Overview

Part I - Getting Started
- Sources of Information
- Source Code Layout
- Build Tools
- Miscellaneous routines
- Debugging

Part II - Virtual Memory
- Overview of Mach Virtual Memory
- The pmap module

Part III - Saving and Restoring State
- Kernel entry and Exit
- Traps, interrupts and system calls
- Continuations

Part IV - User Code
- Libmach
- Cthreads
- Emulator library
- BSD single server
Sources of Information

**Technical questions**

mach3@cs.cmu.edu
Read by CMU, OSF, and 300 other people
To be added contact mach3-request@cs.cmu.edu

**Administrative questions**

mach@cs.cmu.edu
Read by Mach distribution people

**Documentation**

All documentation mentioned is available via anonymous ftp from mach.cs.cmu.edu in the doc directory

Source Code Layout

**MK Source Code Layout**

- **obj**: mirror of src area where object files are placed
- **release**: final release area
- **src**: source area
- **build_tools**: code make, gcc, gmake
- **mk**: micro kernel
- **user**: mach user programs
- **ux**: single server
- **parisc_mach**: machine dependent area for Hewlett-Packard pa-risc machine

**Single Server Code Layout**

- **emulator**: sources for emulation library
- **include**: Makefiles for releasing include files
- **server**: single server sources
- **bsd**: bsd sources
- **conf**: configuration files
- **sys**: UNIX include files
- **ufs**: UNIX file system
- **uxkern**: Mach glue code
- **vm**: virtual memory
- **parisc**: machine dependent code for Hewlett-Packard pa-risc machine
C Shell Tricks

setenv M3BASE /usr0/bobw/M3
setenv M3SRC $M3BASE/src
setenv M3OBJ $M3BASE/obj/parisc_mach

alias ksrc cd $M3SRC/mk/kernel
alias ssrc cd $M3SRC/ux/server
alias esrc cd $M3SRC/ux/emulator
alias kobj cd $M3OBJ/mk/kernel
alias sobj cd $M3OBJ/ux/server
alias eobj cd $M3OBJ/ux/emulator

set cdpath=/($M3SRC/mk
   $M3SRC/mk/kernel
   $M3SRC/ux
   $M3SRC/ux/server
   $M3SRC/ux/emulator)

setenv FAKE “-DKERNEL -I. -I. \
-I$M3OBJ/mk/kernel/STD+ANY-debug”

Strange and Mysterious Build Magic

setvar shell script
Sets environment variables for a specific machine

mk/Makeconf

Tells make where the object area is
MAKEOBJDIR
Tells make where the source area is
MAKESRCDIRPATH

See
Building Mach 3.0
Mary R. Thompson and Richard P. Draves
Available via anonymous ftp

Kernel Build Tools

doconf
Reads MASTER configuration files and produces input for config

config
Generates include files and Makefile for building kernel

MiG
Mach interface generator, the IPC stub generator

makeboot
Binds a kernel and the default pager into a single bootable image

Porting Config

Add configuration type to config.h
#define CONFTYPE_PARISC 18

Add test for configuration type in config.y
else if (!strcmp($2, “parisc”)) {
   conftype = CONFTYPE_PARISC;
   conftypename = “parisc”;
}

Add case in main.c
   case CONFTYPE_PARISC:
      parisc_ioconf();

Add routine for ioconf.c in mkioconf.c
   ifdef CONFTYPE_PARISC
      parisc_ioconf() {} 
   #endif /* CONFTYPE_PARISC */

Add users entry in mkmakemakefile.c
   { 32, 8, 1024 } /* CONFTYPE_PARISC */
The MASTER files

- A simple way to specify configurations
- Read by doconf to create input to config

```
# STD= [ hp700 scsi lan]
# ANY= [ ]

conftype "parisc" # <hp700>
platform HP700 # <hp700>
config mach kernel

options TRAP_COUNTERS # <test>

device sd0 # <scsi>
device sd1 # <scsi>
device sd2 # <scsi>
device sd3 # <scsi>
device lan # <lan>
pseudo-device bpf 16 # <lan>
```

The files File

- Specifies options and files to config
- Paths are relative to mk/kernel

**Syntax:**

```
<OPTIONS | directory>filename \ 
<optional opt | standard> \ 
[device-driver] [ordered] [compiler-options]
```

**Example:**

- OPTIONS/trap counters optional trap counters
- parisc/locore.s standard ordered
- parisc/context.s standard
- parisc/trap.c standard
- parisc/pmap.c standard
- parisc/sd.c optional sd device-driver
- parisc/lan.c optional lan device-driver
- parisc/bpf.c optional bpf device-driver

Config Output

All output from config is in the object area

Makefile internal

```
ioconf.c
platforms.h
  #define HP 700 1
trap_counters.h
  #define TRAP_COUNTERS 0
sd.h
  #define NSD 4
lan.h
  #define NLAN 1
bpf.h
  #define NBPF 16
```

Specifying Options

**Two ways to specify options**

In MASTER file put in an options line

```
options TRAP_COUNTERS #<hp700>
```

If there is an OPTIONS line in the files file then config will produce an include file

```
trap_counters.h
  #define TRAP_COUNTERS 1
```

Otherwise config will add a -D to the compile line

```
-DTRAP_COUNTERS
```

Use include files if the option will change
MiG - The Mach Interface Generator

Stub Generator for Mach IPC

Mig sources are in .defs files

Uses a PASCAL like syntax for historical reasons

Machine specifics gathered by include files
(Do n’t have to “port” Mig)

Most kernel Mig output is put in subdirectories in
the object area. (include in etags)

Don’t try and debug Mig, that’s not the problem

Porting Makeboot and Bootstrap

off_t exec_header_size()
Routine to tell the size of the object file header

int exe_get_header(in_file, is_kernel, lp,
sym_header, sym_header_size)
Routine to read the object file header

void write_exec_header(out_file, kp, file_size)
Routine to write the object file header

Bootstrap only needs to read the object file header
code is very similar to exe_get_header()

Makeboot

Combines the kernel with the default pager into a
single bootable image

At runtime move bootstrap() moves pager out of
kernel data area before the BSS section is cleared

Getting Started

Get build tools working
Fake include files
Fake configuration files
Fake genassym.c

See what comes up undefined
Use grep and etags

Write small utilities like findsym

#! /bin/sh
for file in *.o;
do
echo $file
nm $file | grep $1
done
Kernel Bootstrap

Bootstrapping is the process of establishing a stack, initializing hardware, and calling init().

Init() moves the bootstrap image, zeroes BSS, configures the bus, sizes memory, and calls pmap_bootstrap().

Return from init() enables virtual memory (first fault).

Setup_main() initializes the rest of the machine-independent system, calls machine_init() for machine-dependent initialization (autoconf) after VM is enabled.

Starts additional bootstrap threads and creates a bootstrap task (first syscall).

Bootstrap Routines

**locore**

Establish a stack, initialize hardware, and call init().

**init()**

Move bootstrap image, zero BSS, configure bus, size memory, call pmap_bootstrap().

**return from init()**

Enable virtual memory (first fault).

**setup_main()**

Initializes the rest of the machine-independent system.

Calls machine_init() for machine-dependent initialization (autoconf) after VM is enabled.

Starts additional bootstrap threads.

Creates a bootstrap task (first syscall).

Miscellaneous Routines

**void startrtclock()**

Set the current time of day and start periodic clock interrupts.

**void resettodr()**

Set the time of day clock.

**void halt_cpu()**

Halt this cpu.

**void halt_all_cpus(reboot)**

Halt all processors and optionally reboot.

Device Drivers

Devices are very similar to BSD devices.

One table instead of a cdevsw and bdevsw table.

```c
struct dev_ops {
    char *d_name;
    int (*d_open)();
    int (*d_close)();
    int (*d_read)();
    int (*d_write)();
    int (*d_getstat)();
    int (*d_setstat)();
    int (*d_mmap)();
    int (*d_async_in)();
    int (*d_reset)();
    int (*d_port_death)();
    int d_subdev;
    int (*d_dev_info)();
};
```

```c
struct dev_ops dev_name_list[];
int dev_name_count;
```
Early Stages of a Kernel’s Life

- Kernel links
- Kernel loads and toggles lights
- Printf works
- Debugger works
- Pmap initialized
- Virtual memory enabled (first VM fault)
- First user process (bootstrap)
- First system call (from bootstrap)
- Server loads
- Paging file found
- Init doesn’t die
- First signal (from /bin/sh)
- Single user # prompt
- Multi-user
- Network works
- Compiles lisp

Sophisticated Debugging

Stable machine
Symbolic source level debugging
Debugging scripts

Test machine
Hooks into tty driver
Hooks into trap handler
Small stub
  read/write memory and registers
  single step and set breakpoints

Debugging

Two schools of thought

“Hell yes, I’m from Texas”
core dumps
printf
adb

Debugger: use ddb, it’s just printf

New Yorker approach
Symbolic
Source Level
Debugging scripts

Debugger: use remote GNU gdb

Debugging Tips

Investment
This isn’t the last bug!
Every hour invested in debuggers pays off
Learn how to write GDB scripts
Know your machine

Read the code carefully
Bugs deep in kernels are hard to find

User Level Testing
Interactive testing and scripts
Lex and yacc can build powerful tools
Test modules as you go along

In Kernel Testing
Build a small kernel with printf

Simple Counters
In trap handlers, I/O routines, cache flushes
Do the numbers make sense?
More Debugging Tips

Use assert
assert(addr != 0);

Conditionalized print statements
Powerful if used with the debugger and patched at runtime

if (addr == catchme)
    printf(“addr matches catchme\n”);

Make a special printf syscall
Always prints a string from user space

Build debugging into your system
Flags on interrupt/trap stack frames
Don’t hide registers from user
Make debugging output easy to read (PSW)
Whenever possible write in C not assembly

When you’re frustrated… build a new tool

Recommended Reading

Machine-Independent Virtual Memory Management for Paged Uniprocessors and Multiprocessor Architectures
Richard Rashid et al.
CMU technical report CMU-CS-87-140
(also in ASPLOS II, October 1987)

Architecture-Independent Virtual Memory Management for Parallel and Distributed Environments: The Mach Approach
Avadis Tevanian Jr.’s Ph.D. Thesis
CMU technical report CMU-CS-88-106

Exporting a User Interface to Memory Management from a Communication-Oriented Operating System
Michael Young’s Ph.D. Thesis
CMU technical report CMU-CS-89-202

Mach Virtual Memory

Basic Data Structures

vm_page
Describes a physical page of memory

vm_object
A contiguous repository of data some in backing store, some in memory

vm_map_entry
A mapping of contiguous virtual address space and protection to a vm_object

pmap
A “physical map”, the machine dependent representation for mappings (page tables)

vm_map
A collection of vm_map_entries and a pmap there is one vm_map per task

Mach Virtual Memory

Virtual Memory Data Structures
Resolving a Page Fault
Copy-on-write
Physical Maps (pmaps)
Pmap Routines
Page Reference Bits
Virtual Cache Alignment
Zone Package
Grabbing Physical Pages
Simple VM Example

![Diagram of vm_map and pmap relationships]

Key points
- Memory object represents a piece of data and physical memory is a cache of this data.
- Mapping entries map a contiguous range of virtual addresses with common protection onto a memory object.

Memory Object

![Diagram of vm_object structure]

Fields in vm_object structure
- Size of object
- Reference count
- Pager for object
- Offset into pager
- Pointer to shadow object
- Pointer to copy object
- Miscellaneous flags:
  - temporary: Object can not be changed by an external memory manager
  - canPersist: Object can persist after last reference
  - internal: Created by kernel managed by default pager

Object/Offset Hash Table

![Diagram of vm_page and vm_page_lookup relationships]

Fields in vm_page structure
- Links for page queues
  - double linked page list
- Object and offset for page
- Physical address
- Flags:
  - inactive, active and free
    - Page list page is on
  - busy
    - Page in transit from pager
  - tabled
    - vm_page is in object/offset table
  - fictitious
    - vm_page is placeholder in object

What vm_page is in an object at a specific offset?

Use a hash table

Hash is a function of object and offset

vm_page:1 vm_page_lookup(object, offset)
- Lookup a page in an object
Mapping Entry

Fields in vm_map_entry structure

- Virtual address start and end
  - Always page aligned
- Current and maximum protection
  - read/write/execute
- Inheritance with child on fork
  - shared, copied or none
- Miscellaneous flags
  - needs_copy
    - Region marked as copy-on-write

Virtual Memory Map

Fields in vm_map structure

- Minimum and maximum virtual address
- Size of address map
- Reference count
- Head and tail of mapping entries list
- Hint for mapping entry search
- Pmap associated with map

Resolving a Simple Page Fault

1. Start with map and virtual address
2. Find map entry containing virtual address
3. Get object and offset from map entry
4. Add offset into map entry to offset into object
5. Find vm_page structure from object/offset hash table
6. If vm_page is VM_PAGE_NULL then zero fill and enter mapping into pmap
7. If vm_page resident then enter mapping into pmap
8. If vm_page is paged out then ask pager for page when provided enter mapping into pmap

Copy-on-write

Transparent optimization for copying data

- Access to page is marked read only to both parties
- Writing to a page causes a fault and a new private copy of the page is made
- Can only share on a page granularity

Two forms

- Symmetric copy-on-write
  - Both source and destination treated the same
- Asymmetric copy-on-write
  - Used when an external memory manager is involved
Symmetric Copy-on-write

Copy operation

Point destination mapping entry at source object
Set \textit{needs\_copy} for both mapping entries
Remove write access to all pages in object
\textit{(removed by using physical address)}

Write to Page

Write operation

Causes a protection fault
Shadow object is created and a copy of the faulting page is inserted in the shadow object
Unmodified pages are still in original object
Write by source or destination treated the same

Copy-on-write Shadow Chains

Multiple copy-on-write operations can result in a shadow chain
Attempt is made to collapse chain when possible

External Memory Managers

Problem

Object is backed by an external memory manager
Memory manager wants to see all changes to the object
Original object will not see the changes with symmetric copy-on-write

Solution

Make original object a “copy object”
Copy objects push pages up to a shadow object before they are modified
Copy objects reflects all changes
Asymmetric Copy-on-write

Write operation

If first write to page in copy object then push an unmodified copy to shadow and then modify page in copy object.

If first write to page in shadow object then pull an unmodified copy from copy object and then modify page in shadow object.

Pmap Dictionary Entry

A pmap dictionary entry consists of

- Virtual address
- Physical address
- Protection (read/write/execute)
- Modified flag
- Referenced flag
- Wired flag

Any non-wired entry can be discarded at any time and regenerated by the machine independent data structures when needed.

Physical Maps (pmaps)

pmaps

A pmap is simply a dictionary structure that supports the following operations:

- insert
- remove
- modify
- lookup

Both hardware and the operating system query and modify the dictionary.

Hardware usually dictates the internal format of the dictionary.
The Physical to Virtual Table

The pmap module must find entries given either the virtual or physical address

Length of table is the number of physical pages of memory managed by virtual memory system

Each entry is a linked list of (pmap, virtual address) pairs mapped to that physical page of memory

A Word on Addresses

All addresses, both virtual and physical, are byte addresses unless specifically stated otherwise

Virtual addresses are always qualified by the pmap module they are in

A range of addresses, whether specified as a start and end address or a start and length, always includes the first address and excludes the last address

The addresses for a page of memory will be given as the first address in the page

The page size must be a multiple of the physical page size but need not be the same

Pmap Bootstrap Routines

void pmap_bootstrap()
Called by init to setup enough of the pmap module to allow the kernel to run with virtual memory enabled

pmap_bootstrap is not part of the pmap interface

unsigned int pmap_free_pages()
Return the number of free physical pages that have not been allocated (used to size the object/offset hash table)

void pmap_init()
Called by vm_init() to initialize any structures or zones that the pmap system needs to map virtual memory

Pmap Bootstrap Options

Two options for bootstrapping
Define MACHINE_PAGES in pmap.h if the pmap module wants complete control of page allocation

A useful thing to do is map all of physical memory by the kernel (with block TLB entries if possible)

If you define MACHINE_PAGES then implement

vm_offset_t pmap_steal_memory(size)
Allocate and return the address of a piece of kernel memory that is size bytes long

void pmap_startup(startp, endp)
Allocate and initialize a vm_page_t structure for all physical memory to be managed and return the starting and ending virtual address for the kernel in startp and endp
Non MACHINE_PAGES option

If you don’t define MACHINE_PAGES then implement

**void pmap_virtual_space(startp, endp)**
Return the starting and ending virtual address for the kernel in `startp` and `endp`

**boolean_t pmap_next_page(phys_addr)**
Return TRUE if there is another page of physical memory to be allocated and return the physical address of the page in `phys_addr`

Pmap Create and Delete

**pmap_t pmap_create()**
Create and return a pmap

**void pmap_reference(pmap)**
Increment the reference count of this pmap

**void pmap_destroy(pmap)**
Decrement the pmap’s reference count and delete the pmap if zero

All entries will be removed from the pmap before the final `pmap_destroy` is called

Pmap Context Switch

**void PMAP_ACTIVATE(pmap, thread, cpu)**
Activate the pmap for use by this thread on this cpu

**void PMAP_DEACTIVATE(pmap, thread, cpu)**
Deactivate the pmap used by this thread on this cpu

**void PMAP_CONTEXT(pmap, thread)**
Switch pmap to a new thread in the same task

These are typically `#define` macros in `pmap.h` and are sometimes null macros

In our example `PMAP_ACTIVATE` would just set the root page table pointer, the other two would be null macros

Zero Fill and Copy Physical Pages

**void pmap_zero_page(phys_addr)**
Zero fill a page of memory at the specified physical address

**void pmap_copy_page(src_addr, dst_addr)**
Copy a page of memory at physical address `src_addr` to physical address `dst_addr`

The source page for `pmap_copy_page` may or may not be mapped, the destination page will never be mapped
Miscellaneous Routines

pmap_t pmap_kernel()
   Return the pmap for the kernel

int pmap_resident_count(pmap)
   Return the number of physical pages mapped by
   this pmap

vm_offset_t pmap_phys_address(phys_page)
   Return the byte address of physical page phys_page.
   Note: phys_page is the machine dependent physical
   page number not a byte address

These routines are small enough that they are usually
implemented as #define macros in pmap.h

Pmap Insert Routine

void pmap_enter(pmap, virt_addr, phys_addr,
               min_prot, max_prot, wired)
   Create a mapping in pmap for virtual address
   virt_addr to physical address phys_addr.

   The minimum protection required is min_prot
   which is the protection passed to vm_fault()

   The maximum protection allowed is max_prot

   If the wired flag is set then this mapping must
   never cause a page fault

Pmap_enter is the only routine that can increase
access to a page of memory

min_prot was added for machines with split
instruction and data TLBs that are software loaded

Pmap Lookup Routines

vm_offset_t pmap_extract(pmap, virt_addr)
   Return the physical address mapped by the virtual
   address in the specified pmap or 0 if there is no
   known mapping

boolean_t pmap_is_referenced(phys_addr)
   Return whether the page at the specified physical
   address has been referenced since the last call to
   pmap_clear_reference() was made

boolean_t pmap_is_modified(phys_addr)
   Return whether the page at the specified physical
   address has been modified since the last call to
   pmap_clear_modify() was made

Pmap Modification Routines

void pmap_set_modify(phys_addr)
   Set the modification bit on the page at the
   specified physical address

void pmap_clear_modify(phys_addr)
   Clear the modification bit on the page at the
   specified physical address

void pmap_clear_reference(phys_addr)
   Clear the reference bit on the page at the
   specified physical address

void pmap_change_wiring(pmap, virt_addr, wired)
   Change the wiring status for the specified
   virtual address
Change Protection

```c
void pmap_protect(pmap, start, end, prot)
    Change the protection on the range of virtual addresses in the specified pmap
```

```c
void pmap_page_protect(phys_addr, prot)
    Change the protection for all mappings to the specified physical page
```

A protection of VM_PROT_NONE should remove the mapping

If the caller attempts to increase access then remove the mapping, only `pmap_enter()` can increase access

Machine Specific Attributes

```c
kern_return_t pmap_attribute(pmap, address, size, attribute, value)
    Set a specific attribute on a range of addresses in the given pmap
```

Attributes

- MATTR_CACHE
  - Cachability

Value

- MATTR_VAL_CACHE_FLUSH
  - Flush all caches
- MATTR_VAL_DCACHE_FLUSH
  - Flush data caches
- MATTR_VAL_ICACHE_FLUSH
  - Flush instruction caches

Add machine specific attributes if needed

Optional Pmap Routines

```c
void pmap_collect(pmap)
    Garbage collect pages for this pmap that are no longer used
```

```c
void pmap_copy(dst_pmap, src_pmap, dst_addr, length, src_addr)
    Copy the source pmap entries from for the address range src_addr to src_addr + length into the destination pmap at address dst_addr
```

```c
void pmap_pageable(pmap, start, end, pageable)
    Make the specified pages in the given pmap pageable (or not) as requested. `pmap_enter()` will also specify that these pages are to be wired down if appropriate
```

These routines are optional and may be provided as null macros in pmap.h

Memory Manipulation Routines

```c
void bcopy(src, dst, length)
    Copy length bytes from src to dst
```

```c
void bzero(addr, length)
    Zero length bytes starting at addr
```

```c
kern_return_t copyin(src, dst, length)
    Copy length bytes from the current thread's address src to the kernel address dst
```

```c
kern_return_t copyout(src, dst, length)
    Copy length bytes from the kernel address src to the current thread's address dst
```

```c
kern_return_t copyinmsg(src, dst, length)
```

```c
kern_return_t copyoutmsg(src, dst, length)
```

Same as `copyin` and `copyout` except that `src` and `dst` are word aligned and `length` is a multiple of 4
Bad User Addresses

Bad user addresses in copyin or copyout

Before accessing user space load error recovery routine in recover in thread structure.

Clear recover when completed.

In fault handler if kernel data fault and recover is not null then patch program counter to return to error recovery routine.

Alternate method

Hard code start and ending addressed of copyin and copyout routines and the recovery routine.

Page Reference Bits

If your hardware doesn’t have page reference bits you might find it advantageous to let the machine dependent code simulate them.

To do this add the following two lines to pmap.h

```c
#define pmap_is_referenced(phys) (FALSE)
#define pmap_clear_reference(phys) \n  pmap_page_protect(phys, VM_PROT_NONE)
```

See the paper

*Page Replacement and Reference Bit Emulation in Mach*

by Richard P. Draves.

Virtual Cache Alignment

If you have virtual caches then you can allow the pmap module to influence the placement of shared memory between address spaces.

```c
#define PMAP_ALIGN in pmap.h
```

Requires a few new routines to be written:

- `pmap_align_init`
- `pmap_align_copy`
- `pmap_align_set`
- `pmap_align_propose`

See the pmap module for the Hewlett-Packard parisc machines.

Also see the paper

*Consistency Management for Virtually Indexed Caches*

Bob Wheeler and Brian N. Bershad

CMU technical report CMU-CS-92-182

(also in ASPLOS V, October 1992)

Zone Package

Zones allow fast allocation of a fixed size structure.

```c
zone_t zinit(size, max, alloc, pageable, name)
```

Initialize a new zone with elements of `size` bytes using at most `max` bytes of memory, allocate space in `alloc` byte chunks, `pageable` declares if the zone may be paged while `name` is the name of the zone.

```c
vm_offset_t zalloc(zone)
```

Allocate an element from the zone.

```c
vm_offset_t zget(zone)
```

Allocate an element from the zone without blocking and return 0 if none available.

```c
void zfree(zone, elem)
```

Free an element back to the specified zone.
Grabbing Physical Pages

Routines for grabbing a physical page from the free list

```c
vm_page_t vm_page_grab()
    Remove a page from the free list or return VM_PAGE_NULL if the free list is too small

void vm_page_wait(continuation)
    Wait for a free page to become available
    while ((p = vm_page_grab()) == VM_PAGE_NULL)
    vm_page_wait((void (*)(())) 0);

void vm_page_release(mem)
    Return a page to the free list

int vm_page_grab_phys_addr()
    Grab a page of memory from the free list and return the physical address or -1 if no page is available use this only if the page will never be freed
```

Task and Thread Data Structures

```
| thread   | thread state & scheduling information |
| pcb      | user state on entry to kernel |
| kernel regs | registers saved across context switch |
| kernel stack | thread's kernel runtime stack |
| task     | common task information |
| map      | task's virtual memory map |
| pmap     | task's physical map |
```

Pointers to structures have a “_t” on the end of them, (i.e. task_t, thread_t)

Part III - Saving and Restoring State

Task and thread data structures
Kernel entry and exit
    System calls
    Trap and interrupts
Kernel and interrupt stack
Saved state Structure
    Where to save state
    Continuations
    State routines
    Trap handlers
Asynchronous system traps

Kernel Entry and Exit

Three types of kernel entry
    System call
    Trap
    Interrupt

Calling conventions
    Caller saves registers
    Callee saves registers
**System Calls**

User code saves caller-saves registers before syscall

Only use caller-saves registers in syscall stub or kernel entry and exit code

User registers are saved in the pcb

Should only have to save a few things like the return pointer and the user stack pointer

Switch to kernel address space and onto kernel stack

Kernel code will save callee-saves registers

**Trap or Interrupt in Kernel Mode**

Very similar to user mode trap or interrupt

Don’t have to change address space to kernel

**Kernel and Interrupt Stack**

Each thread has a kernel stack which is typically small (4k bytes)

Each cpu has an interrupt stack which is typically larger (20k - 40k bytes)

Could have only kernel stacks but then each would have to be much larger

Using an interrupt stack allows nested interrupts without overflowing a kernel stack

Thread may block if using kernel stack

Thread may not block if using interrupt stack
Saved State Structure

Layout for registers saved on kernel entry and exit

Add debugging flags such as reason for kernel entry

Add flag to allow partial register reload for debuggers and thread_setstatus()

Make life easier and use the same structure for pcb, kernel stack, interrupt stack and thread_status

Make room for all registers from the start

Continuations

Problem
Kernel stacks must be wired which requires lots of physical memory

Solution
Many threads are blocked in a known state

Discard kernel stack when blocked thread will return immediately to user mode and provide instead a routine to call to leave kernel

Complication
Must save user callee-saves registers if continuation is possible

See the paper
Using Continuations to Implement Thread Management and Communication in Operating Systems
Richard P. Draves, et al.
Thirteenth SOSP, October 1991

Native System Calls

typedef struct {
    int mach_trap_arg_count;
    int (*mach_trap_function)();
    boolean mach_trap_reused;
} mach_trap_t;

mach_trap_t mach_trap_table[];

int mach_trap_count
Continuation Stack Routines

void stack_attach(thread, stack, continuation)
Attach the stack to the thread and set the return
pointer to run the continuation

boolean_t stack_alloc_try(thread, continuation)
Non-blocking attempt to allocate and attach a
kernel stack

void stack_alloc(thread, continuation)
Allocate and attach a kernel stack, may block

void stack_free(thread)
Free a thread’s kernel stack

void stack_collect()
Free excess kernel stacks

Continuation Routines

void call_continuation(routine)
Reset kernel stack pointer to base of kernel stack
and call the specified routine

void thread_syscall_return(return_value)
Place the argument in the syscall return register,
restore state frompcb and return to user mode

void thread_set_syscall_return(return_value)
Set the eventual return value for this syscall

void thread_exception_return()
Restore state from pcb and return to user mode

void thread_bootstrap_return()
Return to user mode for the first time

PCB Routines

void pcb_module_init()
Called at bootstrap time to initialize pcb data
structures

void pcb_init(thread)
Allocate and initialize a pcb and attach it to
the specified thread

void pcb_terminate(thread)
Free the pcb attached to the specified thread

kern_return_t thread_setstatus(thread, flavor,
state, count)
Set the user registers in the pcb

kern_return_t thread_getstatus(thread, flavor,
state, count)
Get the user registers from the pcb

Context Switch

Save and restore callee-saves registers and stack

Save and restore the context from the bottom of
the kernel stack

void load_context(new_thread)
Load the context of the first thread

void switch_context(old_thread, continuation,
new_thread)
Save the context of the old thread, set swap_func
in the old_thread’s thread structure to run the
continuation when resumed, restore the context
of the new_thread

Keep old_thread in arg0 for thread_contiue
and return old_thread for switch_context

stack_handoff(old_thread, new_thread)
Move the stack from the old thread to the new one
Miscellaneous State Routines

**vm_offset_t set_user_regs(stack_base, stack_size, entry, arg_size)**
Allocate argument area, set registers for first user thread and return where to store the arguments on the stack

**vm_offset_t user_stack_low(stack_size)**
Return preferred address of user stack, always returns low address of stack

Asynchronous System Traps

ASTs are a way to force a thread to take a trap when it about to return to user mode

AST state is a per processor state

Used to implement involuntary context switches

If MACHINE_AST is defined then implement

```c
astoff(cpu)
```
called to disable AST trap on cpu

```c
aston(cpu)
```
called to enable AST trap on cpu

Else use the value of need_ast[cpu]

Trap Handlers

Calls made from trap handlers

**Virtual memory faults**

```c
kern_return_t vm_fault(map, vaddr, fault_type, change_wiring, resume, continuation)
```

**Clock interrupt**

```c
void clock_interrupt(usec, usermode, basepri)
```

**Exceptions**

```c
void exception(exception_type, code, subcode)
```

Interrupt Priority Level

Spl is the level of interrupts that we are blocking

**only return from interrupt can lower spl**

kernel uses (from highest to lowest)

```c
int splhigh()
int splclock()
int splsched()
int splbio()
int splimp()
int splitty()
int splsoftclock()
int spl0()
```
block all interrupts
block clock and below
block clock and below
block block I/O and below
block network and below
block terminal and below
block softclock and below
interrupts not blocked

Above routines return old spl level

```c
void splx(s)
```
set spl to level s

```c
void set_softclock()
```
Called from clock_interrupt to schedule a lower level interrupt
Part IV - User Code

Libmach

Contains all the stubs to call the kernel

Machine dependent code

- setjmp and longjmp
- bzero and bcopy
- fork
  Special fork that calls mach_init() in child
- crt0.s
  Special version that calls mach_init() and cthread_init() routines

Cthread Locks

spin_lock_t
  Typedef for a lock

SPIN_LOCK_INITIALIZER
  Static initializer for a lock

spin_lock_init(s)
  Dynamic initializer for a lock

spin_lock_locked(s)
  Test if a lock is locked

Cthread Locks Continued

spin_try_lock(s)
  Try and acquire a lock, return 0 if successful

spin_unlock(s)
  Spin unlock

If you are on a uniprocessor you might want to look at

Fast Mutual Exclusion for Uniprocessors
Brian N. Bershad et.al.
CMU technical report CMU-CS-92-183
(also in ASPLOS V, October 1992)
Cthread Routines

cproc_setup(child, thread, routine)
   Set up the initial state of a cthread so that it
   will invoke routine(child) when it is resumed

void cproc_switch(cur, next, lock)
   Suspend the current thread and resume the
   next one

void cproc_start_wait(parent_context, child,
   stackp, lock)
   Save the current threads state, switch to a new
   stack and call cproc_waiting(child)

void cproc_prepare(child, child_context, stack)
   Create a call frame and context on the given stack
   so that when invoked by cproc_switch it calls
   cthread_body(child)

Emulated System Calls

1. User process executes syscall trap
2. Emulated system call redirected to emulator
3. Emulator builds message, calls mach_msg_send
4. A server thread that previously called
   mach_msg_receive and is waiting in the kernel
   takes message to server
5. The server does mach_msg_send to send a reply
6. The user's thread waiting in the kernel takes the
   reply message to the emulator
7. As an optimization the emulator returns directly
   to the server

Emulated Syscall Data Structures

typedef struct eml_dispatch {
   decl_simple_lock_data(, lock)
   int ref_count;
   int disp_count;
   int disp_min;
   eml_routine disp_vector[1];
} *eml_dispatch;

The emulated syscall dispatch table pointer in
active_threads[0] → task → eml_dispatch

If you cache the emulation dispatch pointer...

void syscall_emulation_sync(task)
   Called when the task's emulation vector changes

Emulator Routines

void emul_setup(task)
   Call task_set_emulation(task, routine,
   syscall_number) for each system call
   Most syscalls are redirected to emul_common
   except e_fork which is directed to emul_save_regs

Positive syscall numbers are UNIX syscalls
   negative numbers are CMU extensions
**Emul_common**

Non-fork system calls

1. Save essential caller-saves registers
2. Acquire `emul_stack_lock`
3. Call `emul_stack_alloc()` to get a stack
4. Release `emul_stack_lock`
5. Switch to emulator stack
6. Call `emul_syscall()` to create message to server
7. Acquire `emul_stack_lock`
8. Turn in emulator stack and return to user stack
9. Release `emul_stack_lock`
10. Check for signals and call signal handler
11. Clean up and return to user

**Emul_save_regs**

Similar to `emul_common` except that you must save and restore argument and syscode registers in parent

In child you must call `child_init()` to initialize the emulator

**Emul_syscall**

Collects arguments and calls MiG stub to start remote procedure call to server

On return checks for system calls to be restarted

Checks for signals and dispatches them if needed

**Signals**

**void take_signal(...)**

Call `bsd_take_signal` to get any signals pending

Build signal context
Fake return so that you go to handler

**sigreturn**

Called by signal handler

if using mapped U area
   call `e_shared_sigreturn()`
else
   call `bsd_sigreturn()`

The server may need assistance from the kernel to restore the state
A few machine specific routines needed for loading executable, delivering signals, ptrace()...

`boolean_t machine_exception(...)`
Where the exception() call ends up, translates a mach exception into a UNIX exception

Create `cdewsw` and `bdewsw` tables in `conf.c`

Most single server devices use generic devices to interface with the kernel