EGH456 Embedded Systems

Lecture 10

Concurrent Access and Contention Problems

Dr Chris Lehnert

Contents

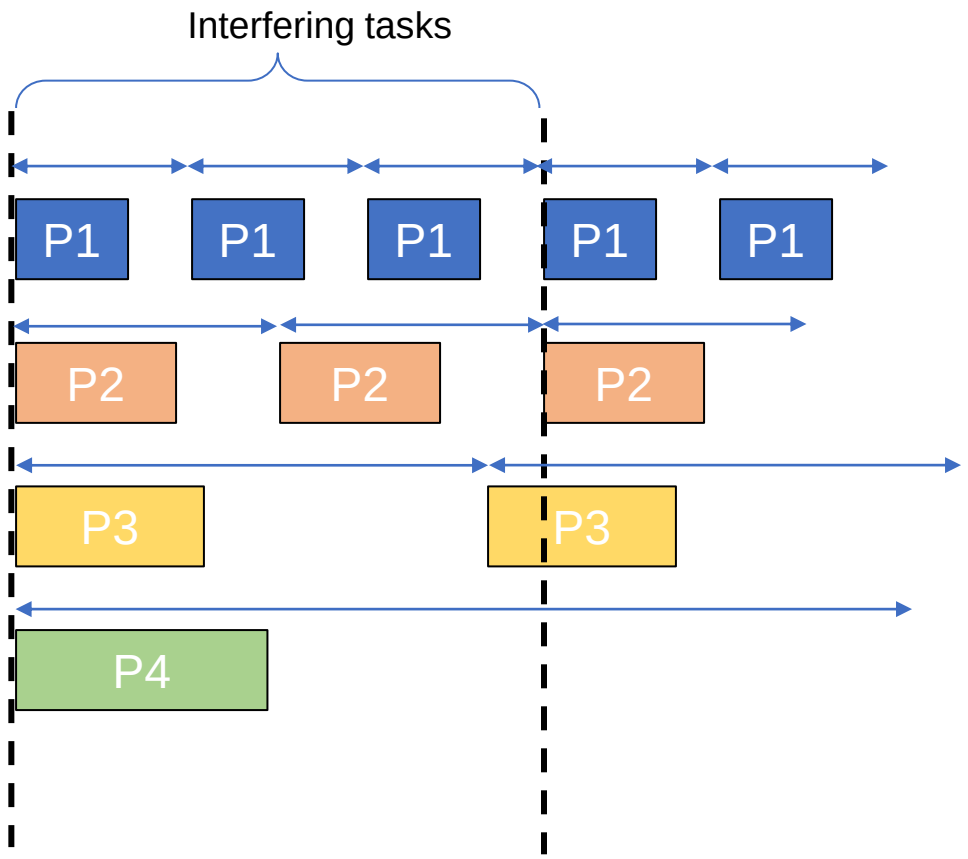
- Concurrent Access Model
- Priority Inversion problem
- Priority Inheritance

Last Lecture - Scheduling Strategies

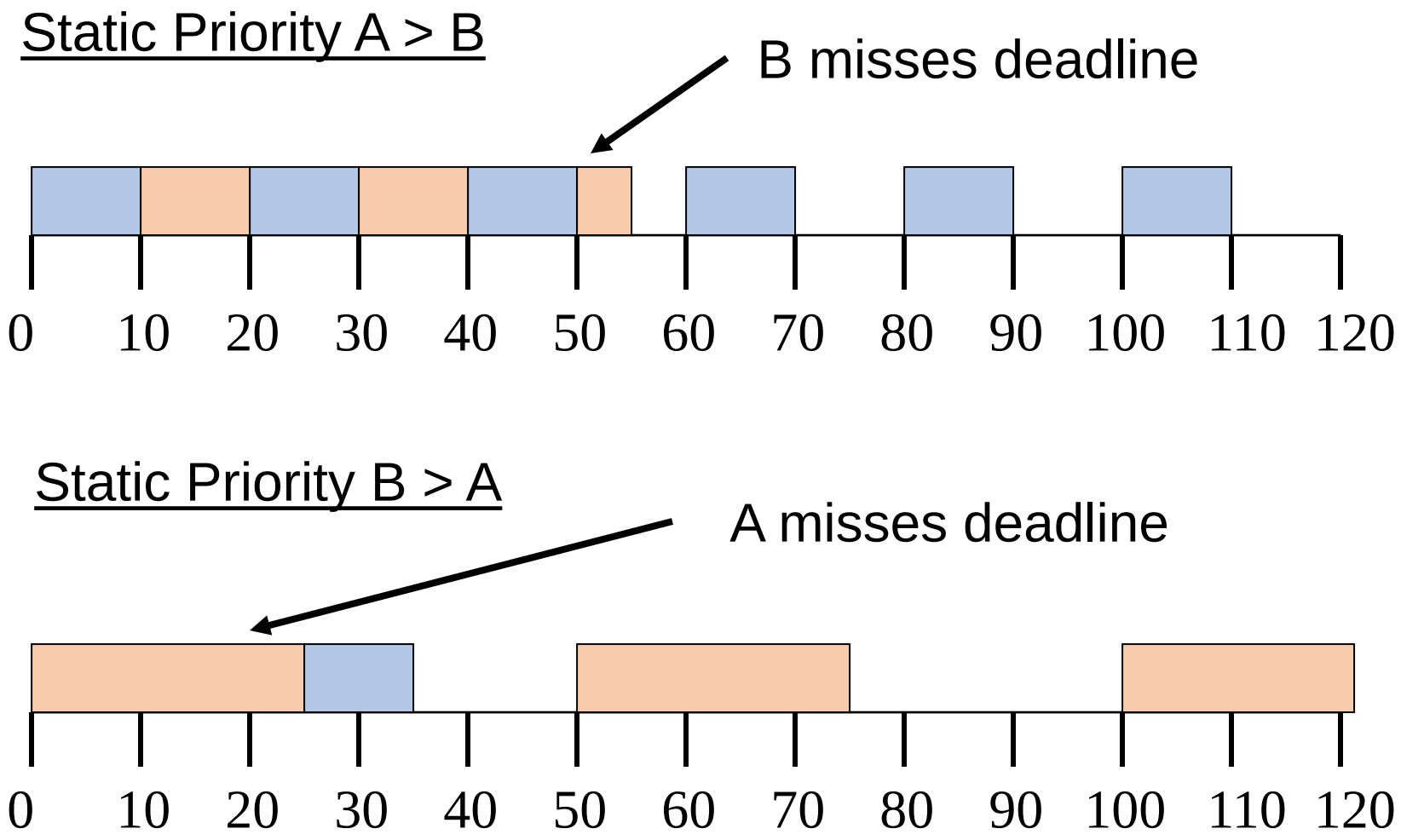
- Rate Monotonic Scheduling
  - Highest priority for highest frequency task
  - Static priority
- Earliest Deadline First
  - Highest priority to closest deadline
  - Dynamic priority
- Stu Monotonic
  - All threads at equal priority, raise threads that don't meet deadlines
- Minimum Laxity, Deadline monotonic and many more…

Critical Instant

- Scheduling state that gives worst response time

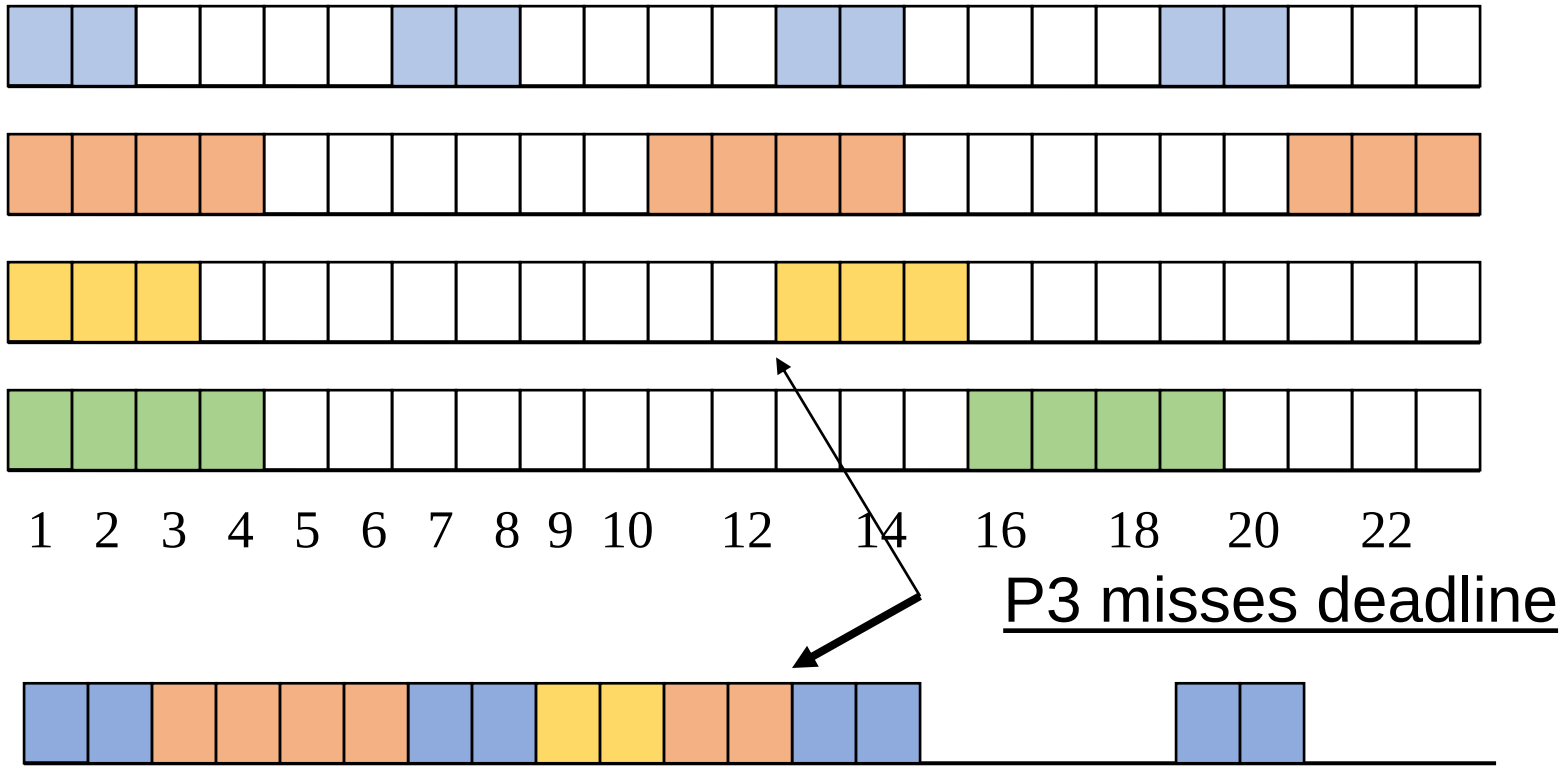


Static Priority



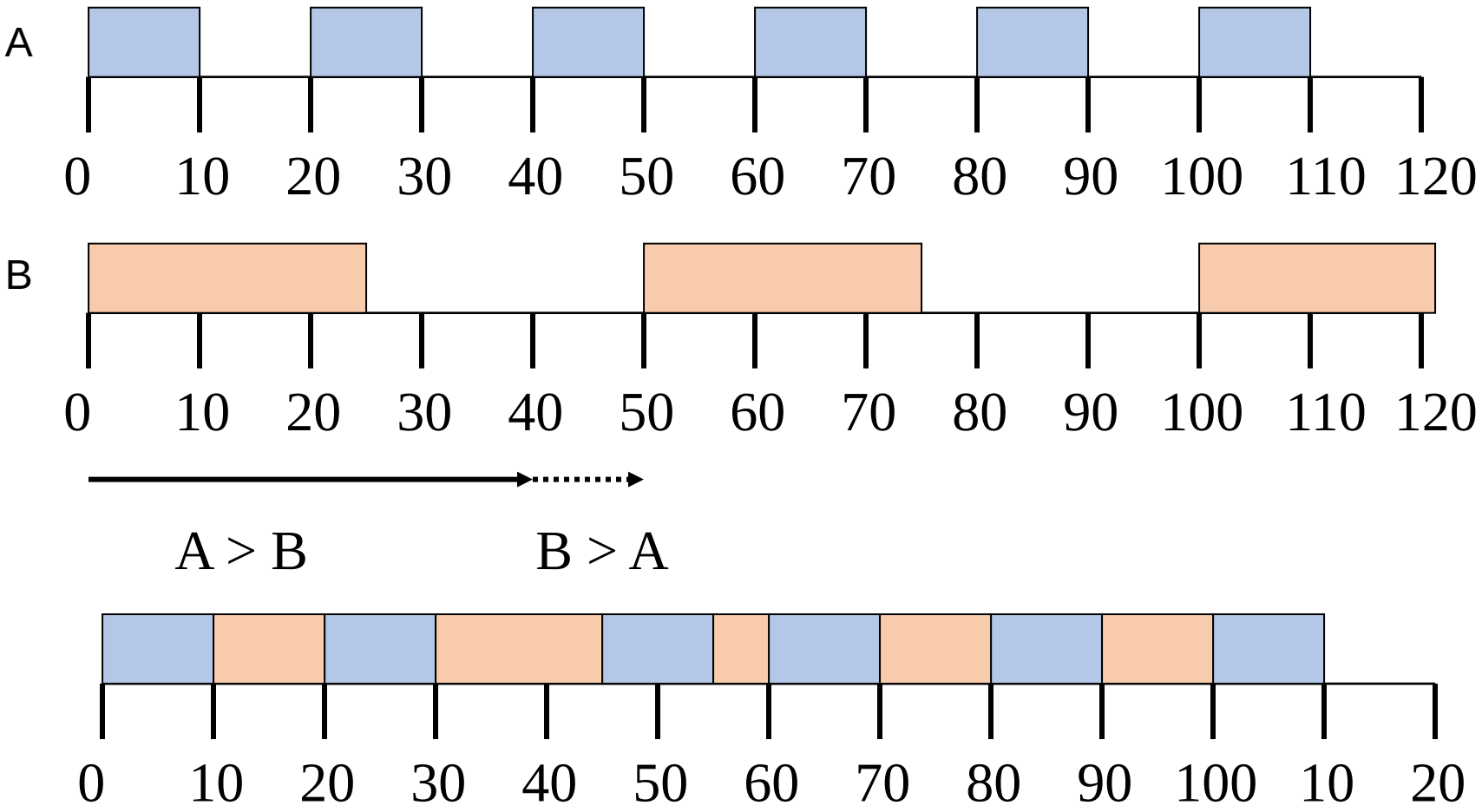
Rate Monotonic Scheduling

Schedulable bound for 2 tasks = 83%. Combined utilization of P1, P2 = 73.33% SATISFIED



P1 and P2 can be guaranteed to meet deadlines. In general CPU utilisations are lower and more tasks can be guaranteed to meet deadlines.

Earliest deadline first

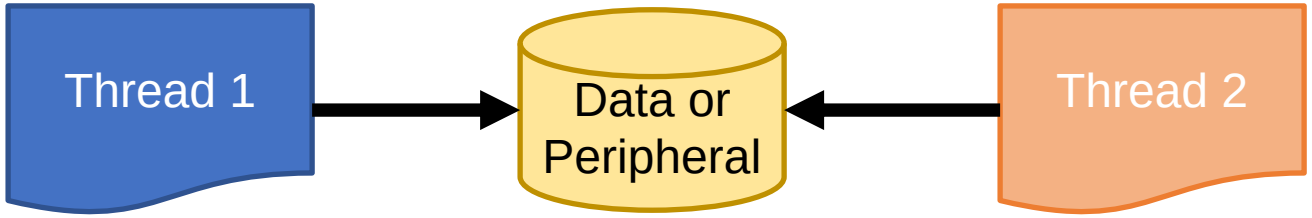


Resource Sharing

Producer – Consumer Model

- Thread 1 produces a buffer or Msg and places into BIOS “container”
- Thread 2 blocks until available, then consumes it when signalled (no contentions

Concurrent Access Model

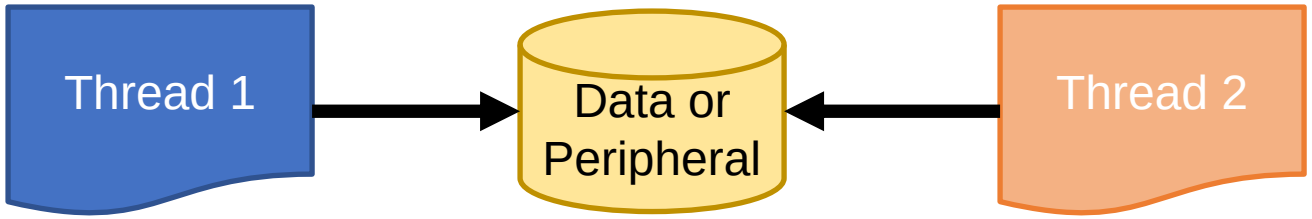


- Any thread could access resource at any time (no structured protocol)
- Pre-emption of one thread by another can cause contention

Concurrent Access Model

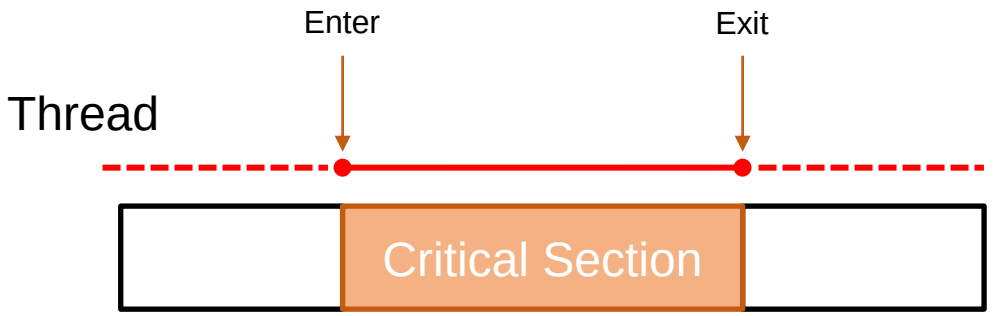
Concurrent Access Model

- Any thread could access resource at any time (no structured protocol)
- Must add protection to avoid contention between different Priority threads
- **Disadvantage:**
  - Pre-emption of one thread by another can cause contention
  - Priority Inversion & Deadlock can occur if not explicitly accounted for
- **Solution**: Scheduler management, MUTEX's, Same priority

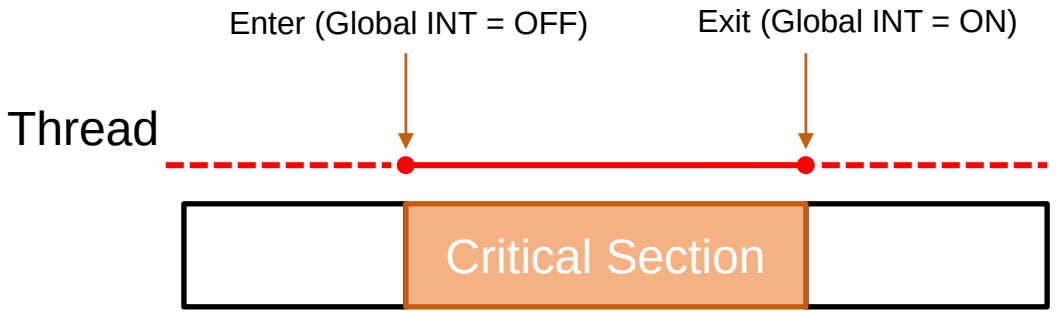


Critical Resource Protection

- For example if a Task and Hwi are sharing a resource



- For example if a Task and Hwi are sharing a resource then the only way to protect a critical section is by turning off interrupts (Hwi can't pend/block)



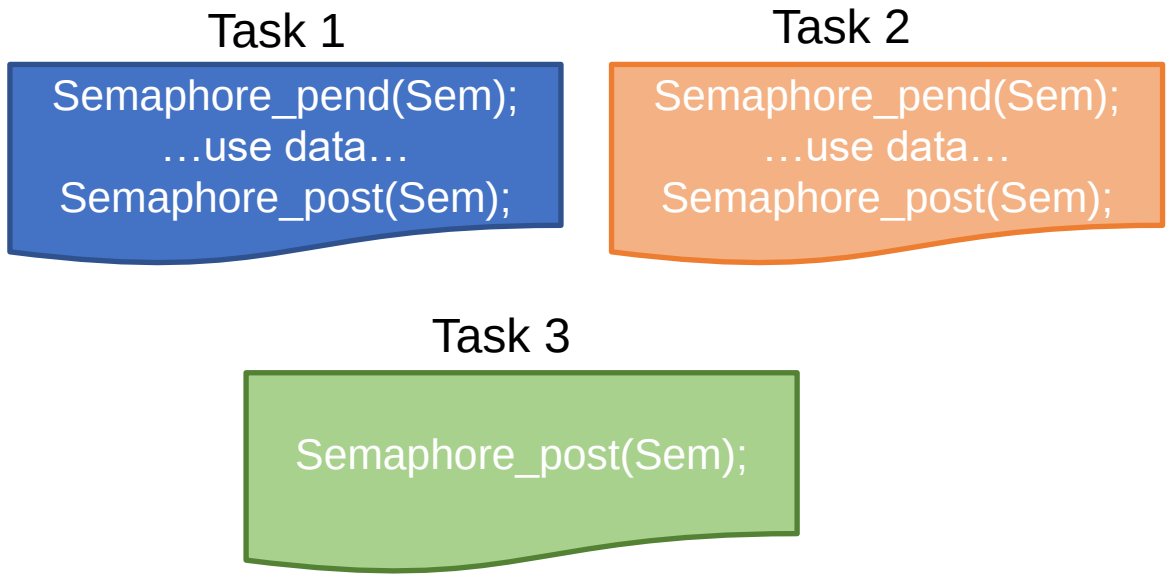
Mutual Exclusion

- MUTEX – **MUT**ual **Ex**clusion
- Goal is to allow only one thread to access a critical section at a time
- Commonly used in systems to protect a critical resource being accessed by multiple threads
- Many different methods can provide Mutual Exclusion

Semaphore ≈ MUTEX

- A **semaphore** can act like a MUTEX using an initial count of 1
  - Pros: Common, simple
  - Cons: Does not protect from priority inversion (Semaphore is FIFO queue), Can be posted by any thread, therefore potentially dangerous.

Semaphore: Sem
Initial Count = 1



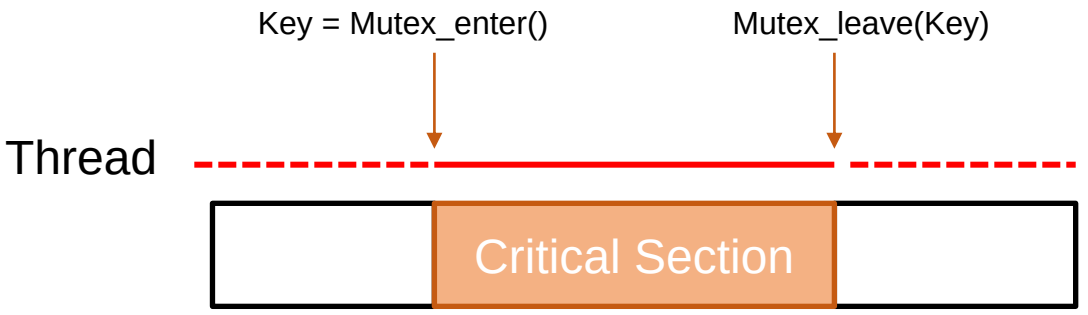
MUTEX Concept

Commonly implemented using a lock and key:

1. A thread locks the MUTEX object and is given a key
2. Thread accesses critical section
3. Thread unlocks the MUTEX using the key

Ensures **key** is owned by the thread that locks the MUTEX

Only the thread with the key can unlock the MUTEX



Semaphore vs MUTEX

- MUTEX
  - **Locking mechanism** used to synchronise access to a resource
  - Only **one** task can acquire the mutex (get gate key)
  - Ownership associated with MUTEX (acquire key) and only the owner can release the lock
- Semaphore
  - **Signalling mechanism** indicating something has happened such as an interrupt or condition has occurred
  - Can also be used to protect critical sections
  - There are not owned by a task and any thread can post or pend to the semaphore. This is potentially dangerous!

TI-RTOS Gates

- Gates are TI-RTOS objects to prevent concurrent access to critical regions
  - Differ in SYS/BIOS based on thread type and how they lock critical regions
- **Gates** disable pre-emption for each thread type
  - **GateHwi** (disables/enables interrupts)
  - **GateSwi** (disables/enables software interrupts)
  - **GateTask** (disables/enables task switching)
- **GateMutex** lock a critical region and **blocks** so only use in Tasks
  - **GateMutex** (uses a binary semaphore)
  - **GateMutexPri** (uses a binary semaphore and priority inheritance)

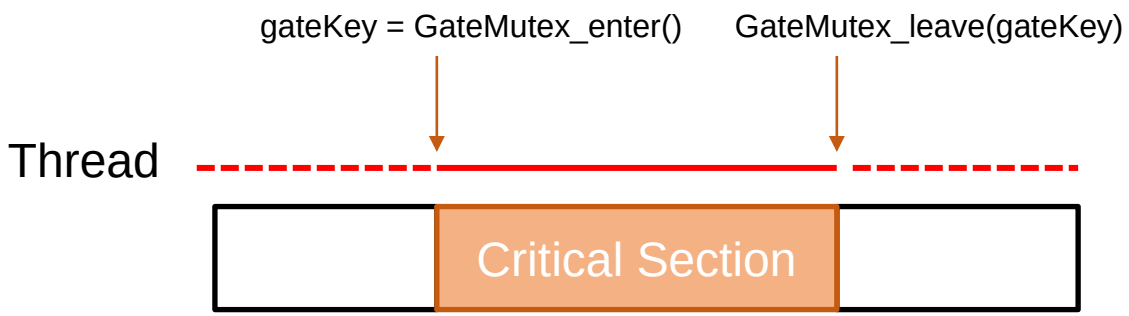
TI-RTOS Gates - Between threads types

Critical Region Shared between	Hwi	Swi	Task
Hwi	GateHwi	GateHwi	GateHwi
Swi	GateHwi	GateSwi	GateSwi
Task	GateHwi	GateSwi	GateTask, GateMutex

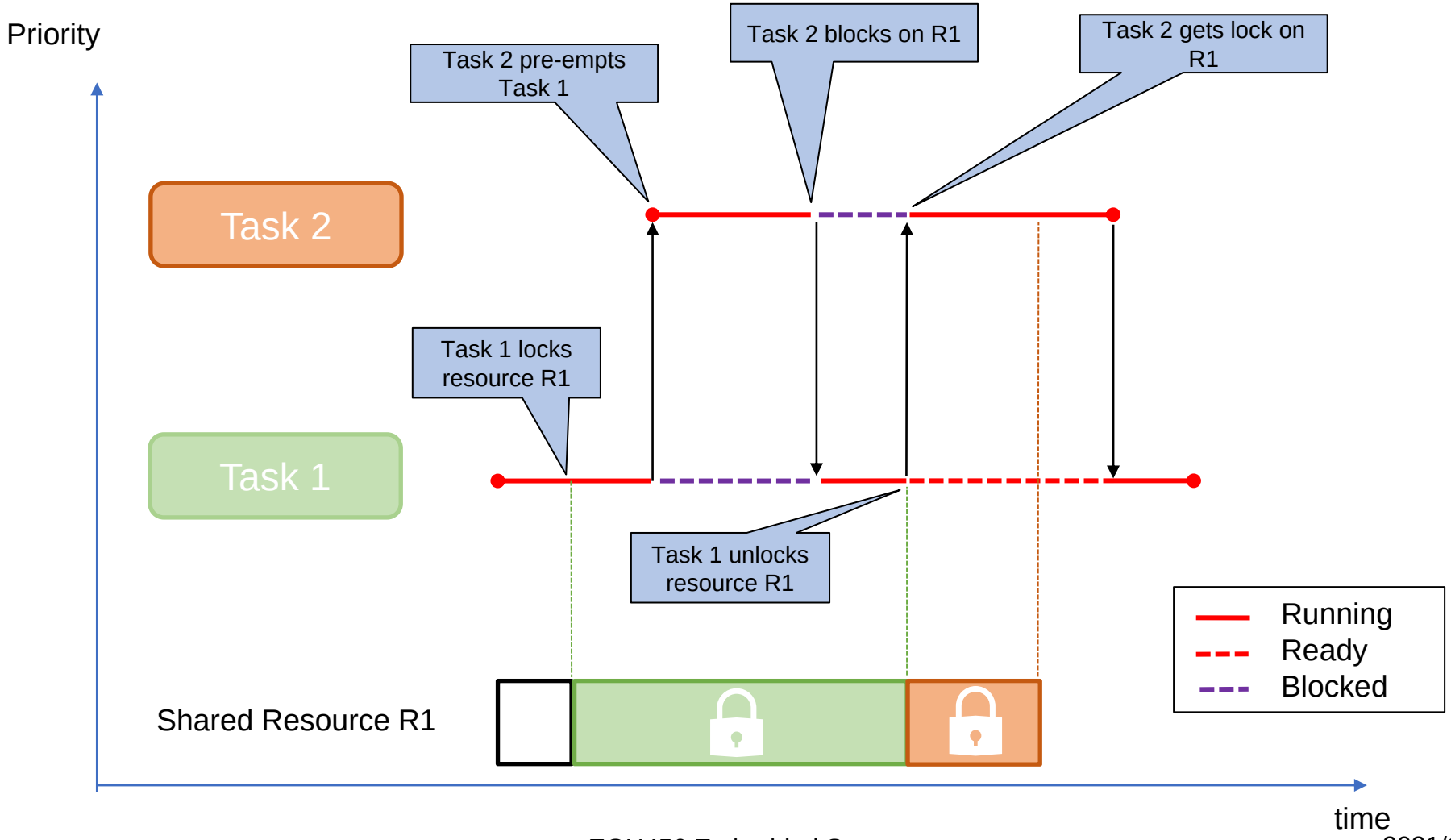
GateMutex Example

- Basic GateMutex example code (same code for both tasks threads)

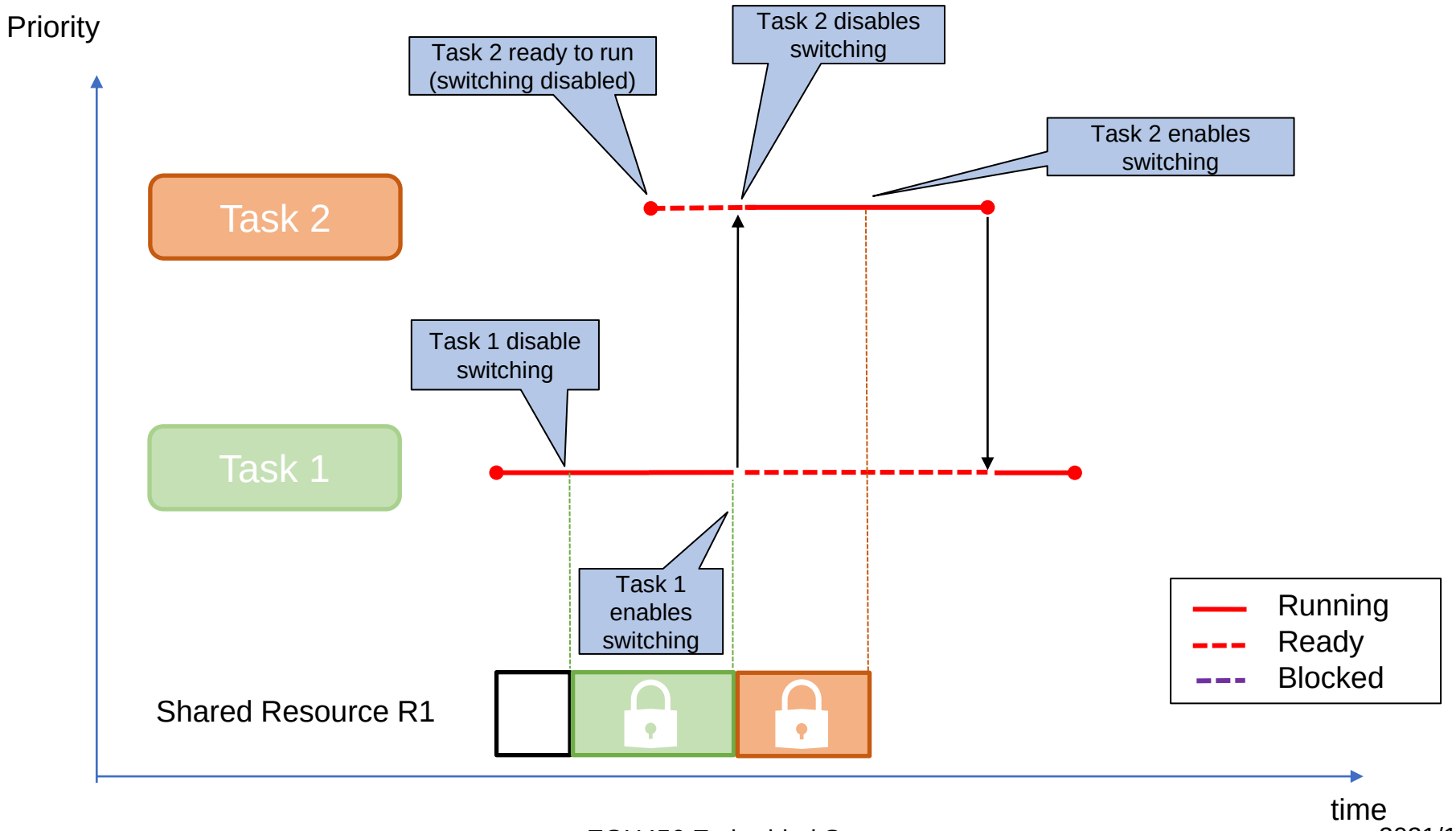
```
gateKey = GateMutex_enter(gateMutex);   // enter Gate

cnt += 1;                               // protected access

GateMutex_leave(gateMutex, gateKey);   // exit Gate
```



GateMutex Example



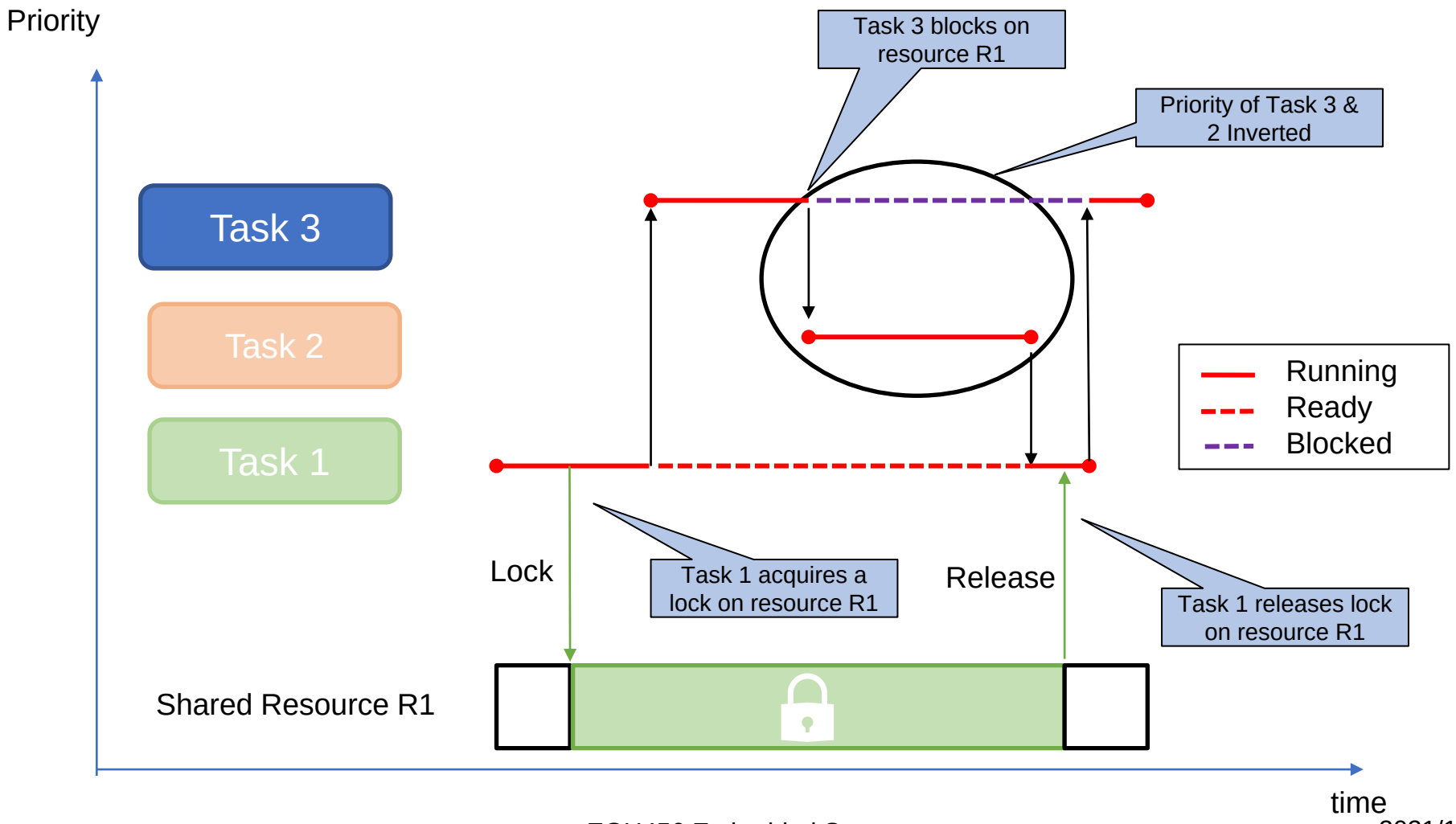
GateTask Example



Potential Issue – Threads waiting in FIFO queue

- Semaphores or MUTEX's by default use a FIFO queue to handle concurrent access
- Usually only while the task holding the resource is in its **critical section**
- What if the waiting task has a **higher** priority?
  - A priority inversion may occur

What's a Priority Inversion?



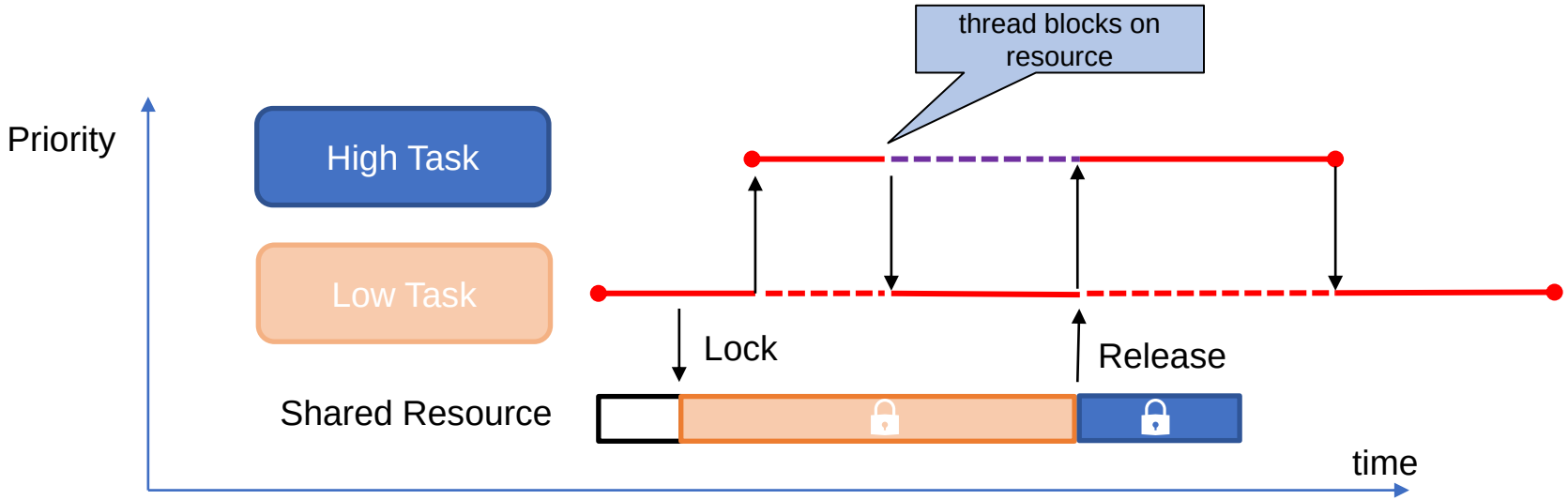
Priority Inversion

- A priority inversion occurs when a high priority task is indirectly pre-empted by a medium priority task **inverting** the relative priorities of the two tasks

- Priority inversion is a situation where a high priority task is prevented from running by a lower priority task because it has to wait for a resource being held by a lower priority task.

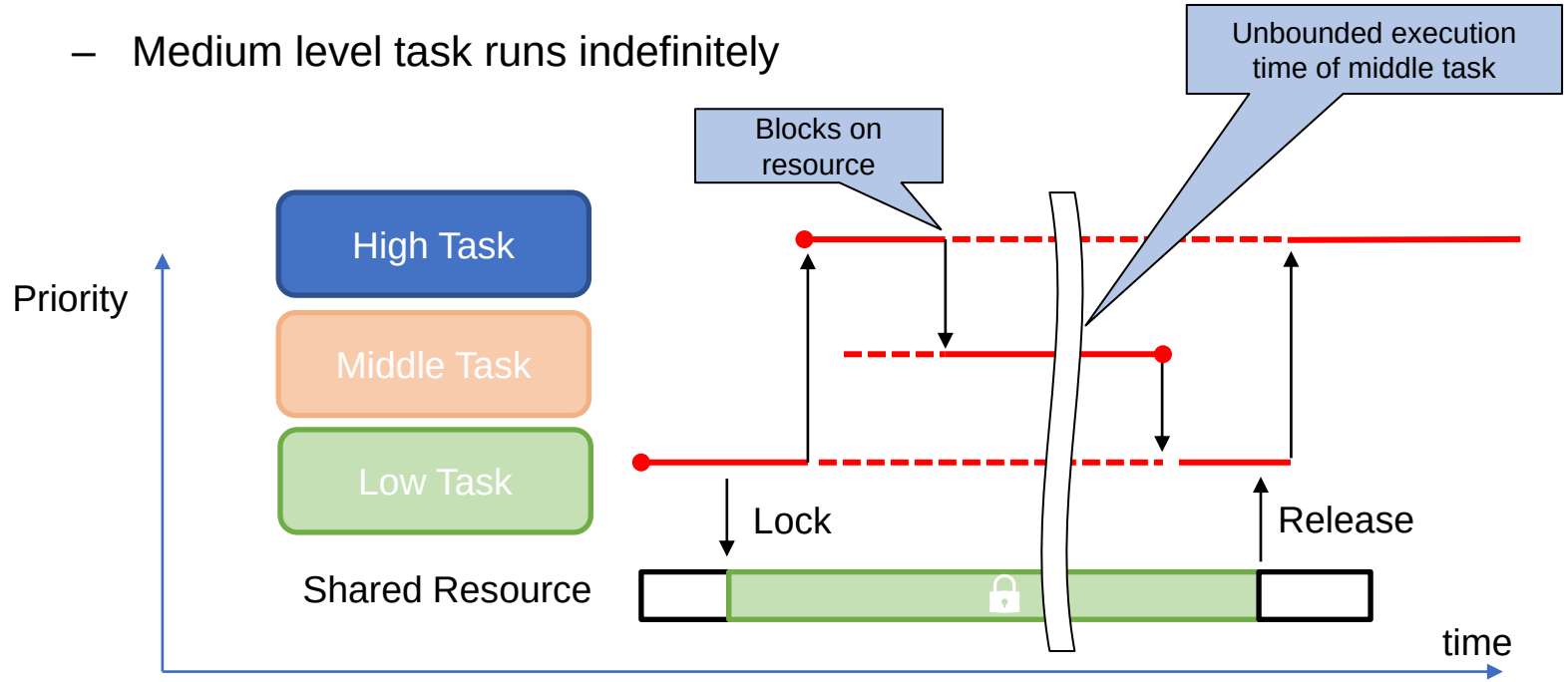
Bounded Priority Inversion

- Low priority task acquires lock but before releasing the resource is pre-empted by higher priority task. Higher task is forced to wait for resource to be released
- lasts a short period of time (time for lower task to finish with resource)



Unbounded Priority Inversion

- Potentially indefinite when an intervening task extends a bounded priority inversion
- For unbounded priority inversion to occur there must be at least 3 tasks.
  - While low level task locks resource, medium task is unblocked, pre-empting the low task
  - Medium level task runs indefinitely

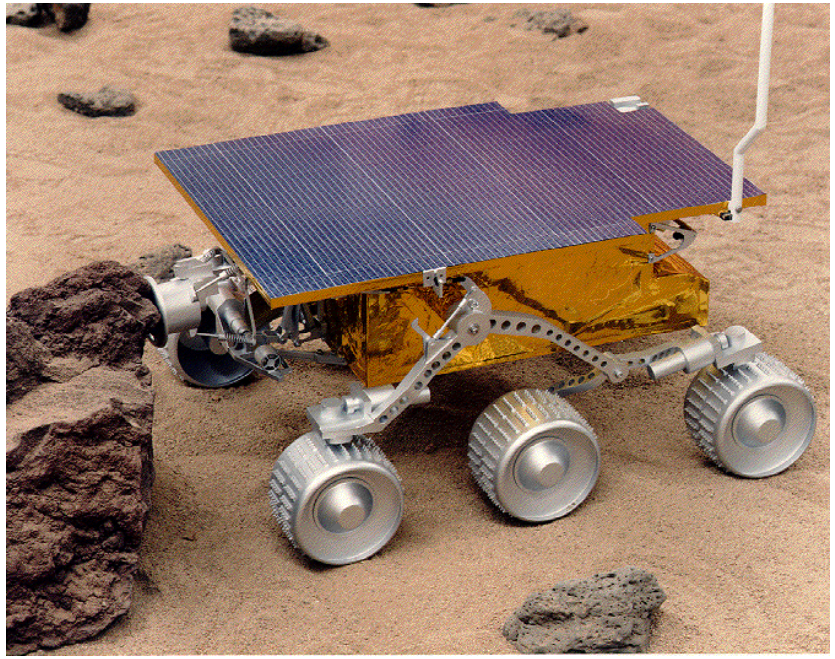


Mars Pathfinder Case

- System used RTOS VxWorks
- Radiation Hardened IBM Risc 6000 Single Chip (Rad6000 SC) CPU with 128 MB of RAM and 6 MB of EEPROM
- In 1997 the priority inversion problem occurred in NASA Mars Pathfinder mission's Sojourner rover vehicle used to explore the surface of Mars.
- Time sliced scheduler for 1/8th sec

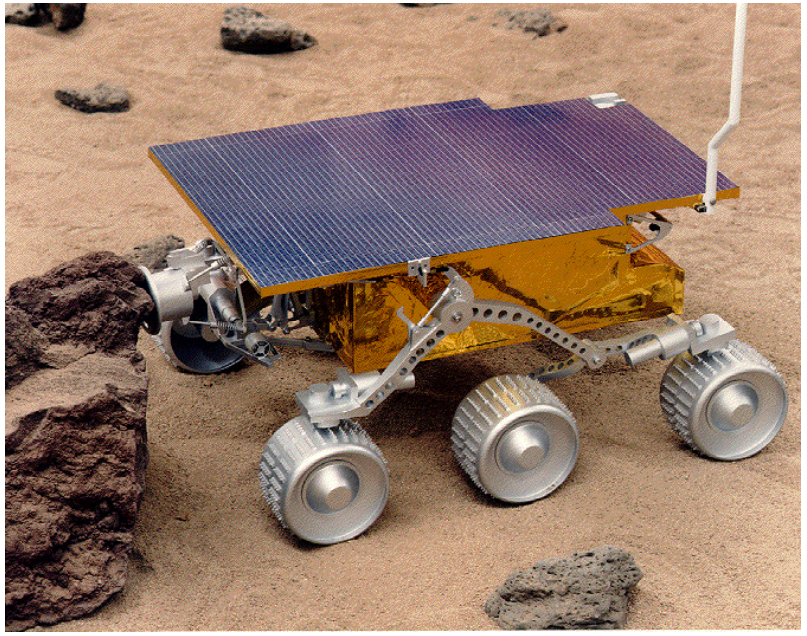
Three threads (among others):

- bus management
  - High priority, high frequency
- Communication thread
  - Medium priority & frequency, ran for a much longer time
- Meteorological thread
  - Atmospheric Structure Instrument/Meteorology Package (ASI/MET)
  - Low priority that ran infrequently.



Mars Pathfinder case

- Within a few days of landing on mars when pathfinder started gathering meteorological data, it began having **system resets**
- JPL engineers had replica on Earth
- After 18 hours of execution with replica the symptom was reproduced



Mars Pathfinder Case - Problem

Problem:

Pathfinder experiences repeated **RESETS** after starting gathering of meteorological data.

Caused by a combination of the following:

1. The meteorological thread required the (shared) bus to publish data. A binary semaphore was used.
2. The meteorological thread acquired it and the bus manager had to wait.
3. The meteorological thread was pre-empted by the communications thread and
4. The bus manager had to wait longer – causing a missed deadline, alarm and system hardware reset.

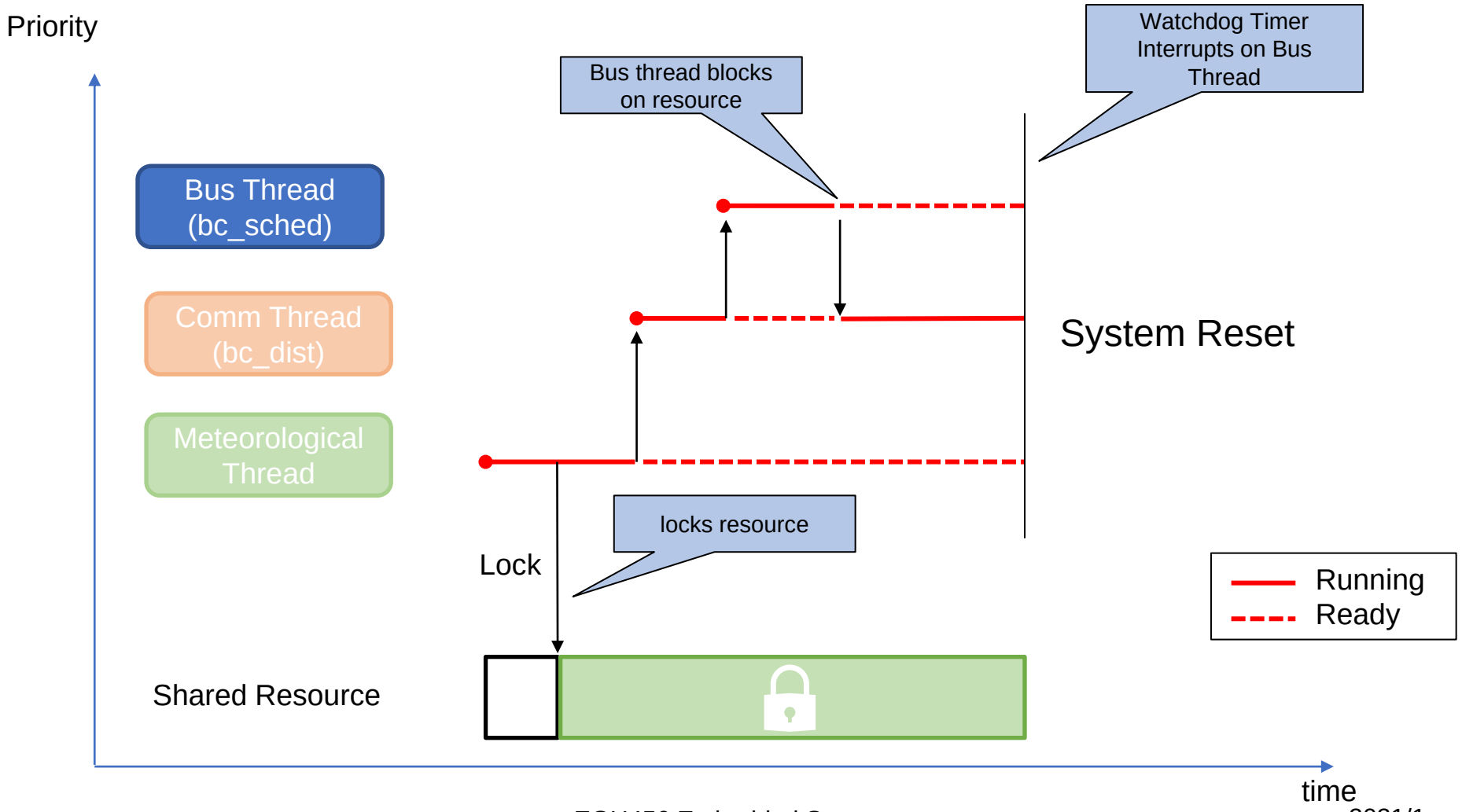
Saved by being Fault tolerant

- Watchdog timer was used to reset the system in the event the RTOS locks up
- Watched for lock ups on the highest priority task

Main issue was Inter-task communication

- Shared resource (memory) was used to pass data from data gatherer (meteorological task) to the communication task via the bus manager task.

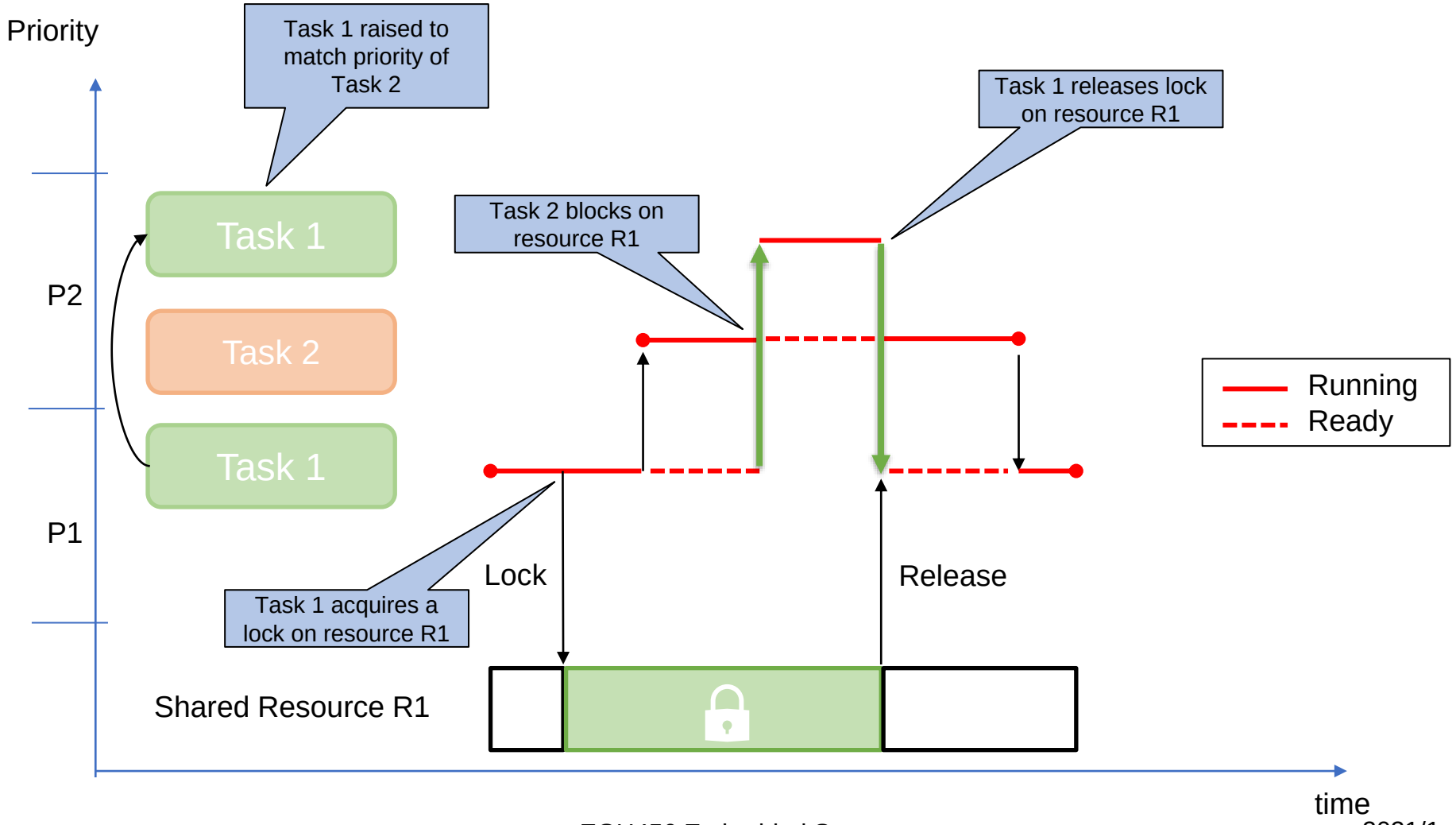
Mars Pathfinder Case



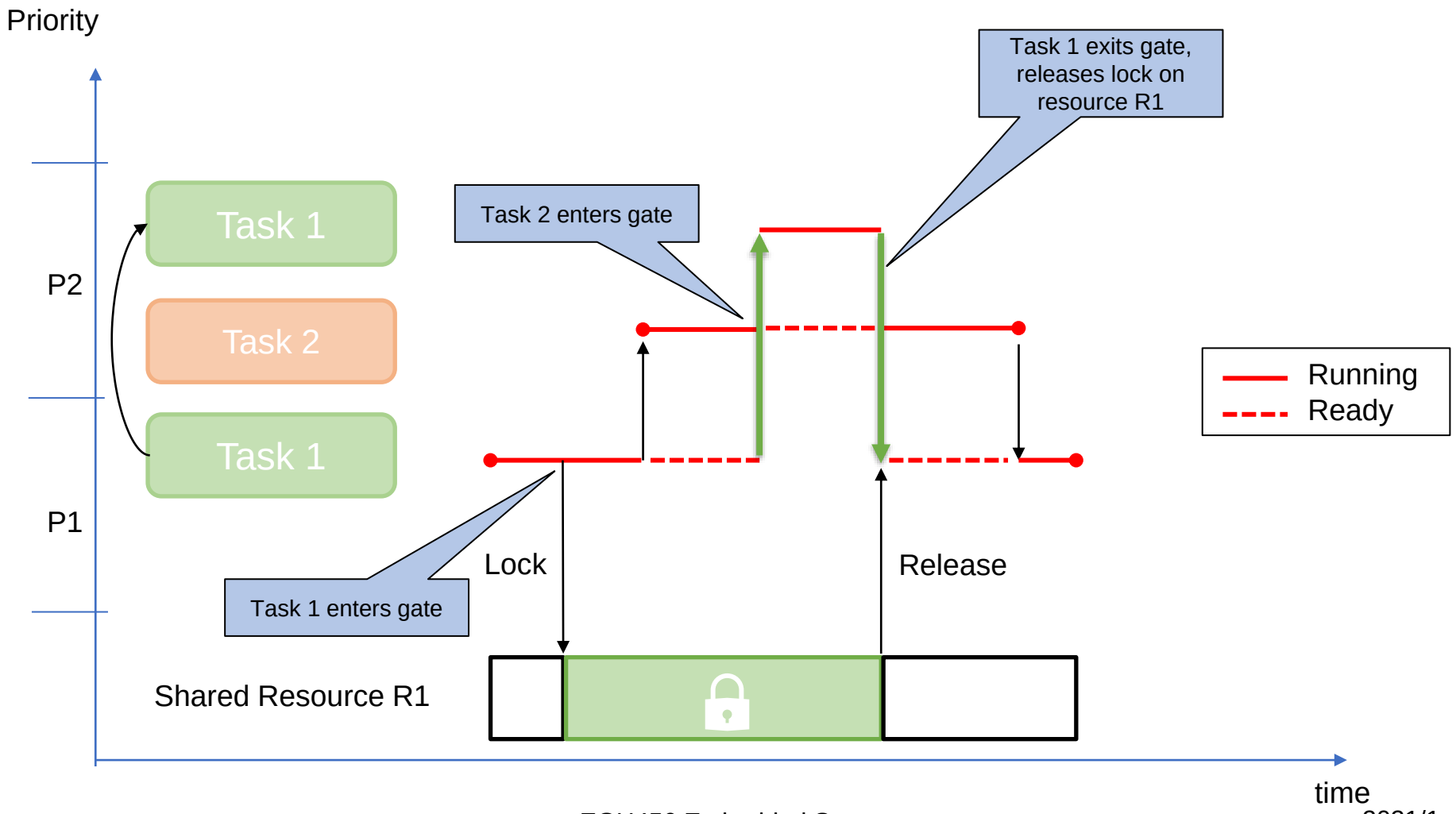
Solutions to Priority Inversion

- **Priority Inheritance Protocol**
  - Priorities of tasks are dynamically changed
  - A task in a critical section inherits the priority of the highest task pending on that critical region.
  - Priority inheritance does not prevent deadlock.
- **Priority Ceiling Protocol**
  - Raise priority of task to predefined ceiling during critical region
  - Stops deadlock
  - Poor response time due to overhead
- **Random Boosting**
  - Ready threads in critical sections priorities randomly boosted (used in Windows)

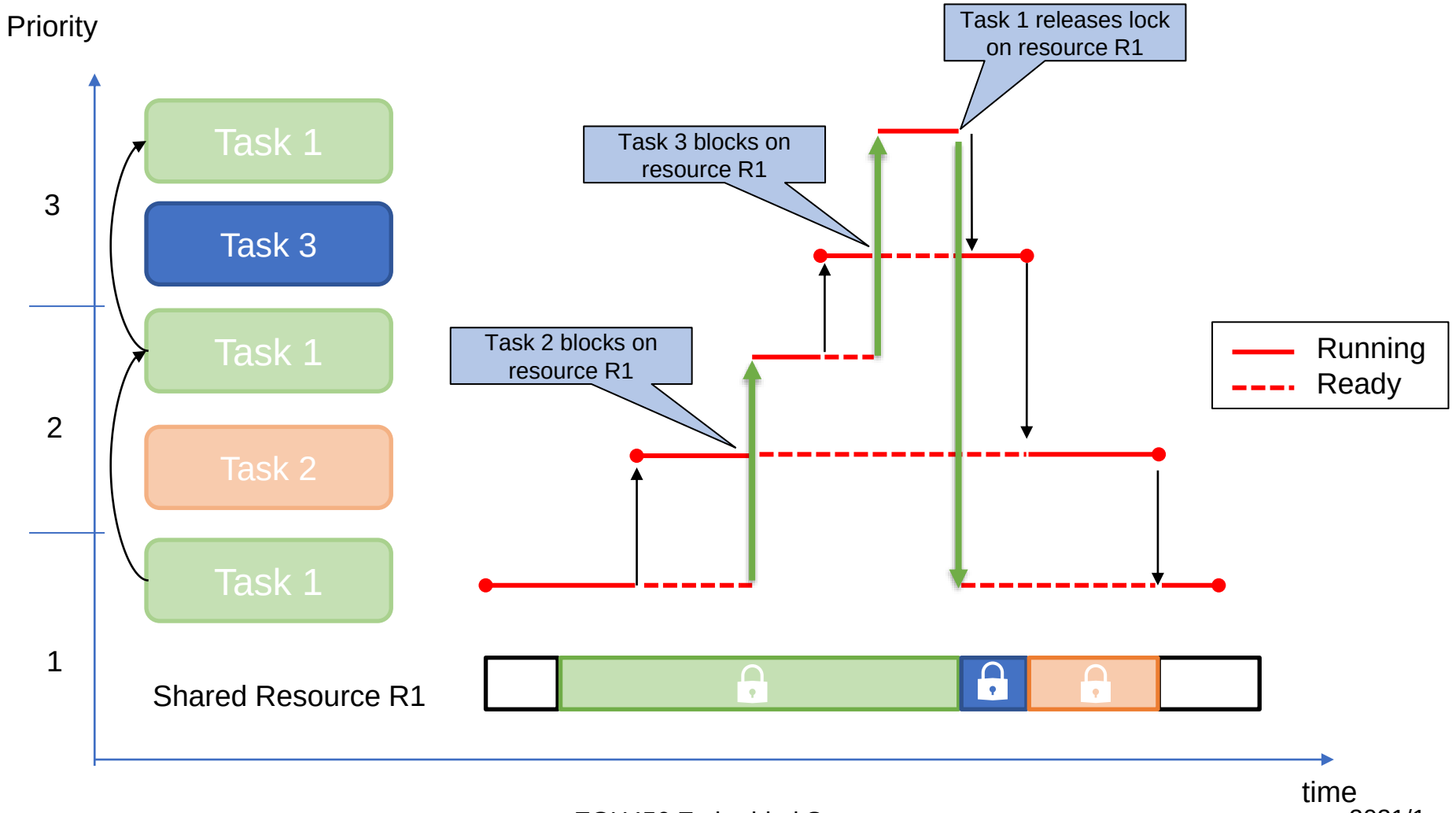
Priority Inheritance Example



Priority Inheritance Example



Priority Inheritance Example 2



Priority Ceiling Protocol

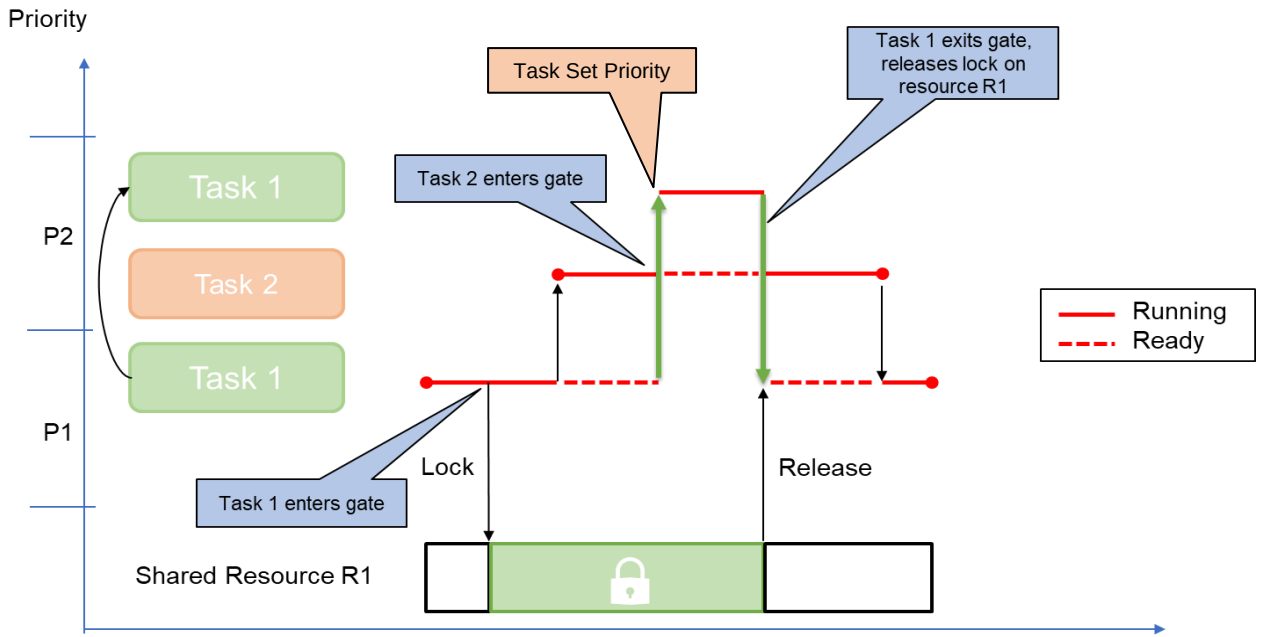
- Avoids unbounded priority inversion and mutual deadlock due to wrong nesting of critical sections.
- Each resource is assigned a priority ceiling
- Priority ceiling = the priority of the highest task that can use this resource.
- This and other methods are outside the scope of this unit

Information found in the reference book by Daniel Lewis. Research papers can be found in ACM and IEEE Transactions.

Priority Mutex Gates

Only if both threads run the **gateMutexPri_enter()** on the same **gateMutexPri** object does task low inherit task high's priority thus avoiding priority inversion.

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



Alternative Simple Solution

If you can then use same priority!