Microkernel Construction
Threads and Address Spaces

Nils Asmussen

04/16/2020
Outline

- Threads
  - Definition
  - Concepts in NOVA
  - Thread Switch in NOVA
- FPU Handling
- Address Spaces
What is a Thread?

- An independent flow of control inside an address space
- Communicates with other threads using IPC
- Characterized by a set of registers and the thread state
- Dispatched by the kernel according to a defined schedule
What is a Thread?

- An independent flow of control inside an address space
- Communicates with other threads using IPC
- Characterized by a set of registers and the thread state
- Dispatched by the kernel according to a defined schedule
- Each thread is bound to one core at a time
- Only one thread per core is running at one point in time
- With $n$ cores, $n$ threads can run at once
- All other threads are inactive, waiting inside the kernel
Implementation in NOVA

Execution Context:
- Register state
- Continuation
- Address Space (PD)
- UTCB (message buffer)
- IPC partner
- FPU state
- prev/next pointer

Scheduling Context:
- Execution Context
- Priority
- Budget
- Remaining budget
- prev/next pointer
Thread Variants

Global Thread
- Needs an scheduling context, i.e., CPU time, to execute
- Causes exception on startup to let creator set register state

Local Thread
- Has no scheduling context
- Are only used to handle portal calls
- Waits in the kernel until someone called an associated portal
• A portal is an IPC endpoint
• Executed by local threads
• CPU time is donated from caller
• Called via system call
• Message is transferred from sender UTCB to receiver UTCB
Thread Switch: Conventional
Thread Switch: Conventional

- User Stack
- Address Space
- User Stack

- Kernel
- User Stack
- Kernel
- Stack

- User Stack
- Kernel
- User Stack

- Regs
- Kernel Stack
- Kernel Stack

- CPU
Thread Switch: Conventional
Thread Switch: Conventional
Thread Switch: Continuation Style

Address Space

User Stack A

User Stack B

Kernel Stack

User Stack

Kernel

User Stack

ECA ECB

EC_A

Kernel Stack

Kernel

CPU

EC_B
Thread Switch: Continuation Style

![Diagram of thread switch]

- User Stack
- User Stack_A
- User Stack_B
- Kernel Stack
- Kernel
- CPU
- Address Space
- Address Space
- Regs
- EC_A
- EC_B

[Diagram showing the transition between user and kernel stacks during thread switching.]
Thread Switch: Continuation Style
Thread Switch: Continuation Style

```
Address
Space
Address
Space
Kernel
CPU
User
Stack
Stack
User
Stack
Kernel
Stack
User
Stack
User
Stack
```

```
EC_A
EC_B
```

```
User Stack_A
User Stack_B
```

```
CPU
```

```
Kernel
```

```
Stack
```

```
Stack
```
Thread Switch: In-Kernel Switch

- Traditional kernels save/restore the current CPU state
- Each thread has own stack → stack frames are kept
- In NOVA, stack frames and CPU state are lost

Part of `sys_call`

```c
current->cont = ret_user_sysexit;
current->set_partner (ec);
ec->cont = recv_user;
ec->regs.set_ip (pt->ip);
ec->regs.set_pt (pt->id);
ec->make_current();
```
void Ec::make_current()
{
    current = this;
    Tss::run.sp0 = reinterpret_cast<mword>(exc_regs());
    pd->make_current();
    asm volatile (
        "mov %0, %%rsp;"
      "jmp *%1;"
      :
      : "g" (CPU_LOCAL_STCK + PAGE_SIZE),
        "rm" (cont) : "memory"
    );
    UNREACHED;
}
Threads

FPU Handling
  - General Idea
  - x86 Details
  - Implementation in NOVA

Address Spaces

- CPU has dedicated functional units for FP computations
- Are accessed with specific instructions
- Have their own state, which is large (512 bytes)
- Each thread has its own FPU state
- → Save/restore FPU on each context switch is expensive
- → However, many OSes on x86 today save it on every switch (vector instructions, LazyFPU vulnerability)
FPU Switch: General Idea

- We want to know if/when a thread uses the FPU
- We only want to save the FPU state if it has been modified
- We don’t want to save the FPU state when switching from a thread that used the FPU to a thread that is not going to use the FPU and then later restore the old (unmodified) FPU state
If CR0.TS (Task Switched) flag is set, FPU instructions are not executed, but cause #NM exception.
Handling the #NM exception

```c
void handle_exc_nm() {
    CR0.TS = 0;
    hzd |= HZD_FPU;
    if (current == fpowner)
        return;
    if (fpowner)
        fpowner->fpu->save();
    if (current->fpu)
        current->fpu->load();
    else {
        current->fpu = new Fpu;
        Fpu::init();
    }
    fpowner = current;
}
```

Before leaving to user

```c
void handle_hazards() {
    if ((hzd & HZD_FPU) &&
        current != fpowner) {
        CR0.TS = 1;
        hzd &= ~HZD_FPU;
    }
}
```
- Threads
- FPU Handling
- **Address Spaces**
  - Virtual Memory Recap
  - x86 Data Structures
  - x86 TLB
  - Implementation in NOVA
Virtual Memory

Virtual Memory

Physical Memory

16
14
11
10

16
11
10

12
17
15
11
10

18
17
16
15
14
13
12
11
10

10 10
11
12
13
14
15
10 10
11 11 11
14
16
15
17
18

Physical Memory
Translation of linear to physical addresses
Done by memory management unit (MMU)
Hardware defines data structures:
- Page Directory Base Register (CR3)
- Page Directory (PDIR)
  - 4KiB page containing 1024 page directory entries (PDEs)
- Page Table (PTAB)
  - 4KiB page containing 1024 page table entries (PTEs)
Paging data structures use physical addresses
Address Translation: 4 KiB pages (x86)

32 Bit Linear Address

Directory | Table | Offset

Page Directory | Page Table | Physical Address

Page Directory Entry | Page Table Entry

CR3 (PDBR)

1024 PDE * 1024 PTE = $2^{20}$ Pages
Address Translation: 4 MiB superpages (x86)

1024 PDE = 2^{10} Superpages
PDEs and PTEs (x86)

<table>
<thead>
<tr>
<th>Page Table Base Address</th>
<th>Avail (G)</th>
<th>P</th>
<th>S</th>
<th>0</th>
<th>A</th>
<th>P</th>
<th>C</th>
<th>D</th>
<th>P</th>
<th>W</th>
<th>T</th>
<th>U</th>
<th>K</th>
<th>R</th>
<th>W</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Page Base Address</th>
<th>Reserved</th>
<th>PAT</th>
<th>Avail G</th>
<th>P</th>
<th>S</th>
<th>D</th>
<th>A</th>
<th>P</th>
<th>C</th>
<th>D</th>
<th>P</th>
<th>W</th>
<th>T</th>
<th>U</th>
<th>K</th>
<th>R</th>
<th>W</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Page Base Address</th>
<th>Avail G</th>
<th>PAT</th>
<th>D</th>
<th>A</th>
<th>P</th>
<th>C</th>
<th>D</th>
<th>P</th>
<th>W</th>
<th>T</th>
<th>U</th>
<th>K</th>
<th>R</th>
<th>W</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Translation Lookaside Buffer (x86)

- Caches recent linear-to-physical translations
- Avoids expensive page-table walk
- Must be kept consistent with the page tables by the OS
- No TLB coherency protocol
- On modifications, OS must flush relevant TLB entries
- TLB flush triggered by CR3 reload or INVLPG instruction
- No TLB flush required when upgrading page attributes
- CR3 reload does not flush global pages
- TLB shootdowns for page tables active on other cores
  - Expensive signaling and synchronization
  - Inter-Processor-Interrupt (IPI)
Memory space of a protection domain

class Space_mem : public Space {
public:
    Hpt hpt;                       // master page table
    Hpt loc[NUM_CPU];              // per-core PTs; synced from master
    Dpt dpt;                      // DMA PT for IOMMU
    union {
        Ept ept;                   // nested PT for Intel (VMX)
        Hpt npt;                   // nested PT for AMD (SVM)
    };
};
Generic page table entry handling

template<typename P, typename E, unsigned L, unsigned B>
class Pte {
    E val;

    P *walk (E virt, unsigned long level, bool add);
    size_t lookup (E virt, Paddr &phys, mword &attr);
    void update (E virt, mword size, E phys,
                 mword attr, bool add);
};

class Hpt : public Pte<Hpt, uint32, 2, 10>;
class Dpt : public Pte<Dpt, uint64, 4, 9>;
class Ept : public Pte<Ept, uint64, 4, 9>;}
Implementation in NOVA – TLB shootdowns

- `cpus` mask stores cores that use the address space
- `core-bit in cpus` is set as soon as Ec is started on a core
- `htlb` is set to `cpus` on permission downgrades
- TLB shootdown sends IPI to all cores in `htlb`
- IPI causes a scheduling to set CR3
- ...and clear core-bit in `htlb`
<table>
<thead>
<tr>
<th>Start</th>
<th>End</th>
<th>core-local</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000_0000</td>
<td>BFFF_FFFF</td>
<td>No</td>
<td>User space</td>
</tr>
<tr>
<td>C000_0000</td>
<td>CFBF_FFFF</td>
<td>No</td>
<td>Code, static data, heap</td>
</tr>
<tr>
<td>CFFF_D000</td>
<td>CFFF_DFFF</td>
<td>Yes</td>
<td>Kernel stack</td>
</tr>
<tr>
<td>CFFF_E000</td>
<td>CFFF_EFFF</td>
<td>Yes</td>
<td>LAPIC</td>
</tr>
<tr>
<td>CFFF_F000</td>
<td>CFFF_FFFF</td>
<td>Yes</td>
<td>Kernel data</td>
</tr>
<tr>
<td>D000_0000</td>
<td>D000_1FFF</td>
<td>No</td>
<td>I/O Bitmap</td>
</tr>
<tr>
<td>E000_0000</td>
<td>FFFF_FFFF</td>
<td>No</td>
<td>Capabilities</td>
</tr>
</tbody>
</table>