Operating systems, OS & RTOS in general terms

**Responsibilities**

For all operating systems
- Task management and scheduling
- Interrupt servicing
- Interprocess communication & synchronization
- Memory management
- Extended responsibilities
  - File management
  - User interaction
  - Communication protocol stacks
  - Handle storage IO

**Real Time operating Systems**

- Task management and scheduling
  - Run-time Priority of the task
  - Latencies due to swapping memory etc
  - Context switching delays
  - Simple & efficient or complex & optimal (maybe fair)
  - Maximum average throughput or deterministic behavior

- Interrupt servicing
  - Hardware & software interrupt
    In principal isn't the OS involved in the interrupt code, but have to deal with the consequences, handle the “deferred interrupt servicing”.

- Interprocess communication & synchronization

- Memory management
  - Memory allocation
  - Memory mapping
  - Memory protection
**Concepts**

- **Kernel space**
- **User space**
- **Real-time space**

- **Monolithic kernel**
  - Runs all OS services in protected mode. (Linux, UNIX, winxx)

- **Micro-kernel**
  - Runs only the “really core” services (scheduling, interrupt handling & interprocess com) in protected mode.

- **Non-preemptible kernel**
  - Kernel code cannot be interrupted by either kernel space tasks or user space tasks.
  - Makes design simple but an undeterministic system (Linux < 2.4)

- **Preemptible kernel**
  - Everything is interruptible. Linux 2.6 has preemptible subsystems but the core is still partly non-preemptible.

- **Memory management (Virtual memory)**
  - Overhead and indeterministic allocation of memory. “Secure”

- **Shared memory**
  - Efficient system. Little overhead. “Insecure”.

- **Operating system for support of “lower-level” functionality.**
  - System calls

- **Language runtime for support of “lower-level” functionality.**
  - Library calls
  - Portable, ready to use and safe
  - Generally heavy, not deterministic and not configurable
**Time**

Realtime:
“Times at which tasks will execute can be *predicted deterministically* ” on the basis of knowledge about the systems hardware & software”

The defenitions “soft” and “hard” realtime is related to the applications and the consequences of meeting the timing constraints.

**Concepts**

**Latency**

Difference between the instants of time when a task should have started and when it really did start.

Depends of:
- Timing properties of the hardware (processor, buses, memory, peripherals etc)
- Scheduling properties of the OS
- Pre-emptiness of the kernel
- System load
- Context switching times

The statistical distribution of latency is called *jitter*

**Indeterminism**

Lots of sources in an ordinary computer system:
- Peripheral IO (harddisk access etc)
- Networking (TCP has delivery control but no timing control whatsoever...)
- Low-resolution timing
- Non real-time device drivers
- memory allocation & management

**Different types of timing constraints**

- Deadlines
  Doesn't matter when from now and until deadline the task is completed
- Zero execution time or “Instant deadline”. The deadline is now!!!
- QoS The task gets a fixed amount of “service” per time unit.
Standards

There are several OS standards, trying to providing a uniform API:
A few examples are: POSIX, UNIX98, EL/IX, OSEK, uITRON, RTSJ (Real-Time Specification for Java), Ada 95, Real-Time CORBA
Some have been implemented in systems, other are so far only definitions.

POSIX
Portable Operating Systems Interface
A standard API for system calls of UNIX-like GPOS (and extended parts for RTOS).

Linux as a real-time system
- Non-preemptible kernel
- “Fair” sharing of resources
- “Timing slicing” scheduling
- Partly possible to change priority of processes making the scheduling nondeterministic
A standard Linux-kernel is a BAD choice for a RTOS

Efforts for making linux a better RTOS
Two different roads:
- Patching the standard kernel with more deterministic scheduling and adding scheduling points in the kernel
- A small Real Time OS underneath linux, running linux as one task (among others). E g Rtlinux & RTAI
Task management & scheduling in linux

Task - Both processes and threads

Processes
An executable entity with its own memory, registers, file descr PID etc. New process is created by fork etc

Threads
An executable entity sharing global variables, file descr, code area, heap etc but with its own thread status, program counter, registers etc. Faster creation, context switch and IPC. Can be prioritized.

pthread_create(...)

Tasklets
A function whose execution can be asked for, will be executed after the running context is done AND before next schedule.

Scheduling

Scheduling
Determining the order and timing with which the tasks should be scheduled

Dispatching
The dispatcher starts and stops the tasks

Scheduling points
When schedule is called: end of interrupt services, processes going to sleep or “scheduling time”.

The primary responsibility of an GPOS is to give all taska a fair amount of CPU and a fast average task throughput
The primary responsibility of an RTOS is to make sure all the tasks meet their timing constraints
Scheduling algorithms

Static Priority based
A static priority is given at task creation and the highest priority task is scheduled
Implies preemption, Lower priority tasks can be interrupted when higher priority tasks requests it.

Dynamic Priority based
- Rate monotonic. Priority depends on every tasks scheduling frequency
- Earliest deadline first. Priority depends on each tasks deadline. The earlier, the higher priority

Priority space's
Normally one for RTOS
Linux have 2
- user space. Preemted by tasks in kernel space
- kernel space. Kernel space have 3 levels
  - Interrupts, highest priority
  - Tasklets next priority
  - All other kernel tasks, which have higher priority than every user space task

Linux scheduler

round-robin
Time slicing. A task is moved to the tail of its priority queue when its time slice is over.

First in, first out
Once a task is scheduled, it keeps the processor until preempted by a higher priority task, sleeps or releases the processor voluntarily.

Other
Linux calculates a “godness” value, based on a number of rules. Default for kernel tasks, always for user tasks
**Interrupts**

Interrupts are processed by ISR, interrupt service routines

Fast ISR -> better real-time performance

Interrupt sources:

- timers
- keyboards
- DAQ's
- cards etc
- Ports, serial or parallel
- The CPU itself (illegal op. or SW exceptions)
RTAI

A module oriented Real Time Application Interface.

Provide us with a Hardware abstraction layer (remaps some of the “real” kernels functions for interaction with the hardware) and real time features through dynamically loaded modules.

The whole package consists of 5 complementary parts:

1. **RTHAL** Real Time Hardware Abstraction Layer
   - gather all the pointers to the required internal data and functions into a single structure, rthal, to allow an easy trapping of all the kernel functionalities that are important for real time applications, mainly related to the hardware, so that they can be dynamically switched to appropriate software emulation functions by RTAI when hard realtime is needed
   - makes available the substitutes of the above grabbed functions and sets rthal pointers to point to them
   - substitutes the original function calls with calls to the rthal pointers in all the kernel functions using them.

2. **Linux compatibility layer** Integrates RTAI tasks with Linux task manager

3. **RTOS core** Offers RT functionality for scheduling and interrupt processing

4. **LX/RT** Makes RT feature available in user space

5. **Extended functionality packages** IPC, networking, drivers, third part toolboxes, interfaces etc
RTAI system consists of a few hundred lines of changes in the Linux kernel code, and a set of modules for providing RT functionality:

- rtai_hal.ko
  - Initializes all of its control variables and structures
  - Makes a copy of the idt_table and of the Linux irq handlers entry addresses
  - Initializes the interrupts chips (ic) management specific functions
  - That does no arm to Linux but paves the way for succeeding easy mount/umount(s) of rtai.

  Nothing happens with Linux or the system until RTAI is “mounted”. After mounting all interrupts are filtered through RTAI:

  - Catch all the interrupts
  - If it is a Linux interrupts hard enable immediately the interrupt and pend it to be processed when there is no real time activity
  - If it is a real time only interrupt process it appropriately your own way, i.e. pass to a real time specific handler
  - If it is a real time interrupt shared also by Linux, process it appropriately your own way but the real time handler must care to end it also for later processing of its Linux part.

- rtai_sched.ko
  - The real time scheduler module, which is in charge of distributing the CPU to different tasks present in the system, including Linux.
  - The scheduling occurs when tasks perform certain system calls and on timer handler activation (incl interrupts)

- rtai_fifo.ko
  - Both between RT tasks and Linux processes

- rtai_ko_ko
  - Both between RT tasks and Linux processes

- rtai_lxr.t.ko
  - Makes all other services available also in user space

- rtai_sem.ko
- rtai_tasklets.ko
- rtai_mbx.ko
- rtai_serial.ko
- etc...etc

All modules are put under /usr/realtime/modules/
Installation

Patch linux-kernel
Build new kernel with support for RTAI
Build RTAI
Install RTAI modules

# cd /usr/src
# wget http://ftp.kernel.org/..../linux-2.6.10.tar.gz
# tar zxvf linux-2.6.10.tar.gz
# rm linux
# ln -s linux-2.6.10 linux
# wget http://http://www.aero.polimi.it/RTAI/rtai-3.3.tar.bz2
# tar xjvf rtai-3.3.tar.bz2
# ln -s rtai-3.3 rtai
# cd /usr/src/linux
# patch -p1 < ../rtai/base/arch/i386/patches/hal-linux-2.6.10-i386-r9.patch
# cp /boot/config-2.6.xx /usr/src/linux/.config (xx är tidigare sparad config-fil)
# make oldconfig  (Suggested answers are mostly the right choice)
# make menuconfig

Make sure that

- "Adeos" is selected (Adeos Support -> Adeos Support)
- "Loadable module support -> Module versioning support" is disabled
- "Kernel hacking -> Compile the kernel with frame pointers" is disabled
- "Processor type and features -> Use register arguments" is disabled (CONFIG_REGPARM)
- On some computer is it necessary to compile the network driver (at A410-labcomps e100) into the kernel, not as a module. Otherwise it won't work... -Have abs no idea why???)

# make

# make modules_install
# cp arch/i386/boot/bzimage /boot/kernel-2.6.10-rtai
# cd /usr/src/rtai
# make menuconfig
# make
# make install

Apply changes in /boot/grub/grub.conf
Reboot now, select the new kernel from the GRUB menu.

Test the new RT system:
# cd /usr/realtime/testsuite/kern/latency/
# ./run
Usage

**Kernel mode**
real time tasks
insmod necessary RT modules
Write your driver with RT parts (sem, mailboxes, tasks etc)
insmod your module

**User mode**
Application tasks, not real time
insmod necessary RT modules
Write your application and communicate with your driver with fifos or equivalent.
Look under /proc/adeos & /proc/rtai/ for a lot of informations about the RTAI-environment

Task management

```c
int rt_task_init (  
    RT_TASK *task,  
    void (*rt_thread)(int)  
    int data,  
    int stack_size,  
    int priority,  
    int uses_fpu,  
    void(*signal)(void)  
);  
```

Task is a pointer to an RT_TASK type structure whose space must be provided by the application. It must be kept during the whole lifetime of the real time task.

rt_thread is the entry point of the task function. The parent task can pass a single integer value data to the new task.

stack_size is the size of the stack to be used by the new task.

priority is the priority to be given the task. The highest priority is 0, while the lowest is RT_LOWEST_PRIORITY.

uses_fpu is a flag. Nonzero value indicates that the task will save the floating point registers at context switches. On a (multi) uni-processor, a real-time task does not save its floating point context by default. However, when the task is created for a symmetric multi-processing system, the floating point context is saved, because the task's context must be save against CPU migration anyway.

signal is an "ASR" function that is called, within the task environment and with interrupts disabled, when the task becomes the current running task after a context switch.
Here is a typical RTAI code for creating and starting a real-time task, from within an \texttt{init} \_\texttt{module()}, that periodically runs the function whose code is in the application dependent function \texttt{task1}():

```c
#define STACK\_SIZE 4096
static RT\_TASK mytask;
int init\_module(void)
{
    rt\_set\_periodic\_mode();
    rt\_task\_init(&mytask, task1, 0, STACK\_SIZE, 0, 1, 0);
    now = rt\_get\_time();
    rt\_task\_make\_periodic( &mytask, now + 2000, 100000000);
    return 0;
}

void cleanup\_module(void)
{
    stop\_rt\_timer();
    rt\_task\_delete(&my_task);
}
module\_init(init\_module);
module\_exit(clean-up\_module);
MODULE\_LICENSE("GPL");

// function that runs periodically in
// the created real-time task:
static void task1(int t) {
...
while (...) {
    ... // do what has to be done each period

    rt\_task\_wait\_period();
}
}
Dont forget exit and freeing resources!!
```
Intertask communication

- Synchronization
- Protecting resources
- Exchanges of data

Semaphores (Mutex), Messages, Mailboxes, fifos, queues

**Semaphores**

- `rt_sem_init(SEM *sem, int value)`
- `rt_sem_wait(SEM *sem)`
  1. Semaphore value decremented
  2. Tested
  3. If still positive return
  4. else block and return as soon someone has released the semaphore
- `rt_sem_signal(SEM *sem)`
  1. Semaphore value incremented
  2. Tested
  3. If not positive Run the first task in semaphores waiting queue
- `rt_sem_delete(SEM *sem)`

**Messages**

- `rt_send(RT_TASK *task, unsigned int msg)`
- `rt_receive(RT_TASK *task, unsigned int msg)`
  Both are blocked until sender/receiver has delivered.

**Fifos**

- `mknod -m 666 /dev/rtf<fifo> c 150 0 ; Where fifo is an int`
- `rtf_create(unsigned int fifo, int size)`
- `rtf_destroy(unsigned int fifo)`
- `rtf_put(unsigned int fifo, void *buf, int count)`

User application code

```c
fifo = open("/dev/rtf0", O_RDONLY)
read(fifo, tecken, sizeof(tecken))
```
Protecting common resources
- Mutual exclusion
- Producer / Consumer  Both mutual exclusion and conditional waiting
- Readers / Writer  1 writer, many readers

Synchronization
- Asymmetric synchronization
  1 task waits untils another task signals its ok to continue
  Ex Producer who signals to a consumer that data has been delivered

```c
sem_init(&sem, 0)
Process P1
{
   while(1) {
      .
      Store data
      signal(&sem)
   }
}
```

```c
Process P2
{
   while(1) {
      .
      wait(&sem)
      Read data
   }
}
```

- Symmetric synchronization
  2 taska waits for each other

```c
sem_init(&semDataReady, 0)
sem_init(&semDataReceived, 0)
Process P1
{
   while(1) {
      .
      Store data
      signal(&semDataReady)
      wait(&semDataReceived)
   }
}
```

```c
Process P2
{
   while(1) {
      .
      wait(&semDataReady)
      Read data
      signal(&semDataReceived)
   }
}
```
Readers / Writer

sem_init(&Mutex, 1)
sem_init(&N_readers_Mutex, 0)

Process writer
{
    while(1){
        wait(&Mutex)
        WRITE DATA
        signal(&Mutex)
    }
}

Process_reader
{
    while(1) {
        wait(&N_readers_Mutex)
        if ( N_readers == 0){
            wait(&Mutex)
        }
        N_readers++
        signal(&N_readers_Mutex)
        READ DATA
        wait(&N_readers_Mutex)
        N_readers--
        if(N_readers == 0){
            signal(&Mutex)
        }
        signal(&N_readers_Mutex)
    }
}