REAL TIME OPERATING SYSTEM
PROGRAMMING-II: Windows CE, OSEK and Real time Linux

Lesson-13:
RT Linux
1. RT Linux
RT Linux

- For real time tasks and predictable hard real time behaviour, an extension of Linux is a POSIX hard real-time environment using a real time core.
- The core is called RTLinuxFree and RTLinuxPro, freeware and commercial software respectively. V. Yodaiken developed RTLinux, later FSM Labs commercialized RTLinuxPro and now Wind River has acquired it.
RT Linux…

- Relatively simple modifications, which converts the existing Linux kernel into a hard real-time environment.
- Deterministic interrupt-latency ISRs execute at RTLinux core and other in-deterministic processing tasks are transferred to Linux.
RT Linux…

- The forwarded Linux functions are placed in FIFO with sharing of memory between RTLinux threads as highest priority and Linux functions running as low priority threads. Figure
RT Linux basic features

RTLinux Core for hard and soft real-time environment

1. Deterministic interrupt-latency ISRs execute at RTLinux core. Other non-deterministic processing tasks transferred to Linux

2. Primitive tasks with only statically allocated memory. No virtual memory allocation. Priority allocation to RT threads

3. Real-time task with no address space protection and run with disabling of interrupts

4. FIFO connects real-time task with Linux processes, shared memory perform synchronization between the hard real-time tasks and the limited size FIFO queues

5. Non-real-time processes of applications run as the Linux processes

6. RTLinux functions, Running RT Thread period definition, Real Time thread functions, Running RT Thread scheduling specifications
Running the task for hard real time performance

- Run the primitive tasks with only statically allocated memory.
- The dynamic memory allocation or virtual memory allocation introduces unpredictable allocation and load timings.
Running the task for hard real time performance…

- Run the real time task with no address space protection.
- The memory address protection involves additional checks, which also introduce the unpredictable allocation and load timings.
Running the task for hard real time performance...

- Run with disabling of interrupts so that other interrupts don’t introduce the unpredictability.
- Run a simple fixed priority scheduler.
Running the task for hard real time performance...

- Run with disabling of interrupts so that other interrupts don’t introduce the unpredictability.
- Run a simple fixed priority scheduler.
Running the task for soft real time performance...

- Applications can be configured to run differently.
- RTLinux allows flexibility in defining real-time task behaviour, synchronization and communication.
- RTLinux kernel designed with modules, which can be replaced to make behaviour flexible wherever possible.
Running the task for non real time tasks

- Applications run as the Linux processes.
2. Programming with RT Linux
Example RT Linux

- include rtl.mk    /* Include RTLinux make file. The rtl.mk file is an include file which contains all the flags needed to compile the code. */
- all: module1.o    /* Object file at module1.o */
- clean: rm -f .o   /* Remove using function rm object files inserted before this file */
- module1.0: module1.c    /* module1.0 is object file of source file module1.c */
- $(cc) ${include} ${cflags} -c module1.c    /* Compile, include, Cflags C module module1.c */
3. Functions in RT Linux
## Module and thread functions in RT Linux

<table>
<thead>
<tr>
<th>Task</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>init insmod, claeup, and rmmod</td>
<td>The <code>init_module()</code>, <code>insmod_module()</code>, <code>cleanup()</code> and <code>rmmod_module()</code> functions are the same as Linux.</td>
</tr>
<tr>
<td>RT Linux functions</td>
<td>1. <code>rtl_hard_enable_irq();</code> /* Enables hard real time interrupts */</td>
</tr>
<tr>
<td></td>
<td>2. <code>rtl_hard_disable_irq();</code> /* Gets current clock schedule */</td>
</tr>
<tr>
<td></td>
<td>3. <code>rtlunix_signal();</code> /* to RTLinux signal handling function of the application */</td>
</tr>
<tr>
<td></td>
<td>4. <code>rtlgetsclock();</code> /* Gets current clock schedule */</td>
</tr>
<tr>
<td></td>
<td>5. <code>rtl_request_irq();</code> /* to install real time interrupt handler */</td>
</tr>
<tr>
<td></td>
<td>6. <code>rtl_restore_interrups();</code> /* Restores the CPU interrupts state */</td>
</tr>
<tr>
<td></td>
<td>7. <code>rtl_stop_interrups();</code> /* Stops the CPU interrupts */</td>
</tr>
<tr>
<td></td>
<td>8. <code>rtl_printf();</code> /* to print text from real time thread */</td>
</tr>
<tr>
<td></td>
<td>9. <code>rtl_no_interrups();</code> /* No CPU interrupts permitted */</td>
</tr>
<tr>
<td>Thread creation and getting thread ID</td>
<td>Creation and getting ID is same as Linux threads <code>pthread_create()</code> and <code>pthread_self()</code>.</td>
</tr>
<tr>
<td>Semaphore and mutex functions</td>
<td>Same as in Linux.</td>
</tr>
<tr>
<td>Thread wait</td>
<td><code>pthread_wait_np();</code> /* thread waits for other thread to complete */</td>
</tr>
</tbody>
</table>
Functions in RT Linux

1. struct sched_param thrd1, thrd2; /* Defines two structures for threads thrd1, thrd2 assignment parameters. */
2. thrd1.sched_priority = 1; /* Defines thrd1 scheduled priority parameter = 1 (=high). Similarly thrd2 priority can be defined */
3. pthread_setschedparam (pthread_self (), SCHED_FIFO, & thrd1; /* Function setschedparam arguments are self function, schedule defines as FIFO and thread structure thrd1. Similarly thrd2 arguments for Function setschedparam can be defined */ thrd1_put(fid, &queuebuff, 1); /* Function thrd1_put arguments are fid (= an integer for FIFO descriptor ID, say, 1, 2, 3, ..), address of queue buffer queuebuff and an option = 1 */
Real Time Thread Functions in RT Linux

1. pthread_attr_setcpu_np(); /* Set CPU pthread attributes. */
2. pthread_attr_getcpu_np(); /* Get CPU pthread attributes. */
3. pthread_attr_setfp_np(); /* Set CPU pthread floating point enable attributes. */
4. pthread_suspend_np(); /* Suspends thread run. */
5. pthread_wakeup_np(); /* Wakes up the thread. */
6. pthread_wait_np(); /* Wait for start (message or signal) the thread. */
7. pthread_setfp_np(); /* Allows floating point arithmetic. */
8. pthread_delete_np(); /* Delete the thread. */
Real Time FIFO functions

1. `rtf_create()`, `rtf_create_rt_handler()`, `rtf_open()`, `rtf_close()` and `rtf_unlink()` are the functions to create FIFO device, create real-time handler, initialize, close and remove a named RT FIFO. `unlink()` does not destroy the queue immediately but prevents the other tasks from using the queue. The queue will get destroyed only if the last task closes the queue. Destroy means to de-allocate the memory associated with queue ECB.

2. `rtf_setattr()` sets the attributes.

3. `rtf_lock()` and `rtf_unlock()` lock and unlock a queue.
Real Time FIFO functions

3. rtf_lock ( ) and rtf_unlock ( ) lock and unlock a queue.

4. rtf_put (fid, &queuebuff, 1); /* Function rtf_put arguments are fid (= an integer for FIFO descriptor ID, say, 1, 2, 3, ..), address of queue buffer queuebuff and an option = 1 */.

5. rtf_get (fid, &queuebuff, 1); /* Function rtf_get arguments are fid (= an integer for FIFO descriptor ID, say, 1, 2, 3, ..), address of queue buffer queuebuff and an option = 1 */.

6. rtf_send ( ) and rtf_receive ( ) to send and receive into a queue. rtf_send four arguments are fid (FIFO device ID), (void *) &qsendingdata (pointer to address of user data), sizeof (qsendingdata) and 0. mq_receive five arguments are msgid (message queue ID), (void *) &qreceivedata (pointer to address of bufferdata), sizeof (qreceivingdata), 0 and 0.
Real Time FIFO functions

7. rtf_notify() signals to a single waiting task that the message is now available. The notice is exclusive for a single task, which has been registered for a notification (registered means later on takes note of the rtf_notify).

8. rtf_getattr() retrieves the attribute of a RT FIFO.

9. rtf_flush() flushes the data of a RT FIFO.

10. rtf_allow_interrupts(); /* Controls real time interrupt handler */.

11. rtf_free_irq(); /* Frees real time interrupt handler */.
Summary
We learnt

- RTLinux provides hard real functionalities in a separate layer, which runs the primitive tasks with only statically allocated memory, no dynamic memory allocation, no virtual memory allocation, no address space protection, run with disabling of interrupts, runs a simple fixed priority scheduler,
We learnt

- It provides for running of real time tasks by RTLinux layer and no deterministic non real time tasks by Linux.
- A FIFO connects real time tasks with Linux processes,
- Synchronization between the hard real time tasks and the limited size FIFO queues is achieved through use of shared memory (not through IPCs).
We learnt

- RTLinux separate functions
  `rtl_hard_enable_irq();`
  `rtl_hard_disable_irq();`
  `rtlinux_sigaction();`
  `rtl_getschedclck();`
  `rtl_request_irq();`
  `rtlRestore_interrups();`
  `rtl_stop_interrupts();`
  `rtl_printf();`
  `rtl_no_interrupts();`
We learnt

- RTLinux supports priority for the real time threads. RTLinux has real time thread wait, thread period definition, thread deletion, priority assignment, and FIFO device-functions.
End Lesson-13 on RT Linux