SJSU SAN JOSÉ STATE UNIVERSITY

Charles W. Davidson College of Engineering

Department of Computer Engineering

Real-Time Embedded System Co-Design

CMPE 146 Section 1

Fall 2024

Task Communication

Communication Methods

- A task may need to pass information to other tasks for processing
- Two general categories of methods for inter-task communication
  - Focus on methods suitable for real-time embedded applications
- Category 1: Task-owned objects (application resources)
  - Shared variable
  - Shared memory region
- Category 2: Kernel objects (system resources)
  - Queue
  - Pipe
  - Signal

Shared Variable

- The simplest approach is to just have a global primitive-data-type variable accessible to all tasks concerned
  - Primitive data type examples in C: char, bool, int, short, etc.
    - Memory read and write operations are atomic (cannot be disrupted by interrupts or preemption)
    - Provides the property of mutual exclusion
  - Method is easy to implement
  - Very fast, no OS overhead
  - Method can be expanded to multiple variables as long as they are not related in any operations (no association and independent of each other)
- There is no access control at software level
  - All tasks have access to the shared resource at any time
- Serious constraints
  - Only one writer task, all others are readers
  - Very limited information can be transferred
  - Not efficient as reader tasks need to constantly monitor the variable

Shared Memory Region

- For larger amount of information to be shared, a memory region is needed
  - Some locking mechanism is required
    - Otherwise, race condition would occur
  - Only one task is allowed to access (even read only) the region at any time
    - Operations on the region must be atomic to preserve data integrity
  - Disabling interrupt would be too costly for large regions
    - Won't work in multi-core systems anyway
- We can use a shared global variable as a lock (like a binary semaphore)
- Use test-and-set instruction to operate on the lock (also called spinlock)
- Test-and-set instruction
  - Reads from a memory location and writes a value to it
  - Read and write operations are parts of a single memory transaction
    - Atomic sequence, cannot be disrupted by interrupt or preemption
  - Mostly supported in various processors and memory systems

```
bool lock = FALSE;
bool test_and_set(bool* lock)
{
    bool value = *lock;
    *lock = TRUE;
    return value;
}
```

Shared Memory Region (cont'd-1)

- Usage:

```
bool lock = FALSE;
Taskx()
{
        ⋮
    while (test_and_set(&lock)); // Get lock
    // Critical section
        ⋮               // Access the memory region
    lock = FALSE;                    // Release lock
        ⋮
}
```

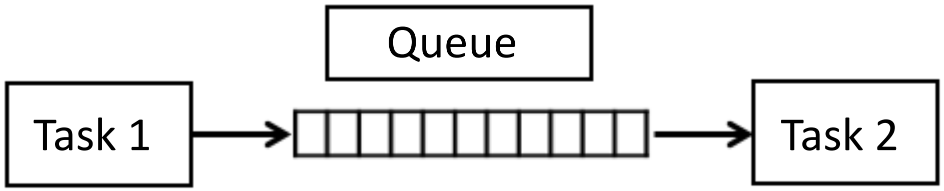
```
bool lock = FALSE;
bool test_and_set(bool* lock)
{
    bool value = *lock;
    *lock = TRUE;
    return value;
}
```

- How it works
  - Lock is initialized to FALSE (region is not locked)
  - First execution of *test_and_set* returns FALSE and lock = TRUE
  - Critical section is entered and memory region can be accessed
  - Other tasks executing *test_and_set* will get TRUE
    - Prevented from entering the critical section
  - When the lock-owner task is done with the critical section, set lock = FALSE
    - Next task executing *test_and_set* will get FALSE, thus gains access to memory region

Shared Memory Region (cont'd-2)

- If C compiler does not support the instruction, we can embed assembly instruction inline
  - For example, use SWP instruction on ARM processors
    - SWP R1, R2, [R0]   ; Swap a word between registers and memory (address in R0)
      ; Read [R0] to R1 and write R2 to [R0]
- Advantages
  - Easy to implement
  - Fast, no OS overhead
- Disadvantages
  - Two separate data structures (lock and shared region) for one purpose
  - Busy-wait on the lock
    - Waste CPU time

Message Queue



- Queues are independent kernel objects
  - Provide a means for tasks to send messages
  - Messages are read in the order as they in the queue
  - Some OSes also support a "broadcast" feature
    - One task sends the same message to multiple tasks
- Each message has an associated priority value (an integer)
  - Messages are queued according to the priorities
  - A new message with a higher priority is inserted before messages with lower priorities
  - A new message is inserted after the queued messages with the same priority

Message Queue (cont'd)

- Main POSIX functions
  - mq_open(): Create a new message queue
  - mq_send(): Send a message to the queue
  - mq_receive(): Receive the message from the head of the queue
- mq_send() can be blocked if no space is available in the queue
  - Calling task will be suspended
- mq_receive() can be blocked if no message is in the queue
  - Calling task will be suspended
- Advantages
  - Access control is all managed by OS
  - Message priorities for real-time control
  - Efficient use of processor time
    - No busy-wait to get access to the queue
    - mq_send() and mq_receive() can be blocking calls

Pipe

- A pipe is essentially identical to a queue, but processes byte oriented data
- Strictly FIFO (First In, First Out) stream of bytes
  - There is no notion of priority
- Provides simple communication channel for unstructured data exchange among tasks
- Main POSIX functions
  - popen(): Create a pipe
  - write(): Write a certain number of bytes
  - read(): Read a certain number of bytes
- write() and read() can be configured to be blocked or not
- Advantage
  - Can be used to handle data from any serial devices
    - Reader task does not need to know the details of the device

Signal

- Signals are indicators to a task that an event has occurred
  - Standard system event examples: memory access violation, divide by zero
  - Application event examples: a switch is turned on, message received from some communication channel, a computation step is done
- A task can perform useful work while waiting for an event
  - When the event occurs, the signal handler in a task is invoked
  - A task can specify a signal handler to execute for a specific event
    - Works like a virtual interrupt
- The signal handler is typically executed within the task's context
  - It runs in the place of the signaled task when it is the signaled task's turn to run
- Main POSIX functions
  - signal(): Set a handler for a signal
  - sigaddset(): Add a signal to a signal set
  - sigqueue(): Queue a signal to a task