Sistemi Operativi

Corso di Laurea in Informatica

a.a. 2019-2020

SAPIENZA
UNIVERSITÀ DI ROMA

Gabriele Tolomei

Dipartimento di Informatica

Sapienza Università di Roma

tolomei@di.uniroma1.it

Problems Seen So Far

- Contiguous allocation
  - Hard to grow or shrink process memory

Problems Seen So Far

- Contiguous allocation
  - Hard to grow or shrink process memory
- Fragmentation
  - Frequent compaction needed

Problems Seen So Far

- Contiguous allocation
  - Hard to grow or shrink process memory
- Fragmentation
  - Frequent compaction needed
- Process entirely loaded
  - Swapping helps but it may be too inefficient

Paging

- A memory management scheme that addresses the problems above

Paging

- A memory management scheme that addresses the problems above
- The logical address space of a process is still **contiguous** but it is divided into **fixed-size blocks**, called **pages**

Paging

- A memory management scheme that addresses the problems above
- The logical address space of a process is still **contiguous** but it is divided into **fixed-size blocks**, called **pages**
- Contiguous allocation is no longer required as logical pages can be mapped to **non-contiguous** physical **frames**

Paging

- A memory management scheme that addresses the problems above
- The logical address space of a process is still **contiguous** but it is divided into **fixed-size blocks**, called **pages**
- Contiguous allocation is no longer required as logical pages can be mapped to **non-contiguous** physical **frames**
- External fragmentation is eliminated because pages have fixed size
  - Internal fragmentation may still occur though

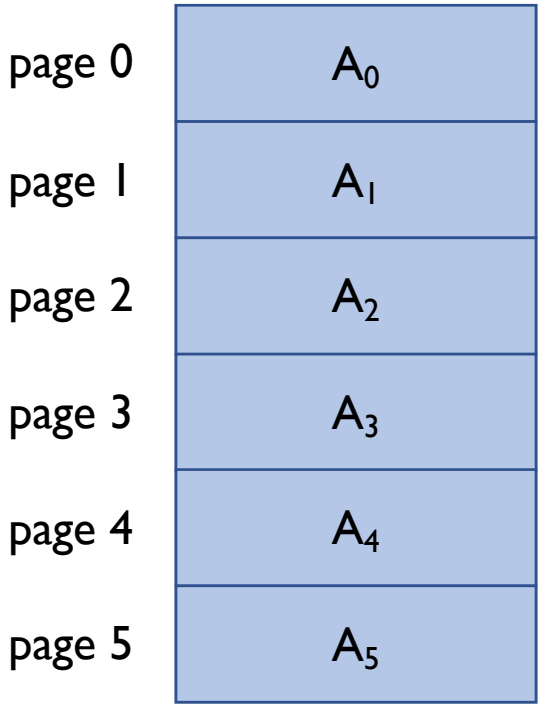
Paging

- A memory management scheme that addresses the problems above
- The logical address space of a process is still **contiguous** but it is divided into **fixed-size blocks**, called **pages**
- Contiguous allocation is no longer required as logical pages can be mapped to **non-contiguous** physical **frames**
- External fragmentation is eliminated because pages have fixed size
  - Internal fragmentation may still occur though
- Keep only those frames that are actually being used by a process

90/10 Rule

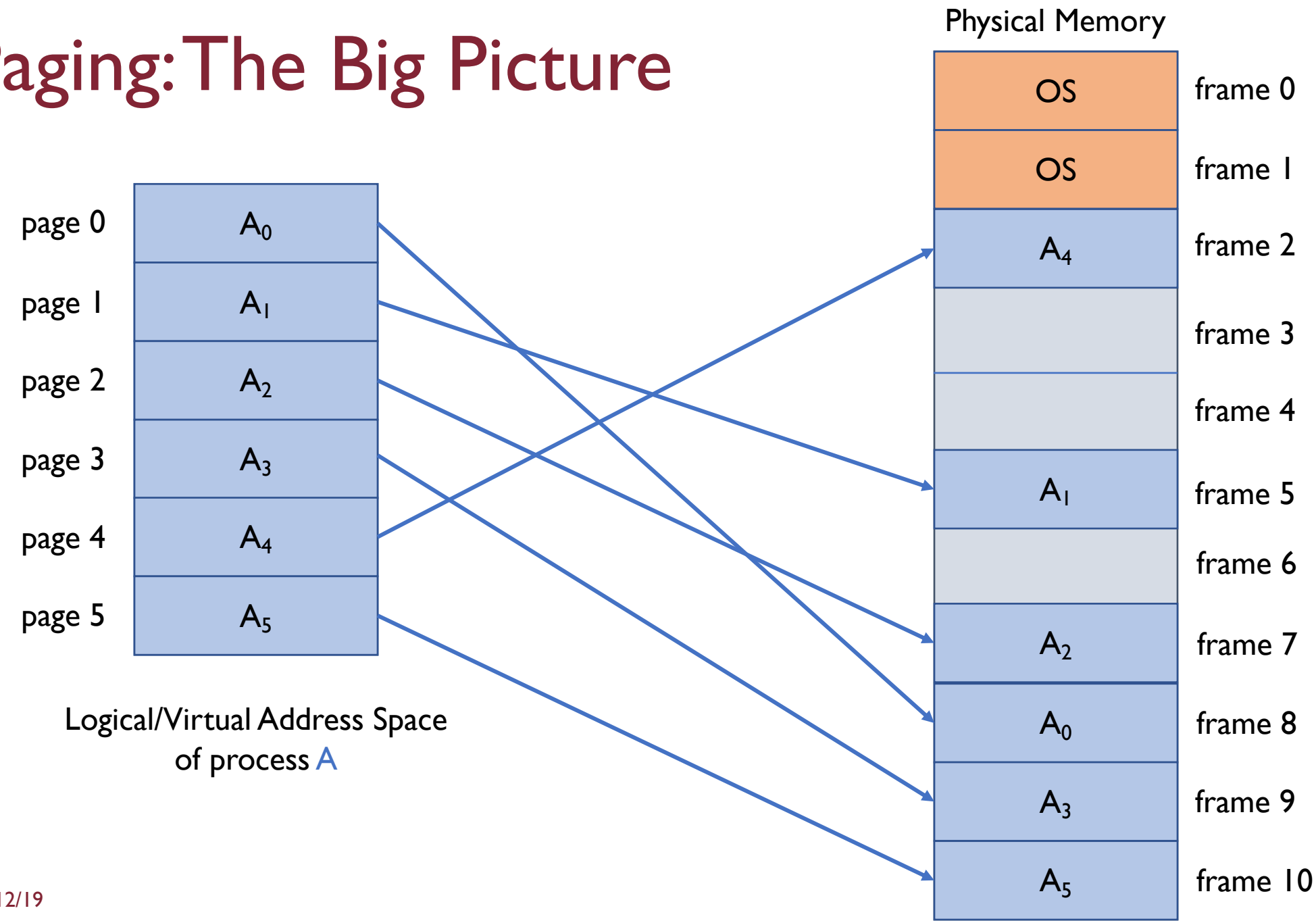
Processes spend 90% of their time accessing only 10% of their allocated memory space

Paging: The Big Picture



Logical/Virtual Address Space
of process A

Paging: The Big Picture



Basic OS Responsibilities for Paging

- The OS has 2 main responsibilities:
  - mapping between logical pages and physical frames
  - translating logical addresses to physical addresses

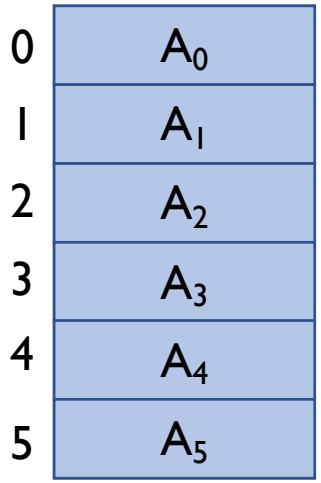
Basic OS Responsibilities for Paging

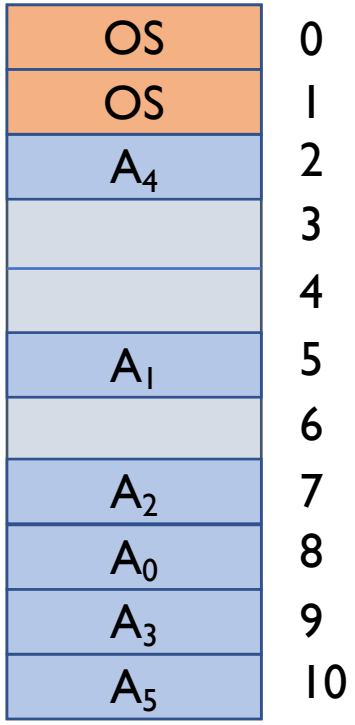
- The OS has 2 main responsibilities:
  - mapping between logical pages and physical frames
  - translating logical addresses to physical addresses
- All of this must be done efficiently!
  - Remember, memory addresses are referenced all the time

Basic OS Responsibilities for Paging

- The OS has 2 main responsibilities:
  - mapping between logical pages and physical frames
  - translating logical addresses to physical addresses
- All of this must be done efficiently!
  - Remember, memory addresses are referenced all the time
- OS needs dedicated support for doing it → **Page Table**

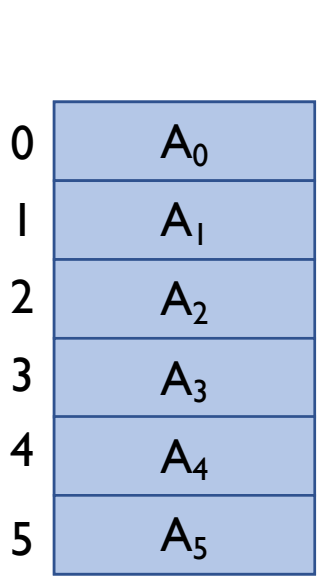
Page Table: Mapping Pages to Frames



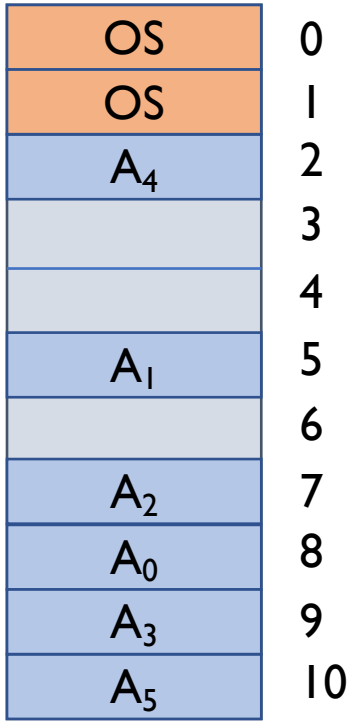


Page Table: Mapping Pages to Frames

Lookup table to efficiently retrieve what frame a page is stored in

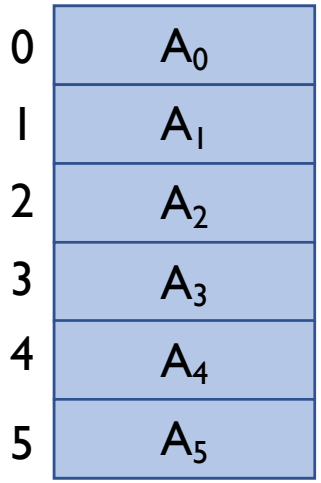


Page	Frame
0	8
1	5
2	7
3	9
4	2
5	10

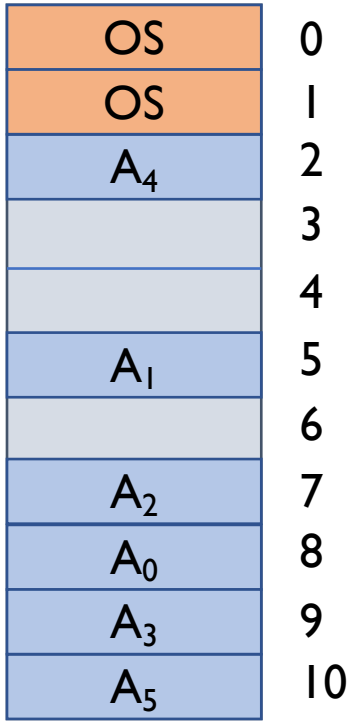


Page Table: Mapping Pages to Frames

Lookup table to efficiently retrieve what frame a page is stored in



Page	Frame
0	8
1	5
2	7
3	9
4	2
5	10



So far, we have simply assumed **all** pages of a process is mapped to physical frames, but we will see this is not always the case

Page Table: Virtual to Physical Address

- Processes use virtual (logical) addresses to refer to memory (not page number!)

Page Table: Virtual to Physical Address

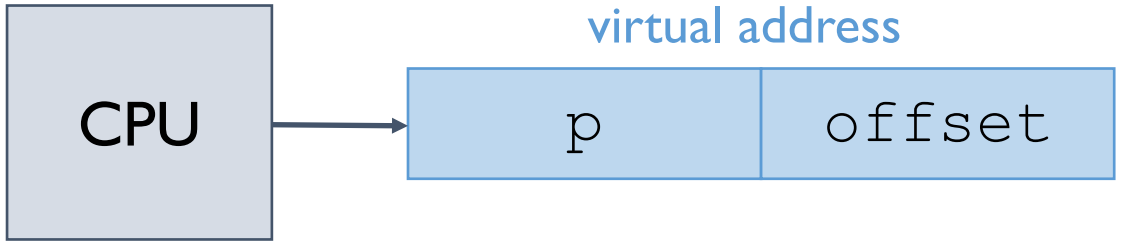
- Processes use virtual (logical) addresses to refer to memory (not page number!)
- Virtual (logical) address space is still contiguous starting from 0

Page Table: Virtual to Physical Address

- Processes use virtual (logical) addresses to refer to memory (not page number!)
- Virtual (logical) address space is still contiguous starting from 0
- Page table must ultimately translate virtual address to physical address

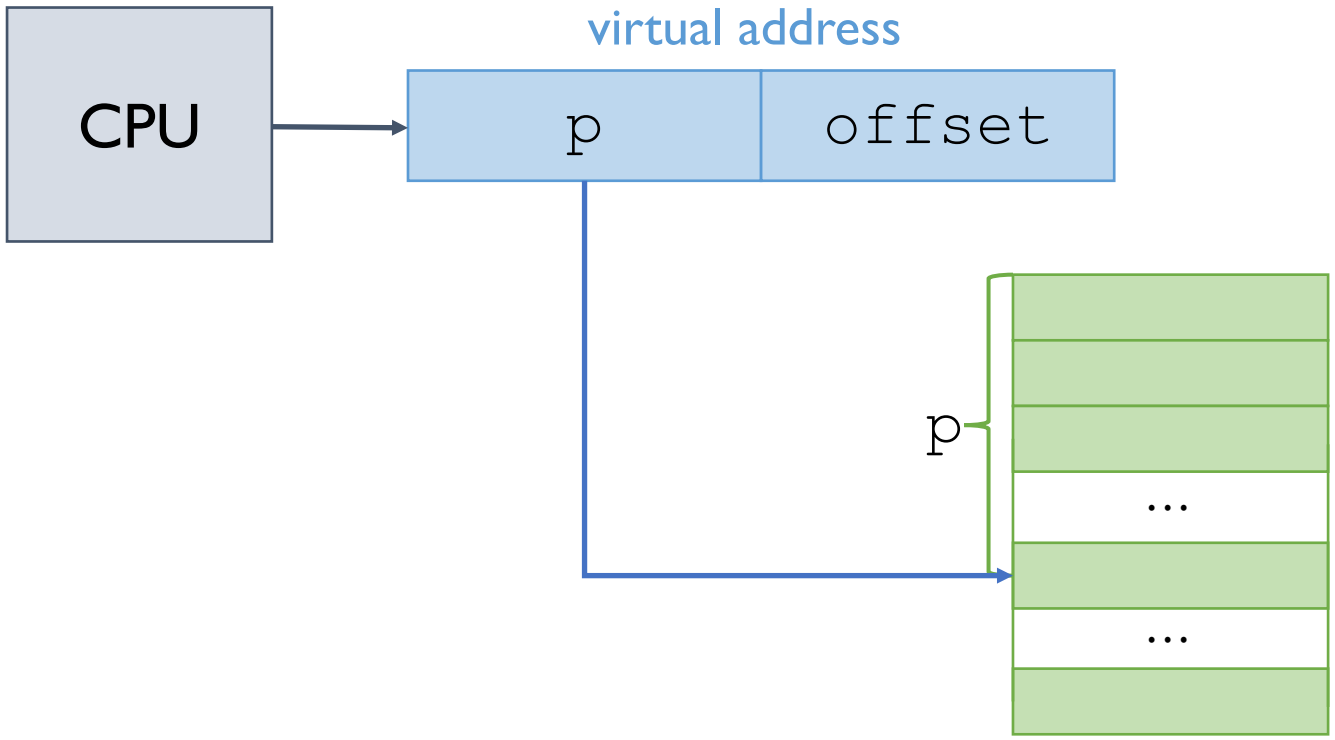
Page Table: Virtual to Physical Address

virtual address consists of 2 parts:
- `p`: page number where the address resides
- `offset`: relative from the beginning of the page



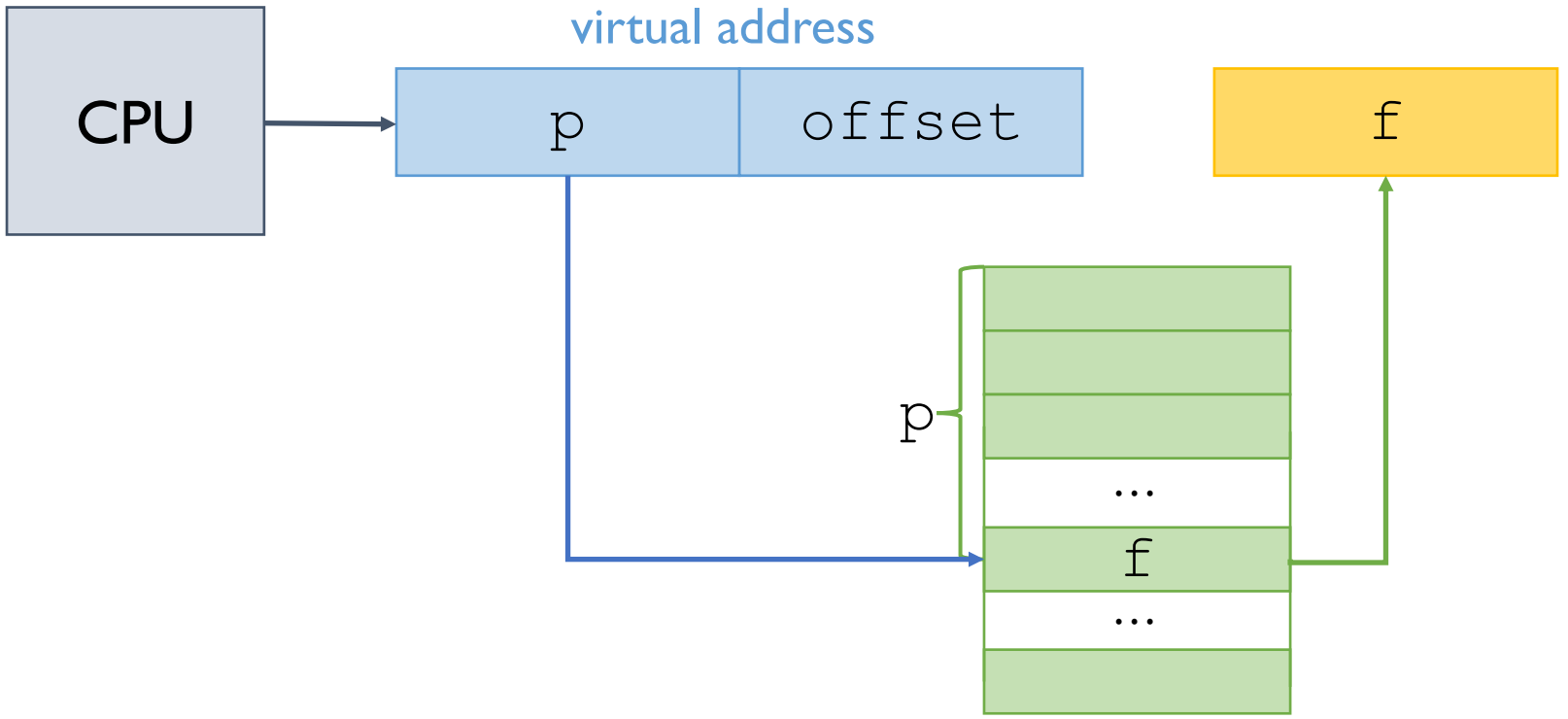
Page Table: Virtual to Physical Address

virtual address consists of 2 parts:
- `p`: page number where the address resides
- `offset`: relative from the beginning of the page



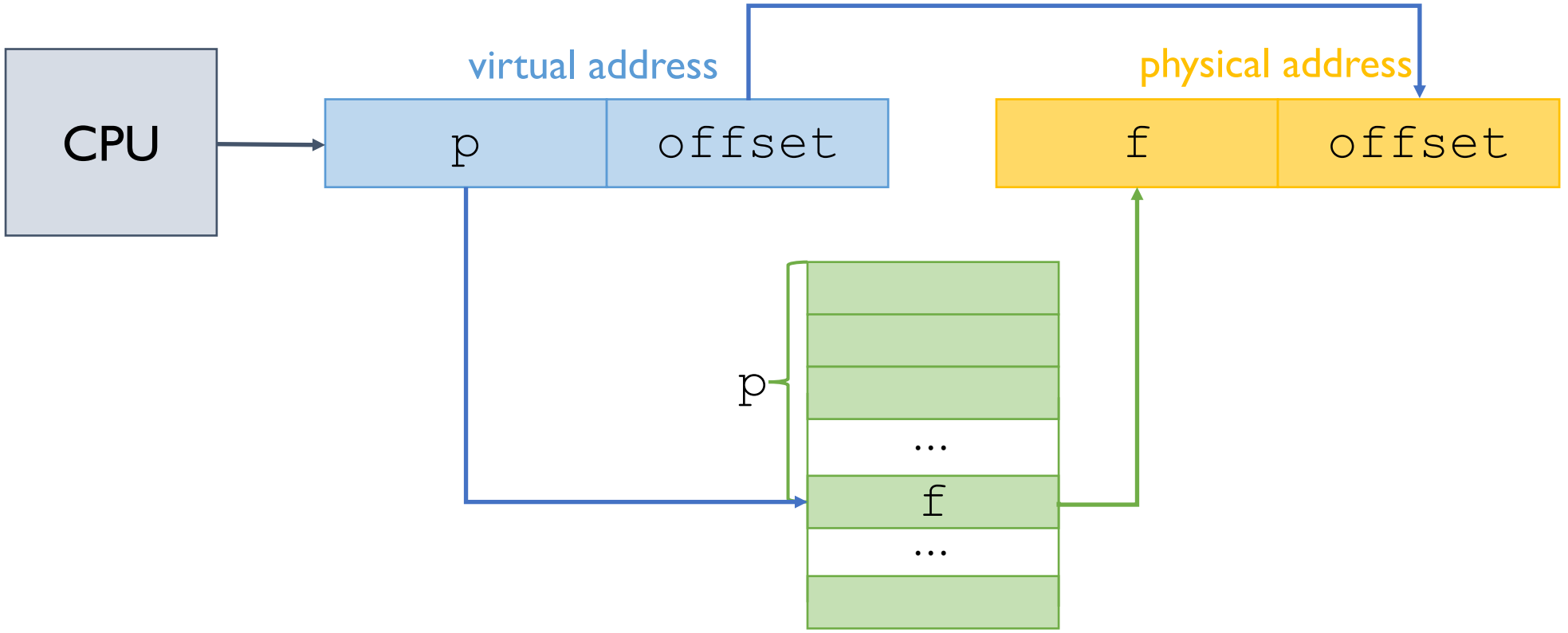
Page Table: Virtual to Physical Address

virtual address consists of 2 parts:
- `p`: page number where the address resides
- `offset`: relative from the beginning of the page



Page Table: Virtual to Physical Address

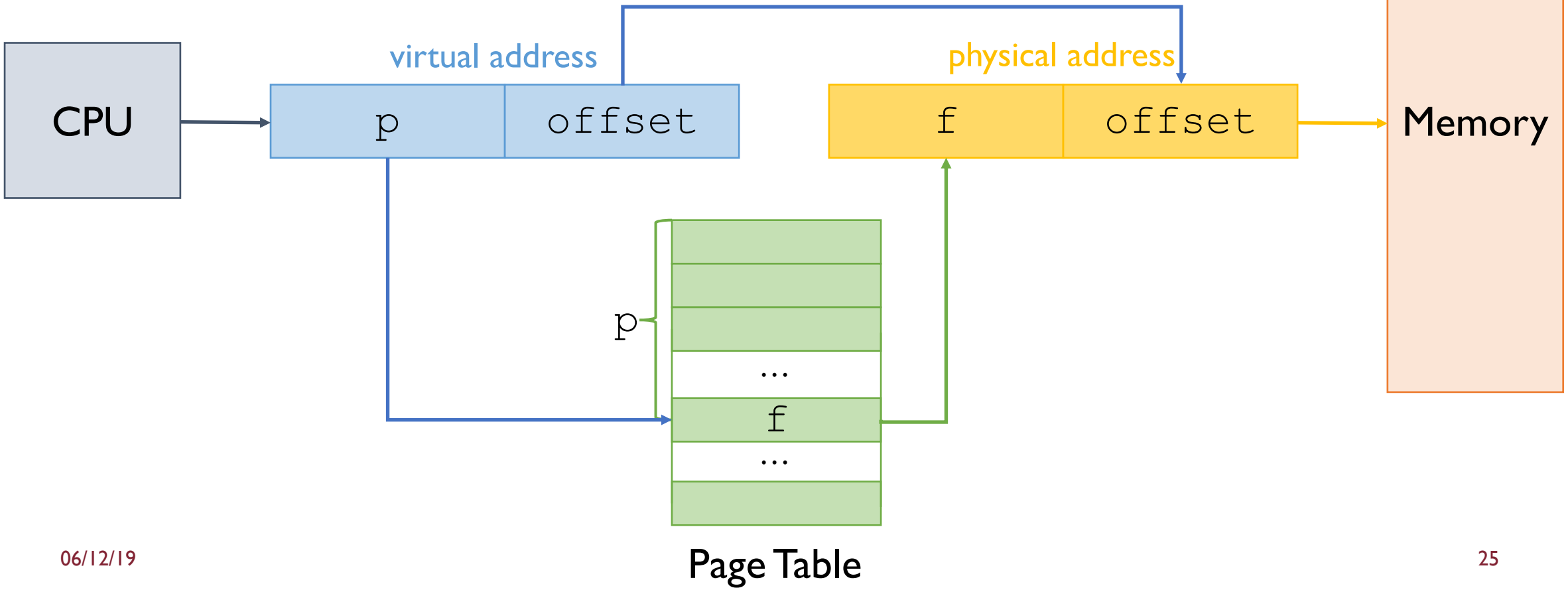
virtual address consists of 2 parts:
- `p`: page number where the address resides
- `offset`: relative from the beginning of the page



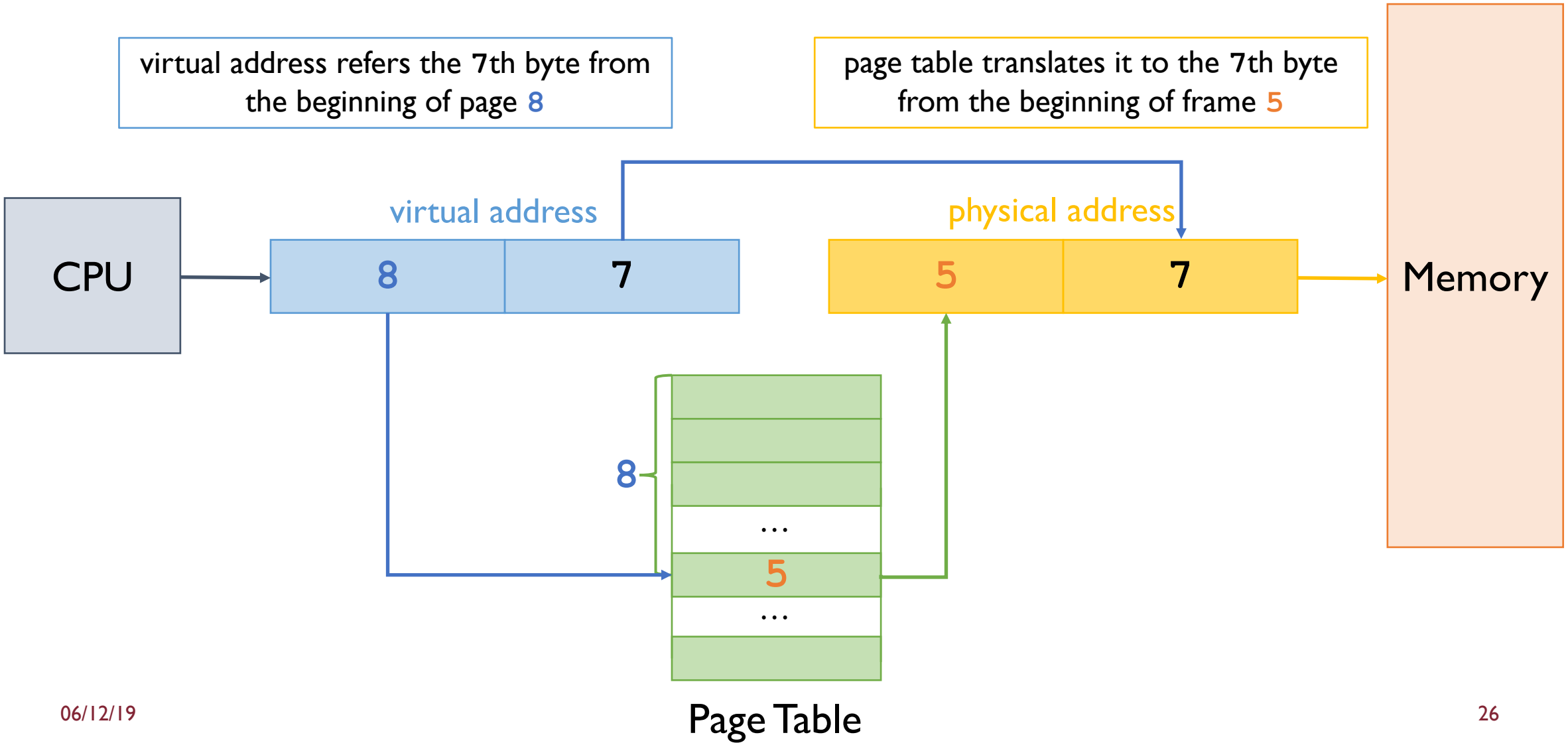
Page Table: Virtual to Physical Address

virtual address consists of 2 parts:
- `p`: page number where the address resides
- `offset`: relative from the beginning of the page

physical address also consists of 2 parts:
- `f`: physical frame number
- `offset`: as above



Page Table: Example of Address Translation



Paging Hardware

- Paging can is a form of dynamic relocation
- Each virtual address is bound by the paging hardware (i.e., page table) to a physical address
- Page table can be seen just as a set of base (relocation) registers, one for each frame
- Mapping is invisible to the user process: the OS maintains the page table and translation happens in hardware
- Protection is provided similarly to dynamic relocation (limit register)

Paging Hardware: Steps

How does page table translate a virtual address **x** into a physical address **y**?

Paging Hardware: Steps

How does page table translate a virtual address **`x`** into a physical address **`y`**?

1. Get the page number (**`p`**) and the **`offset`** where the virtual address **`x`** resides

Paging Hardware: Steps

How does page table translate a virtual address **`x`** into a physical address **`y`**?

1. Get the page number (**`p`**) and the **`offset`** where the virtual address **`x`** resides

2. Use **`p`** to index into the page table to retrieve the frame number **`f`**

Paging Hardware: Steps

How does page table translate a virtual address **`x`** into a physical address **`y`**?

1. Get the page number (**`p`**) and the **`offset`** where the virtual address **`x`** resides

2. Use **`p`** to index into the page table to retrieve the frame number **`f`**

3. Combine **`f`** with **`offset`** to obtain the physical address **`y`**

Paging Hardware: Get `p` and `offset` from `x`

Suppose we have **`50B`** of physical memory available for user processes

Paging Hardware: Get `p` and `offset` from `x`

Suppose we have **`50B`** of physical memory available for user processes

Assume we use paging with page (frame) size **`S = 10B`**

Paging Hardware: Get `p` and `offset` from `x`

Suppose we have **`50B`** of physical memory available for user processes

Assume we use paging with page (frame) size **`S = 10B`**

Each process can generate virtual addresses in the range **`[0, 49]`**

Paging Hardware: Get `p` and `offset` from `x`

Suppose we have **`50B`** of physical memory available for user processes

Assume we use paging with page (frame) size **`S = 10B`**

Each process can generate virtual addresses in the range **`[0, 49]`**

Suppose a process generates virtual address **`x = 27`**

Paging Hardware: Get `p` and `offset` from `x`

Suppose we have **50B** of physical memory available for user processes

Assume we use paging with page (frame) size **S = 10B**

Each process can generate virtual addresses in the range **[0, 49]**

Suppose a process generates virtual address **x = 27**

p = x div S

page number

Paging Hardware: Get `p` and `offset` from `x`

Suppose we have **50B** of physical memory available for user processes

Assume we use paging with page (frame) size **S = 10B**

Each process can generate virtual addresses in the range **[0, 49]**

Suppose a process generates virtual address **x = 27**

p = x div S

page number

p = 27 div 10 = 2

Paging Hardware: Get `p` and `offset` from `x`

Suppose we have **50B** of physical memory available for user processes

Assume we use paging with page (frame) size **S = 10B**

Each process can generate virtual addresses in the range **[0, 49]**

Suppose a process generates virtual address **x = 27**

p = x div S

page number

p = 27 div 10 = 2

offset = x mod S

offset

Paging Hardware: Get `p` and `offset` from `x`

Suppose we have **50B** of physical memory available for user processes

Assume we use paging with page (frame) size **S = 10B**

Each process can generate virtual addresses in the range **[0, 49]**

Suppose a process generates virtual address **x = 27**

p = x div S

page number

p = 27 div 10 = 2

offset = x mod S

offset

offset = 27 mod 10 = 7

Paging Hardware: Get `p` and `offset` from `x`

Suppose we have **50B** of physical memory available for user processes

Assume we use paging with page (frame) size **S = 10B**

Each process can generate virtual addresses in the range **[0, 49]**

Suppose a process generates virtual address **x = 27**

p = x div S

page number

offset = x mod S

offset

p = 27 div 10 = 2

offset = 27 mod 10 = 7

Address translation requires a **div** and a **mod** operation

Paging Hardware: Implementation Details

- Page/frame numbers and page/frame sizes are determined by the architecture

Paging Hardware: Implementation Details

- Page/frame numbers and page/frame sizes are determined by the architecture
- Page/frame sizes are typically a **power of 2,** ranging between 512B and 8192B (i.e., 8KiB)

Paging Hardware: Implementation Details

- Page/frame numbers and page/frame sizes are determined by the architecture
- Page/frame sizes are typically a **power of 2,** ranging between 512B and 8192B (i.e., 8KiB)
- Powers of 2 make the translation from virtual to physical address easy (i.e., no need for **`div`** and **`mod`**)

Paging Hardware: Implementation Details

- Page/frame numbers and page/frame sizes are determined by the architecture
- Page/frame sizes are typically a **power of 2,** ranging between 512B and 8192B (i.e., 8KiB)
- Powers of 2 make the translation from virtual to physical address easy (i.e., no need for **`div`** and **`mod`**)

Why?

Paging Hardware: Why Power of 2?

- Virtual address is made of m bits

Paging Hardware: Why Power of 2?

- Virtual address is made of m bits
  - Then, virtual address space (i.e., the set of bytes addressable by each user process) is 2^m long, and ranges between [0, 2^m-1]

Paging Hardware: Why Power of 2?

- Virtual address is made of m bits
  - Then, virtual address space (i.e., the set of bytes addressable by each user process) is 2^m long, and ranges between [0, 2^m-1]
- Assume page (frame) size is 2^n, n < m

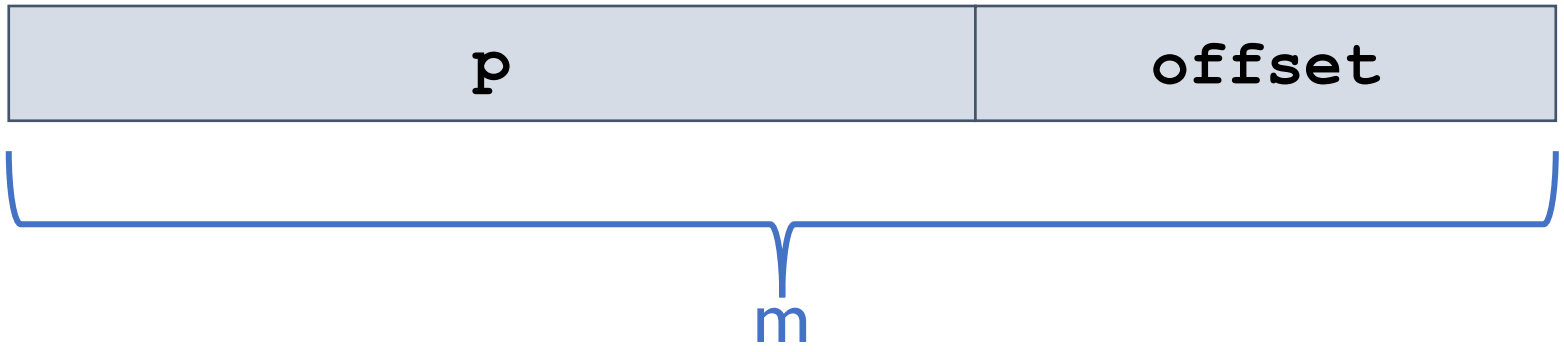
Paging Hardware: Why Power of 2?

- Virtual address is made of m bits
  - Then, virtual address space (i.e., the set of bytes addressable by each user process) is 2^m long, and ranges between [0, 2^m-1]
- Assume page (frame) size is 2^n, n < m
- The higher m-n bits of the virtual address indicates the page number

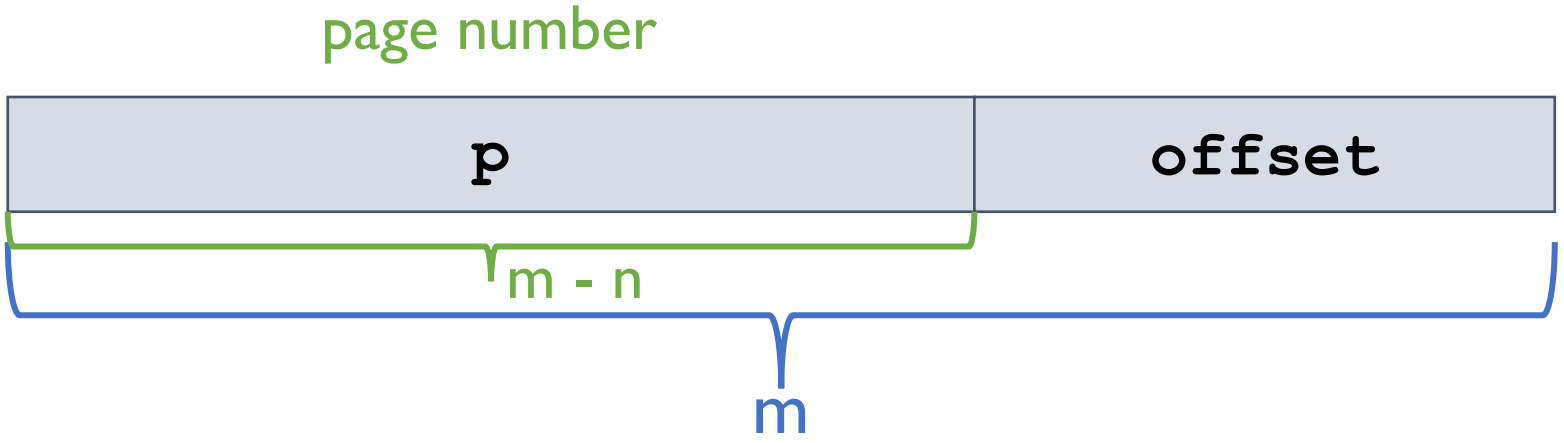
Paging Hardware: Why Power of 2?

- Virtual address is made of m bits
  - Then, virtual address space (i.e., the set of bytes addressable by each user process) is 2^m long, and ranges between [0, 2^m-1]
- Assume page (frame) size is 2^n, n < m
- The higher m-n bits of the virtual address indicates the page number
- The low order n bits represent the offset

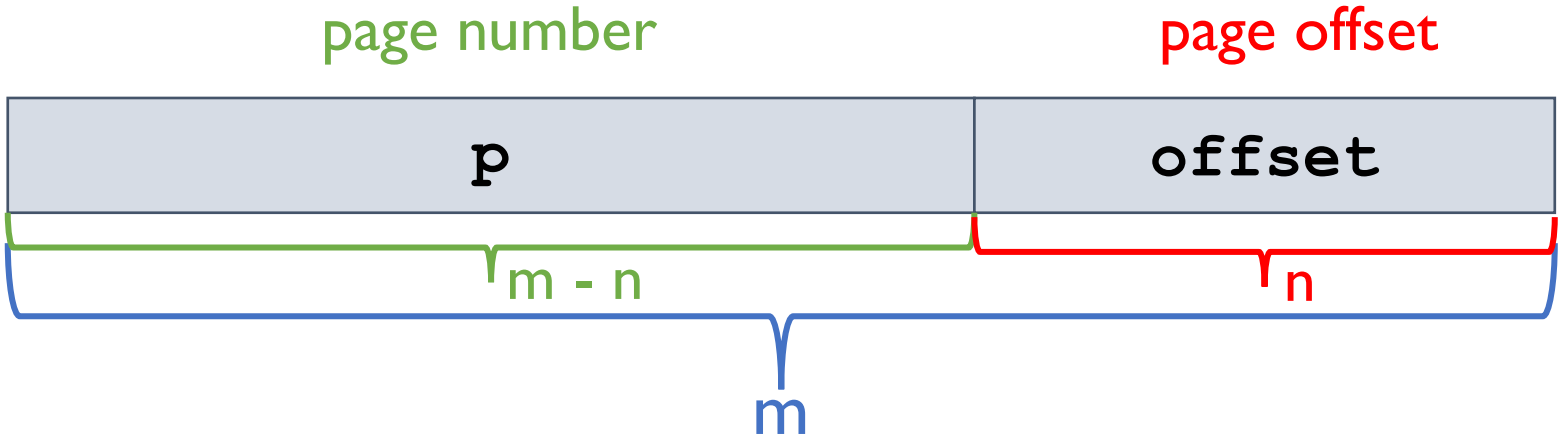
Paging Hardware: Why Power of 2?



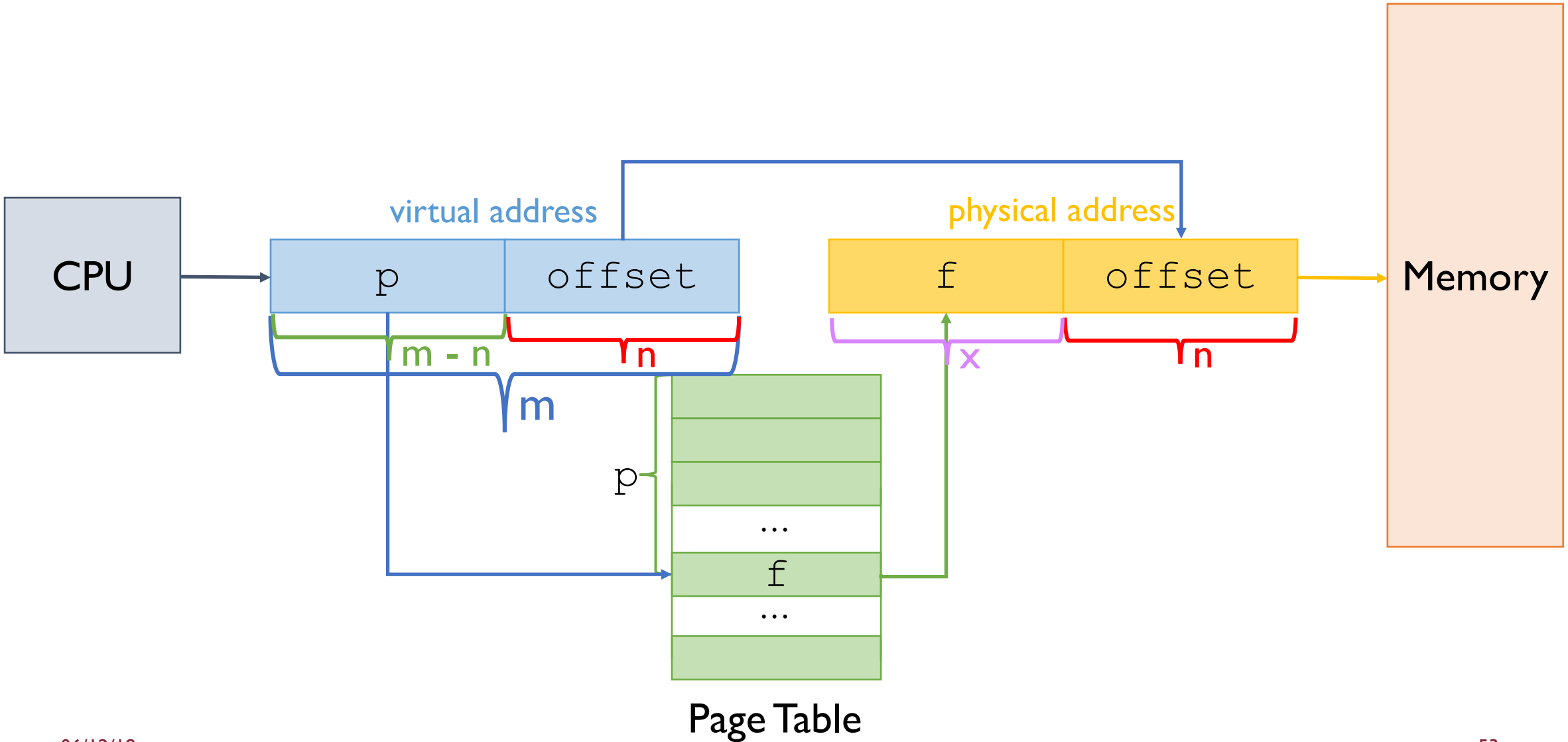
Paging Hardware: Why Power of 2?



Paging Hardware: Why Power of 2?



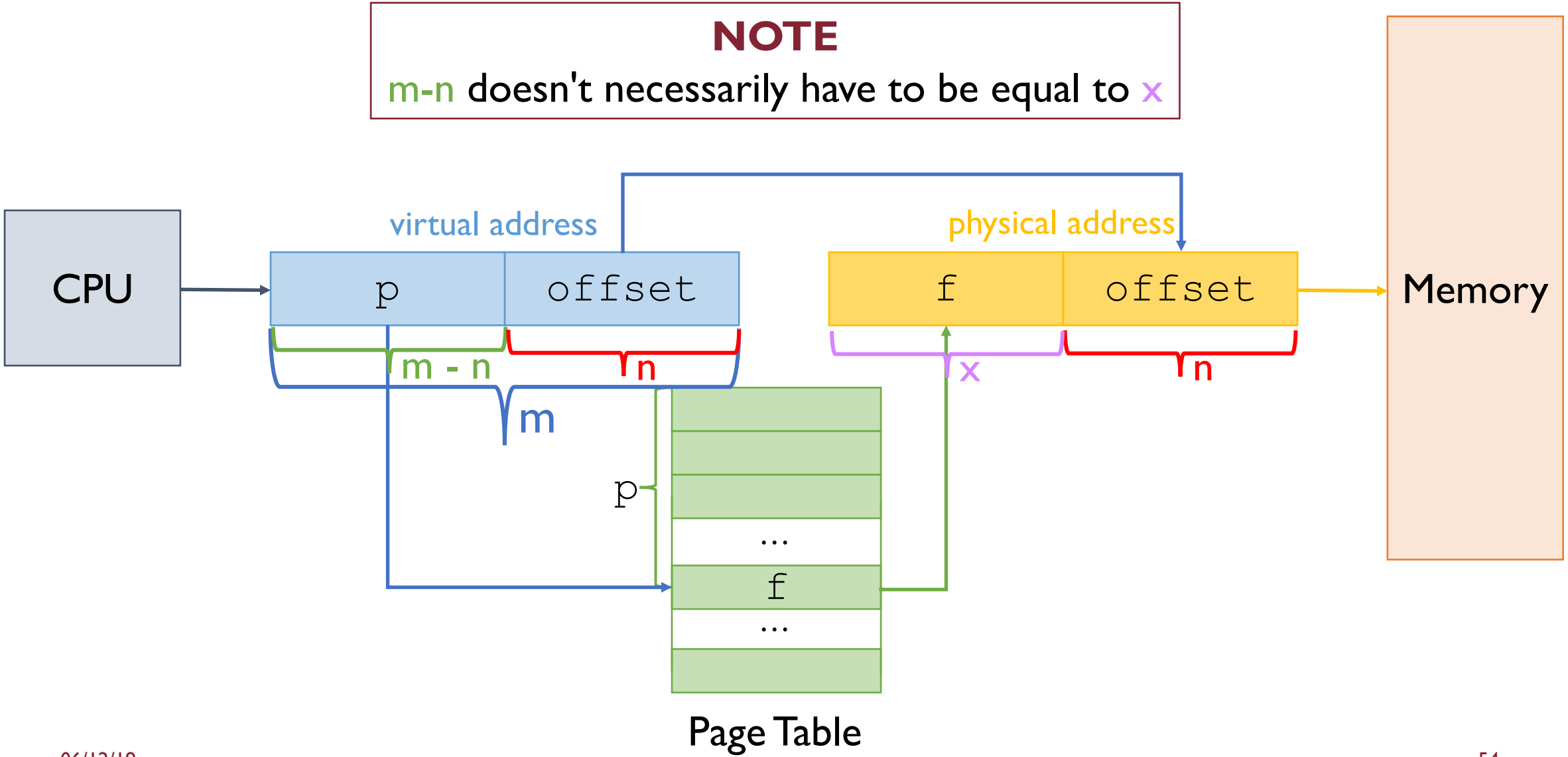
Paging Hardware: Practical Details



Paging Hardware: Practical Details

NOTE

m-n doesn't necessarily have to be equal to x



Paging Hardware: Practical Details

- Typical values of virtual address size is m = 32 or 64 bits
  - That means the virtual address space is 2^{32} = 4GiB or 2^{64} = 16EiB

Paging Hardware: Practical Details

- Typical values of virtual address size is m = 32 or 64 bits
  - That means the virtual address space is 2^{32} = 4GiB or 2^{64} = 16EiB
- Typical values of page/frame sizes is n = 12 bits
  - That means each page/frame is 2^{12} = 4KiB

Paging Hardware: Practical Details

- Typical values of virtual address size is m = 32 or 64 bits
  - That means the virtual address space is 2^{32} = 4GiB or 2^{64} = 16EiB
- Typical values of page/frame sizes is n = 12 bits
  - That means each page/frame is 2^{12} = 4KiB
- Assuming m = 32 bits, there are 2^{m-n} = 2^{20} = 1MiB pages/frames
  - That means page table has 2^{20} entries (i.e., one for each page/frame)

Paging Hardware: Practical Example

Suppose we have a virtual memory and a physical memory, both of size M = 1024B (1KiB)

Q1

How many bits are needed for a virtual/physical address (assuming single-byte addressing)

Paging Hardware: Practical Example

Suppose we have a virtual memory and a physical memory, both of size M = 1024B (1KiB)

Q1

How many bits are needed for a virtual/physical address (assuming single-byte addressing)

R1

10 bits to address M = 1024 bytes (both for virtual and physical address)

Paging Hardware: Practical Example

Now, assume we use paging with page/frame size S =16B

Q2

How big is the page table? (i.e., how many pages/entries does it have to index?)

Paging Hardware: Practical Example

Now, assume we use paging with page/frame size S =16B

Q2

How big is the page table? (i.e., how many pages/entries does it have to index?)

R2

T = M / S = 1024 memory bytes / 16 bytes per page = 64 pages

Paging Hardware: Practical Example

Q3

What is `p` and `offset` (i.e., how many bits for `p` and `offset`?)

Paging Hardware: Practical Example

Q3

What is `p` and `offset` (i.e., how many bits for `p` and `offset`?)

R3

Our logical address is made of m = 10 bits

n = 4 bits are used to represent the `offset`, as each page/frame is S = 16 bytes

m-n = 6 bits are used to represent page number `p`, as there are T = 64 pages

Paging Hardware: Practical Example

Q4

Translate the virtual address `x = 42`, assuming the following page table

page	frame
0	12
1	5
2	37
3	0
…	..
63	29

Paging Hardware: Practical Example

Q4

Translate the virtual address **`x = 42`**, assuming the following page table

page	frame
0	12
1	5
2	37
3	0
…	..
63	29

R4

`p = x div S = 42 div 16 = 2`

Paging Hardware: Practical Example

Q4

Translate the virtual address **x = 42**, assuming the following page table

page	frame
0	12
1	5
2	37
3	0
…	..
63	29

R4

p = x **div** S = 42 **div** 16 = **2**

Paging Hardware: Practical Example

Q4

Translate the virtual address **x = 42**, assuming the following page table

page	frame
0	12
1	5
2	37
3	0
…	..
63	29

R4

p = x **div** S = 42 **div** 16 = **2**

offset = x **mod** S = 42 **mod** 16 = **10**

10th byte from the beginning of frame 37

Paging Hardware: Practical Example 2

Suppose we still have a virtual memory and a physical memory, both of size M = 1024B

Q1

So far, we have assumed that computers work on single-byte (i.e, 8-bit architecture)
Modern computers however operate natively on multiple of bytes (i.e., **words**) rather than single-byte. Typical values of word length is: 16, 32 or 64 bits.
If we assume 32-bit architecture (i.e., word = 32 bits = 4 bytes), virtual addresses refer to words instead of bytes
How many bits are therefore needed to address the number of words available on M?

Paging Hardware: Practical Example 2

Suppose we still have a virtual memory and a physical memory, both of size M = 1024B

Q1

So far, we have assumed that computers work on single-byte (i.e, 8-bit architecture) Modern computers however operate natively on multiple of bytes (i.e., **words**) rather than single-byte. Typical values of word length is: 16, 32 or 64 bits.
If we assume 32-bit architecture (i.e., word = 32 bits = 4 bytes), virtual addresses refer to words instead of bytes
How many bits are therefore needed to address the number of words available on M?

R1

8 bits to address M = 1024/4 = 256 4-byte **words** (both for virtual and physical address)

Paging Hardware: Practical Example 2

Now, assume we still use paging with page/frame size S =16B

Q2

How big is the page table? (i.e., how many pages/entries does it have to index?)

Paging Hardware: Practical Example 2

Now, assume we still use paging with page/frame size S =16B

Q2

How big is the page table? (i.e., how many pages/entries does it have to index?)

R2

T = M / S = 1024 memory bytes / 16 bytes per page = 64 pages

Paging Hardware: Practical Example 2

Q3

What is `p` and `offset` (i.e., how many bits for `p` and `offset`?)

Paging Hardware: Practical Example 2

Q3

What is `p` and `offset` (i.e., how many bits for `p` and `offset`?)

R3

Our logical address is now made of m = 8 bits

n = 2 bits are used to represent the `offset`, as each page/frame is:

S = 16 bytes = 4 * 4-byte **words**

m-n = 6 bits are used to represent page number `p`, as there are still T = 64 pages

Paging Hardware: Practical Example 2

Q4

Translate the virtual address **x = 7**, assuming the following page table

page	frame
0	12
1	5
2	37
3	0
…	..
63	29

Paging Hardware: Practical Example

Q4

Translate the virtual address **x = 7**, assuming the following page table

page	frame
0	12
1	5
2	37
3	0
…	..
63	29

Remember: now virtual address refers to a 4-byte word!

Paging Hardware: Practical Example 2

Q4

Translate the virtual address **x = 7**, assuming the following page table

page	frame
0	12
1	5
2	37
3	0
…	..
63	29

`S` = `16` bytes = `4 * 4-byte` **words**

Must be expressed in terms of number of words

R4

`p = x` **`div`** `S = 7` **`div`** `4 = 1`

Paging Hardware: Practical Example 2

Q4

Translate the virtual address **x = 7**, assuming the following page table

page	frame
0	12
1	5
2	37
3	0
…	..
63	29

R4

p = x **div** S = 7 **div** 4 = **1**

offset = x **mod** S = 7 **mod** 4 = **3**

3rd **word** from the beginning of frame 5

How To Make Paging Efficient?

- Every single time a user process references a (virtual) memory address through the CPU this has to be translated to a physical one

How To Make Paging Efficient?

- Every single time a user process references a (virtual) memory address through the CPU this has to be translated to a physical one
- Where should the page table be stored?

How To Make Paging Efficient?

- Every single time a user process references a (virtual) memory address through the CPU this has to be translated to a physical one
- Where should the page table be stored?
  - **Registers** → PRO: very fast CON: very expensive and limited

How To Make Paging Efficient?

- Every single time a user process references a (virtual) memory address through the CPU this has to be translated to a physical one
- Where should the page table be stored?
  - **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 Efficient?

- Every single time a user process references a (virtual) memory address through the CPU this has to be translated to a physical one
- Where should the page table be stored?
  - **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 solution: **Translation Look-aside Buffer** (**TLB**)

Appendix: 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

Appendix: 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)

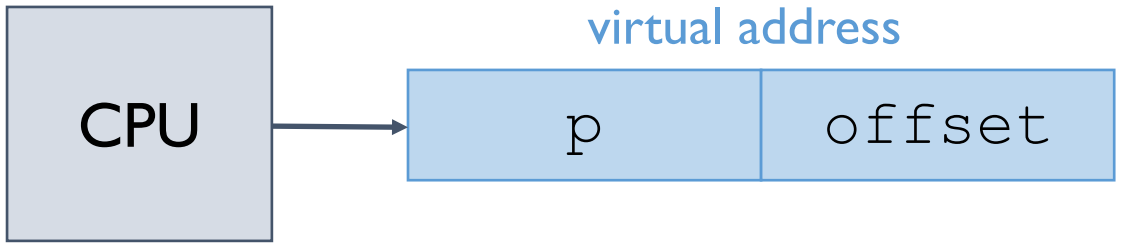
Translation Look-aside Buffer (TLB)

- 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

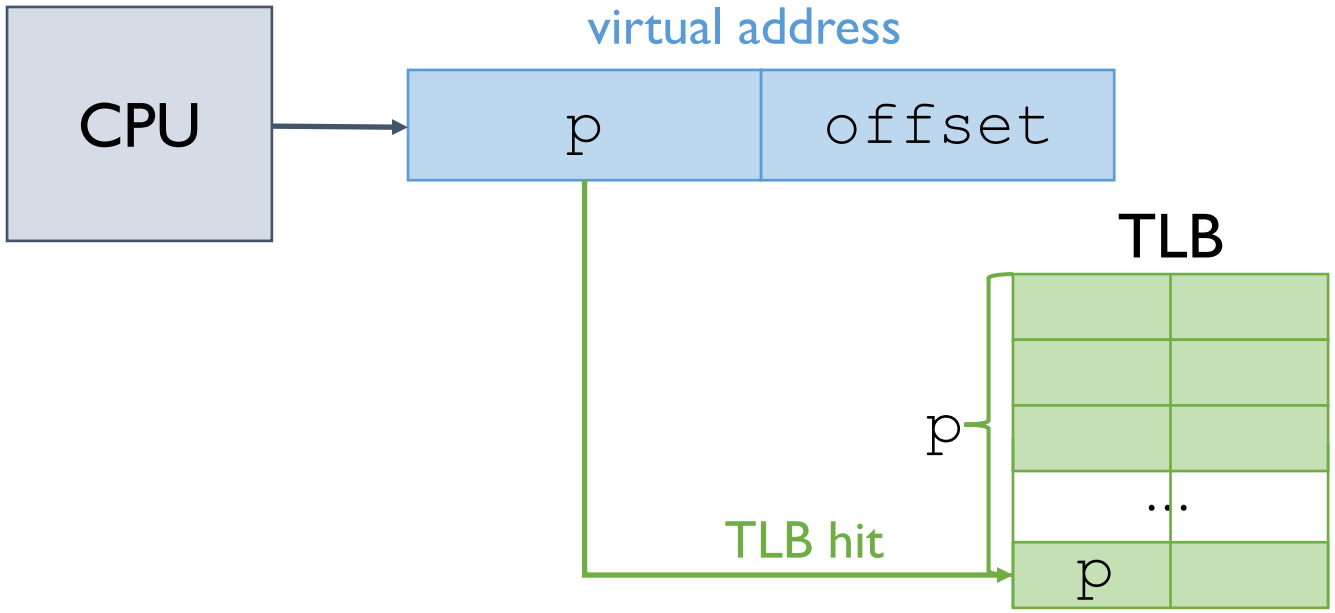
Translation Look-aside Buffer (TLB)

- 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

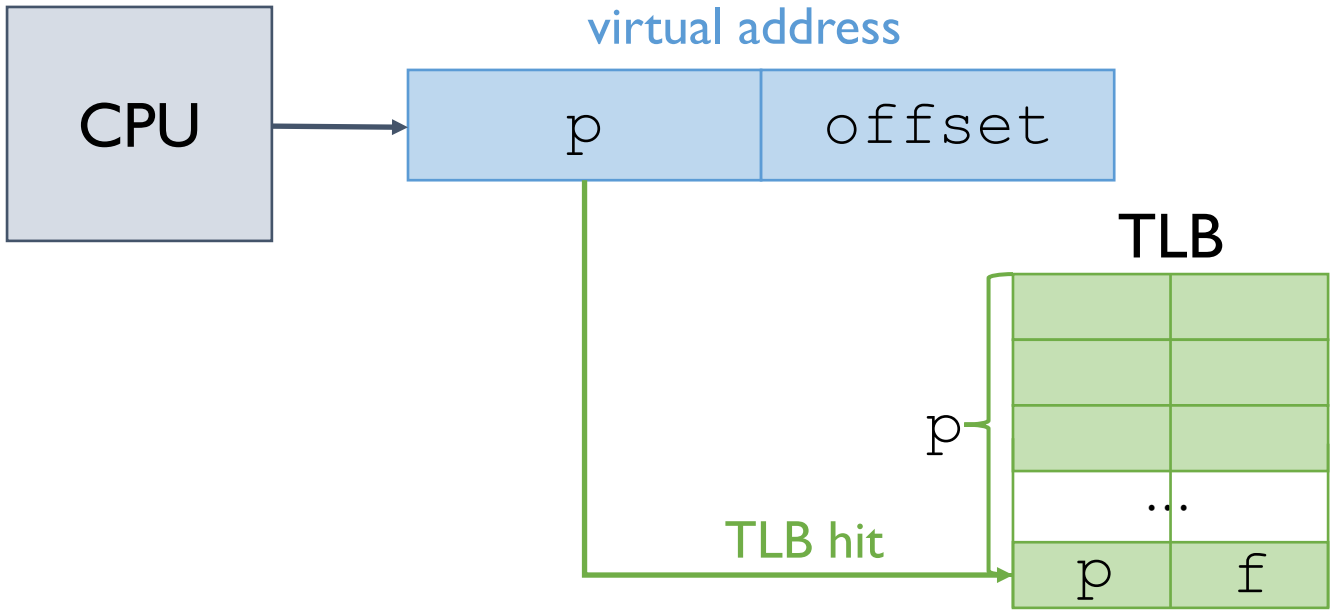
Translation Look-aside Buffer (TLB)



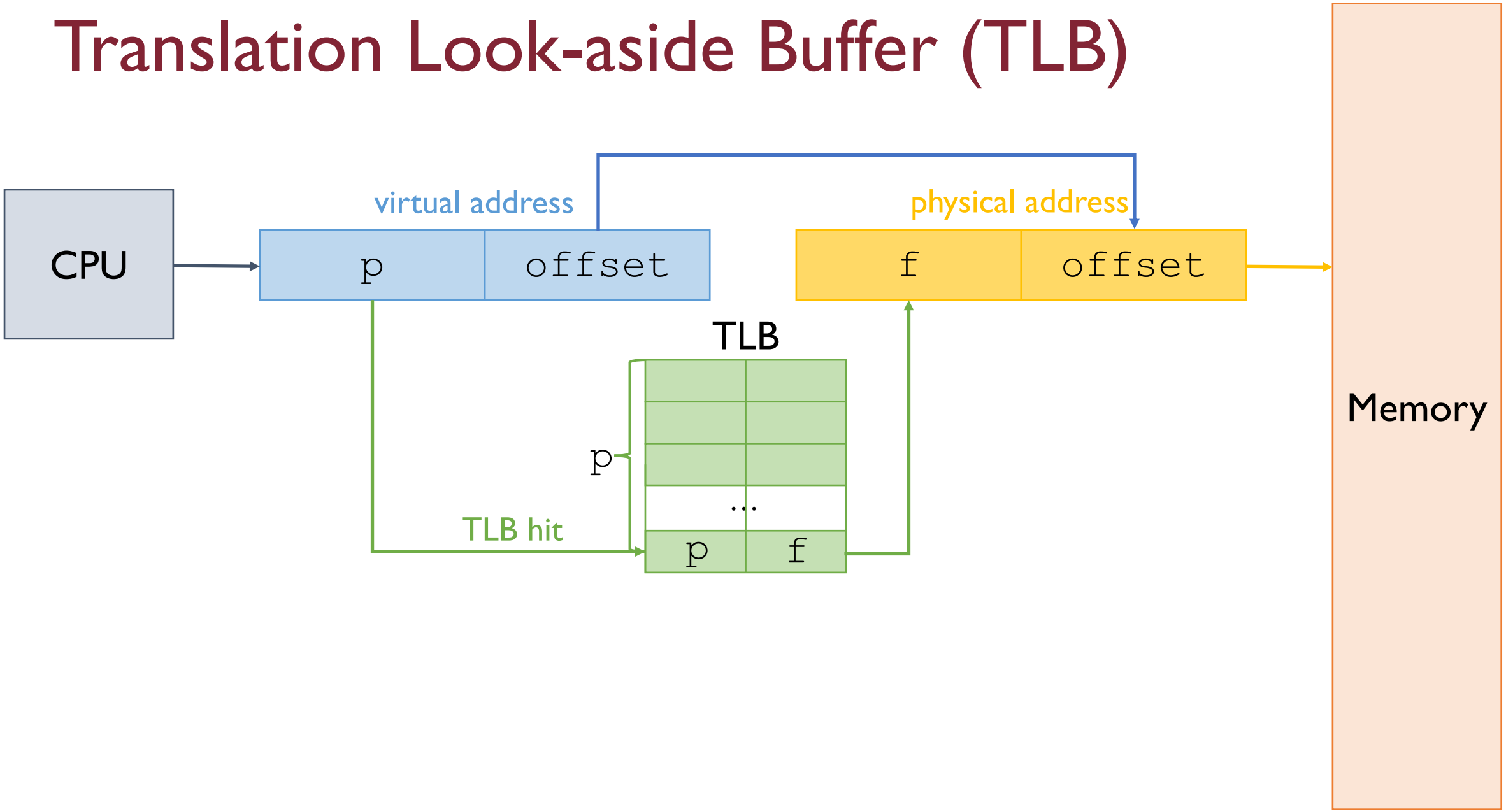
Translation Look-aside Buffer (TLB)



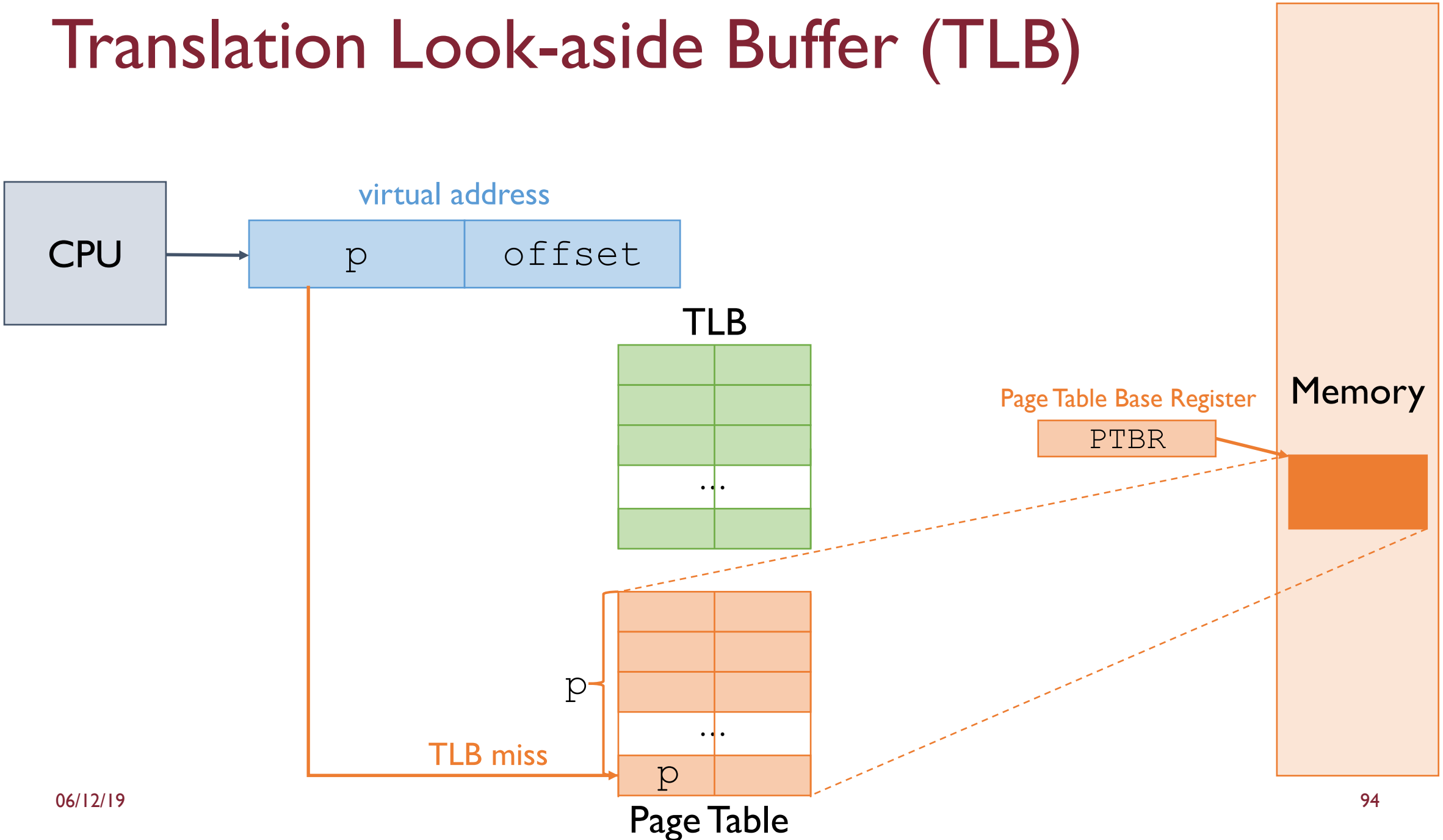
Translation Look-aside Buffer (TLB)



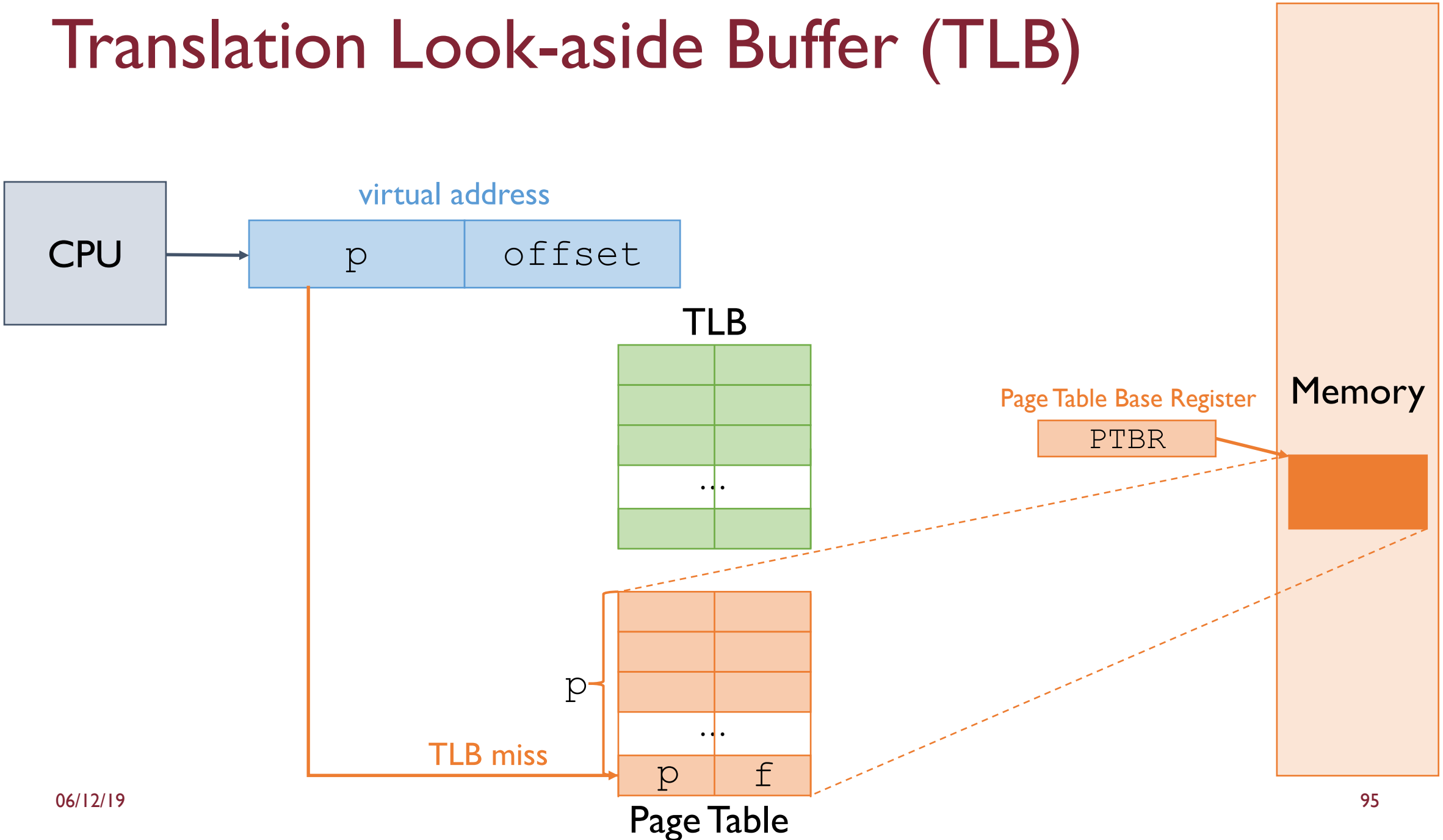
Translation Look-aside Buffer (TLB)



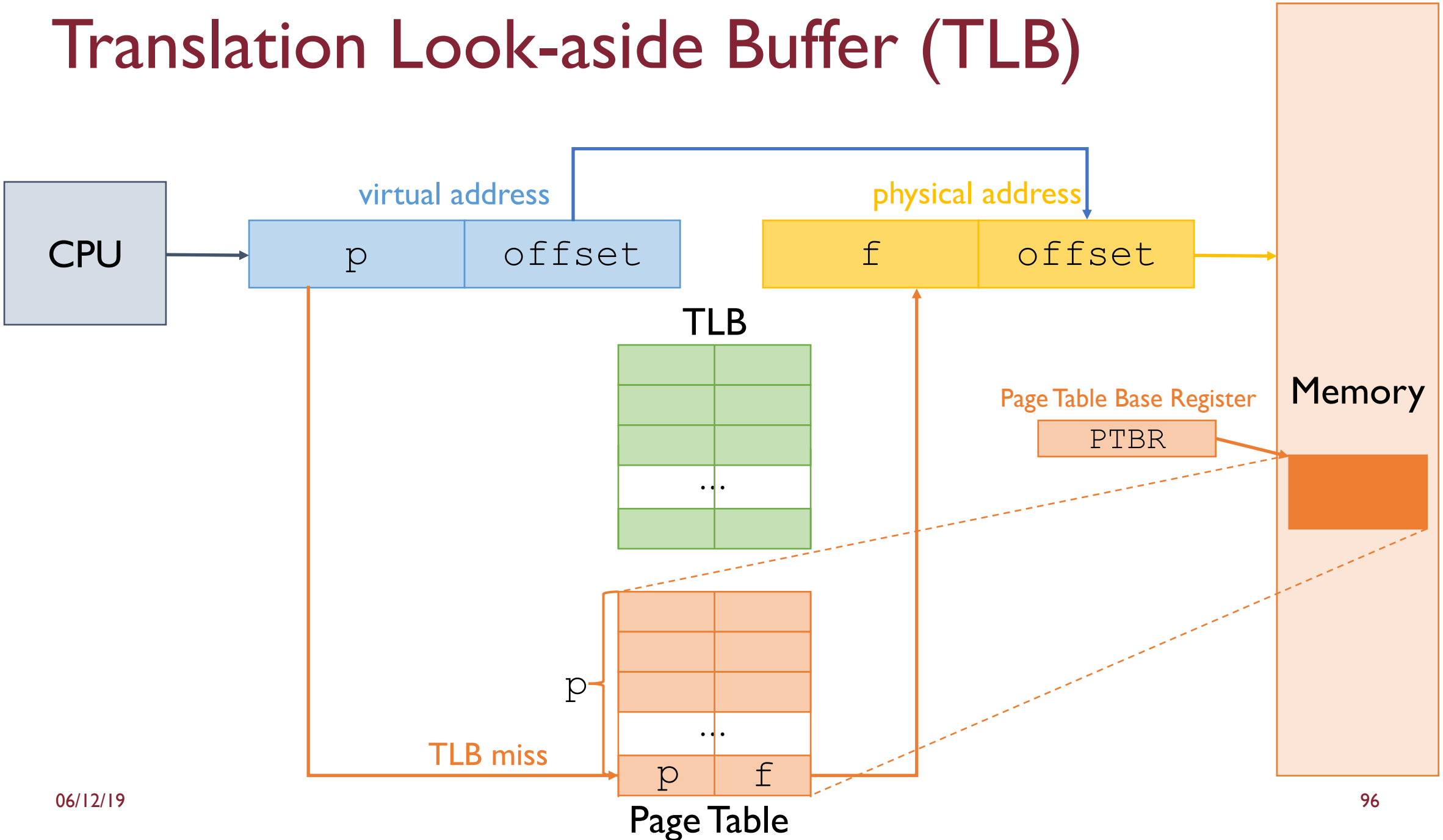
Translation Look-aside Buffer (TLB)



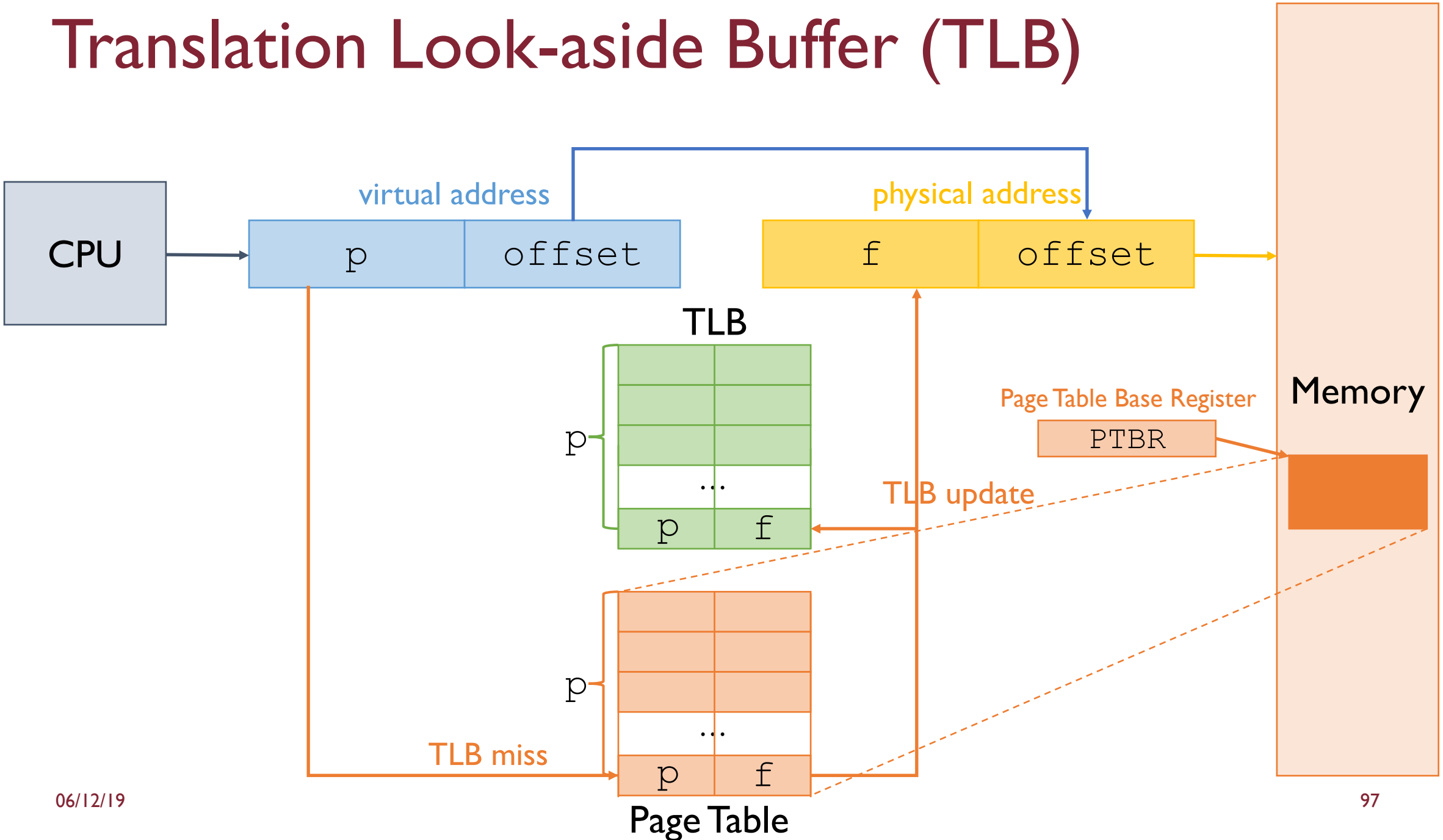
Translation Look-aside Buffer (TLB)



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 which is requesting the translation

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 which is requesting the translation
- How to deal with multiple process and a single TLB? **2 setups**:

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 which is requesting the translation
- 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)

- 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 which is requesting the translation
- 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

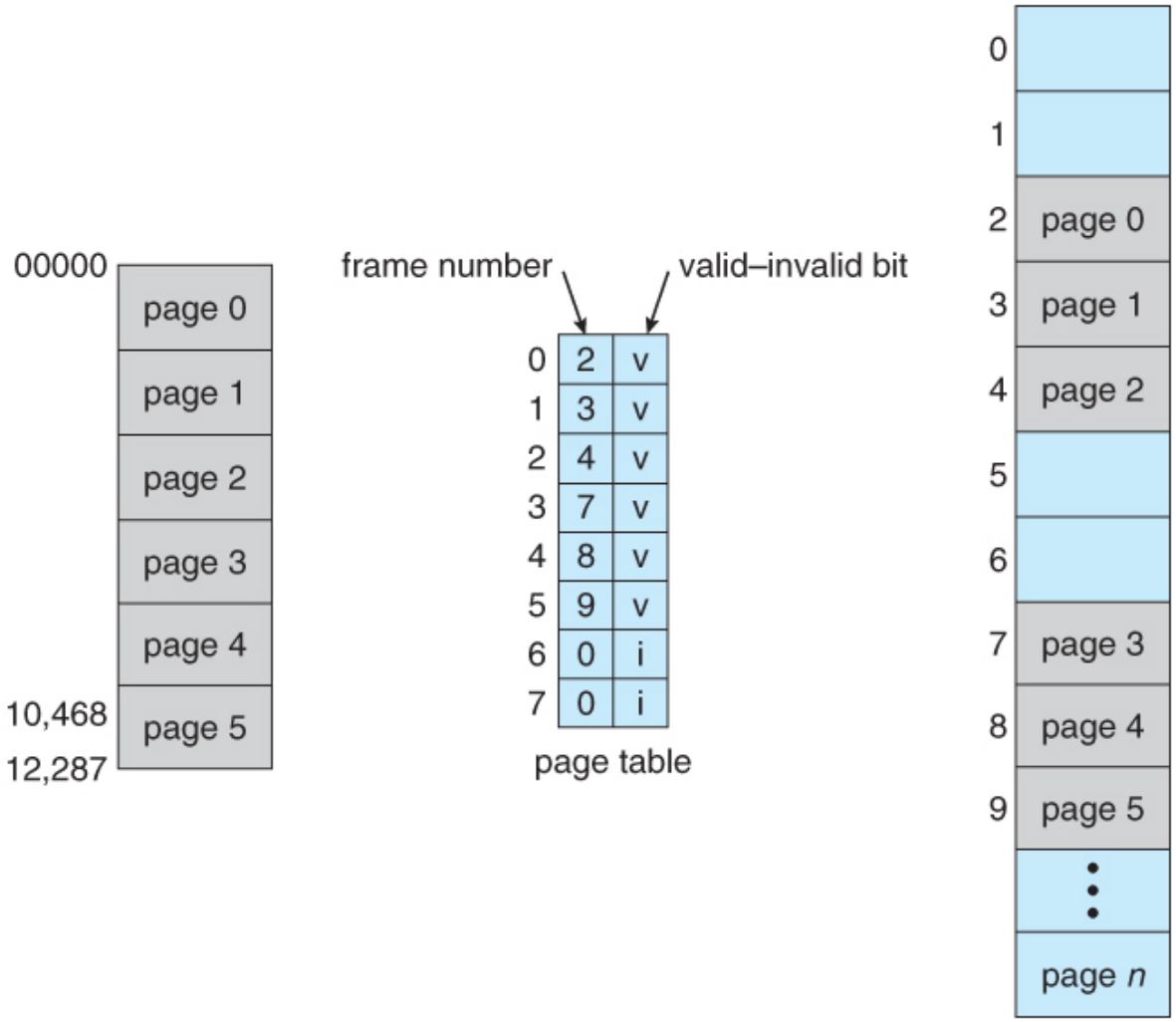
Additional Protection

- The page table can also help to protect processes from accessing memory they shouldn't, or their own memory in correct ways
- 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

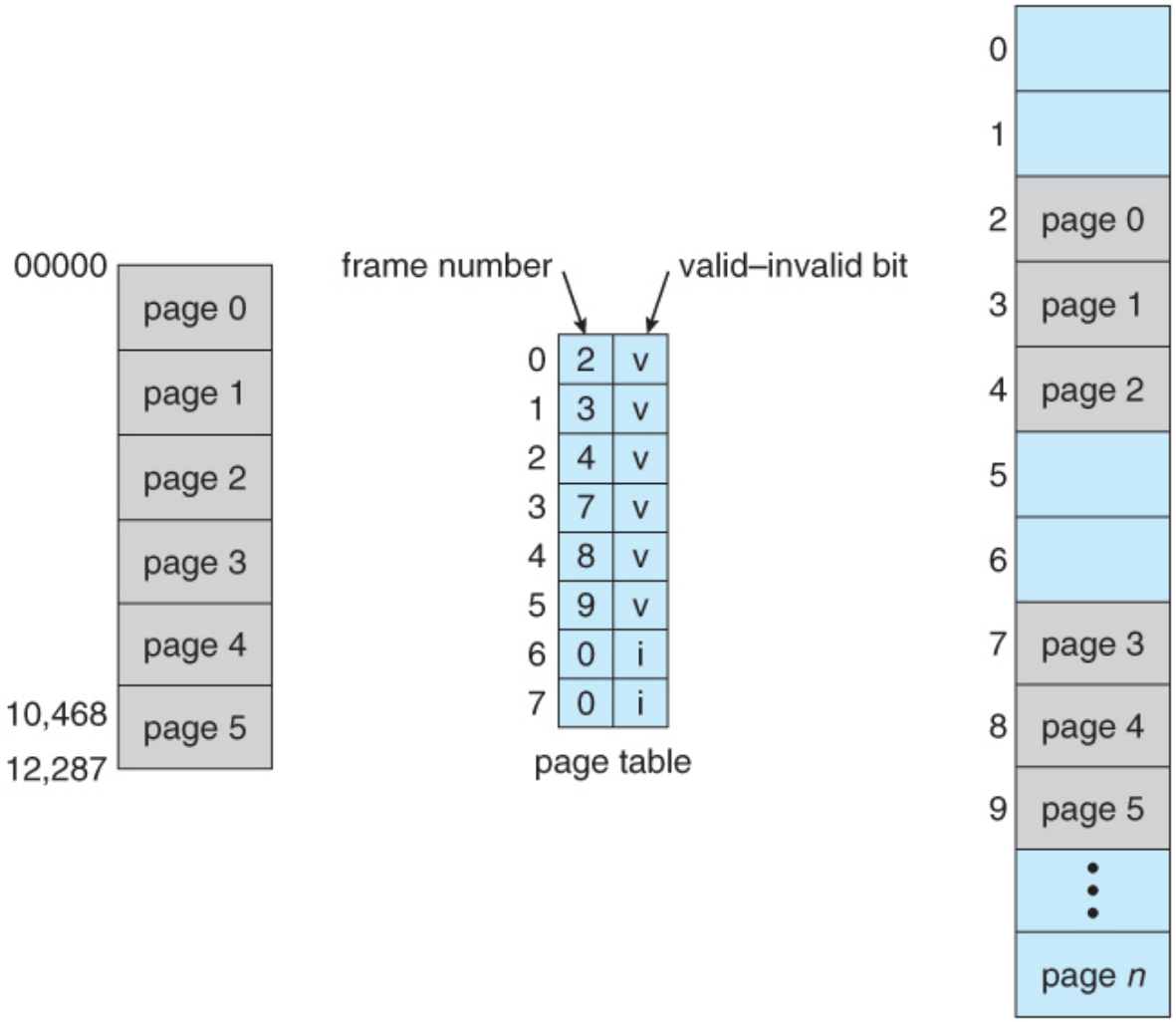
- valid/invalid bits cannot block all illegal memory accesses, due to the internal fragmentation
- Many processes do not use all of the page table entries available, particularly in modern systems with very large potential page tables
- Some systems use a page-table length register (`PTLR`) to specify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

Initializing Memory when Starting a Process

1. Process requests for *k* pages

Initializing Memory when Starting a Process

1. Process requests for *k* pages
2. If *k* frames are free then allocate those to the process, otherwise free frames no longer needed (swapping-out)

Initializing Memory when Starting a Process

1. Process requests for *k* pages
2. If *k* frames are free then allocate those to the process, otherwise free frames no longer needed (swapping-out)
3. OS puts each page into a frame and sets the corresponding mapping into the page table (in main memory)

Initializing Memory when Starting a Process

1. Process requests for *k* pages
2. If *k* frames are free then allocate those to the process, otherwise free frames no longer needed (swapping-out)
3. OS puts each page into a frame and sets the corresponding mapping into the page table (in main memory)
4. OS marks all previous TLB entries as invalid (i.e., flushes the cache) or restores TLB entries from saved PCB

Initializing Memory when Starting a Process

1. Process requests for *k* pages
2. If *k* frames are free then allocate those to the process, otherwise free frames no longer needed (swapping-out)
3. OS puts each page into a frame and sets the corresponding mapping into the page table (in main memory)
4. OS marks all previous TLB entries as invalid (i.e., flushes the cache) or restores TLB entries from saved PCB
5. As process runs, OS loads TLB missed entries possibly replacing existing entries if TLB is full

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, since memory doesn't have to be contiguous anymore

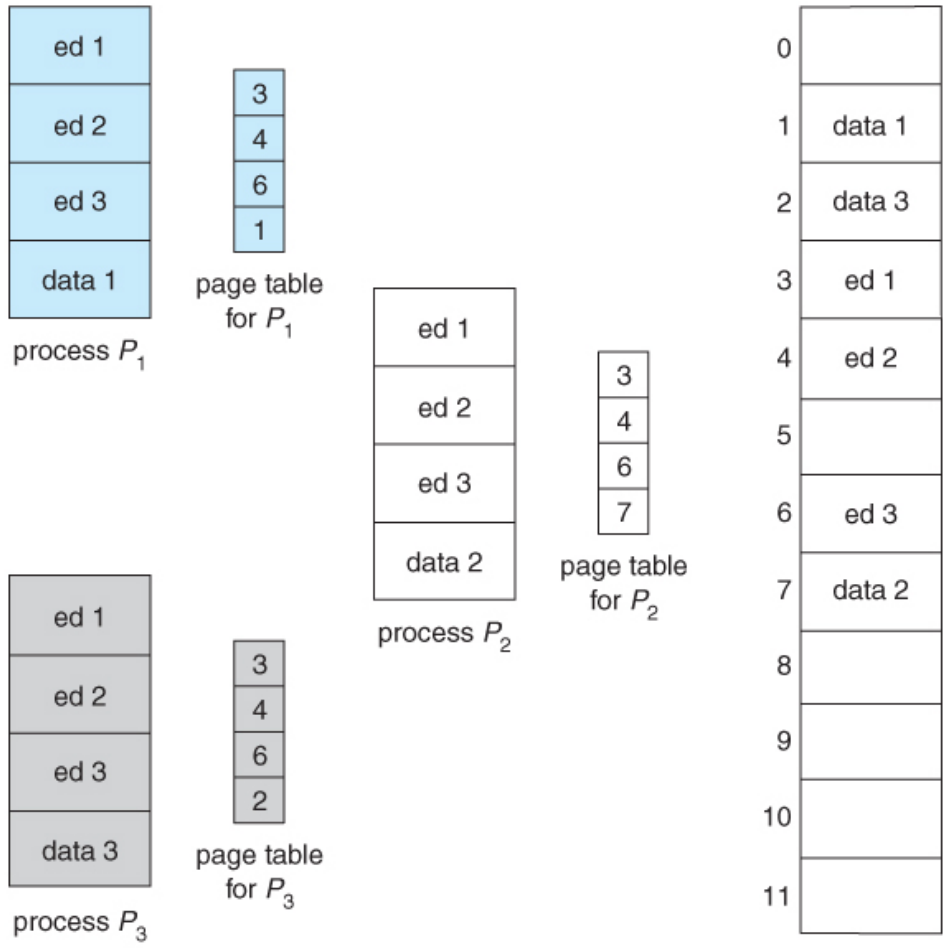
Sharing Pages

- Paging systems can make it very easy to share blocks of memory, since memory doesn't have to be contiguous anymore
- This can be done by simply duplicating page entries of different processes to the same page frames (both for code and data)

Sharing Pages

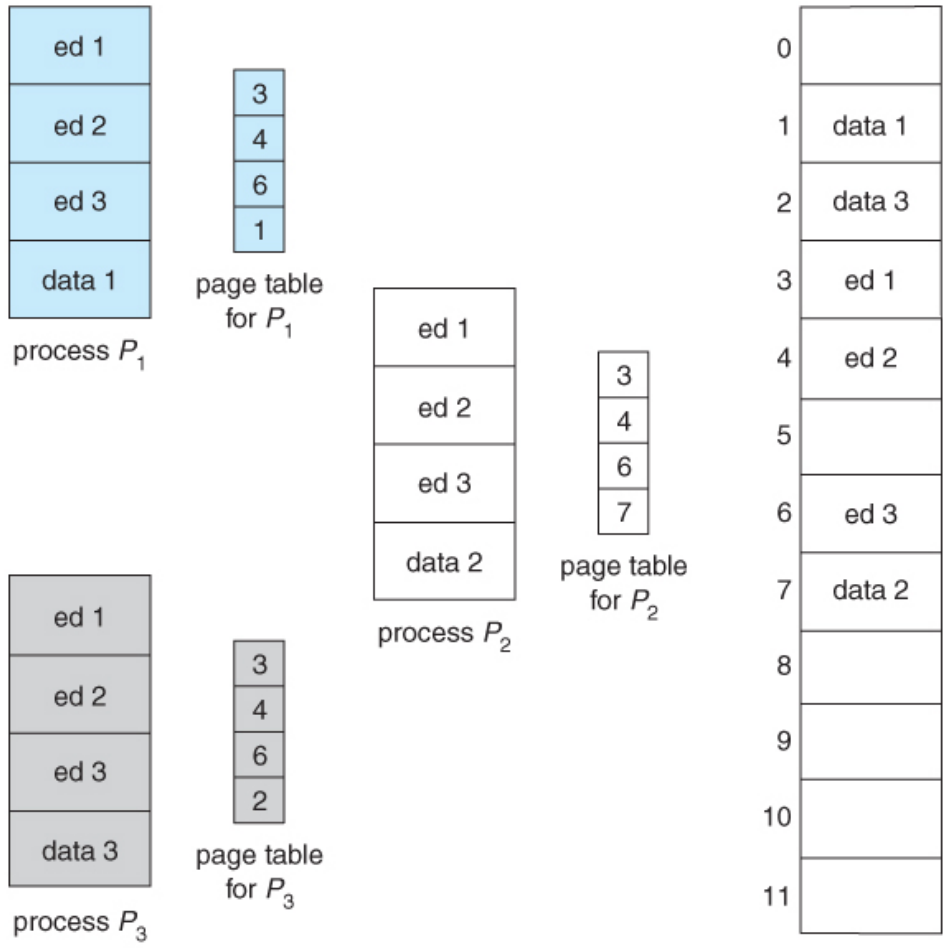
- Paging systems can make it very easy to share blocks of memory, since memory doesn't have to be contiguous anymore
- This can be done by simply duplicating page entries of different processes to the same page frames (both for code and data)
- Only if code is **reentrant**:
  - it does not write to 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: Summary

- 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

Paging: Summary

- 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
- However, paging comes with its costs:
  - Virtual/Physical address translation may be time consuming
  - Hardware support like TLB cache is needed to make it efficient enough
  - OS has to be inevitably more complex

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

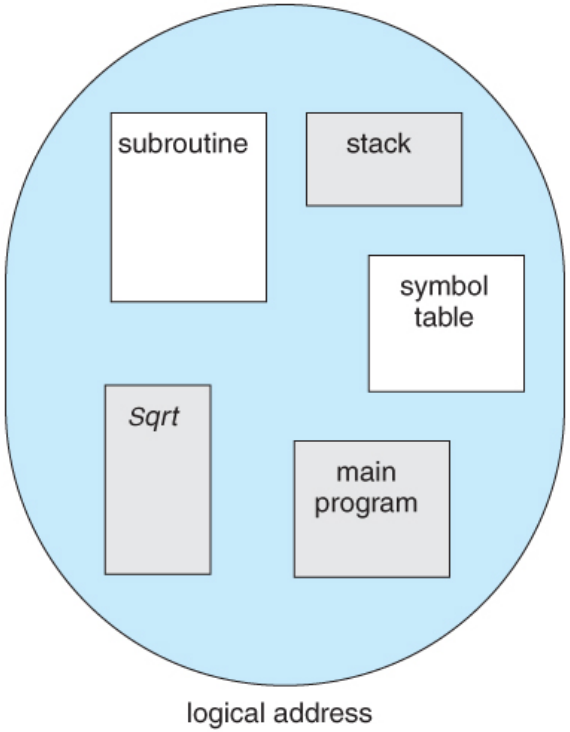
- Most users (programmers) do not think of their programs as existing in one continuous linear address space
- Rather they think of memory divided in multiple **segments**, each dedicated to a specific use, such as code, data, stack, heap, etc.

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** from the beginning of that segment

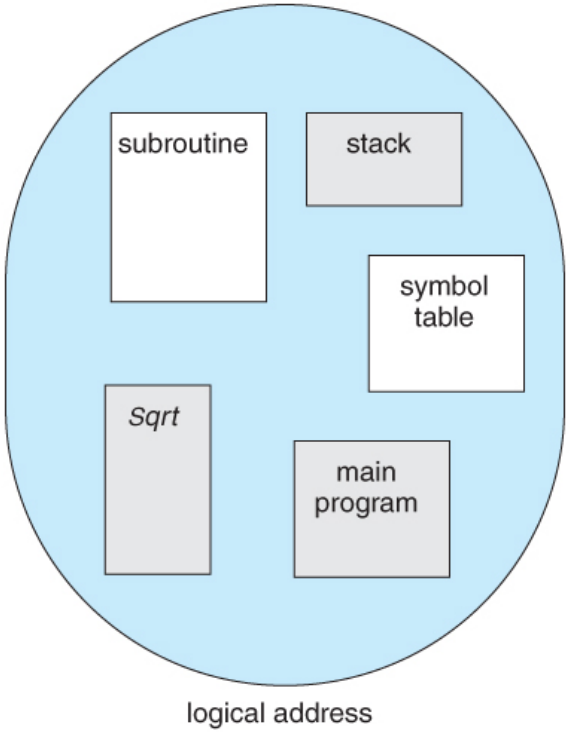
Segmentation: Example

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



Segmentation: Example

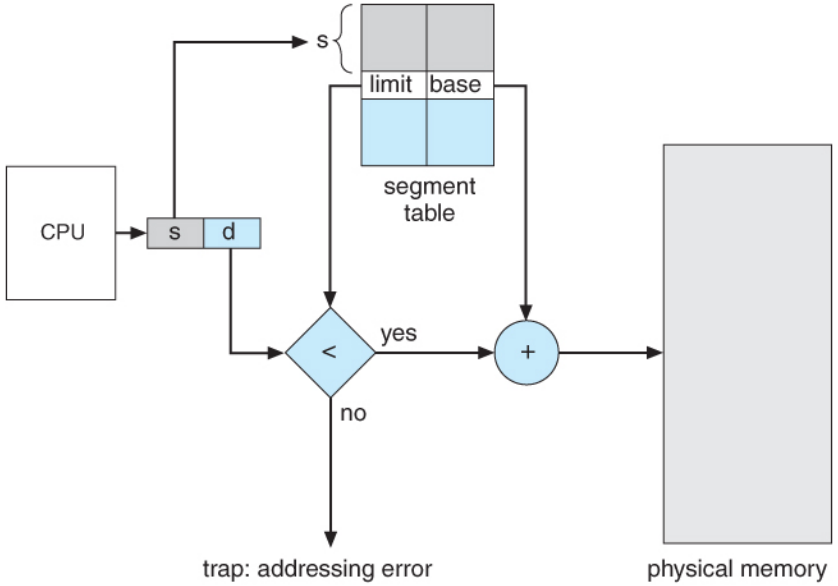
A C compiler generating **5 segments** for the user code, library code, global (static) variables, the stack, and the heap



The compiler generates addresses identifying segments and offset in those

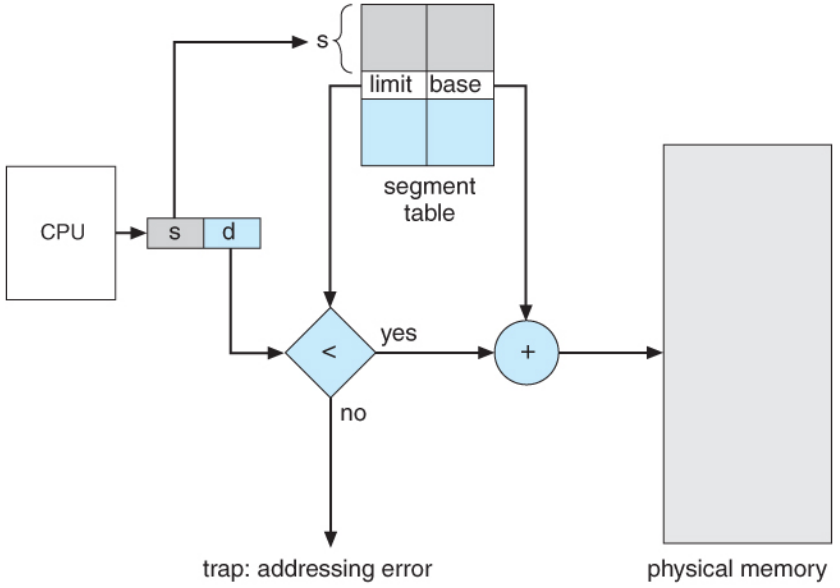
Segmentation Hardware

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



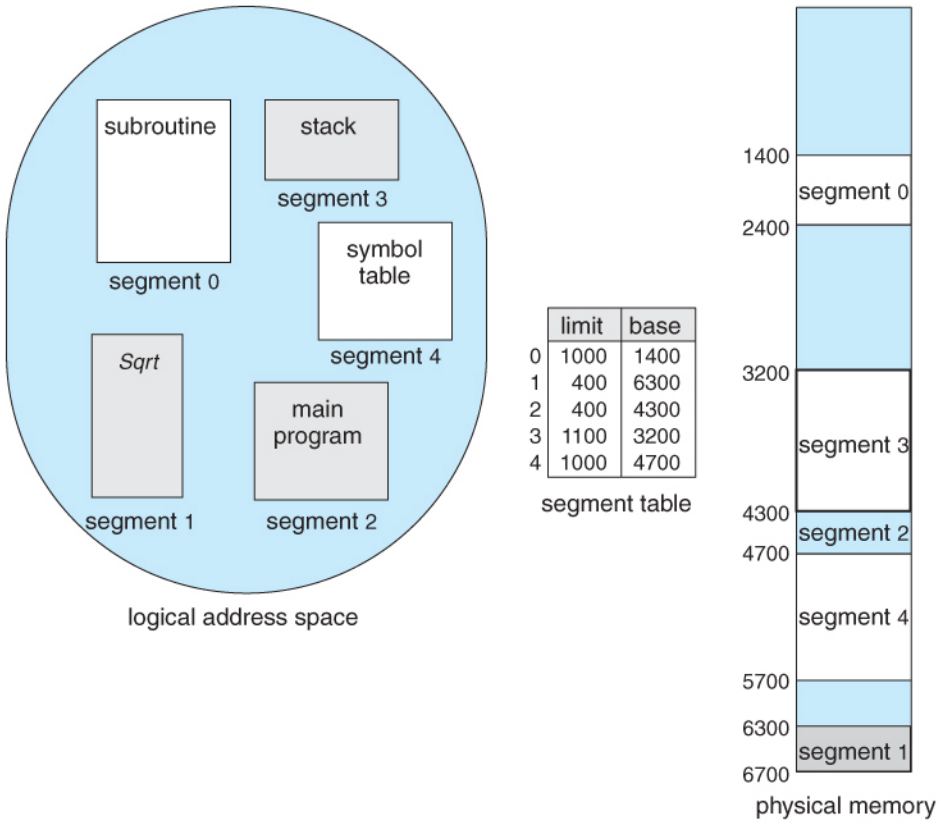
Segmentation Hardware

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



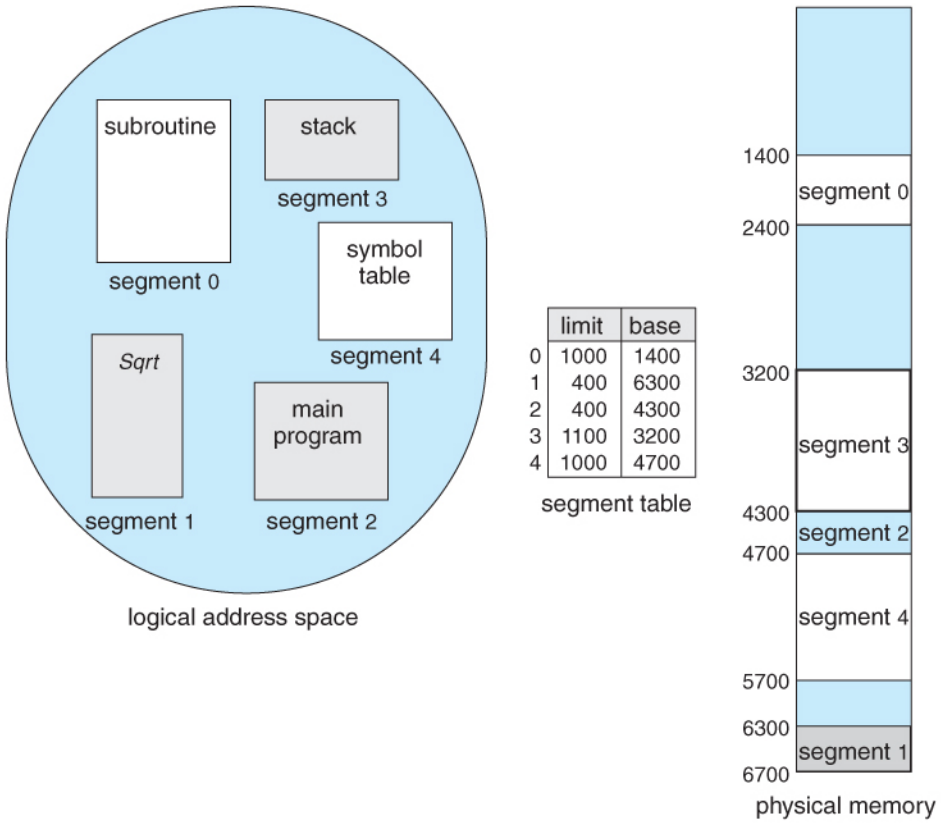
Note that we came back to the assumption that each segment is kept in **contiguous** memory and may be of different sizes…

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)

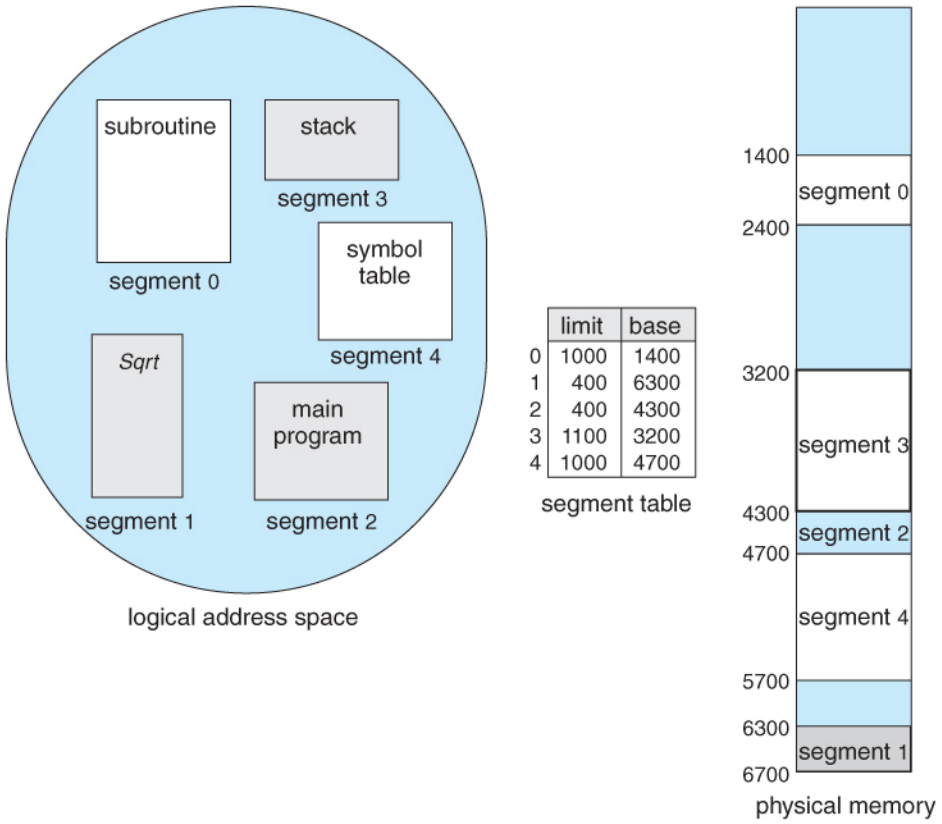
Segment Table



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

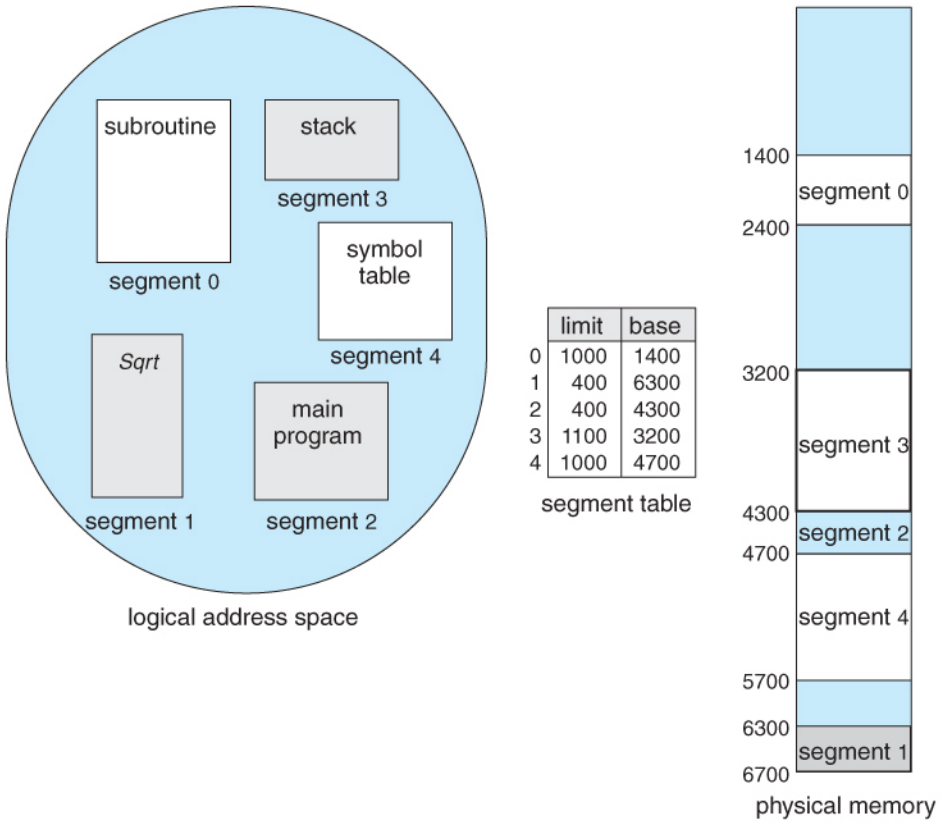


Each entry contains a base address in memory, the length of the segment, plus additional protection information (e.g., sharing, read/write permissions)

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 needs to generate virtual addresses whose top-most significant bits indicate the segment number

Implementing (Basic) Segmentation

- Compiler needs to generate 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 needs to generate 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!

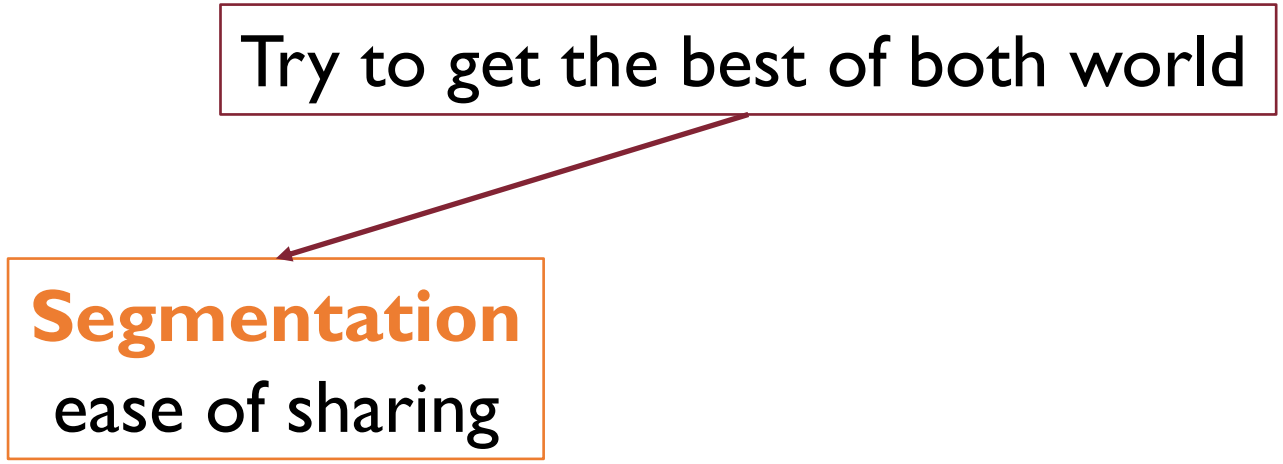
Implementing (Basic) Segmentation

- Compiler needs to generate 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

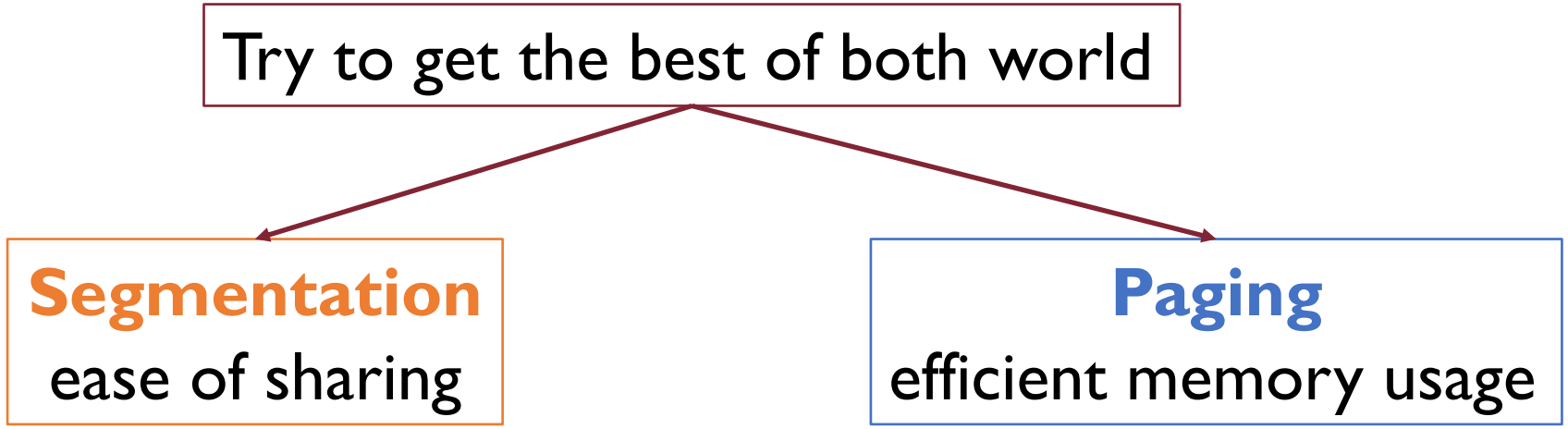
Combine Segmentation with Paging

Try to get the best of both world

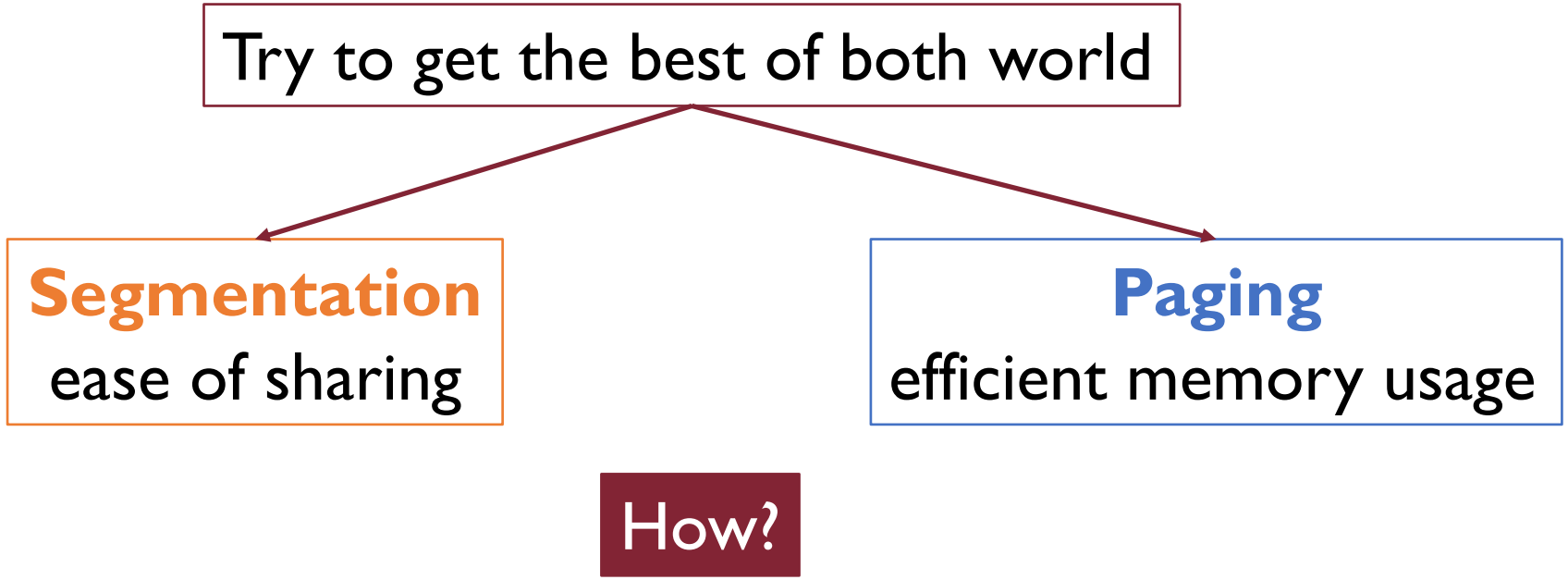
Combine Segmentation with Paging



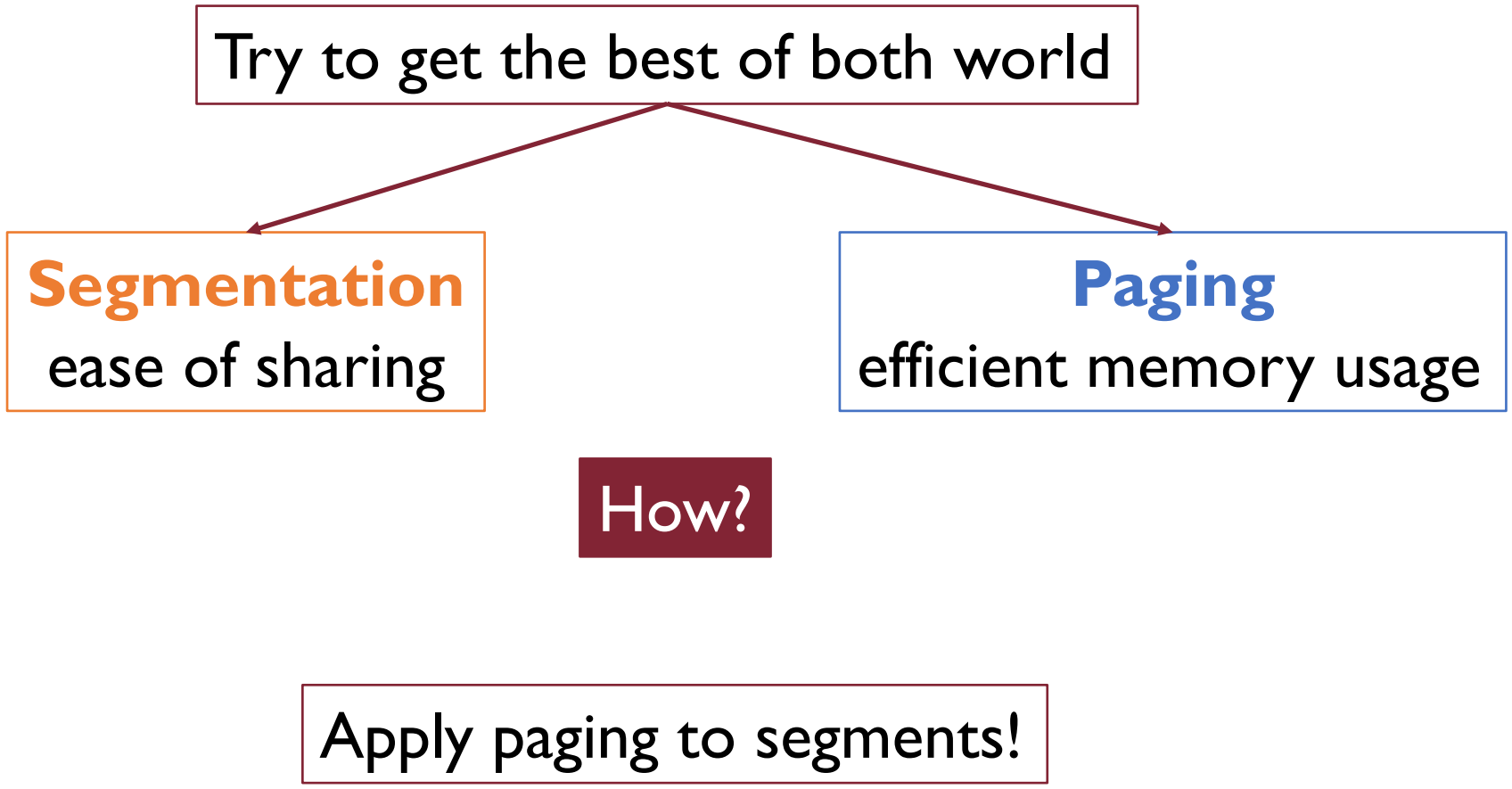
Combine Segmentation with Paging



Combine Segmentation with Paging



Combine Segmentation with Paging



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

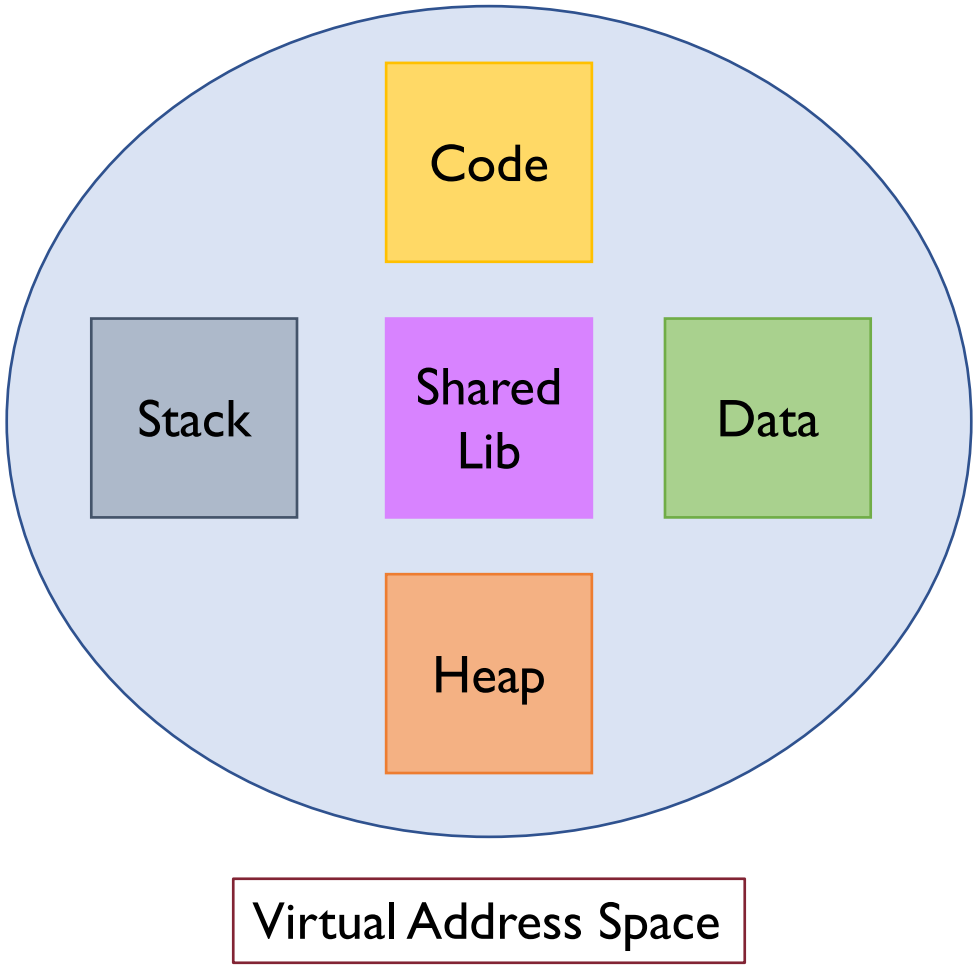
- Each process' virtual address space is seen as a collection of segments (i.e., logical units of arbitrary size)
- Physical address space is still seen as a sequence of fixed-size frames
- Segments are usually larger than physical page frames

Combine Segmentation with Paging

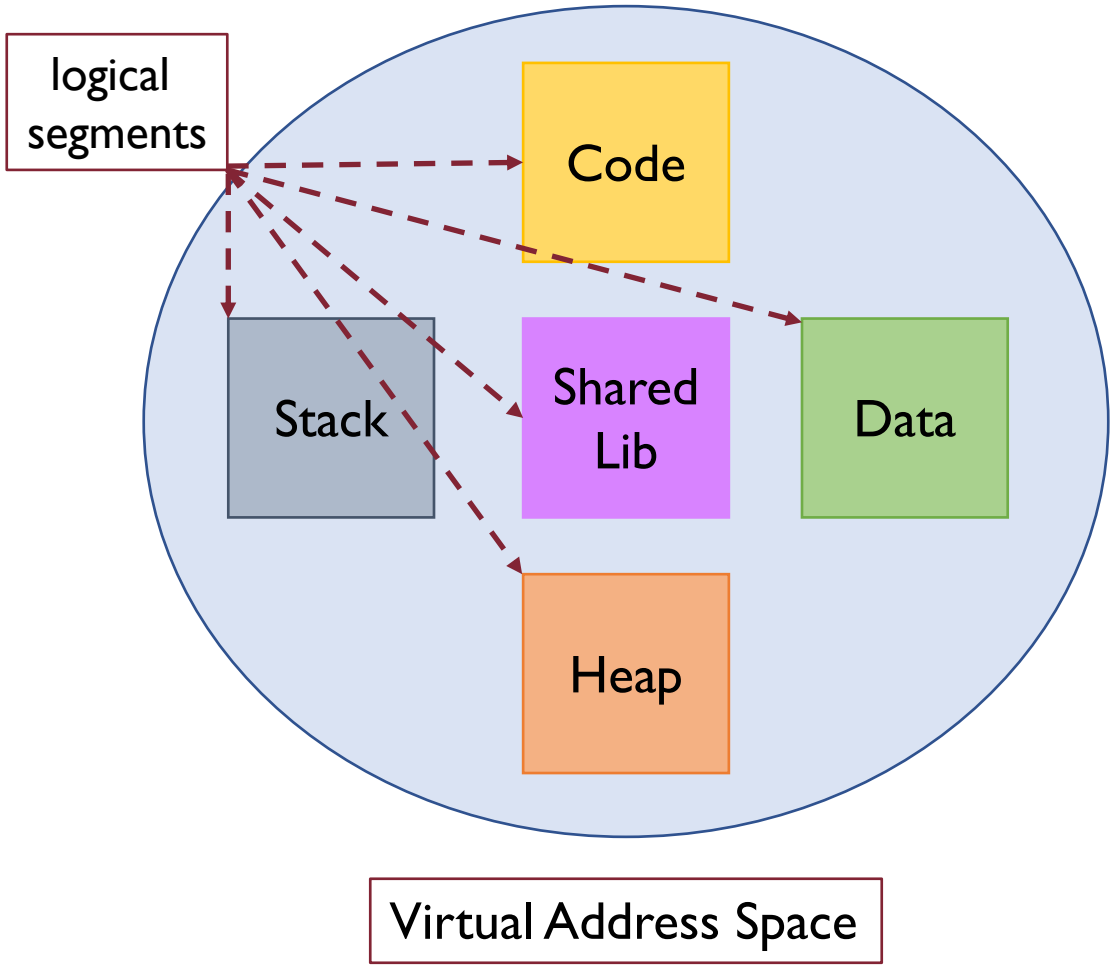
- Each process' virtual address space is seen as a collection of segments (i.e., logical units of arbitrary size)
- Physical address space is still seen as a sequence of fixed-size frames
- Segments are usually larger than physical page frames

Map a logical segment onto multiple page frames

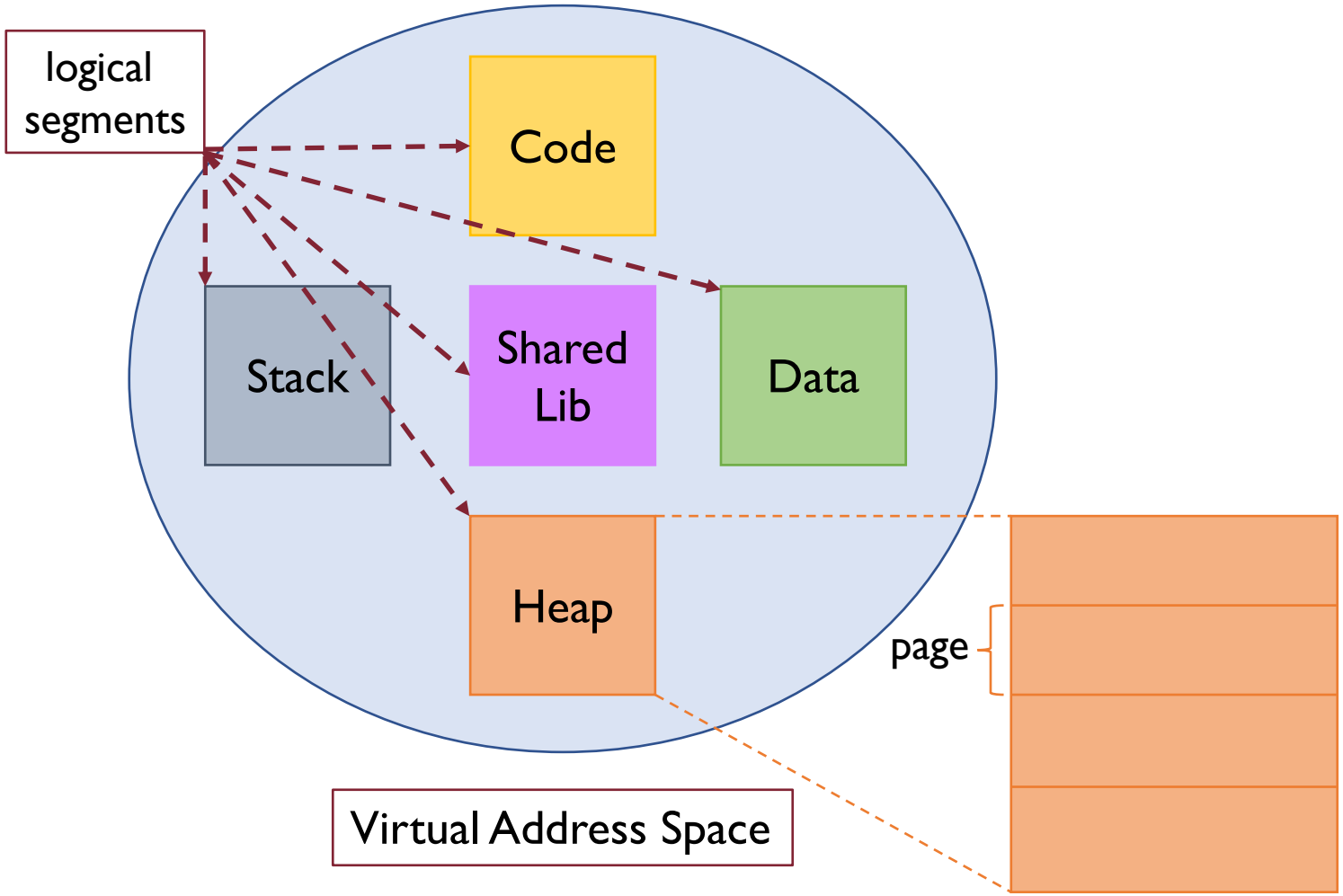
Paging Logical Segments



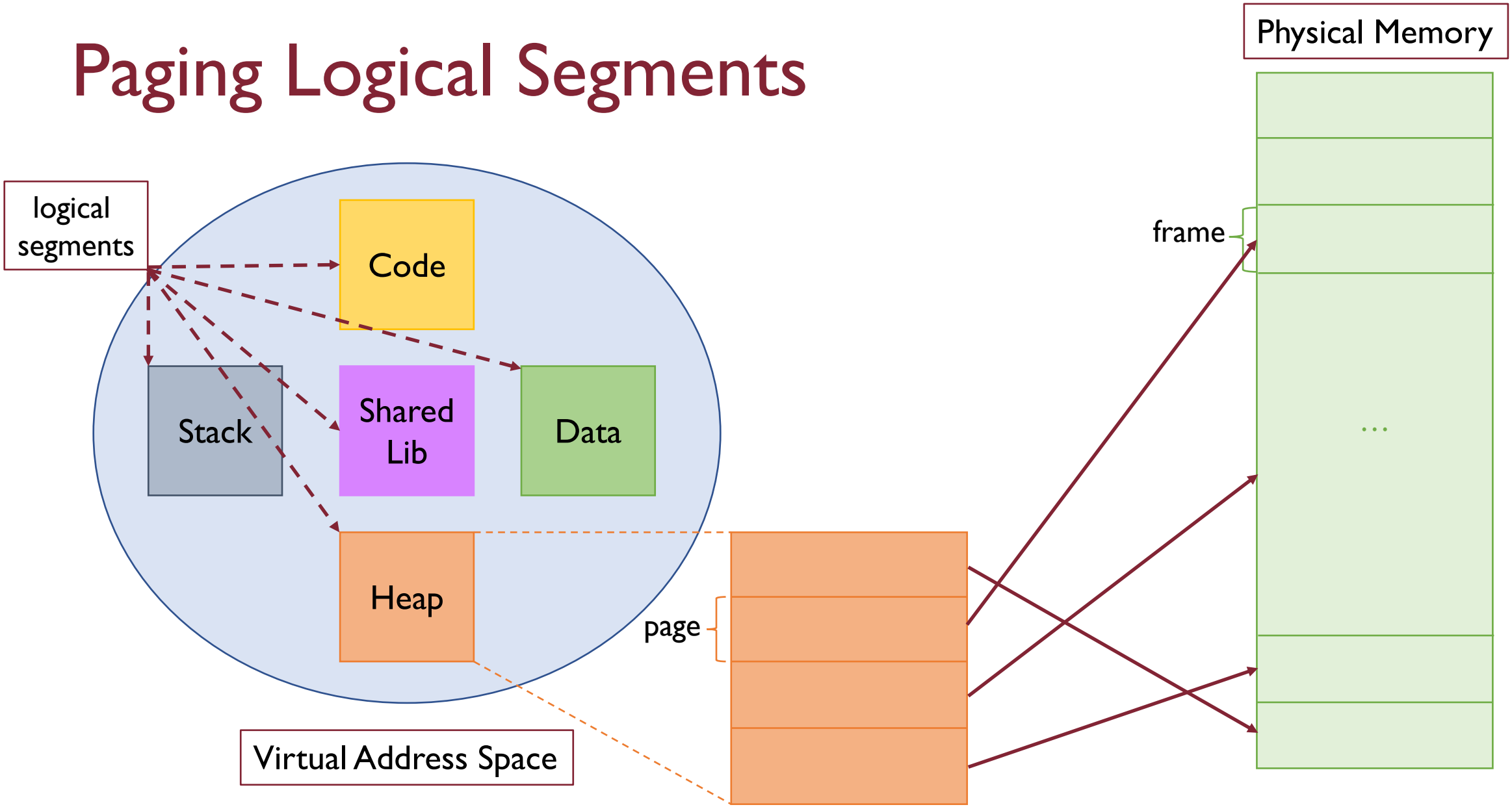
Paging Logical Segments



Paging Logical Segments

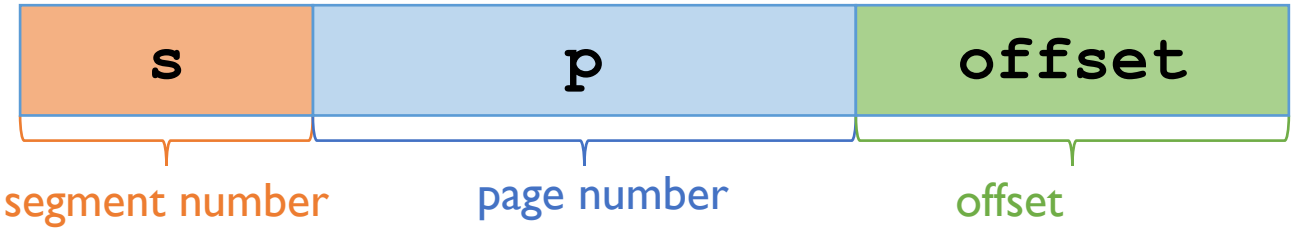


Paging Logical Segments



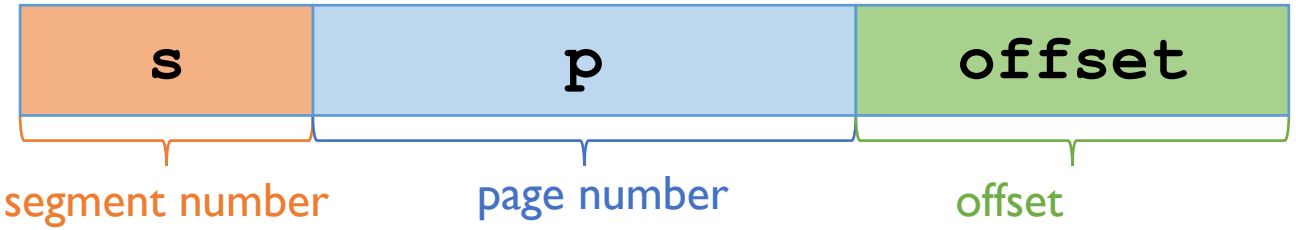
Address Translation with Segmented Paging

A virtual address now becomes:



Address Translation with Segmented Paging

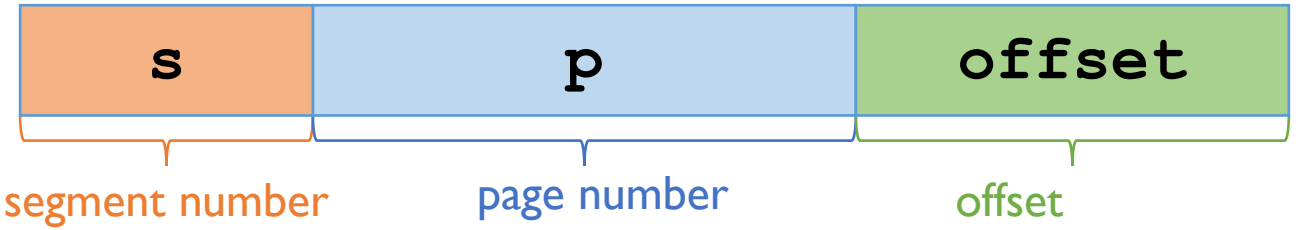
A virtual address now becomes:



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

Address Translation with Segmented Paging

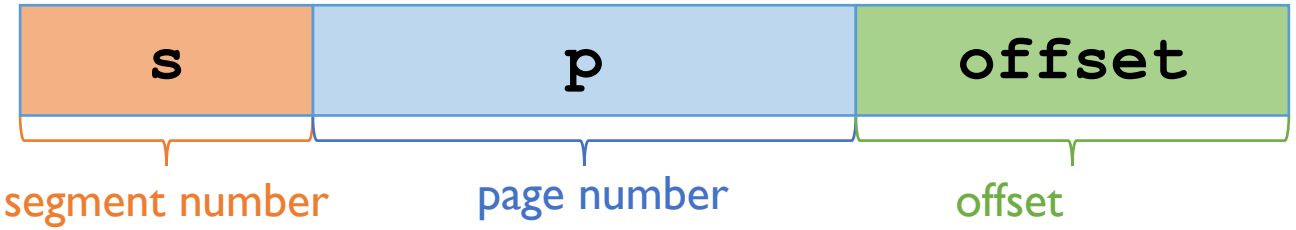
A virtual address now becomes:



- The segment number indexes into the **segment table**, which contains the base address of the **page table** for *that* segment
- Check the page number + offset against the limit of the segment

Address Translation with Segmented Paging

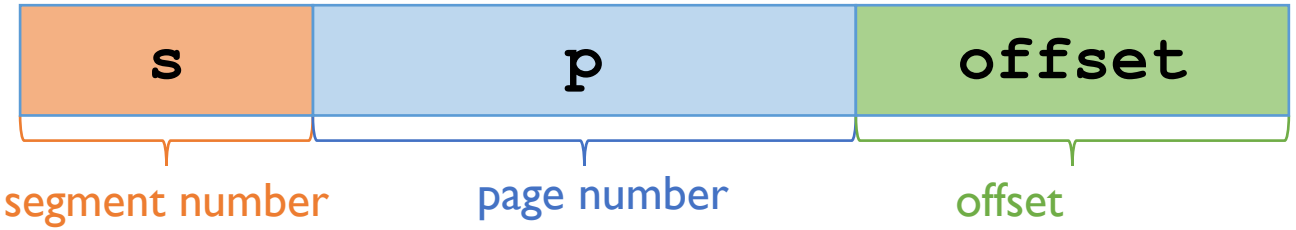
A virtual address now becomes:



- The segment number indexes into the **segment table**, which contains the base address of the **page table** for *that* segment
- Check the page number + offset against the limit of the segment
- Use the page number to index the page table to get the physical frame number

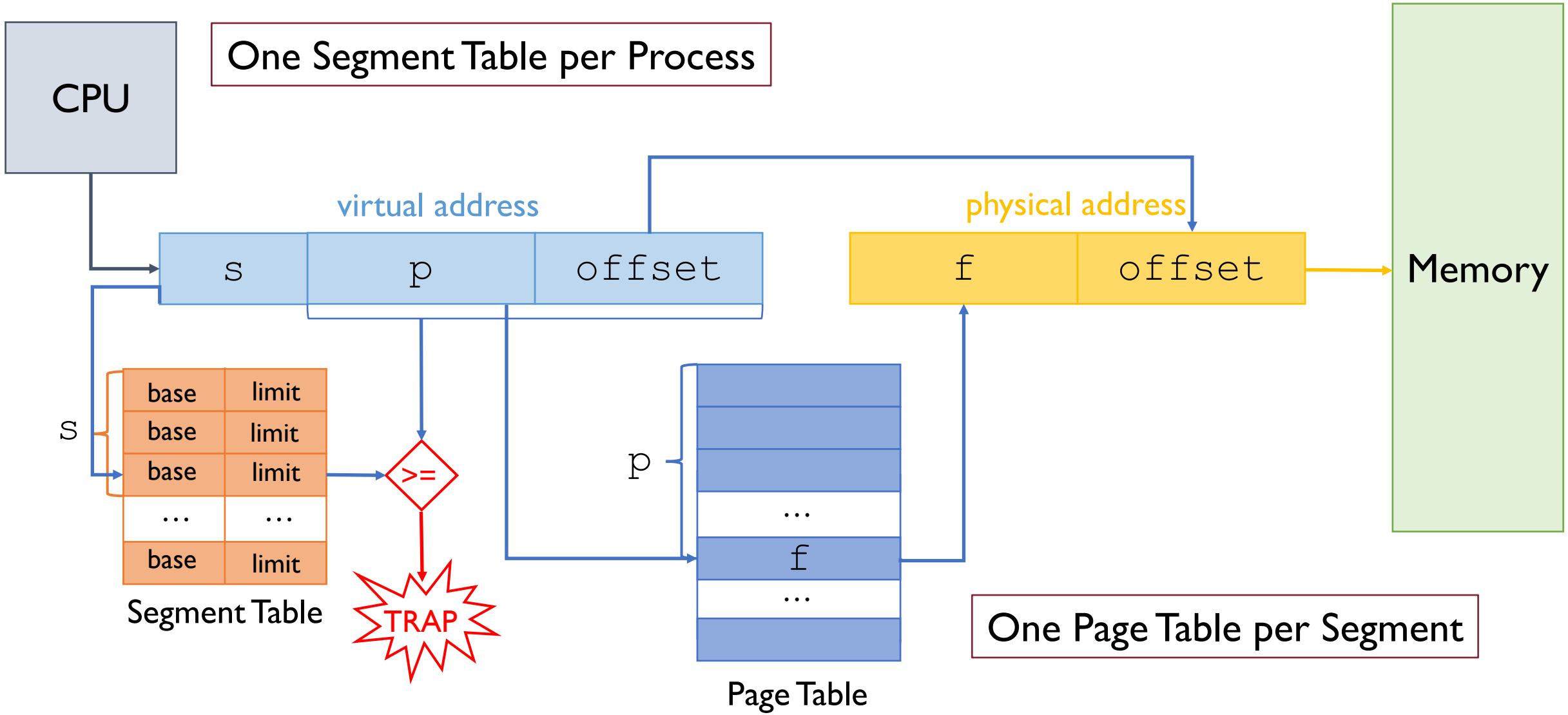
Address Translation with Segmented Paging

A virtual address now becomes:



- The segment number indexes into the **segment table**, which contains the base address of the **page table** for *that* segment
- Check the page number + offset against the limit of the segment
- Use the page number to index the page table to get the physical frame number
- Add the frame number to the offset to get the physical address

Address Translation with Segmented Paging



Segmented Paging: Implementation Issues

Where are **segment tables** and **page tables** stored?

Segmented Paging: Implementation Issues

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 Issues

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

Segmented Paging: Implementation Issues

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 Issues

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

Slower but more flexible

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

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

Q2

How many bits are therefore needed for the virtual 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

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:
  - 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
  - No external fragmentation
  - Sharing memory between processes

Segmented Paging: Benefits and Costs

- **Benefits:**
  - Merge compiler and OS view of memory
  - Flexibility
  - No external fragmentation
  - 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

Advanced Paging

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

Advanced Paging

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

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

Advanced Paging

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

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

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

Advanced Paging

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

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

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

Note that the page table itself must be paged, and with 4KiB page size this would take 2^{10} = 1024 pages just to hold the page table!

Advanced Paging

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

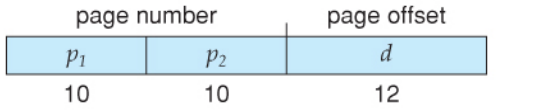
With a 2^{32} address space and 4KiB (2^{12}) page size, there are 2^{20} ~ 1 million entries in the page table

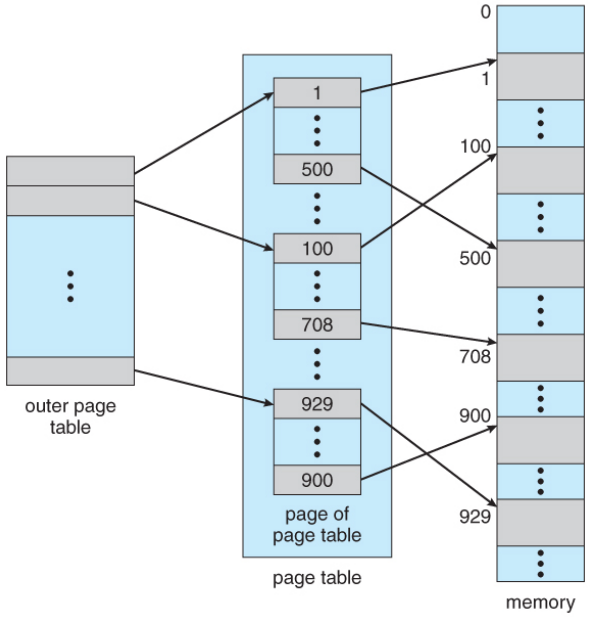
At 4 bytes per entry, this amounts to a 4 MiB page table, which may be too large to reasonably keep in contiguous memory

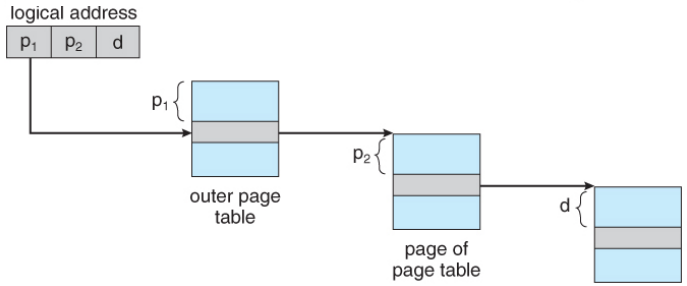
Note that the page table itself must be paged, and with 4KiB page size this would take 2^{10} = 1024 pages just to hold the page table!

More advanced paging structures are needed!

Advanced Paging: Two-Tier Page Table

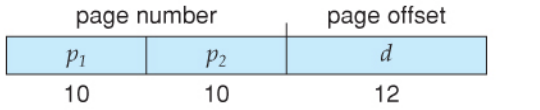


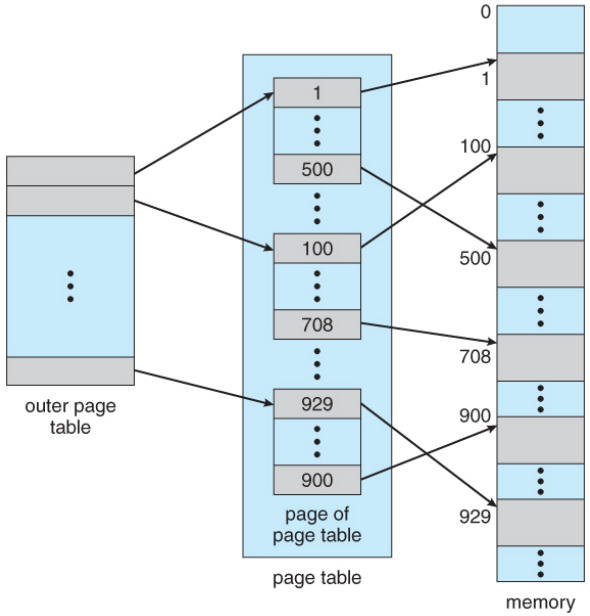


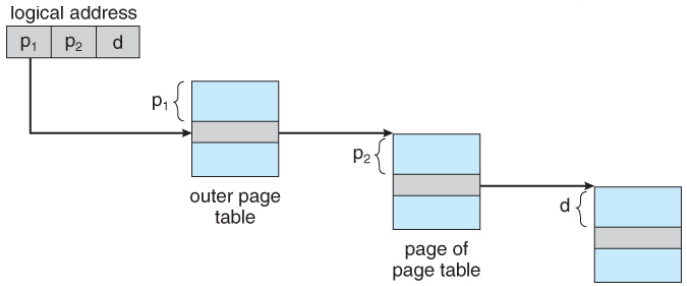


Let's page the page table!

Advanced Paging: Two-Tier Page Table



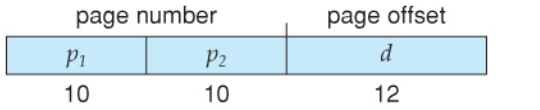


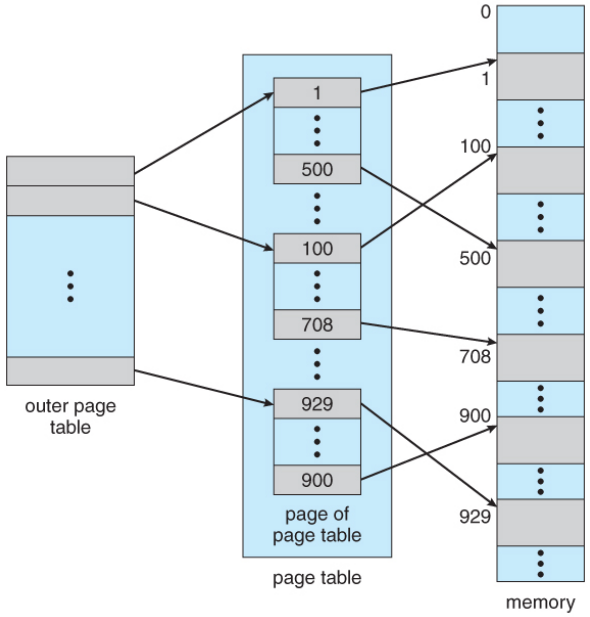


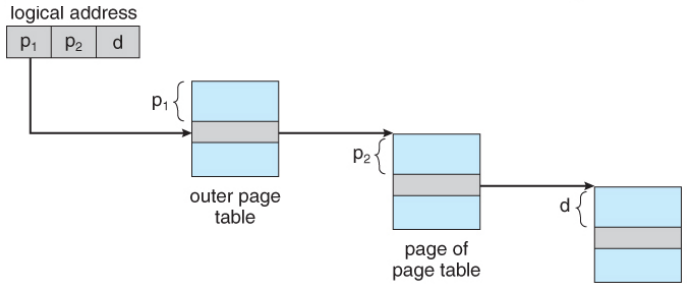
Let's page the page table!

20-bit page number broken into 2 10-bit page numbers

Advanced Paging: Two-Tier Page Table





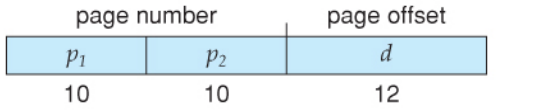


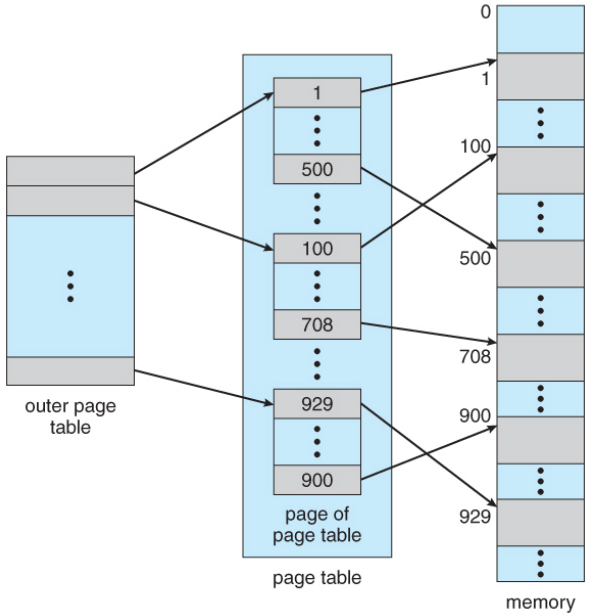
Let's page the page table!

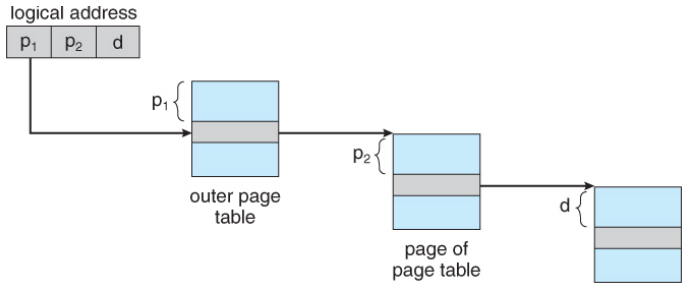
20-bit page number broken into 2 10-bit page numbers

The most significant 10 bits represent an entry in the outer page table, used to find one page of the inner page table

Advanced Paging: Two-Tier Page Table







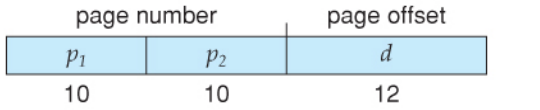
Let's page the page table!

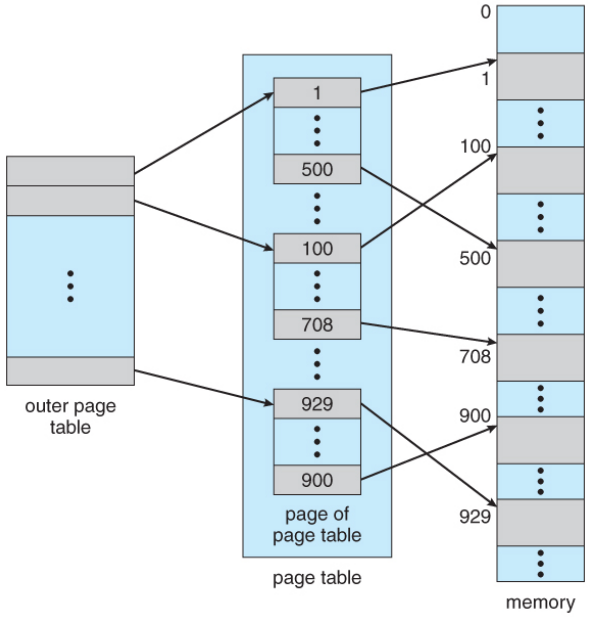
20-bit page number broken into 2 10-bit page numbers

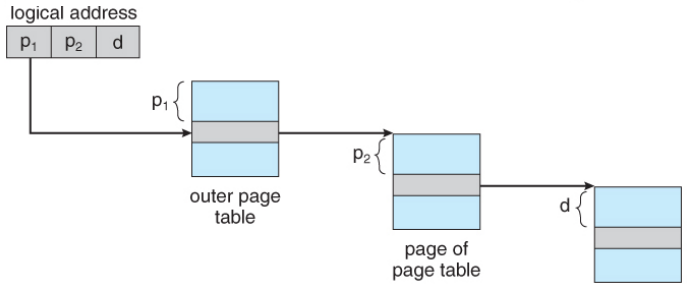
The most significant 10 bits represent an entry in the outer page table, used to find one page of the inner page table

The second 10 bits finds a specific entry in that inner page table, which maps to a frame in physical memory

Advanced Paging: Two-Tier Page Table







Let's page the page table!

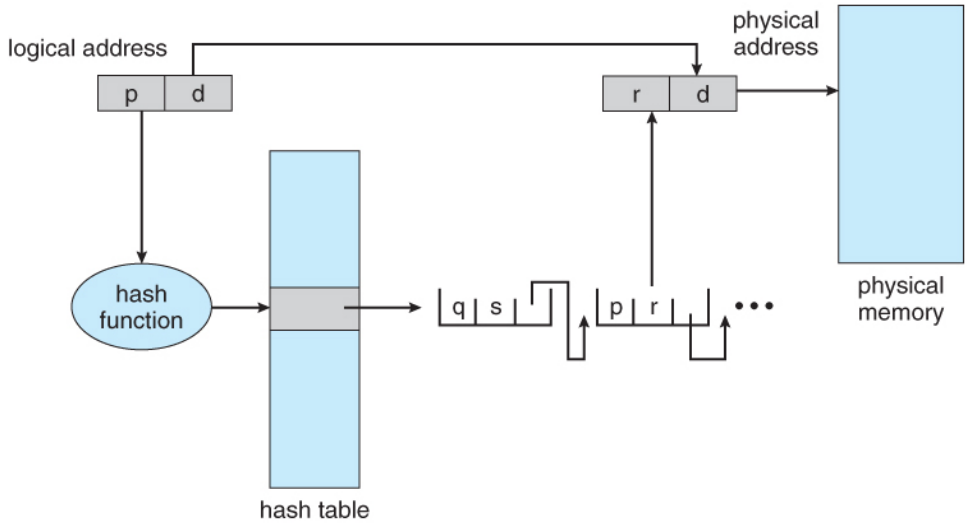
20-bit page number broken into 2 10-bit page numbers

The most significant 10 bits represent an entry in the outer page table, used to find one page of the inner page table

The second 10 bits finds a specific entry in that inner page table, which maps to a frame in physical memory

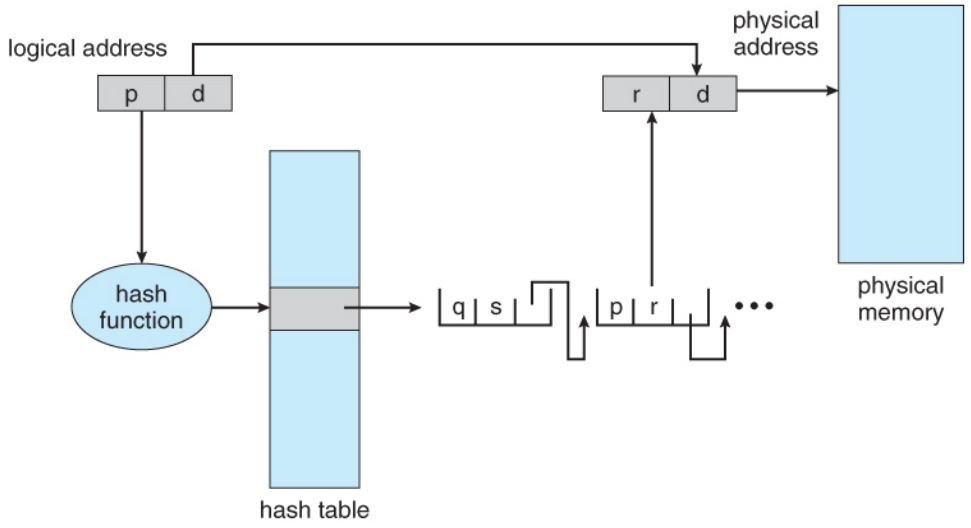
The remaining 12 bits of the 32-bit logical address are still the offset within the 4KiB frame

Advanced Paging: Hashed Page Table



Use hash tables to store highly sparse page tables

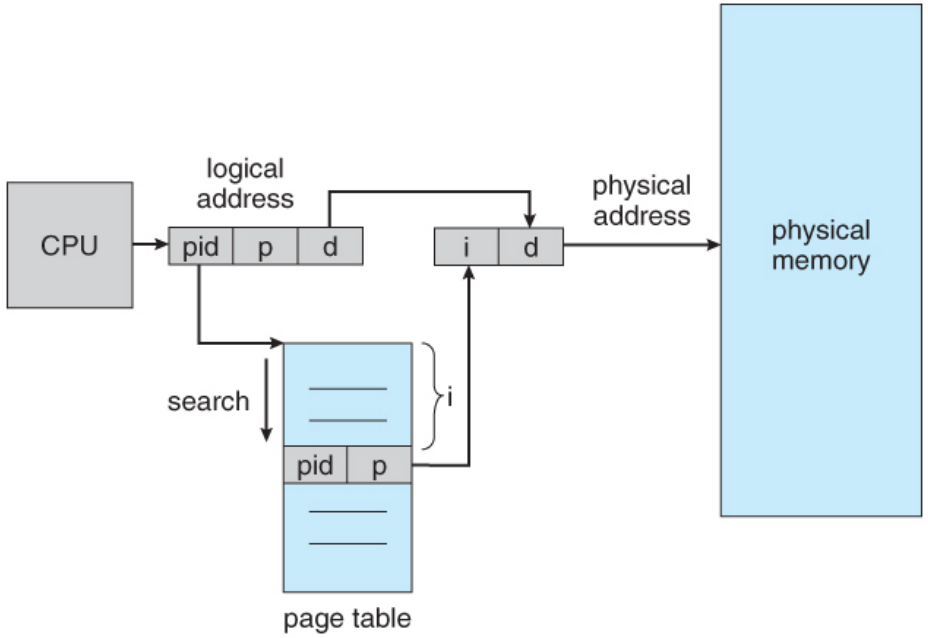
Advanced Paging: Hashed Page Table



Use hash tables to store highly sparse page tables

Indexing via hash function rather than integers

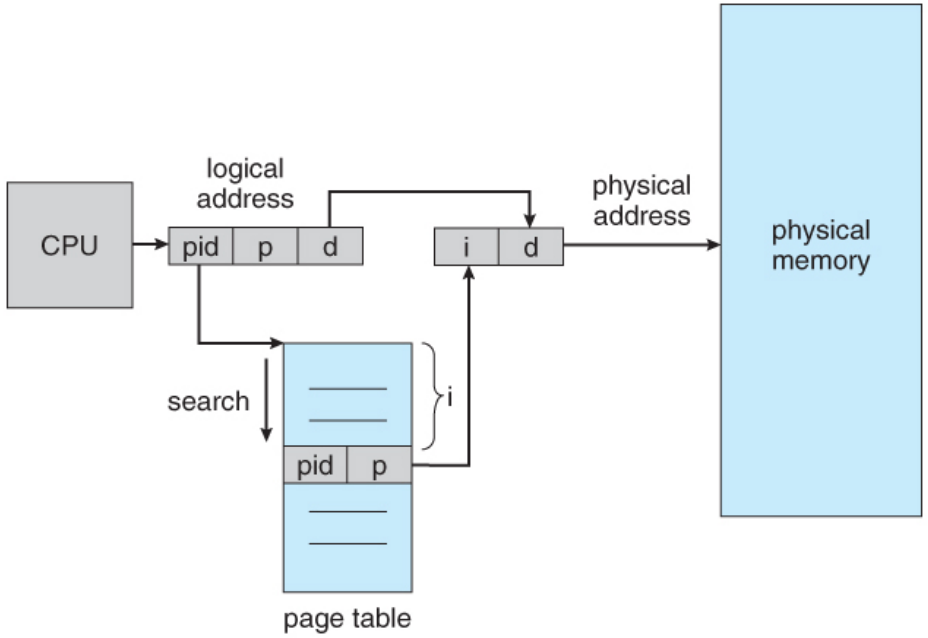
Advanced Paging: Inverted Page Table



An inverted page table lists all of the pages 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



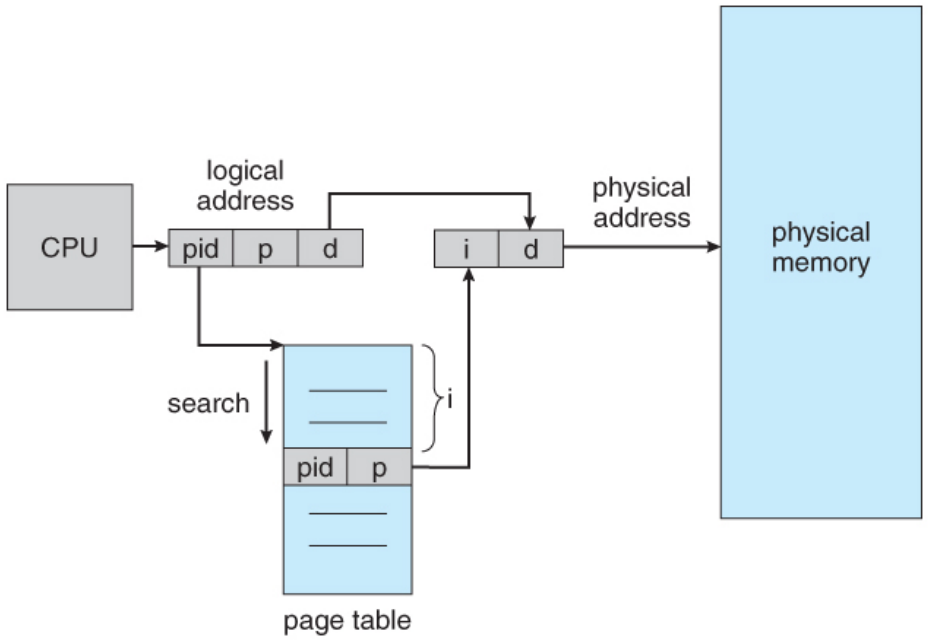
An inverted page table lists all of the pages currently loaded in memory, for all processes

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 can help speedup the search process

Advanced Paging: Inverted Page Table



An inverted page table lists all of the pages 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 can help speedup 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

- **Paging**
  - Simplifies memory allocation by relaxing contiguous assumption
  - Each logical page can be allocated to any physical frame
  - Page tables can be extremely large
- **Segmentation + Paging**
  - Only need to allocate as many page table entries an needed
  - Sharing either at the segment or at the page level
  - Might increase internal fragmentation over pure paging
  - 2 lookups per memory reference are needed