eXecute In Place support in NetBSD

Masao “uebs” Uebayashi
<uebayasi@tombi.co.jp>

BSDCan 2010

2010.5.13
Who am I

NetBSD developer
Japanese
Living in Yokohama
Self-employed
  Since Dec. 15 2008
Tombi Inc.
Agenda

Demonstration
Introduction
Program execution
VM
  Virtual memory management
  Physical memory management
  Fault handler, pager
  :
Design of XIP
Demonstration

XIP on NetBSD/arm (i.MX35)
Introduction

What is XIP?

Execute programs directly from devices
No memory copy
Only about userland programs
( Kernel XIP is another story)
Introduction

Who needs XIP?

Embedded devices
  Memory saving for less power consumption
  Boot time
Mainframes (Linux)
  Memory saving for virtualized instances

"Nothing in between"
Introduction

How to achieve XIP?

Don't copy programs to memory when executing it

“Execute” == \texttt{mmap()}
Goals

No hacks
  Keep code cleanliness
  Keep abstraction
    Including device handling

Performance
  Latency
  Memory efficiency
Program execution

execve(2) → sys_execve()

Prepare
- Read program header using I/O
- Map sections
- Set program entry point

Execute
- Page fault is triggered
  - Load pages using VM
- Execution is resumed
Program execution

I/O part needs no changes
   If block device interface (d_strategy()) is provided
VM part needs changes!!!
Virtual memory management

http://en.wikipedia.org/wiki/Virtual_memory

Virtual memory is a computer system technique which gives an application program the impression that it has contiguous working memory (an address space), while in fact it may be physically fragmented and may even overflow onto disk storage.

Developed for multitasking kernels, virtual memory provides two primary functions:

- Each process has its own address space, thereby not required to be relocated nor required to use relative addressing mode.

- Each process sees one contiguous block of free memory upon launch. Fragmentation is hidden.
Virtual memory management

http://en.wikipedia.org/wiki/Virtual_memory

All implementations (excluding emulators) require hardware support. This is typically in the form of a memory management unit built into the CPU.
Virtual memory management

Resource management
  Virtual address space
  Physical memory
On-demand paging
  Limited resource
Slow operation (I/O)
Object abstraction
  Linear mapping
Page cache
Virtual memory management

Behavior == object oriented + event driven

Preparation
  API
    Kernel API
    User API (== syscall)

Resolution
  Fault handler
  VM objects -> pager
Physical memory management

Structure

`vm_physseg`
- Continuous physical memory segment
- Registered at bootstrap
  - (hotplug is not yet)

`vm_page`
- Per-page metadata
- Page's `state`
- MI vs. MD (`vm_page_md`)
- CPU cache vs. page cache
Fault handler

On-demand operations

Paging (page cache <-> backing store)

getpages() returning *vm_page[]

H/W mapping (TLB <-> page table)

pmap_enter() with vm_page->phys_addr

Optimizations

Pre-fault

Copy-on-write

Relying on H/W assistance (MMU)
Fault handler

Behavior

Suspend the faulting context

Resolve things

Paging (== I/O == slow)

*** do dirty things here ***

H/W mapping

Resume the faulted context
Pager

Object-oriented abstraction
Linear space
PAGE_SIZE wise
Any backing store
  Vnode
  Swap (aobj)
VM object's operation
  "get pages"
  "put pages"
Pager

Device pager
Character device
mmap(2)'ing /dev/XXX
“Unmanaged”
Its own “special” handler
“Steal” the handling onto its region
No generic support
No copy-on-write, etc.
Its own `pmap_enter(9)` callsite (ugly)
Pager

Vnode pager

Glue file/filesystem into VM

Represent file as VM object
Address space vs. filesystem / block address
Paging vs. filesystem state
"genfs" functions

Filesistem is complex but useful component
Pager

Vnode pager for XIP

Return physical address back to the fault handler

But can't use `vm_page` because the physical address is specific to the "activation"

-> Return the physical address by encoding it in "`struct vm_page *`"
   
   `(struct vm_page *)((intptr_t)phys_addr | MAGIC)`

-> "This is part of a device's segment, the offset to it is XXX"
“Device segment” and “device page”

Exist on devices

- Memory-addressable device region
- Attached during device configuration

“Managed”

- Which virtual address is mapped to which physical address?
- Track P->V mapping

Paging?

- Depend on usage? XXX to be considered
Design

Introduce “device segment” and “device page”
Switch XIP by mount point
Mark mount point as XIP
vnode pager checks if vnode is on XIP mount
If yes, return “device pages” back to generic fault handler
Teach (generic) fault handler and vnode pager about “device page” handling
Design

Device driver interface for “device page”

bus_space_physload(9)
business_space_physload_device(9)

Device drivers *must* register their “possibly managed” bus address space

VM allocates “context” internally to keep track of the “managed” device pages
Design

Mount and device driver
-> Interfaced using “device segment”

Device driver registers its segment and gets “device segment” as a cookie (to VM)

When mounting the device, the “device segment” is associated with the mount point

Vnode pager refers to the “device segment” cookie
Implementation

uebayasi-xip branch on anoncvs.NetBSD.org
TODO

More tests, measurements, and tuning
Write more FlashROM device drivers (glues)
Explicit mount option or not
Optimized filesystem (Linux's AXFS)
`xmd(4)` - XIP memory disk
Memory hotplug, NUMA, ...
Convert framebuffers to use `bus_space_physload(9)`
Summary

Basic XIP support for NetBSD is implemented
Available in netbsd-6 (hopefully)
A new concept “device page” is introduced and driver API is provided
  bus_space_physload_device(9)
Clean design
No special MD code
No special device drivers
No hacks