...MAKE YOUR NETWORK SMARTER

Real-time Programming in Embedded Systems

ETRAX

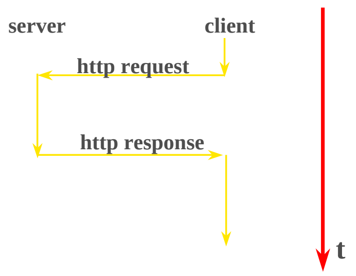
Real-time embedded?

- Real-time
  - Not multi-tasking
  - Not safety-critical
- Embedded
  - Little memory
  - Low speed
  - Runs unattended…
  - …but interacts with reality
  - Development environment often not so well supported
  - Target environment often custom
  - May be located in interesting places

What is a real-time system?

- A system that must react on external/internal events and deliver the right results on time.
- There are of course varying degrees of requirements on timeliness.

Example: HTTP server



Implementation Methods: Polling

```
while (true) {
  if ( packet_received )
     handle_packet();
  if ( timer_expired )
      blink_leds();
}
```

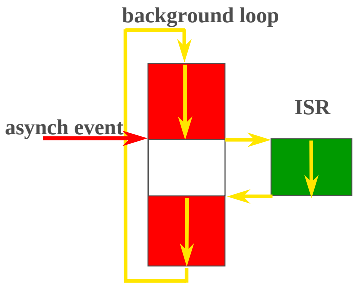
Implementation Methods: Polling

- Very efficient for small systems.
- What happens if a subroutine takes very long time to complete?
- Time consuming subroutine must return to main loop regularly to avoid locking out other tasks.
- The code becomes very complex when number of tasks increase.

Implementation Methods: Foreground/Background

- The background task is similar to the main loop in a polled system.
- The foreground task is interrupt service routines (ISRs) that handle critical events.

Implementation Methods: Foreground/Background



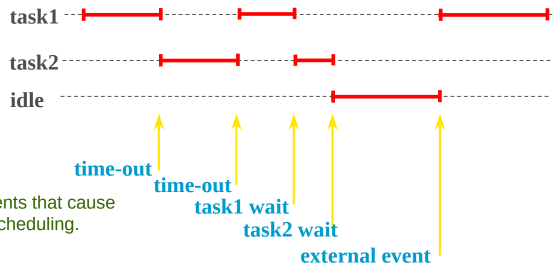
Real-time and other aspects

- Scheduling analysis
- Performance (interrupts, packets etc)
  - Throughput
  - Latency
  - Jitter
- Simplicity, understandability, correctness, provability
- Quality of service

Implementation Methods: Multitasking

- Partition the system into tasks/processes that execute in pseudo parallel.

Implementation Methods: Multitasking



Events that cause rescheduling.

Implementation Methods: Multitasking

- A scheduler determines which task to execute and when.
- Tasks are preempted, i.e., scheduler changes to another task after a certain time, to guarantee fair access to CPU.
- External events can also preempt a running task.

Implementation Methods: Multitasking

- A task can voluntarily release control to other tasks.
  - when waiting for external events.
  - when waiting for a time interval (sleep).
  - when waiting for access to shared resources.
  - when task is finished.

Implementation Methods: Multitasking

- What happens if a task is preempted while updating a shared object?
- Shared objects must be protected.
- Atomic operations are operations that are guaranteed to never be interrupted or preempted.
- In a single processor system this is usually accomplished by disabling interrupts.

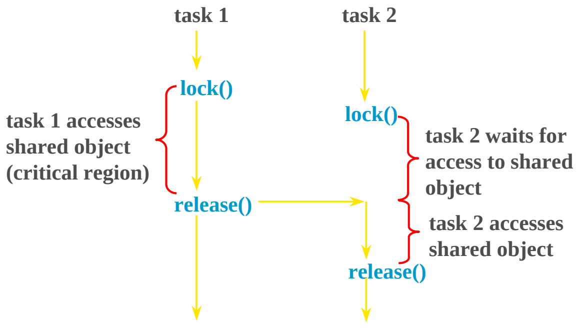
Inter-task Communication: Shared Variables

- Shared variables
  - Reading and writing a byte/dword variable is atomic in most processors and can therefore be used for communicating between tasks.
  - Requires (volatile) declaration in C/C++ to guarantee that variable isn't cached in a register.
  - Access to larger objects can be implemented by disabling interrupts.
  - This method doesn't transfer control to other tasks.

Inter-task Communication: Monitors

- Monitors
  - Protects an object by mutual exclusion (mutex) and also coordinates multiple tasks trying to access the same object.
  - Tasks that try to enter a monitor when it's locked by another task is put in a waiting queue and will be started when the monitor is free.

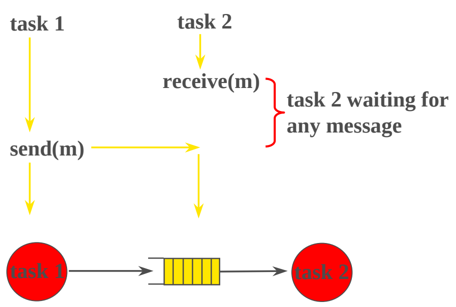
Inter-task Communication: Monitors



Inter-task Communication: Messages

- Message passing
  - Tasks communicate by sending/receiving messages containing data/objects.
  - Multiple messages to a task is usually kept in a message queue.

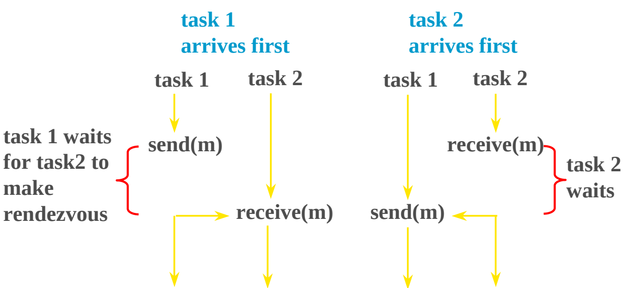
Inter-task Communication: Messages



Inter-task Communication: Events

- Events
  - Events can be considered a special case of message passing. Events are messages without any data and without message queues.
  - Note that, since there is no event queue, it's possible to get an event overrun where multiple events is seen as one by the receiving task.

Inter-task Communication: Rendezvous



Inter-task Communication: Rendezvous

- Both tasks must enter their send/receive call before the message is send.
- Which task comes first to the rendezvous doesn't matter.

Inter-task Communication: Semaphores

- Semaphore is a classical synchronization primitive, introduced by Dijkstra in 1965.
- Since then the name semaphore has been used for many different synchronization methods.

Dijkstra Semaphore

- A semaphore has two atomic operations, P() and V() and an integer state variable S. (Proberen, Verhogen)

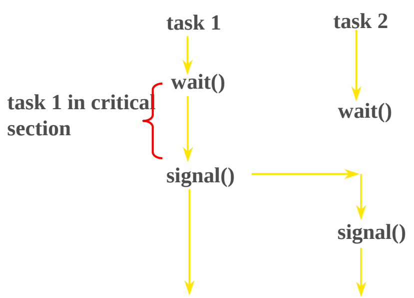
```
P() { // wait()
    while ( S <= 0 ) ;
    S--;
}
```

```
V() { // signal()
    S++;
}
```

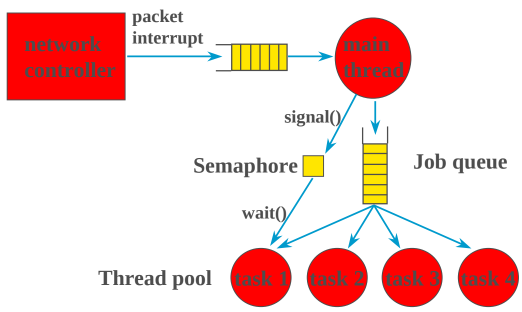
Dijkstra Semaphore

- The example implementation uses busy wait when a task waits for a semaphore. In reality, a semaphore is never implemented like that. Tasks that are waiting for a semaphore are placed in a waiting queue and started one at a time for each signal().

Monitor implemented with a semaphore



Task communication in our server.



Priorities

- One common source of problems in multitasking system is that some tasks are more important than others, e.g., it's more important to handle an Ethernet packet than to blink the status LED.
- One way of reflecting the importance of different tasks is to assigning priorities to all tasks and then letting the scheduler always run the highest priority task.

Problems with priorities

- It can be difficult to assign fixed priorities. Often the importance of a task varies over time.
- Although priorities can be used to ensure mutex, it's not recommended.
- Fixed priorities can lead to deadlock.

Priority inversion problem

- A resource shared between high and low priority tasks.
- During the time the low priority task locks the shared resource, the high priority task will be blocked if it also tries to access the resource.
- The solution is priority inheritance which must be implemented in the synchronization primitives.

Example: The Mars Rover Pathfinder

- Landed July 4th 1997
- Everything was perfect!
- Everything?
- After a few days…
- …the system began to reset sporadically
- L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. In IEEE Transactions on Computers, vol. 39, pp. 1175-1185, Sep. 1990.

Deadlock

- Simplified, deadlock occurs when a set of tasks wait on each others resources to become free.
- There are algorithms both to avoid deadlocks and to detect and resolve deadlocks.
- However, often deadlocks are avoided by a design and debug process which can't guarantee deadlock-free systems.

Starvation

- Starvation happens when a task always becomes locked out from accessing a resource.
- This can happen in a under-dimensioned system.
- It can also be caused by unfair synchronization methods.

Hard vs. Soft

- Hard real-time systems is a class of systems where you must be able to guarantee that the system works under all work-loads, e.g., control system for Ariane's engines.
- Some hard aspects:
  - Maximum response time.
  - Long term and short term frequency deviation in periodic tasks.
  - Maximum task timing jitter.

Scheduling methods

- In hard real-time systems it's necessary to guarantee timing correctness of the system.
- Rate monotonic scheduling (RMS)
  - If the system consists of periodic tasks with fixed-length execution time, rate monotonic scheduling can guarantee that the application meets it's deadlines.
  - RMS: assigns priority according to how frequently a task executes.

Scheduling methods

- Deadline driven scheduling (DDS)
  - When the system also has aperiodic tasks, deadline driven scheduling can be used.
  - DDS: schedule the task with the earliest deadline first.

Performance aspects

- Throughput, both in number of events/s and bytes/s is often an important performance aspect.
- Latency/response time is often also important.
- The *critical path* concept from hardware design can be a useful way of analyzing a system.

Performance Aspects: Task Creation

- Dynamic task creation can be a performance problem, especially if task creation is in the critical path (e.g., http-server that creates a task per connection request).
- Static task creation, i.e., all tasks are created at system start. This is very efficient. Even scheduling and task communications become more efficient. Drawback is of course the difficulty to adapt to system load.

Performance Aspects: Task Creation

- A pool of tasks is often a good solution.
  - Tasks are created in advance and then used upon demand. When the pool is starting to become fully allocated, a chunk of new tasks can be created. The system can therefore adapt to varying demands.

Performance Aspects: Context Switching

- Context switches is often a fairly expensive operation, involving saving/restoring all processor registers etc.
- It's important to use appropriate task communication methods that don't introduce unnecessary context switches, e.g., don't use message passing for accessing a shared database.

Performance Aspects: Context Switching

- Task partitioning can often influence the amount of context switches.
- A partitioning that makes sense for software modularization is not always the best partitioning for high performance, e.g., use one task per protocol layer.

Performance Aspects: Critical Regions

- Non-preemptive scheduling has one interesting performance advantage. Resources shared among the non-preempted tasks doesn't need mutex which reduces overhead in accessing shared data.

Performance Aspects: Critical Regions

- Since mutex is implemented by disabling interrupts, the amount of time when interrupts are disabled, influences how fast an event (e.g., timer or packet interrupt) can be handled.
- It's therefore important to minimize these regions.

Memory Management

- The memory allocation/deallocation system is a very important shared resource.
- In most embedded systems, memory is a scarce resource and therefore reallocation and defragmentation must be used. This results in slower memory allocation.
- Running out of memory must never happen which puts demands on how memory is managed.

Memory Management

- A large part of the memory allocation code must be performed with mutex. Performance therefore makes it important to minimize memory allocation calls.
- One solution is to allocate memory in advance and manage it in block-pools within each sub-system. This reduces allocation calls and also makes it easier to predict memory requirements.

Memory Management

- Memory leaks is a very common problem in complex system. This usually stems from that it's unclear who is responsible for deallocating memory of data that is passed between subsystems.
- Automatic methods for deallocation memory is therefore interesting.

Memory Management

- Smart pointers is one solution to automatic deallocation. By counting how many references there are to a memory block, it's possible to know when to deallocate that block. This can be implemented quite nicely in C++.

Garbage Collect

- Garbage collect is another solution where the system periodically scans for unused memory blocks (i.e., garbage).
- Contrary to what many believe, it's possible to implement garbage collect in C/C++.
- The straight forward garbage collect algorithms halts the system while scanning. This is of course unacceptable in real-time systems.

Garbage Collect

- Special real-time garbage collection algorithms have been developed but this is still a somewhat immature area and consequently not used in many systems.

Debugging Real-time Systems

- The biggest problem in debugging is to avoid influencing the system.
- Debug printouts is very slow and changes behavior, often to a degree where the system stops functioning correctly.
- Source level debugging with breakpoints, single stepping, etc. isn't very useful in a real-time system
- An *in-circuit emulator (ICE)* is very nice but not always feasible

Debugging Real-time Systems

- One non-intrusive method is to use a logic analyzer to trace what is happening in the system. A logic analyzer with large trace depth and good triggering functions can be as useful as a source-level debugger.
- If the processor has an internal cache, the usefulness of a logic analyzer is reduced since much of what's going on inside is hidden.

Debugging Real-time Systems

- There are many other tricks of the trade
  - Prevent and find errors earlier!
    - Design for quality
    - Test-driven development
    - Reviews
    - Static analysis
    - Software instrumentation
  - Post mortem debugging. Trace a small part of what's happening in the system, into memory. After an error has occurred, the trace can be printed.
  - Write trace data (e.g., one byte task-id, state) to an i/o port that can easily be traced with a cheap logic analyzer.
  - Use a memory emulator, that lets you inspect the memory of the running system without significant disturbance.
  - JTAG

Special considerations for embedded

- Exception handling
- Graceful degradation
- Self-testing and watchdog