1 System & Activities

Gerd Liefländer

28. Oktober 2008
System Architecture Group
Roadmap for Today & Next Week

- System Structure
  - System Calls
  - (Java) Virtual Machine

- Basic System Abstractions
  - Address Space
  - Activities
    - Procedures
    - Process, Task
    - Threads
      - Kernel Level Threads
      - User Level Threads

- Assignment Hints

- OS Kernels
  - Monolithic
  - Micro
System Structure

Layered Systems
Privileged OS Kernel
System Interface
System Layers

... what does this ordering imply? Is it a strict layering? What is at the top or at the bottom?

higher abstraction

lower details

End User

Application Programs

Utilities

Operating-System

Computer Hardware

Programmer

Operating-System Designer

higher abstraction

lower details
Major System Components

Application

What happens here?

Operating System Kernel

Hardware

System Structure
The Privileged OS (Kernel)

- Applications should not be able to bypass the OS (apart from the non-privileged CPU instructions)
  - OS can enforce the extended machine
  - OS can enforce its resource management
  - OS prevents applications from interfering with each other
- Some embedded OSes (e.g. PalmOS) do not have privileged components
System Calls

- OS supplies its functionality via \textit{system calls}
- System calls form a well defined interface (API) between applications and OS
  - Applications only need to know these system calls in order to get the requested service from the kernel
  - \textit{How is a system call implemented?}
  - Via a specific, but \textit{non privileged} instruction:
    - \texttt{trap}
    - \texttt{int}
  - The trap instruction needs a specific parameter indicating the target IP within the kernel
  - To enable some control this parameter must be transferred within a \textit{predefined register}
Interrupt Vector Table

Index 3
pagefault
Exception & Interrupts

Index 240
gettimeofday
System calls
OS as a Privileged Component

The System API is often hidden within a user level library, e.g. the Java API

Typical system calls?
# Linux System Calls for Processes

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pid = fork()</code></td>
<td>Create child process</td>
</tr>
<tr>
<td><code>pid=waitpid(pid, &amp;statloc, options)</code></td>
<td>Wait for a child to terminate</td>
</tr>
<tr>
<td><code>s = execve(name, argv, environp)</code></td>
<td>Replace a process’ core image</td>
</tr>
<tr>
<td><code>exit(status)</code></td>
<td>Terminate execution and return status</td>
</tr>
</tbody>
</table>
## Linux System Calls for Files

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fd = open(file, how, …)</code></td>
<td>Open file for reading, writing, or both</td>
</tr>
<tr>
<td><code>s = close(fd)</code></td>
<td>Close an open file</td>
</tr>
<tr>
<td><code>n = read(fd, buffer, nbytes)</code></td>
<td>Read data from a file into a buffer</td>
</tr>
<tr>
<td><code>n = write(fd, buffer, nbytes)</code></td>
<td>Write data from a buffer into a file</td>
</tr>
<tr>
<td><code>position = lseek(fd, offset, whence)</code></td>
<td>Move the file pointer</td>
</tr>
<tr>
<td><code>s = stat(name, &amp;buf)</code></td>
<td>Get the file’s status information</td>
</tr>
</tbody>
</table>
## Linux System Calls for Directories

### Directory Management

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>s = mkdir(name, mode)</code></td>
<td>Create a new directory</td>
</tr>
<tr>
<td><code>s = rmdir(name)</code></td>
<td>Remove an empty directory</td>
</tr>
<tr>
<td><code>s = link(name1, name2)</code></td>
<td>Create new entry name2 → name1</td>
</tr>
<tr>
<td><code>s = unlink(name)</code></td>
<td>Remove a directory entry</td>
</tr>
<tr>
<td><code>s = mount(special, name, flag)</code></td>
<td>Mount a file system</td>
</tr>
<tr>
<td><code>s = umount(special)</code></td>
<td>Unmount a file system</td>
</tr>
</tbody>
</table>
# System Calls for Miscellaneous Tasks

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>s = chdir(dirname)</code></td>
<td>Change the working directory</td>
</tr>
<tr>
<td><code>s = chmod(name, mode)</code></td>
<td>Change a file’s protection bits</td>
</tr>
<tr>
<td><code>s = kill(pid, signal)</code></td>
<td>Send a signal to a process</td>
</tr>
<tr>
<td><code>seconds = time(&amp;seconds)</code></td>
<td>Get elapsed time since Jan. 1, 1970</td>
</tr>
</tbody>
</table>
Interdependencies

Application

Interaction via a function call to a library procedure

System Libraries (e.g. API)

Interaction via System Calls

Operating System Kernel

Hardware
Nested Layered System Structure

Structure of a Virtual Machine on top of the OS kernel

System Structure

Java application
Java Application Interface
Java Runtime Environment

Terminal driver
Process manager
Memory manager
Communication Software
Network Driver
Basic System Terms

Address Space,
Process, Thread, Task,
Thread Types
2 Main Abstractions within Systems

1. How to install „information processing“, i.e. activity „when“ to execute „what“ code

   ⇒ activity, e.g. thread (process*)

2. How to install „protected code and data depositories“, i.e. ”where“ to store ”what“ software entities

   ⇒ address space

*Note: Notion “process” ← “procedere” = “voranschreiten”
Notion “thread” ~ “Faden abwickeln”
Design Parameters for Address Spaces

- Number of data entities
- Boundary checks
- Types of buffers (stack, heap, file, ...)
- Security of data entity (object, protection domain)
- Duration of data entity (volatile/temporary/persistent)
- An address space (AS) provides a *protected domain* for an activity, i.e. an executing program
1. In most cases caller & callee belong to same AS
2. Either caller or callee are running
Why Processes/Threads?

- Suppose your system offers a software tool, enhancing the way how you can edit, compile, and test your programs.

- If this tool allows concurrent editing, compiling, and testing, \( \Rightarrow \) this tool could reduce your work a great deal.

\( \Rightarrow \) Processes/threads help to manage concurrent activities.
Design Parameters for Activities

- Number of activities
  - Static
  - Dynamic
- Types of activities
  - Foreground
  - Background
- Urgency of activities
  - Real time
    - Hard real time
    - Soft real time
  - Interactive
  - Batch
- Degree of interdependency
  - Isolated
  - Dependant
- ...
Thread

- Basic entity of pure activity
- Object of scheduling
  - Internal scheduling in the kernel
  - External scheduling in a runtime system

Basic characteristics of a thread?
Characteristics of Threads

- Protected domain
  - The kernel address space is domain for all kernel threads
  - A user address space is domain for all threads of this application, i.e. each application has its own user address space

- Code
- Instruction pointer
- Stack
- Stack pointer
- Thread control block TCB
Additional Attributes of Threads

- Internal state (context)
- External state (running, ready, waiting, ...)
- Priority
- Creation time
- Start time
- Deadline
- Waiting time
- Exit time
Process

- Single threaded
- Address space (Unix terminology)
- Additional resources

Address space = protected area
Task

- Entity of an “application” consisting of
  - $t \geq 1$ thread(s)
  - Address space
  - Resources

![Diagram showing 1 or more threads]
The Activity Models

- Process Mode
- Procedure versus Thread
- Process versus Task
- Shared Memory
- Java Threads
Multiprogramming of 4 programs, each program is located in an extra address space

Conceptually 4 independent, *sequential* processes

However, on a single processor only one process is running at any instant
Procedure vs. Thread

Assumption: Given program with a simple procedure call to compute data needed for the program to progress

Program

Procedure()

call()

Procedure()

return()

How does return find the fitting return address?
Assumption: Given program with two threads, one computes data that the other thread needs for its progress

Thread 1

Wait_For_Data() -> Provide_Data()

Thread 2

Data is stored in an independent object both threads have access to

does something else ...
Thread Model

Thread

- Thread = abstraction for a pure activity (e.g. being executed on a CPU) ⇒

- Thread includes code and private data (e.g. a stack)

- A thread may also need some environment
  - Address space
  - Files, I/O-devices and other resources
  - It may even share this environment with other threads

Example: A file server may consist of \( t \) identical threads, each thread serving only one client’s request.
Process versus Task Model

(a) Three processes (each task with only one thread)
(b) One task with three threads
Process versus Task

Process model
- create and delete need more
  - time
  - space, e.g.
    new address space
- Cooperation via IPC or shared memory (→)

+ well-separated from each other

Thread model
- Might destroy each others data

+ create and delete need less
  - time
  - space, e.g. only new
    - stack and TCB

+ easier to work together on common data
Shared Memory (0)

Process 1
- Stack 1
- Data 1
- Code 1

Process 2
- Stack 2
- Data 2
- Code 2

IPC Queue

Kernel
Shared Memory (1)

Process 1
(producer)

Stack 1

Shared Memory

Data 1

Code 1

Process 2
(consumer)

Stack 2

Shared Memory

Data 2

Code 2

Shared Memory Management

Kernel
Shared Memory (2)

Process 1
(producer)

Stack 1
Shared Memory
Data 1
Code 1

Process 2
(consumer)

Stack 2
Shared Memory
Data 2
Code 2

Kernel
Shared Memory (3)

Task
(producer/consumer)

Stack 1
Stack 2
Shared Memory
Data 2
Data 1
Code 2
Code 1

Kernel
Thread Life-Cycle in Java

The predicate `isAlive()` can be used to test if a thread has been started but not terminated. Once terminated, it cannot be restarted.

`new Thread()` causes the thread to call its `run()` method.

`stop()`, or `run()` returns
Thread Alive States in Java

Once started, an *alive* thread has a number of substates:

- **Running**
  - Transitioned from the *Runnable* state by the `start()` method.
  - Can transition to the *Runnable* state by the `yield()` method.
  - Can transition to the *Non-Runnable* state by the `sleep()` or `wait()` methods.

- **Runnable**
  - Transitioned from the *Non-Runnable* state by the `dispatch` method.

- **Non-Runnable**
  - Transitioned from the *Runnable* state by the `notify()` method.
Thread Models

Pure User Level
Kernel Level
Hybrid
Types of Threads

- **Kernel Level** Threads (KLT)
  - Known to the system wide thread management *implemented* inside the kernel, i.e. the corresponding TCBs are located inside the kernel

- **User Level** Threads (PULT)
  - Known only within one task or one sub system, often implemented by a thread library, i.e. the corresponding TCBs are located inside an instance of the thread library, i.e in user-land

*This notion is KA-specific*
Kernel Level Threads

Types of Threads

TCBs of all known kernel level threads

TaskCBs of 2 tasks

Task table

Thread table
Kernel Level Threads

- Supported by the Kernel

- Examples
  - Windows 95/98/NT/2000
  - Solaris
  - Tru64 UNIX
  - BeOS
  - Linux
User Level Threads

Types of Threads

Diagram showing the relationship between tasks, threads, user space, kernel space, run-time system, thread table, and task table.
User Level Threads

- Thread management done by user-level thread library

- Examples
  - POSIX  \textit{Pthreads}
  - Mach  \textit{C-threads}
  - Solaris  \textit{threads}
Analysis of Kernel-Level Threads

**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

**Inconveniences:**
Thread switching within same task involves the kernel. We have 2 mode switches per thread switch!!

Discuss this very carefully
Analysis of User-Level Threads

**Advantages**
- Thread switch does not involve the kernel: ⇒ no mode switching
- Scheduling policy can be application specific: ⇒ best fitting policy
- PULTs can run on any OS, if there is thread library

**Inconveniences**
- Many system calls are blocking, ⇒ all threads of the task will be blocked
- Kernel can only assign tasks to processors ⇒
- 2 pure user level threads of the same task can never run on two processors simultaneously
OS Kernels
What’s Inside a Kernel?

Depends on the type of kernel

- Monolithic Kernel (traditional approach)
  - Lot of things, e.g.
    - File system
    - Network stack
    - Device Driver
    - Memory management

- Microkernel (our view)
  - Only what’s needed
  - 2 major system abstraction + IPC mechanism
Monolithic Kernel

Documents, stacks & heaps, symbols, windows, variables, coroutines, statements, procedures, arrays & structures.

Linux

Address space, socket, file, event, semaphore, priority, region, process, IPC, task, thread, mutex, page, monitor, schedule.

Hardware

Bit, byte, word, TLB, instruction, register, interrupt.
Microkernel

documents  stacks & heaps  modules
threads symbols windows
variables coroutines statements
arrays & structures procedures

App
TCP/IP
EXT2
NetDrv
IDEDrv
L4 μ-kernel
Hardware

Multi-Server System

address space  thread

bit

instruction

byte

word

Monolithic System

Application documents
windows
threads
coroutines
symbols
variables
stacks & heaps
statements
arrays & structures
procedures

Arrays & Structures

Variables

Symbols

Coroutines

Windows

Threads

Documents

Modules

Stacks & Heaps

Microkernel

TCP/IP

EXT2

Net Drv

IDE Drv

L4 µ-kernel

Hardware

Server

Mutex

Socket

File

Address Space

Thread

Bit

Instruction

Byte

Word

Register

© 2006 Universität Karlsruhe (TU), System Architecture Group