Today's topic: RTOS

Why OS?
- To run a single program is easy
- What to do when several programs run in parallel?
  - Memory areas
  - Program counters
  - Scheduling (e.g. one instruction each)
  - ... Communication/synchronization/semaphores
  - Device drivers
- OS is a program offering the common services needed in all applications
  - (e.g. Enea's OSE kernel)

Operating System Provides
- Environment for executing programs
- Support for multitasking/concurrency
- Hardware abstraction layer (device drivers)
- Mechanisms for Synchronization/Communication
- Filesystems/Stable storage

We will focus on concurrence and real-time issues
First, a little history

Batch Operating Systems
- Original computers ran in batch mode:
  - Submit job & its input
  - Job runs to completion
  - Collect output
  - Submit next job
- Processor cycles very expensive at the time
- Jobs involved reading, writing data to/from tapes
- Cycles were being spent waiting for the tape!

Timesharing Operating Systems
- Solution
  - Store multiple batch jobs in memory at once
  - When one is waiting for the tape, run the other one
- Basic idea of timesharing systems
- Fairness, primary goal of timesharing schedulers
  - Let no one process consume all the resources
  - Make sure every process gets "equal" running time

Overall Structure of Computer Systems
Real-Time Is Not Fair

- Main goal of an RTOS scheduler:
  - meeting timing constraints e.g. deadlines
- If you have five homework assignments and only one is due in an hour, you work on that one
- Fairness does not help you meet deadlines

Do We Need OS for RTS?

- Not always
- Simplest approach: cyclic executive

```plaintext
loop
  do part of task 1
  do part of task 2
  do part of task 3
end loop
```

Cyclic Executive

- Advantages
  - Simple implementation
  - Low overhead
  - Very predictable
- Disadvantages
  - Can't handle sporadic events (e.g. interrupt)
  - Everything must operate in lockstep
  - Code must be scheduled manually

Real-Time Systems and OS

- We need an OS
  - For convenience
  - Multitasking and threads
  - Cheaper to develop large RT systems
- But - don't want to loose ability to meet deadlines (timing and resource constraints in general)
- This is why RTOS comes into the picture

Requirements on RTOS

- Determinism
- Responsiveness (quoted by vendors)
  - Fast process/thread switch
  - Fast interrupt response
- User control over OS policies
  - Mainly scheduling, many priority levels
  - Memory support (especially embedded)
- Reliability

Basic functions of OS kernel

- Process management
- Memory management
- Interrupt handling
- Exception handling
- Process Synchronization (IPC)
- Process scheduling
Process, Thread and Task

- A process is a program in execution.
- A thread is a "lightweight" process, in the sense that different threads share the same address space, with all code, data, process status in the main memory, which gives shorter creation and context switch times, and faster IPC.
- Tasks are implemented as threads in RTOS.

Basic functions of RTOS kernel

- Task management
- Interrupt handling
- Memory management
  - no virtual memory for hard RT tasks
- Exception handling (important)
- Task synchronization
  - Avoid priority inversion
- Task scheduling
- Time management

Micro-kernel architecture

Task: basic notion in RTOS

- Task = thread (lightweight process)
  - A sequential program in execution
  - It may communicate with other tasks
  - It may use system resources such as memory blocks
  - We may have timing constraints for tasks

Typical RTOS Task Model

- Each task a triplet: (execution time, period, deadline)
- Usually, deadline = period
- Can be initiated any time during the period
Task Classification (1)

- **Periodic tasks**: arriving at fixed frequency, can be characterized by 3 parameters (C,D,T) where
  - C = computing time
  - D = deadline
  - T = period (e.g. 20ms, or 50Hz)
  Often D=T, but it can be D<T or D>T

  Also called Time-driven tasks, their activations are generated by timers

Example: Fly-by-wire Avionics:
Hard real-time system with multi-rate tasks

<table>
<thead>
<tr>
<th>Sensors</th>
<th>Signal Conditioning</th>
<th>Control laws</th>
<th>Actuating</th>
<th>Actuators</th>
</tr>
</thead>
<tbody>
<tr>
<td>INU 1kHz</td>
<td></td>
<td>Pitch control 500 Hz</td>
<td>Aileron 1 1 kHz</td>
<td>Aileron</td>
</tr>
<tr>
<td>GPS 20 Hz</td>
<td></td>
<td>Lateral Control 250 Hz</td>
<td>Aileron 2 1 kHz</td>
<td>Aileron</td>
</tr>
<tr>
<td>Air data 1 kHz</td>
<td></td>
<td>Throttle Control 250 Hz</td>
<td>Elevator 1 kHz</td>
<td>Elevator</td>
</tr>
<tr>
<td>Stick 500 Hz</td>
<td></td>
<td></td>
<td>Rudder 1 kHz</td>
<td>Rudder</td>
</tr>
</tbody>
</table>

Task Classification (2)

- **Non-Periodic** or aperiodic tasks = all tasks that are not periodic, also known as Event-driven, their activations may be generated by external interrupts

- **Sporadic tasks** = aperiodic tasks with minimum interarrival time \( T_{\text{min}} \) (often with hard deadline)
  - worst case = periodic tasks with period \( T_{\text{min}} \)

Task states (1)

- Ready
- Running
- Waiting(blocked/suspended) ...
- Idling
- Terminated

Task states (2)

Task states (Ada, delay)
Task states (Ada95)

Basic functions of RT OS

Task managment

Basic functions of RT OS
Interrupts
- Interrupt: environmental event that demands attention
  - Example: "byte arrived" interrupt on serial channel
- Interrupt routine: piece of code executed in response to an interrupt

Interrupt Service Routines
- Most interrupt routines:
  - Copy peripheral data into a buffer
  - Indicate to other code that data has arrived
  - Acknowledge the interrupt (tell hardware)
- Longer reaction to interrupt performed outside interrupt routine
- E.g., causes a process to start or resume running

Handling an Interrupt
1. Normal program execution
2. Interrupt occurs
3. Processor state saved
4. Interrupt routine runs
5. Interrupt routine terminates
6. Processor state restored
7. Normal program execution resumes

Interrupt Handling
- Types of interrupts
  - Asynchronous (or hardware interrupt): by hardware event (timer, network card...) the interrupt handler runs as a separated task in a different context.
  - Synchronous (or software interrupt, or a trap): by software instruction (call in ARM, int in Intel 80x86), a divide by zero, a memory segmentation fault, etc. The interrupt handler runs in the context of the interrupting task
- Interrupt latency
  - The time delay between the arrival of interrupt and the start of corresponding ISR.
  - Modern processors with multiple levels of caches and instruction pipelines that need to be reset before ISR can start might result in longer latency.
  - The ISR of a lower-priority interrupt may be blocked by the ISR of a high-priority interrupt

Basic functions of RT OS
- Task management
- Interrupt handling
- Memory management
- Exception handling
- Task synchronization
- Task scheduling
- Time management

Memory Management/Protection
- Standard methods
  - Block-based, Paging, hardware mapping for protection
- No virtual memory for hard RT tasks
  - Lack all pages in main memory
- Many embedded RTs do not have memory protection – tasks may access any blocks – Hope that the whole design is proven correct and protection is unnecessary
  - to achieve predictable timing
  - to avoid time overheads
- Most commercial RTOS provide memory protection as an option
  - Run into "fail-safe" mode if an illegal access trap occurs
  - Useful for complex reconfigurable systems
Basic functions of RT OS

- Task management
- Interrupt handling
- Memory management
- Exception handling
- Task synchronization
- Task scheduling
- Time management

Exception handling

- Exceptions e.g. missing deadline, running out of memory, timeouts, deadlocks
  - Error at system level, e.g. deadlock
  - Error at task level, e.g. timeout
- Standard techniques:
  - System calls with error code
  - Watch dog
  - Fault-tolerance (later)
- However, difficult to know all scenarios
  - Missing one possible case may result in disaster
  - This is one reason why we need Modelling and Verification

Watch-dog

- A task, that runs (with high priority) in parallel with all others
- If some condition becomes true, it should react ...

  ```
  Loop
  begin
  ....
  end
  until condition
  ```
- The condition can be an external event, or some flags
- Normally it is a timeout

Example

- Watch-dog (to monitor whether the application task is alive)

  ```
  Loop
  if flag==1 then
  {
  next :=system_time;
  flag :=0
  }
  else if system_time> next+20s then WARNING;
  sleep(100ms)
  end loop
  ```

Task Synchronization

- Synchronization primitives
  - Semaphore: counting semaphore and binary semaphore
    - A semaphore is created with initial_count, which is the number of allowed holders of the semaphore lock. (initial_count=1: binary sem)
    - Sem_wait will decrease the count; while sem_signal will increase it.
    - A task can get the semaphore when the count > 0; otherwise, block on it.
  - Mutex: similar to a binary semaphore, but mutex has an owner.
    - A semaphore can be "welled for" and "unwelled" by any task, while only the task that has taken a mutex is allowed to release it.
  - Spinlock: lock mechanism for multi-processor systems,
    - A task wanting to get spinlock has to get a lock shared by all processors.
  - Read/write locks: protect from concurrent write, while allow concurrent read
    - Many tasks can get a read lock; but only one task can get a write lock.
    - Before a task gets the write lock, all read locks have to be released.
  - Barrier: to synchronize a list of tasks,
    - they should wait until all of them have reached a certain "barrier."

- Application-task
  - flag:=1 .... computing something .... flag:=1 .... flag:=1 ....
Task Synchronization

- Challenges for RTOS
  - Critical section (data, service, code) protected by lock mechanism e.g. Semaphore etc. In a RTOS, the maximum time a task can be delayed because of locks held by other tasks should be less than its timing constraints.
  - Race condition – deadlock, livelock, starvation: Some deadlock avoidance/prevention algorithms are too complicated and indeterminate for real-time execution. Simplicity is preferred, like allow each task to hold only one resource.
  - Priority inversion: using priority-based task scheduling and locking primitives should know the “priority inversion” danger: a medium-priority job runs while a high-priority task is ready to proceed.

IPC: Data exchanging

- Semaphore
- Shared variables
- Bounded buffers
- FIFO
- Mailbox
- Message passing
- Signal

Semaphore, Dijkstra 60s

- A semaphore is a simple data structure with
  - a counter
  - the number of “resources”
  - binary semaphore
  - a queue
  - Tasks waiting

and two operations:

- P(S): get or wait for semaphore
- V(S): release semaphore

Implementation of Semaphores: SCB

- SCB: Semaphores Control Block

<table>
<thead>
<tr>
<th>Counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queue of TCBs (tasks waiting)</td>
</tr>
<tr>
<td>Pointer to next SCB</td>
</tr>
</tbody>
</table>

The queue should be sorted by priorities (Why not FIFO?)

Implementation of Semaphores: P-operation

- P(scb):
  - Disable-interrupt;
  - If scb.counter>0 then
    - scb.counter := -1;
  - end then
  - else
    - save-context();
    - current-tcb.state := blocked;
    - insert(current-tcb, scb.queue);
    - dispatch();
  - end else
  - Enable-interrupt

Implementation of Semaphores: V-operation

- V(scb):
  - Disable-interrupt;
  - If not-empty(scb.queue) then
    - tcb := get-first(scb.queue);
    - tcb.state := ready;
    - insert(tcb, ready-queue);
    - save-context();
    - schedule(); /* dispatch invoked*/
    - load-context();
  - end then
  - else scb.counter += 1;
  - end else
  - Enable-interrupt
Advantages with semaphores

- Simple (to implement and use)
- Exists in most (all?) operating systems
- It can be used to implement other synchronization tools
  - Monitors, protected data type, bounded buffers, mailbox etc

Exercise/Questions

- Implement Mailbox by semaphore
  - Send(mbox, receiver, msg)
  - Get-msg(mbox,receiver,msg)
- How to implement hand-shaking communication?
  - V(S1)P(S2)
  - V(S2)P(S1)
- Solve the read-write problem
  - (e.g max 10 readers, and at most 1 writer at a time)

Disadvantages (problems) with semaphores

- Deadlocks
- Loss of mutual exclusion
- Blocking tasks with higher priorities (e.g. FIFO)
- Priority inversion!

Priority inversion problem

- Assume 3 tasks: A, B, C with priorities A_p<B_p<C_p
- Assume semaphore: S shared by A and C
- The following may happen:
  - A gets S by P(S)
  - C wants S by P(S) and blocked
  - B is released and preempts A
  - Now B can run for a long long period ......
  - A is blocked by B, and C is blocked by A
  - So C is blocked by B
- The above scenario is called ‘priority inversion’
- It can be much worse if there are more tasks with priorities in between B_p and C_p, that may block C as B does!

Solution?

- Task A with low priority holds S that task C with highest priority is waiting.
- Task A can not be forced to give up S, but A can be preempted by B because B has higher priority and can run without S

So the problem is that ‘A can be preempted by B’

- Solution 1: no preemption (an easy fix) within CS sections
- Solution 2: high A’s priority when it gets a semaphore shared with a task with higher priority! So that A can run until it release S and then gets back its own priority

Resource Access Protocols

- Highest Priority Inheritance
  - Non preemption protocol (NPP)
- Basic Priority Inheritance Protocol (BIP)
- POSIX (RT OS standard) mutexes
- Priority Ceiling Protocols (PCP)
- Immediate Priority Inheritance
  - Highest Locker’s priority Protocol (HLP)
  - Ada95 (protected object) and POSIX mutexes
Basic functions of RT OS

- Task management
- Interrupt handling
- Memory management
- Exception handling
- Task synchronization

Task scheduling
- Time management

Task states

- Ready
- Running
- Blocked
- Preemption
- Dispatch
- Wait
- Signal
- Activate
- Idling
- Terminate
- Timeout
- Delay
- Signal
- Terminate

Priority-based Scheduling

- Typical RTOS based on fixed-priority preemptive scheduler
- Assign each process a priority
- At any time, scheduler runs highest priority process ready to run
- Process runs to completion unless preempted

Scheduling algorithms

- Sort the READY queue according to
  - Priorities (HPF)
  - Execution times (SCF)
  - Deadlines (EDF)
  - Arrival times (FIFO)

- Classes of scheduling algorithms
  - Preemptive vs non preemptive
  - Off-line vs on-line
  - Static vs dynamic
  - Event-driven vs time-driven

Task Scheduling

- Scheduler is responsible for time-sharing of CPU among tasks.
- A variety of scheduling algorithms with predictable behaviors exist.
- The general trade-off: the simplicity and the optimality.

Challenges for an RTOS

- New tasks
- Preemption
- Dispatching
- Running
- Termination

Schedulability

- A schedule is an ordered list of tasks (to be executed) and a schedule is feasible if it meets all the deadlines
- A queue (or set) of tasks is schedulable if there exists a schedule such that no task may fail to meet its deadline

- How do we know all possible queues (situations) are schedulable? we need task models (next lecture)
**Priority-based scheduling in RTOS**

- **Static priority**
  - A task is given a priority at the time it is created, and it keeps this priority during the whole lifetime.
  - The scheduler is very simple, because it looks at all wait queues at each priority level, and starts the task with the highest priority to run.
- **Dynamic priority**
  - The scheduler becomes more complex because it has to calculate task's priority on-line, based on dynamically changing parameters.
  - Earliest-deadline-first (EDF) --- A task with a closer deadline gets a higher scheduling priority.
  - Rate-monotonic scheduling --- A task gets a higher priority if it has to run more frequently.
  - This is a common approach in case that all tasks are periodic. So, a task that has to run every n milliseconds gets a higher priority than a task that runs every m milliseconds when n>m.

**Basic functions of RT OS**

- Task management
- Interrupt handling
- Memory management
- Exception handling
- Task synchronization
- Task scheduling
- Time management

**Time management**

- A high resolution hardware timer is programmed to interrupt the processor at fixed rate — Time interrupt.
- Each time interrupt is called a system tick (time resolution):
  - Normally, the tick can vary in microseconds (depend on hardware)
  - The tick may (not necessarily) be selected by the user
  - All time parameters for tasks should be the multiple of the tick
  - Note: the tick may be chosen according to the given task parameters
  - System time = 32 bits
    - One tick = 1 usec: your system can run 50 days
    - One tick = 50ms: your system can run 1000 days = 2.7 years
    - One tick = 200ms: your system can run 2500 days = 7 years

**Time interrupt routine**

- Save the context of the task in execution
  - Increment the system tick by 1, if current time > system lifetime, generate a timing error
  - Update timers (reduce each counter by 1)
  - A queue of timers
  - Activation of periodic tasks in idling state
  - Schedule again -- call the scheduler
  - Other functions e.g.
    - (Remove all tasks terminated -- deallocate data structures e.g TCBS)
    - (Check if any deadline misses for hard tasks, monitoring)
  - load context for the first task in ready queue

**Features of current RTOS: SUMMARY**

- Multi-tasking
- Priority-based scheduling
  - Application tasks should be programmed to suit ...
- Ability to quickly respond to external interrupts
- Basic mechanisms for process communication and synchronization
- Small kernel and fast context switch
- Support of a real time clock as an internal time reference
Existing RTOS: 4 categories

- Priority based kernel for embedded applications e.g. OSE, VxWorks, QNX,VRTX32, pSOS ... Many of them are commercial kernels
  - Applications should be designed and programmed to suit priority-based scheduling e.g. deadlines as priority etc.
- Real Time Extensions of existing time-sharing OS e.g. Real time Linux, Real-time NT by e.g. locking RT tasks in main memory, assigning highest priorities etc.
- Research RT Kernels e.g. SHARK, TinyOS ...
- Run-time systems for RT programming languages e.g. Ada, Erlang, Real-Time Java ...

RT Linux: an example

RT Linux is an operating system, in which a small real-time kernel co-exists with standard Linux kernel:

- The real-time kernel sits between standard Linux kernel and the I/O. The standard Linux kernel sees this RT layer as actual I/O.
- The real-time kernel intercepts all hardware interrupts. Only for those RTLinux-related interrupts, the appropriate ISR is run. All other interrupts are held and passed to the standard Linux kernel as software interrupts when the standard Linux kernel runs.
- The real-time kernel assigns the lowest priority to the standard Linux kernel. Thus the real-time tasks will be executed in real-time.
- User can create realtime tasks and achieve correct timing for them by deciding on scheduling algorithms, priorities, execution freq, etc.
- Realtime tasks are privileged (that is, they have direct access to hardware), and they do NOT use virtual memory.

Scheduling

- Linux contains a dynamic scheduler
- RT-Linux allows different schedulers
  - EDF (Earliest Deadline First)
  - Rate-monotonic scheduler
  - Fixed-priority scheduler

Time Resolution

- RT tasks may be scheduled in microseconds
- Running RT Linux V3.0 Kernel 2.2.19 on the 486 allows stable hard real-time operation:
  - 17 nanoseconds timer resolution.
  - 6 microseconds interrupt response time (measured on interrupts on the parallel port).
- High resolution timing functions give nanosecond resolution (limited by the hardware only)

Linux v.s. RTLinux

- Linux Non-real-time Features
  - Linux scheduling algorithms are not designed for real-time tasks
  - But provide good average performance or throughput
  - Unpredictable delay
  - Uninterruptible system calls, the use of interrupt disabling, virtual memory support (context switch may take hundreds of milliseconds).
- Linux Timer resolution is coarse, 10ms
- Linux Kernel is non-preemptible.

- RTLinux Real-time Features
  - Support real-time scheduling: guarantee hard deadlines
  - Predictable delay (by its small size and limited operations)
  - Finer time resolution
  - Pre-emptible kernel
  - No virtual memory support