6. Real-Time Operating Systems

6.2 Real-Time Extensions for Linux

Roadmap of Section 6.1

- History, Overview
- Real-Time Linux
  - Concept, Architecture
  - API, Modules
    - Physical Memory Access, I/O, Interrupt Handling, Periodic Threads, IPC
  - Samples
- RTAI vs. RTLinux vs. RTLinux Pro
- Literature

RTLinux History

- Developed at the New Mexico Institute of Technology by Michael Barabanov under the direction of Professor Victor Yodaiken
- Development, ownership and rights were moved FSMLabs (Finite State Machine Labs)
- Version 2 introduced in October 1999
- Version 3 February 2001
  - Available for X86, PowerPC and Alpha (MIPS beta)
- Lot of RTLinux'es available all using the same idea

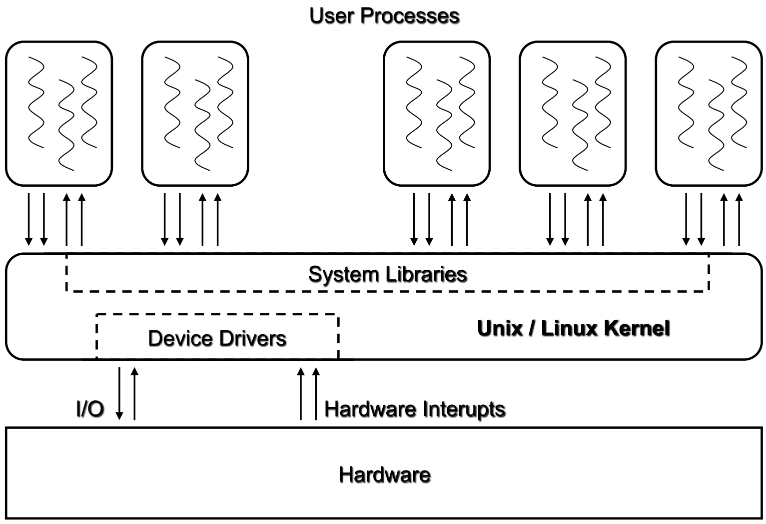
RTLinux Overview

- Adds hard real-time capabilities to Linux
  - Interrupt Emulation / Kernel Source Patch
  - Normal Linux Processes run as idle task of RT Core
  - Real-Time IPC between RT-tasks and Linux Processes
  - Periodic Threads
  - High Resolution Timer
  - Real-Time Scheduler

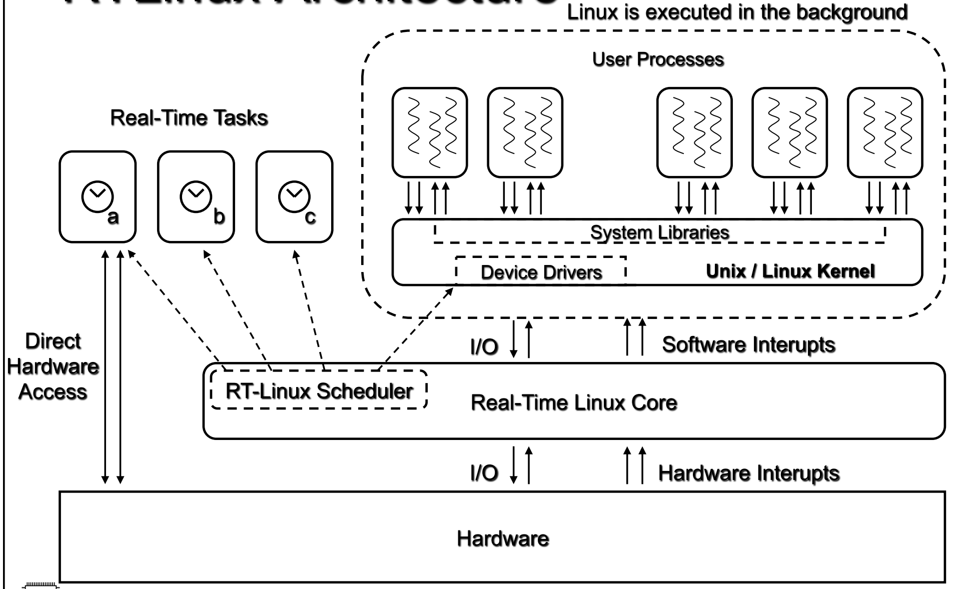
Motivation for a Real-Time Linux

- Standard operating system offers rich set of services, tools
- Usage of standard development tools (gcc, gdb)
- Linux is a free operating system, while most special purpose OS are expensive
- RT-Kernel Code changes possible
- Usage of existing Know-How
- POSIX compatible

Linux Architecture



RTLinux Architecture



Basic Concept: Interrupt Emulation

- Layer of emulation software between the Linux kernel and the interrupt controller hardware
- In the Linux source code all occurrences of cli, sti and iret instructions are replaced with emulating macros S_CLI S_STI and S_IRET
- All hardware interrupts are caught by the emulator
- Linux has no direct control over the interrupt controller it does not influence processing of realtime interrupts that do not pass through the emulator

Interrupt Emulation: Soft Interrupts

- Disabling a hardware interrupt resets a variable within the emulator
- When an interrupt occurs the variable is checked and if set the Linux interrupt handler routine invoked
- If the variable is disabled the handler will not be invoked and a bit is set in the variable that holds the information about all pending interrupts
- Re-enabling interrupts causes all pending Linux interrupt handlers to be invoked

```
S_CLI:  movl $0, SFIF

S_STI:  sti
        pushfl
        pushl $KERNEL_CS
        pushl $1f
        S_IRET
1:
```

Embedded Systems Programming

9

Interrupt Emulation: S_IRET Macro

- Save data register to access global variables
- Bitmask representing all unmasked pending interrupts is scanned for a set bit
- If no pending interrupt was found the interrupt state variable is set and a hard return from interrupt is performed
- If an interrupt was found a jump is made to the Linux handler

```
S_IRET: push %ds
        pushl %eax
        pushl %edx
        movl $KERNEL_DS ,%edx
        mov %dx,%ds
        cli
        movl SFREQ, %edx
        andl SFMASK, %edx
        bsfl %edx, %eax
        jz 1f
        S_CLI
        sti
        jmp  SFIDT(,%eax,4)
1:      movl $1, SFIF
        popl %edx
        popl %eax
        pop %ds
        iret
```

Embedded Systems Programming

10

Interrupt Emulation: Interrupt Handler

```
if(real-time linux handler registered)
  call real-time linux handler
if(softinterrupts enabled)
  call linux interrupt handler
else mark interrupt as pending
iret
```

- interrupt vector table overwriten by real-time linux patch

RTLinux Modules

- rtl_core.o – main module
- rtl_time.o-controls processor clocks
- rtl_sched.o-implements a real-time scheduler
- rtl_posixio.o-provides a POSIX-like interface to device drivers
- rtl_fifo.o-creates a real-time non-blocking FIFO implementation between real-time modules and user-space processes
- mbuff.o-provides a shared memory between real-time tasks and user-space processes
- rtl_ipc.o-provides POSIX-style blocking mutexes and semaphores
- rtl_debug.o-adds support for a source-level debugger
- rtl_com.o-interface with serial ports

Threads

- Posix Thread API for Real-Time Threads
- All real-time tasks are threads in one rt-process per processor

```
int pthread_create(pthread_t * thread,
                   pthread_attr_t * attr,
                   void * (*thread_code)(void*),
                   void * arg);
```

- **pthread_join()**
- **pthread_delete_np()**
- **pthread_attr_getcpu_np()**
- **pthread_attr_setcpu_np()**

Threads Scheduling

```
int pthread_setschedparam(pthread_t thread, int policy,
   const struct sched_param *param);

int sched_get_priority_max(int policy); // 1000000
int sched_get_priority_min(int policy); // 0 = min prio
struct itimerspec {
   struct timespec it_interval; /* timer period */
   struct timespec it_value; /* timer expiration */
};

int pthread_make_periodic_np(pthread_t thread, const
   struct itimerspec *its);

int pthread_wait_np(void);
```

```
#include <rtl.h>
#include <time.h>
#include <pthread.h>
pthread_t thread;
void * thread_code(void)
{
   pthread_make_periodic_np(pthread_self(),gethrtime(),10000000);

   while (1) {
        pthread_wait_np ();
        rtl_printf("Hello World\n");
   }
   return 0;
}
int init_module(void) {
   return pthread_create(&thread, NULL, thread_code, NULL);
}
void cleanup_module(void) {
   pthread_delete_np(thread);
}
```


Scheduler Implementations

- Original scheduler: priority-based FIFO, one-shot
  - 1000000 priorities
  - Not a good performance with tasks > 20
- EDF and RMS scheduler available
- One-shot mode
  - Reprogramming of timer chip at each scheduling decision
- Periodic Modes – timer chip programmed once
  - Better performance, not all periods available

RTLinux Inter Process Communication

- Real-Time FIFOs
  - Implemented using soft interrupts
  - Non-blocking real-time interface
  - Communication between real-time and non-real-time tasks
  - Character device for normal Linux processes
- Shared Memory
  - Support of mmap() in posixio.o

Real-Time FIFOS API

```
int rtf_create(unsigned int fifo, int
  size);
// fifo is a value unique within the
  system, and must be less than RTF_NO
int rtf_create_handler(unsigned int fifo,
  int (* handler)());
int rtf_get(unsigned int fifo, char * buf,
  int count);
int rtf_put(unsigned int fifo, char * buf,
  int count);
int rtf_destroy(unsigned int fifo);
```

Synchronisation: Mutex

- Initialisation

```
int pthread_mutex_init(pthread_mutex_t
*mutex, const pthread_mutexattr_t *attr);
```

- Locking a Mutex

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

- Unlocking a Mutex

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- Mutex options:
  - Lock counts, error checks, priority ceiling

Synchronisation: Semaphores

- Initialisation of an unnamed Semaphore

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared,
             unsigned int value);
```

- Signal a semaphore (unblock)

```
int sem_post(sem_t *sem);
```

- Synchronous Wait

```
int sem_wait(sem_t *sem);
```

- Non-Blocking Wait

```
int sem_trywait(sem_t *sem);
```

Physical Memory and I/O Port Access

- Output a byte to a port:

```
#include <asm/io.h>
void rtl_outb(char value, short port)
```

- Output a word to a port:

```
#include <asm/io.h>
void rtl_outw(unsigned int value, unsigned short
port)
```

- Read a byte from a port:

```
#include <asm/io.h>
char rtl_inb(unsigned short port)
```

- Read a word from a port:

```
#include <asm/io.h>
short rtl_inw(unsigned short port)
```


Interrupt Handling: Soft Interrupts

- Soft interrupts are normal Linux kernel interrupts
- some Linux kernel functions can be called from them safely
- do not provide hard real-time performance

```
// allocates a virtual irq number and installs the
   handler function for it
int rtl_get_soft_irq(void
      (*handler)(int, void *, struct pt_regs *),
      const char * devname);
//triggers virtual interrupts
void rtl_global_pend_irq(int ix);

void rtl_free_soft_irq(unsigned int irq);
```

Interrupt Handling: Hard Interrupts

- **Very low latency**
- **Usage of very limited function set**

```
#include <rtl_core.h>
int rtl_request_irq(unsigned int irq,
  unsigned int (*handler) (unsigned int,
  struct pt_regs *));
```

- handler will be executed with hardware interrupts disabled
- We have to reenable the interrupt line with the method **`rtl_hard_enable_irq()`**

```
int rtl_free_irq(unsigned int irq);
```


Timing API

```
#include <rtl_time.h>
int clock_gettime(clockid_t clock_id, struct timespec
   *ts);
hrtime_t clock_gethrtime(clockid_t clock);
```

Currently supported clocks are:

- CLOCK_MONOTONIC: This POSIX clock runs at a steady rate, and is never adjusted or reset.
- CLOCK_REALTIME: standard POSIX realtime clock.
- CLOCK_RTL_SCHED: The clock that the scheduler uses for task scheduling.

CLOCK_8254: Used on non-SMP x86 machines for scheduling.

- CLOCK_APIC: Used on SMP x86 machines. This corresponds to the local APIC clock of the processor that executes clock_gettime. You cannot read or set the APIC clock of other processors.

Implementing RTLinux Applications

- Only hard real time tasks should be implemented as RT-modules
- Do as much as possible in non-real time Linux processes
  - GUI, File System I/O, Networking, DB-Access…
- Be careful while implementing real-time tasks
  - Whole system can hang
  - Use debugger
- There is no memory protection in kernel space

Higher Striker
Real-Time Linux and Periodic Threads

- rtLinux can schedule Threads up to 40 kHz periodically / low jitter (100Mhz CPU)
- Buffers are read/written each period
- Experiment data must be sampled every 13µs because of sampling theorem
- Table shows write ahead buffer that must be used

Iterations / period	Busy waiting	13 µs	26 µs	260 µs
100000 ~ 2s	1	1	41	48
1000000 ~ 26s	1	40	50	59
other processes / interactive reaction time	almost not active	very slow	slow	Almost normal

J.Gressmann, B. Kaufmann 2004

Real-Time Linux

```
for (r = 0; r < runs; ++r) {
   initialize(writeAheadBuffer, writeAhead);
   start();
   for (i = 0; i < iterations; ++i) {
        LukasResult result;
        writeMS(byte); readLS();
        status = readStatus();
        if (TEST_EMPTY_MS(status))
        update(result);
        while (TEST_EMPTY_LS(status)) {
                pthread_wait_np();
        status = readStatus();
   }
}
stop();
rtf_put(fifo, &result, sizeof(LukasResult));
}
```


Installing RTLinux

- Download new Kernel Source
- Patch Kernel witch RTLinux Patch
- Configure Kernel
- Build new patched Kernel
- Install Kernel
- Reboot
- Start RTLinux modules
- Insert your own RT module
- Changed API for some modules

Real-Time Linux Implementations

- RT-Linux
  - ftp://ftp.fsmlabs.com/pub/rtlinux
- RTAI
  - ftp://www.aero.polimi.it/RTAI/
- KURT
  - http://www.ittc.ukans.edu/kurt/
- Linux/RK
  - http://www.cs.cmu.edu/~rajkumar/linux-rk.html
- RED-Linux
  - http://linux.ece.uci.edu/RED-Linux/SDK/
- ART Linux
  - http://www.etl.go.jp/etl/robotics/Projects/ART-Linux/
- SMART-Linux
  - http://www.ime.usp.br/~dilma
- Linux-SRT
  - http://www.uk.research.att.com/~dmi/linux-srt/
- QLinux
  - http://www.cs.umass.edu/~lass/software/qlinux/

Real-Time Application Interface (RTAI)

- Developed at the Dipartimento di Ingeneria Aerospaziale, Politecnico di Milano by Professor Paolo Mantegazza
- Common approach to rtLinux, supports original rtLinux API, extended features
- Floating point support
- Supports x86, PowerPC, Arm, MIPS, Cris
- Open Source

Typical Performance

- RTAI on a Pentium II, 233MHz
- simultaneously servicing Linux, which was working under a heavy load
- Maximum periodic task iteration rate: 125KHz
- Typical sampling task rate: 10KHz (Pentium 100)
- Jitter at maximum task iteration rate: 0-13µs UP, 0-30µs SMP
- One-shot interrupt integration rate: 30KHz (Pentium-class CPU), 10KHz (486-class CPU)
- Context switching time: approximately 4µs

FSMLabs RTLinuxPro 2.0 Features

- Improved scheduler performance
- Lnet: hard real-time networking API for communication over Ethernet or Firewire
- Improved documentation
- Test/validation tools for RT-modules
- Removed kernel module semantic of rt-modules
- Standard C-programs can be real-time
  - rtl module loader

RTLinuxPro FIFOs

```
int mkfifo(const char *pathname, int mode);
```

- Extended FIFO implementation
- Integration into the Linux file system
- Support of security attributes

```
mkfifo("/mydev2", 0777)
fd2 = open("/mydev2",O_NONBLOCK);
ftruncate(fd2, 4096);
```


RTLinuxPro Example: A Real-Time Thread

```
pthread_t thread;

void *thread_code(void *t)
{
        struct timespec next;
        int count = 0;

        clock_gettime( CLOCK_REALTIME, &next );

        while ( 1 ) {
                timespec_add_ns( &next, 1000*1000 );
                clock_nanosleep( CLOCK_REALTIME, TIMER_ABSTIME,
                                 &next, NULL);
                count++;
                if (!(count % 1000))
                        printf("woke %d times\n",count);
        }

        return NULL;
}
```

RTLinuxPro Example: Main

```
int main(void)
{
        pthread_create( &thread, NULL, thread_code, (void *)0 );

          rtl_main_wait();

          pthread_cancel( thread );
          pthread_join( thread, NULL );

          return 0;
   }

./hello.rtl
```


Low Latency Kernel Patches

- Monolithic kernel and interrupt handling causes long scheduling delays,
- Low Latency Patch
  - Insertion of rescheduling points into kernel (cooperative scheduling)
  - latency = max. time between 2 rescheduling points
  - RED Linux
- Preemptable Linux
  - Allow more than one execution flow in kernel
  - Kernel structures have to be protected by synchronization mechanisms (Mutex, Spinlock)

CPU-Shielding - Real-Time for SMP

- Developed by Concurrent Computer Corporation
- Implemented in RedHawk Linux, Suse Enterprise Real-time Linux
- Applicable for symmetric multiprocessor systems
- High-priority tasks and interrupts are bound to a more shielded CPU
- Shielded CPU's are protected/shielded from unpredictable processing activities
- Configuration via processor affinity (processes and interrupts)

Literature

- "A Linux-based Real-Time Operating System", Michael Barabanov (Thesis)
- "The RTLinux Manifesto", Victor Yodaiken
- Finite State Machine Labs : www.fsmlabs.com
  - Tutorials, Manuals, RTLinux Sources
- http://www.mrao.cam.ac.uk/~dfb/doc/rtlinux/GettingStarted/node42.html
- RTAI : www.rtai.org