Zero Volume 0 - The Kernel

Design and Implementation

DRAFT 1, Revision 9

Tuomo Petteri Venäläinen

August 21, 2013

Part I

Overview

Contents

Chapter 1

Preface

Goal

The goal of the Zero project is a new, portable, high performance, UNIX-inspired operating system. Such systems typically consist of a [relatively] small kernel and supporting user software such as editors, compilers, linkers and loaders, and other software development tools.

Rationale

Whereas different UNIX-like operating systems are doing strong for many users, the world is a different place from when UNIX and some of the related operating systems had their initial designs laid out. We have outstanding graphics (and physics) processors, high quality audio interfaces, lots of memory, plenty of disk space, and so forth. Also, the trend is leaning towards multiprocessor systems with new requirements and possibilities. I think and feel it's worth designing a new operating system for modern computers.

Design

Zero is a multithreaded multiprocessor-enabled kernel. Design goals include fast response to user actions as well as high multimedia performance.

1.1 Acknowledgements

Contributors

At the moment, I have kept Zero a one-man project on purpose. I feel it's too much of a moving target to spend other people's time working on things that may change any moment. I will do my best to bring Zero ready for others to work on - I have been offered help, and I want to thank you guys (who know yourself) here. :)

Open Source Community

First and foremost, I want to thank **the developers** of open and free software for their work and courage to release their work for others to use and modify. Keep the spirit strong!

TODO: thanks and greetings etc.

1.2 Background

Zero has its roots in old, still simple and elegant versions of the UNIX operating system. I still see many good, timeless things about the UNIX design worth reusing in a new operating system. One thing of particular attraction is the ëverything is a filep̈hilosophy; I plan to use and possibly extend that idea in Zero.

Chapter 2

System Concepts

This chapter is a glossary of some terminology used throughout the book.

2.1 General Terminology

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.

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.

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.

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.

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.

Task

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

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.

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).

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.

2.2 X86 Terminology

APIC

APIC stands for [CPU-local] 'advanced programmable interrupt controller.'

GDT

A GDT is 'global descriptor table', a set of memory segments with different permissions.

HPET

HPET is short for high precision event timer (aka multimedia timer).

IDT

IDT means interrupt descriptor table (aka interrupt vector).

ISR

ISR stands for interrupt service routine, i.e. interrupt handler.

LDT

An LDT is local descriptor table', a set of memory segments with different permissions.

PIT

PIT stands for programmable interrupt timer.

Chapter 3

System Features

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.

3.4 Compile-Time Configuration

The table below lists some features of the Zero kernel that you can configure at compile-time. The list is not complete; for more settings, consult <**kern/-conf.h**>.

Macro	Brief	Notes
SMP	symmetric multiprocessor support	Not functional yet
HZ	scheduler timer frequency	default value is 250
ZEROSCHED	default thread scheduler	do not change yet
NPROC	maximum number of simultaneous processes	default is 256
NTHR	maximum number of simultaneous threads	default is 4096
NCPU	number of CPU units supported	default is 8 if **SMP** is non-zero

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

5.2.5 Zone Allocator

5.2.5.1 Page Replacement Algorithm

Chapter 6

System Call Interface

TODO

Keep in mind, that the interface described here is currently **incomplete**; therefore, please consult the final interface later.

The most notable missing things at the moment are support for sockets as well as semaphores.

6.1 Conventions

6.1.1 IA-32

On IA-32 architectures, up to 3 system call parameters are passed in

6.2 Process Control

6.2.1 halt

void sys_halt(long flg);

The halt system call shuts the system down. If the **flg** argument has the **HALT_REBOOT** bit set, the system will be restarted after performing a shutdown.

6.2.2 sysctl

long sys_sysctl(long cmd, long parm, void *arg);

6.2.3 exit

long sys_exit(long val, long flg);

The exit system call terminates the calling process. The process returns **val** as its exit status. If **flg** has the **EXIT_DUMPACCT** bit set, process accounting information is dumped into the **/var/log/acct.log** system log file.

6.2.4 abort

void sys_abort(void);

The abort system call terminates the calling process in an abnormal way. If the limit for core dump size is set to be big enough, a memory image of the process is dumped into a **core** file. The location of this file may be either the local directory or one configured in **/etc/proc/core.cfg**.

6.2.5 fork

long sys_fork(long flg);

The fork system call creates a new child process. If **flg** has the **FORK_VFORK** bit set, the new process shall share the parent's address space; otherwise, the child's address space will be a clone of the parent's address space. If **flg** has the **FORK_COW** bit set, the new process will only clone pages as they are written on.

6.2.6 exec

long sys_exec(char *path, char *argv[], ...);

The exec system call replaces the calling process by an instance of the program **path**. The argument vector **argv** holds argument strings for the program to be executed; the table must be terminated by a final **NULL** pointer.

If a third argument is given, it shall be **char **** used as **environment** strings for the program; the table must be terminated by a final **NULL** pointer.

6.2.7 throp

long sys_throp(long cmd, long parm, void *arg);

The throp system call provides thread control. The **cmd** argument is one of the values in the following table.

cmd	parm	arg	Notes
THR_NEW	class	struct thrarg *	pthread_create()
THR_JOIN	thrid	struct thrjoin *	pthread_join()
THR_DETACH	N/A	N/A	pthread_detach()
THR_EXIT	N/A	N/A	pthread_exit()
THR_CLEANUP	N/A	N/A	cleanup; pop and execute handlers etc.
THR_KEYOP	cmd	struct thrkeyop *	create, delete
THR_SYSOP	cmd	struct thrsys *	atfork, sigmask, sched, scope
THR_STKOP	thrid	struct thrstk *	stack; addr, size, guardsize
THR_RTOP	thrid	struct thrrtop *	realtime thread settings
THR_SETATTR	thrid	struc thrattr *	set other attributes

6.2.8 pctl

long sys_pctl(long cmd, long parm, void *arg);

The pctl system call provides process operations. The following table lists possible values for the **cmd** argument.

cmd	parm	arg	Notes
PROC_GETPID	N/A	N/A	getpid()
PROC_GETPGRP	N/A	N/A	getpgrp()
PROC_WAIT	procid	N/A	wait()
	PROC_WAITPID	N/A	wait for pid
	PROC_WAITCLD	N/A	wait for children in the group pid
	PROC_WAITGRP	N/A	wait for children in the group of caller
	PROC_WAITANY	N/A	wait for any child process
PROC_USLEEP	milliseconds	N/A	usleep()
PROC_NANOSLEEP	nanoseconds	N/A	nanosleep()

6.2.9 sigop

long sys_sigop(long cmd, long parm, void *arg);

The sigop system call provides control over signals and related program behavior. The diffent values for **cmd** as well as related values for **parm** are shown in the following table.

cmd	parm	arg	Notes
SIG_WAIT	N/A	N/A	pause()
SIG_SETFUNC	sig	struct sigarg *	signal()/sigaction()
SIG_SETMASK	N/A	sigset_t *	sigsetmask()
SIG_SEND	N/A	sigset_t *	raise() etc.
SIG_SETSTK	N/A	struct sigstk *	sigaltstack()
SIG_SUSPEND	N/A	sigset_t *	sigsuspend(), sigpause()

Structure Declarations

```
/* flg bits */
```

```
#define SIG_NOCLDSTOP 0x01 // no SIGCHLD on stop or cont
#define SIG_ONSTACK   0x02 // use sigaltstk() stack
#define SIG_RESETHAND 0x04 // reset handler to SIG_DFL
#define SIG_RESTART   0x08 // no EINTR behavior
#define SIG_SIGINFO   0x10 // func(int, siginfo_t, void *)
struct sigarg {
    long  sig; // signal ID
    long  flg; // see SIG_-macros above
    void *func; // signal disposition
};
```

6.3 Memory Interface

6.3.1 brk

long sys_brk(void *adr);

The brk system call sets the current break, i.e. top of heap, of the calling process to **adr**. The return value is 0 on success, -1 on failure.

6.3.2 map

void *sys_map(long desc, long flg, struct sysmem *arg);

The map system call is used to map [zeroed] anonymous memory or files to the calling process's virtual address space. For compatibility with existing systems, mapping the device special file **/dev/zero** is similar to using the **flg** value of **MAP_ANON**.

flg	Notes
MAP_FILE	object is a file (may be /dev/zero)
MAP_ANON	map anonymous memory set to zero
MAP_SHARED	changes are shared
MAP_PRIVATE	changes are private
MAP_FIXED	map to provided address
MAP_SINGLE	map buffer mapped to single user process and kernel
MEM_NORMAL	normal behavior
MEM_SEQUENTIAL	sequential I/O buffer
MEM_RANDOM	random-access buffer
MEM_WILLNEED	don't unmap after use; keep in buffer cache
MEM_DONTNEED	unmap after use
MEM_DONTFORK	do not share with child processes

Structure Declarations

```
struct sysmem {
    void *base; // base address
    long  ofs; // offset in bytes
```

```
    long  len; // length in bytes
    long  perm; // permission bits
};
```

6.3.3 umap

long sys_umap(void *adr, size_t size);

The umap system call unmaps memory regions mapped with sys_map().

6.3.4 mhint

long sys_mhint(void *adr, long flg, struct sysmem *arg);

The mhint system call is used to hint the kernel of a memory region use patterns. The possible bits for **flg** are shown in the table below; for **struct sysmem** declaration, see **map**.

MEM_NORMAL	default behavior
MEM_SEQUENTIAL	sequential I/O buffer
MEM_RANDOM	random-access buffer
MEM_WILLNEED	don't unmap after use; keep in buffer cache
MEM_DONTNEED	unmap after use
MEM_DONTFORK	do not share with forked child processes

6.4 Shared Memory

The shared memory interface of Zero is modeled after the POSIX interface.

6.4.1 shmget

long sys_shmget(long key, size_t size, long flg);

The shmget system call maps a shared memory segment; it returns a shared memory identifier (usually a long-cast of a kernel virtual memory address). If **key** is **IPC_PRIVATE**, a new segment and its associated book-keeping data are created. If **flg** has the **IPC_CREAT** bit set and there's no segment associated with **key**, a new segment is created in concert with the relevant data.

6.4.2 shmat

void *sys_shmat(long id, void *adr, long flg);

The shmat system call atteaches the shared memory segment identified by **id** to the address space of the calling process. If **adr** is NULL, the segment is attached to the first address selected by the system. If **adr** is not NULL and **flg** has the **SHM_RND** bit set, the segment is mapped to **adr** rounded down

to the previous multiple of **SHMLBA**. If **adr** is not NULL and **flg** does **not** have the **SHM_RND** bit set, the segment is mapped to **adr**. If **flg** has the **SHM_RDONLY** bit set, the segment is attached **read-only**; otherwise, provided the user process has read and write permissions, the segment is attached **read-write**.

6.4.3 shmdt

long sys_shmdt(void *adr);

The shmdt system call detaches the shared memory segment at **adr** from the address space of the calling process.

6.4.4 shmctl

sys_shmctl(long id, long cmd, void *arg);

The shmctl system call provides control operations for shared memory segments. The possible values for **cmd** are listed in the following table.

cmd	arg	brief
IPC_STAT	struct shmid_ds *	read segment attributes
IPC_SET	struct shmid_ds *	set segment permissions (uid, gid, mode)
IPC_RMID	N/A	destroy shared memory segment

TODO: shared memory, message queues, semaphores, events

6.5 Semaphores

6.6 Message Queues

6.7 Events

6.8 I/O

Part III

User Environment

6.9 Process Environment

6.10 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, i.e. the addresses grow upwards
- the stack segment grows downwards in memory
- the BSS segment is allocated at run-time
- segments are shown in descending address order