Introduction à la programmation temps-réel pour la robotique
Robotics Real-Time Programming

Ludovic Saint-Bauzel

Polytech’Paris UPMC

2012-2013
Introduction

Robotics Software Frameworks

OS : Tasks And Threads

OS Driver Programming

Application

Documentations
Introduction

What is a robot?

Definition

Hardware

Software

First Idea: From Scratch

Second Idea: Operating System Approach

Robotics Software Frameworks

OS: Tasks And Threads

OS Driver Programming

Application

Documentations
Definition: An Autonomous Robot is a complex system that has the ability to decide its actions on its environment from its sensing, state in order to fill his aims. Survive, Assist ...

- Sensors
- Decision
- Action

Physical ability: read information from sensor, send orders to actuators, have a regulation (Time based computation) and a decision computation unit (policy).

Figure: Classic (up) and Autonomous (down) Robot Models
A computer science way to see a robot is to say it is a computer with peripherals that bring sensing, actuation ability and because of real life interaction need to put the stress on the time. So like a computer we will have to deal with:

- bus
- memory
- cpu
Introduction

What is a robot?

Ludovic Saint-Bauzel (Polytech'Paris UPMC)
Structure Choice criteria

- Critical path?
  - the main loop
  - delayed sensors

- Method
  - Tasks
  - Pipes (read only, read write, write)

- Parameters
  - Computation
  - Time constraints
  - Memory needs
  - Precision of measure
Monocomputer

- Properties
  - Limited Computation Ability
  - Delays are limited too
  - Simple but efficient
- Recent Improvements
  - GPU
  - Multicore CPU
- Example

Figure: mc2e
Monocomputer

- Properties
  - Limited Computation Ability
  - Delays are limited too
  - Simple but efficient

- Recent Improvements
  - GPU
  - Multicore CPU

- Example

**Figure:** mc2e Architecture
Multi computer

- **Properties**
  - High scalability of computation
  - High time lost: Bus communication
- **Recent Improvement**
  - Distributed algorithms development
- **Example**

Staubli
Multi computer

- Properties
  - High scalability of computation
  - High time lost: Bus communication
- Recent Improvement
  - Distributed algorithms development
- Example

![Diagram of multi computer system]

UDP

Robot
Sensors
Effectors
Control
Supervision / Decision
Staubli

Robot Models
- Kinematics
- Velocities
- Trajectory Control

GUI To choose trajectories
Introduction

What is a robot?

Master Computer - Slave Computing Units

- Master is connected to slave unit
- Often a bus
- Image
**Bus**

- **Definition**
  - Physical connection
  - Multiple devices

- **Common elements:**
  - Base Address called ports
  - Local Address

- **Examples:**
  - Serial
    - RS485
    - USB
    - CAN
  - Parallel
    - ISA
    - PCI
    - ...

Ludovic Saint-Bauzel (Polytech’Paris UPMC) Robotics Real-Time Programming 2012-2013 14 / 334
How the robot can make environment interaction?

- Categories
  - Perception
  - Decision
  - Actions

- Computer Science Name: tasks
Introduction

What is a robot?

Real-Time Programming

- Low Level Access
- High Time constraints
  - Don’t mean quick!
  - Mean ability to respect those constraints
Low Level Programming

LINUX

```c
#include <asm/io.h>
...
unsigned char inb (unsigned short port);
void outb (unsigned char byte, unsigned short port);
unsigned char insb (unsigned short port, void *addr,
                      unsigned long count);
void outsb (unsigned short port, void *addr,
                      unsigned long count);
```

DOS

```c
#include <dos.h>
...
unsigned char inputb (unsigned short port);
void outputb (unsigned short port, unsigned char byte);
```
Introduction

What is a robot?
First Idea: From Scratch
Example
x86 example
From scratch approach
Second Idea: Operating System Approach

Robotics Software Frameworks

OS: Tasks And Threads

OS Driver Programming

Application

Documentations
RobModex

- ISA Card
  - 0x100 : Base Address
  - 0x01 - 0x04 : Timer (IRQ 5)
  - 0x05 - 0x08 : DAC1
  - 0x09 - 0x12 : DAC2
  - 0x13 - 0x14 : 2 Wheel Encoder Registers
Closed Loop PID

variables \( q_{mes}, q_{des} \)
\[ \text{error}(k) = \]
\[ \text{error\_cumul}(k) = \]
\[ \text{error\_diff}(k) = \]

\[ q_{cons} = \]
\[ \text{olderror} = \text{error} \]
Simple IRQ x86

<table>
<thead>
<tr>
<th>IRQ</th>
<th>IRR</th>
<th>IMR</th>
<th>ISR</th>
<th>IVT</th>
</tr>
</thead>
<tbody>
<tr>
<td>#0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>#1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>#2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>#3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Addr Interrupt Routine
...
...
...
Simple IRQ x86

<table>
<thead>
<tr>
<th>IRQ</th>
<th>IRR</th>
<th>IMR</th>
<th>ISR</th>
</tr>
</thead>
<tbody>
<tr>
<td>#0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>#1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>#2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>#3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Memory (0x0-0x3FF)

Addr Interrupt Routine

...
Simple IRQ x86

PIC Registers

<table>
<thead>
<tr>
<th>IRQ</th>
<th>IRR</th>
</tr>
</thead>
<tbody>
<tr>
<td>#0</td>
<td>0</td>
</tr>
<tr>
<td>#1</td>
<td>0</td>
</tr>
<tr>
<td>#2</td>
<td>1</td>
</tr>
<tr>
<td>#3</td>
<td>0</td>
</tr>
</tbody>
</table>

IMR

<table>
<thead>
<tr>
<th>IMR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

ISR

<table>
<thead>
<tr>
<th>ISR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

IVT

Addr Interrupt Routine

...
## Simple IRQ x86

### PIC Registers

<table>
<thead>
<tr>
<th>IRQ</th>
<th>IRR</th>
<th>IMR</th>
<th>ISR</th>
</tr>
</thead>
<tbody>
<tr>
<td>#0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>#1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>#2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>#3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

### Memory (0x0-0x3FF)

<table>
<thead>
<tr>
<th>IVT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addr Interrupt Routine</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

### PIC Registers Memory (0x0-0x3FF)

- 0x20
  - Command Register
  - Status Register
- 0x21
  - Int Mask Register
  - In Service Register

### System Memory

- Interrupt Vector Table (4 bytes)
  - Byte 0 : Offset Low Address of the **Interrupt Routine** (Handler)
  - Byte 1 : Offset High Address of the IR
  - Byte 2 : Segment Low Address of the IR Offset
  - Byte 3 : Segment High Address of the IR Offset

---

Ludovic Saint-Bauzel (Polytech’Paris UPMC) Robotics Real-Time Programming 2012-2013 22 / 334
**turboC DOS example**

```c
# define IMR 0x21
void _install_int_function(int IRQn,
    void interrupt (*_new_int_function)())
{
    int inter = IRQn + 8;
    _disable(); // disable interrupts
    // save the old interrupt vector
    _old_int_function=_dos_getvect(inter);
    // install the new interrupt vector
    _dos_setvect(inter,_new_int_function);
    // save the state of the 8259A IMR register
    _old_mask=inportb(IMR);
    // Set new value for IMR register
    outportb(IMR,_old_mask&_interrupt_mask(IRQn));
    _enable(); // enable interrupts
}
```
Introduction  First Idea : From Scratch

I/O APIC : Advanced Programmable Interrupt Controller

Pentium 4 and Intel Xeon Processors
Processor Core
Local APIC
Interrupt Messages → Local Interrupts
Interrupt Messages ← System Bus
Bridge
PCI
I/O APIC
System Chip Set

Pentium and P6 Family Processors
Processor Core
Local APIC
Interrupt Messages → Local Interrupts
External Interrupts
I/O APIC
System Chip Set
3-Wire APIC Bus
IRQ

Timer

Timer Mode
00: One-shot
01: Periodic
10: TSC-Deadline

Address: FEE0 0320H
Value after Reset: 0001 0000H

Delivery Status
0: Idle
1: Send Pending

Delivery Mode
000: Fixed
010: SMI
100: NMI
111: ExdINT
101: INIT
All other combinations are Reserved

Interrupt Input Pin Polarity

Remote IRR

Trigger Mode
0: Edge
1: Level

LINT0

LINT1

Error

Performance

Vector

Vector

Vector

Vector

Vector

Vector

Vector

Vector

Vector
Bus: ISA

- Ax: Addresses
- Dx: Data
- IRQs: 3, 4, 5, 6, 7
- AEN: DMA
#define ISAADDR
short data;
void interrupt _new_int_function()
{
    _disable(); //disable interrupts
data = inb(ISAADDR); //getDataFromISA
    //that the ISA Component has requested
    //to tell the system the interrupt service function has finished
    _end_interrupt();
    _enable(); //enable interrupts again
}
Digital/Analog Converter

**Figure:** Digital Analog Converter Schematic
RobModex Application

```c
#define ISAADDR ..... 
short data;

void interrupt _new_int_function()
{
_disable(); // disable interrupts
data=inb();
}

outb(),
_end_interrupt();
-enable(); // enable interrupts again
}
```
Conclusion

- More Complex = More code
- No-reusability of pieces of code

Figure: RUPI
Introduction

What is a robot?

First Idea: From Scratch

Second Idea: Operating System Approach

Introduction

Properties

Simple

Multitasks

Real-Time

Robotics Software Frameworks

OS: Tasks And Threads

OS Driver Programming

Application
Operating System few words

A software:
or precisely the first software executed by your computer

Consequences:
Manager

► Schedule task order
► Manage resources in order to fulfil every needs
► Use of all the resources.
Resources

Computer Science:
- Processor: CPU, IRQ
- Communication: BUS
- Memory
- ...

Robotics:
- Motor
- Sensors
What is RealTime

Two examples:

A navigation system computing the best path for a boat.

A navigation system computing a boat position during its sailing.

- First case: Computing time is not a constraint, the result is this only thing that is interesting
- Second case, if computing time is too long, position is false.
An informal definition

**Real-Time system**: A system where the behaviour relies on precision of computation and also the time when it is produced.

In other words, a delay is considered as an error that can lead to severe consequences.
A Robotic definition

A Real-Time System: a system that bring possibility to control a robot in a finite time and let us manage urgent informations in a high level of priority. That means a real time system for control and management of events with priority possibilities.
Deadline and realtime system

Different kind of deadline

- **Hard Deadline**: a delay generate an exception (error event and computation)
- **Soft Deadline**: a delay don’t lead to an exception

Different possible real-time systems

- **Hard Real-Time systems**: exceeded deadlines must not arise. (VxWorks, RT-Linux, RTAI, Xenomai...)
- **Soft Real-Time**: exceeded deadlines can sometimes happen. (DOS, Microsoft Robotic Studio?)
Environment and RealTime System

**System** : set of activities linked to one or more computing tasks that are done sequentially or concurrently and that can communicate each other.

A Real-Time System interacts in its environment

- **Sensors** : signals and their measurement tools
- **Computing Unit**
- **Actuators** : act on the environment at precise times
Approaches

Centralised
- A CPU work on information coming from sensors and send the result to actuators

Multiple-systems
- Controlled systems and a controlling system interacting with sensors and actuators
Time scheduling

- time-driven system: measure of time leads to actions
- event-driven system: arising of an event leads to actions
- reactive system: constrained time to process
Predictability determinism reliability

Real time system : in worst case.

**Predictability** : Ability to identify in advance if a system is able to respect its time constraints Knowledge of parameters related to computation.

**Determinism** : remove any uncertainty in individual task behaviour and when they are together
- Variation of task execution time
- IO duration, reaction time
- Interruption reaction

**Reliability** : Behaviour and Fault tolerance
Example GSM Mobile

System must combine:

- Hardware layer: emission, reception, voice coder activation, reception strength measure, etc
- Software layer: DB communication for localisation update, look up calling messages, etc...

Real-Time constraints: 577 $\mu$s of talking emitted and reception every 4.6 ms

Mobility constraints: emits with an earlier signal if distance increase and later if it decreases.

System must be enough reactive in order to the system comfortable for the user.
Classical system limits

Classical systems are based on multiple tasks not well adapted to real-time constraints:

- scheduling policy that aims to share time in a balance way for each task
- access mechanisms that shares resources and synchronizes time uncertainty
- interruption management is not optimal
- Virtual memory and cache memory management generate some delays
- timer management is not very accurate
Services

Let’s consider a simple system :

- a single software
- One CPU
- One memory
- One IO controller
- One communication bus

Software interacts with external world

- Data acquisition
- Computation
- Returning the results

How can a software react to an event that can arise at any time?
Cyclic look up

Look up regularly each peripheral devices: called interacting from a cyclic look up

```
  do
  do
    check sensors
  while (no data are available)
  read sensors
  compute data
  do
    activate actuators
  while (action to do)
  while (system halts)
```
Introduction  Second Idea : Operating System Approach

Cyclic look up

Pros

▶ predictable : one iteration time (easy to know)
▶ easy to program : with a few peripheral devices while reaction time of each device are almost the same

Cons

▶ if there is many devices and reaction time are very different, sub-optimal (near the slower device), sub loop to treat each device
▶ bad in a computer science point of view : software become unreadable, close to be impossible to maintain : a new device needs to rewrite almost all the software.

Ludovic Saint-Bauzel (Polytech’Paris UPMC)  Robotics Real-Time Programming  2012-2013  46 / 334
Interruptions

How to warn the software that something happens outside? mechanism to stop the regular computation is called **interruption**

- Initially hardware event leads to some changing in software computation
- Next, extended to internal events and events coming from software
- In general point of view, an event modify the behaviour of your software

Deal with interruptions

- Save context
- Interruption is computed
- Restore context
Interruptions

```plaintext
do
do
check sensors
while (WA_DATA)
read memorised data
while (system halts)

INTERUPTION //from a sensor
read data
write WA_DATA is true
acknowledge interruption
```

- Interruption deals with external events
- Cyclic look up software is delayed
- Good response time if interruption computation needs a few time
Limits

CPU can spend time to compute different things than the software

Example: UART (Universal Asynchronous Receiver Transmitter)

- Interruptions to emit and receive through two registers (parallel computation ability)
- Emission of a ASCII Char every millisecond: a lot time to wait until we have to emit again
- For a new emission: WAIT
- When UART is available: Interruption SIGNAL
- Software can restart

Can do new things while waiting: **multiple-task**
Introduction

Second Idea : Operating System Approach

Synchronous or asynchronous approaches

- need a system that manage the software : start and synchronizes tasks according to events and context evolution
- 2 kind of approach :
  - **synchronous one** : direct link between a task and its activating device, without taking care of conflicts that can arise between this task and the others during execution of the software.
  - **asynchronous one** : all events are considered, including during computation of a task and reacts according to the context.
Synchronous approach

- "simple" algorithm
- does not work without behavioural and time error emission
  - No pre-emption neither concurrent access to share resources
  - "synchronous" hypothesis: task activating time is NULL (instantaneous reaction)
- discrete time view, scheduled by events arising, in the opposite of continuous time view (hard time)
- simultaneity linked to observation of current events at the same time
- simple and powerful model
Asynchronous approach

- more complex algorithms to implement
- hard to model for checking the behaviour, need to evaluate every possible pre-emption
- scheduling policy: event appearance, CPU load decision between:
  - activated task by event
  - running task when event arise
  - another task waiting for the CPU
- sometimes it is possible to check off-line according to formal rules
- priority definition is based on some criteria like: deadline, period, level of importance
Services of a multitasks system

Conclusion: A single program rarely use all CPU load
Used those idle times to compute some concurrent other tasks from other software: a part of what an operating system must do

» running concurrent tasks
» exchanging with outside world
» managing memory resources
» sharing hardware, synchronization, communication
» time management

What are the limits for real-time system?
Standard operating systems manage process that can be described as follow:

- separated memory room
- special channels for communication
- high time cost spent for creation of a new one
- protection mechanisms are also time costing
- switch between two processes need a lot of memory switches
CPU Management - Task, process

One process works on many memory areas

- computing area (instructions of the software)
- a heap data area
- a stack area for temporary data (variable)

**Context of a process** is composed of registers, a name, a state, etc.

A **thread** only own individual context data, everything else is shared with father process
Management multiple task and scheduling

A activated process can have many states, here are the main one :

- **created**
- **terminated**
- **running** : task is currently using the CPU. We say it is **elected**.
- **waiting** : task is asking for CPU. It is **electable**.
- **blocked** : task is blocked waiting for an event to arise.
Management multiple task and scheduling

State transitions are:

- **waking**: blocked $\rightarrow$ waiting
- **allocating**: waiting $\rightarrow$ running
- **unallocating** or **pre-empted**: running $\rightarrow$ waiting
- **blocking**: running $\rightarrow$ blocked
Scheduling

Scheduler must manage how the task are allocated to the CPU. Schedule in the best way is very hard.

A scheduler called with request or pre-emptive if running process can be unallocated by scheduler.

Non pre-emptive systems can do context switch only when a task is terminated or when it is asking for context switch.

Scheduler works most of the time on-line. That means that a policy must exists and be applied.
Scheduling policy

Classical scheduling rules:

- First In First Out (FIFO)
- Each one its turn (tourniquet, *Round Robin*: RR)
- Priority based

Example

<table>
<thead>
<tr>
<th>Task</th>
<th>Length</th>
<th>Priority</th>
<th>Start time</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>6</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>T2</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>T3</td>
<td>4</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Scheduling example 1

- **FIFO**
  
  T1  T1  T1  T1  T1  T1  T2  T2  T2  T3  T3  T3  T3

- **RR**
  
  T1  T2  T3  T1  T2  T3  T1  T2  T3  T1  T3  T1  T1

- **Priority**
  
  T2  T2  T2  T1  T1  T1  T1  T1  T1  T3  T3  T3  T3
Scheduling example 1

Scheduler with task interactions

<table>
<thead>
<tr>
<th>Task 1 (prio = 2)</th>
<th>Task 2 (prio = 2)</th>
<th>Task 3 (prio = 3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T[1.1] wait (event)</td>
<td>T[2.1] Signal (Task 1, event)</td>
<td>(not started) T[3.1] wait (1 sec.)</td>
</tr>
<tr>
<td>T[1.2]</td>
<td>T[2.2] Starting Task 3</td>
<td>T[3.2]</td>
</tr>
</tbody>
</table>
## Scheduling example 2

### Pre-emption after OS signal

<table>
<thead>
<tr>
<th>Task 1 (prio = 2)</th>
<th>Task 2 (prio = 2)</th>
<th>Task 3 (prio = 3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T[1.1] Wait (evt)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T[2.1] Signal T1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T[1.2]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>T[3.2]</td>
</tr>
</tbody>
</table>
## Scheduling example 2

### Without pre-emption

<table>
<thead>
<tr>
<th>Task 1 (prio = 2)</th>
<th>Task 2 (prio = 2)</th>
<th>Task 3 (prio = 3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T[1.1] Wait (evt)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>T[2.1] Signal T1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>T[2.2] Start T3</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>T[3.2]</td>
</tr>
</tbody>
</table>
Interruptions

- 2 kind
  - **asynchronous**: emitted by external context (hardware) can arise at any time
  - **synchronous**: emitted by a specific instruction can only arrive in known date for the developer

- processed by the same mechanism
Introduction

Second Idea: Operating System Approach

Interruption management

The way interruptions are processed have a large influence on efficiency of the OS

- Control of interruptions: mostly hardware
  - Context switch
  - Source identification

- Processing interruption
  - Short: same context
  - Long: task are called: signal/wait

- Back to original process
Those mechanisms are used to give one access to one resource in one time.

- Define some sections as **atomic**: **critical sections**
- Mask interruptions (hardware)
- Scheduler is disabled during this critical section (software)
  - Mutex
    - Block other tasks whatever priority they are
    - Problem to choose the task when Mutex is released
- Counting Semaphores to manage a set of devices
Synchronization

Force an order of processing instructions of a process

- Mutex
- Rendezvous
- Product/Consume

Uses the following mechanisms:

- Signal an event
- Semaphores use
- Mailboxes
- Rendezvous
Synchronization Limits

Maintain coherency of data: creation of waiting queues

- "Balanced" Systems inefficient
- Priority: Synchronization brings some priority issues where a low priority can block a high priority task

Some solutions:

- Priority based waiting management: unlock view the priority level of the task
- Flag like "availability test"
- Time out on blocking flags

Ludovic Saint-Bauzel (Polytech’Paris UPMC)
Input/Output

Different policies for CPU scheduling and for Input/Output (IO)

- Access through a specialized layer
- Synchronous processing
- Asynchronous processing
- Difficult to manage priorities
- Bad time management
Definition: a software that schedules computing of tasks in one shot way, manage devices of the system and bring development libraries (Application Programming Interface or API).

It is organized with:

- a kernel that schedules tasks and manage devices
- often it proposes services packaged as modules for example. These service can bring the ability of the system to manage File Systems, network devices or any services that are needed by software.
Components

The main components of a RTOS are:

- scheduler, inside every kernel, it is the component that apply different algorithms to manage access to the CPU
- kernel interfaces that give developers the ability to create software
  - tasks
  - semaphores
  - message queue...
- services: actions that a kernel can do on a device or on the system like time measurement, interruption, bus controller,...
Introduction

Robotics Software Frameworks
RUPI
RUPI Point Of View
Design Pattern
OROCOS
ROS
Frameworks

OS : Tasks And Threads

OS Driver Programming

Application

Documentations
Challenging Scenario

Figure: RUPI
Layout

Introduction

Robotics Software Frameworks
  RUPI
  Design Pattern
    Software Recipes
  OROCOS
  ROS
  Frameworks

OS : Tasks And Threads

OS Driver Programming

Application

Documentations
Find the tasks!

- No Fuzzy properties
  - Real-Time
  - Good Refresh rate
- Nothing is perfect
  - Properties
    - Time (Loop (Freq?), Oneshot)
    - Load (CPU)
  - Boundaries
    - Delays
    - Memory
- Task repeated every milliseconds with acceptable delay of 0.01 ms
Classical Robotics Tasks

- **Control Task**
  - Loop, Delay
  - Variable Parameters
  - Static parameters

- **User Interface**
  - Supervision
  - Send Orders (Asynchronous, Parameters)
MVC

- View
  - Supervision
  - GUI
- Controller
  - Send Orders
  - Control Task Information
  - GUI
- Model
  - Synchronization
  - Communication System
  - Database
- RT Task
Introduction

Robotics Software Frameworks
  RUP1
  Design Pattern
  OROCOS
    Open RObotics COntrol Software
  ROS
  Frameworks

OS : Tasks And Threads

OS Driver Programming

Application

Documentations
OROCOS Project

Figure: Orocos Project

Kinematics and Dynamics Library
- Kinematic chains, real-time inverse and forward kinematics, Python bindings

Bayesian Filtering Library
- Dynamic Bayesian Networks, (Extended) Kalman Filters, Particle Filters (or Sequential Monte Carlo methods)

Orocos Toolchain
- Real-time software components, interactive scripting, state machines, distributed processes, code generation
History

- December 2000, as an idea of Herman Bruyninckx
- European Funding
- European Labs
  - K.U.Leuven in Belgium (Orocos@KUL)
  - LAAS Toulouse in France (Orocos@LAAS)
  - KTH Stockholm in Sweden (Orocos@KTH)
Orocos Toolchain = RTT + OCL

- Real-Time Toolkit
- Orocos Component Library

Figure: Orocos Component/Plugins Architecture
Task Context

- Provides
- Requires
- Plugin
- Flow Port

Operations

OperationCallers

Services Interface

maps to

C/C++ functions:
- Callbacks
- Algorithms

Dynamic functions:
- State Charts
- Program scripts

Configuration Interface

Input Ports

Data Flow

Output Ports

Data Flow
OCL -> Logging Tools

- 2 Solutions
  - Custom Component
    - + Complete control of the generated file
    - - Some additional code that is not the heart of the controller
  - Listen a component through logging tools
    - In C++ code:
    - In deployment:
Task State Diagram

```
public API
user code

cleanup()
cleanupHook()

Default
state

Stopped

Init

PreOperational

configure()
configureHook()

Running

start()
startHook()

update()
updateHook()
```
Activity vs Threads

An Activity object executes the ExecutionEngine, which in turn processes incoming messages, plugin functions and finally updateHook() is called.

Figure: Component Activity Diagram
Data Flow

Figure: Components Connections
Example: Definition

- **Supervisor**
  - Initialisation
  - Waiting
  - Standing-Up
  - Walking
  - Alert

- **Sensors**
  - Force Handles
  - Orientation of the arm
  - Orientations of the wheels ($\beta_L, \theta_L, \beta_R, \theta_R$)

- **Controls**
  - Impedance Control
  - Position Control
  - Speed Control
  - Models
Example: Modelling State Machine
Example: Modelling Data Flows
Example: State Machine (Excerpt)

```plaintext
StateMachine OurStateMachineType
{
   var bool detected;
   initial state INIT {
      entry {
         ConsoleOut.display("[INIT]");
         /*...*/ Plant.configure();
      }
      transition select Waiting;
   }
   state Waiting {
      entry {
         STSDetector.start();
      }
      transition STSDetected(dected) if (detected) then select SitToStand;
   } /*...*/
RootMachine OurStateMachineType statemachine
```
Example: Component Skeleton (Excerpt)

class Supervisor : public TaskContext {
protected:
    InputPort<bool> STSDetected; /*...*/
    string stateMachineName;
public:
    Supervisor(string const & name) : TaskContext(name),
        stateMachineName("statemachine")
    {
        this->addEventPort("STSDetected", STSDetected);
        this->addProperty("statemachineName", stateMachineName);
    }

    bool configureHook() {
        return true;
    }
    void updateHook() { } /*...*/
    void errorHook() { }
    void stopHook() { }
    void cleanupHook() { }
};
Robotics Software Frameworks

Example: Deployment File (Excerpt)

```xml
<properties>
  <struct name="Supervisor" type="Supervisor">
    <struct name="Activity" type="Activity">
      <simple name="Period" type="double"><value>0.1</value></simple>
      <simple name="Priority" type="short"><value>50</value></simple>
      <simple name="Scheduler" type="string"><value>ORO_SCHED_RT</value></simple>
    </struct>
    <struct name="Peers" type="PropertyBag">
      <simple type="string"><value>STSDetector</value></simple>
      <!--... [EVERY PEERS that will be touched by this component] ...-->
    </struct>
    <struct name="Ports" type="PropertyBag">
      <simple name="STSDetected" type="string"><value>STSDetected</value></simple>
    </struct>
    <simple name="RunScript" type="string"><value>deployment/smSTSandWALK.osd</value></simple>
  </struct>
</properties>
```

Ludovic Saint-Bauzel (Polytech’Paris UPMC)
Layout

Introduction

Robotics Software Frameworks
- RUPI
- Design Pattern
- OROCOS
- ROS
  - Robotics Operating System

Frameworks

OS: Tasks And Threads

OS Driver Programming

Application

Documentations

Ludovic Saint-Bauzel (Polytech'Paris UPMC)
ROS (Robot Operating System) provides libraries and tools to help software developers create robot applications. It provides hardware abstraction, device drivers, libraries, visualizers, message-passing, package management, and more.

Willow Garage
- Created in 2006
- 2008 first stable release of ROS
- 2010 ROS 1.0
Connecting Middleware

- Nodes
  - Server (roscore)
  - Output (rosout)
- Topics
  - Publishers
  - Subscribers
- Parameterized Connections
  - TCP
  - SharedMemory
  - Serial (Arduino RosSerial)

*Figure:* rxgraph of /Example node with its topics
Meta-Operating System

- Meta-Operating-System
  - rosdep
  - rosmake
  - roscd
  - rosed
  - rosrunc
- Usefull tools
  - rxgraph
  - rxplot

Figure: rxplot of /Example node with its topics
<package>
  <description brief="depth_reg">
    depth_reg
  </description>
  <author>bonjean</author>
  <license>BSD</license>
  <review status="unreviewed" notes=""/>
  <url>http://ros.org/wiki/depth_reg</url>
  <depend package="cv_bridge"/>
  <depend package="roscpp"/>
  <depend package="sensor_msgs"/>
</package>
Introduction

Robotics Software Frameworks
- RUPI
- Design Pattern
- OROCOS
- ROS

Frameworks
- Conclusion

OS: Tasks And Threads

OS Driver Programming

Application

Documentations

Ludovic Saint-Bauzel (Polytech’Paris UPMC)
Connectivity Question

- Exercice :
- Description
  - Kinect
  - OpenCV
  - SLAM
  - Directions
- Aim
  - When someone do hello then go to its direction
Example

- **Exercice**
  - **Description**
    - Kinect
    - OpenCV
    - SLAM
    - Directions
  - **Aim**
    - When someone do hello then go to its direction
- **Result**
  - Kinect -> OpenCV -> Position -> SLAM -> Directions
Connectivity Solutions

- In OROCOS:
  - Omni ORB
  - From Scratch Components Connection
    - RPC
    - CORBA
    - Sockets -> Server/Client Approach

- ROS
Conclusion

- Overview Implementation
  - Controller on one Computer: OROCOS
  - Communication between computing units: ROS

- Method
  - Identify the states of your system (Tasks, loops)
  - Think of the time, delays and how to overlay some of them
  - Write the data flow for each state
  - Look at the Computing Units: At least one node per Unit (ROS)
  - Therefore, the components are described (inside one node)
Layout

Introduction

Robotics Software Frameworks

OS : Tasks And Threads
Scheduling
  Model
  Algorithms
  Thread Scheduling
Management
Time Services
Synchronization
Messages and Communication
IO Control

OS Driver Programming
Scheduler

**Aim**: address any needs of a real-time software like emergency stop, high level action or reactivity need

**Taxonomy**:
- Off-line/On-line algorithm: static/dynamic algorithm
- Static/Dynamic priorities
- pre-emptive/non pre-emptive algorithms:
  - Ability to have mutual exclusion of a device
  - Scheduler cost low
  - Small efficient
Properties that we look for:

- **Feasibility**: ability to decide "a priori" that all the constraints will be fulfilled.
- **Predictability**: response time of tasks is predictable.
- **Optimality**: Optimal if able to find a schedule of every set of feasibility tasks.
- **Complexity**: feasibility test is long and difficult?
- **Easiness to implement**
Scheduler

Scheduler works only on active tasks:

- Schedule Table: off-line scheduling
- Off-line priorities definition, root of the scheduling
- On-line analysis and scheduling
Analysis recipe

Scheduling problem:
- To Model tasks of the system and their constraints
- To Choose a scheduling algorithm
- To Validate feasibility on a set of tasks
  - Theoretical feasibility: scheduling ability and complexity
  - Empirical feasibility: implement scheduler
Task model

Task families:

- Linked task or not (Precedence constraint).
- Important task or not
- Repetitive or periodic task: activated regularly
- Non periodic task
  - Sporadic: irregular activation but a minimum time between each activation
  - Aperiodic: deadline is less strict and no minimum time between each activation
Task model

Parameters for task $i$

- $S_i$: start time when task arrives in the scheduler
- $C_i$: Computation time needed by the task (Capacity).
- $P_i$: Period
- $D_i$: Deadline of the task
- $R_i$: Earliest activation time

- Aperiodic task is defined by:
  $$(S_i, C_i, D_i, R_i)$$

- Periodic task is defined by:
  $$(S_i, C_i, D_i, P_i)$$
Let’s consider a simplified model:

- Periodic task
- Independent (No precedence constraint)
- Starting time (worst case): $S_i = 0$
- Deadline equal to Period: $P_i = D_i$
Scheduler Structure

Computation of scheduling information
- On-line or Off-line
- Period, Deadline...

Waiting queue management
- One queue for each priority
- FIFO policy
- Round Robin policy

Election phase
- Priority, deadline, ...
Rate Monotonic Algorithm

Off-line analysis, fixed priority, periodic tasks: Static software

**Rationale**

- Computation phase: priority = \( \frac{1}{\text{period}} \)
- Election phase: highest priority is chosen

**Properties**

- Low Complexity
- Optimal Algorithm in fixed priority algorithm set
Rate Monotonic Algorithm

Example

- Task 1: \(C_1 = 6, \ P_1 = 10\)
- Task 2: \(C_2 = 9, \ P_1 = 30\)
- Pre-emptive case
- Non pre-emptive case

Can be scheduled if load rate of the CPU \(U\) agrees with the following sufficient condition:

\[
U = \sum_{i}^{n} \frac{C_i}{T_i} \leq n \times (2^{\frac{1}{n}} - 1)
\]
Critical time theorem

If every tasks both arrive at the same time in a system if they respect their first deadline, then every other following deadlines will be respected whatever time they arrive in the system

- it’s a necessary and sufficient condition if every tasks arrive at the same time
- else, it is a sufficient condition

If \( D_i = T_i \), finishing test is:

\[
\forall i, 1 \leq i \leq n \quad \min_{0 \leq t \leq D_i} \sum_{j=1}^{i} \frac{C_j}{t} \left\lceil \frac{t}{T_j} \right\rceil \leq 1
\]
Response time computation

**Response time TR**: Duration between time when a task begin and time when it is finished. Result can be exact relying on task model.

\[
TR_i = C_i + \sum_{j \in hp(i)} l_j
\]

\[
TR_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{TR_i}{P_j} \right\rceil C_j
\]

Where \( hp(i) \) represents a set of tasks with a higher priority than \( i \).
Response time computation

Computation method: iterative way of evaluation

\[ w_{i}^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{P_j} \right\rceil C_j \]

- Let start with \( w_i^0 = C_i \)
- Fail if \( w_i^n > P_i \)
- Achieved if \( w_i^{n+1} = w_i^n \)

**Example**: with \( P_1 = 7, C_1 = 3, P_2 = 12, C_2 = 2, P_3 = 20, C_3 = 5 \). We assume that every task have the same priority and are ran in this order

\[ TR_1 = 3 \quad TR_2 = 5 \quad TR_3 = 18 \]
Aperiodic tasks with high priorities

Periodic task dedicated to these tasks.

- bring feasibility in the worst case
- dedicated task is not activated if there is no aperiodic task to compute
- easy solution, but sometime we loose some CPU load
Earliest Deadline First (EDF) algorithm

- Periodic task and aperiodic
- On-line Algorithm, more useful than Rate Monotonic

Rationale:

- **Computation phase**: Deadline computation
  With \( \text{priority}_i(t) \) current priority at \( t \) and \( i \) the task:
  - Aperiodic task: \( \text{priority}_i(t) = S_i + D_i - t \)
  - Periodic task: \( \text{priority}_i(t) = S_i(t) \) \{last start activation time when time is \( t \}\) + \( P_i - t \)

- **Election phase**: Earliest deadline is chosen. (Min)
EDF Example

Let consider 2 tasks :

\[ T_1 : C_1 = 6, \quad P_1 = 10 \]

\[ T_2 : C_2 = 9, \quad P_2 = 30 \]

- Pre-emptive case
- Non pre-emptive case
EDF Properties

**Ability to schedule**: Pre-emptive case, periodic and independent tasks

- Necessary and sufficient condition if \( \forall i, D_i = P_i \); Only necessary if \( \exists i, D_i \leq P_i \):

  \[
  U = \sum_{j=1}^{n} \frac{C_j}{P_j} \leq 1
  \]

- Sufficient condition if \( \exists i, D_i \leq P_i \):

  \[
  U = \sum_{j=1}^{n} \frac{C_j}{D_j} \leq 1
  \]
Server for aperiodic and sporadic jobs

- Periodic server:
  - Idle if waiting queue is empty
  - Running in its time

![Diagram showing periodic server with tasks e1 and e2]
Server for aperiodic and sporadic jobs

- Deferred server:
  - Time Budget
  - Waiting for task
  - Used to be in highest priority

![Diagram of server with time budget examples](image)

- $T_{DS} = (p_S=3, e_S=1)$
- $T_1 = (\phi=2, p=3.5, e=1.5)$
- $T_2 = (p=6.5, e=0.5)$
Server for aperiodic and sporadic jobs

- Sporadic server:
  - Like Deferred server
  - But extra computation is done in background (lowest priority)
Properties:

- Threads and process capability
- Fixed priorities, pre-emptive $\Rightarrow$ RM easy. A minimum of 32 levels is a mandatory
- One waiting queue for each priority and scheduling policy (SCHED_FIFO, SCHED_RR, SCHED_OTHERS).
- Available Services to specific users (like root)

Standard say that scheduling policies must be able to be applied each time the choice thread/process exists (ex. : choice of a thread/process to release a semaphore).
Policies

POSIX.4 policies:

```c
#define SCHED_OTHER 0
#define SCHED_FIFO 1
#define SCHED_RR 2
```

- Parameter(s): extensible to future policies

```c
struct sched_param{
   int sched_priority;
   ...
};
```

- Parameters modification:
  - Thread: creation of a thread from an attribute or modification of a running thread.
  - Inherit from a fork() or modify of a running process.
### API

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sched_get_priority_max</code></td>
<td>Get max priority value.</td>
</tr>
<tr>
<td><code>sched_get_priority_min</code></td>
<td>Get min priority value.</td>
</tr>
<tr>
<td><code>sched_rr_get_interval</code></td>
<td>Get duration of one time unit.</td>
</tr>
<tr>
<td><code>sched_yield</code></td>
<td>Free the CPU of this thread.</td>
</tr>
<tr>
<td><code>sched_setscheduler</code></td>
<td>Choose the scheduler policy.</td>
</tr>
<tr>
<td><code>sched_getscheduler</code></td>
<td>Get the scheduler policy value.</td>
</tr>
<tr>
<td><code>sched_setparam</code></td>
<td>Set parameters of the scheduler.</td>
</tr>
<tr>
<td><code>sched_getparam</code></td>
<td>Get parameters of the scheduler.</td>
</tr>
<tr>
<td><code>pthread_setschedparam</code></td>
<td>Set parameters of the scheduler for thread.</td>
</tr>
<tr>
<td><code>pthread_getschedparam</code></td>
<td>Get parameters of the scheduler for thread.</td>
</tr>
</tbody>
</table>

The last 2 function are working only on threads, other functions can be applied on process/thread.
Example

```c
struct sched_param parm;
int res = -1;
...
/* Task T1 ; P1=10 */
parm.sched_priority = 15;
res = sched_setscheduler(pid_T1, SCHED_FIFO, &parm)
if(res < 0)
    perror("sched_setscheduler task T1");
/* Task T2 ; P2=30 */
parm.sched_priority = 30;
res = sched_setscheduler(pid_T2, SCHED_FIFO, &parm)
if(res < 0)
    perror("sched_setscheduler task T2");
```
Introduction

Robotics Software Frameworks

OS : Tasks And Threads
Scheduling
Management
Process
Posix Thread
Time Services
Synchronization
Messages and Communication
IO Control

OS Driver Programming

Application

Ludovic Saint-Bauzel (Polytech’Paris UPMC)
Fork
Kill - Signal
POSIX Threads

Defined in chapter POSIX.4a (Portable Operating System Interface). This chapter describes both threads and synchronization tools that are close to threads (ex: mutex).

Properties:

- A POSIX Thread is defined with a identifier that is local to the process. It owns its stack, its context and a set of attributes that describe its behaviour.

- A POSIX Thread can be implemented in user space or in kernel space ⇒ Standard describe only the interface not the way it is coded.
POSIX Threads

- Code must be re-entrant (or "thread safe") $\Rightarrow$ code is able to be computed in multiple instances in a safe way

- re-entrant code means:
  - don’t manipulate shared variables.
  - or it manipulate shared variables in a critical section (that mean no other is able to access this variable when one is manipulating it).

- Everything must be re-entrant user code but also system libraries (ex. libc.so)
### POSIX threads

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pthread_create</code></td>
<td>Creation of a thread. Parameters: code, attributes, arg.</td>
</tr>
<tr>
<td><code>pthread_exit</code></td>
<td>End of a thread. Parameter: return error value.</td>
</tr>
<tr>
<td><code>pthread_self</code></td>
<td>Returns id of the running thread.</td>
</tr>
<tr>
<td><code>pthread_cancel</code></td>
<td>Destroys a thread. Parameter: id of the thread.</td>
</tr>
<tr>
<td><code>pthread_join</code></td>
<td>Suspend a thread until another is not finished.</td>
</tr>
<tr>
<td><code>pthread_detach</code></td>
<td>Suppress parent link between threads.</td>
</tr>
</tbody>
</table>
POSIX Threads

<table>
<thead>
<tr>
<th>pthread_kill</th>
<th>Emit a signal to a thread.</th>
</tr>
</thead>
<tbody>
<tr>
<td>pthread_sigmask</td>
<td>Modify signal mask of a thread.</td>
</tr>
</tbody>
</table>

In a POSIX.4a system, some services have a different semantic.

- fork() : create a new process containing a thread where the main() function is computed
- exit() : finish a process and all threads it contains.
POSIX Threads

Creation and join example of a thread

```c
#include <pthread.h>
void* th(void* arg)
{
    printf("I am thread %d ;
    process %d\n", 
    pthread_self(), getpid());
    pthread_exit(NULL);
}
```
POSIX Threads

Creation and join example of a thread

```c
#include <pthread.h>
void* th(void* arg)
{
    printf("I am thread %d ;
    process %d
", 
    pthread_self(), getpid());
    pthread_exit(NULL);
}
```
POSIX Threads

```c
#include <pthread.h>

int main(int argc, char* argv[])
{
    pthread_t id1, id2;
    pthread_create(&id1, NULL, th, NULL);
    pthread_create(&id2, NULL, th, NULL);
    pthread_join(id1, NULL);
    pthread_join(id2, NULL);
    printf("End of main thread %d ; process %d\n",
              pthread_self(), getpid());
    pthread_exit(NULL);
}
```

Ludovic Saint-Bauzel (Polytech'Paris UPMC)
POSIX Threads

► With Solaris:

```bash
gcc -D_REENTRANT create.c -lpthread -lrt
>a.out
  I am thread 4 ; process 5539
  I am thread 5 ; process 5539
  End of main thread 1 ; process 5539
```

► With Linux:

```bash
gcc -D_REENTRANT create.c -lpthread
>a.out
  I am thread 1026 ; process 1253
  I am thread 2051 ; process 1254
  End of main thread 1024 ; process 1251
```
Thread Attributes: properties of a thread that are defined during its creation. No heritage between father and son threads.

Example of attributes:

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>detachstate</td>
<td>pthread_join possible or not</td>
</tr>
<tr>
<td>policy</td>
<td>scheduler policy</td>
</tr>
<tr>
<td>priority</td>
<td>level of priority of the thread ($\geq 0$)</td>
</tr>
<tr>
<td>stacksize</td>
<td>Size of the allocated stack</td>
</tr>
</tbody>
</table>
Thread Attributes

During creation of a thread, an structure of type pthread_attr_t can be filled and given if we want to specify none default attributes to the new thread:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pthread_attr_init</td>
<td>Creation of a default attribute structure.</td>
</tr>
<tr>
<td>pthread_attr_delete</td>
<td>Destruction of attribute structure.</td>
</tr>
<tr>
<td>pthread_attr_setATT</td>
<td>set the value of &quot;ATT&quot; attribute.</td>
</tr>
<tr>
<td>pthread_attr_getATT</td>
<td>set the value of &quot;ATT&quot; attribute.</td>
</tr>
</tbody>
</table>

where ATT is replace by the name of the attribute.
Thread Attributes

```c
#include <pthread.h>
void* th(void* arg)
{
    printf("I am thread %d \n", pthread_self());
}
```
Thread Attributes

```c
int main(int argc, char* argv[]) {
    int i;
    pthread_t id;
    pthread_attr_t attr;
    struct sched_param param;
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
    pthread_attr_setschedpolicy(&attr, SCHED_FIFO);
    param.sched_priority = 1;
    pthread_attr_setschedparam(&attr, &param);
    for(i=1;i<10;i++)
        pthread_create(&id,&attr,th,NULL);
}
```
Thread Attributes

The TSD (Thread Specific Data area): is an area of memory where are stored specific information of each thread.

Allow extension of regular attributes.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pthread_key_create</td>
<td>Creation of a key.</td>
</tr>
<tr>
<td>pthread_key_delete</td>
<td>Destruction of the key.</td>
</tr>
<tr>
<td>pthread_getspecific</td>
<td>get pointer value linked to the key of the running thread.</td>
</tr>
<tr>
<td>pthread_setspecific</td>
<td>set pointer value linked to the key of the running thread.</td>
</tr>
</tbody>
</table>
Thread Attributes

Creation of a new attribute:

```c
pthread_key_t cd_key;

int pthread_cd_init(void)
{
    return pthread_key_create(&cd_key, NULL);
}

char* pthread_get_cd(void)
{
    return (char*)pthread_getspecific(cd_key);
}
```
int pthread_set_cd(char* cd)
{
    char* mycd = (char*)malloc(sizeof(char)*100);
    strcpy(mycd, cd);
    return pthread_setspecific(cd_key, mycd);
}

int main(int argc, char* argv[])
{
    pthread_cd_init();
    pthread_set_cd("/here/dir");
    printf("My local directory is %s\n", pthread_get_cd());
}
Introduction

Robotics Software Frameworks

OS: Tasks And Threads

Scheduling
Management

Time Services

- Posix Time
- Xenomai Time services

Synchronization
Messages and Communication
IO Control

OS Driver Programming

Application

Ludovic Saint-Bauzel (Polytech’Paris UPMC)
Time manipulation

Time linked services:

- What time is it?
- Block a thread/process during a defined time.
- Wake up a thread/process regularly (timer) ⇒ periodic tasks.

Precision of these services:

- Linked to what hardware we have (clock circuit), its features (interruption period) and also to the software that use it (interruption handler).
- Ex: Linux Intel: circuit activated periodically (10 ms) ⇒ waking between 10 to 12 ms.
Many timers are available ⇒ many physical clocks, profiling.

Need at least one real-time clock: CLOCK_REALTIME (less than 20 ms precise).

timespec structure ⇒ "theoretical" precision close to micro-seconds ....

Available services:

- Get and set clocks.
- Put a task in sleep mode.
- Link periodic timer to UNIX signals; and maybe with real-time signals. With or without automatic restart.
## Time Manipulation

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>clock_gettime</code></td>
<td>Get clock value.</td>
</tr>
<tr>
<td><code>clock_settime</code></td>
<td>Set clock value.</td>
</tr>
<tr>
<td><code>clock_getres</code></td>
<td>Get clock resolution.</td>
</tr>
<tr>
<td><code>timer_create</code></td>
<td>Create a timer.</td>
</tr>
<tr>
<td><code>timer_delete</code></td>
<td>Remove a timer.</td>
</tr>
<tr>
<td><code>timer_getoverrun</code></td>
<td>Give the number of unprocessed signals</td>
</tr>
<tr>
<td><code>timer_settime</code></td>
<td>Activate a timer.</td>
</tr>
<tr>
<td><code>timer_gettime</code></td>
<td>Get time to the end of the timer</td>
</tr>
<tr>
<td><code>nanosleep</code></td>
<td>Block a process/thread during a defined duration.</td>
</tr>
</tbody>
</table>
int main(int argc, char * argv[]){
    timer_t monTimer; struct sigaction sig;
    struct itimerspec ti;
    timer_create(CLOCK_REALTIME, NULL, &monTimer);
    sig.sa_flags = SA_RESTART;
    sig.sa_handler = trop_tard;
    sigemptyset(&sig.sa_mask);
    sigaction(SIGALRM, &sig, NULL);
    ti.it_value.tv_sec = capacite;
    ti.it_value.tv_nsec = 0;
    ti.it_interval.tv_sec = 0; // here timer is not automatically restarted
    ti.it_interval.tv_nsec = 0;
    timer_settime(monTimer, 0, &ti, NULL);
Time Manipulation : Example

```c
printf("Begin\n");
while(go>0)
    printf("I am working .....\n");
printf("Unblocked by timer : deadline missed ..\n");
}

int go=1;
void too_late(int sig){
    printf("Signal %d received\n",sig);
    go=0;
}
```

Execution :

<table>
<thead>
<tr>
<th>Begin</th>
<th>I am working .....</th>
</tr>
</thead>
<tbody>
<tr>
<td>I am working .....</td>
<td>I am working .....</td>
</tr>
<tr>
<td>I am working .....</td>
<td>Signal 14 received</td>
</tr>
<tr>
<td>I am working .....</td>
<td>Unblocked by timer : deadline missed</td>
</tr>
</tbody>
</table>
API Timer Management System

API:

- void `rt_timer_spin` (RTIME ns) : busy clock
- int `rt_timer_set_mode` (RTIME nstick) : TM_ONESHOT or period time

Hardware level:

- oneshot
- periodic
Layout

Introduction

Robotics Software Frameworks

OS : Tasks And Threads
Scheduling
Management
Time Services
Synchronization
  Mutex
  Variable Conditionnal
  Counting Semaphore
  Xenomai Native API Synchronisation tools
Messages and Communication
IO Control

OS Driver Programming
Mutex

- Optimized Semaphores to implement critical section.
- Waiting queue manager: depends of scheduling policy (SCHED_FIFO, SCHED_OTHER, ...). Default behaviour: threads are waked in decreasing order of priority.
- Behaviour is defined by a set of attributes, main parameters are:

<table>
<thead>
<tr>
<th>Attribute name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>protocol</td>
<td>Inheriting Protocol used</td>
</tr>
<tr>
<td>pshared</td>
<td>Inter-process or inter-thread mutex</td>
</tr>
<tr>
<td>ceiling</td>
<td>Priority ceiling</td>
</tr>
</tbody>
</table>
Rationale

Programming mechanism that can block a thread while a condition is not true.

- Solution is based on a mutex and a Var Cond
- Waking signal has no memory: if nothing is waiting for the condition, waking signal is lost (≠ counting semaphores).
- Waiting queue management: depend on scheduling policy (SCHED_FIFO, SCHED_OTHER, etc ...). Default behaviour: threads are waked in decreasing order of priority.
## Mutex

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pthread_mutex_init</td>
<td>Initialize a mutex.</td>
</tr>
<tr>
<td>pthread_mutex_lock</td>
<td>Lock the mutex in blocking manner eventually.</td>
</tr>
<tr>
<td>pthread_mutex_trylock</td>
<td>Non blocking try to lock the mutex</td>
</tr>
<tr>
<td>pthread_mutex_unlock</td>
<td>Release the mutex lock.</td>
</tr>
<tr>
<td>pthread_mutex_destroy</td>
<td>Destruct the mutex.</td>
</tr>
<tr>
<td>pthread_mutexattr_init</td>
<td>Initialize an attribute structure</td>
</tr>
<tr>
<td>pthread_mutexattr_setATT</td>
<td>Set attribute ATT.</td>
</tr>
<tr>
<td>pthread_mutexattr_getATT</td>
<td>Get attribute ATT value.</td>
</tr>
</tbody>
</table>

- ATT is one of the attributes of the mutex.
## API

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pthread_cond_init</td>
<td>Initialize a variable.</td>
</tr>
<tr>
<td>pthread_cond_destroy</td>
<td>Destruct a variable.</td>
</tr>
<tr>
<td>pthread_cond_wait</td>
<td>Wait a waking signal coming from a condition.</td>
</tr>
<tr>
<td>pthread_cond_signal</td>
<td>Signal that a condition is true to one thread.</td>
</tr>
<tr>
<td>pthread_cond_broadcast</td>
<td>Signal that a condition is true to all threads.</td>
</tr>
</tbody>
</table>
A classical program where a thread modify a variable: a mutex is used (var_mutex) and a condition is used (var_cond).

```c
pthread_mutex_lock(&var_mutex);
/* Modify the variable in critical section */
var = .... ;

/* If the condition is fulfilled, we alert the blocked thread */
if(condition(var))
    pthread_cond_signal(&var_cond);

pthread_mutex_unlock(&var_mutex);
```
Classical program that wait for a variable to be modified:

```c
pthread_mutex_lock(&var_mutex);
while (!condition(var)) {
    /* If the condition is not true, wait */
    pthread_cond_wait(&var_cond, &var_mutex);
}
/* use the variable in critical section */
.....

pthread_mutex_unlock(&var_mutex);
```

Attention `pthread_cond_wait` is making some implicit lock and unlock of the mutex.
Example

```c
int y=2, x=0;
pthread_mutex_t mut;
pthread_cond_t cond;
void* th(void* arg)
{
    int cont=1;
    while(cont){
        pthread_mutex_lock(&mut);
        x ++;
        printf("x ++\n");
        if (x > y){
            pthread_cond_signal(&cond);
            cont=0; }
        pthread_mutex_unlock(&mut);
    }
}
```
Example

```c
int main(int argc, char* argv){
    pthread_t id;
    pthread_mutex_init(&mut,NULL);
    pthread_cond_init(&cond,NULL);
    pthread_mutex_lock(&mut);
    pthread_create(&id,NULL,th,NULL);
    while (x <= y)
        pthread_cond_wait(&cond,&mut);
    printf("x>y is true\n");
    pthread_mutex_unlock(&mut);}
```

Execution:

<table>
<thead>
<tr>
<th>x++</th>
<th>x++</th>
</tr>
</thead>
<tbody>
<tr>
<td>x++</td>
<td>x&gt;y is true</td>
</tr>
</tbody>
</table>
Counting semaphore

- Semaphore: waiting queue + counter.
- No management of priority inversion: case must be treated in developer level.
- Used for synchronization and inter-process or inter-thread mutual exclusion.
- Waiting queue management: depend on scheduling policy (SCHED_FIFO, SCHED_OTHER, etc...). Default behaviour: threads are waked in decreasing order of priority.
- Two kind of semaphore: named and unnamed semaphore.
### Counting Semaphore

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sem_open</td>
<td>Connection to a named semaphore.</td>
</tr>
<tr>
<td>sem_close</td>
<td>Disconnection to a named semaphore.</td>
</tr>
<tr>
<td>sem_unlink</td>
<td>Destruction of a named semaphore.</td>
</tr>
<tr>
<td>sem_init</td>
<td>Initialisation of a unnamed semaphore.</td>
</tr>
<tr>
<td>sem_destroy</td>
<td>Destruction of a unnamed semaphore.</td>
</tr>
<tr>
<td>sem_post</td>
<td>Freeing a semaphore.</td>
</tr>
<tr>
<td>sem_wait</td>
<td>Acquiring a semaphore.</td>
</tr>
<tr>
<td>sem_trywait</td>
<td>Non blocking acquiring of a semaphore.</td>
</tr>
</tbody>
</table>
# include <pthread.h>
#include <semaphore.h>

sem_t sem;

int main(int argc, char* argv[]){
    pthread_t id;
    struct timespec delay;
    sem_init(&sem, 0, 0);
    pthread_create(&id, NULL, th, NULL);
    deli.tv_sec=4; deli.tv_nsec=0;
    nanosleep(&delay, NULL);
    printf("main thread %d : free from
        the other thread \n", pthread_self());
    sem_post(&sem);
    pthread_exit(NULL);
}
Counting Semaphore Example

```c
void* th(void* arg)
{
    printf("thread %d waiting\n", 
        pthread_self());
    sem_wait(&sem);
    printf("thread %d unblocked \n", 
        pthread_self());
}
```

Execution

```text
thread 4 waiting
main thread 1 : free from
the other thread
thread 4 unblocked
```
API Mutexes

```c
int rt_mutex_create (RT_MUTEX *mutex, const char *name)
int rt_mutex_acquire (RT_MUTEX *mutex, RTIME timeout)
int rt_mutex_release (RT_MUTEX *mutex)
```
API Condition variables

```c
int rt_cond_create (RT_COND *cond, const char *name)
int rt_cond_signal (RT_COND *cond)
int rt_cond_broadcast (RT_COND *cond)
int rt_cond_wait (RT_COND *cond, RT_MUX *mutex, RTIME timeout)
```
API Counting Semaphores

```c
int rt_sem_create (RT_SEM *sem, const char *name,
                   unsigned long icount, int mode)
int rt_sem_p     (RT_SEM *sem, RTIME timeout)
int rt_sem_v     (RT_SEM *sem)
int rt_sem_broadcast (RT_SEM *sem)
```
Layout

Introduction

Robotics Software Frameworks

**OS : Tasks And Threads**
- Scheduling
- Management
- Time Services
- Synchronization

**Messages and Communication**
- Signals
- Mail Queue
- Xenomai Native API

**IO Control**

**OS Driver Programming**
Real-Time Signals

**Signal**: event delivered in asynchronous way to a process/thread (software interruption).

- Blocked signals (or masked) pending or delivered.
- Default behaviour can be modified by user.
- One signal register for each thread/process.
Real-Time Signals

Existing mechanism in POSIX.1 but with the following issues:

- Implementation pending signal register = unsafe delivery (possible loss).
- Emitting order not respected when they are delivered.
- Does not contain many information and few of them are available for user: not adapted to IPC implementation.
- Not efficient (slow: high latency).
Real-Time Signals : POSIX Interface

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>kill</td>
<td>Emission of a signal.</td>
</tr>
<tr>
<td>sigaction</td>
<td>Connection of a handler to a signal.</td>
</tr>
<tr>
<td>sigemptyset</td>
<td>Initialize with all signals disabled. (mask all signals)</td>
</tr>
<tr>
<td>sigfillset</td>
<td>Initialize with all signals enabled. (unmask all signals)</td>
</tr>
<tr>
<td>sigaddset</td>
<td>Add a signal in the set (unmask this signal).</td>
</tr>
<tr>
<td>sigdelset</td>
<td>Remove a signal from the set (mask this signal).</td>
</tr>
<tr>
<td>sigismember</td>
<td>Check if the signal is enabled in the set.</td>
</tr>
<tr>
<td>sigsuspend</td>
<td>Block a process until a signal is received.</td>
</tr>
<tr>
<td>sigprocmask</td>
<td>Install a mask.</td>
</tr>
</tbody>
</table>
Signals: Non RT Example

```c
void handler(int sig)
{
    printf("Signal %d received\n", sig);
}

int main(int argc, char * argv[])
{
    struct sigaction sig;
    sig.sa_flags = SA_RESTART;
    sig.sa_handler = handler;
    sigemptyset(&sig.sa_mask);
    sigaction(SIGUSR1, &sig, NULL);
    while(1); // program computes !!
}
```

Execution:

```
$ sig &
$ kill -USR1 14090
Signal 10 received
```
Real-Time signals: POSIX Extended

Range of new signals numbered from SIGRTMIN to SIGRTMAX (that represent a minimum of RTSIG_MAX signals)

- Possibility to use a value linked to this signal.
- No loss of signal: using of a queue for pending signals.
- Ordered delivery: respecting scheduling policy chosen + priority linked to a signal ⇒ SIGRTMIN has the highest priority.
- Emitted with a kill, sigqueue, with a timer or by an asynchronous IO.
Real-Time Signals: POSIX Extended

Complementary interfaces of POSIX.4:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sigqueue</td>
<td>Emit a real-time signal.</td>
</tr>
<tr>
<td>sigwaitinfo</td>
<td>Wait for a signal without handler execution</td>
</tr>
<tr>
<td>sigtimedwait</td>
<td>Idem above + timeout on blocking time.</td>
</tr>
</tbody>
</table>
int main(int argc, char * argv[]){
    struct sigaction sig;
    union sigval val;
    int cpt;
    sig.sa_flags=SA_SIGINFO;
    sig.sa_sigaction=handler;
    sigemptyset(&sig.sa_mask);
    if(sigaction(SIGRTMIN,&sig,NULL)<0)
        perror("sigaction");
    for(cpt=0;cpt<5;cpt++){
        struct timespec delai;
        delai.tv_sec=1; delai.tv_nsec=0;
        val.sival_int=cpt;
        sigqueue(0,SIGRTMIN,val);
        nanosleep(&delai,NULL);
    }
}
RealTime Signals : RT Example

```c
void handler ( int sig , siginfo_t *sip , void *uap )
{
    printf("Received signal %d, val = %d \n", sig, sip->si_value.sival_int);
}
```

Execution :

```
$rt-sig
Received signal 38, val = 0
Received signal 38, val = 1
Received signal 38, val = 2
Received signal 38, val = 3
Received signal 38, val = 4
```
With UNIX, Communication mechanism is the pipeline (named or not) ⇒ too abstract for real-time constraints.

Mail queues in POSIX.4 have the same features :

- Priority in emission/reception.
- Can work in Non-blocking
- Can preallocate resources.
- Can communicate in Inter-Process and Inter-Thread mode.
- Unfortunately, priority inversion issues are not addressed.
### API

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mq_open</td>
<td>Creation or connection to a queue.</td>
</tr>
<tr>
<td>mq_unlink</td>
<td>Destruction of a queue.</td>
</tr>
<tr>
<td>mq_receive</td>
<td>Reception of the oldest and the most important mail.</td>
</tr>
<tr>
<td>mq_send</td>
<td>Emission of a mail with a given priority</td>
</tr>
<tr>
<td>mq_close</td>
<td>Disconnection to a queue.</td>
</tr>
<tr>
<td>mq_notify</td>
<td>Notification that there is a new mail.</td>
</tr>
<tr>
<td>mq_setattr</td>
<td>Set attributes of the queue.</td>
</tr>
<tr>
<td>mq_getattr</td>
<td>Get attributes of the queue.</td>
</tr>
</tbody>
</table>
Example

```c
/* Emission with a priority 1 (from 0 to MQ_PRIO_MAX) */
  mq_send(id,buff,100,1);
pthread_create(&tid,NULL,consumer,NULL);
pthread_exit(NULL);
}

void* consumer(void* arg){
  mqd_t id;
  char buff [100];
  id=mq_open("/myqueue",O_RDONLY);
mq_receive(id,buff,100,NULL);
  printf("msg = %s\n",buff);
mq_unlink("/myqueue");
pthread_exit(NULL);
}
```
# Example

```c
#include <pthread.h>
#include <mqueue.h>
int main(int argc, char* argv[])
{
    pthread_t tid;
    mqd_t id;
    char buff[100];
    struct mq_attr attr;
    attr.mq_maxmsg=100;
    attr.mq_flags=0;
    attr.mq_msgsize=100;
    id=mq_open("/mafile", O_CREAT | O_WRONLY, 444, &attr);
    strcpy(buff,"Hi!!");
    ... 
}
```
API Event flags groups

- Synchronizing object based on a long word structure.
- Any bit in the word can be used as a user flag
- Task oriented signal mechanism
- Conjunctive or Disjunctive
API Event flags groups

```c
int rt_event_create (RT_EVENT *event, const char *name,
                     unsigned long ivalue, int mode)
int rt_event_signal (RT_EVENT *event, unsigned long mask)
int rt_event_wait (RT_EVENT *event, unsigned long mask,
                   unsigned long *mask_r, int mode, RTIME timeout)
int rt_event_clear (RT_EVENT *event, unsigned long mask,
                    unsigned long *mask_r)
```
API Messages queue

```c
int rt_queue_create (RT_QUEUE *q, const char *name,
                      size_t poolsize, size_t qlimit, int mode)

void *rt_queue_alloc (RT_QUEUE *q, size_t size)

int rt_queue_free (RT_QUEUE *q, void *buf)

int rt_queue_send (RT_QUEUE *q, void *mbuf, size_t size, int mode)

int rt_queue_write (RT_QUEUE *q, const void *buf, size_t size, int mode)

ssize_t rt_queue_receive (RT_QUEUE *q, void **bufp, RTIME timeout)

ssize_t rt_queue_read (RT_QUEUE *q, void *buf, size_t size, RTIME timeout)
```
API Messages pipe

int        rt_pipe_create (RT_PIPE *pipe, const char *name,
                         int minor, size_t poolsize)
ssize_t   rt_pipe_receive (RT_PIPE *pipe,
                          RT_PIPE_MSG **msgp, RTIME timeout)
ssize_t   rt_pipe_read (RT_PIPE *pipe, void *buf, size_t size, RTIME timeout)
ssize_t   rt_pipe_send (RT_PIPE *pipe, RT_PIPE_MSG *msg,
                       size_t size, int mode)
ssize_t   rt_pipe_write (RT_PIPE *pipe, const void *buf, size_t size,
                       int mode)
ssize_t   rt_pipe_stream (RT_PIPE *pipe, const void *buf, size_t size)
RT_PIPE_MSG * rt_pipe_alloc (RT_PIPE *pipe, size_t size)
int       rt_pipe_free (RT_PIPE *pipe, RT_PIPE_MSG *msg)
int       rt_pipe_flush (RT_PIPE *pipe, int mode)
Example: Pipe Kernel side

```c
#include <sys/types.h>
#include <fcntl.h>
#include <string.h>
#include <stdio.h>
#include <native/pipe.h>
#define TASK_PRIO 0 /* Highest RT priority */
#define TASK_MODE T_FPU | T_CPU(0) /* Uses FPU, bound to CPU #0 */
#define TASK_STKSZ 4096 /* Stack size (in bytes) */

RT_TASK task_desc;
RTPIPE pipe_desc;

void task_body(void){
  RTPIPE_MSG *msgout, *msgin;
  int err, len, n;
  for (; ; ){ /* ... */
    len = sizeof("Hello");
    /* Get a message block of the right size in order to
    initiate the message-oriented dialog with the
    */
```
Example: Pipe Kernel side

```c
#include <native/pipe.h>
RTPIPE pipe_desc;
int init_module (void)
{
    int err;
    /* Connect the kernel-side of the message pipe to the
    special device file /dev/rtp7. */
    err = rt_pipe_create (&pipe_desc , "MyPipe" , 7 , NULL);
    ...
}
```
Example: Pipe User side

From a regular Linux process:
```
#include <native/pipe.h>
int pipe_fd;
int main (int argc, char *argv[])
{
    /* Open the Linux side of the pipe. */
    pipe_fd = open("/dev/rtp7", O_RDWR);
    /* ou grace au service de registres*/
    pipe_fd = open("/proc/xenomai/registry/native/pipes/MyPipe", O_RDWR);
    ...  
    /* Write a message to the pipe. */
    write(pipe_fd,"hello world",11);
    ...
}
```
Introduction

Robotics Software Frameworks

**OS : Tasks And Threads**
- Scheduling
- Management
- Time Services
- Synchronization
- Messages and Communication
- IO Control
  - Memory
  - IO

**OS Driver Programming**

Application

Ludovic Saint-Bauzel (Polytech’Paris UPMC)
A shared time system leads to possible time indeterminism because:

- Dynamic memory allocation.
- Page swap: swapin=swapout.

Solution: limit dynamic memory allocation and lock pages in central memory (POSIX.4 Interface):

- mlockall()=munlockall(): lock/unlock swap with all memory pages of this process.
- mlock()=munlock(): lock/unlock a range of addresses.

**Warning**: mlock() is not portable (because there is no memory model in POSIX standard).
API Memory heap

```c
int rt_heap_create (RT_HEAP *heap, const char *name, size_t heapsize, int mode)
int rt_heap_alloc (RT_HEAP *heap, size_t size, RTIME timeout, void **blockp)
int rt_heap_free (RT_HEAP *heap, void *block)
```
Posix AIO

- aio_read
- aio_write
- aio_return, aio_error
- aio_cancel
- aio_fsync
- aio_suspend
Example:

```c
sa.sa_sigaction = aioSigHandler;
if (sigaction(IO_SIGNAL, &sa, NULL) == -1) errExit("sigaction");
aiocb.aio_nbytes = BUF_SIZE;
aiocb.aio_reqprio = 0;
aiocb.aio_offset = 0;
aiocb.aio_sigevent.sigev_notify = SIGEV_SIGNAL;
aiocb.aio_sigevent.sigev_signo = IO_SIGNAL;
aiocb.aio_sigevent.sigev_value.sival_ptr = &ioValue;
aio_read(&aiocb);
err = aio_error(&aiocb);
switch (err) {
    case 0: printf("I/O succeeded\n"); break;
    case EINPROGRESS: printf("In progress\n"); break;
    case ECANCELED: printf("Canceled\n"); break;
    default: errMsg("aio_error"); break;
}
aio_return(&aiocb);
```
API Interrupt management

```c
int rt_intr_create (RT_INTR *intr, const char *name, unsigned irq,
                    rt_isr_t isr, rt_iack_t iack, int mode) [Kernel]
int rt_intr_create (RT_INTR *intr, const char *name, unsigned irq, int mode) [User]
int rt_intr_enable (RT_INTR *intr)
int rt_intr_disable (RT_INTR *intr)
int rt_intr_wait (RT_INTR *intr, RTIME timeout)
```
Example: User Interruption

```c
#include <sys/mman.h>
#include <native/task.h>
#include <native/intr.h>

#define IRQ_NUMBER 7 /* Intercept interrupt #7 */
#define TASK_PRIO 99 /* Highest RT priority */
#define TASK_MODE 0 /* No flags */
#define TASK_STKSZ 0 /* Stack size (use default one) */

RT_INTR intr_desc;
RT_TASK server_desc;

void irq_server (void *cookie){
    for (; ; ) {
        /* Wait for the next interrupt on channel #7. */
        err = rt_intr_wait(&intr_desc, TM_INFINITE);
        if (!err) {
            /* Process interrupt. */
        }
    }
}
```
Example: User Interruption

```c
int main (int argc, char *argv[]){
    int err;
    mlockall(MCL_CURRENT|MCL_FUTURE);
    /* ... */
    err = rt_intr_create(&intr_desc,"MyIrq",IRQ_NUMBER,0);
    /* ... */
    err = rt_task_create(&server_desc,
                "MyIrqServer",
                TASK_STKSZ,
                TASK_PRIO,
                TASK_MODE);
    if (!err)
        rt_task_start(&server_desc,&irq_server,NULL);
    /* ... */
}
void cleanup (void){
    rt_intr_delete(&intr_desc);
    rt_task_delete(&server_desc);}
```
OS Driver Programming

General overview

Introduction

Linux Kernel Introduction

Kernel

Tools

User Space Mechanisms

Kernel/Hardware Interface

Driver (Module)
Context

**Figure:** Robot Unified Platform Initiative

- OS Hardware
- Native Robot S/W
- O/S
- Hardware
- Robot Middleware for RUPI

**RUPI Server Applications**
- Standard Interface
- Voice
- Image
- Navigation

**Contents Server**

**RUPI Server Middleware**

**Server Platform**

**Backend Server**
- Authentication
- Robot/User management
- Billing...

**URS Server**

**URC Robot**

**URC Server**

**BcN**

**WG**

**Wireless AP (WiFi, WiBro...)**

**Robot Applications**
- Standard Interface
- System Component
- Application Component

**Context**

Ludovic Saint-Bauzel (Polytech’Paris UPMC)
OS Driver Programming

State of Art

- **OS Hardware**
  - Linux, WindowsCE, VxWorks
- **Native Robot SW**
  - Xenomai, RTAI, RTLinux, OROCOSRT
- **MiddleWare**
  - RoSta Robotics Standards: OROCOS, Microsoft Robotics Studio, URBI, Robotic Operating System (ROS), ORCA, MARIE...
- **Communication Framework**
  - OROCOS, Microsoft Robotics Studio, URBI, Robotic Operating System, OpenJaus
History

Creator: Linus Torvalds for personal reasons

- understand Operating System
- funding no cheap OS present on the market place
- Minix had too many limits

Main initial choices taken by Linus

- using GNU licenses tools compiler and tools
- share sources on Internet
- GNU GPL License used
- collaborative work $\Rightarrow$ quick emergence of a developer community
Key dates

- First stable version (1.0) in 1994;
- POSIX compatibility added in version 1.2 (1995);
- Support Multi Processors (SMP) and portability improvement in versions 2.0 (1996);
- Improvement of general performances, increase of number of platform and number of supported devices in versions 2.2 (1999);
- Increasing performances with SMP architectures and network layer in versions 2.4 (2001);
- Wide set of improvements internally and in the API (pre-emptive, 64 bit/embedded architectures, unified devices management, ...) (2003).
Features

- Monolithic Kernel
- UNIX type architecture all files concept
- MultiTasks
- MultiUsers
- MultiProcessors
- Multiplatform
Properties

- Dynamic load of kernel modules
- Inter Processes memory protection
- Load of executable on demand
- Page sharing between each executable
- Dynamic cache size management
- Shared libraries supported
- ...
Layout

Introduction

Robotics Software Frameworks

OS : Tasks And Threads

OS Driver Programming

General overview

Kernel
  Main Concepts
  Components and mechanisms
  Developing environment

Tools

User Space Mechanisms

Kernel/Hardware Interface

Driver (Module)
User/Kernel World

Material protection using CPU mechanisms.

Two distinct worlds

- Kernel space where all is permitted even the worst
- User space where possibilities are restricted with safety mechanisms (ex: memory access)
Schema
Three kind of transitions user/kernel:
- system calls;
- interruptions ⇒ Interruption controller in the kernel;
- exceptions (ex: illegal memory access, illegal instructions ...).

Limited number of entry points in the kernel (∼190) for users ⇒ system calls;

Each user process can be divided in two main parts:
- a kernel part ⇒ System calls (ex: `open()`, `read()`...);
- a User part ⇒ all the other things (ex: management, algorithms...).
Rationale

Main rule
"Always try to bring complexity outside of the kernel"

Consequences

- kernel must stay as small as possible
- system calls must be limited in their numbers
- all the complexity are in the library called libc or
- in user software
Motivations

Kernel device drivers or user mode software?

Kernel device drivers:
- Direct Access: Memory and devices;
- quickness (less software layers);
- manage concurrent accesses to devices;
- manage access rights.

User software solution:
- keep complexity outside of the kernel;
- software are running in safe mode;
- easy to debug.

Ex: Ghostscript (printers), Sane (scanners)…
Kernel Structures

Many structures are commonly used in the kernel:

- **Complex structures** (ex: `task_struct` defined in `<linux/sched.h>`);
- **complex linked lists** (ex: structures list `vm_area_struct` organised as AVL tree - `<linux/mm.h>`);
- **pseudo-objects** composed of pointers on functions structures as a method of it (ex: `module_structure` defined in `<linux/module.h>`).
Memory management

Kernel address space
- linear view of the RAM no segments
- no check of illegal accesses

User address space
- virtual dont represent real memory
- private for each process
- in advance allocation real allocation when really in use
- no size limit except hardware ones sometimes software if it doesnt fit the hardware (ex: DOS)
Memory Translation Virtual-Real

Virtual Address
Segment Number
Displacement

Segmentation table
information fields
Base Address (or NIL)

Addition
Real Address

Virtual Address
Page Number
Displacement

Page table
information fields
Block number (or NIL)

Block Number
Displacement
Real Address
Process

Elementary Execution Unit of the Operating System
Two kinds

- **process** based on a copy of resources coming from father process improved with the use of copy on write mechanisms or vfork

- **thread** share the maximum part of the resources coming from the main process $\Rightarrow$ context switch accelerated
Scheduling

Pre-emptive Multi-task:
Kernel is pre-emptive itself (from 2.6 version);
UNIX Policy to share CPU time between each processes;
Real-Time API in POSIX.1b;
Linux is nevertheless a real-time kernel:
  ▶ Execution time not **predictive**;
  ▶ non-determinist system. Priorities management (41 normal levels + 99 real-time levels).
Interruption management

Linked to a device (ex: bus, interface, timer...); Material interruptions can pre-empt a process even if it work in kernel mode;

Two phases computing:

- top-half: high-speed part and critical (ex: acknowledge an interruption for a device), and

- bottom-half: slow part in a queue of tasks that is emptied by the scheduler (ex: processing associated to the interruption). Possibility to share IRQ.
Mutual Exclusion

Linux Kernel is re-entrant and pre-emptive; there is consequently many possible concurrencies:

- between each CPU (SMP architecture);
- between Interruption Controller and System Calls;
- between different System Calls (because of explicit or implicit pre-emption);

⇒ Blocking mechanism (locking) for protection:

- global variables,
- non-re-entrant functions,
- critical hardware accesses.
File Systems

Fundamental of every UNIX based systems;
Two main kind:

- real: Associated to a real storage device (ex. hdd, CDROM ...);
- virtual: emulates a storage device to have access to advantages of files mechanisms (ex: drivers, /proc ...).
Abstraction layer to implement writing mechanism with file system (VFS) :

- can be based on every bloc device ;
- basic mechanism are present : * open()/close(), read()/write(), lseek()... ; *inheriting generic mechanisms if not implemented.
- need of specific tools (ex : formatting, check, tuning...)
One of the big asset of Linux system; Many devices and protocols are supported; Divided in two set of sub-systems:

- Driver of interfaces that fit with physical level (1) and Data level (2) of OSI Model (ex: Ethernet, Token Ring, PPP...);
- Protocols corresponding to layers Net (3) and Transport (4) (ex: TCP, IP, IPX, Appletalk...).
Interface drivers use device drivers to have access to them; Communication between device drivers, interface drivers and protocols is done with packets (ex: structure sk_buff defined in `<linux/skbuff.h>`); IP layer integrate route tables and name resolution.
Dynamic Kernel Modules: object files (.o/.ko) implementing a functionality of the kernel;

Used to reduce the size of the Kernel and bring it more flexibility;

We can load and unload in dynamic way (during run-time) a new functionality (ex: driver, protocol, file system ...). Module Dependencies management with `modprobe` tool;

Management of dynamic link edition with `insmod`.
Device drivers

Piece of software plugged in the kernel to bring an interface between kernel and hardware devices;

There must be two different interfaces:
- one for users that is generic and stable;
- one to communicate with the kernel.

With Linux, user interface is provided as a special file;

So we are able to use classic system call like `open()`, `read()`, `write()`;

Can be implemented as a module dynamically linked "on demand" or statically compiled in the kernel.
Aim of the part

Sources organization;

Developing tools;

Debugging methods in the kernel;

Interfaces with the community;

Licenses.
Distribution

Sources archives in files linux-X.Y.Z.tar.bz2

- X = VERSION (~ 4 5 years) : number of version ;
- Y = PATCHLEVEL (~ 1 2 years) : number of sub-version (even ⇒ stable, odd ⇒ unstable) ;
- Z = SUBLEVEL (~ 1 6 months) : number of sub-sub-version.

Update patches in files like patch-X.Y.Z+1.bz2 ;

Files .sign to check signature of archives and patches
Ex for signature :

gpg -keyserver wwwkeys.pgp.net -recv-keys 0x517D0F0E
gpg -verify linux-X.Y.Z.tar.bz2.sign linux-X.Y.Z.tar.bz2
TreeView

Documentation/ : documentation related to the kernel and its sub-systems (ex: sound/, DocBook/...) ;

arch/ : specific part for architectures (ex: alpha/, arm/, i386/...) ;

drivers/ : device drivers (ex: usb/, net/...) ;

fs/ : file systems (ex: xt2/, fat/...) et VFS ; include/ : headers files (ex: asm-i386/, linux/, net/...) ;

init/ : initialisation process (boot) of the kernel ;
TreeView

ipc/ : inter-processes communication mechanisms (ex: shared memories, semaphores...) ;

kernel/ : core of the system (ex: scheduler, signals...) ;

lib/ : mini C library to be used in the kernel (ex: strcmp(), sprintf()...) ;

mm/ : memory management ;

net/ : interfaces and network protocols (ex: ethernet/, ipv4/...) ;

scripts/ : useful scripts for configuration and compilation of the kernel.
In 2.6 version, new directories:

- **crypto**: generic API cryptography and their implementations;
- **security**: security mechanisms running in the kernel and implementations (SELinux);
- **sound**: sound sub-systems (in drivers before).
Introduction

Robotics Software Frameworks

OS : Tasks And Threads

**OS Driver Programming**
- General overview
- Kernel
- Tools
  - Dev. tools
  - Debugging
  - Practical
- User Space Mechanisms
- Kernel/Hardware Interface
- Driver (Module)
Compilation

Only supported tools for compilation:

- GCC (with its specific optimization set...);
- GNU Make (syntax...).

Lower versions of tools ⇒ Documentation/Changes;

Use optimization flag -O of GCC to have inline in headers interpreted.
Ex Makefile

CC = gcc
CCOPTS = -Wall -Wstrict-prototypes -O2
KSRC = /lib/modules/$(shell uname -r)/build
MODVERSIONS = -DMODVERSIONS
MODVERSIONS += -include $(KSRC)/include/linux/modversions.h
CFLAGS = -I$(KSRC)/ ... $(CCOPTS) -DMODULE $(MODVERSIONS)
CFLAGS += -D__KERNEL__

%.o : %.c
  $(CC) $(CFLAGS) -c $< -o $@

Ex : Makefile
C Language and C library

No libC for kernel programming;

Useful Functions in the directory lib/ of kernel sources;

Coding standards described in Documentation/CodingStyle;

It is advised to split big C files into many smaller files with well defined functions inside (ex: software layer);

Object files (.o) can then be merged in one module (one module_init()...) with the linker;

Ex:

```
> ld -r part1.o part2.o -o module.o
```
Patches

Collecting modifications in one or more files;

Must keep the original file or directories;

Ex: creation of a patch

```
diff -urN <orig> <dest> > patch_<description>
```

Ex: using the patch

```
cd <origine>
patch -p1 < patch_<description>
```
Kernel function **printk()** to send information in standard output (console);

Multiple levels of debugging are defined in `<linux/kernel.h>` (ex: `KERN_EMERG`, `KERN_INFO`...);

Messages are logged by syslogd daemon (entry kern.*);

**dmesg** software display and check the round buffer used by **printk()**;

Prototype:

```c
int printk (const char *fmt, ...);
```
kgdb patch for the kernel (http://kgdb.sourceforge.net);

Need of a second machine:

- link the target with a serial connection;
- use GDB-client to debug the target.

Easy to use and quick way to debug (source mode debugging).
KDB

Embedded in the kernel debugging tool;

Developed by SGI (http://oss.sgi.com/projects/kdb/);

It includes:

- A set of user commands that allow to debug the kernel and real-time kernels;
- a kernel debugger in assembly mode (not in source mode)

Useful debugger but hard to use because of its complexity.
Profiling

It is possible to analyse time spent by each function of the kernel:

- boot parameter: `profile=n`;
- binary information in `/proc/profile`;
- can become readable with `readprofile` tool.

Allow to identify:

- structural misbehaviour;
- to be optimised parts of the driver.

2.6 kernel provides Oprofile, a more powerful kernel/user profiler.
Other possibilities

On request Debugging:

- virtual file in /proc;
- system call ioctl().

GDB debugging read only mode the virtual file /proc/kcore;

Using Linux User Mode for hardware independent parts;

Analysing outputs of the software with strace;

In oops case, analysing et interpretation with ksymoops tool (use the file System.map).
Get sources of a linux kernel on www.kernel.org
  - Extract kernel sources: `tar xvfj linux-X.Y.Z.tar.bz2`
  - Go to the root of the sources: `cd linux-X.Y.Z`
Apply some patches: `patch -p1 < /path/to/patch-foo`
Configure the kernel: `make xconfig`
  - Compile (2.4 kernel): `make dep clean bzImage modules`
  - Compile (2.6 kernel): `make`
Install: `make modules_install install`
Reboot.
Layout

Introduction

Robotics Software Frameworks

OS : Tasks And Threads

OS Driver Programming
  General overview
  Kernel
  Tools
  User Space Mechanisms
    Review
    System Calls
    Hardware access
  Kernel/Hardware Interface
  Driver (Module)
Threads

Signals

Systems Calls

Hardware access
- Threads: You know!
- Signals: You know!
- Systems Calls
- Hardware access
ioctl : Devices Control

#include <sys/ioctl.h>

int ioctl(int d, int request, ...)

Example:

0x00000606 LPGETIRQ int *
0x00000608 LPWAIT int
0x00000609 LPCAREFUL int
0x0000060A LPABORTOPEN int
#include <syscall.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>

int main(void) {
    long ID1, ID2;
    /* -----------------------------*/
    /* direct system call */
    /* SYS_getpid (func no. is 20) */
    /* -----------------------------*/
    ID1 = syscall(SYS_getpid);
    printf("syscall(SYS_getpid)=\l\d\n", ID1);
    /* -----------------------------*/
    /* "libc" wrapped system call */
    /* SYS_getpid (Func No. is 20) */
    /* -----------------------------*/
    ID2 = getpid();
    printf("getpid()=\l\d\n", ID2);
    return(0);
}
Input/Output Ports and Authorization

Address range used by present devices on IO bus (ex: controlling registers);

We can access these ports from the kernel with functions:

- `in{b,w,l}()/out{b,w,l}()`: read/write 1, 2 or 4 octets on an IO port,
- `in{b,w,l}_p()/out{b,w,l}_p()`: read/write 1, 2 or 4 octets on an IO port and pause until the I/O completes,
- `ins{b,w,l}()/outs{b,w,l}()`: read/write sequences of 1, 2 or 4 octets on IO ports.

To be authorize to access data in user space you have to ask for permission with the functions:

- `int ioperm(unsigned long from, unsigned long num, int turn_on)`: only the first 0x3ff I/O ports can be specified in this manner.
- `int iopl(int level)`: changes the I/O privilege level of the calling process, Warning !! not advised.
Prototypes

Prototypes x86 (<asm/io.h>):

- `unsigned char inb (unsigned short port);`
- `void outb (unsigned char byte, unsigned short port);`
- `unsigned char insb (unsigned short port, void *addr, unsigned long count);`
- `void outsb (unsigned short port, void *addr, unsigned long count);`
Mapping addresses

**Memory**

- `mmap`
- `mremap`
- `mprotect`

**IO** In some cases, it is mandatory to map addresses range of IO in linear address space of the kernel (ex: PCI ⇒ addresses > PAGE_OFFSET):
  - `ioremap()` : map physical addresses range to a linear address range (similar to `vmalloc()`),
  - `iounmap()` : free an address range already mapped.
Memory Mapping

```c
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]){
    char *addr;
    int fd;
    struct stat sb;
    off_t offset, pa_offset;
    size_t length;
    ssize_t s;
    fd = open(argv[1], O_RDONLY);
    if (fstat(fd, &sb) == -1) /* To obtain file size */
        offset = atoi(argv[2]);
    pa_offset = offset & ~(sysconf(_SC_PAGE_SIZE) - 1);
    /* offset for mmap() must be page aligned */
```
if (argc == 4) {
    length = atoi(argv[3]);
    if (offset + length > sb.st_size)
        length = sb.st_size - offset;
    /* Can't display bytes past end of file */
} else {
    /* No length arg ==> display to end of file */
    length = sb.st_size - offset;
}
addr = mmap(NULL, length + offset - pa_offset,
            PROT_READ, MAP_PRIVATE, fd, pa_offset);
s = write(STDOUT_FILENO,
           addr + offset - pa_offset, length);
exit(EXIT_SUCCESS);
}
**Kernel Services**

(`/sys` : system / driver status information)

```
> cat /sys/bus/usb/devices/usb4/4-2/product
USB-PS/2 Optical Mouse
```

(`/proc` : Hardware Status Information)

```
> cat /proc/iomem
...
fec00000-fec00fff : IOAPIC 0
...
fee00000-fee00fff : Local APIC
...
> cat /proc/ioports
0000-001f : dma1
0020-0021 : pic1
0040-0043 : timer0
0050-0053 : timer1
0060-0060 : keyboard
...
Introduction

Robotics Software Frameworks

OS : Tasks And Threads

OS Driver Programming
  General overview
  Kernel
  Tools
  User Space Mechanisms
  Kernel/Hardware Interface
    Devices
    Interruptions and events
    Waiting queues
  Driver (Module)
IO Port Reservation

Drivers can access IO ports that are not there own devices (ex : probing);

In order to avoid conflicts, kernel can reserve ports for drivers:

- int request_region() : reserve a IO port;
- check_region() : check if the port is not yet reserved (deprecated in 2.6);
- release_region() : free an IO port.

List of ports already reserved appears in /proc/ioports.
Memory Reservation

To avoid conflicts, kernel can reserve IO memory region for some device driver:

- `request_mem_region()` : reserve one memory range of address,
- `check_mem_region()` : check if the region is already reserved (deprecated in 2.6),
- `release_mem_region()` : free this region.

Listing of memory regions reserved appears in `/proc/iomem`;
Prototypes

Prototypes (<asm/io.h> and <linux/ioport.h>):
- void * ioremap (unsigned long offset, unsigned long size);
- void iounmap (void *addr);
- struct resource * request_{mem_}region (unsigned long start, unsigned long n, const char *name);
- int check_{mem_}region (unsigned long start, unsigned long n);
- void release_{mem_}region (unsigned long start, unsigned long n);
Hardware access functions

It is then possible to access to shared memory of IO with the following functions:

- `readb,w,l()/writeb,w,l()` : read/write respectively 1, 2 ou 4 octets in IO memory;
- `memcpy_{from,to}io()` : read/write octet continuous block in IO memory;
- `memset_io()` : fill a specific range of data with a fixed value;
- `virt_to_bus()/bus_to_virt()` : translate between a virtual linear address and real address on the bus.
Prototypes

x86 prototypes (\texttt{<asm/io.h>}) :

\begin{itemize}
  \item char \texttt{readb (void *addr)} ;
  \item void \texttt{writeb (char byte, void *addr)} ;
  \item void * \texttt{memcpy_{from,to}io (void *dest, const void *src, size\_t count)} ;
  \item void * \texttt{memset\_io (void *addr, int pattern, size\_t count)} ;
  \item unsigned long \texttt{virt\_to\_phys (volatile void *addr)} ;
  \item void * \texttt{phys\_to\_virt (unsigned long addr)} ;
\end{itemize}
Kernel memory

Many methods exists in kernel to allocate memory :

- `kmalloc()`/`kfree()` : allocate/de-allocate a real continuous memory block in the kernel (`GFP_KERNEL ⇒ can sleep, GFP_ATOMIC ⇒ atomic`),

- `__get_free_pages()`/`free_pages()` : allocate/de-allocate an integer number (2^n) of continuous real memory pages (idem previous but for biggest needs - limit : 128 ko),

- `vmalloc()`/`vfree()` : allocate/de-allocate a virtual memory block composed by many discontinuous memory blocks of real memory.
Prototypes

Prototypes (\textless linux/slab.h\textgreater  et \textless linux/vmalloc.h\textgreater ) :

\begin{itemize}
  \item void * kmalloc (size\_t size, int flags) ;
  \item void kfree (const void *addr) ;
  \item unsigned long \_\_get\_free\_pages (int gfp\_mask, unsigned long order) ;
  \item void free\_pages (unsigned long addr, unsigned long order) ;
  \item void * vmalloc (unsigned long size) ;
  \item void vfree (void *addr) ;
\end{itemize}
Supervision of Input/Output events, I/O often asynchronous and unpredictable (ex: keyboard);
Two methods to supervise I/O:
- active waiting (polling);
- interruptions.

In polling mode, CPU is released with schedule() function after each unsuccessful test;
Interruptions et events

Ex : polling

```c
for ( ; ;) {
    if (read_state(carte) & ETAT_END)
        break ;
    schedule() ;
}
```

In interruptible mode, calling process is asleep and put in a waiting queue ; Interruption manager will have to wake the process (or processes according to its policy)
Interruptions et events

Interruption manager is called interrupt handler. Properties:

- must be fast;
- can’t call sleeping function (ex: kmalloc() non atomic).

For these reasons, interruption management is divided in 2 parts:

- interrupt handler: fast and uninterrupted part, and
- bottom-half: slowest part, manage waiting queue of tasks.

First part can exist without second part but inverse is not possible;
Interruptions et events

Interrupt handler is declared with function request_irq();
It is freed by free_irq();
Prototypes (<linux/sched.h>, <linux/interrupt.h>):

- int request_irq (unsigned int irq, irqreturn_t (*handler)(int, void *,
  struct pt_regs *), unsigned long flags /* SA_SHIRQ */ , const char
  *device, void *dev_id);
- void free_irq (unsigned int irq, void *dev_id);
Interruptions et events

When irq arises, function handler() is called; dev_id field is used to identify devices when IRQ is shared (SA_SHIRQ); It can be used to send specific structure to driver; List of already declared IRQ is available in /proc/interrupts; Returning code of request_irq must be IRQ_NONE or IRQ_HANDLED.
Interruptions et events

bottom-halves
Kernel functions used to manage asynchronous tasks;
Executed after every return of system call, exception or interrupt handler;
Two possible "implementations":
- tasklets: can run on different CPU but one instance at one time
- softirqs: can run on different CPU and many instances can work in parallel
Interruptions et events

Ex:

```c
void my_routine_bh(unsigned long)
{
/* ... bottom-half code ... */
}

DECLARE_TASKLET(ma_tasklet, ma_routine_bh, 0);

void my_handler_irq(int irq, void *dev_id,
                      struct pt_regs *regs)
{
    ...
    tasklet_schedule(& ma_tasklet);
    ...
}
```
A process can wait for a kernel event (ex: data);

Process is going in a waiting queue and release the CPU;

When the event arises, processes in the corresponding queue are awaking.
Manipulation methods on waiting queue are as follow:

- `wait_event_interruptible_timeout()` : put the running process in sleep mode, in interruptible way or not and with/without time out limit, sleeping process is put in waiting queue
Interruptible Implementation

'interruptible' version check if a signal has been sent to a process and return an error code in this case (-ERESTARTSYS).
This error code should be propagated as a return of the system call.
Initialisation of a waiting queue (type wait_queue_head_t):
  ▶ during declaration ⇒ DECLARE_WAIT_QUEUE_HEAD(), or
  ▶ during execution (runtime) ⇒ init_waitqueue_head().
Prototypes (<linux/wait.h>):
  ▶ DECLARE_WAIT_QUEUE_HEAD (name);
  ▶ void init_waitqueue_head (wait_queue_head_t *wq);
Prototypes (\textless \texttt{linux/sched.h}\textgreater , \textless \texttt{linux/wait.h}\textgreater ) :

- \texttt{void wait\_event (wait\_queue\_head\_t \*wq, condition)} ;
- \texttt{int wait\_event\_interruptible (wait\_queue\_head\_t \*wq, condition)} ;
- \texttt{int wait\_event\_interruptible\_timeout (wait\_queue\_head\_t \*wq, condition, long timeout)} : 
- \texttt{void wake\_up\_interruptible (wait\_queue\_head\_t \*wq)} :
- \texttt{void wake\_up\_interruptible\_nr (wait\_queue\_head\_t \*wq, int nr)} :
- \texttt{void wake\_up\_interruptible\_all (wait\_queue\_head\_t \*wq)} ;

Ludovic Saint-Bauzel (Polytech’Paris UPMC)
Example

Ex:

```c
static DECLARE_WAIT_QUEUE_HEAD(ma_file);

int function_reading (void *addresse) {
    ...
    ret = wait_event_interruptible(& ma_file, readb() != 0);
    if (ret <$ 0)
        return ret ;
    ...
}
void my_handler(int irq, void *priv, struct pt_regs *regs) {
    ...
    wake_up_interruptible(& my_file);
    ...
}
```
Layout

Introduction

Robotics Software Frameworks

OS: Tasks And Threads

OS Driver Programming

General overview
Kernel
Tools
User Space Mechanisms
Kernel/Hardware Interface
Driver (Module)
Manipulation
Basic Routines
Char Devices
Memory Management
Xenomai RTOS example

Ludovic Saint-Bauzel (Polytech’Paris UPMC)
Manipulation

Set of tools to manipulate modules ⇒ modutils (kernel 2.4) or module-init-tools (kernel 2.6):

▶ lsmod : listing of modules loaded in the kernel;
▶ ins,rmmod : manual insertion/extraction of a module in the kernel;
▶ modprobe : automatic insertion/extraction of a module in the kernel with dependency resolving (loading all needed modules);
Manipulation

- `modinfo`: general information about a module (ex: author, description, parameters...);
- `depmod`: generate a list of dependencies for a set of modules;

Modules are automatically loaded when it is needed according to the hotplug mechanism that runs `modprobe` in user mode.
Basic Macros

Descriptions and parameters managed by dedicated macros (<linux/module.h>) :

- MODULE_AUTHOR() ;
- MODULE_DESCRIPTION() ;
- MODULE_LICENSE() ;
- MODULE_PARM() ;
- MODULE_PARM_DESC().

In kernel 2.6, MODULE_PARM() is replaced by (<linux/moduleparam.h>) :

- module_param(name, type, perm) ;

Ex :
MODULE_PARM(my_integer, "i") ;
MODULE_PARM_DESC(my_integer, "My integer parameter") ;
Initialisation and release

Macros defined in file `<linux/init.h>`:

- `module_init()`: declaration of initialisation function (type `int foo(void);`);
- `module_exit()`: declaration of quitting function (type `void foo(void);`).

`MODULE` is a constant that allow the compiler to make the difference between module versus in kernel compilation (no use for you!).
The initialisation function must contain all necessary information and computation to initialise a module (ex: allocations, hardware initialisations, resources reservation...) ;
In the opposite, exiting function must undo all the things done by initialisation one (ex: freeing reserved memory, hardware deactivation, freeing resources...) ;

Implementation
Linux uses optimisations of GCC to free in memory some unused part of the code after they have been instantiated (<linux/init.h>):

- \_\_init: initialisation functions (freed after finishing module\_init());
- \_\_exit: exit functions (ignored if statically linked to the kernel);
- \_\_initdata: used data in initialisation functions (freed after finishing module\_init());
- \_\_exitdata: used data in exit functions (ignored if statically linked to the kernel);

Ex:

```c
int \_\_init my\_function\_init(void)
{
    /* ...code... */
}

module\_init(my\_function\_init);
```
Usage counter

Each module is associated to a usage counter;
Avoid that a module is unloaded while being used;
Counter modification is done when the module is called;
The module can be manipulated with functions described in (\texttt{\langle linux/module.h\rangle}) :

- \texttt{MOD\_INC\_USE\_COUNT} : increase the usage counter;
- \texttt{MOD\_DEC\_USE\_COUNT} : decrease the usage counter;
- \texttt{MOD\_IN\_USE} : bring the value of the usage counter.

Kernel 2.6 provides instead (\texttt{\langle linux/module.h\rangle}) :

- \texttt{try\_module\_get(THIS\_MODULE)} : try to increase the usage counter.
- \texttt{module\_put(THIS\_MODULE)} : decrease the usage counter.
- \texttt{module\_refcount(THIS\_MODULE)} : bring the value of the usage counter.
Symbols

Exported symbols by the kernel are available in /proc/ksyms (/proc/kallsyms in 2.6);

Used by insmod to do dynamical linking;

Most of the symbols are exported in kernel/ksyms.c;

Modules can declare new symbols according to macro EXPORT_SYMBOL() (<linux/module.h>);

ksyms tool give a more precise and readable view of /proc/ksyms.
Integration to kernel sources

Choose an appropriate directory in kernel tree;

Create a sub-tree if necessary (ex: many elements);

In 2.4, edit Config.in and Makefile files of upper directory;

En 2.6, edit Kconfig et Makefile files of upper directory;

Add an entry inspired by the other;

Add a help entry in Documentation/Configure.help (2.6 only, in 2.6 content is in Kconfig);

Syntax is defined in Documentation/kbuild/...;
Implementation basic routines

In kernel 2.4, while a statical compilation, parameters of a module must be defined by macro \_\_setup() (in <linux/init.h>) :
void \_\_setup (char *append, int (*extract) (char *)) ;

In kernel 2.6, module\_param() automatically do this, parameters are directly given as inline parameter with :

module.param=valuer
Practical

Creation of the simplest module (HelloWorld);
Add some parameters to this module;
Link it to the kernel in a static point of view.
You can access to drivers with the use of specials files (device nodes also called inode);
A structure file_operations provide a way to define any entry point (<linux/fs.h>);
Access classic functions can be overloaded by driver own access functions;
Structure Associated

Initialisation of structure variables is done with the new C syntax (.var = value) (in old times a specificity of GCC) Some not implemented methods are replaced by default functions (ex : open(), close()) Others not implemented functions return -EINVAL ;
Ex :

```c
static struct file_operations mondriver_fops = {
    .owner = THIS_MODULE ,
    .read = mondriver_read ,
    .write = mondriver_write ,
    .open = mondriver_open ,
    .release = mondriver_release }
```

Majors et minors

Special files are created with command mknod;
A *inode* file is described by a unique identifier (*i_rdev*, of type *kdev_t*, size 16 bits in 2.4, 32 bits in 2.6) separated in two numbers (8+8 / 12+20):

- **major**: identifier of the driver kind (ex: IDE, /dev/hd* ⇒ Major=3),
- **minor**: identifier of one driver in a set (ex: IDE, /dev/hda1 ⇒ minor=1).

Majors 0 to 255 are reserved;
Majors and minors

List of majors and minors already used is updated in Documentation/devices.txt and `<linux/major.h>`;
Macros MAJOR() and MINOR() allow you to know what are these information of your inode;
Ex:

```c
static int mondriver_open(struct inode *inode,
struct file *file)
{
unsigned int minor = MINOR(inode->i_rdev);
/* or file->f_dentry->d_inode->i_rdev */
}
```

Kernel 2.6 have also functions : imajor(inode) and iminor(inode)
Majors and minors

Classical use, a major describes one driver for a specific device; Minors are used to identify different devices of the same kind or controlled sub-set; Can manage same major identical cards with this habit:

- structures that describes each device (ex: buffers, mutex...);
- array of structures indexed with minors;
- n files in /dev with the same major and different minors.
Returning Value

Negative means error defined in `<linux/errno.h>` and `<asm/errno.h>`; Positive or zero means working; This standard is used in every functions in the kernel; Error values transmitted between each functions;
Returning Value

Ex : invalid argument

```c
if (arg > 3) return -EINVAL;
```

Ex : error transmission

```c
int ret;
ret = fonction_noyau();
if (ret < 0) return ret;
```
Data Transfer kernel/user

Kernel uses physical or virtual memory;

Processes use their own virtual memory space;

Need to transfer data from/to kernel memory space to/from calling process memory space;

Main functions are:

- `get, put_user()` : transfer a variable from/to user memory space (using with lvalue), and
- `copy_{from,to}_user()` : transfer a buffer from/to user memory space.
These functions are all calling function access_ok() that checks validity of a user buffer;
This call can be avoided if we used those functions prefixed by __ (not advised). Prototypes (<asm/uaccess.h>):

- int get_user (lvalue, addr);
- int put_user (expression, addr);
- void copy_{to,from} _user (unsigned long dest, unsigned long src, unsigned long len);
- int access_ok (int type, unsigned long addr, unsigned long size);
Motivations

- Linux based (GPL)
- Developing tools (Multi-Interfaces)
- Dynamic community (Forum and mailing list\(^1\))

\(^1\)Quality to be observed when we want to use a free tool in long term point of view
Main Structure

- Patched Kernel (ADEOS)
- HAL
- nucleus
- Multi-skins
  - RTAI,
  - VXWorks,
  - POSIX,
  - RTDM,
  - pSOS+,
  - VRTX,
  - uITRON ...
Architecture

syscall interface

Native  POSIX  VxWorks  pSOS  uTRON  VRTX

Real-time nucleus

HAL

Adeos
ADEOS - Virtual Domains

Interrupts & Traps

Highest Priority Domain X

Root Domain

Lowest Priority Domain Y

per-CPU Adeos pipeline

Linux Kernel
ADEOS - Applied to Xenomai

per-CPU Adeos pipeline

- Interrupts & Traps
- Xenomai (primary)
- Interrupt shield
- Linux (secondary)
ADEVOS - HAL to manage communications

```c
void realtime_eth_handler (unsigned irq, void *cookie)
{
    rthal_irq_enable(irq);
    rthal_irq_host_pend(irq);
    /* This irq has been marked as pending for Linux */
}

void linux_eth_handler (int irq, void *dev_id, 
                        struct pt_regs *regs)
{
    /* process the IRQ normally. */
}
```
NATIVE API - Main

- Compactness: <100 services
- Orthogonality:
  - developer memory :)
  - Non-ambiguity
  - More functions combination than more complex functions
- Context independent:
  - Kernel Space
  - User space: privileged
- Small Semantic for localisation: Same names
  - Kernel (ex. xeno_native.ko),
  - User (ex. libnative.so)
NATIVE API - Services

- Tasks Management
- Time Service
- Synchronisation tools
- Messages and communication
- Control I/O
- Registers
NATIVE API - Services

- Tasks Management
  - Increasing Priorities 1-99
- Time Service
  - Timers,
  - Time,
  - **Alarm**…
- Synchronisation tools
  - Counting Semaphores,
  - Mutex,
  - Conditional Variables,
  - Event Flags
NATIVE API - Services

- Message and communication
  - Inter-tasks synchronous Messages
  - Messages queue
  - Pipes: Message streaming
- I/O control
  - Interruptions
  - Heap memory
  - Memory Access of a device in user space
- Registers
  - Small semantic, developer basis
  - /proc/xenomai/registry/...
NATIVE API - Compilation Flags

CFLAGS=\$(shell xeno-config --xeno-cflags)
LDFLAGS= -lnative \$(shell xeno-config --xeno-ldflags)

myapp: myapp.c
  \$(CC) -o $@ myapp.c \$(CFLAGS) \$(LDFLAGS)
Common Functions

- Create / Delete
- Bind / Unbind ⇒ Registry and binding
- Inquire ⇒ Inquiries
Registry and binding

Multiple processes unified sharing of objects.
In Kernel Space and in User Space (T_PRIMARY).

- 
  
  - "/proc/xenomai/registry"
  
  - \( \text{int rt\_object\_bind} (\text{RT\_OBJECT \*obj, const char \*name, RTIME timeout}) \)
  
  - \( \text{static int rt\_object\_unbind} (\text{RT\_OBJECT \*obj}) \)

Ex:

```c
main1.c:
RT_TASK t;
rt_task_create(&t,"myTask",0,0,0);

main2.c:
RT_TASK *tt;
rt_task_bind(tt, "myTask", (RTIME)10000);
... Execution ...
rt_task_unbind(tt);
```
Inquiries

State informations of any object.

- \textbf{int \texttt{rt\_object\_inquire} (RT\_OBJECT *obj, RT\_OBJECT\_INFO *info)}

Ex:

```c
...  
RT_TASK *curtask;  
RT_TASK_INFO curtaskinfo;  
curtask=rt_task_self();  
rt_task_inquire(curtask,&curtaskinfo);  
rt_printf("Task name : %s \n", curtaskinfo.name);  
...  
```
Introduces a collection of utilities aimed at forming a Real-Time Development Kit for userland usage.

```c
int rt_vfprintf(FILE *stream, const char *format, va_list args);
int rt_vprintf(const char *format, va_list args);
int rt_fprintf(FILE *stream, const char *format, ...);
int rt_printf(const char *format, ...);

int rt_print_init(size_t buffer_size, const char *name);
void rt_print_cleanup(void);
void rt_print_auto_init(int enable);
const char *rt_print_buffer_name(void);
```
Real Time Driver Model RTDM

- User API
- Driver Development API
  - Inter-Driver API
  - Device Registration Services + Synchronisation Services
  - Clock Services
  - Task Services
  - Timer Services
  - Synchronisation Services
  - Interrupt Management Services
  - Non-Real-Time Signalling Services
  - Utility Services

- User API
- Driver Development API
  - Inter-Driver API
  - Device Registration Services + Synchronisation Services
  - Clock Services
  - Task Services
  - Timer Services
  - Synchronisation Services
  - Interrupt Management Services
  - Non-Real-Time Signalling Services
  - Utility Services
Real Time Driver Model

- Device Profiles
  - CAN Devices
  - Serial Devices
  - Testing Devices
Introduction

Robotics Software Frameworks

OS : Tasks And Threads

OS Driver Programming

Application

Examples

Examples

Documentations
MC2e

- **Aim:**
  - Surgical Robotics
  - 3rd Hand
  - Minimal invasive operation
  - Co-manipulation

- **Technical notes:**
  - Spherical kinematics
  - Force based robot
MC2e: Tasks Architecture

Shared Time

User

UI

robotCmd

robotState

Constrained Time

Kernel

Asserv
MC2e : Synchro Solution

Only one shared memory with all the state and command information.
⇒ leads to only one synchronization mechanism.

Ludovic Saint-Bauzel (Polytech’Paris UPMC)  Robotics Real-Time Programming  2012-2013  312 / 334
Monimad: Description

- **Aim:**
  - Sit-To-Stand
  - Walking
  - Disbalance recover
  - Old and Diseased people
  - Assistance
Elements:

- CAN Bus
- Force Sensors (Serial, Acquisition card)
- UI
Monimad : Synchro Solutions

Shared Time

- UI : Trajectory, Decision
- Com
- Force Sensor Driver
- Shared Memory
- Mutex

Constrained Time

- Kernel
- MPC555 : Pos Asserv
- Char Device
- CAN Bus
- Shared Structure
Cardioloc : Description

▶ Aim :
  ▶ Heart Surgery
  ▶ Compensation
  ▶ Visual Servoying

▶ technical information :
  ▶ 2000Hz
  ▶ PiezoElectric Actuator
  ▶ High frequency camera
Cardioloc : Tasks Architecture

Elements :

- GUI
- Camera Driver
- Threads
Cardiocolc : Synchro Solutions

Shared Time | Constrained Time

UI

Com

User

Camera Driver

Fifo

Shared Structure

Asserv

Shared Variable

Mutex

Kernel

RobotCmd

robotState

getImageCMD

image

robotCmd

robotState

getimageCMD

robotCmd

robotState
RobModex: Description

- **Aim:**
  - Pedagogic Solution
  - Polytechnique School of Engineering (Palaiseau)

- **Technical**
  - Fixed Wheel Structure
  - Usb Camera
RobModex : Tasks

Shared Time

- User
- Kernel
- UI
- Asserv

Constrained Time

- robotStartStopCmd

Elements :
- USB Driver
- Threads
RobModex : Synchro Solution

Notice :
There is no Synchronization needed
Introduction

Robotics Software Frameworks

OS : Tasks And Threads

OS Driver Programming

Application

Documentations

Doc

Interface with community
References
Licences
The Documentation/ directory in kernel sources;
The documentation Linux project

- LDP "http://www.tldp.org": guides, HOWTOs, FAQs, manpages...;

Personal web sites of maintainers (see newsgroups);
Other documents from web (http://www.google.com/linux/).
Information

Specialised Mailing Lists (newsgroups http://www.uwsg.indiana.edu/search/):
- linux-kernel,
- kernel-newbies,
- linux-net...

Week summary of discussions of linux-kernel mailing list:
- Kernel Traffic (http://kt.zork.net/kernel-traffic/), and
- Kernel Notes (http://www.kernelnotes.org).
Interface with community

FAQ Linux kernel (http://www.tux.org/lkml/); Discussion Forums: comp.os.development.kernel...; Personal email maintainers or authors of a part of source code.
References

Use the source, Luke!
The answer of all your questions are inside the kernel sources...
References

Books - Linux Device Drivers 2nd edition (O’Reilly)
http://www.xml.com/ldd/chapter/book/index.html:
  ▶ Understanding the Linux Kernel (O’Reilly):
  ▶ "Programmation Linux 2.0 "(Eyrolles).
References

E-Books - Linux Kernel Internals (kernel 2.4) [http://www.tldp.org/LDP/lki/]:

- Linux Kernel Module Programming Guide (kernels 2.0 et 2.2) [http://www.tldp.org/LDP/lkmpg/]:
- Linux Kernel Hackers’ Guide (kernels 2.0 et 2.2) [http://www.tldp.org/LDP/khg/]:
- The Linux Kernel (kernel 2.0) [http://www.tldp.org/LDP/tlk/]:
- The Linux Programmer’s Guide (API user space) [http://www.tldp.org/LDP/lpg/].
Articles and documents

- Articles on how import drivers from 2.4 to 2.6
  [http://lwn.net/Articles/driver-porting/]
- Links of kernelnewbies [http://www.kernelnewbies.org/links/]:
- Articles of kerneltrap [http://kerneltrap.org/]:
- Conceptual architecture of Linux Kernel
  [http://plg.uwaterloo.ca/~itbowman/CS746G/a1/]:
- Concrete architecture of Linux kernel
  [http://plg.uwaterloo.ca/~itbowman/CS746G/a2/].
References

- http://www.xenomai.org
  - Native-API-Tour-rev-C.pdf
  - Life-with-Adeos-rev-B.pdf
  - RTDM-and-Applications.pdf
GNU General Public License (GPL) Maintained et defended by Free Software Foundation (FSF);
Give the right (and it is also a duty !) to copy, modify et redistribute the software only under the same licence terms (GPL).
This rule includes also software linked to codes licensed under GPL;
No free of charge are described, only freedom in using;
It is possible to sell a software under GPL but not to forbid them to modify it and redistribute (free of charge or not).
LGPL

- GNU Lesser General Public License (LGPL) Maintained and defended by Free Software Foundation (FSF);
- Less restricted than GPL; Possibility to call LGPL code from a non-free program (ex: software under licence incompatible with GPL);
- Used mainly for libraries (ex: GNU libC).
Licence of Linux kernel is GPL with added terms;
Core of the kernel is GPL;
But using and redistribution of binaries only inside are allowed;
Must fit with standard interfaces of modules;
A lot of work has been done to clarify the GPL/non GPL frontier: EXPORT_SYMBOL_GPL, MODULE_LICENSE...
Licences

Technically very difficult to provide compatible binary modules because of all possible configurations of the kernel (ex: versions, external patches...); Legal but not advised (no support from community).