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 $\mu$K construction?
Processor

IP
SP
FLAGS

IP
SP
FLAGS

user mode A

kernel

tcb A

IP
SP
FLAGS
user mode A
kernel
user mode A

kernel
user mode A
kernel
user mode A

kernel
Construction conclusion

From the view of the designer there are two alternatives.

**Single Kernel Stack**

- Only one stack is used all the time.

**Per-Thread Kernel Stack**

- Every thread has a kernel stack.
Single Kernel Stack
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, NICTA Pistachio, OKL4

- 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!
  - 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

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

- but larger cache footprint
- larger memory footprint
Kernel Entry/Exit

- A look at mechanics of kernel entry and exit
- Optimisations
- Context switching
enter kernel (IA32)

CPU

- esp
- eip
- eflags
- eax ebx ecx edx ebp esi edi

user mode

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

points to the running threads kernel stack

user stack

tcb A

ekernelp code

esp0

/\square6
trap / fault occurs (\textit{INT} \textit{n} / exception / interrupt)
- push user esp on to kernel stack, load kernel esp
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)
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 eip, load kernel entry eip

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

- **CPU**
  - esp
  - eip
  - eflags
  - eax ebx ecx edx ebp esi edi

- **tcb A**
  - X eip cs flg esp ss

- **kernel code**
  - esp0

- **user stack**

**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 eip, load kernel entry eip
  - push X : error code (hw, at exception) or kernel-call type

---

*hardware programmed, single instruction*
**Enter Kernel (IA32)**

CPU

- esp
- eip
- eflags
- eax ebx ecx edx ebp esi edi

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 eip, load kernel entry eip
- push X : error code (hw, at exception) or kernel-call type
- push registers (optional)

**Hardware programmed, single instruction**
System call (IA32)

int 0x32

push X
pusha
...
...
popa
add $4, esp
iret

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

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

Kernel esp

align tcbs on a power of 2:

esp

tcb

esp0
Remember:

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

To find out the starting address from the tcb.

align tcbs:

```assembly
mov esp, ebp
and -sizeof tcb, ebp
```
Thread switch (IA32)

Thread A

```
push X
pusha
mov esp, ebp
and -sizeof tcb, ebp

dest tcb address -> edi
```

Thread B

```
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 esp0 so that next enter kernel uses new kernel stack**

**switch current kernel stack pointer**

int 32

int 32
Switch threads (IA32)
Switch threads (IA32)

- int 0x32, push registers of the green thread
Switch threads (IA32)

- int 0x32, push registers of the green thread
- switch kernel stacks (store and load esp)
Switch threads (IA32)

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

<table>
<thead>
<tr>
<th>Exc PC</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>r31</td>
<td>k0</td>
</tr>
<tr>
<td>r30</td>
<td>k1</td>
</tr>
<tr>
<td>r29</td>
<td></td>
</tr>
<tr>
<td>r28</td>
<td></td>
</tr>
<tr>
<td>r27</td>
<td></td>
</tr>
<tr>
<td>r26</td>
<td></td>
</tr>
<tr>
<td>r25</td>
<td></td>
</tr>
<tr>
<td>r24</td>
<td></td>
</tr>
<tr>
<td>r23</td>
<td></td>
</tr>
<tr>
<td>r22</td>
<td></td>
</tr>
<tr>
<td>r21</td>
<td></td>
</tr>
<tr>
<td>r20</td>
<td></td>
</tr>
<tr>
<td>r19</td>
<td></td>
</tr>
<tr>
<td>r18</td>
<td></td>
</tr>
<tr>
<td>r17</td>
<td></td>
</tr>
<tr>
<td>r16</td>
<td></td>
</tr>
<tr>
<td>r15</td>
<td></td>
</tr>
<tr>
<td>r14</td>
<td></td>
</tr>
<tr>
<td>r13</td>
<td></td>
</tr>
<tr>
<td>r12</td>
<td></td>
</tr>
<tr>
<td>r11</td>
<td></td>
</tr>
<tr>
<td>r10</td>
<td></td>
</tr>
<tr>
<td>r9</td>
<td></td>
</tr>
<tr>
<td>r8</td>
<td></td>
</tr>
<tr>
<td>r7</td>
<td></td>
</tr>
<tr>
<td>r6</td>
<td></td>
</tr>
<tr>
<td>r5</td>
<td></td>
</tr>
<tr>
<td>r4</td>
<td></td>
</tr>
<tr>
<td>r3</td>
<td></td>
</tr>
<tr>
<td>r2</td>
<td></td>
</tr>
<tr>
<td>r1</td>
<td></td>
</tr>
<tr>
<td>r0 = 0</td>
<td></td>
</tr>
</tbody>
</table>
Problems

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

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

  move k1, sp          /* Save previous stack pointer in k1 */
  mfc0 k0, c0_status   /* Get status register */
  andi k0, k0, CST_up  /* Check the we-were-in-user-mode bit */
  beq k0, $0, 1f      /* If clear, from kernel, already have stack */
  nop                  /* delay slot */

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

1:

  mfc0 k0, c0_cause    /* Now, load the exception cause. */
  j common_exception   /* Skip to common code */
  nop
exception:

    move k1, sp        /* Save previous stack pointer in k1 */
    mfc0 k0, c0_status /* Get status register */
    andi k0, k0, CST_Kup /* Check the we-were-in-user-mode bit */
    beq k0, $0, 1f     /* If clear, from kernel, already have stack */
    nop                 /* delay slot */

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

1:

    mfc0 k0, c0_cause   /* Now, load the exception cause. */
    j common_exception  /* Skip to common code */
    nop                 /* delay slot */
common_exception:

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

/*
 * Allocate stack space for 37 words to hold the trap frame,
 * plus four more words for a minimal argument block.
 */
addi sp, sp, -164
/* The order here must match mips/include/trapframe.h. */

sw ra, 160(sp) /* dummy for gdb */
sw s8, 156(sp) /* save s8 */
sw sp, 152(sp) /* dummy for gdb */
sw gp, 148(sp) /* save gp */
sw k1, 144(sp) /* dummy for gdb */
sw k0, 140(sp) /* dummy for gdb */
sw k1, 152(sp) /* real saved sp */
nop /* delay slot for store */

mfc0 k1, c0_epc /* Copr.0 reg 13 == PC for
sw k1, 160(sp) /* real saved PC */

These six stores are a “hack” to avoid confusing GDB
You can ignore the details of why and how
The order here must match mips/include/trapframe.h. */

sw ra, 160(sp) /* dummy for gdb */
sw s8, 156(sp) /* save s8 */
sw sp, 152(sp) /* dummy for gdb */
sw gp, 148(sp) /* save gp */
sw k1, 144(sp) /* dummy for gdb */
sw k0, 140(sp) /* dummy for gdb */

sw k1, 152(sp) /* real saved sp */
nop /* delay slot for store */

mfc0 k1, c0_epc /* Copr.0 reg 13 == PC for exception */
sw k1, 160(sp) /* real saved PC */
sw t9, 136(sp)
sw t8, 132(sp)
sw s7, 128(sp)
sw s6, 124(sp)
sw s5, 120(sp)
sw s4, 116(sp)
sw s3, 112(sp)
sw s2, 108(sp)
sw s1, 104(sp)
sw s0, 100(sp)
sw t7, 96(sp)
sw t6, 92(sp)
sw t5, 88(sp)
sw t4, 84(sp)
sw t3, 80(sp)
sw t2, 76(sp)
sw t1, 72(sp)
sw t0, 68(sp)
sw a3, 64(sp)
sw a2, 60(sp)
sw a1, 56(sp)
sw a0, 52(sp)
sw v1, 48(sp)
sw v0, 44(sp)
sw AT, 40(sp)
sw ra, 36(sp)

Save all the registers on the kernel stack
/**
 * Save special registers.
 */
mfhi t0
mflo t1
sw t0, 32(sp)
sw t1, 28(sp)

/**<*
 * Save remaining exception context information.
 */
sw k0, 24(sp)  /* k0 was loaded with cause earlier */
mfc0 t1, c0_status    /* Copr.0 reg 11 == status */
sw t1, 20(sp)
mfc0 t2, c0_vaddr  /* Copr.0 reg 8 == faulting vaddr */
sw t2, 16(sp)

/**<*
 * Pretend to save $0 for gdb's benefit.
 */
sw $0, 12(sp)
/*
 * Prepare to call mips_trap(struct trapframe *)
 */

addiu a0, sp, 16           /* set argument */
jal mips_trap              /* call it */
nop                       /* delay slot */
By creating a pointer to here of type \texttt{struct trapframe *}, we can access the user’s saved registers as normal variables within ‘C’.
enter kernel: (Mips)

Load kernel stack pointer if trap from user mode

```
and     k1, t0, st_ksu_mask
IFNZ    k1
        mov    t2, sp
        mov    sp, kernel_stack_bottom(k0)
FI
mov     t1, C0_exception_ip
mov     [sp-8], t2
add     t1, t1, 4
mov     [sp-16], t1
mov     [sp-24], t0
IFZ     AT, zero
        sub     sp, 24
        jmp     k_ipc
FI
```

Push old sp (t2), ip (t1), and status (t0)

```
mov     t0, k1
srl     k1, 5  /* clear IE, EXL, ERL, KSU */
sll     k1, 5
mov     C0_status, k1
```
TCB structure

- Thread Id
- Local Id = UTCB
- All threads ready to execute
- Round Robin Scheduler
- Address Space
- Optimization IA32: %CR3

MyselfGlobal
MyselfLocal
State
Resources
KernelStackPtr
Scheduling
  ReadyList
  TimesliceLength
  RemainingTimeslice
  TotalQuantum
  Priority
  WakeupList
Space
PDirCache
...
Stack[]

Optimization IA32: %CR3
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 $\rightarrow$ 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.
- 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.
- TLB working set grows: 4 entries to cover 4000 threads.
- Nevertheless much better than 1 TLB for 8 threads like in direct address.
AS Layout  32bits, virt tcb, entire PM

- user regions
- shared system regions
  - other kernel tables
  - physical memory
  - kernel code
  - tcbs
- per-space system regions

phys mem
Limitations

- number of threads
- physical mem size

32bits, virt tcb, entire PM

L4Ka::Pistachio/ia32:
- 262,144 threads
- 256 M physical memory

Nearly every desktop PC has more than 256 M physical memory

<table>
<thead>
<tr>
<th>3 G</th>
<th>512 M</th>
<th>256 M</th>
<th>256 M</th>
</tr>
</thead>
<tbody>
<tr>
<td>phys mem</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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
    fpu_owner := current

Kernel

FPU

Locked

current  fpu_owner

pacman()

finit
fld

finit
fld

fcos
fst
IPC

Functionality & Interface
What IPC primitives do we need to communicate?

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

- Two threads can communicate
- Can create specific protocols without fear of interference from other threads
- Other threads block until it’s their turn

Problem:
- How to communicate with a thread unknown a priori
  (e.g., a server’s clients)
What IPC primitives do we need to communicate?

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

- **Call**
  - (send to, receive from specified thread)

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

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

**Are other combinations appropriate?**

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

- **Atomic** operation for optimization reasons. Typically used by servers to reply and wait for the next request (from anyone).
What message types are appropriate?

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

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

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

- **Strings** *(optional)*
  - In-memory 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, v x.0)
  - snd timeout, rcv timeout
IPC - API

- **Timeouts** $(v2, v.x.0)$
  - snd timeout, rcv timeout
    - snd-pf timeout
      - specified by sender

- Attack through receiver’s pager:
IPC - API

- Timeouts (V2, Vx.0)
  - snd timeout, rcv timeout
    - snd-pf / rcv-pf timeout
      - specified by receiver

- Attack through sender’s pager:
Timeout Issues

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

Timeout values

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

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

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

send/receive timeout = \[
\begin{cases} 
\infty & \text{if } e = 0 \\
4^{15-m} & \text{if } e > 0 \\
0 & \text{if } m = 0, e \neq 0
\end{cases}
\]
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>$m = 1$</th>
<th>$m = 255$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$\infty$</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>268,435456</td>
<td>68451,04128</td>
</tr>
<tr>
<td>2</td>
<td>67,108864</td>
<td>17112,76032</td>
</tr>
<tr>
<td>3</td>
<td>16,777216</td>
<td>4278,19008</td>
</tr>
<tr>
<td>4</td>
<td>4,194304</td>
<td>1069,54752</td>
</tr>
<tr>
<td>5</td>
<td>1,048576</td>
<td>267,38688</td>
</tr>
<tr>
<td>6</td>
<td>0,262144</td>
<td>66,84672</td>
</tr>
<tr>
<td>7</td>
<td>0,065536</td>
<td>16,71168</td>
</tr>
<tr>
<td>8</td>
<td>0,016384</td>
<td>4,17792</td>
</tr>
<tr>
<td>9</td>
<td>0,004096</td>
<td>1,04448</td>
</tr>
<tr>
<td>10</td>
<td>0,001024</td>
<td>0,26112</td>
</tr>
<tr>
<td>11</td>
<td>0,000256</td>
<td>0,06528</td>
</tr>
<tr>
<td>12</td>
<td>0,000064</td>
<td>0,01632</td>
</tr>
<tr>
<td>13</td>
<td>0,000016</td>
<td>0,00408</td>
</tr>
<tr>
<td>14</td>
<td>0,000004</td>
<td>0,00102</td>
</tr>
<tr>
<td>15</td>
<td>0,000001</td>
<td>0,000255</td>
</tr>
</tbody>
</table>

- Up to 19h with ~4.4min granularity
- 1μs – 255μs with 1μs granularity
IPC - API

- Timeouts (V2, v x.0)
  - snd timeout, rcv timeout
    - snd-pf / rcv-pf timeout
  - timeout values
    - 0
    - infinite
    - 1us ... 19 h (log)
  - Compact 32-bit encoding
IPC - API

- Timeouts (v x.2, v 4)
  - snd timeout, rcv timeout, xfer timeout snd, xfer timeout rcv
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 deceive as
- Intended receiver of deceived IPC
Ideally Encoded in Registers

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

Sender Registers

- EAX
- ECX
- EDX
- EBX
- EBP
- ESI
- EDI

Receiver Registers

Universität Karlsruhe (TH)
Call-reply example

Thread A

IPC call

pre

IPC reply & wait

post

Thread B

pre

IPC reply & wait

pre

post
Send and Receive Encoding

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

Sender Registers

- EAX: destination
- EDX: receive specifier

**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 deceive as
- Intended receiver of deceived IPC
Message Transfer

- Assume that 64 extra registers are available
  - Name them $MR_0$ ... $MR_{63}$ (message registers 0 ... 63)
  - All message registers are transferred during IPC
To Encode for IPC

- 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 construction

- Messages are stored in registers ($MR_0 \ldots MR_{63}$)
- First register ($MR_0$) acts as message tag
- Subsequent registers contain:
  - Untyped words ($u$), and
  - Typed words ($t$)
    (e.g., map item, string item)

![Message Tag Diagram]

- Number of untyped words
- Number of typed words
- Various IPC flags
- 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
  - Typed words (t)
    (e.g., map item, string item)
Message construction

- 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
Map and Grant items

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

Semantics will be explained during memory management lecture
String items

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

```
String Pitems
/square 6
Max Psize P4MB P(per Pstring) /
square 6
Compound Pstrings Psupported /
/square 6
Allows Pscatter-gather /
/square 6
Incorporates Pcacheability Phints /
/square 6
Reduce Pcache Ppollution Pfor Plong Pcopy Poperations
```

```
String Plength
String Ppointer
String PItem
```

```
"hh" Pindicates Pcacheability Phints Pfor Pthe Pstring
```

```
string pointer
```

```
string length
```

```
MR_{i+1}
```

```
MR_i
```

```
String Item
```

```
“hh” Pindicates Pcacheability Phints Pfor Pthe Pstring
```
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

"hh" indicates cacheability hints for the string
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
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 deceived IPC
String Receival

- Assume that **34 extra registers** are available
  - Name them $BR_0 \ldots BR_{33}$ (buffer registers 0 ... 33)
- Buffer registers specify
  - Receive strings
  - Receive window for mappings
Receiving messages

- Receiver buffers are specified in registers ($BR_0 \ldots BR_{33}$)

- First $BR$ ($BR_0$) contains "Acceptor"
  - May specify receive window (if not nil-fpage)
  - May indicate presence of receive strings/buffers (if s-bit set)
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.

If C-bit in string item is set, it indicates presence of more receive buffers.

The s-bit set indicates presence of string items acting as receive buffers.

string pointer  

string pointer  

string pointer  

string length \(0 - 1\)  

string pointer  

string length \(0\)  

receive window  

Acceptors

BR\(_4+j\)  

BR\(_5\)  

BR\(_4\)  

BR\(_3\)  

BR\(_2\)  

BR\(_1\)  

BR\(_0\)  

Universität Karlsruhe (TH)
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 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
**IPCP Result**

- IPC errors flagged in $MR_0$
- Senders thread ID stored in register

<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>ESP</strong></td>
<td></td>
</tr>
<tr>
<td><strong>EDI</strong></td>
<td></td>
</tr>
</tbody>
</table>
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 deceive as
- Intended receiver of deceived IPC
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
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
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.

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
    - 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
  \texttt{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

Sender Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>destination</td>
</tr>
<tr>
<td>ECX</td>
<td>timeouts</td>
</tr>
<tr>
<td>EDX</td>
<td>receive specifier</td>
</tr>
<tr>
<td>EBX</td>
<td>MR₁</td>
</tr>
<tr>
<td>EBP</td>
<td>MR₂</td>
</tr>
<tr>
<td>ESI</td>
<td>MR₀</td>
</tr>
<tr>
<td>EDI</td>
<td>UTCB</td>
</tr>
</tbody>
</table>

Receiver Registers

<table>
<thead>
<tr>
<th>Description</th>
<th>EMR₁</th>
<th>EMR₂</th>
<th>EMR₀</th>
<th>UTCB</th>
</tr>
</thead>
<tbody>
<tr>
<td>from</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MR₁</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MR₂</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MR₀</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UTCB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Free Up Registers for Temporary Values

- Kernel need registers for temporary values
- $MR_1$ and $MR_2$ are the only registers that the kernel may not need

### Sender Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>destination</td>
</tr>
<tr>
<td>ECX</td>
<td>timeouts</td>
</tr>
<tr>
<td>EDX</td>
<td>receive specifier</td>
</tr>
<tr>
<td>EBX</td>
<td>MR_1</td>
</tr>
<tr>
<td>EBP</td>
<td>MR_2</td>
</tr>
<tr>
<td>ESI</td>
<td>MR_0</td>
</tr>
<tr>
<td>EDI</td>
<td>UTCB</td>
</tr>
</tbody>
</table>

### Receiver Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>from</td>
</tr>
<tr>
<td></td>
<td>MR_1</td>
</tr>
<tr>
<td></td>
<td>MR_2</td>
</tr>
<tr>
<td></td>
<td>MR_0</td>
</tr>
<tr>
<td></td>
<td>UTCB</td>
</tr>
</tbody>
</table>
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>EAX</td>
<td>destination</td>
</tr>
<tr>
<td>ECX</td>
<td>timeouts</td>
</tr>
<tr>
<td>EDX</td>
<td>receive specifier</td>
</tr>
<tr>
<td>EBX</td>
<td>~</td>
</tr>
<tr>
<td>EBP</td>
<td>~</td>
</tr>
<tr>
<td>ESI</td>
<td>MR₀</td>
</tr>
<tr>
<td>EDI</td>
<td>UTCB</td>
</tr>
<tr>
<td></td>
<td>from</td>
</tr>
<tr>
<td></td>
<td>MR₁</td>
</tr>
<tr>
<td></td>
<td>MR₂</td>
</tr>
<tr>
<td></td>
<td>MR₀</td>
</tr>
<tr>
<td></td>
<td>UTCB</td>
</tr>
</tbody>
</table>
IPCP Register Encoding

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

Sender Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>destination</td>
</tr>
<tr>
<td>ECX</td>
<td>timeouts</td>
</tr>
<tr>
<td>EDX</td>
<td>receive specifier</td>
</tr>
<tr>
<td>EBX</td>
<td>~</td>
</tr>
<tr>
<td>EBP</td>
<td>~</td>
</tr>
<tr>
<td>ESI</td>
<td>MR₀</td>
</tr>
<tr>
<td>EDI</td>
<td>UTCB</td>
</tr>
</tbody>
</table>

Receiver Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>from</td>
</tr>
<tr>
<td></td>
<td>MR₁</td>
</tr>
<tr>
<td></td>
<td>MR₂</td>
</tr>
<tr>
<td></td>
<td>MR₀</td>
</tr>
<tr>
<td></td>
<td>UTCB</td>
</tr>
</tbody>
</table>