Language Support for Real Time Programming

- Concurrency (Ada tasking)
- Communication & synchronization (Ada Rendezvous)
- Consistency in data sharing (Ada protected data type)
- Real time facilities (Ada real time packages)

What to do if we don’t have Ada?

Today’s topic: OS Support
for Real Time Programming

Why OS?

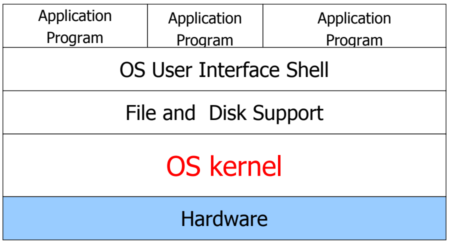
- To run one single program is easy
- Two, or three is fine, but more than five would be difficult without help
- you need to take care
  - memory areas
  - Program counters
  - Scheduling, run one instruction each?
  - ....
  - Communication/synchronization
  - Device drivers
- OS is nothing but a program offering the functions needed in all applications e.g. the start of Enea’s OSE

An example nano-kernel:
a single task cyclic executive

```
Setup-timer
c=0;
While (1) {suspend until timer expires
        c++;
        compute tasks due every cycle
        if ((c%2)==0) compute tasks due every 2nd cycle
        if ((c%3)==0) compute tasks due every 3rd cycle
       }
```

Overall Stucture of Computer Systems

In most cases, RTOS=OS Kernel



Application Program	Application Program	Application Program
OS User Interface Shell		
File and Disk Support		
OS kernel		
Hardware		

Basic functions of OS

- Process mangement
- Memory management
- Interrupt handling
- Exception handling
- Process Synchronization (IPC)
- Process schedulling
- Disk management

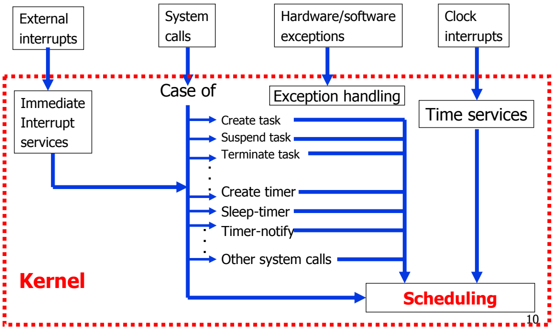
Process, Thread and Task

- A process is a program in execution ...
- Starting a new process is a heavy job for OS: memory has to be allocated, and lots of data structures and code must be copied.
    - memory pages (in virtual memory and in physical RAM) for code, data, stack, heap, and for file and other descriptors; registers in the CPU; queues for scheduling; signals and IPC; etc.
- A thread is a "*lightweight*" process, in the sense that different threads share the same address space.
    - They *share global and "static" variables*, file descriptors, signal bookkeeping, code area, and heap, but they have own thread status, program counter, registers, and stack.
    - ***Shorter creation and context switch times, and faster IPC.***
        - to save the state of the currently running task (registers, stack pointer, PC, etc.), and to restore that of the new task.
- Tasks are mostly threads

Basic functions of RTOS kernel

- Task mangement
- 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



External interrupts

System calls

Hardware/software exceptions

Clock interrupts

Immediate Interrupt services

Case of

Exception handling

Time services

Create task

Suspend task

Terminate task

Create timer

Sleep-timer

Timer-notify

Other system calls

Kernel

Scheduling

Basic functions of RTOS kernel

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

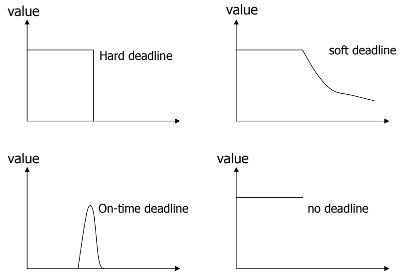
Task: basic notion in RTOS

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

Task Classification by deadlines (i.e. timing constraints)

- Hard RT task must be computed within the given deadline (otherwise system failure), e.g ABS control
  - The course will mainly deal with Hard RT tasks
- Soft RT task may be computed after the given deadline e.g. Multi-media, the web etc
- On-time RT tasks (e.g alarm, robotics)
- Non RT task (background tasks)

RT task characterization



Task Classification by release rate (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

---

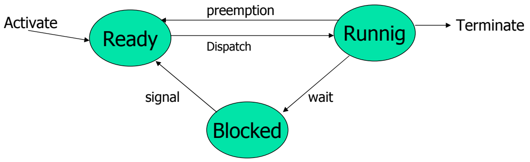
Task Classification by release rate (2)

- Non-Periodic or aperiodic tasks = all tasks that are not periodic, also known as Event-driven, their activations are generated by interrupts
- Sporadic tasks = aperiodic tasks with minimum interarrival time T_{min} (often with hard deadline)
  - worst case = periodic tasks with period T_{min}

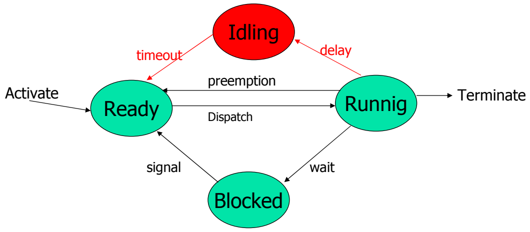
Task states (1)

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

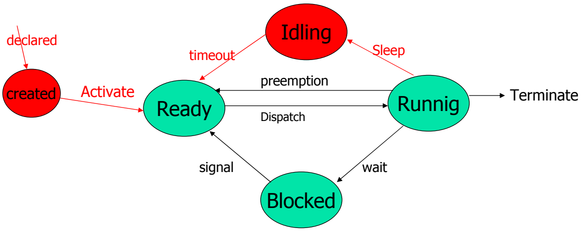
Task states (2)



Task states (Ada, delay)



Task states (Ada95)



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 (preceding, next, waiting queues etc)

Basic functions of RT OS

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

Task managment

- Task creation: create a newTCB
- Task termination: remove the TCB
- Change Priority: modify the TCB
- ...
- State-inquiry: read the TCB

Task mangement

- Challenges for an RTOS
  - Creating an RT task, it has to get the memory without delay: this is difficult because memory has to be allocated and a lot of data structures, code seqment must be copied/initialized
  - The memory blocks for RT tasks must be locked in main memoery to avoid access latencies due to swapping
  - Changing run-time priorities is dangerous: it may change the run-time behaviour and predictability of the whole system

Basic functions of RT OS

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

Interrupt Handling

- **Types of interrupts**
  - Asynchronous (or hardware interrupt) by hardware event (timer, network card …) the interrupt handler as a separated task in a different context.
  - Synchronous (or software interrupt, or a trap) by software instruction (swi 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
- **Challenges in RTOS**
  - ***Interrupt latency***
    - The time 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.
  - ***Interrupt enable/disable***
    - The capability to enable or disable ("mask") interrupt individually.
  - ***Interrupt priority***
    - to block a new interrupt if an ISR of a higher-priority interrupt is still running.
    - the ISR of a lower-priority interrupt is preempted by a higher-priority interrupt.
    - The priority problems in task scheduling also show up in interrupt handling.

Interrupt Handling

- ***Interrupt nesting***
  - an ISR servicing one interrupt can itself be pre-empted by another interrupt coming from the same peripheral device.
- ***Interrupt sharing***
  - allow different devices to be linked to the same hardware interrupt.
    - check a status register on each of the devices that share the interrupt
    - calling in turn all ISRs that users have registered with this IRQ.

Basic functions of RT OS

- Task mangement
- 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
  - Lock 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 unneccessary
  - to achive 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 mangement
- 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 senarios
  - 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
```

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

Basic functions of RT OS

- Task mangement
- Interrupt handling
- Memory management
- Exception handling
- Task synchronization
- Time management
- CPU 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. (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.
  - ***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 lot of tasks,
    - they should wait until *all* of them have reached a certain "barrier."

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 complicate and indeterministic for real-time execution. Simplicity is preferred, like
    - *all tasks* always take locks in the *same order*.
    - 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 highpriority task is ready to proceed.*

IPC: Data exchanging

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

Semaphore is the most primitive and widely used construct for
Synchronization and communicatioin in all operating systems

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

Counter
Queue of TCBs (tasks waiting)
Pointer to next SCB

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 !

Priority inversion problem

- Assume 3 tasks: A, B, C with priorities Ap<Bp<Cp
- 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 senario is called 'priority inversion'
- It can be much worse if there are more tasks with priorities in between Bp and Cp, that may block C as B does!

Solution?

- Task A with low priority holds S that task C with highest priority is waiting.
- Tast 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

---

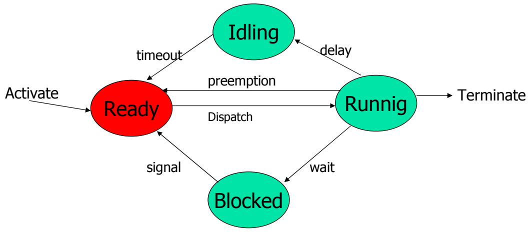
Resource Access Protocols

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

Basic functions of RT OS

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

Task states



Scheduling algorithms

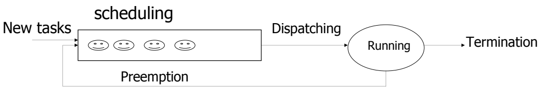
- Sort the READY queue acording to
  - Priorities (HPF)
  - Execution times (SCF)
  - Deadlines (EDF)
  - Arrival times (FIFO)
- Classes of scheduling algorithms
  - Preemptive vs non preemptive
  - Off-line vs on-line
  - Static vs dynamic
  - Event-driven vs time-driven

Task Scheduling

- **Scheduler is responsible for time-sharing of CPU** among tasks.
  - A variety of scheduling algorithms have been explored and implemented.
  - The general trade-off: the simplicity and the optimality.
- **Challenges for an RTOS**
  - ***Different performance criteria***
    - GPOS: maximum *average* throughput,
    - RTOS: *deterministic* behavior (also small memory usage, low power consumption ...)
  - ***A theoretically optimal schedule does not exist***
    - Hard to get *complete knowledge* – task requirements and hard properties
    - the requirements can be *dynamic* (i.e., *time varying*) – adaptive scheduling
  - *How to garuantee 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 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 mangement
- Interrupt handling
- Memory management
- Exception handling
- Task synchronization
- Task scheduling
- Time management

Time mangement

- 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 = 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 RT OS

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

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 kernal and fast context switch
- Support of a real time clock as an internal time reference

Existing RTOS: 4 categories

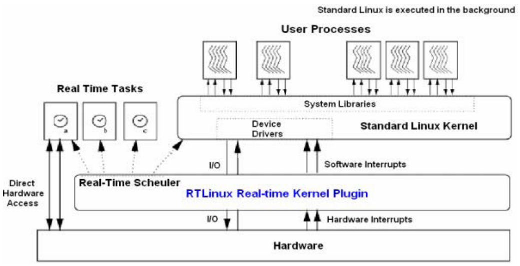
- Priority based kernel for embbeded applications e.g. OSE, VxWorks, QNX, VRTX32, pSOS .... Many of them are commercial kernels
  - Applications should be designed and programmed to suite priority-based scheduling e.g deadlines as priority etc
- Real Time Extensions of existing time-sharing OS e.g. Real time Linux, Real time NT, real time Mach etc by e.g locking RT tasks in main memory, assigning highest priorities etc
- Research RT Kernels e.g. MARS (Vienna univ), Spring (univ of Massachusetts) ...
- Run-time systems for RT programmingn 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 *h/w*. The standard Linux Kernel sees this RT layer as actual h/w.
- – 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 realtime 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*.

RT Linux



Scheduling

- **Linux contains a dynamic scheduler**
- **RT-Linux allows different schedulers**
  - EDF (Earliest Deadline First)
  - Rate-monotonic scheduler
  - Fixed-prioritiy 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 microsecond).
  - – 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