Today’s topic: RTOS

(Real Time Operating Systems)
Operating System Provides

- Hardware abstraction layer
  - Interrupt processing
  - Device drivers: I/O Libraries - 10 times bigger than a minimal OS
    e.g. the firmware on an automotive ECU is 10% RTOS and 90% device drivers
  - API – Application Programming Interface -- System calls
- Environment for executing program(s)
  - Process/thread management, process communication
- Stable storage
  - Filesystems
Overall Structure of Computer Systems

<table>
<thead>
<tr>
<th>Application Program</th>
<th>Application Program</th>
<th>Application Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>API/OS User Interface/Shell</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Filesystem and Disk management</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS kernel</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hardware</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Requirements on RTOS

- **Determinism**
  - Deterministic system calls

- **Responsiveness (quoted by vendors)**
  - Fast context switch (#instructions)
  - Short interrupt latency (#CPU cycles)

- **Support for timely concurrent processing**
  - Real-time
  - Multi-tasking
  - Synchronization

- **User control over OS policies**
  - CPU scheduling (e.g. many priority levels)
  - Support for memory management
    - E.g. pages locked in main memory

- **Controlled code size (~1MB)**
  - E.g. Micro kernel, Contiki, 1000 loc, OSE small kernel, 2k
Micro-kernel architecture

External interrupts

System calls

Hardware/software exceptions

Clock interrupts

Immediate Interrupt services

Case of

Exception handling

Create task
Suspend task
Terminate task
Create timer
Sleep-timer
Timer-notify
Other system calls

Time services

Kernel

Scheduling
Typical footprints

- POSIX
- PSE54
- POSIX
- PSE51/52
- OSEK/VDX
- Linux
- real-time
- SHaRK
- eCOS
- VXworks
- L4
- threadX
- ERIKA
- OSE
- tinyOS
- Contiki

code size

- 1000kb
- 100kb
- 10kb
- 1kb
Existing RTOS: 4 categories

- **Priority based kernel for embedded applications** e.g. POSIX (IEEE standard 1003.1-1988, UNIX based), OSE (cell phone), VxWorks (space and robotics), OSEK/VDX (automotive), QNX (automotive and multimedia) .... Many of them are commercial kernels
  - Applications should be designed and programmed to suite 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. L4 (some of the kernels are commercial), SHARK, TinyOS, ...

- **Run-time systems** for RT programming languages e.g. Ada, Erlang, Real-Time Java ...
POSIX: an example RTOS

- POSIX: **Portable Operating System Interface for Unix**
  - IEEE standard 1003.1-1988
  - A typical footprint around 1M
- Use **profiles** to support subsets of the standard
- A profile lists a set of services typically used in a given environment
- POSIX real time profiles are specified by the ISO/IEEE standard 1003.13
POSIX 1003.13 profiles

- PSE51  minimal real-time system profile (around 50-150 Kbytes)
  - no file system
  - no memory protection
  - Mono-process multi-thread kernel
- PSE52  real-time controller system profile
  - PSE51 + file system + asynchronous I/O
- PSE53  dedicated real-time system profile
  - PSE51 + process support and memory protection
- PSE54  multi-purpose real-time system profile
  - PSE53 + file system + asynchronous I/O
RT Linux: an example RTOS
Linux v.s. RTLinux

- **Linux Non-real-time Features**
  - Dynamic scheduling, not designed for real-time tasks -- good average performance or throughput
  - Non-preemptible.
  - Un-predictable delays
    - Uninterruptible system calls, the use of interrupt disabling, virtual memory support (context switch may take hundreds of microsecond).
  - “Coarse” timer resolution 1-10ms (e.g. 4ms)

- **RTLinux Real-time Features**
  - Real-time scheduling (EDF, RMS, FPS): guarantee *hard deadlines*
  - Pre-emptible kernel
  - Predictable delays (by its small size and limited operations)
  - “Finer” timer resolution (can be nanos/too much overheads, 50micros ...)
OSEK /VDX: an example RTOS (automotive applications)

- Founded in May 1993
  - Joint project of the German automotive industry
  - OSEK (German): Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug
  - Initial project partners: BMW, Bosch, DaimlerChrysler, Opel, Siemens, VW

- 1994 PSA & Renault joined OSEK
  - Similar project of the French automotive industry
  - VDX → Vehicle Distributed eXecutive

- OSEK/VDX resulted from the merge of the two projects

- [http://www.osek-vdx.org](http://www.osek-vdx.org)
Basic functions of RTOS

- **Time management**
  - Task management
  - Interrupt handling
  - Memory management
  - Exception handling
  - Task scheduling
  - Task synchronization
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
  - The tick may be selected/configured by the user
  - All time parameters for tasks should be the multiple of the tick
  - OBS: too fine → too much overheads, too coarse → too long latency
- System time = 32 bits
  - One tick = 1ms: your system can run 50 days
  - One tick = 20ms: your system can run 1000 days = 2.5 years
  - One tick = 50ms: your system can run 2500 days= 7 years
- The Lego processor/ARM-7: 48MHZ (automotive systems/Bosch, 200MHZ)
  - 48 000 000 CPU cycles (~12 000 000 instructions!) per second
  - ~12 000 instructions/1ms (per tick, if time resolution is 1ms)
Time interrupt routine

- **Save the context of the task in execution**
  - Increment the system time 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**
Basic functions of RTOS kernel

- Time management
- **Task management**
  - Interrupt handling
  - Memory management
  - Exception handling
  - Task scheduling
  - Task synchronization
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**
Periodic tasks

- Described with 3 parameters (C,D,T) where
  - C = resource budget
  - 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
Sporadic tasks

- Described with 3 parameters \((C, D, T_{\text{min}})\) where
  - \(C\) = resource budget
  - \(D\) = deadline
  - \(T_{\text{min}}\) = minimum interarrival time

Also known as \textit{Event-driven}: their activations are generated by interrupts, but separated with minimum interarrival time \(T_{\text{min}}\). In the worst case, it is the same as a periodic task with period \(T_{\text{min}}\).
Timing Constraints

- **Hard real-time** — systems where it is absolutely imperative that responses occur within the required deadline. E.g. Flight control systems, automotive systems, robotics etc.

- **Soft real-time** — systems where deadlines are important but which will still function correctly if deadlines are occasionally missed. E.g. Banking system, multimedia etc.

A single system may have both hard and soft real-time tasks. In reality many systems will have a cost function associated with missing each deadline.
Timing Constraints

- **Hard deadline**: Constant value.
- **Soft deadline**: Value decreases over time.
- **On-time deadline**: Value peaks at the deadline.
- **No deadline**: Value remains constant.
Task states (1)

- Ready
- Running
- Waiting/blocked/suspended ...
- Idling
- Terminated
Task states (2)
Task states (Ada, delay)
Task states (Ada95)

Declared (e.g. in Ada)

- **Ready**
  - Activated
  - Preemption
  - Wait

- **Blocked**
  - Idling
  - Preemption
  - Signal

- **Running**
  - Dispatch
  - Sleep
  - Terminate

- **Idling**
  - Timeout

- **Terminate**
TCB (Task Control Block)

- Id
- Task state (e.g. Idling)
- Task type (hard, soft, background ...)
- Priority
- Other Task parameters
  - period
  - comuting time (if available)
  - Relative deadline
  - Absolute deadline
- Context pointer
- Pointer to program code, data area, stack
- Pointer to resources (semaphors etc)
- Pointer to other TCBs (preceeding, next, waiting queues etc)
Task management

- Task creation: create a new TCB
- Task termination: remove the TCB
- Change Priority: modify the TCB
- ...
- State-inquiry: read the TCB
Basic functions of RTOS

- Time management
- Task management
- **Interrupt handling**
  - Memory management
  - Exception handling
  - Task scheduling
  - Task synchronization
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
Basic functions of RTOS

- Time management
- Task management
- Interrupt handling

**Memory management**
- Exception handling
- Task scheduling
- Task synchronization
Memory Management/Protection

- Standard methods
  - Block-based, Paging, hardware mapping for protection
- **No virtual memory** for hard RT tasks
  - Lock all pages in main memory
- Many embedded RTS do not have memory protection – tasks may access any block – **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 RTOS

- Time management
- Task management
- Interrupt handling
- Memory management

**Exception handling**
- Task scheduling
- Task synchronization
Exception handling

- Exceptions e.g. missing deadline, running out of memory, timeouts, deadlocks, divide by zero, etc.
  - Error at system level, e.g. deadlock
  - Error at task level, e.g. timeout

- Standard techniques:
  - System calls with error code
  - Watch dog
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)
  Every 100ms, it should check: if flag=1 then OK
  otherwise if flag has been 0 for 20s, send out a warning

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

- Application-task
  - flag:=1 ... ... computing something ... ... flag:=1 ..... flag:=1 ....
Basic functions of RTOS

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

**Task scheduling**
- Task synchronization
Task states

- Ready
- Blocked
- Idling
- Running
- Terminate

- Activate
- timeout
- delay
- preemption
- Dispatch
- signal
- wait
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
Basic functions of RTOS

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

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. \( \text{initial\_count}=1 \): 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 “waited for” and “signaled” 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.

- **Barrier**: to synchronize a lot of tasks,
  - they should wait until all of them have reached a certain “barrier.”
Potential problems in task synchronization

- **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**.

- **Deadlock, livelock, starvation** Some deadlock avoidance/prevention algorithms are too complicate and indeterministic for real-time execution. Simplicity preferred, e.g.
  - all tasks always take locks in the same order.

- **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.