Real-Time Programming Languages

Ada: Features for Real-Time and Safety

Martin Becker

TU München
Institute for Real-Time Computer Systems (RCS)

December 15, 2015

Lehrstuhl für
Realzeit-Computersysteme

Technische Universität München

Outline

Section 1

Recap: Processes and Scheduling

Process vs. Thread (1)

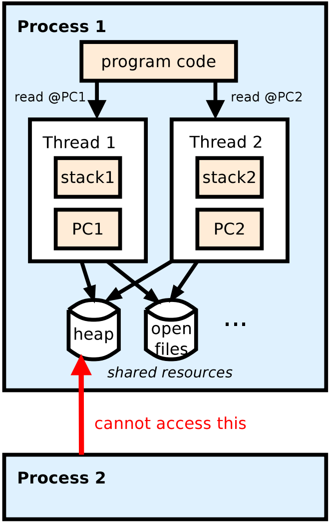
General Concept

Process

- one instance of your program code
- has its own memory space and resources (heap, open files, ...)
- consists of at least one *thread*
- other processes cannot access its resources

Thread

- a flow of execution ⇒ has its own program counter (PC) and stack
- runs in parallel to other threads



Remark: *Inter-Process Communication* for resource sharing between processes.

Process vs. Thread (2)

Implementation in Linux

- in Linux kernel **everything is a process**
- threads are *light-weight processes*
  - = a process sharing resources with other light-weight process(es)
- every process (light-weight or not), has its own PCB:
  - stack, program counter, ...
- processes can have children
  - can be light-weight or normal processes
  - in Linux, the “root” process is called `init`
    - everything starts here
    - try `pstree -p 1`

Process vs. Thread (3)

Demo: creating a process (see `process.c` on Moodle)

Build+run `process-demo`:

```
$ gcc process.c o pdemo
$ ./pdemo

P: my process id is 14635
P: created child
P: created child
C: changed the variable to: 42
C: closed the file.
C: paused (press key)
C: exit
P: The variable is now: 9
P: Read from the file: t
P: exit
```

Inspect process tree:

```
$ pstree -p 14635
pdemo(14635)---pdemo(14636)
```

Process has created a new child process.

This creates a new (heavy-weight) process, which is completely independent of the caller.

Process vs. Thread (4)

Demo: creating a thread (see `thread.c` on Moodle)

Build+run `thread-demo`:

```
$ gcc thread.c o tdemo
$ ./tdemo

P: my process id is 23375
P: created child
P: paused (press key)
C: my process id is 23376
C: changed the variable to: 42
C: closed the file.
C: paused (press key)
P: The variable is now: 42
P: READ failed: Bad file
     descriptor
```

Inspect process tree:

```
$ pstree -p 23375
tdemo(23375)---tdemo(23376)
```

For Linux this is just another process.

This creates a new (light-weight) process, which shares resources with the caller.

Process vs. Thread (4)

Demo: creating a thread (see `thread.c` on Moodle)

Build+run `thread-demo`:

```
$ gcc thread.c o tdemo
$ ./tdemo

P: my process id is 23375
P: created child
P: paused (press key)
C: my process id is 23376
C: changed the variable to: 42
C: closed the file.
C: paused (press key)
P: The variable is now: 42
P: READ failed: Bad file
     descriptor
```

Inspect process tree:

```
$ pstree -p 23375
tdemo(23375)---tdemo(23376)
```

For Linux this is just another process.

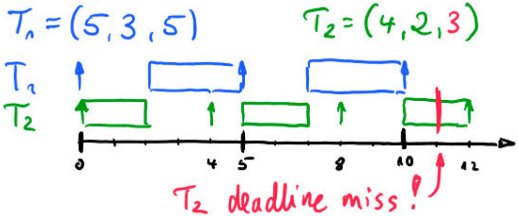
This creates a new (light-weight) process, which shares resources with the caller. **Remark:** The example uses a low-level API of Linux. Usually new threads are not created like this, but with the *pthread* library.

Process vs. Thread (5)

Summary:
In Linux everything is a process. We only need to deal with how to schedule processes.

Schedulability Analysis

- processes have to take turns on the CPU
- OS decides how they take turns, using a *scheduling policy*
  - round-robin, priority-based, earliest-deadline first, etc.
- in real-time systems, processes usually have *deadlines* - if they do not finish computing by then, something bad might happen
- **schedulability analysis** is a method to determine, whether all processes can keep their deadlines



task := (P, e, D)
P = period (release time)
e = WCET
D = relative deadline

Context Switch

scheduling = switching between processes

- when suspending one process and waking up another, scheduler performs a *context switch*:
  1. pause the currently running process
  2. save the state of the retiring process to PCB:
     - PC, registers, I/O status, privileges, open files, ...
  3. load state of the waking process from its PCB
  4. resume the waking process

PCB in Linux (2% of it)

```
struct task_struct {
volatile long state;
void *stack;
        // ...
int on_cpu;
int prio;
pid_t pid;
u64 start_time; /* monotonic time in nsec */
/* open file information */
struct files_struct *files;
/* signal handlers */
struct signal_struct *signal;
/* CPU-specific state of this task */
struct thread_struct thread;
};
```

(https://github.com/torvalds/linux/blob/master/include/linux/sched.h)

Context Switch (2)

- switching takes time
  - several μs per switch, depending on data size (cache!)
  - several hundrets of switches per second (`grep ctxt /proc/<pid>/status`), depending on the scheduler settings
- switching between threads is much faster than switching between processes
- an incoming interrupt also causes a context switch

Usually...

...one would talk about *mutex* and *semaphore*

- these are mechanisms provided by the OS
- they allow to synchronize threads
  - avoid race conditions on shared data
  - ...see Lecture 7

Usually...

...one would talk about *mutex* and *semaphore*

- these are mechanisms provided by the OS
- they allow to synchronize threads
  - avoid race conditions on shared data
  - ...see Lecture 7

... but in Ada we have more high-level mechanisms which replace these

- internally, these may build on mutex and semaphore
- you may still use mutex and semaphore directly, if you insist (‘‘Florist’’ library)

Section 2

Ada’s Real-Time & Safety Features

Ada features

- *certified* Ada compilers are available since 1983
- before Ada 95 the compilers had problems with tasking and efficiency
- since Ada 95, most innovations fall into the real-time area to fix these:
  - better concurrency model
  - new dispatching policies (such as non-preemptive, round-robin, EDF)
  - Timing Events
  - monitoring task execution times
  - high-integrity profiles, such as *Ravenscar*
- **today:** Ada is used for highly safety-critical systems

Famous users of Ada

Famous users:

- Air Traffic Management: DFS
- Aviation: Boeing 777 (99.9%!)
- Railway: TGV
- Rockets: Ariane
- Satellites: INMARSAT
- Banking: Reuters
- Medical: JEOL Nuclear MRI
- Military: Eurofighter combat aircraft
- . . .

Why projects decide for Ada

User Reports:

- once it compiles, it works $\Rightarrow$ less development time in total (more coding, much less debugging time)
- harder to mix different data types (units, casts, etc)
- programmers have to think harder, which yields better code

Moreover (today's lecture)

- amenable to static analysis $\Rightarrow$ reduce testing effort
- small memory footprint
- Ada compilers usually have to pass a huge test suite (''ACVC'', > 3,800 tests), whereas almost all C compilers (including gcc) are known to have hundreds of bugs

Exceptions

In case of an unexpected program state

1. the control flow of the causing thread is interrupted
2. the control flow is handed over to an *exception handler*
3. if there is no handler, the thread terminates

Typical exceptions:

- `Constraint_Error`: exceeding variable's type bounds
- `Program_Error`:
- ...

Exceptions (2)

This causes an exception at line 5:

```
procedure main is
   type mydays is Integer range 1 .. 31
   d : mydays := 31;
begin
5  d := d + 1; -- raises an exception
   Put_Line("day is:" & d'img); -- never reached
end main;
```

Some exceptions are foreseen by the compiler. Others are not. To handle them:

```
procedure main is
   type mydays is Integer range 1 .. 31
   d : mydays := 31;
begin
   d := d + 1; -- raises an exception
   Put_Line("day is:" & d'img); -- never reached
exception
   -- control flow continues here
   -- do something to recover
end main;
```

Exceptions (3)

- some types of exceptions can be turned off by pragmas, e.g., Integer Overflow
- how are they implemented?
  - the compiler inserts checks into the program
  - if checks fail, programm jumps to the exception handler
  - **therefore, exceptions consume memory space and execution time**
- in embedded systems: no space and no time
  - prove absence of exceptions
  - turn those off that can be proven
  - handle the remainder
    - Ada: `last_chance_handler`
    - one exception handler, global for the entire program
    - ‘‘last chance’’ before program terminates

Real-Time Annex (since Ada95)

```
with Ada.Real_Time;
```

- Ada implementations offering above package must comply to that annex (GNAT does, but coverage is platform-dependent)
- additional semantics, such as
  - integrated priority-based interrupt handling
  - deterministic scheduling via fixed and dynamic priority
  - preemptive scheduling and monitoring execution time
  - forbidden: `terminate, abort, ...`
  - protection against priority inversion (...Pathfinder Mars rover...)
  - minimizes risk of race conditions (...Therac-25 medical radiation machine...)
  - multi-processor support
  - ...

Real-Time Annex (2)

Interesting:

- `pragma Priority(...)`: Tasks can be given priorities
- `pragma Queuing_Policy(...)`: entry queues can support *priorities* on top of FIFO
- `pragma Max_Task_Entries(...)`: limit queue length
- *hi-res timer*: as described earlier (`Clock, Time, Duration`)
  - RM: "real time is defined to be the physical time as observed in the external environment" $\Rightarrow$ scheduled programming model
  - implementation must document: upper bound on the size of a clock jump, rate drift, tick length, ...
- `Ada.Execution_Time.Timers`: get notified when task exceeds a certain execution time
- **clearly, if OS is used then it must be a Real-Time OS**

Ada Language

- Ada is a language with many features
- some features are hard to analyze (`select`...)
- the run-time libraries of Ada can incur high cost
- the language offers to ban features by putting *restriction pragmas*
  - a program that uses a restricted feature will not compile
  - unused features are removed from the run-time ⇒ smaller and faster program

To obtain efficient and analyzable programs for real-time systems, which features shall we turn off?

a village in Great Britain, location of 8th International Real-Time Ada Workshop (IRTAW), 1997.

Ravenscar

a village in Great Britain, location of 8th International Real-Time Ada Workshop (IRTAW), 1997.

- discussion at workshop about *what needs to be done in Ada, to target high-integrity, efficient real-time systems*
- participants agreed on a **list of forbidden features** in the Ada language, that hinder analysis of such systems $\Rightarrow$ The *Ravenscar* Profile

The *Ravenscar* Profile (since Ada 2005)

```
pragma Profile (Ravenscar);
```

- Ada can be complex (`select`, `terminate`, ...)
- this profile: constrain tasking features to an analyzable subset
  - forbids `select, ATC, requeue, dynamic prios ...`
  - this is why we do not look at those features in detail
- **only concerned with tasking features of Ada**, nothing else
- goals:
  - full determinism
  - easier schedulability analysis
  - memory boundedness

Ravenscar - Task Model

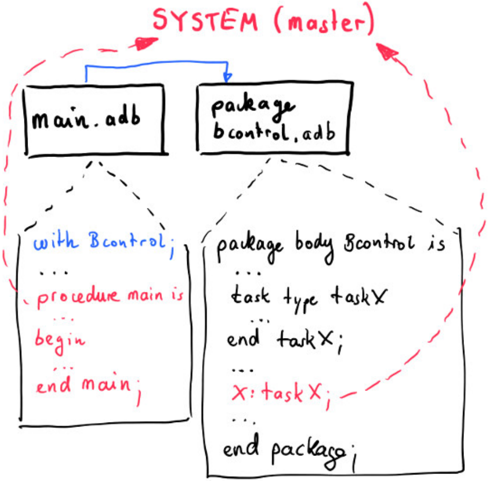
Tasks are either

1. *time-triggered* (periodic) or
   - released by a *single* `delay until`
2. *event-triggered* (sporadic)
   - released by a *single* protected *entry*
   - e.g., to realize interrupts

In addition:

- **no task hierarchies:** master of all tasks is ‘‘system’’
- **no *rendezvous*:** all task interaction via *protected types*
- **static task set**: no dynamic task allocation, with static properties (e.g., task priorities), never terminate
- **entry queues**: have a capacity of exactly one task

Ravenscar - Tasks, no hierarchy



- tasks are "all at same level"
- master is always the system
- for details see *blinker* code from Moodle

Ravenscar - Tasks, single release point

```ada
with Ada.Real_Time; use Ada.Real_Time;
task body my_periodic_task is
  Next_Time : Time;
  Period : constant Time_Span := Milliseconds(100);
begin
  Next_Time := Ada.Real_Time.Clock + Period;
  loop
    delay until Next_Time; -- without "until" is illegal!
    -- Do work...
    Next_Time := Next_Time + Period;
  end loop;
exception -- handler goes here
end my_periodic_task;
```

```ada
task body my_sporadic_task is
begin
  loop
    Monitor.Wait_Event; -- protected entry
    -- Do work...
  end loop;
exception -- handler goes here
end my_sporadic_task;
```

Ravenscar - Tasks, single release point

```
with Ada.Real_Time; use Ada.Real_Time;
task body my_periodic_task is
   Next_Time : Time;
   Period : constant Time_Span := Milliseconds(100);
begin
   Next_Time := Ada.Real_Time.Clock + Period;
   loop
      delay until Next_Time; -- without "until" is illegal!
      -- Do work...
      Next_Time := Next_Time + Period;
   end loop;
exception -- handler goes here
end my_periodic_task;
```

```
task body my_sporadic_task is
begin
   loop
      Monitor.Wait_Event; -- protected entry
      -- Do work...
   end loop;
exception -- handler goes here
end my_sporadic_task;
```

Ravenscar - Interrupts

- in embedded systems: a timer, a UART message, input capture, ...
- interrupts always share data with some other process $\Rightarrow$ race conditions
- Ravenscar: protected object to encapsulate shared data
- ISR is a protected procedure, cannot be called directly
- no race condition possible because of mutual exclusion

Ravenscar - Interrupts

- in embedded systems: a timer, a UART message, input capture, ...
- interrupts always share data with some other process ⇒ race conditions
- Ravenscar: protected object to encapsulate shared data
- ISR is a protected procedure, cannot be called directly
- no race condition possible because of mutual exclusion

```
protected Receiver is
  entry Wait (Msg: access Message);
  pragma Interrupt_Priority (
        UART_Priority);
private
  procedure My_ISR;
  pragma Attach_Handler(My_ISR,
        UART_Data_Arrival);
  Buffer: Contents(Message_Size);
  Length: Natural := 1;
  Message_Ready : Boolean := False;
end Receiver;
```

- `pragma`: ensure that ceiling priority of handler $\geq$ max. possible prio of the interrupt to be handled

...and the body:

```
protected body Receiver is
  procedure My_Isr is -- called when UART gets data
  begin
    -- ... access hardware, set 'Length' and 'Buffer'
    Message_Ready := True;
  end My_Isr;

  entry Wait (Msg: access Message) when Message_Ready is
  begin
    -- copy data to caller
    Msg.Value (1 .. Length) := Buffer (1 .. Length);
    Msg.Length := Length;
    -- reset buffer, it's empty now:
    Length := 1;
    Message_Ready := False;
  end Wait;
end Receiver;
```

... and the matching main task:

```
task Message_Processor is
  pragma Priority (System.Priority'First);
end Message_Processor;
task body Message_Processor is
  Next_Message : aliased Message (Size => Message_Size'Last)
       ;
begin
  loop
    Receiver.Wait (Next_Message'Access); -- "pointer"
    -- do something in response...
  end loop;
end Message_Processor;
```

Note the *single release point* is the entry function in this example. This is a *sporadic* task, since we do not know when it will execute.

Ravenscar - Scheduling Policy for Tasks

Ravenscar: `FIFO_Within_Priorities`, preemptive

Additionally: Static CPU assignment on multi-core systems.

$\Rightarrow$ OS must support this. Linux: kernel RT patchset

Ravenscar - Locking Policy for Shared Resources

Ravenscar: `Ceiling_Locking`

- =Priority Ceiling Protocol (see Lecture 7)
- protected objects get a ceiling priority, which is higher than that of all tasks that are accessing it
- caller inherits the priority of the protected object

Ravenscar - Effect of Policies

- highest-priority runnable task in the system can always run
- no mutex locks or semaphores required
- tasks never block when calling a protected object
- can be analyzed with simple algorithms such as *rate-monotonic* schedulability test:

$$\text{task set schedulable if} \quad u = \sum_{i=1}^{n} \frac{e_i}{P_i} \leq n(\sqrt[n]{2} - 1) \qquad (1)$$

u=processor load, n=number of tasks, $\forall i : D_i = P_i$

Ravenscar - Dynamic Memory

- memory for each task (e.g., stack) must be resolved latest at link time
- i.e., tasks and protected types must not be dynamically allocated
- *implicit* memory allocation not allowed (compiler must obey this)
- **no restrictions** for large or dynamic-size objects, i.e., keyword `new` is allowed

Ravenscar - Protected Objects

Queues in protected objects can hold at most one task

- i.e., each `entry` inside can only have one caller
- (`procedure`s or `function`s do not have queues)
- program will compile, but ...
- if one task is waiting in the queue and another one wants to enqueue ⇒ Exception `Program_Error`

Protected objects cannot have more than one entry

- multiple functions and procedures are okay
- ⇒ at most one task can be waiting for a guard

workarounds:

- see *Guide for the use of the Ada Ravenscar Pro le in high-integrity systems*

Ravenscar - Potentially Blocking Operations

pragma Detect_Blocking is part of Ravenscar:

- *potentially* blocking operations cannot be called from within protected objects
  - a lot of them are already banned by the profile: *select*, *task entry*, *delay*, ...
- however, reading from files, etc. could block
- currently, this is a run-time check
  - with this pragma, every potentially blocking operation raises an exception *when called*
  - compiler does not check this!
- **simplifies WCET computation**

Race Conditions & Deadlocks

- all communication/synchronization between tasks must be protected:
  - via protected object or suspension object or atomic object
  - only one task at a time can work on shared data
  - access is serialized through FIFOs
  - $\Rightarrow$ no race conditions
- no potentially blocking operations from protected entries, and PCP
  - a task that is blocking other tasks cannot get blocked again
  - no “cycles” possible
  - $\Rightarrow$ no deadlocks possible

Ravenscar - Further Restrictions

- there are more forbidden features:

  `No_Dynamic_Attachment, No_Local_Protected_Objects, No_Local_Timing_Events, No_Requeue_Statements, No_Specific_Termination_Handlers, Simple_Barriers, Max_Entry_Queue_Length => 1, Max_Protected_Entries => 1, Max_Task_Entries => 0, No_Dependence => Ada.Asynchronous_Task_Control, No_Dependence => Ada.Calendar, No_Dependence => Ada.Execution_Time.Group_Budgets, No_Dependence => Ada.Execution_Time.Timers, No_Dependence => Ada.Task_Attributes, No_Dependence => System.Multiprocessors.Dispatching_Domains`
- as Ada grows, new features enter this list

Ravenscar - Rationale

The *Guide for the use of the Ada Ravenscar Profile in high integrity systems*, A. Burns, B. Dobbing and T. Vardanega, 2003 provides a rationale for each of the restrictions.

Ravenscar - Implications

With the described task model, scheduling policy and other restrictions, implementation is easy:

- little or no RTOS required
- minimal program is only $\approx$2kB of object code
- deterministic tasking
- reduced footprint of run-time
  - smaller run-time means lower certification cost
  - means smaller memory and thus less power consumption
  - means shorter boot time
- such implementations are well-studied and certifiable up to highest assurance levels
  - e.g., "DO-178B Level A" for commercial avionics (failure of level-A software results in catastrophic consequences, such as aircraft crash or unintended release of weapons)

Ada: Even more safety available...

- annex "Safety and Security"/"High Integrity Systems" provides *pragmas* that the developer can turn on:
  - `Normalize_Scalars`: set uninitialized variables to an out-of-range default value
  - `Reviewable`: provide implementation information for analysis, e.g.,
    - execution time, memory usage and mapping from source to object codes, presence of run-time checks, ... ⇒ **useful for WCET analysis**
  - many others (no protected types, no allocators (`new` etc), no exceptions, no float, no fixed, no delay, no recursion, detect blocking) ...
- attribute `'valid` to check whether value of object is within legal range, e.g., to recognize bit flips from cosmic rays, hardware errors etc.

Section 3

Conclusion

Conclusion

For safety-critical real-time systems in Ada, use

- a compiler supporting **annex Real-Time** $\Rightarrow$ to get appropriate features
  - requires Real-Time OS
- the ***Ravenscar* profile** $\Rightarrow$ ban tasking features to get determinism and enable schedulability analysis
  - enforces a simple program structure
  - requires Real-Time kernel (scheduling policy ''FIFO within Priorities'') to work correctly
- a compiler supporting **annex High Integrity Systems** $\Rightarrow$ to get information about the object code and restrict language even further, if desired

Next Lecture...

- verification of Ravenscar programs
  - static analysis of concurrent code
  - scheduling analysis
  - (WCET)
- dynamic and static analysis of sequential Ada code
  - sequential code is not addressed in Ravenscar
- formal verification (similar to Model Checking in Esterel) for Ravenscar programs
  - all-in-one for the restricted Ravenscar subset

Section 4

References

References

- *Ada95 Lovelace tutorial*, David A. Wheeler (who wrote that in his free time), `http://www.adahome.com/Tutorials/Lovelace/lovelace.htm`
- *Ada Wikibook*, `http://en.wikibooks.org/wiki/Ada_Programming`
- *The Boeing 777 Flies on 99.9% Ada*, `http://archive.adaic.com/projects/atwork/boeing.html`
- *Ada 95 Eliminates Race Conditions*, J.G.P. Barnes, Parallel and Distributed Real-Time Systems, 1995.
- *Programming Real-Time with Ada 2005*, P. Rogers, `http://www.embedded.com/design/prototyping-and-development/4025713/Programming-real-time-with-Ada-2005`, 2006.
- *Guide for the use of the Ada Ravenscar Profile in high-integrity systems*, A. Burns et. al., University of York, Technical Report YCS-2003-348, 2003.
- *Ada 95 Rationale: The Language - The Standard Libraries*, J. Barnes, Springer, 1995.

All online resources as of December 2015.