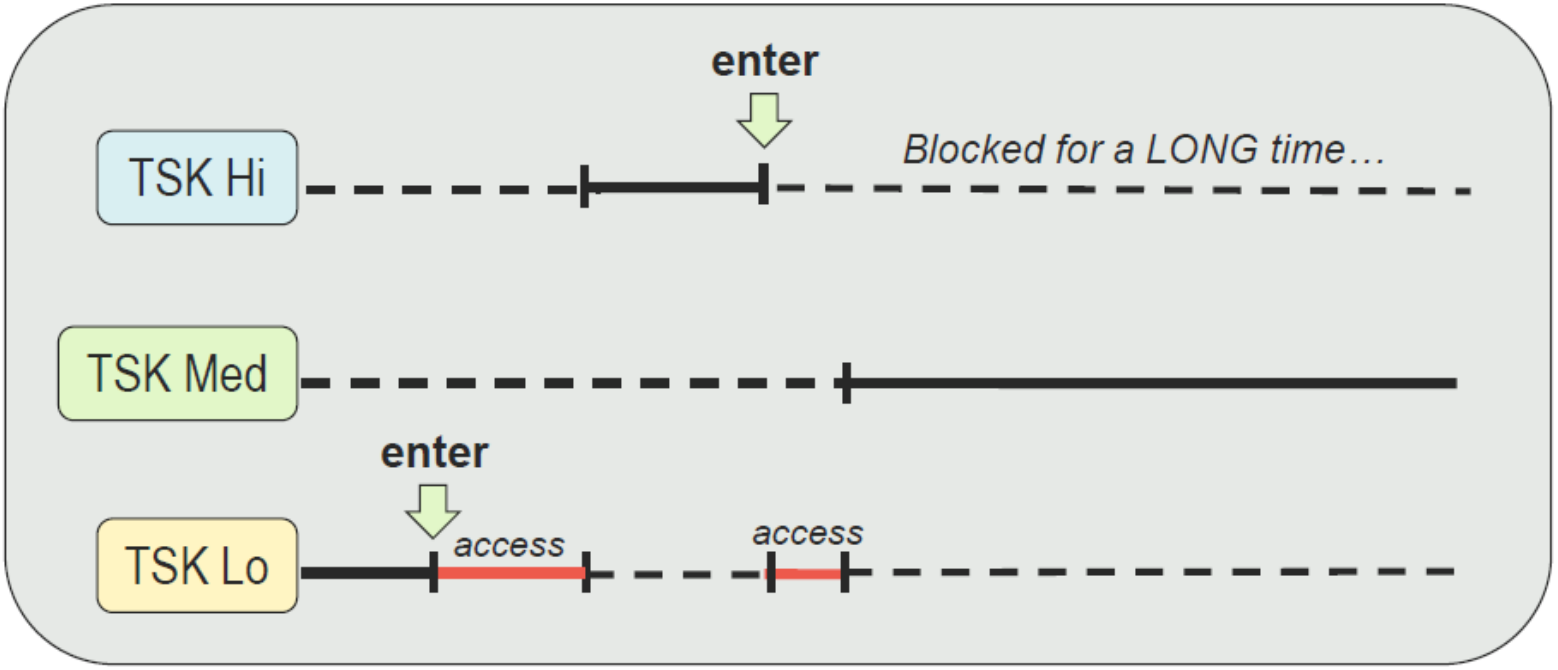
This way, only one thread at time will be able to modify the variable *cnt*.

```
#include <semaphore.h>

void thread_A(void) {
    data = fecth_outside_data();
    push_data(data);
    sem_post(sem_t* semaphore);
}
```

```
void thread_B(void) {
    sem_wait(sem_t* semaphore);
    data = pop_data();
}
```

Priority inversion

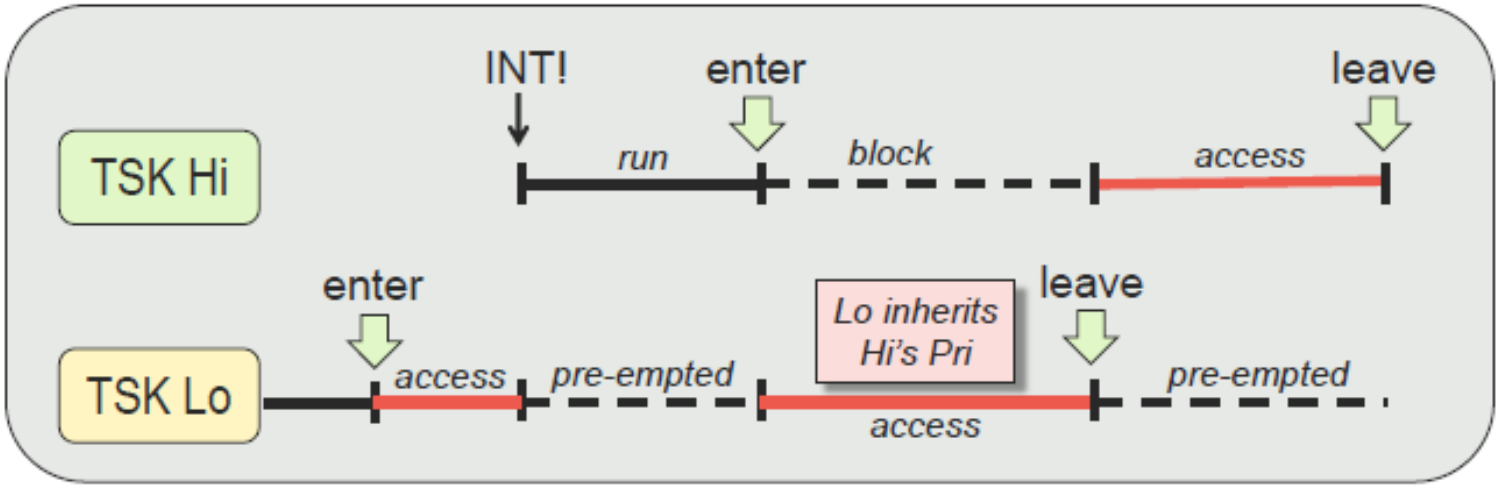


- TSK Lo enters critical section (using Semaphores or Gates)
- TSK Hi pre-empts, but blocks to let TSK Lo leave critical section
- TSK Med pre-empts TSK Lo *and holds off TSK Hi indefinitely*

We still need to think thoroughly the interaction with other threads!

[1] https://www.rapitasystems.com/blog/what-really-happened-to-the-software-on-the-mars-pathfinder-spacecraft

TI-RTOS solution: mutex gates with priority inheritance



- Use the following code in BOTH (TSK Hi and TSK Lo)

```
gateKey = GateMutexPri_enter(gateMutexPri);  // enter Gate
cnt += 1;                                    // protected access
GateMutexPri_leave(gateMutexPri, gateKey);   // exit Gate
```

- TSK Lo inherits priority of TSK Hi if TSK Hi requests resource access (enter) and then returns to original Pri after "leave".
- Advantages: simple to code, does automatic Task_setPri() of TSK Lo

Memory management

Static Memory

- **Link Time:**
  - Allocate Buffers
- **Execute:**
  - Read data
  - Process data
  - Write data

- Allocated at **LINK** time
- \+ Easy to manage (less thought/planning)
  \+ Smaller code size, faster startup
  \+ Deterministic, atomic (interrupts won't mess it up)
- \- Fixed allocation of memory resources
- Optimal when most resources needed concurrently

Dynamic Memory (HEAP)

- **Create:**
  - Allocate Buffers
- **Execute:**
  - R/W & Process
- **Delete:**
  - FREE Buffers

- Allocated at **RUN** time
- \+ Limited resources are SHARED
  \+ Objects (buffers) can be freed back to the heap
  \+ Smaller RAM budget due to re-use
- \- Larger code size, more difficult to manage
  \- NOT deterministic, NOT atomic
- Optimal when multi threads share same resource or memory needs not known until runtime

SYS/BIOS allows either method