Sharing Data Safely (with Mutexes)

Overview – Compiler is not Omniscient

Simplify: first consider CPU which can operate on data in memory (not only load/store)

Vulnerability	Volatile Data: things outside this function can change data	Critical sections: non-atomic memory object updates **All** memory data object updates are critical sections
Fault Trigger	*Compiler Optimization: Assume data is not volatile, so remove redundant loads or stores*	*Preemption: Interleaving critical sections from different threads for same object* *Load/Store Architecture: makes updates of **all** memory objects into critical sections*
Solution	*"volatile" data type modifier for compiler*	*Prevent preemption of critical sections (interrupt/scheduler locking, mutex)*

Definitions

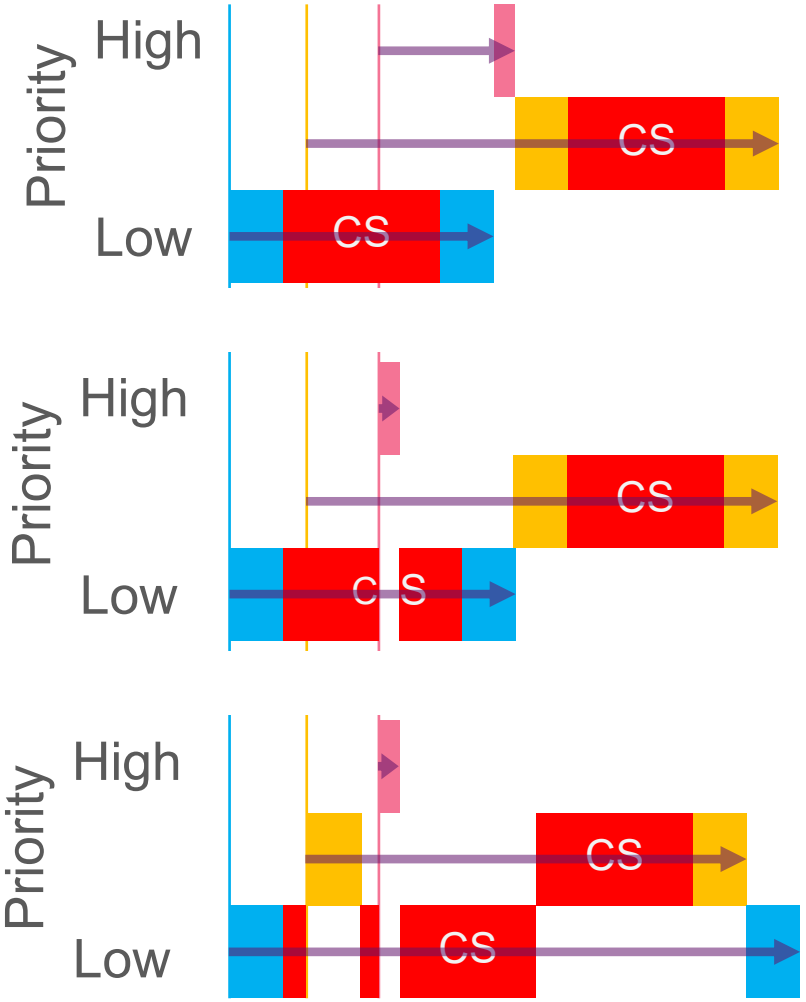
- Race condition
  - An ordering of read and write operations in multiple threads (or ISRs) which causes code to behave anomalously.
  - Is there a way to jump back and forth between two threads and get the wrong answer?

- Critical section
  - A section of code which creates a possible race condition.
    - *Any access to a shared object in a system with preemption is a critical section of code*
  - Only one critical section per shared object can be executed at a time.
    - OK to execute multiple critical sections concurrently if they access different shared objects.
  - Some synchronization mechanism is required at the entry and exit of each critical section to ensure exclusive use.

Mutual Exclusion Concepts

Range of Possible Solutions for Sharing Resource

Code accessing same shared resource is called a Critical Section

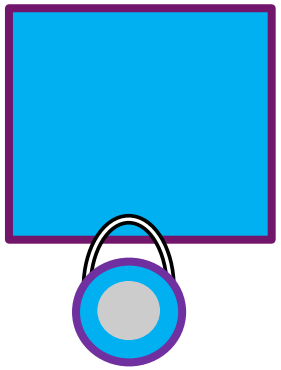


- Simple solution: Don't let any tasks preempt each other
  - Disadvantage: All higher priority tasks have to wait longer and finish later
- Slightly better solution: Don't let tasks which might access shared resource preempt each other
  - Disadvantage: Higher priority tasks which use shared resource have to wait longer and finish later
- Better solution: Let tasks preempt each other unless it will interleave accesses (critical sections) to same shared resource
  - If gold task wants to run its critical section now and it preempted blue task when it was using running the critical section, let blue task finish its critical section, and then let gold task run its critical section
  - Higher priority tasks finish sooner – just what we wanted

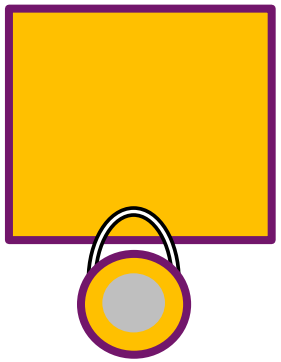
Using a Lock for Mutual Exclusion

Unlocked, resource is available

Locked by blue, only blue can use resource



Locked by gold, only gold can use resource



- Use a *lock* to *protect* each shared resource by only allowing a single thread to access it at a time
- Access Rules
  - We **must** get permission (acquire the lock) before we try to access the resource
  - We **must** release that permission (release the lock) when we are done accessing the resource
- ***What is*** the lock?
  - Regular shared variable? Vulnerable to concurrency bugs from preemption
  - Use OS-protected variable prevent them. Which type?
    - Event flag? Semaphore -- binary? counting? Something else?

```
my_data_type SharedResource …;
my_lock_type lockSR;

void T_Rose(void) {
  …
  do_work_without_SharedResource();
  …
}
void T_Gold(void) {
  …
  if (LockAcquire(lockSR)==OK) {
    // start of Gold's crit. section
    SharedResource.field1 += 37;
    // end of Gold's crit. section
    LockRelease(lockSR);
  } else handle error
  …
}
void T_Blue(void) {
  …
  if (LockAcquire(lockSR)==OK) {
    // start of Blue's crit. section
    SharedResource.field1 -= 12;
    // end of Blue's crit. section
    LockRelease(lockSR);
  } else handle error
  …
}
```

Lock with a Binary Semaphore?

```
my_data_type SharedResource …;
osSemaphoreId_t lockSR;

void init(void) {
  // create binary semaphore, init to available (1)
  lockSR = osSemaphoreNew(1,1, NULL);
}
void T_Gold(void) {
  …
  if (osSemaphoreAcquire(lockSR, osWaitForever)==OK)
{
    // start of Gold's critical section
    SharedResource.field1 += 37;
    // end of Gold's critical section
    osSemaphoreRelease(lockSR);
  } else handle error
  …
}
```

```
void T_Blue(void) {
  …
  if (osSemaphoreAcquire(lockSR, osWaitForever)==OK)
{
    // start of Blue's critical section
    SharedResource.field1 -= 12;
    // end of Blue's critical section
    osSemaphoreRelease(lockSR);
  } else handle error
  …
}

void T_Rose(void) {
  …
  do_work_without_SharedResource();
  …
}
```

- Works fine for this simple problem

Scaling Up? Complications

- Scaling up threads sharing a resource
  - Thread quantity
  - Priority inversion
  - Timing requirements vs. allowed slack
- Scaling up critical sections
  - CS quantity
  - Locations in source code
  - Sequencing behavior
- Other wants
  - Simplify programs with complex lock behaviors
  - Handle thread ending without releasing mutex

	Characteristic	Simple Case	Not so Easy…
Threads using Shared Resource	Quantity	Two or a few	Many threads
	Timing slack allowed. For how much extra time can resource be locked before hurting system performance?	Much slack	Little slack.
Critical Sections in Threads for Shared Resource	Quantity	Few	Many
	Locations	Known	Not so clear
	Sequence of execution	Known	Unknown, many possibilities

Mutex Motivation

- Let only one thread at a time access a shared resource using multiple critical sections
  - Examples: data structure with multiple fields which are independent of each other, some types of shared peripherals
  - **Note:** We trust the programmer not interleave the critical sections in this thread.
- Let multiple critical sections of **that** thread access that shared resource (but don't let other threads in until this one is done).
  - Don't want to have to track number or ordering of critical sections, since complicated.
  - Want to protect each critical section individually since much easier to implement in program.

Use a Semaphore?

- Binary semaphore?
  - Maybe? Complicates programming because of extra conditions
  - Acquire semaphore before **first** access/ start of critical section
  - Must protect intervening critical sections somehow -- but can't increment binary semaphore past one
    - Add code before each critical section to check to confirm we have the semaphore?
      - What should we do if we **don't** have it?
    - Assume we have semaphore and do CS anyhow?
      - Must check program very carefully, bugs likely
  - Then release semaphore after ***last*** access/ end of critical section. Which access is the last?
- Counting semaphore?
  - Fails because it allows different threads to interleave their critical sections, breaking atomic critical sections.

Use Two Binary Semaphores?

- Sem1: my thread has rights to use the shared resource
- Sem2: my thread has started but not finished a critical section

- Before each critical section, maybe acquire sem1 or sem2

- After each critical section, maybe release sem1 or sem2

Binary Semaphore + Counter Variable?

- Add a counter variable
  - Track number of active critical sections: started but not finished
  - Pending “releases”
- Before each critical section, *maybe* acquire sem
  - Counter == 0?
    - Acquire the semaphore, since this is thread’s first acquire.
    - Else skip the acquire, since this thread already has the semaphore.
  - Increment counter
- After each critical section, *maybe* release sem
  - Decrement counter
  - Counter == 0?
    - Last release, so release the binary semaphore.
    - Else skip the release, since we haven’t finished all the critical sections.

Mutex Implementation

- Want a **counting semaphore** which can only be used by one thread at a time

- Mutex is like two semaphores tied together

1. Binary semaphore to limit ownership to one thread at a time
2. Counting semaphore (lock counter) to track depth of recursive acquisitions, know when to release binary semaphore.

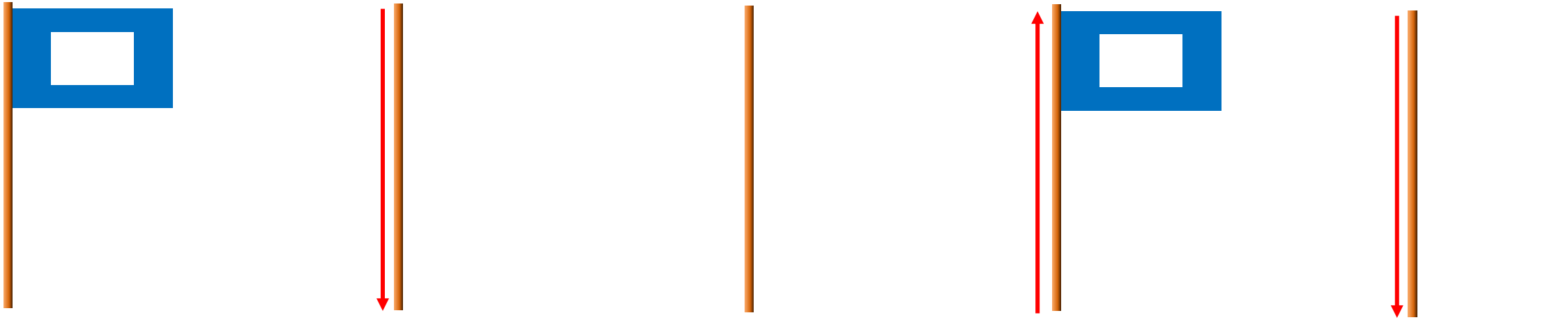
Operation:

1. When a thread acquires a mutex, that mutex "belongs" to the thread.
2. The thread can then do additional nested/recursive acquisitions, which increment the mutex's lock counter (a special hidden semaphore token counter variable) and always succeed immediately without blocking.
3. When the thread releases the mutex, the lock counter is decremented.
4. When the lock counter finally reaches zero, the thread doesn't own the mutex anymore, and another thread can acquire it successfully.

Mutex can also add priority inheritance/ceiling to owning thread if enabled.

Mutex OS Mechanism

Scheduler Behavior for Mutex



1. Start: Mutex available (count = 1)
2. Task A acquires mutex (count = 0)
3. Task B tries to acquire mutex but can't
4. Task A releases mutex (count = 1)
5. Scheduler resumes Task B, gives it the mutex (count = 0)

- Wait/Pend until mutex is free, then *acquire* it
  - If mutex available (1), take it and continue executing
  - Otherwise, OS puts calling task into *waiting* state until mutex is available
- Release the mutex
  - If a task is waiting for this mutex, OS moves that task to the *ready* state

CMSIS-RTOS2 Mutex Functions and Macros

- osMutexId_t osMutexNew(osMutexAttr_t * attr)
  - Creates mutex and makes it available (initializes it to 1)
- osStatus_t osMutexAcquire(osMutexId_t mutex_id, uint32_t timeout)
  - Wait for until mutex is available, then takes it (decrements count)
  - Optional timeout value (in scheduler ticks)
    - To never timeout, use osWaitForever
  - Return type is osStatus
    - osOK, osErrorTimeoutResource, osErrorResource, etc.
- osStatus_t osMutexRelease(osMutexId_t mutex_id)
  - Releases (signals) the mutex
  - Increments count to 1
  - Can't release a mutex you don't have! Error
  - Return type is osStatus
- osStatus osMutexDelete(osMutexId_t mutex_id)
  - Deletes mutex, frees internal memory it used
  - Return type is osStatus
- Note: Can't call mutex functions from ISR!

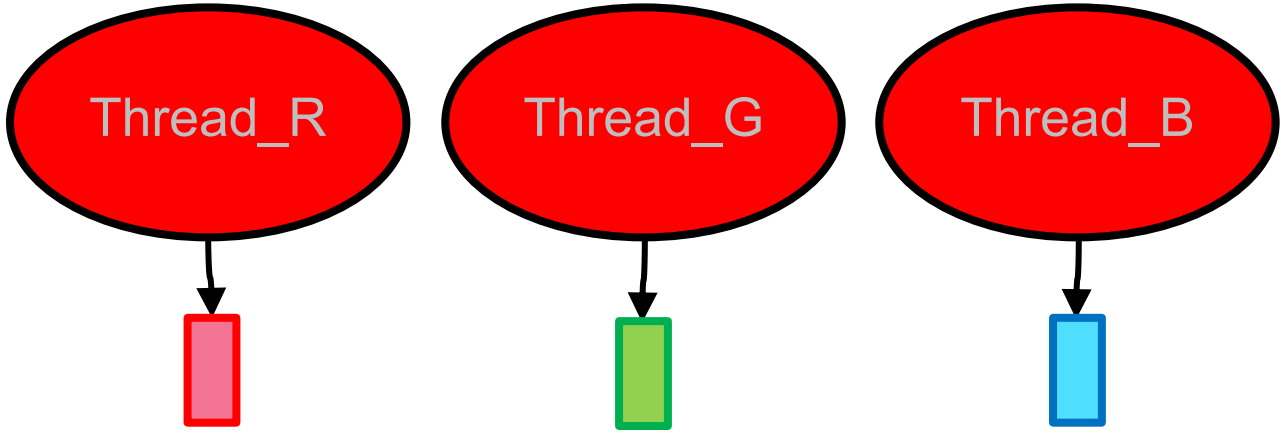
Using a CMSIS-RTOS2 Mutex Object

- Use a *mutex lock* to *protect* each shared variable, object or resource
  - A mutex provides mutual exclusion - Only a single task can have the mutex at a time
  - Mutexes are like binary semaphores with extra features
- Access Rules
  - We **must** get permission (acquire the mutex) before we try to access the resource
  - We **must** release that permission (release the mutex) when we are done accessing the resource

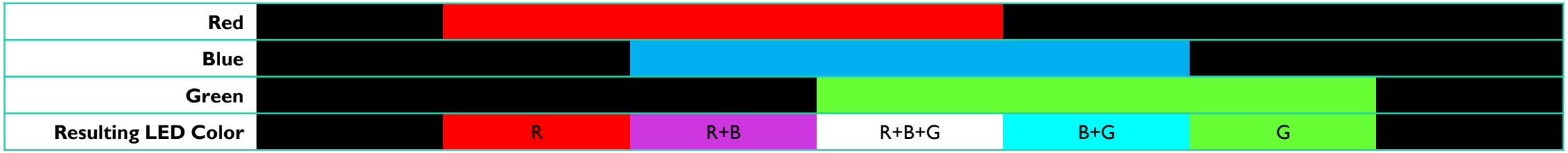
```
my_data_type SharedResource …;
my_mutex_type mutexSR;

void T_Rose(void) {
  …
  do_work_without_SharedResource();
  …
}
void T_Gold(void) {
  …
  if (osMutexAcquire(mutexSR)==OK) {
    // start of Gold's critical section
    SharedResource.field1 += 37;
    // end of Gold's critical section
    osMutexRelease(mutexSR);
  } else handle error
  …
}
void T_Blue(void) {
  …
  if (osMutexAcquire(mutexSR)==OK) {
    // start of Blue's critical section
    SharedResource.field1 -= 12;
    // end of Blue's critical section
    osMutexRelease(mutexSR);
  } else handle error
  …
}
```

RTX5 Demo: Mutex of LEDs

- Three threads running, each lighting its LED
- Requirement: Light at most one LED at a time – e.g. power constraint
- Threads are running independently, don't coordinate their activities
  - So may have more than one LED on

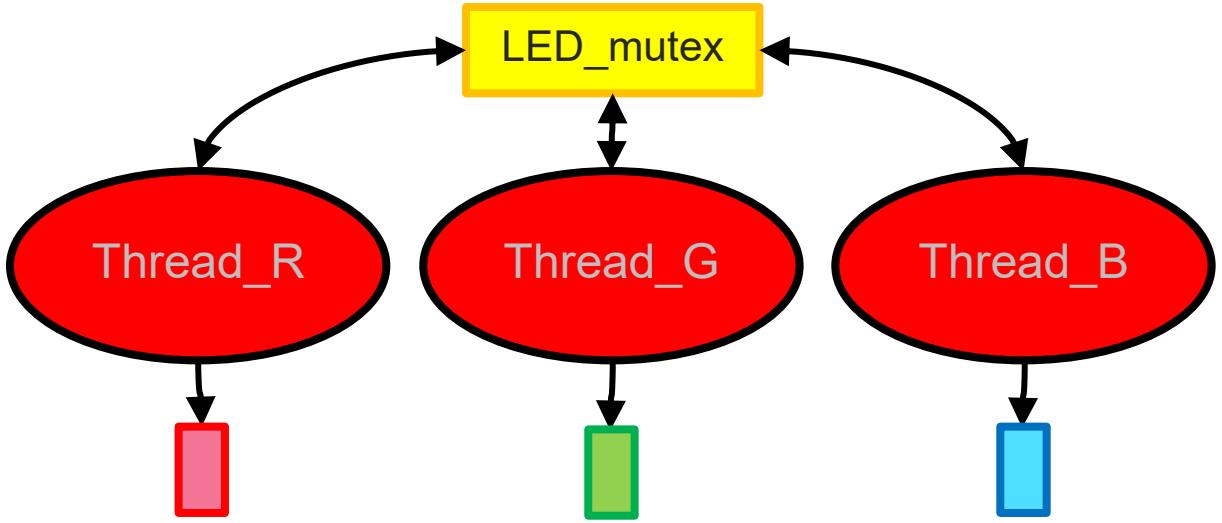


Without Mutual Exclusion



RTX5 Demo: Mutex of LEDs

- Use mutex to indicate **permission** to light an LED
  - Thread **blocks on** (waits for) the mutex before lighting the LED.
  - Thread **releases** mutex after turning off LED



With Mutual Exclusion

LED_mutex	1: available	0: held by Thread_R		0: held by Thread_B	0: held by Thread_G	1: available
Thread_R	blocking on osDelay	has mutex		blocking on osDelay		
Red						
Thread_B	blocking on osDelay		blocking on mutex	has mutex	blocking on osDelay	
Blue						
Thread_G	blocking on osDelay		blocking on mutex		has mutex	blocking on osDelay
Green						
Resulting LED Color		R		B	G	

Mutual Exclusion: Binary Semaphore vs. Mutex

	Binary Semaphore	Mutex
Meaning of counter value	1 = Shared object is available	1 = Shared object is available
Initial counter value	Must explicitly initialize to 1	Typically initialized to 1 when created
Who can acquire it?	Any thread	Any threa.
Who can release it?	Any thread	Only owner thread. Attempts by other threads will be ignored.
Which thread owns it?	No owner thread	Thread which acquired but hasn't yet released the mutex
Repeated acquires (recursive/nested locking)	Not allowed, since binary semaphore counts up to only 1	Allowed for owner thread (configurable)
Priority elevation (inheritance or ceiling)	Not provided	Provided by RTOS (configurable).

- Further discussion by Michael Barr at http://www.barrgroup.com/Embedded-Systems/How-To/RTOS-Mutex-Semaphore

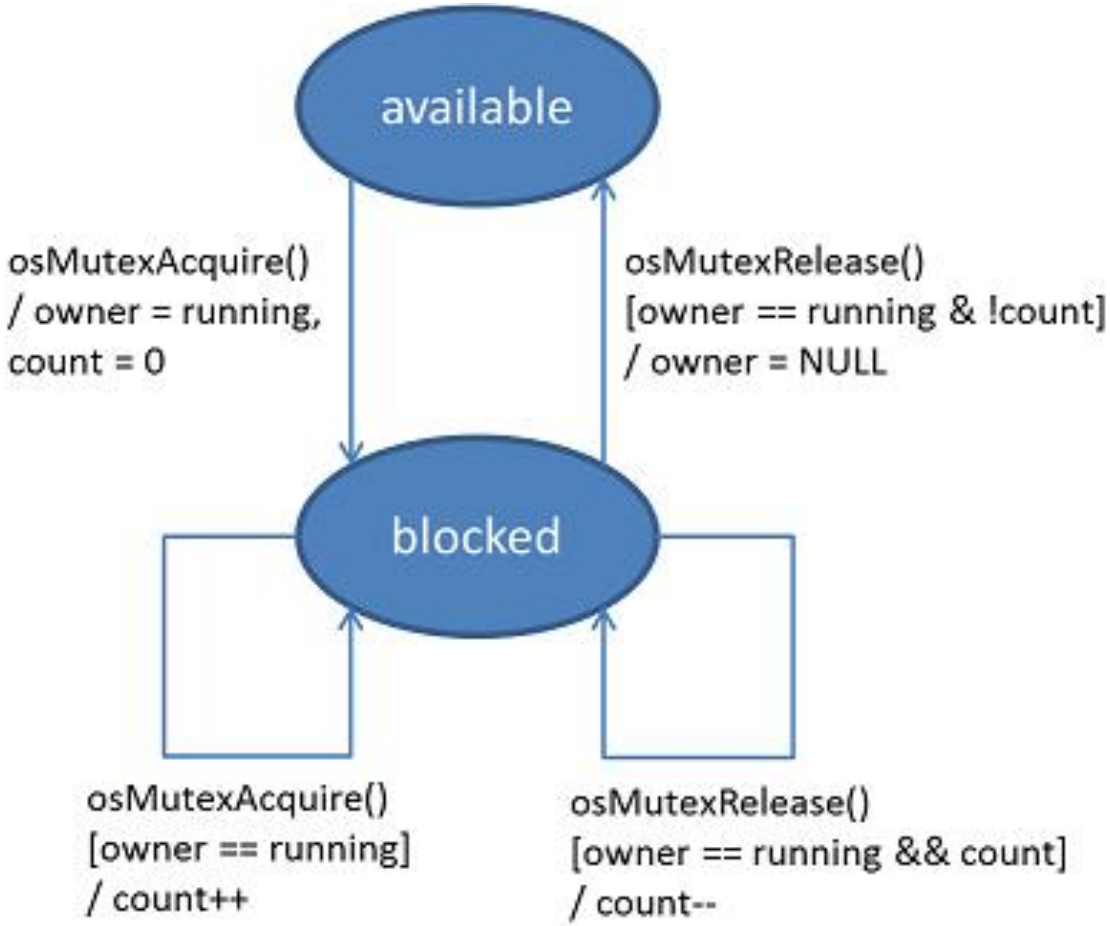
Mutex Options

- Recursive mutex allows nested locking
  - Thread can acquire mutex more than once before releasing it
- Priority inheritance
  - Thread temporarily inherits priority of higher-priority thread waiting on mutex
- Robust mutex
  - Release mutex automatically when owning thread terminates

- Defined by attr_bits in osMutexAttr_t when calling osMutexNew
- Details online at https://www.keil.com/pack/doc/CMSIS/RTOS2/html/group__CMSIS__RTOS__MutexMgmt.html#structosMutexAttr__t

Mutex and Thread Ownership

- *"When a thread acquires a mutex and becomes its owner, subsequent mutex acquires from that thread will succeed immediately without any latency (if osMutexRecursive is specified). Thus, mutex acquires/releases can be nested."* – CMSIS-RTOS2 documentation
  - http://www.keil.com/pack/doc/CMSIS/RTOS2/html/group__CMSIS__RTOS__MutexMgmt.html#details
- Semaphores don't provide this



Dangers: Volatile Data

ISR		Mem [do_work]	Thread_1	
Source Code	Assembly Code		Assembly Code	Source Code

- Compiler optimizes program by deleting instrs. which are useless from *function's point of view*
- Compiler assumes that this function's code runs without certain disruptions.
  - ISRs will never change function's state (e.g. local variables, registers, condition code flags, memory used by function) or global variables.
  - But subroutine calls may change **global variables**
- So, don't need to repeat instructions if **this function** hasn't changed their input data – just reuse the previous results
- Result: Compare do_work to 0 once, and reuse the result of that comparison

Solution: Warn the Compiler with "volatile"

ISR		Mem [do_work]	Thread_1	
Source Code	Assembly Code		Assembly Code	Source Code

- **volatile** keyword modifies variable's **storage class**
  - *Haunted!* **Something else** can change this variable's value **as this function executes**.
- For volatile variables, compiler generates code to access variable in memory **whenever** variable appears in source code.
  - Don't optimize by reusing first loaded value, or result of first comparison, etc.
- Should apply volatile to variables which either…
  - can be changed by …
    - Code in an ISR or a preempting thread
    - DMA (direct memory access) transfer
  - or which are actually mapped to hardware registers whose contents can change spontaneously

OLD: Dangers: Volatile Data

To Do:
- Introduce with memory/reg ISA to show general problem without load/store complications
- Add column with assembly code for each snippet to illustrate load/store optimization

```
volatile int32_t do_work=0;
```

```
ISR {
    // Event happened,
    // so ask work to be done
    do_work = 1;
}
```

```
Thread_1{
    while (1){
        if (do_work > 0) { // Any work?
            do_work = 0;
            // Do the work here
        }
    }
}
```

- Compiler tries to optimize program by deleting useless instructions (useless from *function's* point of view)
  - Compiler assumes that this function's code runs without certain disruptions.
    - ISRs will never change function's state (e.g. local variables, registers, condition code flags, memory used by function) or global variables.
    - Subroutine calls by the function may change global variables
  - So, don't need to repeat instructions if this function hasn't changed their input data – just reuse the previous results
  - Compare do_work to 0 once, and reuse the result of that comparison

- **Volatile** keyword (storage class modifier)
  - Indicates **something else** can change this variable's value as this function executes. (*It's haunted!*)
    - Code: in an ISR or another thread (if preemptive threads)
    - Hardware: DMA, or if variable is a hardware register
  - **Volatile** tells compiler to access variable **whenever** specified in source code.
    - Don't optimize by reusing first loaded value, result of first comparison, etc.

Dangers: Load/Store and Atomic

To Do:
- Split slide to revise flow/structure
  - Non-atomic data: requires multiple operations to update (e.g. string, struct, array) may be invalid during update
    - Example (using mem/reg ISA)
  - Load/Store ISA uses multiple instructions, makes all data updates non-atomic
    - Example (this one)
      - Add column with assembly code for each snippet to show load/store architecture makes increments non-atomic
      - Add arrows showing fail sequence
      - Add boxes around critical sections

Thread_1			Mem [counter]	Thread_2	
Source Code	Assembly Code			Assembly Code	Source Code

```
Thread_1{
    counter = counter + 1;
    // load r0 from memory
    // add 1 to r0
    // store r0 to memory
}
```

```
volatile int32_t counter=3;
```

```
Thread_2{
    counter = counter + 1;
    // load r0 from memory
    // add 1 to r0
    // store r0 to memory
}
```

- emory) to be
- thout interruption or preemption
- Any memory-resident variable modification uses at least 3 instructions: read (load), modify, write (store)
  - Modifying these variables is not atomic
  - This creates *critical section from the load instruction to the store instruction* (inclusive)
  - Even single-word variables are vulnerable to corru e variable
  - Incre
- So threa ory variable
  - **Any variables used in shared memory communication must be protected somehow**

Solution:

ISR		Mem [do_work]	Thread_1	
Source Code	Assembly Code		Assembly Code	Source Code

Dangers: Load/Store and Atomic

```
volatile int32_t
```

```
Thread_1{
    counter = counter + 1;
    // load r0 from memory
    // add 1 to r0
    // store r0 to memory
}
```

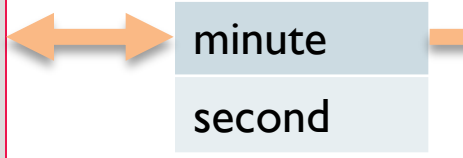
To Do:
- Split slide to revise flow/structure
  - Non-atomic data: requires multiple operations to update (e.g. string, struct, array) may be invalid during update
    - Example (using mem/reg ISA)
  - Load/Store ISA uses multiple instructions, makes all data updates non-atomic
    - Example (this one)
      - Add column with assembly code for each snippet to show load/store architecture makes increments non-atomic
      - Add arrows showing fail sequence
      - Add boxes around critical sections

```
}
```

- ARM is a Load/Store architecture
  - Variables must be in registers (not memory) to be tested or modified
  - Need to load variable into register from memory before using it
- **Atomic** operations are performed without interruption or preemption
- Any memory-resident variable modification uses at least 3 instructions: read (load), modify, write (store)
  - Modifying these variables is not atomic
  - This creates *critical section from the load instruction to the store instruction* (inclusive)
  - Even single-word variables are vulnerable to corruption
- What if two threads try to increment the same variable in memory?
  - Incrementing 3 twice results in 4 or 5
- So threads communicating with shared memory variables are vulnerable to *race conditions*
  - **Any variables used in shared memory communication must be protected somehow**

Preemption and a Data Structure

```
void Timer_ISR(void){
    TimerVal.second++;
    if TimerVal.second >= 60 {
        TimerVal.second -= 60;
        TimerVal.minute ++;
    }
}
```

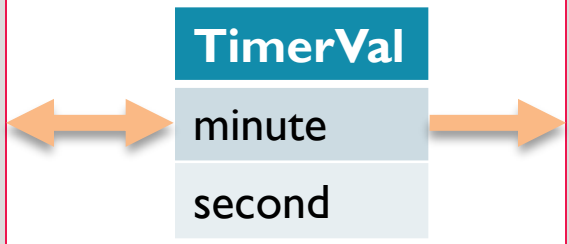



```
void GetTime(TimeType * T){
    T->minute = TimerVal.minute;
    T->second = TimerVal.second;
}
```

- System structure
  - TimerVal structure holds elapsed time
  - TimerVal's fields are updated by periodic timer ISR
  - Thread calls GetTime to copy time into time stamp
- Problem
  - An interrupt at the wrong time will lead to wrong data in T – some is old, some is new

Fail

```
void Timer_ISR(void){
    TimerVal.second++;
    if TimerVal.second >= 60 {
        TimerVal.second -= 60;
        TimerVal.minute ++;
    }
}
```




```
void GetTime(TimeType * T){
    T->minute = TimerVal.minute;
    T->second = TimerVal.second;
}
```

- TimerVal is {0, 59}
- Thread code calls GetTime(), which starts copying the TimerVal fields to T: minute = 0
- A timer interrupt occurs, which updates TimerVal to {1, 0}
- GetTime() resumes, copying the remaining TimerVal field to T: second = 0
- T now has a corrupted time stamp of {0, 0} (old, new)
- ***The system thinks time just jumped backwards one minute***

Data Corruption from Preemption

To Do:
- Old data vs. corrupted data vs. new data concepts
- Add big picture of how preemption can corrupt data:
1. Data variables in program (this slide)
2. Peripherals
    1. Preemption within operation (e.g. write byte to SD or LCD controller)
    2. Preemption between sequence of operations (e.g. send command + data to SD or LCD controller = sequence of byte writes)

- What could possibly go wrong with a ***shared data***?
  - Data may be overwritten partway through being read or written

# Readers	# Writers	Overwrite during Read?	Overwrite during Write?
>0	1	**Possible**	Impossible
>0	>1	**Possible**	**Possible**

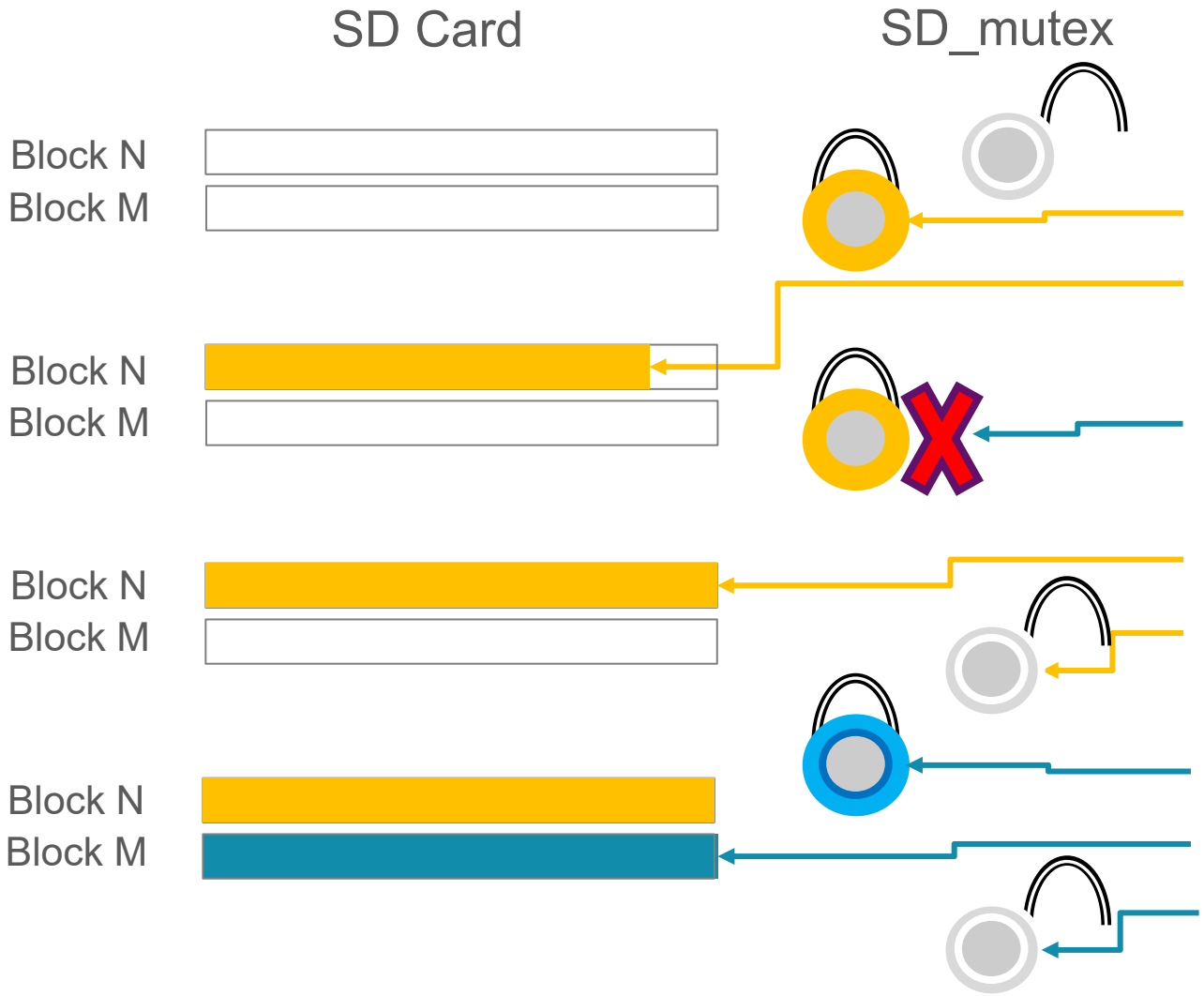
  - Other corruptions: e.g. two threads incrementing same variable
- What could possibly go wrong with a ***shared peripheral***?
  - All sorts of problems! Depends on the peripheral and device
- You must ensure indivisible (*atomic*) access to the shared objects
  - Don't let a thread or ISR access an object until an ongoing update has completed

Preemption and a Peripheral: SPI and a µSD Card

- Preemption gives *interleaved* task execution

- Example: Two tasks can access SD card via SPI
- Possible failure:
  - Task 1 starts reading data from SD card block N (sending 0xFF to clock out data) but is switched out by scheduler before finishing
  - Task 2 starts writing new data to SD card block M
  - Scheduler switches out Task 2 to run Task 1
  - Task 1 resumes reading from SD card, sending 0xFF to clock out data. SD Card interprets 0xFF as data to write to block M.
  - Task 1 finishes and is switched out
  - Task 2 resumes and tries to complete by writing rest of data, but will not succeed.
  - Result: Task 2's SD card block is corrupted, with some blocks overwritten by 0xFF. And SD card controller is probably stuck.

Solution: Task Locks Resource(s) When in Use



- SD_mutex is initially unlocked
- Task 1 locks SD_mutex
- Task 1 starts updating SD card
- Task 2 preempts Task 1, starts running
- Task 2 tries to lock SD_mutex but fails and blocks
- Scheduler switches back to Task 1
- Task 1 finishes updating SD card
- Task 1 unlocks SD_mutex
- Scheduler switches context to Task 2
- Task 2 locks SD_mutex
- Task 2 updates SD card
- Task 2 unlocks SD_mutex
- Task 2 completes and blocks
- Task 1 finishes other (non-SD Card) work