Zero Volume 0 - The Kernel

Design and Implementation

DRAFT 1, Revision 13

Tuomo Petteri Venäläinen

January 3, 2014

Contents

Part I

Overview

Zero is an operating system project; at the time of this writing, the project has several pieces under construction, some of them rather good already.

Zero is open source. The choice of license is the so-called New BSD License; I chose it over the GPL in the hopes of possibly making it easy to adopt this software for business projects - after all, we all have to make a living. I do, however, plan to keep the base system (the kernel and support software) open source and free of charge; I have the feeling tailoring Zero for needs of clients might be fruitful for business.

At the moment, I have implemented the so-far-relevant parts of the C standard library implementation for 3 platforms: 32-bit X86 (IA-32), 64-bit X86 (AMD64), and some versions of the ARM microprocessor family. I would love to port the kernel to all of these as well as other platforms such as some 64-bit architectures common in UNIX workstations during the old times. Alpha, MIPS, Sparc, PowerPC etc. With such interests, donations of old server and workstation hardware would be very welcome, provided I have the room to set them up. :)

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 interaction as well as high multimedia performance.

1.1 Acknowledgements

1.1.1 Contributors

At the moment, I have kept Zero mostly 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. :)

At the time being, I want to thank **Craig Robbins** for some graphical effects, **Leonardo Taglialegne** for some audio hacks, and **Ivan Rubinson** for some graphics. :)

1.1.2 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 strong in spirit!

As their work has been used for the purpose of inspiration, thanks go to **Carsten 'raster' Haitzler** and his team for their work on **Enlightenment** and their mission to reinvent graphical user interfaces. Keep up the good work, guys! :)

1.2 Background

Zero has its roots in early, simple and elegant versions of the UNIX operating system. I still see many good, timeless things about UNIX design worth reusing in a new operating system. One thing of particular attraction is the "everything is a file" philosophy; 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. Buffer memory is allocated from a separate buffer cache, which may be either static or dynamic size. On Zero, 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.

Event Task

Certain events trigger threads or processes listening to them to run for a short period of time immediately. The timer used may be the same as for interval tasks (see **Fast Timer**). The idea is to let user interaction events 'steal' a bit of time from the rest of the system.

Fast Timer

A timer separate from the standard thread scheduling timer. In default configuration, the fast timer fires an interrupt 1,000 times a second, whereas normal threads are scheduled slices of 4 millisecond at the frequency of 250 Hz. On X86 platforms, HPET event timers are utilised.

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.

Hybrid Scheduler

See **Event Task**, **Fast Timer**, and **Interval Task** for more information.

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.

Signal

Signals are asynchronous events. Signals may be delivered by the kernel, by other processes, or by a process itself. Typical uses include notifying processes of things such as division by zero (SIGFPE), undefined memory access (SIGSEGV and SIGBUS) or illegal machine instructions (SIGILL).

Two predefined signals exists for application use; SIGUSR1 and SIGUSR2.

Task

In the context of Zero, the term task is usually used to refer to either **a process or a thread**.

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. This timer may be used to schedule threads on uniprocessor systems; for modern PCs, look into **APIC**.

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	multiprocessor support	Not functional yet
HZ	timer frequency	default value is 250
ZEROSCHED	default scheduler	do not change yet
NPROC	maximum simultaneous processes	default is 256
NTHR	maximum 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 code modules, some notable ones of which are listed below.

Module	Desription
tmr	hardware timer interface
proc	process 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

Processor Support

Initially, Zero shall support 32-bit and 64-bit X86 architectures. The plan is to port Zero at least to ARM in addition.

5.1 Thread Scheduler

5.1.1 Thread Classes

The following code snippet lists thread classes in decreasing order of priority.

TODO: add thread classes when final

5.1.2 Priorities

Priority Queue

Zero scheduler uses a queue with $THRNCLASS * THRNPRIO$ priorities.

Priority Adjustment

Every time a thread stops running, its priority is adjusted with the function below.

TODO: implement a simple hybrid priority adjustment algorithm

5.1.3 Data Structures

The structures **struct proc** and **struct thr** are declared in <**kern/obj.h**>. The example implementations below are for the **IA-32** architecture.

struct proc

The process data structure, of the type **struct proc** consists of per-process book-keeping data shared between all threads of the process.

IA-32 Implementation

TODO: add final proc structure

struct thr

The thread data structure, of the type **struct thr**, consists of machine-specific and portable members.

IA-32 Implementation

TODO: add final thr structure

5.2 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.

The symbolic [macro] names for interrupts and IRQs (interrupt requests) are declared in <**kern/unix/x86/trap.h**.

5.2.1 X86 Interrupts

The following table lists X86 interrupts along with their associated signals.

Macro	Number	Description	Signal
TRAPDE	0x00	divide error; fault	SIGFPE
TRAPDB	0x01	reserved; fault/trap	N/A
TRAPNMI	0x02	non-maskable interrupt; interrupt	N/A
TRAPBP	0x03	breakpoint; trap	SIGTRAP
TRAPOF	0x04	overflow; trap	N/A
TRAPBR	0x05	bound range exceeded; fault	SIGBUS
TRAPUD	0x06	invalid opcode; fault	SIGILL
TRAPNM	0x07	no FPU; fault	SIGILL
TRAPDF	0x08	double-fault; fault, error == 0	N/A
TRAPRES1	0x09	reserved	N/A
TRAPTS	0x0a	invalid tss; fault + error	N/A
TRAPNP	0x0b	segment not present; fault + error	SIGSEGV
TRAPSS	0x0c	stack-segment fault; fault + error	SIGSTKFLT
TRAPGP	0x0d	general protection; fault + error	SIGSEGV
TRAPPF	0x0e	page-fault; fault + error	N/A
TRAPRES2	0x0f	reserved	N/A
TRAPMF	0x10	FPU error; fault + number	SIGFPE
TRAPAC	0x11	alignment check; fault, error == 0	SIGBUS
TRAPMC	0x12	machine check; abort	SIGABRT
TRAPXF	0x13	SIMD exception; fault	SIGFPE

5.2.2 IRQs

The following table lists default use of IRQs 0 through 0x0f (15). Zero redirects these to interrupts 0x20 through 0x2f (32 to 47).

Macro	Number	Description
IRQTIMER	0	timer interrupt
IRQKBD	1	keyboard interrupt
IRQCOM2AND4	3	serial ports 2 and 4
IRQCOM1AND3	4	serial ports 1 and 3
IRQLPT	5	parallel port
IRQFD	6	floppy disk drive
IRQRTC	8	real-time clock
IRQMOUSE	12	mouse
IRQFPU	13	floating point unit
IRQIDE0	14	IDE controller 1
IRQIDE1	15	IDE controller 2

5.2.3 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 handler address, protection ring (system or user), and certain other attribute flags.

Chapter 6

Memory

6.1 Segmentation

Zero uses what is traditionally called a flat memory model; the use of segmentation is kept to the bare minimum required for successful code execution. Protection is done at page level.

A process may use either a global descriptor table with its physical address in the GDT-register or a local one with the address in the LDT-register. Processes are required to declare a handful of entries into their descriptor tables.

A segment descriptor is a CPU-specific data structure. On **IA-32** and **X86-64** architectures, the descriptors have base address and limit fields, permission bits, and other such values.

6.2 Paging Data Structures

Protection and other control of memory appears on per-page basis.

6.2.1 Page Directory

The page directory points to page tables; page tables, in turn, point to pages that may or may not be present in physical memory. This provides a hierarchical view of memory; by providing a virtual address, it is possible to obtain a physical one for memory access.

6.2.2 Page Tables

6.3 Page Daemon

6.3.1 Zone Allocator

The Zero memory manager was crafted to use aggressive buffering of allocations. The higher buffer level is a **Bonwick**-style **magazine** layer consisting of allocation stacks for sub-slab blocks. The lower level is a typical **slab allocator** dealing with slabs of power of two sizes to make book-keeping easier.

6.3.2 Page Replacement Algorithm

Chapter 7

I/O

7.1 Hardware I/O

7.1.1 Character I/O Transfers

Some devices operate in character based (raw) mode. This is typically feasible for so-called slow devices such as terminals. It is noteworthy that a system call needs to be triggered for every character.

7.1.2 Block I/O Transfers

Block I/O works using a buffer cache mechanism. Instead of doing single-character operations, I/O is buffered using bigger blocks (typically 4 to 64 kilobytes) to dramatically reduce the number of system calls involved in I/O operations.

Block based devices include hard drives, optical drives, and network interfaces. In case of network packets, the system caches checksums for static data to avoid redoing it on retransmittal.

Chapter 8

Inter-Process Communications (IPC)

Zero provides typical IPC primitives such as mutexes and generic semaphores, shared memory, and message queues.

Mutexes (binary semaphores) and other semaphores are used to restrict access to critical regions; e.g., linked lists need to be locked to avoid simultaneous access by multiple threads to ensure consistency of list data.

Shared memory is a fast form of IPC; shared memory is physical memory mapped to virtual address spaces of the kernel and user processes. It's recommended to use shared memory where possible to obtain zero-copy (or close) message passing.

Chapter 9

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.

9.1 Conventions

By convention, system calls return -1 on failure.

9.1.1 System Call Mechanism

9.1.1.1 Interrupt 0x80

On **IA-32** architectures, all of up to 3 system call parameters are passed in registers; the system call number is in **EAX**. On **X86-64** all system call arguments are passed in registers. On both architectures, system calls return a value for **errno**; the value is returned in **EAX** on IA-32, and in **RAX** on X86-64. **Failures** are indicated by setting the **CF-bit** (carry) in the **EFLAGS**-register. System calls are, in the first implementation, triggered by **interrupt 0x80**.

9.1.1.2 SYSENTER and SYSEXIT

TODO: sysenter + sysexit

9.2 Process Control

9.2.1 sysctl

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

The sysctl system call is used to trigger certain system control operations. The commands, their arguments, and the return values are listed in the next table.

cmd	**parm**	**arg**	Returns
SYS_INIT	runlevel	N/A	new runlevel

TODO: SYS_INIT (SYS_HALT, SYS_REBOOT), ...

9.2.2 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.

9.2.3 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**.

9.2.4 fork

long sys_fork(long flg);

The fork system call creates a new child process. The return value is the process ID of the new child process in the parent, and 0 (zero) in the 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** (copy on write) bit set, the new process will only clone pages as they are written on.

9.2.5 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.

9.2.6 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_SETATR	thrid	struc thratr *	set other attributes

9.2.7 pctl

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

TODO: pid as parm, special values for the choices

The pctl system call provides process operations. The following tables list provided functionality.

cmd	parm	Notes
PROC_GETPID	N/A	getpid()
PROC_GETPGRP	N/A	getpgrp()
PROC_WAIT	-1	wait for any child
	pid > 0	wait for pid
	pid == 0	wait for children in the group of caller
	pid < -1	wait for children in the group abs(pid)
PROC_USLEEP	milliseconds	usleep()
PROC_NANOSLEEP	nanoseconds	nanosleep()

Notes

For **PROC_WAIT**, if **arg** is NULL or **struct rusage *** for collecting resource usage statistics.

cmd	parm	arg	Notes
PROC_SETSCHED	attribute bits	struct syssched *	set scheduling parameters
PROC_STAT	pid or RUSAGE_SELF	struct rusage *	get resource usage statistic
PROC_GETLIM	limit bits	struct rlimit *	read process limits
PROC_SETLIM	limit bits	struct rlimit *	set process limits

9.2.8 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
};
```

9.3 Memory Interface

9.3.1 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.

9.3.2 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**.

The return value is user virtual memory address of the mapped region or ((void *)-1) on failure.

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 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
};
```

9.3.3 umap

long sys_umap(void *adr, size_t size);

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

9.3.4 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

9.4 Shared Memory

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

9.4.1 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.

9.4.2 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**.

9.4.3 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.

9.4.4 shmctl

void 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

9.5 Semaphores

long sys_seminit(void);

The seminit system call initialises a semaphore at the given virtual user address to unlocked state. The kernel maps the semaphore to its own virtual address space and returns the **ID of the new semaphore**.

The return value is the ID of the new semaphore on success, -1 on failure.

9.5.1 semup

long sys_semup(long id);

The semup system call atomically increases the value of a semaphore. The return value is the new semaphore value or -1 on failure:

9.5.2 semdown

long sys_semdown(long id);

The semup system call atomically decreases the value of a semaphore. The return value is the new semaphore value or -1 on failure.

9.5.3 semop

long sys_semop(long id, long cmd, long flg);

The semop system call performs a semaphore operation. See the table below for defined values.

cmd	**flg**	brief
IPC_STAT	N/A	obtain semaphore value
IPC_SET	permissions	set semaphore permissions
IPC_RMID	N/A	remove semaphore

9.6 Message Queues

TODO: MQSYNC-priority (0)

9.6.1 mqinit

long sys_mqinit(long perm, long prio, long nbyte);

The mqinit system call initialises a message queue with permissions and priority as supplied. The nbyte parameter determines the size of the region allocated for queue messages in octets.

mqinit returns the ID of the new message queue or -1 on failure

9.6.2 mqsend

long sys_mqsend(long qid, void *buf, long nbyte);

The mqsend system call sends nbyte bytes of message data from buffer to the specified queue. If the queue is too full for the message to fit, the system call will block until the transmittal can be processed; if, however, the queue was initialised with MQASYNC, the system call will return and the data shall be transmitted asynchronously.

The return value is the number of bytes sent or -1 on failure.

9.6.3 mqrecv

long sys_mqrecv(long qid, void *buf, long nbyte);

The return value is the number of bytes received, 0 if the message queue is empty, and -1 on failure.

9.6.4 mqop

long sys_mqop(long qid, long cmd, long parm);

9.7 Events

/* TODO: EVSHM, EVMQ */

9.7.1 evreg

long evreg(long ev, long flg, void (*func)(void *buf));

The evreg system call registers a handler function for the supplied event type.

9.7.2 evsend

long evsend(void *buf, long flg, long dest);

The evsend system call sends an event stored in a buffer to a receiver such as a desktop window (shared memory queue) or message queue (local or remote over a network)

9.7.3 evrecv

long evrecv(void *buf, long flg, long src);

The evrecv system call reads an event from a source such as shared memory (local) or a message queue (local or remote).

9.7.4 evop

long sys_evop(long cmd, long flg, void *atr);

9.8 I/O

Part III

User Environment

9.9 Process Environment

9.10 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