REAL TIME OPERATING SYSTEMS

Ege Üniversitesi
Müh. Fakültesi Bilgisayar Mühendisliği Bölümü
Gerçek Zamanlı İşletim Sistemleri
2014
Yar. Doç. Dr. Mustafa Engin

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/semaphors
  - Device drivers
- OS is a program offering the common services needed in all applications

Operating System Provides

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

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

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

```
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

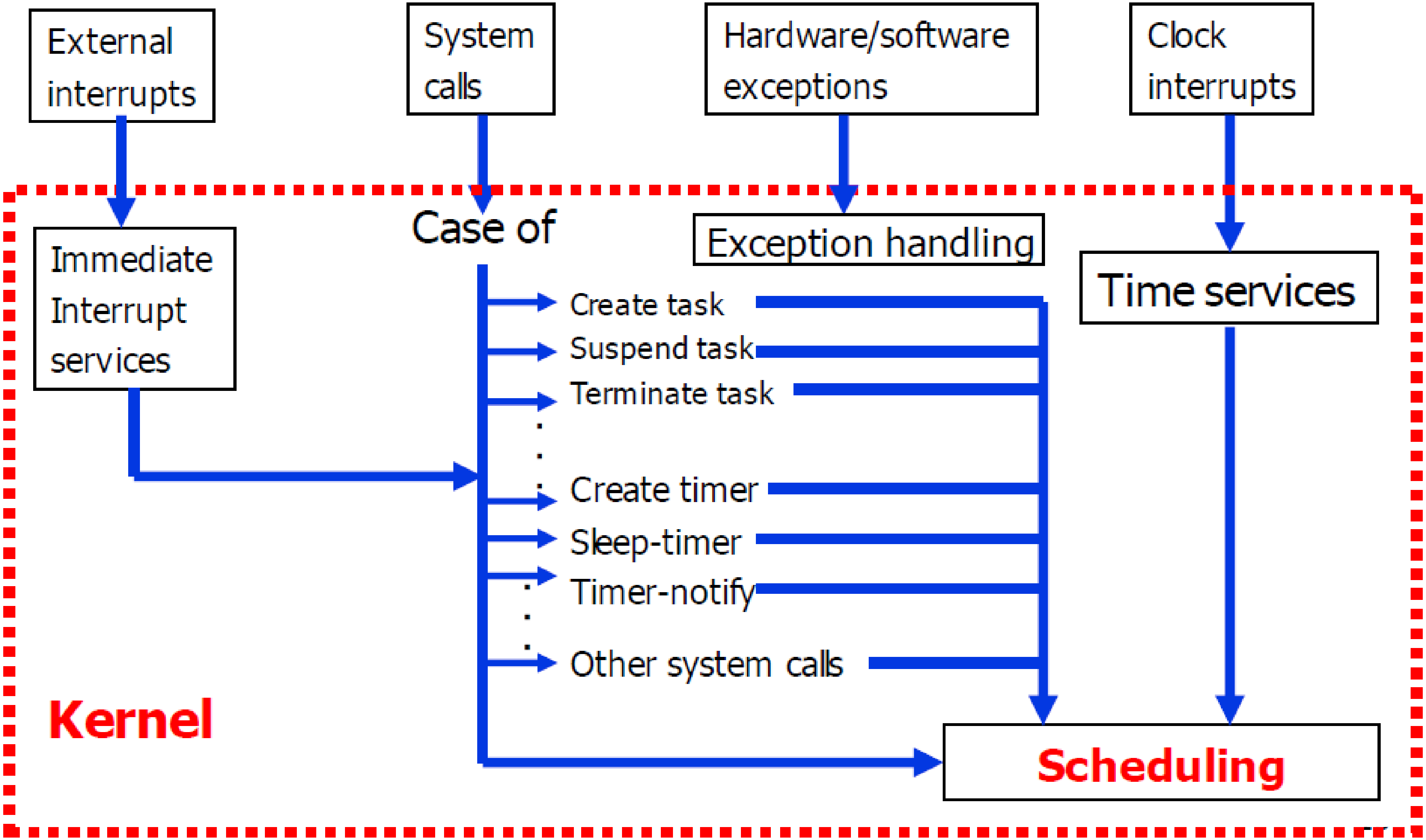
Process, Thread and Task

- 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 (Inter-process communication)
- 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
  - A void priority inversion
- Task scheduling
- Time management

Micro-kernel architecture

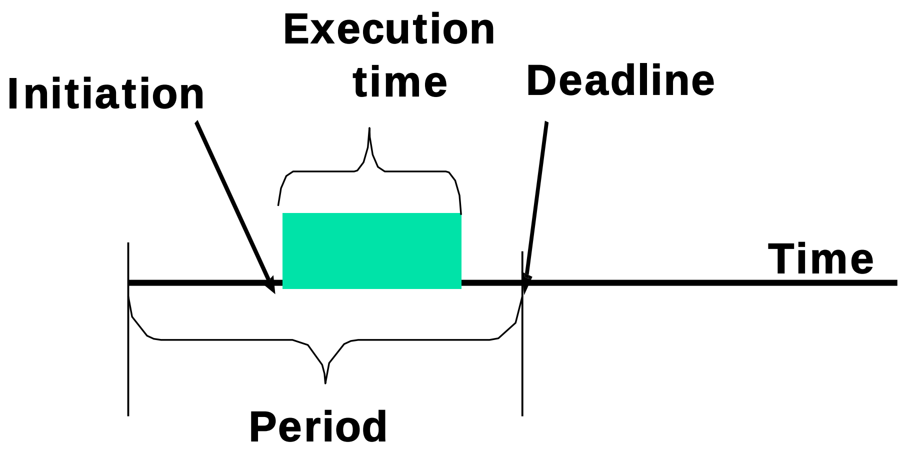


Task: basic notion in RTOS

- 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

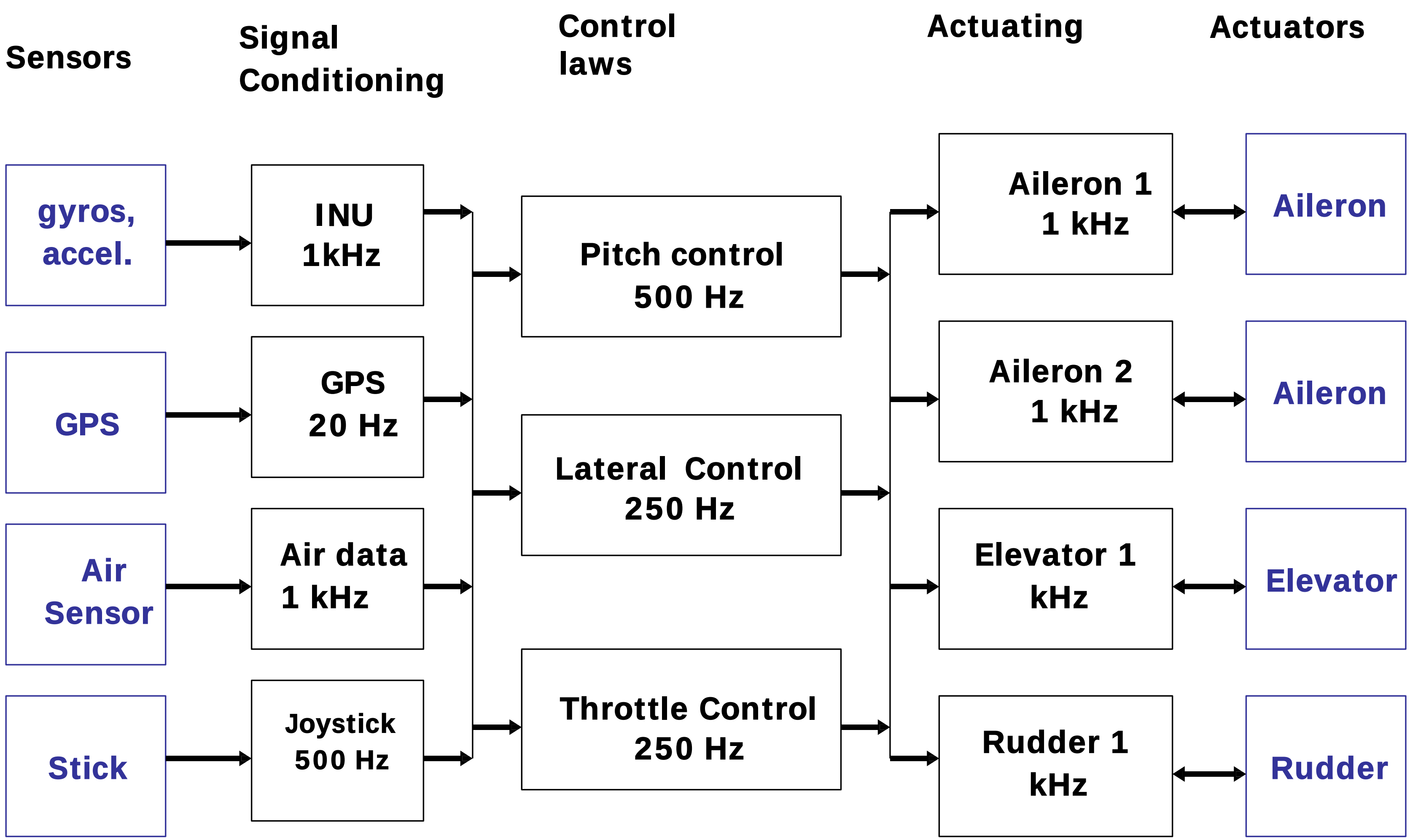


Task Classification (1)

- 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



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

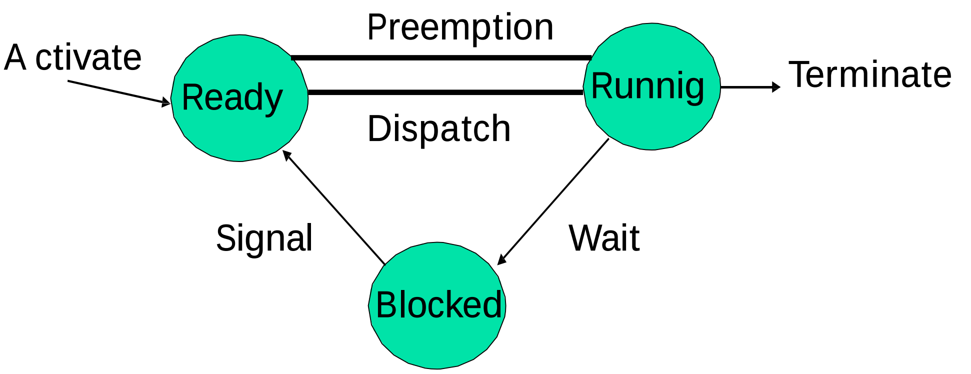
interarrival time T_{min} (often with hard deadline)

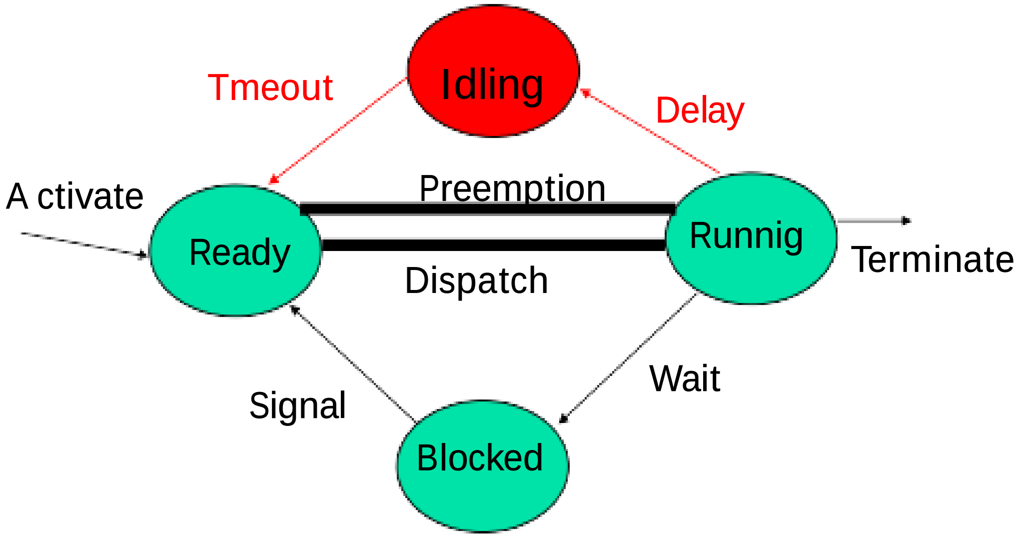
  - worst case = periodic tasks with period T_{min}

Task states

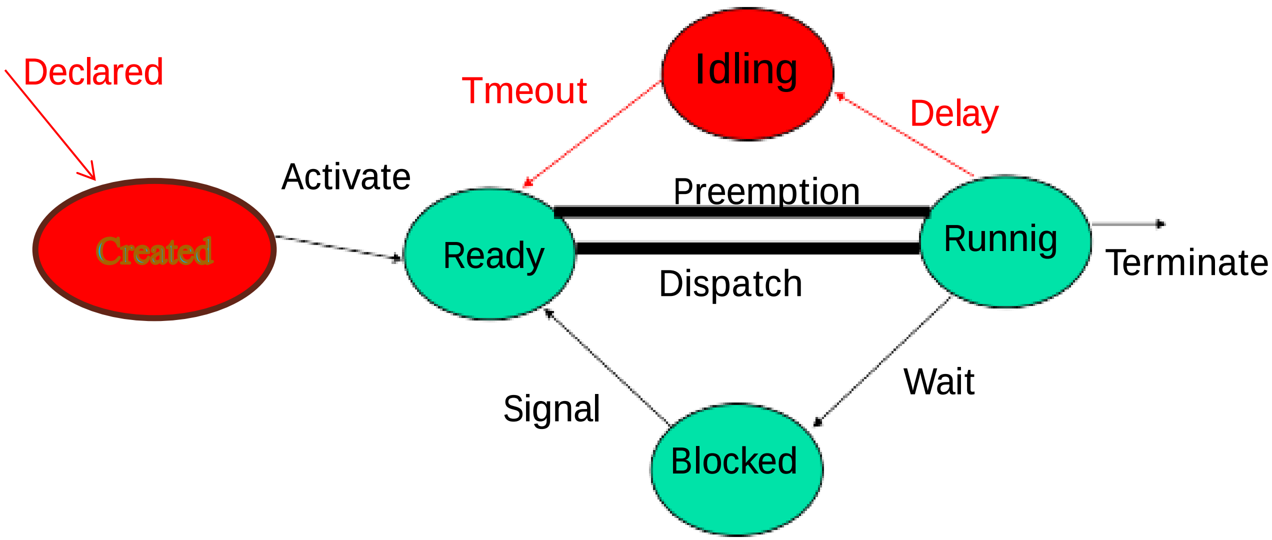
- Ready
- Running
- Waiting/blocked/suspended ...
- Idling
- Terminated

Idling: Rölantide çalışma





Task states


Declared
Tmeout
Idling
Delay
Activate
Preemption
Created
Ready
Runnig
Terminate
Dispatch
Wait
Signal
Blocked

TCB (Task Control Block)

- Id
- Task state (e.g. Idling)
- Task type (hard, soft, background ...)
- Priority
- Other Task parameters
  - period
  - commuting 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 mangement

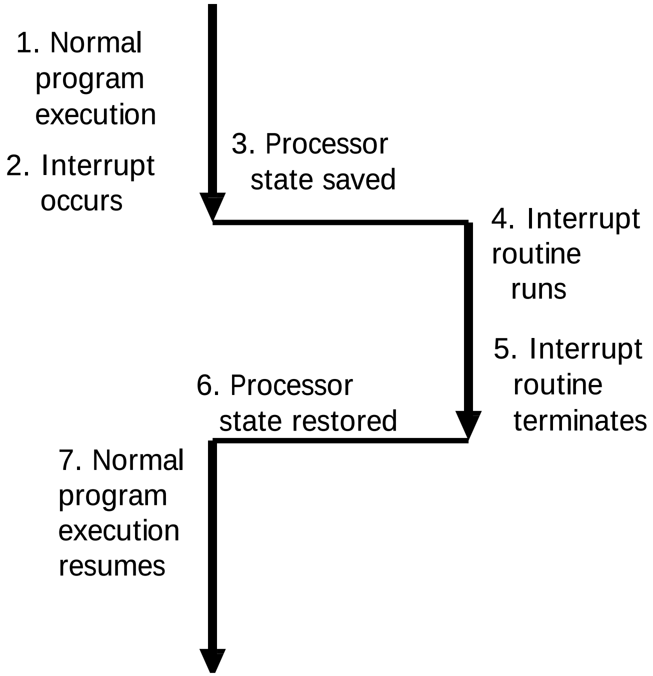
- Task creation: create a new TCB
- Task termination: remove the TCB
- Change Priority: modify the TCB
- State-inquiry: read the TCB
- Challenges for an RTOS
  - The memory blocks for RT tasks must be locked in main memory to avoid access latencies due to swapping
  - Creating an RT task, it has to get the memory without delay: this is difficult because memory has to be allocated and a lot of data structures, code segment must be copied/initialized
  - Changing run-time priorities is dangerous: it may change the run-time behavior and predictability of the whole system

Basic functions of RT OS

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

Interrupt and Interrupt Handling

- Interrupt: environmental event that demands attention.
  - Example: “byte arrived” interrupt on serial channel.
- Interrupt routine: piece of code executed in response to an interrupt.
- 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.



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
- 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

Basic functions of RT OS

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

Memory Management

- 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 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 (tuzak) 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
- 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 management
- Interrupt handling
- Memory management
- Exception handling
- **Task synchronization**
- Task scheduling
- Time management

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* 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 high priority task is ready to proceed.*

Data exchanging

- Semaphore
- Shared variables
- Bounded buffers
- FIFO
- Mailbox
- Message passing
- Signal
- Semaphore is the most primitive and widely used construct for Synchronization and communication in all operating systems

Semaphore

- 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

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(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

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 & Disadvantages semaphores

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

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 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 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.
- 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.

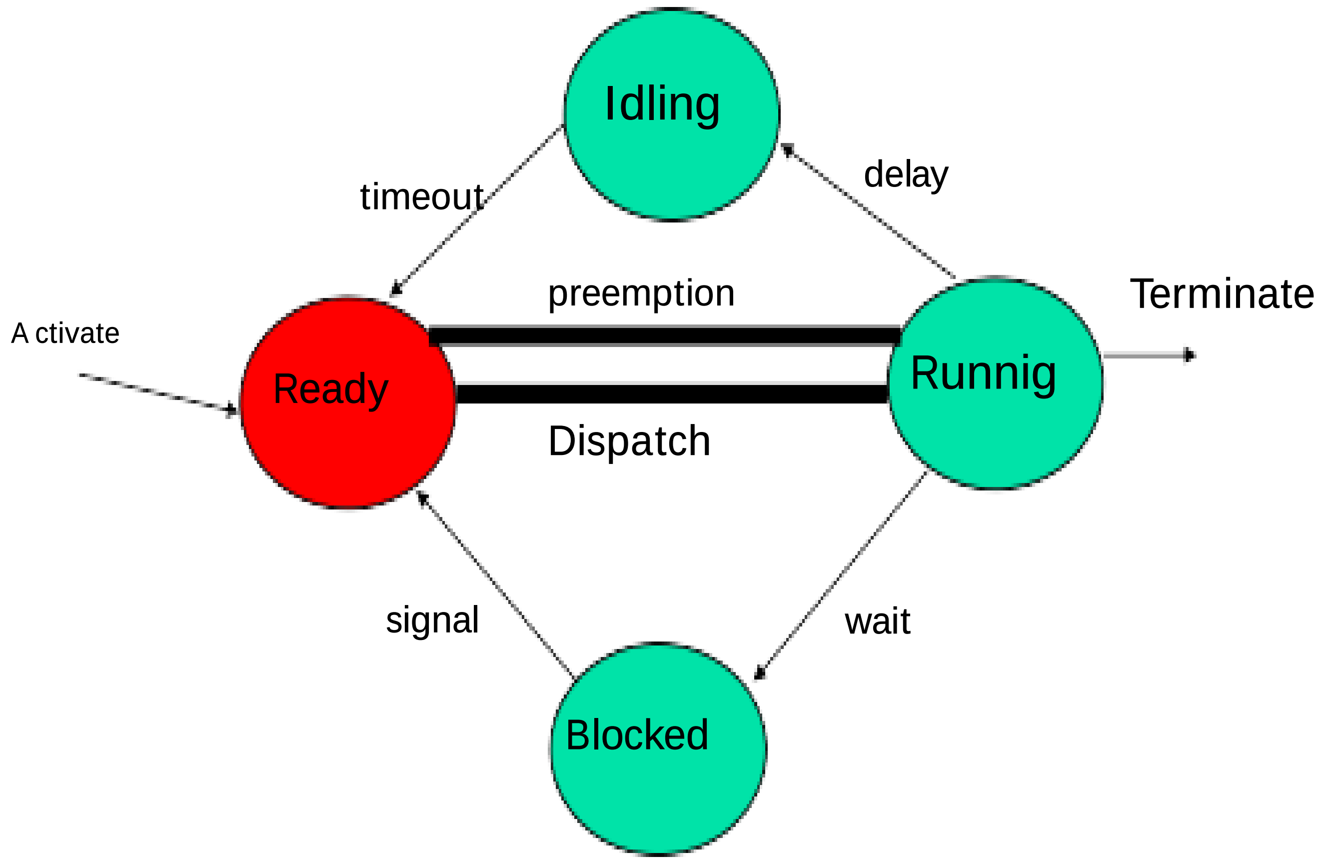
Resource Access Protocols

- Highest Priority Inheritance
  - Non preemption protocol (NPP)
- Basic Priority Inheritance Protocol (BIP)
  - POSIX (RT OS standard) mutexes
- Priority Ceiling Protocols (PCP)
- Immedate 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


Idling
timeout
delay
preemption
Terminate
A ctivate
Ready
Runnig
Dispatch
signal
wait
Blocked

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

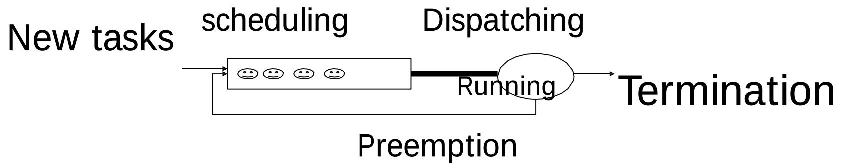
- Sort the READY queue acording to
  - Priorities (HPF)
  - Execution times (SCF)
  - Deadlines (EDF)
  - Arrival times (FIFO)
- Classes of scheduling algorithms
  - Preemptive vs non preemptive
  - Off-line vs on-line
  - Static vs dynamic
  - Event-driven vs time-driven

Task Scheduling

- Scheduler is responsible for time-sharing of CPU among tasks.
- A variety of scheduling algorithms with predictable behaviors exist.
- The general trade-off: the simplicity and the optimality.
- Challenges for an RTOS
- Different performance criteria
- GPOS: maximum average throughput
- RTOS: deterministic behavior
- 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 management
- 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

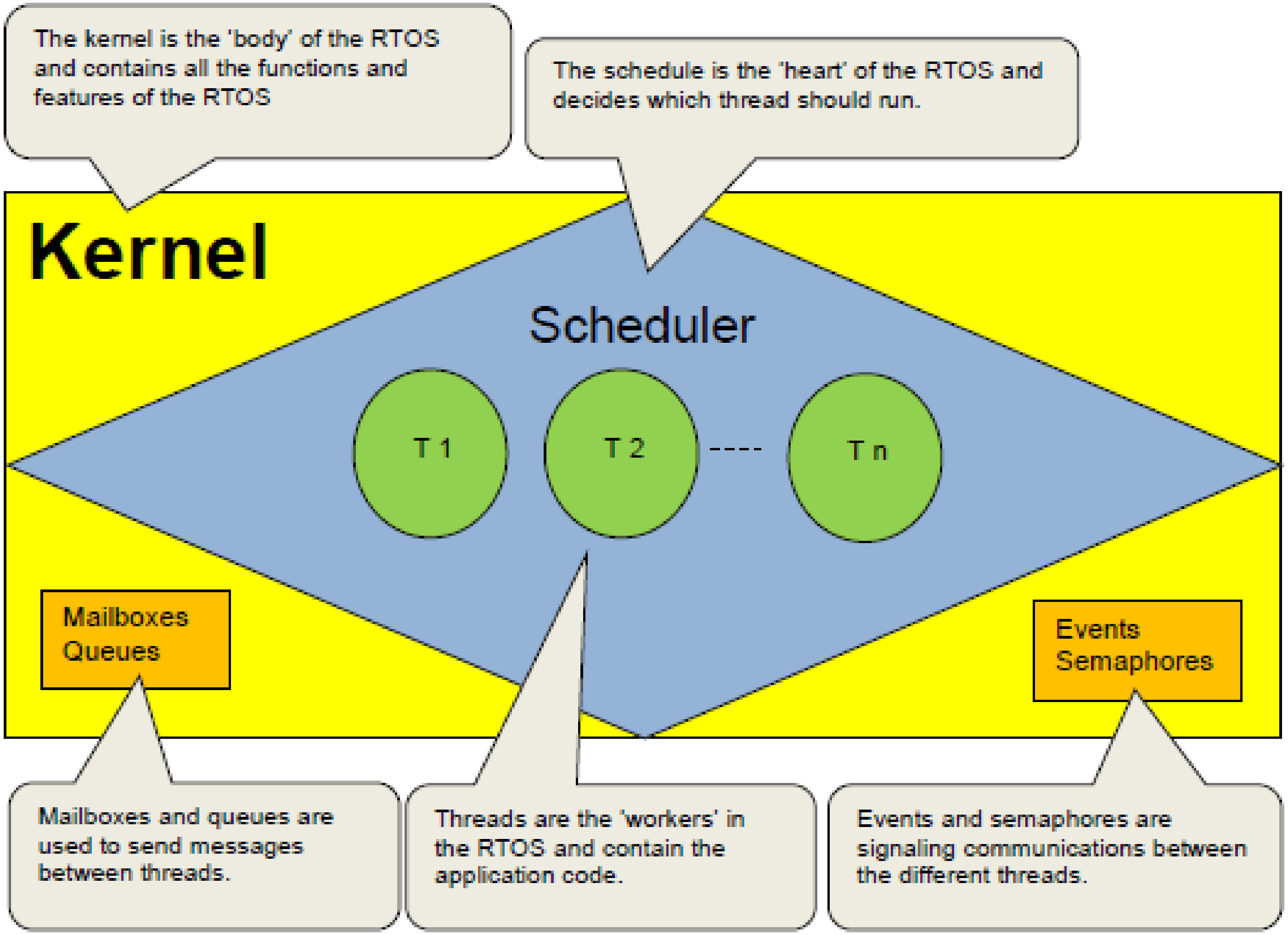
EXAMPLE SYSTEM

- Engine temperature
- Oil pressure
- Rotation per minute (RPM)
- User input

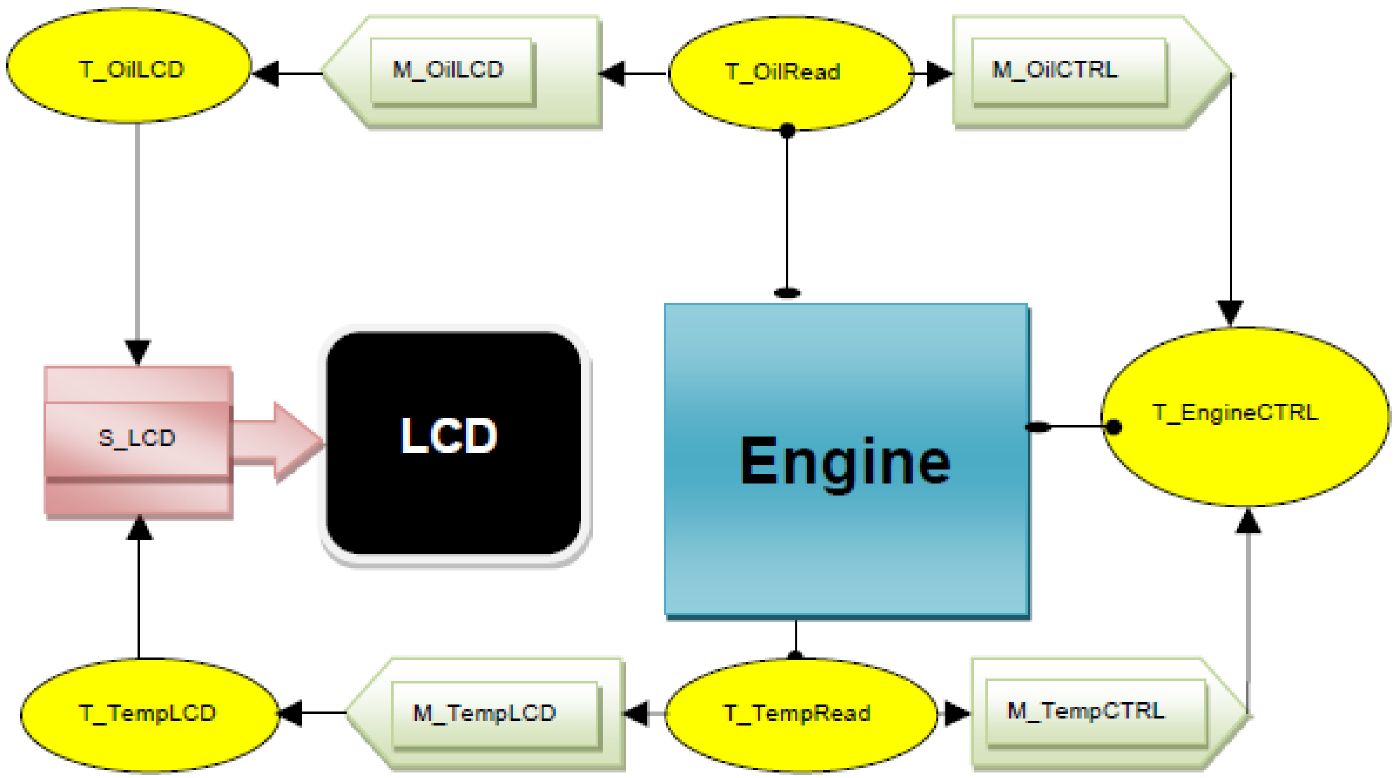
These modules can then be set up as threads, or can be divided into sub-threads. For example:

- Engine temperature
  - Read engine temperature
  - Update LCD with current temperature
- Oil pressure
  - Read current oil pressure
  - Conduct emergency engine shutdown
- RPM
  - Read RPM
  - Update LCD with current RPM
- User input
  - Get gas pedal angle
  - Get current gear
- This division into sub-threads can continue until work can be handled by a single thread.

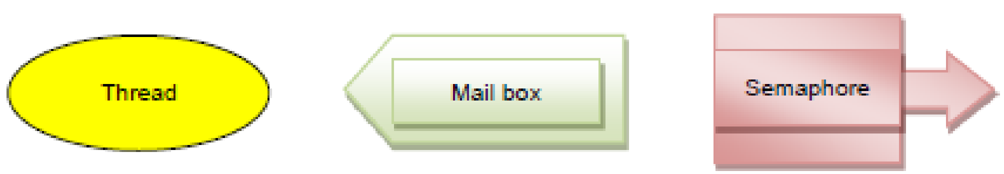
EXAMPLE SYSTEM



EXAMPLE SYSTEM



And the different symbols used are defined as follows…



RTOS

- Real-time Linux
- Microsoft Windows XP ,Microsoft Windows CE Embedded 6.
- Micrium µC/OS-III
- Wind River VxWorks
- Real Time Application Interface (RTAI)
- QNX Neutrino
- FreeRTOS
- TinyOS
- OSE, VRTX32, pSOS

COMPARISON OF RTOS

	Win XP	Win CE	Neutrino	µC/OS-II	Linux	RTAI	VxWorks
A	200µs	20µs	20µs	1,92µs	13,89µs	5µs	3,85µs
B	848µs	99µs	35,2µs	3,2µs	98µs	11,4µs	13,4µs
C	700µs	88,8µs	32µs	2,32µs	77,6µs	7.01µs	10,4µs

A: Response Time, B: Latency, C: Latency Jitter