ECEN 3753: Real-Time Operating Systems

Shared Resources

Resource Contention Examples to be discussed

1. Data Races
2. Deadlock
3. Priority Inversion

Example of Mutual Exclusion : Data Races

- Generally occur when the order of CPU stores (writes) and loads (reads) are non-deterministic.
- Example 1 (conflicting writes)
  - void Task1(void*) {global_val = 1;}
  - void Task2(void*) {global_val = 2;}
  - When both tasks are finished, what is the value of global_val?

More On Data Races. . .

- Example 2 (read-modify-write with pass by value)
  - void Task1(void*) {global_val = AddOne(global_val);}
  - void Task2(void*) {global_val = SubtractOne(global_val);}
  - If the initial value of global_val is 0, what is its value when both tasks are complete?
  - Suppose Task1 runs, loads (reads) global_val and decides that it wants to set a new value of 1, but gets swapped out before completing its store (write)
    - Task2 now runs to completion, and captures the global_val parameter by value (global_val=0) before Task1 has rewritten it to 1.
    - Task2 runs to completion, changing global_val from 0 to -1
    - Task1 resumes, and sets global_val to what it had originally intended to write (global_val=1)

See visual representation of this type of problem, in IPC Lecture Topic.

The Solution to Data Races

If we can cause multiple potential users of a resource to mutually exclude their uses during each others', the result will be no unsafe/inappropriate sharing of data via race conditions!

The fundamental building block for Mutual Exclusion is the **Atomic Lock**.

Atomic and Volatile Data in C/C++

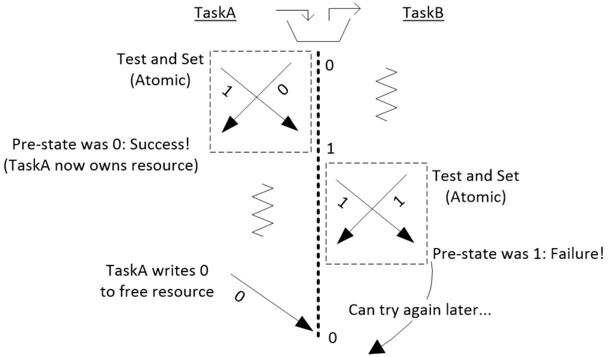
- Caveat: Other programming languages define these terms differently
- Volatile in C/C++
  - Does not prevent data races.
  - Used to prevent the compiler from optimizing out a read|write.
    - Memory mapped HW peripherals
- Atomic in C/C++
  - Can be used to prevent data races.
  - Helps enforce an order on loads/stores to memory
    - A happens before B (e.g. enumerated steps)
    - A synchronizes with B (e.g. indicator to block B)

More on C/C++ Atomics

- Typically implemented with CPU word-wide default accesses for small data, without needing an OS mutex lock.
- For larger data, may use an OS mutex lock as part of the solution.
- Uses compare_and_exchange (alternatively: test_and_set) operations for read-modify-write
  - Loads data and compares to expected value
  - If the compare succeeds, the new value is written, otherwise the value that was just read is returned and no write occurs.
  - A task may check whether the compare_and_exchange succeeded to guarantee that it was the task that completed the write successfully.
  - If unsuccessful, a task may loop using the value it read (which is data written by some other task) as the new expected value, and retry its write.

What is Test_And_Set?

Preferably **Hardware** operation on a bit, for use in mutual exclusion. (See Intel:"XCHG"", ARM:"SWP"")


TaskA
TaskB
Test and Set (Atomic)
1
0
0
Pre-state was 0: Success! (TaskA now owns resource)
1
Test and Set (Atomic)
1
1
Pre-state was 1: Failure!
TaskA writes 0 to free resource
0
Can try again later...
0

Design Pattern: Critical Sections

- Enter critical region
  - Access shared resources
- Exit critical region

The shared resource MIGHT be the CPU itself.

Atomic Flag to Protect a Critical Region

- One way to protect a critical region is to use an atomic flag ensure that only one task at a time may enter.
  - Setup an atomic boolean flag, normally cleared to indicate that no tasks are inside the critical region.
  - When a task wants to enter the critical region it tries to do an atomic test_and_set
    - If the test_and_set succeeds, the task may enter the critical region and is responsible for re-clearing the flag on the way out.
    - If the test_and_set fails, some other task succeeded on its set and is inside the critical region. The current task must wait until the other task exits the critical region.

When might blocking on a mutually-exclusive lock via the OS be better/worse than spinyielding?

Mutual Exclusion Strategies

- Mutex
- Semaphore (binary)
- Disable interrupts
- Disable the OS scheduler

Mutex

A Mutex is a **Lock** for supporting Mutual Exclusion on a Shared Resource.

- With a mutex, a task can indicate when it is entering/exiting a critical code region.
- While one task is inside the critical region, other tasks may not enter this region, but interrupts still may occur.
- To enter the critical region, a task acquires (lock/pend) the mutex.
    - If some other task in the system tries to acquire a mutex owned by another task, it blocks waiting for the mutex to be released.
    - We may choose to wait for either an infinite duration or with a timeout.
    - In rare cases, a task may acquire a mutex more than once (recursive mutex)
- To exit the critical region, the task (or recursion) that acquired the mutex releases (unlock/post) it.

One Critical Region…

Or Many.

Up until here, I described ONE piece of code that would access a shared resource, but in general there could be MANY that access the same shared resource. Locking must be done by each section to keep ALL conflicting sections from accessing the shared resource at the same time.

Mutexes in Micrium

- Creating a mutex
  - void OSMutexCreate(OS_MUTEX* p_mutex, CPU_CHAR* p_name, RTOS_ERR* p_err);
- Acquiring a mutex. Calling task may block.
  - void OSMutexPend(OS_MUTEX* p_mutex, OS_TICK timeout, OS_OPT opt, CPU_TS* p_ts, RTOS_ERR* p_err);
- Releasing a mutex
  - void OSMutexPost(OS_MUTEX* p_mutex, OS_OPT opt, RTOS_ERR* p_err);

Semaphore (repeat from ITC Lecture)

Semaphores are a **signaling mechanism**, which (in “binary” mode) can be used to implement mutual exclusion on shared resources, but are not explicitly associated with data.

- Semaphores are not ‘owned’ by any particular task.
- Semaphores are initialized with an initial counter value.
- Acquiring (“**Wait/Pend**-ing”) a semaphore when the counter is:
    - Zero–causes our task to block.
    - Non-zero–decrements the counter, and our task continues to run.
- Releasing (“**Signal/Post**-ing”) a semaphore may increment the counter, if there are no other tasks waiting for the semaphore.
    - If a task(s) are waiting for the semaphore, the counter remains 0 and at least one waiting task is unblocked.
    - Semaphore options may be set to unblock either ‘one’ or ‘all’ waiting tasks when a semaphore is released.

Semaphores For Mutual Exclusion

- A binary semaphore may be used exactly like a mutex
  - Create the semaphore with an initial count of 1
  - Always pair up acquire and release operations on all tasks
    - A task acquires the semaphore.
    - The task that acquired the semaphore subsequently releases it.
    - All other tasks must acquire then release in the same order.

Disabling Interrupts

- On ARM, typically implemented as masking off the IRQ (and potentially FIQ) bit(s) in CPSR.
- Disabling interrupts prevents any type of preemption
- Necessary when resources are shared between tasks and interrupt handlers
- Must consider the consequences
  - Interrupt latency
  - Missed interrupts
- Make sure to re-enable interrupts

Implemented in Micrium by macros OS_ENTER_CRITICAL() and OS_EXIT_CRITICAL()

Disabling the Scheduler

- Can be used when a resource is shared between tasks, but not shared with an ISR (IRQs are NOT blocked by disabling the scheduler)
- Prevents the OS from changing task states, effectively allowing one task to run indefinitely
  - No other tasks may unblock.
  - No other tasks may transition from READY to RUNNING (even if a higher priority, or unrelated to the region being protected).
- Repeated/recursive disabling is possible. Scheduling is held off until ALL locks are removed.

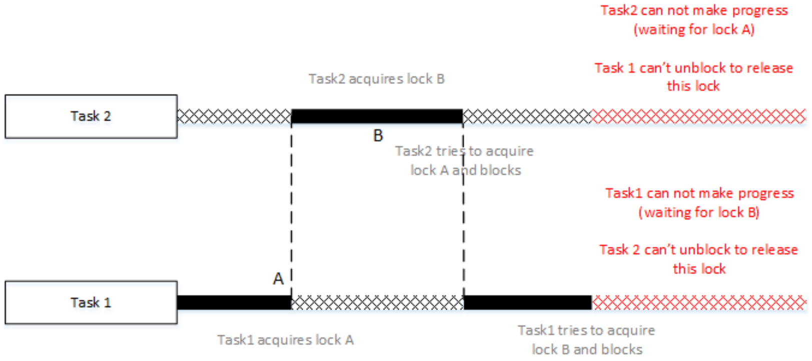
Implemented in Micrium by OSSchedLock()/OSSchedUnlock().

What would happen if you make a blocking call from a task while the Scheduler is disabled?

Consequences of Disabling Interrupts/Scheduling

- Late handling of interrupts
- Missed interrupts can lead to lost data
- Interference with timing requirements for unrelated tasks
- Higher priority tasks may not run, even if they have no relation to the critical resource being protected

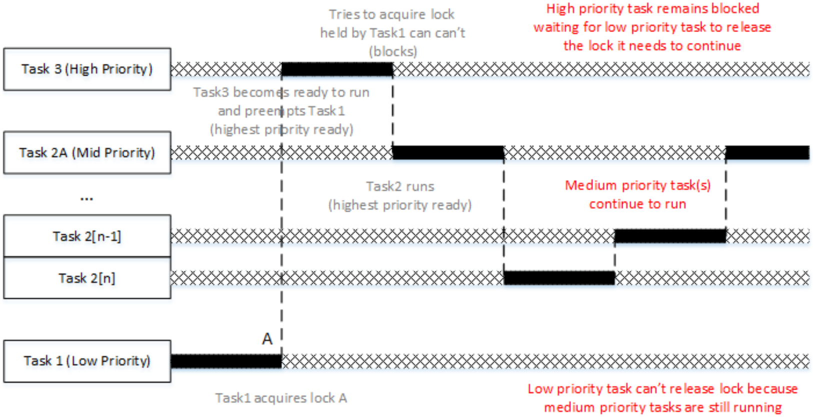
Resource Contention Issue #2: Deadlock


Task2 can not make progress
(waiting for lock A)
Task 1 can't unblock to release this lock
Task2 acquires lock B
Task 2
B
Task2 tries to acquire lock A and blocks
Task1 can not make progress
(waiting for lock B)
Task 2 can't unblock to release this lock
A
Task 1
Task1 acquires lock A
Task1 tries to acquire lock B and blocks

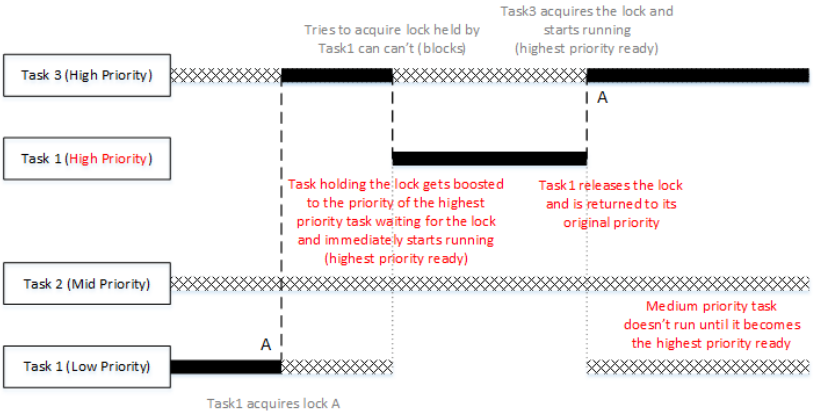
Preventing Deadlock

- Always acquire locks in the same order throughout the system
    - If a task needs to lock A before B, no one should ever lock B before A
    - Some static analysis tools (e.g. Coverity) are able to catch this heuristically; E.g. if your code locks A before B 95% of the time, chances are that the 5% case where B is locked before A is a bug.
- Always release locks in the reverse order that they were acquired.
    - If a task locks A then B, it should release B then A.
- Avoid creating circular dependencies between tasks. (More on this in a couple lectures)

Resource Contention Issue #3 : Priority Inversion



Priority Inheritance


Task3 acquires the lock and starts running (highest priority ready)
Tries to acquire lock held by Task1 can can't (blocks)
Task 3 (High Priority)
A
Task 1 (High Priority)
Task holding the lock gets boosted to the priority of the highest priority task waiting for the lock and immediately starts running (highest priority ready)
Task1 releases the lock and is returned to its original priority
Task 2 (Mid Priority)
Medium priority task doesn't run until it becomes the highest priority ready
A
Task 1 (Low Priority)
Task1 acquires lock A

Shared Resource Programming Mistakes

- Entering but forgetting to exit a critical region
    - Can occur when a function has more than one return point (if (error); return error;)
    - C++: Can use RAII (resource acquisition is initialization) to guarantee that all function exit paths exit critical regions.
- Improper nesting of OS locking calls
    - One function starts a critical region and calls to a second function that tries to exit the same critical region.
    - Prefer to avoid recursive mutexes, but if they are used, always enter/exit the critical region in the same function.
- Implementing critical region protection around too 'large' a region.
- Leaky API design: Allowing a pointer to protected data to leak out of a function.
- Inadvertently trying to block or wait inside an ISR.
- Others?