SJSU SAN JOSÉ STATE UNIVERSITY

Charles W. Davidson College of Engineering

Department of Computer Engineering

Real-Time Embedded System Co-Design

CMPE 146 Section 1

Fall 2024

Task Synchronization

Motivation

- Not all tasks run entirely independently
- Some tasks may need to use common resources with other tasks
  - Some protection mechanism is needed for resolving conflicts that happen in sharing common resources
    - To preserve data integrity: data object must truly represent what is intended for
    - To use devices or I/O exclusively
- Co-operating tasks can use synchronization mechanism to
  - Ensure correct program execution orders
  - Regulate data flow
- OS can provide services to help achieve those objectives
  - Programmers do not need to worry about the details of mechanism
- Common system objects for protection and synchronization in tasks
  - Semaphore
  - Mutex

Semaphore

- A system object managed by the OS for controlling access to resources
  - A task must acquire the semaphore first before accessing
- A semaphore, when created, can be specified with an initial value denoted by *count*
- At any time, a semaphore is available only if *count* > 0
  - It is unavailable when *count* ≤ 0
- Depending on the application, one can impose a constraint on the maximum value that *count* may take
  - Such maximum value is referred to as capacity
- A semaphore has an associated task waiting list, containing tasks that are blocked because of the unavailability of the semaphore
- There are two main operations associated with semaphores
  - POSIX uses the terms "*wait*" and "*post*"

Semaphore (cont'd)

- A task executes the *wait* operation on a semaphore to access resource
  - If the semaphore is currently available (i.e., *count* > 0), the semaphore value is decreased by 1, and the task is granted access
  - If the semaphore is currently unavailable (i.e., count ≤ 0), the task is added to the task waiting list
  - In POSIX, it is the *sem_wait()* function
- A task executes the *post* operation on a semaphore to release access
  - If there are no tasks in the waiting list, this operation simply increases the semaphore value by 1 (up to the capacity count)
  - If there are tasks in the waiting list, one task waiting for the semaphore is unblocked and will return from its *sem_wait* call
    - Note that in this case, we have *count* = 0 both before and after the *post* operation
  - In POSIX, it is the *sem_post()* function
- Generally, there are three types of semaphore
  - Binary
  - Counting
  - Mutually Exclusion (Mutex)

Binary Semaphore

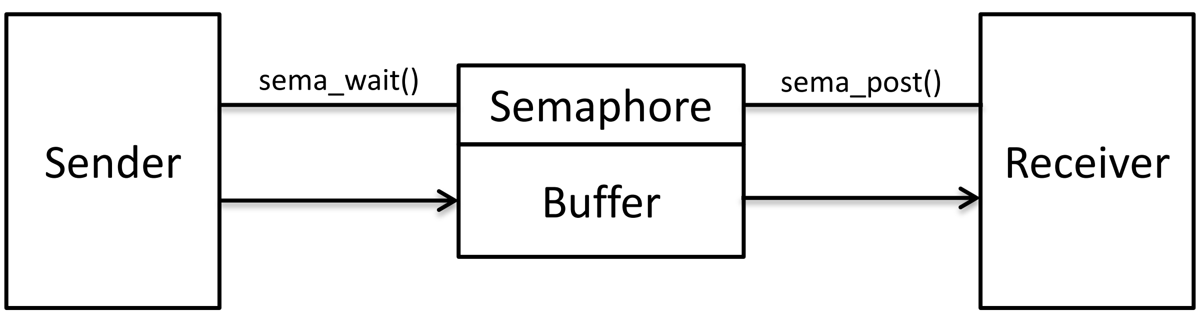
- We can use binary semaphores to synchronize two tasks
  - A binary semaphore's *count* can only be 0 or 1
- Two tasks need to know if each other has reached certain point before moving forward (a need to synchronize)
- Two semaphores are used: *sem1* and *sem2*
  - Both are initialized to 0 upon creation
  - Task1 needs to acquire *sem1* before proceeding
  - Task2 needs to acquire *sem2* before proceeding
- When a task reaches the synchronization point
  - It posts (increases the count of) the other semaphore
  - Then waits for its "own" semaphore

```
Task1()
{
        ⋮
     sem_post(sem2);
     sem_wait(sem1);
        ⋮
}
```

```
Task2()
{
        ⋮
     sem_post(sem1);
     sem_wait(sem2);
        ⋮
}
```

Counting Semaphore

- We can use a counting semaphore to regulate data flow from one task to another
  - A counting semaphore's *count* indicates the amount of resource available
- Two tasks: sender and receiver
  - Both share a buffer that can hold up to N data packets
  - Sender sends data packets to buffer; receiver retrieves data packets from buffer
- A counting semaphore's value indicates how many empty slots available in the buffer
  - Initially, the semaphore count is N
  - Sender waits for semaphore before sending data packet
  - Receiver posts semaphore after retrieving data packet



Mutex

- A mutex is a special type of binary semaphore to control access to a resource that is shared between two or more tasks
- Mutex has ownership whereas semaphore does not have that
- Characteristics of mutex
  - A task gains the ownership of a mutex when it first locks the mutex
  - A mutex can be unlocked (released) only by its owner task
  - If configured so, a mutex can be locked by the owner thread repeatedly
    - Normally, the same task attempting to lock again will result a deadlock
- POSIX functions
  - *pthread_mutex_lock()* locks a mutex object
    - Task (Thread) acquires ownership of lock
  - *pthread_mutex_unlock()* releases a mutex object (by its owner)
    - Task (Thread) relinquishes ownership of lock

Mutex Usage

A typical sequence in the use of a mutex

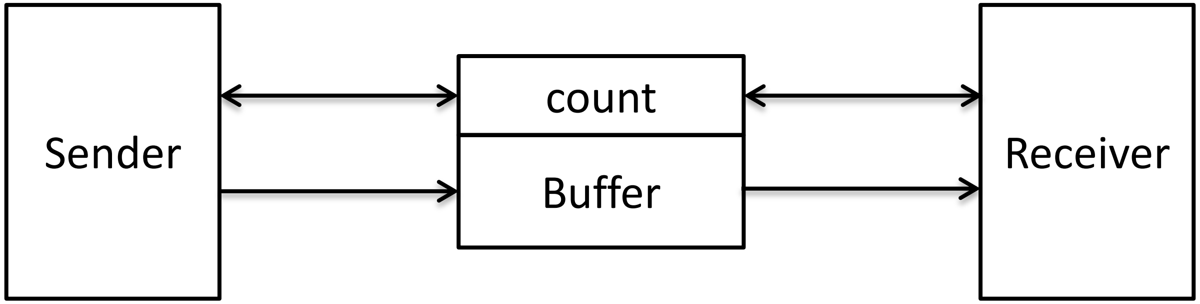
1. Create and initialize a mutex
2. Several tasks attempt to lock the mutex
3. Only one succeeds and now owns the mutex
4. The owner performs a set of actions
5. The owner unlocks the mutex
6. Another task acquires the mutex and repeats Steps 4-5
7. Finally, the mutex is returned to the OS when it is no longer needed

Mutex vs. Semaphore

- Mutex has ownership whereas semaphore does not
  - A mutex can be unlocked only by its owner task
  - Any task can call sem_post() to “unlock” a semaphore
- Mutex is a locking mechanism whereas semaphore is a signaling mechanism
  - Mutex allows exclusive access to some shared resource by its owner (mutual exclusion)
  - Semaphore indicates the availability of some shared resource
- Once a task owns a mutex, it can repeatedly call pthread_mutex_lock() in different part of the task without being blocked
  - Newer calls just return immediately
  - With semaphore, repeated calls to sem_wait() may suspend the task

Mutex Example

- Same sender-receiver example as before
- Use a variable *count* to indicate the number of data packets in the buffer



- Sender increments *count* upon adding a data packet to buffer
- Receiver decrements *count* upon removing a data packet from buffer

```
Sender()
{
        ⋮
    count++;
        ⋮
}
```

```
Receiver()
{
        ⋮
    count--;
        ⋮
}
```

Mutex Example (cont'd-1)

- ***count++*** and ***count--*** are (mostly) not atomic actions
- Action consists of three separate instructions:
  - Processor reads *count* from memory to register
  - Processor increments/decrements register
  - Processor writes register back to memory

```
Sender()                Receiver()
{                       {
        ⋮                       ⋮
    count++;                count--;
        ⋮                       ⋮
}                       }
```

- A race condition exists
  - Both *Sender* and *Receiver* may try to update *count* at the same time or very close to each other in time
  - The three-instruction sequence may be disrupted, thus producing incorrect *count* value
- One scenario:
  - *count* = 10
  - Receiver reads 10 from *count* into processor register
  - Receiver task is preempted (after only finished first part of the sequence)
  - Sender task reads 10 from *count* and writes 11 (*count++*) back to *count*
  - Receiver task resumes and eventually writes 9 (*count*--) back to *count*
  - *count* should be equal to 10 after those two task actions

Mutex Example (cont'd-2)

- Solution: Use a mutex to protect the data integrity of *count*
  - Task must acquire a lock (mutex) first before update can be done

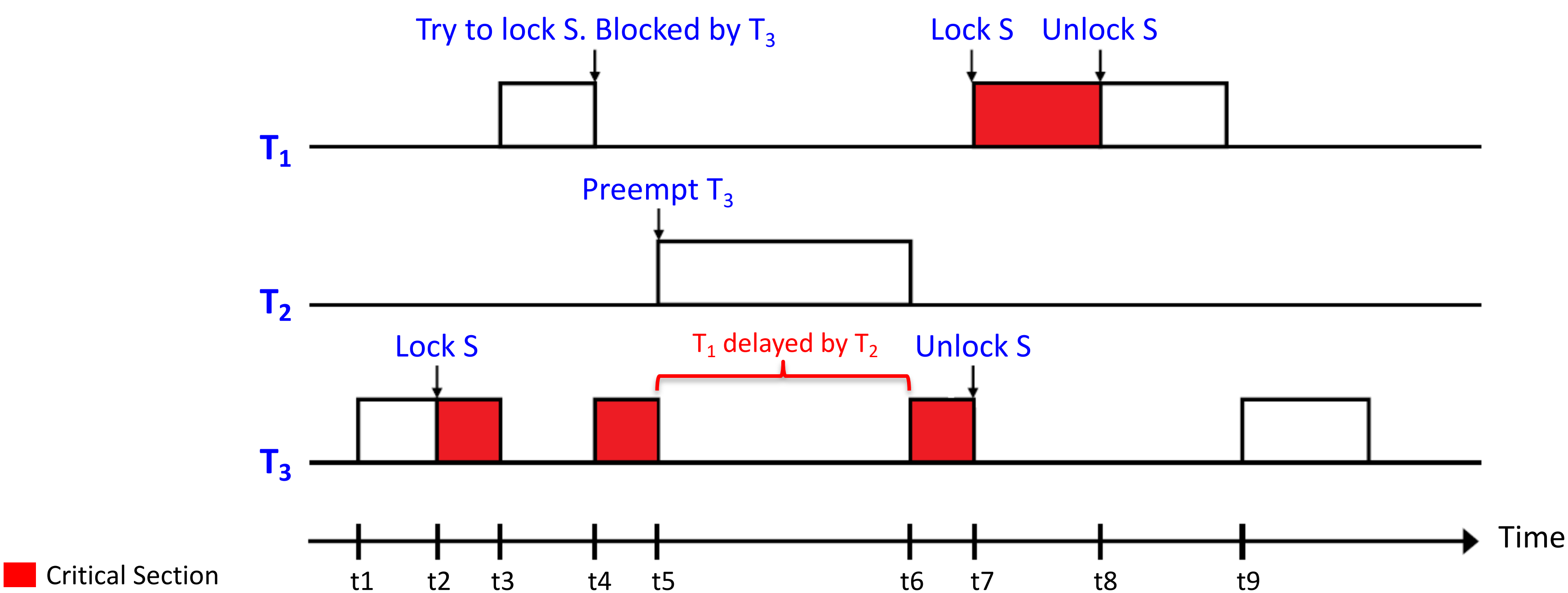
```
Sender()
{
      ⋮
    pthread_mutex_lock(mutex);
    count++;
    pthread_mutex_unlock(mutex);
      ⋮
}
```

```
Receiver()
{
      ⋮
    pthread_mutex_lock(mutex);
    count--;
    pthread_mutex_unlock(mutex);
      ⋮
}
```

- Region of instructions between the lock and unlock calls is referred to as critical section
- ***count++*** and ***count--*** are now atomic actions as far as the tasks are concerned
  - The three-instruction sequence can no longer be disrupted

Priority Inversion Problem

- Basic synchronization techniques may produce an unwanted side effect
  - Higher-priority tasks can be blocked by lower-priority tasks
- Example
  - Three tasks: T_1, T_2, T_3
  - Priorities: $T_1 > T_2 > T_3$
  - T_3 locks (call *sema_wait()*) the semaphore S (with initial count of 1)
  - T_1 preempts T_3 and then attempts to lock S, but is blocked
  - T_2 preempts T_3, delaying T_1

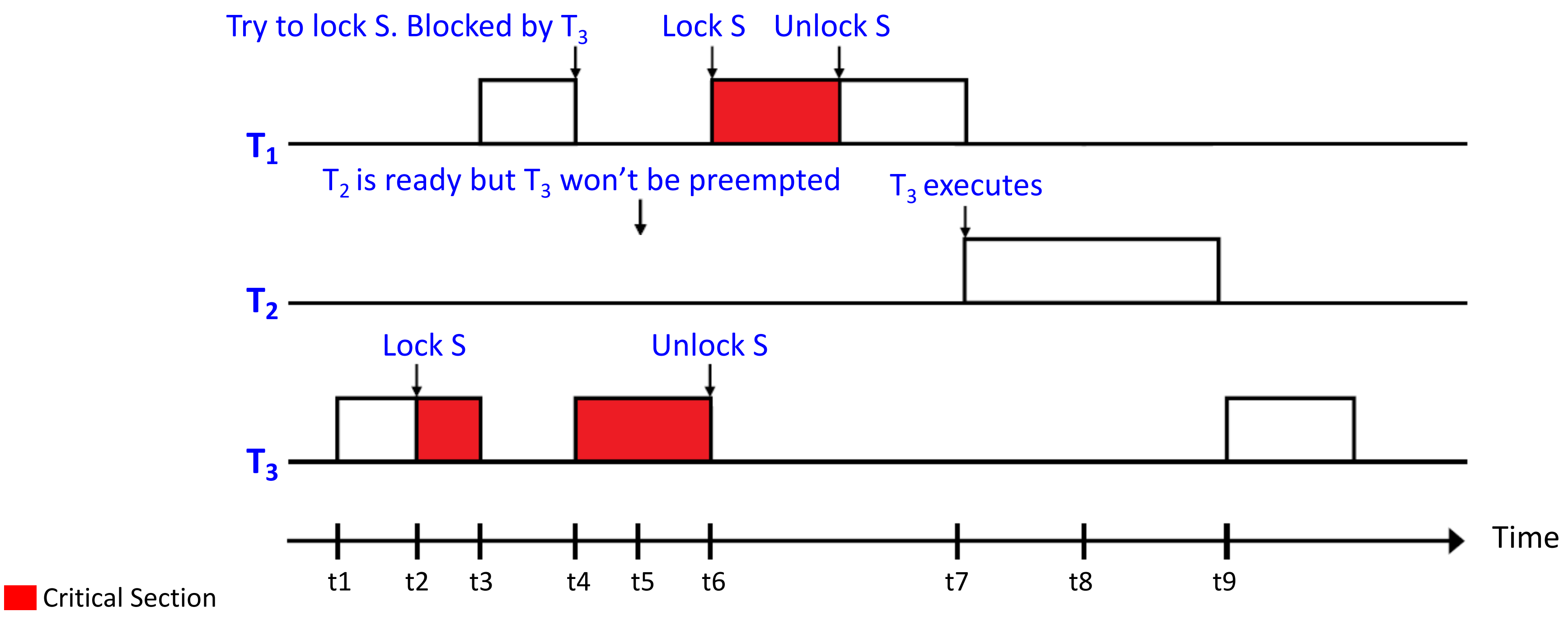


Priority Inheritance Protocol

- To solve the priority inversion problem, we could disallow preemption in a critical section
  - A simple but not good solution
  - Higher-priority tasks may suffer unnecessary blockings
- Priority Inheritance Protocol is a better solution
  - When a task blocks one or more higher priority tasks, its original priority can be raised
  - The task executes its critical section at the highest priority level of all the tasks it is blocking
- Priority inheritance is transitive
  - Only applies to critical sections

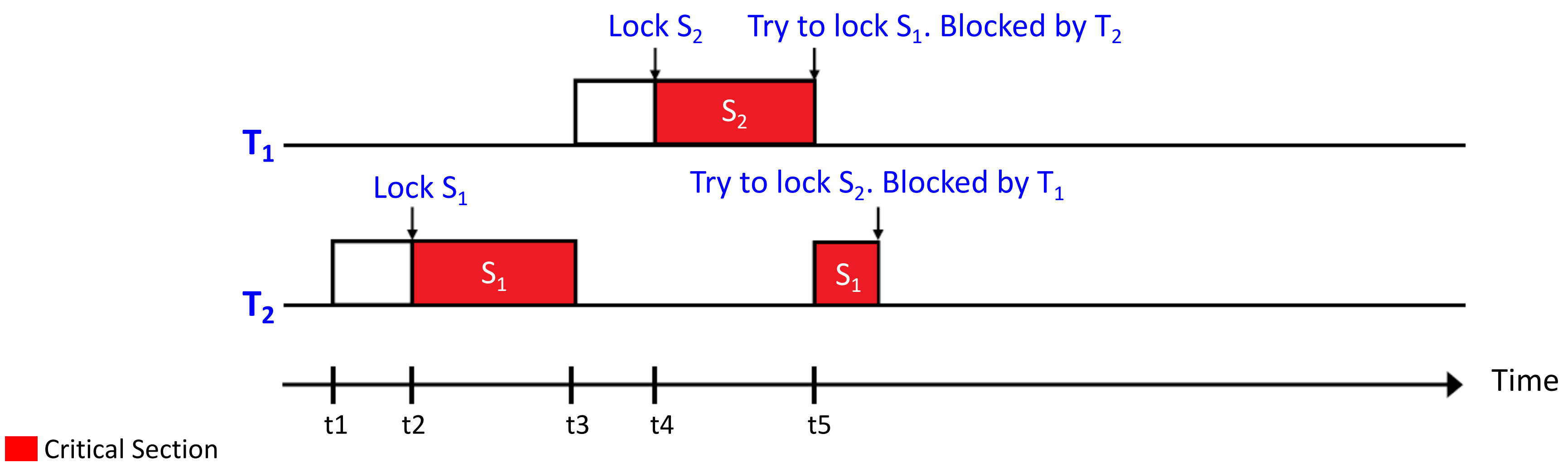
Priority Inheritance Protocol (cont'd)

- Use the same example as before
- T_3 inherits T_1's priority from t4 to t6 temporarily
- T_2 cannot preempt T_3 at t5
- Lower-priority T_2 can no longer delay higher-priority T_1



Deadlock Problem

- Besides priority inversion problem, the basic synchronization techniques can also create deadlocks
  - Something that Priority Inheritance Protocol cannot solve
- Two tasks (T_1 and T_2) share two locks (S_1 and S_2)
- Deadlock in critical sections:



Priority Ceiling Protocol

- Priority Ceiling Protocol can prevent deadlocks
- Based on Priority Inheritance Protocol
- A priority ceiling is assigned to each semaphore
  - Priority ceiling = the highest priority of tasks that use the semaphore
- Task T is allowed to start a new critical section
  - Only if T's priority is higher than all priority ceilings of all the semaphores locked by tasks other than T

Priority Ceiling Protocol (cont'd-1)

- Let's see how Priority Ceiling Protocol handles deadlock
- Recall the previous example that resulted in deadlock
  - T_1 locks S_2 and then S_1 in the critical section
  - T_2 locks S_1 and then S_2 in the critical section
- Priority_Ceiling(S_1) = 1, Priority_Ceiling(S_2) = 1 (both semaphores used by T1)
  - Lower number means higher priority. Priority(T_1)=1, Priority(T_2)=2
- At t4, T_1 cannot lock S_2 because its priority is not higher than locked semaphores' priority ceilings
  - T_2 inherits T_1's higher priority