# Zero Kernel

Design and Implementation
DRAFT 1, Revision 7

Tuomo Petteri Venäläinen

August 16, 2013

# Part I

# Overview

# Contents

# Chapter 1

# Preface

## 1.1 Acknowledgements

## 1.2 History

# Chapter 2

# System Concepts

## 2.1 Terminology

**Trap**

A trap is a hardware- or software generated event. Other names for traps include **interrupts**, **exceptions**, **faults**, and **aborts**. As an example, keyboard and mouse input may generate interrupts to be handled by interrupt service routines (**ISR**s).

**Process**

A process is a running instance of a program. A process may consist of several threads. Threads of a process share the same address space, but have individual execution stacks.

**Thread**

Threads are the basic execution unit of programs. To utilise multiprocessor-parallelism, a program may consist of several threads of execution, effectively letting it do several computation and I/O operations at the same time.

**Task**

In the context of Zero, the term task is synonymous the term process.

**Interval Task**

Short-lived, frequent tasks such as audio and video buffer synchronisation are scheduled with priority higher than normal tasks. It is likely such tasks should be given a slice of time shorter than other threads.

**Virtual Memory**

Virtual memory is a technique to map physical memory to per-process address space. The processes see their address spaces as if they were in total control of the machine. These per-process address spaces are protected from being tampered by other processes. Address translations are hardware-level (the kernel

mimics them, though), so virtual memory is transparent to application, and most of the time, even kernel programmers.

**Page**

A page is the base unit of memory management. Hardware protection works on per-page basis. Page attributes include read-, write-, and execute-permissions. For typical programs, the text (code) pages would have read- and execute-permissions, wherease the program [initialised] data pages as well as [unitialised] BSS-segment pages would have read- and write- but not execute-permissions. This approach facilitates protection against overwriting code and against trying to execute code from the data and BSS-segments.

**Segment**

Processes typically consist of several segments. These include a text segment for [read-only] code, a data segment for initialised global data, a BSS-segment for uninitialised global data, and different debugging-related segments. Note that the BSS-segment is runtime- allocated, whereas the data segment is read from the binary image. Also Note that in a different context, segments are used to refer to hardware memory management.

**Buffer**

Buffers are used to avoid excess I/O system calls when reading and writing to and from I/O devices. This memory is allocated from a separate buffer cache, which can be either static or dynamic size. Buffer pages are 'wired' to memory; they will never get paged out to disk or other external devices, but instead buffer eviction leads to writing the buffer to a location on some device; typically a disk.

**Event**

Events are a means for the kernel and user processes to communicate with each other. As an example, user keyboard input needs to be encoded to a well known format (Unicode or in the case of a terminal, ISO 8859-1 or UTF-8 characters) and then dispatched to the event queues of the listening processes. Other system events include creation and destroyal of files and directories.

Zero schedules certain events on timers separate from the timer interrupt used for scheduling threads. At the time of such an event, it is dispatched to the event queues of [registered] listening processes; if an event handler thread has been set, it will be given a short time slice of the event timer. Event time slices should be shorter than scheduler time slices to not interfere with the rest of the system too much.

# Chapter 3

# System Interface

## 3.1 UNIX Features

### Concepts

Zero is influenced and inspired by AT&T and BSD UNIX systems. As I think many of the ideas in these operating systems have stood the test of time nicely, it feels natural to base Zero on some well-known and thoroughly-tested concepts.

### Nodes

Nodes are similar with UNIX 'file' descriptors. All I/O objects, lock structures needed for IPC and multithreading, as well as other types of data structures are called nodes, collectively. Their [64-bit] IDs are typically per-host kernel **memory addresses** (pointer values) for kernel **descriptor data structures**.

## 3.2 POSIX Features

### Threads

Perhaps the most notable POSIX-influenced feature in Zero kernel is threads. POSIX- and C11-threads can be thought of as light-weight 'processes' sharing the parent process address space but having unique execution stacks. Threads facilitate doing several operations at the same time, which makes into better utilisation of today's multicore- and multiprocessor-systems.

## 3.3 Zero Features

### Events

Possibly the most notable extension to traditional UNIX-like designs in Zero is the event interface. Events are interprocess communication messages between kernel and user processes. Events are used to notify of user device (keyboard,

mouse/pointer) input, filesystem actions such as removal and destroyal of files or directories, as well as to communicate remote procedure calls and data between two processes (possibly on different hosts).

Events are communicated using message passing; the fastest transport available is chosen to deliver messages from a process to another one; in a scenario like a local display connection, messages can be passed by using a shared memory segment mapped to the address spaces of both the kernel and the desktop server.

# Part II

# Basic Kernel

# Chapter 4

# Kernel Layout

### Monolithic Kernel

Zero is a traditional, monolithic kernel. It consists of several parts, some of which are highlighted below.

| Module | Operation |
|---|---|
| **tmr** | hardware timer interface |
| **thr** | thread scheduler |
| **vm** | virtual memory manager |
| **page** | page daemon |
| **mem** | kernel memory allocator |
| **io** | I/O primitives |
| **buf** | block/buffer cache management |

The code modules above will be discussed in-depth in the later parts of this book.

# Chapter 5

# Kernel Environment

This chapter describes the kernel-mode execution environment. Hardware-specific things are described for the IA-32 and X86-64 architectures.

## 5.1 Processor Support

### 5.1.1 Thread Scheduler

#### 5.1.1.1 Thread Data Structure

### 5.1.2 Interrupt Vector

The interrupt vector is an array of interrupt descriptors. The descriptors contain interrupt service routine base address for the function to be called to handle the interrupt.

#### 5.1.2.1 Interrupt Descriptors

Entries in the interrupt vector, i.e. interrupt descriptor table (**IDT**), are called interrupt descriptors. These descriptors, whereas a bit hairy format-wise, consist of interrupt service address, protection ring (system or user), and certain other attribute flags.

## 5.2 Memory

### 5.2.1 Overview

### 5.2.2 Segment Descriptor Tables

#### 5.2.2.1 Segment Descriptors

### 5.2.3 Paging Data Structures

#### 5.2.3.1 Page Directory Entry

#### 5.2.3.2 Page Table Entry

#### 5.2.3.3 Page Directory

#### 5.2.3.4 Page Tables

### 5.2.4 Page Daemon

#### 5.2.4.1 Page Replacement Algorithm

## 5.3 User Environment

### 5.3.1 Memory Map

| Segment | Brief | Parameters |
|---|---|---|
| stack | process stack | read, write, grow-down |
| map | memory-mapped regions | read, write |
| heap | process heap (sbrk()) | read, write |
| bss | uninitialised data | read, write, allocate |
| data | initialised data | read, write |
| text | process code | read, execute |

**Notes**

- memory regions are shown from highest to lowest address
- the stack segment grows downwards in memory
- the BSS segment is allocated at run-time
- segments are shown in descending address order

# Chapter 6

# IA-32 Implementation