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?*
A “thread of control” has

- register set
  - e.g. general registers, IP and SP
- stack
- status
  - e.g. FLAGs, privilege,
  - OS-specific states (prio, time...)
- address space
- unique id
- communication status
Construction Conclusions (1)

♦ Thread state must be saved / restored on thread switch.
♦ We need a thread control block (tcb) per thread.
♦ Tcbs must be kernel objects.

(at least partially, we found some good reasons to implement parts of the TCB in user memory.)

♦ Tcbs implement threads.

♦ We need to find
  ■ any thread’s tcb starting from its uid
  ■ the currently executing thread’s tcb (per processor)
Thread Switch A → B

user mode A
Thread Switch A → B

Processor

user mode A

kernel

Thread Switch A → B

user mode A

kernel

tcb B
Thread Switch A \( \rightarrow \) B

user mode A

\textit{kernel}

Processor

\begin{tabular}{|c|}
\hline
IP \\
SP \\
FLAGS \\
\hline
\end{tabular}

\begin{tabular}{|c|}
\hline
IP \\
SP \\
FLAGS \\
\hline
\end{tabular}

tcb A

tcb B
Thread Switch A → B

Processor

user mode A
kernel

tcb A

tcb B
Thread Switch A → B

user mode A
kernel
user mode B
Thread Switch A → 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 the thread A on A’s TCB
- The next step is to load the status of thread B from B’s TCB.
- Leave kernel mode and thread B is running in user mode.
user mode A
user mode A

kernel
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.
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 thread's 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

```c
example(arg1, arg2) {
  P1(arg1, arg2);
  if (need_to_block) {
    thread_block();
    P2(arg2);
  } else {
    P3();
  }
  /* return control to user */
  return SUCCESS;
}
```
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

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

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

```c
msg_send_rcv(msg, option,
    send_size, rcv_size, ...) {

    rc = msg_send(msg, option,
        send_size, ...);

    if (rc != SUCCESS)
       return rc;

    rc = msg_rcv(msg, option, rcv_size, ...);
    return rc;
}
```

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

Block inside msg_rcv if no message available
IPC examples - Continuations

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

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

- 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
- difficult to exchange kernel on-the-fly

**Conclusion:**
Either no persistent tcbs or tcbs must hold virtual addresses
enter kernel (IA32)

CPU

- \text{esp}
- \text{eip}
- \text{eflags}
- \text{eax, ebx, ecx, edx, ebp, esi, edi}

user mode

- trap / fault occurs (\text{INT n} / \text{exception} / \text{interrupt})

points to the running threads kernel stack

user stack
**enter kernel (IA32)**

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

- **tcb A**
  - esp
  - ss

- **kernel code**
  - esp0

- **user stack**

---

**kernel mode**

- trap / fault occurs (\( \text{INT} \ n / \text{exception} / \text{interrupt} \))
  - 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 ($I=0, S=0$)
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

hardware programmed, single instruction
enter kernel (IA32)

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

CPU

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

user stack

kernel code

tcb A

esp0

kernel mode

hardware programmed, single instruction
enter kernel (IA32)

- trap / fault occurs (INT n / exception / interrupt)
  - push user esp on to kernel stack, load kernel esp
  - push user eflags, reset flags (I=0, S=0)
  - push user eip, load kernel entry eip
  - push X: error code (hw, at exception) or kernel-call type
  - push registers (optional)
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 \neq myself is current !**
  - (my kstack below [esp] is also current when in kernel mode.)

One thread is running and all the others are in their kernel-state and can analyze their stacks. All processes except the running are in kernel mode.
Kernel-stack state

Uniprocessor:

- **Any kstack ≠ myself is current !**
  
  (my kstack below [esp] is also current when in kernel mode.)

- X permits to differentiate between stack layouts:
  - interrupt, exception, some system calls
  - ipc
  - V86 mode
Kernel-stack state

Uniprocessor:

- **Any kstack ≠ myself is current !**
  - (my kstack below [esp] is also current when 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

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:

```
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

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

Thread B

switch current
kernel stack pointer

int 32

switch esp0 so that next
enter kernel uses new
kernel stack

int 32

Universität Karlsruhe (TH)
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
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
  ```
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

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

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

Load kernel stack pointer if trap from user mode

mov k1, C0_status
and k0, k1, exc_code_mask
sub k0, syscall_code
IFNZ k0

mov k0, kernel_base
jmp other_exception

no syscall trap

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)
TCB structure

- Thread Id
- Thread 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[]
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 $\rightarrow$ 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:

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

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.
AS Layout 32bits, virt tcb, entire PM

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

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 phys mem
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...

- Map and unmap
  - copy IPC

- Page tables
  - TCBs

- KDB output
  - Mem Dump
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 \(~1.5\)KB

- 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

 fcos

 fst

 finit

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

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

*Can be combined*
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

[Version 4, Version X.2]
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, V x.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
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 + $n \times$ page fault timeout

- Worst-case for a careless implementation is unbound
  - If pager can respond with null mapping that does not resolve the fault
IPC - API

- **Timeouts** (*v x.2, v 4*)
  - snd timeout, rcv timeout, xfer timeout snd, xfer timeout rcv

![Diagram showing IPC API with timeouts and send/receive messages](image)
IPC - API

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

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

  \[2^e \cdot m \mu s\]
IPC - API

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

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

  - absolute timeout values

  \[\text{clock} = m_{10} + 0\]

  \[\text{clock} + 2^{(e+10)} \neq m_{10}\]
## Timeout Range of Values (seconds) [V 4, V X.2]

<table>
<thead>
<tr>
<th>e</th>
<th>m = 1</th>
<th>m = 1023</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0,000001</td>
<td>0,001023</td>
</tr>
<tr>
<td>1</td>
<td>0,000002</td>
<td>0,002046</td>
</tr>
<tr>
<td>3</td>
<td>0,000008</td>
<td>0,008184</td>
</tr>
<tr>
<td>5</td>
<td>0,000032</td>
<td>0,032736</td>
</tr>
<tr>
<td>7</td>
<td>0,000128</td>
<td>0,130944</td>
</tr>
<tr>
<td>9</td>
<td>0,000512</td>
<td>0,523776</td>
</tr>
<tr>
<td>11</td>
<td>0,002048</td>
<td>2,095104</td>
</tr>
<tr>
<td>13</td>
<td>0,008192</td>
<td>8,380416</td>
</tr>
<tr>
<td>15</td>
<td>0,032768</td>
<td>33,521664</td>
</tr>
<tr>
<td>17</td>
<td>0,131072</td>
<td>134,086656</td>
</tr>
<tr>
<td>19</td>
<td>0,524288</td>
<td>536,346624</td>
</tr>
<tr>
<td>21</td>
<td>2,097152</td>
<td>2145,386496</td>
</tr>
<tr>
<td>23</td>
<td>8,388608</td>
<td>8581,545984</td>
</tr>
<tr>
<td>25</td>
<td>33,554432</td>
<td>34326,18394</td>
</tr>
<tr>
<td>27</td>
<td>134,217728</td>
<td>137304,7357</td>
</tr>
<tr>
<td>29</td>
<td>536,870912</td>
<td>549218,943</td>
</tr>
<tr>
<td>31</td>
<td>2147,483648</td>
<td>2196875,772</td>
</tr>
</tbody>
</table>

1µs – 1023µs with 1µs granularity
Up to ~610h with ~35min granularity
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
Ideally Encoded in Registers

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

Sender Registers

Receiver Registers

<table>
<thead>
<tr>
<th>EAX</th>
<th>ECX</th>
<th>EDX</th>
<th>EBX</th>
<th>EBP</th>
<th>ESI</th>
<th>EDI</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Call-reply example

Thread A

pre

IPC call

IPC reply & wait

post

Thread B

pre

IPC reply & wait

pre

post
Send and Receive Encoding

- \textbf{0} (\textit{Nil ID}) is a reserved thread ID
- Define \textbf{-1} as a \textit{wildcard} thread ID

\begin{itemize}
  \item \textbf{Nil ID} means no send operation
  \item \textbf{Nil ID} means no receive operation
  \item \textbf{Wildcard} means receive from any thread
\end{itemize}
Why use a single call instead of many?

- The implementation of the individual send and receive is very similar to the combined send and receive
  - We can use the same code
    - We reduce cache footprint of the code
    - We make applications more likely to be in cache
To Encode for IPC

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

- Assume that 64 extra registers are available
- Name them $\text{MR}_0 \ldots \text{MR}_{63}$ (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).

![Diagram of message tag structure](image-url)
Message construction

- Messages are stored in registers \(\text{MR}_0 \ldots \text{MR}_{63}\).
- First register \(\text{MR}_0\) acts as message tag.
- Subsequent registers contain:
  - Untyped words \((u)\), and
  - Typed words \((t)\)
    (e.g., map item, string item)

![Message Register Diagram]
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

“hh” indicates cacheability hints for the string.
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 Receipt

- 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.
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
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 $MR_0$
- 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
- 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 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
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  
  `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
  \texttt{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

<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>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>

from

MR₁
MR₂
MR₀
UTCB
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>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>$MR_1$</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>UT CB</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>$MR_1$</td>
</tr>
<tr>
<td></td>
<td>$MR_2$</td>
</tr>
<tr>
<td></td>
<td>$MR_0$</td>
</tr>
<tr>
<td></td>
<td>UT CB</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>UT CB</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>UT CB</td>
</tr>
</tbody>
</table>
IPC Register Encoding

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

**Sender Registers**
- EAX: destination
- ECX: timeouts
- EDX: receive specifier
- EBX: ~
- EBP: ~
- ESI: MR\_0
- EDI: UTCB

**Receiver Registers**
- from
- ~
- ~
- MR\_1
- MR\_2
- MR\_0
- UTCB