System Architecture

6 Thread Switching

Yielding, General Switching

November 10 2008
Winter Term 2008/2009
Gerd Liefländer
Agenda

- Review & Motivation
- Thread Switching Mechanisms
  - Cooperative PULT Scheduling + Thread Switch
  - Cooperative KLT Scheduling
  - KLT Thread Switch of KLTs
- Additional Design Parameters
  - User and Kernel Stack
  - Idle Thread
  - Initialization/Termination
Review & Motivation
Problems to Solve

- *How to design mechanism* `thread_switch`?

- *How to schedule* threads, and for how long?

- *Do we need* `time slices` in every computer?
Influence of CPU Switching

- CPU switching back and forth among threads:
  - Rate at which a thread performs its computation will not be uniform
  - Nor will it be reproducible if the same set of threads will run again
  - Its timing (e.g. waiting times) can depend on other application- or system-activities

⇒ Threads should never be programmed with built-in assumptions about timing
Conclusion

- **Never** accept a solution relying on timing conditions

- If you program portable application don’t rely on
  - specific scheduling policies
  - number of processors
  - ...

- In case, you can rely on a specific platform offering different scheduling policies, try to get the most promising one

- In each system the scheduling policies should be supported by a policy-free dispatching mechanism
Pult Scheduling
Cooperative Scheduling

**Assumption:** Given 2 pure CPU-bound threads T1 and T2, 
**one CPU** and **no interference** with a device.

Dispatch only cooperatively via `yield()`

```
yield(T1)  running
yield(T2)
running
```
Simplified User Level Yield (1)

yield(T2)

T1

call yield

yield(T1)

reload context of T2

yield(T2)

!!! A bit tricky !!!
Return to another caller

time

save context of T1

yield()

return from yield
Simplified UL-Yield (2)

What is happening at?
procedure yield(NT:thread) 
{
    ... 
    save context of CT 
    ... 
    load context of NT 
    ... 
    return;
}
Assumption: Both threads T1 and T2 already have called yield() once before
Corollary: Each thread gets and gives up control within the procedure yield at exactly the same (user land) instruction

How to solve this problem?

Part of yield still runs under control of caller

Part of yield will run under control of the next thread
procedure yield(NT:thread)
{
  ...
  save context of CT
  CT.sp := SP; SP := NT.sp;
  load context of NT
  ...
  return;
}

Part of yield still runs under control of caller

Part of yield will run under control of the next thread

\[ SP = \text{stack pointer register} \]

\[ sp = \text{entry in TCB} \]
Stack Contents during UL-Yield (1)

SP

Local Variables of T1

Local Variables of T1

T1 runs

Local Variables of yield

Local Variables of yield

Return Address to T2

Parameter T1

Local Variables of T2

Local Variables of T2
Stack Contents during UL-Yield (2)
Stack Contents during UL-Yield (3)

- Local Variables of yield
- Local Variables of yield
- Return address to T1
- Parameter T2
- Local Variables of T1
- Local Variables of T1

- Local Variables of yield
- Local Variables of yield
- Return Address to T2
- Parameter T1
- Local Variables of T2
- Local Variables of T2
Stack Contents during UL-Yield (4)

Save to current TCB, i.e. to TCB of T1

Switch Stack Pointer

Local Variables of yield
Local Variables of yield
Return address to T1
Parameter T2
Local Variables of T1
Local Variables of T1

Save Context of T1

Local Variables of yield
Local Variables of yield
Return Address to T2
Parameter T1
Local Variables of T2
Local Variables of T2

© 2008 Universität Karlsruhe (TH), System Architecture Group
Stack Contents during UL-Yield (5)

Switch Stack Pointer

Load from NT.TCB

SP' →

Local Variables of yield
Local Variables of yield
Return address to T1
Parameter T2
Local Variables of T1
Local Variables of T1

Local Variables of yield
Local Variables of yield
Return address to T2
Parameter T1
Local Variables of T2
Local Variables of T2

© 2008 Universität Karlsruhe (TH), System Architecture Group
Stack Contents during UL-Yield (6)

- Local Variables of T1
- Return address to T1
- Parameter T2
- Local Variables of T1
- Local Variables of T1

Local Variables of yield

Context of T2

Local Variables of yield

Return address to T2

Parameter T1

Local Variables of T2

Local Variables of T2
Stack Contents during UL-Yield (7)
Stack Contents during UL-Yield (8)

- SP'
- Local Variables of yield
- Local Variables of yield
- Parameter T2
- Return address to T1
- Local Variables of T1
- Local Variables of T1

T2 runs

- Local Variables of T2
- Local Variables of T2

Yield
Summary of a PULT-Yield

Assumption:
Suppose we have a single processor system, and yield is the only dispatching possibility ⇒

- Only the stack of the running thread is “visible”
- Number of involved stack elements as well as their order is the same
- Content of involved stack elements differ a bit
- Of course, $T_1$ or $T_2$ can have different local variables
Thread Library Contents

- Can contain code for:
  - Creating and destroying PULTs
  - Passing messages between PULTs
  - Scheduling thread execution
  - Synchronizing with other PULTs
  - Saving/restoring context of a PULT
Potential Kernel Support for PULTs

Though the kernel is not aware of a PULT, it is still managing the activity of the task that hosts the PULT.

Example:
When a “PULT” does a “blocking system call” ⇒ kernel blocks its whole task

From the point of view of the PULT scheduler this PULT is still in the PULT thread state running !*

Thesis:
PULT thread states are independent of task states
Cooperative Scheduling of KLTs

Anthony D. Joseph

http://inst.eecs.berkeley.edu/~cs162
Causes for a Thread Switch

Additional reasons for switching to another thread:

- Current Thread (CT) terminates
- CT calls synchronous I/O, must wait for result
- CT waits for a message from another thread
- CT is cooperative, hands over CPU to another thread

- CT exceeds its time slice
- CT has lower priority than another ready thread:
  - CT interrupted by a device waking up another thread
  - A higher-priority thread’s sleep time is exhausted
  - CT creates a new thread with higher priority
- CT gets a software interrupt from another thread
Needed: External Events

- What might happen if a KLT never does any I/O, never waits for anything, and never calls `yield()`?
  - Could the ComputePI program grab all resources and never release the processor?
    - What if it didn’t print to console?
  - Must find a way that the kernel dispatcher regains control

- Answer: Utilize External Events
  - Interrupts: signals from hardware or software that stop the running code and jump to kernel
  - Timer: like an alarm clock that goes off every x milliseconds
  - If we ensure that external events occur frequently enough, the dispatcher can gain control again
Events triggering a Thread Switch

Exceptions (all synchronous events):

- Faulty event (reproducible)
  - Division by zero (during instruction)
  - Address violation (during instruction)

- Unpredictable event
  - Page fault (before instruction)

- Breakpoint
  - Data (after instruction)
  - Code (before instruction)

- System call
  - Trap
Events triggering a Thread Switch

Interrupts (all asynchronous events\(^1\)):

- Clock
  - End of time slice
  - Wake up signal

- Printer
  - Missing paper
  - Paper jam, ...

- Network
  - Packet arrived, ...

- Another CPU\(^1\)
  - Inter-Processor signal
  - Software Interrupt

\(^1\)From the point of view of the interrupted CPU
Nested Interrupt Handling (2)

API C sits in between CPU and peripherals

- IR-"Input" register
  - Pending interrupts are listed here (as “1” bits)

- MR-"Mask" register
  - Where IRs can be masked out

- IR-"Compare" register
  - Helps to decide whether interrupting the current interrupt handling is allowed

- Dynamic or static interrupt scheme
  - Rotating or fixed priorities
Thread Control

**Linux Interrupt Handling**

**Linux: Bottom-Half Handler**

- **HW-topic**
- **current thread**

**Top-Half Handler/tasklet**

- **interrupt-handler**
- **next thread**
- **RTI**
- **next thread**

*Depending on system and/or interrupt, sometimes next thread = current thread*
Nested Exception Handling (1)

Bug in exception handler?

Division by 0

1, 2, 3, current application

Division by 0 handler

Remark:
Some systems allow application-specific exception handlers.
Nested Exception Handling (2)

- Division by 0
- Divison by 0 handler
- 1,2,3, current application
- Invalid Address

? Bug in exception handler?
At least somewhere in the kernel

Remark:
Some systems allow two to three nested exceptions, but not more.
Construction Conclusion

Due to these events we need a centralized control instance in the

- Microkernel or
- Kernel

Due to the sensitivity of these events, thread switching and thread controlling need special protection:

- Kernel Mode
- Code and Data inside Kernel Address Space

Let’s study the case:
Current KLT CT has consumed its complete time slice
Objective: Establishing Fair Scheduling

Assumption: No other thread-switching events to be discussed in detail

Simplification: No detailed clock interrupt handling
Simplified Thread Switch

Return from Internal call

Clock Interrupt

Clock Interrupt Handling

Internal Call

Thread Switch

Return from Internal call

Return from ? ? ? to User Mode

Kernel Level

User Level

Current thread

New Current Thread

Again: Just switch the stack pointer
Simplified Thread Switch

- Internal Call
- Return from Internal call
- Thread Switch
- Return from ? ? ? to User Mode
- New Current Thread
- ? ? ? to User Mode
- Clock Interrupt
- Clock Interrupt Handling
- Current thread
- User Level
- Kernel Level
- time
Simplified Thread Switch

Old Current Thread

Clock Intr Handling

Thread Switch

Current Thread

Green → Blue

Thread Switch

Time
Simplified Thread Switch

green → blue

Old Current Thread

Clock Intr Handling

Thread Switch

Current Thread

Clock Intr Handling

Clock Interrupt

Internal call

Thread Switch

???

???

? ? ?

© 2008 Universität Karlsruhe (TH), System Architecture Group
Thread Switch

Simplified Thread Switch

- Clock Interruption Handling
- Thread Switch
- Internal Call
- Clock Interruption Handling

Old Current Thread
- Current Thread

Time

© 2008 Universität Karlsruhe (TH), System Architecture Group
Simplified Thread Switch

Old Current Thread

Current Thread

New Current Thread

Thread Switch

Clock Intr Handling

Clock Intr Handling

Clock Intr Handling

Thread

Internal call

Return to User Mode

time
Simplified Thread Switch

Thread Switch

Old Current Thread

Clock Intr Handling

New Current Thread

Current Thread

Clock Intr Handling

Clock Interrupt

Internal call

Thread Switch

Thread Switch finish

Return to User Mode

time
Simplified Thread Switch

Clock

Thread Switch

Current Thread

Intr Handling

New Current Thread

Return to User Mode

Old Current Thread

Clock Intr Handling

click Intr post

time
Simplified Thread Switch

Clock Intr Handling

Thread Switch

New Current Thread

Old Current Thread

Thread Switch

Return to User Mode

Post

time
Thread Switch

Simplified Thread Switch

Old Current Thread

New Current Thread

Thread Switch

Clock Intr Handling

clock Intr

post

Return to User Mode

post

Return to User Mode

Old Current Thread

New Current Thread

Old Current Thread

New Current Thread

Old Current Thread

New Current Thread

Old Current Thread

New Current Thread

Old Current Thread

New Current Thread

Old Current Thread

New Current Thread

Old Current Thread

New Current Thread
Simplified Thread Switch

Old Current Thread

Clock Intr Handling

Return to User Mode

New Current Thread

© 2008 Universität Karlsruhe (TH), System Architecture Group
Task Address Space

PULTs

TCB1

TCB2

T1

T2

yield()

thread_switch

Task Address Space

KLTs

TCB1

TCB2

T1

T2

T1'

T2'

thread_switch

timeslice

yield()
Thread Switch Implementation

Assumption¹:

- Whenever entering the kernel, i.e. via
  - interrupt
  - exception
  - system call

- the HW automatically pushes SP, IP and status flags, e.g. the user-context of the current thread, e.g. CT = T₁ onto the kernel stack (T₁)

- Kernel stack is implemented in its related TCB, e.g. TCB₁

¹Some processors use shadow register instead of
Current thread T1 is running

Processor

RAM Memory

IP of T1
SP of T1
Flags of T1

TCB T1

Kernel SP

TCB T2

Kernel SP

Thread Switch

Data
CIH Data
CIH Data
Saved IP
Saved SP
Saved Flags

Code T1
Code T2

Stack T1
Stack T2

Note: As long as T1 is running in user mode, the kernel stack is nearly empty. However, at least the start address of the kernel stack is kept in TCB.T1.SP
CT T1 is running in user mode. Clock interrupt saves context of T1 and ...

Thread Switch

Saved IP
Saved SP
Saved Flags
CT T1 is running in user mode. Clock interrupt saves context of T1 and loads context of clock IH.
CT T1 is running in user mode. Clock interrupt saves context of T1 and loads context of clock IH. CIH decides end of time slice for T1 calling thread_switch(T2)
CT T1 is running in user mode. Clock interrupt saves context of T1 and loads context of clock IH. CIH decides end of time slice for T1 calling `thread_switch(T2)`

Save kernel SP(TCB1) and ...

```
Processor
  IP of TSw
  SP of TSw
  Flags of TSw

Memory
  Clock IH
  Thread Switch

TCB T1
  Kernel SP
  Thread Switch
  Data
  CIH Data
  CIH Data
  Saved IP
  Saved SP
  Saved Flags

Code T1
  Stack T1

Code T2
  Stack T2

TCB T2
  Kernel SP
  Thread Switch
  Data
  CIH Data
  CIH Data
  Saved IP
  Saved SP
  Saved Flags
```
CT T1 is running in user mode. Clock interrupt saves context of T1 and loads context of clock IH. CIH decides end of time slice for T1 calling thread_switch(T2). Save kernel SP(TCB1) and ...
CT T1 is running in user mode. Clock interrupt saves context of T1 and loads context of clock IH. CIH decides end of time slice for T1 calling thread_switch(T2).

Save kernel SP(TCB1) and load kernel SP(TCB 2)
CT T1 is running in user mode. Clock interrupt saves context of T1 and loads context of clock IH. CIH decides end of time slice for T1 calling thread_switch(T2). Save kernel SP(TCB1) and load kernel SP(TCB 2)

Next Steps? Complete for yourself.
Additional Design Parameter
Implementation Alternatives

Number of kernel stacks involved:

- 1 Kernel stack for all threads
- Each KLT/process has a kernel stack

Discuss carefully!!
Stack Management

- Each process/KLT has two stacks
  - Kernel stack
  - User stack
- Stack pointer changes when entering/exiting the kernel

Why is this necessary?
Open Questions

- If thread T2 not known in advance ⇒ need for scheduling policy? (see later chapters)
- If we know thread T2, where do we get its TCB? (see exercise)
- If kernel stack is part of TCB ⇒ danger of stack overflow?
- How to handle thread initiation and termination?

Remark:
Limitation on kernel stack size is no real problem in practice.
If your system suffers from a kernel stack overflow
⇒ obvious sign of a severe kernel bug
Examine your kernel design and implementation,
before playing around with increasing kernel stack sizes
Open Questions

*How to handle thread initiation and termination?*

**General remark (Principle of Construction):**

“Solve special cases with the normal-case solution”
Thread Termination

How to terminate a thread?

- Do all necessary work for cleaning up thread’s environment
- Switch to another thread, never return to exiting thread
- No additional mechanisms required
Thread Initialization

What to do, when switching to a brand new thread for the very first time?
Thread Initialization

- Initialize new thread’s (T2) kernel stack with the second part of the thread_switch and the exit function
- Returning from thread_switch leads to second part of system call exit, $\Rightarrow$
  - ‘return’ to T2 in user mode, and
  - start with the first instruction of T2
Idle CPU Problem

What to do, when there is no thread to switch to?

Solution:

Avoid that situation by introducing an idle thread that is always runnable

Question:

Major properties of an idle thread?
Idle Thread

- **When to install?**
  - Before booting
  - While booting
  - After booting

- **How to guarantee that idle thread is always runnable?**
  - Avoid any wait events in the idle thread
    - ...
    - ...
    - ...
    - ...
Summary: Kernel-Level Threads

- All thread management is done by the kernel.
- No thread library, but API to kernel thread facility.
- Kernel maintains TCBs for the task and threads.
- Switching between threads requires kernel.
- Scheduling on thread basis.
Pros/Cons of KLTs

Advantages:
Kernel can simultaneously schedule threads of same task on different processors

A blocking system call only blocks the calling thread, but no other thread from the same application

Even “kernel” tasks can be multi-threaded

Disadvantages:
Thread switching within same task involves the kernel. We have two additional mode switches per thread switch!!

This can result in a significant slow down!!
Summary

Thread-Switching Environment:

- Kernel Entry + Mode Switch (User → Kernel)
- Changing Old Thread State
- Select New Thread (optional)
- **Thread_Switch** (context switch)
- Changing New Thread State
- Kernel Exit + Mode Switch (Kernel → User)

**Remark:** (Only needed for kernel-level threads)
Preview

- Thread Control
- Thread Representation
- Thread Switch
- Thread States orthogonal to Task States
- Dispatching of Threads