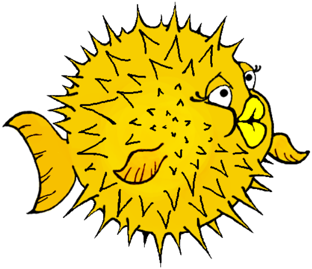
OpenBSD Kernel Internals

The Hitchhiker's Guide

Vladimir Kirillov
proger@uaoug.org.ua



$osys: sys.tex,v 1.43 2009/07/31 22:38:22 proger Exp $

Why?

- Security paranoia makes me want to know what's happening inside
- Want to learn system programming on a good free example
- Want to be able to help the project some day
  (so why not start studying and tell everybody to make it easier for such subsequest tries?)
- I'm keen on OpenBSD and system programming

- Chicks dig OpenBSD (-:

Introduction

Operating system kernel overview

- Foundation component of the OS

Multitasking kernel responsibilities

- Managing the system resources:
  - cpu time
  - memory
  - peripherals
- Access mediation between user-level software and hardware as an abstraction layer
- Communication facilities
- Providing basic security and protection

OpenBSD Kernel

OpenBSD kernel

- Inherits 4.4BSD (and NetBSD) Unix kernel architecture
- Monolithic (big, fast, easy to maintain, everything is in one address space) (with `LKM(4)` support)
- Provides the interface to software via system calls
- Supports plenties of HW architectures by separating the code to `MD` and `MI` parts
- Has integrated strong `crypto(9)` framework which is used (almost) everywhere

Source tree layout

/sys/ (MACHINE_ARCH=i386)

`kern`
: main kernel subroutines (`clock_`, `exec_`, `init_`, `kern_`, `sched_`, `subr_`, `sys_`, `syscalls.master`, `tty_`, `uipc_`, `vfs_`, `vnode_`)

`sys`
: kernel-wide include interfaces

`lib`
: kernel libraries (libc (libkern), libsa, libz)

`dev`
: device drivers

`dev/ic`
: bus-independent device drivers code

`{dev/$bus,arch/i386/$bus}`
: $bus driver code

`{net*,altq}`
: network stacks, pf code

`uvm`
: UVM virtual memory subsystem

Source tree layout (continued)

/sys/ (MACHINE_ARCH=i386)

`{isofs,miscfs/*,msdosfs,nfs,nnpfs,ntfs,ufs/{ext2fs,ffs,mfs,ufs}}`
: filesystems

`crypto`
: crypto framework implementation

`ddb`
: kernel debugger

`compat`
: other UNIXes compatibility interfaces

`arch/i386`
: i386 MD kernel code

`{arch/i386/stand,stand}`
: bootloaders: `mbr(8)`, `biosboot(8)`, `boot(8)`, depend on `libsa`

`arch/i386/include`
: MD include interfaces (referenced as `#include <machine/$file.h>`

`{arch/i386/conf,conf}`
: kernel configurations and params sources

Configuration

config(8)

- uses config(8)-syntax based kernel configuration
- uses files.conf(5)-based file lists for Makefile generation
- generates header files for kernel use
- generates device lists files (ioconf.c) for autoconf(9) framework

Files (MACHINE_ARCH=i386)

```
conf/files
arch/i386/conf/files.i386
dev/$bus/files.$bus

find /sys | egrep 'files.*'
```

more later

Kernel Organization

Kernel organization (system services)

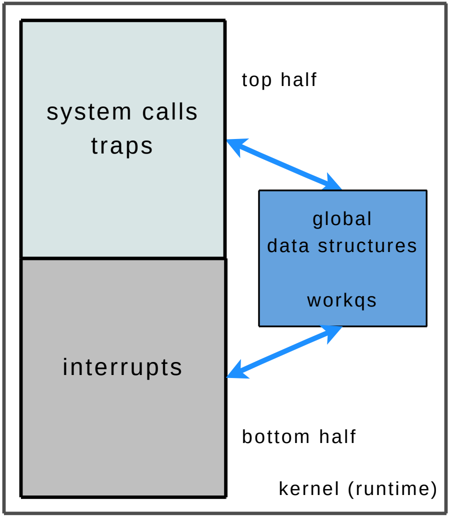
MD

- timer, system clock handling, process and descriptor management
- memory management
- descriptor operations
- filesystem
- terminal handling
- IPC (sockets)
- networking

MI

- low-level startup actions
- trap/fault handling
- low-level runtime context manipulation
- hardware configuration and initialization
- runtime support for I/O hardware

Kernel organization (runtime)



Top half

- works in a process context
- runs on a per-process kernel stack in process address space
- can block to wait for resources

Bottom half

- runs on kernel stack in kernel address space
- controls top half behaviour with clock interrupts

System entry points

Hardware interrupt

- arise from external events (devices)
- asynchonous, not related to process
- served by "bottom half" of the kernel
- i386: handlers are installed in and chosen by IDT, programmed into APIC or PIC

Hardware trap

- appear synchronous to process in the result of process actions
  (e.g.: division by zero, page fault)
- i386: trap handler is installed in IDT (first 20 entries)
  asm handlers are in `locore.s`, which perform initial handling and calling `trap(struct trapframe)`

System entry points (continued)

Software trap

- implemented either as hw-generated interrupt or flag (checked on each priviledge level drop)
- used by system to force the scheduling of events (deferred interrupts)
- special case of software trap - a system call

All kernel entries require machine state saving before processing

System calls

- a way for process to perform privileged system actions
- served in kernel top half (using the per-process kernel stack)
- appear synchronous to process

Implementation (`i386`)

- called by pushing arguments onto the stack, syscall number to `%eax` and triggering `int $0x80`

  (performed by libc routines, syscalls appear to user as libc functions)
- `IDTVEC(syscall)` routine (`locore.s`) saves machine state and passes to `syscall(struct trapframe)` function
- `syscall()` function performs checks, copies arguments to kernel space, picks a system call entry from `sysent[]` table and calls the handler

  if a `syscall()` gets interrupted by a signal/trap — it may result in syscall restart or exiting with `EINTR`
- on error: libc routine sets `errno` on error and returns -1
- on success: libc routine returns either 0 or return value, process continues execution

System calls implementation — kernel

sys/systm.h

```
/* system call handlers prototype */
typedef int       sy_call_t(struct proc *, void *, register_t *);

/* system call table */
extern struct sysent {
        short              sy_narg;      /* number of args */
        short              sy_argsize;   /* total size of arguments */
        int                sy_flags;
        sy_call_t         *sy_call;      /* implementing function */
} sysent[];
```

example: creating a new syscall "call"

kern/syscalls.master

```
310     STD              { int sys_call(int arg); }
```

- kern/makesyscalls.sh regenerates header files by creating arguments structure, prototypes for libc and rebuilds system call table

System calls implementation — kernel (continued)

kern/init_sysent.c: primary syscall table

```
struct sysent sysent[] = {
        /* ... */
        { 1, sizeof(struct sys_call_args), 0,
            sys_call },                         /* 310 = call */
};
```

sys/syscallargs.h

```
struct sys_call_args {
        syscallarg(int) arg;
};
```

syscall source code

```
int
sys_call(struct proc *p, void *v, register_t *retval)
{
        struct sys_call_args *uap = v;
        if (SCARG(uap, arg) == 0)
                return (EAGAIN);
        printf("sys_call(arg=%d): pid %u\n", SCARG(uap, arg), p->p_pid);
        return (0);
}
```

System calls implementation — libc

Hooking syscall into libc

1. install syscall headers regenerated by syscalls.master from `/sys/sys` to `/usr/include/sys`

   or `make includes`

2. add stub syscall object filename to `${ASM}` variable to `src/lib/libc/sys/Makefile.inc`

   e.g. ASM+= call.o

3. rebuild libc, the syscall symbols object file will be created and linked into libc:

```
${ASM}: ${LIBCSRCDIR}/arch/${MACHINE_ARCH}/SYS.h /usr/include/sys/syscall.h
    printf '#include "SYS.h"\nRSYSCALL(${.PREFIX})\n' | \
   ${CPP} ${CFLAGS:M-ID*} ${AINC} | ${AS} -o ${.TARGET}.o
    ${LD} -x -r ${.TARGET}.o -o ${.TARGET}
    rm -f ${.TARGET}.o
```

System calls implementation — libc (continued)

Syscall binding object

```
proger src/lib/libc 0 % nm obj/call.o
         U __cerror
00000006 T _thread_sys_call
00000006 W call

proger src/lib/libc 0 % objdump -S obj/call.o

obj/call.o:     file format elf32-i386

Disassembly of section .text:

00000000 <_thread_sys_call-0x6>:
   0:   e9 fc ff ff ff          jmp    1 <_thread_sys_call-0x5>
   5:   90                      nop

00000006 <_thread_sys_call>:
   6:   b8 36 01 00 00          mov    $0x136,%eax
   b:   cd 80                   int    $0x80
   d:   72 f1                   jb     0 <_thread_sys_call-0x6>
   f:   c3                      ret
```

debug: option `SYSCALL_DEBUG`

reference: `syscall(9)`

Clock handling

- clock is the most frequent hardware interrupt source
- `tc_init(9)` is the generic `MI` framework for `MD` timecounter handling
- the main system timecounter is handled via `hardclock(9)` which is interrupted `hz * ncpu` times per second
  `hardclock(9)` is used to update system time, control process timers and launch other timers if needed
- other clock handling functions are `statclock()`, `softclock()`, `schedclock()`
- `softclock()` is called as software interrupt, processes `timeout(9)` queue

Software interrupts

- used to handle deferred interrupt tasks in level queues
- three levels: `softclock, softnet, softtty`
- executed on each acceptable priviledge level drop

- application example: `timeout_set(9)`
- task deferring can be also achieved via workqs (`workq_create(9)`) which allows executing a task in process context (processed by a kernel thread)

Synchronisation

- Interrupt priority level: `spl(9)`
- Process-context read/write locks: `rwlock(9)`
- Inter-CPU mutexes: `mutex(9)`
- Legacy locking interfaces: `lockmgr(9)`

Process Management

Process management

Definitions

process
: a thread of control within it's own address space

thread
: a thread of control sharing the address space with another control thread

process context
: everything used by the kernel in providing services for the process (process data structues)

process control block
: current execution state of a process, defined by machine architecture

context switch
: switching among processes in an effort to share CPU(s) resources

task (`i386`)
: a unit of work that a processor can dispatch, execute, and suspend, can be used to execute a program, a task or process, an operating-system service utility, an interrupt or exception handler, or a kernel or executive utility

Process data structures

`/sys/sys/proc.h`

`/sys/arch/i386/include/proc.h`

`struct process`

- threads (`TAILQ of struct proc`)
- owner credentials
- limits

`struct proc`

- pid / pgid / sid
- VM space (`struct vmspace`)
- FD table
- scheduling information
- process state
- signal state / actions
- MD state information (`struct mdproc`)
- user structure (`struct user`)

Process data structures (continued)

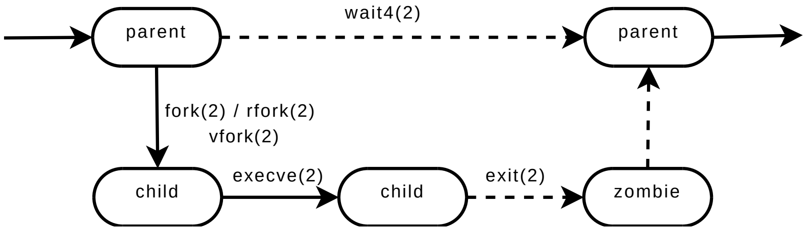
/sys/sys/user.h
/sys/arch/i386/include/pcb.h

struct user

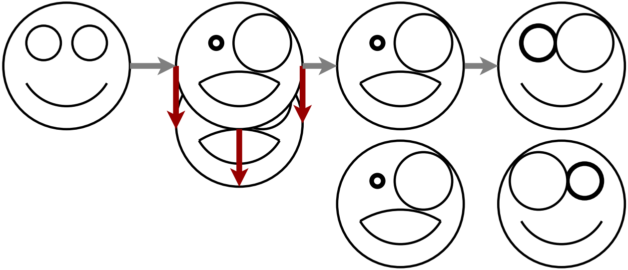
- process control block (`struct pcb`)
- resource accounting and statistics
- core dump information
- tracing information

- structure sharing is done with refrence counting

Essential process management-related syscalls



Process creation



- syscalls: `fork(2)` / `vfork(2)` / `rfork(2)` (implemented via `fork1(9)`)
- achieved via copying parent process data structures, including address space
- new execution program image is loaded via `execve(2)` / `exect(2)` syscalls

fork1(9)

- check process count limits
- allocate process data structures
- zero/copy process data structures fields
- init process timeouts
- call `uvm_fork(9)` function to share/copy virtual address space
  - call `cpu_fork()` to create PCB and make child ready to run
- init process timers (`virttimer/proftimer`)
- update stats (`uvmexp`)
- pick `PID` for process
- hook new process into scheduler, pick a run queue and cpu
- initialize tracing if needed

Kernel threads

- special case of a process: runned in system with kernel privileges, linked into kernel executable
- always cloned from process 0 (`swapper`)
- share memory map, limits
- have a copy of FD table
- can not be swapped (kernel memory is wired)
- do not receive broadcast/group signals

- created using `kthread_create(9)`
  - implemented via `fork1(9)`

Process state

Process control block (PCB)

- represented by `struct pcb` in `struct user`
- contains MD process state data (i386):
  - TSS - task state segment data, keeps track of segments of 3 PLs, GPRs, LDT, CRs
  - FPU status
  - I/O bitmap
  - VM86 mode flags, etc
- i.e. defines process context for switching

`struct proc p_stat` values

```
#define SIDL        1       /* Process being created by fork. */
#define SRUN        2       /* Currently runnable. */
#define SSLEEP      3       /* Sleeping on an address. */
#define SSTOP       4       /* Process debugging or suspension. */
#define SZOMB       5       /* Awaiting collection by parent. */
#define SDEAD       6       /* Process is almost a zombie. */
#define SONPROC     7       /* Process is currently on a CPU. */
```

Scheduling

- relies on clock:
  `hardclock()`:
  - `statclock()` gets called if no other timer for it
  - `roundrobin()` gets called every `hz / 10` ticks (100 ms for now (`hz = 1000`)) to call `need_resched()` (which toggles AST for `preempt()`)

  `statclock()`:
  - `p->p_cpticks` get increased
    so are other process tick stats
  - `schedclock()` gets called to adjust process priority
    active processes get higher priority
- process priority is calculated in `resetpriority()`:

  ```
  newpriority = PUSER + p->p_estcpu + NICE_WEIGHT * (p->p_nice - NZERO);
  p->p_usrpri = min(newpriority, MAXPRI);
  ```

  priority affects the run queue the process is put into
- process may be either in sleep queue or in run queue according to its status (waiting for resources or ready to run respectively)

Scheduler queues

Run queues

- each CPU has `SCHED_NQS` (32) run queues
- the queue for the process is picked according to priority
  `int queue = p->p_priority >> 2;`
- defined as a `TAILQ` array for each `struct schedstate_percpu`

Sleep queue

- defined as a global `TAILQ` array (`TABLESIZE = 128`, used for hashing wait channel pointers to speed up the lookup)
- handled by `tsleep(9)/wakeup(9)`
- the process is put back on run queue on wakeup

Context switching

Context switch cases

- forced - AST in user mode in result of spending the process time slice
- voluntary - calling `yield()` (`sched_yield()` syscall)
- involuntary - in result of getting into sleep queue after calling `tsleep(9)`

`mi_switch()`

- implements the machine-independent prelude to context switch
- counts resource usage stats
- chooses next process (`sched_chooseproc()`)
- performs `cpu_switchto()`

`cpu_switchto()`

- MD function of context switch (implemented in `locore.s` on `i386`)
- saves old process PCB, loads new and starts rolling

Threading

pthreads

- user-level N:1 threding implementation
- POSIX standard
- uses user-level scheduler implemented on top of per-process timers (`ITIMER_VIRTUAL/SIGVTALRM`, `ITIMER_PROF/SIGPROF`, `setitimer(2)`)
- makes no use of SMP for threads
- when one thread waits for resources — others block

rthreads

- kernel-level 1:1 threading implementation
- based on `rfork(RFTHREAD)` system call
- system scheduler handles each thread
- removes all pthreads limitations
- librthread is binary compatible to libpthreads
- currently in development

Signals

- designed as hardware interrupts for software
- allow process to respond to asynchronous external events
- higher-level `psignal(9)` is used to post signals
- sent to process via MD routine `sendsig()`
  - immediately save process exec frame
  - run handler from signal table
  - process handles signal if possible and calls `sigreturn()`, which restores normal execution

Tracing and debugging

ktrace(2)

option KTRACE

- enables kernel trace logging for processes
- trace is written to a file, may be controlled via ktrace(1), kdump(1) tools

ptrace(2)

option PTRACE

- allows one process (the **tracing** process) to control another (the **traced** process)
- most of the time, the **traced** process runs normally, but when it receives a signal (sigaction(2)), it stops
- the **tracing** process is expected to notice this via wait(2) or the delivery of a SIGCHLD signal
- e.g. gdb(1) is implemented via ptrace(2)

Memory Management

Physical memory management (`i386`)

- the memory that the processor addresses on its bus is called **physical memory**
- physical memory is organized as a sequence of 8-bit bytes
- each byte is assigned a unique address, called a **physical address**

- OpenBSD kernel does not address physical memory directly, it uses **segmented** memory model with **paging**

Physical memory management (i386) — definitions

linear address space
: processor's addressable memory space

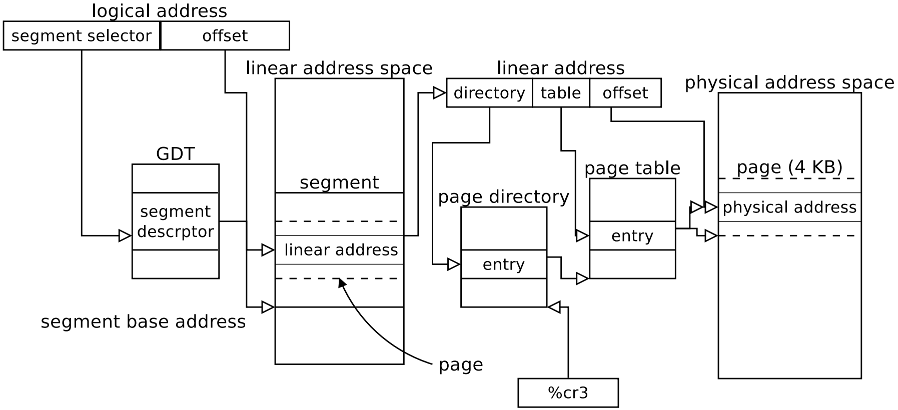
segment
: smaller protected address space; each program can be assigned its own set of segments

logical address (far pointer)
: consists of a segment selector and an offset; used to locate a byte in a particular segment

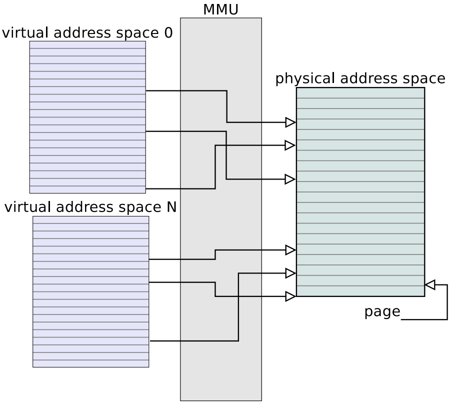
linear address space
: segment base address plus the logical address offset

paging
: technique used to store and retrieve data from secondary storage for use in main memory; used in implementing virtual memory; transparent to program execution

Physical memory management (i386) — scheme

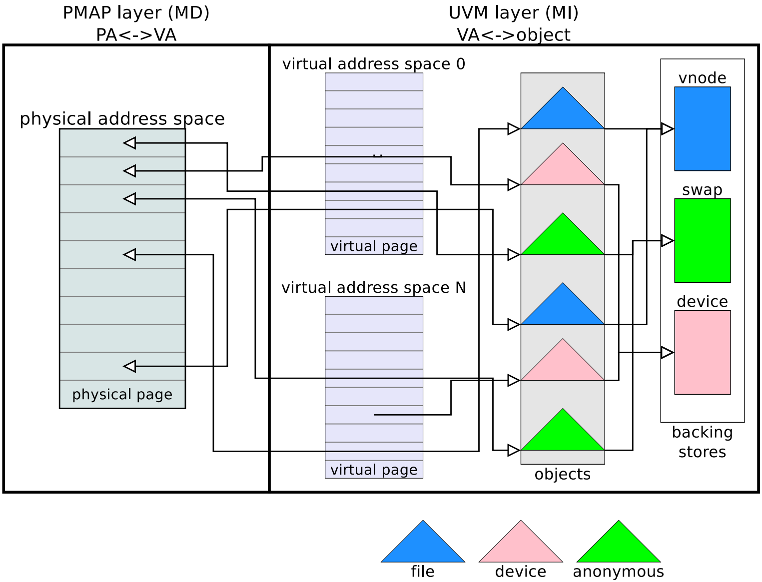


Virtual memory

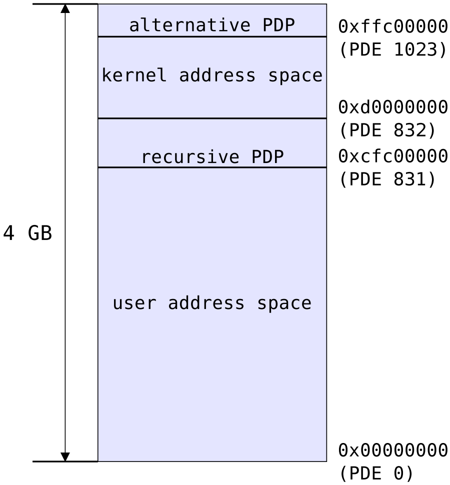


- each process gets its own address space
  (which in fact may be physically fragmented or even overflow on to disk storage)
- used to organize memory protection between processes
- allows mapping of either files or devices into virtual
- each virtual address is converted to physical in hardware
  (using **Memory Management Unit**)

Memory management in OpenBSD



PMAP (i386)



- the lower layer of VM system
- describes a process' 4GB virtual address space
- maintains VA<->PA mappings
- can be viewed as a big array of mapping entries that are indexed by virtual address to produce a physical address and flags (flags describe the page's protection, whether the page has been referenced or modified, etc)

`pmap(9)`

PMAP data structures (i386)

`struct pmap`
: describes an address space of a thread

`struct pv_entry`
: describes `<PMAP, VA>` mapping for `PA`

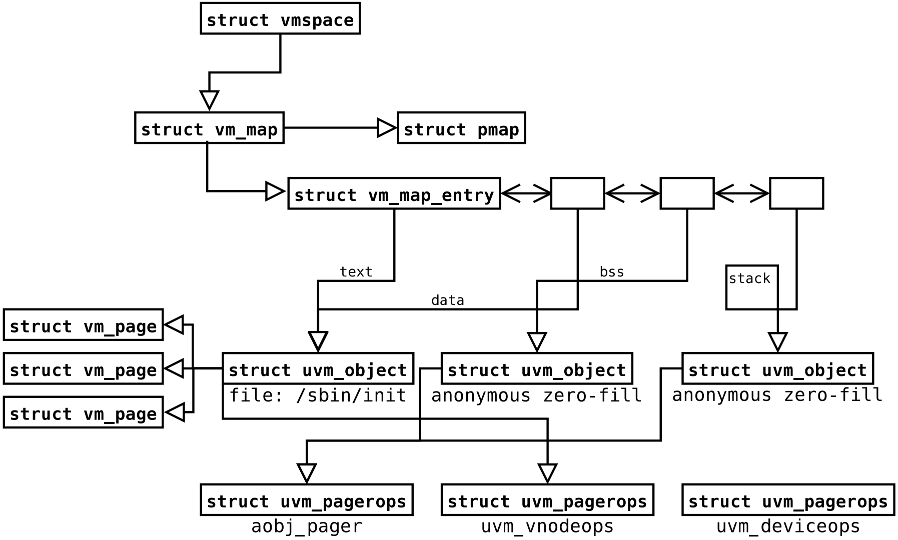
`struct pv_head`
: points to a list of `struct pv_entry`, which describes all `<PMAP, VA>` pairs for one page

`struct pv_page / struct pv_page_info`
: `struct pv_entry`'s are allocated out of `struct pv_page`

UVM

- the upper level of VM system
- manages `VA<->object` mappings
  (object can be either anonymous mapping, file or device)
- handles **page faults**
- designed as an evolution of old BSD VM system to eliminate its limitations while retaining its positive design aspects:
  - `MD-MI` separation
  - copy-on-write technique
  - several data structures
- new data movement techniques:
  - page loanout (process<->process)
  - page transfer (kernel<->process)
  - map entry passing (process<->process)
- implemented with fine-grained locking which is good for SMP systems

UVM data structures



Page faults

- hardware trap caused by MMU when no physical memory page is mapped at starting memory address
  - process accesses an unmapped/improperly mapped memory in its VA space
  - processor MMU generates `#PF` trap
  - MD routine asks MMU to provide VA, which triggered PF and access type
  - MI `uvm_fault()` routine gets called
    - VM system lookups mappings at that address
    - if the mapping is invalid/access control error -> send `SIGSEGV`
    - otherwise **fault in** the page into physical memory and continue process execution

uvm_fault()

```
look up faulting address' map entry;
if (data is in amap layer) { /* case 1 */
        handle any copy-on-write processing;
        map page and return success;
} else if (data is in uvm_object layer) { /* case 2 */
        handle any copy-on-write processing;
        map page and return success;
} else { /* missing data */
        return error;
}
```

Kernel memory management

- UVM routines: `uvm_km_*` (see `uvm(9)`)
- `malloc(9)` - generic memory allocator (like `malloc(3)`), implemented on top `uvm_km_*` functions, with statistics support

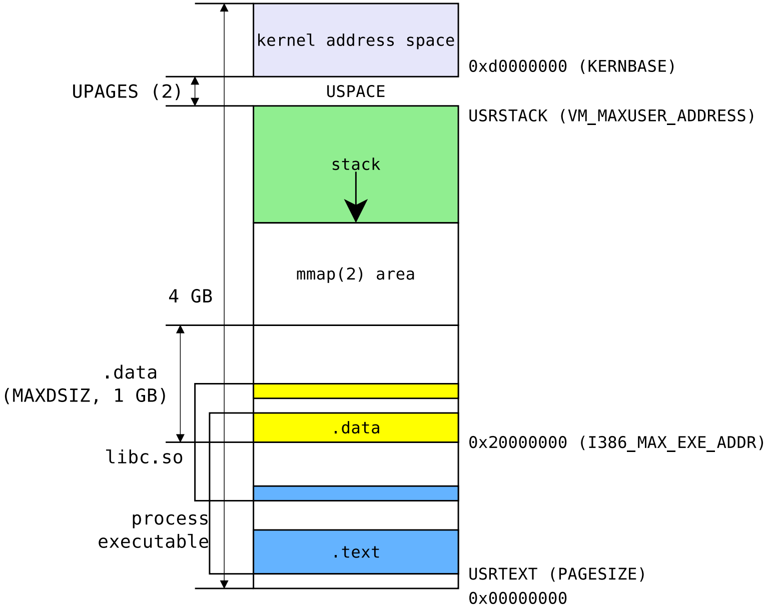
  ```
  void *malloc(unsigned long size, int type, int flags);
  ```

- `pool(9)` - resource pool manager; provides management of pools of fixed-sized areas of memory. Resource pools set aside an amount of memory for exclusive use by the resource pool owner.
- `extent(9)` - provides management of areas of memory or other enumerable spaces (such as I/O ports) (implemented on top of `pool(9)`)
- `mbuf(9)` - buffer management for networking

Debugging tricks

```
option UVMHIST
option UVMHIST_PRINT
```

Process VM space layout (i386)



Memory management interfaces in userspace

System calls

```
void *mmap(void *addr, size_t len, int prot, int flags, int fd, off_t offset);
int   msync(void *addr, size_t len, int flags);
int   munmap(void *addr, size_t len);
int   mprotect(void *addr, size_t len, int prot);
int   madvise(void *addr, size_t len, int behav);
int   mlock(void *addr, size_t len);  /* mlockall(int flags) */
int   munlock(void *addr, size_t len);/* munlockall(void) */
int   minherit(void *addr, size_t len, int inherit);
int   mincore(void *addr, size_t len, char *vec);
void *mquery(void *addr, size_t len, int prot, int flags, int fd, off_t offset);
```

- libc `malloc(3)` is implemented via `mmap(2)`

Kernel Bootstrap (i386)

`mbr(8)`

- locates in first 512 bytes of hard disk
- gets loaded by `BIOS` at `0000:7C00`
- relocates to `07A0:0000` (**chainloading**)
- scans partition table for the first active one
- reads `PBR` (OpenBSD: `biosboot(8)`) to memory
- `ljmp`

- written in assembler
- relies on `BIOS` routines
- works in real addressing mode

`/sys/arch/i386/stand/mbr/mbr.S`

biosboot(8)

- written in assembler
- relies on BIOS routines
- works in real addressing mode
- capable of reading ELF `boot(8)` binary from FFS partition while aware of its position (patched by `installboot(8)`)

```
/sys/arch/i386/stand/biosboot/biosboot.S
/sys/arch/i386/stand/installboot/installboot.c
```

boot(8)

- responsible of setuping protected mode environment
- does basic devices probing and memory detection, a20 gate activation
- supports interactive configuration + `boot.conf(8)`
- uncompresses kernel image and copies it into memory
- passes device probing information and arguments to the kernel
- `ljmp`!

```
/sys/lib/libsa/
/sys/stand/boot/
/sys/arch/i386/stand/libsa/
/sys/arch/i386/stand/boot/
```

In kernel: `locore.s`

- processor detection
- creating bootstrap kernel virtual address space, initializing initial paging support
  (kernel should be relocated to `KERNBASE (0xd00000000)`, so it is mapped twice first)
- setup new stack for process 0 and future kernel startup
- wire 386 chip for unix operation: `init386()`
- call `main()`

`/sys/arch/i386/i386/locore.s`

machdep.c: init386()

- enumerate processor address spaces with `extent(9)`: `ioport_ex`, `iomem_ex`
- create new bootstrap `GDT`
- create `IDT` and hook trap handlers
- if system has `isa(4)` call `isa_defaultirq()` to program PIC (`i8259`)
- initialize console
- bootstrap pmap / count physical memory
- init `ddb` / `kgdb`; throw into ddb if asked
- init soft interrupts

`/sys/arch/i386/i386/machdep.c`

init_main.c: main()

- initialize timeouts
- init autoconf structures
- init UVM
- init disk
- init tty
- `cpu_startup()`: init dmesg buffer, fill cpu0 data structures, start first rt clock
  - drop into UKC if requested in `boot(8)`
- init IPC goo: sockets, pipes, mbufs
- init filedescriptors
- fill in process 0 (“`swapper`”) context and data structures
- init scheduler
- init workqs
- `cpu_configure()`: run devices autoconfiguration
- init nfs/vfs
- init clocks
- init SysV features (shm, msg queues, semaphores)
- configure/attach pseudo devices (like `pf` or `crypto`)

`init_main.c: main()` (continued)

- init networking
- init exec feature
- start scheduler
- fork `init(8)` process (call `start_init()` which will `tsleep(9)` until everything else is configured
- create deferred kthreads
- wait until autoconfiguration finished
- mount root
- start other generic kernel threads (`pagedaemon, reaper, etc`)
- boot application processors
- wakeup init thread
- enter `uvm_scheduler` as main `swapper` (proc0) job

`/sys/kern/init_main.c`

Driver architecture

`autoconf(9)`: driver framework

autoconf(9) by example

mainbus(4)

- definition: /sys/arch/i386/conf/files.i386

```
define mainbus {apid = -1}
device mainbus: isabus, eisabus, pcibus, mainbus
attach mainbus at root
file arch/i386/i386/mainbus.c mainbus
```

- configuration: /sys/arch/i386/conf/GENERIC

```
mainbus0 at root
```

Autoconfiguration data structures: /sys/sys/device.h
Autoconfiguration subroutines: /sys/kern/subr_autoconf.c

autoconf(9) by example (continued)

mainbus(4)

- driver structures: /sys/arch/i386/i386/mainbus.c

```
struct cfattach mainbus_ca = {
    sizeof(struct device), mainbus_match, mainbus_attach
};

struct cfdriver mainbus_cd = {
    NULL, "mainbus", DV_DULL
};
```

- attach rountine should scan for children devices and install interrupt handlers/etc jobs

Kernel frameworks (subsystems)

- buses: `PCI/USB/SBUS/ISA/SDMMC/I2C/GPIO/...`
- device classes: `SCSI/ATA`
- networking layers: `net/netinet/netinet6/net80211/netbt/netatalk/netnatm/netmpls`
- highlevel driver-hooking frameworks:
  - `wscons(4)`: input and display abstracting/multiplexing
  - `drm(4)`: direct rendering management
  - `bio(4)`: kernel block I/O storage abstraction
  - `tty(4)`: terminal handling
  - sensors (`sensor_attach(9)`: device status sensors
  - `audio(9)`: framework for audio drivers
  - `radio(9)`: sound/video tuners
  - `ifmedia(4)`: handling network interface options
- interfacing to a kernel subsystem normally takes the form of filling out a few structures and perhaps callbacks in attach code then using provided functions.

Exploring and debugging

- `man -k`
- `*DEBUG` macros / hidden compile options (like `UVMHIST`, `BIOS_DEBUG` etc)
- `ddb / kgdb`
- serial console
- `*stat` userland tools
- cool text editor (`vim`) + `ctags` (`cd /sys/arch/i386; make tags`)
- `grep / ack`
- Google
- OpenBSD mailing lists
- `while true; do cd /usr/src; cvs up; done`

Thanks to:

- The OpenBSD Project
- ATMNIS for OpenKyiv organization
- Sergey Prysiazhnyi for helping and guiding
- My friends for support
- My girlfriend for drawing a Puffy on my 1st slide

Future work: practice

Thank You