Today’s topic: **RTOS**

### 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, Windows</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>
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
- Environment for executing programs
  - Support for multitasking/concurrency
  - Mechanisms for Synchronization/Communication
- Filesystems/Stable storage
- API – Application Programming Interface

Requirements on RTOS

- Determinism
  - Deterministic system calls
- Responsiveness (quoted by vendors)
  - Fast process/thread switch
  - Fast interrupt response
- Support for concurrency and real-time
  - Multi-tasking
  - Real-time
  - Synchronization
- User control over OS policies
  - Mainly scheduling, many priority levels
  - Memory support (especially embedded)
    - E.g. pages locked in main memory
    - E.g. cache partitioning/coloring on multicore
- Controlled code size
  - E.g. Micro kernel, Contiki, 1000 loc, OSE small kernel, 2k
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 (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. SHARK, TinyOS ...

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

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

- The RT kernel sits between standard Linux kernel and HW.
  - the standard Linux Kernel sees this RT layer as actual HW

- The RT kernel intercepts all hardware interrupts
  - Only for those RT Linux-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 RT kernel assigns the lowest priority to the standard Linux kernel.
  - Thus the realtime tasks will be executed in real-time

- User can create RT tasks and achieve correct timing
  - The user can decide on scheduling algorithms, priorities, execution freq, etc.
  - RT tasks have direct access to hardware, and they do NOT use virtual memory.
RT Linux: an example RTOS

Linux v.s. RTLinux

- **Linux Non-real-time Features**
  - Linux scheduling algorithms (dynamic scheduler) 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 microsecond).
  - Linux Timer resolution is coarse, e.g. 10ms
  - Linux Kernel is Non-preemptible.

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

Basic functions of RT OS

- 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 (depend on hardware)
  - The tick may 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 = 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

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

What is a “Task”?
Process vs. Thread & Task

- A process is “a program in execution”, corresponding to one application e.g. “mail browser”, “web browser” etc.

- A thread is a “lightweight” process, in the sense that different threads share the same address space (within the same application), 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.

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 triple: (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
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 classification (3)

- **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.
Classification of RTS’s

Task states (1)

- Ready
- Running
- Waiting/blocked/suspended ...
- Idling
- Terminated
Task states (2)

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

- Ready
- Blocked
- Runnig
- Dispatch
- Idling
- Terminate
- preemption
- signal
- wait
- timeout
- declared
- created
- Activate
- Idling

TCB (Task Control Block)

- Id
- Task state (e.g. Idling)
- Task type (hard, soft, background ...)
- Priority
- Other Task parameters
  - period
  - computing 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 (preceding, next, waiting queues etc)
Basic functions of RT OS

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

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 RT OS

- 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 RT OS

- 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 commercialRTOS 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

- 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

- **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)
  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;
        flag :=0
      }
    else if system_time> next+20s then WARNING;
    sleep(100ms)
  end loop
- Application-task
  - flag:=1 ... ... computing something ... ... flag:=1 ..... flag:=1 ....
Basic functions of RT OS

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

---

**Task states**

![Task States Diagram]

- Ready
- Blocked
- Running
- Idling
- Terminate

- Activate
- Dispatch
- Signal
- Wait
- Preemption
- Delay
- Timeout

---
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

Challenges for RTOS

- Different performance criteria
  - GPOS: maximum average throughput
  - RTOS: deterministic behavior

- Optimal schedules difficult to find
  - Hard to get complete knowledge

- How to guarantee Timing Constraints?
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 feasible schedule.

How do we know all possible queues (situations) are schedulable?
→ Scheduling theory (later lectures)

Basic functions of RT OS

- Time management
- Task management
- Interrupt handling
- Memory management
- Exception handling
- Task scheduling
- Task synchronization
Challenges for RTOS (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.

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 "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."
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();
    load-context();
  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!