Sistemi Operativi I

Corso di Laurea in Informatica

2025-2026

SAPIENZA
UNIVERSITÀ DI ROMA

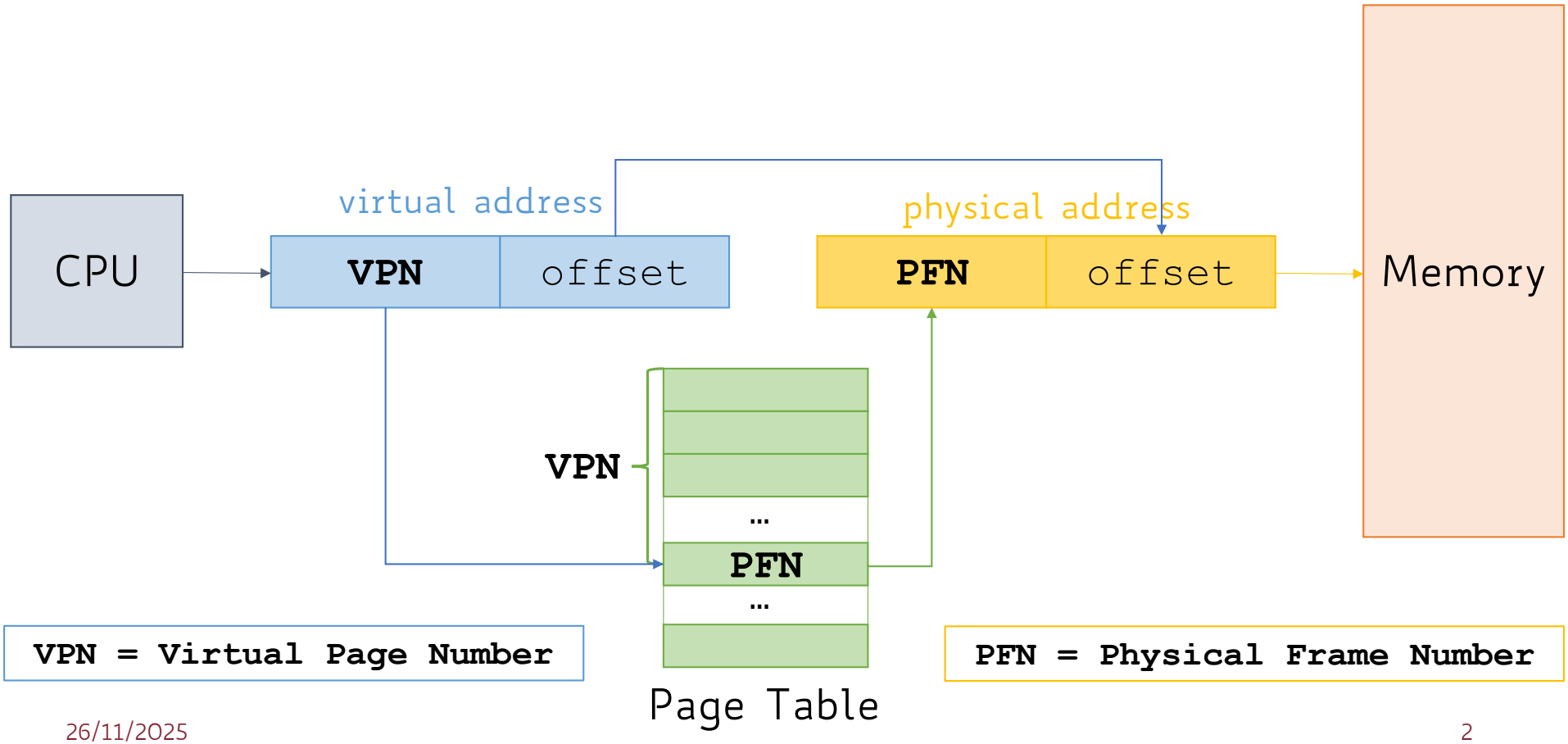
Gabriele Tolomei

Dipartimento di Informatica

Sapienza Università di Roma

tolomei@di.uniroma1.it

One-Slide Paging



The Overhead of Paging

- A user process generates (virtual) memory addresses almost constantly through the CPU

The Overhead of Paging

- A user process generates (virtual) memory addresses almost constantly through the CPU
- Every time a user process references a virtual memory address this must be translated to a physical one

The Overhead of Paging

- A user process generates (virtual) memory addresses almost constantly through the CPU
- Every time a user process references a virtual memory address this must be translated to a physical one
- To do so, the MMU must access the page table

How To Make Paging Faster?

- Where should the page table be stored, then?

How To Make Paging Faster?

- Where should the page table be stored, then?
  - **Registers** → PRO: very fast CON: very expensive and limited

How To Make Paging Faster?

- Where should the page table be stored, then?
  - **Registers** → PRO: very fast CON: very expensive and limited
  - **Main Memory** → PRO: highest capacity CON: quite slow (every memory translation requires one extra memory access!)

How To Make Paging Faster?

- Where should the page table be stored, then?
  - **Registers** → PRO: very fast CON: very expensive and limited
  - **Main Memory** → PRO: highest capacity CON: quite slow (every memory translation requires one extra memory access!)
  - **Trade-off** → Keep page tables in main memory and cache a subset of them into a **Translation Look-aside Buffer** (TLB)

Registers and Main Memory

- All memory accesses are equivalent: the memory hardware doesn't know what a particular part of memory is being used for

Registers and Main Memory

- All memory accesses are equivalent: the memory hardware doesn't know what a particular part of memory is being used for
- CPU can only access its registers and main memory (any access to other devices, e.g., hard drive, requires data to be moved into main memory first)

Registers and Main Memory

- All memory accesses are equivalent: the memory hardware doesn't know what a particular part of memory is being used for
- CPU can only access its registers and main memory (any access to other devices, e.g., hard drive, requires data to be moved into main memory first)
- Access to registers is very fast, generally one clock cycle

Registers and Main Memory

- All memory accesses are equivalent: the memory hardware doesn't know what a particular part of memory is being used for
- CPU can only access its registers and main memory (any access to other devices, e.g., hard drive, requires data to be moved into main memory first)
- Access to registers is very fast, generally one clock cycle
- Access to main memory is comparatively slow, and may take several clock cycles to complete

Cache Memory

- Bridge the gap between fast registers and slower main memory

Cache Memory

- Bridge the gap between fast registers and slower main memory
- Cache Memory: on-chip (thereby, fast!) intermediary memory built into most modern CPUs

Cache Memory

- Bridge the gap between fast registers and slower main memory
- Cache Memory: on-chip (thereby, fast!) intermediary memory built into most modern CPUs
- Several chunks of memory transferred from main memory to the cache

Cache Memory

- Bridge the gap between fast registers and slower main memory
- Cache Memory: on-chip (thereby, fast!) intermediary memory built into most modern CPUs
- Several chunks of memory transferred from main memory to the cache
- Access individual memory locations one at a time from the cache rather than from memory directly

Translation Look-aside Buffer (TLB)

- Essentially, a very fast L1-cache

Translation Look-aside Buffer (TLB)

- Essentially, a very fast L1-cache
- Fully-associative memory that stores page numbers (keys) and frame numbers (values) where the former are stored

Translation Look-aside Buffer (TLB)

- Essentially, a very fast L1-cache
- Fully-associative memory that stores page numbers (keys) and frame numbers (values) where the former are stored
- Memory accesses obey to the "locality" principle (memory references are often "close" to each other)

Translation Look-aside Buffer (TLB)

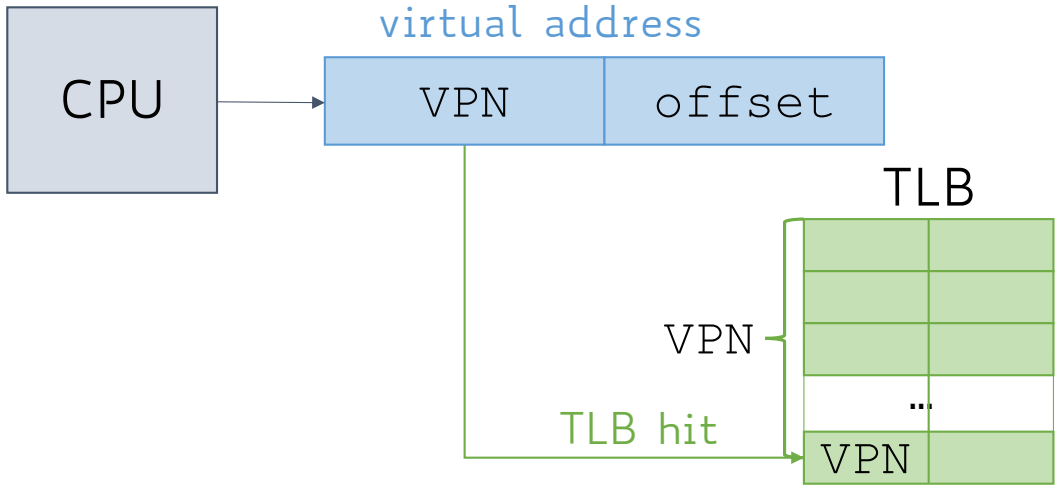
- Essentially, a very fast L1-cache
- Fully-associative memory that stores page numbers (keys) and frame numbers (values) where the former are stored
- Memory accesses obey to the "locality" principle (memory references are often "close" to each other)
- Locality still holds for address translation

Translation Look-aside Buffer (TLB)

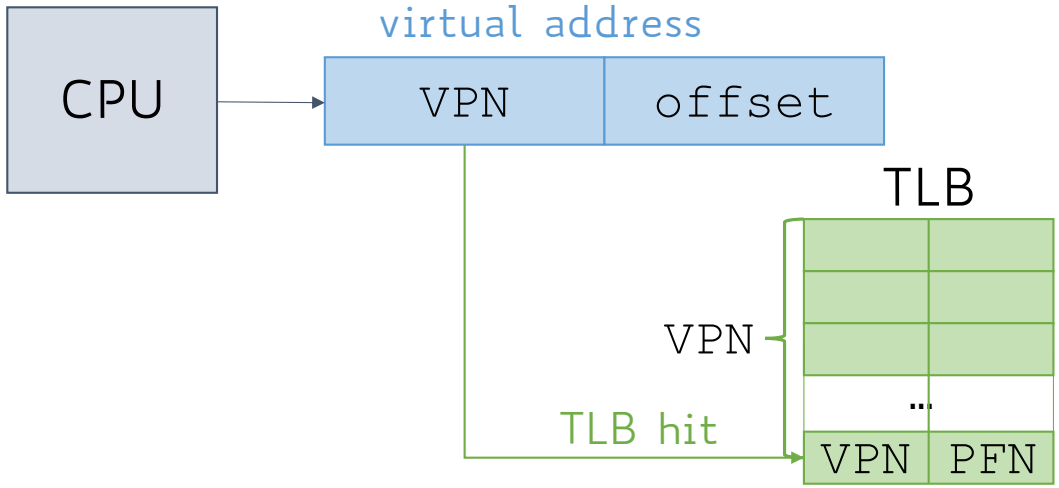
- Essentially, a very fast L1-cache
- Fully-associative memory that stores page numbers (keys) and frame numbers (values) where the former are stored
- Memory accesses obey to the "locality" principle (memory references are often "close" to each other)
- Locality still holds for address translation
- Typical TLB sizes range from 8 to 2048 entries

Translation Look-aside Buffer (TLB)

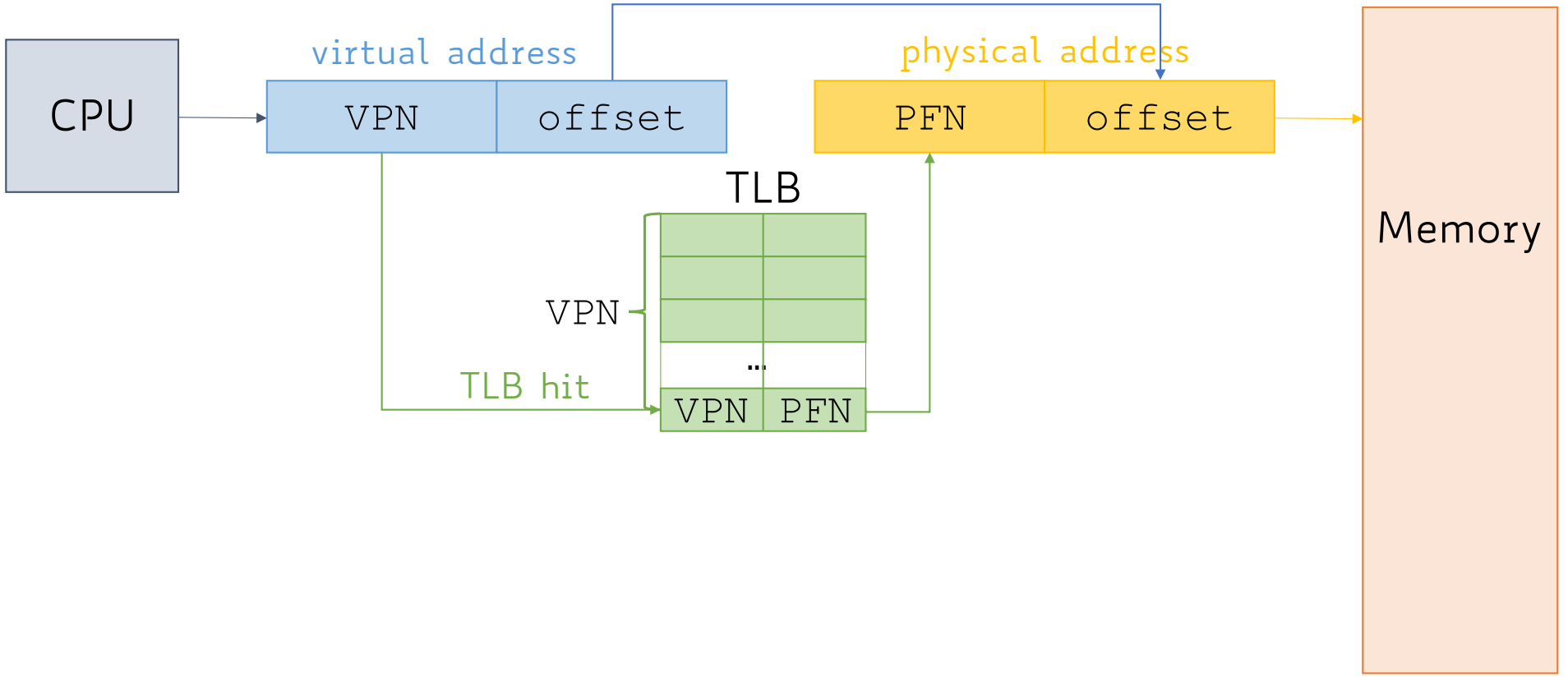
Translation Look-aside Buffer (TLB)



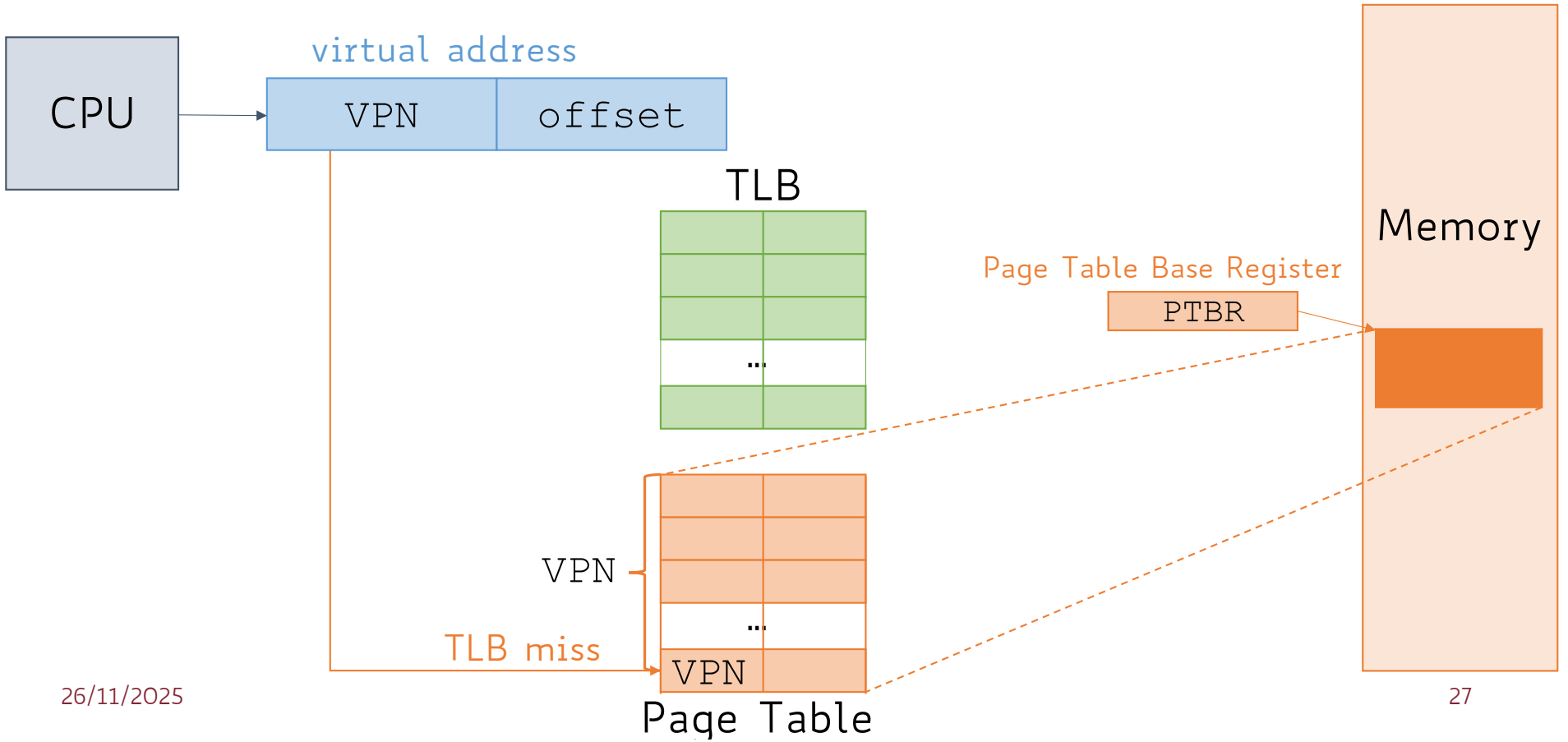
Translation Look-aside Buffer (TLB)



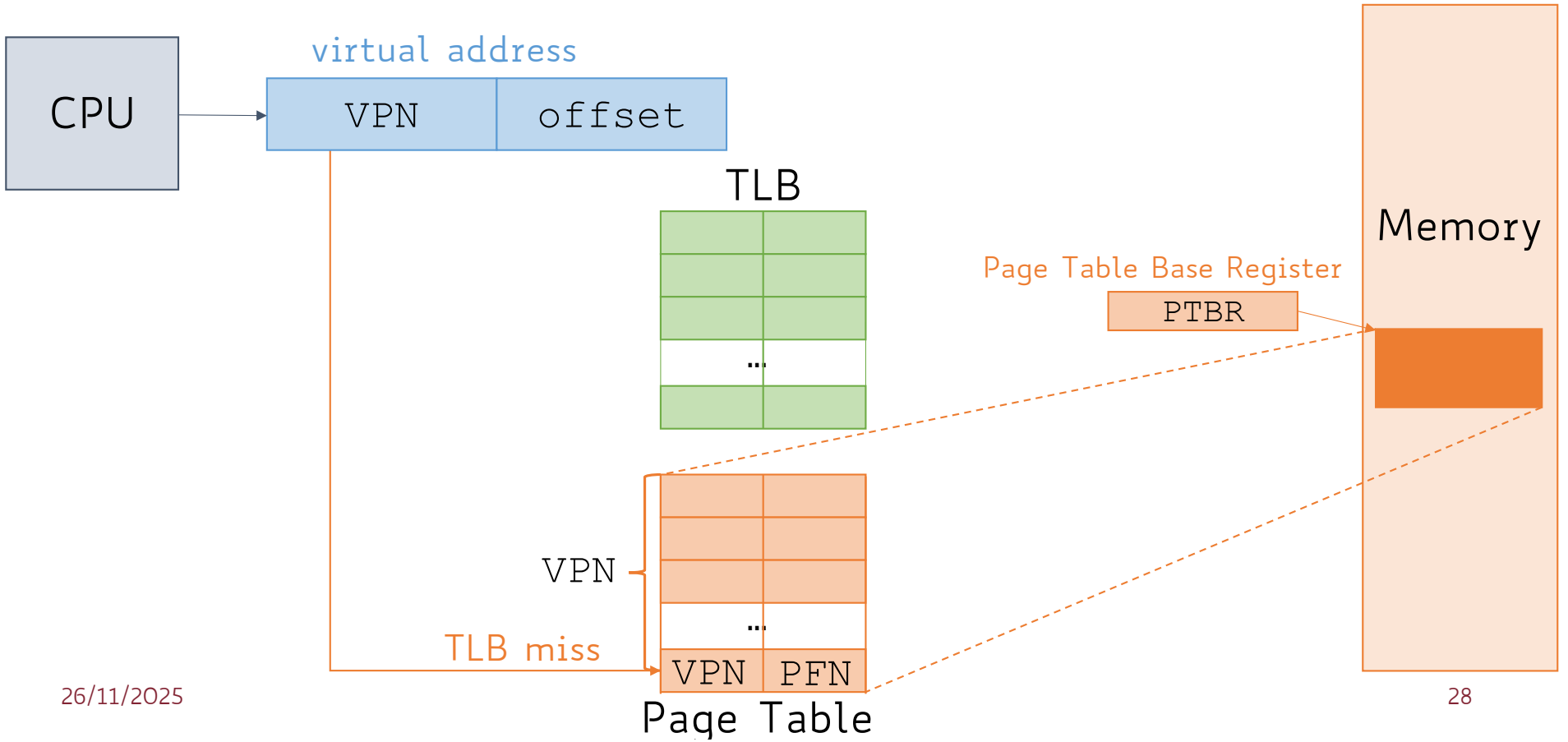
Translation Look-aside Buffer (TLB)



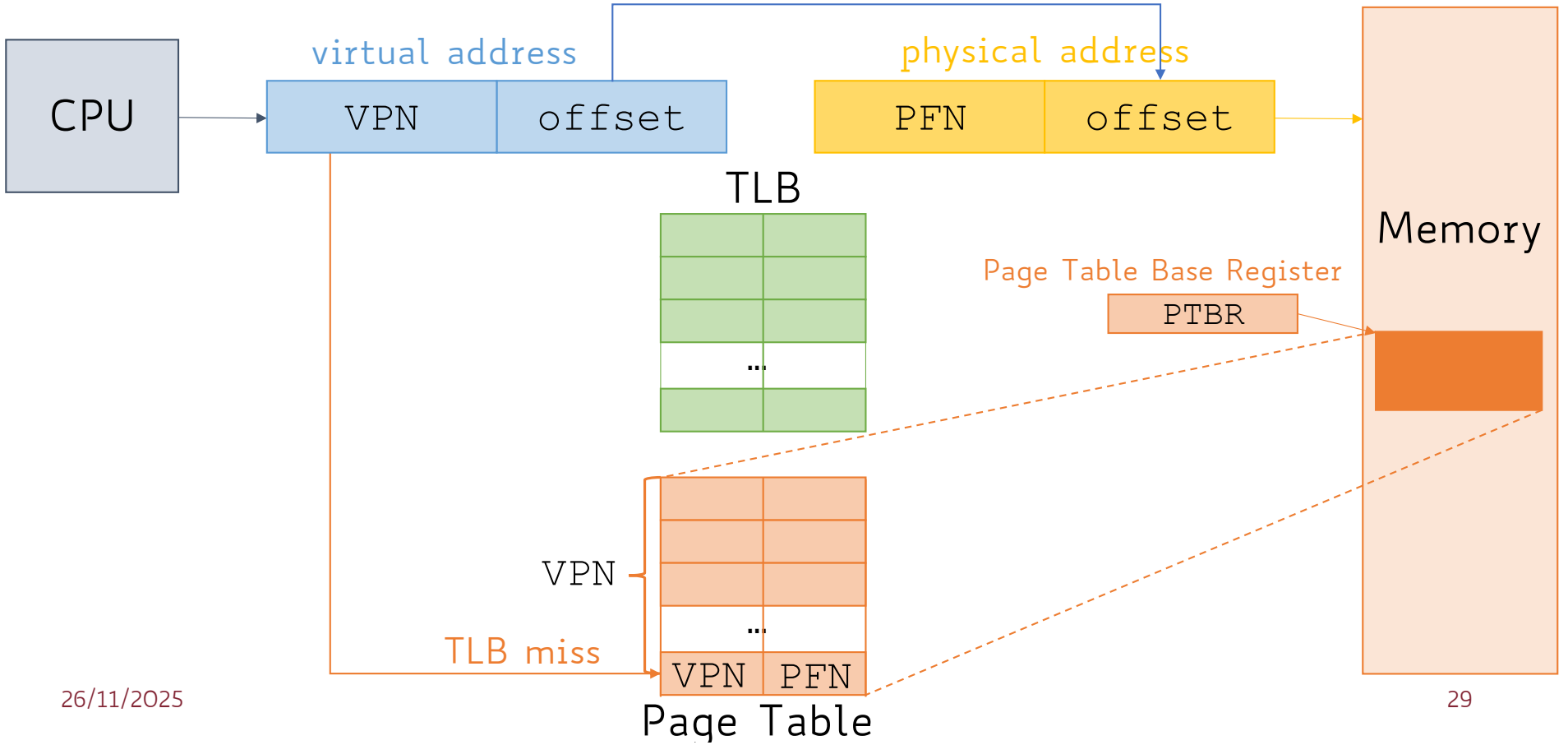
Translation Look-aside Buffer (TLB)



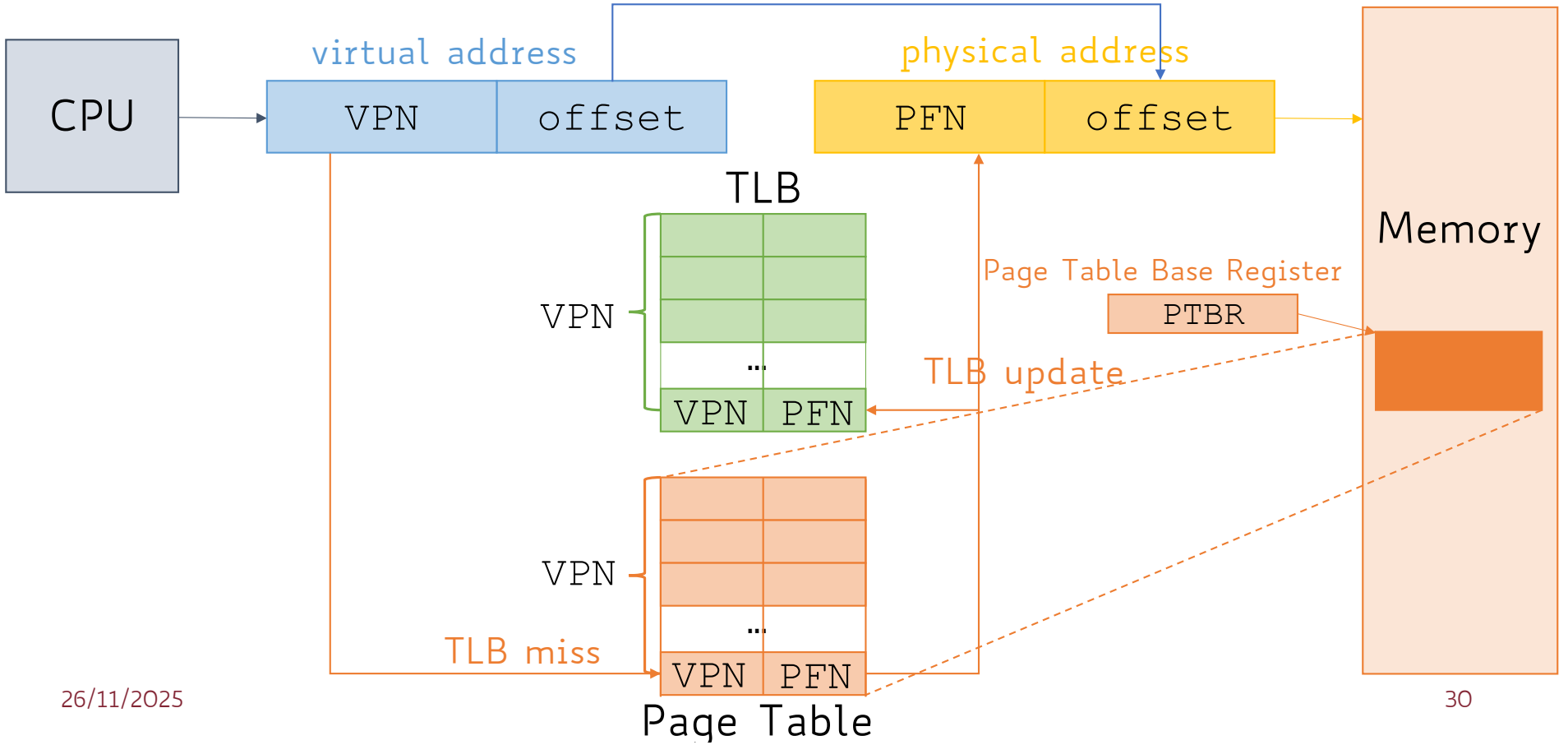
Translation Look-aside Buffer (TLB)



Translation Look-aside Buffer (TLB)



Translation Look-aside Buffer (TLB)



Translation Look-aside Buffer (TLB)

- TLB is a piece of hardware that is shared across all the processes

Translation Look-aside Buffer (TLB)

- TLB is a piece of hardware that is shared across all the processes
- The same page number can be mapped to different frame number depending on the process running

Translation Look-aside Buffer (TLB)

- TLB is a piece of hardware that is shared across all the processes
- The same page number can be mapped to different frame number depending on the process running
- We must ensure the TLB content is up-to-date w.r.t. the currently running process

Translation Look-aside Buffer (TLB)

How to deal with multiple process and a single TLB?

Translation Look-aside Buffer (TLB)

How to deal with multiple process and a single TLB?

- **2 setups:**
  - **basic:** at each context switch the content of the TLB is fully flushed and cleaned (cold-start → the first accesses will generate all TLB misses)

Translation Look-aside Buffer (TLB)

How to deal with multiple process and a single TLB?

- **2 setups:**
  - **basic:** at each context switch the content of the TLB is fully flushed and cleaned (cold-start → the first accesses will generate all TLB misses)
  - **advanced:** TLB entries dumped and restored within the PCB or adding a so-called process context ID (PCID) to each entry (the CPU will use a TLB entry iff the PCID of that entry corresponds to the ID of the running process)

Memory Access Cost

t_{MA} = physical memory access time
t_{TLB} = lookup time on the TLB cache
(NOTE: $t_{TLB} \ll t_{MA}$)
p = probability of TLB cache hit (i.e., *hit ratio*)
T_{MA} = total time required to *actually* get to physical memory each time a virtual address is referenced

Memory Access Cost

t_{MA} = physical memory access time
t_{TLB} = lookup time on the TLB cache
(NOTE: $t_{TLB} \ll t_{MA}$)
p = probability of TLB cache hit (i.e., *hit ratio*)
T_{MA} = total time required to *actually* get to physical memory each time a virtual address is referenced

without TLB
(i.e., Page Table full in memory)

Memory Access Cost

t_{MA} = physical memory access time
t_{TLB} = lookup time on the TLB cache
(NOTE: $t_{TLB} \ll t_{MA}$)
p = probability of TLB cache hit (i.e., *hit ratio*)
T_{MA} = total time required to *actually* get to physical memory each time a virtual address is referenced

without TLB
(i.e., Page Table full in memory)

$$T_{MA} = 2 * t_{MA}$$

Memory Access Cost

t_{MA} = physical memory access time
t_{TLB} = lookup time on the TLB cache
(NOTE: $t_{TLB} \ll t_{MA}$)
p = probability of TLB cache hit (i.e., *hit ratio*)
T_{MA} = total time required to *actually* get to physical memory each time a virtual address is referenced

without TLB
(i.e., Page Table full in memory)

$$T_{MA} = 2 * t_{MA}$$

with TLB

Memory Access Cost

t_{MA} = physical memory access time
t_{TLB} = lookup time on the TLB cache
(NOTE: $t_{TLB} \ll t_{MA}$)
p = probability of TLB cache hit (i.e., *hit ratio*)
T_{MA} = total time required to *actually* get to physical memory each time a virtual address is referenced

without TLB
(i.e., Page Table full in memory)

$$T_{MA} = 2 * t_{MA}$$

with TLB

$$T_{MA} = p*(\underbrace{t_{MA} + t_{TLB}}_{\text{TLB hit}}) + (1-p)*\underbrace{(2 * t_{MA} + t_{TLB})}_{\text{TLB miss}}$$

Memory Access Cost

t_{MA} = physical memory access time
t_{TLB} = lookup time on the TLB cache
(NOTE: $t_{TLB} \ll t_{MA}$)
p = probability of TLB cache hit (i.e., *hit ratio*)
T_{MA} = total time required to *actually* get to physical memory each time a virtual address is referenced

without TLB
(i.e., Page Table full in memory)

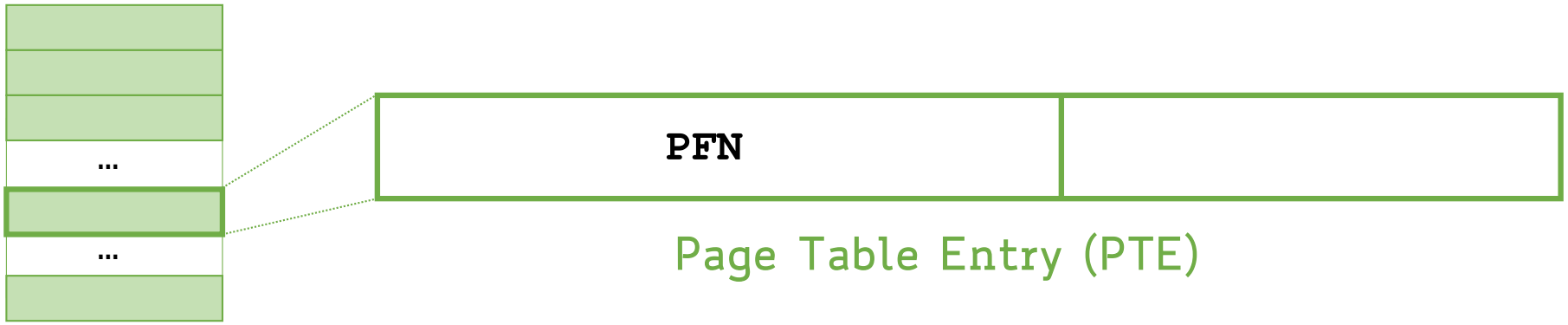
$$T_{MA} = 2 * t_{MA}$$

with TLB

$$T_{MA} = p*(\underbrace{t_{MA} + t_{TLB}}_{\text{TLB hit}})+(1-p)*\underbrace{(2 * t_{MA} + t_{TLB})}_{\text{TLB miss}}$$

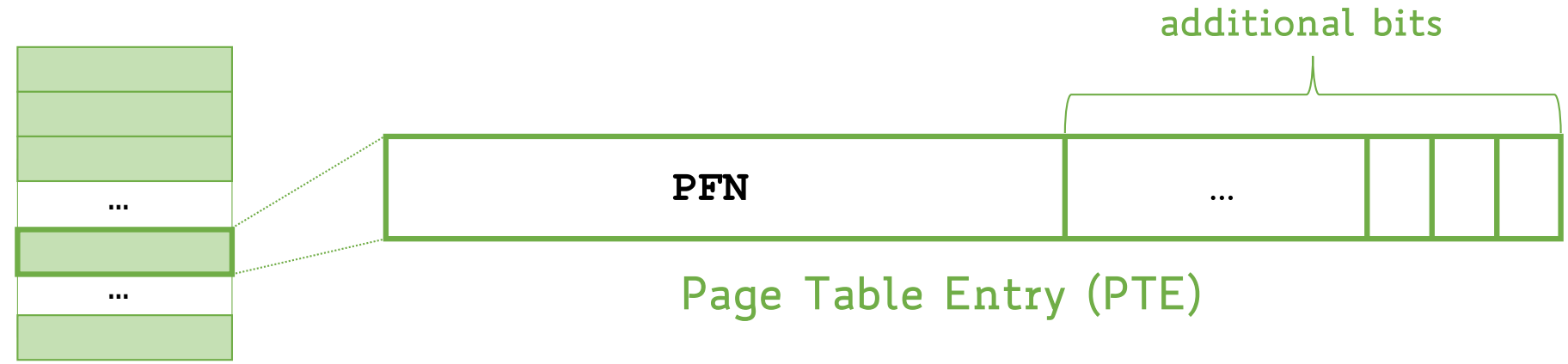
The larger the TLB the higher the probability p of hit ratio, thereby decreasing the average memory access cost

What's Inside the Page Table?



At least, the mapping to the Physical Frame Number (PFN)…

What's Inside the Page Table?



valid/invalid bit → check if the translation is legitimate

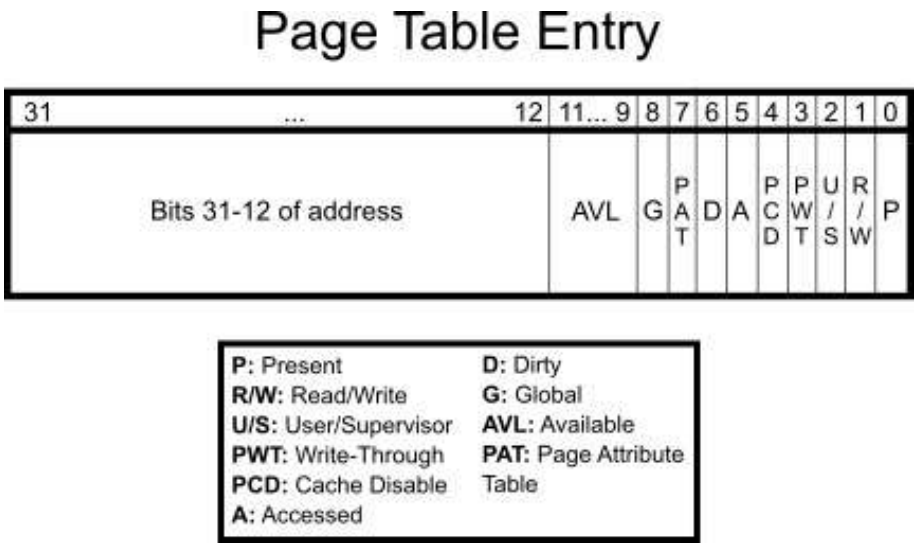
protection bits → read (R), write (W), execute (X)

present bit → the page is in RAM or swapped out to disk

reference bit → the page has been accessed (recently)

...

32-bit x86 PTE



source: https://wiki.osdev.org/Paging

Additional Protection

- The page table can also help to protect processes from accessing memory they shouldn't

Additional Protection

- The page table can also help to protect processes from accessing memory they shouldn't
- A bit or bits can be added to the page table to classify a page as read-write, read-only, read-write-execute, or combination of those

Additional Protection

- The page table can also help to protect processes from accessing memory they shouldn't
- A bit or bits can be added to the page table to classify a page as read-write, read-only, read-write-execute, or combination of those
- Each memory reference can be checked to ensure it is accessing the memory in the appropriate mode

Additional Protection

- The page table can also help to protect processes from accessing memory they shouldn't
- A bit or bits can be added to the page table to classify a page as read-write, read-only, read-write-execute, or combination of those
- Each memory reference can be checked to ensure it is accessing the memory in the appropriate mode
- Valid/invalid bits can be added to "mask off" entries in the page table that are not in use by the current process

Additional Protection

- Valid/Invalid bits cannot block all illegal memory accesses, due to the internal fragmentation

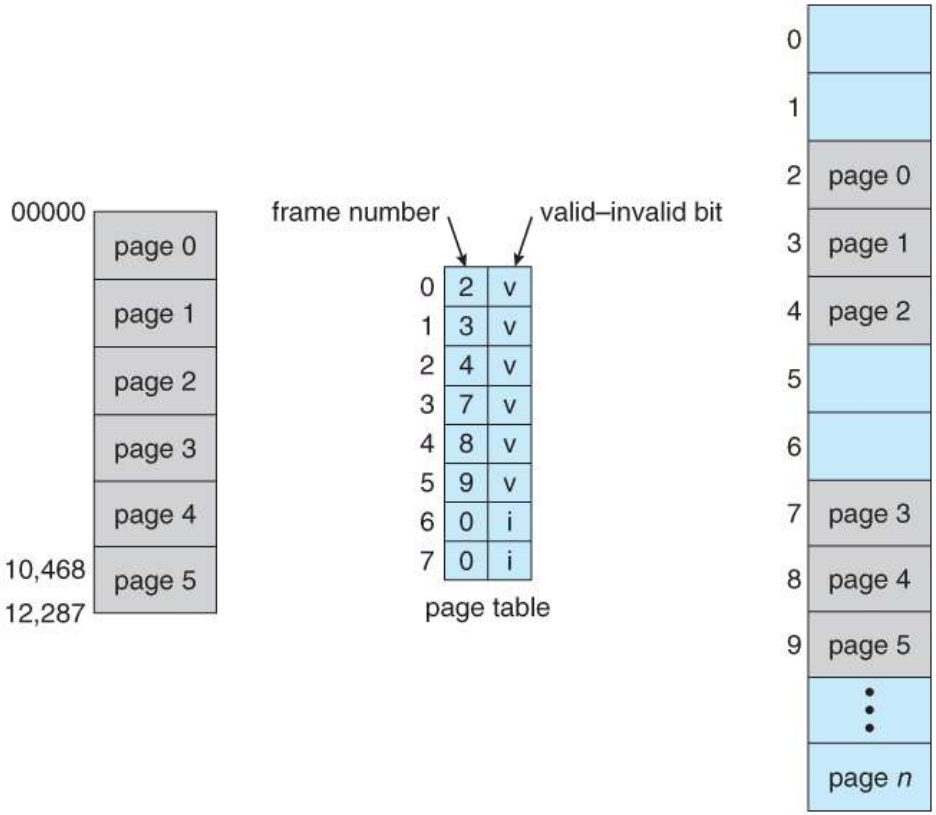
Additional Protection

- Valid/Invalid bits cannot block all illegal memory accesses, due to the internal fragmentation
- Many processes do not use all of the page table entries available, particularly in modern systems with very large potential page tables

Additional Protection

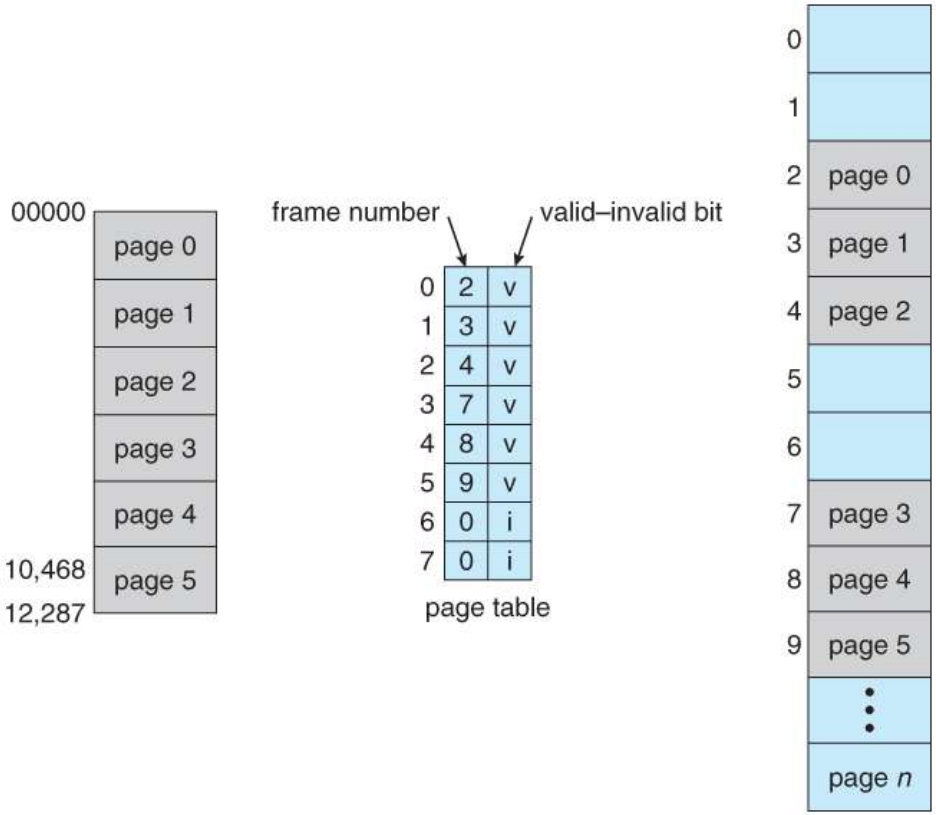
- Valid/Invalid bits cannot block all illegal memory accesses, due to the internal fragmentation
- Many processes do not use all of the page table entries available, particularly in modern systems with very large potential page tables
- Some systems use a page-table length register (PTLR) to specify the length of the page table

Additional Protection



valid/invalid bits can be used to flush TLB entries upon context switch if basic setup is used

Additional Protection



valid/invalid bits can be used to flush TLB entries upon context switch if basic setup is used

any entry whose invalid bit is set will be discarded (and updated)

Saving/Restoring Memory Upon Context Switch

- The PCB must now contain:
  - The value of the Page Table Base Register (`PTBR`)
  - Possibly a copy of the TLB entries

Saving/Restoring Memory Upon Context Switch

- The PCB must now contain:
  - The value of the Page Table Base Register (`PTBR`)
  - Possibly a copy of the TLB entries
- On a context switch:
  - Copy the `PTBR` value to the PCB
  - Copy the TLB to the PCB (optional)
  - Flush the TLB (if TLB is not saved to/restored from the PCB)
  - Restore the `PTBR` (i.e., with the value of the new running process)
  - Restore the TLB (if it was previously saved)

Sharing Pages

- Paging systems can make it very easy to share blocks of memory

Sharing Pages

- Paging systems can make it very easy to share blocks of memory
- Memory doesn't have to be contiguous anymore!

Sharing Pages

- Paging systems can make it very easy to share blocks of memory
- Memory doesn't have to be contiguous anymore!
- Just duplicate page entries of different processes to the same page frames (both for code and data)

Sharing Pages

- Only if code is **reentrant**!

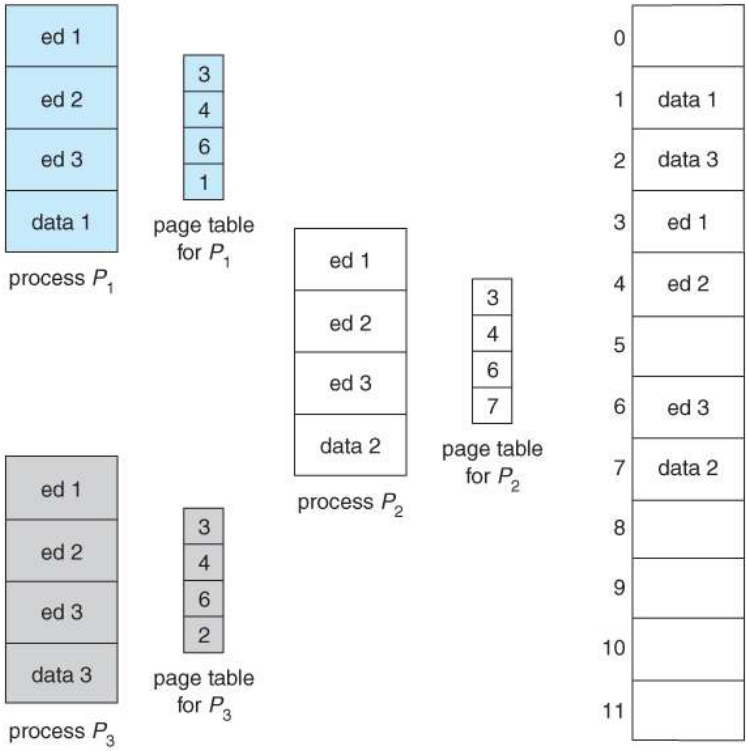
Sharing Pages

- Only if code is **reentrant**!
- It does not write to or change the code (i.e., it is non self-modifying)

Sharing Pages

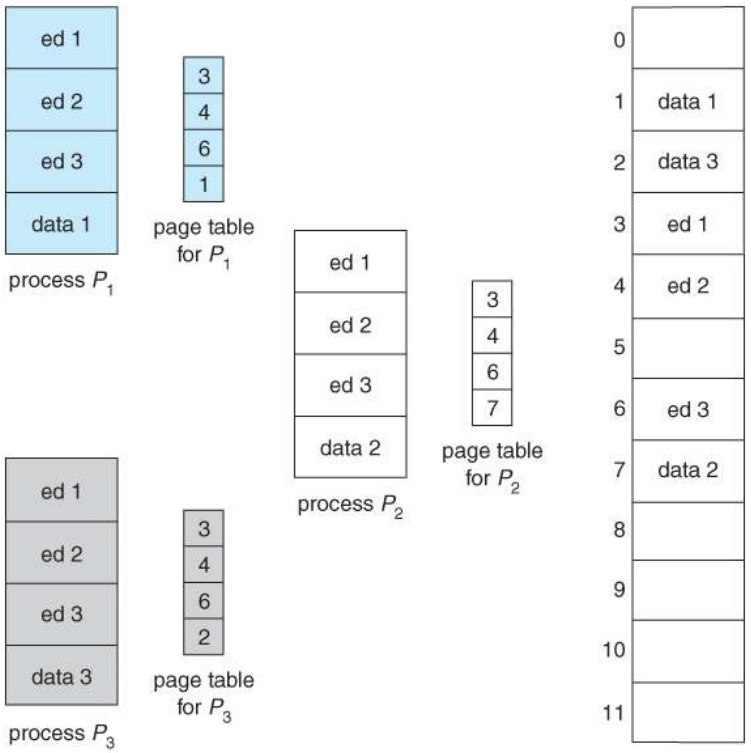
- Only if code is **reentrant**!
- It does not write to or change the code (i.e., it is non self-modifying)
- The code can be shared by multiple processes, as long as each has their own copy of the data and registers, including the instruction register

Sharing Pages: Example



3 user processes are using the editor program `ed`

Sharing Pages: Example



3 user processes are using the editor program `ed`

Only a single copy of the code of `ed` is actually loaded in main memory

Paging: PROs

- A big improvement over **relocation**
- Eliminates the problem of external fragmentation and therefore the need for compaction
- Allows code sharing among processes, reducing memory footprint
- Enables processes to run when they are partially loaded (more on this later…)

Paging: CONs

- Virtual/Physical address translation may be time consuming
- Hardware support like TLB cache is needed to make it fast enough
- OS has to be inevitably more complex
- Simple linear page stored in (kernel) memory can be prohibitive for large and highly sparse VAS systems

How Large Can Grow a Page Table?

Most modern computer systems support logical address spaces of 2^{32} to $2^{48} \div 2^{57}$ (i.e., 32-/64-bit machines)

How Large Can Grow a Page Table?

Most modern computer systems support logical address spaces of 2^{32} to $2^{48} \div 2^{57}$ (i.e., 32-/64-bit machines)

With a 2^{32} address space and 4KiB (2^{12}) page size, there are 2^{20} ~ 1 million page table entries (PTEs) in the page table

How Large Can Grow a Page Table?

Most modern computer systems support logical address spaces of 2^{32} to $2^{48} \div 2^{57}$ (i.e., 32-/64-bit machines)

With a 2^{32} address space and 4KiB (2^{12}) page size, there are 2^{20} ~ 1 million page table entries (PTEs) in the page table

At 4 bytes per PTE, this amounts to a 4 MiB page table, which may be too large to reasonably keep in contiguous memory

How Large Can Grow a Page Table?

Most modern computer systems support logical address spaces of 2^{32} to $2^{48} \div 2^{57}$ (i.e., 32-/64-bit machines)

With a 2^{32} address space and 4KiB (2^{12}) page size, there are 2^{20} ~ 1 million page table entries (PTEs) in the page table

At 4 bytes per PTE, this amounts to a 4 MiB page table, which may be too large to reasonably keep in contiguous memory

In a system with hundreds of processes this means a lot of space in RAM just to hold their *entire* page tables

How Large Can Grow a Page Table?

Most modern computer systems support logical address spaces of 2^{32} to $2^{48} \div 2^{57}$ (i.e., 32-/64-bit machines)

With a 2^{32} address space and 4KiB (2^{12}) page size, there are 2^{20} ~ 1 million page table entries (PTEs) in the page table

At 4 bytes per PTE, this amounts to a 4 MiB page table, which may be too large to reasonably keep in contiguous memory

In a system with hundreds of processes this means a lot of space in RAM just to hold their *entire* page tables

Plus, most of such space would be wasted as the process' VAS is sparse and only a fraction of its pages should be mapped to RAM

Can We Make Page Tables Smaller?

1. Using segmentation
2. Using more advanced page table structures beyond simple linear arrays

Can We Make Page Tables Smaller?

1. **Using segmentation**
2. Using more advanced page table structures beyond simple linear arrays

A Quick Step Back: Segmentation

- Most users (programmers) do not think of their programs as existing in one continuous linear address space

A Quick Step Back: Segmentation

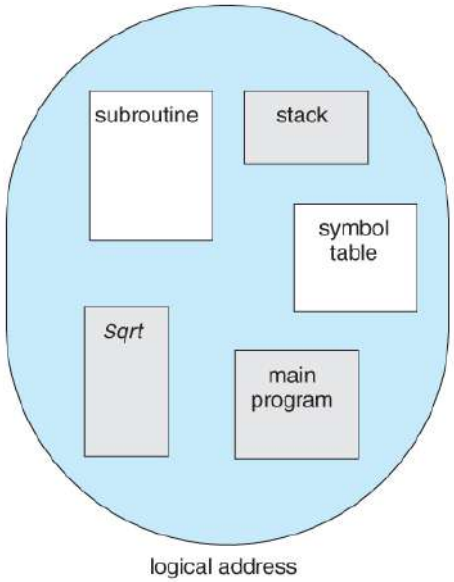
- Most users (programmers) do not think of their programs as existing in one continuous linear address space
- Rather they think of memory divided in multiple **segments**, each dedicated to a specific use, such as code, data, stack, heap, etc.

A Quick Step Back: Segmentation

- Most users (programmers) do not think of their programs as existing in one continuous linear address space
- Rather they think of memory divided in multiple **segments**, each dedicated to a specific use, such as code, data, stack, heap, etc.
- Memory segmentation supports this view by providing addresses with a **segment number** (mapped to a segment base address) and an **offset**

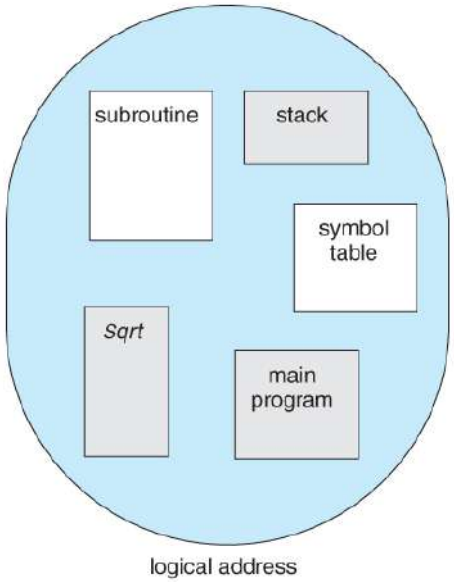
Segmentation: Example

A C compiler generating 5 segments for the user code, library code, global (static) variables, the stack, and the heap



Segmentation: Example

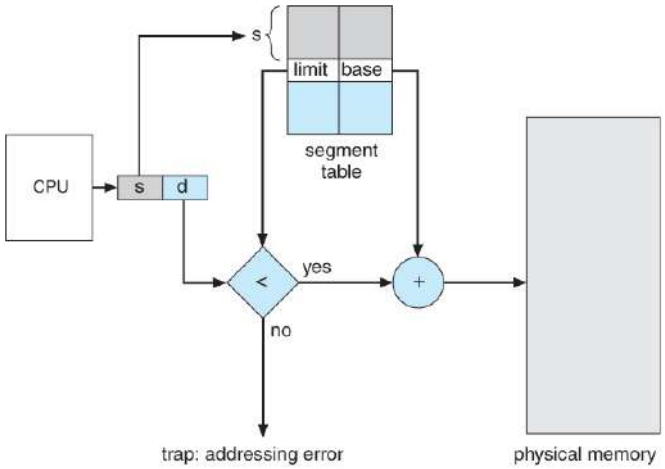
A C compiler generating 5 segments for the user code, library code, global (static) variables, the stack, and the heap



The compiler generates addresses identifying segments and offset

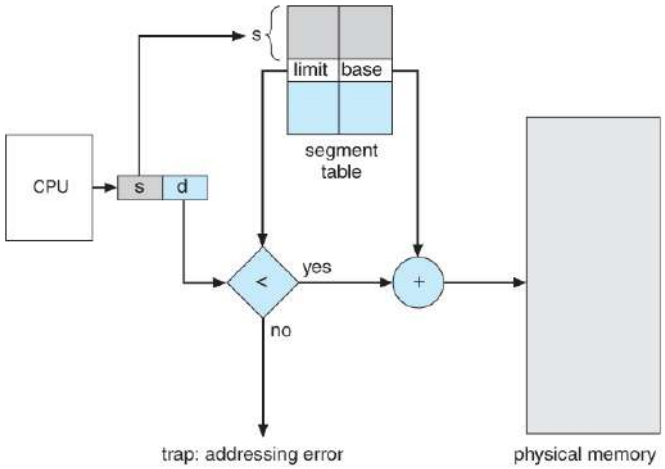
Segmentation Hardware

A segment table maps segment-offset addresses to physical addresses, and simultaneously checks for invalid addresses, using a system similar to the page tables and relocation base registers discussed previously



Segmentation Hardware

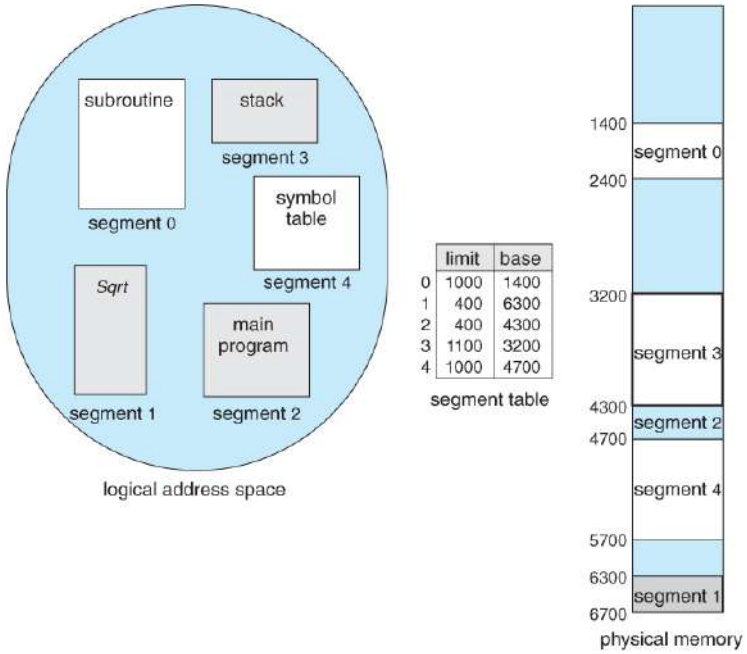
A segment table maps segment-offset addresses to physical addresses, and simultaneously checks for invalid addresses, using a system similar to the page tables and relocation base registers discussed previously



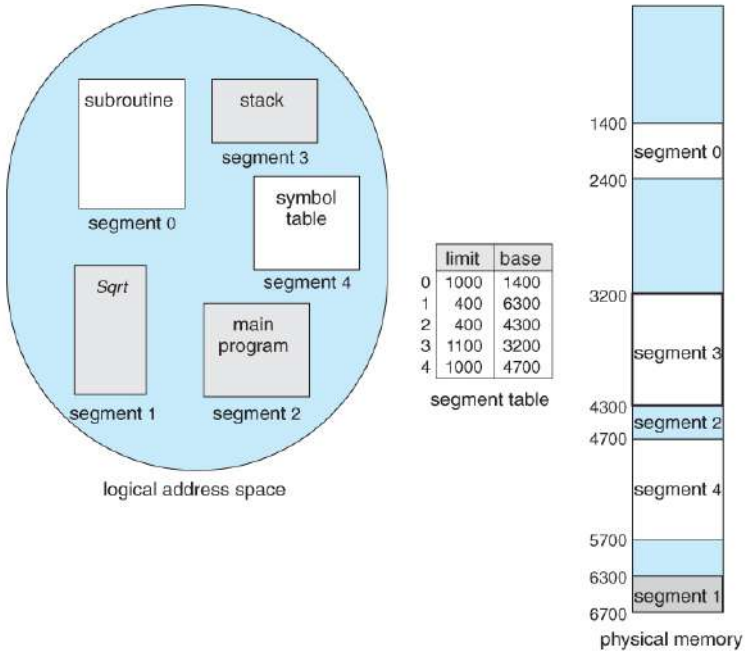
Note that we came back to the assumption that each segment is kept in contiguous memory and may be of different size…

Segment Table

Each entry contains a base address in memory, the length of the segment, plus additional protection information (e.g., sharing, read/write permissions)



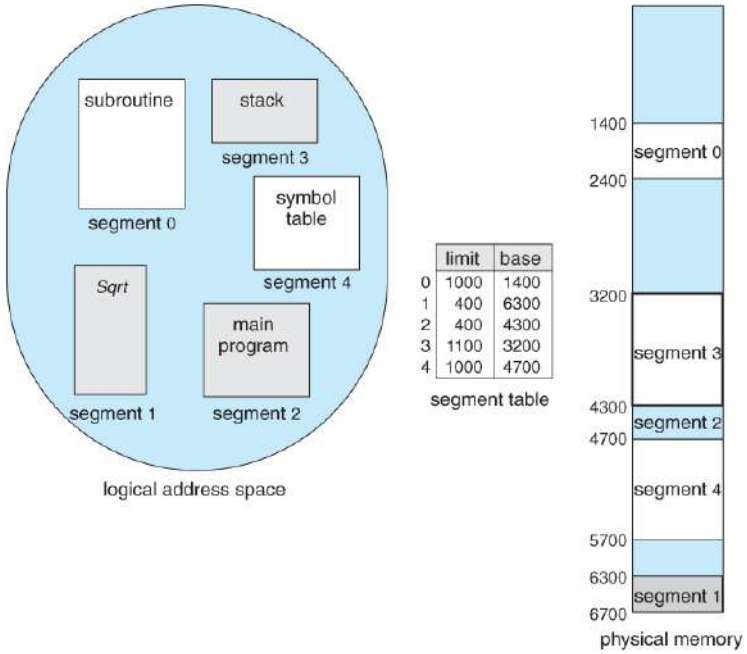
Segment Table



Each entry contains a base address in memory, the length of the segment, plus additional protection information (e.g., sharing, read/write permissions)

Segment Table can be stored using few base/limit hardware registers

Segment Table

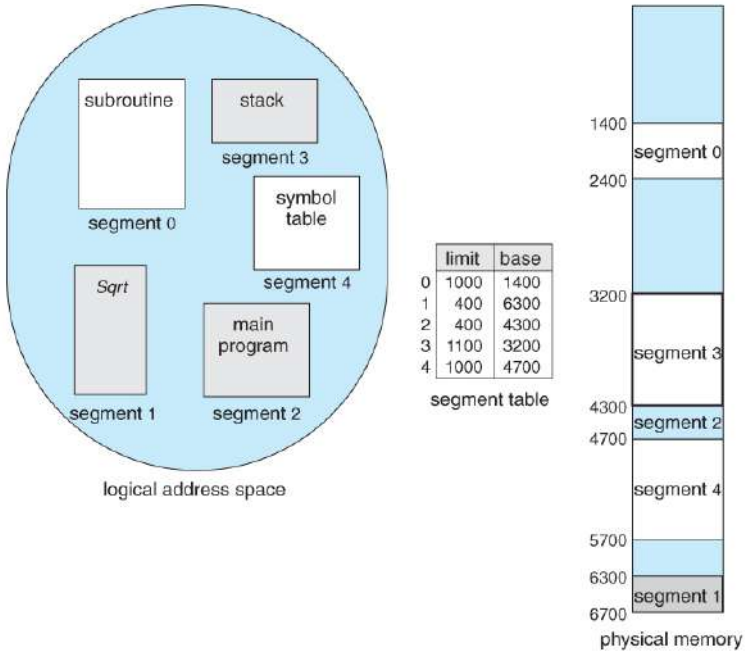


Each entry contains a base address in memory, the length of the segment, plus additional protection information (e.g., sharing, read/write permissions)

Segment Table can be stored using few base/limit hardware registers

Page Table cannot be stored using hw registers as there might be potentially too many page entries

Segment Table



Each entry contains a base address in memory, the length of the segment, plus additional protection information (e.g., sharing, read/write permissions)

Segment Table can be stored using few base/limit hardware registers

Page Table cannot be stored using hw registers as there might be potentially too many page entries

Segment Table, instead, must store a very limited amount of segments per process (3÷5)

Implementing (Basic) Segmentation

- Compiler generates virtual addresses whose top-most significant bits indicate the segment number

Implementing (Basic) Segmentation

- Compiler generates virtual addresses whose top-most significant bits indicate the segment number
- Segmentation can be combined both with static or dynamic relocation

Implementing (Basic) Segmentation

- Compiler generates virtual addresses whose top-most significant bits indicate the segment number
- Segmentation can be combined both with static or dynamic relocation
- Each segment is allocated a contiguous block of memory
  - External fragmentation can be an issue again!

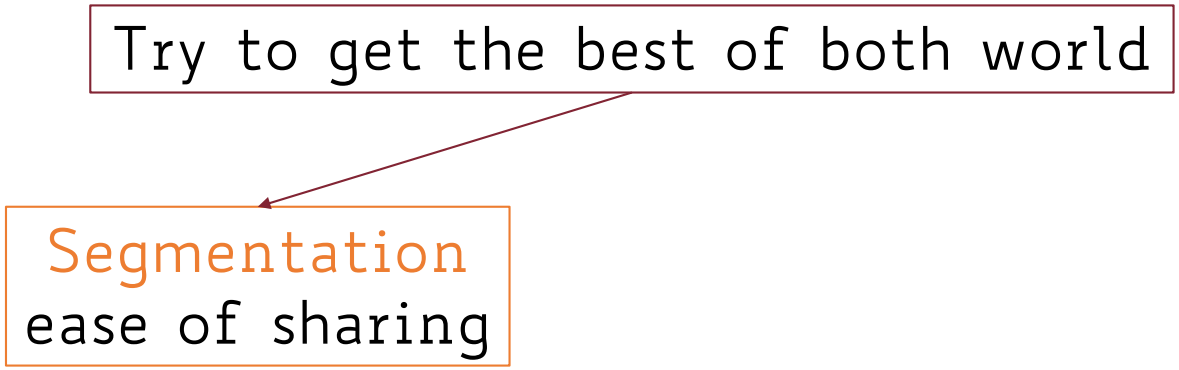
Implementing (Basic) Segmentation

- Compiler generates virtual addresses whose top-most significant bits indicate the segment number
- Segmentation can be combined both with static or dynamic relocation
- Each segment is allocated a contiguous block of memory
  - External fragmentation can be an issue again!
- Additional HW (like TLB cache) might be needed if programs use many logical segments

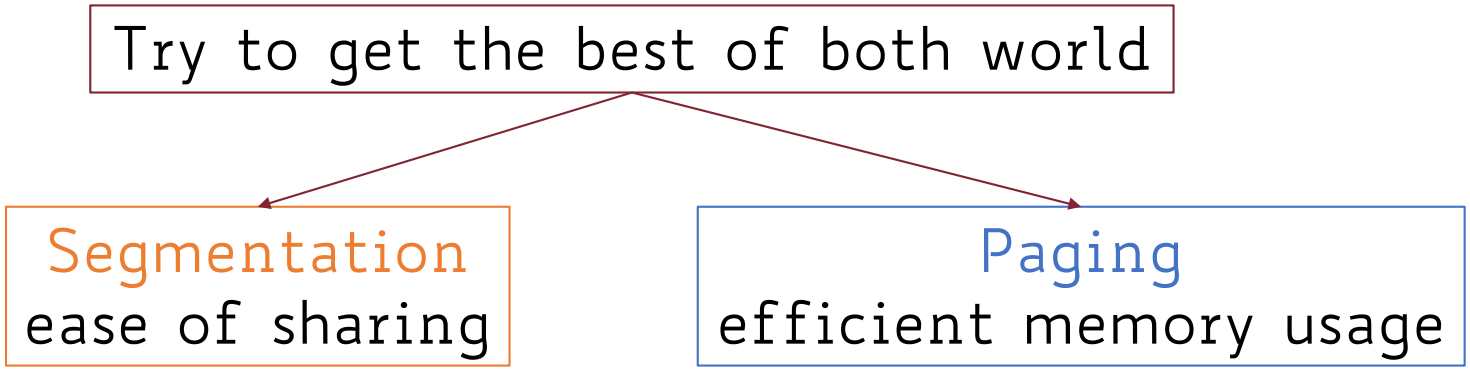
Combine Segmentation with Paging

Try to get the best of both world

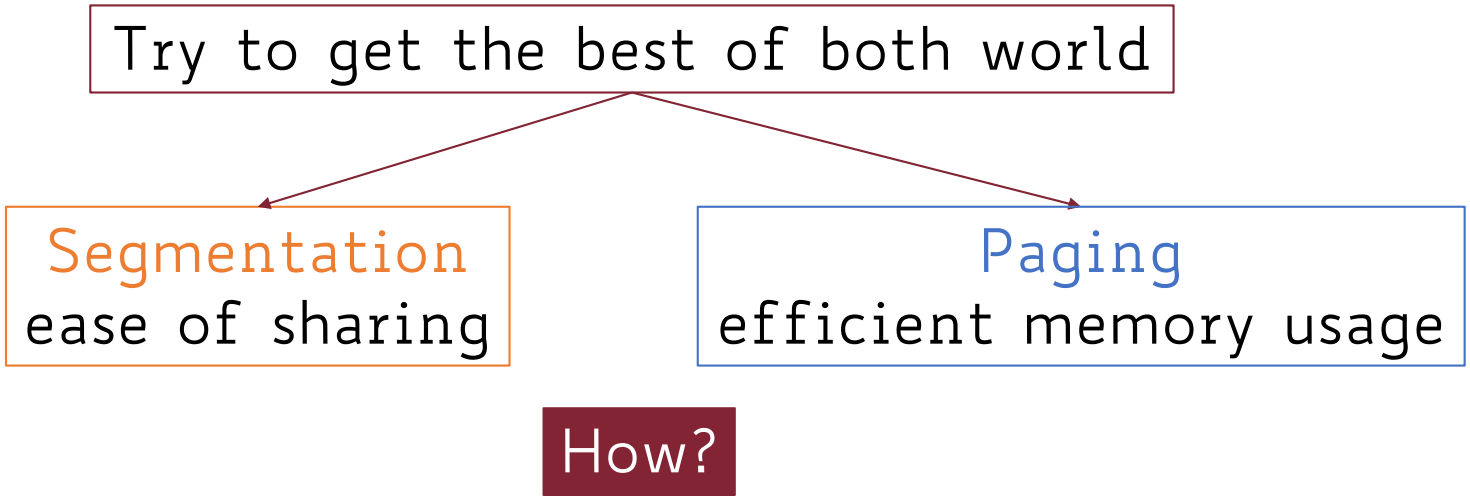
Combine Segmentation with Paging



Combine Segmentation with Paging

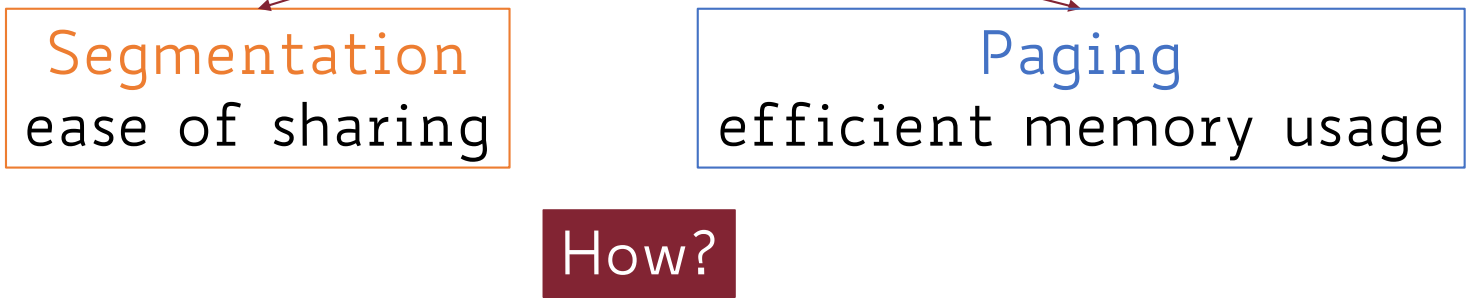


Combine Segmentation with Paging



Combine Segmentation with Paging

Try to get the best of both world



How?

Apply paging to segments!

Combine Segmentation with Paging

- Each process' virtual address space is seen as a collection of segments (i.e., logical units of arbitrary size)

Combine Segmentation with Paging

- Each process' virtual address space is seen as a collection of segments (i.e., logical units of arbitrary size)
- Physical address space is still seen as a sequence of fixed-size frames

Combine Segmentation with Paging

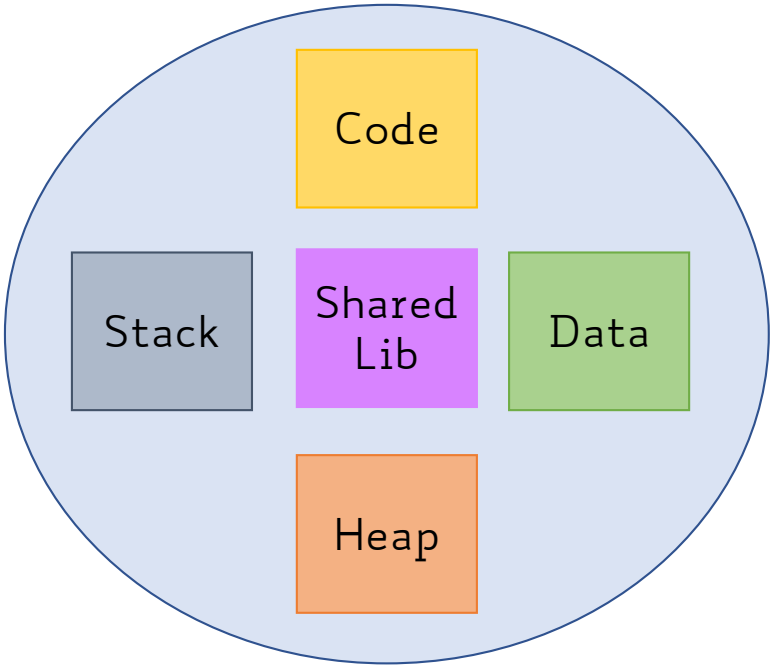
- Each process' virtual address space is seen as a collection of segments (i.e., logical units of arbitrary size)
- Physical address space is still seen as a sequence of fixed-size frames
- Segments are usually larger than physical page frames

Combine Segmentation with Paging

- Each process' virtual address space is seen as a collection of segments (i.e., logical units of arbitrary size)
- Physical address space is still seen as a sequence of fixed-size frames
- Segments are usually larger than physical page frames

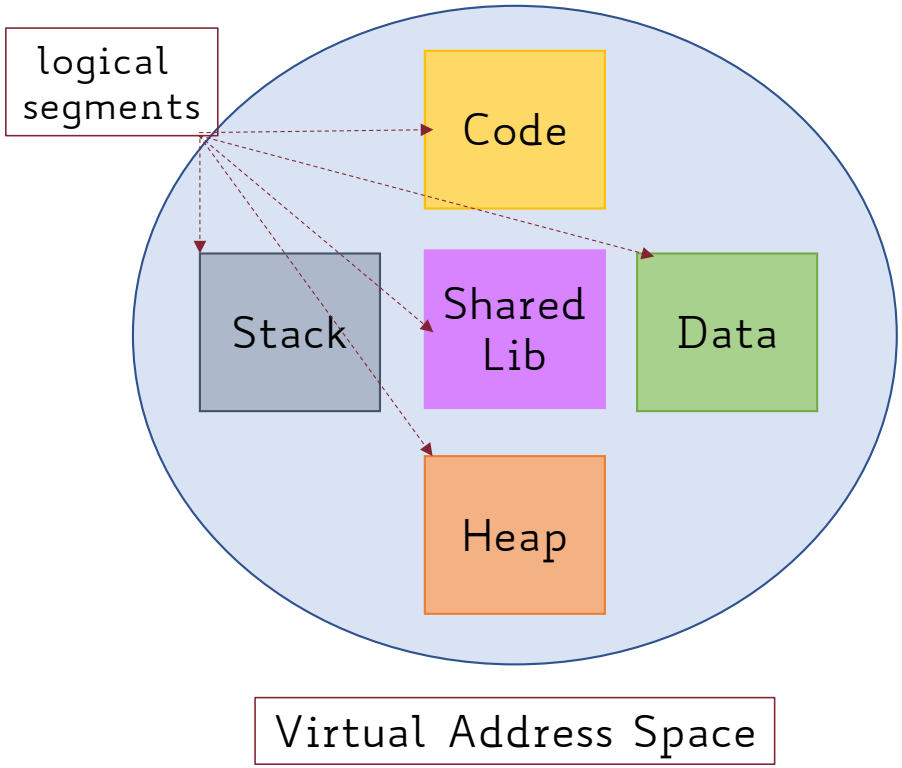
Map a logical segment onto multiple page frames

Paging Logical Segments

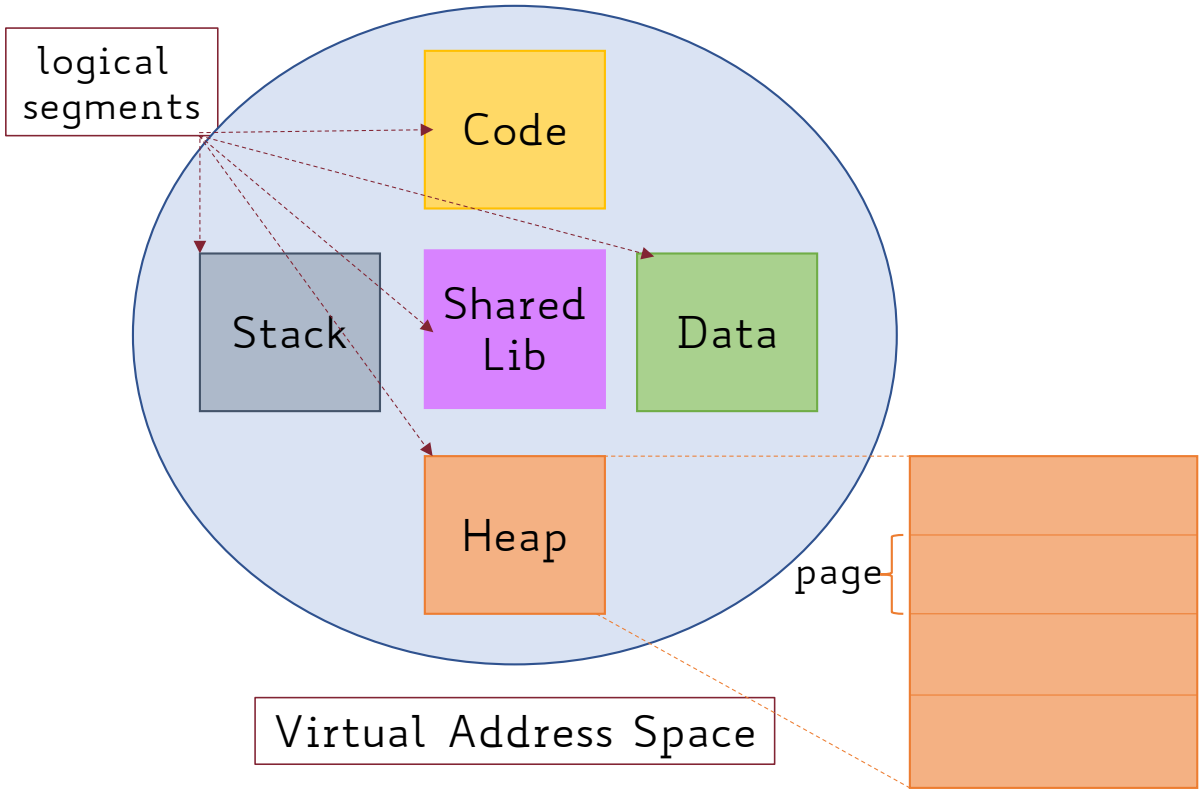


Virtual Address Space

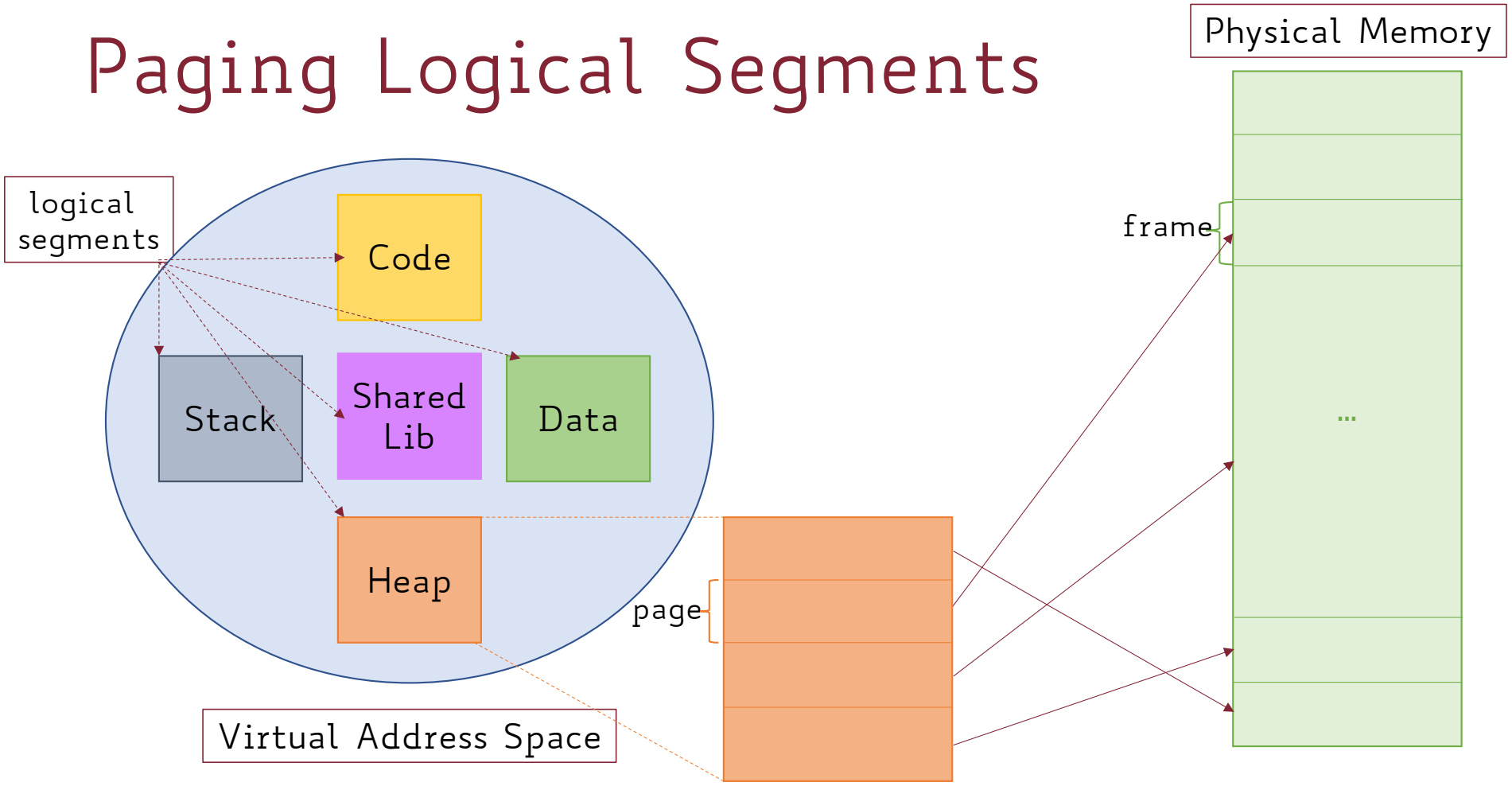
Paging Logical Segments



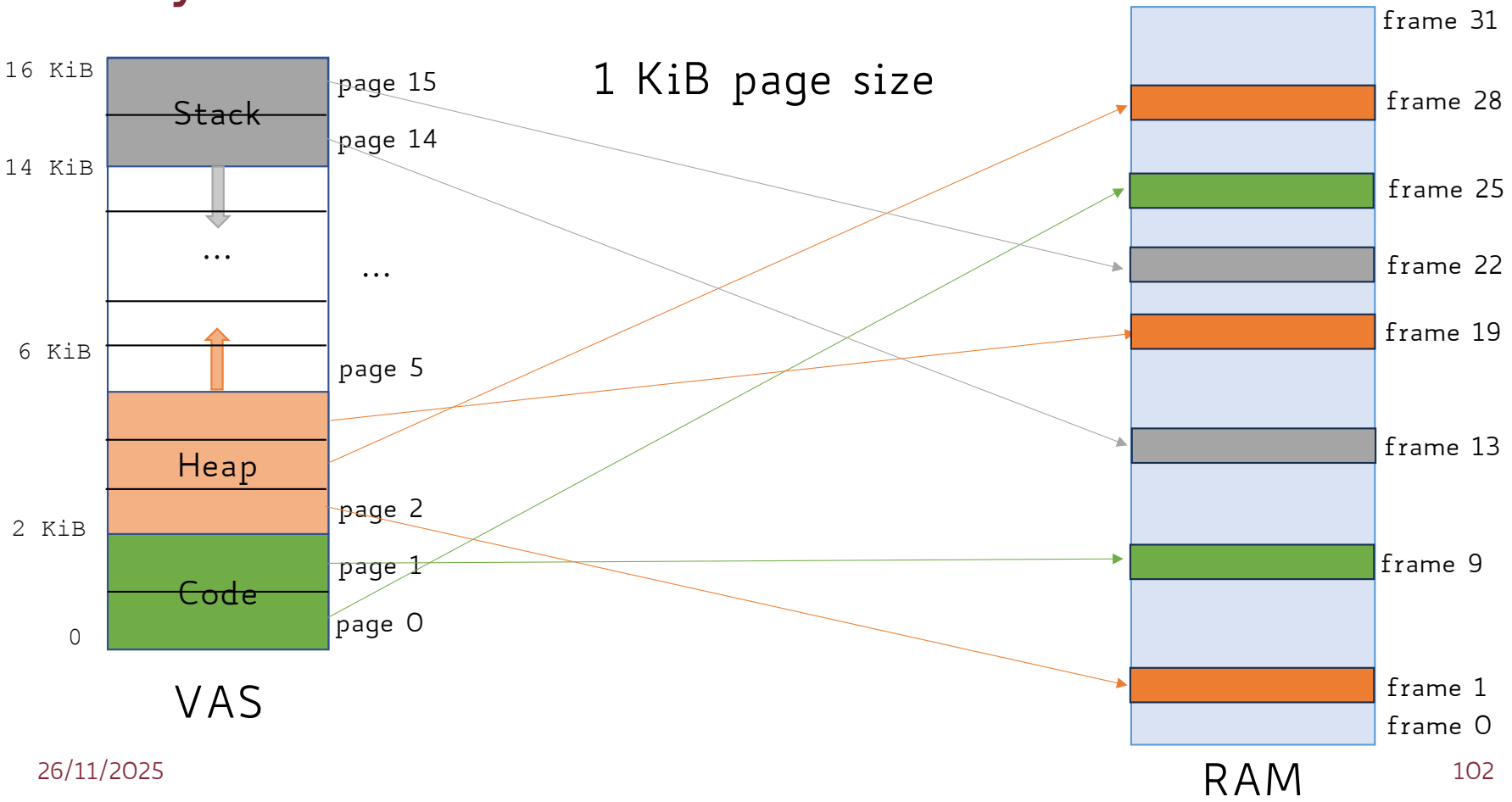
Paging Logical Segments



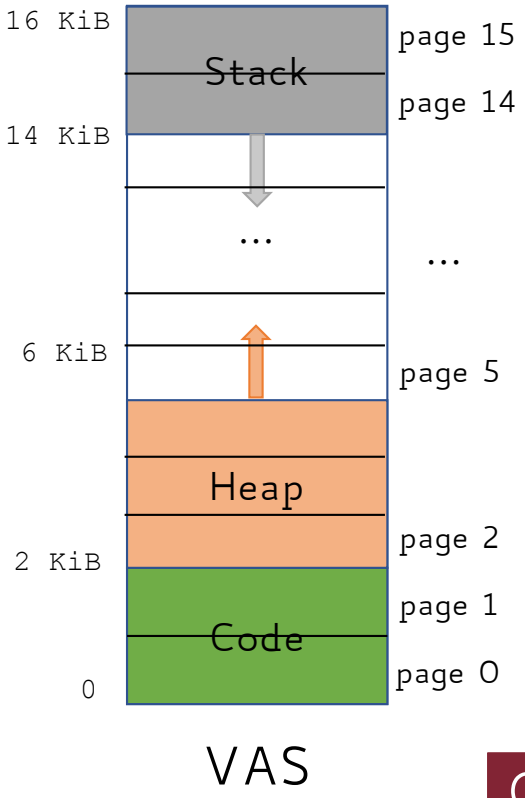
Paging Logical Segments



Why Does This Make PT Smaller?



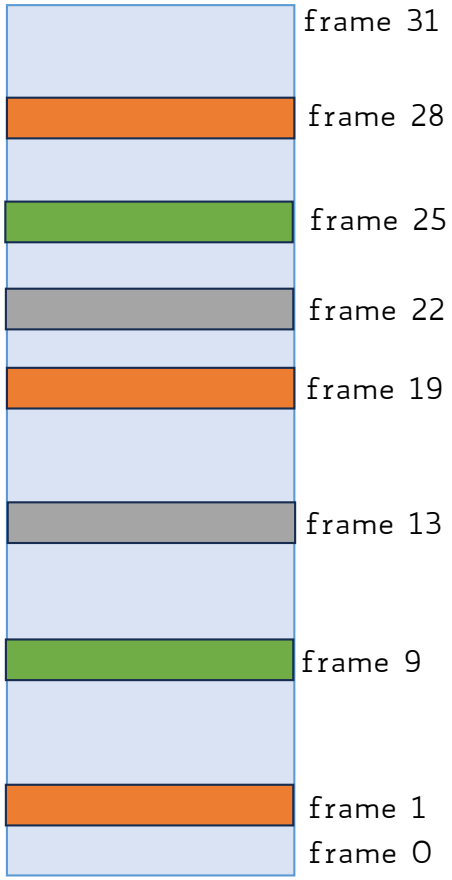
Why Does This Make PT Smaller?



VAS

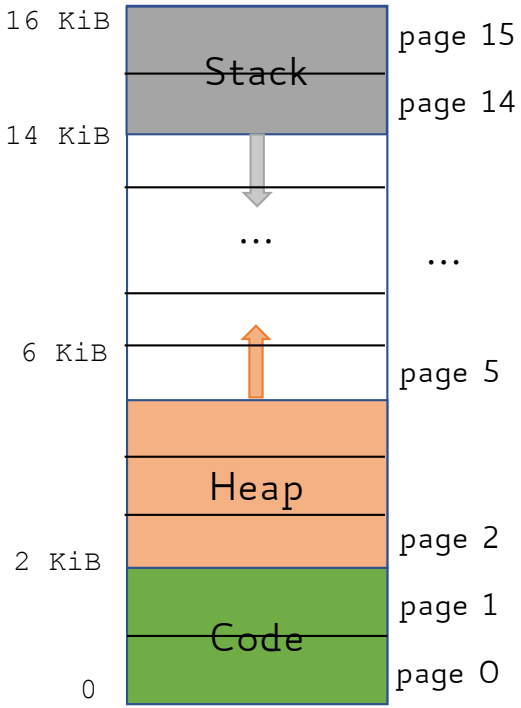
	PFN	valid	prot.	pres.
0	25	1	r-x	1
1	9	1	r-x	1
2	1	1	rw-	1
3	28	1	rw-	1
4	19	1	rw-	1
5	-	0	-	-
6	-	0	-	-
...				
13	-	0	-	-
14	13	1	rw-	1
15	22	1	rw-	1

One Single Linear Page Table



RAM

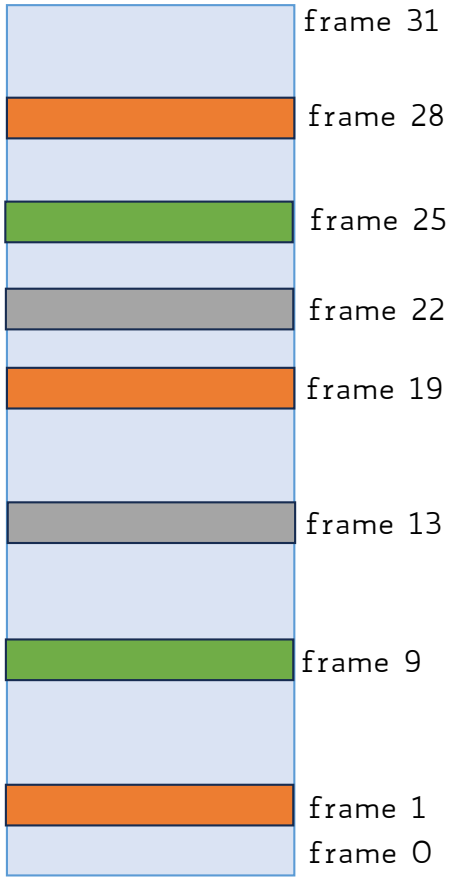
Why Does This Make PT Smaller?



VAS

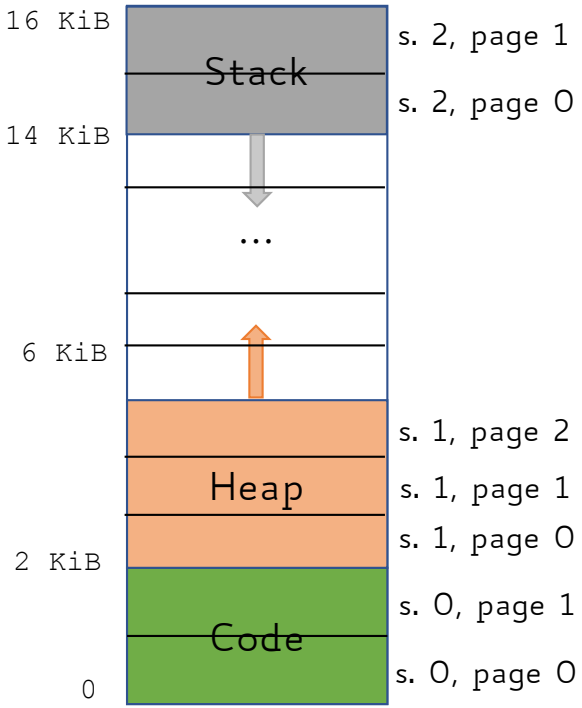
	PFN	valid	prot.	pres.
0	25	1	r-x	1
1	9	1	r-x	1
2	1	1	rw-	1
3	28	1	rw-	1
4	19	1	rw-	1
5	-	0	-	-
6	-	0	-	-
	...			
13	-	0	-	-
14	13	1	rw-	1
15	22	1	rw-	1

Wasted!



RAM

Why Does This Make PT Smaller?



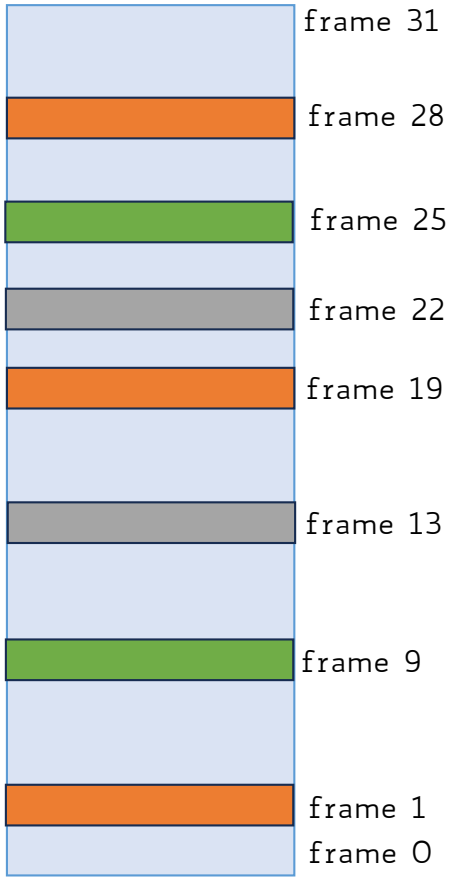
VAS

	PFN	valid	prot.	pres.
0	25	1	r-x	1
1	9	1	r-x	1

	PFN	valid	prot.	pres.
0	1	1	rw-	1
1	28	1	rw-	1
2	19	1	rw-	1

	PFN	valid	prot.	pres.
0	13	1	rw-	1
1	22	1	rw-	1

3 Linear Page Tables (one per segment)



RAM

Per-Segment Page Tables

- PRO:
  - Memory Saving → unallocated pages between stack and heap won't require (invalid) PTEs as with a single linear page table

Per-Segment Page Tables

- PRO:
  - Memory Saving → unallocated pages between stack and heap won't require (invalid) PTEs as with a single linear page table
- CONs:
  - Sparse Segments → sparse heap or weird stack allocation may still cause memory waste

```
int *arr = malloc(4 TB);
arr[0] = 1;
arr[500000000] = 2;
```

```
void f() {
    char big[4 * 1024 * 1024];   // 4 MB array
    big[0] = 1;                  // first page touched only
}
```

Per-Segment Page Tables

- PRO:
  - Memory Saving → unallocated pages between stack and heap won't require (invalid) PTEs as with a single linear page table
- CONs:
  - Sparse Segments → sparse heap or weird stack allocation may still cause memory waste

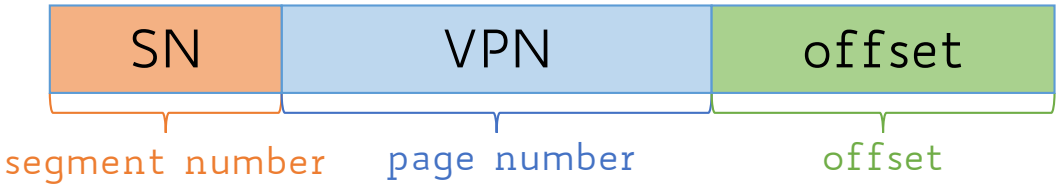
```
int *arr = malloc(4 TB);
arr[0] = 1;
arr[500000000] = 2;
```

```
void f() {
    char big[4 * 1024 * 1024];   // 4 MB array
    big[0] = 1;                  // first page touched only
}
```

  - External fragmentation → page tables may have different sizes

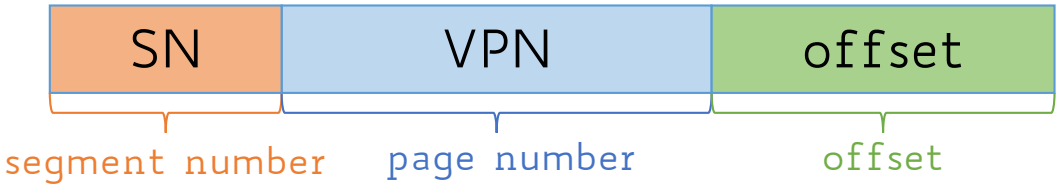
Address Translation with Segmented Paging

A virtual address now becomes:



Address Translation with Segmented Paging

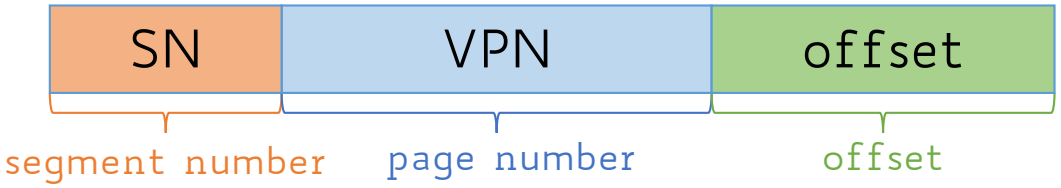
A virtual address now becomes:



- The segment number (SN) indexes into the **segment table**, which contains the base address of the **page table** for *that* segment

Address Translation with Segmented Paging

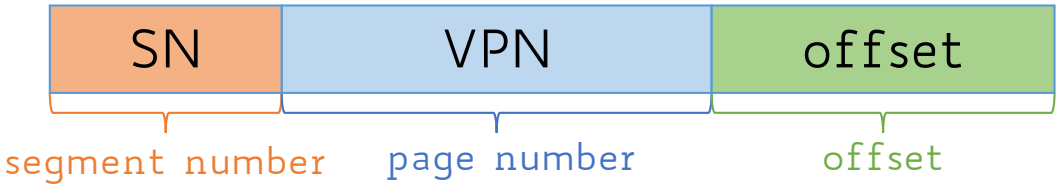
A virtual address now becomes:



- The segment number (SN) indexes into the **segment table**, which contains the base address of the **page table** for *that* segment
- Check the VPN against the limit of the segment (i.e., the **number of PTEs** in the page table for *that* segment)

Address Translation with Segmented Paging

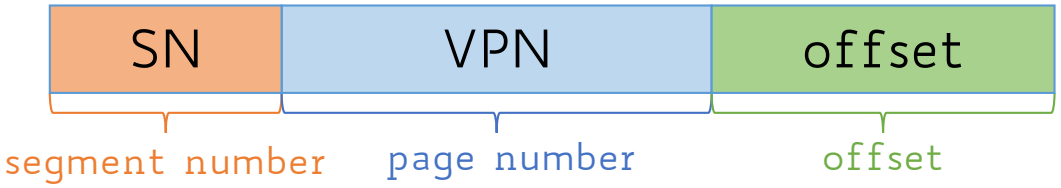
A virtual address now becomes:



- Use the VPN to index the page table to get the physical frame number (PFN)

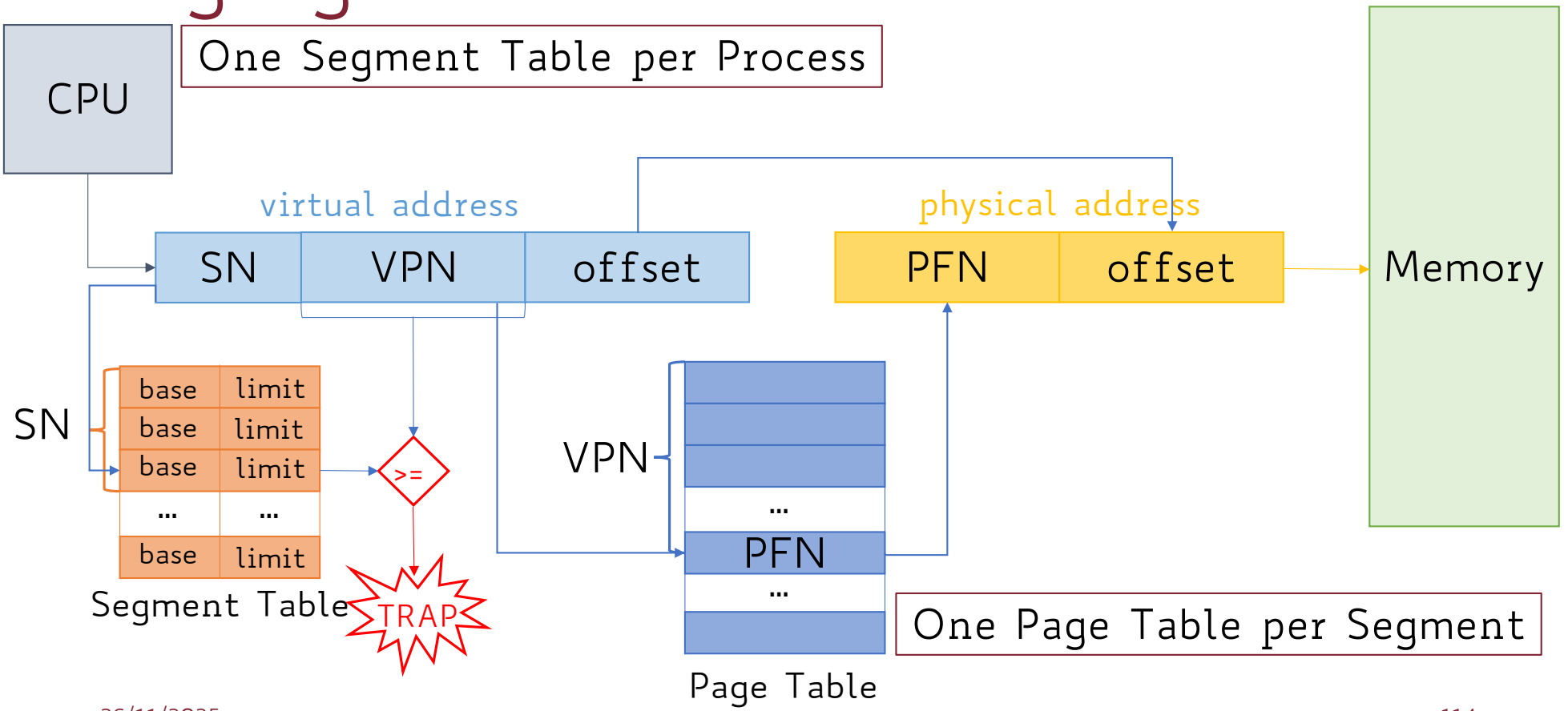
Address Translation with Segmented Paging

A virtual address now becomes:



- Use the VPN to index the page table to get the physical frame number (PFN)
- Add the frame number (PFN) to the offset to get the physical address

Address Translation with Segmented Paging



Segmented Paging: Implementation

Where are segment tables and page tables stored?

Segmented Paging: Implementation

Where are segment tables and page tables stored?

Option 1

segment tables in a small number of registers

page tables in main memory with TLB cache

Segmented Paging: Implementation

Where are segment tables and page tables stored?

Option 2

both segment tables and page tables in main memory with TLB cache

TLB lookup done using segment index and page index

Segmented Paging: Implementation

Where are segment tables and page tables stored?

Option 1

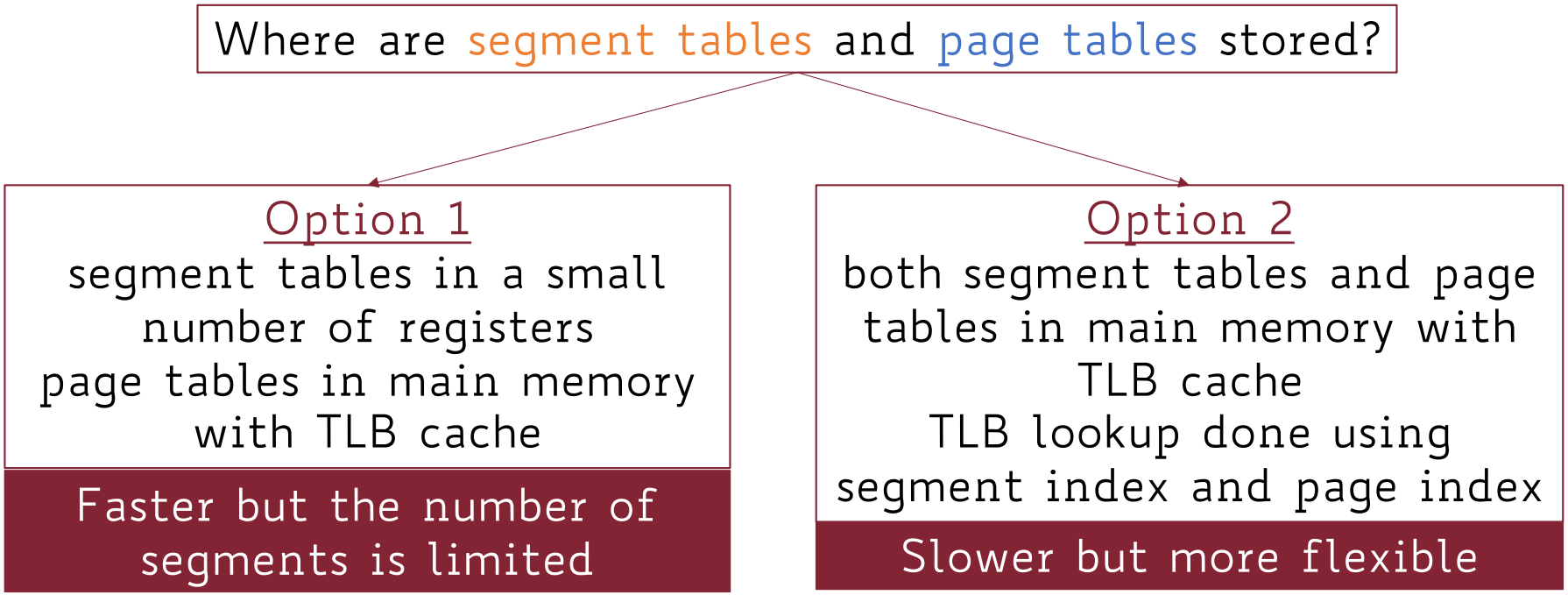
segment tables in a small number of registers
page tables in main memory with TLB cache

Faster but the number of segments is limited

Option 2

both segment tables and page tables in main memory with TLB cache
TLB lookup done using segment index and page index

Segmented Paging: Implementation



Segmented Paging Hardware: Practical Example 3

Suppose a physical memory of 1024 addressable words (assuming 1 word = 1 byte)
Frame size is 64 words (i.e., 64 bytes)
Page table size (i.e., number of entries) is thus 1024 bytes/64 bytes per frame = 16
8 logical segments

Segmented Paging Hardware: Practical Example 3

Suppose a physical memory of 1024 addressable words (assuming 1 word = 1 byte)

Frame size is 64 words (i.e., 64 bytes)

Page table size (i.e., number of entries) is thus 1024 bytes/64 bytes per frame = 16

8 logical segments

Q1

How many bits are therefore needed for the physical address?

Segmented Paging Hardware: Practical Example 3

Suppose a physical memory of 1024 addressable words (assuming 1 word = 1 byte)

Frame size is 64 words (i.e., 64 bytes)

Page table size (i.e., number of entries) is thus 1024 bytes/64 bytes per frame = 16

8 logical segments

Q1

How many bits are therefore needed for the physical address?

R1

10 bits to address M = 1024/1 = 1024 1-byte words

Segmented Paging Hardware: Practical Example 3

Q2

How many bits are therefore needed for the virtual address?

Segmented Paging Hardware: Practical Example 3

Q2

How many bits are therefore needed for the virtual address?

R2

3 bits to address 8 logical segments (s)

4 bits to address 16 entries of the page table

6 bits to address 64 individual words (i.e., bytes) within each page

Segmented Paging Hardware: Practical Example 3

Q2

How many bits are therefore needed for the virtual address?

R2

3 bits to address 8 logical segments (s)

4 bits to address 16 entries of the page table

6 bits to address 64 individual words (i.e., bytes) within each page

13 bits (virtual address) vs. 10 bits (physical address)

Sharing Pages and Segments

- We already showed individual pages can be shared across processes by copying page entries

Sharing Pages and Segments

- We already showed individual pages can be shared across processes by copying page entries
- Segments can also be shared by sharing segment table entries:

Sharing Pages and Segments

- We already showed individual pages can be shared across processes by copying page entries
- Segments can also be shared by sharing segment table entries:
  - Two or more processes map the shared segment on its own segment table (one segment table per process) to the same page table

Sharing Pages and Segments

- We already showed individual pages can be shared across processes by copying page entries
- Segments can also be shared by sharing segment table entries:
  - Two or more processes map the shared segment on its own segment table (one segment table per process) to the same page table
  - Since there is one page table for each segment, we can share a segment by simply sharing the page table this points to

Sharing Pages and Segments

- We already showed individual pages can be shared across processes by copying page entries
- Segments can also be shared by sharing segment table entries:
  - Two or more processes map the shared segment on its own segment table (one segment table per process) to the same page table
  - Since there is one page table for each segment, we can share a segment by simply sharing the page table this points to
- Even more flexible!

Segmented Paging: Benefits and Costs

- **Benefits:**
  - Merge compiler and OS view of memory
  - Flexibility
  - Less memory waste for sparse VAS w.r.t linear page tables
  - Sharing memory between processes

Segmented Paging: Benefits and Costs

- **Benefits:**
  - Merge compiler and OS view of memory
  - Flexibility
  - Less memory waste for sparse VAS w.r.t linear page tables
  - Sharing memory between processes
- **Costs:**
  - Slower context switches (why?)
  - Slower address translation (why?)

Internal Fragmentation Still There…

- Paging allows getting rid of external fragmentation

Internal Fragmentation Still There…

- Paging allows getting rid of external fragmentation
- Internal fragmentation is still an issue

Internal Fragmentation Still There…

- Paging allows getting rid of external fragmentation
- Internal fragmentation is still an issue
- On pure paging (no segmented), assuming process' memory footprint is random, internal fragmentation amounts to 0.5 page per process (on average)

Internal Fragmentation Still There…

- Paging allows getting rid of external fragmentation
- Internal fragmentation is still an issue
- On pure paging (no segmented), assuming process' memory footprint is random, internal fragmentation amounts to 0.5 page per process (on average)
- On segmented paging, we can lose 0.5 page per process' segment

Internal Fragmentation Still There…

- Paging allows getting rid of external fragmentation
- Internal fragmentation is still an issue
- On pure paging (no segmented), assuming process' memory footprint is random, internal fragmentation amounts to 0.5 page per process (on average)
- On segmented paging, we can lose 0.5 page per process' segment
- The larger the page size the higher the chance of internal fragmentation

Can We Make Page Tables Smaller?

1. Using segmentation
2. **Using more advanced page table structures beyond simple linear arrays**

Multi-Level Page Tables

- Turns linear page tables into a **tree**
- **Basic Idea:**
  - Divide up the page table into page-sized units (i.e., applying paging to the page table!)
  - If an entire page of PTEs is invalid do not allocate that page at all!
  - Use a new structure (**page directory**) to track if a page of the page table is valid

Multi-Level Page Tables

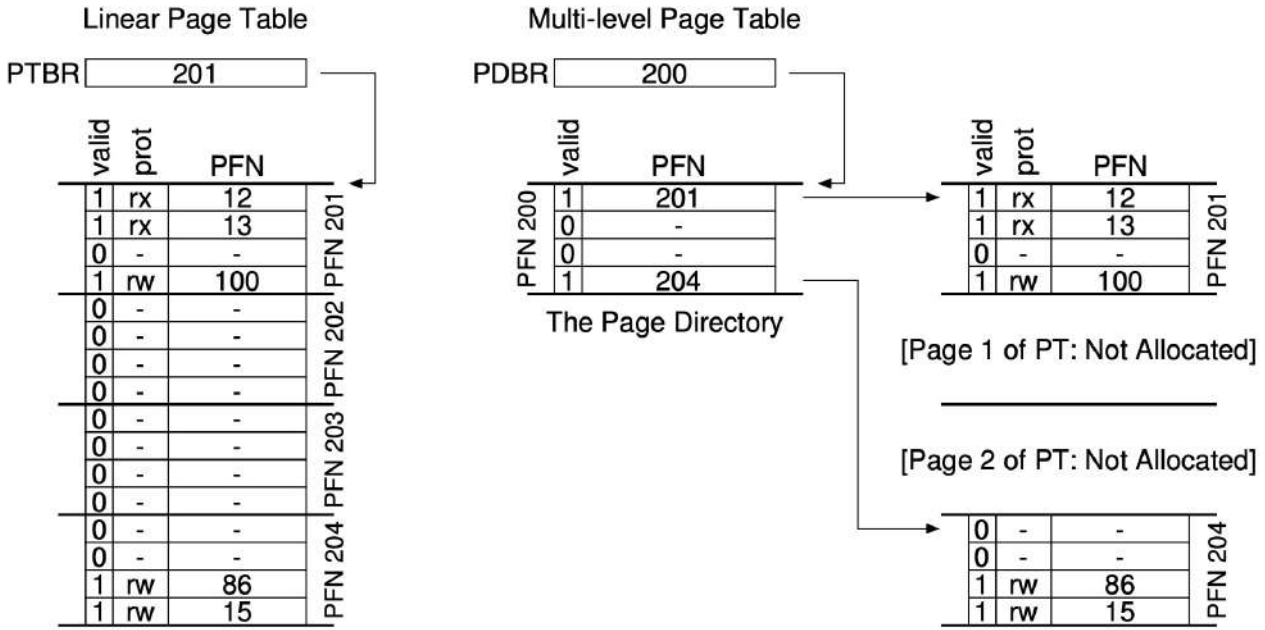


Figure 20.3: **Linear (Left) And Multi-Level (Right) Page Tables**

source: Operating Systems: Three Easy Pieces

Page Directory

- Consists of a number of page directory entries (PDEs)
- In a two-level table: one PDE per page of the page table
- Each PDE at least contains:
  - A valid bit
  - A page frame number (similar to a PTE)
- If the PDE valid bit is set, at least one of the pages that the PDE refers to is valid

Multi-Level Page Tables: PROs

- Allocates only page table space that is proportional to the (sparse) address space used
- If properly designed, each portion of the page table fits the size of a page
- For linear (non-paged) page tables, this array must be allocated contiguously in physical memory
- With multi-level paging, pages of the same page table can be allocated scatteredly in RAM

Multi-Level Page Tables: CONs

- When combined with TLB in case of a cache miss, two extra memory accesses are needed
  - The first one to get to the PDE
  - The second one to get to the PTE
- Page table lookup on TLB miss is typically handled by the OS (rather than just via HW through the MMU)
  - Increased complexity

Beyond Two-Level Page Tables

- Suppose m = 30-bit VAS and page size = 512 B

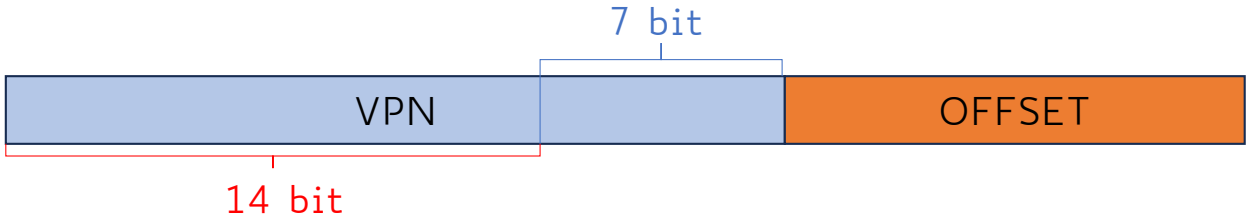
Goal: Divide the entire page table into chunks, so that each chunk fits one page

How many levels do we need?

Beyond Two-Level Page Tables

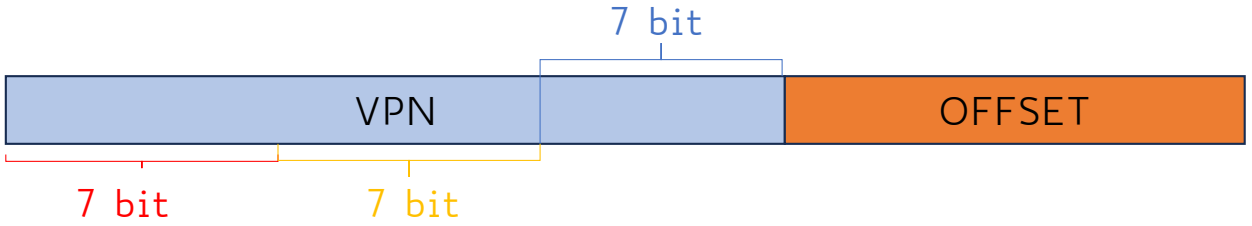
- Let's start by determining how many PTEs fit within a page
  - Given our page size is 512 B and each PTE requires 4 B
  - A single page will contain 512 B/4 B = 128 PTEs
- That means that to index within a single page of the page table we need $\log_2(128)$ bits = 7 bits
  - The least significant 7 bits of the VPN are used as index

Beyond Two-Level Page Tables



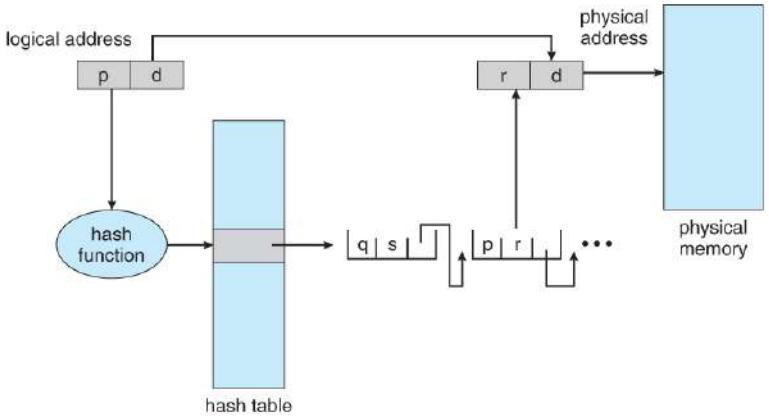
- We are left with 21-7 = 14 bits for the page directory
- 2^{14} page directory entries each of size 4 B would require $2^{14} * 2^2 = 2^{16}$ B
- 2^{16} B is way larger than one page size (2^9 B = 512 B)
- The page directory would require 2^{16} B/2^9 B = 2^7 = 128 pages rather than one as our goal was!

Beyond Two-Level Page Tables



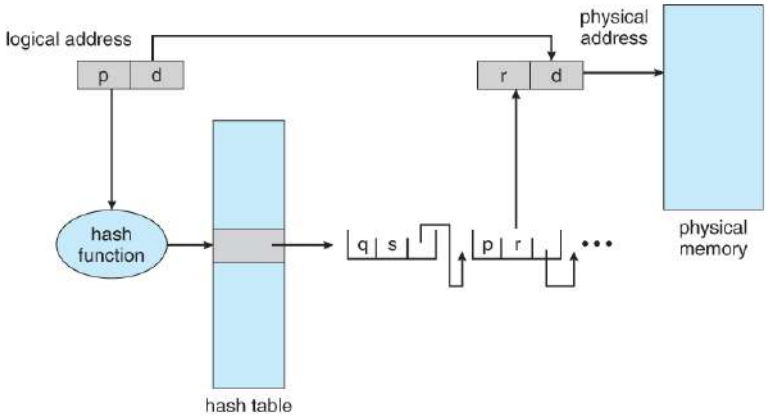
- **Solution:** We add another level to our tree!
- The first 7 bits are used to index the root (level 1) page directory (PD1) and fetch the corresponding PDE
  - This PDE contains the PFN of the base of level-2 page directory (PD2)
- The second 7 bits are used to index in PD2 and fetch the PDE
  - This PDE contains the PFN of the base of level-3 page table (PT)
- The third 7 bits are used to index in PT and fetch the final PTE

Advanced Paging: Hashed Page Table



Use hash tables to store highly sparse page tables

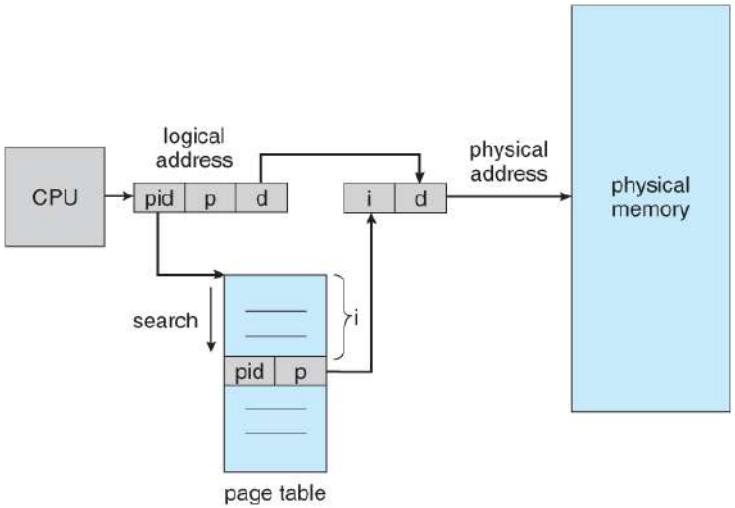
Advanced Paging: Hashed Page Table



Use hash tables to store highly sparse page tables

Indexing via hash function rather than integers

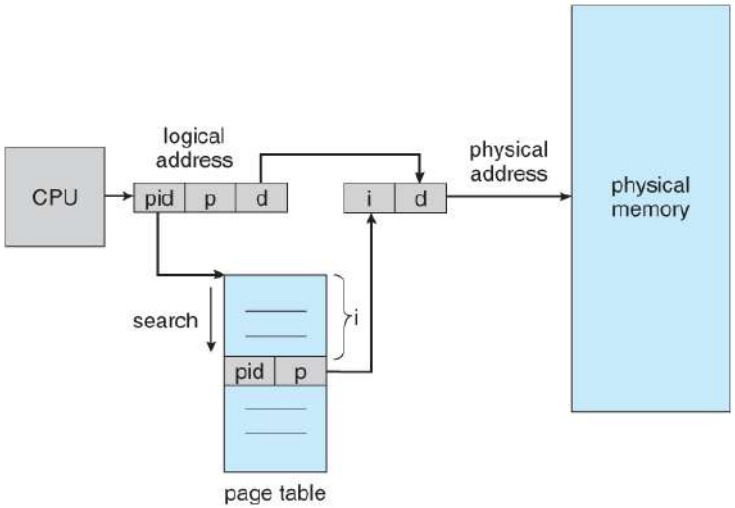
Advanced Paging: Inverted Page Table



An inverted page table lists all of the frames currently loaded in memory, for all processes

Instead of a table listing all of the pages for a particular process

Advanced Paging: Inverted Page Table



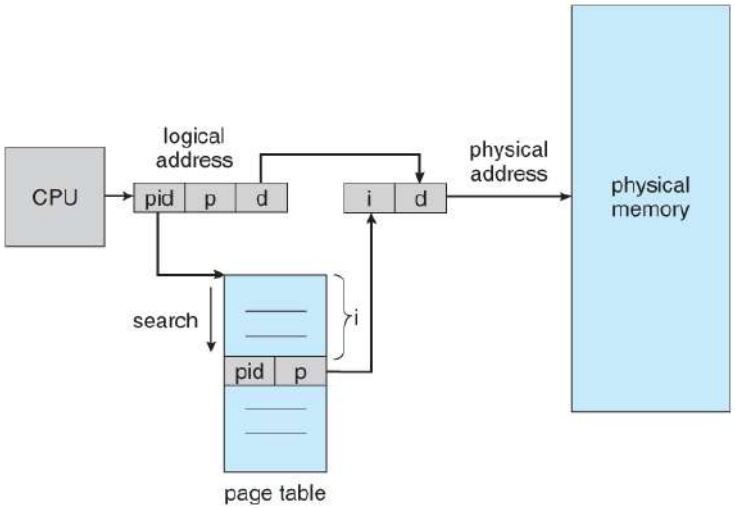
An inverted page table lists all of the frames currently loaded in memory, for all processes

Instead of a table listing all of the pages for a particular process

Access to an inverted page table can be slow (linear search)

Hashing the table speeds up the search process

Advanced Paging: Inverted Page Table



An inverted page table lists all of the frames currently loaded in memory, for all processes

Instead of a table listing all of the pages for a particular process

Access to an inverted page table can be slow (linear search)

Hashing the table speeds up the search process

Inverted page tables do not easily allow mapping multiple logical pages to a common physical frame (page sharing)

Each frame is mapped to exactly one process

Summary

- **Relocation** using base and limit registers
  - Simple yet inflexible

Summary

- **Relocation** using base and limit registers
  - Simple yet inflexible
- **Segmentation**
  - Compiler's logical view of memory presented to the OS
  - Segment tables tend to be small enough to be stored in registers
  - Contiguous memory allocation is expensive and complicated (first-fit, best-fit, or worst-fit)
  - Compaction is needed to solve external fragmentation

Summary

- **Paging**
  - Simplifies memory allocation by relaxing contiguous assumption
  - Each logical page can be allocated to any physical frame
  - Page tables can be extremely large

Summary

- **Segmentation + Paging**
  - One page table per segment vs. of one page table per VAS
  - Segments can still be sparse (e.g., heap)
  - Sharing either at the segment or at the page level
  - Might increase internal fragmentation over pure paging
  - 2 lookups per memory reference are needed

Summary

- **Multi-Level Paging**
  - Paging applied to the page table!
  - From linear page table (array) to a tree
  - Goal: Create as many levels as needed to fit each page table chunk into a page
  - Minimize memory waste (only touched pages are materialized)
  - On a TLB miss, as many memory accesses as the number of levels