UNIVERSITÀ DEGLI STUDI FIRENZE
DINFO
Dipartimento di Ingegneria dell'Informazione

Software Engineering for Embedded Systems

Real-time operating systems

Laura Carnevali

Software Technologies Lab
Department of Information Engineering
University of Florence
`http://stlab.dinfo.unifi.it/carnevali`
laura.carnevali@unifi.it

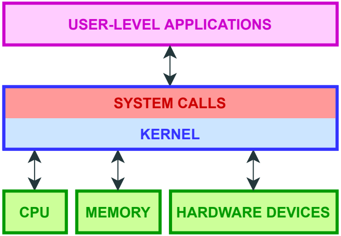
Outline

1. Standards for RTOSs
2. VxWorks
3. Introducing RT programming: recap on the c language
4. Credits & References

Standards for RTOSs

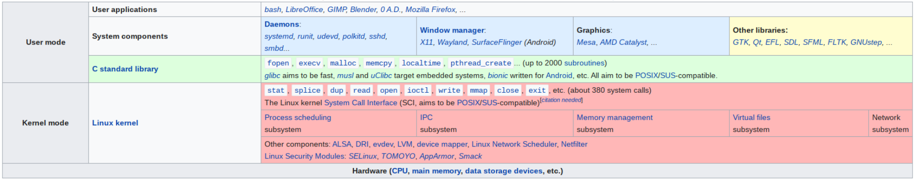
Operating systems: kernel & system calls

- The **kernel** is the core of the Operating System (OS)
  - It is the portion of the OS code that is always resident in memory
  - It acts as an interface between user-level applications and hardware resources
- It must not be confused with the Basic Input/Output System (BIOS), the built-in core processor software responsible for booting up the system
- It provides the **system calls** allowing user-level programs access OS services
  - They provide the services of the OS to use-level applications
  - They comprise an Application Programming Interface (API)
  - They are the only entry-point into the kernel



Memory: kernel space vs user space

- **Kernel space**: memory area where the kernel code is stored and executed
  - The kernel has access to all of the memory (both kernel space and user space)
  - The **kernel mode** is a privileged CPU operating mode
- **User space**: memory area where the user-level code is stored and executed
  - The user-level programs have access to the user space only
  - The **user mode** is non-privileged CPU operating mode for user-level programs
  - The user-level programs can access a limited kernel part via the system calls
  - If a user-level program invokes a system call, a **software interrupt** is sent to the kernel which runs the appropriate **interrupt handler** (in kernel mode) and then returns the control to the user-level program (executed in use mode)



User mode	**User applications**	*bash*, *LibreOffice*, *GIMP*, *Blender*, *0 A.D.*, *Mozilla Firefox*, ...				
	System components	**Daemons**: *systemd*, *runit*, *udevd*, *polkitd*, *sshd*, *smbd*...	**Window manager**: *X11*, *Wayland*, *SurfaceFlinger* (Android)	**Graphics**: *Mesa*, *AMD Catalyst*, ...	**Other libraries:** *GTK*, *Qt*, *EFL*, *SDL*, *SFML*, *FLTK*, *GNUstep*, ...	
	C standard library	`fopen`, `execv`, `malloc`, `memcpy`, `localtime`, `pthread_create`... (up to 2000 subroutines) *glibc* aims to be fast, *musl* and *uClibc* target embedded systems, *bionic* written for Android, etc. All aim to be POSIX/SUS-compatible.				
Kernel mode	**Linux kernel**	`stat`, `splice`, `dup`, `read`, `open`, `ioctl`, `write`, `mmap`, `close`, `exit`, etc. (about 380 system calls) The Linux kernel System Call Interface (SCI, aims to be POSIX/SUS-compatible)[citation needed]				
		Process scheduling subsystem	IPC subsystem	Memory management subsystem	Virtual files subsystem	Network subsystem
		Other components: ALSA, DRI, evdev, LVM, device mapper, Linux Network Scheduler, Netfilter Linux Security Modules: *SELinux*, *TOMOYO*, *AppArmor*, *Smack*				
Hardware (CPU, main memory, data storage devices, etc.)						

Image from `https://en.wikipedia.org/wiki/User_space`

Real-time operating systems

- **General-purpose OS (GPOS)**: an OS not suitable for RT applications
- **Real-Time Operating System (RTOS)**: OS suitable for RT applications
  - **Hard** RTOS: missing a deadline may have catastrophic effects on the system (manages hard RT tasks, with reaction time in the order of 1 ms or less)
  - **Soft** RTOS: missing a deadline results in degraded performance (manages soft RT tasks, with reaction time in the order of $\leq$ 100 ms or less)
- Main features of an RTOS
  - **Predictability**
  - Determinism
  - High performance
  - Safety and security features
  - Priority-based scheduling
  - Small footprint

Standards for GPOSs and RTOSs

- They define (syntax and semantics of) the system calls
- They provide **portability** of applications from one platform to another
  - Promote competition among kernel providers ⇒ Increase quality of platforms
  - Portability is specified at the source-code level ⇒ Recompile for each platform
- Main standards for GPOSs and RTOSs
  - POSIX
  - RT-POSIX
  - OSEK/VDX
  - ARINC-APEX
  - μITRON

POSIX (Portable Operating System Interfacte for UniX)

- Family of standards intended to provide portability of applications at source code level and thus maintain compatibility between OSs
- Developed by the Portable Applications Standards Committee (PSSC) of the IEEE Computer Society and formally termed **IEEE Std 1003**
  - IEEE Std 1003.1, also termed **POSIX.1**: core services
  - IEEE Std 1003.1b, also termed **POSIX.1b**: **real-time extensions** (priority scheduling, clocks and timers, semaphores, . . .)
  - IEEE Std 1003.1c, also termed **POSIX.1c**: thread extensions
  - IEEE Std 1003.2, also termed **POSIX.2**: shell and utilities
  - . . .
- Supports many different **levels of compliance**
- Supports the paradigm **“program by contract”**

RT-POSIX

- Real-time extension of POSIX, enabling portability of real-time applications
- Provides services for **concurrent programming** and **time predictability**
  - Mutual exclusion synchronization through priority inheritance
  - Prioritized message queues for inter-task communication
  - Fixed-priority preemptive scheduling
  - ...
- **Real-time profiles** defined by **POSIX.13**
  - **Minimal Real-Time System profile (PSE51)**: for small embedded systems; it supports threads but not processes; it provides I/O via predefined device files
  - **Real-Time Controller profile (PSE52)**: for small embedded systems; it extends PSE51 with support for a simplified file system
  - **Dedicated Real-Time System profile (PSE53)**: for large embedded systems; it extends PSE52 with support for multiple processes with protection system
  - **Multi-Purpose Real-Time System profile (PSE54)**: for general purpose systems; it supports applications made of real-time and non-real-time tasks

Other standards

- OSEK/VDX: standard for automotive application software
  - Jointly developed by many automotive industries
  - Enables portability and reusability of distributed control software in vehicles
- ARINC-APEX: standard for avionics application software
  - It describes avionics, cabin systems, protocols, and interfaces used by more than 10 000 air transport and business aircraft worldwide
  - Physical memory is divided into partitions, each allocated to an application and temporally isolated from the other partitions
  - Communication between processes in different partitions is performed through message passing over logical ports and physical channels
- μITRON: family of standards for embedded application software
  - It aims at maximizing portability while maintaining scalability (e.g., improving portability of interrupt handlers while limiting overhead)

Types of RTOS

- **Commercial** RTOSs
  - VxWorks
  - QNX
  - Neutrino
  - OSE
  - ...
- **Linux-related** RTOSs
  - RTLinux
  - RTAI
  - ...
- **Open-source real-time research** RTOSs
  - SHARK
  - MaRTE
  - ERIKA
  - ...

VxWorks

Main features of VxWorks

- Developed by WindRiver (headquarters in Alameda, CA, USA)
- 32/64 bits on Arm/Intel/MIPS/PowerPC
- Proprietary RTOS, POSIX PSE52
- Kernel/user space separation, user space optional
- C/C++11/14, possible to develop kernel C++ modules and user applications
- Safety certifiable: DO-178, ISO 26262, IEC 61508
- Proprietary build system
- Kernel shell
- Eclipse-based IDE, Windows/Linux hosts

Examples of industry applications using VxWorks

COMAU
KUKA
SIEMENS
MITSUBISHI ELECTRIC
ABB
NASA
YASKAWA
NXC100 robot controller offers high performance, open communication, and integrated cell control
NXC100
ROBOT CONTROLLER
BOSCH
Schneider Electric
Honda Robotics

Image from the presentation “ROS2 on VxWorks” by A. Kholodnyi at the ROS-Industrial Conference 2019

Clock and scheduling

- The **kernel clock** determines the resolution of scheduling actions
  - The kernel clock has default frequency of 60 Hz
  - The minimum and maximum of the kernel frequency depend on the hardware
- VxWorks supports:
  - Priority-based preemptive scheduling
  - Round-robin scheduling
  - Up to 256 priority levels

Inter-task communication

- VxWorks supports:
    - Shared memory among tasks (similar to threads)
    - Binary semaphore and counting semaphores
    - Mutexes (POSIX interfaces)
    - Message queues and pipes
    - Sockets and Remote Procedure Calls (RPCs)
    - Signals
- Mutex semaphores support the Priority Inheritance Protocol (PIP)

Introducing RT programming: recap on the c language

The c programming language

- A c program specifies a computation through 3 constructs:
    - Variables
    - Expressions
    - Statements

1/3: Variables

- A **Variable** holds a **Value** of some **Type**
  - The value varies along the computation
  - The type remains unchanged
- A Type defines a set of Values and a set of Operations
  - Predefined basic Types: int, float, char, . . . with modifiers
  - Pointers-to and arrays-of
  - User-defined Types: struct (see later)
- A Variable has a Lifecycle
  - Declaration
  - Reference
  - For user-defined types only, there is also a Definition (see later)
- The identity of a Variable is resolved as the address

1/3: Variables

- A **Declaration** introduces a Variable
  - A Type, a location in memory, a name
- Declaration applies not only to variables: the role of semantic modifiers
  - The rule: start from the name, first move right, and then move left

```
int a; // a is an int
int* ptr; // ptr is a pointer to an int
int A[64]; // A is an array of 64 int
int f(void); // f is a function returning an int
```

- The point of Declaration identifies the context
  - Scope of visibility of the name
  - Life time of the declared entity

1/3: Variables

- User-defined Types
  - Defined by aggregation of multiple declarations . . .
  - . . . with possible recursion through pointers

```
struct list {
    int value;
    struct list * nextPtr;
};
```

1/3: Variables

- A Reference identifies a (declared) Variable
  - By name

    ```
    int a;
    a = 7;
    ...
    ```

  - By dereferencing a an expression that returns an address:

    ```
    int* ptr;
    ...
    *ptr = 7;
    *(ptr+1) = 14;
    ...
    ```

  - By arithmetic offset over an Array (address)

    ```
    int A[64];
    A[0] = 14;
    ...
    ```

  - Don't mess Variables and References to Variables!

2/3: Expressions

- Syntax of an **Expression**:
  an Expression combines Constants and references to Variables by Operators
- The semantics of an Expression consists of 2 elements:
  - An Expression returns a value (of some Type)
  - An Expression produces side effects on Variables
- The two aspects of semantics are more or less intuitive
  - A few examples (assume that `x` holds 2 before each expression is evaluated)

Expression	Returned value	Side effects	Remarks
`x+3`	5		
`x<=3`	1		no Boolean
`x>=3`	0		
`x=3`	3	`x`<−3	= is an Operator
`x++`	2	`x`<−3	
`++x`	3	`x`<−3	

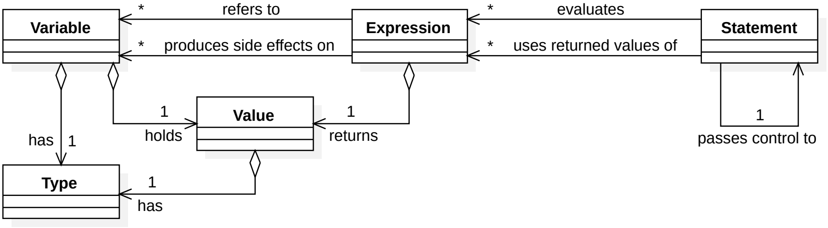
- Don't mess side effects and returned values

2/3: Expressions - Functions

- Functions are a kind of Expression
  - Syntax: they are a combination of a Constant (the name) with Values returned by Expressions (the actual parameters) by an Operator ((...))
  - Semantics: they return a value and produce side effects
- In the syntax perspective, `f(x,3)` is somehow like `x+3`
  - But, the returned value and the side effects remain hidden
  - And, the operation semantics is user defined (leading to Function definition)
- More on functions later

3/3: Statements

- A **Statement** specifies 2 aspects:
  - The Expressions to be evaluated
  - The next Statement to be executed, which may depend on Values returned by Expressions



- An example:

```
int count, sum, A[64];
for(count=0,sum=0; count<64; count++)
    sum+=A[count];
printf("%f", sum);
```

3/3: Statements

- Statements are:
  - `Expr;`

    e.g. x=3;
  - `Statement1 Statement2`

    e.g. x=3; y=x;
  - `Statement1 Statement2`

    e.g. x=3; y=x;
  - `if(Expr) Statement`
  - `for(Expr1; Expr2; Expr3) Statement`
  - `while(Expr) Statement`
  - `do Statement while(Expr);`
  - `return Expr;`
  - `break;`
  - `goto label;`
  - `switch(Expr)case const:  Statement ...`
- Remark: statements and declarations have similar syntax

3/3: Statements - More on Functions

- A function has a lifecycle
  - Definition (being a kind of user defined operator)
  - Declaration (to introduce its name in a context)
  - Invocation (to let it be evaluated)
- Function Definition specifies
  - Specifies returned type, name, formal parameter declarations list, body
  - The body may include among others:
    local variable declarations, return statements,
    further function invocations
    int add(int x, int y) int z; z=x+y; return z;
- Function Declaration (also known as prototype)
  - Introduces the function name, returned type, and signature in a context (usually global), to make the function referrable
  - int add(int x, int y);

3/3: Statements - More on Functions

- A Function invocation is a kind of Expression
  - Made of the address where the function is stored
    followed by the list of actual parameters
  - The function address is usually specified by the declared name
  - The actual parameters are expressions
    returning a value for each formal parameter in the definition
    ... add(x+3,y) ...
- Invocation and parameters binding
  - For each formal parameter, a Variable is created (on the stack)
    and initialized with the Value of the corresponding actual parameter
    (which is termed binding by value)
  - Local Variables declared in the body are allocated,
    and body Statements are executed,
    until exhaustion of the code or reach of a return Statement
- Expression semantics
  - Returned value: the value returned by the expression on return
  - Side effects: those produced in the body execution

The c language summary - 1/3

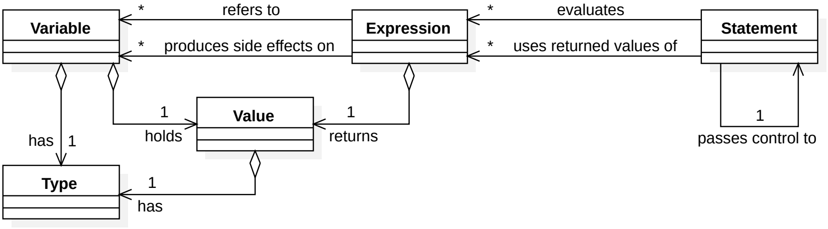
- A serialized story (a resonable roadmap to learn the language)
  - Types, values, and constants
  - Variables
  - Expressions, side-effects and returned values
  - Pointers (a kind of variable)
  - Arrays (another kind of variable), static or dynamic allocation
  - Functions (a kind of expression), binding technique
  - Statements
  - Structured types (a kind of user-defined type)

The c language summary - 2/3

- The real story behind
  - Types and Values
    - Elementary types
    - User-defined Types (struct); Type definition
  - Variables
    - Declaration and reference
    - Pointers; Arrays; static or dynamic allocation
  - Expressions
    - Operators; side-effects and returned values
    - Functions; binding technique; definition, declaration, and reference
  - Statements
    - Program structure

The c language summary - 3/3

- A mechanism with 3 parties
  - Variables encode Values (of some Type)
  - Expressions return a Value and produce side effects on Variables; Variables affect returned Values and side-effects
  - Statements control the flow of execution of Expressions; Values returned by Expressions affect Statements



Credits & References

Credits

- The slides of the section titled “Introducing RT programming: recap on the c language” are taken from the slides of the lectures of this course given by Prof. Enrico Vicario in the A.Y. 2019/2020
- These slides are authorized for personal use only
- Any other use, redistribution, and profit sale of the slides (in any form) requires the consent of the copyright owners

References

- Giorgio Buttazzo, “Hard Real-Time Computing Systems - Predictable Scheduling Algorithms and Applications”, Third Edition, Springer, 2011
    - Chapters 1, 2, 12
- Stefano Berretti, Laura Carnevali, Enrico Vicario, “Fondamenti di Programmazione: linguaggio c, strutture dati e algoritmi elementari, c++”, Società Editrice Esculapio, Bologna, 2017
    - Chapter 1