System Architecture

5 Threads

Thread Model, Implementation

November 5 2008
Winter Term 2008/09
Gerd Liefländer
Agenda

- Motivation
- Thread Models
- Thread Types
- Problems with Threads
- Controlling Threads
- Implementing Threads (TCB)
Motivation
Single-Threaded Example

- Imagine the following C program:

```c
main() {
    ComputePI("pi.txt");
    PrintClassList("clist.txt");
}
```

- *What behavior?*
Multithreaded Example

- Version of previous program with(out) Threads:

\[
\begin{align*}
\text{main}() \{ \\
    \text{(ComputePI("pi.txt") \};} \\
    \text{(PrintClassList("clist.txt") \);} \\
\}
\end{align*}
\]

- What does “CreateThread” do?
  - Start an independent thread running the function `ComputePI("pi.txt")`
  - Start an independent thread running the function `PrintClassList("clist.txt")`

- How many threads? What behavior?
Memory Footprint of Example

- If we stopped this program and examined it with a debugger, we would see:
  - Three sets of CPU registers
  - Three sets of Stacks

- Problems:
  - How to position stacks?
  - Maximum size of stacks?
  - How to handle stack overflow?
#include <pthread.h>
#include <stdio.h>

void * run (void * d) {
    int q = *((int *) d);
    int v = 0;
    for (int i = 0; i < q; i++) {
        v = v + expensiveComputation(i);
    }
    return (void *) v;
}

main() {
    pthread_t t1, t2;
    int r1, r2;
    pthread_create (&t1, NULL, run);
    pthread_create (&t2, NULL, run);
    pthread_wait (&t1, (void *) &r1);
    pthread_wait (&t2, (void *) &r2);
    printf ("r1 = %d, r2 = %d\n", r1, r2);
}
import java.lang.*;
class Worker extends Thread implements Runnable {
    public Worker (int q) { this.q = q; this.v = 0; }
    public void run() {
        int i;
        for (i = 0; i < q; i++) { v = v + i; }
    }
    public int v;
    private int q;
}

class Example {
    public static void main(String args[]) {
        Worker t1 = new Worker (100);
        Worker t2 = new Worker (100);
        try {
            t1.start();
            t2.start();
            t1.join();
            t2.join();
        } catch (InterruptedException e) {} 
        System.out.println ("r1 = " + t1.v + ", r2 = " + t2.v); 
    }
}
Classifying “Threaded” Systems

- One or many address spaces
- One or many threads per AS

© 2008 Universität Karlsruhe (TH), System Architecture Group
Processes versus Task

(a) 3 processes with 1 main thread
(b) 1 task with 3 threads
Potential Benefits of Threads

- **Responsiveness**
  - Reaction time on external or internal events

- **Convenience of programming**
  - Special threads for different activities
  - Install appropriate attributes for a thread
    - Access rights
    - Scheduling info

- **Economy**
  - Resource Sharing
  - Cheaper creation and deletion

- **Better utilization of SMPs**
  - Heavily depending on thread model
Thread Model
Abstract Thread Model

Remark: In whatever thread model, each thread has its own user stack.
Common Properties of Thread Models

- Code and data regions of a thread are **always** located in the user space of the surrounding task
  - Code of a thread can be located in the shared code segment of the task or in a private code segment

- A user stack is always private to its thread, i.e. **no other thread** of the same AS can access its contents with a usual stack operation, **BUT** ....
  - it can violate the others user stacks contents via pointer operations, when it can guess the stack addresses of the other user stacks

- Thread data can be part of the public data region (e.g. heap, global variables)
Word “processor” with three threads:

- Controlling keyboard input and updating word document
- Displaying current content
- Updating word document on the disk
Multithreaded Application (2)

A multithreaded Web server

Thread Model

User Address Space

Web server process

Dispatcher thread

Worker thread

Web page cache

Network connection

Kernel

User space

Kernel space

Shared resource
Implementation of Threads

- **Thread Control Block (TCB)** contains
  - Execution Context
    - CPU registers, instruction pointer, stack pointer
    - HW specific
  - Scheduling Info
    - Scheduling State (more later), priority, estimated CPU time
    - Policy specific
  - Resource Usage Info
    - Already used CPU time
    - HW & OS specific
  - Various Pointers:
    - Implementing thread states, pointer to enclosing TaskCB
    - Implementation specific
  - ...

Thread Model
Task versus Thread

- **Items shared by all threads of the same task in most thread models**
  - Address space
  - Global variables
  - Open files
  - Children (what type of?)
  - Pending alarms
  - Signals and signal handlers
  - Accounting information

- **Items private to each thread**
  - Instruction pointer ("program counter")
  - Registers, flags etc. ⇒ context of thread
  - Stack pointer & stack (user stack, kernel stack?)
  - Thread state (external state)
Thread Models

- Pure User-Level Threads (PULT*)
  - Known only outside the kernel (e.g. within the task, or subsystem, or runtime system), often implemented by a thread library, i.e. its TCBs are located inside user space

- “Pure” Kernel-Level Threads (KLT*)
  - Every KLT is explicitly known to the kernel, its TCB (at least parts of it) is located inside the kernel

- Hybrid Threads (HLT)
  - Take advantage of both pure models

*often called user threads, user-space threads, green threads …

*This term is KLT specific, also called native threads (or even kernel threads, which is completely misleading)
Types of Threads

- Kernel-Level Threads (KLT)
- Pure User-Level Threads (PULT)
- Hybrid-Level Threads (HLT)
- Kernel(-Mode) Threads (KMT)

Examples
Implementing KLTs

Types of Threads

- Task
- Thread
- TCBs of all known kernel-level threads
- Task CBs of 2 tasks
- Task table
- Thread table
Management of KLTs

- Kernel knows about tasks and the KLTs
  - Threads of type KLT are objects of CPU scheduling
  - TaskCB contains info on shared resources, AS + set of its threads
  - TCB contains info on CPU context, state + task affiliation
  - TCB might be moved between thread states

- TCBs can be made smaller than a TaskCB, e.g.
  - Linux TaskCB has 106 fields
  - Linux TCB would only require 24 fields (*but????*)
Advantages of KLTs

- KLTs from the same task can be assigned to different CPUs on a SMP (⇒ real parallelism)
  - Always a speed up?
  - Speed up even on a single processor?

- A blocking system call only blocks the calling KLT (not the complete task)

- A `thread_switch` between 2 KLTs of the same task is faster than ~ between 2 KLTs of different tasks, because you do not need an AS-Switch with
  - Switching paging information
  - TLB flushing & restoring
Disadvantages of KLTs

- Each thread related operation requires overhead, it always needs kernel entrance/kernel exit
- Relatively high initialization costs
- All KLTs must live with the more or less flexible kernel scheduling policy
  - ∃ few systems with customizable kernel scheduling policies
- Besides the entities “process” and “task” the kernel also has to know about threads
  - Kernel is more complicated
  - More kernel space is needed for TaskCBs + TCBs
Examples Systems offering KLTs

- Windows NT/XP/2000
- Solaris 9 and later versions
- Tru64 UNIX
- BeOS
- MacOS X
- Linux
  - Implements the Posix 1003.1c package
  - Using the \texttt{clone()} system call
  - To take full advantage of SMPs
When to use KLTs?

What parts of a program should or can be threaded?

- A few rules of thumb:
  - Groups of lengthy or special operations, e.g.
    - painting a window
    - printing a doc
    - responding to a mouse-click
    - calculating a spreadsheet column
  - Amount of shared data is relatively small, i.e. mutually obstruction neither occurs too often nor too long
  - Preview: However, you should be prepared to worry about deadlocks and race conditions
Pure User-Level Threads

Types of Threads

Java green threads (classic VM)
Controlling PULTs

All **PULT related operations** are done inside the thread library, i.e. in user mode land

- **Advantages:**
  - Fast thread manipulation, typically 10-100x faster than slipping into the kernel and back again
  - UTCB can be smaller (e.g. IP, PSW, and SP)
  - Each application might use a tailored scheduling policy for its PULTs
  - Usable in an OS that does not offer KLTS (often called native threads)
Controlling PULTs

- **Disadvantages:**
  - Only one single PULT of a multi-threaded task can run on an SMP at any given time.
  - A task always gets the same amount of the CPU independently of the number of its PULTs.
  - A blocking system call (e.g. synchronous I/O) blocks the entire task, not only the calling PULT.
  - Some say: Kernel might preempt a task with a PULT holding a system wide lock (thus slowing system down).
Analysis

- As long as the kernel is involved in granting a lock to a PULT it can notify this fact in the related TaskCB (e.g. setting a "non preemption flag")

- Whenever the kernel wants to preempt a task, it can control whether the "non preemption flag" of the corresponding task is set

- But, assume a shared memory concept and a common lock - handled in user land- for more than 1 AS

- Then the kernel might preempt a PULT that holds a lock, some other PULTS from the related ASes are waiting for

  \[\rightarrow\text{increased latency}\]
CROSS-AS-LOCK at User Level

AS1

AS2

AS3

SHARED MEMORY

User Level Lock

Kernel Scheduler
Hybrid Threads

Map user-level threads to “kernel-level threads”
Example: Solaris 2 Threads

Types of Threads

m:n mapping of ULTs to KLTs

kernel thread

user-level thread

lightweight process

kernel

CPU
Cons of Hybrid Thread Model*

- OS must offer native threads
- 2 scheduling layers
  - The “KLT part” of the application is managed by the kernel
  - The user-level part is managed by a scheduler in the library
  - Kernel scheduler and library scheduler have to cooperate

**Problem:**
Fundamental reliance on kernel (lower layer) calling procedures in user space (higher layer) via so called “UPCALLS”, violating the principle of layered systems

Pros of Hybrid Thread Model* 

- Two ways to handle blocking system calls 
  1. A blocking system call in the user-level thread only blocks its counterpart in the kernel 
    - Only those user-level threads are blocked which are mapped to this blocked KLT 
    - When the blocking criteria no longer holds, the kernel will unblock the related KLT as usual
Pros of Hybrid Thread Model*

2. Kernel blocks current KLT temporarily, recording the relevant blocking criteria in the kernel

- But instead of **blocking the KLT for a while**, it returns control to the user-level scheduler (via a scheduler activation)
- This user-level scheduler blocks the responsible ULT and can switch to another **ready ULT** which has been planned for this KLT
- When the blocking situation no longer holds, the kernel informs the corresponding user-level-scheduler, which then can unblock the related user-level thread and can put it to the ready list for its KLT
Hybrid Thread Blocking

Types of Threads

One way to handle the above situation are scheduler activations (see: Anderson et al)
PThreads

- POSIX standard (IEEE 1003.1c) API for thread creation and synchronization.

- API specifies behavior of the thread library; implementation is up to the developer of the library.

- Common in UNIX operating systems.
K42 Hybrid Threads

Types of Threads

Address Space

Dispatcher

User-Level Threads

User-Level Threads

User-Level Threads

User-Level Threads

Dispatcher

CPU Domain

Dispatcher Descriptor

Scheduling Class 1

Scheduling Class 2

User Space

Kernel Space

© 2008 Universität Karlsruhe (TH), System Architecture Group

1 read the white papers: http://www.research.ibm.com/K42/
Kernel (-Mode) Threads (KMT)

- Yet another type of threads?
- Just to confuse you completely?

Always executing in kernel-mode

Pure user-level threads or "green threads"

Kernel-level threads or native threads
Kernel-Mode Threads

- Already in early Unix versions, some kernel "processes" (daemons, e.g. the swapper completely mapped to KAS) and always running in kernel-mode
- **root** = owner of UNIX kernel processes
- Modern OS may use kernel-mode threads

**Additional problems:**
- *How (when)*to schedule kernel-mode threads?
- *How to interact with*kernel-mode threads?
- *How to protect sensitive kernel data from misbehaving kernel-mode threads?*
Thread Programming

Concurrent Programming

Examples of Problems

Common Problems
**Why to study Thread Programming?**

- Concurrent correct application programs are not yet very widespread

- Some of them have been (still are) **error-prone**\(^1,2\) ⇒

- Skill of concurrent programming has to be trained

- Famous Mars Pathfinder problem
  - Problems with the correct synchronization of three concurrent processes

\(^1\)For further software horror stories see: 
http://www.cs.tau.ac.il/~nachumd/horror.html

\(^2\)http://courses.cs.vt.edu/~cs3604/lib/Therac_25/Therac_1.html
What can go **wrong** with Threads?

- Safety hazards
  - Program does the wrong thing due to *race conditions*

- Liveliness hazards
  - Program never does the right thing (*live lock, deadlock*)

- Performance hazards
  - Program is **too slow** due to excessive synchronization
Preview: Thread Scheduling

- Scheduler decides when to run a “ready” thread

![Diagram showing potential execution sequence of threads T1, T2, T3]

- Programs should make no assumption about the scheduler
  - Scheduler is a “black box”
Threads Safety

- A program is thread safe if it behaves correctly when accessed from multiple threads, regardless of the scheduling or interleaving of those threads.

- **Race condition**: program’s output is different depending on scheduler’s interleaving.
  
  - Such a behavior is a program bug.
  
- Study the following program example on your own.
1. public class ThreadTest extends Thread {
2.  private static int x = 1;
3.  public void run() {
4.      x++;
5.  }
6.  public static void main(String[] args) {
7.      thread t1 = new ThreadTest();
8.      thread t2 = new ThreadTest();
9.      t1.start();
10.     t2.start();
11.     try {
12.         t1.join; // wait for t1 to finish
13.         t2.join; // wait for t2 to finish
14.     }
15.     catch(InterruptedException iex) {} 
16.     printf("Value of x == %d", x);
17.  }
18.}
Safety Hazard: $x++$

- Key point: $x++$ is not an atomic instruction, but consist of multiple CPU instructions, e.g.
  - Load $x$ from memory
  - Increment $x$
  - Store $x$ to memory

- In MIPS assembly this could mean:
  - `LW $t, offset($s)`
  - `ADDI $t,$t,1`
  - `SW $t, offset($s)`
Unsafe Thread Schedule

- Given threads t1 and t2:

  ```
  LW $t, offset($s)     //for t1
  LW $t, offset($s)     //for t2
  ADDI $t,$t,1          //for t2
  SW $t, offset($s)     //for t1
  ADDI $t,$t,1          //for t1
  SW $t, offset($s)     //for t1
  ```

- Final result: Value of x == 2
Thread Cancellation

- “Killing” a thread before it has finished
- Two general approaches:
  - Asynchronous cancellation terminates the thread immediately
  - Deferred (lazy) cancellation allows the thread to periodically check if it should be cancelled, e.g. whenever control flows enters the kernel
Signal Handling

- Signals are used in UNIX systems to notify a process that a particular event has occurred

- *How to signal PULTs or KLTs of a task?*

- A signal handler is used to process appropriate actions after the signal has arrived
  1. Signal is generated by a particular event
  2. Signal is delivered to a task/process
  3. Signal is handled

- Design options:
  - Deliver the signal to the thread to which the signal applies
  - Deliver the signal to every thread in the task
  - Deliver the signal to certain threads in the task
  - Assign a specific thread to receive all signals for the task
Thread Pools

- Create a number of “worker threads” in a pool where they are waiting for work

- You might need different pools of worker threads

  **Advantages:**
  - Usually slightly faster to service a request with an existing waiting thread instead of creating a new pop-up thread before starting the service
  - Allows the number of threads in the application(s) to be bound to the size of the pool

  **Disadvantage:**
  - Some of these worker threads might never be needed
Problems: Programming Threads

Types of Threads

Conflicts between threads using a global variable, e.g.
Unix's errno (contains error code of current system call)
Thread Control Block (TCB)
Thread Representation

Implementing thread control, we need a data structure representing a thread, i.e. something representing the existence of a thread including its

\[ \text{TID} = \text{thread identifier} \] (see our passport)

Why and when do we really need a TID?

Hint: Compare a TCB with your passport

- Crossing borders
- Controlling/arresting suspicious people
- Additional unique and non forgeable attributes?
Thread Representation

TID all together with other useful information characterizing a thread is collected within the thread control block or **TCB**

**Definition:**
A thread attribute describes or characterizes a thread

**Example:**
A thread intensively using I/O is called “I/O-bound”

**Remark:**
Different systems might have different attributes in their TCBs
TCBs can vary concerning size and/or internal structure
Implementing Threads

Minimal TCB

- Thread Identifier (TID)
- Instruction Pointer (IP)
- Stack Pointer (SP)
- Status Flags (SF)

\[ <IP, SP, SF> = \text{minimal context of a thread} \]

Don’t say program counter!!!
Potential TCB Attributes

- Thread identifier TID
- Context related
  - User registers
  - Kernel registers
- Scheduling related
- Stack related
- Additional private “resources”
  - private global data
Thread Control Information

• User-Visible Registers
  - Stack-Related
  - General Purpose
  - Floating-Point
  - Index

• Control and Status Registers

Stacks are used to support procedure and system calls (establishing local variables, transfer of parameters etc.)

A stack pointer points to the top of the stack.
Thread Control Information

• User-Visible Registers
  - Stack-Related
  - General Purpose
  - Floating-Point
  - Index

• Control and Status Registers

  Instruction Pointer (address of next instruction)

  Condition Codes  (results of previous instruction, e.g. equal bit, overflow bit)

  Status Information (execution mode etc.)
Thread Control Information

- Events related to a thread’s execution, e.g. waiting for a specific I/O result (⇒ scheduling)
  
- Priorities

- Inter-Process Communication (IPC)

- Mapping Information (task-specific)

- Current resource holder (e.g. lock holder)

- Resource ownership and utilization (task-specific)
Implementing Sets of TCB’s

Implementation techniques

- Contiguous table of TCB’s (thread table)
  - Real array
  - Virtual array

- Structured list of TCB’s

Hint:
Considering different system constraints, discuss appropriate data structures for implementing a structured list of TCBs
Traditional Location of KLT TCBs

User-level land

Kernel-level land

Kernel

μ-Kernel

respectively in a
Literature

Bacon, J.: Operating Systems (PI, 4)
Stallings, W.: Operating Systems (3, 4)
Silberschatz, A.: OS Concepts (2)
Tanenbaum, A.: MOS (2)

- http://linuxdevices.com/articles/AT6753699732.html
- http://www.gridbus.org/~raj/asc98.html