µ-Kernel Construction

Fundamental Abstractions

A thread is an independent flow of control inside an address space. Threads are identified by unique identifiers and communicate via IPC. Threads are characterized by a set of registers, including at least an instruction pointer, a stack pointer and a state information. A thread’s state also includes the address space in which the thread currently executes.

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.

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

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

Thread Switch A → B

user mode A

Processor

IP
SP
FLAGS

tcb A

IP
SP
FLAGS

tcb B
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.
Per-Thread Kernel Stack
Processes Model

- 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 threads stack until thread is resumed
- Resuming is simply switching back to the original stack
- Preemption is easy
- No conceptual difference between kernel mode and user mode

```
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 {
    P3();
  }
  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, ...);
  return rc;
}
```

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

IPC examples - Continuations

```
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;
}
```

```
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;
}
```

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 – 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;
}
```

Set user-level PC to restart `msg_rcv` only.

Single Kernel Stack

- **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
  - 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!

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!

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!
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 point
  - 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

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 $0(esp), edx
- sti
- sysexit

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 $0(esp), edx
  - sti
  - sysexit
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.

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

Thread switch (IA32)

Thread A

push X
pusha
mov esp, ebp
add esp, -sizeof tcb, ebp
mov [ebp].thr_esp, esp
mov esp, eax
add sizeof tcb, eax
add sizeof tcb, eax
mov eax, [ebp].thr_esp
popa
add $4, esp
iret

Thread B

switch esp
so that next enter kernel uses new kernel stack

switch current
kernel stack pointer
Switch threads (IA32)

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

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

- Trick:
  MSR points to esp0
  mov (esp), esp
Case study: IA-64

Thread Switching and Kernel Entry

Thread Switching Overhead
- All registers must be saved on context switches.
- More than 3.2KB of register contents.
- Certain optimizations made possible by hardware.

IA-64 User Accessible Registers

Thread Switching Overhead
- $r_{g0}$ fixed to zero.
- On thread switch:
  - Static registers must be saved explicitly.
  - Stacked registers handled by register stack engine (RSE).
- "Only" 2.5KB of register contents left.

Thread Switching Overhead
- $r_f$, and $r_f$, fixed.
- Remaining floating point registers can be handled lazily.
- "Only" ~0.5KB of register contents left.

Thread Switch Example
(pistachio/kernel/include/glue/v4-ia64/tcb.h)

About 50 instructions. Leave register save/restore up to compiler.
Exception Handling

- Bank 1 used normally
- Automatic switch to bank 0 on exceptions
- Frees up registers for storing context
- Can switch manually

Banked Registers

General Registers

Floating-point Registers

Predicates

Branch Registers

Application Registers

Instruction Pointer

User Mask

Current Frame Marker

Bank 1 used normally
Automatic switch to bank 0 on exceptions
Frees up registers for storing context
Can switch manually

Exception Handling

- Run on bank 1
- Exception
  - Switches to bank 0
  - Store other registers
  - Switch to bank 1
  - Store remaining registers

Must not receive interrupts or raise exceptions while storing exception context

Kernel Entry

- Kernel entry by exception is slow
- Must flush instruction pipeline
- IA-64 provides an epc instruction
  - Raises privileges to kernel mode
  - Continues execution on next instruction
  - Can only be executed in special regions of virtual memory

Mips R4600

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

- Thread Id
- MyselfGlobal
- MyselfLocal
- State
- Resources
- KernelStackPtr
- Scheduling
  - ReadyList
  - TimesliceLength
  - RemainingTimeslice
  - TotalQuantum
  - 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
  - other kernel tables
  - physical memory
  - kernel code
- tcb

per-space system regions
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
- Limit valid physical range to remap size (256M)
- Issue occurs only when kernel accesses physical memory

Kernel Debugger (not performance critical)

- Walk page table in software
- Remap on demand (4MB)
- Optimization: check if already mapped

Physical-to-virtual Pagetable

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

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
- If fpu_owner != current
  - Save current state to fpu_owner
  - Load new state from current
  - fpu_owner := current

Lazy FPU switching diagram:

- Kernel
- FPU
- fpu_owner
- pacman
- finit
- fld
- fcos
- fsub
- fadd
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)

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)
- 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?

- 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)
  - In-memory messages we construct to send

- Indirect strings (optional)
  - In-memory messages not in page

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

- Can be combined

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)
    - In-memory messages we construct to send
  - Indirect strings (optional)
    - In-memory messages not in page
  - 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 deal with threads that are:
  - Uncooperative
  - Malfunctioning
  - Malicious
- That might result in an IPC operation never completing?

**IPC - API**
- Timeouts (V2, V X.0)
  - snd timeout, rcv timeout

**IPC - API**
- Timeouts (V2, V X.0)
  - snd timeout, rcv timeout
  - snd-pf timeout
  - specified by sender

**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) [V 2, V X.0]

<table>
<thead>
<tr>
<th>e</th>
<th>m</th>
<th>(10^4)</th>
<th>(10^6)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(\infty)</td>
<td>(\infty)</td>
<td>(\infty)</td>
</tr>
<tr>
<td>1</td>
<td>288,435,469</td>
<td>68,451,041,280</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>87,108,641</td>
<td>17,112,760,320</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>16,777,216</td>
<td>4,278,190,080</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4,194,304</td>
<td>1,069,547,520</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1,048,576</td>
<td>267,386,880</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>262,144</td>
<td>66,846,720</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>64,000</td>
<td>17,116,640</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>16,384</td>
<td>4,279,488</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>4,096</td>
<td>1,069,312</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1,024</td>
<td>267,424</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>256</td>
<td>171,104</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>64</td>
<td>42,928</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>16</td>
<td>10,752</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>4</td>
<td>2,710</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>684</td>
<td></td>
</tr>
</tbody>
</table>

IPC - API

- Timeout values:
  - 0
  - infinite
  - 1us … 19h (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 + \(n\) × 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

- Timeout values:
  - 0
  - 1us to 255µs with 1µs granularity
  - Up to 19h with ~4.4min granularity
  - 1us – 255µs with 1µs granularity

IPC - API

- Timeout values:
  - 0
  - infinite
  - 1us … 19h (log)
  - Compact 32-bit encoding
### IPC - API

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

- **relative timeout values**
  - 0
  - infinite
  - 1µs … 610 h (log)

- **absolute timeout values**
  - 0
  - infinite
  - 1µs … 610 h (log)

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

<table>
<thead>
<tr>
<th>e</th>
<th>m</th>
<th>µs</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0.000001</td>
<td>0.00127</td>
</tr>
<tr>
<td>6</td>
<td>0.000002</td>
<td>0.00254</td>
</tr>
<tr>
<td>7</td>
<td>0.000003</td>
<td>0.00381</td>
</tr>
<tr>
<td>8</td>
<td>0.000004</td>
<td>0.00508</td>
</tr>
<tr>
<td>9</td>
<td>0.000005</td>
<td>0.00635</td>
</tr>
<tr>
<td>10</td>
<td>0.000006</td>
<td>0.00762</td>
</tr>
<tr>
<td>11</td>
<td>0.000007</td>
<td>0.00889</td>
</tr>
<tr>
<td>12</td>
<td>0.000008</td>
<td>0.01016</td>
</tr>
<tr>
<td>13</td>
<td>0.000009</td>
<td>0.01143</td>
</tr>
<tr>
<td>14</td>
<td>0.000010</td>
<td>0.01270</td>
</tr>
<tr>
<td>15</td>
<td>0.000011</td>
<td>0.01397</td>
</tr>
<tr>
<td>16</td>
<td>0.000012</td>
<td>0.01524</td>
</tr>
<tr>
<td>17</td>
<td>0.000013</td>
<td>0.01651</td>
</tr>
<tr>
<td>18</td>
<td>0.000014</td>
<td>0.01778</td>
</tr>
<tr>
<td>19</td>
<td>0.000015</td>
<td>0.01905</td>
</tr>
<tr>
<td>20</td>
<td>0.000016</td>
<td>0.02032</td>
</tr>
<tr>
<td>21</td>
<td>0.000017</td>
<td>0.02159</td>
</tr>
<tr>
<td>22</td>
<td>0.000018</td>
<td>0.02286</td>
</tr>
<tr>
<td>23</td>
<td>0.000019</td>
<td>0.02413</td>
</tr>
<tr>
<td>24</td>
<td>0.000020</td>
<td>0.02540</td>
</tr>
<tr>
<td>25</td>
<td>0.000021</td>
<td>0.02667</td>
</tr>
<tr>
<td>26</td>
<td>0.000022</td>
<td>0.02794</td>
</tr>
<tr>
<td>27</td>
<td>0.000023</td>
<td>0.02921</td>
</tr>
<tr>
<td>28</td>
<td>0.000024</td>
<td>0.03048</td>
</tr>
<tr>
<td>29</td>
<td>0.000025</td>
<td>0.03175</td>
</tr>
<tr>
<td>30</td>
<td>0.000026</td>
<td>0.03302</td>
</tr>
<tr>
<td>31</td>
<td>0.000027</td>
<td>0.03429</td>
</tr>
</tbody>
</table>

1µs = 100µs with 1µs granularity

Up to ~610h with ~30min 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**
- **Specify deceiting 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

---

### Call-reply example

**Thread A**
- **pre**
- IPC call & wait

**Thread B**
- **pre**
- IPC reply & wait

**Thread A**
- **post**
- **pre**
- IPC reply & wait

---

### Parameters in Registers

<table>
<thead>
<tr>
<th>Sender Registers</th>
<th>Receiver Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>EAX</td>
</tr>
<tr>
<td>ECX</td>
<td>ECX</td>
</tr>
<tr>
<td>EDX</td>
<td>EDX</td>
</tr>
<tr>
<td>EBX</td>
<td>EBX</td>
</tr>
<tr>
<td>EBP</td>
<td>EBP</td>
</tr>
<tr>
<td>ESi</td>
<td>ESi</td>
</tr>
<tr>
<td>EDI</td>
<td>EDI</td>
</tr>
</tbody>
</table>
**Send and Receive Encoding**

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

```
Send and Receive Encoding

<table>
<thead>
<tr>
<th>Sender Registers</th>
<th>Receiver Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>EBX</td>
</tr>
<tr>
<td>8-bit</td>
<td>16-bit</td>
</tr>
<tr>
<td>destination</td>
<td>receive specifier</td>
</tr>
</tbody>
</table>

Nil ID means no send 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
- Reserve 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 deceit 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)

```
Number of untyped words
Number of typed words
Various IPC flags
Message Tag
Freely available (e.g., request type)
```
Message construction

- Messages are stored in registers (MR0, ..., MR63)
- First register (MR0) acts as message tag
- Subsequent registers contain:
  - Untyped words (u), and MR
  - 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

String items

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

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
- Specify deceited IPC
- Thread ID to deceit as
- Intended receiver of deceited IPC
Timeouts

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

Timeout 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
    - 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

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 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

String length
- Receive string start for each string
- Receive string size for each string

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)

Receive from thread ID
- Specify deceiting IPC
- Thread ID to deceit as
- Intended receiver of deceited 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 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

**IPC Result**

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

**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
    - No page faults
    - Registers always accessible
- Preserved by kernel during context switch

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

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

IPC Register Encoding

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

What About IA-64?

- All other registers are undefined