CS202

Agenda

1. Finish Virtual Memory

2. Context Switching
   - Demand Paging
     LRU Mechanism
   - Page Replacement Policy
     - What Page to Evict
   - FIFO, MIN, LRU
   - Clock
Small Physical
Bad access pattern
Lot of allocated virtual memory
Beladi's Anomaly

FIFO

\[ \begin{align*}
P_0 & : A D E h \\
P_1 & : A B C h \\
P_2 & : C B h D \\
P_3 & : A h E D \\
P_4 & : B h A E \\
P_5 & : C B C
\end{align*} \]

Context Switch

Thread 0

\[ \begin{align*}
\text{Registers} & : \%rip \rightarrow \text{Next thing to exec} \\
\%rsp & \rightarrow \%rbp \\
\%rbp & \cdots
\end{align*} \]

Process Address Space

\[ \text{Stack} \rightarrow \text{Ret addr} \rightarrow \text{Ret addr} \]

Thread 1

\[ \begin{align*}
\text{Registers} & : \%rip \rightarrow \text{Next thing to exec} \\
\%rsp & \rightarrow \%rbp \\
\%rbp & \cdots
\end{align*} \]

\[ \text{Stack} \rightarrow \text{Ret addr} \rightarrow \text{Ret addr} \]
Switching & Scheduling in Weensy OS
User Level Threading

Where?
- Go
- Rust
...

How?
Somewhat similar to Weenys OS but
- No changing CR3
- Cannot access all registers in CR3
(switch to handout)

S S S S S S

Sched

Library

S S
Core i7 Page Table Translation

![Diagram of page table translation for the Core i7 processor.]

- **CR3**: Physical address of L1 PT
- **VPN**: Virtual page numbers
- **L1 PT**: Page global directory
- **L2 PT**: Page upper directory
- **L3 PT**: Page middle directory
- **L4 PT**: Page table
- **VPN**: Virtual page number
- **L1 PTE**: L1 page table entry
- **L2 PTE**: L2 page table entry
- **L3 PTE**: L3 page table entry
- **L4 PTE**: L4 page table entry
- **VPN**: 9
- **L1 PT**: 9
- **L2 PT**: 9
- **L3 PT**: 9
- **L4 PT**: 12
- **VPO**: 12
- **PPN**: 40
- **PPO**: 12
- **VPN 1**: 9
- **VPN 2**: 9
- **VPN 3**: 9
- **VPN 4**: 9
- **VPO**: 12
- **Physical address of L1 PT**: 40
- **Offset into physical and virtual page**: 40
- **Physical address of page**: 40
- **VPN**: 9
- **L1 PT**: 9
- **L2 PT**: 9
- **L3 PT**: 9
- **L4 PT**: 12
- **VPO**: 12
- **Physical address**: 40
- **VPN**: 9
- **L1 PT**: 9
- **L2 PT**: 9
- **L3 PT**: 9
- **L4 PT**: 12
- **VPO**: 12
- **Physical address of page**: 40
- **VPN**: 9
- **L1 PT**: 9
- **L2 PT**: 9
- **L3 PT**: 9
- **L4 PT**: 12
- **VPO**: 12
- **Physical address**: 40

Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition
Core i7 Level 4 Page Table Entries

Each entry references a 4K child page. Significant fields:

P: Child page is present in memory (1) or not (0)

R/W: Read-only or read-write access permission for this page

U/S: User or supervisor mode access

WT: Write-through or write-back cache policy for this page

A: Reference bit (set by MMU on reads and writes, cleared by software)

D: Dirty bit (set by MMU on writes, cleared by software)

Page physical base address: 40 most significant bits of physical page address (forces pages to be 4KB aligned)

XD: Disable or enable instruction fetches from this page.
Core i7 Level 1-3 Page Table Entries

<table>
<thead>
<tr>
<th>63</th>
<th>62</th>
<th>52</th>
<th>51</th>
<th>12</th>
<th>11</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>XD</td>
<td>Unused</td>
<td>Page table physical base address</td>
<td>Unused</td>
<td>G</td>
<td>PS</td>
<td>A</td>
<td>CD</td>
<td>WT</td>
<td>U/S</td>
<td>R/W</td>
<td>P=1</td>
<td></td>
<td></td>
<td>P=0</td>
<td></td>
</tr>
</tbody>
</table>

Available for OS

Each entry references a 4K child page table. Significant fields:

**P**: Child page table present in physical memory (1) or not (0).

**R/W**: Read-only or read-write access access permission for all reachable pages.

**U/S**: user or supervisor (kernel) mode access permission for all reachable pages.

**WT**: Write-through or write-back cache policy for the child page table.

**A**: Reference bit (set by MMU on reads and writes, cleared by software).

**PS**: Page size: if bit set, we have 2 MB or 1 GB pages (bit can be set in Level 2 and 3 PTEs only).

**Page table physical base address**: 40 most significant bits of physical page table address (forces page tables to be 4KB aligned)

**XD**: Disable or enable instruction fetches from all pages reachable from this PTE.
### Figure 4-12. Page-Fault Error Code

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>P: 0 The fault was caused by a non-present page. 1 The fault was caused by a page-level protection violation.</td>
</tr>
<tr>
<td>15</td>
<td>W/R: 0 The access causing the fault was a read. 1 The access causing the fault was a write.</td>
</tr>
<tr>
<td>5</td>
<td>U/S: 0 A supervisor-mode access caused the fault. 1 A user-mode access caused the fault.</td>
</tr>
<tr>
<td>4</td>
<td>RSVD: 0 The fault was not caused by reserved bit violation. 1 The fault was caused by a reserved bit set to 1 in some paging-structure entry.</td>
</tr>
<tr>
<td>3</td>
<td>I/D: 0 The fault was not caused by an instruction fetch. 1 The fault was caused by an instruction fetch.</td>
</tr>
<tr>
<td>2</td>
<td>PK: 0 The fault was not caused by protection keys. 1 There was a protection-key violation.</td>
</tr>
<tr>
<td>1</td>
<td>SGX: 0 The fault is not related to SGX. 1 The fault resulted from violation of SGX-specific access-control requirements.</td>
</tr>
</tbody>
</table>
End-to-end Core i7 Address Translation

CPU

Virtual address (VA)

VPN

VPO

36

12

TLBT

TLBI

TLB

hit

miss

L1 TLB (16 sets, 4 entries/set)

VPN1

VPN2

VPN3

VPN4

9

9

9

9

CR3

PTE

Page tables

VPN1

VPN2

VPN3

VPN4

32

4

32/64

Result

L2, L3, and main memory

L1 hit

L1 miss

L1 d-cache
(64 sets, 8 lines/set)

Physical address (PA)

CR3

PTE

Page tables

VPN1

VPN2

VPN3

VPN4

32

4

32/64

Result

L2, L3, and main memory

L1 hit

L1 miss

L1 d-cache
(64 sets, 8 lines/set)

Physical address (PA)

CPU

Virtual address (VA)

VPN

VPO

36

12

TLBT

TLBI

TLB

hit

miss

L1 TLB (16 sets, 4 entries/set)

VPN1

VPN2

VPN3

VPN4

9

9

9

9

CR3

PTE

Page tables

VPN1

VPN2

VPN3

VPN4

32

4

32/64

Result

L2, L3, and main memory

L1 hit

L1 miss

L1 d-cache
(64 sets, 8 lines/set)

Physical address (PA)
Cute Trick for Speeding Up L1 Access

**Observation**

- Bits that determine CI identical in virtual and physical address
- Can index into cache while address translation taking place
- Cache carefully sized to make this possible: 64 sets, 64-byte cache blocks
- Means 6 bits for cache index, 6 for cache offset
- That’s 12 bits; matches \( VPO, PPO \) → One reason pages are \( 2^{12} \) bits = 4 KB
Virtual Address Space of a Linux Process

- Process-specific data structs (ptables, task and mm structs, kernel stack)
- Physical memory
- Kernel code and data
- User stack
- Memory mapped region for shared libraries
- Runtime heap (malloc)
- Uninitialized data (.bss)
- Initialized data (.data)
- Program text (.text)

Identical for each process

Kernel virtual memory

Process virtual memory

%rsp
brk
0x00400000
Return path

File: kernel.c line 26

// schedule
// Pick the next process to run and then run it.
// If there are no runnable processes, spins forever.

void schedule(void) {
    pid_t pid = current->p_pid;
    while (1) {
        pid = (pid + 1) % NPROC;
        if (processes[pid].p_state == P_RUNNABLE) {
            run(&processes[pid]);
        }
        // If Control-C was typed, exit the virtual machine.
        check_keyboard();
    }
}

// run(p)
// Run process `p`. This means reloading all the registers from
// `p->p_registers` using the `popal`, `popl`, and `iret` instructions.
// As a side effect, sets `current = p`.

void run(proc* p) {
    assert(p->p_state == P_RUNNABLE);
    current = p;

    // Load the process's current pagetable.
    set_pagetable(p->p_pagetable);

    // This function is defined in k-exception.S. It restores the process's
    // registers then jumps back to user mode.
    exception_return(&p->p_registers);

    spinloop: goto spinloop;  // should never get here
}
File: k-exception.S line 158

exception_return:
  movq %rdi, %rsp
  popq %rax
  popq %rcx
  popq %rdx
  popq %rbx
  popq %rbp
  popq %rsi
  popq %rdi
  popq %r8
  popq %r9
  popq %r10
  popq %r11
  popq %r12
  popq %r13
  popq %r14
  popq %r15
  popq %fs
  popq %gs
  addq $16, %rsp
  iretq
Entry path

File: k-exception.S line 134

generic_exception_handler:

```
pushq %gs
pushq %fs
pushq %r15
pushq %r14
pushq %r13
pushq %r12
pushq %r11
pushq %r10
pushq %r9
pushq %r8
pushq %rdi
pushq %rsi
pushq %rbp
pushq %rbx
pushq %rdx
pushq %rcx
pushq %rax
movq %rsp, %rdi
call exception
# `exception` should never return.
```

File kernel.c line

```
void exception(x86_64_registers* reg) ...
typedef struct x86_64_registers {
    uint64_t reg_rax;
    uint64_t reg_rcx;
    uint64_t reg_rdx;
    uint64_t reg_rbx;
    uint64_t reg_rbp;
    uint64_t reg_rsi;
    uint64_t reg_rdi;
    uint64_t reg_r8;
    uint64_t reg_r9;
    uint64_t reg_r10;
    uint64_t reg_r11;
    uint64_t reg_r12;
    uint64_t reg_r13;
    uint64_t reg_r14;
    uint64_t reg_r15;
    uint64_t reg_fs;
    uint64_t reg_gs;
    uint64_t reg_intno;     // (3) Interrupt number and error
    uint64_t reg_err;      // code (optional; supplied by x86
                         // interrupt mechanism)
    uint64_t reg_rip;      // (4) Task status: instruction pointer,
    uint16_t reg_cs;       // code segment, flags, stack
    uint16_t reg_padding2[3]; // in the order required by `iret`
    uint64_t reg_rflags;
    uint64_t reg_rsp;
    uint16_t reg_ss;
    uint16_t reg_padding3[3];
} x86_64_registers;
1. User-level threads and swtch()

We'll study this in the context of user-level threads.

Per-thread state in thread control block:

```c
typedef struct tcb {
    unsigned long saved_rsp;    /* Stack pointer of thread */
    char *t_stack;       /* Bottom of thread's stack */
    /* ... */
};
```

Machine-dependent thread initialization function:

```c
void thread_init(tcb **t, void (*fn) (void *), void *arg);
```

Machine-dependent thread-switch function:

```c
void swtch(tcb *current, tcb *next);
```

Implementation of swtch(current, next):

```c
# gcc x86-64 calling convention:
# on entering swtch():
#  register %rdi holds first argument to the function ("current")
#  register %rsi holds second argument to the function ("next")
# Save call-preserved (aka "callee-saved") regs of 'current'
pushq %rbp
pushq %rbx
pushq %r12
pushq %r13
pushq %r14
pushq %r15

# store old stack pointer, for when we swtch() back to "current" later
movq (%rdi), %rsp
# %rsp = %rdi->saved_rsp

# Restore call-preserved (aka "callee-saved") regs of 'next'
popq %r15
popq %r14
popq %r13
popq %r12
popq %rbx
popq %rbp

# Resume execution, from where "next" was when it last entered swtch()
ret
```

2. Example use of swtch(): the yield() call.

A thread is going about its business and decides that it’s executed for long enough. So it calls yield(). Conceptually, the overall system needs to now choose another thread, and run it:

```c
void yield() {
    tcb* next = pick_next_thread(); /* get a runnable thread */
    tcb* current = get_current_thread();
    swtch(current, next);
    /* when 'current' is later rescheduled, it starts from here */
}
```

3. How do context switches interact with I/O calls?

This assumes a user-level threading package.

The thread calls something like "fake_blocking_read()". This looks to the _thread_ as though the call blocks, but in reality, the call is not blocking:

```c
int fake_blocking_read(int fd, char* buf, int num) {
    int nread = -1;
    while (nread == -1) {
        /* this is a non-blocking read() syscall */
        nread = read(fd, buf, num);
        if (nread == -1 && errno == EAGAIN) {
            /* read would block. so let another thread run */
            yield();
        }
    }
    return nread;
}
```