Introduction to Real-Time Operating Systems

Mahesh Balasubramaniam

What is an RTOS?

- An RTOS is a class of operating systems that are intended for real time-applications
- What is a real time application?
- A real time application is an application that guarantees both correctness of result and the added constraint of meeting a deadline

So what is an RTOS?

- An operating system which follows the Real Time criteria.
- ² *Efficiency, Predictability* and *Timeliness* – important
- – All components of an RTOS must have these properties.
- Some tasks which may delay things:
- – Interrupt Processing, Context Switching, Inter-task communication,

So what is an RTOS ?(contd)

- IO
- To cut back on (variable) overhead for these tasks:
- – Multiprogramming, Memory Management, File (and other) IO, IPC,
- etc.

So what makes an RTOS special?

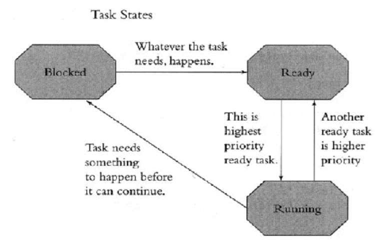
- An RTOS will provide facilities to guarantee deadlines will be met
- An RTOS will provide scheduling algorithms in order to enable deterministic behavior in the system
- An RTOS is valued more for predictability than throughput

Design Philosophies

- Some of the design philosophies of an RTOS are with respect to:
- Scheduling
- Memory allocation
- Inter task communication
- Interrupt handlers

Tasks

- Task States:
  - Running
  - Ready (possibly: suspended, pended)
  - Blocked (possibly: waiting, dormant, delayed)
  - Scheduler – schedules/shuffles tasks between Running and Ready states
  - Blocking is self-blocking by tasks, and moved to Running state via other tasks' interrupt signaling (when block-factor is removed/satisfied)
  - When a task is unblocked with a higher priority over the 'running' task, the scheduler 'switches' context immediately



Scheduling

- The data structure of the ready list in the scheduler is designed so as to minimize the worst-case length of time spent in the scheduler's critical section
- The critical response time, sometimes called the flyback time, is the time it takes to queue a new ready task and restore the state of the highest priority task. In a well-designed RTOS, readying a new task will take 3-20 instructions per ready queue entry, and restoration of the highest-priority ready task will take 5-30 instructions.

Intertask Comm. & resource sharing

- It is "unsafe" for two tasks to access the same specific data or hardware resource simultaneously.
- 3 Ways to resolve this:
- Temporarily masking/disabling interrupts
- Binary Semaphores
- Message passing

Memory Allocation

- Speed of allocation
- Memory can become fragmented

Interrupt Handling

- Interrupts usually block the highest priority tasks
- Need to minimize the unpredictability caused

Linux as an RTOS

- Is Linux an RTOS?
- Linux provides a few basic features to support real-time applications
- Provides soft-real time guarantees
- SCHED_FF and SCHED_RR are 2 scheduling policies provided

Problems with Linux

- Use of Virtual Memory
- Use of shared memory
- Does not support priority inheritance

RTLinux and RTAI

- Variants of Linux with support for real-time applications
- They both use a real-time kernel which interacts with the main Kernel
- They treat the Linux OS as the lowest running task

RTLinux : Mechanics behind the Kernel

Sudhanshu Sharma

Outline **RTLinux**

➔Build Up

Real time Linux – The various forms

RTLinux – Architecture

RTLinux - Internals

Examples

Build Up RTLinux

Kernel (Wikipedia) : "As a basic component of an operating system, a kernel provides the lowest level of abstraction layer for the resources (especially memory, processors and I/O devices) that applications must control to perform their function"

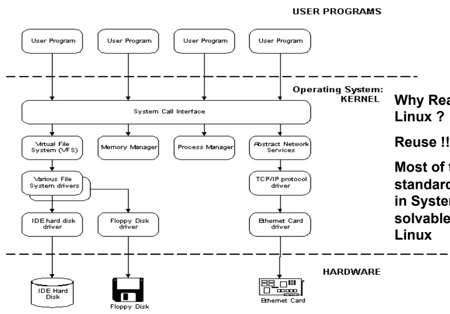
-Process Management

-Memory Management

-Device Management

-System Calls

Build Up — RTLinux



Why Real Time Linux ?

Reuse !!!

Most of the standard problems in Systems already solvable through Linux

Build Up — RTLinux

Any RT system application can be divided into 2 parts

-Real Time task/process (Temporal properties Imp.)

-Non Real Time task/process (Temporal properties not as Imp.)

Ideology behind RTLinux :

Extend the existing source to provide the standard functionalities at the same time provide a framework that can guarantee Hard Real Time requirements to be fulfilled.

Linux an obvious choice ---

- Open source
- Vast User/Developer base of Linux

Outline — RTLinux

Build Up

➔Real Time Linux Approaches

RTLinux – Architecture

RTLinux - Internals

Examples

Real Time Linux Approaches — RTLinux

3 broader paradigms to solve RTOS problem :

1) Providing Non real time Services to the basic real time kernel (eg. VxWorks)

2) Preemption Improvement in Standard kernel (preempt patch for Linux kernel)

3) Virtual Machine Layer to make standard kernel Pre-emptable (RTLinux /RTAI)

Real Time Linux Approaches — RTLinux

RTAI & RTLinux comparisons

-In essence both RTAI and RTLinux execute Real Time tasks in the kernel memory space preventing RT threads to be swapped out

- Dynamic Memory allocation in RTAI while RTlinux still uses static allocation
- Shared Memory (Linux <-> RTLinux) provided by both
- IPC functions in RTAI are more extensive FIFO , Mailboxes, Messg. Q's,net_rpc
- POSIX Mutex , Conditional Variables, Semaphores provided in both
- User space real time (Protection) – Provided only in RTAI called LXRT services

-RTLinux only provides user space real time signals.

No interaction with RTservices or Linux System Calls possible in those handlers.

Outline — RTLinux

Build Up

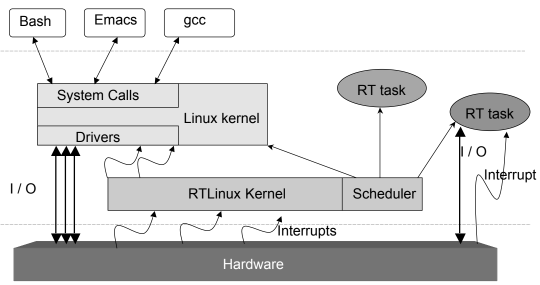
Real time Linux – The various forms

➔RTLinux – Architecture

RTLinux - Internals

Examples

Architecture — RTLinux



Architecture — RTLinux

- Standard time sharing OS and hard real time executive running on same machine

- Kernel provided with the emulation of Interrupt control H/W

-RT tasks run at kernel privilege level to provide direct H/W access

- RTLinux runs Linux kernel for booting , device drivers ,networking, FS, process control and loadable kernel modules

- Linux kernel runs as a low priority task on the RT kernel hence making sure it cannot preempt any RT task

Architecture — RTLinux

-RT task allocated fixed memory for data and code (Pain Sounds Familiar !!!!)

-RT tasks cannot use Linux system calls , directly call routines or access ordinary data structures in Linux Kernel

-RTProcesses without memory protection features (RTLinux Pro has some PSDD now)

- The RT kernel is actually patched over the Linux kernel and then recompiled to work as a RTLinux system

- Completely configurable RTLinux kernel

Architecture — RTLinux

VM layer only emulates the Interrupt Control

The 0 level OS does not provide any basic services that can be provided by Linux

- Only RT services
- No primitives for process creation, switching or MM

Uses software stack for switching and not the expensive H/W switching

RT KERNEL IS NOT PREEMPTABLE

Outline — **RTLinux**

Build Up

Real time Linux – The various forms

RTLinux – Architecture

➔RTLinux - Internals

Examples

Internals — RTLinux

Here comes the nuts and bolts of implementation …

We will cover 4 important aspects of the RTLinux internals

1) Interrupt Handling

2) IPC toolsets – RT- FIFO & Shared Memory

3) Clock and Timers

4) Scheduling

Internals — RTLinux

Interrupt Handling

- Traditional calls (PSW) of sti() and cli() are modified to use the Software interrupts
- Wrapper routines written to save and restore state at return from software Interrupt
- Interrupt Handlers in RT executive perform whatever functions are required and passes interrupts to Linux
- In Most I/O , RT device interrupts simply notify Linux

Internals — RTLinux

Interrupt Handling

- Timer interrupt increments timer variable and determines whether RT task needs to run and passes interrupts to Linux at appropriate intervals
- All Linux "wrappers" modified to fix stacks to cheat Linux to believe H/W interrupt and hence kernel mode execution ensured
- S_IRET is used to save minimal state and look for pending interrupts (call other wrappers) otherwise restore registers and return from Interrupt

Internals — RTLinux

Interrupt Handling – APIs

rtl_request_irq() - Add RT Interrupt Handler

rtl_free_irq () – Remove RT Interrupt Handler

rtl_get_soft_irq ()- Install Software interrupt Handler

rtl_free_soft_irq ()- Remove Software interrupt Handler

rtl_global_pend_irq ()- Schedule a Linux Interrupt

Internals — RTLinux

RT- FIFO

This provides the primary mechanism through which the RT processes interact with the Linux processes

RT tasks should pre allocate the default size of FIFO and the number

RTKernel Config has options for–>

- Pre Allocated FIFO Buffer
- Max number of FIFO

Internals — RTLinux

RT- FIFO - Asynchronous I/O

Example – Linux process producing data and RT process is the consumer tht writes nothing back

```
void fifo_handler (int sig,rtl_siginfo_t *sig,void *v){
    char msg[64];
    read(sig->si_fd,&msg,64);
}
void fifo(void)
{
    int fd; struct rtl_sigaction sigaction;
```

Internals — RTLinux

RT- FIFO - Asynchronous I/O

```
...  //create FIFO
    mkfifo("/myfifo",0755); //if 2 arg. 0 then Linux cant see it

    // open FIFO for read
    fd = open("/myfifo",0_RDONLY|0_NONBLOCK);
    // register a SIGPOLL handler for FIFO
    sigaction.sa_sigaction = fifo_handler;
    //file that we want signal for
    sigaction .sa_fd =fd;
    //write event notification
    sigaction.sa_flags = RTL_SA_RDONLY | RTL_SA_SIGINFO;
    //install handlers
    rtl_sigaction(SIGPOLL,&sigaction,NULL);
    unlink("/myfifo");
}
```

Internals — RTLinux

RT- FIFO

Only one SIGPOLL handler installed at a given time

Many fd share the same FIFO but only one SIGPOLL handler

Internals — RTLinux

Shared Memory

Almost the same principle it uses POSIX RT extensions

- shm_open("file",0_CREATE,0)   // 0755 to allow Linux to use it
- shm_unlink()
- mmap() //Area created needs to be mapped

Reference Count

- Maintained so that shm_unlink() / unlink() don't wipe out the resource in use across Linux or RTLinux processes

Internals — RTLinux

Clocks & Timers

- Clocks used to manage time in computers –Clocks control API's
- Timers is H/w or S/w allow functions to be evoked at specified time in future
- Multi task systems need timers for each one of them hence S/w timers used
- Timer Interrupt will trigger task schedule at specified moments (One shit timers) – Timer management API support

Internals — RTLinux

Schedulers

RM Scheduler provided

EDF Scheduler provided

Also the Scheduler can be loaded at run time hence more complex extensions are possible

Examples — RTLinux

```
// Program in Linux writing to FIFO //rtf3 for
Sound

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

#define DELAY 30000

void make_tone1(int fd)
{
        static char buf = 0;
        write (fd, &buf, 1);
}
void make_tone2(int fd)
{
        static char buf = 0xff;
        write (fd, &buf, 1);
}
…..
```

```
…..

main()
{
        int i, fd = open ("/dev/rtf3", O_WRONLY);
        while (1)
        {
                for (i=0;i<DELAY;i++);
                make_tone1(fd);
                for (i=0;i<DELAY;i++);
                make_tone2(fd);
        }
}
```

Examples — RTLinux

```
// RT process doing the same work

#include <rtl.h>
#include <pthread.h>
#include <rtl_fifo.h>
#include <time.h>

#define FIFO_NO 3
#define DELAY 30000
pthread_t thread;
void * sound_thread(int fd)
{
        int i;
        static char buf = 0;
        while (1)
        {
                for(i=0; i<DELAY; i++);
                                buf = 0xff;
                rtf_put(FIFO_NO, &buf, 1);
                for(i=0;i<DELAY;i++);
                buf = 0x0;
                rtf_put(FIFO_NO, &buf, 1);
        }
        return 0;
}
```

```
int init_module(void)
{
        return pthread_create(&thread, NULL, sound_thread,
NULL);
}

void cleanup_module(void)
{
        pthread_delete_np(thread);
}

// Finally a kernel module that will be a RT thread and
// will read the char device //rtf3 to produce sound
```

References

RTLinux

Yodaiken, Victor . An Introduction to RTLinux [www]
<http://www.linuxdevices.com/articles/AT3694406595.html>

Dougan,Court and Matt Sherer. RTLinux POSIX API for IO on Real Time FIFOs and Shared Memory

Yodaiken, Victor and Barabanov, Michael .A Real-Time Linux

V. Esteve, I. Ripoll and A. Crespo. Stand-Alone RTLinux-GPL

She Kairui, Bai Shuwei, Zhou Qingguo, Nicholas Mc Guire, and Li Lian.
Analyzing RTLinux/GPL Source Code for Education

Ismael Ripoll (2002). RTLinux versus RTAI

Yodaiken, Victor and Barabanov, Michael .FSMLabs RTLinux PSDD: Hard Realtime with Memory Protection

Michael Pettersson,Markus Svensson .Memory Management in VxWorks compared to RTLinux