5. Real-time operating systems

5.1 Functionality

Standard functions should not be re-implemented for each and every new application.
→ Real-time operating systems (RTOS)

RTOS should have the following functionality:

- **Flexible configuration features**

  Large variation of embedded systems + required efficiency → specialised RTOS

- **Direct control of I/O devices**

  Facts:

  - For ES, only few bytes have to be transfered per input/output operation.

  - In most of the cases, I/O devices are immediate devices.

  - There are no 'users' that have to be protected against each other.

  - OS calls come with a major overhead
  → allow for direct I/O by processes.

- **Interrupts can be used by every process**

  Example: a certain interrupt should start a certain process immediately

- **Time services**

- **Scheduling and synchronisation**

  Scheduling =
  mapping: processes → time values.

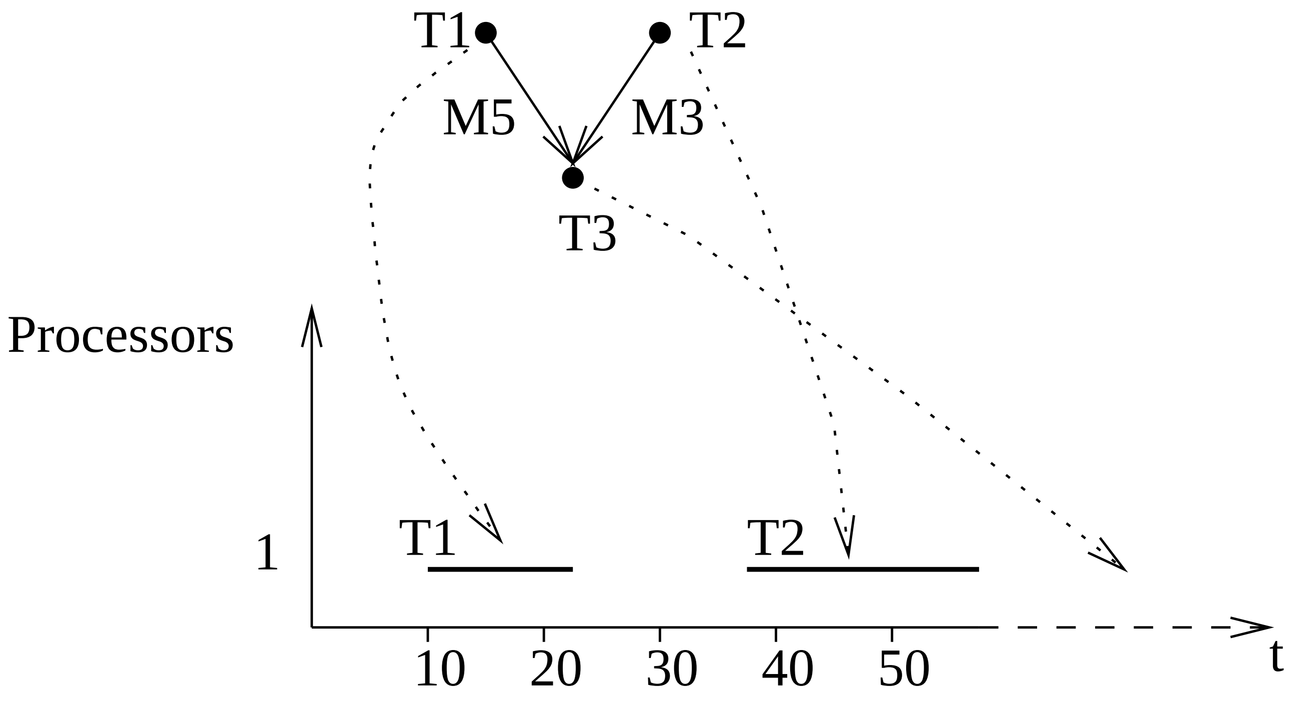
  For real-time systems: most processes and their dependencies are known.

  Timing constraints have to be met.

  In order to guarantee timing constraints, the **worst case execution time** (WCET) has to be computed for every process.

  Using WCETs, **static scheduling** can be used to guarantee meeting the timing constraints.

Example:

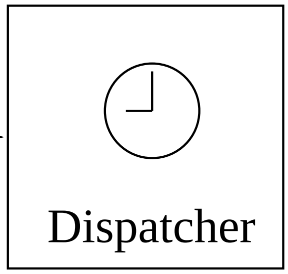


Schedule will be stored in tables; these tables will be used by RTOS for allocating the processor.

Term according to Kopetz: time-triggered system

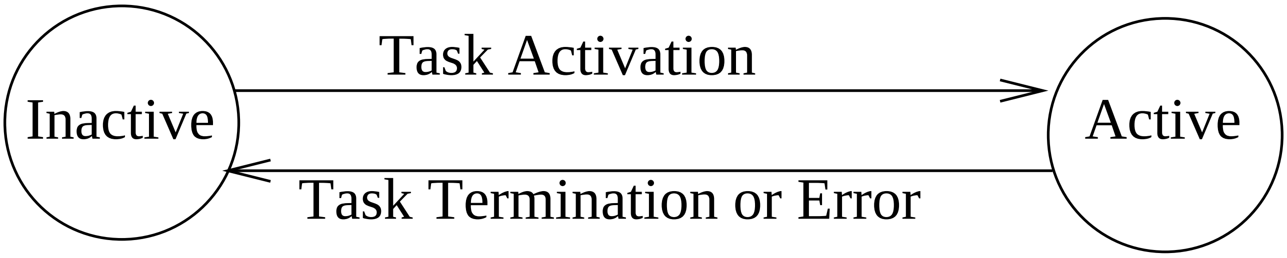
In an entirely time-triggered system, the temporal control structure of all tasks is established a priori *by off-line support-tools. This temporal control structure is encoded in a* Task-Descriptor List (TDL) *that contains the cyclic schedule for all activities of the mode. This schedule considers the required precedence and mutual exclusion relationships among the tasks such that an explicit coordination of the tasks by the operating system at run time is not necessary.*

Time	Action	WCET
10	start T1	12
17	send M5	
22	stop T1	
38	start T3	20
47	send M3	



The dispatcher is activated by the synchronized clock tick. It looks at the TDL, and then performs the action that has been planned for this instant.

A task of a TT system with non-preemptive S-Tasks (task without interprocess communication) is in one of the two states; inactive or active:



More complex in the case of interprocess communication and dynamic scheduling.

Advantages of static scheduling:

- Meeting time constraints can be guaranteed.
- RTOS is very simple.

For satisfying timing constraints in hard real-time systems, predictability of the systems behavior is the most important concern; pre-run-time scheduling is often the only practical means of providing predictability in a complex system. [Xu, Parnas]

Disadvantage: Slow response to urgent events.

RTOS Examples:

- **Fast, proprietary kernels**

  For complex embedded systems, these kernels are inadequate, because they are designed to be fast, rather than to be predictable in every respect [Gupta98].

  Examples: QNX, PDOS, pSOS, VCOS, VRTX32, VxWORKS (*Wind River Systems*).

- **Real time extension to standard OSes**

  Slower and behaviour not quite predictable.

  More comprehensive functionality, including standard APIs.

  Not the correct approach because too many basic and inappropriate underlying assumptions still exist such as optimizing for the average case (rather than worst case), assigning resources on demand, ignoring most if not all semantic information about the application, and independent CPU scheduling and resource allocation possibly causing unbounded blocking

  Examples: RT-Unix, RT-POSIX, RT-MACH, Real time version of CHORUS.

- **Research systems**

  Examples: MARS, Spring, MARUTI, ARTS, HARTOS, DARK, Lehrstuhl III, GMD.

- **OS/9**

  OS/9 RTOS with Unix-Like file system.

- **Real time kernels as add-ons to Windows or DOS**

  Based on the idea of using low-cost PCs.

  Real-time kernels implement several virtual machines.

  Windows can be executed on one of these virtual machines.

  Maintenance requires significant efforts.

- **Windows CE**: Announced to be used in the embedded market.

- **Java**

  Allows several threads. However, scheduling is not deterministic.

6 Software development for real time systems

Software development for real time systems
$= \text{software development} + {'x'}$

$'x'$:

1. **Scheduling** (see discussion of RTOS)
2. **Increased validation effort**

   Normal software can be validated offline.

   Testing embedded software requires testing in the environment.

   May even be dangerous.
3. **Continous operation of software**

   No memory leaks.

   Correct response to exceptions.

   ...
4. **Dependable software required**

Emphasis on item 1.

6.1 Real time scheduling

6.1.1 Terms

Purpose: generation of task/process schedule satisfying all constraints (including resource constraints, deadlines, dependencies).

Classification:

- **Dynamic scheduling:** decisions at run time.
- **Static scheduling**: generation of schedule at design time; Generation of a table containing process start and termination times.
- **Non-preemptive scheduling**
- **Preemptive scheduling**
- **Centralized scheduling**
- **Distributed scheduling**

Terms:

- **Periodic processes**: process T_i has to be executed once during period p_i.
- c_i: computation time of process i.
- d_i: *deadline interval*, time between T_i becoming executable and time at which T_i has to have finished execution.
- $d_i - c_i$: *laxity* of task T_i.
  $d_i = c_i$: task i has to be executed immediately.

In the following: restriction to periodic tasks.

Necessary condition for the existance of a schedule for m processors:

$$\mu = \Sigma \frac{c_i}{p_i} \leq m$$

Scheduler is **optimal** $\iff$
it will find a schedule if one exists.

6.1.2 Dynamic scheduling

6.1.2.1 Independent processes

Standard: *rate monotonic scheduling* (Liu, 1973).

Preemptive dynamic scheduler, fixed priorities.

Assumptions:

1. All tasks $\{T_i\}$ with deadlines are periodic.
2. All tasks are independent.
3. $d_i = p_i$, for all tasks i.
4. c_i is known and constant, for all tasks i.
5. Context switching times can be ignored.
6. The following holds for processor utilization μ for $m = 1$:

$$\mu = \sum_{i=1}^{n} \frac{c_i}{p_i} \leq n(2^{1/n} - 1)$$

$$\lim_{n \to \infty} (2^{1/n} - 1) = ln(2) (=\sim 0.7)$$

Task with shortest execution has highest priority etc.

Priority is descending function of execution time.

Dispatcher will select task with highest priority.

Properties

- All tasks keep their deadlines.

- For mono-processor systems, rate monotonic scheduling is optimal.

Steps of the proof: (c.f. Liu)

1. Considers times (critical times), at which all requests arrive simultaneously.

   It can be shown that the task with the highest priority keeps its deadline.

2. Next, the task with the second largest priority is considered, etc.

3. In a second phase, it is shown that all deadlines are met, if they are met for critical times.

If all tasks have a period which is a multiple of the period of the process with the highest priority, then schedules can be found also for $\mu = 1.0$

In this case, assumption 6 can be replaced by:

$$\mu = \sum_{i=1}^{n} \frac{c_i}{p_i} \leq 1$$

Rate monotonic scheduling has been extended.

Alternatives:

- *Earliest deadline first scheduling* (EDF):
  dynamic priorities, preemptive scheduler,
  for mono-processors: EDF is optimal,
  even for $\mu = 1.0$

- *Least laxity scheduling* (LL):
  task with smallest value of $l_i = d_i - c_i$ is assigned to CPU.
  For mono-processors: LL is optimal.

For multi-processors: neither EDF nor LL are optimal.

6.1.2.2 Dependent tasks

Problem: **Priority inversion**

Given: tasks $T1, T2$ and $T3$,
$T1$ has highest priority, $T3$ lowest priority
rate monotonic scheduling
$T1$ and $T3$ require exclusive use of some resource, via semaphore S.
Assumption: $T3$ in its critical section
Now, $T2$ preempts $T3$
$\rightarrow$: S will not be reset
$\rightarrow$: $T1$ cannot be executed
$\rightarrow$: $T2$ prevents execution of $T1$
$\rightarrow$: **priority inversion**.

First attempt to avoid priority inversion:

Priority inheritance protocol

For critical sections: task priority is maximum of task priority of dependent tasks.

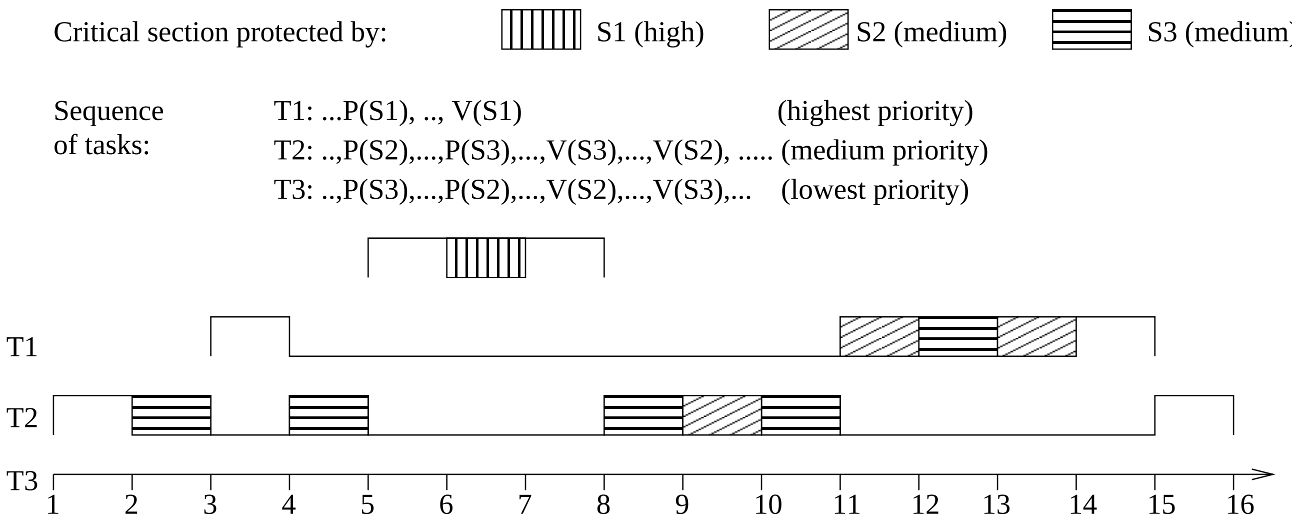
Similar to ADA concept: priority in rendez-vous is the highest priority of involved tasks.

Results in deadlocks and chains of blocked tasks.

Priority ceiling protocol:

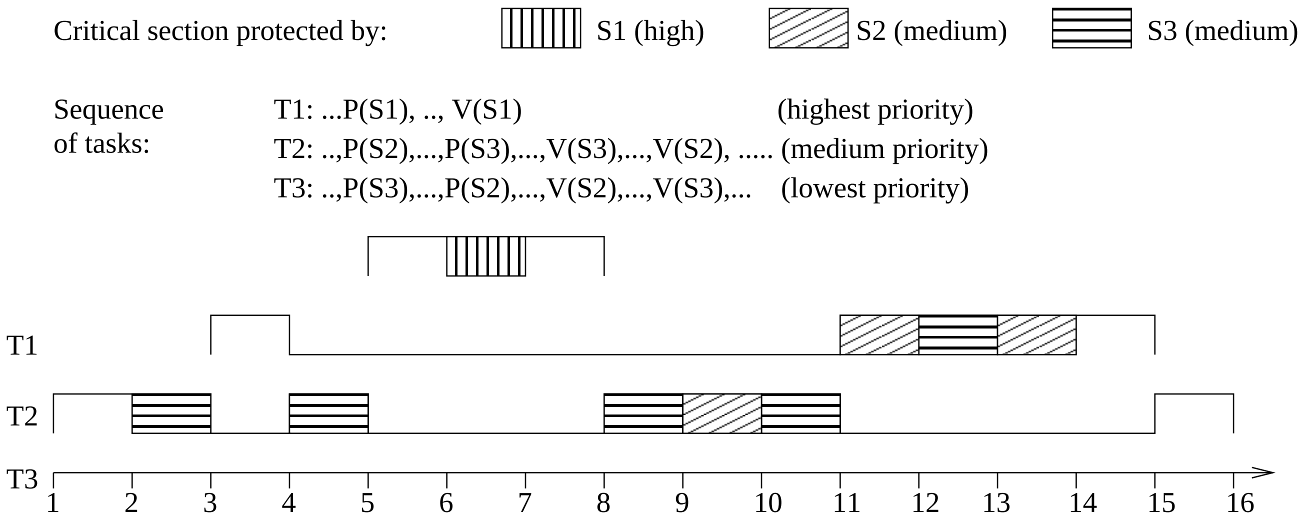
- For each semaphore, there is an upper bound on its priority.
- Semaphores can only be used by tasks with the same or a lower priority
- Critical sections can be entered only, if the task priority is larger than those of other active semaphores.
- Task priority = assigned priority, except within critical sections; Within critical sections: tasks blocking others tasks inherit their largest priority.

Example:



1. T3 starts execution
2. T3 sets S3
3. T2 starts execution and preempts T3
4. T2 attempts setting S2, is preempted, since its priority is not larger than the priority of S3. T3 regains control, executes its critical section with the inherited priority of T2.

- Semaphores can only be used by tasks with the same or a lower priority
- Critical sections can be entered only, if the task priority is larger than those of other active semaphores.
- Within critical sections: tasks blocking others tasks inherit their largest priority.



5. T1 starts execution and preempts T3
6. T1 sets S1. Priority of T1 > priorities of active semaphores.
7. T1 resets S1.
8. T1 terminates, T3 is executed with the priority of T2.
9. T3 sets S2.
10. T3 resets S2.
11. T3 resets S3 and returns to low priority. T2 sets S2.
12. T2 sets S3.
13. T2 resets S3.
14. T2 resets S2.

15. T2 terminates, T3 regains control.

16. T3 terminates.

Formal verification of properties of this protocol required.

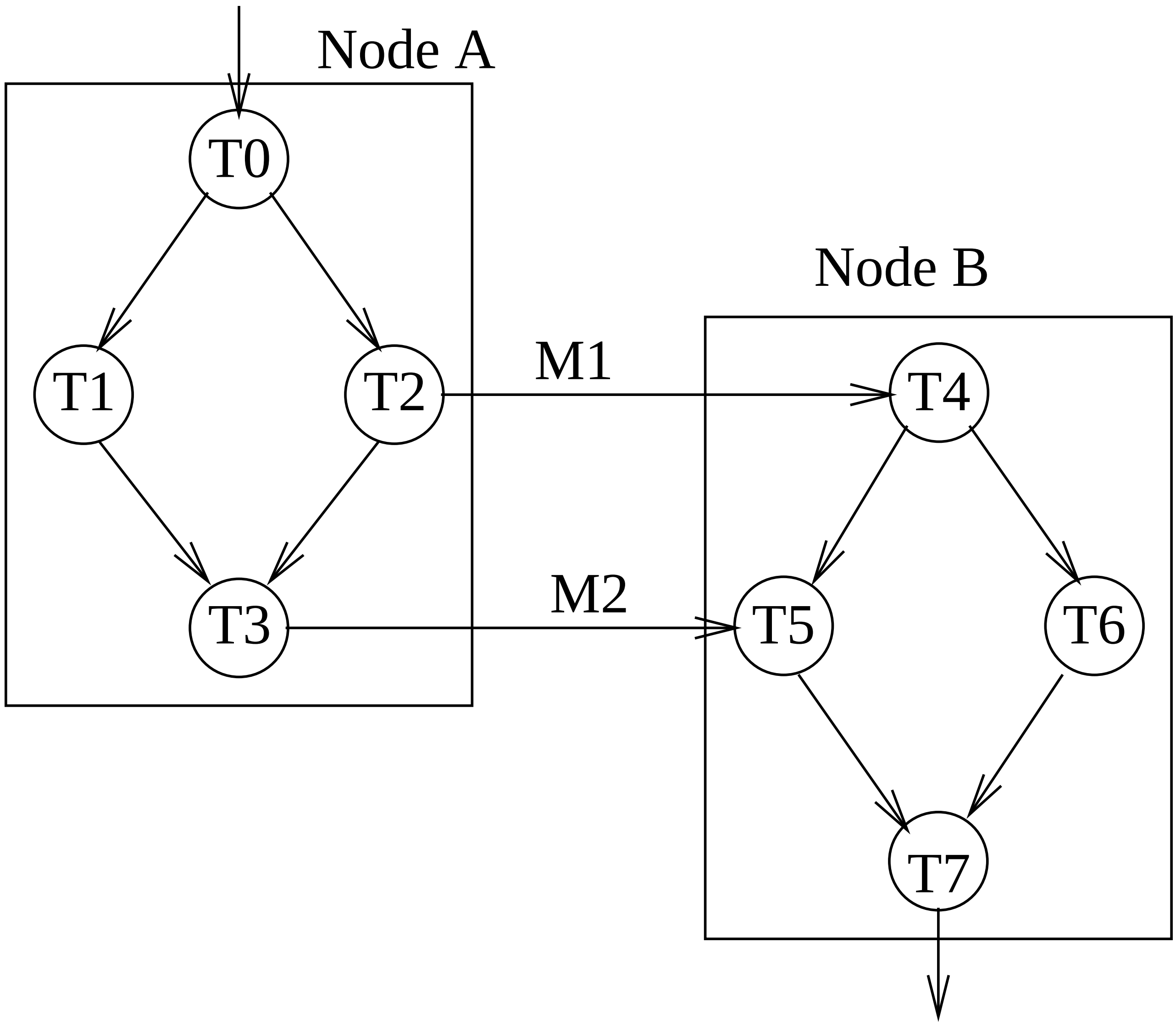
General property of dynamic scheduling: checking of timing constraints difficult, especially for multiprocessor systems.

6.1.3 Static scheduling

Sequence of execution defined during software development.

Consideration of precedence constraints and mutual exclusion.

Precedence constraints represented by precedence graphs.



Sequence represented by table.

Table contains times and actions.

Timer will generate interrupts and these will cause the table to be checked.

No other interrupts.

Times = multiples of smallest time unit.

Repeated for each **schedule period**.

Optimal scheduling for a single processsor is NP complete

Systematic search for schedules.

Times				
1			T0	
2		T1		T2
3		T2		T1&M1
4		M1&T3		T3&T4
5		M2&T4		M2&T6
6	T5		T6	T5
7	T6		T5	T7
8	T7		T7	

Assumption: identical execution and communication times.

Only potentially minimal schedules considered.