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.

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.
- 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 \(\rightarrow\) 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.
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 the thread is resumed.
- Resuming is simply switching back to the original stack.
- Preemption is easy.
- No conceptual difference between kernel mode and user mode.

```example
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 does one use a single kernel stack to support many threads?
- Issue: How are system calls that block handled?
  - Either continuations
  - Or stateless kernel (interrupt model)
    - Ford et al. Interface and Execution Models in the Fluke Kernel. Proc 3rd OSDI

```continuations
example(arg1, arg2) {
  P1(arg1, arg2);
  if (need_to_block) {
    save_context_in_TCB;
    thread_block(example_continue);
    /* NOT REACHED */
  } else {
    P2();
  }
  thread_syscall_return(SUCCESS);
}
example_continue() {
  recover_context_from_TCB;
  P2(recovered arg2);
  thread_syscall_return(SUCCESS);
}
```

### 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.

```continuations
example(arg1, arg2) {
  P1(arg1, arg2);
  if (need_to_block) {
    save_context_in_TCB;
    thread_block(example_continue);
    /* NOT REACHED */
  } else {
    P2();
  }
  thread_syscall_return(SUCCESS);
}
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 context 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.

```stateless
example(arg1, arg2) {
  P1(arg1, arg2);
  if (need_to_block) {
    // Modify user-state such that syscall is restarted when resources become available
    // Stack context is discarded
    thread_block();
    P2(recovered arg2);
  } else {
    P3();
  }
  thread_syscall_return(SUCCESS);
}
```

### IPC examples – Per thread stack

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

```ipc
msg_send_rcv(msg, option, send_size, recv_size, ...) {
  rc = msg_send(msg, option, send_size, ...);
  if (rc != SUCCESS)
    return rc;
  rc = msg_rcv(msg, option, recv_size, ...);
  return rc;
}
```

### IPC examples – Continuations

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

```ipc
msg_send_rcv(msg, option, send_size, recv_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.recv_size = recv_size;
  ...
  rc = msg_rcv(msg, option, recv_size, ..., msg_rcv_continue);
  return rc;
}
msg_rcv_continue(cur_thread) {
  msg = cur_thread->continuation.msg;
  option = cur_thread->continuation.option;
  recv_size = cur_thread->continuation.recv_size;
  ...
  rc = msg_rcv(msg, option, recv_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

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

Conclusion:

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

CPU

- enter kernel (IA32)

```
CPU
  esp
  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

User 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)
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

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
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.

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

align tcb's on a power of 2:

To find the starting address from the tcb:
- mov esp, [ebp].thr_esp
- mov [edi].thr_esp, esp
- mov esp, eax
- add sizeof tcb, eax
- mov [esp].ptr, esp
- popa
- add $4, esp
- iret

Thread switch (IA32)

push X
pusha
mov esp, ebp
and -sizeof tcb, ebp
dst 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, [esp].ptr
pops
add $4, esp
int 32

Thread B

switch esp
so that next enter kernel uses 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

Sysenter/Sysexit

- Emulate int instruction (ECX=USP, EDX=UIP)
  - mov esp0, esp
  - sub $20, esp
  - mov ecx, 16 esp
  - mov edx, 4 esp
  - mov $5, esp
- Emulate int instruction
  - mov 16 esp, ecx
  - mov 4 esp, edx
  - sti
  - sysexit

Trick:
- MSR points to esp0
- mov (esp), 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
- pop orange registers, return to new user thread
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

Thread Switching Overhead
- \( gr_0 \) 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
- \( fr_0 \) and \( fr_1 \) 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]

```
asm volatile ("
; ;

st8 [sp2] = r_bsp, 16
mov r_psr = psr
mov r_pr = pr
mov r_rnat = ar.rnat
st8 [sp2] = r_cfm, 16
st8 [sp1] = r_pfs, 16
1: alloc r_cfm = ar.pfs, 0, 0, 0, 0
// Store context into switch frame
br.call.sptk.many rp = 1f

mov rr[r0] = %[dest_rid]
// Set region id
mov "MKSTR(r_PHYS_TCB_ADDR)" = %[dest_tcb_phys]
mov "MKSTR(r_LOCAL_ID)" = %[dest_lid]
mov "MKSTR(r_GLOBAL_ID)" = %[dest_gid]
// Set thread ids and ksp for new thread
st8 [%[this_stack_ptr]] = sp
add sp1 = %[offset_pfs], sp
mov ar.rs = 0
// Make sure that stacked reg is not used
mov r_ip = 2f
mov r_rp = rp
mov ar.pfs = r_pfs
// Move context into general registers
r_psr = r22
r_pr = r21
r_unat = r20
r_cfm = r17
r_ip = r15
r_bsp = r14
"
;

movl r_ip = 2f
mov r_rp = rp
mov ar.pfs = r_pfs
// Move context into general registers
"
;

r_psr = r22
r_pr = r21
r_unat = r20
r_cfm = r17
r_ip = r15
r_bsp = r14
"
;

аш爱你
movl r_ip = 2f
mov r_rp = rp
mov ar.pfs = r_pfs
// Move context into general registers
"
;

r_psr = r22
r_pr = r21
r_unat = r20
r_cfm = r17
r_ip = r15
r_bsp = r14
"
;

аш爱你
movl r_ip = 2f
mov r_rp = rp
mov ar.pfs = r_pfs
// Move context into general registers
"
;

r_psr = r22
r_pr = r21
r_unat = r20
r_cfm = r17
r_ip = r15
r_bsp = r14
"
;

аш爱你
movl r_ip = 2f
mov r_rp = rp
mov ar.pfs = r_pfs
// Move context into general registers
"
;

r_psr = r22
r_pr = r21
r_unat = r20
r_cfm = r17
r_ip = r15
r_bsp = r14
"
;

аш爱你
movl r_ip = 2f
mov r_rp = rp
mov ar.pfs = r_pfs
// Move context into general registers
"
;

r_psr = r22
r_pr = r21
r_unat = r20
r_cfm = r17
r_ip = r15
r_bsp = r14
"
;

аш爱你
movl r_ip = 2f
mov r_rp = rp
mov ar.pfs = r_pfs
// Move context into general registers
"
;

r_psr = r22
r_pr = r21
r_unat = r20
r_cfm = r17
r_ip = r15
r_bsp = r14
"
;

аш爱你
movl r_ip = 2f
mov r_rp = rp
mov ar.pfs = r_pfs
// Move context into general registers
"
;

r_psr = r22
r_pr = r21
r_unat = r20
r_cfm = r17
r_ip = r15
r_bsp = r14
"
;

аш爱你
movl r_ip = 2f
mov r_rp = rp
mov ar.pfs = r_pfs
// Move context into general registers
"
;

r_psr = r22
r_pr = r21
r_unat = r20
r_cfm = r17
r_ip = r15
r_bsp = r14
"
;

аш爱你
movl r_ip = 2f
mov r_rp = rp
mov ar.pfs = r_pfs
// Move context into general registers
"
;

r_psr = r22
r_pr = r21
r_unat = r20
r_cfm = r17
r_ip = r15
r_bsp = r14
"
;

аш爱你
movl r_ip = 2f
mov r_rp = rp
mov ar.pfs = r_pfs
// Move context into general registers
"
;

r_psr = r22
r_pr = r21
r_unat = r20
r_cfm = r17
r_ip = r15
r_bsp = r14
"
;

аш爱你
movl r_ip = 2f
mov r_rp = rp
mov ar.pfs = r_pfs
// Move context into general registers
"
;

r_psr = r22
r_pr = r21
r_unat = r20
r_cfm = r17
r_ip = r15
r_bsp = r14
"
;

аш爱你
movl r_ip = 2f
mov r_rp = rp
mov ar.pfs = r_pfs
// Move context into general registers
"
;

r_psr = r22
r_pr = r21
r_unat = r20
r_cfm = r17
r_ip = r15
r_bsp = r14
"
;

аш爱你
movl r_ip = 2f
mov r_rp = rp
mov ar.pfs = r_pfs
// Move context into general registers
"
;

r_psr = r22
r_pr = r21
r_unat = r20
r_cfm = r17
r_ip = r15
r_bsp = r14
"
;

аш爱你
movl r_ip = 2f
mov r_rp = rp
mov ar.pfs = r_pfs
// Move context into general registers
"
;

r_psr = r22
r_pr = r21
r_unat = r20
r_cfm = r17
r_ip = r15
r_bsp = r14
"
;

аш爱你
movl r_ip = 2f
mov r_rp = rp
mov ar.pfs = r_pfs
// Move context into general registers
"
;

r_psr = r22
r_pr = r21
r_unat = r20
r_cfm = r17
r_ip = r15
r_bsp = r14
"
;

аш爱你
movl r_ip = 2f
mov r_rp = rp
mov ar.pfs = r_pfs
// Move context into general registers
"
;

r_psr = r22
r_pr = r21
r_unat = r20
r_cfm = r17
r_ip = r15
r_bsp = r14
"
;

аш爱你
movl r_ip = 2f
mov r_rp = rp
mov ar.pfs = r_pfs
// Move context into general registers
"
;

r_psr = r22
r_pr = r21
r_unat = r20
r_cfm = r17
r_ip = r15
r_bsp = r14
"
;

аш爱你
Exception Handling

- 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
  - Store remaining registers

Exception Handling

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

Kernel Entry

- Kernel entry by exception is slow
- Must flush instruction pipeline
- IA-64 provides an \texttt{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

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

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

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
Limitations
- 32-bit, virt tcb, entire PM
- number of threads
- physical mem size

Physical Memory
- Kernel uses physical for:
  - Application’s Page tables
  - Kernel memory
  - Kernel debugger
- Issue occurs only when kernel accesses physical memory
  - Limit valid physical range to remap size (256M)
  - Or...

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

Kernel Debugger (not performance critical)
- Walk page table in software
- Remap on demand (4MB)
- Optimization: check if already mapped

FPU Context Switching
- Strict switching
  - Thread switch:
    - Store current thread’s FPU state
    - Load new thread’s FPU state
- Extremely expensive
  - IA-32’s full SSE2 state is 512 Bytes
  - IA-64’s floating point state is ~1.5KB
- May not even be required
  - Threads do not always use FPU

Lazy FPU switching
- Lock FPU on thread switch
- Unlock at first use – exception handled by kernel
  - Unlock FPU
  - If fpu_owner != current
    - Save current state to fpu_owner
    - Load new state from current
    - fpu_owner := current
  - FPU
  - Kernel
  - pacman()
What IPC primitives do we need to communicate?

- Send to (a specified thread)
- Receive from (a specified thread)
- Two threads can communicate
- Can create specific protocols without fear of interference from other threads
- Other threads block until it's their turn
- Problem:
  - How to communicate with a thread unknown a priori (e.g., a server's clients)

Scenario:

- A client thread sends a message to a server expecting a response.
- The server replies expecting the client thread to be ready to receive.
- Issue: The client might be preempted between the send to and receive from.

What message types are appropriate?

- Register
  - Short messages we hope to make fast by avoiding memory access to transfer the message during IPC
  - Guaranteed to avoid user-level page faults during IPC
- Direct strings (optional)
- Indirect strings (optional)
- Map pages (optional)
  - Messages that map pages from sender to receiver

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

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).
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
- Attack through receiver’s pager:
  - snd timeout, rcv timeout
  - snd-pf timeout
  - rcv-pf timeout
  - specified by sender

IPC - API
- Attack through sender’s pager:
  - snd timeout, rcv timeout
  - snd-pf / rcv-pf timeout
  - specified by receiver

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

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

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+1</th>
<th>m+288</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>284.3356</td>
<td>64851.04728</td>
</tr>
<tr>
<td>1</td>
<td>87.10864</td>
<td>17112.76032</td>
</tr>
<tr>
<td>2</td>
<td>16.77216</td>
<td>4278.19008</td>
</tr>
<tr>
<td>3</td>
<td>4.194304</td>
<td>1069.54752</td>
</tr>
<tr>
<td>4</td>
<td>1.048576</td>
<td>267.38688</td>
</tr>
<tr>
<td>5</td>
<td>0.262144</td>
<td>66.84672</td>
</tr>
<tr>
<td>6</td>
<td>0.065536</td>
<td>16.71168</td>
</tr>
<tr>
<td>7</td>
<td>0.016384</td>
<td>4.17792</td>
</tr>
<tr>
<td>8</td>
<td>0.004096</td>
<td>1.04448</td>
</tr>
<tr>
<td>9</td>
<td>0.001024</td>
<td>0.26112</td>
</tr>
<tr>
<td>10</td>
<td>0.000256</td>
<td>0.06528</td>
</tr>
<tr>
<td>11</td>
<td>0.000064</td>
<td>0.01632</td>
</tr>
<tr>
<td>12</td>
<td>0.000016</td>
<td>0.00408</td>
</tr>
<tr>
<td>13</td>
<td>0.000004</td>
<td>0.00102</td>
</tr>
<tr>
<td>14</td>
<td>0.000001</td>
<td>0.000255</td>
</tr>
</tbody>
</table>

IPC - API

- Timeouts (V 2, V X.0)
  - snd timeout, rcv timeout
  - snd-pf / rcv-pf timeout
    - 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

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

Timeouts (V X.2, V 4)
- snd timeout, rcv timeout, xfer timeout snd, xfer timeout rcv
  - relative timeout values
    - 0
    - infinite
  - 1us … 610h (log)
  - absolute timeout values
    - 0
    - infinite

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

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

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

Call-reply example
- Thread A
  - pre
  - IPC call
  - post
- Thread B
  - pre
  - IPC reply & wait
  - post
Send and Receive Encoding

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

- 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 desiring IPC
- Thread ID to deseed 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 (HR0, ..., HRn)
- First register (HR0) acts as message tag
- Subsequent registers contain:
  - Untyped words (u), and
  - Typed words (t)
- (e.g., map item, string item)
- Freely available (e.g., request type)
- Number of typed words
- Number of untyped words
- Various IPC flags
- Message Tag
- Label
Message construction

- Messages are stored in registers (MR0, ..., MR63).
- First register (MR0) acts as message tag.
- Subsequent registers contain:
  - Untyped words (u), and MRi
  - Typed words (t) (e.g., map item, string item)

Semantics will be explained during memory management lecture.

Map and Grant items

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

String items

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

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 decepting IPC
- Thread ID to decoy as
- Intended receiver of decepted 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
- 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 BR0 ... BR33 (buffer registers 0 ... 33)
  - Buffer registers specify
    - Receive strings
    - Receive window for mappings

Receiving messages

- Receiver buffers are specified in registers (BR0 ... BR33)
  - First BR (BR0) contains "Acceptor"
    - May specify receive window (if not nil-page)
    - May indicate presence of receive strings/buffers (if s-bit set)

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
- 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

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
- Message and buffer registers
- Most architectures do not have 64+34 spare registers
- What about predefined memory locations?
- Must be thread local

Virtual Registers

- What about predefined memory locations?
- Must be thread local

IPC Result

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

IPC Redirection

- Redirection/deceiting IPC flagged by bit in the message tag
- Fast check
- When redirection bit set
- Thread ID to deceit as and intended receiver ID stored in predefined memory locations
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
- UTCBs can not be paginated
- No page faults
- Registers always accessible

Preserved by
switching UTCB
on context switch

Preserved by
kernel during
context switch

VEBX
VEBP
VESI
EPBX
EBP
ESI

Physical Registers

Virtual Registers

UTCB

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

<table>
<thead>
<tr>
<th>EAX</th>
<th>destination</th>
<th>from</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>timeouts</td>
<td></td>
</tr>
<tr>
<td>EDS</td>
<td>receive specifier</td>
<td></td>
</tr>
<tr>
<td>ESI</td>
<td>MR1</td>
<td>MR0</td>
</tr>
<tr>
<td>ESI</td>
<td>MR2</td>
<td>MR1</td>
</tr>
<tr>
<td>ESI</td>
<td>MR3</td>
<td>MR2</td>
</tr>
<tr>
<td>ESI</td>
<td>MR4</td>
<td>MR3</td>
</tr>
<tr>
<td>EDI</td>
<td>UTCB</td>
<td>UTCB</td>
</tr>
</tbody>
</table>

Free Up Registers for Temporary Values

- Kernel needs registers for temporary values
- MR1 and MR2 are the only registers that the kernel may not need

<table>
<thead>
<tr>
<th>EAX</th>
<th>destination</th>
<th>from</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>timeouts</td>
<td></td>
</tr>
<tr>
<td>EDS</td>
<td>receive specifier</td>
<td></td>
</tr>
<tr>
<td>ESI</td>
<td>MR1</td>
<td>MR0</td>
</tr>
<tr>
<td>ESI</td>
<td>MR2</td>
<td>MR1</td>
</tr>
<tr>
<td>ESI</td>
<td>MR3</td>
<td>MR2</td>
</tr>
<tr>
<td>ESI</td>
<td>MR4</td>
<td>MR3</td>
</tr>
<tr>
<td>EDI</td>
<td>UTCB</td>
<td>UTCB</td>
</tr>
</tbody>
</table>
Free Up Registers for Temporary Values

- `S`ys`e`x`i`t instruction requires:
  - ECX = user IP
  - EDX = user SP

<table>
<thead>
<tr>
<th>EAX</th>
<th>destination</th>
<th>from</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECX</td>
<td>timeouts</td>
<td></td>
</tr>
<tr>
<td>EDX</td>
<td>receive specifier</td>
<td></td>
</tr>
<tr>
<td>ESI</td>
<td>MR0</td>
<td>MR1</td>
</tr>
<tr>
<td>EDI</td>
<td>UTCB</td>
<td>UTCB</td>
</tr>
</tbody>
</table>

IPC Register Encoding

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

<table>
<thead>
<tr>
<th>EAX</th>
<th>destination</th>
<th>from</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECX</td>
<td>timeouts</td>
<td></td>
</tr>
<tr>
<td>EDX</td>
<td>receive specifier</td>
<td></td>
</tr>
<tr>
<td>ESI</td>
<td>MR0</td>
<td>MR1</td>
</tr>
<tr>
<td>EDI</td>
<td>MR2</td>
<td>MR3</td>
</tr>
<tr>
<td>ESI</td>
<td>MR4</td>
<td>MR5</td>
</tr>
<tr>
<td>EDI</td>
<td>MR6</td>
<td>MR7</td>
</tr>
<tr>
<td>ESI</td>
<td>MR8</td>
<td>MR9</td>
</tr>
<tr>
<td>EDI</td>
<td>MR10</td>
<td>MR11</td>
</tr>
</tbody>
</table>

What About IA-64?

- All other registers are undefined