EGH456 Embedded Systems

Lecture 11

Concurrent Access and Contention Problems Continued

Dr Chris Lehnert

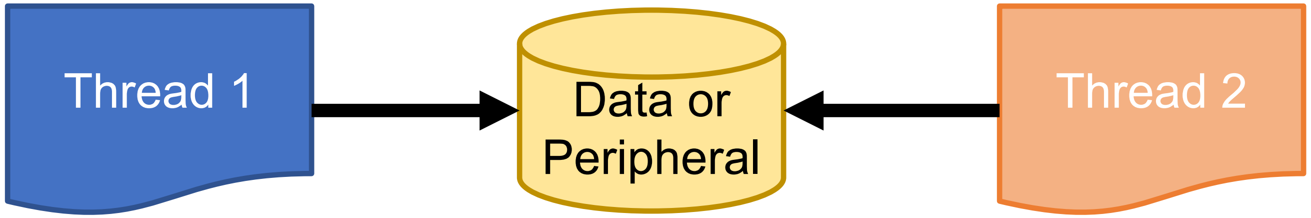
Contents

- Lecture 10 recap
- Deadlock
- Assignment Help

Last Lecture - Concurrent Access Model

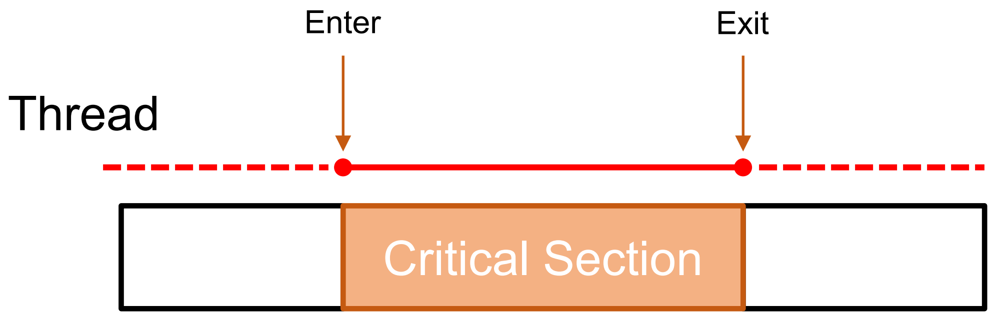
Concurrent Access Model

- 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

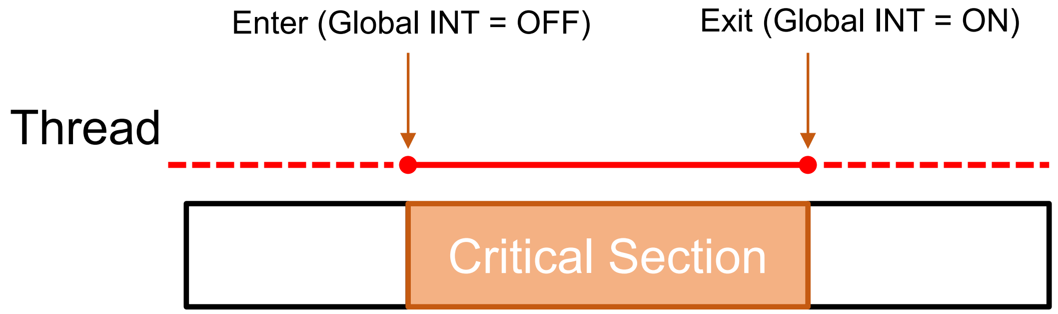


Critical Resource Protection

- 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)



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

Task 1

```
Semaphore_pend(Sem);
…use data…
Semaphore_post(Sem);
```

Task 2

```
Semaphore_pend(Sem);
…use data…
Semaphore_post(Sem);
```

Task 3

```
Semaphore_post(Sem);
```

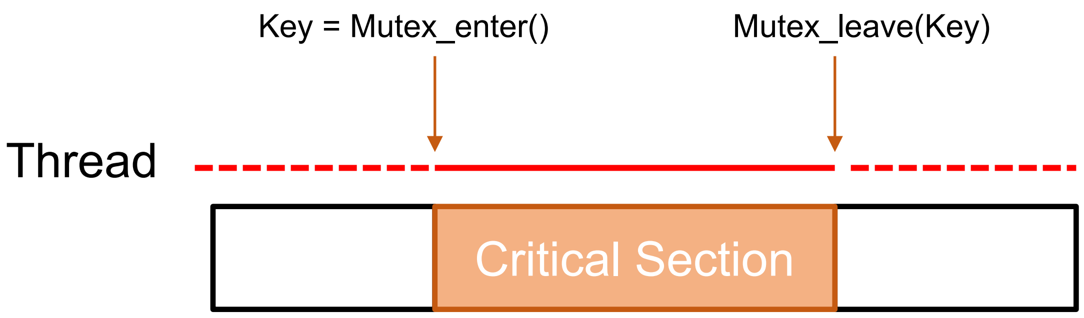
MUTEX Concept

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



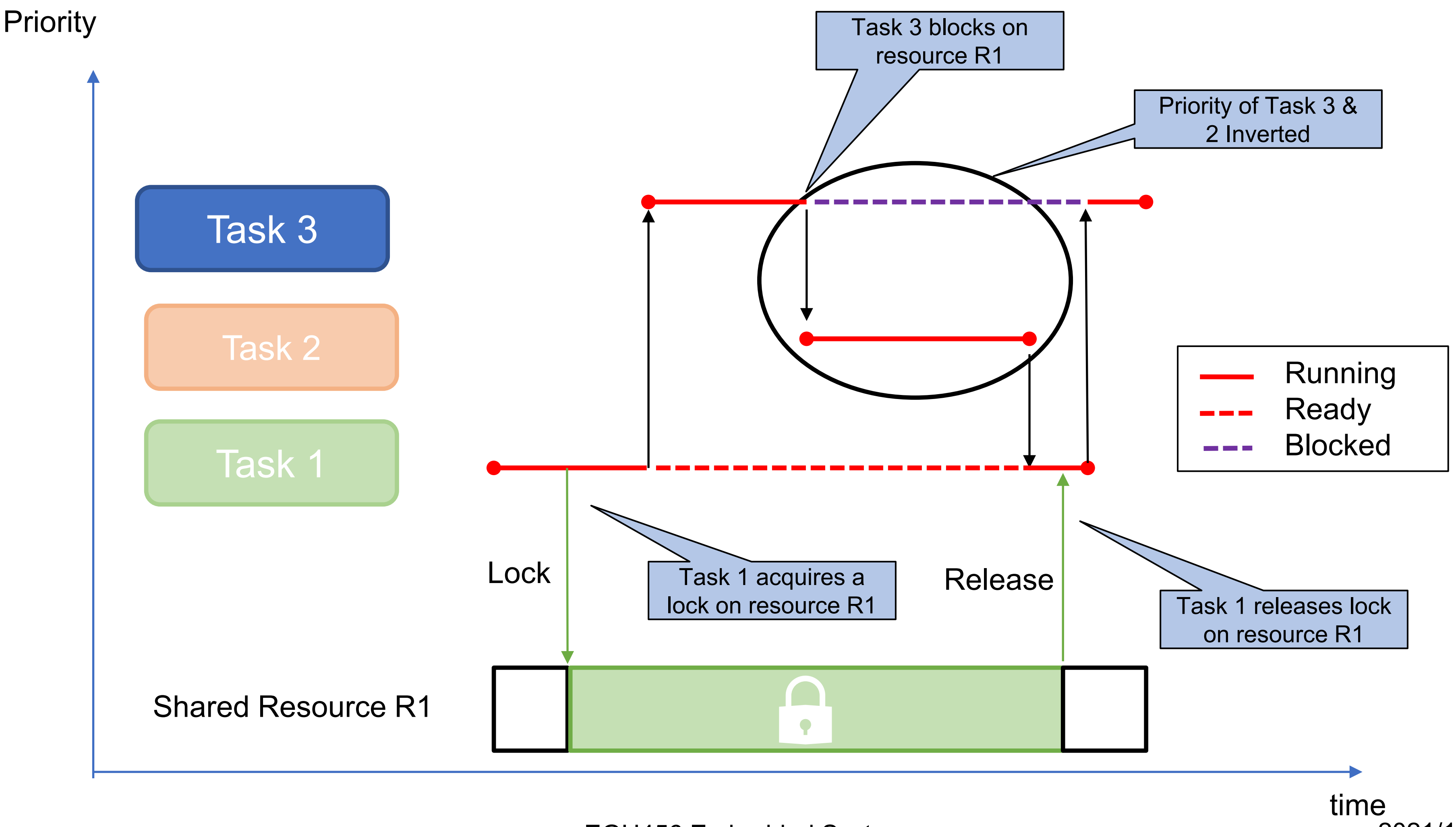
Semaphore vs MUTEX

- 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)

What's a Priority Inversion?



Priority Inversion

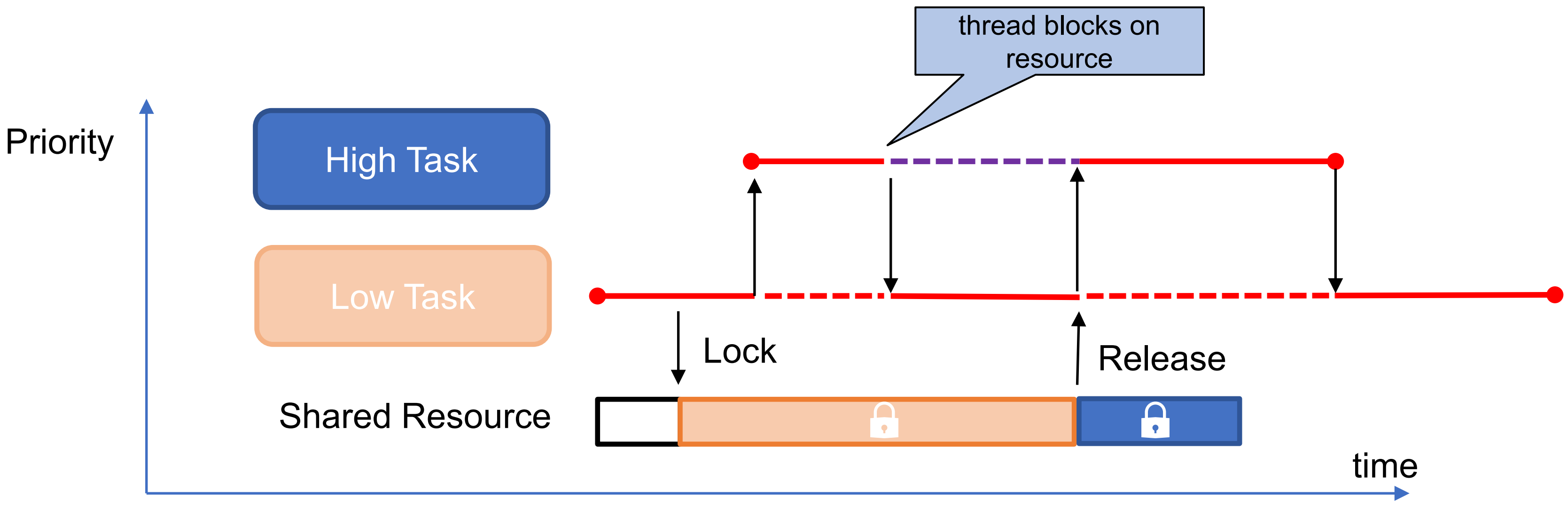
- A priority inversion occurs when a high priority task is indirectly pre-empted by a medium priority task **inverting** the relative priorities of the two tasks

- 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

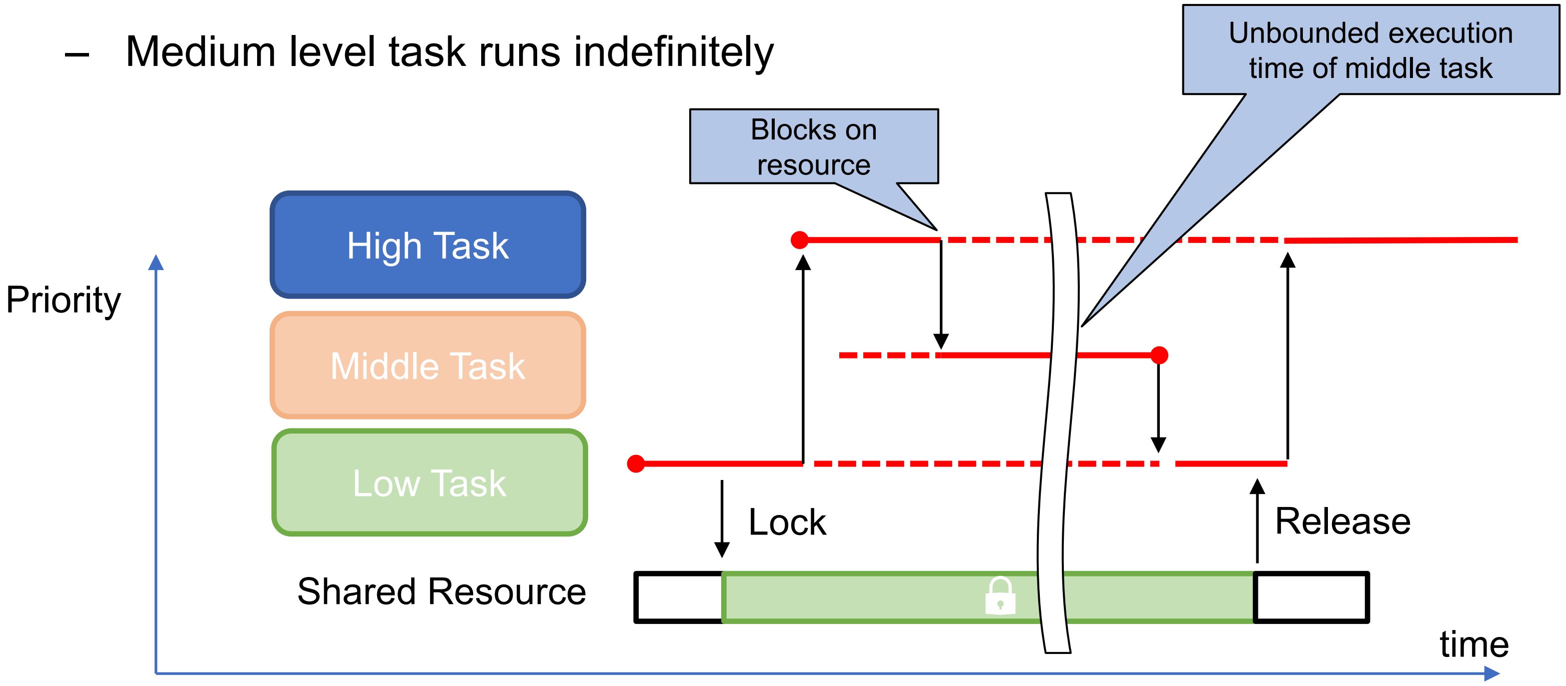
– Low priority task acquires lock but before releasing the resource is pre-empted by higher priority task. Higher task is forced to wait for resource to be released

– lasts a short period of time (time for lower task to finish with resource)



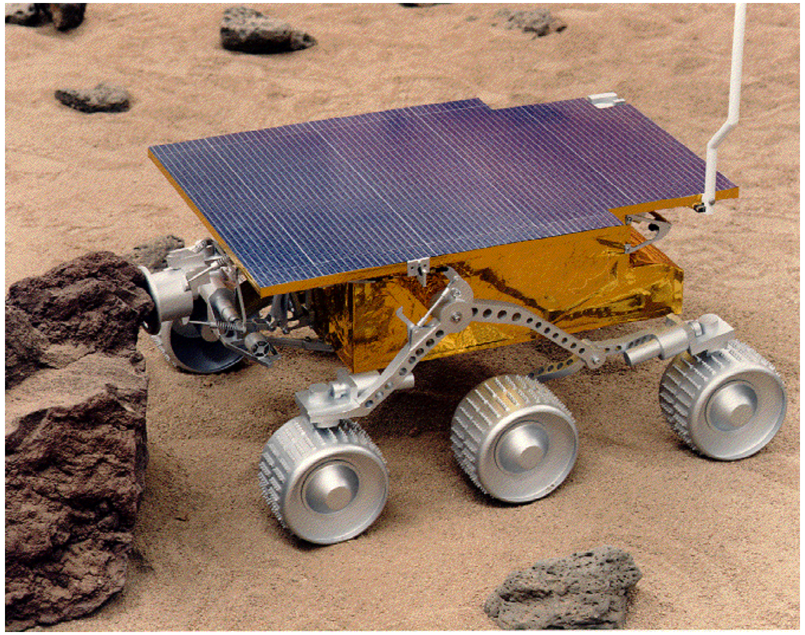
Unbounded Priority Inversion

- 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

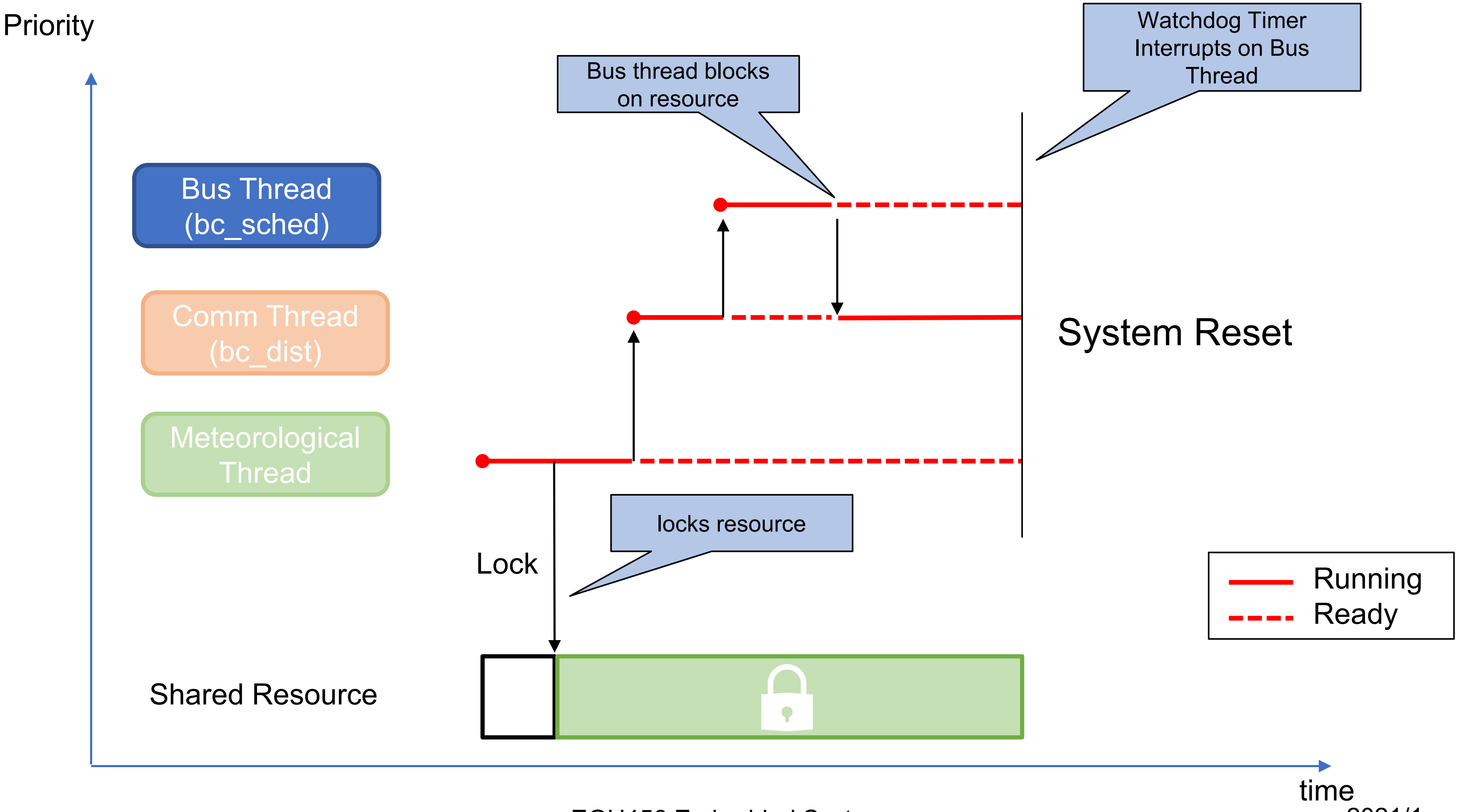


Mars Pathfinder case

- 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



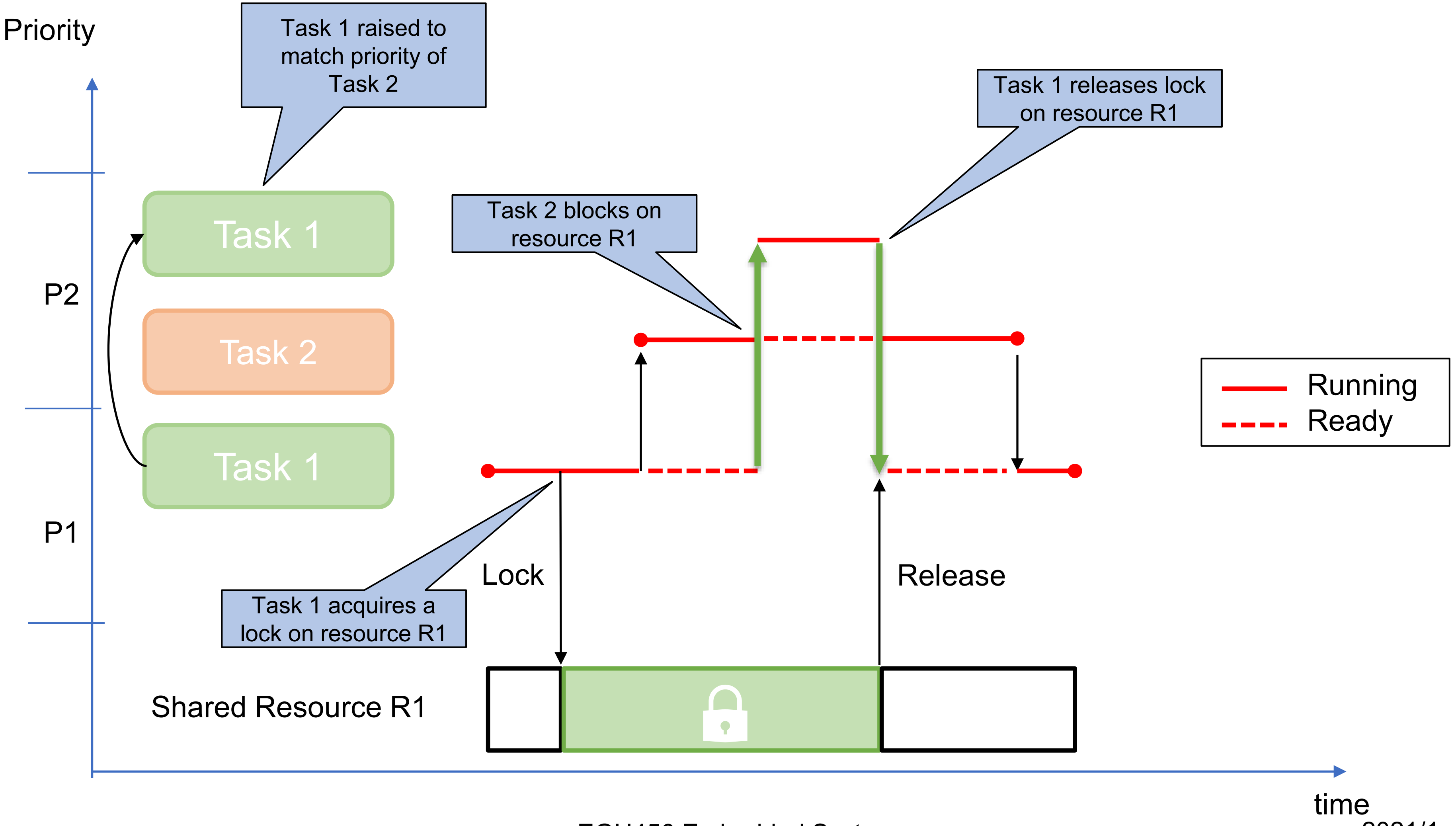
Mars Pathfinder Case



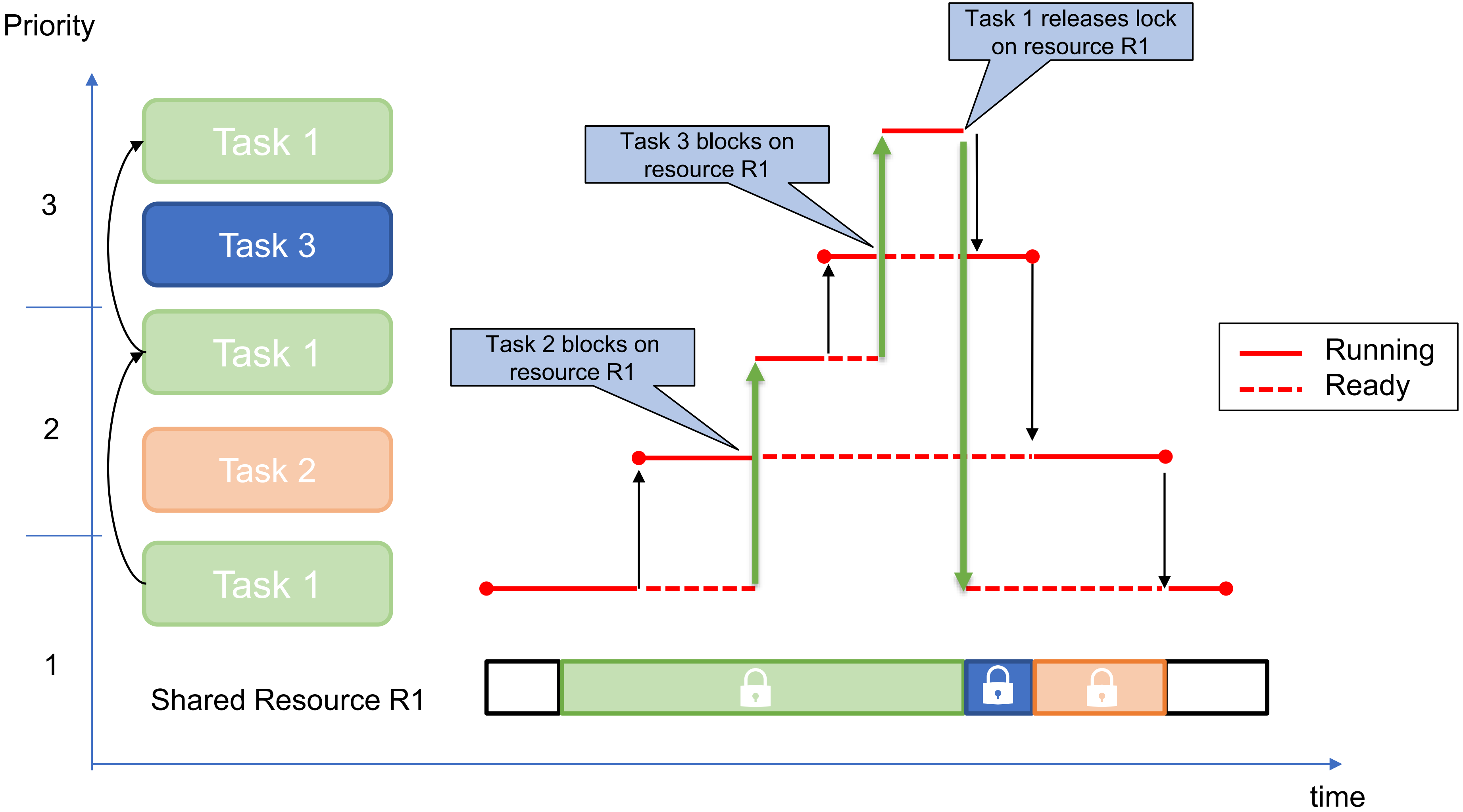
Solutions to Priority Inversion

- **Priority Inheritance Protocol**
  - Priorities of tasks are <u>dynamically changed</u>
  - A task in a critical section inherits the priority of the <u>highest task pending</u> 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



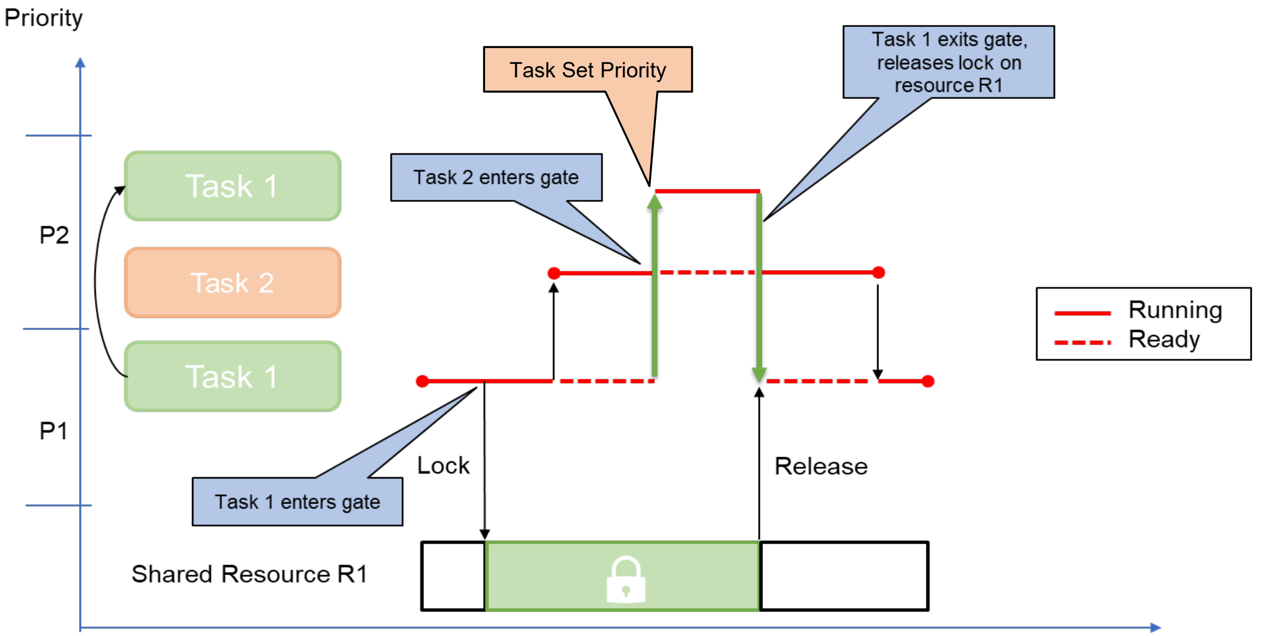
Priority Inheritance Example 2



Priority Mutex Gates

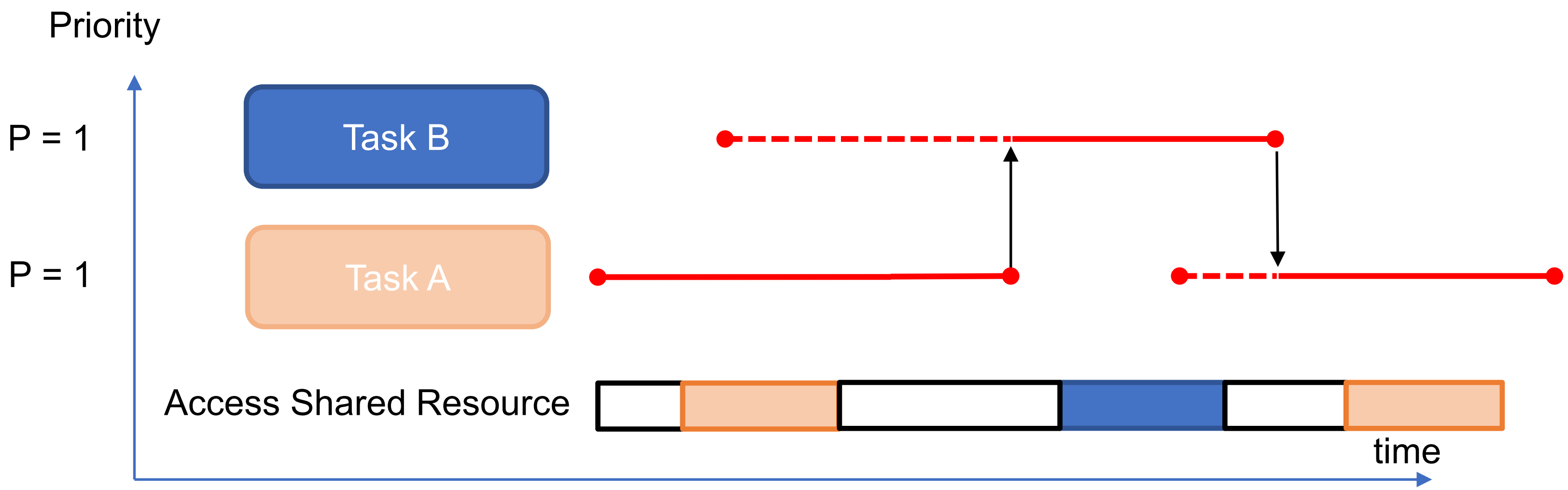
Only if both threads run the **gateMutexPri_enter()** on the same **gateMutexPri** object does task low inherit task high's priority thus avoiding priority inversion.

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



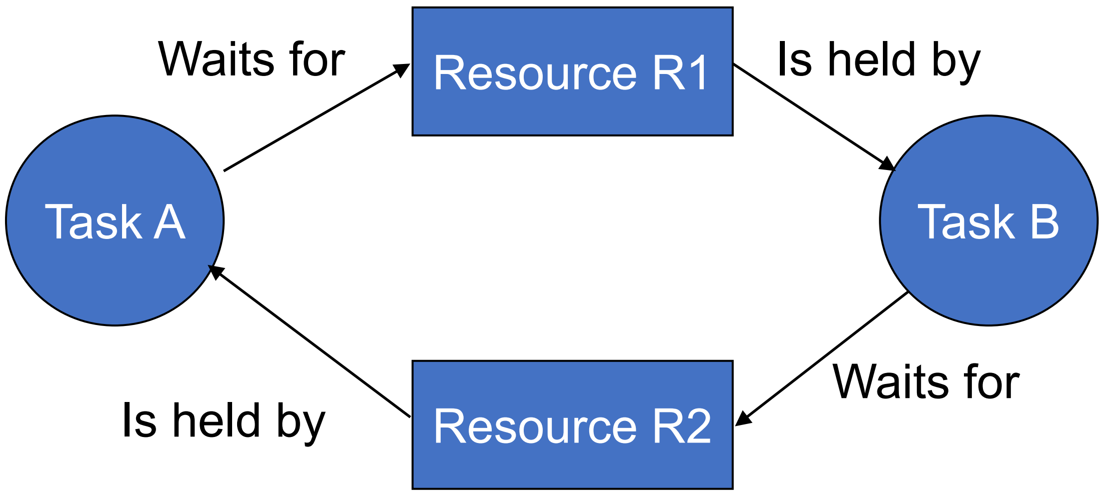
Alternative Simple Solution

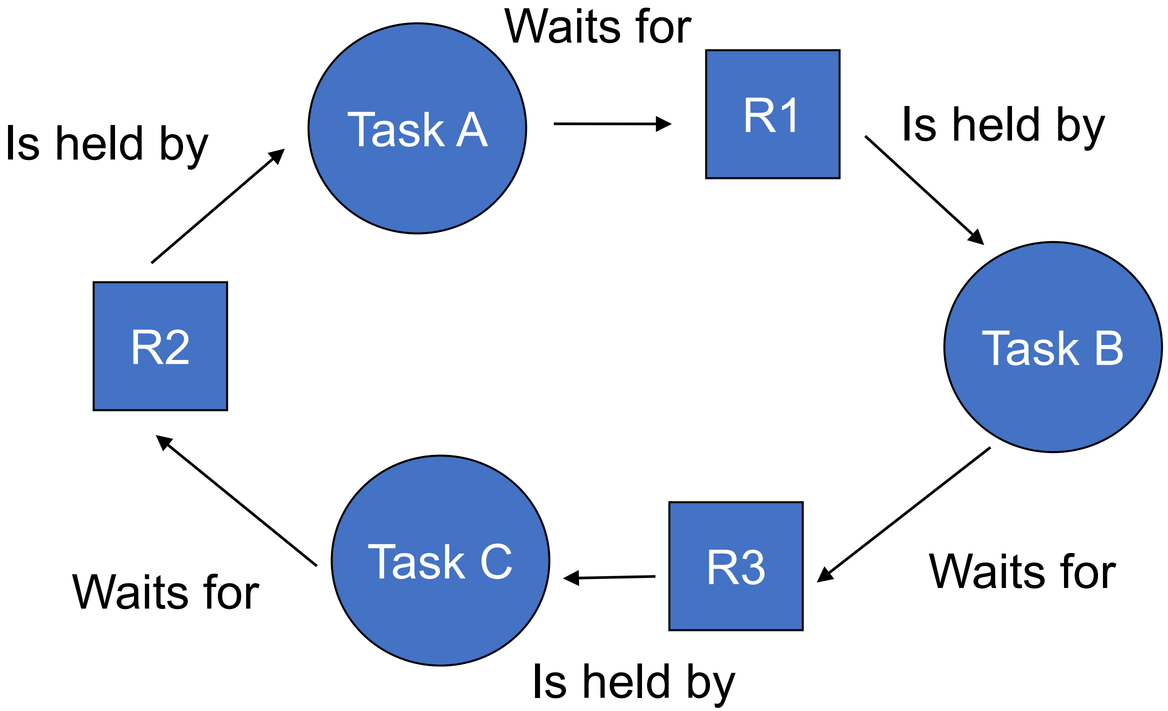
If you can then use same priority!



Deadlock

- Chains of blocking may be formed and the blocking duration can become substantial or infinite.
  - A waits for B, B waits for A, etc.





Other Problems - Deadlock

- Occurs when two threads block each other
  - Use of MUTEX with multiple resources (with circular pending)
  - Threads at different priorities

Task A

```
Sem_pend(res_1);
//  use resource1
     STUCK ?
Sem_pend(res_2);
//  use resource2
Sem_post(res_1);
Sem_post(res_2);
```

Task B

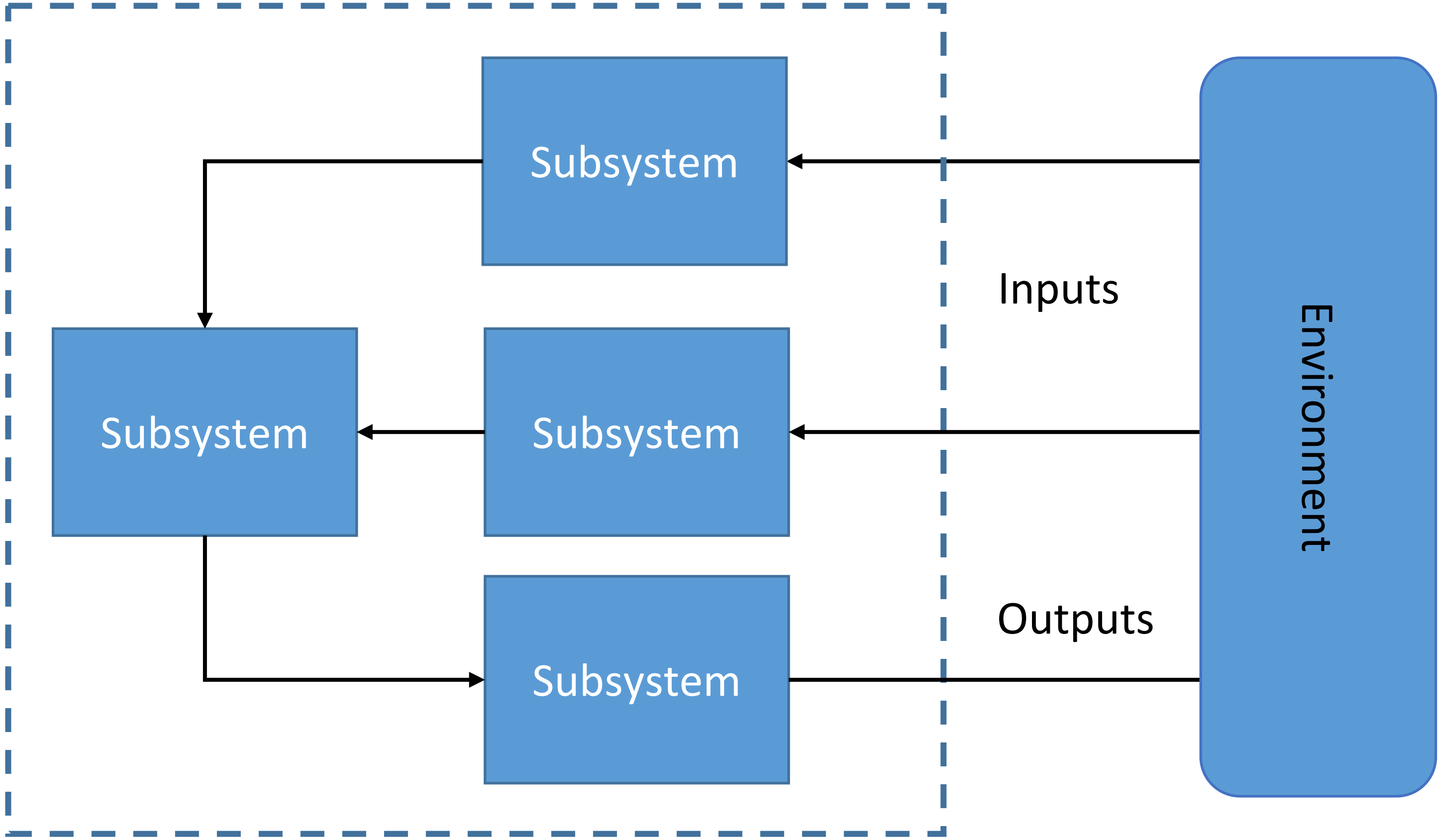
```
Sem_pend(res_2);
//  use resource2
     STUCK ?
Sem_pend(res_1);
//  use resource1
Sem_post(res_2);
Sem_post(res_1);
```

Solutions:

- Use timeouts on _pend
- Use same ordering in both threads – 1, 2, 3
- Lock one resource at a time, or ALL of them

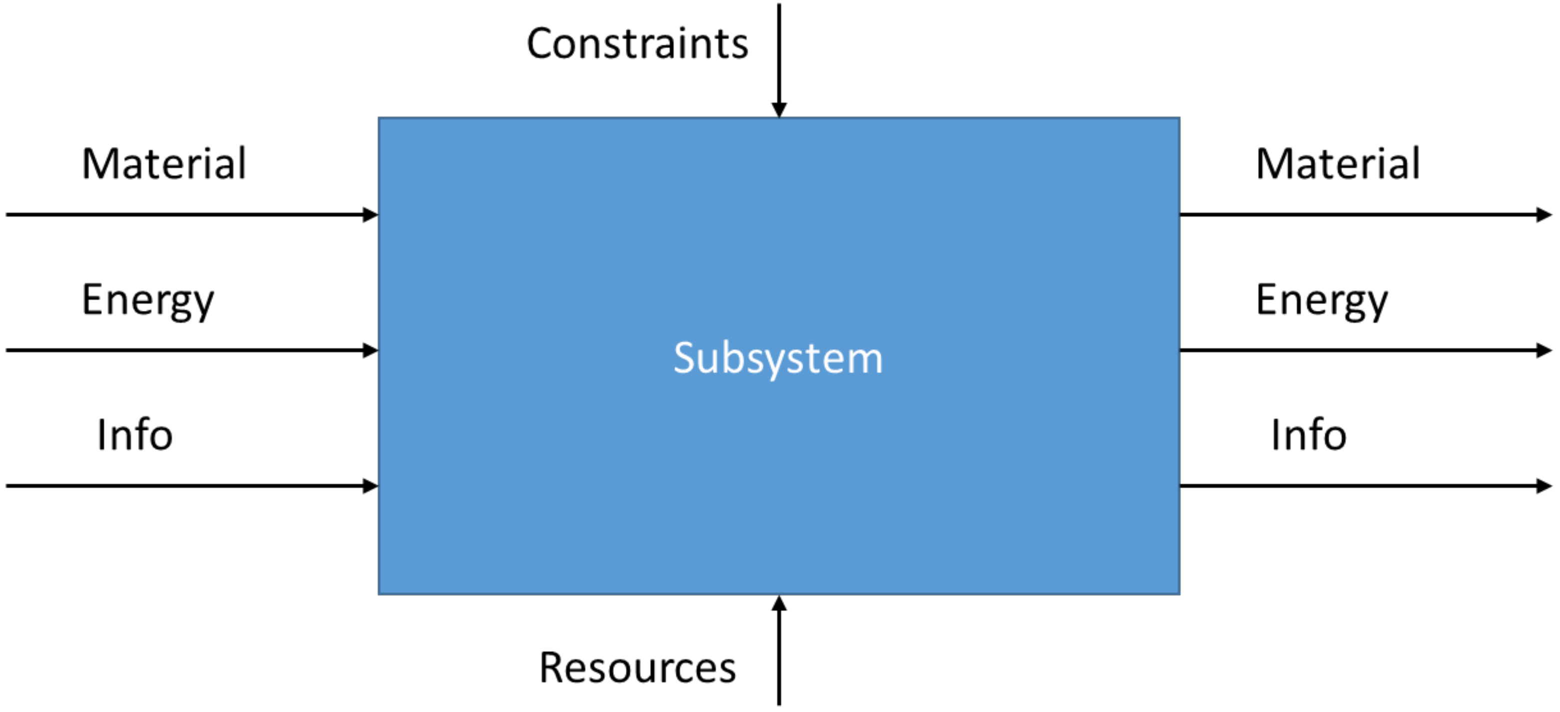
Neither Task will be awakened

System Architecture

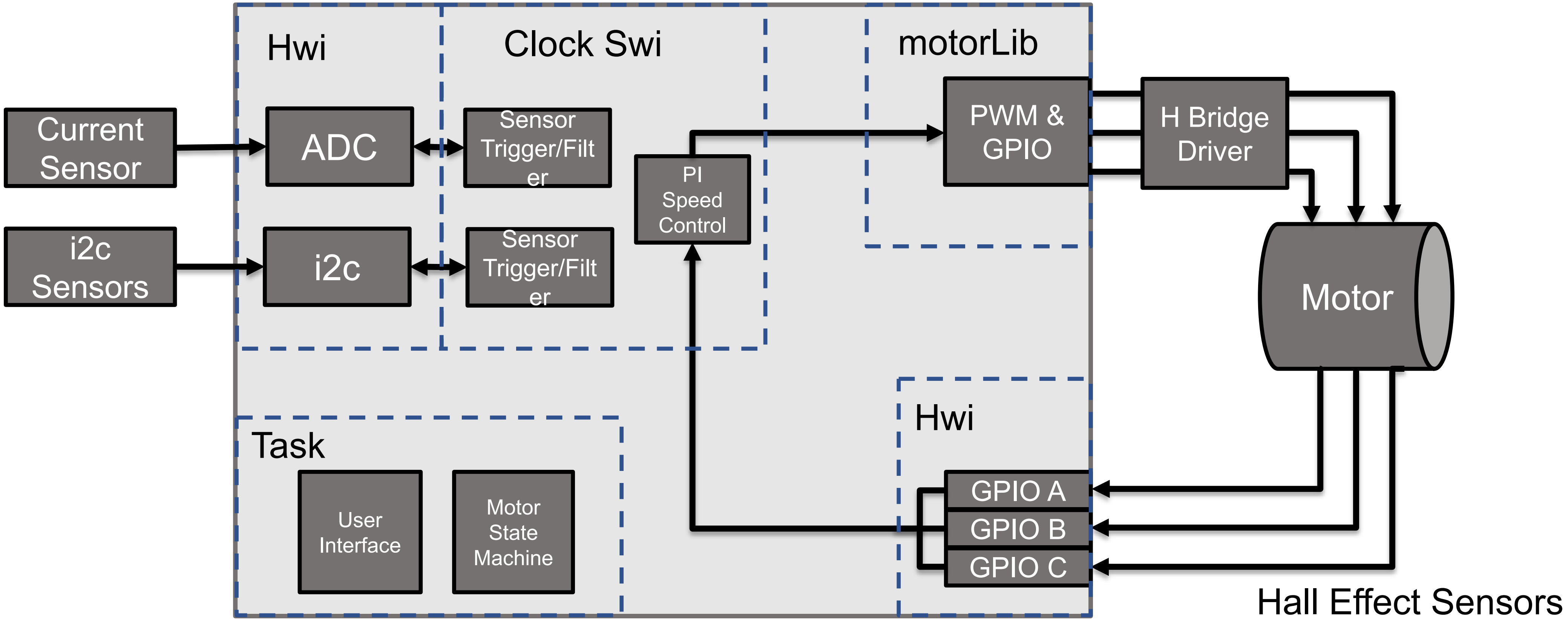


Functional Analysis

- Black box functions
  - Define sub systems as functions with **inputs** and **outputs**, given constraints and resources.

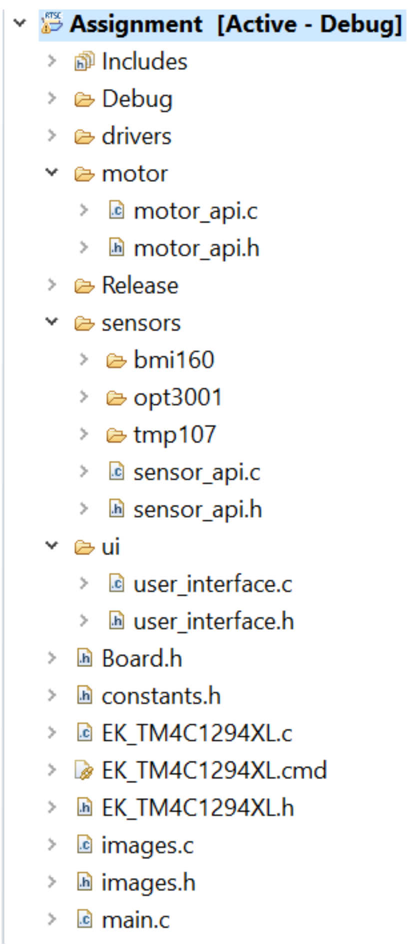


Assignment Structure Example



Project Structure

- Recommended to structure solution into different drivers
  - Use folders to clean up project
  - Use **comments**!
  - Sensor, Motor and User Interface folders
  - Think about re-entrant or function behaviours when sharing your code to the team
  - Plan and structure drivers using an api



Sensors.h

- `initSensors()`
- `getLight()`
- `getBoardTemp()`
- `getCurrent()`

Motors.h

- `initMotor()`
- `getSpeed()`
- `startMotor()`
- `setSpeed()`

UserInterface.h

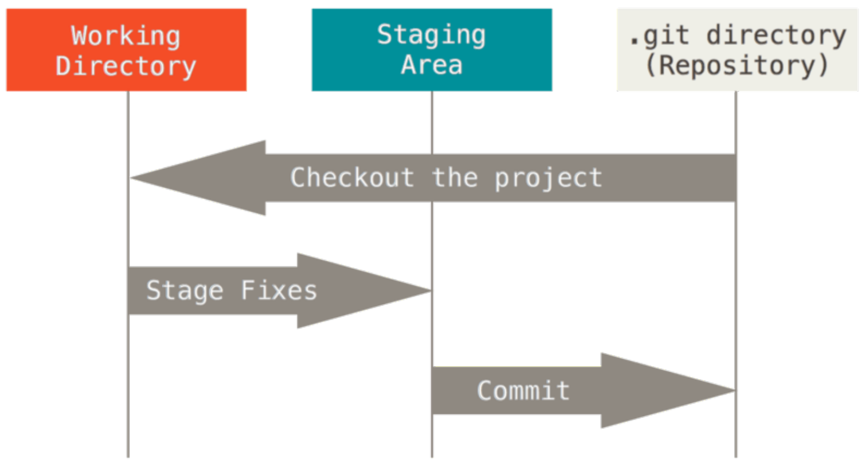
- `initUserInterface()`
- `DrawMenuScreen()`
- `drawGraph()`
- `setSpeed()`

Version Control

- Advantages of using git or svn repositories
  - Keeps track of code history
    - Easily visualise code changes
  - Develop in parallel using branches
    - Good for teams working on single software project
  - **Quickly and easily revert back to working code!**
  - Bitbucket vs Github
    - Private vs public
  - Try Git Kraken

  https://www.gitkraken.com/

Git in a nutshell




```
$ git clone
https://github.com/libgit2/libgit2

$ git pull

$ git status
        On branch master Your branch is up-to-
date with 'origin/master'.
        Changes not staged for commit:
        modified: changed_file.cpp

$ git add changed_file.cpp

$ cat .gitignore *.[oa] *~

$ git commit -m 'my first commit of
file'

$ git push
```