RTOS - SHARED DATA AND RESOURCES

EL-GY 6483 Real Time Embedded Systems

THE SHARED DATA PROBLEM

DATA SHARING IN C

Data is shared if the same memory location is accessed by multiple tasks, or by multiple instances of a task.

```
static int g_st_int;

int g_int;

int g_int_init = 4;

void *g_ptr;

void fn1 (int p1, int *p2) {
        static int st_loc;
        int loc;
}
```

Which of these might be shared if used by multiple tasks? The answer depends on both scope and memory storage.

DATA SHARING IN C

- g_st_int is global (is in scope throughout file) and static (stored at fixed memory location).
- g_int is global in scope and also stored at fixed memory location.
- g_int_init is global in scope and also stored at fixed memory location.
- g_ptr - the pointer is global in scope and stored at fixed memory location, and so is the data in the address it points to.
- p1 is on the stack. It's local to **fn1**(). If **fn1**() is called multiple times, each will have its own copy of p1.
- p2 is on the stack. Any instance of **fn1**() can change the address in **p2** without affecting others. But, changing the value at the address pointed to by **p2** might be problematic.
- st_loc is local - can only be accessed by instances of **fn1**() - but at a fixed memory location and is shared between all instances of **fn1**().
- loc is on the stack. It's local to **fn1**(). If **fn1**() is called multiple times, each will have its own copy of **p1**.

DATA SHARING IN MULTITASKING

With multitasking, data may be shared between multiple tasks:

- Multiple threads in an RTOS read/write to same data
- A main() and ISR both read/write to same data

Common programming patterns involving shared data:

- Setting flags and taking action based on flag status
- Producer/consumer: one task generates data and puts it into buffer, another task consumes data (removes it from buffer and processes it

TERMINOLOGY

A program or system has a race condition if its output depends on the sequence or timing of events that may occur concurrently.

If we know exactly in what order events will occur, data sharing is not as problematic. But with multitasking and interrupts, we often don't know the order.

The critical section is the part of the code that must not be concurrently run by more than one thread of execution (i.e., the part that accesses shared resource).

RACE CONDITION BUGS ARE THE WORST!

- Bugs typically appear randomly, often with low probability
- Exhaustive testing generally impossible
- Often causes symptoms that appear completely unrelated
- Porting code to a new architecture can unexpectedly make atomic operations non-atomic

EXAMPLE

```
static int temps[2];
void timer_irq () {
        temps[0] = IDR0 // read in value
        temps[1] = IDR1 // read in value
}
int main () {
int t0, t1;
        while (1) {
                t0 = temps[0];
                t1 = temps[1];
                if (t0 != t1) {
                        // do something
                }
        }
}
```

EXAMPLE

Does this help?

```
static int temps[2];

void timer_irq () {
        temps[0] = IDR0 // read in value
        temps[1] = IDR1 // read in value
}

int main () {
        while (1) {
                if (temps[0] != temps[1]) {
                // do something
                }
        }
}
```

Will the two values be accessed by the same (assembly) instruction?

THREAD SAFETY

THREAD SAFE

A piece of code is thread-safe if it guarantees safe (correct) execution by multiple threads at the same time

THREAD LOCAL STORAGE

Use variables that “look” like global variables (e.g. to other functions) but exist once per thread.

```
__thread int i;
extern __thread struct state s;
static __thread char *p;
```

Needs special support from compiler and linker.

REENTRANCY

Rules for reentrancy:

- A reentrant function may not use variables in a nonatomic way unless they are stored on the stack of the task that called the function or are otherwise private.
- A reentrant function may not call any other functions that are not themselves reentrant. Many standard library calls are nonreentrant:
    - malloc
    - printf (most I/O or file function)
- A reentrant function may not use the hardware in a nonatomic way.

REENTRANT FUNCTIONS

Not reentrant:

```
int t;
void swap(int* x, int* y) {
        t = *x;
        *x = *y;
        // hardware interrupt might invoke isr() here while
        // the function has been called from main or other
        // non-interrupt code
        *y = t;
}
void isr {
        int x=1, y=2;
        swap(&x, &y);
}
```

ATOMIC OPERATIONS

Shared resource is accessed using atomic operations – operations that execute in a single CPU instruction.

```
// set PD12 thru PD15
GPIOD->ODR |= 0xF000;

// atomic set PD12 thru PD15
GPIOD->BSRRL = 0xF000;

// what about this one?
GPIOD->BSRRL |= 0xF000;
```

MUTUAL EXCLUSION

Use synchronization primitives to ensure that only one task is in critical section at a time.

TECHNIQUES FOR THREAD SAFETY

- Thread-local storage
- Reentrancy
- Atomic operations
- Mutual exclusion

The first two techniques work by avoiding data sharing. The second two techniques support shared state.

SYNCHRONIZATION PRIMITIVES

BINARY SEMAPHORE

- A semaphore, s, is a specific memory location that acts as a lock to protect critical regions.
- A binary semaphore has two states:
    - Not taken.
    - Taken.

Two system calls are used either to take or to release the semaphore.

- take(&s) (or wait(&s))
- release(&s) (or signal(&s))

BINARY SEMAPHORE

Direct access to any semaphore is not allowed - it has to go through the kernel.

```
if (lock == 0) {
        // semaphore free - take it
        lock = myPID;
        // do stuff
}
```

This code does not guarantee that the task has the lock!

EXAMPLE: BINARY SEMAPHORE

```
/* Task_1 */
...
wait(&s); // wait until A/D available
select_channel(acceleration);
a_data=ad_conversion();
signal(&s) // release A/D
...
/* Task_2 */
...
wait(&s); // wait until A/D available
select_channel(temperature);
t_data=ad_conversion();
signal(&s) // release A/D
```

COUNTING SEMAPHORE

A counting semaphore allows an arbitrary number of tasks to access shared resource together. It has an internal state defined by a counter. The sign of the counter value (N) has the following meaning:

- Negative: there are exactly -N threads queued on the semaphore.
- Zero: no waiting threads, a wait operation would put in queue the invoking thread.
- Positive: no waiting threads, a wait operation would not put in queue the invoking thread.

MUTEX

Similar to a binary semaphore, but has ownership: a mutex can be unlocked only by the thread that owns it. (A semaphore does not have ownership).

LOCK PROBLEMS

DEADLOCK

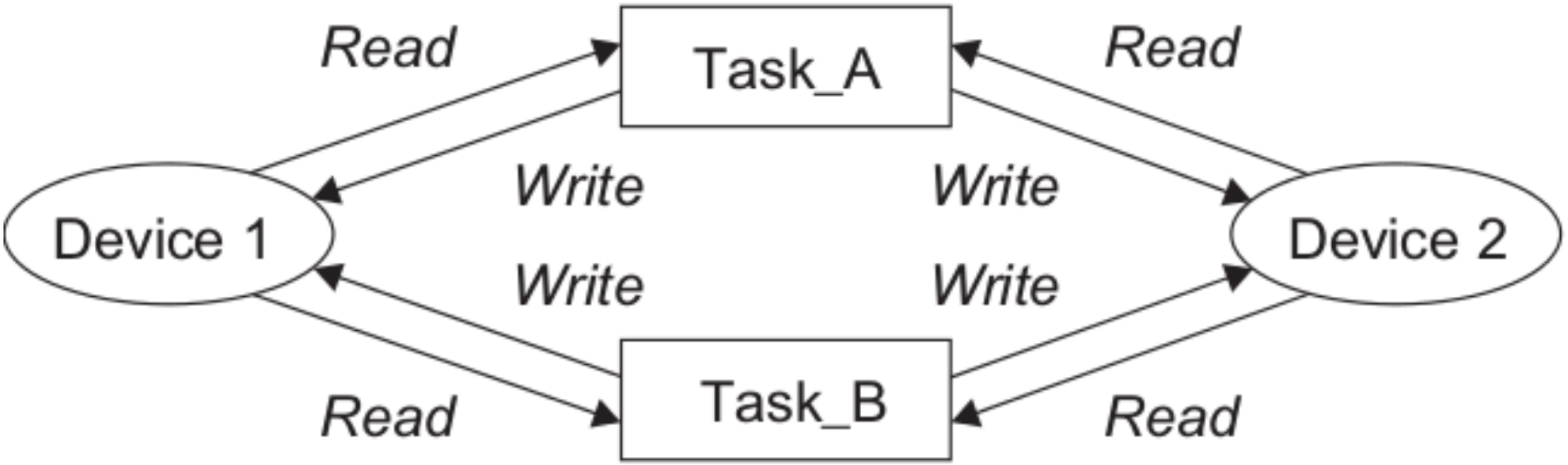
Tasks are blocked, each waiting on resources held by others.

```
/ * Task_A * /
wait( & s1);
wait( & s2);
// do stuff
signal( & s2);
signal( & s1);

/ * Task_B * /
wait( & s2);
wait( & s1);
// do stuff
signal( & s1);
signal( & s2);
```

DEADLOCK

Loops are a common cause of deadlock:



CONDITIONS FOR DEADLOCK

Deadlock occurs if all of these are satisfied:

- Mutual exclusion: a shared resource is held in non-shareable mode
- Circular wait: chain of tasks that holds resources needed by other asks further down the chain
- Hold and wait: tasks request a resource and then lock that resource until other subsequent resource requests are also filled
- No resource preemption: a resource can be released only voluntarily by the task that holds it

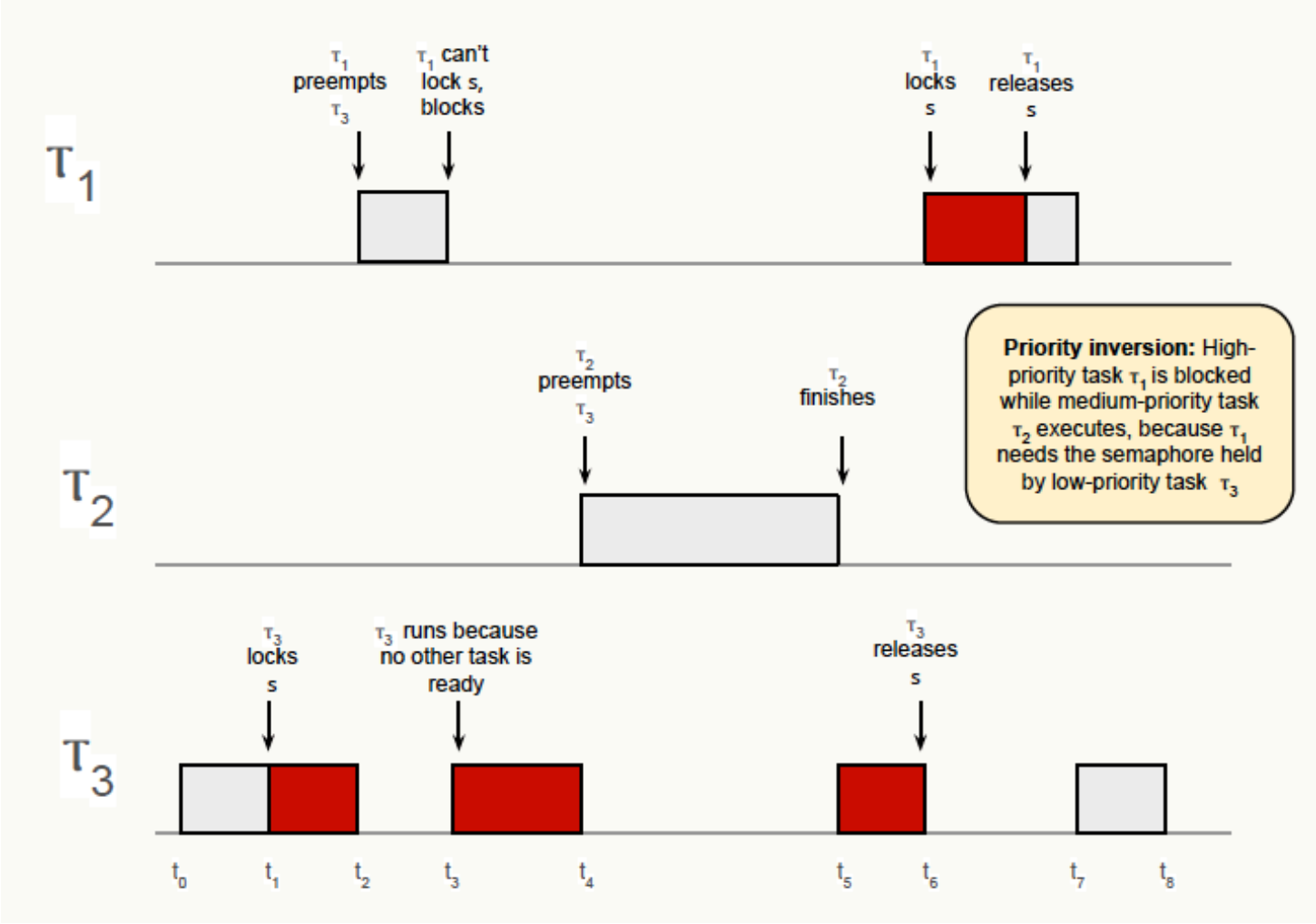
PRIORITY INVERSION PROBLEM

Suppose three tasks, τ_1, τ_2, τ_3, have decreasing priorities (τ_1 has highest priority).

- τ_1 and τ_3 share a resource that requires exclusive access, while τ_2 does not interact with either of the other two tasks

- Access to the critical section is carried out through the wait and signal operations on semaphore s

PRIORITY INVERSION PROBLEM



PRIORITY INVERSION PROBLEM

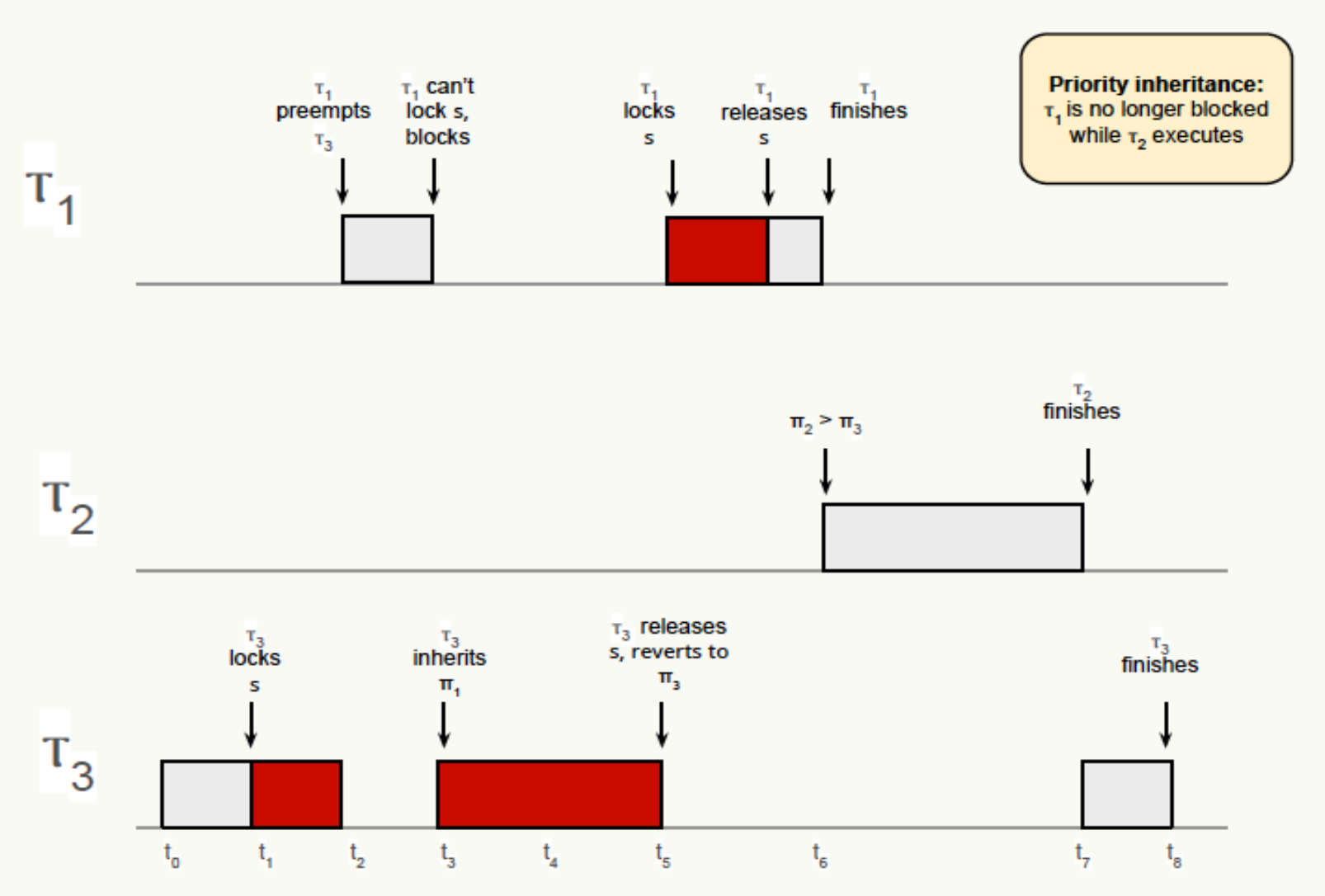
- τ_3 starts at t_0 and locks **s** at t_1
- At t_2, τ_1 arrives and preempts τ_3 inside its critical section
- τ_1 tries to acquire **s** at t_3, but is blocked because it is held by τ_3. So, τ_3 resume execution.
- At t_4, τ_2 arrives and preempts τ_3
- The execution time of τ_2 increases waiting time of τ_1 even though τ_1 has a higher priority than τ2!
- A priority inversion is said to occur within the time interval $[t_4, t_5]$, during which the highest priority task, τ_1, is prevented from execution by a medium-priority task τ_2.

PRIORITY INHERITANCE

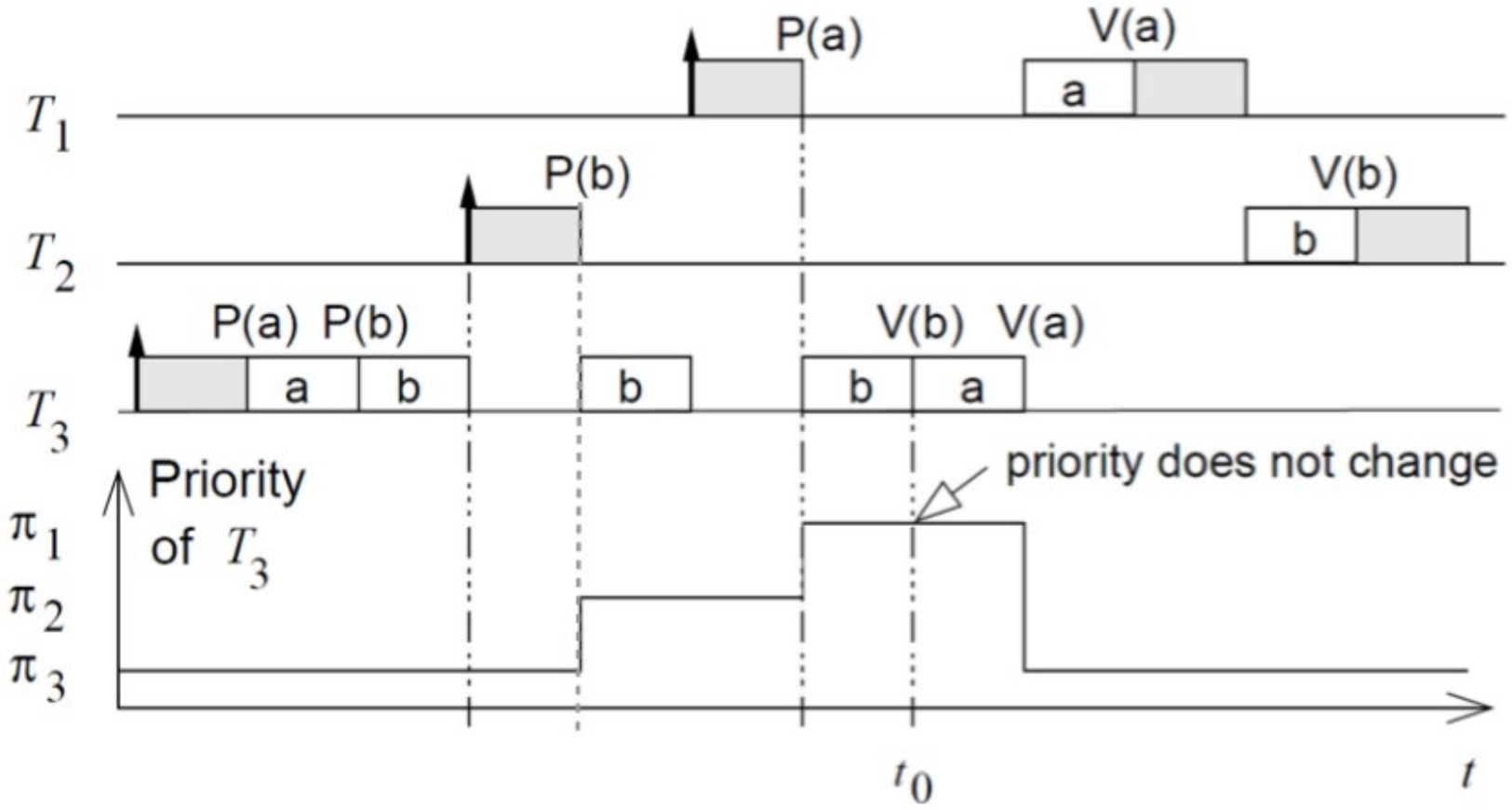
Priority inheritance prevents unbounded priority inversion.

- When a task, τ_1 blocks one or more higher-priority tasks, it temporarily inherits the highest priority of the blocked tasks (for only as long as it blocks the other tasks).
- The highest-priority task relinquishes the CPU whenever it seeks to lock the semaphore guarding a critical section that is already locked by some other task.
- Priority inheritance is transitive: if τ_3 blocks τ_2 that blocks τ_1, τ_3 inherits the priority of τ_1 via τ_2.

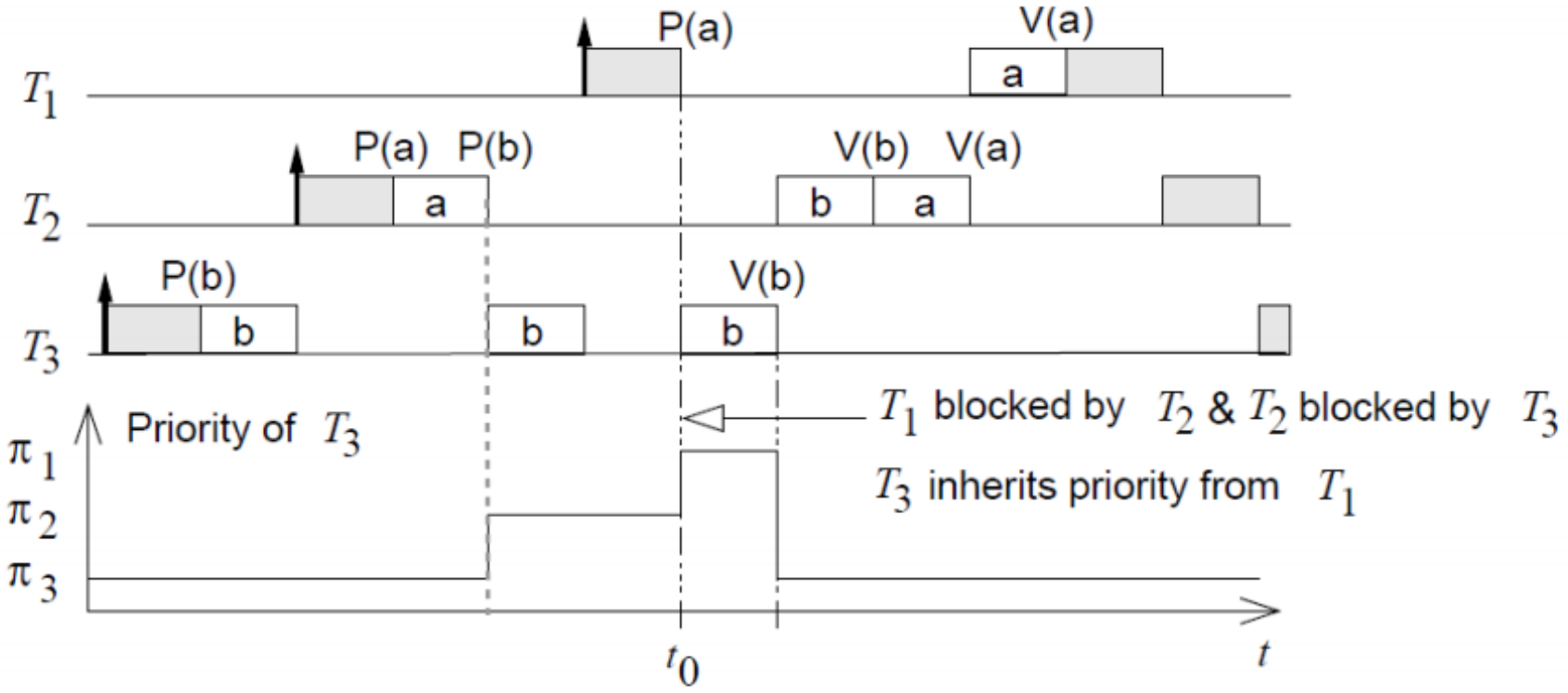
PRIORITY INHERITANCE



PRIORITY INHERITANCE WITH NESTED CS



PRIORITY INHERITANCE TRANSITIVITY



PRIORITY CEILING PROMOTION

Priority ceiling protocol extends to the priority inheritance protocol so that no task can enter a critical section in a way that leads to blocking it.

A task, τ_i, is blocked from entering a critical section if there exists any semaphore currently held by some other task whose priority ceiling is greater than or equal to the priority of τ_i.

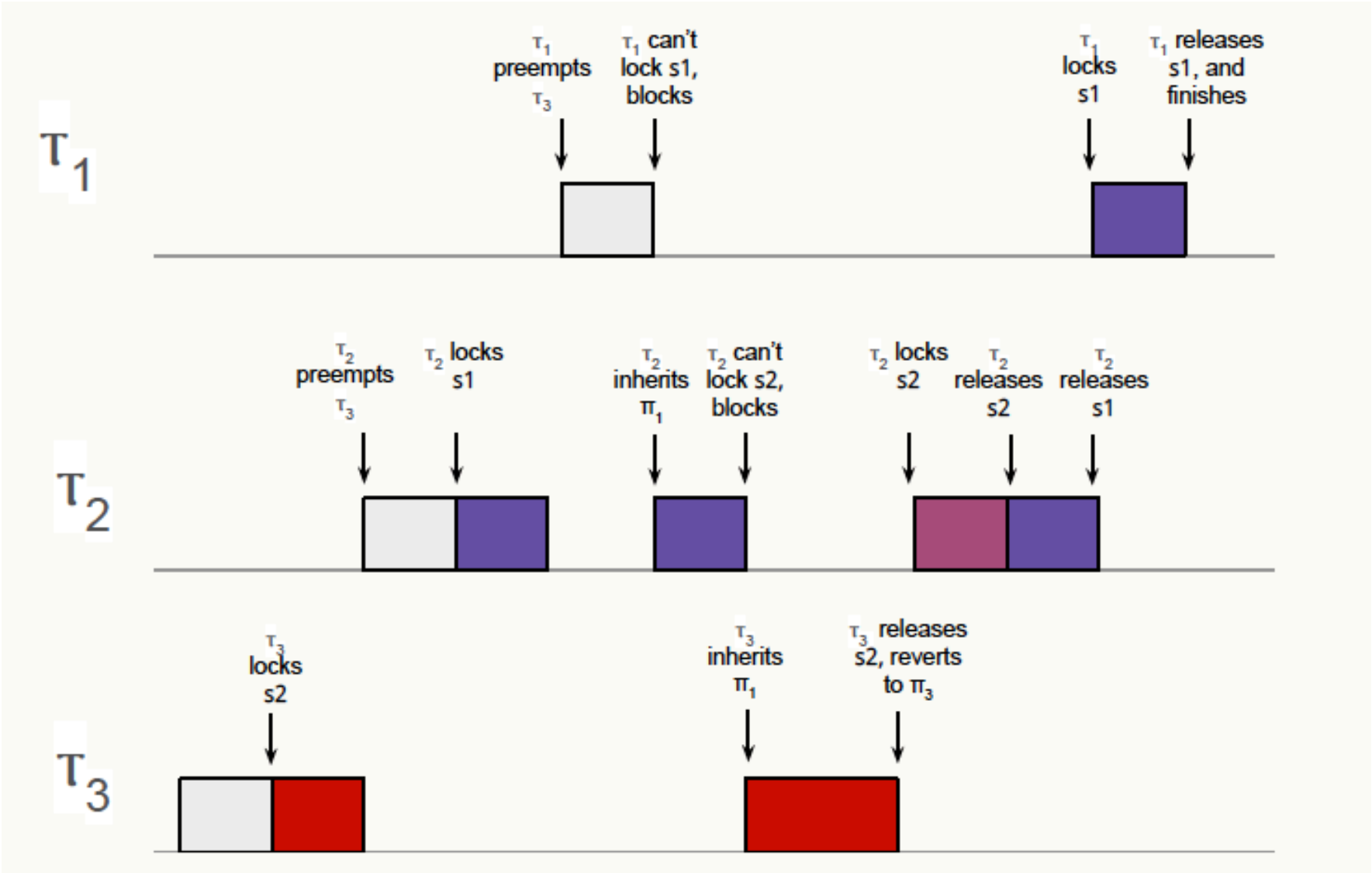
EXAMPLE

Consider three tasks of decreasing priority ($\pi_1 > \pi_2 > \pi_3$):

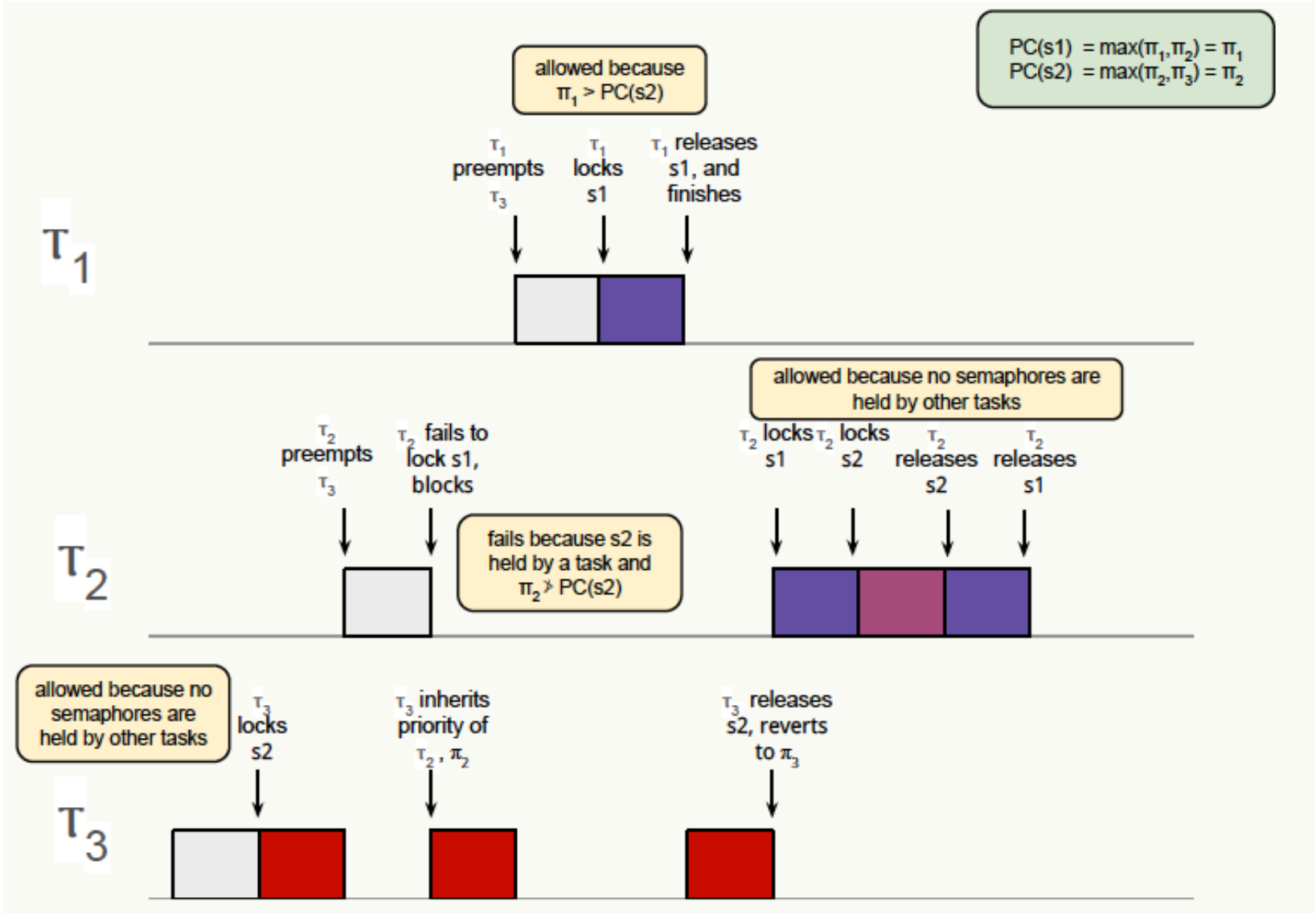
	τ_1		τ_2		τ_3
1	begin	1	begin	1	begin
2	lock s1	2	lock s1	2	lock s2
3	work on CS (s1)	3	work on CS (s1)	3	work on CS (s2)
4	unlock s1	4	lock s2	4	unlock s2
5	-	5	work on CS (s1, s2)	5	
6	-	6	unlock s2	6	
7	-	7	work on CS (s1)	7	
8	-	8	unlock s1	8	

What if their arrival is such that τ_3 locks s2 first, then τ_2 locks s1, and finally τ_1 tries to lock s1?

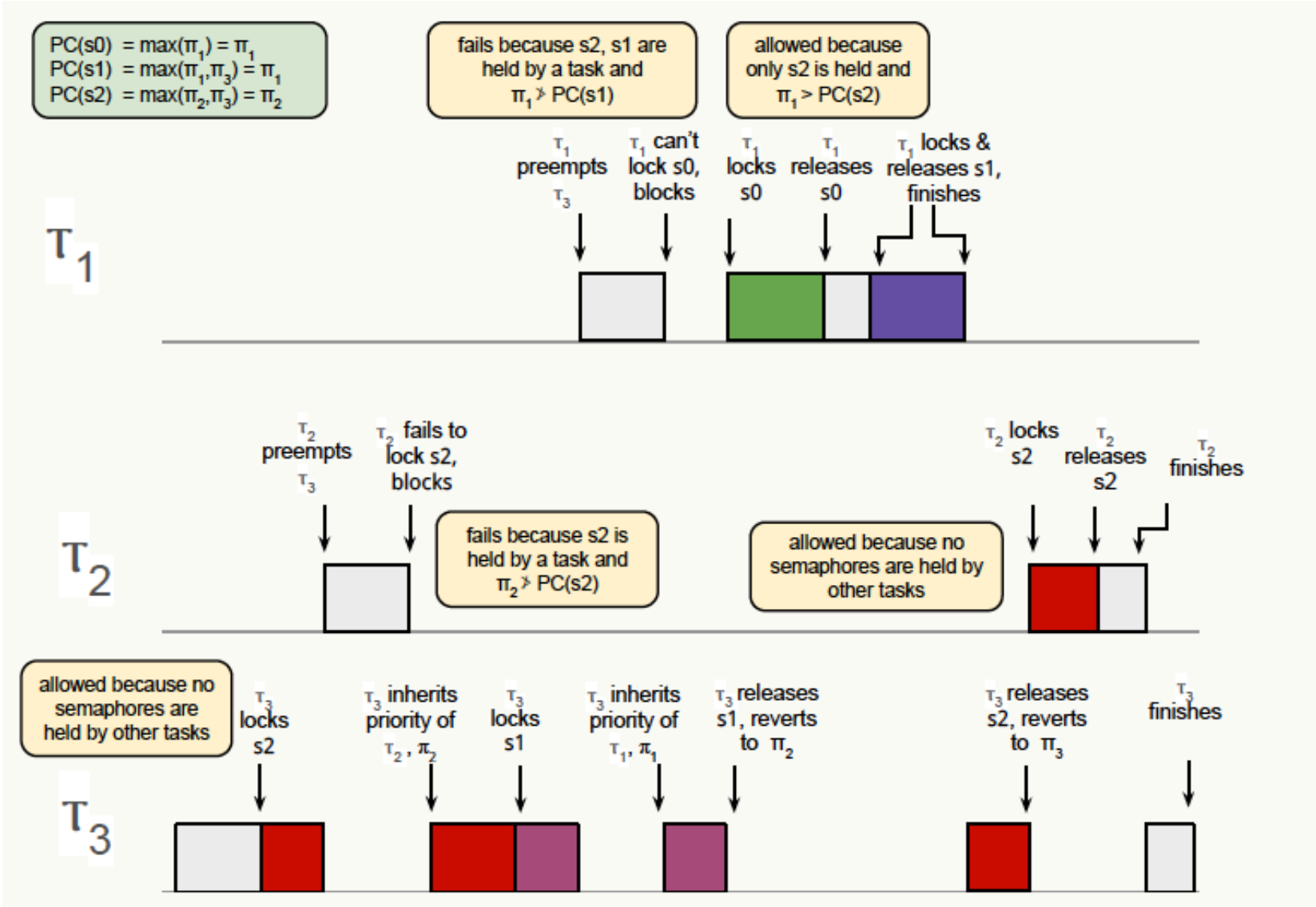
EXAMPLE 1: WITH PRIORITY INHERITANCE



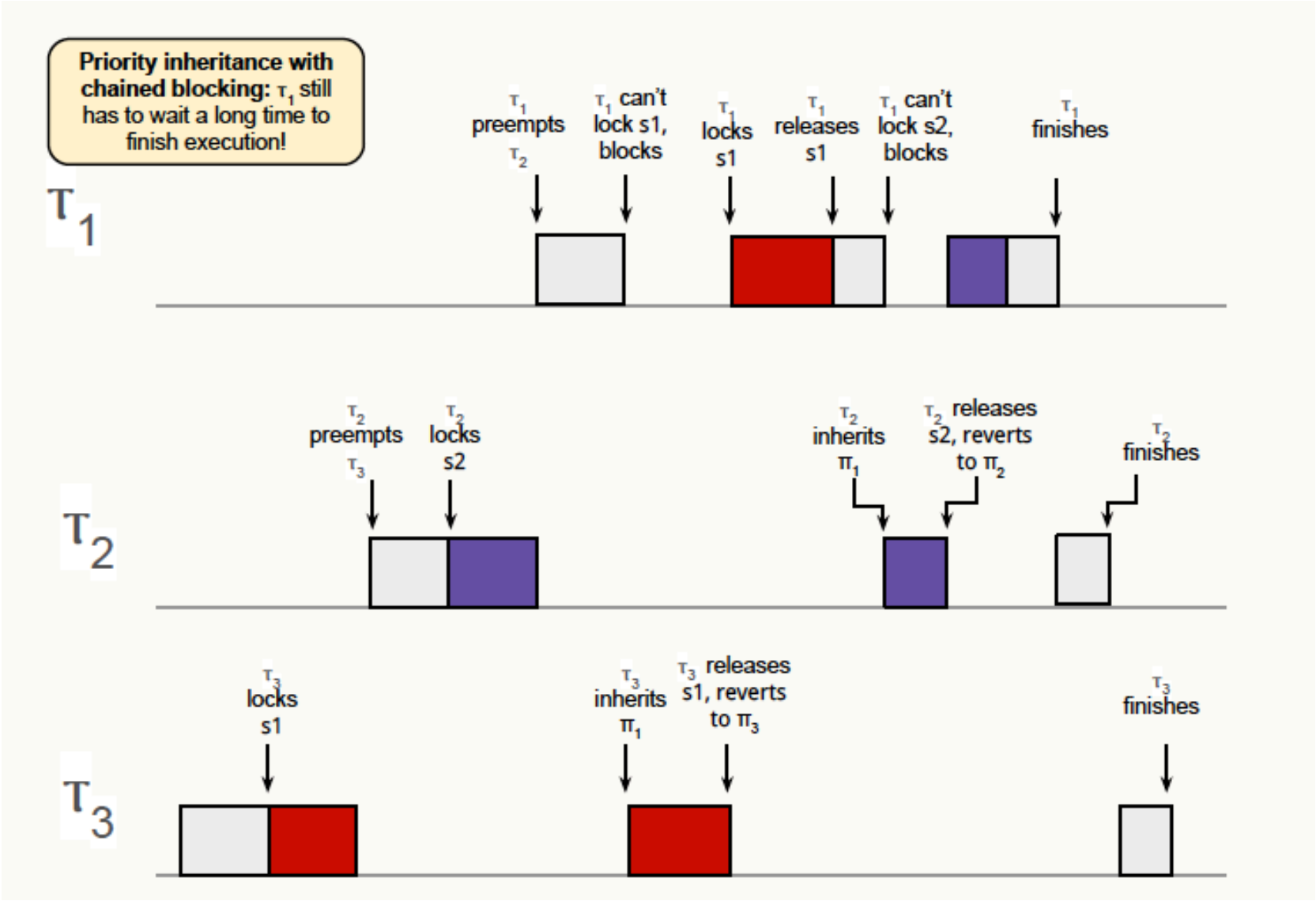
EXAMPLE 1: WITH PRIORITY CEILING PROMOTION



EXAMPLE 2: PRIORITY CEILING PROMOTION



EXAMPLE 3: PRIORITY INHERITANCE



EXAMPLE 3: PRIORITY CEILING PROMOTION