μ-Kernel 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

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 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 Switch A ➔ B

user mode A

IP
SP
FLAGS
tcb A

IP
SP
FLAGS
tcb B

Thread Switch A ➔ B

user mode A

IP
SP
FLAGS
tcb A

IP
SP
FLAGS
tcb B
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.

- A thread's kernel state is implicitly encoded in the kernel activation stack.
- If the thread must block in-kernel, we can simply switch from the current stack to another thread's stack until the thread is resumed.
- Resuming is simply switching back to the original stack.
- Preemption is easy.
- No conceptual difference between kernel mode and user mode.
Singleg Kernel Stack 

- 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]

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 - 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_cont)
    if (rc != SUCCESS) return rc;
    msg_rcv_cont(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_cont) 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_rcv_entry);
    rc = msg_rcv(cur_thread);
    if (rc != SUCCESS) return rc;
    return SUCCESS;
}
```

Continuations

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

```c
def example(argc, argv) {
    P1(argc, argv):
    if (need_to_block) {
        thread_block(example_cont);
    } else {
        P2();
        thread_syscall_return(SUCCESS);
    }
    /* NOT REACHED */
}
```

```c
example_cont() {
    recover_context_from_TCB;
    P2(recovered argv);
    thread_syscall_return(SUCCESS);
}
```

IPC examples – Per thread stack

```c
msg_send_rcv(msg, option, send_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_cont) {
        msg = cur_thread->continuation.msg;
        option = cur_thread->continuation.option;
        rcv_size = cur_thread->continuation.rcv_size;
        ... return rc;
    }
}
```

Block inside msg_rcv if no message available
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 (Dravek), L4:Strawberry, NICTA Pistachio

- or stateless kernel
  - no kernel threads, kernel not interruptible, difficult to program
  - request of potentially required resources prior to execution
  - blocking syscall must always be re-sizable
  - processor-provided stack management can get in the way
  - system calls need to be kept simple “atomic”
  - kernel can be exchanged on-the-fly
  - e.g. the Net 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 tcb or tcb must hold
Virtual addresses

CPU

Single Instruction

User Stack

Kernel Stack

Enter Kernel (IA32)

User Mode

- trap / fault occurs (INT n / exception / interrupt)

Kernel Mode

- push user esp on to kernel stack, load kernel esp

- push user eflags, reset flags (I=0, S=0)

- 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)**

- 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

**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 (ECX, 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.)
Kernel-stack state

Uniprocessor:
- Any kernelstack ≠ myself is current!
- X permits to differentiate between stack layouts:
  - interrupt, exception, some system calls
  - ipc
  - V86 mode

Thread switch (IA32)

Thread A

switch esp down to kernel stack

int 32

Thread B

Switch current kernel stack pointer

push X
push esp
mov esp, [esp+4]

Add sizeof tcb, esp

push eax
pop esp

Switch threads (IA32)
**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 int instruction
  - mov 16(esp), ecx
  - mov 4(esp), edx
  - stk
  - sysexit

**Mips R4600**

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

**Register**:

- `eax`
- `edx`
- `esi`
- `edi`
- `esp`
- `ebp`
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

Enter kernel:

(Mips)

```
move k1, C0_status
and k1, C0_status
jal sp, kernel_stack_bottom()

ifnz k1
    jal other_exception

move k0, kernel_base
move k1, k0
sub k1, 0
or k1, k1, 1
or k1, k1, 1
or k1, k1, 1

jal switch

add sp, sp, 24
```
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
- Via MMU
  - MMU
  - no table access
  - TLB entry per TCB
- **TCB pointer array** requires 1M virtual memory for 256K potential threads
- 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 maps 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**
- 32 bits, virt tcb, entire PM
- User regions
- Shared system regions
- Per-space system regions
- Other kernel tables
- Physical memory
- Kernel code
- TCBs

**Limitations**
- 32 bits, virt tcb, entire PM
- Number of threads
- Physical mem size
- IA-32 External Memory
- L-4KA::Pistachio/a32
- 262,144
- Nearly even doubling PC
- Has more than 2GB

**Physical Memory**
- Kernel uses physical for:
  - Application’s Page tables
  - Kernel memory
  - Kernel debugger
- Page tables
- TCBs
- KDEx output
- Mem Dump
- 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 points to current
    - Set fpu_owner to current
    - Load new state from current

**IPC**

**Functionality & Interface**

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

- Send to
  - (a specified thread)
- Receive from
  - (a specified thread)
- Send to & Receive
  - (send to, receive from and any thread)
- Send to, Receive from
  - (send to, receive from specified different threads)

- 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)

- Are other combinations appropriate?

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

- Send to
  - (a specified thread)
- Receive from
  - (a specified thread)
- Send to & Receive
  - (send to, 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.

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

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

**Atomic operation to ensure that server's (caller'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).**

**Classic client vs server send and receive diagram.**

- Classic client send and receive diagram.
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 strings**
  - String message we construct to send

- **Indirect strings (optional)**
  - In-memory messages sent in place

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

What message types are appropriate? [Version 4, Version X.2]

- **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

- **Strings (optional)**
  - String message we construct to send

- **Indirect strings (optional)**
  - In-memory 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, v3, v4)**
  - snd timeout, rcv timeout

IPC - API

- **Timeouts (v2, v3, v4)**
  - snd timeout, rcv timeout
    - snd-pf timeout
    - specified by sender

Attack through receiver’s pager:
IPC - API

- Timeouts [v2, v3, v4]
  - snd timeout, rcv timeout
  - snd-pf / rcv-pf timeout
    - specified by receiver

- Attack through sender’s pager:

   Pager

   PF

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

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

Timeout Range of Values (seconds) [v2, v3, v4]

<table>
<thead>
<tr>
<th>e</th>
<th>m = 1</th>
<th>m = 255</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>268,435,046</td>
<td>68451,041,298</td>
</tr>
<tr>
<td>1</td>
<td>67,108,864</td>
<td>17,112,760,332</td>
</tr>
<tr>
<td>2</td>
<td>16,177,276</td>
<td>4,221,270,508</td>
</tr>
<tr>
<td>4</td>
<td>4,104,304</td>
<td>1,069,547,652</td>
</tr>
<tr>
<td>8</td>
<td>1,046,876</td>
<td>267,386,888</td>
</tr>
<tr>
<td>16</td>
<td>256,2144</td>
<td>66,946,72</td>
</tr>
<tr>
<td>32</td>
<td>65,536</td>
<td>16,711,68</td>
</tr>
<tr>
<td>64</td>
<td>0.016384</td>
<td>0.041732</td>
</tr>
<tr>
<td>128</td>
<td>0.004096</td>
<td>0.096448</td>
</tr>
<tr>
<td>256</td>
<td>0.001024</td>
<td>0.025112</td>
</tr>
<tr>
<td>512</td>
<td>0.000256</td>
<td>0.06328</td>
</tr>
<tr>
<td>1024</td>
<td>0.0000516</td>
<td>0.01632</td>
</tr>
<tr>
<td>2048</td>
<td>0.000004</td>
<td>0.000403</td>
</tr>
<tr>
<td>4096</td>
<td>0.000001</td>
<td>0.000025</td>
</tr>
</tbody>
</table>

- Page fault timeout has no mantissa

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

- Compact 32-bit encoding

IPC - API

- Timeouts [v2, v3, v4]
  - 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 + time × page fault timeout
- Worst-case for a careless implementation is unbound
  - If paper can respond with null mapping that does not resolve the fault

**IPC - API**

- Timeouts (VX.2, V4)
  - snd timeout, rcv timeout, xfer timeout snd, xfer timeout rcv
- relative timeout values
  - 0
  - infinite
  - 1us ... 610 h (log)

**Timeout Range of Values (seconds) [V4, VX.2]**

<table>
<thead>
<tr>
<th>Value</th>
<th>Send</th>
<th>Receive</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.000001</td>
<td>0.001024</td>
</tr>
<tr>
<td>1</td>
<td>0.000002</td>
<td>0.002048</td>
</tr>
<tr>
<td>2</td>
<td>0.000004</td>
<td>0.004096</td>
</tr>
<tr>
<td>3</td>
<td>0.000008</td>
<td>0.008192</td>
</tr>
<tr>
<td>4</td>
<td>0.000016</td>
<td>0.016384</td>
</tr>
<tr>
<td>5</td>
<td>0.000032</td>
<td>0.032768</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>31</td>
<td>2.097152</td>
<td>2.1439852</td>
</tr>
<tr>
<td>32</td>
<td>4.194305</td>
<td>4.2879704</td>
</tr>
<tr>
<td>64</td>
<td>33.554432</td>
<td>33.928864</td>
</tr>
<tr>
<td>128</td>
<td>343.6683</td>
<td>349.2680</td>
</tr>
<tr>
<td>256</td>
<td>3492.1854</td>
<td>3549.3757</td>
</tr>
<tr>
<td>319</td>
<td>3549.3757</td>
<td>3606.7197</td>
</tr>
<tr>
<td>320</td>
<td>3606.7197</td>
<td>3664.0632</td>
</tr>
</tbody>
</table>

- Up to ~610h with 1us granularity
- Up to ~35min granularity

**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
  - Receive from thread ID
- Specify destination IPC
- Destination thread ID
- Source thread ID
- Send registers
  - Number of send strings
  - Send string size for each string
  - Send string size for each string
  - Number of receive strings
  - Receive string size for each string
  - Receive string size for each string
  - Intended receiver of desected IPC
Ideally Encoded in Registers
- Parameters in registers whenever possible
- Make frequent/simple operations simple and fast

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

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
  - Send from
  - Receiver
  - 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
  - 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 after timeout
  - Receive after timeout
  - Receive from thread ID
  - Specify devolving IPC
  - Thread ID to devolve as
  - Extended receiver of devolved 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
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 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
- Receiver timeout
- Send Xfer timeout
- Receive Xfer timeout
- Receive from thread ID
- Specify denoting IPC
- Thread ID to decode as
- Intended receiver of described IPC

Message construction
- Messages are stored in registers (MRi ... MRi+k)
- First register (MRi) acts as message tag
- Subsequent registers contain:
  - Untyped words (u), and
  - Typed words (t)
    - (e.g., map item, string item)
- Various IPC flags
- Number of send/recv registers
- Number of send/recv string
- String register size
- String register alignment

Map and Grant items
- Two words:
  - Send base
  - Fpage
- Lower bits of send base

String items
- Max size 4MB (per string)
- Compound strings supported
- Allows scatter-gather
- Incorporates cachability hints
- Reduce cache pollution for long copy operations

Semantics will be explained during memory management lecture
String items

- New string specifier may of course contain substrings
- Different size compound strings require a new string specifier
- All substrings are of same size

"hi" indicates cacheability hints for the string.

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
- 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
- IPC Xfer timeout
- Send window
- Receive window
- Thread ID to discard
- Extended receiver of dedicated IPC

Timeouts

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

String Receival

- 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 (BR4, ..., BR33)
  - First BR (BR4) contains
    - May specify receive window (if not nil override)
    - May indicate presence of receive strings/buffers
      (if s-bit set)
Receiving messages

- If 0-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.
- If 1-bit in string item is set, it indicates presence of more receive buffers.
- The 0-bit set indicates presence of string items acting as receive buffers.

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 timer timeout
- Receive timer timeout
- Send from thread ID
- Specify sending IPC
- Thread ID to deceive as
- Intended receiver of deceived IPC

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

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

IPC Result

- IPC errors flagged in MR
- Sends thread ID stored in register

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

IPC 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
- Send to
- Receive from
- Call
- Send to & Receive
- Send to, Receive from
- Examine 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
- Range for each map page
- Attribute flags for each map page
- IPC result code
- Send timeout
- Receive timeout
- Receive from thread ID
- Source from thread ID
- Specify deserializing IPC
- Thread ID to device as
- Extended receiver of deserializing IPC

Virtual Registers
- What about message and buffer registers?
  - Most architectures 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’t 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 needs 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

IPC Register Encoding

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