μ-Kernel Construction

A "thread of control" has
- **internal properties**
  - register set
  - e.g. general registers, IP and SP
  - stack
  - status
  - e.g. FLAGS, privilege,
  - OS-specific states (prio, time...)
- **external properties**
  - 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 (per processor)

Thread Switch A → B
- Processor
- tcb A
- user mode A
- tcb B

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?
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 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.

```
example(arg1, arg2) {
    P1(arg1, arg2);
    if (need_to_block) {
        thread_block();
        P2(arg2);
    } else {
        P3();
    }
    /* 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(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);
}
```

 IPC examples – Per thread stack

```
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, ...);
  if (rc != SUCCESS)
    return rc;
  return SUCCESS;
}
```

 IPC Examples – stateless kernel

```
msg_send_rcv(msg, option, send_size, rcv_size, ...) {
  rc = msg_send(msg, option, send_size, ...);
  if (rc != SUCCESS)
    return rc;
  set_pc(msg, msg_rcv_entry);
  rc = msg_rcv(msg, option, rcv_size, ...);
  if (rc != SUCCESS)
    return rc;
  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.
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 (Shaves), L4Ka:Strawberry, NICTA Pistachio
- 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:**
Either no persistent tcbs or tcbs must hold virtual addresses!
**System call (IA32)**

- **Enter kernel (IA32)**
  - Trap / fault occurs (\textit{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

- **Kernel-stack state**
  - Uniprocessor:
    - Any kstack ≠ myself is current!

- **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
    - \texttt{EIP = MSR(Kernel IP)}
    - \texttt{ESP = MSR(Kernel SP)}
    - Eflags.I = 0, FLAGS.S = 0
  - Sysexit
    - \texttt{ESP = ECX}
    - \texttt{EIP = EDX}
    - Flags undefined
  - Kernel has to re-enable interrupts

- **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
    - Sysexit

- **Emulate iret instruction**
  - \texttt{Mov 16(esp), ecx}
  - \texttt{Mov 4(esp), edx}
  - \texttt{Sti}
  - \texttt{Sysexit}

- **Kernel-stack state**
  - Uniprocessor:
    - Any kstack ≠ myself is current!
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

align tcbs on a power of 2:

Thread switch (IA32)

Thread A
push X
pusha
mov esp, ebp
and -sizeof tcb, ebp
mov [ebp].thr_esp, esp
mov esp, eax
and -sizeof tcb, eax
add sizeof tcb, eax
mov [eax].esp0_ptr, esp
popa
add $4, esp
int 32

Thread B
switch current
kernel stack pointer
push X
pusha
mov esp, ebp
and -sizeof tcb, ebp
mov [ebp].thr_esp, esp
mov esp, eax
and -sizeof tcb, eax
add sizeof tcb, eax
mov eax, esp0_ptr
popa
add $4, esp
int 32
Switch threads (IA32)

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

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

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

TCB structure

<table>
<thead>
<tr>
<th>Thread Id</th>
<th>MyselfGlobal</th>
<th>MyselfLocal</th>
<th>State</th>
<th>Resources</th>
<th>KernelStackPtr</th>
<th>Scheduling</th>
<th>ReadyList</th>
<th>TimesliceLength</th>
<th>RemainingTimeslice</th>
<th>TotalQuantum</th>
<th>Priority</th>
<th>Scheduler</th>
<th>Address Space</th>
<th>Round Robin</th>
<th>Scheduler</th>
</tr>
</thead>
</table>

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 all threads’ 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
- Via 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
- 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
  - other kernel tables
  - physical memory
  - kernel code
  - tcb

TLB working set grows: 4 entries to cover 4000 threads.
Nevertheless much better than 1 TLB for 8 threads like in direct address.
**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 != current
- Save current state to fpu_owner
- Load new state from current

Unlock FPU:

- fpu_owner := current

pacman()
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)

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.

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
  - Guaranteed to avoid user-level page faults during IPC

- Direct strings (optional)

- Indirect strings (optional)

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

- Strings (optional)

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

[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?

IPC - API

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

- Message Types
  - Registers
  - Strings
  - Map pages

IPC - API

- Message Types
  - Registers
  - Strings
  - Map pages

IPC - API

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

- Message Types
  - Registers
  - Strings
  - Map pages

Timeouts (V2, V X.0)

- snd timeout, rcv timeout

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 = \[ e \cdot 2^m \]
  \[ e = 0 \]
  \[ 4 \cdot 2^m \]
  \[ e > 0 \]
  \[ 0 \]
  \[ m = 0, e = 0 \]

Page fault timeout has no mantissa

- Page fault timeout = \[ \infty \]
  \[ p = 0 \]
  \[ 4 \cdot 2^p \]
  \[ 0 < p < 15 \]
  \[ 0 \]
  \[ p = 15 \]

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>0</td>
<td>268,439,450</td>
<td>986,821,000</td>
<td>17,772,398</td>
<td>4,194,036</td>
<td>1,048,576</td>
<td>1.048576</td>
<td>0.262144</td>
<td>0.065536</td>
<td>0.016384</td>
<td>0.004096</td>
<td>0.001024</td>
<td>0.000256</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>

IPC - API

- Timeouts (V X.0, V X.1, V X.2)
  - snd timeout, rcv timeout
    - xfer timeout snd, xfer timeout rcv

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 unbounded.
  - If pager can respond with null mapping that does not resolve the fault.

IPC - API
IPC - API

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

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

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

To Encode for IPC
- Send to
- Receive from
- Call
- Send to & Receive
- Send to, Receive from
- Destination thread ID
- Source thread ID
- 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
- Send from thread ID
- Specify deceiving IPC
- Thread ID to deceit as
- Intended receiver of deceived IPC

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

Sender Registers

Receiver Registers

Call-reply example

Thread A

IPC call

pre

Thread B

IPC reply & wait

pre

post

IPC reply & wait

post
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
- Receive from
- Call
- Send to & Receive
- 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
- IPC result code
- Send timeout
- Receive timeout
- Send Xfer timeout
- Receive Xfer timeout
- Send from thread ID
- Receive from thread ID
- Specify deceiting IPC
- Thread ID to deceit as
- Intended receiver of deceited 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
- 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
- IPC result code
- Send timeout
- Receive timeout
- Send Xfer timeout
- Receive Xfer timeout
- Send from thread ID
- Receive from thread ID
- Specify deceiting IPC
- Thread ID to deceit as
- Intended receiver of deceited 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 (MR_i... MR_i+6).
- First register (MR_0) acts as message tag.
- Subsequent registers contain:
  - Untyped words (u), and
  - Typed words (t) (e.g., map item, string item).

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 indicates map or grant item

String items

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

To Encode for IPC

- Send to
- Receive from
- Source thread ID
- Destination thread ID
- Send to & Receive
- 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
- IPC result code
- Send timeout
- Receive timeout
- Send after timeout
- Receive after timeout
- Specified thread ID
- Specify deceiving IPC
- Thread ID to deceive as
- Intended receiver of deceived IPC

Semantics will be explained during memory management lecture.
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
- 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
- Send string start 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
- 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

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 (BRi ... BRj)
- First BR (BR0) contains "Acceptor"
  - May specify receive window (if not nil-page)
  - May indicate presence of receive strings/buffers (if n-bit set)

To Encode for IPC

- Number of map pages
- Page range for each map page
- 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
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
- Send to
- Receive from
- Call
- Send to & Receive
- Send to, Receive from
- Destination thread ID
- Source thread ID
- Sender 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 Xfer timeout
- Receive Xfer timeout
- Receive from thread ID
- Specify deceiting IPC
- Thread ID to deceit as
- Intended receiver of deceived IPC

IPC Result
- IPC errors flagged in MR0
- Senders thread ID stored in register

IPC Redirection
- Redirection/deceiting 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

Virtual Registers
- What about messages and buffer registers?
- Most architectures do not have 64+34 spare registers
- Define as Virtual Registers
- What about predefined memory locations?
- Must be thread local
- Define as Virtual Registers
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

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>8AX</td>
<td></td>
</tr>
<tr>
<td>8CX</td>
<td></td>
</tr>
<tr>
<td>8DX</td>
<td></td>
</tr>
<tr>
<td>8BX</td>
<td></td>
</tr>
<tr>
<td>8SI</td>
<td></td>
</tr>
<tr>
<td>8DI</td>
<td></td>
</tr>
<tr>
<td>destination</td>
<td></td>
</tr>
<tr>
<td>timeouts</td>
<td></td>
</tr>
<tr>
<td>receive specifier</td>
<td>MR0</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>MR1</td>
<td>MR1</td>
</tr>
<tr>
<td>UT CB</td>
<td>UT CB</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>8AX</td>
<td></td>
</tr>
<tr>
<td>8CX</td>
<td></td>
</tr>
<tr>
<td>8DX</td>
<td></td>
</tr>
<tr>
<td>8BX</td>
<td></td>
</tr>
<tr>
<td>8SI</td>
<td></td>
</tr>
<tr>
<td>8DI</td>
<td></td>
</tr>
<tr>
<td>destination</td>
<td>from</td>
</tr>
<tr>
<td>timeouts</td>
<td></td>
</tr>
<tr>
<td>receive specifier</td>
<td>MR0</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>MR1</td>
<td>MR1</td>
</tr>
<tr>
<td>UT CB</td>
<td>UT CB</td>
</tr>
</tbody>
</table>