Tasking Beneath the Linux Kernel

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

Multi-Tasking Beneath the Linux Kernel

A "Bare-Machine" Implementation of Ada

6 June 1999
Talk for Ada-Europe'99
Acknowledgement

Mike Kamrad and Barry Spinney of Top Layer Networks, Inc.

This project benefited from the work on restricting Gnarl done by

Hongfeng Shen (FSU), Arnaud Chartier (ACT), Ted Baker (FSU)
Outline

1. Motivation
2. Linux and RT Linux
3. Ada runtime system implementation for RT Linux
4. Performance
Motivation

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

Hong Feng Shen (FSU), Arnaud Charlet (ACRT), Ted Baker (FSU)
Objectives

• maximum code reuse
• simplicity
• cheap platform (Pentium PC)
• very predictable timing
• low overhead
Design Issues

Avoid virtual memory management (pasting)

Avoid dynamic memory allocation

Avoid writing new device drivers

Hardware timer must operate in programmer interval mode

Ada tasking primitives must be on bare hardware
Timer Interrupt = Timer Interrupt
next desired wakeup delay

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

Plan
http://ft.tjinux.cs.nmt.edu/~ftjinux/ftpextendedpaper/short.htm

- Easily replaceable scheduler
- FIFO buffers for communication between fore and back
- Fine-grained clock and inter-ratelimer driver
- Postpones delivery hardware interrupts to Linux
- RTL Linux schedules real-time tasks in foreground,
  Linux OS runs in background
  Add-on to Linux OS

RTL Linux (Victor Yodziken)

Hongfeng Shen (FSU), Amund Charlet (ACT), Ted Baker (FSU)
RT Linux Organization
Linux Kernel Modules

- RT Linux is implemented in kernel modules
- `cleanmod` module is run when module is removed
- `insmod` module is run when module is inserted
- Run in kernel address space
- Dynamically loadable and unloadable
- OS extensibility mechanism

Hongfeng Shen (FSU), Arnaud Charlet (AC'T), Ted Baker (FSU)
Example of Kernel Module in C

```c
#include <linux/module.h>

int init_module(void)
{
    printk("Hello, World!");
    return 0;
}

void cleanup_module(void)
{
    printk("Goodbye, World!")
}

#define MODULE
```
be done with non-restricted runtime

add compiler optimizations that exploit simple case but can also

restrict Ada runtime system as necessary

disable support for priority ceiling locks

follow implementation model of Ada 95 Rationale

replace RT Linux scheduler by Ada scheduler

reuse RT Linux timer and FIOS

Ada application as Linux Kernel module

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

Challenges
end hello:
procedure clean-up (c, clean-up-module, "clean-up-module")
procedure clean-up (c, init-module, "init-module")
function init-module return integer;
procedure print (c, print, "print")
procedure print (message : string);
procedure clean-up (c, mod-use-count, "mod-use-count")
mod-use-count : integer;
procedure clean-up (c, kernel-version, kernel-version);
kernel-version : constant aliased string := "2.0.33" & character'val (0);
ilong integer type aliased-string is array (positive range <>)
package hello is

Ada Kernel Module

Hengfei Shen (FSU), Arnaud Cherlet (ACCT), Ted Baker (FSU)
end Hello; end Cleanup-Module;

procedure Cleanup-Module is begin
    procedure Cleanup-Module is
        begin
            return 0;

    printt ("Hello, world!", Character Val (10));
    Hello-Tab;
        begin
            function Int-Module (return Integer is
                procedure Import (Ada, Hello-Tab, "Hello---Tab");
                procedure Hello-Tab;
                Hello is
                    package body Hello is

Ada Kernel Module (body)

Hong Feng Shen (FSU), Armand Chartier (ACI) Ted Baker (FSU)
unlock operation = restore previous priority & check preemption

lock operation = raise priority to ceiling of object

(only) while it holds the lock.

A task that is holding a lock inherits the priority ceiling of the lock.

A task is not permitted to obtain a lock if its active priority is

higher than the ceiling of the lock.

Only ready tasks can hold locks.

Scheduling is strictly preemptive.

Ada 95 Rationale Scheduling Model
### Ada 95 Application Program Model

<table>
<thead>
<tr>
<th>Machine</th>
<th>Operating System</th>
<th>Threads Layer</th>
<th>GNATLI</th>
<th>GNARL</th>
<th>Ada 95 Application Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>RT-Linux GNU/CLI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Diagram:**

Ada 95 Application Program Model

Focusing beneath the Linux Kernel
Plan

(Top level Network project converged here)

• Restrict GNURL as necessary to make it fit
• Test GNURL directly
• Port GNURL to RT Linux
Simplifications

- preallocated array of task control blocks
- single task/thread control block
Tasking Beneath the Linux Kernel

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

**GNU C Operations**

Adi Efraimou, Florida State University (FSU), Armand Charlet, ACT; Ted Baker, FSU
Lock Operation with Threads

procedure Write-Lock

begin

result : Interfaces.C.int;

result = pthread_mutex_lock (L.l.lock access);

Getting-Violation = result /= 0;

end Write-Lock;
end Write-Lock;
end if;

if Current-Task.LL.Outer-Lock = L.All.Unchecked-Access,

if lock is not nested, record a pointer to it

null the

if Current-Task.LL.Outer-Lock = null then

Current-Task.LL.Active-Priority = L.Getting-Priority;

L.Pre-Locking-Priority = Pr1o;
end if;

return;

if getting-Violation = True,

if pr1o > L.Getting-Priority then

getting-Violation = False;

begin

Current-Task.LL.Active-Priority = pr1o : constant System.Any-Priority := L : access Lock ; getting-Violation : out Boolean)

procedure Write-Lock

Lock Operation on Bare Machine

9 June 1990
Unlock Operation on Bare Machine

end Unlock;
Result = pthread_mutex_unlock (l, l'access);
begin
Result = Interfaces.C.int;
procedure Unlock Unlock (l : access Lock) is
Lock Operation on Bare Machine

end unlock;
end if;
CALL Scheduler;
Restore-Flags (Flags);
Insert-In-Ready-Queue (Current-Task);
Delete-From-Ready-Queue (Current-Task);
-- Masks interrupts
CALL ;
-- Saves interrupt mask.
if Current-Task.LL.Active-Priority then
begin
Flags : Integer ;
procedure Unlock (L : access Lock) is
else
Current-Task.LL.Active-Priority := L.Pre-Locking-Priority;
else
Current-Task.LL.Outer-Lock := null;
Current-Task.LL. Current-Priority :=
Current-Task.LL. Active-Priority :=
Current-Task.LL. Outer-Lock := L. All Unchecked-Access then
procedure Unlock (L : access Lock) is
procedure steep (Self-ID : Task-ID; Reason : Task-States) is
begin
  Result := Interfaces.C.int;
end steep;

pragma assert (Result = 0 or else Result = EINTR);

result := pthread-cond-wait
end if;

Self-Priority (Self-ID, Self-ID.Base-Priority);
Self-ID.Base-Priority := Self-ID.New-Boss-Priority;
Self-ID.Pending-Priority-Change := Raise;
if Self-ID.Pending-Priority-Change then

end Sleep;

Sleep Operation with Threads
procedure step

if Self-ID.Running-Queue = null then
    Self-ID.Running-Queue = Ready-Queue (Self-ID);
    begin
        flags : integer;
        save-flags (flags);
        Self-ID.State := Reason;
    end
end step;

Write-Lock (Self-ID);
Call-Scheduler;
Restore-Flags (flags);
end it;

Self-ID.LL.ACTIVE-PRIORITY := Self-ID.LL.CURRENT-PRIORITY;
else
    Self-ID.LL.Outer-Lock := null;
    if Self-ID.LL.OUTER-LOCK = Self-ID.LL.CURRENT-PRIORITY
        then
            Delete-From-Ready-Queue (Self-ID);
            if Self-ID.LL.Outer-Lock = Self-ID.LL.LL.Acress then
                Critical;
                restore-flags (flags);
                Self-ID.State := Reason;
            end if;
        else
            begin
                flags : integer;
                save-flags (flags);
                Self-ID.State := Reason;
            end if;
end if;

end step

Procedure Operation On Bare Machine
Wakeup Operation with Threads

```
end wakeup;

procedure wakeup (T : Task-Id; Reason : Task-States) is
begin
    Result : Interfaces.C.int;
    Result := pthread_cond_signal (T.LL.CV, access);
    pragma assert (Result = 0);
end wakeup;
```
end wakeup;

CALL Scheduler;

Restore-Flags (Flags);

Insert-In-Ready-Queue (T);

Delete-From-Timer-Queue (T);
end if;
end if;

Set-Timer (T.LL.Succ.LL.Resume-Timer);
else

NO-Timer;
if T.LL.Succ = Timer-Queue then

if Timer-Queue.LL.Succ = T then

CALL -- Disable Interrupts.
Save-Flags (Flags);
T.State := Reason;
begin

Flags : Integer;
System.Tasking.Task-States is:

procedure Wakeup (T : Task-ID; Reason :)

Wakeup Operation on Bare Machine

Hongfei Shen (FU), Arnaud Carlet (ACI), Ted Baker (F5U)
- Single lock runtime system
- Entry with simple Boolean variable as barrier

- Protected object with no entries
  - Optimize some special cases, e.g.,
  - Avoid some dynamic storage allocations

GNARL and Compiler Changes
97% CPU utilization on 900MHz/33MHz Pentium PC

utilization is reached

load level is adjusted, by bisecction, until maximum schedulable

load level parameter

work depends on load level parameter

pattern: lock; work; unlock; lock; work; unlock; lock; work; unlock;

harmonic task set: 320, 160, 80, 40, 20, 10 Hz

Real-Time Predictability and Overhead: Bare GnuLiP

Adel-Europe 99
Hongfei Shen (FSU), Arnaud Charet (ACI), Ted Baker (FSU)
Lock/unlock performance: Bare CPU

I. 06 microseconds per cycle, on 90MHz/33MHz Pentium PC.
These figures are on 166MHz/66MHz Pentium PC.

<table>
<thead>
<tr>
<th>FSU threads Linux threads</th>
<th>RT-Linux</th>
<th>Simple protected procedure</th>
<th>Protected procedure</th>
<th>rendezvous</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 µs</td>
<td>4.3 µs</td>
<td>2.3 µs</td>
<td>5.5 µs</td>
<td>4.9 µs</td>
</tr>
<tr>
<td>4.7 µs</td>
<td>4.7 µs</td>
<td>3.9 µs</td>
<td>4.9 µs</td>
<td>6.2 µs</td>
</tr>
</tbody>
</table>
Conclusions
Dynamic loading/unloading of kernel partitions
- use FIFOs for inter-partition communication mechanism
  - other partitions are processes
  - one or more partitions are in kernel
  - distributed systems annex implementation?

Experiment with alternative scheduling: e.g., sporadic server
- improve ease of use
- further optimizations by compiler
- further simplifications (e.g., single lock)

Further Work