Virtual memory - Paging

Johan Montelius

KTH

2018
Memory layout for a 32-bit Linux process
Segments - a could be solution

Processes in virtual space

Address translation by MMU (base and bounds)

Physical memory
Segments - a could be solution

Processes in virtual space

Address translation by MMU (base and bounds)

Physical memory
Segments - a could be solution

Processes in virtual space

Address translation by MMU (base and bounds)

Physical memory
Segments - a could be solution

Processes in virtual space

Address translation by MMU (base and bounds)

Physical memory
Segments - a could be solution

Processes in virtual space

Address translation by MMU
(base and bounds)

Physical memory
Segments - a could be solution

Processes in virtual space

Physical memory

Address translation by MMU (base and bounds)
Segments - a could be solution

Processes in virtual space

Address translation by MMU (base and bounds)

Physical memory
Segments - a could be solution

Processes in virtual space

Physical memory

Address translation by MMU
(base and bounds)
Segments - a could be solution

Processes in virtual space

Address translation by MMU (base and bounds)

Physical memory
Segments - a could be solution

Processes in virtual space

Address translation by MMU (base and bounds)

Physical memory
Segments - a could be solution

Processes in virtual space

Address translation by MMU (base and bounds)

Physical memory
Segments - a could be solution

Processes in virtual space

Address translation by MMU (base and bounds)

Physical memory
Segments - a could be solution

Processes in virtual space

Address translation by MMU (base and bounds)

Physical memory
Segments - a could be solution

Processes in virtual space

Address translation by MMU (base and bounds)

Physical memory
Segments - a could be solution

Processes in virtual space

Address translation by MMU
(base and bounds)

Physical memory
one problem

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

Solution: allocate larger segments ...
one problem

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

Solution: allocate larger segments ... internal fragmentation.
another problem

virtual space

physical memory
another problem

virtual space

code

physical memory
another problem

virtual space

physical memory
another problem

virtual space

physical memory

code
another problem

virtual space

used

code

physical memory
another problem

virtual space

used

code

physical memory

not used?
We’re reserving physical memory that is not used.
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.
Let’s try again

It’s easier to handle fixed size memory blocks.
It’s easier to handle fixed size memory blocks.

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

MMU

virtual addr.  // offset

index

segment table

physical address

exception

no

yes

within bounds
The paging MMU

virtual addr.
MMU

virtual addr.
The paging MMU

MMU

virtual addr.
//

page table
The paging MMU

MMU

virtual addr.

VPN

page table
The paging MMU

MMU

virtual addr.  offset

VPN

page table
The paging MMU

MMU

virtual addr.  offset

VPN

page table

physical address
The paging MMU

MMU

virtual addr. \( \rightarrow \) offset

VPN

page table

\( \rightarrow \) physical address

\( /\)
The paging MMU

MMU

virtual addr. // offset

VPN

page table

exception

available

physical address
the MMU

virtual address

Segmentation
virtual address

Segmentation

linear address
the MMU

virtual address → within bounds

Segmentation

linear address

exception
the MMU

virtual address → exception

Segmentation → within bounds → linear address

Paging
the MMU

virtual address \[\rightarrow\] exception

within bounds

Segmentation

linear address \[\rightarrow\] Paging

physical address
the MMU

virtual address → exception

within bounds → linear address

Segmentation

physical address → exception

page available → Paging

Physical Address Translation Process:

1. Virtual Address to Linear Address (Segmentation)
2. Linear Address to Physical Address (Paging)

Bounds Checking:
- Check if virtual address is within segment bounds.
- Linear address is generated if within bounds.

Exceptions:
- Raise an exception if virtual address is out of bounds.
- Raise an exception if page is not available.
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.
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 manage thread local storage and CPU specific data.
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 x86-32 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 x86-32 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.
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 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 manage *thread local storage* and *CPU specific data*. 
Processes in virtual space

Physical memory
Processes in virtual space

Physical memory
Processes in virtual space

Physical memory
Processes in virtual space
Processes in virtual space

Physical memory
Processes in virtual space
Processes in virtual space

Physical memory
the process

Processes in virtual space

Physical memory
Processes in virtual space
the process

Processes in virtual space
Processes in virtual space

Physical memory
the process

Processes in virtual space

Physical memory
Processes in virtual space

Only pages actually used need to be in memory.

Physical memory
virtual space

physical memory
virtual space

physical memory
three pages

virtual space

physical memory
virtual space

available

not available (page fault)

physical memory
virtual space

available

not available (page fault)

physical memory

not allocated (segmentation fault)
example Linux on (32bit) x86

31
example Linux on (32bit) x86

If the page index is 20 bits, does the frame number need to be 20 bits?
example Linux on (32bit) x86

If the page index is 20 bits, does the frame number need to be 20 bits?
The page table entry

example Linux on (32bit) x86

31  12

20-bit frame number

R/W

Present

If the page index is 20 bits, does the frame number need to be 20 bits?
If the page index is 20 bits, does the frame number need to be 20 bits?
If the page index is 20 bits, does the frame number need to be 20 bits?
The page table entry

example Linux on (32bit) x86

If the page index is 20 bits, does the frame number need to be 20 bits?
If the page index is 20 bits, does the frame number need to be 20 bits?
In 1995 the x86 architecture provided 24-bit frame numbers. The CPU could thus address 64 GiB 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 GiB.

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 TiB). 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.
Physical Address Extension (PAE)

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

In 1995 the x86 architecture provided 24-bit frame numbers. The CPU could thus address 64 Gibyte 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 Gibyte.

The x86_64 architecture supports 48-bit virtual address space and up to 52-bit physical address space.
In 1995 the x86 architecture provided 24-bit frame numbers. The CPU could thus address 64 Gibyte 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 Gibyte.

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 Tibyte). Check your address space in /proc/cpuinfo.
In 1995 the x86 architecture provided 24-bit frame numbers. The CPU could thus address 64 Gibyte 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 Gibyte.

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

- we need a page table base register, PTBR

```c
movl 0x11111222, %eax
```
we need a page table base register, PTBR

the virtual page number, VPN, is 0x11111

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)
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
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
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
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
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.*
The CPU keeps a *translation look-aside buffer*, TLB, with the most recent page table entries.
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*. 
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!
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.
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 raises 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.
Who handles a TLB miss

**RISC architecture**

- MIPS, Sparc, ARM
- The hardware raises an interrupt.
Who handles a TLB miss

**RISC architecture**
- MIPS, Sparc, ARM
- The hardware rises an interrupt.
- The operating system jumps to a *trap handler*. 
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.
Who handles a TLB miss

<table>
<thead>
<tr>
<th>RISC architecture</th>
<th>CISC architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIPS, Sparc, ARM</td>
<td>x86</td>
</tr>
<tr>
<td>The hardware rises an interrupt.</td>
<td></td>
</tr>
<tr>
<td>The operating system jumps to a <em>trap handler</em>.</td>
<td></td>
</tr>
<tr>
<td>The operating system will access the TLB and update the TLB.</td>
<td></td>
</tr>
</tbody>
</table>
Who handles a TLB miss

**RISC architecture**
- MIPS, Sparc, ARM
- The hardware raises 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).
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.
What happens when we switch process?

The TLB contains the cached translations of the running process, when switching processes 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?
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.
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?
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?
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?
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?
virtual addr.
The paging MMU with TLB

virtual addr. //
The paging MMU with TLB
The paging MMU with TLB

virtual addr. \[\rightarrow\] VPN \[\rightarrow\] TLB
The paging MMU with TLB

virtual addr. \[\rightarrow\] offset

VPN \[\downarrow\] TLB

PFN \[\rightarrow\] physical address
The paging MMU with TLB
The paging MMU with TLB

virtual addr. → offset → VPN → TLB → PFN → + → physical address

VPN → PTBR → + → VPN
The paging MMU with TLB

- Virtual address
- Offset
- VPN
- TLB
-PFN
- Physical address
- PTBR
- Page table in memory
The paging MMU with TLB

virtual addr. \[\rightarrow\] offset \[\rightarrow\] VPN \[\rightarrow\] TLB \[\rightarrow\]PFN \[\rightarrow\] physical address

VPN \[\rightarrow\] PTE

Page table in memory
Using 4 Kibyte pages (12 bits) for a 4 Gibyte address space (32 bits) will result in 1Mi (20 bits) page table entries.
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.
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.
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.
Using 4 Kibyte pages (12 bits) for a 4 Gibyte address space (32 bits) will result in 1 Mi (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.
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!
Why not use pages of size 4 Mibyte?
The solution - not.

Why not use pages of size 4 Mibyte?

- Use a 22 bit offset and 10 bit virtual page number.
Why not use pages of size 4 Mbyte?

- Use a 22 bit offset and 10 bit virtual page number.
- Page table 4 Kibyte (1024 entries, 4 byte each).
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!
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.
Map only the areas that are actually used.
Map only the areas that are actually used.
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.
Hybrid approach - paged segmented memory

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

| seg | 18-bit page number |
What if each segment was rarely larger than 1Ki pages of 4Kibyte.

| seg | 18-bit page number | 12-bit offset |
Hybrid approach - paged segmented memory

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

<table>
<thead>
<tr>
<th>seg</th>
<th>18-bit page number</th>
<th>12-bit offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>29</td>
<td>12</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
What if each segment was rarely larger than 1Ki pages of 4Kibyte.

<table>
<thead>
<tr>
<th>31</th>
<th>29</th>
<th>12</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>seg</td>
<td>18-bit page number</td>
<td>12-bit offset</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>base/bound</th>
</tr>
</thead>
</table>
Hybrid approach - paged segmented memory

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

<table>
<thead>
<tr>
<th>seg</th>
<th>18-bit page number</th>
<th>12-bit offset</th>
</tr>
</thead>
</table>

*Base/bound* → *bound* → *page table* → *base*
Hybrid approach - paged segmented memory

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

<table>
<thead>
<tr>
<th>31</th>
<th>29</th>
<th>12</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>seg</td>
<td>18-bit page number</td>
<td>12-bit offset</td>
<td></td>
</tr>
</tbody>
</table>

Diagram:

- base/bound
- bound
- page table
- frame number
- base
Hybrid approach - paged segmented memory

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

<table>
<thead>
<tr>
<th>seg</th>
<th>18-bit page number</th>
<th>12-bit offset</th>
</tr>
</thead>
</table>

- **seg**: 31 29
- **18-bit page number**: 12
- **12-bit offset**: 0

Diagram:
- **base/bound**
- **bound**
- **base**
- **frame number**
- **page table**
- **page**
Hybrid approach - paged segmented memory

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

```
<table>
<thead>
<tr>
<th>seg</th>
<th>18-bit page number</th>
<th>12-bit offset</th>
</tr>
</thead>
</table>
```

```
base/bound
```

```
bound
```

```
page table
```

```
frame number
```

```
page
```
Multi-level page table

31

Used by Intel 80386
### Multi-level Page Table

<table>
<thead>
<tr>
<th>31</th>
<th>22</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>10-bit directory index</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Used by Intel 80386*
Multi-level page table

<table>
<thead>
<tr>
<th>31</th>
<th>22</th>
<th>12</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>10-bit directory index</td>
<td>10-bit page index</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Multi-level page table**

<table>
<thead>
<tr>
<th>31</th>
<th>22</th>
<th>12</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>10-bit directory index</td>
<td>10-bit page index</td>
<td>12-bit offset</td>
<td></td>
</tr>
</tbody>
</table>

*Used by Intel 80386*
### Multi-level page table

<table>
<thead>
<tr>
<th>31</th>
<th>22</th>
<th>12</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>10-bit directory index</td>
<td>10-bit page index</td>
<td>12-bit offset</td>
<td></td>
</tr>
</tbody>
</table>

**Page directory**

*Used by Intel 80386*
## Multi-level page table

<table>
<thead>
<tr>
<th>31</th>
<th>22</th>
<th>12</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>10-bit directory index</td>
<td>10-bit page index</td>
<td>12-bit offset</td>
<td></td>
</tr>
</tbody>
</table>

*Used by Intel 80386*
Multi-level page table

<table>
<thead>
<tr>
<th>31</th>
<th>22</th>
<th>12</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>10-bit directory index</td>
<td>10-bit page index</td>
<td>12-bit offset</td>
<td></td>
</tr>
</tbody>
</table>

Used by Intel 80386
Multi-level page table

Used by Intel 80386
Multi-level page table

10-bit directory index  10-bit page index  12-bit offset

page directory → page table → page

Used by Intel 80386
Multi-level page table

31  22  12  0
10-bit directory index  10-bit page index  12-bit offset

Used by Intel 80386
Mostly empty space

page directory

virtual address space
page directory

virtual address space
Mostly empty space

page directory

virtual address space
Mostly empty space

page directory

virtual address space
Mostly empty space

page directory

virtual address space
Mostly empty space

page directory

Each page table can map 4 Mibyte

Virtual address space
More than two levels

Scheme used in PAE, where each entry has a 24-bit physical base address. Each page table entry was 8 bytes wide.

Trace the translation of a 32-bit virtual address to a 36-bit physical address.
More than two levels

2-bit page global directory index

Scheme used in PAE, where each entry has a 24-bit physical base address. Each page table entry was 8 bytes wide.
More than two levels

Scheme used in PAE, where each entry has a 24-bit physical base address. Each page table entry was 8 bytes wide.

Trace the translation of a 32-bit virtual address to a 36-bit physical address.

9-bit page middle directory index

2-bit page global directory index
More than two levels

Scheme used in PAE, where each entry has a 24-bit physical base address. Each page table entry was 8 bytes wide.

Trace the translation of a 32-bit virtual address to a 36-bit physical address.

2-bit page global directory index

9-bit page middle directory index

9-bit page table index
Scheme used in PAE, where each entry has a 24-bit physical base address. Each page table entry was 8 bytes wide.

Trace the translation of a 32-bit virtual address to a 36-bit physical address.
Scheme used in PAE, where each entry has a 24-bit physical base address. Each page table entry was 8 bytes wide.
Scheme used in PAE, where each entry has a 24-bit physical base address. Each page table entry was 8 bytes wide.

Trace the translation of a 32-bit virtual address to a 36-bit physical address.
The 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.

- Linux can only handle a physical base address of 34 bits i.e 46 bit physical address.
The 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 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.
The 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.

Linux can only handle a physical base address of 34 bits i.e 46 bit physical address.
Inverted page tables

Why not do something completely different?

We will probably not have more than say 8 Gibyte 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.
Inverted page tables

Why not do something completely different?
Why not do something completely different?

- We will probably not have more than say 8 Gabyte of main memory.
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.
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.
Why not do something completely different?

- We will probably not have more than say 8 Gabyte 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).
Inverted page tables

Why not do something completely different?

- We will probably not have more than say 8 Gibyte 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.
Segmentation is not an ideal solution (why?).
Summary

- Segmentation is not an ideal solution (why?).
- Small fixed size pages is a solution.
- Segmentation is not an ideal solution (why?).
- Small fixed size pages is a solution.
- Speed of translation is a problem (what is the solution?)
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).
• 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.
TLB - dynamite, makes paging possible.