Presented material is based on

- www.rtai.org and the material downloadable from there
There are more than 100 hundred commercial RTOS with memory footprints from few hundred kilobytes to large multiprocessor systems.

Most of them support concurrent programming and static priority preemptive scheduling.

Only a small number supports EDF-driven scheduling and PI-avoidance.

Commonly their distributions come along with a tool chain for developing and analyzing RT-applications.

Commercial RT-OS
Examples of commercial RT-OS

* **VxWorks** implements
  * FP preemptive and Round Robin scheduling
  * Inter-task communication via shared variables, semaphores (with a priority inheritance protocol), message queues, pipes sockets and RPC (remote procedure calls).
  * Conforms to the RT-POSIX 1003.1b standard

* **QNX Neutrion** implements
  * Only signals, timers and scheduler as kernel routines, other components (drivers, file system mg., ...) run outside the kernel.
  * It supports PIP to avoid priority inversion
  * Communication among tasks is performed through message passing
  * Conforms to the RT-POSIX 1003.1b standard
  * Conforms to the RT-POSIX 1003.1b standard

* **OSE** is a family of RT kernels each implementing the OSE API at different levels.
  * It allows different families of processes which run under different scheduling principles: interrupt and priority-based processes are FP scheduled, timer interrupt processes are triggered cyclically background processes are served according RR (round robin)
  * Communication among processes is performed via message queues
* Synchronization?
  System calls can not be blocked (before vers. 2.6), e.g. fork holds back a preemption for tens of milliseconds, malloc.....

* Paging?
  Swapping pages in and out of virtual memory has (in principle) an unbounded execution time

* Fairness?
  For excluding starvation, low priority tasks are eventually executed and might block high prio. Tasks from being executed.

* Request reordering?
  Linux reorders IO-requests from processes to HW.

* Batching?
  May lead to extreme waiting times, e.g. garbage collection of kernel

* Linux, a Real-time System?
* Since version 2.6 the system calls of the Linux Kernel are preemptable, where pre-emption is only possible for dedicated instructions (preemption points).

* To this end, the Linux Kernel can be extended in several ways (customized) for fitting the needs of a desired RT-system.

* Well-known distributions are:
  * RTLinux
  * Real-Time Application Interface (RTAI)
  * Linux Resource/Kernel
  * Many other research kernels, e.g. SHARK, MaRTE and ERIKA

* Linux-based RT-OS
RTAI it is not a real time operating system, such as VXworks or QNX. It is based on the Linux kernel, providing the ability to make it fully pre-emptable. It follows the ideas of RTLinux.

The main features of RTAI are not POSIX compliant, although RTAI implements a compliant subset of POSIX 1003.1.c
* With RTAI the standard Linux Kernel is executed as the lowest priority task, i.e. it is allowed to execute whenever there is no RT task to be scheduled.

* The modification to the Linux Kernel are limited. RTAI executes on a virtual HW, which is implemented by another layer.

* This layer is denoted as RT Hardware Abstraction Layer (RTHAL).
RT-Kernel

Linux Kernel

Pending Linux interrupts

RT interrupts

Non-RT interrupts

RTHAL

Hardware (Architecture)

Kernel Space

*RTAI: Principles
Interrupt Dispatcher that intercepts processes and redirects HW interrupts

* A RT interrupt is immediately served by the RT-kernel via invoking the resp. RT-interrupt handler
* Non-RT interrupts are handled by the Linux kernel, i.e., once now RT task is scheduled
* This yields a fully preemtable Linux kernel and solely modifying 20 lines of source code and adding about 50 lines of new code to the sources of the standard Linux kernel.

* Interception of all interrupts imposes additional overhead, however, this is compensated by improving determinism and responsiveness of RT.

* RTHAL
<table>
<thead>
<tr>
<th>Module</th>
<th>Purpose</th>
<th>Order of loading</th>
</tr>
</thead>
<tbody>
<tr>
<td>rtai_hal</td>
<td>The RTAI Adeos interface layer</td>
<td>1st</td>
</tr>
<tr>
<td>rtai_ksched</td>
<td>The link to the scheduler</td>
<td>2nd</td>
</tr>
<tr>
<td>rtai_up</td>
<td>The uni-processor scheduler</td>
<td>&gt; 2nd</td>
</tr>
<tr>
<td>rtai_mbx</td>
<td>Mailboxes</td>
<td>&gt; 2nd</td>
</tr>
<tr>
<td>rtai_msg</td>
<td>Messaging</td>
<td>&gt; 2nd</td>
</tr>
<tr>
<td>rtai_sem</td>
<td>Semaphores</td>
<td>&gt; 2nd</td>
</tr>
<tr>
<td>rtai_fifos</td>
<td>RT FIFOS</td>
<td>&gt; 2nd</td>
</tr>
<tr>
<td>rtai_shm</td>
<td>Shared Memory</td>
<td>&gt; 2nd</td>
</tr>
<tr>
<td>rtai_lxrt</td>
<td>User space RT tasks</td>
<td>&gt; 2nd</td>
</tr>
</tbody>
</table>
Kernel space real-time tasks

* Semaphores: conventional counting semaphores
* Mailboxes are RT-queues
* Messaging for direct task-to-task communication with unblocking send and blocking receive
* Full-duplex version of a remote procedure call (RPC)
RT FIFOs for RT task to Linux communication

* user space processes treat FIFOs like character devices, i.e. you have a file handler and execute a write and read on it.

* Kernel space tasks access the FIFO via the API of the RTAI

```c
rtf_create_handler(fifo_numver, X_FIFO_HANDLER(x_handler));
```

The handler code is likely to be a kind of:

```c
int x_handler(unsigned int fifo, int rw);
{
    if (rw == 'r') {
        // do stuff for a call from read and return appropriate value.
    } else {
        // do stuff for a call from write and return appropriate value.
    }
}
```
* **Multiple reader/writer:**
  It is important to note that FIFOs allow multiple readers/writers. Hence, the select/poll mechanism to synchronize with in/out data can lead to unexpected blocks for such cases.

* **Example:**
you poll and get that there are data available, then read/write them sure not to be blocked, meanwhile another user gets into and stoles all of your data, when you ask for them you get blocked.

* **RT FIFOs**
To coordinate the access to FIFOs (without a scheduler), RTAI implements binary semaphores, (fd is the file descriptor of the FIFO):

```c
rtf_sem_init(fd, init_val);
rtf_sem_wait(fd);
rtf_sem_trywait(fd);
rtf_sem_timed_wait(fd, ms_delay);
rtf_sem_post(fd);
rtf_sem_destroy(fd);
```

The binary semaphores can be used for many other things, e.g. to synchronize shared memory access or in place of using blocking FIFOs.