6.181: Using Virtual Memory

Adam Belay <abelay@mit.edu>
Today’s focus

Cool things you can do with virtual memory:

1. Virtual memory recap
2. Lazy page allocation
3. Better performance/efficiency
   • E.g. One zero-filled page
   • E.g. Copy-on-write w/ fork()
4. New features
   • E.g. Memory-mapped files
Recap: Memory’s many layers of abstraction

Focus today

Hardware

CPU/OS

Compiler/Library

Language/Runtime

RAM and I/O

Address Spaces

Stack and Heap

Garbage Collection, ARC, or Smart Pointers
Recap: Key ideas for address spaces

1. Address spaces can have holes
2. Address spaces can have permissions
3. Combine RAM and devices
4. Virtual memory (today)
5. Cache coherence and consistency (later)
Recap: Process isolation

• Primary goal: Isolation – each process has its own address space
• But… virtual memory provides a level of indirection that allows the kernel to do cool stuff
Page table entries (PTE)

<table>
<thead>
<tr>
<th>63</th>
<th>54</th>
<th>53</th>
<th>28</th>
<th>27</th>
<th>19</th>
<th>18</th>
<th>10</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>10</td>
<td>26</td>
<td>9</td>
<td>9</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.18: Sv39 page table entry.

Some important bits:

- **Physical page number (PPN):** Identifies 44-bit physical page location; MMU replaces virtual bits with these physical bits
- **U:** If set, userspace can access this virtual address
- **W:** writeable, **R:** readable, **X:** executable
- **V:** If set, an entry for this virtual address exists
- **RSW:** Ignored by MMU
RISC-V page faults

• RISC-V supports 16 exceptions
  • Three related to paging

• Exceptions are controlled transfers into the kernel
  • Seen in previous and future lectures

• Information we might need to handle a page fault:
  1. The VA that caused the fault
  2. The type of violation that caused the fault
  3. The instruction where the fault occurred
<table>
<thead>
<tr>
<th>Intr</th>
<th>Exception Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Instruction address misaligned</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Instruction access fault</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>Illegal instruction</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>Breakpoint</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td>Reserved</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>Load access fault</td>
</tr>
<tr>
<td>0</td>
<td>6</td>
<td>AMO address misaligned</td>
</tr>
<tr>
<td>0</td>
<td>7</td>
<td>Store/AMO access fault</td>
</tr>
<tr>
<td>0</td>
<td>8</td>
<td>Environment call</td>
</tr>
<tr>
<td>0</td>
<td>9-11</td>
<td>Reserved</td>
</tr>
<tr>
<td>0</td>
<td>12</td>
<td>Instruction page fault</td>
</tr>
<tr>
<td>0</td>
<td>13</td>
<td>Load page fault</td>
</tr>
<tr>
<td>0</td>
<td>14</td>
<td>Reserved</td>
</tr>
<tr>
<td>0</td>
<td>15</td>
<td>Store/AMO page fault</td>
</tr>
<tr>
<td>0</td>
<td>&gt;16</td>
<td>Reserved</td>
</tr>
</tbody>
</table>
STVAL register

• Contains exception-specific information
• Some exceptions don’t use it (set to zero)
• Page faults set it to the faulting address!
• Use r_stval() in xv6 to access
Gathering info to handle a pgfault

1. The VA that caused the fault?
   • STVAL, or r_stval() in xv6

2. The type of violation that caused the fault?
   • Encoded in SCAUSE, or r_scause() in xv6
   • 12: page fault caused by an instruction fetch
   • 13: page fault caused by a read
   • 15: page fault cause by a write

3. The IP and privilege mode where fault occurred?
   • User IP: tf->epc
   • U/K: SSTATUS, or r_sstatus() & SSTATUS_SPP in xv6
xv6 user memory layout

Figure 3.4: Memory layout of a user process with its initial stack.
Idea: On-demand page allocation

- Problem: `sbrk()` is old-fashioned
  - Allocates memory that may never be used

- Modern OSes allocate memory lazily
  - Insert physical pages when they’re accessed instead of in advance
On-demand page allocation demo
Caveats

• Page faults below user stack are invalid
• Must not fault in pages above `brk`
• What about `copyin()` and `copyout()`?
• And many more caveats...

• Real kernels are difficult to build, every detail matters
Optimization: Zero pages

• Observation: In practice, some memory is never written to
• All memory gets initialized to zero
• Idea: Use just one zeroed page for all zero mappings
• Copy the zero page on write
Feature: Stack guard pages

• Observation: Stack has a finite size
• Push too much data and it could overflow into adjacent memory
• Idea: Install an empty mapping (PTE_V cleared) at the bottom of the stack
• Could automatically increase stack size in page fault handler
Optimization: Copy-on-write fork()

• Observation: Fork() copies all pages in new process
• But often, exec() is called immediately after fork()
  • Wasted copies
• Idea: modify fork() to mark pages copy-on-write
  • All pages in both processes become read-only
  • On page fault, copy page and mark R/W
  • Extra PTE bits (RSV) useful for indicating COW mappings
Optimization: Demand paging

• Observation: exec() loads entire object file into memory
  • Expensive, requires slow disk block access
  • Maybe not all of the file will be used

• Idea: Mark mapping as demand paged
  • On page fault, read disk block and install PTE

• Challenge: What if file is larger than physical memory?
Feature: Support more virtual memory than physical RAM

• Observation: More disk capacity than RAM
• Idea: “Page in” and out data between disk and RAM
  • Use page table entries to detect when disk access is needed
  • Use page table to find least recently used disk blocks to write back
• Works well when working set fits in RAM
Opportunity is large

Software-Defined Far Memory in Warehouse-Scale Computers
Lagar-Cavilla et. Al. ASPLOS’19.
Feature: Memory-mapped files

- Normally files accessed through read(), write(), and lseek()
- Idea: Use load and store to access file instead
  - New system call mmap() can place file at location in memory
  - Use memory offset to select block rather than seeking
- Any holes in file mappings require zeroed pages!
Feature: Distributed shared memory

• Idea: Use virtual memory to share physical memory between several machines on the network
Translation Lookaside Buffers (TLBs)

• Virtual memory translations are stored in RAM
• **Problem:** RAM is slow!
  • Imagine walking the page table for each memory access
• **Solution:** Cache the page table (i.e.) a TLB
TLB management

• xv6 flushes entire TLB during user/kernel transitions
  • Why?
• RISC-V TLB is more sophisticated in reality
  • \texttt{PTE\_G}: global TLB bits
  • \texttt{SATP}: takes ASID number
  • \texttt{sfence.vma}: ASID number, addr
  • \textbf{Large pages}: 2MB and 1GB support
Virtual memory is still evolving

Recent Linux Kernel Changes:

• Support for up to 5-level page tables
  • 57 virtual address bits!
  • In RISCV: sv39 (3 levels), sv48 (4 levels), and sv57 (5 levels)

• Support for ASIDs
  • TLB can cache multiple page tables at a time

• New isolation mechanisms like MPK
  • Allows fast changes to permissions within an address space
Conclusion

• There’s no one way to use virtual memory
  • Many different use cases
  • Enables powerful features and optimizations

• xv6 presents one example
  • It lacks many features of real OSes
  • But still quite complex!

• Our goal: Teach you ideas so you can extrapolate