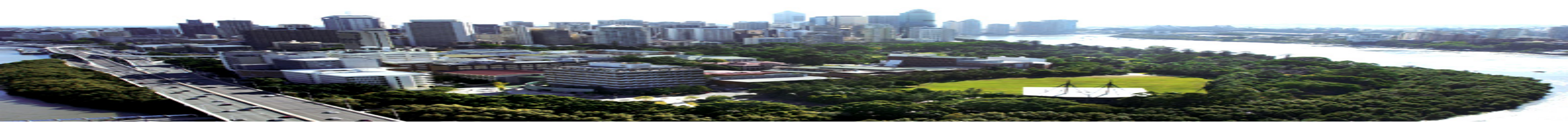
ENB 350 Real-time Computer based systems

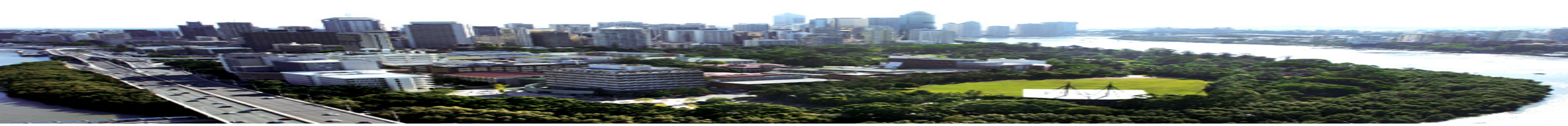
Lecture 9 – SHARED RESOURCES AND SEMAPHORES

Prof. V. Chandran



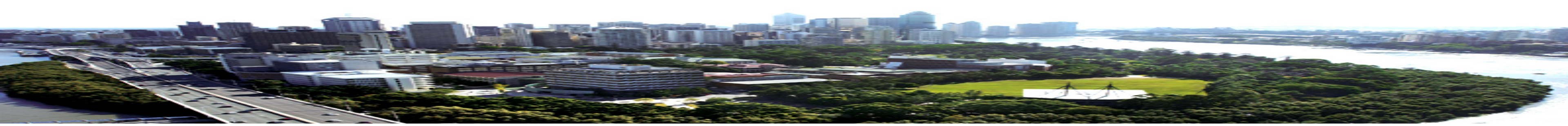
Contents

- Sharing resources
- Critical Sections
- ISR-ISR, ISR-Thread and Thread-Thread
- Semaphores
- Using Semaphores
- Using Mailboxes



Shared resources

- Two or more threads may share
  - **Buffers**
  - **Serial port**
  - **Non re-entrant function**
  - **global variables**
- If access is not coordinated, data streams can get mixed together, structures can become inconsistent



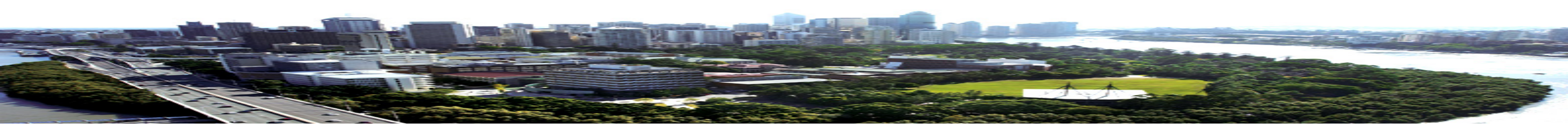
Critical sections

- **Critical sections** of code that manipulate a shared resource must be protected from pre-emption by other code that manipulates the same resource.

Shared memory

- Threads and ISRs may share **memory**, using a common data structure. This can get corrupted.

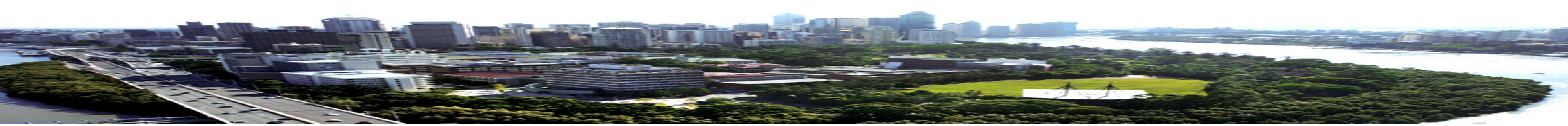
- Between 2 ISR s – if one interrupts the other while in the critical section
- Between ISR and thread – if the thread’s critical section is interrupted to execute the ISR
- Between threads – if access is not coordinated and a context switch occurs while one is in its critical section



Demonstration

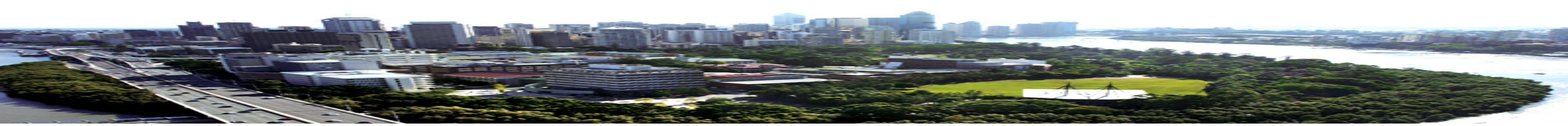
- Shared data and functions
  - Pre-emptive multitasking – Case: Not OK

    In this example, there is no protection. Two tasks share a buffer. They also need to write to the (same) display calling a shared function. The demo program shows corruption from unprotected access to the buffer.



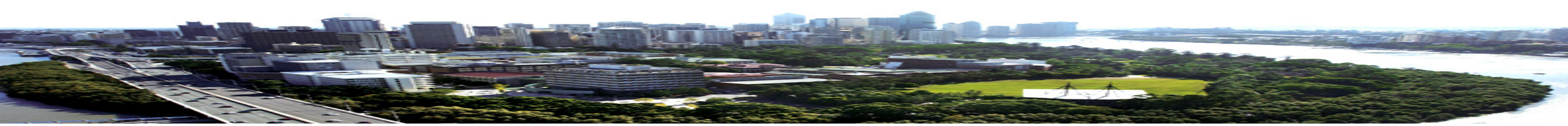
Co-operative multitasking

- Inherently greater protection because context switching is under programmer control
- No kernel calls that can cause a context switch (directly or indirectly) should be made in a critical section
- ISRs ?



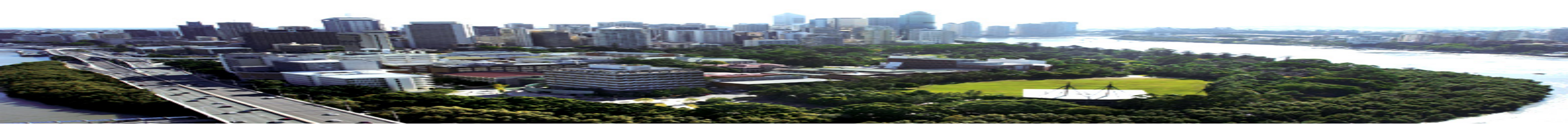
Co-operative multitasking

- ISRs return to the same task
- However, ISRs may modify shared data.
- Atomicity (uninterrupted execution) should be ensured in some cases. The ‘shared’ keyword in Dynamic C allows this.



Pre-emptive systems

- Context switches can be triggered by an interrupt.
- Solutions
  - Disable interrupts or
  - Arbitrate access
    - Flags with atomic test and set
    - Interrupt masking
    - Disabling task switching
    - Spin locks
    - Semaphores

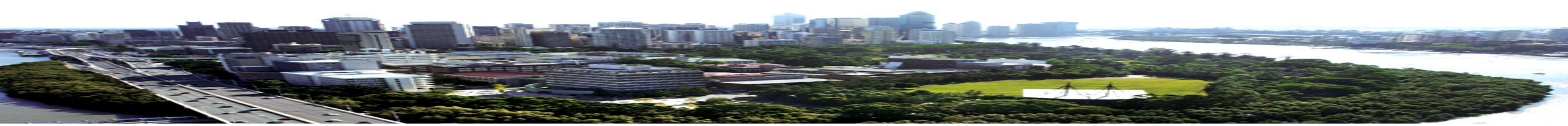


Disabling interrupts

- Okay for short sections of code. For long critical sections, it will degrade response time.
- Blocks all other threads and ISRs even if they don't need the shared resource
- Will work for all : ISR-ISR, ISR-thread and thread-thread situations
- Is the only solution for protecting shared resources in the ISR-ISR or ISR-thread situations.*

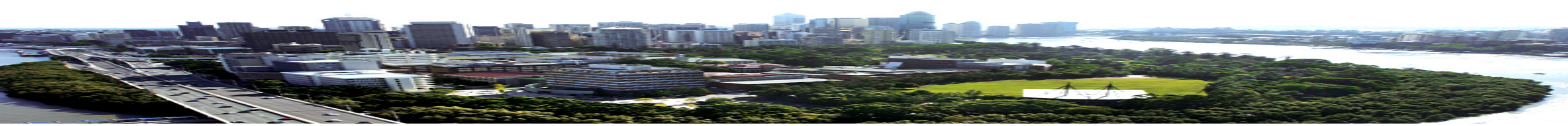
* ISRs can post but not pend on semaphores

ALL shared variables may not need protection

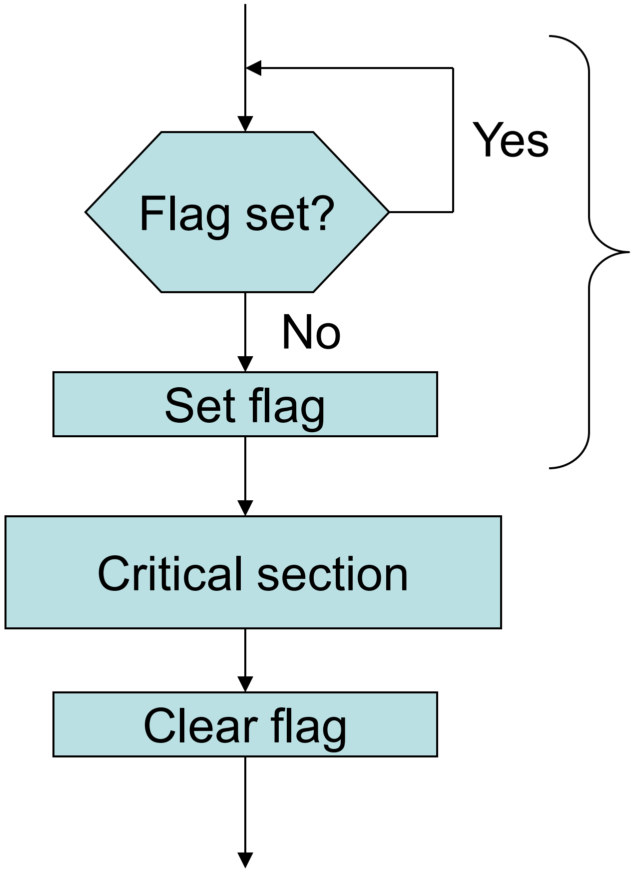


Disabling task switching

- Will protect a shared resource manipulated by two threads
- Is undesirable for real-time systems
- Will not work for ISR and ISR or ISR and thread cases



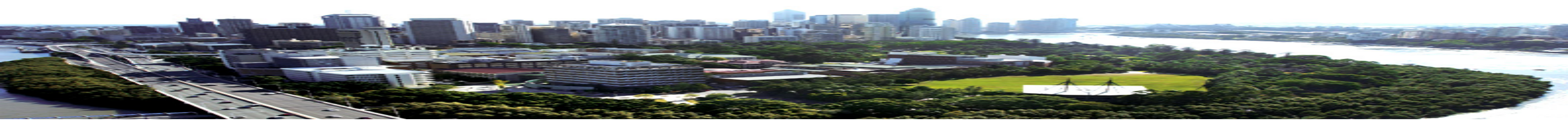
A SPIN LOCK




```
do {
    disable();
    ok = !flag;
    flag = TRUE;
    enable();
    } while (!ok) ;
```

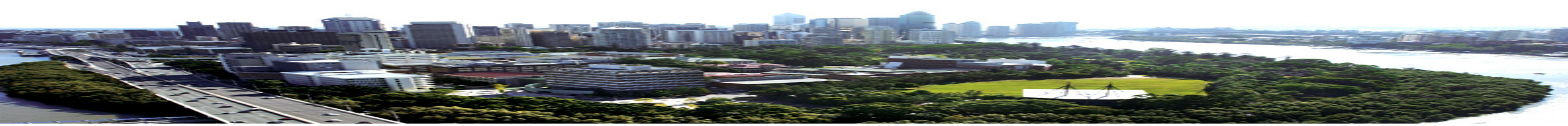
SPIN LOCK IN C

If flag = 0 set it to 1 and enter the critical section, else the resource is in use and therefore try again in a loop.



Spin lock

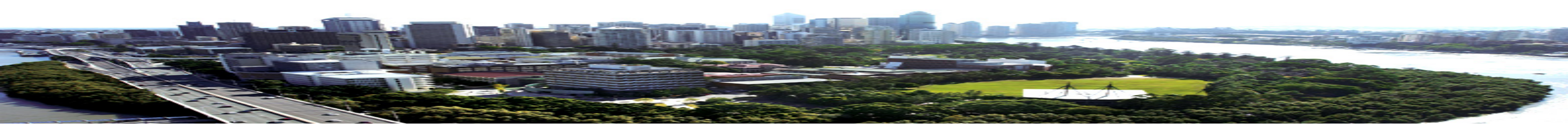
- There should not be a context switch (interrupt) between the checking and the setting of the flag.
- The “test and set” operation must be atomic (no interrupts are serviced in between and no context switch takes place).
- If the thread keeps on checking within a loop, CPU is wasted. Yield statements can be used to block the task and resume it after a time delay.



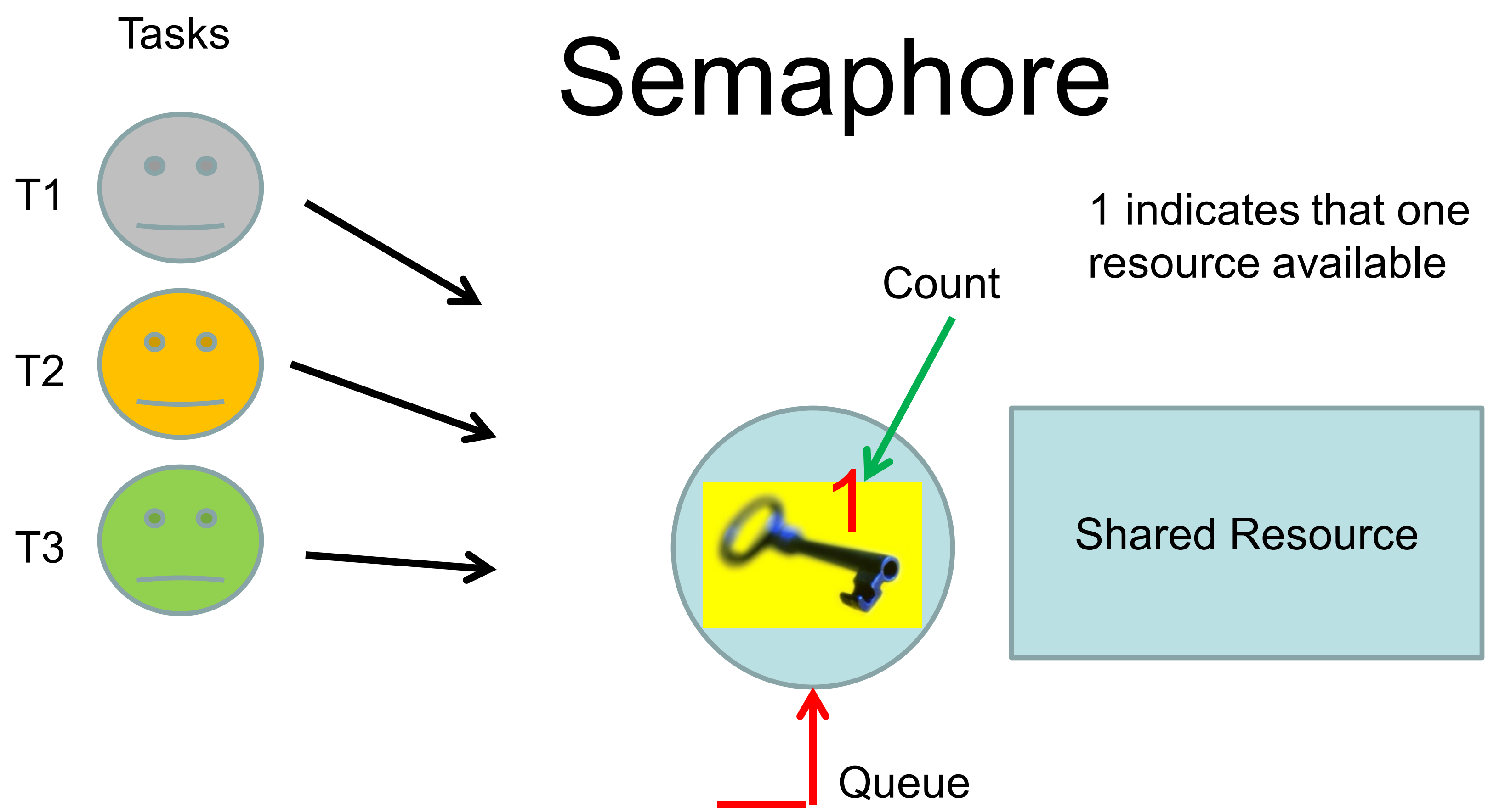
Semaphore

- A protected variable or abstract data type proposed by Edsger Dijkstra in the 1960s as a solution to the mutual exclusion problem, preventing race conditions and protecting shared resources

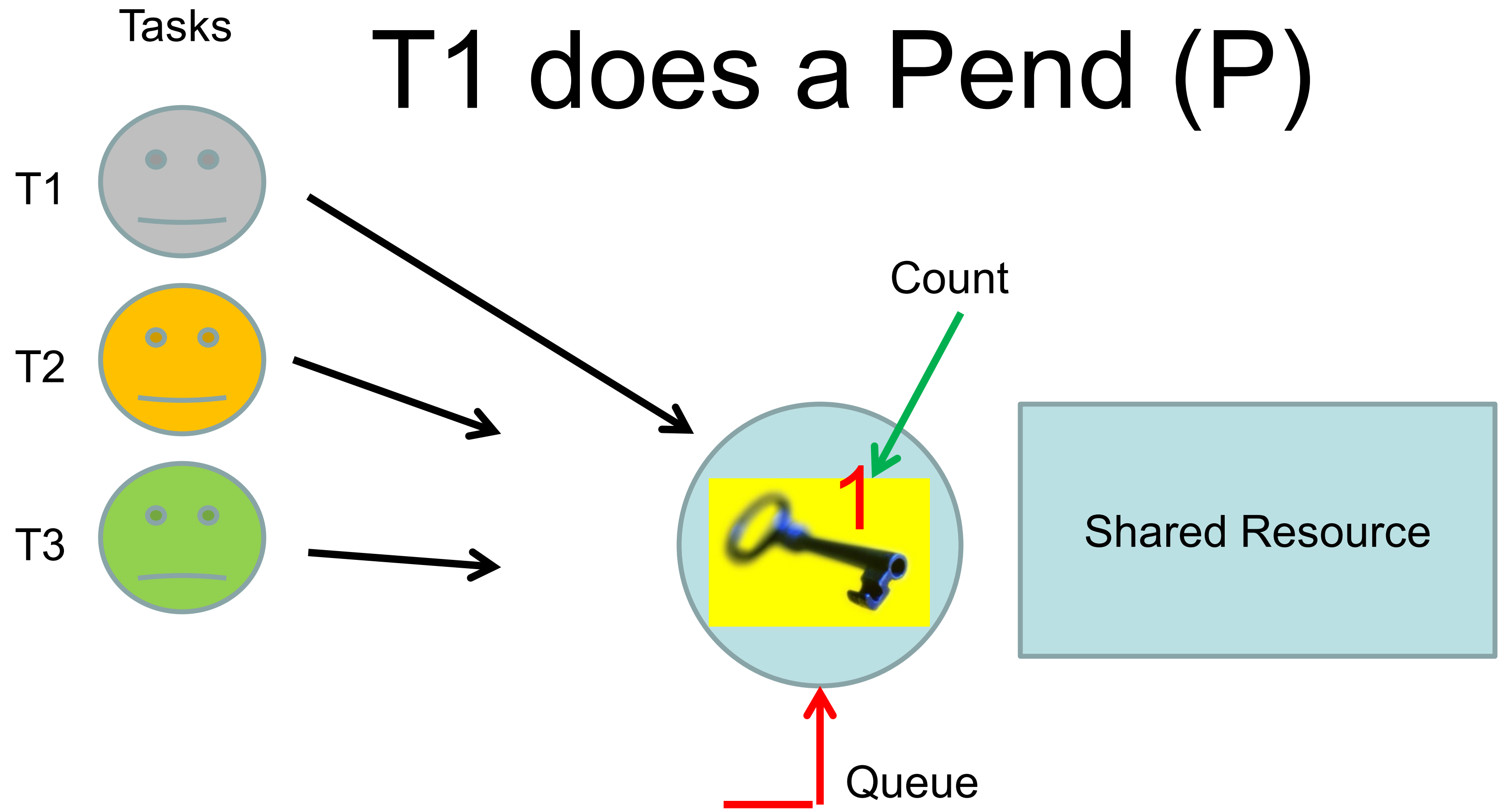
- Supports <u>P and V operations</u> that decrement and increment the value of <u>a count field</u>. Also called Pend and Post.



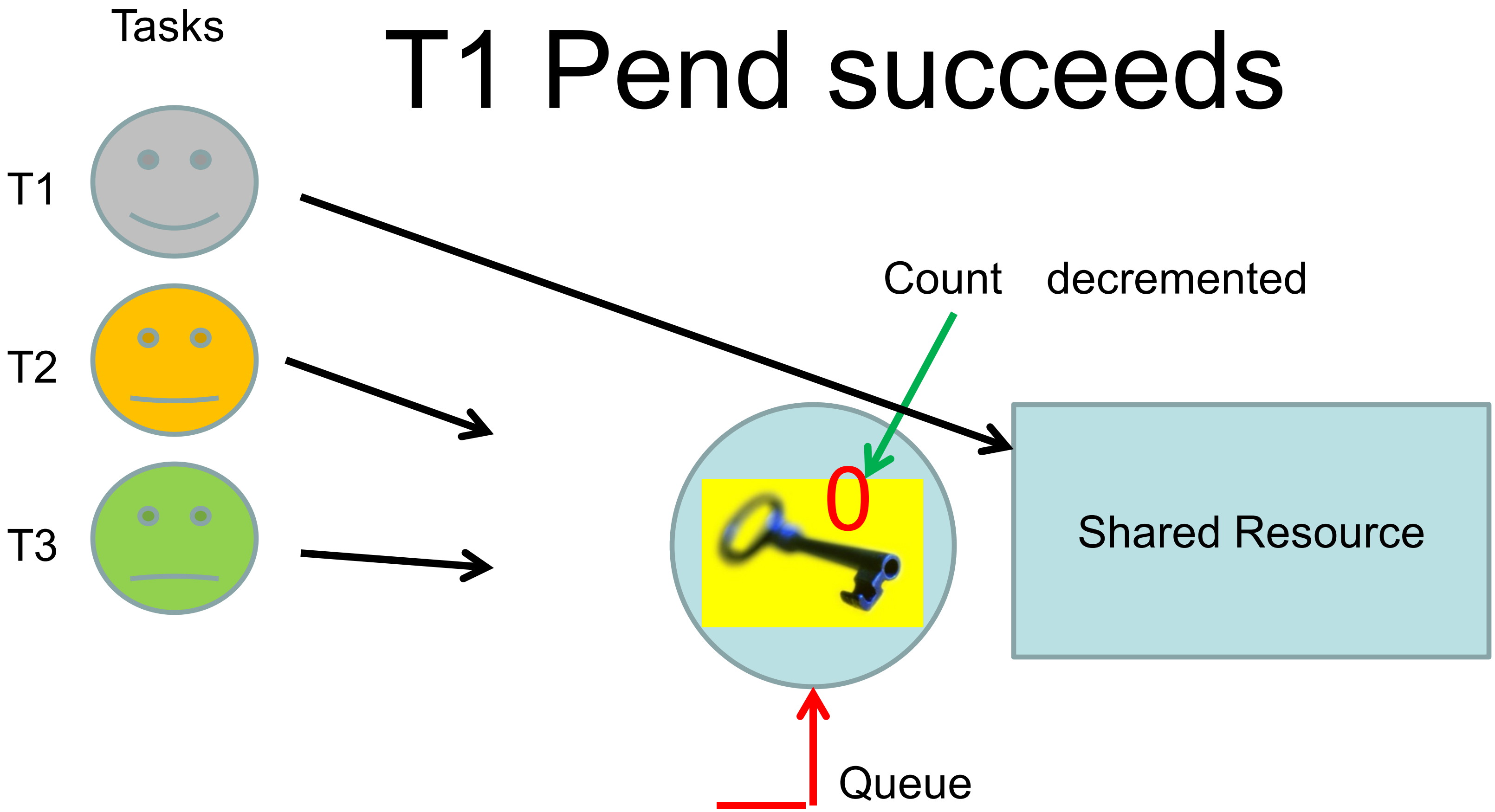
Semaphore


Tasks
T1
T2
T3
Count
1 indicates that one resource available
1
Shared Resource
Queue

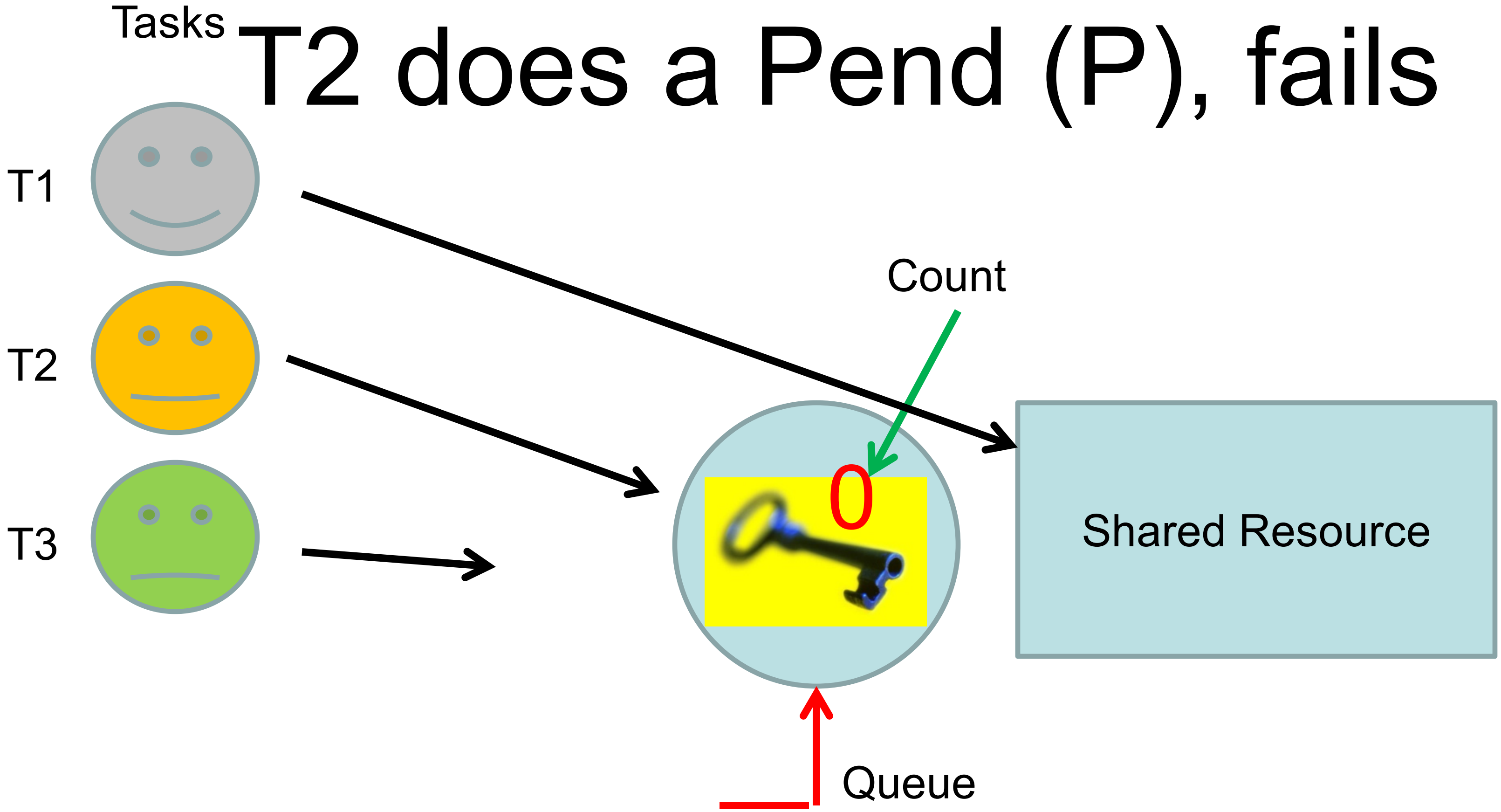
T1 does a Pend (P)



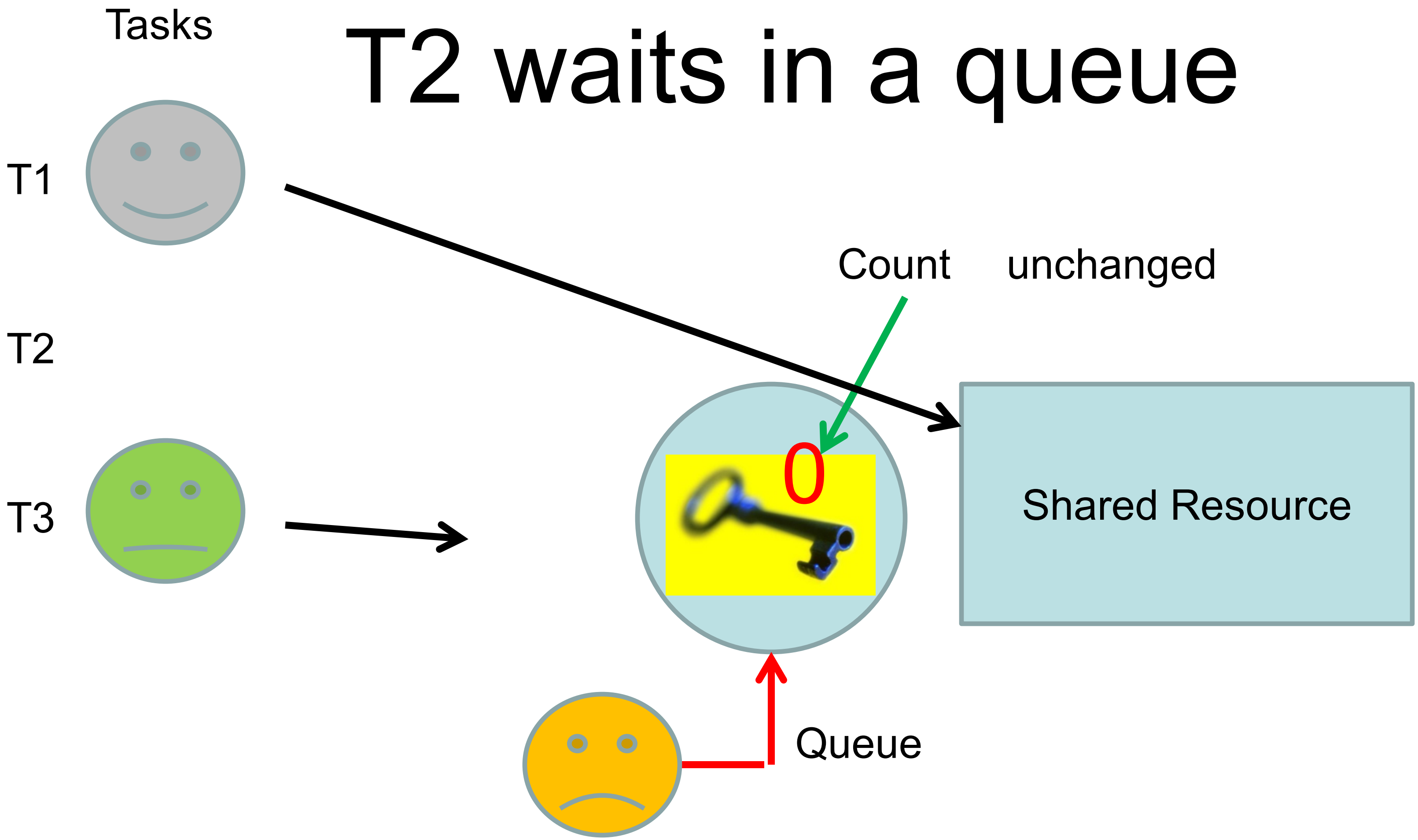
T1 Pend succeeds



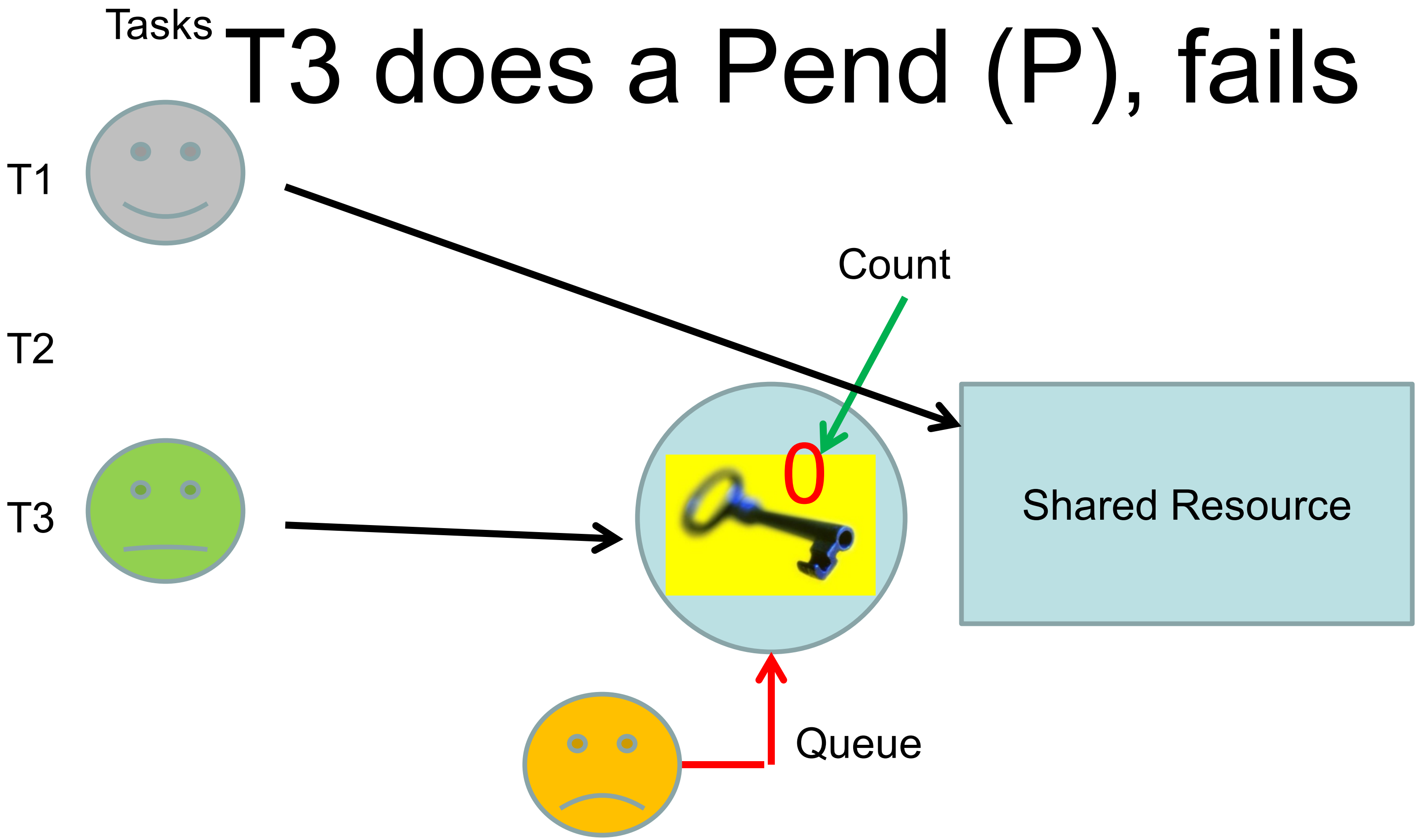
T2 does a Pend (P), fails



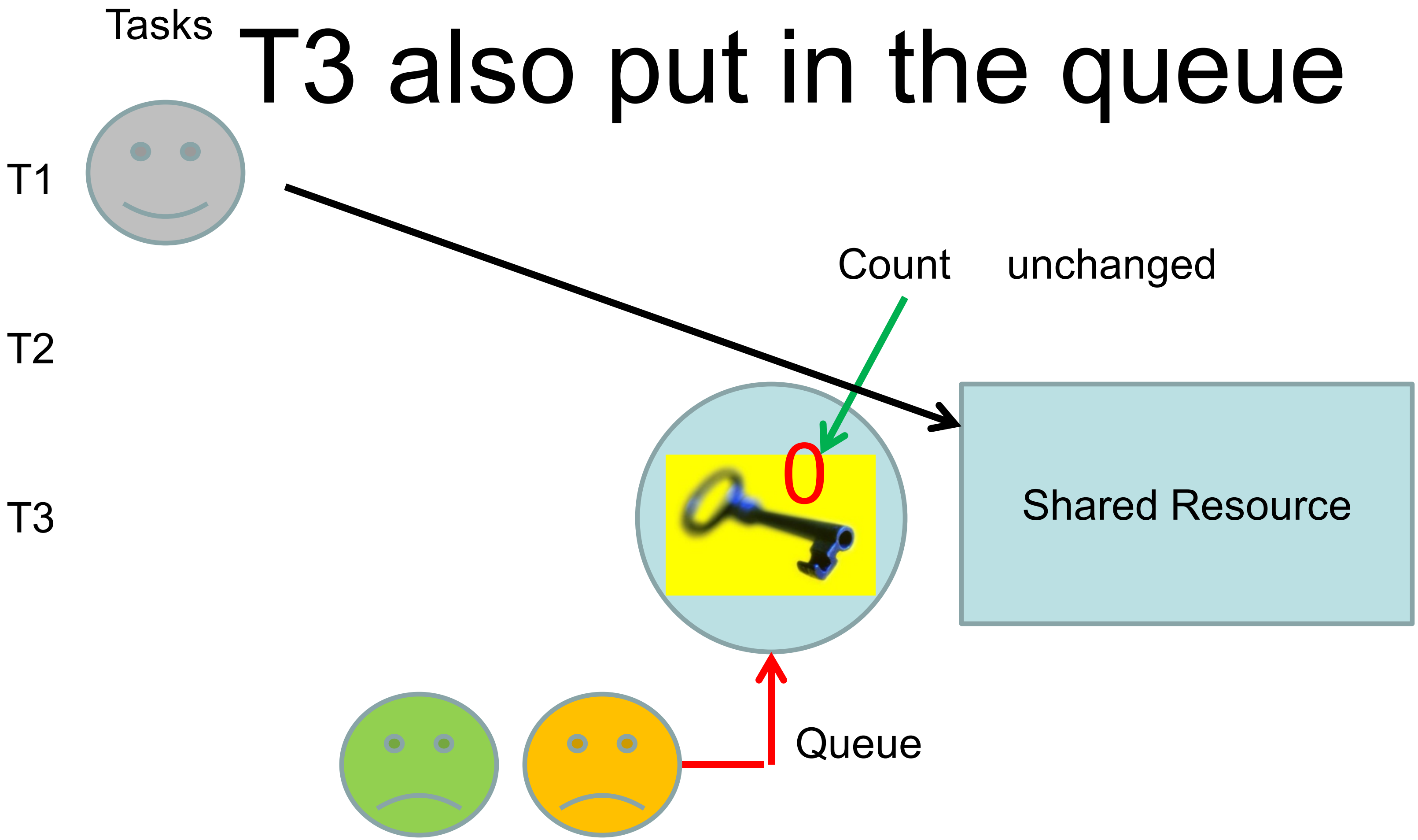
T2 waits in a queue



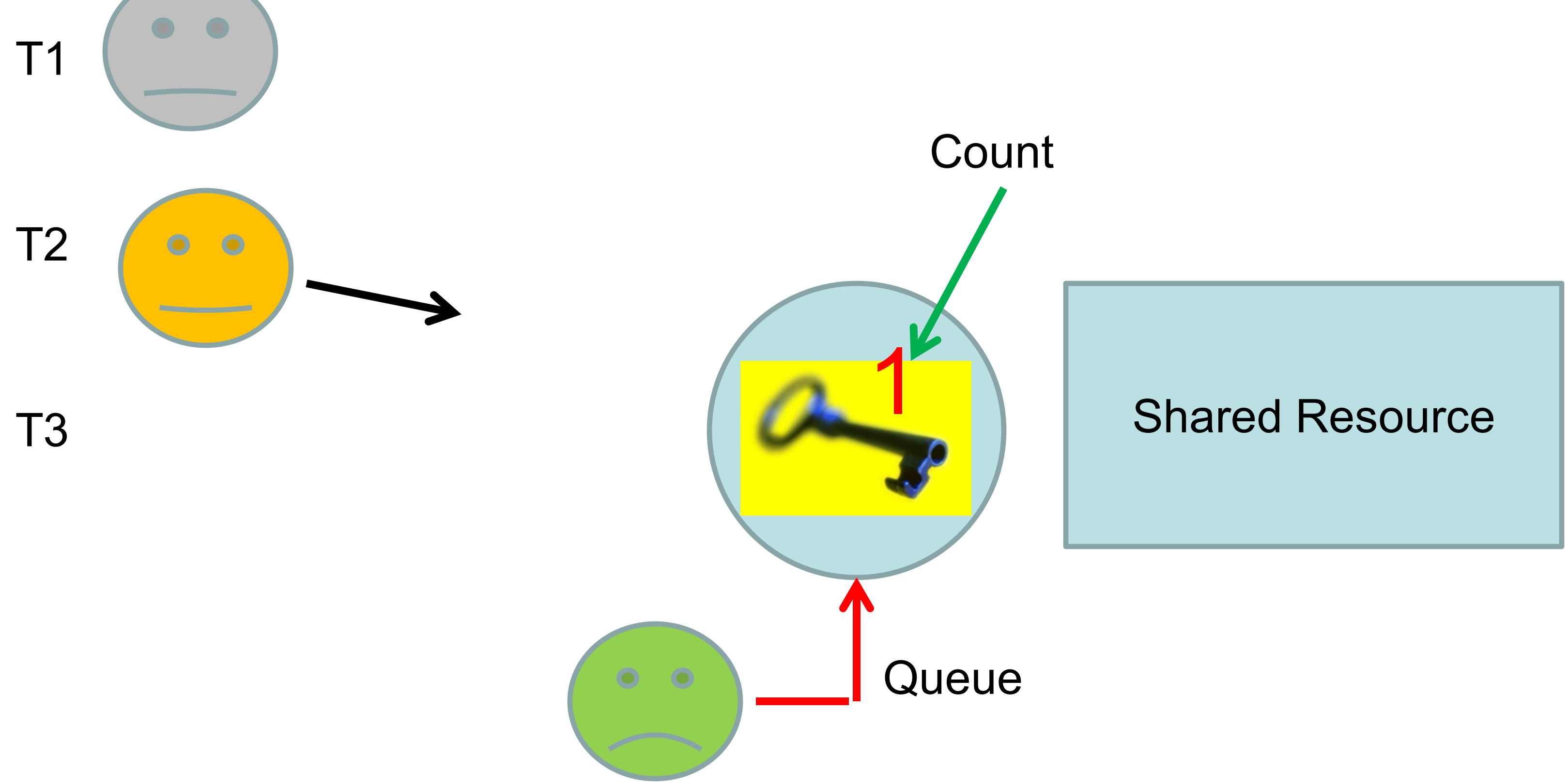
T3 does a Pend (P), fails



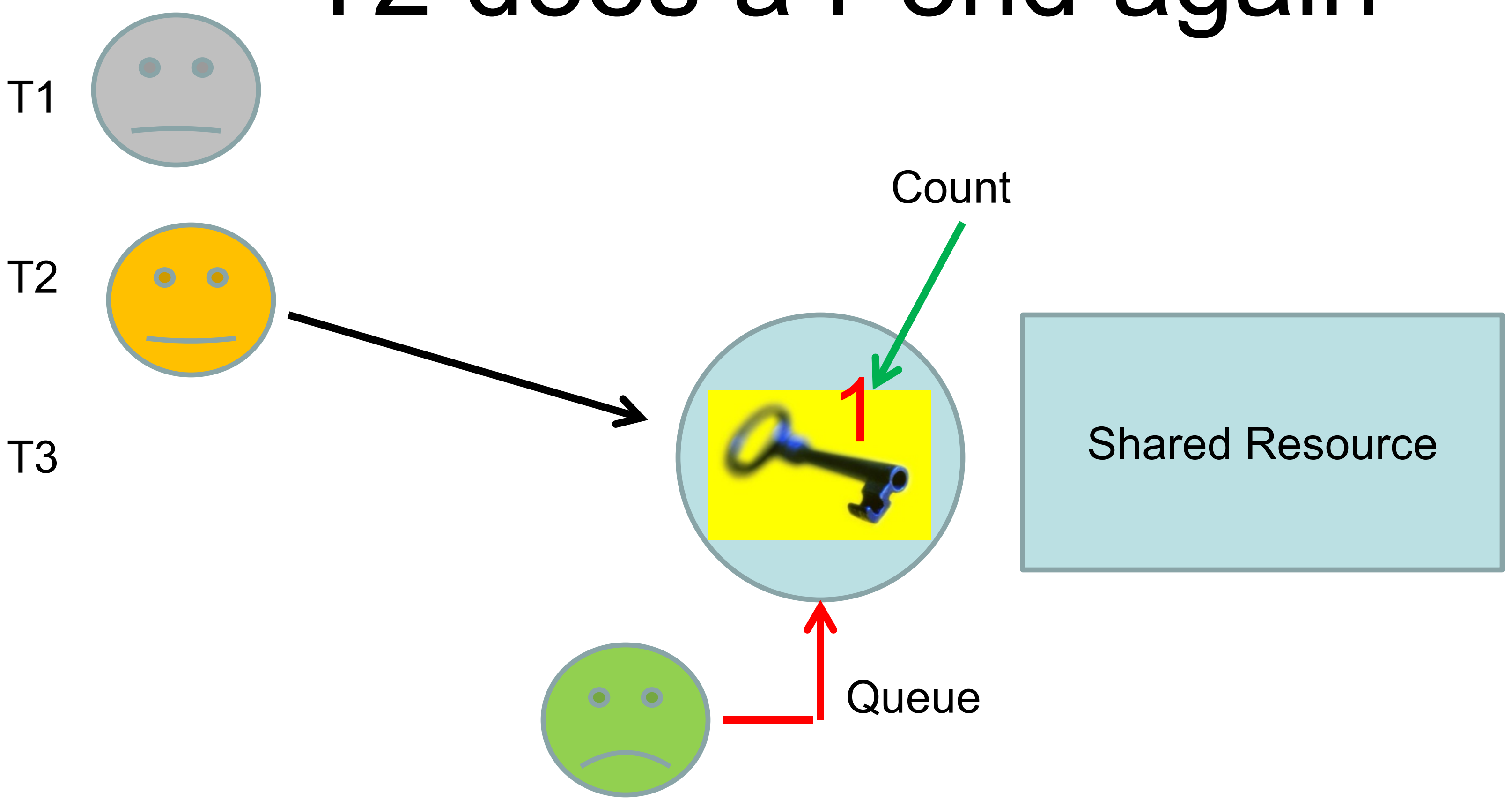
T3 also put in the queue



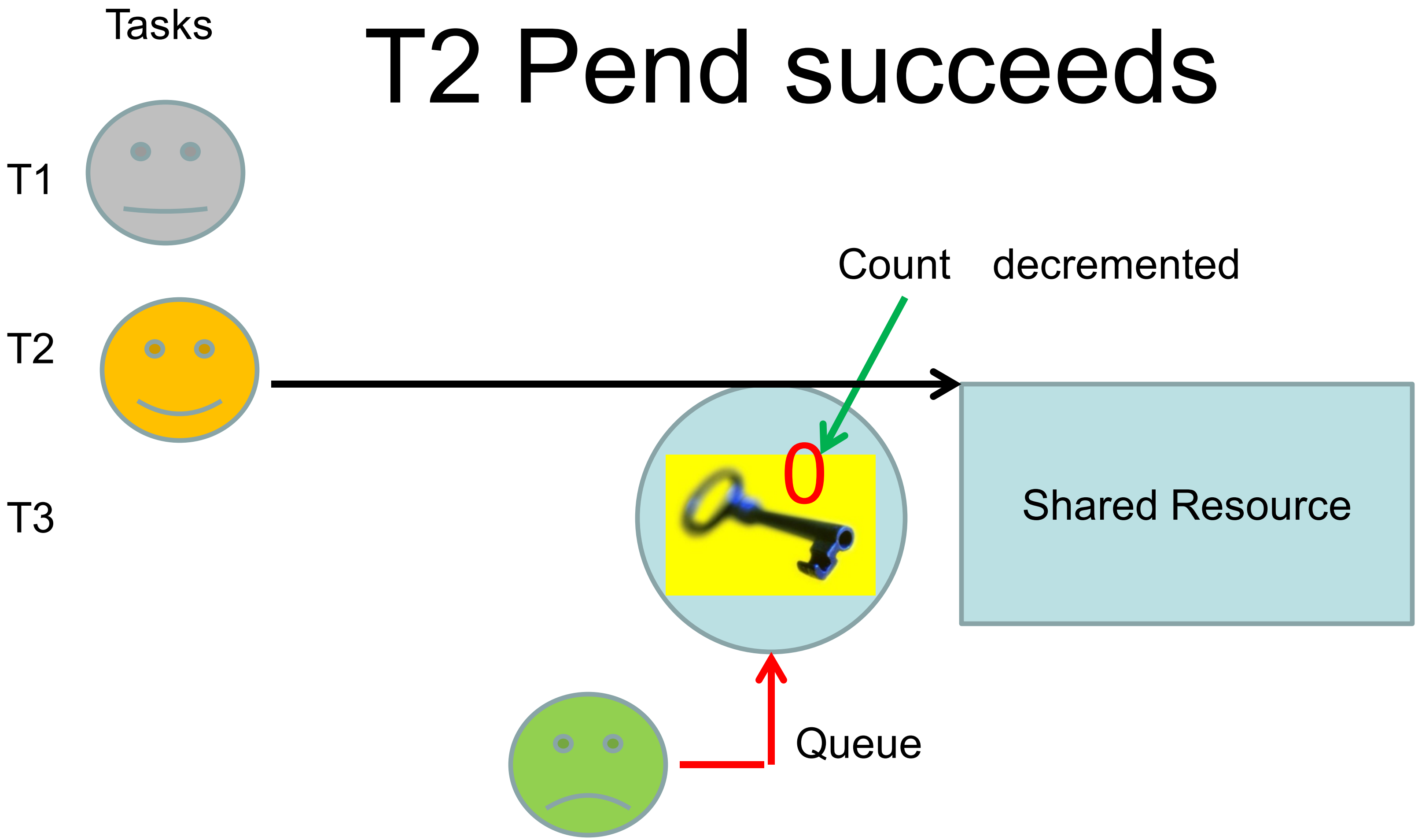
T1 does a Post (V)



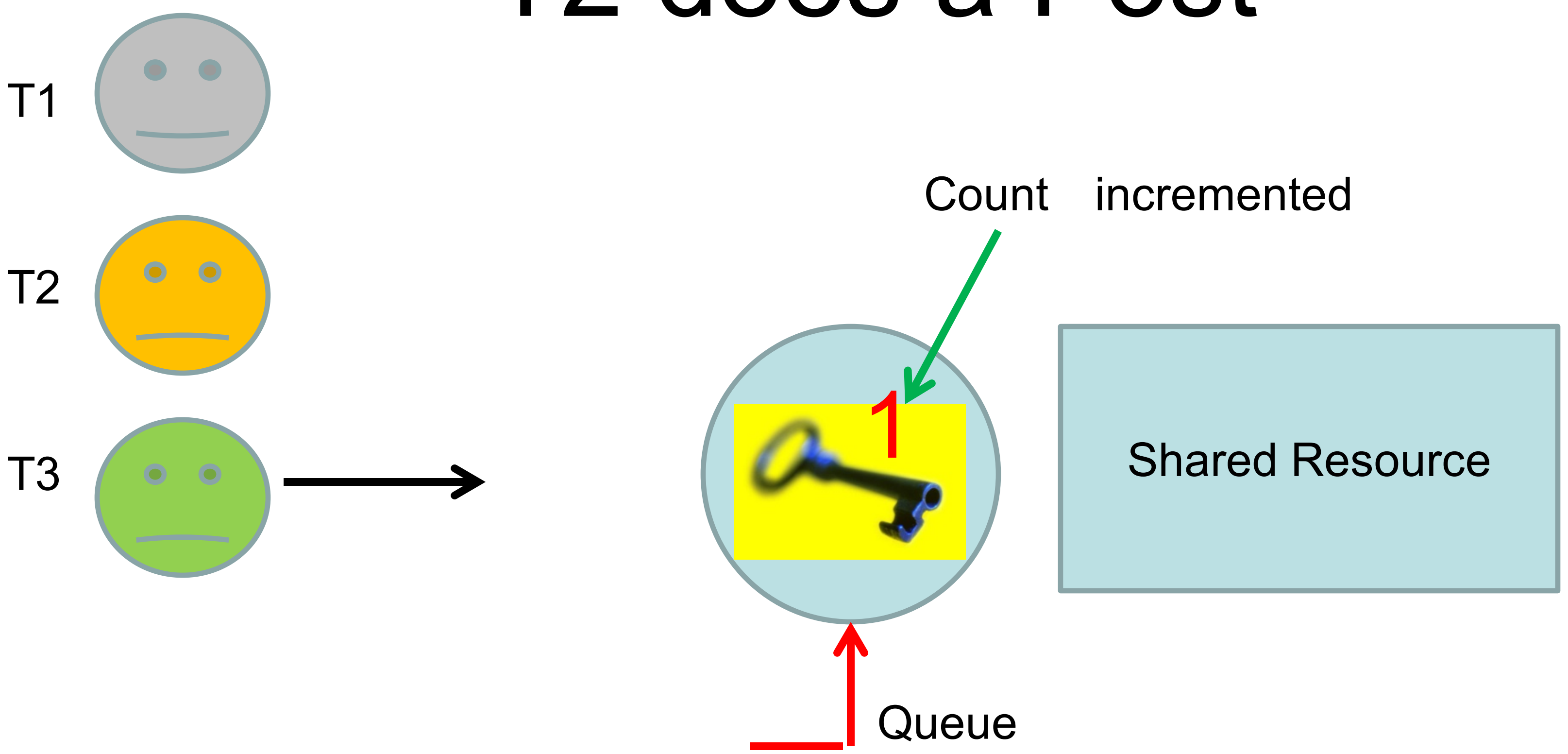
T2 does a Pend again



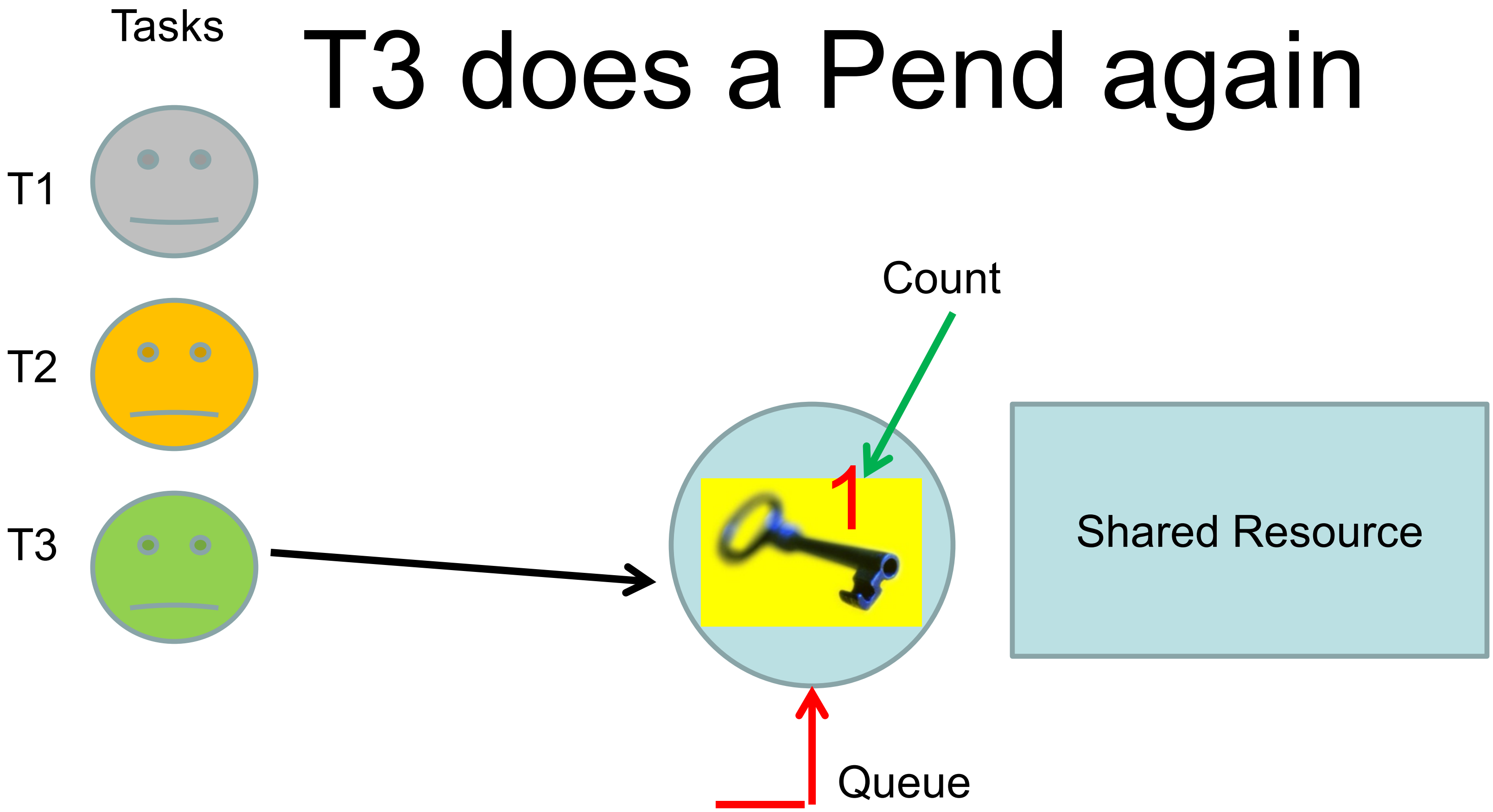
T2 Pend succeeds



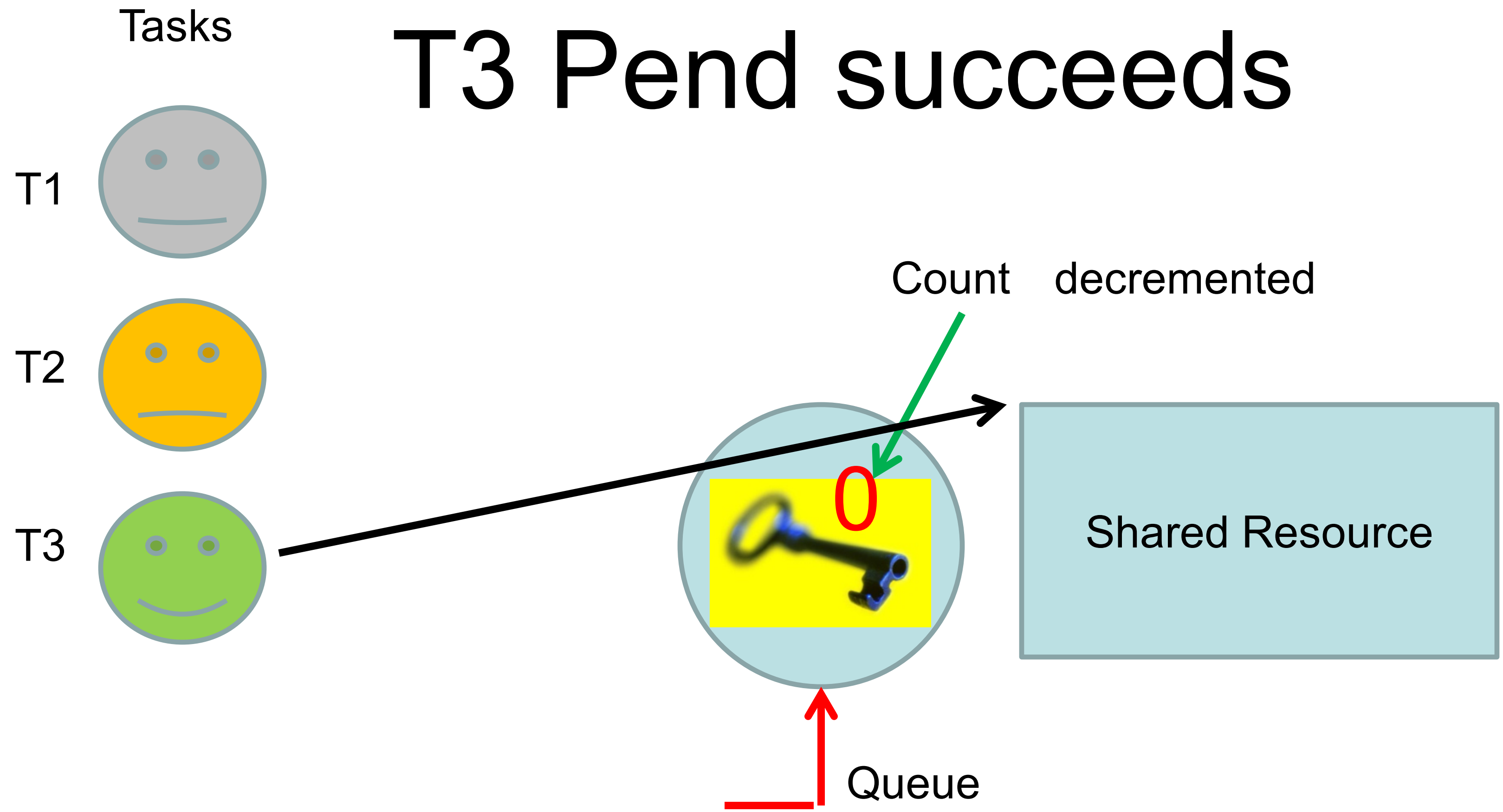
T2 does a Post

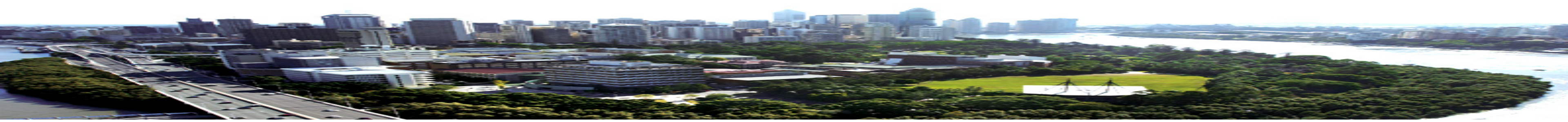


T3 does a Pend again

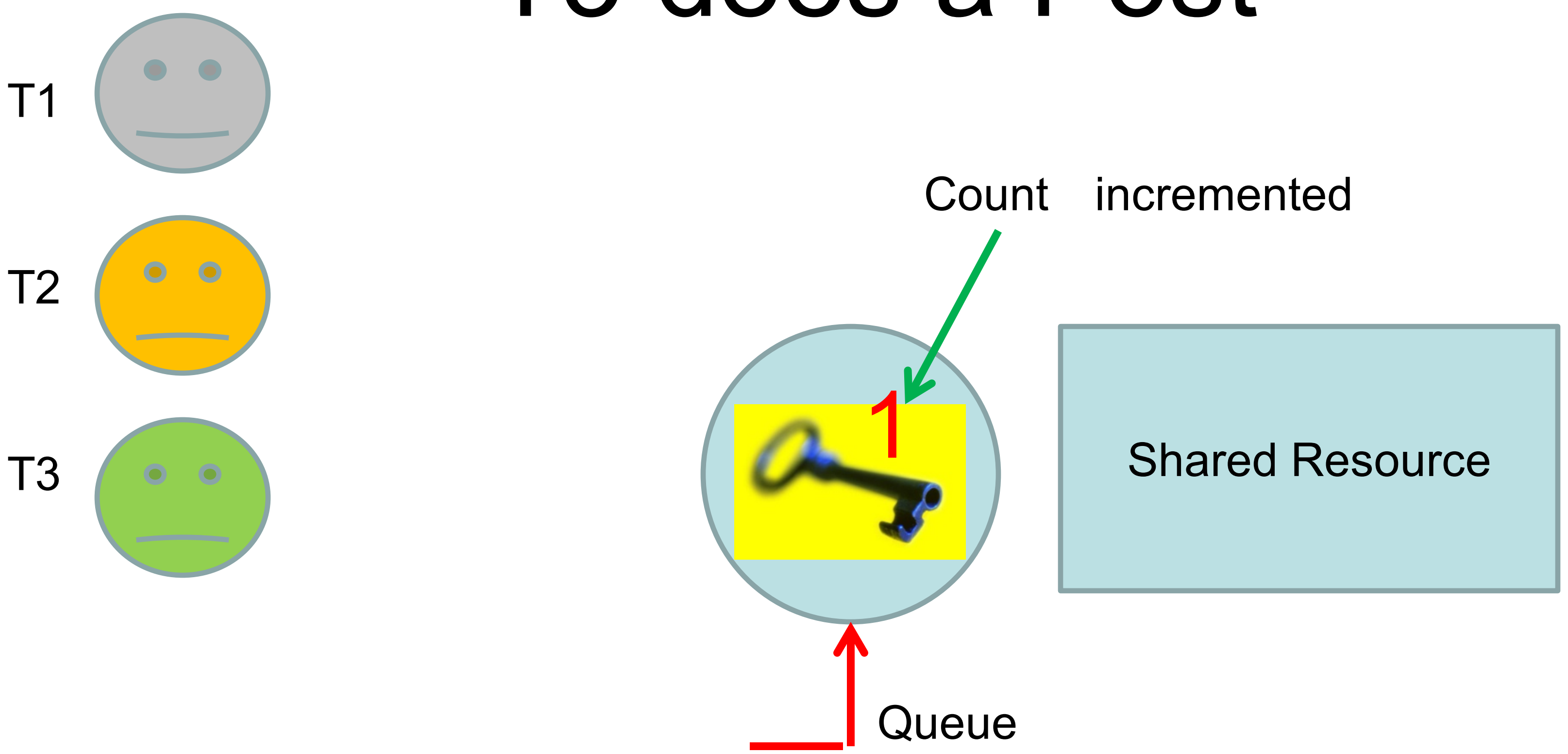


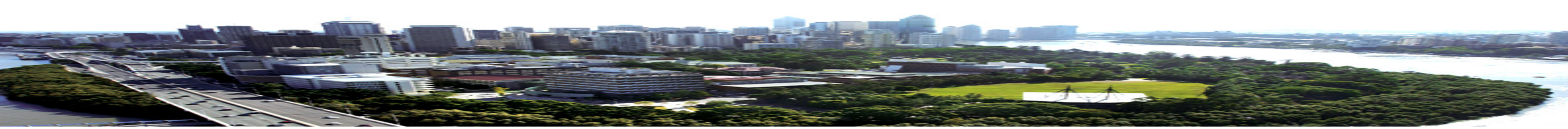
T3 Pend succeeds





T3 does a Post

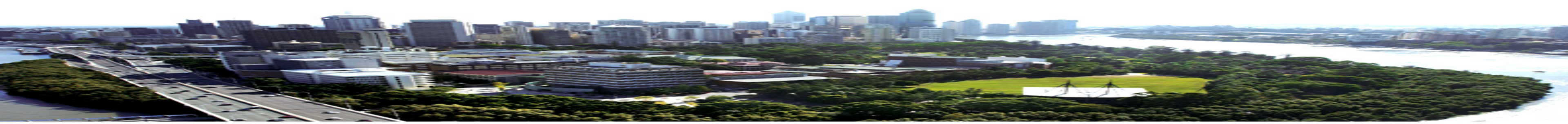




Semaphore

- P and V operations must be atomic. Also referred to as ‘pend’ and ‘post’ operations.

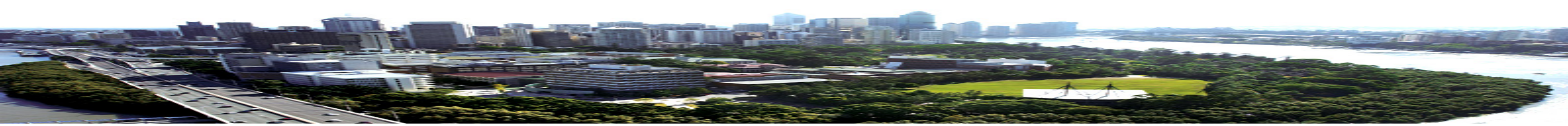
- Semaphores solve the mutual exclusion problem but do not guard against deadlock or priority inversion.



P and V operations

```
P(Semaphore s)                                    // Acquire Resource
{ if (s.count > 0)  s.count- -;
else put pointer to the calling process in the queue (s.queue).
}

V(Semaphore s)                                    // Release Resource
{ s.count++;
  release a process from the queue}

Init(Semaphore s, Integer ResCount)               // Initialize
{ s.count := ResCount;
 s.Queue = NULL;
}
```

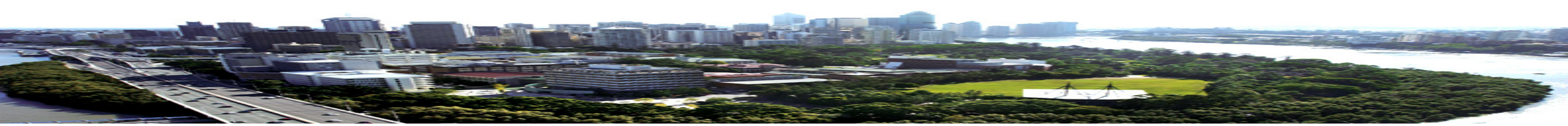
Again, operations must be atomic.



Counting and binary semaphores

- In a **counting semaphore**, the count is initialized to the number of resources available – for example, a shared buffer of N blocks will have a semaphore initialized to N.
- If count > 0 it implies that at least one resource is available

- When **N = 1**, the count reduces to just a flag or value that is either 1 or 0. Then the semaphore is called a **binary semaphore**.

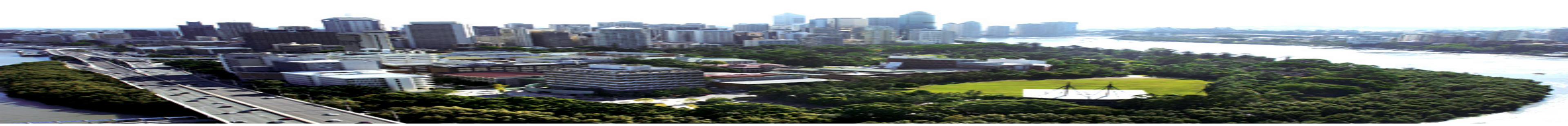


Mutex

- Is a binary semaphore.
- For “mutual exclusion”
- When one resource needs to be shared between threads and only one should enter a critical section at a time
- In ucos-II, the implementation of “mutex” also supports priority inheritance.

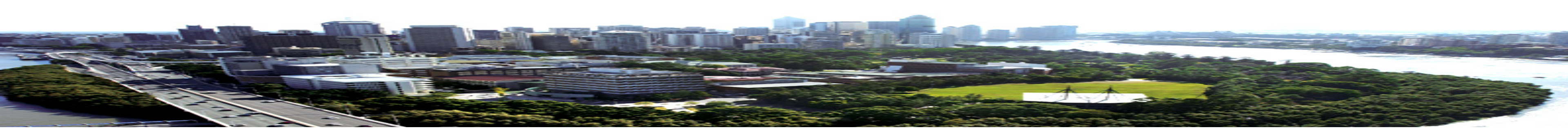
Dynamic C support

- The shared keyword allows atomic updates of multi-byte variables.
- Boolean variables with waitfor() can be used to synchronize threads
- Semaphores and Messages are supported through the RTOS micro-C/OS-II add on.



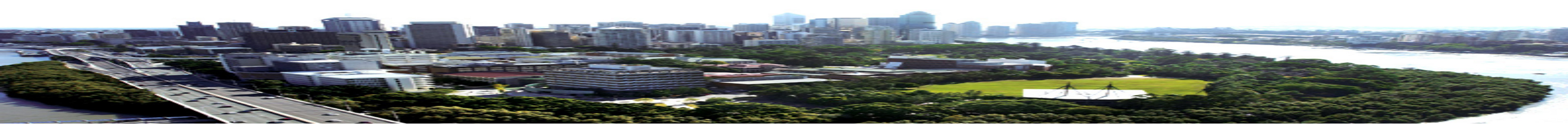
Synchronization in Dynamic C

```
main()
{
char semaphore;                         // THIS IS NOT A TRUE S SEMAPHORE. IT IS ONLY A FLAG VARIABLE – DOES NOT QUEUE TASKS.

initPort();
semaphore=TRUE;

while (1)
        {
        costate turn_on_DS1 always_on // task that turns DS1 LED on
                    {
                    for (;;)
                                    {
                                    waitfor(semaphore);
                                    DS1led(ON);
                                    waitfor(DelayMs(TIME_DS1_ON));
                                    semaphore = FALSE;              // NOTE: atomicity of such instructions is not always guaranteed.
                    } // for
        } // costate

        costate turn_off_DS1 always_on // task that turns DS1 LED off
                    {
                    for (;;)
                                    {
                                    waitfor(!semaphore);
                                    DS1led(OFF);
                                    waitfor(DelayMs(TIME_DS1_OFF));
                                    semaphore=TRUE; /               / NOTE: atomicity of such instructions is not always guaranteed.
                    } // for
        } // costate
} // while
} //main
```

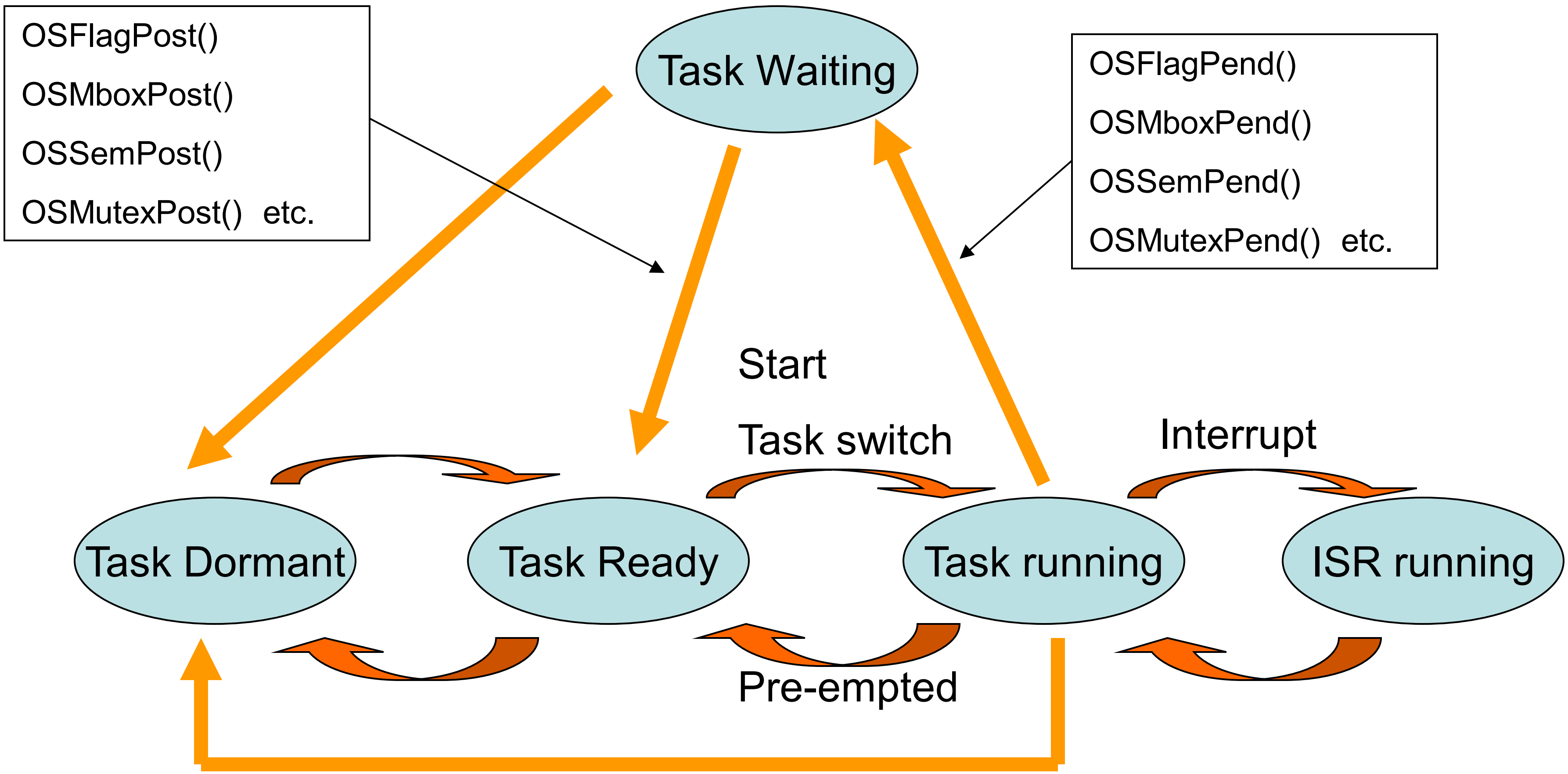


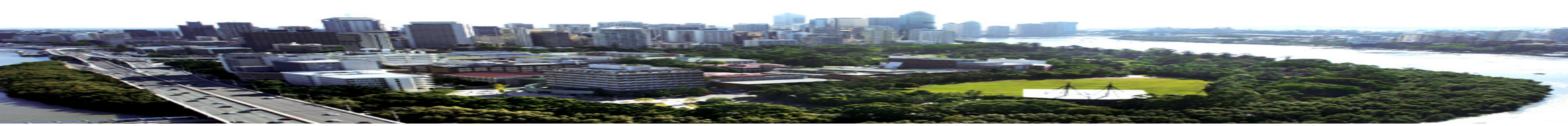
ucos-II support

- Task management
- Time management
- Semaphores
- Mailboxes



Task states in ucos-II





Semaphores in ucos-II

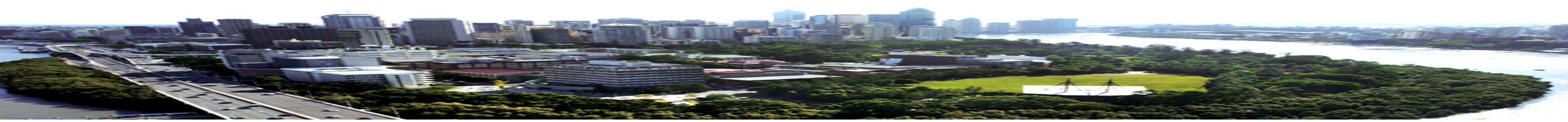
- ucos-II semaphores comprise
  - 16 bit integer holding the count (0 to 65536)
  - List of tasks waiting for the count to be > 0

- Semaphore services are enabled when OS_SEM_EN is set to 1

- A semaphore is created by calling OSSemCreate() and specifying the initial count

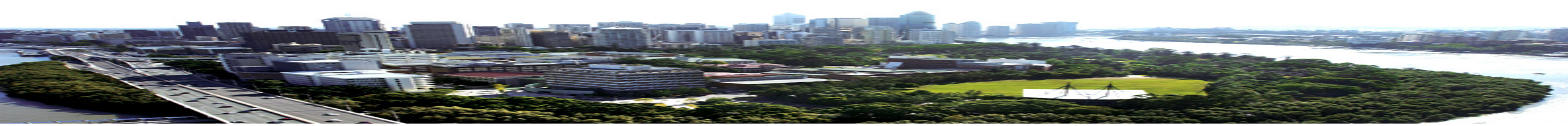
- OSSemPend() allows specification of a timeout in case the signal from the semaphore fails to arrive.

  It does not make sense to call OSSemPend() from an ISR because an ISR cannot be made to wait. OSSemAccept() will obtain a semaphore without putting the task to sleep when it is not available.



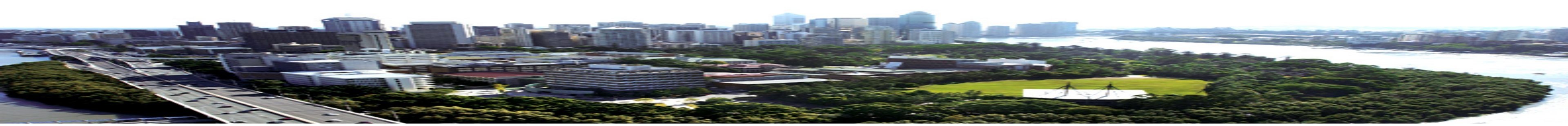
Semaphore usage – code fragments

```
void main (void)
{
   ClearScreen();
   OSInit();                              // Initialize uC/OS-II
   RandomSem = OSSemCreate(1);    // Random number semaphore
   PrintSem = OSSemCreate(1);
   OSTaskCreate(TaskStart, (void *)0, TASK_STK_SIZE, 10);
   OSStart();                               // Start multitasking
}


nodebug void Task (void *data)
{
         auto UBYTE x;
         auto UBYTE y;
          auto UBYTE err;
          auto UBYTE num[2];

   for (;;) {
      OSSemPend(RandomSem, 0, &err);          // Acquire semaphore to perform random numbers
      x = (int)(rand() * 5);                 // Find X position where task number will appear
      y = (int)(rand() * 5);                 // Find Y position where task number will appear
      OSSemPost(RandomSem);                  // Release semaphore

      sprintf(num, "%c", *((char *)data));
                     DispStr(x + 39, y + 5, num);                        // Display the task number on the screen
                     OSTimeDly(25);
   }
}
```



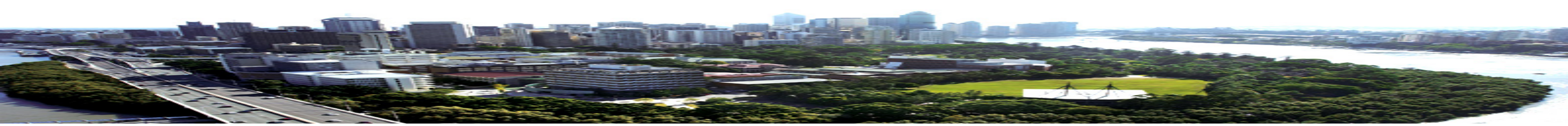
Explanation

- rand() is not re-entrant
- It cannot be interrupted and resumed later – otherwise there will be loss of data. Arises from use of global variables
- It must be protected by a semaphore, such that no other thread will call it while one thread is using it.



Demonstration

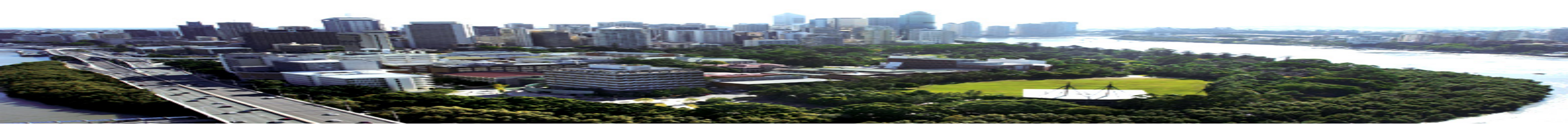
- Shared data? Re-entrant ?
  - Pre-emptive multitasking – OK with semaphores



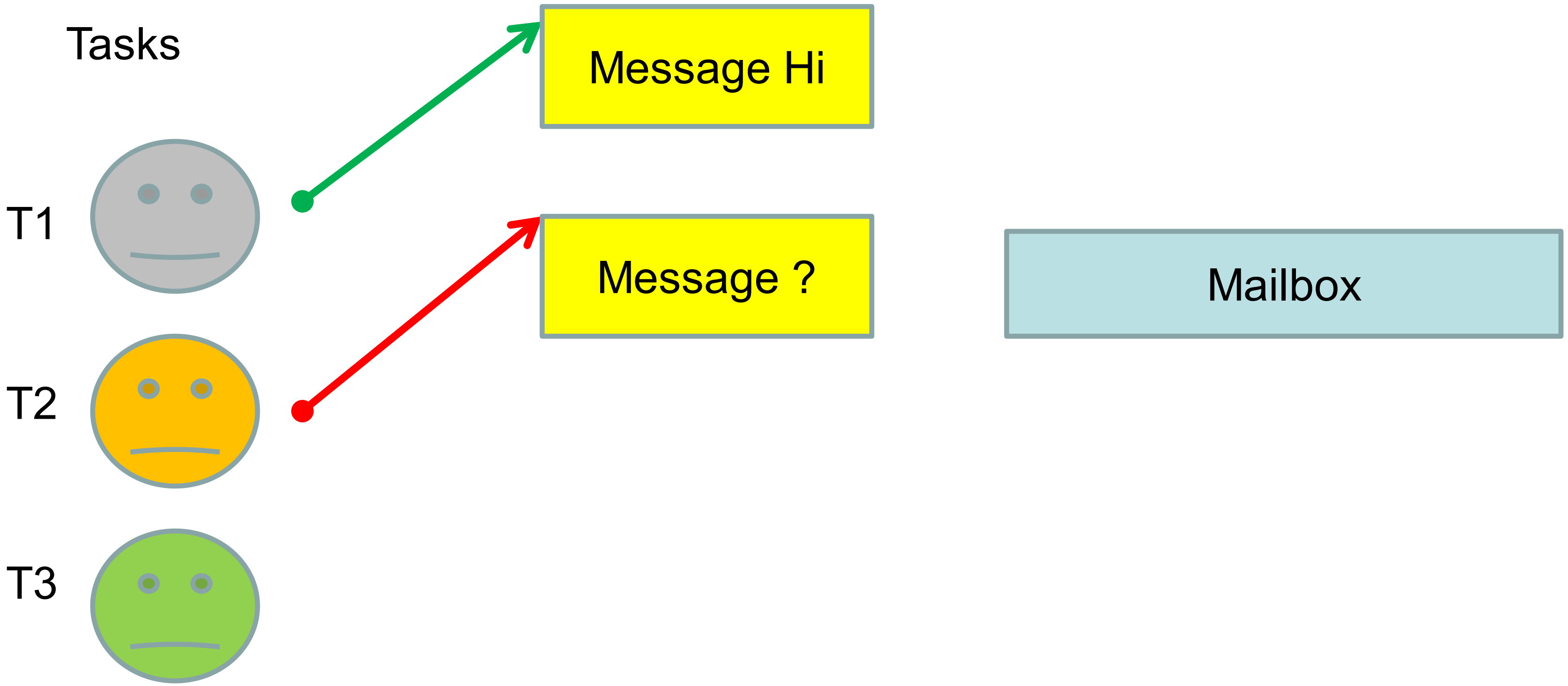
Mailboxes

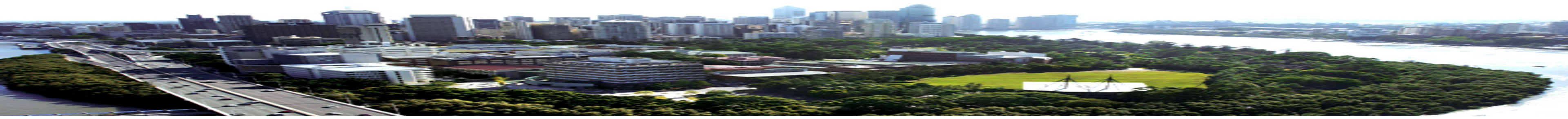
- A message mailbox (or mailbox) in ucos-II is an object that allows a task or ISR to send a pointer sized variable to another task.

- This pointer should point to some application specific “message” data structure.

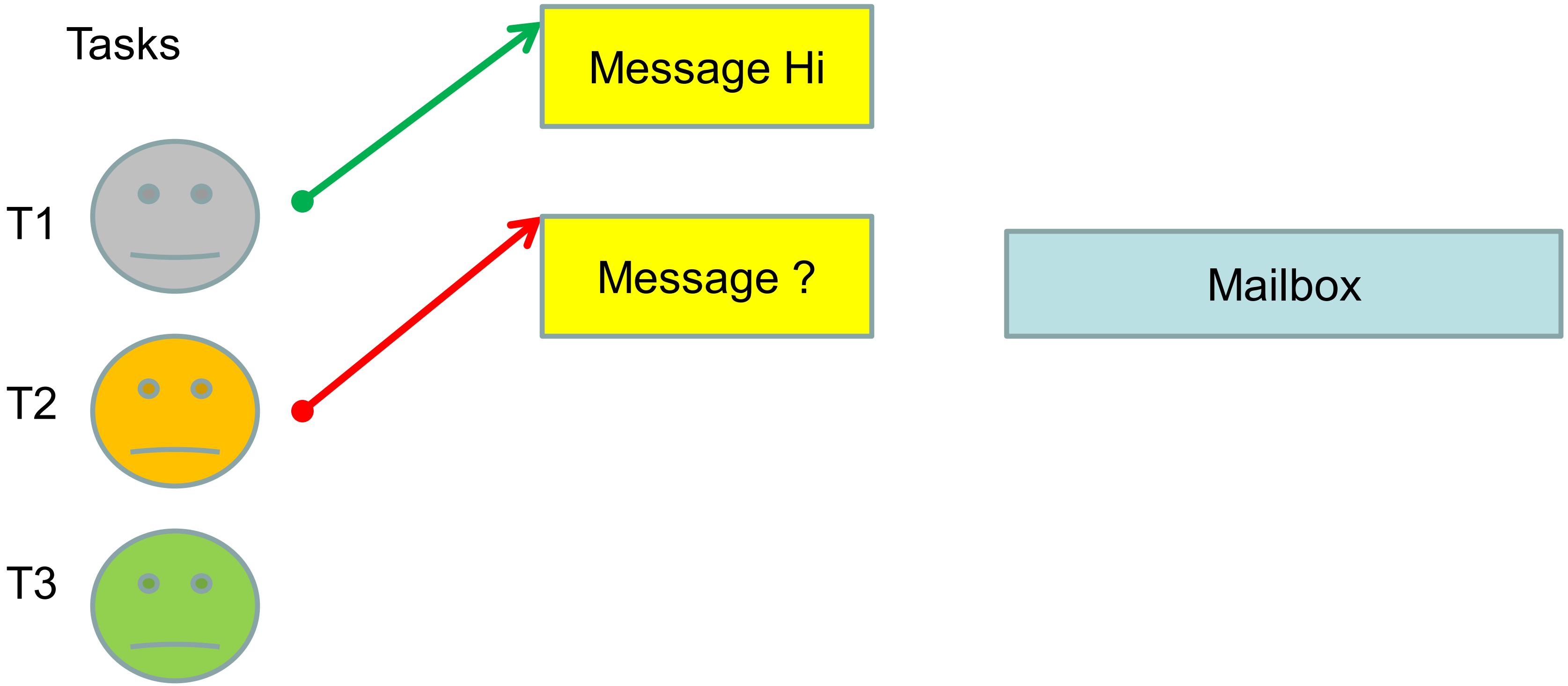


Mailbox

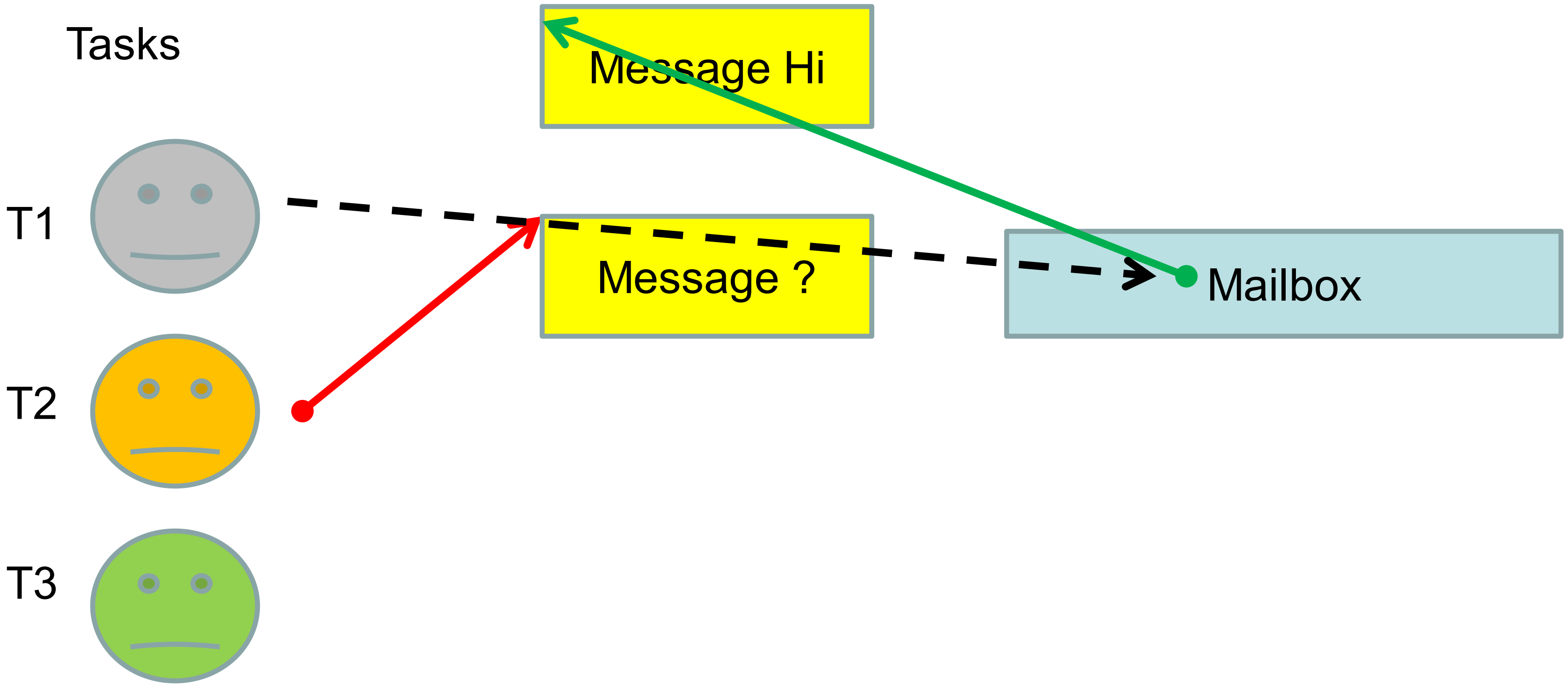




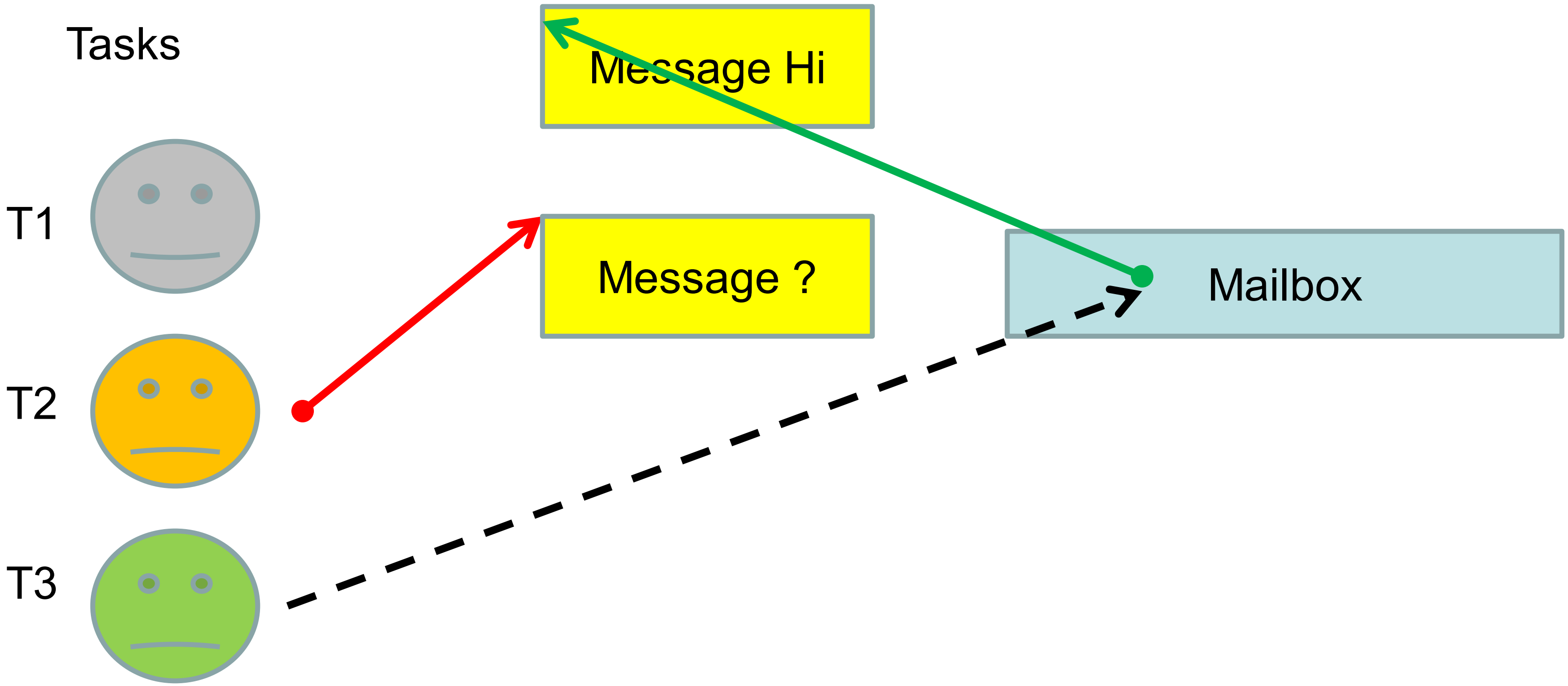
Mailbox



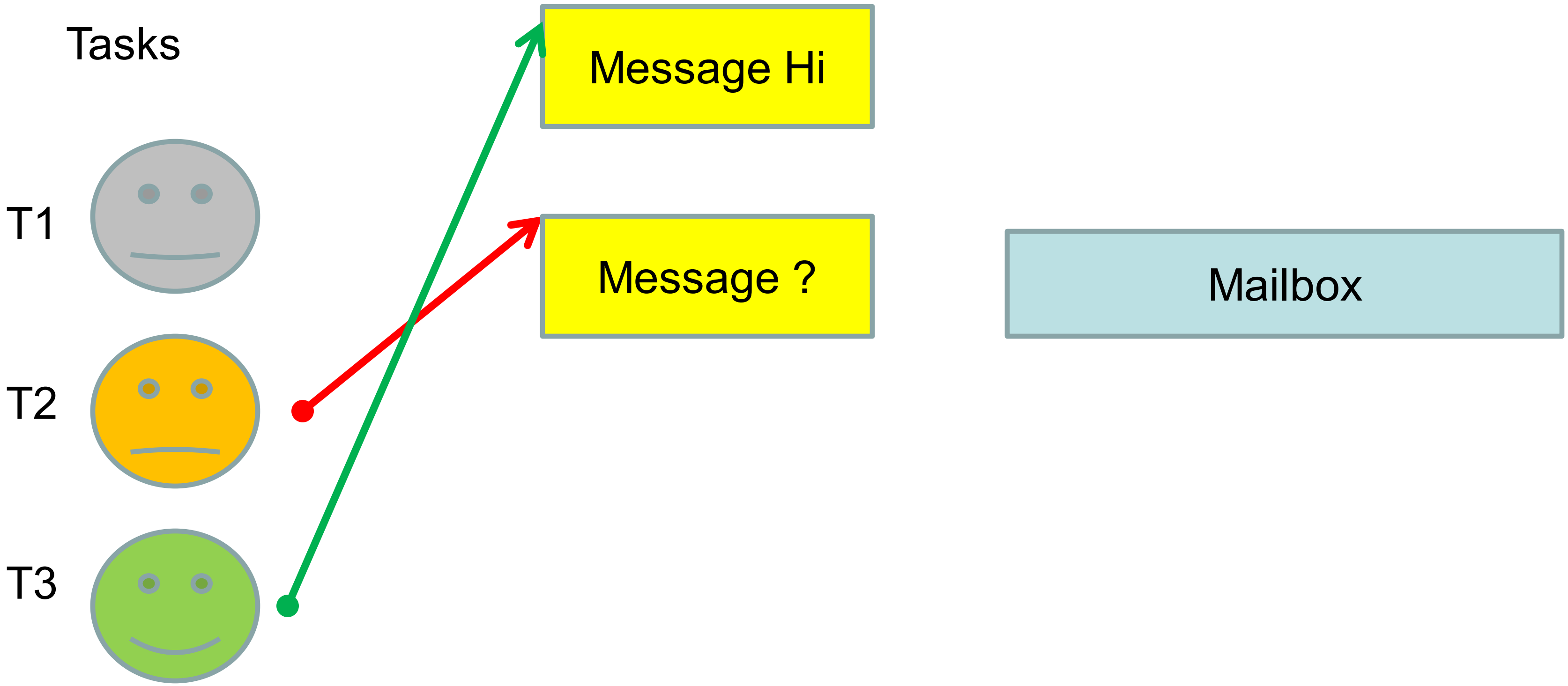
Mailbox T1 Posts



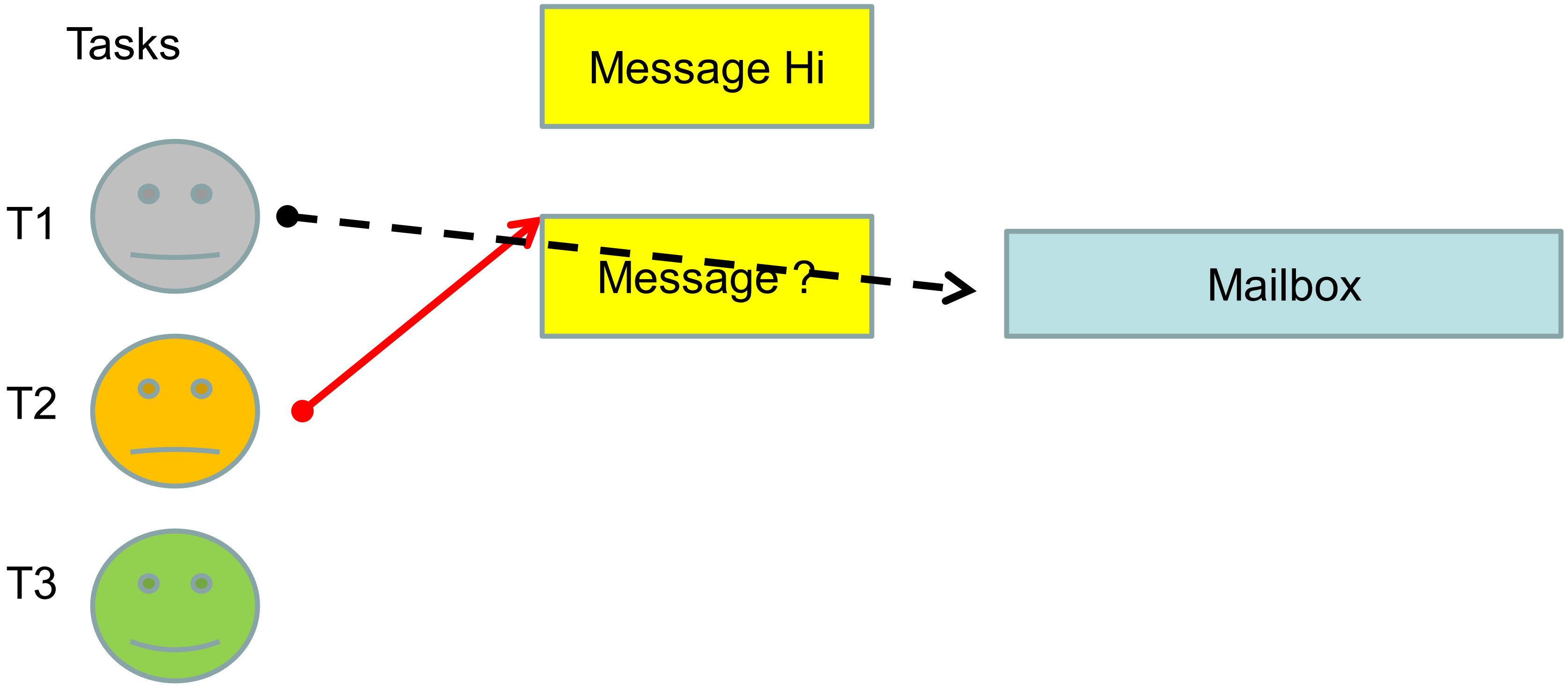
Mailbox T3 attempts Pend

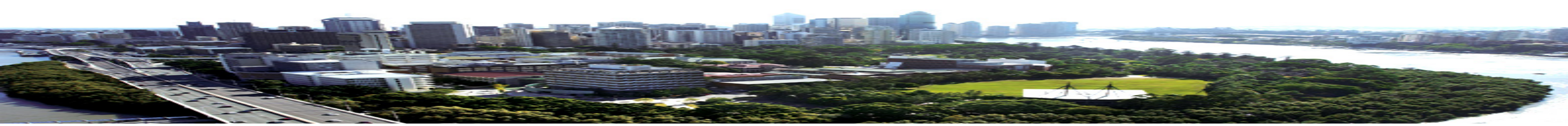


Mailbox T3 Pends (reads)

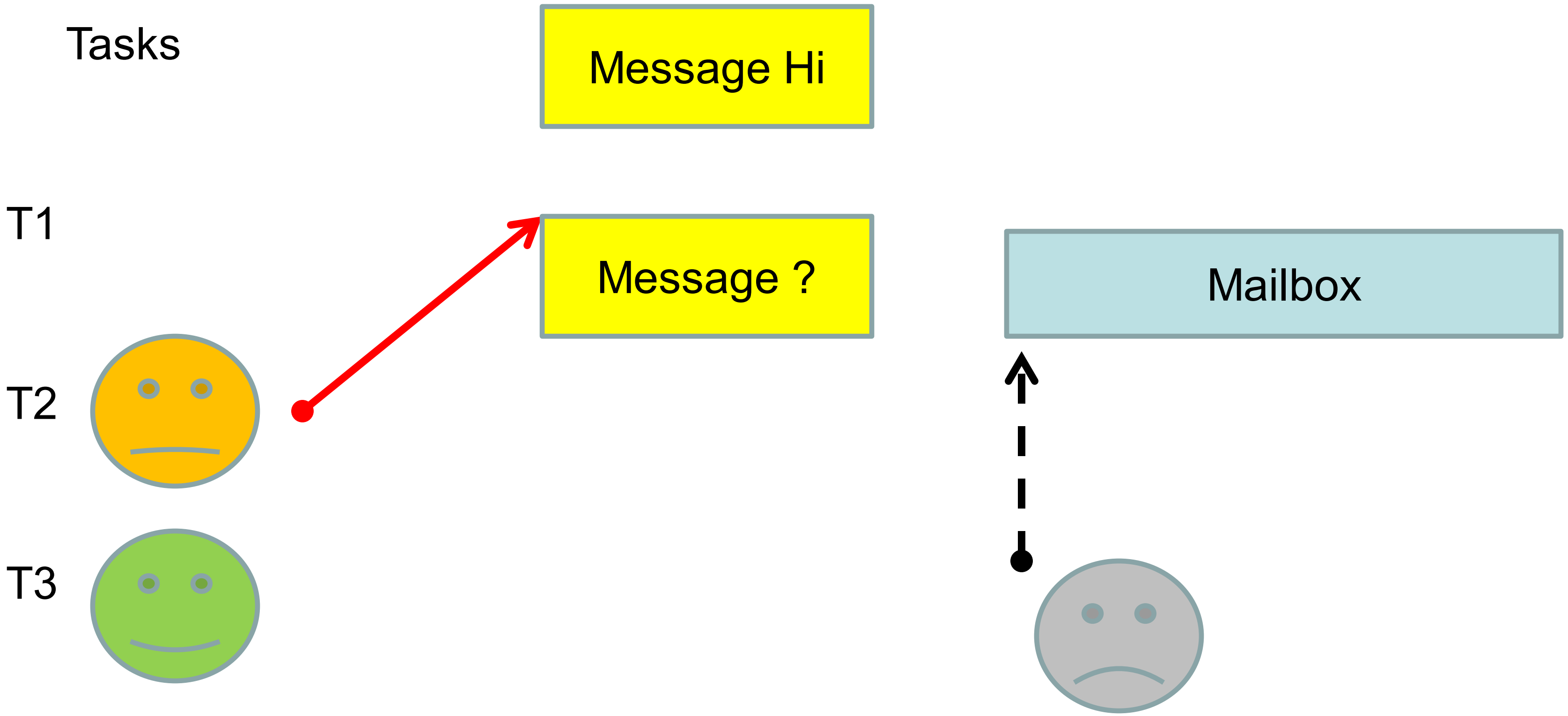


Mailbox T1 attempts Pend

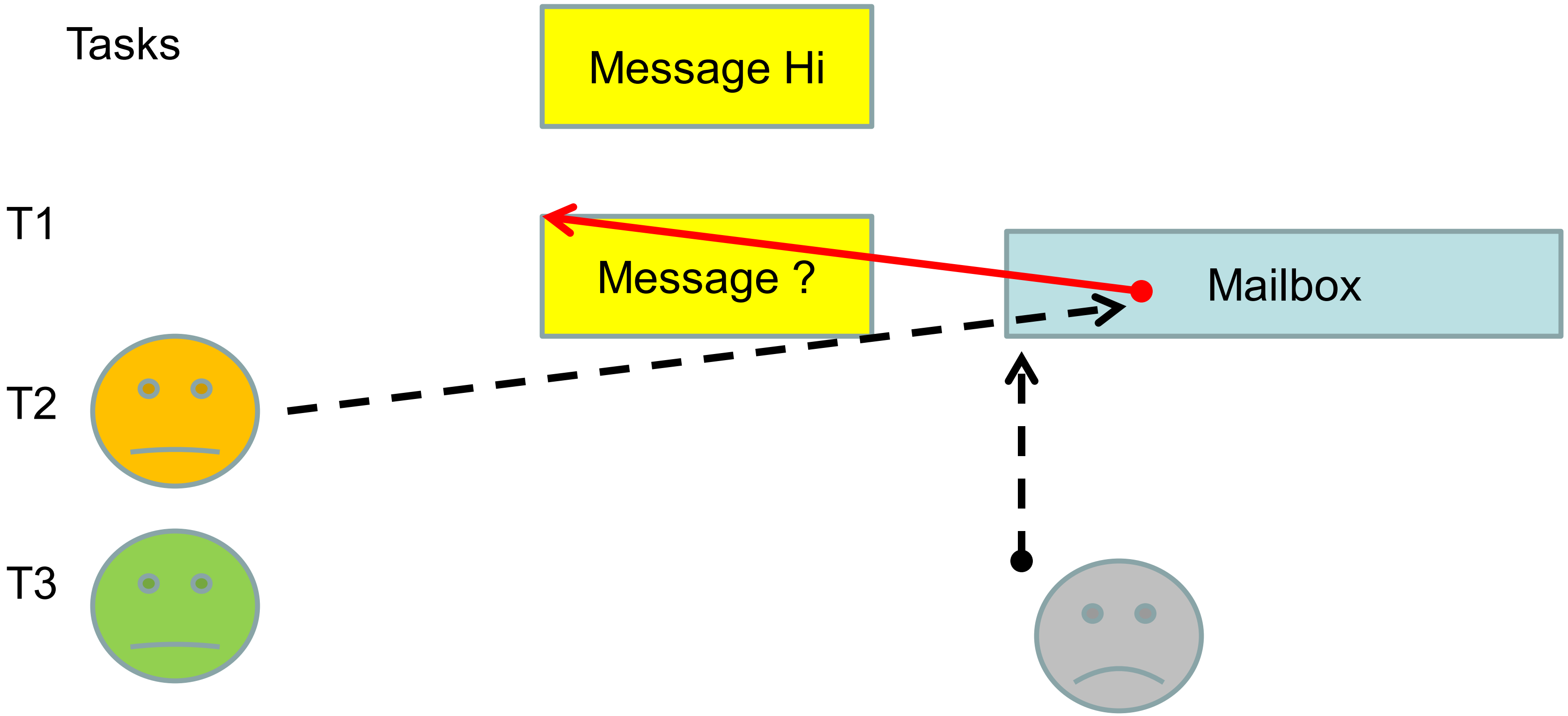




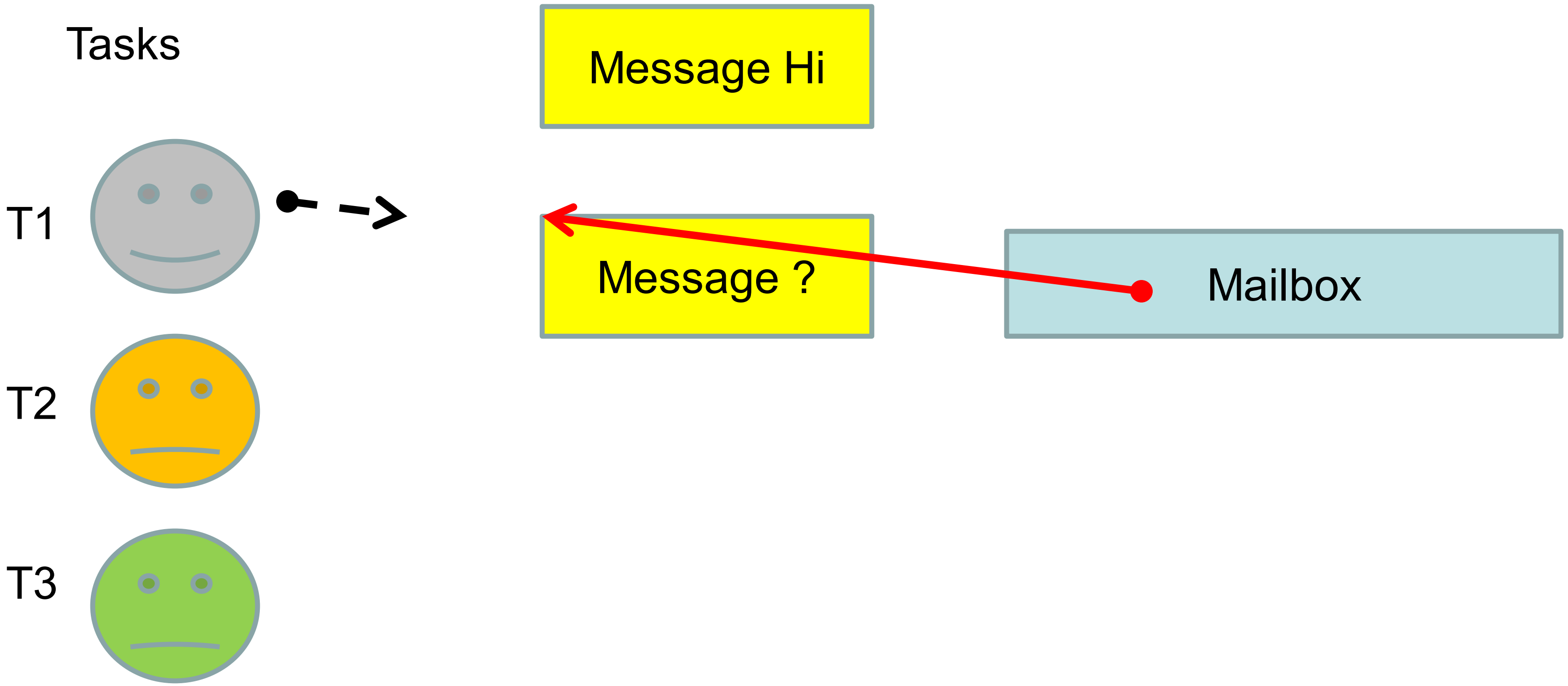
Mailbox T1 Pend fails



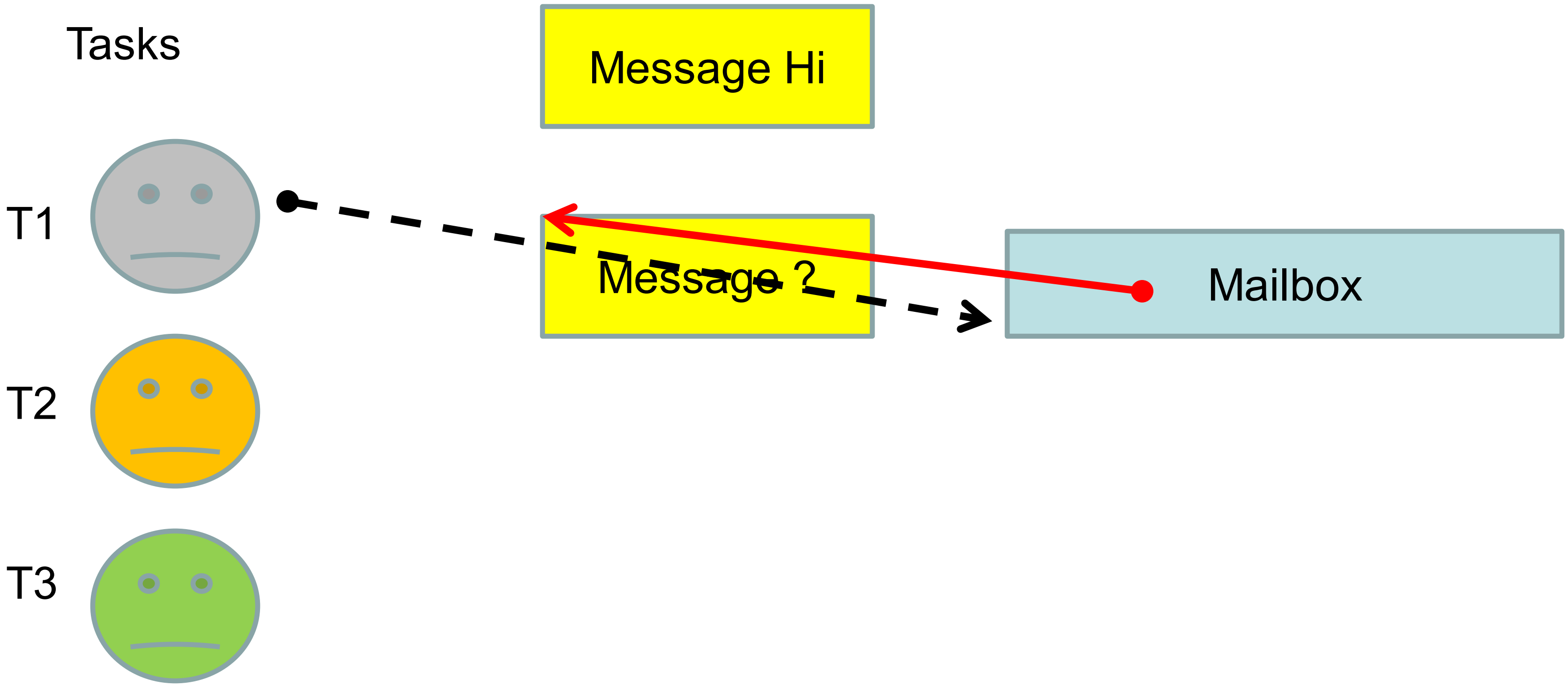
Mailbox T2 Posts

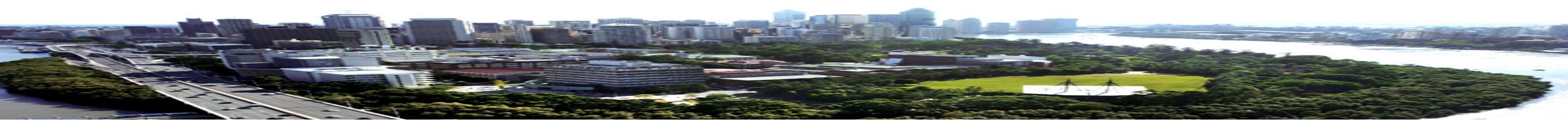


Mailbox T2 Posts

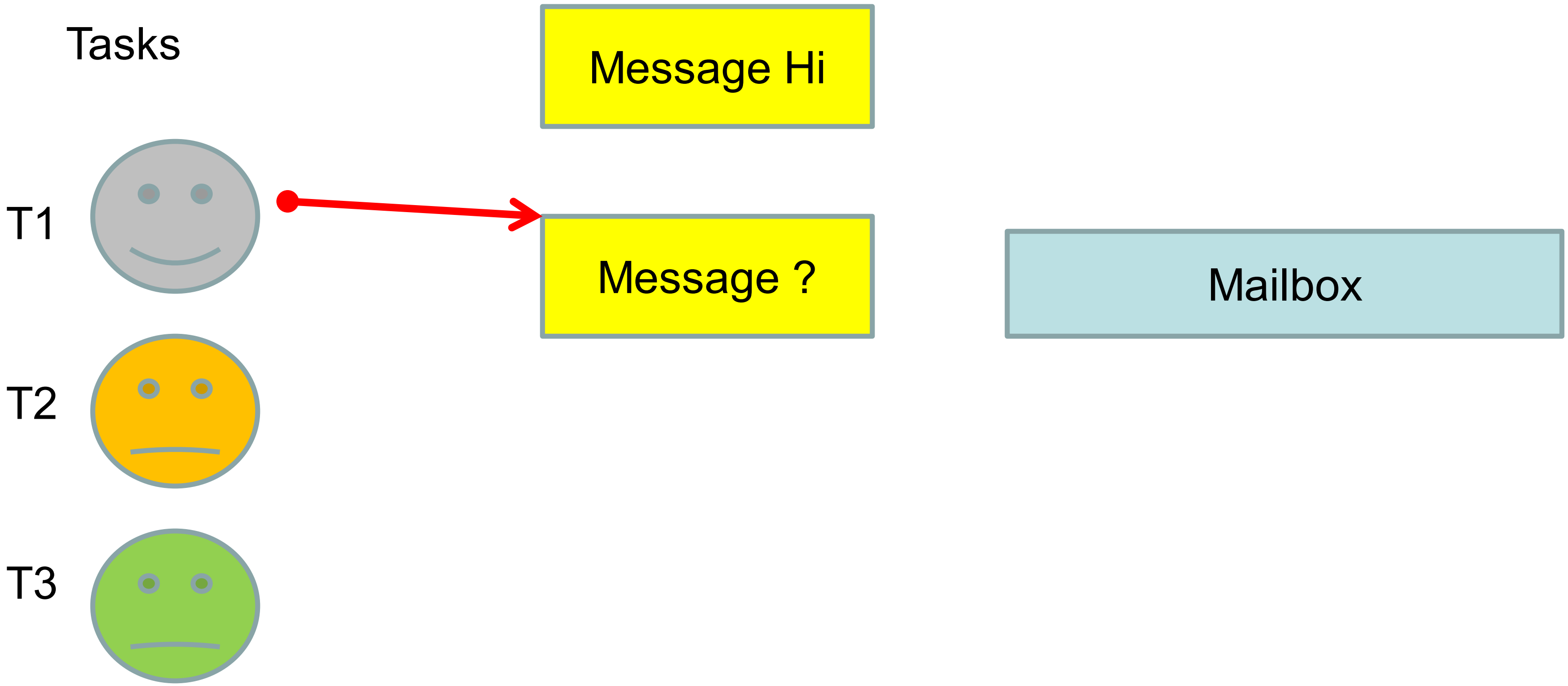


Mailbox T1 attempts Pend





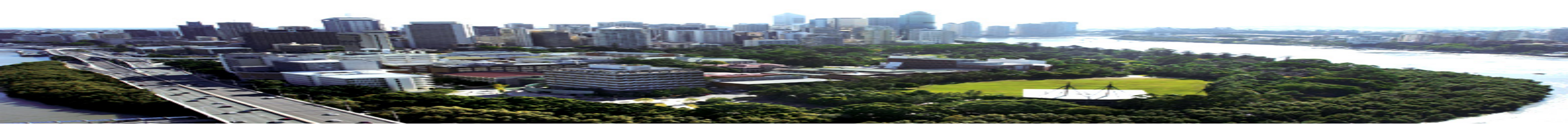
Mailbox T1 Pends (reads)



Mailboxes

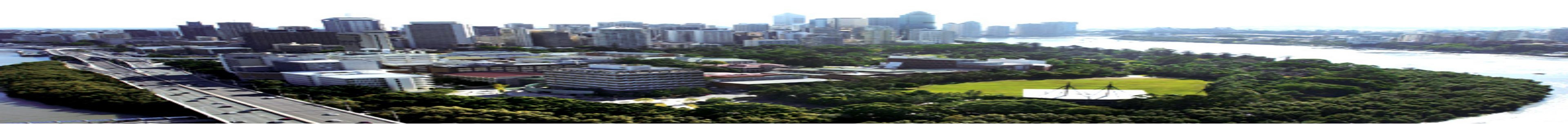
- Waiting for a message with OSMboxPend() is similar to that for a semaphore (blocking). It returns to the caller with the pointer that was in the mailbox, replacing it with a NULL.

- Sending a message with OSMboxPost() involves specification of a pointer to an event and a pointer to a message.



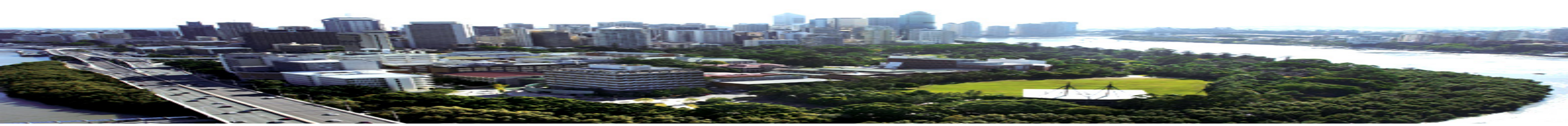
Mailbox usage – code fragments

```
#define OS_MBOX_EN          1       // Enable mailboxes
#define OS_MBOX_POST_EN     1       // Enable Post

OS_EVENT    *AckMbox;               /* Message mailboxes for Tasks #4 and #5    */
OS_EVENT    *TxMbox;
```



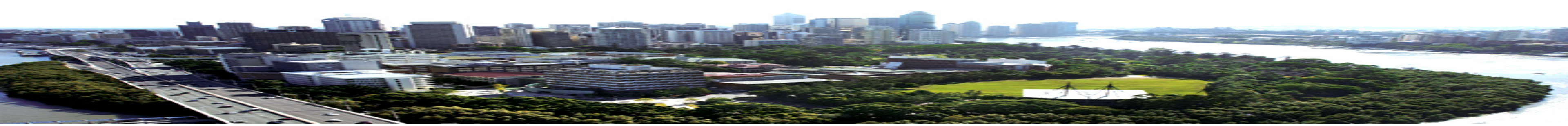
Mailbox usage – Sending task

```
void  Task4 (void *data)
{
    static char txmsg;
        auto INT8U  err, i;


    data  = data;
    txmsg = 'A';
    for (;;) {
        while (txmsg <= 'Z') {
            OSMboxPost(TxMbox, (void *)&txmsg);      /* Send message to Task #5  */
            OSMboxPend(AckMbox, 0, &err);             /* Wait for acknowledgement from Task #5  */
            txmsg++;                                  /* Next message to send     */
        }
        txmsg = 'A';                                  /* Start new series of messages  */
    }
}
```



Mailbox usage – receiving task

```
void  Task5 (void *data)
{
    auto char  *rxmsg;
    auto INT8U  err;
    auto char  buf[2];

    data = data;
    for (;;) {
        rxmsg = (char *)OSMboxPend(TxMbox, 0, &err);        /* Wait for message from Task #4 */
                    sprintf(buf, "%c", *rxmsg);
        DispStr(70, 17, buf);
        OSTimeDlyHMSM(0, 0, 1, 0);                          /* Wait 1 second                 */
        OSMboxPost(AckMbox, (void *)1);                     /* Acknowledge reception of msg  */
    }
}
```



Demonstration

- Program ucosdemo2.c from Samples

  Here Tasks 4 and 5 communicate using a mailbox. We will take a look at this.