μ-Kernel Construction

Fundamental Abstractions

- Thread
- Address Space

What is a thread?
How to implement?

What conclusions can we draw from our analysis with respect to μK construction?
Construction conclusion

From the view of the designer there are two alternatives.

Single Kernel Stack
- Only one stack is used all the time.

Per-Thread Kernel Stack
- Every thread has a kernel stack.

Single Kernel Stack

- either continuations
  - complex to program
  - must be conservative in state saved (any state that might be needed)
  - Mach (Davies), L4:Strawberry, NICTA Pistachio, Oke4
- or stateless kernel
  - no kernel threads, kernel not interruptible, difficult to program
  - request all potentially required resources prior to execution
  - blocking system calls must always be restartable
  - Processor-provided stack management cannot be the way
  - system calls need to be kept simple "atomic"
  - kernel can be exchanged on-the-fly
    - e.g. the fluke kernel from Utah
- low cache footprint
  - always the same stack is used
  - reduced memory footprint

Per-Thread Kernel Stack

- simple, flexible
- kernel can always use threads, no special techniques required for keeping state while interrupted / blocked
- no conceptual difference between kernel mode and user mode
  - e.g. L4
- but larger cache footprint
- larger memory footprint

Kernel Entry/Exit

- A look at mechanics of kernel entry and exit
- Optimisations
- Context switching
trap / fault occurs (INT n / exception / interrupt)
  push user esp on to kernel stack, load kernel esp

push user esp on to kernel stack, load kernel esp
push user efflags, reset flags (I=0, S=0)
push user eip, load kernel entry eip

trap / fault occurs (INT n / exception / interrupt)
push user esp on to kernel stack, load kernel esp
push user efflags, reset flags (I=0, S=0)
push user esp, load kernel entry eip

trap / fault occurs (INT n / exception / interrupt)
push user esp on to kernel stack, load kernel esp
push user efflags, reset flags (I=0, S=0)
push user esp, load kernel entry eip
push X : error code (hw, at exception) or kernel-call type

int 0x32
push X
pusha
...  
popa
add $4, esp
iret

Error code e.g. 3 means page fault
Push all, the register content to the stack
Pop all, see below
esp = esp + 4
the old esp

Interrupt return

System call (IA32)
**Sysenter/Sysexit**

- Fast kernel entry/exit
  - Only between ring 0 and 3
  - Avoid memory references specifying kernel entry point and saving state
- Use Model Specific Register (MSR) to specify kernel entry
  - Kernel IP, Kernel SP
  - Flat 4GB segments
  - Saves no state for exit
- Sysenter
  - EIP = MSR(Kernel IP)
  - ESP = MSR(Kernel SP)
  - Eflags.I = 0, FLAGS.S = 0
- Sysexit
  - ESP = ECX
  - EIP = EDX
  - Eflags.S = 3
- User-level has to provide IP and SP
- by convention – registers (EDX, EDX?)
- Flags undefined
- Kernel has to re-enable interrupts

**Kernel-stack state**

Uniprocessor:

- Any kstack ≠ myself is current!
  - (my kstack below [esp] is also current when in kernel mode)

One thread is running and all the others are in their kernel-state and can analyze their stacks. All processes except the running one are in kernel mode.

**Sysenter/Sysexit**

- Emulate int instruction (ECX=USP, EDX=UIP)
  - sub $20, esp
  - mov ecx, 16(esp)
  - mov edx, 4(esp)
  - mov $5, (esp)
- Emulate int instruction
  - mov 16(esp), ecx
  - mov 4(esp), edx
  - sti
  - sysexit

**Thread switch (IA32)**

To find out the starting address from the tcb:

```plaintext
align tcb:
mov esp, ebp
and -sizeof tcb, ebp
```

**Kernel-stack state**

Remember:

- We need to find
  - any thread's tcb starting from uid
  - the currently executing thread's tcb
SwitchPthreadsP(IA32)

CPU

Switch threads (IA32)

- int 0x32, push registers of the green thread
- switch kernel stacks (store and load esp)

Mips R4600

- 32 Registers
- no hardware stack support
- special registers
  - exception IP, status, etc.
  - single registers, unstacked!
- Soft TLB !!

Kernel has to parse page table
Exceptions on MIPS

- On an exception (syscall, interrupt, ...)
- Loads Exc PC with faulting instruction
- Sets status register
  - Kernel mode, interrupts disabled, in exception.
- Jumps to 0xffffffff80000180

To switch to kernel mode

- Save relevant user state
- Set up a safe kernel execution environment
- Switch to kernel stack
- Able to handle kernel exceptions
- Potentially enable interrupts

Problems

- No stack pointer???
  - Defined by convention sp (r29)
- Load/Store Architecture: no registers to work with???
  - By convention k0, k1 (r31, r30) for kernel use only

System Calls - Kernel Side

- Things left to do
  - Change to kernel stack
  - Preserve registers by saving to memory (the stack)
  - Leave saved registers somewhere accessible to
    - Read arguments
    - Store return values
    - Do the "read()"
    - Restore registers
    - Switch back to user stack
    - Return to application

```assembly
exception:
  move k1, sp          /* Save previous stack pointer in k1 */
  sfd0 k0, c2_status   /* Get status register */
  andi k0, k0, CST_Kup /* Check the we-were-in-user-mode bit */
  beq k0, $0, 1f       /* If clear, from kernel, already have stack */
  nop                  /* delay slot */

/* Coming from user mode - load kernel stack into sp */
  la k0, curkstack
  lw sp, 0(k0)         /* get its value */
  nop                  /* delay slot */

1:
  sfd0 k0, c2 cause    /* Now, load the exception cause. */
  j common_exception   /* Skip to common code */
  nop                  /* delay slot */
```

Note k0, k1 registers available for kernel use
common_exception:

/*
 * At this point:
 *  - Interrupts are off. (The processor did this for us.)
 *  - k0 contains the exception cause value.
 *  - k1 contains the old stack pointer.
 *  - sp points into the kernel stack.
 *  - All other registers are untouched.
 */

/*
 * Allocate stack space for 37 words to hold the trap frame,
 * plus four more words for a minimal argument block.
 */
addi sp, sp, -164

The order here must match mips/include/trapframe.h. */
sw ra, 160(sp) /* dummy for gdb */
sw s8, 156(sp) /* save s8 */
sw sp, 152(sp) /* dummy for gdb */
sw sp, 148(sp) /* save sp */
sw k1, 144(sp) /* dummy for gdb */
sw k0, 140(sp) /* dummy for gdb */
sw k1, 152(sp) /* real saved sp */
nop /* delay slot for store */
mfc0 k1, c0_epc /* Copr.0 reg 13 == PC for exception */
sw k1, 160(sp) /* real saved PC */

These six stores are a "hack" to avoid confusing GDB.
You can ignore the details of why and how.

The real work starts here

Save all the registers on the kernel stack

We can now use the other registers (t0, t1) that we have preserved on the stack

Create a pointer to the base of the saved registers and state in the first argument register
By creating a pointer to here of type struct trapframe *, we can access the user’s saved registers as normal variables within “C.”

**TCB structure**

- Thread Id
- Cal Id = UTCB

**Construction Conclusions (1)**

- Thread state must be saved / restored on thread switch.
- We need a thread control block (TCB) per thread.
- TCBs must be kernel objects.
  - TCBs implement threads.

- We need to find
  - **any thread’s tcb starting from its uid**
  - the currently executing thread’s TCB (per processor)

**Thread ID**

- thread number
  - to find the tcb
- thread version number
  - to make thread ids “unique” in time

**Thread ID → TCB (a)**

- Indirect via table

```
  mov     t0, k1
  sll t1, 5
  add t0, t1, 4
  sub sp, 24
  mov %rax, %rbx
  jnz invalid_thread_id
```
Thread ID → TCB (b)

- **direct address**

```
mov    thread_id, %eax
mov    %eax, %ebx
and    mask_thread_no, %eax
add    offset tcb_array, %eax
cmp    %ebx, OFF_TCB_MYSELF(%eax)
jmp    invalid_thread_id
```

**Trick:**

Allocate physical parts of table on demand, dependent on the max number of allocated tcb.
Map all remaining parts to a 0-filled page.
Any access to corresponding threads will result in "invalid thread id".
However: requires 4K pages in this table.

- **TCB pointer array** requires 1M virtual memory for 256K potential threads.

**AS Layout**

32bits, virt tcb, entire PM

- user regions
- shared system regions
- per-space system regions

**FPU Context Switching**

- **Strict switching**
  
  **Thread switch:**
  
  Store current thread’s FPU state
  
  Load new thread’s FPU state

- Extremely expensive
  
  - IA-32’s full SSE2 state is 512 Bytes
  
  - IA-64’s floating point state is ~1.5KB

- May not even be required
  
  - Threads do not always use FPU
Lazy FPU switching

- Lock FPU on thread switch
- Unlock at first use – exception handled by kernel
  Unlock FPU
  If fpu_owner is current
  Load new state from fpu_owner
  fpu_owner = current

Kernel
FPU

pacman
lock

IPC

Functionality & Interface

What IPC primitives do we need to communicate?

- Send to
  (a specified thread)
  Receive from
  (a specified thread)

- Two threads can communicate
- Can create specific protocols without fear of interference from other threads
- Other threads block until it’s their turn
- Problem:
  How to communicate with a thread unknown a priori
  (e.g., a server’s clients)

What IPC primitives do we need to communicate?

- Send to
  (a specified thread)
  Receive from
  (a specified thread)

- Scenario:
  A client thread sends a message to a server
  Expecting a response.
  The server replies
  Expecting the client thread to be ready to receive.
  Issue: The client might be preempted between the
  send to and receive from.

What IPC primitives do we need to communicate?

- Send to
  (a specified thread)
  Receive from
  (a specified thread)

- Receive
  (from any thread)

- Call
  (send to, receive from specified thread)

- Send to & Receive
  (send to, receive from any thread)

- Send to, Receive from
  (send to, receive from specified different threads)

Are other combinations appropriate?

Atomic operation to ensure that server’s (callee’s) reply cannot arrive before client (caller) is ready to receive

Atomic operation for optimization reasons. Typically used by servers to reply and wait for the next request (from anyone).

What message types are appropriate?

- Register
  Short messages we hope to make fast by avoiding memory access to transfer the message during IPC

- Direct
  Short messages that can be transferred to or from a process

- Can be combined
  Short messages can be transferred to or from a process

- Map pages (optional)
  Messages that map pages from sender to receiver
What message types are appropriate?

- Register
  - Short messages we hope to make fast by avoiding memory access to transfer the message during IPC
- Strings (optional)
  - Memory message we construct to send
- Indirect strings (optional)
  - Temporary messages sent in place
- Map pages (optional)
  - Messages that map pages from sender to receiver

IPC - API

- Operations
  - Send to
  - Receive from
  - Receive
  - Call
  - Send to & Receive
  - Send to, Receive from

Message Types

- Registers
- Strings
- Map pages

Problem

- How to we deal with threads that are:
  - Uncooperative
  - Malfunctioning
  - Malicious
- That might result in an IPC operation never completing?

IPC - API

- Timeouts (V2, X.2)
  - snd timeout, rcv timeout

IPC - API

- Attack through receiver’s pager:

IPC - API

- Attack through sender’s pager:
Timeout Issues

- What timeout values are typical or necessary?
- How do we encode timeouts to minimize space needed to specify all four values.

Timeout values

- Infinite
  - Client waiting for a server
- 0 (zero)
  - Server responding to a client
- Polling
- Specific time
  - 1us – 19 h (log)

To Compact the Timeout Encoding

- Assume short timeout need to finer granularity than long timeouts
- Timeouts can always be combined to achieve long fine-grain timeouts

\[
\text{send/receive timeout} = \begin{cases} 
\infty & \text{if } e = 0 \\
4^{15+m} & \text{if } e > 0 \\
0 & \text{if } m = 0, e = 0
\end{cases}
\]

Page fault timeout has no mantissa

\[
\text{page fault timeout} = \begin{cases} 
\infty & \text{if } p = 0 \\
4^{15-p} & \text{if } 0 < p < 15 \\
0 & \text{if } p = 15
\end{cases}
\]

Timeout Range of Values (seconds) [V2, V X.0]

<table>
<thead>
<tr>
<th>( e )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>( m )</td>
<td>( \infty )</td>
<td>268,435,460</td>
<td>287,508,864</td>
<td>16,711,260,320</td>
<td>1,048,576</td>
<td>64,512</td>
<td>4,096</td>
<td>256</td>
<td>16</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>0.5</td>
<td>0.25</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0.000001</td>
<td>0.000004</td>
<td>0.000016</td>
<td>0.000064</td>
<td>0.000256</td>
<td>0.001024</td>
<td>0.004096</td>
<td>0.016384</td>
<td>0.065536</td>
<td>0.256</td>
<td>1.024</td>
<td>4.096</td>
<td>16</td>
<td>64</td>
<td>256</td>
<td></td>
</tr>
</tbody>
</table>

IPC - API

- Timeouts (V2, V X.0)
  - snd timeout, rcv timeout
  - snd-pf / rcv-pf timeout

Timeout values

- 0
- Infinite
- 1us – 19 h (log)

Compact 32-bit encoding
To Encode for IPC
- Send to
- Receive from
- Call
- Send to & Receive
- Send to, Receive from
- Destination thread ID
- Source thread ID
- Send registers
- Receiver registers
- Number of send strings
- Send string start for each string
- Send string size for each string
- Number of receive strings
- Receiver string start for each string
- Receiver string size for each string
- Number of map pages
- Page range for each map page
- Receiver window for mappings
- IPC result code
- Send timeout
- Receiver timeout
- Send 0 timeout
- Receiver 0 timeout
- Receiver from thread ID
- Specify deallocating IPC
- Thread ID to deallocate
- Intended receiver of deallocated IPC

Ideally Encoded in Registers
- Parameters in registers whenever possible
- Make frequent/simple operations simple and fast

![Sender Registers vs Receiver Registers](image)

Call-reply example
- Thread A
  - pre
  - IPC call
  - post
- Thread B
  - pre
  - IPC reply & wait
  - post
  - IPC reply & wait

Send and Receive Encoding
- 0 (Nil ID) is a reserved thread ID
- Define -1 as a wildcard thread ID

![Sender Registers](image)

Why use a single call instead of many?
- The implementation of the individual send and receive is very similar to the combined send and receive
- We can use the same code
  - We reduce cache footprint of the code
  - We make applications more likely to be in cache

To Encode for IPC
- Send to
- Receive from
- Call
- Send to & Receive
- Send to, Receive from
- Destination thread ID
- Source thread ID
- Send registers
- Receiver registers
- Number of send strings
- Send string start for each string
- Send string size for each string
- Number of receive strings
- Receiver string start for each string
- Receiver string size for each string
- Number of map pages
- Page range for each map page
- Receiver window for mappings
- IPC result code
- Send timeout
- Receiver timeout
- Send 0 timeout
- Receiver 0 timeout
- Receiver from thread ID
- Specify deallocating IPC
- Thread ID to deallocate
- Intended receiver of deallocated IPC
Message Transfer

- Assume that 64 extra registers are available
  - Name them MR_0 ... MR_63 (message registers 0 ... 63)
  - All message registers are transferred during IPC

To Encode for IPC

- Number of map pages
- Page range for each map page
- Receive window for mappings
- IPC result code
- Send timeout
- Receive timeout
- Send timeout
- Receive timeout
- Receive from thread ID
- Specify deserializing IPC
- Thread ID to deserial as
- Intended receiver of deserial IPC

Message construction

- Messages are stored in registers (MR_0 ... MR_63)
- First register (MR_0) acts as message tag
- Subsequent registers contain:
  - Untyped words (u), and
  - Typed words (t)
    - (e.g., map item, string item)

Message construction

- Messages are stored in registers (MR_0 ... MR_63)
- First register (MR_0) acts as message tag
- Subsequent registers contain:
  - Untyped words (u), and
  - Typed words (t)
    - (e.g., map item, string item)

Map and Grant items

- Two words:
  - Send base
  - Fpage
- Lower bits of send base indicate map item
- Semantics will be explained during memory management lecture

Message construction

- Typed items occupy one or more words
- Three currently defined items:
  - Map item (3 words)
  - Grant item (2 words)
  - String item (2+ words)
- Typed items can have arbitrary order
String items
- Max size 4MB (per string)
- Compound strings supported
  - Allows scatter-gather
  - Incorporates cacheability hints
    - Reduce cache pollution for long copy operations

To Encode for IPC
- Send to
- Receiver from
- Receiver
- Call
- Send to & Receive
- Send to, Receiver from
- Destination thread ID
- Source thread ID
- Send registers
- Receiver registers
- Number of send strings
- Send string start for each string
- Send string size for each string
- Number of receive strings
- Receiver string start for each string
- Receiver string size for each string

Timeouts
- Send and receive timeouts are the important ones
  - Xfer timeouts only needed during string transfer
  - Store Xfer timeouts in predefined memory location

String Receive
- Assume that 34 extra registers are available
  - Name them BR_0 ... BR_33 (buffer registers 0 ... 33)
  - Buffer registers specify
    - Receive strings
      - Receive window for mappings
Receiving messages

- Receiver buffers are specified in registers (BR₀, ... BRₐ)
- First BR (BR₀) contains “Acceptor”
- May specify receive window (if not nil-page)
- May indicate presence of receive strings/buffers (if 8-bit set)

To Encode for IPC

- Send to
- Receive from
- Receive
- Call
- Send to & Receive
- Send to, Receive from
- Destination thread ID
- Source thread ID
- Send registers
- Receiver registers
- Number of send strings
- Send string start for each string
- Number of receive strings
- Receive string size for each string
- Number of map pages
- Page range for each map page
- Receive window for mapstrings
- IPC result code
- Send timeout
- Receive timeout
- Receive after timeout
- Receive from thread ID
- Specify deviating IPC
- Thread ID to deviate as
- Intended receiver of deviated IPC

IPC Result

- Error conditions are exceptional
  - Error
  - Bit in received message tag indicate error
- Fast check
- Exact error code store in predefined memory location
- No need to optimize for error handling

IPC errors flagged in MRₐ
- Senders thread ID stored in register

Destination Registers
- senders
- timeouts
- receive specifier

Receiver Registers
- from
**IPCP Redirection**
- Redirection/deceiving IPC flagged by bit in the message tag
  - Fast check
  - When redirection bit set
  - Thread ID to deceive as and intended receiver ID stored in predefined memory locations

**To Encode for IPC**
- Number of map pages
- Page range for each map page
- IPC result code
- IPC result mappings
- Send timeout
- Receive timeout
- Send like timeout
- Receive like timeout
- Send from thread ID
- Receive from thread ID
- Specify deceiving IPC
- Thread ID to deceive as
- Intended receiver of deceived IPC

**Virtual Registers**
- What about message and buffer registers?
  - Most architecture have 64+ SP
- What about predefined memory locations?
  - Must be thread local

**What are Virtual Registers?**
- Virtual registers are backed by either
  - Physical registers, or
  - Non-pageable memory
- UTCBs hold the memory backed registers
  - UTCBs are thread local
  - UTCB can not be paged
  - No page faults
  - Registers always accessible

**Other Virtual Register Motivation**
- Portability
  - Common IPC API on different architectures
- Performance
  - Historically register only IPC was fast but limited to 2-3 registers on IA-32, memory based IPC was significantly slower but of arbitrary size
  - Needed something in between

**Switching UTCBs (IA-32)**
- Locating UTCB must be fast
  (avoid using system call)
- Use separate segment for UTCB pointer
  - `mov %gs:0, %edi`
- Switch pointer on context switches
Switching UTCBs (IA-32)

- Locating UTCB must be fast (avoid using system call)
- Use separate segment for UTCB pointer
  
  ```
  mov %gs:0, %edi
  ```
- Switch pointer on context switches

Message Registers and UTCB

- Some MRs are mapped to physical registers
- Kernel will need UTCB pointer anyway – pass it

Free Up Registers for Temporary Values

- Kernel need registers for temporary values
- MR0 and MR1 are the only registers that the kernel may not need

IPC Register Encoding

- Parameters in registers whenever possible
- Make frequent/simple operations simple and fast