A "Bare-Machine" Implementation of Ada Multi-Tasking Beneath the Linux Kernel

Hongfeng Shen (FSU), Arnaud Charlet (ACT), Ted Baker (FSU)

talk for Ada-Europe'99

9 June 1999

Acknowlegement

This project benefited from the work on restricting GNARL done by Mike Kamrad and Barry Spinney of Top Layer Networks, Inc.

Outline

1. motivation
2. Linux and RT Linux
3. Ada runtime system implementation for RT Linux
4. performance

Motivation

- GNAT implementation for real-time systems course
- Ada 95 tasking as it was intended, for bare machine
- Ravenscar profile
- basis for further real-time Ada and OS experimentation

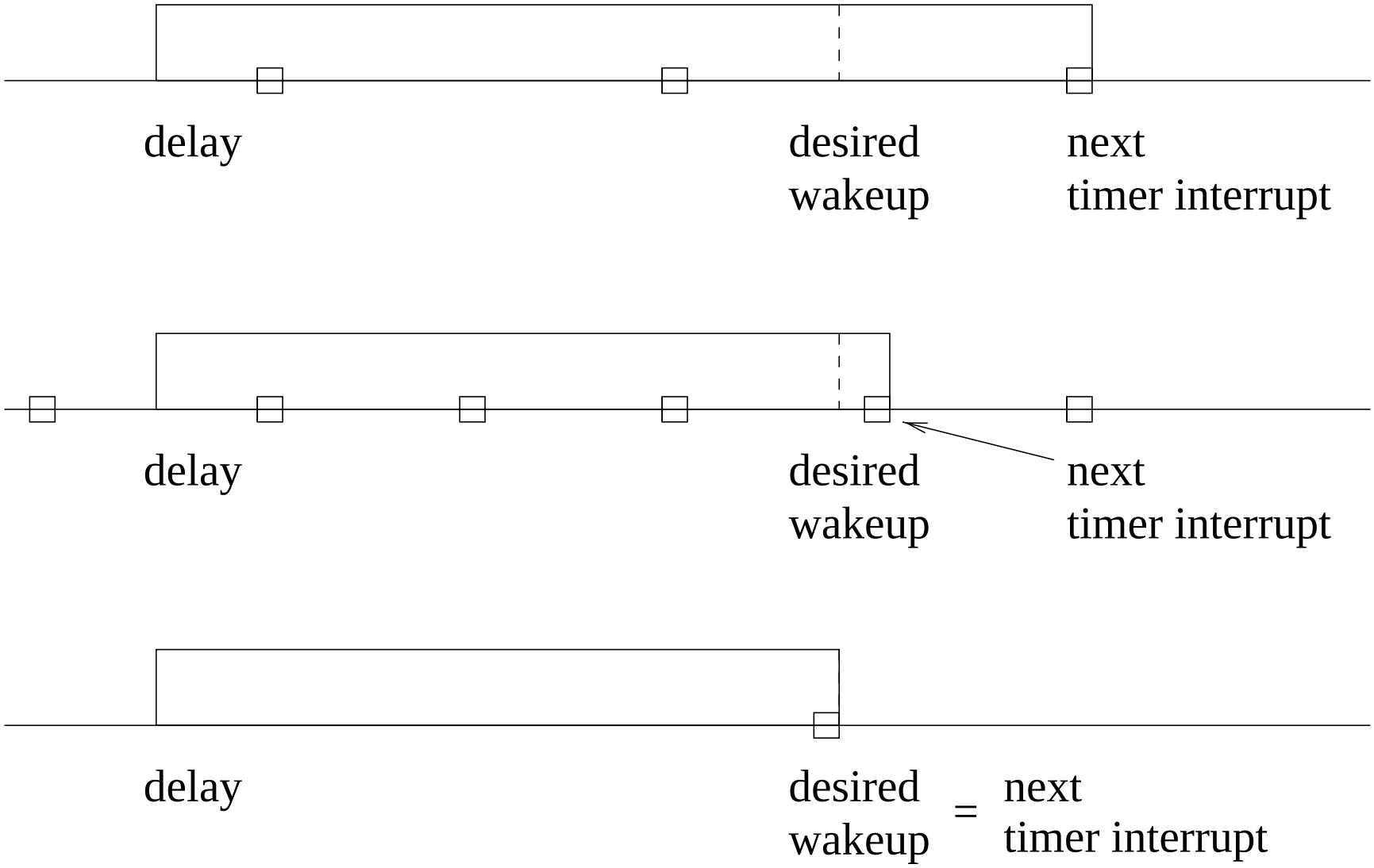
Objectives

- low overhead
- very predictable timing
- cheap platform (Pentium PC)
- simplicity
- maximum code reuse

Design Issues

- Ada tasking primitives must be on bare hardware
- hardware timer must operate in programmed interval mode
- avoid writing new device drivers
- avoid dynamic memory allocation
- avoid virtual memory management (paging)

Periodic *vs.* Interval Mode



Plan

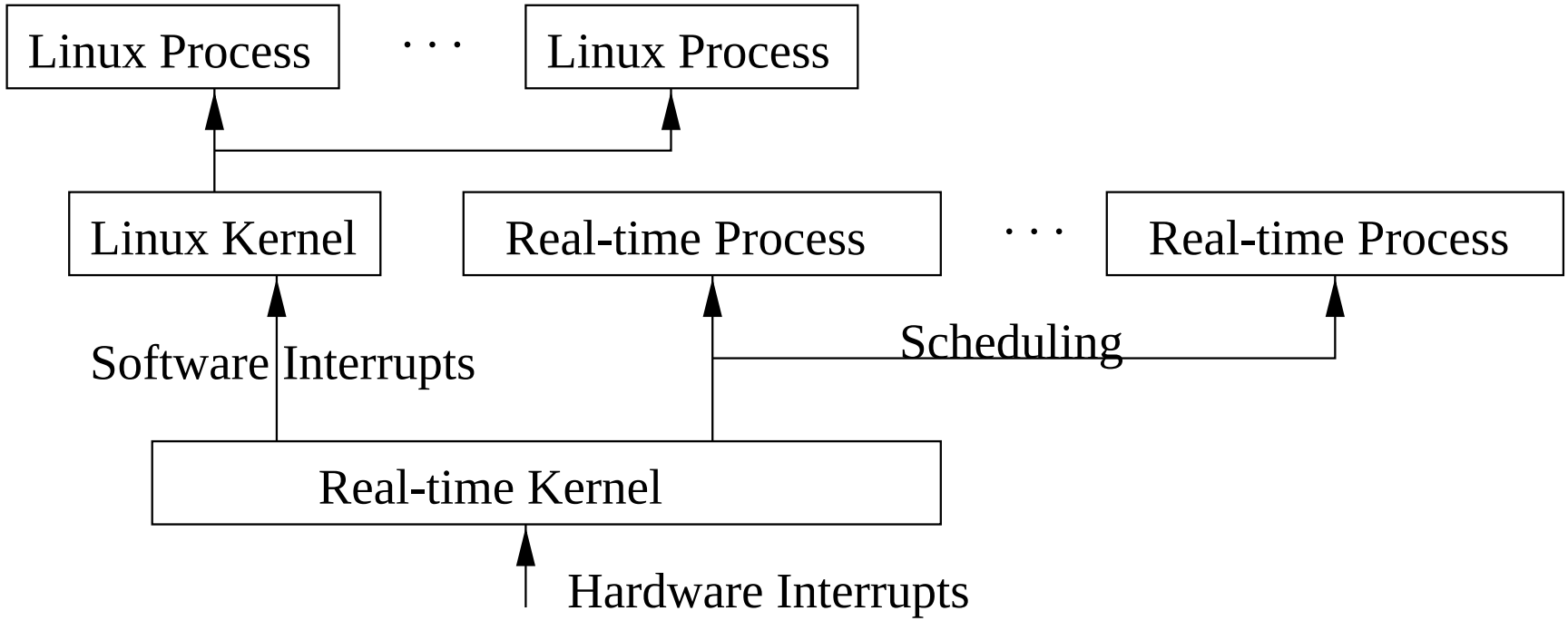
- run hard-real-time tasks in foreground
- no system call traps
- no hardware memory management context switches
- run conventional OS in background
- let conventional OS provide most device drivers
  especially those with ugly timing problems
  like disk drive and network interface
- original choice of OS: DOS
- later decision: Linux

RT Linux (Victor Yodaiken)

- add-on to Linux OS
- Linux OS runs in background
- RT Linux schedules real-time tasks in foreground, in kernel address space
- postpones delivery hardware interrupts to Linux
- fine-grained clock and interval-timer driver
- FIFO buffers for communication between fore and back
- easily replaceable scheduler

`http://rtlinux.cs.nmt.edu/~rtlinuxwhitepaper/short.html`

RT Linux Organization



Linux Kernel Modules

- OS extensibility mechanism
- dynamically loadable and unloadable
- run in kernel address space
- `init_module` is run when module is inserted
- `cleanup_module` is run when module is removed
- RT Linux is implemented in kernel modules

Example of Kernel Module in C

```
#define MODULE
#include <linux/module.h>
int init_module (void)
{ printk ("Hello, World\n"); return 0; }
void cleanup_module (void)
{ printk ("Goodbye, World!$\backslash$ n"); }
```

Plan

- Ada application as Linux kernel module
- reuse RT Linux timer and FIFOs
- replace RT Linux scheduler by Ada scheduler
- follow implementation model of Ada 95 Rationale
- add support for priority ceiling locks
- restrict Ada runtime system as necessary
- add compiler optimizations that exploit simple casse but can also be done with non-restricted runtime

Challenges

- no experience writing kernel modules in Ada
- cannot directly use C kernel header files
- running in kernel imposes restrictions
- kernel modules must be complete
  - may refer to symbols in kernel & previously loaded modules
  - no forward references
- kernel memory is limited (not paged)
  - original GNAT runtime system is large, intertwined
- cannot make OS service calls from inside kernel
  - no `malloc()` or I/O functions
  - GNAT does a lot of dynamic storage allocation

Ada Kernel Module

```
package Hello is
   type Aliased_String is array (Positive range <>)
     of aliased Character;
   pragma Convention (C, Aliased_String);
   Kernel_Version : constant Aliased_String
      := "2.0.33" & Character'Val (0);
   pragma Export (C, Kernel_Version, kernel_version");
   Mod_Use_Count : Integer;
   pragma Export (C, Mod_Use_Count, "mod_use_count_");
   procedure Printk (Message : String);
   pragma Import (C, Printk, "printk");
   function Init_Module return Integer;
   pragma Export (C, Init_Module, "init_module");
   procedure Cleanup_Module;
   pragma Export (C, Cleanup_Module, "cleanup_module");
end Hello;
```

Ada Kernel Module (body)

```
package body Hello is
   procedure Hello_Elabb;
   pragma Import (Ada, Hello_Elabb, "hello___elabb");
   function Init_Module return Integer is
   begin
      Hello_Elabb;
      Printk ("Hello, World!" & Character'Val(10));
      return 0;
   end Init_Module;
   procedure Cleanup_Module is
   begin
      Printk ("Goodbye, World!" & Character'Val(10));
   end Cleanup_Module;
end Hello;
```

Ada 95 Rationale Scheduling Model

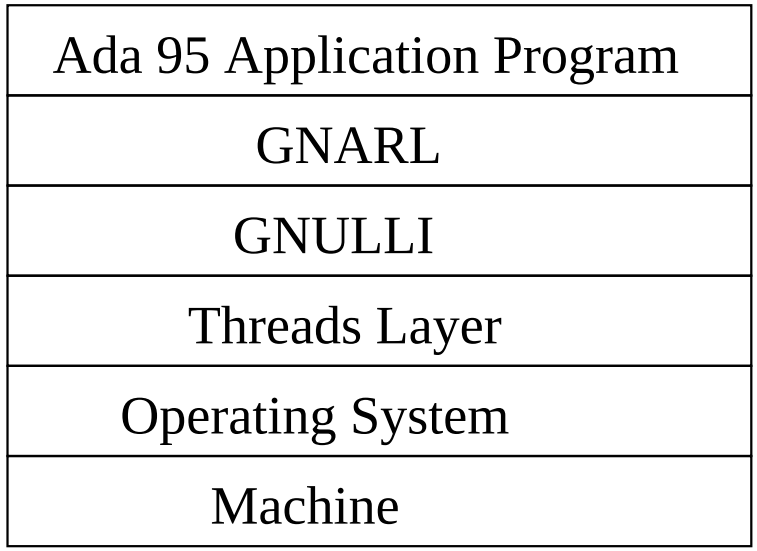
- Scheduling is strictly preemptive
- Only ready tasks can hold locks.
- A task is not permitted to obtain a lock if its active priority is higher than the ceiling of the lock.
- A task that is holding a lock inherits the priority ceiling of the lock (only) while it holds the lock.

---

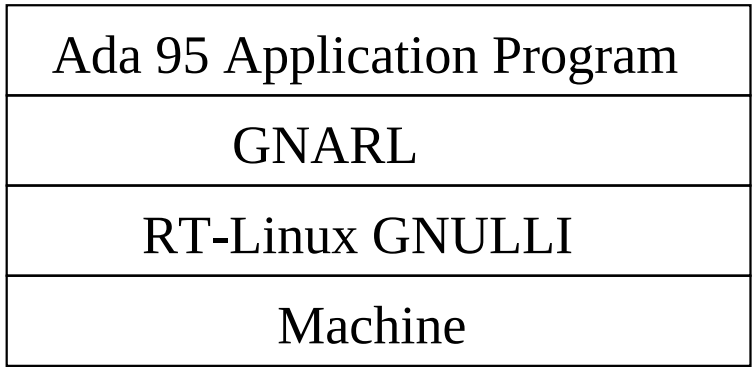
lock operation = raise priority to ceiling of object

unlock operation = restore previous priority & check preeemption

GNARL Implementation Model



(a)



(b)

Plan

- port GNULLI to RT Linux
- test GNULLI directly
- restrict GNARL as necessary to make it fit
  (Top Level Networks project converged here)

Simplifications

- single task/thread control block
- preallocated array of task control blocks
-

GNULLI Operations

- create/destroy task
- lock/unlock
- (timed) sleep/wakeup

Lock Operation with Threads

```
procedure Write_Lock
  (L : access Lock; Ceiling_Violation : out Boolean) is
   Result : Interfaces.C.int;
begin
   Result := pthread_mutex_lock (L.L'Access);
   Ceiling_Violation := Result /= 0;
end Write_Lock;
```

Lock Operation on Bare Machine

```
procedure Write_Lock
  (L : access Lock; Ceiling_Violation : out Boolean) is
   Prio : constant System.Any_Priority :=
     Current_Task.LL.Active_Priority;
begin
   Ceiling_Violation := False;
   if Prio > L.Ceiling_Priority then
      Ceiling_Violation := True;
      return;
   end if;
   L.Pre_Locking_Priority := Prio;
   Current_Task.LL.Active_Priority := L.Ceiling_Priority;
   if Current_Task.LL.Outer_Lock = null then
      --  if lock is not nested, record a pointer to it
      Current_Task.LL.Outer_Lock := L.all'Unchecked_Access;
   end if;
end Write_Lock;
```

Unlock Operation on Bare Machine

```
procedure Unlock (L : access Lock) is
   Result  : Interfaces.C.int;
begin
   Result := pthread_mutex_unlock (L.L'Access);
end Unlock;
```

Lock Operation on Bare Machine

```
procedure Unlock (L : access Lock) is
   Flags : Integer;
begin
   if Current_Task.LL.Outer_Lock = L.all'Unchecked_Access then
      Current_Task.LL.Active_Priority :=
        Current_Task.LL.Current_Priority;
      Current_Task.LL.Outer_Lock := null;
   else
      Current_Task.LL.Active_Priority := L.Pre_Locking_Priority;
   end if;
   if Current_Task.LL.Active_Priority
     < Current_Task.LL.Succ.LL.Active_Priority then
      Save_Flags (Flags); --  Saves interrupt mask.
      Cli;                --  Masks interrupts
      Delete_From_Ready_Queue (Current_Task);
      Insert_In_Ready_Queue (Current_Task);
      Restore_Flags (Flags);
      Call_Scheduler;
   end if;
end Unlock;
```

Sleep Operation with Threads

```
procedure Sleep (Self_ID : Task_ID; Reason  : Task_States) is
   Result : Interfaces.C.int;
begin
   if Self_ID.Pending_Priority_Change then
      Self_ID.Pending_Priority_Change := False;
      Self_ID.Base_Priority := Self_ID.New_Base_Priority;
      Set_Priority (Self_ID, Self_ID.Base_Priority);
   end if;
   Result := pthread_cond_wait
     (Self_ID.LL.CV'Access, Self_ID.LL.L.L'Access);
   pragma Assert (Result = 0 or else Result = EINTR);
end Sleep;
```

Sleep Operation On Bare Machine

```
procedure Sleep
  (Self_ID : Task_ID; Reason : System.Tasking.Task_States) is
   Flags : Integer;
begin
   Self_ID.State := Reason;
   Save_Flags (Flags);
   Cli;
   Delete_From_Ready_Queue (Self_ID);
   if Self_ID.LL.Outer_Lock = Self_ID.LL.L'Access then
      Self_ID.LL.Active_Priority := Self_ID.LL.Current_Priority;
      Self_ID.LL.Outer_Lock := null;
   else
      Self_ID.LL.Active_Priority := Self_ID.LL.L.Pre_Locking_Priority;
   end if;
   Restore_Flags (Flags);
   Call_Scheduler;
   Write_Lock (Self_ID);
end Sleep;
```

Wakeup Operation with Threads

```
procedure Wakeup (T : Task_ID; Reason : Task_States) is
   Result : Interfaces.C.int;
begin
   Result := pthread_cond_signal (T.LL.CV'Access);
   pragma Assert (Result = 0);
end Wakeup;
```

Wakeup Operation on Bare Machine

```
procedure Wakeup (T : Task_ID; Reason :
System.Tasking.Task_States) is
   Flags : Integer;
begin
   T.State := Reason;
   Save_Flags (Flags);
   Cli;  --  Disable interrupts.
   if Timer_Queue.LL.Succ = T then
      if T.LL.Succ = Timer_Queue then
         No_Timer;
      else
         Set_Timer (T.LL.Succ.LL.Resume_Time);
      end if;
   end if;
   Delete_From_Timer_Queue (T);
   Insert_In_Ready_Queue (T);
   Restore_Flags (Flags);
   Call_Scheduler;
end Wakeup;
```

GNARL and Compiler Changes

- avoid some dynamic storage allocations
- optimize some special cases, e.g.
  - protected object with no entries

  - entry with simple Boolean variable as barrier
  - single lock runtime system

Real Time Predictability and Overhead: Bare GNULLI

- harmonic task set: 320, 160, 80, 40, 20, 10 Hz
- pattern: `lock; work; unlock; work; lock; work; unlock;`
- work depends on *load level* parameter
- *load level* is adjusted, by bisection, until maximum schedulable utilization is reached

---

97% CPU utilization, on 90MHz/33MHz Pentium PC

Lock/Unlock Performance: Bare GNULLI

1.06 microseconds per cycle, on 90MHz/33MHz Pentium PC.

Tasking Performance: Full GNARL

	FSU threads	Linux threads	RT-Linux t
simple protected procedure	2.3 μs	3.1 μs	
protected procedure	3.5 μs	4.7 μs	
po "rendezvous"	4.6 μs	9.7 μs	

These figures are on 166MHz/66MHz Pentium PC.

Conclusions

- Linux kernel modules can be written in Ada
- Ada tasking can be implemented within the Linux kernel
- Ravenscar restrictions are helpful, but not required
- timing predictability is as desired
- overhead is very low
- RT Linux code is now merged into GNAT development baseline

Further Work?

- further simplifications (e.g., single lock)
- further optimizations by compiler
- improve ease of use
- experiment with alternative scheduling: e.g., sporadic server
- distributed systems annex implementation?
  - one or more partitions are in kernel
  - other partitions are processes
  - use FIFOs for inter-partition communication mechanism
- dynamic loading/unloading of kernel partitions?