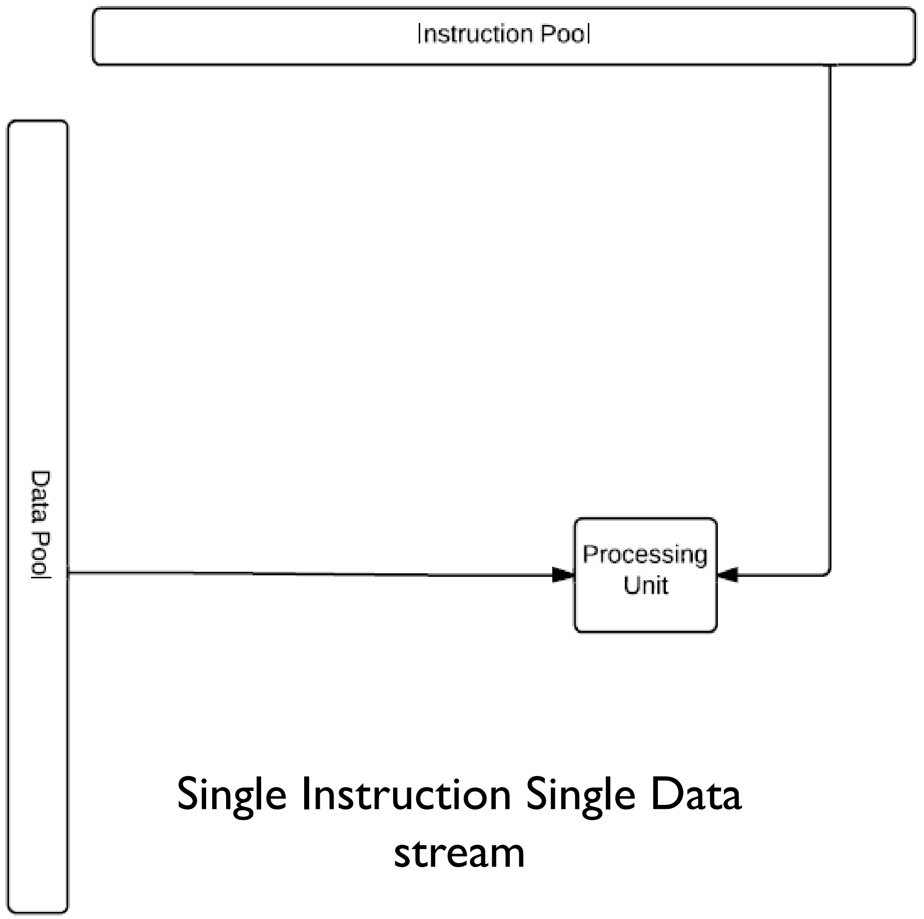
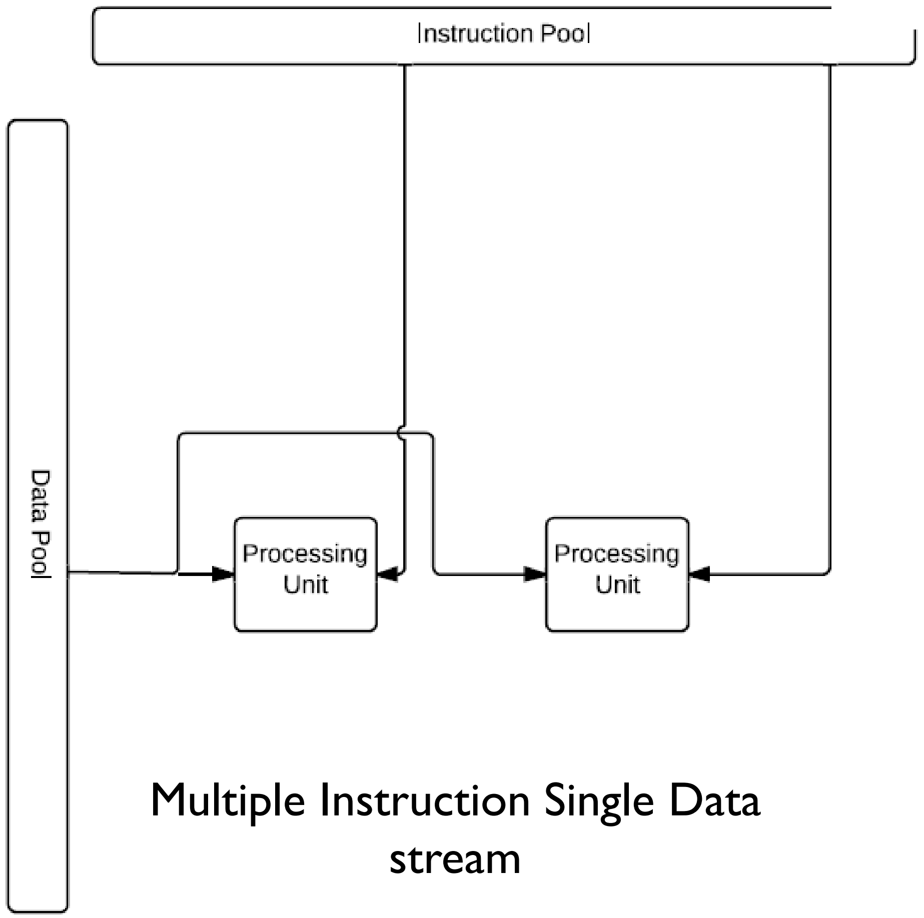
Concurrency

Ravi Suppiah
Lecturer, NUS SoC

Multiprocessing

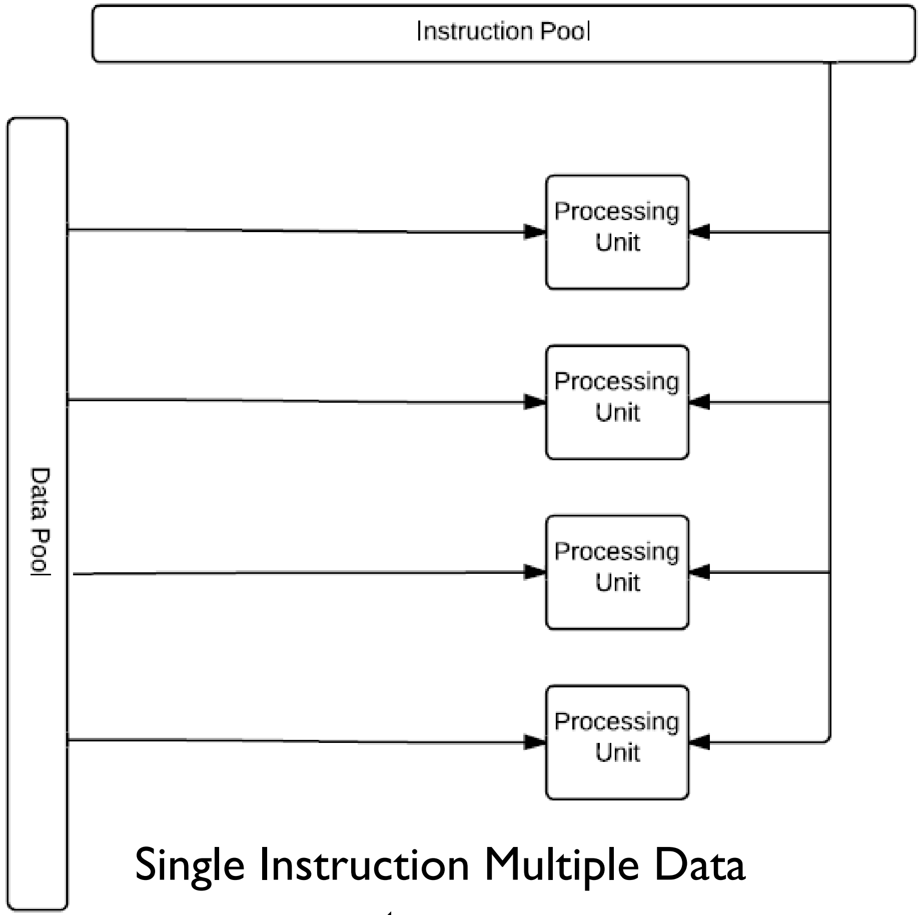


Single Instruction Single Data stream

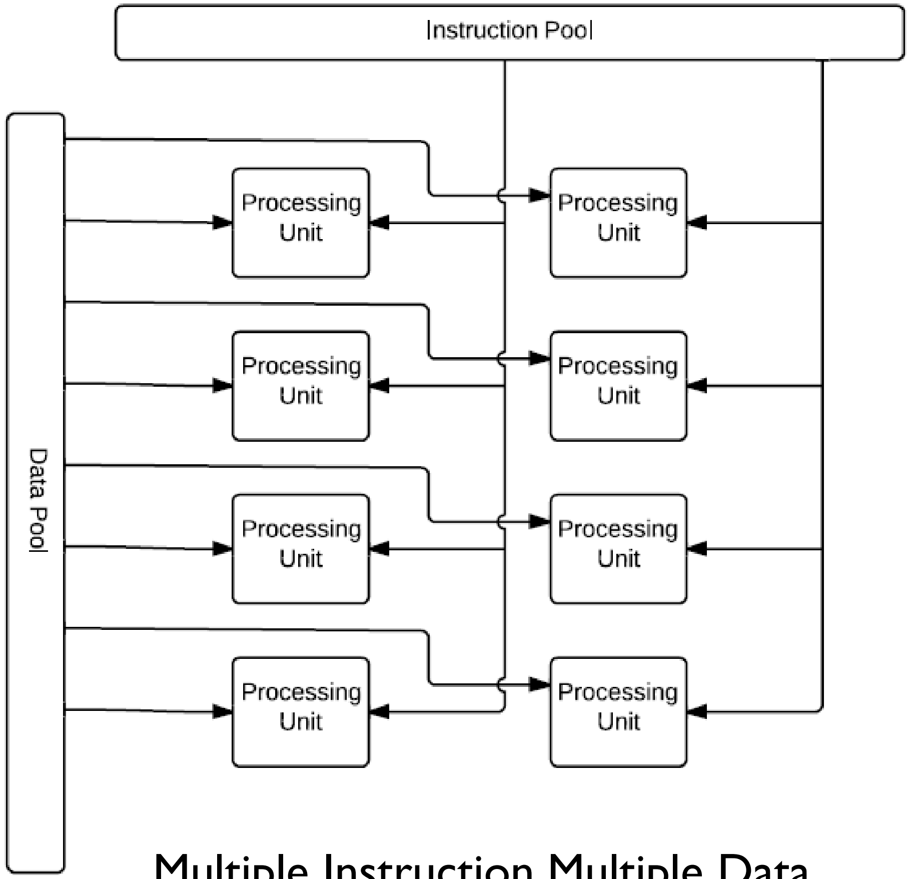


Multiple Instruction Single Data stream

Multiprocessing



Single Instruction Multiple Data streams



Multiple Instruction Multiple Data streams

Multiprocessing

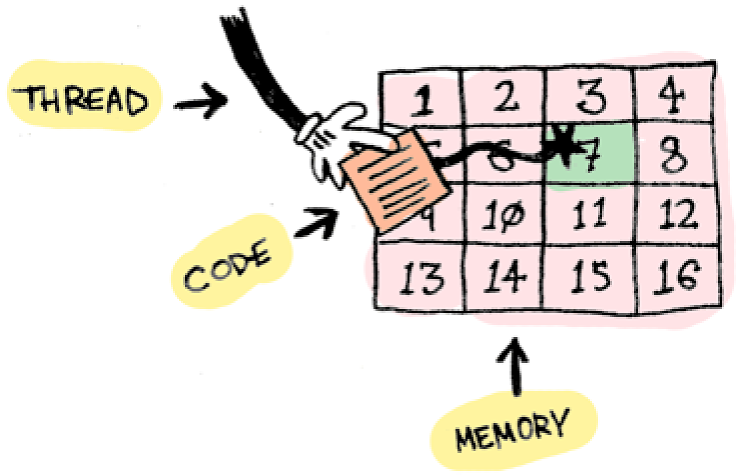
- SISD
  - Sequential without parallelism
  - Concurrency at hardware level
- SIMD
  - A single program on many processing units e.g. Graphics Processing Units (GPUs), vector processors
- MISD
  - Rarely used
- MIMD
  - Many processors executing different instructions on different data
  - Distributed systems
  - Shared memory

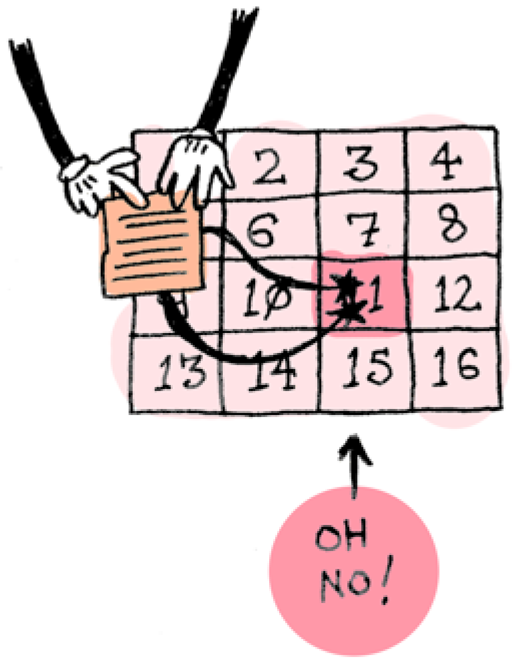
Operating Systems on Single Processors

- OSs on Single Processors
  - Easier Memory / Data Management
  - Consistent Scheduling Policy

Single-Threaded versus Multi-threaded

- Communication is not straightforward with multiple parallel/concurrent threads
- The threads might over-write a memory location





Race Condition

- In concurrent systems, when a set of processes access shared resources to carry out a computation and the results of the computation depend on the exact way the processes interleave, there is a race condition

- Shared resources can refer to
  - Shared variables residing in memory
  - Shared objects such as files and devices

- Race conditions undermines the correctness of any concurrent system and should be eliminated

Race Condition

```
int k;                      //global variable
void inck (void)            // inck ( ) called by multiple processes
{
    k = k + 1;
}
void funcA(void)
{
    while(1){
        inck();
        // do other things}
}
void funcB(void)
{
    while(1){
        inck();
        // do other things}
}
osThreadNew(funcA, NULL, NULL);
osThreadNew(funcB, NULL, NULL);
```

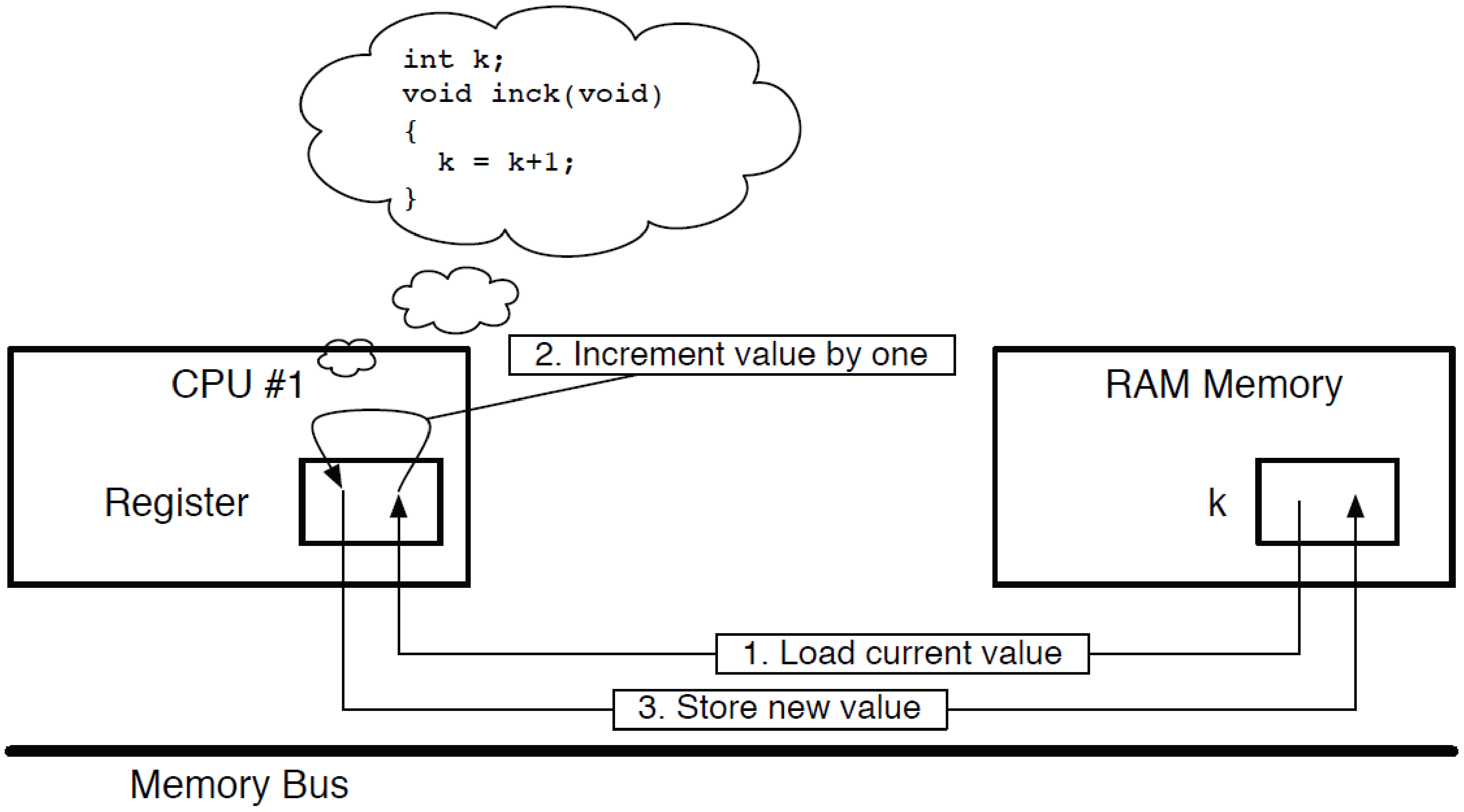
The Issue: Non-atomicity of operation

- Real-world CPU cannot execute `k = k + 1` statement in a single, invisible step (atomically) when compiled into ordinary assembly language instructions in a RISC system
- No practical consequence if code is executed sequentially
- Can have serious consequences in concurrent systems

```
Assembly language code for k = k +1
Assume variable k is in memory location M

load  r, M      // Load k from memory location M to processor register r
add   r, r, #1  // Add 1 to the value in register r
store r, M      // store the incremented value back in memory location M
```

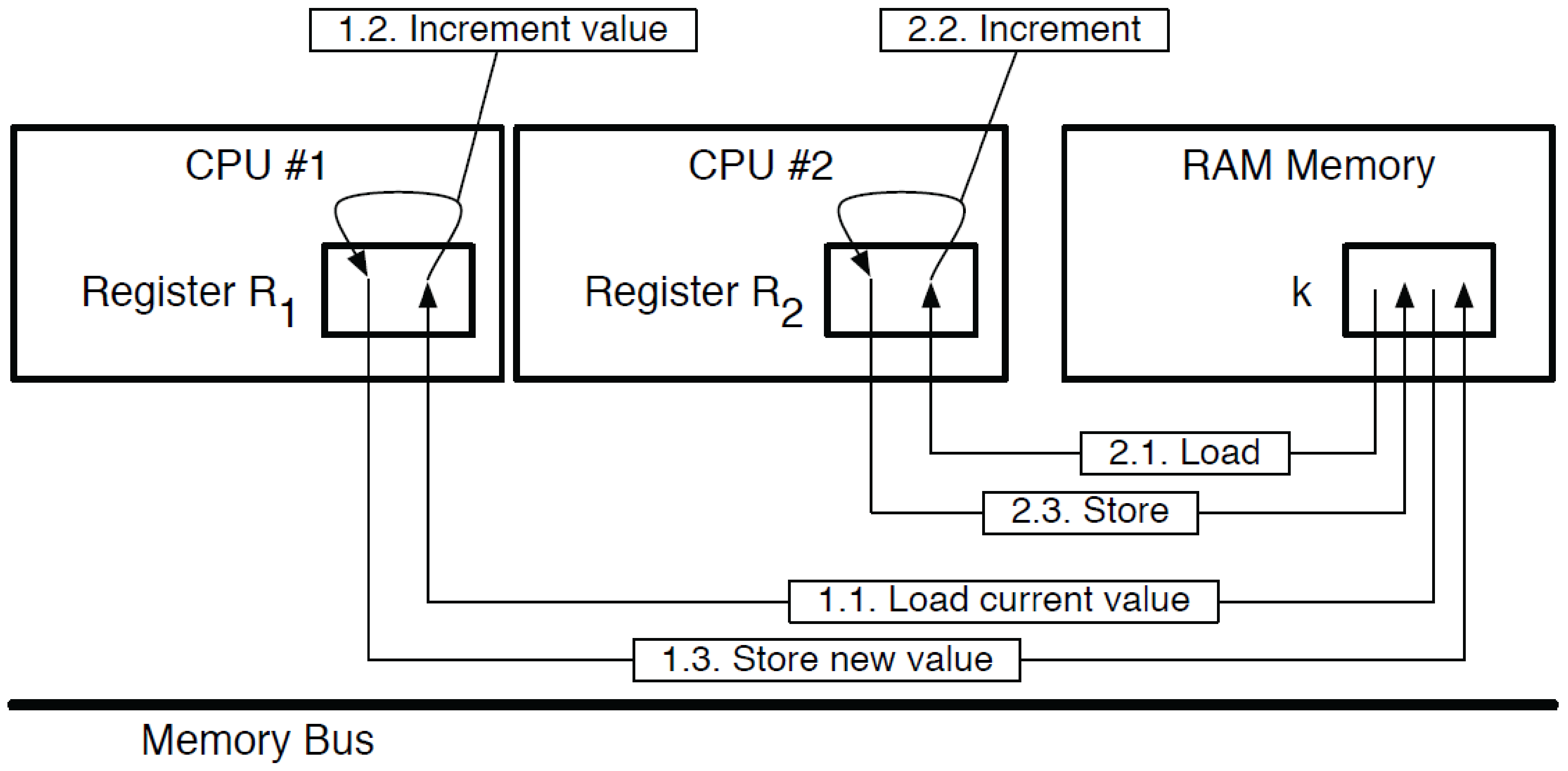
Non-atomic Execution of Increment



```
int k;
void inck(void)
{
  k = k+1;
}
```

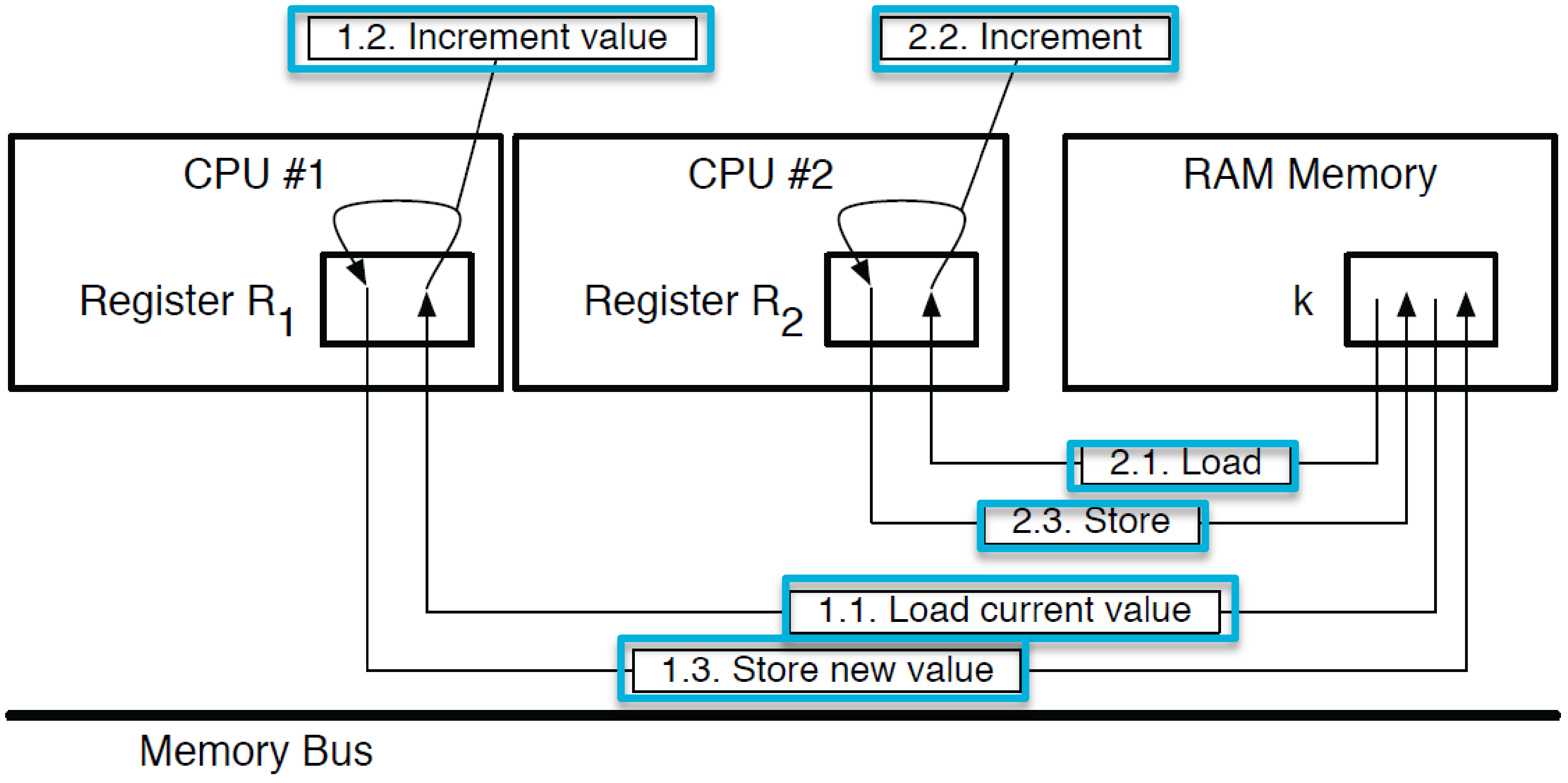
Multiple processes doing Increment

- We first assume parallel execution on two CPU cores
- Initially `k = 0;` expectation `k = 2`



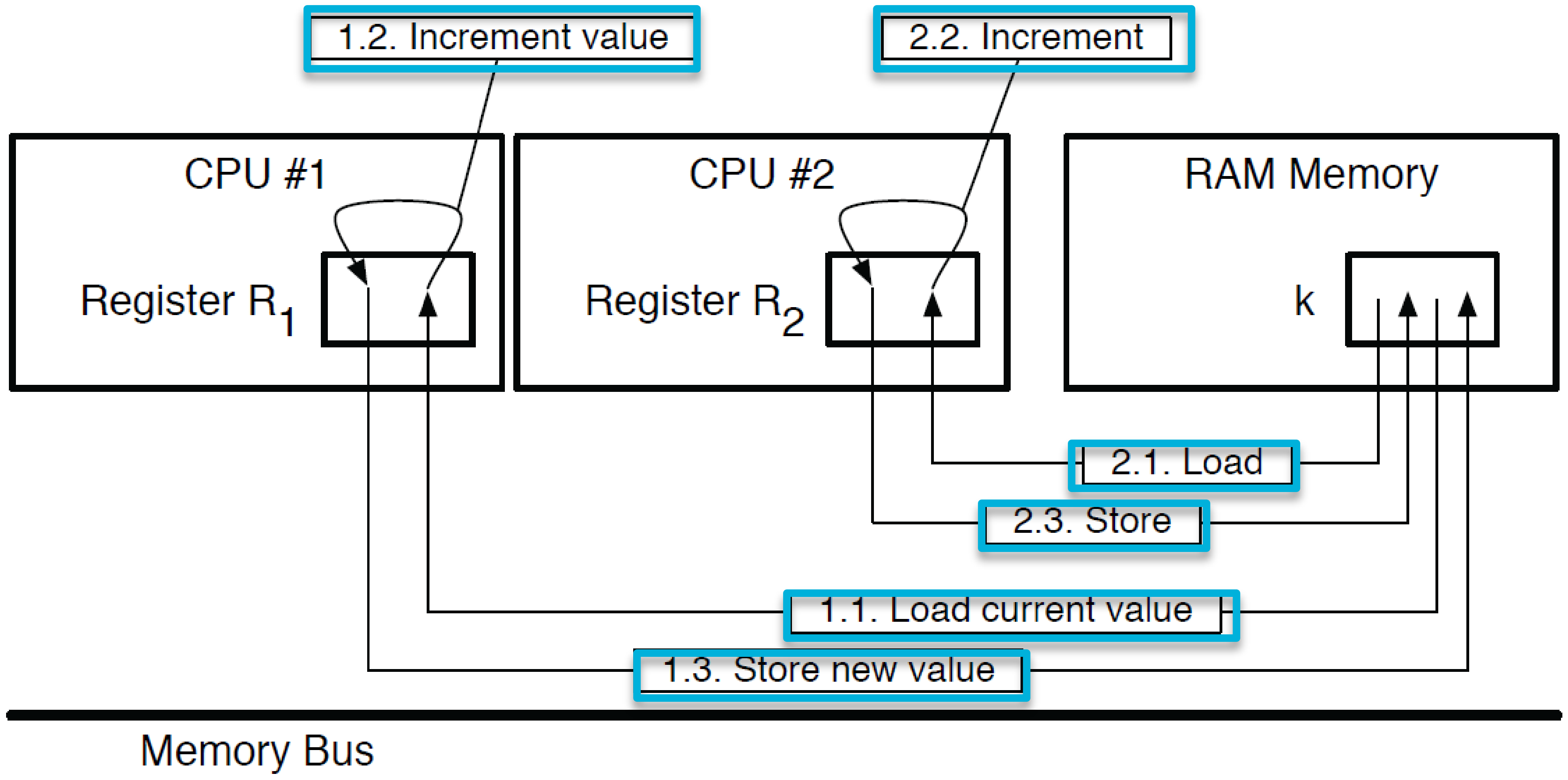
Expected Result

- Execution Sequence: 1.1 → 1.2 → 1.3 → 2.1 → 2.2 → 2.3 OR
  2.1 → 2.2 → 2.3 → 1.1 → 1.2 → 1.3



Unexpected Result

- Execution Sequence: 1.1 → 1.2 → 2.1 → 2.2 → 2.3 → 1.3 OR
  1.1 → 2.1 → 2.2 → 2.3 → 1.2 → 1.3



Lessons Learnt

- Taking a correct piece of sequential code and using it for concurrent programming may not work as expected
- Results are incorrect only sometime
  - value and correctness of result depend on how the elemental steps of the update performed by one process interleave with the steps performed by the other
  - Completely non-deterministic as depends on the precise timing relationship between the processes
  - Hard to find and fix bugs
- How to eliminate race conditions?

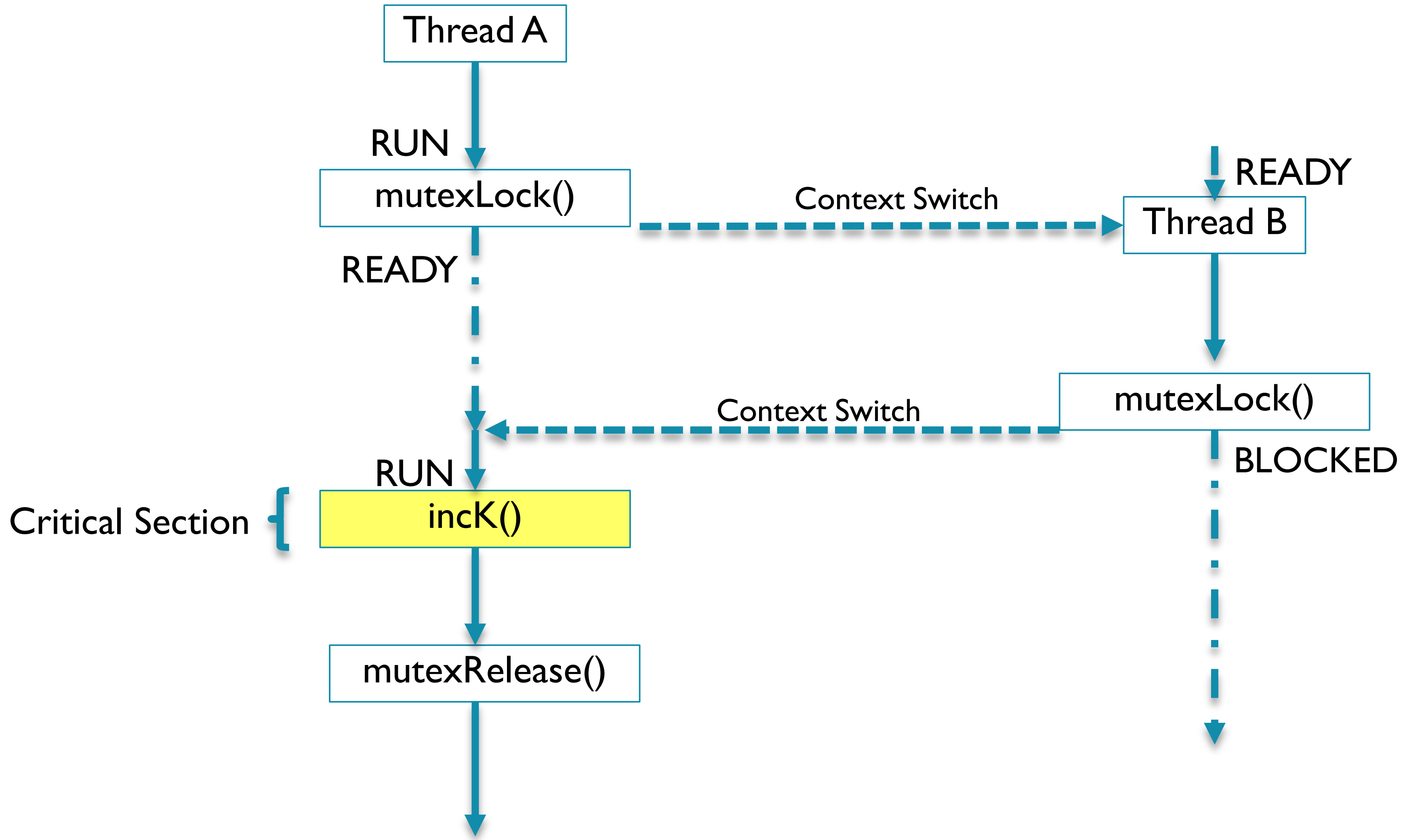
Critical Section

- Sometimes a process executes a region of code that makes access to shared resources
- Regions of code accessing shared resources may lead to race conditions
- Regions of code accessing shared resources are called critical regions or critical section

Mutual Exclusion

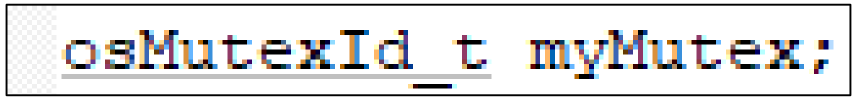
- Mutually exclusive resources such as *shared* variables, or printers (a hardware device)
- Mutex controls access to *shared* resources, you get the expected value as if processed by a single-threaded system
- Enforced to ensure only one thread of execution can have access at any time
- Critical Section – the code that accesses the mutually exclusive resource
- Requirements for good mutex:
  - Enforced
  - The only reason a request to a critical section is rejected/delayed is that another process is accessing already
  - Process can only access the critical section for a finite time
  - No deadlock and starvation

Mutual Exclusion



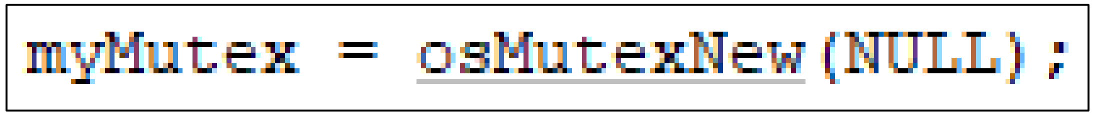
Mutex Routines

- First Step is to declare a Mutex ID Globally.



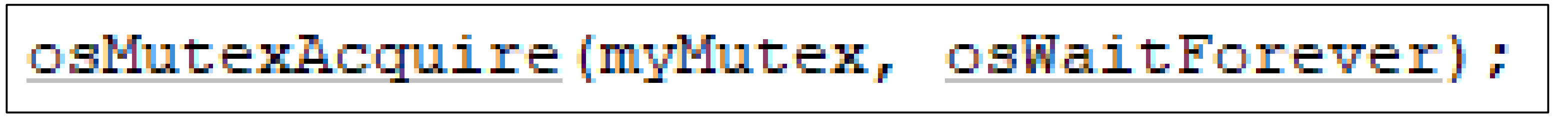
```
osMutexId_t myMutex;
```

- Second Step is to create a new Mutex with the ID.

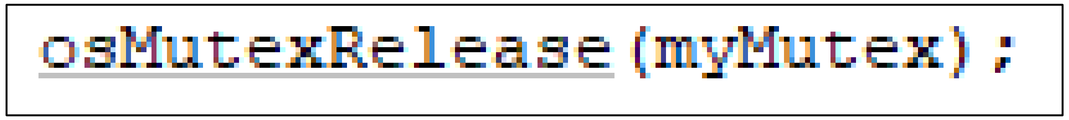


```
myMutex = osMutexNew(NULL);
```

- We can Acquire / Release the Mutex using the following calls.



```
osMutexAcquire(myMutex, osWaitForever);
```



```
osMutexRelease(myMutex);
```

Semaphore

- A semaphore is a container of a number of tokens.
- Acquire a token first, then access a resource.
- To finish with the resource, return the token.
- Also used to synchronize tasks or protect variables and resources.
- Very sophisticated and comprehensive ways of using semaphores.

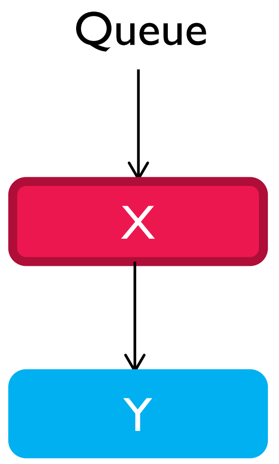
Atomic Primitives for Semaphore

- P(s) --- before accessing shared object
  - Check whether the value of semaphore s is (strictly) greater than 0
  - If yes, decrement the value by one and return without blocking
  - If not, (a) put the calling process into the queue associated with the semaphore, (b) blocks the process by moving it into blocked state

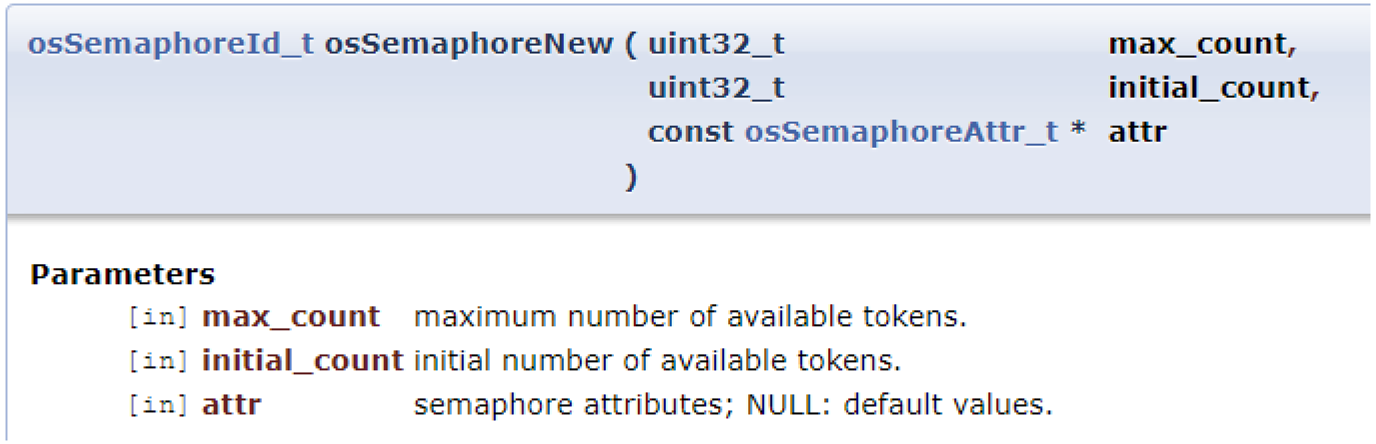
- V(s) --- after accessing shared object
  - Check whether the queue associated with semaphore (s) is empty
  - If yes, increment the value of the semaphore by 1
  - If not, pick one of the blocked processes in the queue and make it ready for execution by moving it into the ready state

Semaphore Example

- S = 3 // 3 resources
- P(S) → S = 2 // use resource
- P(S) → S =1 // use resource
- V(S) → S =2 // done using resource
- P(S) → S = 1 // use resource
- P(S) → S = 0 // use resource
- P(S) → S= 0; process X blocked and put in queue
- P(S) → S= 0; process Y blocked and put in queue
- V(S) → S = 0 // done using resource; unblock process X
- V(S) → S =0 // done using resource; unblock process Y
- V(S) → S = 1 // done using resource
- V(S) → S = 2 // done using resource
- V(S) → S = 3 // done using resource



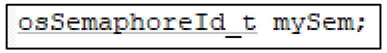
Semaphore Routines



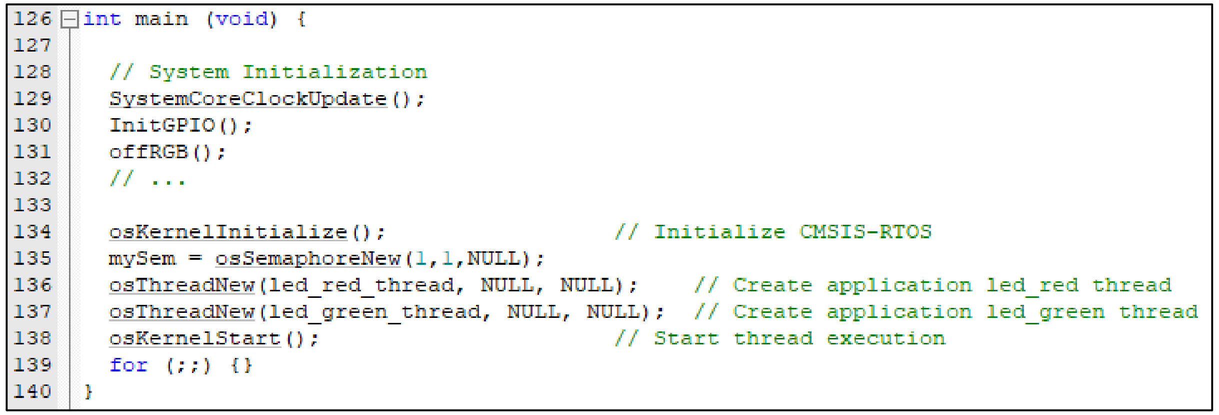
```
osSemaphoreId_t osSemaphoreNew ( uint32_t                     max_count,
                                 uint32_t                     initial_count,
                                 const osSemaphoreAttr_t *    attr
                               )
```

Parameters

[in]	**max_count**	maximum number of available tokens.
[in]	**initial_count**	initial number of available tokens.
[in]	**attr**	semaphore attributes; NULL: default values.



```
osSemaphoreId_t mySem;
```

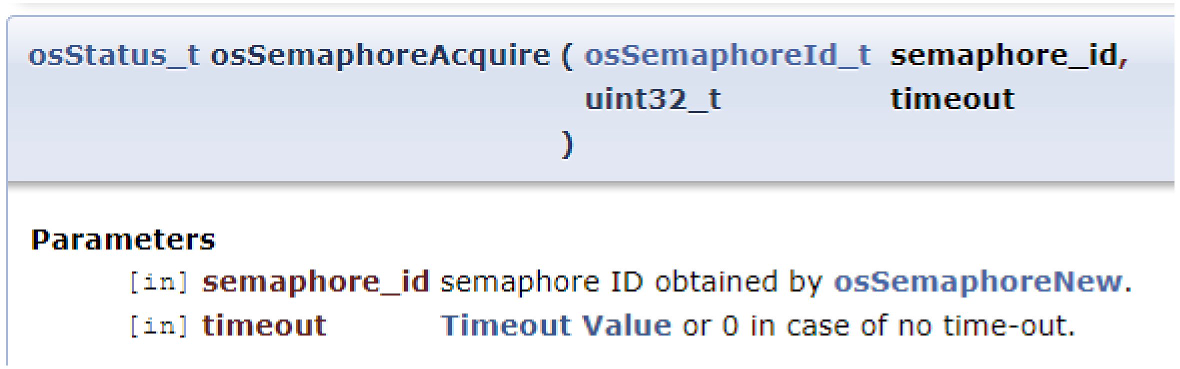


```
int main (void) {

  // System Initialization
  SystemCoreClockUpdate();
  InitGPIO();
  offRGB();
  // ...

  osKernelInitialize();                 // Initialize CMSIS-RTOS
  mySem = osSemaphoreNew(1,1,NULL);
  osThreadNew(led_red_thread, NULL, NULL);     // Create application led_red thread
  osThreadNew(led_green_thread, NULL, NULL);   // Create application led_green thread
  osKernelStart();                      // Start thread execution
  for (;;) {}
}
```

Semaphore Routines

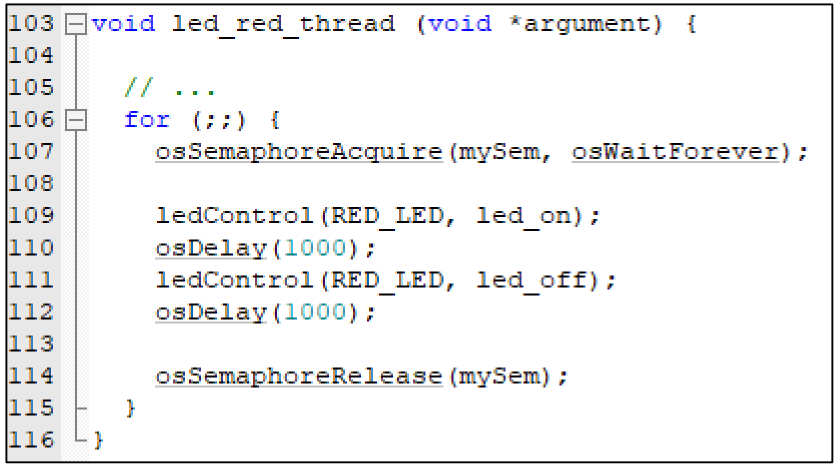


osStatus_t osSemaphoreAcquire (osSemaphoreId_t semaphore_id,
uint32_t timeout
)

Parameters

[in] **semaphore_id** semaphore ID obtained by **osSemaphoreNew**.

[in] **timeout** **Timeout Value** or 0 in case of no time-out.

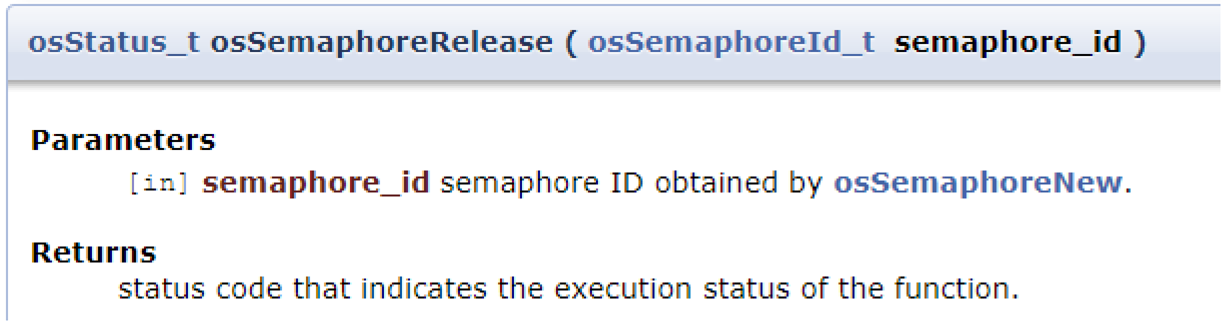


```
void led_red_thread (void *argument) {

  // ...
  for (;;) {
    osSemaphoreAcquire(mySem, osWaitForever);

    ledControl(RED_LED, led_on);
    osDelay(1000);
    ledControl(RED_LED, led_off);
    osDelay(1000);

    osSemaphoreRelease(mySem);
  }
}
```



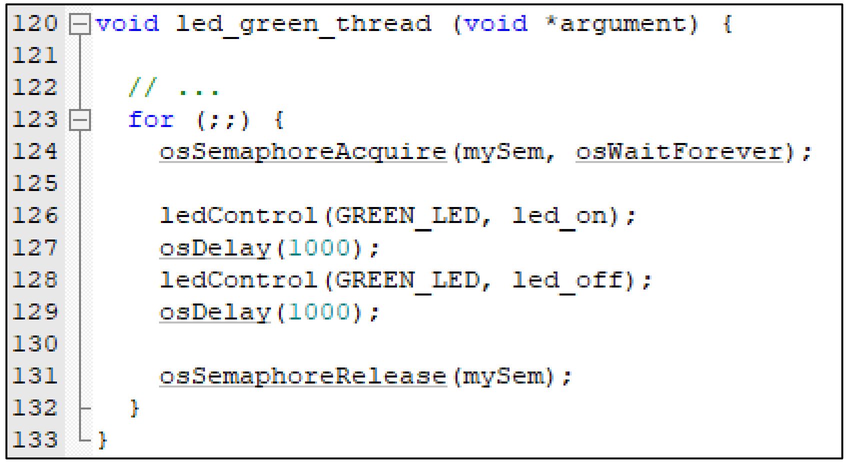
osStatus_t osSemaphoreRelease (osSemaphoreId_t semaphore_id)

Parameters

[in] **semaphore_id** semaphore ID obtained by **osSemaphoreNew**.

Returns

status code that indicates the execution status of the function.



```
void led_green_thread (void *argument) {

  // ...
  for (;;) {
    osSemaphoreAcquire(mySem, osWaitForever);

    ledControl(GREEN_LED, led_on);
    osDelay(1000);
    ledControl(GREEN_LED, led_off);
    osDelay(1000);

    osSemaphoreRelease(mySem);
  }
}
```

Mutex vs Semaphore

- Semaphore
  - Used for signaling from tasks or ISRs to waiting tasks
  - Can be initialized to 0, meaning "The event hasn't happened yet"

- Mutex
  - Used to ensure mutually exclusive access to a shared object
  - Is a binary semaphore with priority inheritance and some other changes
  - Is initialized to 1, meaning "The object isn't being used now"

- Common pitfall
  - Semaphore handles several copies of equivalent resources whereas mutex handles one?
  - Semaphore does not keep track of the order of access – effectively treating all resources identically!

Deadlock

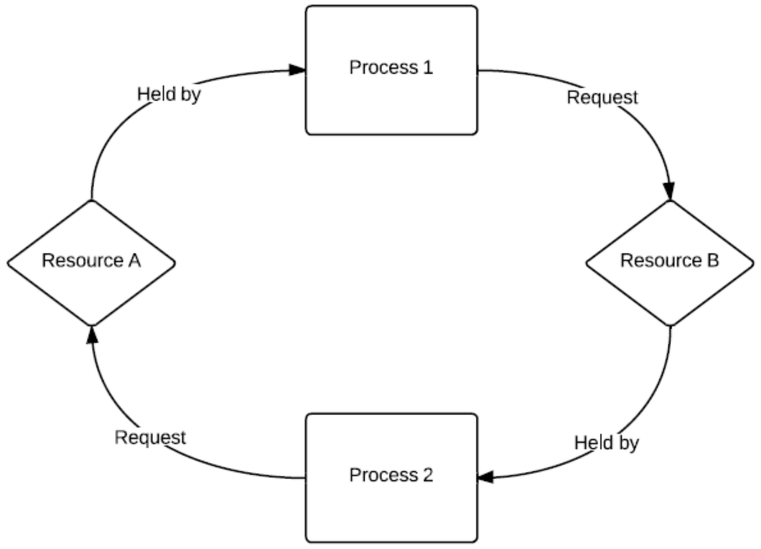
- All processes are waiting for others to finish so that required resources will be released, while reluctant to give up the resources that are also required by others. Thus nobody ever finishes.

- Generally, no perfect solutions yet.

Conditions for Deadlock

- If the following conditions are all true, deadlock may occur:
  - Mutex: exclusive resource, non-shareable
  - Resource holding: request additional resources while holding one
  - No preemption: resource can not be de-allocated or forcibly removed
  - Circular wait: circular dependency or a closed chain of dependency

$\left.\begin{array}{l}\text{Mutex}\\\text{Resource holding}\\\text{No preemption}\end{array}\right\}$ necessary but not sufficient conditions



The End!

- Next, Let’s look at the Lab