About me

- **Martin Bis**
- GNU/Linux (from administration to kernel programming)
- Embedded Linux
- Trainings, consulting, support (http://bis-linux.com)

- Industrial appliances (Linux + Real-Time)
Linux (and Open Source) is being increasingly used in industrial control devices (previously reserved for classic RTOSes).

- Hardware capable of running Linux become cheaper.
- Linux supports lots of hardware, as well as communication protocols.
- Security is easy to achieve (security \(\neq\) safety critical)
- Code is easy to develop and reusable.

More and more such applications must work in Real-Time.
Weight dosing in injection-molding process (plastics industry). Application based on **TNKernel**.

- [http://www.tnkernel.com/](http://www.tnkernel.com/)
- compact and very fast real-time kernel for the embedded 32/16/8 bits microprocessors (Atmel AT91 chip in our case).
- Open-Source software
- Elegant code.
- Well tested (many previous success-stories).

**But:** - customer needs:

- a touchscreen
- logs pulled via FTP
- remote control
- „nicer icons”
- to connect a barcode scanner

Too difficult to implement (within a reasonable time).

**Out of the box in Linux** (+userspace) - but will it fit?
What Real-Time really means?

Popular definition:

Correctness of operation depends not only on whether performed without error, but also on the time (the upper limit) in which the operation completed.
Practical definition:

**RT system is one in which can be proved that any required operation will be completed in a certain time.**

- Mathematical proof would be perfect. Unfortunately systems are so complex, it is not possible.
- System is tested (TDD). If deadlines are met (under load) for all use-cases, system is Real-Time.

Note: In some cases (eg. certification for safety-critical tasks), full-code coverage would be needed!
Linux kernel is designed to be „democratic”
- resources are equally disposed
- eg.: scheduler avoids process starvation

Usually, determinism is not taken into account

throughput is.
Most layers and subsystems are complex:
Latency tests
What are we testing?
Input is triggered on falling and rising edge, output state changes according to input.

In this case, we are using GPIO pin-s. Input can be other external or internal: timer, camera, network PHY, ADC etc.
Driver design

01_inout.c

02_uinout.c

Code is on GitHub:
https://github.com/marcinbis/mb-rt-data.git
Let’s add some load

$ cat /proc/loadavg
5.02 3.76 2.04 2/47 432

- I/O on SD card:
  cat /dev/mmcblk0p1 > /dev/null
- sending ASCII data to serial console:
  cat /dev/zero | od -v
- send network packages:
  ping -f <ip address>

**WARNING!**: these tests just generate IRQ, they are not showing real-life load.
Use real-case tests.
Results under load
Linux:

Separates:

- logic (in userspace)
- mechanisms (provided by kernel)
Interrupt-based I/O

- registered on module loading
- unregistered on module unloading

file_operations {
  .open
  .release
  .read
  /* ... */
}

read() waits for request on waitqueue
returns data and wakes process

(optional)
bottom half
defered data handling
triggers processes on waitqueue

IRQ registered on module loading (request_irq()) and associated with handler
IRQ freed on module unloading

irqreturn_t (*handler)
IRQ handler
receives data from device
schedules bottom halve

IRQ line on current CPU is locked
In case of GPIO: `/sys/class/gpio/` can be used as well (`poll()`, `read()`, `write()`).
Userspace - results under load
Userspace - results under load
Real-Time
**Deadline**

Point in time, before which the action (system response) must occur.

- **Hard Real-Time** - deadline must be meet (fatal error if not).
- **Firm Real-Time** - deadline should be meet (system response is useless otherwise).
- **Soft Real-Time** - deadline should be meet, but nothing critical will happen if not (eg. decreased user experience, sample drop ...).

---

**Latency**

The time between the moment in which the action was to occur, and in which, in fact, occurred.
Jitter
Undesired deviation of latency. For various reasons, latency is not constant. Too large jitter, renders system unusable for data acquisition.

Predictability
How much time, the action will take (eg. from IRQ occurred to handler finished executing). O(1) algorithms should be used.

Worst Case
Due to imperfect nature of real-world systems, we are considering the Worst Case.

We have to know the latency in worst possible case.
Where is latency coming from?

![Diagram showing the stages of task processing and latency](image-url)
How to achieve Real-Time in Linux?

1. Micro-kernel approach:
   - Xenomai - http://www.xenomai.org/ - provides various APIs.

2. In-kernel approach:
   - RT PREEMPT - https://rt.wiki.kernel.org
     http://www.kernel.org/pub/linux/kernel/projects/rt/
I-Pipe take control over all hardware interrupts
All system calls are passed through (I-Pipe)
Events are dispatched to different I-pipe domains.
$ cat /proc/ipipe/Xenomai

|+----- Sticky
||+---- Locked
|||+- Exclusive
||||+- Virtual

[IRQ]  ||||
   38: W..X.
   418: W...V

[Domain info]
id=0x58454e4f
priority=topmost

$ cat /proc/ipipe/Linux

  0: A....
  1: A....
...
priority=100
There are actually two kernels:

- Xenomai
- Linux

process (its scheduling) can migrate between them.
RTAI use-case: LinuxCNC

- RTAI demonstration running on PC
- Ubuntu (LiveCD) + RTAI + Applications
  - GUI
  - G language
  - Process visualisation
  - Software PLC (not only for learning)
  - Drivers for certain hardware . . .
  - . . . or timer-based stepping

Let this test run for a few minutes, then note the maximum Jitter. You will use it while configuring emc2.

While the test is running, you should "abuse" the computer. Move windows around on the screen. Surf the web. Copy some large files around on the disk. Play some music. Run an OpenGL program such as glxgears. The idea is to put the PC through its paces while the latency test checks to see what the worst case numbers are.

<table>
<thead>
<tr>
<th>Max Interval (ns)</th>
<th>Max Jitter (ns)</th>
<th>Last interval (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Servo thread (1.0ms):</td>
<td>994484</td>
<td>3204</td>
</tr>
<tr>
<td>Base thread (25.0μs):</td>
<td>31018</td>
<td>6773</td>
</tr>
</tbody>
</table>

Reset Statistics
LinuxCNC
RT PREEMPT
1. Standard kernel

Interrupts as threads

Kernel Features --->
Preemption Mode (Complete Preemption ()) --->
(X) Complete Preemption (Real-Time)

-** Thread Softirqs /* 2.6.33 */
-** Thread Hardirqqs
04 under load
04 under load
Real-Time != Real Fast
Maximum latency (Worst Case) is limited, but minimum latency is bigger.

Kernel with RT-PREEMPT patch, does not make the whole system Real-Time

Specially designed application and POSIX RT-API should be used:
- Defined: IEEE 1003.1b. Linux supports it.
- Scheduler
- Memory locking
- Shared memory
- RT signals
- Semaphores (priority inheritance)
- Timers (esp. CLOCK_MONOTONIC)
- AIO
ICS

Industrial control system
Loose material (or fluid) is loaded into containers.

- The main tank is suspended on weight (tensometer).
- Conveyor or robot provides containers, appearance of the container triggers interrupt.
- The valve opens, and the material is poured into a container.
- Amount of material is measured by reading data from the weight.
- Main tank has a limited capacity, it can be replenished from main silo by turning on vacuum.
- If vacuum is turned on, it has to work for some minimum time, while vacuum is working, material cannot be poured.
Weight-dosing process can be modelled as a finite state machine.

Process starts on \texttt{WAIT\_ON\_TRIGGER} state. If triggered, it runs on timer (1ms).
System boot constraint: must be operational under 10s.
Results: 2.5s RT task, 8-9s GUI.
Weight-dosing process - implementation

Hardware
- PC for development, Debian GNU/Linux 6
- Custom AT91SAM9263 board for production, 2.6.33.7.2-rt30
  http://www.osadl.org/ Latest Stable
- KT-SBC-SAM9-1, BeagleBoard-xM for testing, 2.6.33 and 3.0

GUI part
- written in QT/C++
- userspace components provided by: Buildroot
- ext4 on SD card as primary storage
- optimized bootlader

Real-Time process (PREEMPT-RT)
- implemented as separate process in C
- communicates with GUI using POSIX shared memory and message queue, in a lockless way:
  - two control structures are stored in SHM
  - one is utilized by running process, other can be changed by GUI
  - then structs are switched
Other cases...

Weight dosing - pneumatics are used (actuator latency is 15ms).

![Diagram: main() ↔ worker()](image)

Threads - share virtual memory, have different scheduling settings.

Welding machine - μs
Medical laser controller - μs (or even less)

![Diagram: main() (Linux task) ↔ worker() (Xenomai task)](image)

Xenomai provides better latency and predictability.

![Diagram: main() (Linux task) ↔ fast serial connection ↔ worker() (dedicated μC)](image)

Special hardware can be utilized too:
- two processors: eg. additional Cortex-M for running worker task
- multicore systems: eg. Freescale Vybrid (Cortex-A5 + Cortex-M4)
Use appropriate programming language

- **C** - but make it object-oriented (for reference - Linux kernel: buses, drivers, classes etc.)
- **C++** - would be nice too
  - cannot be utilized inside kernel or as Xenomai kernel process
  - can be executed as bare-metal μC or in userspace
- Utilize design patterns.

Set the proper scheduler class and priority

```c
struct sched_param sp;
sp.sched_priority = MY_PRIORITY;
ret = sched_setscheduler(0, SCHED_FIFO, &sp);
```

- Interrupts run in threads, default to: SCHED_FIFO/50.
- ...do not forget to fine-tune them.
- SCHED_DEADLINE can be helpful too.
Lock all memory (mlock)

mlockall(MCL_CURRENT|MCL_FUTURE);

Try to cause page_fault (allocated memory, data from files)

buf = malloc(BUF_SIZE);
memset(buf, 0, BUF_SIZE);

- Memory is locked, so it stays on place.
Prefault the stack (it can be shared within process we have forked from)

```c
/* GCC will not inline this function */
__attribute__((noinline)) void stack_prefault(void)
{
    unsigned char tab[MAX_SAFE_STACK];
    /* GCC will omit optimizations */
    asm(""");
    memset(tab, 0, MAX_SAFE_STACK);
}
/*...*/
stack_prefault();
```
Use POSIX timer to do the fsm step (in a proper way)

```c
#define NSEC_IN_SEC 1000000000L
#define INTERVAL 1000000L
struct timespec timeout;

clock_gettime(CLOCK_MONOTONIC, &timeout);
while (1) {
    do_fsm_step(&some_data);
    timeout.tv_nsec += INTERVAL;
    if (timeout.tv_nsec >= NSEC_IN_SEC) {
        timeout.tv_nsec -= NSEC_IN_SEC;
        timeout.tv_sec++;
    }
    clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME,
                     &timeout, NULL);
}
```
clock_nanosleep(CLOCK_MONOTONIC, 0, &interval, NULL);

clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &timeout, NULL);
Utilize AIO to write or read data (eg. sensor data, production logs)

```c
struct aiocb {
    int aio_fildes // File descriptor.
    volatile void *aio_buf // Location of buffer.
    /* ... */
};
aio_write(struct aiocb *);
aio_return(struct aiocb *);
```
ZATRZYMAMIE AWARYJNE... means 'Emergency Stop' in Polish.