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?

A "thread of control" has

- register set
  - e.g. general registers, IP and SP
  - stack
  - status
    - e.g. FLAGS, privilege,
    - OS-specific states (prio, time...)
  - address space
  - unique id
  - communication status

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

Thread Switch A → B

user mode A

user mode A

In Summary:

- Thread A is running in user mode.
- Thread A has experienced an end-of-time-slice or is preempted by an interrupt.
- We enter kernel mode.
- The microkernel has to save the status of the thread A on A's TCB.
- The next step is to load the status of thread B from B's TCB.
- Leave kernel mode and thread B is running in user mode.
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.

### Per-Thread Kernel Stack

#### Processes Model

```c
example(arg1, arg2) {
    P1(arg1, arg2);
    if (need_to_block) {
        thread_block();
    } else {
        P2(arg2);
    }
    /* return control to user */
    return SUCCESS;
}
```
Single Kernel Stack
“Event” or “Interrupt” Model

- How do we use a single kernel stack to support many threads?
- Issue: How are system calls that block handled?
  - either continuations
    - Using Continuations to Implement Thread Management and Communication in Operating Systems. [Draves et al., 1991]
  - or stateless kernel (interrupt model)
    - Interface and Execution Models in the Fluke Kernel. [Ford et al., 1999]

Continuations

- State required to resume a blocked thread is explicitly saved in a TCB
  - A function pointer + variables
  - Stack can be discarded and reused to support new thread
  - Resuming involves discarding current stack, restoring the continuation, and continuing

Example

```c
example(arg1, arg2)
{ 
  P1(arg1, arg2);
  if (need_to_block) {
    save_context_in_TCB;
    thread_block(example_continue);
    /* NOT REACHED */
  } else {
    P2();
    thread_syscall_return(SUCCESS);
  }
}
```

```c
example_continue()
{ 
  recover_context_from_TCB;
  P2(recovered arg2);
  thread_syscall_return(SUCCESS);
}
```

Stateless Kernel

- System calls can not block within the kernel
  - If syscall must block (resource unavailable)
    - Modify user-state such that syscall is restarted when resources become available
    - Stack content is discarded
  - Preemption within kernel difficult to achieve.
    - Must (partially) roll syscall back to (a) restart point
  - Avoid page faults within kernel code
    - Syscall arguments in registers
      - Page fault during roll-back to restart (due to a page fault) is fatal.

IPC examples – Per thread stack

```c
msg_send_rcv(msg, option, send_size, rcv_size, ...)
{ 
  rc = msg_send(msg, option, send_size, ...);
  if (rc != SUCCESS)
    return rc;
  rc = msg_rcv(msg, option, rcv_size, ...);
  return rc;
}
```

Block inside msg_rcv if no message available

- Send and Receive system call implemented by a non-blocking send part and a blocking receive part.

IPC examples – Continuations

```c
msg_send_rcv(msg, option, send_size, rcv_size, ...)
{ 
  rc = msg_send(msg, option, send_size, ...);
  if (rc != SUCCESS)
    return rc;
  cur_thread->continuation.msg = msg;
  cur_thread->continuation.option = option;
  cur_thread->continuation.rcv_size = rcv_size;
  ...
  rc = msg_rcv(msg, option, rcv_size, ..., msg_rcv_continue);
  return rc;
}
```

```c
msg_rcv_continue(cur_thread)
{ 
  msg = cur_thread->continuation.msg;
  option = cur_thread->continuation.option;
  rcv_size = cur_thread->continuation.rcv_size;
  ...
  rc = msg_rcv(msg, option, rcv_size, ..., msg_rcv_continue);
  return rc;
}
```

IPC Examples – stateless kernel

```c
msg_send_rcv(cur_thread)
{ 
  rc = msg_send(cur_thread);
  if (rc != SUCCESS)
    return rc;
  set_pc(cur_thread, msg_send_entry);
  return SUCCESS;
}
```

Set user-level PC to restart msg_rcv only
Single Kernel Stack
per Processor, event model

- either continuations
  - complex to program
  - must be conservative in state saved (any state that might be needed)
  - Mach (Draves), L4Ka::Strawberry

- or stateless kernel
  - no kernel threads, kernel not interruptible, difficult to program
  - request all potentially required resources prior to execution
  - blocking syscalls must always be re-startable
  - Processor-provided stack management can get in the way
  - 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
- difficult to exchange kernel on-the-fly

**Conclusion:**
We have to look for a solution that minimizes the kernel stack size!

**Conclusion:**
Either no persistent tcbs or tcbs must hold virtual addresses

---

**CPU**

**enter kernel (IA32)**

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

**enter kernel (IA32)**

- kernel code
- esp
- ebp
- ebx
- ecx
- eax
- eflags

**User stack**
- tcb A
- user esp
- user stack
- points to the running threads kernel stack

**CPU**

**enter kernel (IA32)**

- trap / fault occurs (INT n / exception / interrupt)
- push user esp on to kernel stack, load kernel esp

**CPU**

**enter kernel (IA32)**

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

**Hardware programmed, single instruction**
enter kernel (IA32)

CPU

kernel mode

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

Hardware programmed, single instruction

user stack

enter kernel (IA32)

CPU

kernel mode

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

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
    - Systemcall
    - EIP = ECX
    - ESP = EDX
    - Eflags.S = 3
  - Sysexit
    - ESP = ECX
    - EIP = EDX
    - Eflags.I = 0, FLAGS.S = 0

Sysenter/sysexit

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

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 are in kernel mode.
Kernel-stack state
Uniprocessor:
- Any kstack ≠ myself is current!
- X permits to differentiate between stack layouts:
  - interrupt, exception, some system calls
  - ipc
  - V86 mode

Remember:
- We need to find
  - any thread’s tcb starting from its uid
  - the currently executing thread’s tcb

To find the starting address from the tcb:
- mov esp, [ebp].thr_esp
- mov [edi].thr_esp, esp
- mov esp, eax
- and -sizeof tcb, eax
- add sizeof tcb, eax
- mov eax, [esp0_ptr]
- popa
- add $4, esp
- iret

Switch threads (IA32)
System call (Sysenter/Sysexit)

- Emulate int instruction (ECX=USP, EDX=UIP)
  - `mov esp0, esp`  
  - `sub $20, esp`  
  - `mov ecx, 16(esp)`  
  - `mov edx, 4(esp)`  
  - `mov $5, (esp)`  

- Emulate iret instruction
  - `mov 16(esp), ecx`  
  - `mov 4(esp), edx`  
  - `sti`  
  - `sysexit`

- Emulate int instruction
  - `mov (esp), esp`  

- Trick: MSR points to esp0

- Set esp0 to new kernel stack

Switch threads (IA32)

- int 0x32, push registers of the green thread
- switch kernel stacks (store and load esp)
- set esp0 to new kernel stack
- pop orange registers, return to new user thread

Sysenter/Sysexit

- Emulate int instruction (ECX=USP, EDX=UIP)
  - `mov esp0, esp`
  - `sub $20, esp`
  - `mov ecx, 16(esp)`
  - `mov edx, 4(esp)`
  - `mov $5, (esp)`

- Emulate iret instruction
  - `mov 16(esp), ecx`
  - `mov 4(esp), edx`
  - `sti`
  - `sysexit`

- Emulate int instruction
  - `mov (esp), esp`

- Trick: MSR points to esp0

- Set esp0 to new kernel stack

- Pop orange registers, return to new user thread

Switch threads (IA32)

- int 0x32, push registers of the green thread
- switch kernel stacks (store and load esp)
- set esp0 to new kernel stack
- pop orange registers, return to new user thread

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 Exe 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

TCB structure

- Thread Id
- MyselfGlobal
- MyselfLocal
- State
- Resources
- KernelStackPtr
- Scheduling
  - ReadyList
  - TimeSliceLength
  - RemainingTimeSlice
  - TotalQuartum
  - Priority
  - WakeupList
- Space
- PDirCache
- Stack()

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 thread_id, %eax
  - mov %eax, %ebx
  - and mask thread_no, %eax
  - mov tcb_pointer_array[%eax*4], %eax
  - cmp OFS_TCB_MYSELF(%eax), %ebx
  - 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, OFS_TCB_MYSELF(%eax)
  - jnz invalid_thread_id

Thread ID translation

- Via table
  - no MMU
  - table access per TCB
  - TLB entry for table
  - TCB pointer array requires 1M virtual memory for 256K potential threads

- Via MMU
  - MMU
  - no table access
  - TLB entry per TCB
  - virtual resource TCB array required, 256K potential threads need 128M virtual space for TCBs

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
TLB working set grows: 4 entries to cover 4000 threads.
Nevertheless much better than 1 TLB for 8 threads like in direct address.

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

AS Layout

32bits, virt tcb, entire PM

- user regions
- shared system regions
  - system
  - other kernel tables
  - physical memory
  - kernel code
  - tcb
Limitations
- 32bits, virt tcb, entire PM
- number of threads
- physical mem size

Physical Memory
- Kernel uses physical for:
  - Application’s Page tables
  - Kernel memory
  - Kernel debugger
- Issue occurs only when kernel accesses physical memory
  - Limit valid physical range to remap size (256M)
  - Or...

Physical-to-virtual Pagetable
- Dynamically remap kernel-needed pages
- Walk physical-to-virtual ptab before accessing
- Costs???
  - Cache
  - TLB
  - Runtime

Kernel Debugger (not performance critical)
- Walk page table in software
- Remap on demand (4MB)
- Optimization: check if already mapped

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 $\neq$ current
      - Save current state to fpu_owner
      - Load new state from current
      - fpu_owner $\leftarrow$ current

Physical Memory Diagram
- Map and unmap
- Copy IPC
- Page tables
- TCBs
- XDB output
- Mem Dump

Physical-to-virtual Pagetable Diagram
- Walk page table in software
- Remap on demand (4MB)
- Optimization: check if already mapped

FPU Context Switching Diagram
- 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 Diagram
- Lock FPU on thread switch
- Unlock at first use – exception handled by kernel
  - Unlock FPU
    - If fpu_owner $\neq$ current
      - Save current state to fpu_owner
      - Load new state from current
      - fpu_owner $\leftarrow$ current
**IPC**

**Functionality & Interface**

**What IPC primitives do we need to communicate?**

- **Send to** (a specified thread)
- **Receive from** (a specified thread)
- **Receive** (from any 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.

**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 message types are appropriate?**

- **Register**
  - Short messages we hope to make fast by avoiding memory access to transfer the message during IPC
  - Guaranteed to avoid user-level page faults during IPC

- **Direct string (optional)**
  - In-memory message we construct to send

- **Indirect string (optional)**
  - In-memory messages built in chunks

- **Map pages (optional)**
  - Messages that map pages from sender to receiver

**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).**

**Version 4, Version X.2**
Problem
- How to deal with threads that are:
  - Uncooperative
  - Malfunctioning
  - Malicious
- That might result in an IPC operation never completing?

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
- Assume page fault timeout granularity can be much less than send/receive granularity

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

Timeout Range of Values (seconds) [V 2, V X.0]

<table>
<thead>
<tr>
<th>m</th>
<th>*10^2</th>
<th>*10^6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>\infty</td>
<td>\infty</td>
</tr>
<tr>
<td>1</td>
<td>268,435,459</td>
<td>6,845,041</td>
</tr>
<tr>
<td>2</td>
<td>87,108,841</td>
<td>2,517,120</td>
</tr>
<tr>
<td>3</td>
<td>16,777,218</td>
<td>427,190</td>
</tr>
<tr>
<td>4</td>
<td>419,430</td>
<td>106,575</td>
</tr>
<tr>
<td>5</td>
<td>1,048,576</td>
<td>26,738</td>
</tr>
<tr>
<td>6</td>
<td>2,621,444</td>
<td>6,846</td>
</tr>
<tr>
<td>7</td>
<td>0,062,539</td>
<td>0,167</td>
</tr>
<tr>
<td>8</td>
<td>0,016,384</td>
<td>0,042</td>
</tr>
<tr>
<td>9</td>
<td>0,004,096</td>
<td>0,011</td>
</tr>
<tr>
<td>10</td>
<td>0,001,024</td>
<td>0,004</td>
</tr>
<tr>
<td>11</td>
<td>0,000,256</td>
<td>0,001</td>
</tr>
<tr>
<td>12</td>
<td>0,000,064</td>
<td>1e-05</td>
</tr>
<tr>
<td>13</td>
<td>0,000,016</td>
<td>0,000</td>
</tr>
<tr>
<td>14</td>
<td>0,000,004</td>
<td>0,000</td>
</tr>
<tr>
<td>15</td>
<td>0,000,001</td>
<td>0,000</td>
</tr>
</tbody>
</table>

1µs ~ 255µs with 1µs granularity

Up to 19h with ~4.4min granularity

IPC - API

- Timeouts (V 2, V X.4)
  - snd timeout, rcv timeout
  - snd-pf / rcv-pf timeout

- timeout values
  - 0
  - infinite
  - 1us ... 19 h (log)
- Compact 32-bit encoding

Timeout Problem

- Worst case IPC transfer time is high given a reasonable single page-fault timeout
  - Potential worst-case is a page fault per memory access
  - IPC time = Send timeout + n x page fault timeout
- Worst-case for a careless implementation is unbound
  - If pager can respond with null mapping that does not resolve the fault

IPC - API

- Timeouts (V X.2, V 4)
  - snd timeout, rcv timeout, xfer timeout snd, xfer timeout rcv

- time
  - wait for send
  - send message (xfer)
  - wait for reply
  - receive message (xfer)
  - min (xfer to, xfer to)
  - rcv to
IPCs - API

Timeouts (V 4, V X.2)
- snd timeout, rcv timeout, xfer timeout snd, xfer timeout rcv

Relative timeout values
- 0
- infinite
- 1µs ... 610h (log)

Absolute timeout values
- 0
- infinite
- 1µs ... 610h (log)

Timeout Range of Values (seconds) [V 4, V X.2]

<table>
<thead>
<tr>
<th>n</th>
<th>em</th>
<th>m+10e12</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0001001</td>
<td>0.0001001</td>
</tr>
<tr>
<td>1</td>
<td>0.0001002</td>
<td>0.0001002</td>
</tr>
<tr>
<td>2</td>
<td>0.0001003</td>
<td>0.0001003</td>
</tr>
<tr>
<td>3</td>
<td>0.0001004</td>
<td>0.0001004</td>
</tr>
<tr>
<td>4</td>
<td>0.0001005</td>
<td>0.0001005</td>
</tr>
<tr>
<td>5</td>
<td>0.0001006</td>
<td>0.0001006</td>
</tr>
<tr>
<td>10</td>
<td>0.0001010</td>
<td>0.0001010</td>
</tr>
<tr>
<td>15</td>
<td>0.0001015</td>
<td>0.0001015</td>
</tr>
<tr>
<td>20</td>
<td>0.0001020</td>
<td>0.0001020</td>
</tr>
<tr>
<td>25</td>
<td>0.0001025</td>
<td>0.0001025</td>
</tr>
<tr>
<td>50</td>
<td>0.0001050</td>
<td>0.0001050</td>
</tr>
<tr>
<td>100</td>
<td>0.0001100</td>
<td>0.0001100</td>
</tr>
<tr>
<td>200</td>
<td>0.0001200</td>
<td>0.0001200</td>
</tr>
<tr>
<td>400</td>
<td>0.0002400</td>
<td>0.0002400</td>
</tr>
<tr>
<td>800</td>
<td>0.0004800</td>
<td>0.0004800</td>
</tr>
<tr>
<td>1600</td>
<td>0.0009600</td>
<td>0.0009600</td>
</tr>
<tr>
<td>3200</td>
<td>0.0019200</td>
<td>0.0019200</td>
</tr>
<tr>
<td>6400</td>
<td>0.0038400</td>
<td>0.0038400</td>
</tr>
<tr>
<td>12800</td>
<td>0.0076800</td>
<td>0.0076800</td>
</tr>
<tr>
<td>25600</td>
<td>0.0153600</td>
<td>0.0153600</td>
</tr>
<tr>
<td>51200</td>
<td>0.0307200</td>
<td>0.0307200</td>
</tr>
<tr>
<td>102400</td>
<td>0.0614400</td>
<td>0.0614400</td>
</tr>
<tr>
<td>204800</td>
<td>0.1228800</td>
<td>0.1228800</td>
</tr>
<tr>
<td>409600</td>
<td>0.2457600</td>
<td>0.2457600</td>
</tr>
<tr>
<td>819200</td>
<td>0.4915200</td>
<td>0.4915200</td>
</tr>
<tr>
<td>1638400</td>
<td>0.9830400</td>
<td>0.9830400</td>
</tr>
<tr>
<td>3276800</td>
<td>1.9660800</td>
<td>1.9660800</td>
</tr>
<tr>
<td>6553600</td>
<td>3.9321600</td>
<td>3.9321600</td>
</tr>
<tr>
<td>13107200</td>
<td>7.8643200</td>
<td>7.8643200</td>
</tr>
</tbody>
</table>

To Encode for IPC
- Send to
- Receive from
- Call
- Send to & Receive
- Send to, Receive from
- Destination thread ID
- Source thread ID
- Send registers
- Receive registers
- Number of send strings
- Send string size for each string
- Number of receive strings
- Receive string size for each string
- Receive from thread ID
- Specify deceiting IPC
- Send timeout
- Receive timeout
- Send Xfer timeout
- Receive Xfer timeout
- Receive window for mappings
- IPC result code
- Specify deceiting IPC
- Thread ID to deceit as
- Intended receiver of deceited IPC

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

Call-reply example
- Thread A
- Send pre
- IPC call
- post
- IPC reply & wait
- Thread B
- Send pre
- IPC reply & wait
- post
Send and Receive Encoding

- 0 (Nil ID) is a reserved thread ID
- Define -1 as a wildcard thread ID

<table>
<thead>
<tr>
<th>Register</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>destination</td>
</tr>
<tr>
<td>EDX</td>
<td>receive specifier</td>
</tr>
</tbody>
</table>

- Nil ID means no send operation
- Nil ID means no receive operation
- Wildcard means receive from any thread

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
- Receive
- Call
- Send to & Receive
- Send to, Receive from
- Destination thread ID
- Source thread ID
- Send registers
- Receive registers
- Number of send strings
- Send string start for each string
- Send string size for each string
- Number of receive strings
- Receive string start for each string
- Receive string size for each string
- Number of map pages
- Page range for each map page
- Receive window for mappings
- IPC result code
- Send timeout
- Receive timeout
- Send Xfer timeout
- Receive Xfer timeout
- Receive from thread ID
- Specify deceiting IPC
- Thread ID to deceit as
- Intended receiver of deceived IPC

Message Transfer

- Assume that 64 extra registers are available
  - Name them MR0 … MR63 (message registers 0 … 63)
- All message registers are transferred during IPC

Message construction

- Messages are stored in registers (MR0 … MR63)
- First register (MR0) acts as message tag
- Subsequent registers contain:
  - Untyped words (u), and
  - Typed words (t)
    - (e.g., map item, string item)
  - Freely available (e.g., request type)
Message construction

- Messages are stored in registers (MR0, ..., MR8)
- First register (MR0) acts as message tag
- Subsequent registers contain:
  - Untyped words (u), and
  - Typed words (t)
  (e.g., map item, string item)

Typed items occupy one or more words

- Three currently defined items:
  - Map item (2 words)
  - Grant item (2 words)
  - String item (2+ words)
- Typed items can have arbitrary order

Message

Map and Grant items

- Two words:
  - Send base
  - Fpage
- Lower bits of send base indicates map or grant item

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
- Receive from
- Call
- Send to & Receive
- Source Thread ID
- Destination Thread ID
- Send registers
- Receive registers
- Number of send strings
- Send string start for each string
- Send string size for each string
- Number of receive strings
- Receive string start for each string
- Receive string size for each string

Number of map pages
- Page range for each map page
- Receive window for mappings
- IPC result code
- Send timeout
- Receive timeout
- Send Xfer timeout
- Receive Xfer timeout
- Specify deceited IPC
- Thread ID to deceit as
- Intended receiver of deceited IPC

"hh" indicates cacheability hints for the string
Timeouts
- Send and receive timeouts are the important ones
- Xfer timeouts only needed during string transfer
- Store Xfer timeouts in predefined memory location

Timeouts values are only 16 bits
- Store send and receive timeout in single register

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
- Receive registers
- Number of send strings
- Send string start for each string
- Send string size for each string
- Number of receive strings
- Receive string start for each string
- Receive string size for each string
- Number of map pages
- Page range for each map page
- Receive window for mappings
- IPC result code
- Send timeout
- Receive timeout
- Send Xfer timeout
- Receive Xfer timeout
- Receive from thread ID
- Specify deceiting IPC
- Thread ID to deceit as
- Intended receiver of deceited IPC

String Receive
- Assume that 34 extra registers are available
  - Name them BR0 ... BR33 (buffer registers 0 ... 33)
  - Buffer registers specify
    - Receive strings
    - Receive window for mappings

Receiving messages
- Receiver buffers are specified in registers (BR0 ... BR33)
- First BR (BR0) contains “Acceptor”
  - May specify receive window (if not nil-fpage)
  - May indicate presence of receive strings/buffers
    (if s-bit set)
  - String length
  - String pointer
  - String length
  - String pointer
  - 0
  - 0 0
  - 0hhC
  - 0hhC
  - BR1
  - BR4
  - 0hh1
  - BR5
  - BR4+j
- A receive buffer can of course be a compound string
- The s-bit set indicates presence of string items acting as receive buffers
- If C-bit in string item is cleared, it indicates that no more receive buffers are present

Receiving messages
- If C-bit in string item is cleared, it indicates that no more receive buffers are present
- A receive buffer can of course be a compound string
- The s-bit set indicates presence of string items acting as receive buffers

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
- Receive registers
- Number of send strings
- Send string start for each string
- Send string size for each string
- Number of receive strings
- Receive string start for each string
- Receive string size for each string
- Number of map pages
- Page range for each map page
- Receive window for mappings
- IPC result code
- Send timeout
- Receive timeout
- Send Xfer timeout
- Receive Xfer timeout
- Receive from thread ID
- Specify deceiting IPC
- Thread ID to deceit as
- Intended receiver of deceited IPC
IPC Result

- Error conditions are exceptional
  - I.e., not common case
  - No need to optimize for error handling
- Bit in received message tag indicate error
- Fast check
- Exact error code store in predefined memory location

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 Xfer timeout
- Receive Xfer timeout
- Specify deceiting IPC
- Intended receiver of deceited IPC

IPC Redirection

- Redirection/deceiting IPC flagged by bit in the message tag
- Fast check
- When redirection bit set
- Thread ID to deceit as and intended receiver ID stored in predefined memory locations

Virtual Registers

- What about message and buffer registers?
  - Most architectures do not have 64+34 spare registers
- 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
- Registers always accessible

Preserved by kernel during context switch

- Physical Registers
- UTCB

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
- MR1 and MR2 are the only registers that the kernel may not need
Free Up Registers for Temporary Values

- `Sysexit` instruction requires:
  - ECX = user IP
  - EDX = user SP

<table>
<thead>
<tr>
<th>Sender Registers</th>
<th>Receiver Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>EAX</strong></td>
<td>destination</td>
</tr>
<tr>
<td><strong>ECX</strong></td>
<td>timeouts</td>
</tr>
<tr>
<td><strong>EDX</strong></td>
<td>receive specifier</td>
</tr>
<tr>
<td><strong>EBX</strong></td>
<td>-</td>
</tr>
<tr>
<td><strong>EBP</strong></td>
<td>-</td>
</tr>
<tr>
<td><strong>ESI</strong></td>
<td>MR0</td>
</tr>
<tr>
<td><strong>EDI</strong></td>
<td>MR1</td>
</tr>
<tr>
<td><strong>ESP</strong></td>
<td>UTCB</td>
</tr>
</tbody>
</table>

IPC Register Encoding

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

<table>
<thead>
<tr>
<th>Sender Registers</th>
<th>Receiver Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>EAX</strong></td>
<td>destination</td>
</tr>
<tr>
<td><strong>ECX</strong></td>
<td>timeouts</td>
</tr>
<tr>
<td><strong>EDX</strong></td>
<td>receive specifier</td>
</tr>
<tr>
<td><strong>EBX</strong></td>
<td>-</td>
</tr>
<tr>
<td><strong>EBP</strong></td>
<td>-</td>
</tr>
<tr>
<td><strong>ESI</strong></td>
<td>MR0</td>
</tr>
<tr>
<td><strong>EDI</strong></td>
<td>MR1</td>
</tr>
<tr>
<td><strong>ESP</strong></td>
<td>UTCB</td>
</tr>
</tbody>
</table>