Chapter 7
Concurrency

 © Prof. Dr. J. Friedrich, University of Applied Sciences Esslingen

Objectives

What are you about to learn?

Knowledge Objectives

- Understand the concept of concurrency in real-time systems.
- Know about the different ways to implement parallelism in real-time computer systems.
- Understand the concept of a kernel and tasks in real-time operating systems.
- Understand the principles of task management in real-time operating systems.

Skill Objectives

- Ability to model concurrency using UML activity diagrams.
- Ability to program a simple multi-tasking system based on the RMOS or DOSEK operating system.

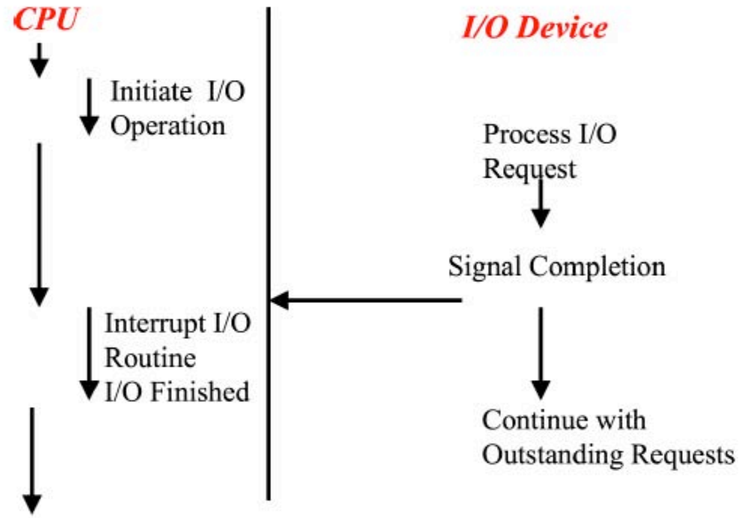
7.1 Concurrency with Real-Time Operating Systems

In many cases, the controlled object of a real-time systems exhibits parallelism. Devices in the real word operate in parallel.

Example: simple system to control and timestamp the temperature of a chemical process.

- Process T reads the temperature via an analogue-to-digital converter and controls a heater which influences the temperature of the chemicals in the vessel
- Process U receives a DCF77 signal and with it synchronizes an internal clock
- Process O displays temperature and time to the operator, and permits the operator to enter the temperature set point

In this example, temperature and time change independent from each other. The operator might enter different set values for the temperature any time.



7.1 Concurrency with Real-Time Operating Systems

We could implement this simple system in three ways:

1. The logical concurrency of T, U, and O is ignored; we use a single program to successively work on each technical process.
2. We use a programming language which supports concurrency based on a run-time support system (Ada, Java, Esterel, Argos, Lustre).
3. We use a standard sequential programming language (C, C++) and a real-time operating system to express the concurrency of the three entities T, U, and O in form of tasks.

We have used the first approach in previous chapters and in our lecture on computer architecture (foreground/background system, cyclic executive). This will hold only for simple real-time systems.

The second solution (using a programming language supporting concurrency) is not wide spread in the real-time industry. It is being used in special applications like defense systems, nuclear reactors, railway systems, airplanes, and in the aerospace industry.

We will focus on the last and most widespread alternative: using a real-time operating system (RTOS) in conjunction with a standard sequential programming language (C).

7.1 Concurrency with Real-Time Operating Systems

Concurrent Programs

A concurrent program is a collection of autonomous sequential processes that execute (logically) in parallel.

There are three alternatives for implementation of a collection of processes:

- Multiprogramming
- Multiprocessing
- Distributed processing

Multiprogramming: Processes multiplex their executions on a single processor.

Multiprocessing: Processes multiplex their executions on a multiprocessor system where each processor has access to shared memory.

Distributed processing: Processes multiplex their executions on a distributed multiprocessor system, where processors do not share a common memory area.

Multiprogramming does not provide true concurrency, while the other two implementations do.

In this class, we will focus on multiprogramming.

Logical Control Flows

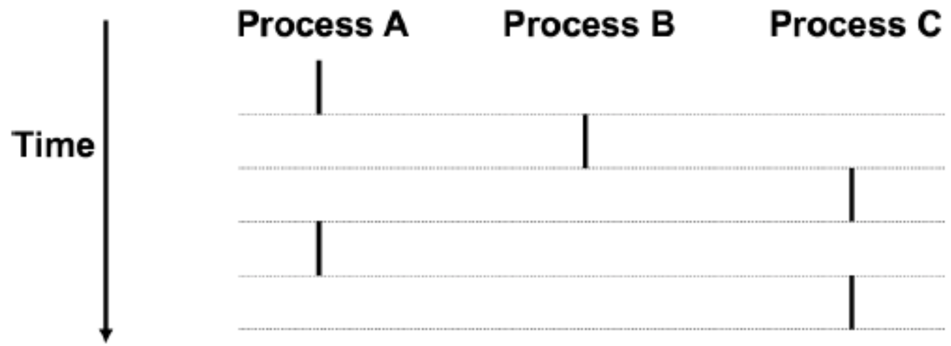
Each process has its own logical control flow.

Two processes run concurrently (are concurrent) if their flows overlap in time.

Otherwise they are sequential.

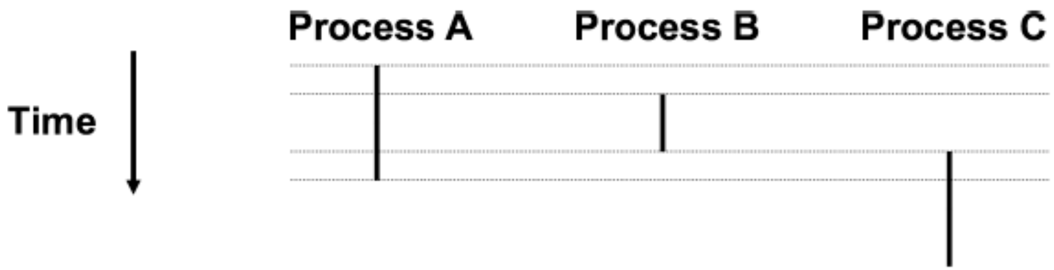
Example:

Better: threads



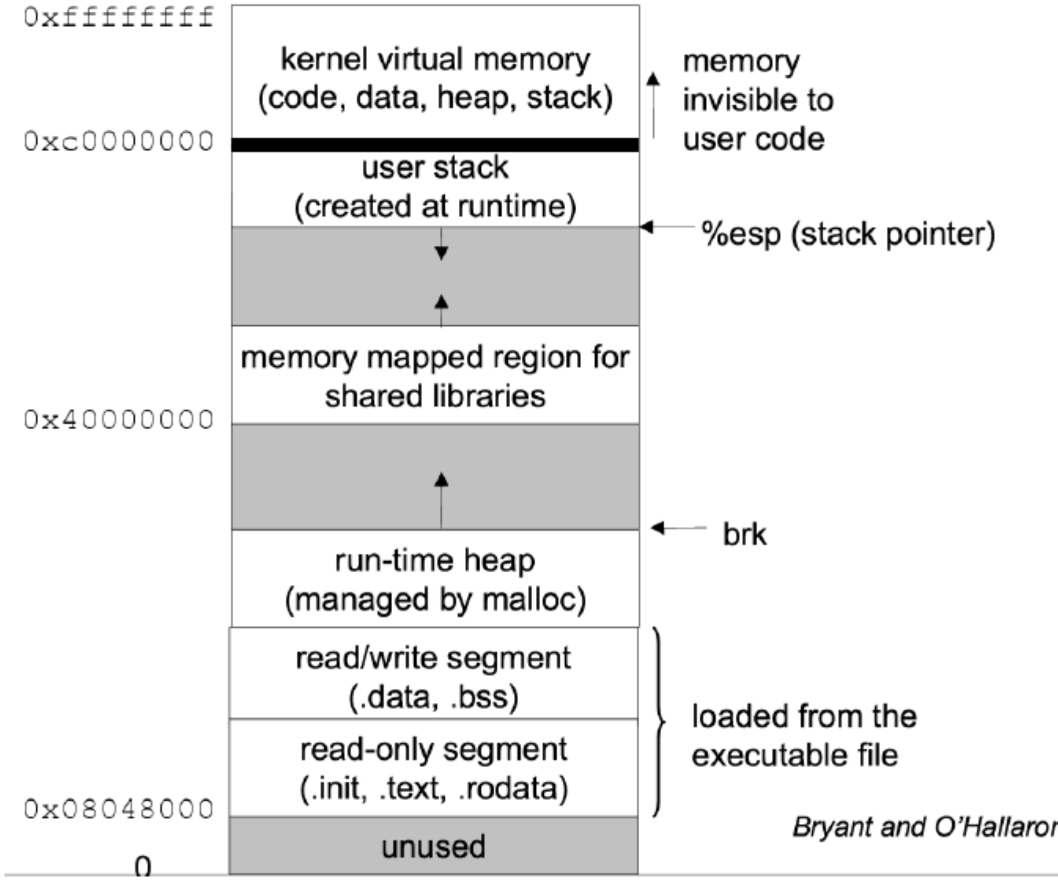
Concurrent: A & B, A & C
Sequential: B & C

User view: we can think of concurrent processes as running in parallel with each other:



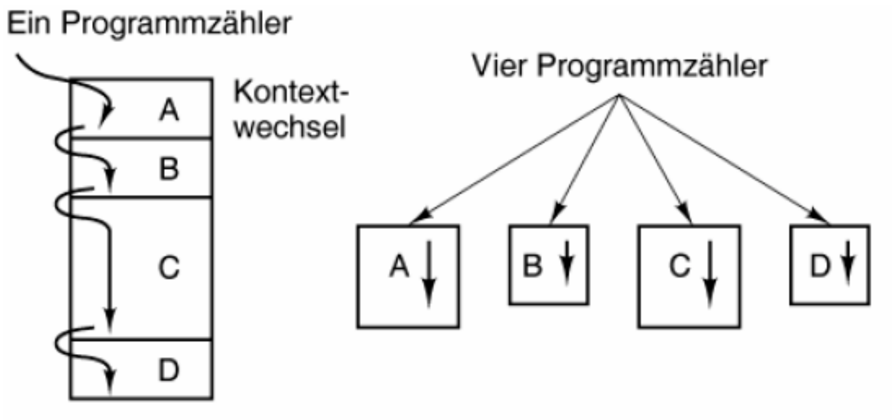
7.2 Processes, Threads, and Multitasking

Computational processes consist of an address space in which one or more threads can be executed, plus all required resources for its execution (IEEE P1003.1 POSIX).



Bryant and O'Hallaron 2003

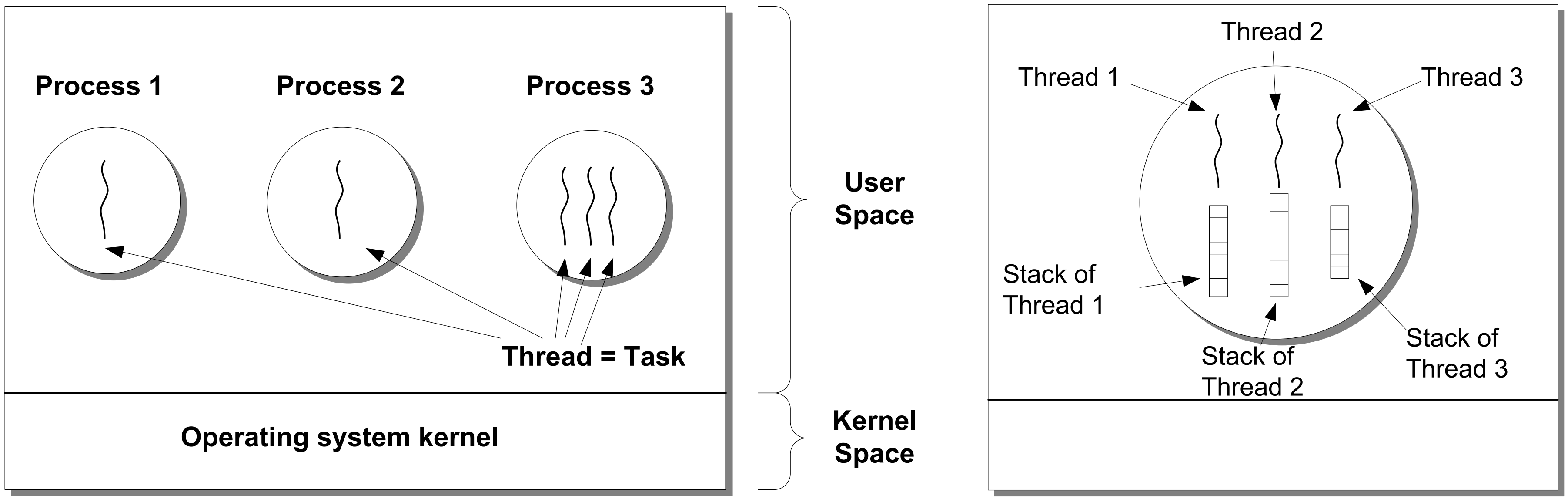
- Each process owns a virtual CPU.
- In reality, the real CPU is time shared between several processes (time multiplexing).
- Many real-time computer systems do not support isolated address spaces for each process (requires MMU).



7.2 Processes, Threads, and Multitasking

Threads represent a single, unique control flow within a computational process. All threads of a computational process share a single address space (IEEE 1003.1 POSIX).

Note: Many RTOS and hardware do not support separation of address spaces and virtual memory which is required for IEEE 1003 processes. Thus, they basically have no processes (or one logical process), and one or more threads. These threads are called “tasks”.



From here on, we call a unique control flow within a multiprogrammed computer system a “task”.

A task is represented within an RTOS by a data structure called a “task control block” (TCB).

Example: DOSEK TCB structure:

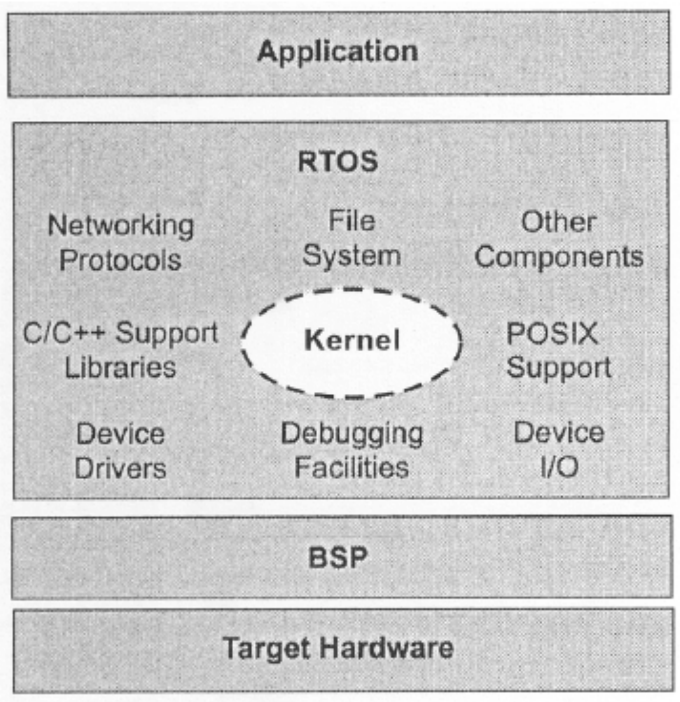
```
typedef struct os_tcb
{
  /* Task stack */
  OS_STK   *TCBStackPtr;    /* current actual stack address */
  OS_STK   *TCBTaskStack;   /* stack start */

  void *TCBTaskAdr;    /* function pointer */

  INT8U TCBStat;       /* task state */
  INT8U TCBPrio;       /* task priority */
  INT8U TCBStdPrio;
  INT8U TCBId;         /* task id */

  /* Events */
  struct os_event TCBEvent;       /* pending event */
  struct os_event TCBWaitEvent;   /* which event are we waiting for? */

  /* linking to other TCB's */
  struct os_tcb *nextTCB;   /* pointer to next task */
  struct os_tcb *prevTCB;   /* pointer to previous task */

  /* if bitmap scheduling is active */
  #if ( SCHEDULER == MLQS )

    INT8U TCB_Y;
    INT8U TCB_BIT_Y;
    INT8U TCB_X;
    INT8U TCB_BIT_X;

  #endif

} OS_TCB;
```

Kernel and Context Switching

Tasks are managed by a part of the operating system called the "kernel".
Common tasks the kernel takes care of are:

- Scheduling tasks
- Managing objects
- Providing services

Microkernel architectures keep kernel functionality at a minimum, and try to provide all other services as tasks. This increases reliability of the kernel.

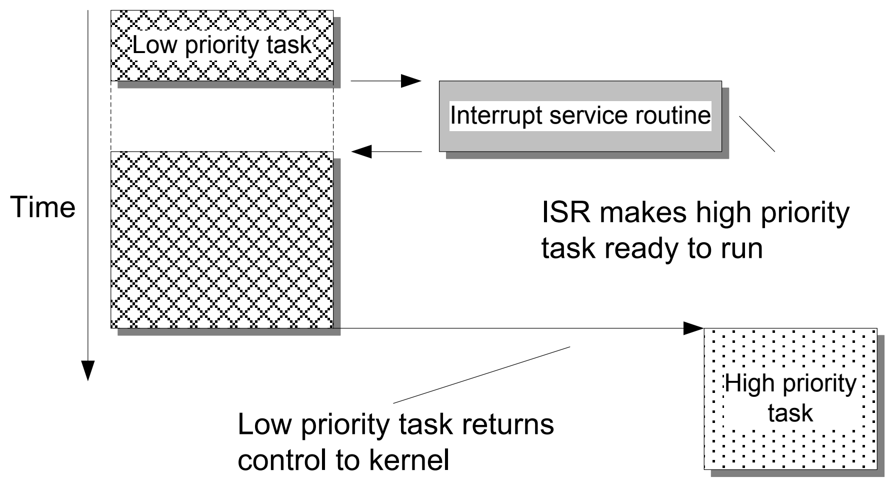


Some RTOS do not provide services like file system support, or networking protocols.

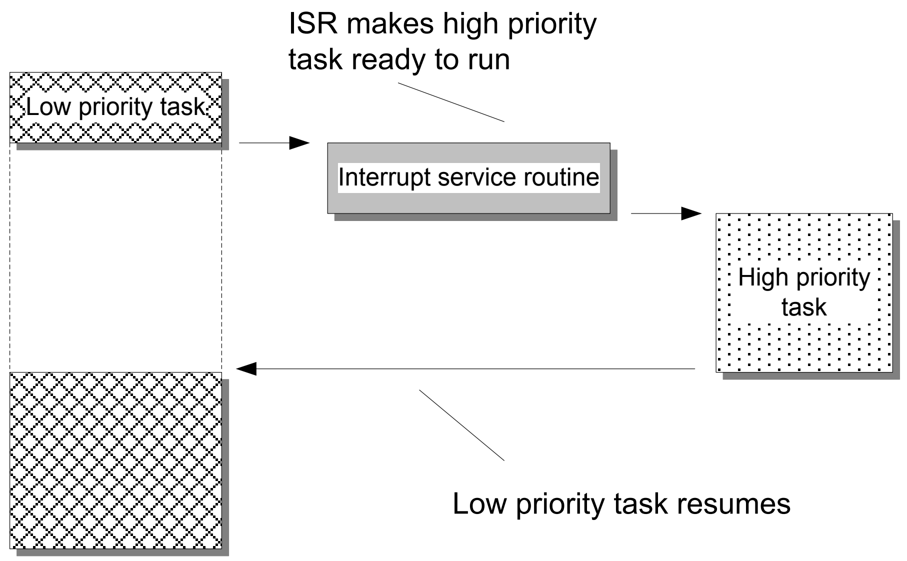
If they just provide basic functionality they are called "real-time kernels".

Kernels supporting preemptive task can interrupt a task any time and can switch to another task.

Kernels that just support non-preemptive tasks rely on the user task to give control back to the kernel.

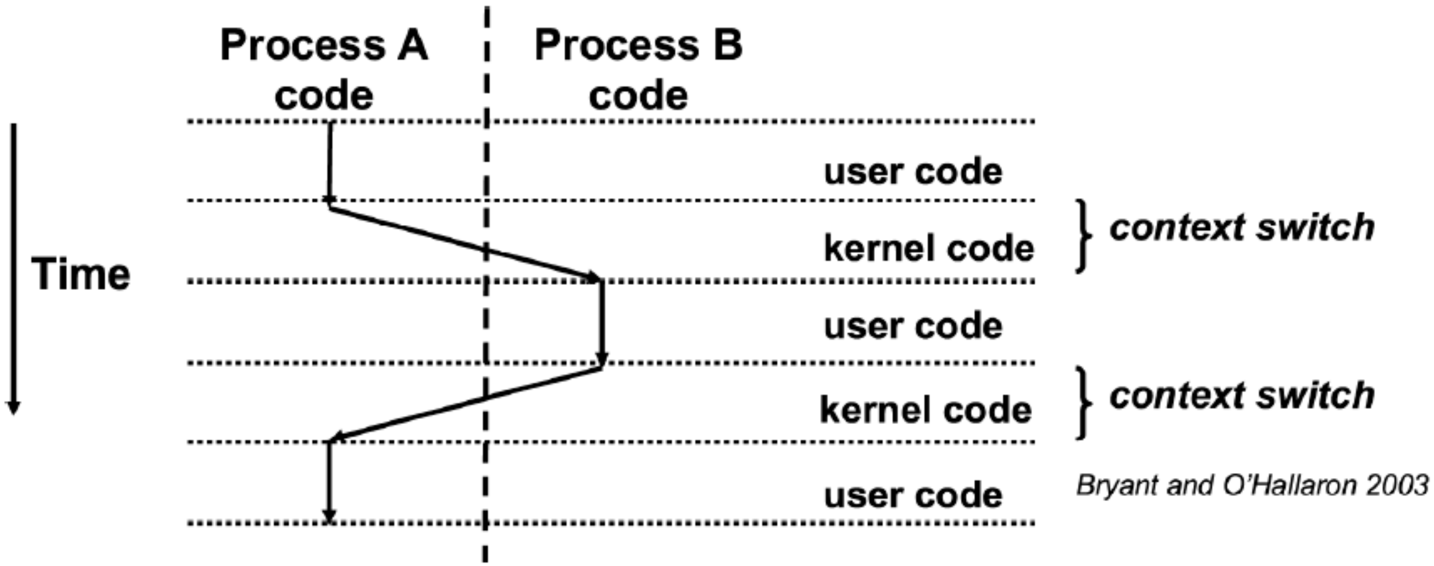


a) Non-preemptive tasks



b) Preemptive tasks

Control flow passes from one task to another task via a context switch:

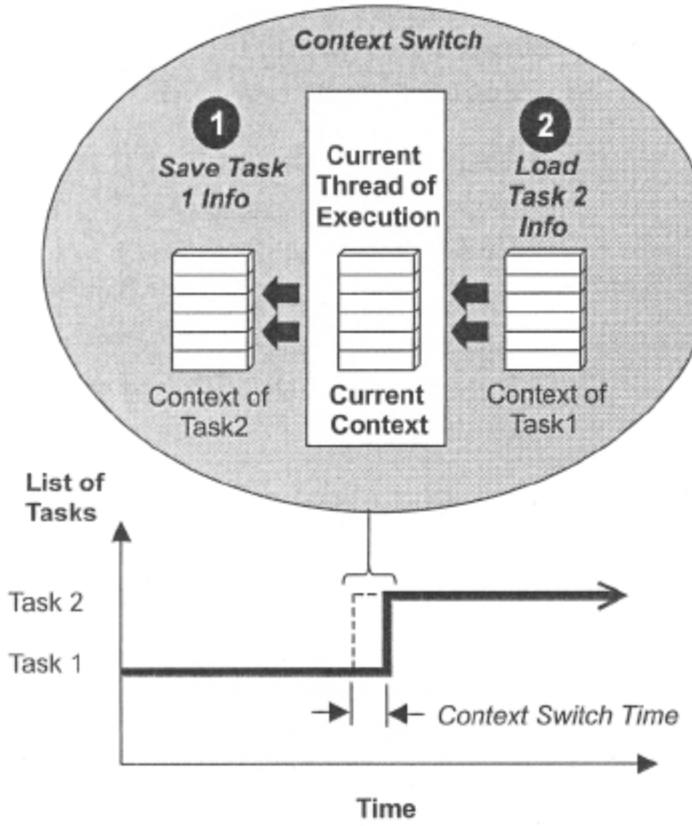


The context consists of:

- All regular registers of the CPU
- The stack pointer
- The instruction pointer

The context switch is one of the few parts of an RTOS that cannot be programmed in a high level programming language.

The context switch is the process of placing the context of an executing task onto its stack and loading the saved context of another task into the CPU registers.



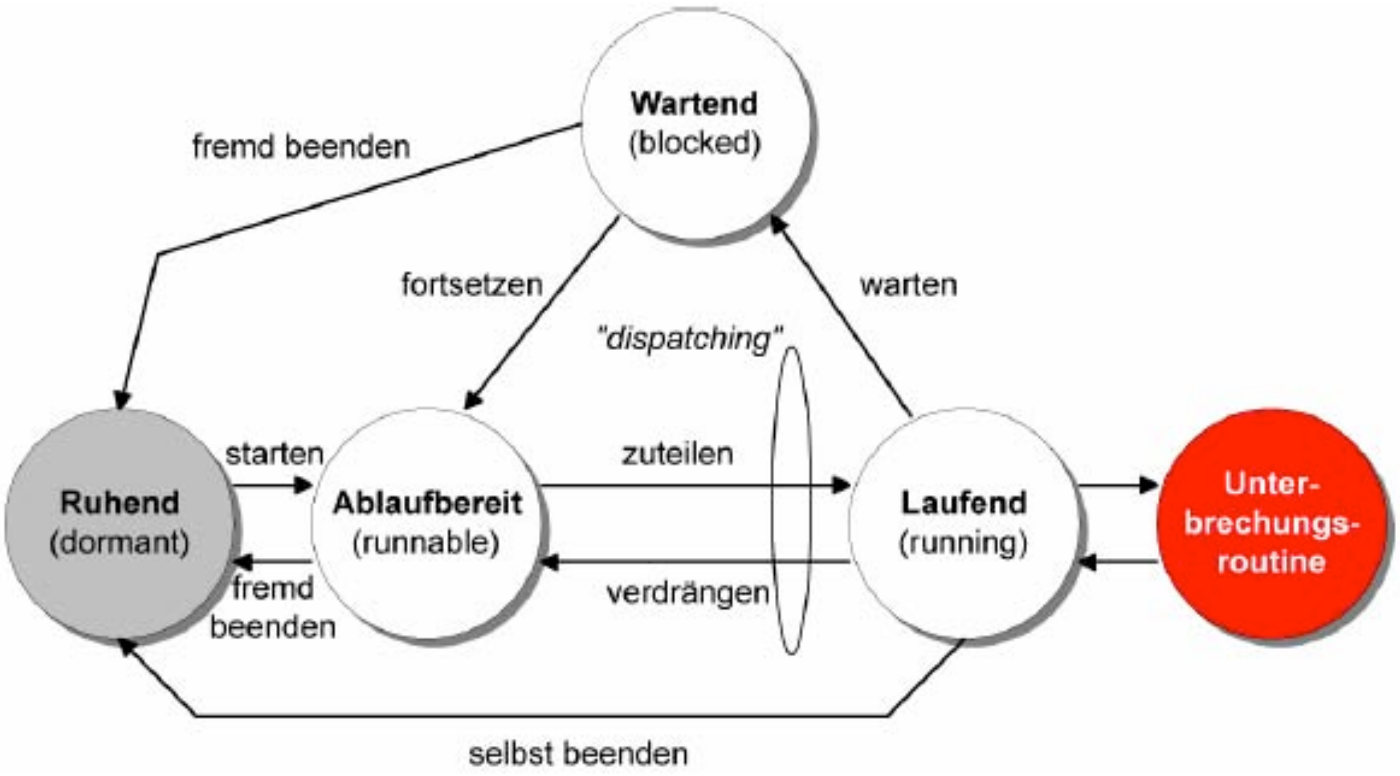

```
/*--------------------------
//Context Switch
/*--------------------------
#pragma TRAP_PROC
void ContextSwitch(void)
{
    __asm
    {
      pshy
      pshx
      pshd
      pshc

      ldx    pTCBPreRun
      sts    0,x

      ldx    pTCBCurRun
      lds    0,x

  }
}
```

Since context switches can occur quite frequently, one measure for the quality of an RTOS is the context switch time.

Task States

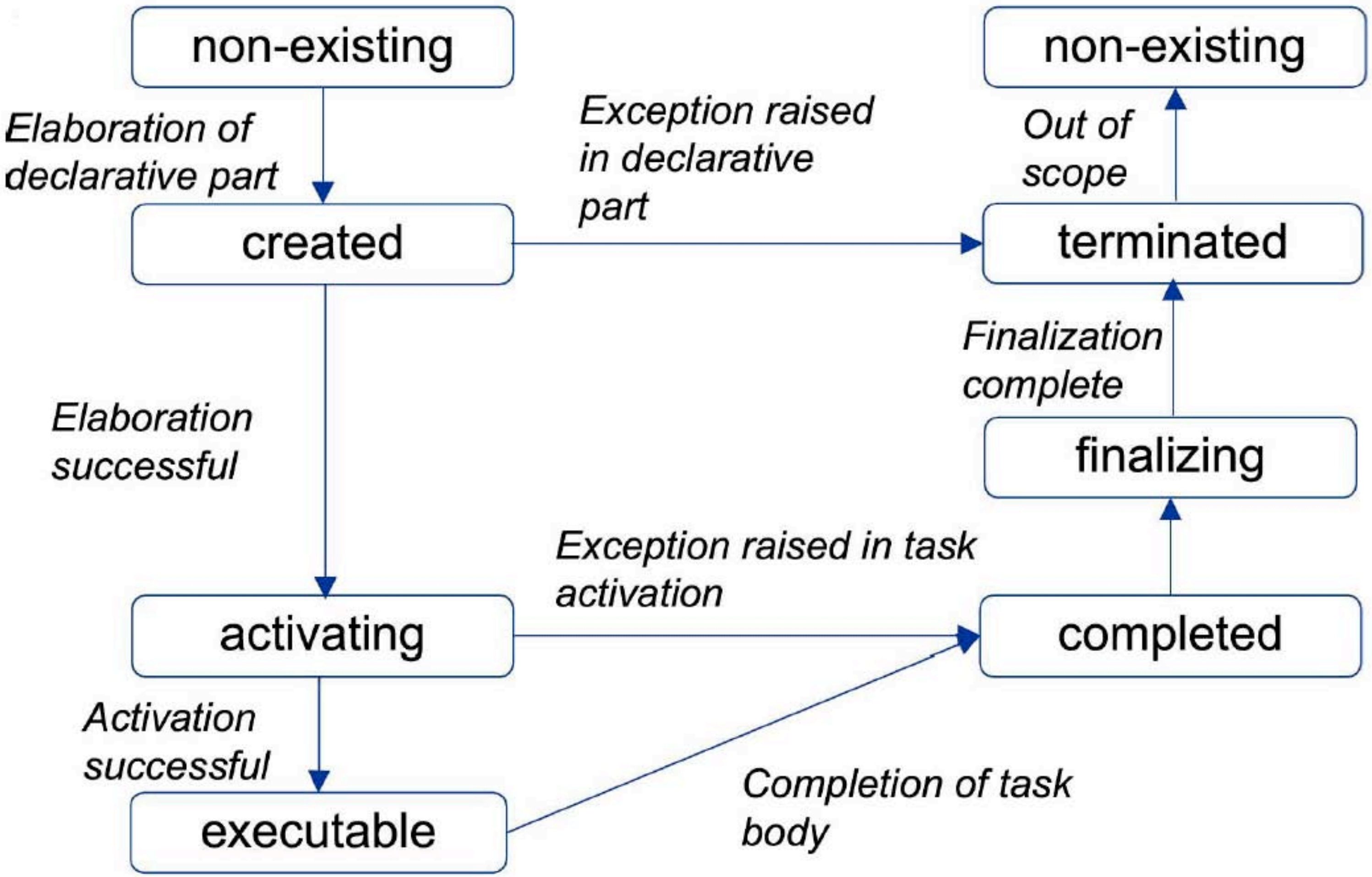
Tasks are always in one out of a set of permitted states.

A typical task state diagram looks like this:

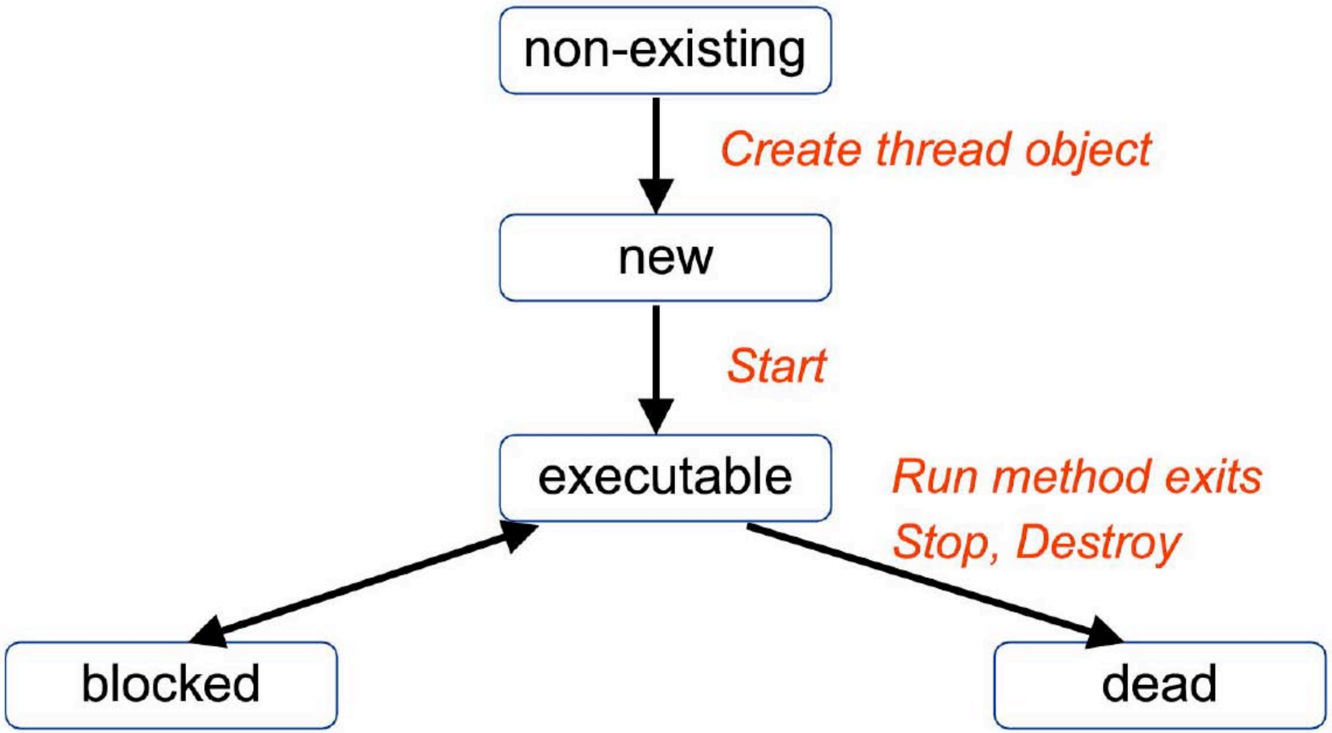


Note: on a single CPU system, at any point in time only one task can be in state “running”!
The task state is managed in a data field in the task control block (TCB).

Example: Ada task states:



Example: Java thread states:



In OSEK, there can be "basic tasks" and "extended tasks". Basic tasks cannot be in state "blocked".

In UNIX, tasks cannot be in state "dormant". After task creation by "fork", tasks are immediately "runnable".

Task state "ready":

- Task could run if it would be assigned computing resources.
- More than one task can be in this state.
- The scheduler decides which task out of the set of ready tasks gets the CPU.

Task state "running":

- Task is currently executing on a CPU.
- Exactly one task per CPU can be in this state.
- Tasks can switch to state "blocked" due to
  - Request of a resource that is not available
  - Waiting on an event (like data from an input device)
  - Pausing (waiting on a timer event)
- Tasks can switch to state "ready" when they are preempted by another task.

Task state "waiting":

- Task waits on an event, e.g., release of a system resource, an interrupt, timer expiration.
- Tasks can bring themselves into this state (pause), or can be placed into this state by other tasks.
- This is one of the most important states, since it enables other tasks to run.
- If high priority tasks wouldn't enter this state every so often, they would starve out other, less important tasks.

Task state "dormant":

- Task does not request or occupy any system resources.
- This state is entered directly after task creation.
- This state does not exist in all operating systems (e.g., UNIX).

Task state "interrupted":

- Actually, this task state is not managed by the kernel and strictly speaking does not belong into this illustration.

Tasks and Objects

We distinguish between

- Active objects: equivalent to tasks. Example in Java: implements Runnable interface.
- Reactive objects: only perform actions when invoked (passive data)
  - Resources: can control order of actions and access to internal states. Example: Java "synchronized" keyword.
  - Passive objects: no control over order of access.

Basic Task Structures

There are two basic task structures:

- Run to completion: runs once and then terminates. Such a task is typically started by another task or an interrupt service routine as a service.

```
void runToCompletion () {
   doSomethingHere();
   RmDeleteTask(RM_OWN_TASK);
}
```

- Endless loop: permanently works on some computational task, e.g. waits for some in-put and generates some output. An endless loop task must have at least one point where it must wait (go into a “blocked” state).

```
void endlessLoopTask () {
   for (;;) {
      RmGetBinSemaphore(); /* blocking call */
      doSomeThingHere();
   }
}
```

Note: In most RTOS tasks are just regular functions and become tasks by associating their function pointer with a task control block!

7.3 Real-Time Operating Systems

There are a number of reasons for using a real-time operating system:

- Reuse of software; otherwise some services are reprogrammed with each application
- Good support for concurrency (multitasking)
- Code proven over a long time with a large user base
- Time to market

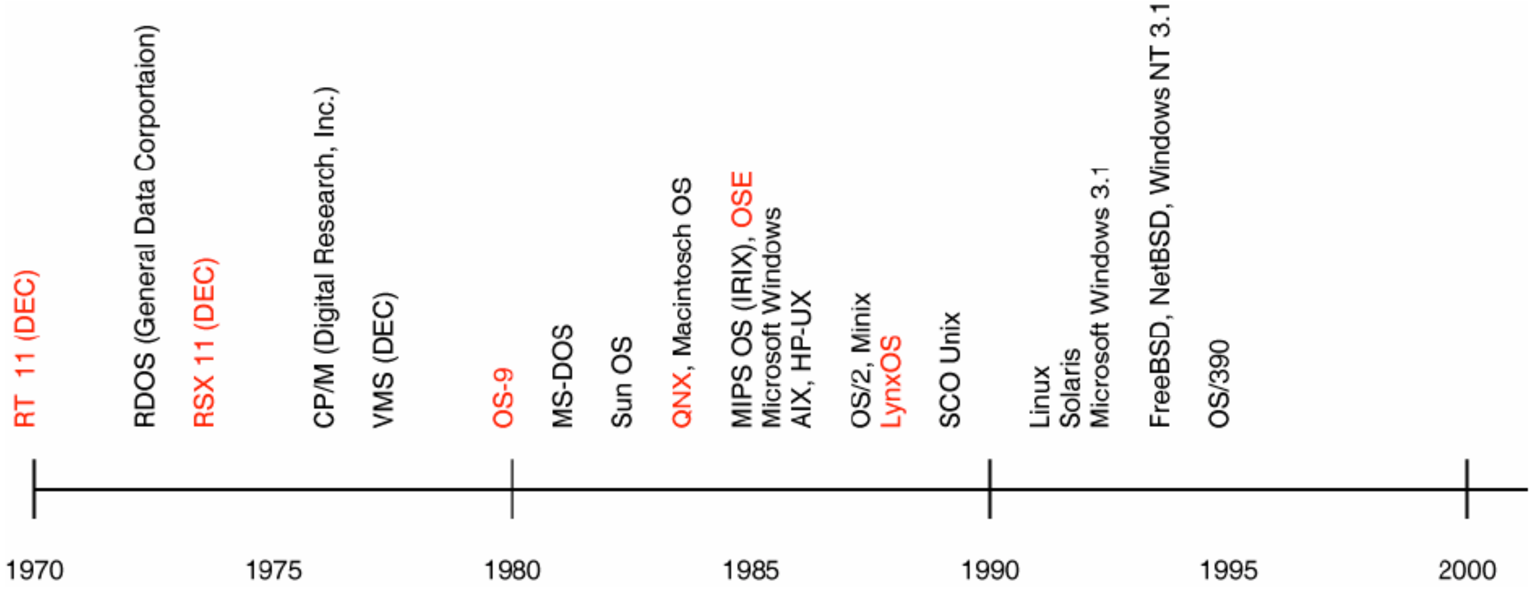
Considerations in Choosing a Real-Time Operating System

This is typically a strategic decision, with large impact.
Check availability for your hardware platforms.
Check tools support (compilers, debuggers, simulation on work station).
Check market base and time on market.
Check for components like network protocols, graphical user interfaces, device drivers.
Check for design-time vs. run-time license fees.
Check for access to source code, helps you to find bugs.
Check for statically vs. dynamically configured RTOS.
Check for scheduling policy support.
Check for time-triggered support.

 © Prof. Dr. J. Friedrich, University of Applied Sciences Esslingen

History of Real-Time Operating Systems

Most real-time operating systems in use today are rather old.



Examples for popular RTOS:

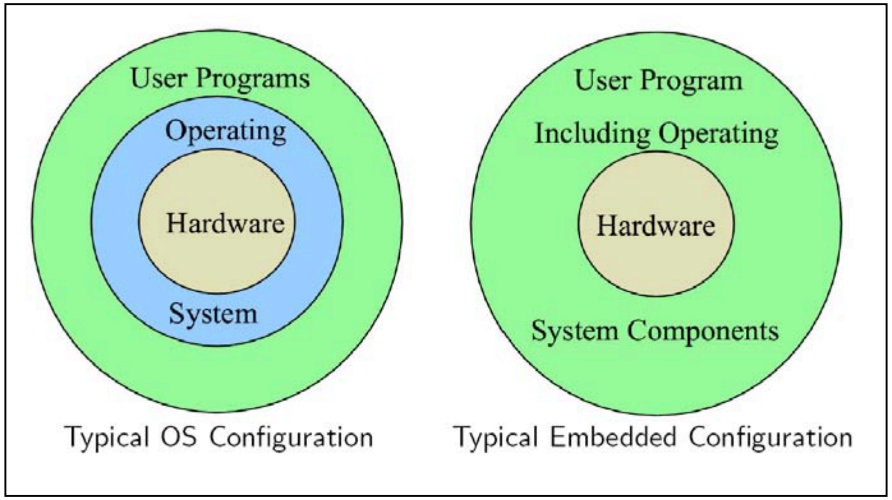
- General: VxWorks, Nucleus, eCOS, QNX Neutrino, Windows CE, microC/OS-II
- Mobile phone: Symbian, Windows Mobile
- Automotive: OSEK, AUTOSAR (specification only)
- Automation: RMOS (in our laboratory)

7.3 Real-Time Operating Systems

Specific Properties of Real-Time Operating Systems

There are some characteristics that distinguish RTOS from General Purpose Operating Systems (GPOS):

- Higher reliability in embedded systems
- Availability, measured in “9s”: 6 nines (99,9999%) means 31 seconds downtime per year
- Scalability, down to very small systems
- Small memory resource demand
- Small kernel down to a few kByte machine code
- High performance, e.g. when switching tasks
- Real-time scheduling algorithms
- Support for diskless systems
- Good portability to different processor platforms
- Predictable performance



Run-Time Support System

Run-time support systems (RTSS) are an alternative to using an OS for achieving concurrency. They are being used in conjunction with a concurrent programming language.

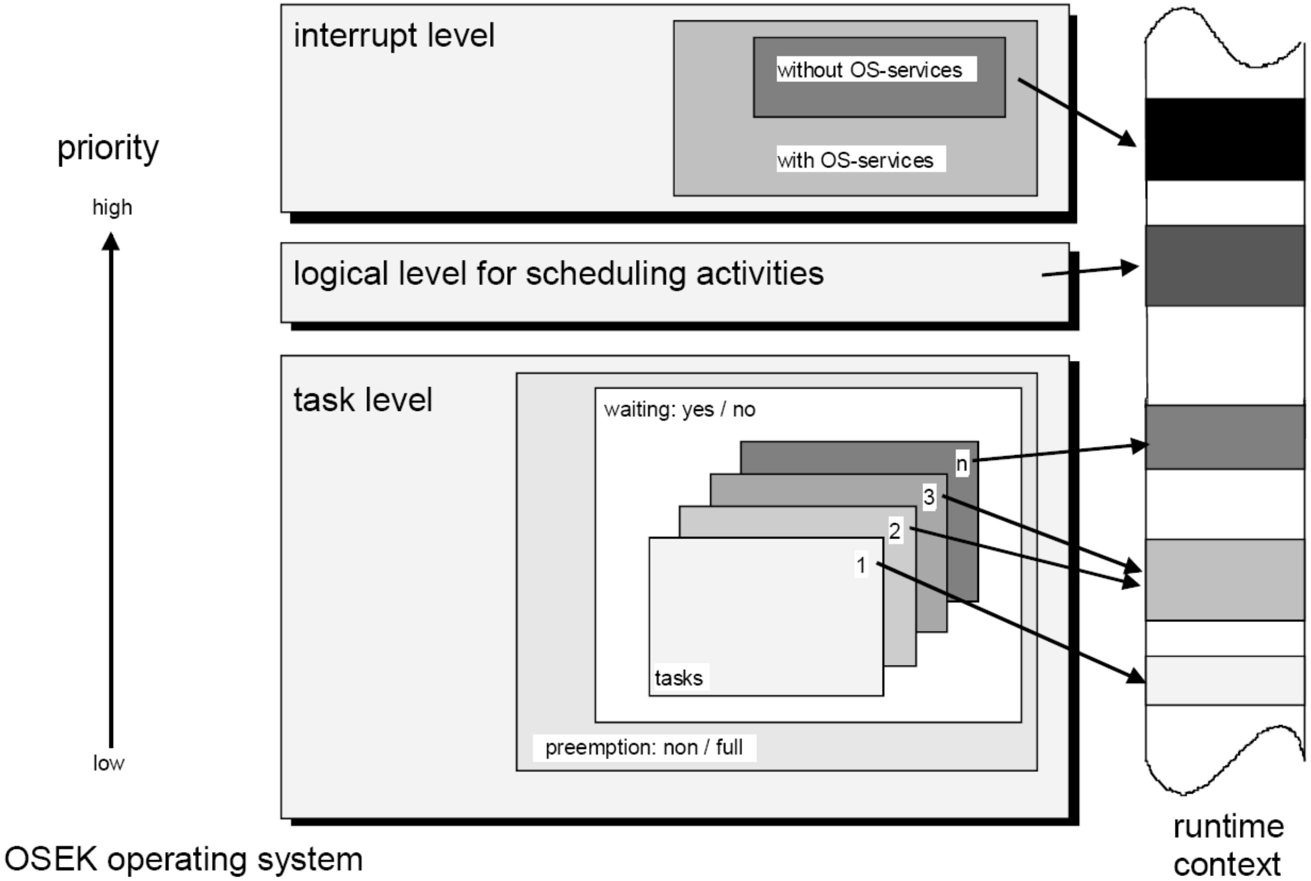
An RTSS acts similar to a scheduler in an operating system.

An RTSS is located logically between hardware and application software.

For well-constructed programs, the logical behavior of the software should not depend on the RTSS.

Processing Levels

Most real-time operating systems structure their services according to the following processing levels (here from example OSEK):



In most cases, tasks are scheduled based on a priority. The highest priority task in "ready" state is scheduled to execute.

7.4 Task Management

Creating Tasks

There are two classes of operating systems when it comes to creating tasks:

- Task creation during run-time
- Task creation during compile-time

Most operating systems permit task creation during run-time. OSEK is a representative for an OS (specification) that forces developers to specify all tasks during compile time in a special configuration file (OIL file). In many embedded real-time systems, there should be no surprises during run-time, which would necessitate creating a task not previously accounted for, so this approach makes sense.

During task creation, the operating system creates the task data structures, in particular the task control block for that task. Example for RMOS:

```c
#include <rmapi.h>
#include <stdio.h>
void main ()
{
  int status;                               /* Statusvariable fuer SVCs */
  char name_taskP1 [] = "TASKP1" ;  /* Name von taskP1 */
  static unsigned int id_taskP1 , id_task_main ;
  xinitt (0, 1, 0, 1, 0, 1, NULL);  /* Initialisiere Konsole */

  printf("Die eigene Taskid ist %d\n", x_cr_gettaskid()); /* Eigene Task-ID */

  /* taskP1 erzeugen: taskP1 = Task-Funktion */
  status = RmCreateTask(name_taskP1, STK_SIZE, PRIO,
                        taskP1, &id_taskP1) ;
```

Making Tasks Ready

Operating systems supporting the “dormant” state require to explicitly start a task so that it will be ready for execution. Making a task “ready” does not mean that it will run immediately!

Example for RMOS:

```
    /* TASKP1 starten */
   status = RmStartTask (RM_WAIT_READY, id_taskP1, RM_TCDPRI, 0, 0); /* taskP1 starten
```

Example for OSEK:

```
TASK(TASK1)
{
        StatusType status;
        status = ActivateTask(TASK2);
        status = GetTaskState(TASK5, &State);
        if(State == SUSPENDED)                              /* the LCD :
        {                                                   /* resource
           status = ChainTask(TASK5);
        }
        status = ActivateTask(TASK3);

        status = TerminateTask();  /* terminates itself */
}
```

Pausing Tasks

A task can pause itself or another task for some time. The time granularity is determined by the operating system basic clock tick (typically 5 or 10 msec).

Example for RMOS:

```
void far taskP1 (void)
{. . . . . . . . . . . .
/*
************************************************
* Warten auf Fortsetzung (dauernd = 255 Stunden)
************************************************
*/
 status = RmPauseTask (RM_HOUR(255));
 exit (0);
}
```

OSEK does not support pausing of tasks.

RMOS distinguishes between pausing the currently running task via RmPauseTask(…), and pausing another task via RmSuspendTask(…).

Resuming Paused Tasks

A task can resume another task, i.e., terminate its pause.

Example for RMOS:

```
void far taskP2 (void)
{. . . . . . . . . . . .
/***************************************
* Fortsetzen nach Warten
****************************************
*/
 status = RmResumeTask (id_taskP1);
 . . . . . . . . . . . .
   exit (0);
}
```

Terminating Tasks

When a task is terminated, it enters a pre-run state (e.g., “dormant” or “suspended”). In some OS, all related resources are deleted when a task terminates. A task can terminate it-self (suicide), or can be terminated by another task.

Example for RMOS (line 28):

```
26      status = RmDeleteTask (id_taskP1);
27
28      exit(0);
```

Example for OSEK:

```
TerminateTask();
```

7.4 Task Management

Deleting Tasks

In dynamically configured operating systems, tasks can be deleted. In statically configured operating systems like OSEK, tasks cannot be deleted.

Example for RMOS (line 26):

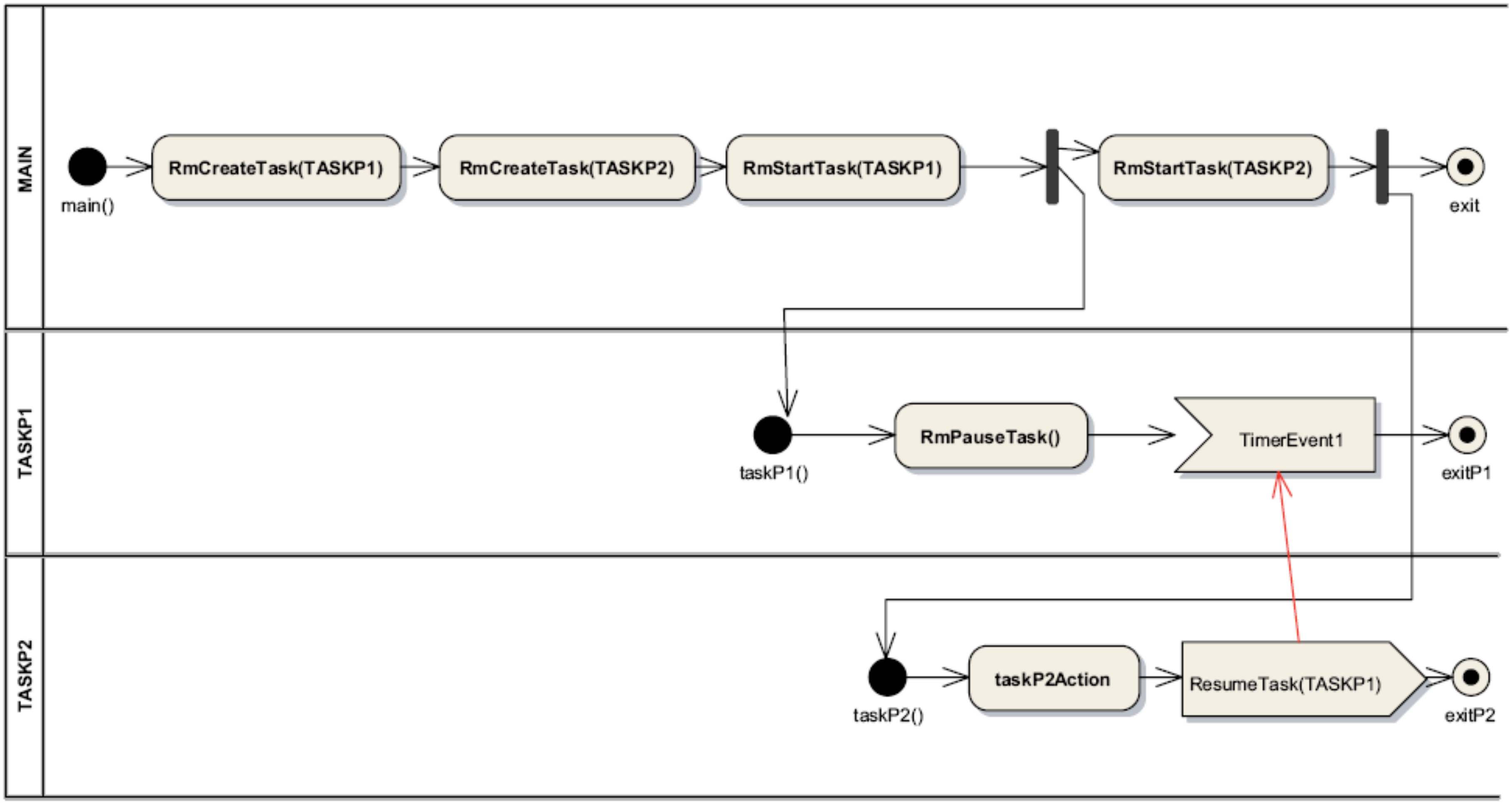
```
26    status = RmDeleteTask (id_taskP1);
27
28    exit(0);
```

In RMOS, a task can terminate and delete itself with:

```
RmDeleteTask (RM_OWN_TASK);
```

Complete Example for RMOS

The activity diagram:



7.4 Task Management

```c
#include <rmapi.h>
#include <stdio.h>
void main ()
{
  int status;                          /* Statusvariable fuer SVCs */
  char name_taskP1 [] = "TASKP1" ;  /* Name von taskP1 */
  static unsigned int id_taskP1 , id_task_main ;
  xinitt (0, 1, 0, 1, 0, 1, NULL);  /* Initialisiere Konsole */

  printf("Die eigene Taskid ist %d\n", x_cr_gettaskid()); /* Eigene Task-ID */

  /* taskP1 erzeugen: taskP1 = Task-Funktion */
  status = RmCreateTask(name_taskP1, STK_SIZE, PRIO,
                        taskP1, &id_taskP1) ;
  if (status != RM_OK)                                  /* "status" pruefen */
    {
      printf("RmCreateTask-ERROR: %s   Status = %x\n", name_taskP1,
             status);
    }

  /* TASKP1 starten */
  status = RmStartTask (RM_WAIT_READY, id_taskP1, RM_TCDPRI, 0, 0); /* taskP1 starten
  status = RmDeleteTask (id_taskP1);        /* TASKP1 löschen mit Uncatalog */

  exit(0);
}
```

Points to Remember