Virtual memory - Paging

Johan Montelius
KTH
2017

The process

0x00000000 0xC0000000 0xffffffff
code (.text) data heap stack kernel

Memory layout for a 32-bit Linux process

Segments - a could be solution

Processes in virtual space

Physical memory

Address translation by MMU (base and bounds)

External fragmentation: free areas of free space that is hard to utilize.

Solution: allocate larger segments ... internal fragmentation.
Let's try again

It's easier to handle fixed size memory blocks.

Can we map a process virtual space to a set of equal size blocks?

An address is interpreted as a *virtual page number* (VPN) and an *offset*.

---

Remember the segmented MMU

The paging MMU
The x86-32 architecture supports both segmentation and paging. A virtual address is translated to a linear address using a segmentation table. The linear address is then translated to a physical address by paging.

Linux and Windows do not use segmentation to separate code, data, nor stack.

The X86-64 (the 64-bit version of the x86 architecture) has dropped many features for segmentation.

Still used to control access rights of memory operations.

Still used to manage thread local storage and CPU specific data.
**The page table entry**

Example Linux on (32bit) x86

<table>
<thead>
<tr>
<th></th>
<th>User/Supervisor</th>
<th>Present</th>
<th>R/W</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20-bit frame number</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>R/W</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*If the page index is 20 bits, does the frame number need to be 20 bits?*

---

**Physical Address Extension (PAE)**

In 1995 the x86 architecture provided 24-bit frame numbers. The CPU could thus address 64 Gbyte of physical address space (24-bit frame, 12-bit offset).

Each process still had a 32-bit virtual address space, (20-bit page number, 12-bit offset) i.e. 4 Gbyte.

The x86_64 architecture supports 48-bit virtual address space and up to 52-bit physical address space.

Linux supports 48-bit virtual address (47-bit user space) and up to 46-bit physical address space (64 Tbyte). Check your address space in `/proc/cpuinfo`.

Physical memory is in reality limited by chipset, motherboard, memory modules etc. Check your available memory in `/proc/meminfo`.

---

**largest server**

Largest server on the market, SGI 3000, can scale up to 256 CPUs and 64 Tbyte of RAM (NUMA) - running Linux.

---

**Speed matters**

```
 movl 0x11111222, %eax
```

- we need a page table base register, PTBR
- the virtual page number, VPN, is 0x11111
- read the page table entry from PTBR + (0x11111 * 8)
- extract frame number PFN from the entry
- the offset is 0x222
- read the memory location at (PFN << 12) + 0x222

_An extra memory operation for each memory reference._
TLB: hardware support

The CPU keeps a translation look-aside buffer, TLB, with the most recent page table entries.

The buffer is implemented using a content-addressable memory keyed by the virtual page number.

If the page table entry is found - great!

If the page table entry is not found - access the real page table in memory.

Who handles a TLB miss

RISC architecture
- MIPS, Sparc, ARM
- The hardware rises an interrupt.
- The operating system jumps to a trap handler.
- The operating system will access the TLB and update the TLB.

CISC architecture
- x86
- The hardware "knows" where to find the page table (CR3 register).
- The hardware will access the page table and updates the TLB.

TLB replacement policy

Which TLB entry should we remove if the TLB is full?

Process switching

What happens when we switch process?

The TLB contains the cached translations of the running process, when switching process the TLB must (in general) be flushed.

Do we have to flush the whole TLB?

Is this best handled by the hardware or operating system?

Can we do pre-fetching of page table entries?
### The paging MMU with TLB

![Diagram of paging MMU with TLB](image)

- **virtual addr.**
- **offset**
- **VPN**
- **PFN**
- **physical address**
- **Page table in memory**

### Size matters

Using 4 Kibyte pages (12 bits) for a 4 Gibyte address space (32 bits) will result in 1Mi (20 bits) page table entries.

Each page table entry is 4 bytes.

A page table has the size of 4 Mibyte.

Each process has its own page table.

For 100 processes we need room for 400 Mibyte of page tables.

Problem!

### The solution - not.

Why not use pages of size 4 Mibyte?
- Use a 22 bit offset and 10 bit virtual page number.
- Page table 4 Kibyte (1024 entries, 4 byte each).
- Case closed!

4 Mibyte pages are used and do have advantages but it is not a general solution.

### Mostly empty space

Map only the areas that are actually used.
Hybrid approach - paged segmented memory

What if each segment was rarely larger than 1Ki pages of 4Kibyte.

Multi-level page table

Mostly empty space

More than two levels

Scheme used in PAE, where each entry has a 24-bit physical base address.
Trace the translation of a 32-bit virtual address to a 36-bit physical address.
Linux on x86_64 architectures

- A 64-bit address but only 48-bits are used.
- Bits 63-47 are either 1, kernel space, or 0, user space.
- The 48 bits are divided into:
  - 9-bit page global directory index
  - 9-bit page upper directory index
  - 9-bit page lower directory index
  - 9-bit page table index
  - 12-bit offset
- A page table entry is 8 bytes and contains a 40-bit physical address base address.
- The 40-bit base is combined with the 12-bit index to a 52-bit physical address.

Inverted page tables

- Why not do something completely different?
  - We will probably not have more than say 8 Gbyte of main memory.
  - If we divide this into 4 Kibyte frames we have 2 Mi frames.
  - Assume maintain a table with 2 Mi entries that describes which process and page that occupies the frame.
  - To translating a virtual address we simply search the table (efficient if we use a hash table).
  - Used by some models of PowerPC, Ultra Sparc and Itanium.

Summary

- Segmentation is not an ideal solution (why?).
- Small fixed size pages is a solution.
- Speed of translation is a problem (what is the solution?)
- The size of the page table is a problem (and you know how to solve it).
- Inverted page tables - an alternative approach.

AC/DC - TLB

- TLB - dynamite, makes paging possible.