Systems and Networking – Unit I

B.Sc. in Applied Computer Science and Artificial Intelligence

2021-2022

Gabriele Tolomei

Department of Computer Science

Sapienza Università di Roma

tolomei@di.uniroma1.it

SAPIENZA
UNIVERSITÀ DI ROMA

Contiguous Memory Allocation

- So far, we have assumed each process is allocated into a contiguous space of physical memory

Contiguous Memory Allocation

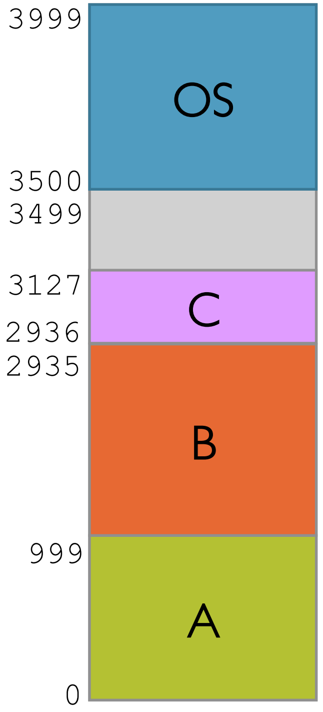
- So far, we have assumed each process is allocated into a contiguous space of physical memory
- One simple method is to divide upfront all available memory dedicated to user processes into equally-sized segments/partitions
  - Assign each process to a segment
  - Implicitly restricts the grade of multiprogramming (i.e., the number of simultaneous processes) and their size
  - No longer used!

Contiguous Memory Allocation

An alternative approach is for the OS to keep track of free (unused) memory segments, as processes enter the system, grow, and terminate

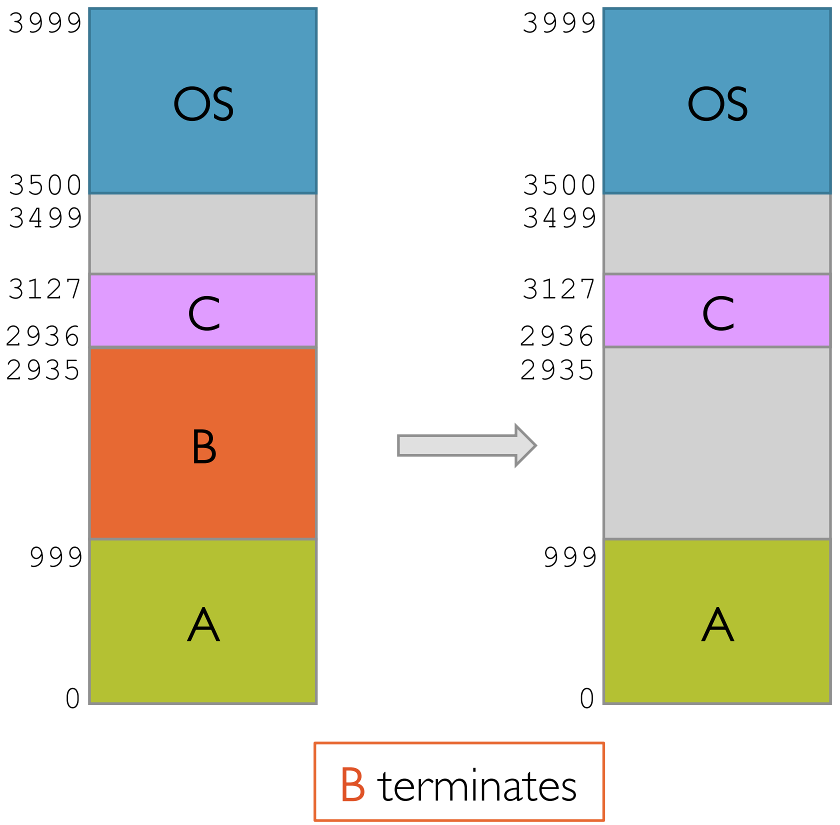
Contiguous Memory Allocation

An alternative approach is for the OS to keep track of **free** (unused) memory segments, as processes enter the system, grow, and terminate



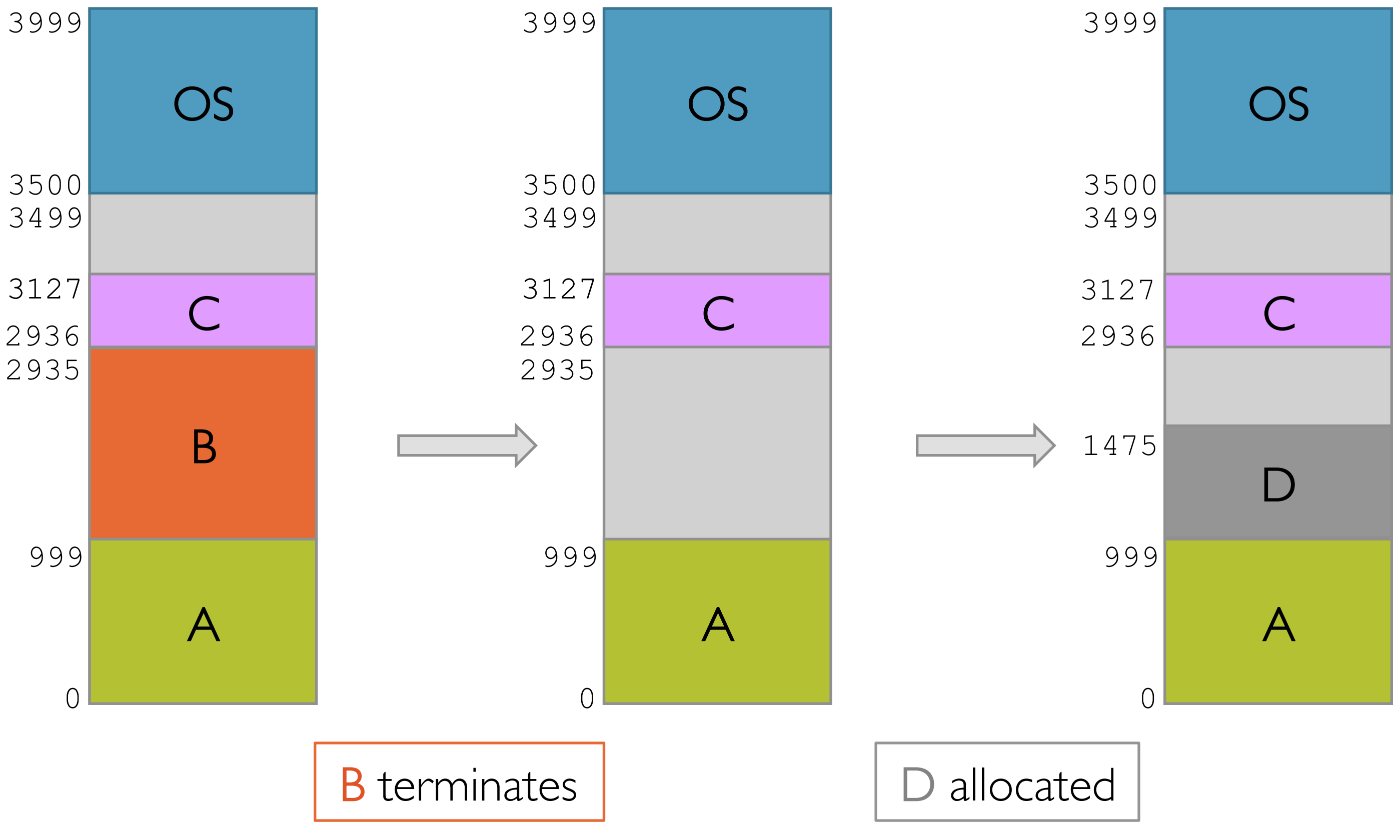
Contiguous Memory Allocation

An alternative approach is for the OS to keep track of free (unused) memory segments, as processes enter the system, grow, and terminate



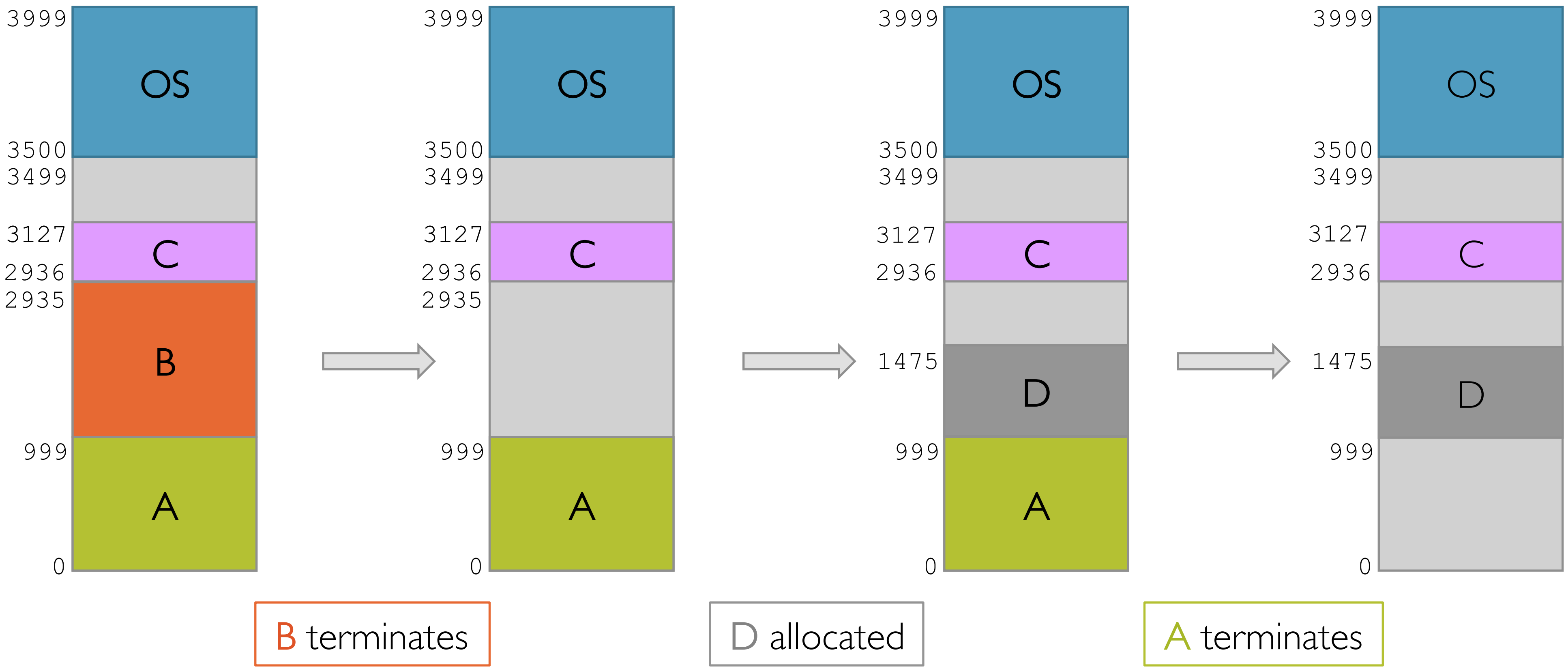
Contiguous Memory Allocation

An alternative approach is for the OS to keep track of free (unused) memory segments, as processes enter the system, grow, and terminate

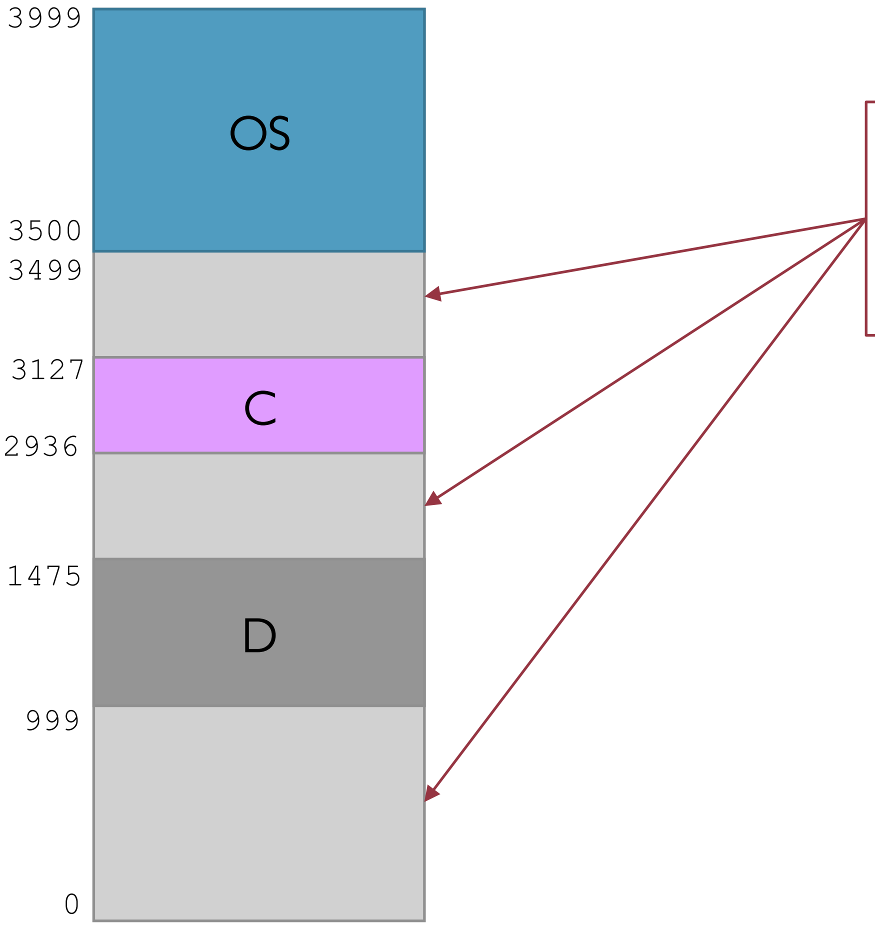


Contiguous Memory Allocation

An alternative approach is for the OS to keep track of **free** (unused) memory segments, as processes enter the system, grow, and terminate

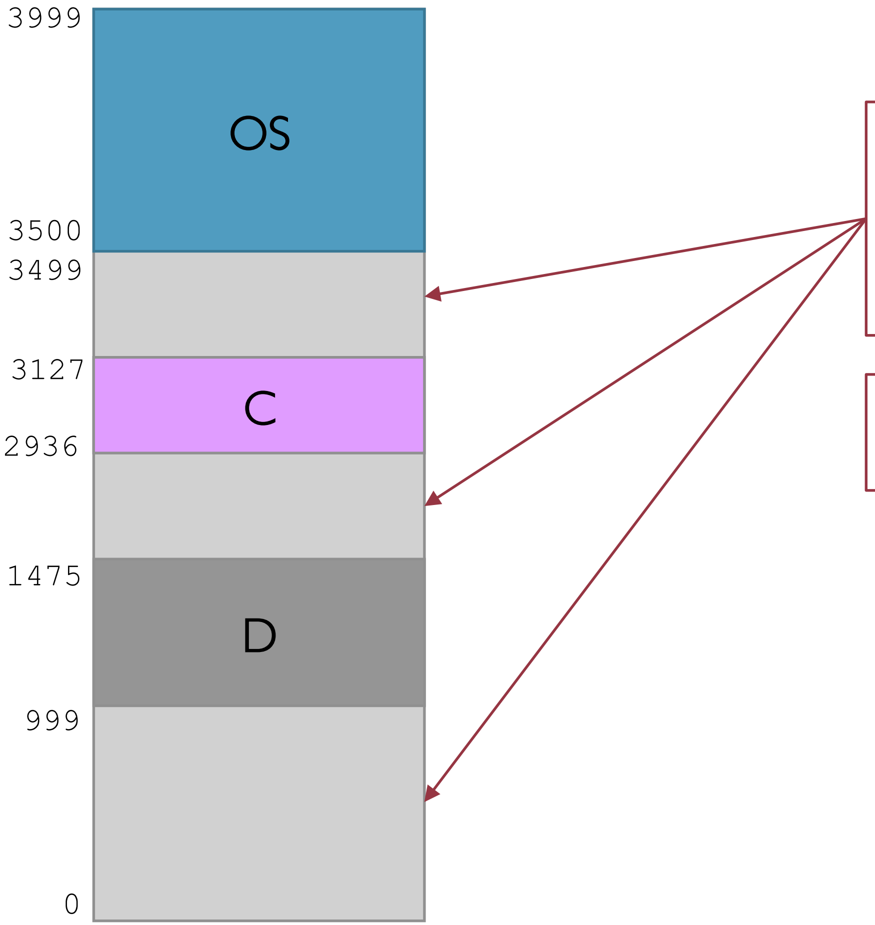


Contiguous Memory Allocation



Unused segments of memory are called **holes**

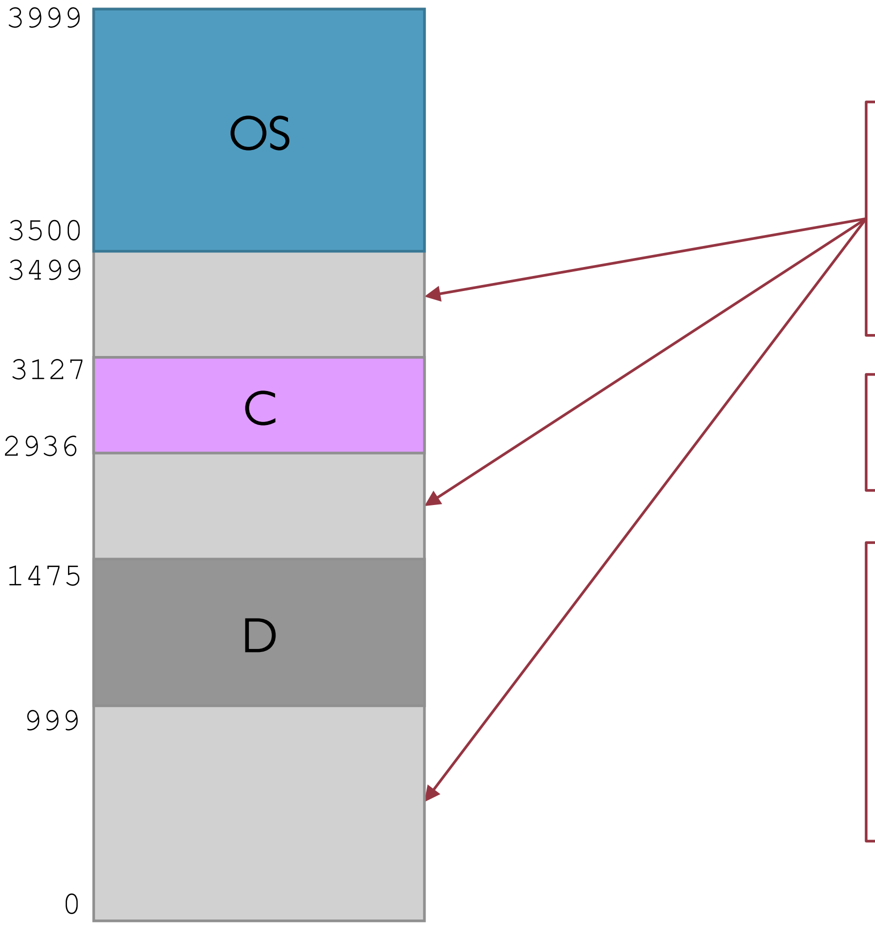
Contiguous Memory Allocation



Unused segments of memory are called **holes**

The OS keeps a list of holes

Contiguous Memory Allocation

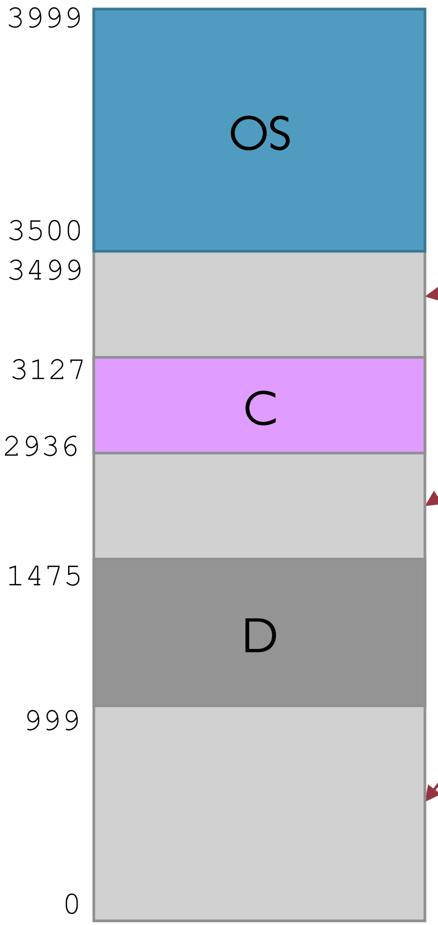


Unused segments of memory are called **holes**

The OS keeps a list of holes

Whenever a process has to be loaded, the OS must select a hole of suitable size

Contiguous Memory Allocation



Unused segments of memory are called **holes**

The OS keeps a list of holes

Whenever a process has to be loaded, the OS must select a hole of suitable size

How?

Memory Allocation Policies: First-Fit

- Linearly scan the list of holes until one is found that is big enough to satisfy the request

Memory Allocation Policies: First-Fit

- Linearly scan the list of holes until one is found that is big enough to satisfy the request
- Subsequent requests may either start from the beginning of the list or from the end of previous search

Memory Allocation Policies: First-Fit

- Linearly scan the list of holes until one is found that is big enough to satisfy the request
- Subsequent requests may either start from the beginning of the list or from the end of previous search
- Complexity: $O(n)$, where n is the number of holes

Memory Allocation Policies: First-Fit

Suppose process X needs 100B of memory to be loaded, and the list of holes is as follows:

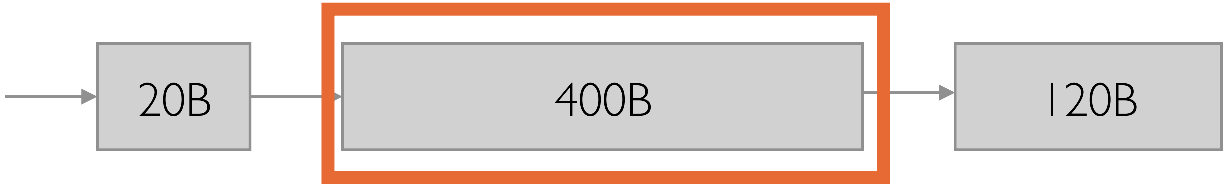
Memory Allocation Policies: First-Fit

Suppose process X needs 100B of memory to be loaded, and the list of holes is as follows:

Which segment will be picked by the OS using first-fit?

Memory Allocation Policies: First-Fit

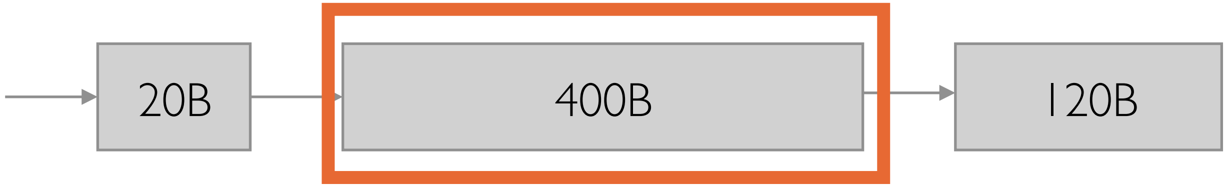
Suppose process X needs 100B of memory to be loaded, and the list of holes is as follows:



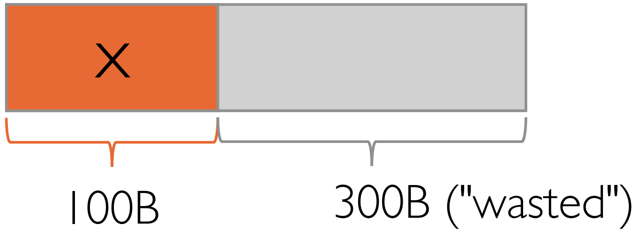
Which segment will be picked by the OS using first-fit?

Memory Allocation Policies: First-Fit

Suppose process X needs 100B of memory to be loaded, and the list of holes is as follows:

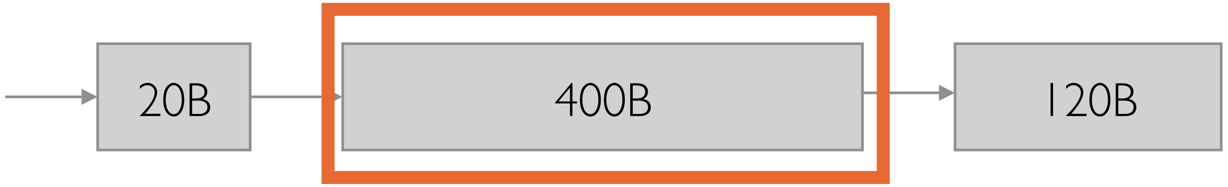


Which segment will be picked by the OS using first-fit?

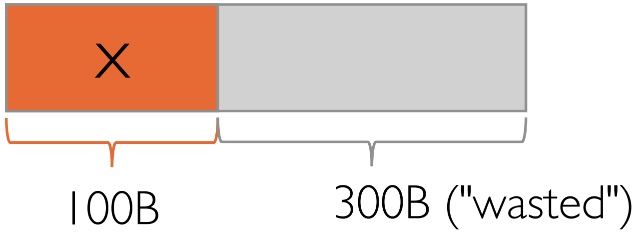


Memory Allocation Policies: First-Fit

Suppose process X needs 100B of memory to be loaded, and the list of holes is as follows:



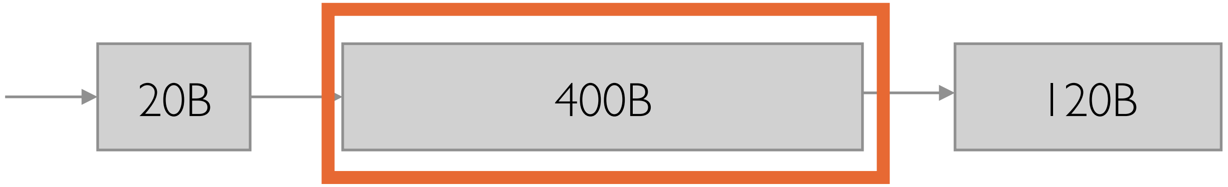
Which segment will be picked by the OS using first-fit?



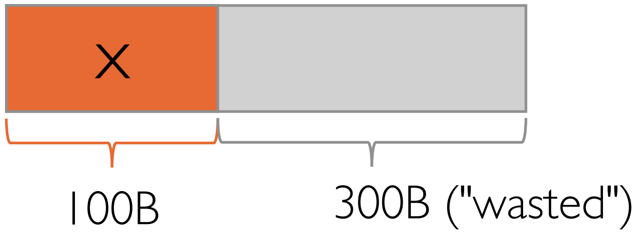
What if afterwards process Y requires 350B?

Memory Allocation Policies: First-Fit

Suppose process X needs 100B of memory to be loaded, and the list of holes is as follows:



Which segment will be picked by the OS using first-fit?



What if afterwards process Y requires 350B?

We will not be able to satisfy this request even if theoretically we could

Memory Allocation Policies: Best-Fit

- Allocate the smallest hole that is big enough to satisfy the request

Memory Allocation Policies: Best-Fit

- Allocate the smallest hole that is big enough to satisfy the request
- This saves large holes for other process requests that may need them

Memory Allocation Policies: Best-Fit

- Allocate the smallest hole that is big enough to satisfy the request
- This saves large holes for other process requests that may need them
- However, the resulting unused portions of holes may be too small to be of any use, and will therefore be wasted

Memory Allocation Policies: Best-Fit

- Allocate the smallest hole that is big enough to satisfy the request
- This saves large holes for other process requests that may need them
- However, the resulting unused portions of holes may be too small to be of any use, and will therefore be wasted
- Complexity: still O(n) but can be O(log n) if the list of holes is kept sorted

Memory Allocation Policies: Best-Fit

- Allocate the smallest hole that is big enough to satisfy the request
- This saves large holes for other process requests that may need them
- However, the resulting unused portions of holes may be too small to be of any use, and will therefore be wasted
- Complexity: still O(n) but can be O(log n) if the list of holes is kept sorted

Do you know how which data structure can be used to achieve this?

Memory Allocation Policies: Best-Fit

- Allocate the smallest hole that is big enough to satisfy the request
- This saves large holes for other process requests that may need them
- However, the resulting unused portions of holes may be too small to be of any use, and will therefore be wasted
- Complexity: still O(n) but can be O(log n) if the list of holes is kept sorted

Do you know how which data structure can be used to achieve this?

Binary Search Tree (BST)

Memory Allocation Policies: Best-Fit

Suppose process X needs 100B of memory to be loaded, and the list of holes is as follows:

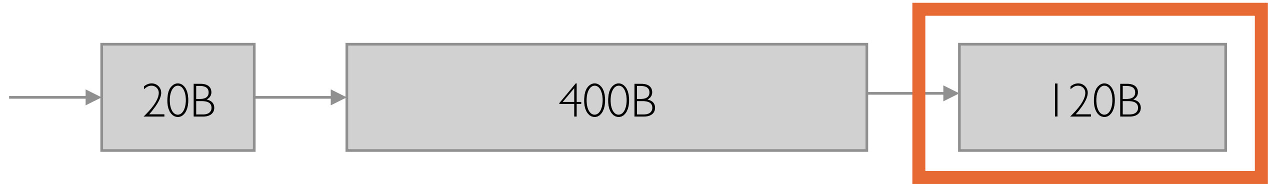
Memory Allocation Policies: Best-Fit

Suppose process X needs 100B of memory to be loaded, and the list of holes is as follows:

Which segment will be picked by the OS using best-fit?

Memory Allocation Policies: Best-Fit

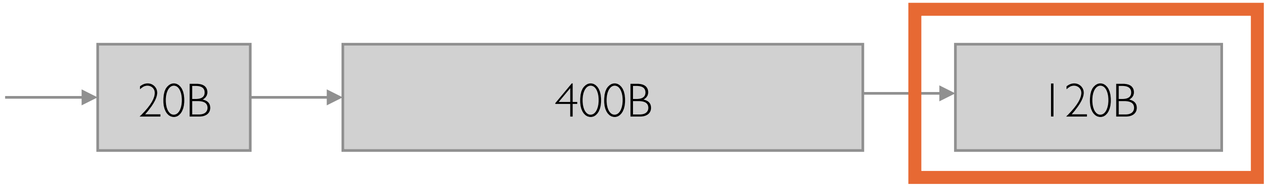
Suppose process X needs 100B of memory to be loaded, and the list of holes is as follows:



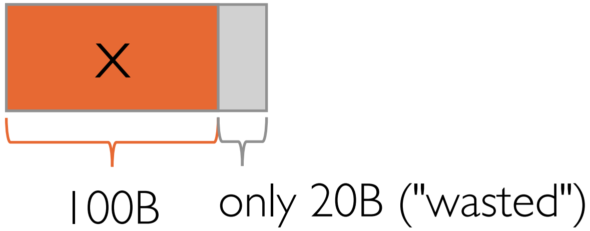
Which segment will be picked by the OS using best-fit?

Memory Allocation Policies: Best-Fit

Suppose process X needs 100B of memory to be loaded, and the list of holes is as follows:

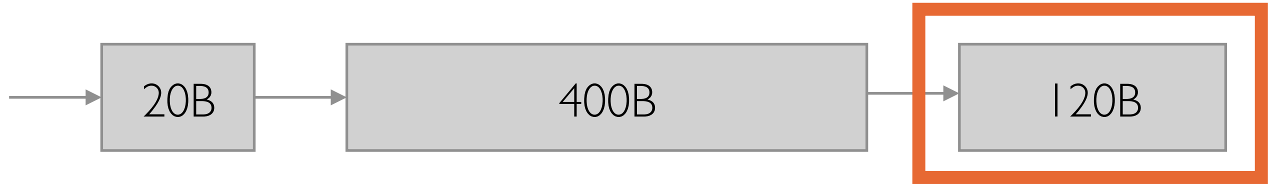


Which segment will be picked by the OS using best-fit?

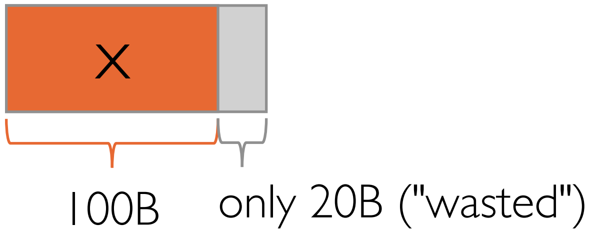


Memory Allocation Policies: Best-Fit

Suppose process X needs 100B of memory to be loaded, and the list of holes is as follows:



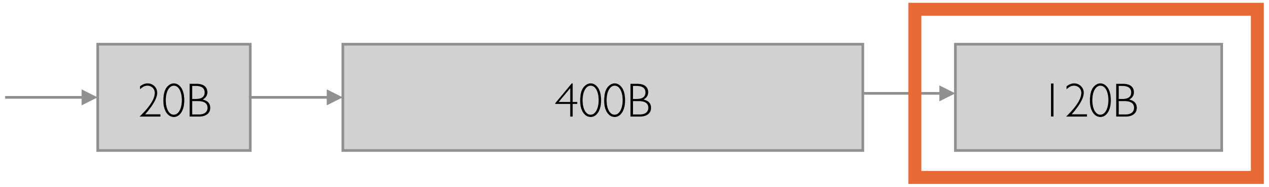
Which segment will be picked by the OS using best-fit?



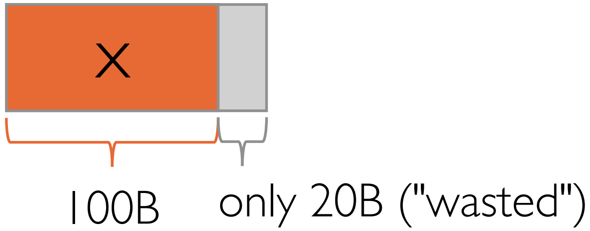
What if afterwards process Y requires 350B?

Memory Allocation Policies: Best-Fit

Suppose process X needs 100B of memory to be loaded, and the list of holes is as follows:



Which segment will be picked by the OS using best-fit?



What if afterwards process Y requires 350B?

We can now assign it the second available hole segment (400B)

Memory Allocation Policies: Worst-Fit

- Allocate the largest hole available

Memory Allocation Policies: Worst-Fit

- Allocate the largest hole available
- Might sound counterintuitive but this increases the likelihood that the remaining portion will be usable for satisfying future requests

Memory Allocation Policies: Worst-Fit

- Allocate the largest hole available
- Might sound counterintuitive but this increases the likelihood that the remaining portion will be usable for satisfying future requests
- Simulations show that First-Fit and Best-Fit usually work best

Memory Allocation Policies: Worst-Fit

- Allocate the largest hole available
- Might sound counterintuitive but this increases the likelihood that the remaining portion will be usable for satisfying future requests
- Simulations show that First-Fit and Best-Fit usually work best
- First-Fit is also generally faster than Best-Fit

Fragmentation

Problem

Individual holes may be too small to serve a process request but they can be large enough if combined together

Fragmentation

<u>Problem</u>

Individual holes may be too small to serve a process request but they can be large enough if combined together

External Fragmentation

Fragmentation

Problem

Individual holes may be too small to serve a process request but they can be large enough if combined together

External Fragmentation

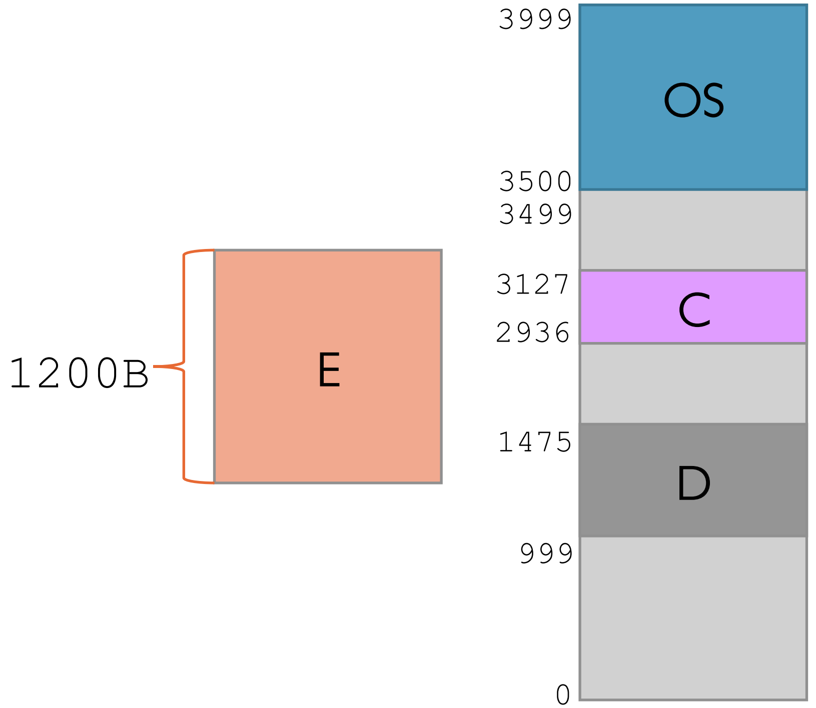
Internal Fragmentation

External Fragmentation

- Frequent loading and unloading processes causes holes to be broken into small (i.e., unusable) chunks

External Fragmentation

- Frequent loading and unloading processes causes holes to be broken into small (i.e., unusable) chunks
- It happens when there is enough memory to load a process in memory but space is not contiguous

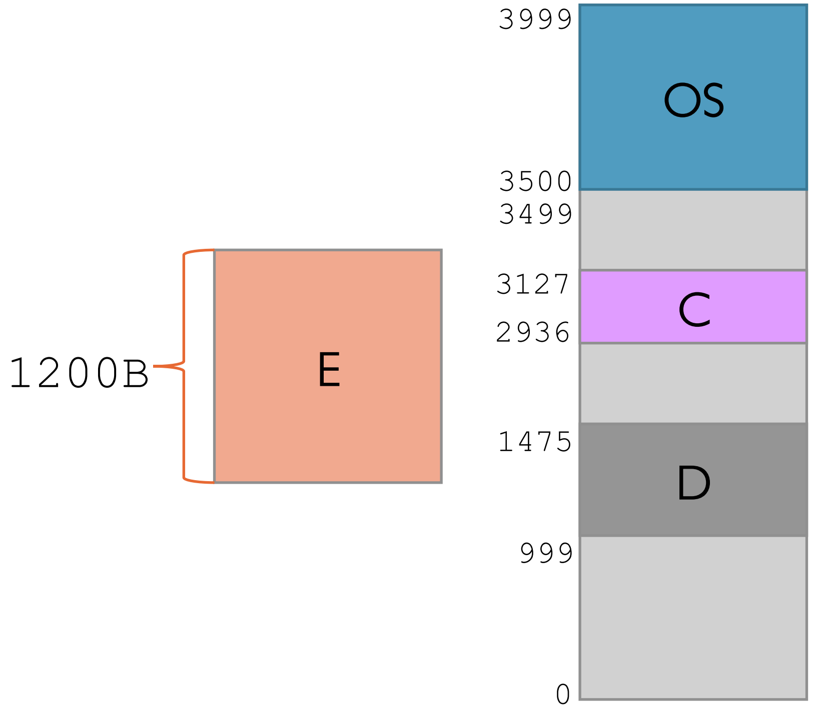


External Fragmentation

- Frequent loading and unloading processes causes holes to be broken into small (i.e., unusable) chunks
- It happens when there is enough memory to load a process in memory but space is not contiguous

Simulations show that for every 2N allocated blocks, N are lost due to external fragmentation

1/3 of memory space is wasted on average



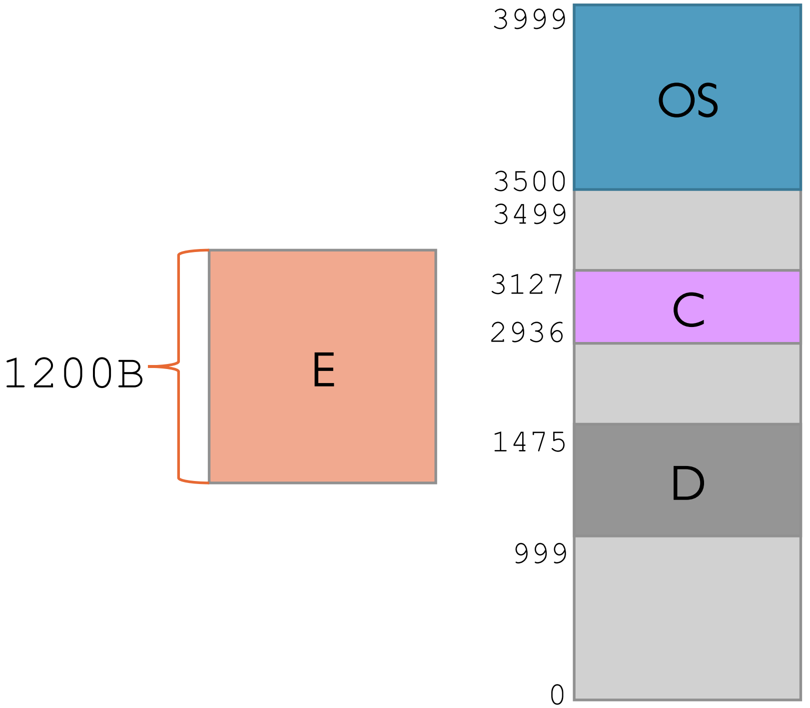
External Fragmentation

- Frequent loading and unloading processes causes holes to be broken into small (i.e., unusable) chunks
- It happens when there is enough memory to load a process in memory but space is not contiguous

Simulations show that for every 2N allocated blocks, N are lost due to external fragmentation

1/3 of memory space is wasted on average

<u>Goal:</u>
Allocation policy that minimizes wasted space!



Internal Fragmentation

- It happens when memory internal to a segment is wasted

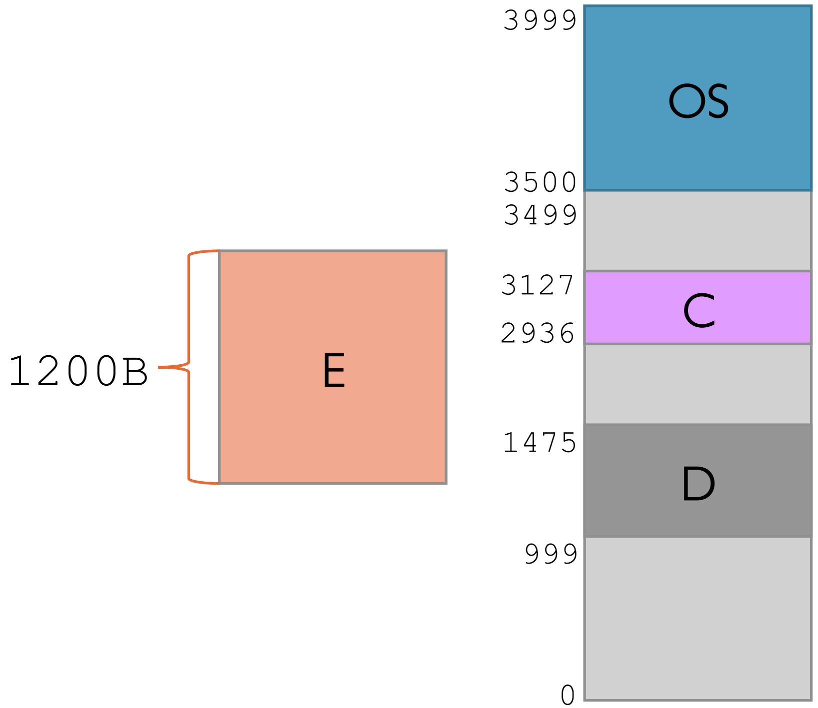
Internal Fragmentation

- It happens when memory internal to a segment is wasted
- For example, consider a process whose size is 8,846B and a hole of size 8,848B

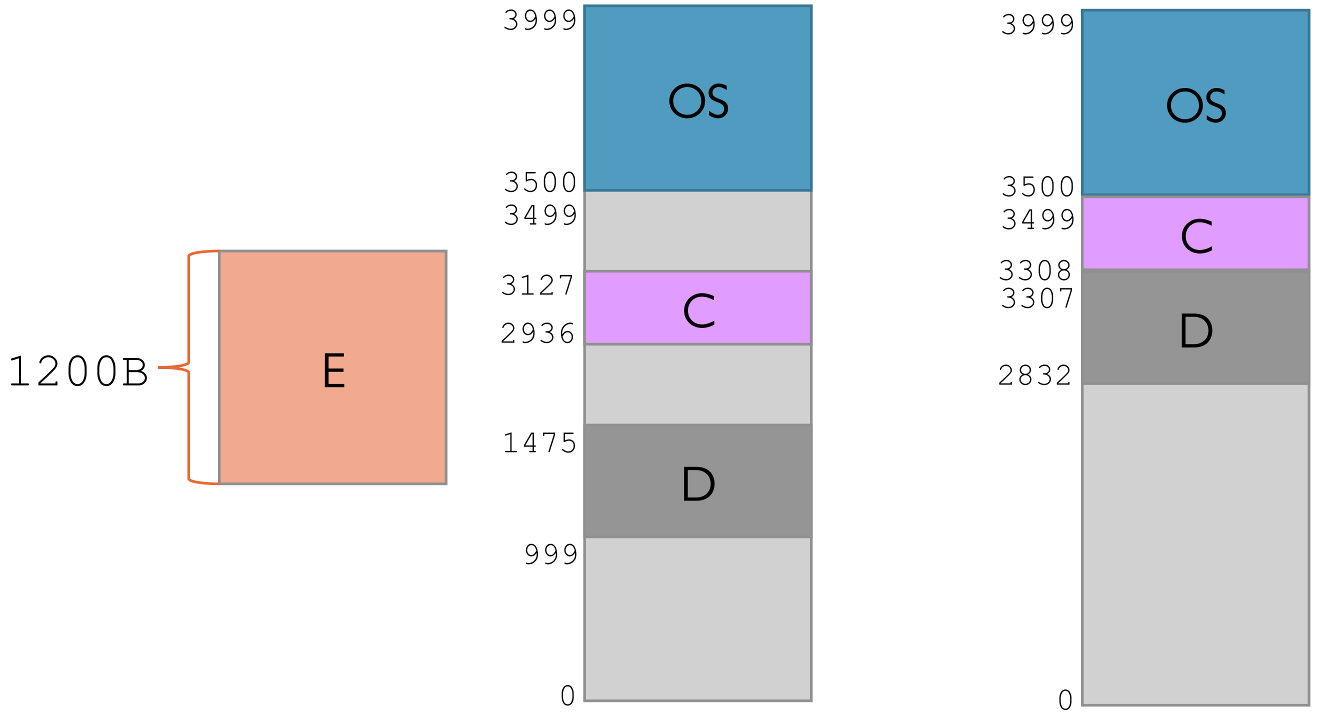
Internal Fragmentation

- It happens when memory internal to a segment is wasted
- For example, consider a process whose size is 8,846B and a hole of size 8,848B
- It may be much more efficient to allocate the process the whole block (and waste 2B) rather than keep track of a tiny 2B hole

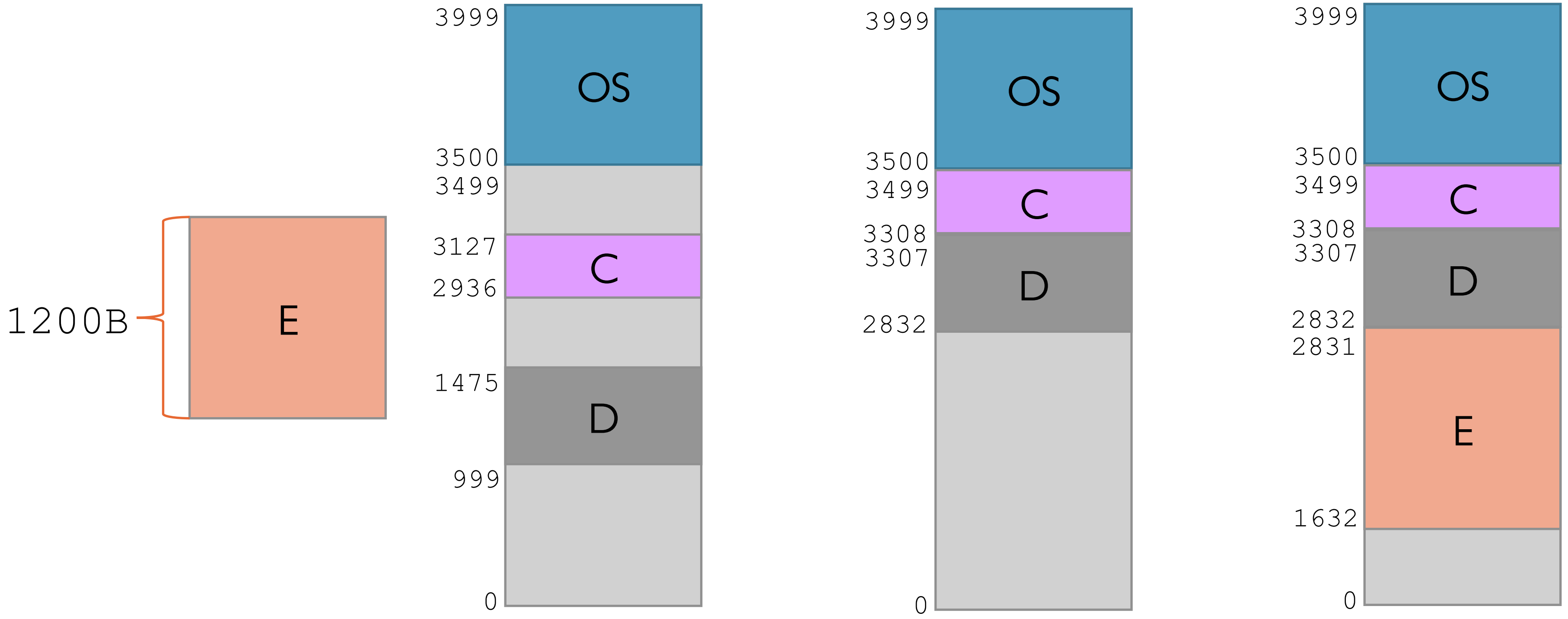
Solution to Fragmentation: Full Compaction



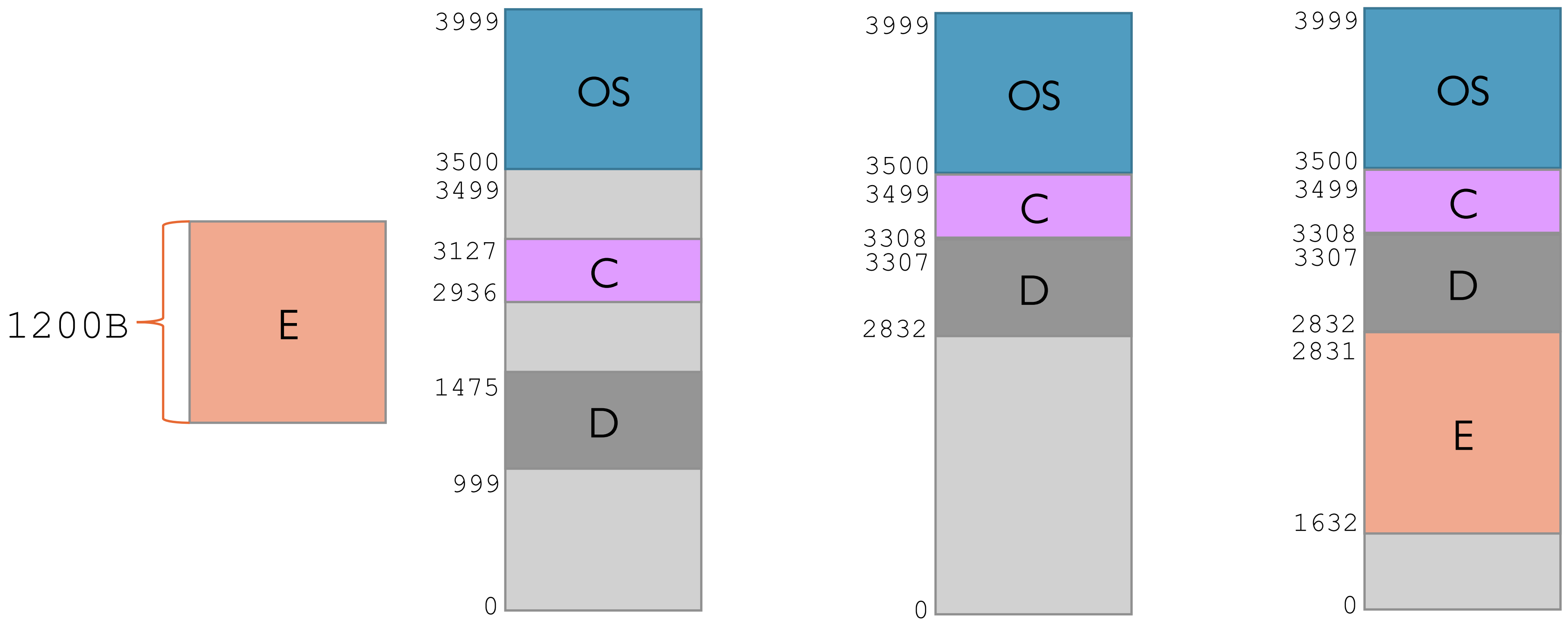
Solution to Fragmentation: Full Compaction



Solution to Fragmentation: Full Compaction

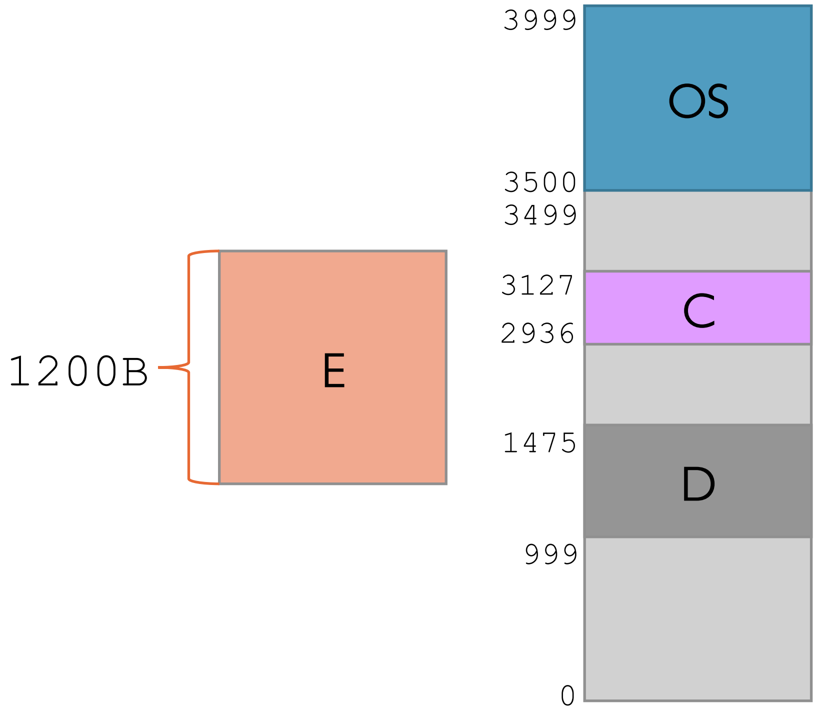


Solution to Fragmentation: Full Compaction

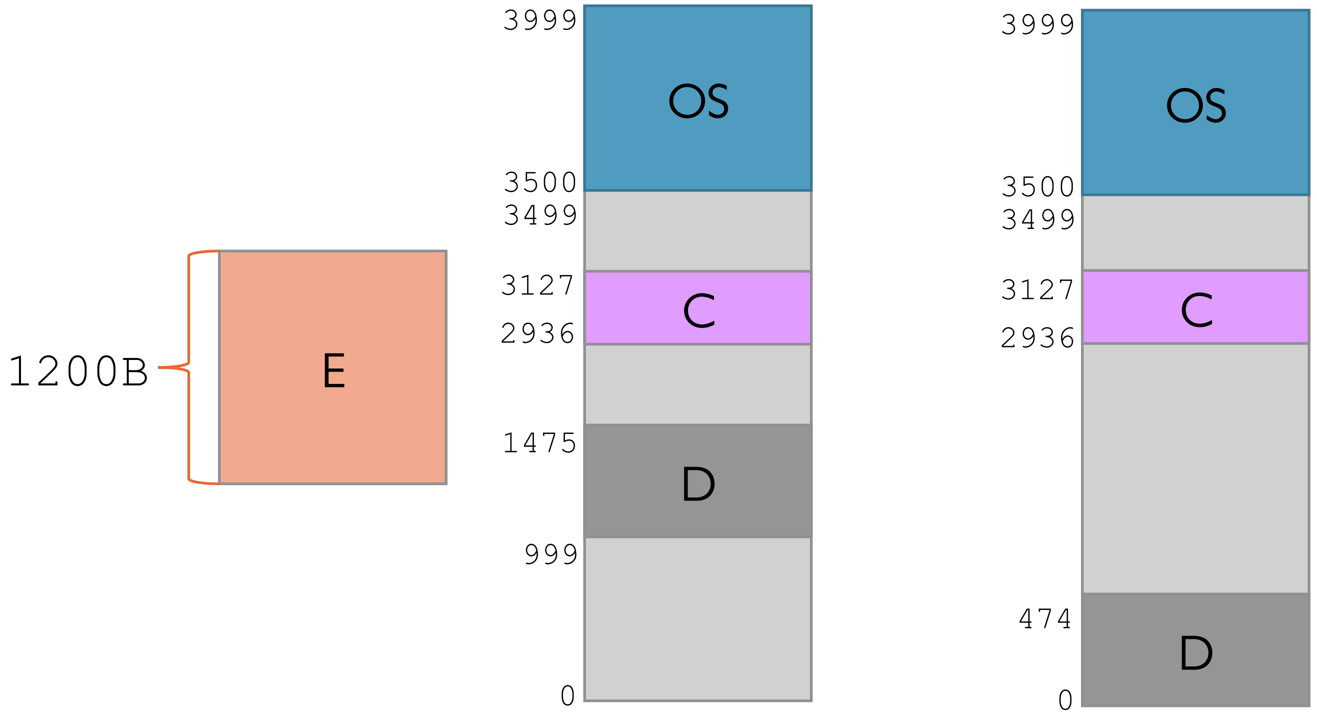


Only one hole is left but two processes need to be moved (C and D)

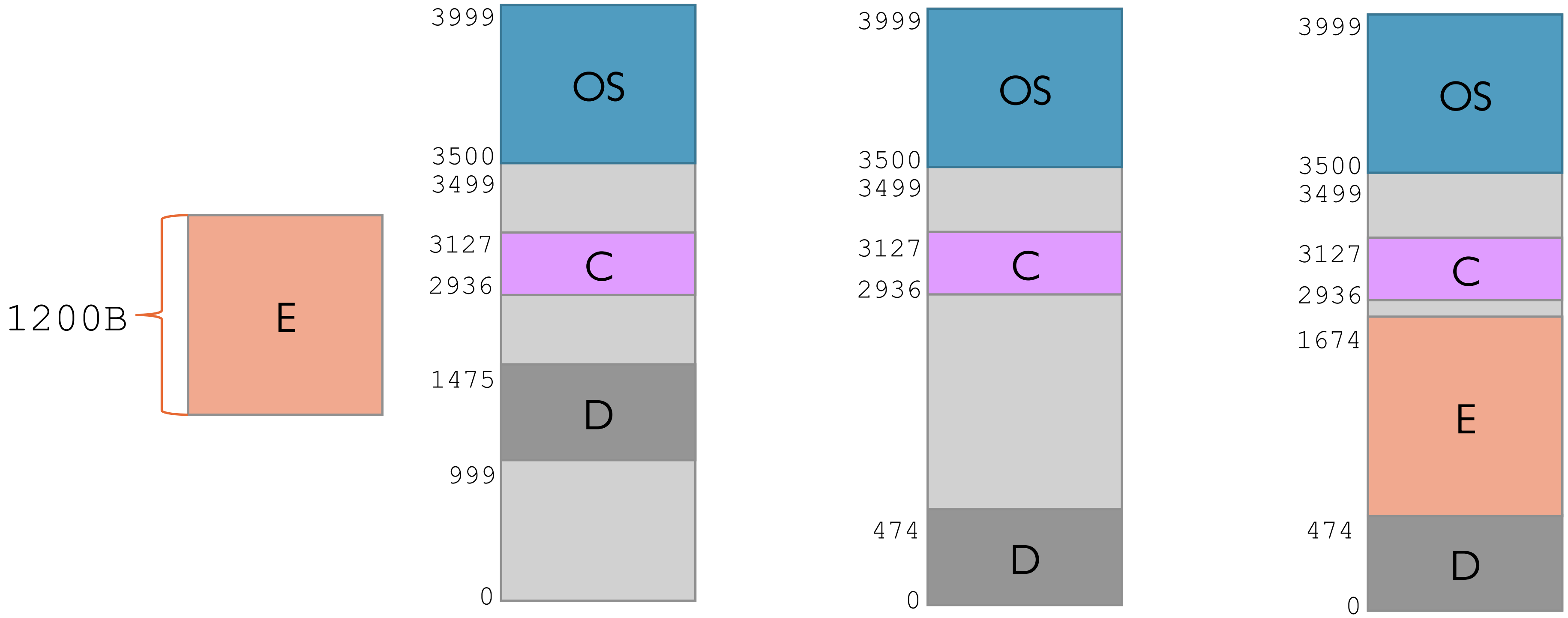
Solution to Fragmentation: Partial Compaction



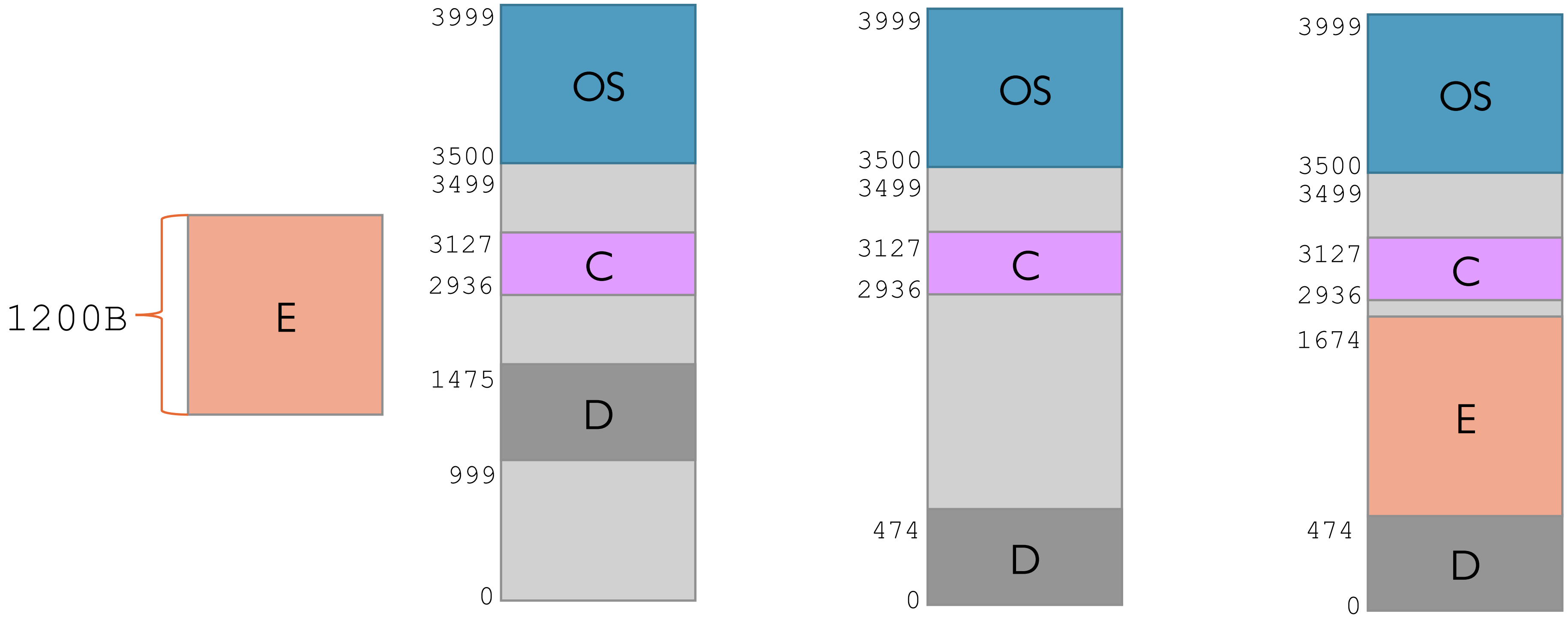
Solution to Fragmentation: Partial Compaction



Solution to Fragmentation: Partial Compaction



Solution to Fragmentation: Partial Compaction



Still some holes left but only one process is moved (D) rather than two

Swapping

- So far, we have assumed all processes are entirely loaded in memory (of course, when loaded!)

Swapping

- So far, we have assumed all processes are entirely loaded in memory (of course, when loaded!)
- Remember: A process needs to sit physically in main memory only if the CPU executes its instructions and accesses its data

Swapping

- So far, we have assumed all processes are entirely loaded in memory (of course, when loaded!)
- Remember: A process needs to sit physically in main memory only if the CPU executes its instructions and accesses its data
- If a process blocks (e.g., due to an I/O call) it doesn't need to be in memory while I/O is running

Swapping

- So far, we have assumed all processes are entirely loaded in memory (of course, when loaded!)
- Remember: A process needs to sit physically in main memory only if the CPU executes its instructions and accesses its data
- If a process blocks (e.g., due to an I/O call) it doesn't need to be in memory while I/O is running
- That process can be "swapped out" from memory to disk to make room for other processes

Swapping

- Once process becomes ready again, the OS must reload it in memory

Swapping

- Once process becomes ready again, the OS must reload it in memory
- Swap in depends on the address binding used:
  - compile- or load-time: must be swapped back into the same memory location from which they were swapped out

Swapping

- Once process becomes ready again, the OS must reload it in memory
- Swap in depends on the address binding used:
  - compile- or load-time: must be swapped back into the same memory location from which they were swapped out
  - execution-time: can be swapped back into any available location (updating base and limit registers)

Swapping

- Once process becomes ready again, the OS must reload it in memory
- Swap in depends on the address binding used:
  - compile- or load-time: must be swapped back into the same memory location from which they were swapped out
  - execution-time: can be swapped back into any available location (updating base and limit registers)
- Using swapping, fragmentation can be tackled easily
  - Just run compaction before swapping-in a process

Swapping: Example

- Swapping is a very slow process compared to other operations due to the interaction with hard disk (more on this later)

Swapping: Example

- Swapping is a very slow process compared to other operations due to the interaction with hard disk (more on this later)
- Example:
  - 10 MB user process
  - disk transfer rate = 40 MB/sec (250 msec just to do the data transfer)

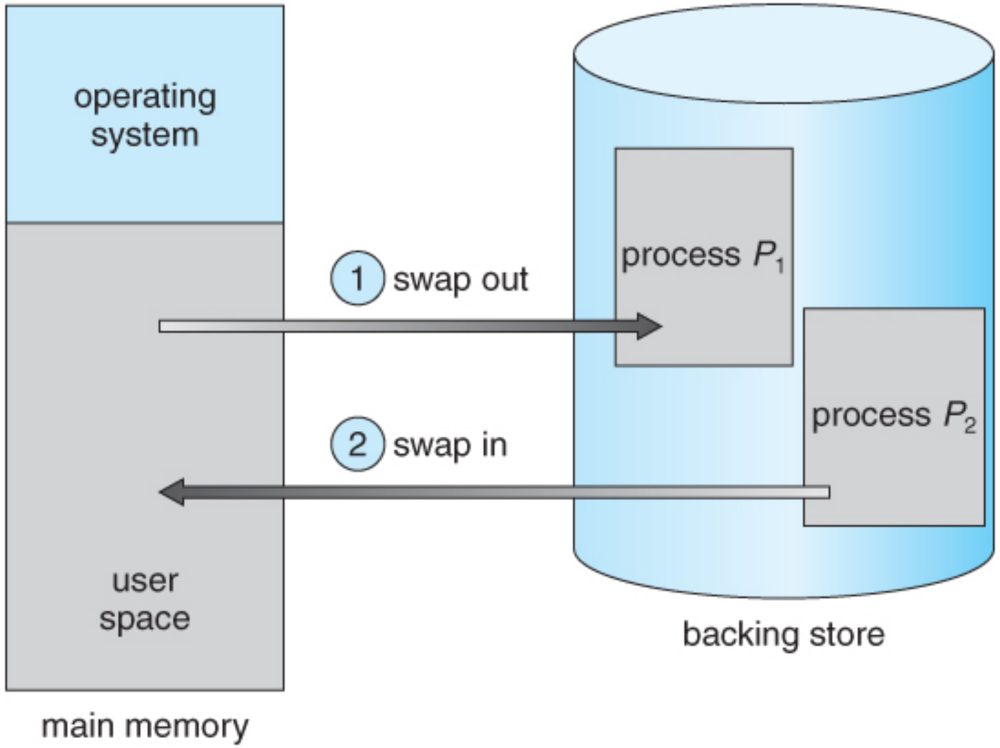
Swapping: Example

- Swapping is a very slow process compared to other operations due to the interaction with hard disk (more on this later)
- Example:
  - 10 MB user process
  - disk transfer rate = 40 MB/sec (250 msec just to do the data transfer)
- Since swap-in may involve swapping-out another process, the overall time required will be ~500 msec

Swapping: Example

- Swapping is a very slow process compared to other operations due to the interaction with hard disk (more on this later)
- Example:
  - 10 MB user process
  - disk transfer rate = 40 MB/sec (250 msec just to do the data transfer)
- Since swap-in may involve swapping-out another process, the overall time required will be ~500 msec
- Time slice is usually way smaller than that!

Swapping



Most modern OSs no longer use swapping, because it is too slow and there are faster alternatives available (e.g., paging)

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

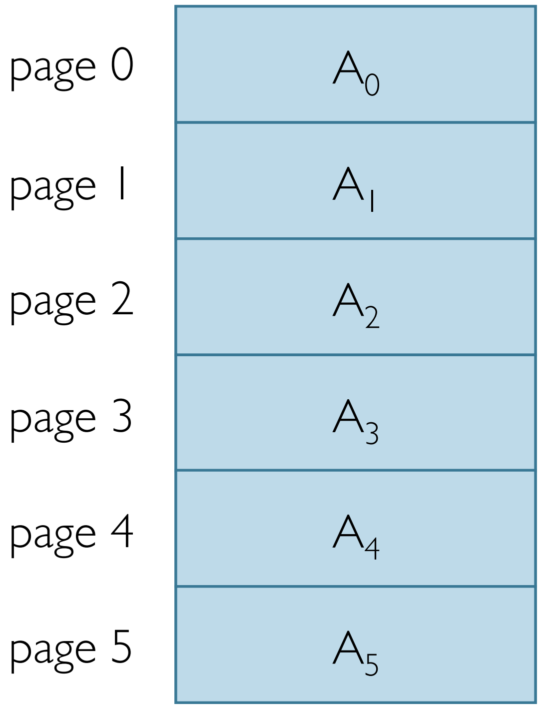
Paging

- 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

90/10 Rule

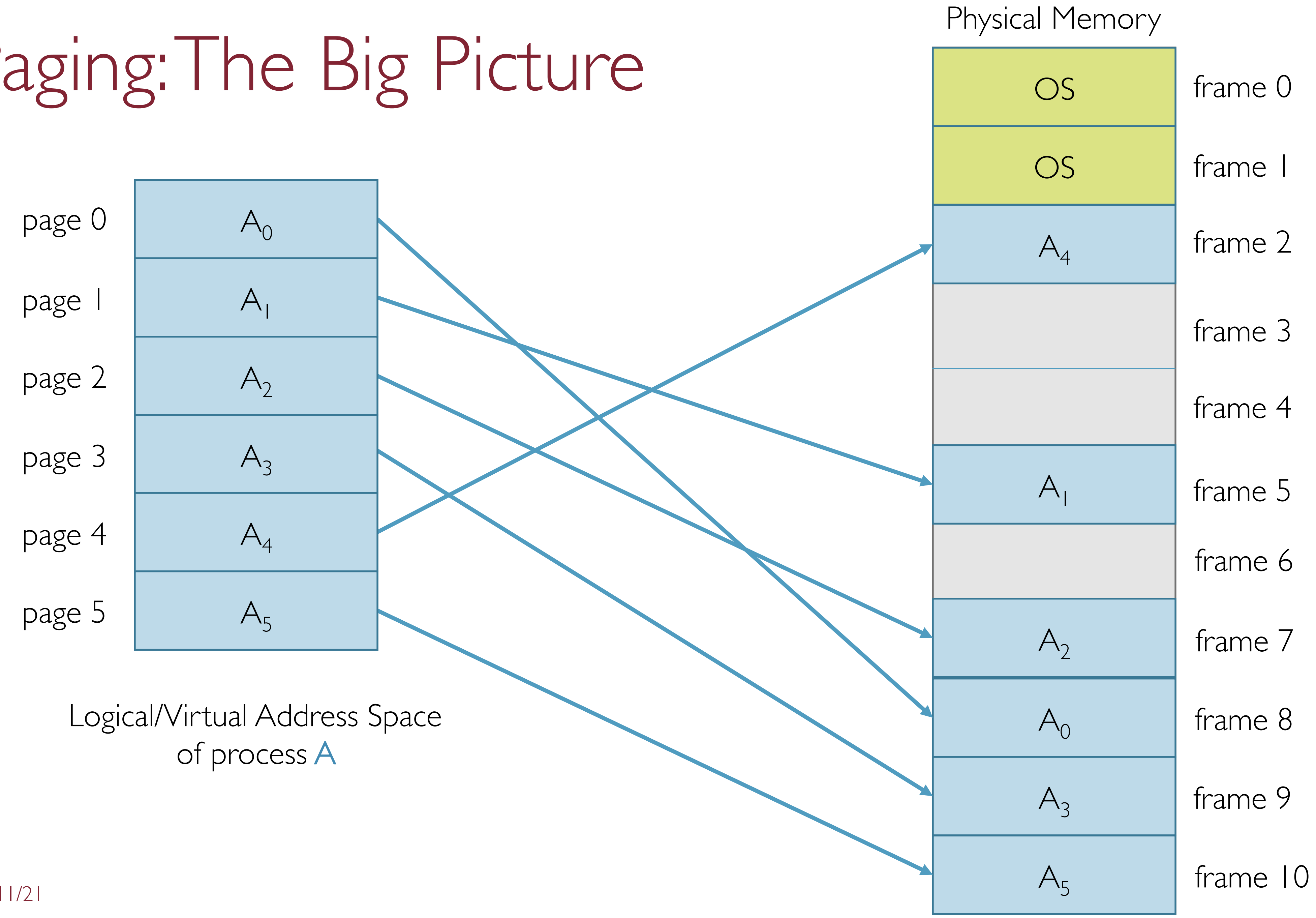
Processes spend 90% of their time accessing only 10% of their allocated memory space

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

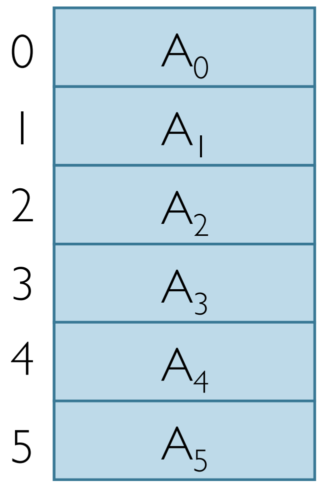
Basic OS Responsibilities for Paging

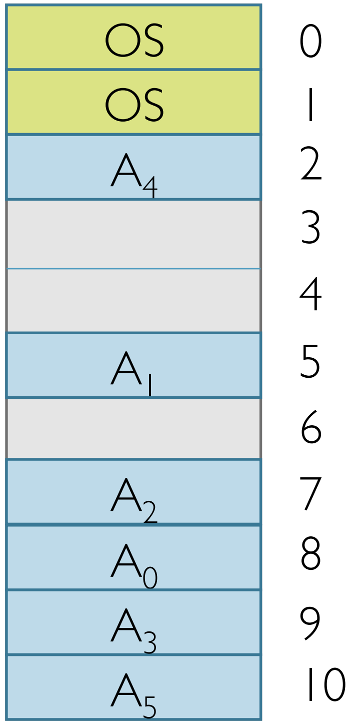
- The OS has 2 main responsibilities:
  - mapping between logical pages and physical frames
  - translating logical addresses to physical addresses
- All of this must be done efficiently!
  - Remember, memory addresses are referenced all the time

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

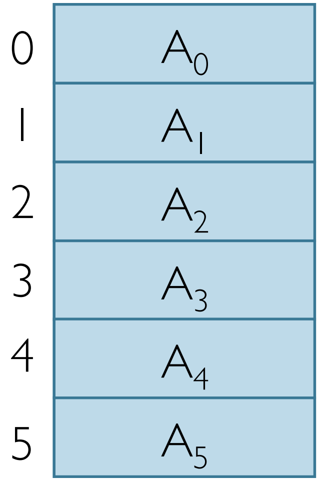
Page Table: Mapping Pages to Frames



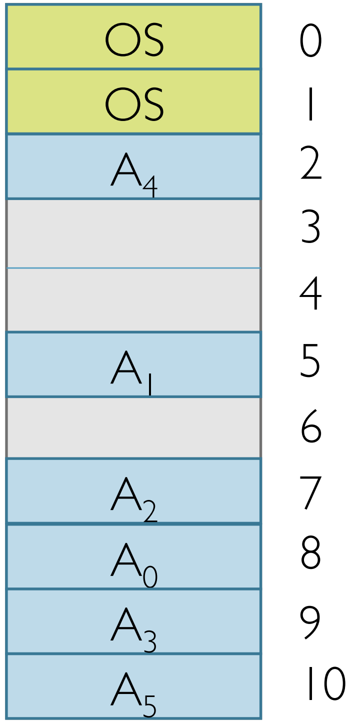


Page Table: Mapping Pages to Frames

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

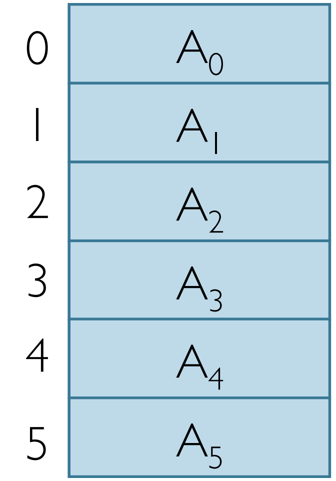


Page	Frame
0	8
1	5
2	7
3	9
4	2
5	10

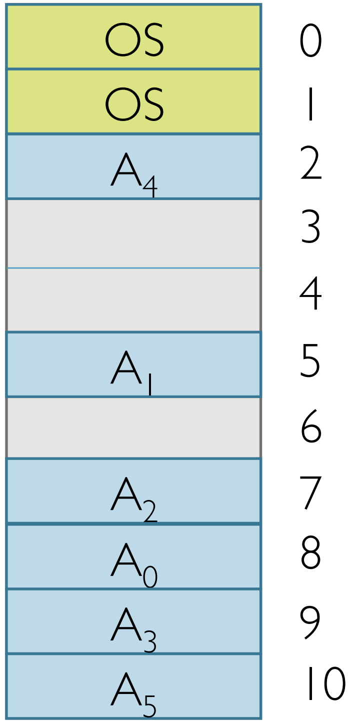


Page Table: Mapping Pages to Frames

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

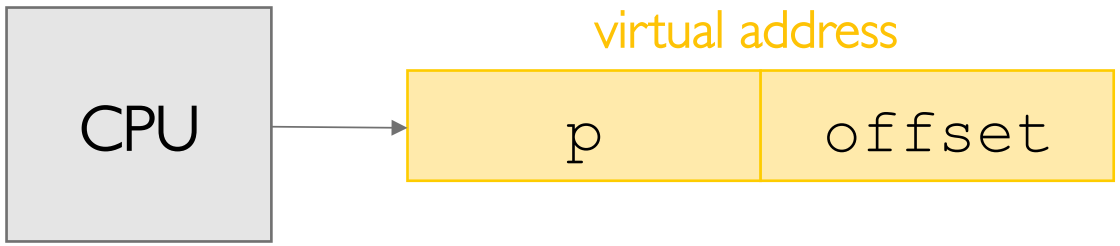
- Processes use virtual (logical) addresses to refer to memory (not page number!)
- Virtual (logical) address space is still contiguous starting from 0

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

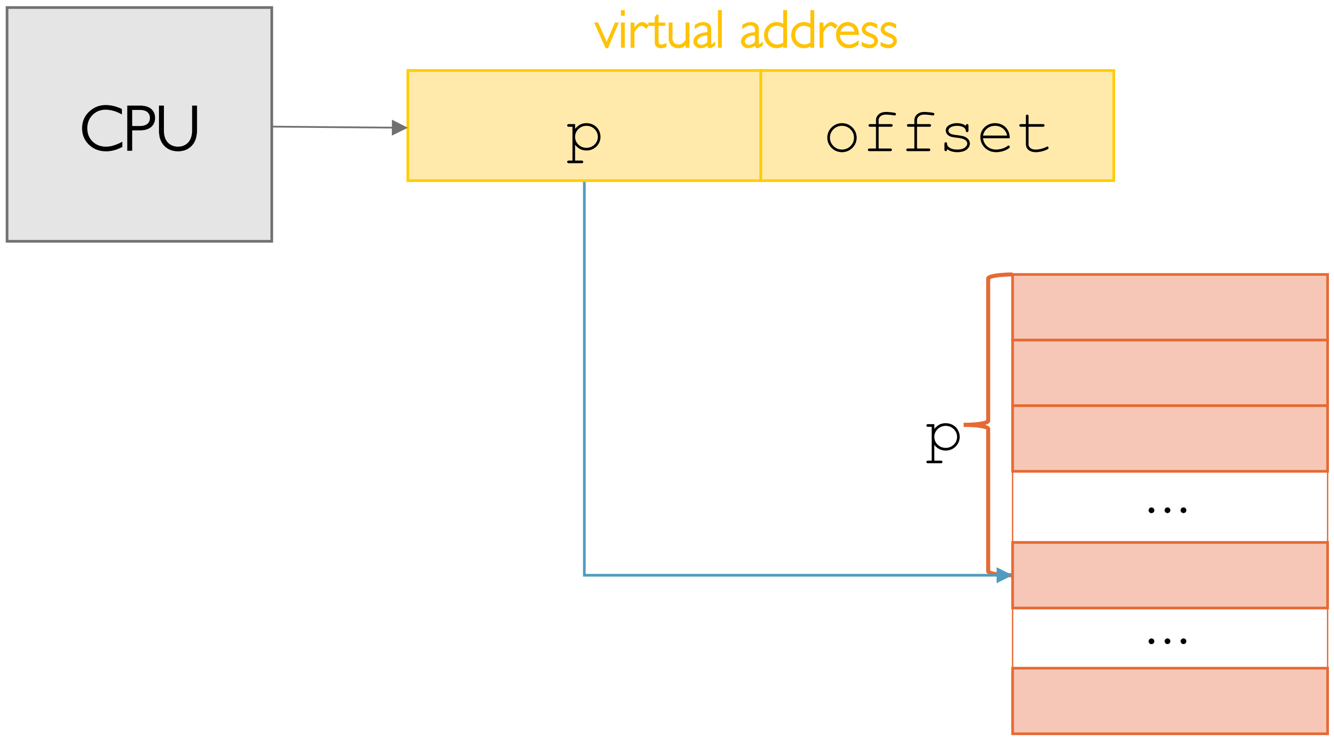
Page Table: Virtual to Physical Address

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

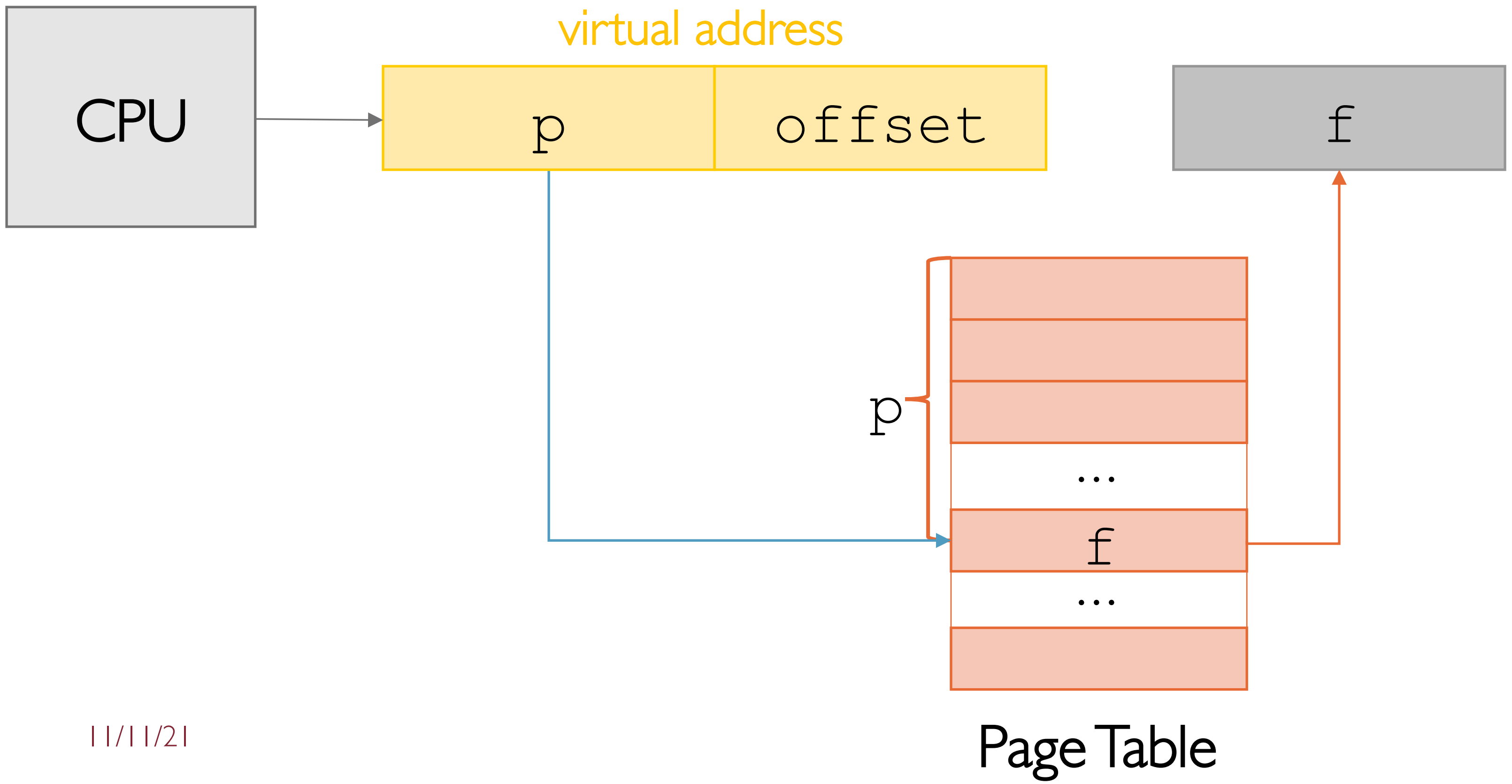
virtual address consists of 2 parts:
- `p`: page number where the address resides
- `offset`: relative from the beginning of the page



Page Table

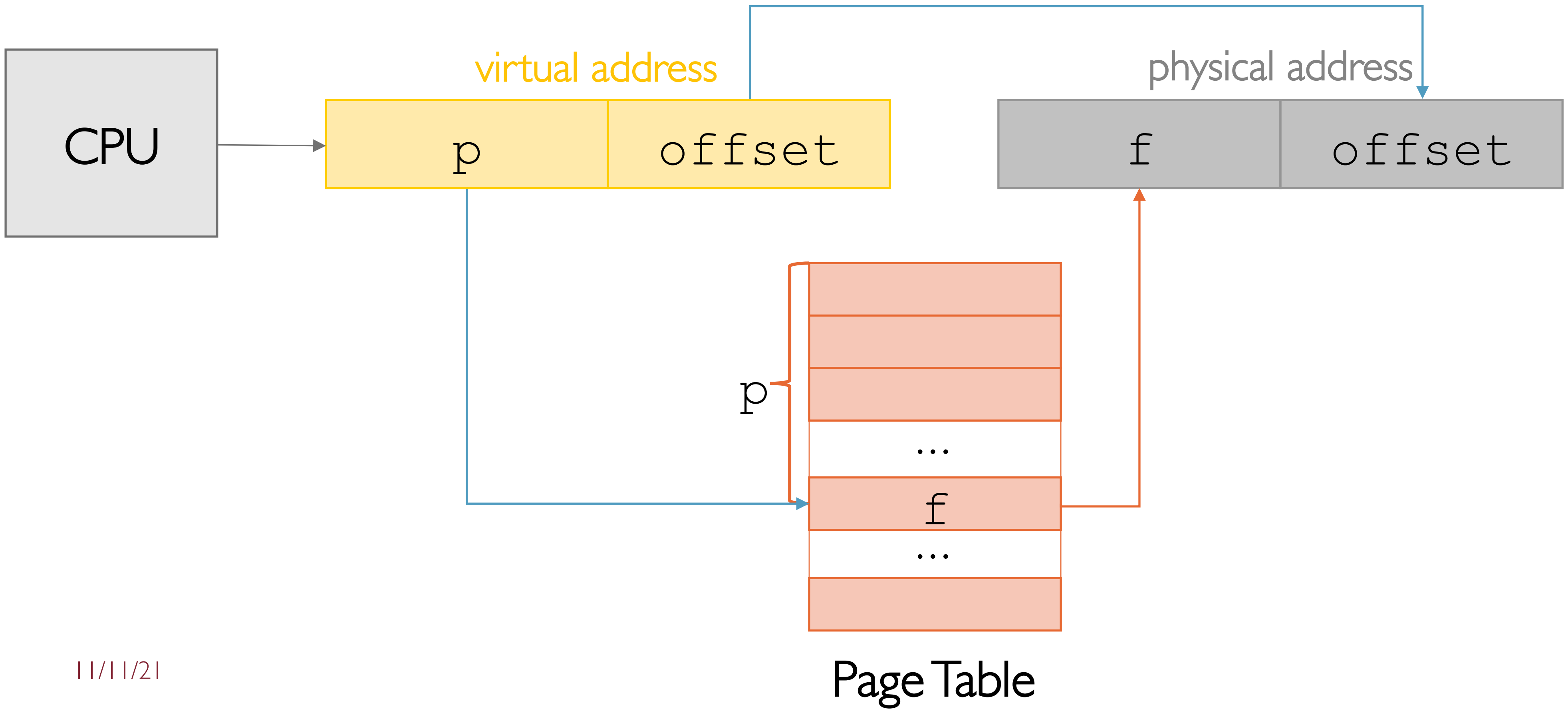
Page Table: Virtual to Physical Address

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

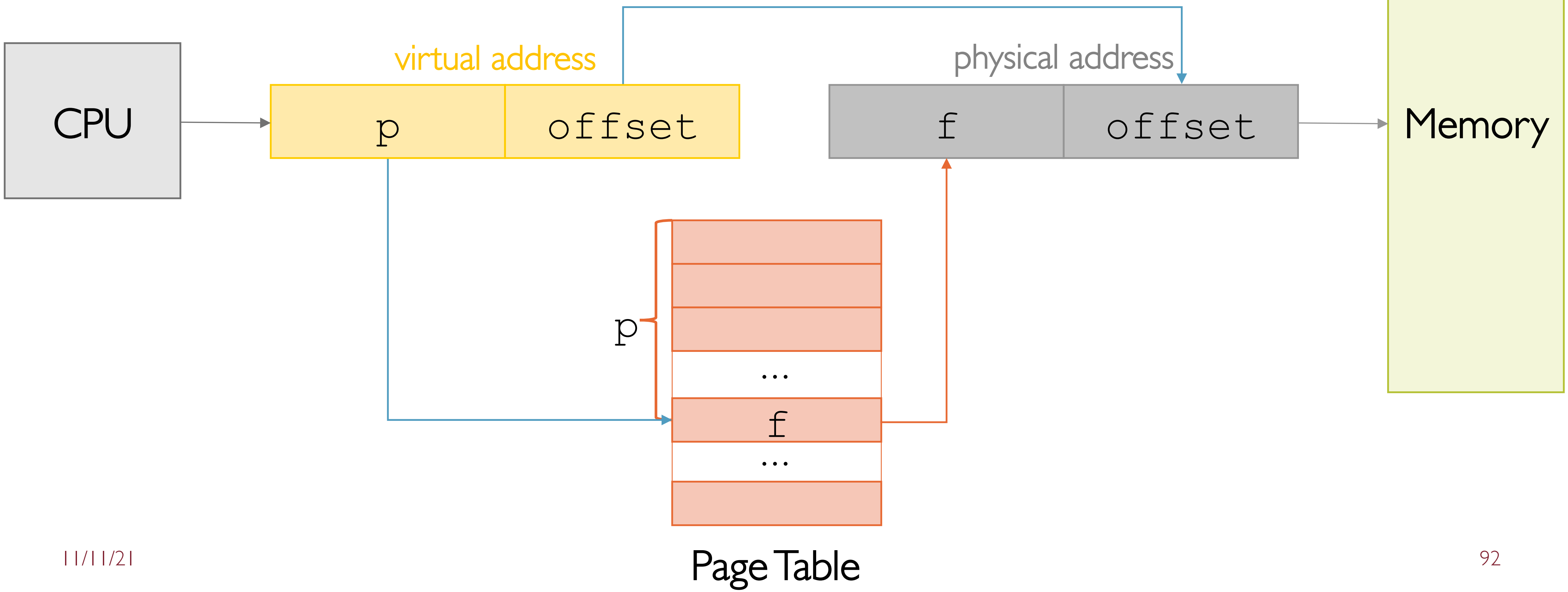
virtual address consists of 2 parts:
- p: page number where the address resides
- offset: relative from the beginning of the page



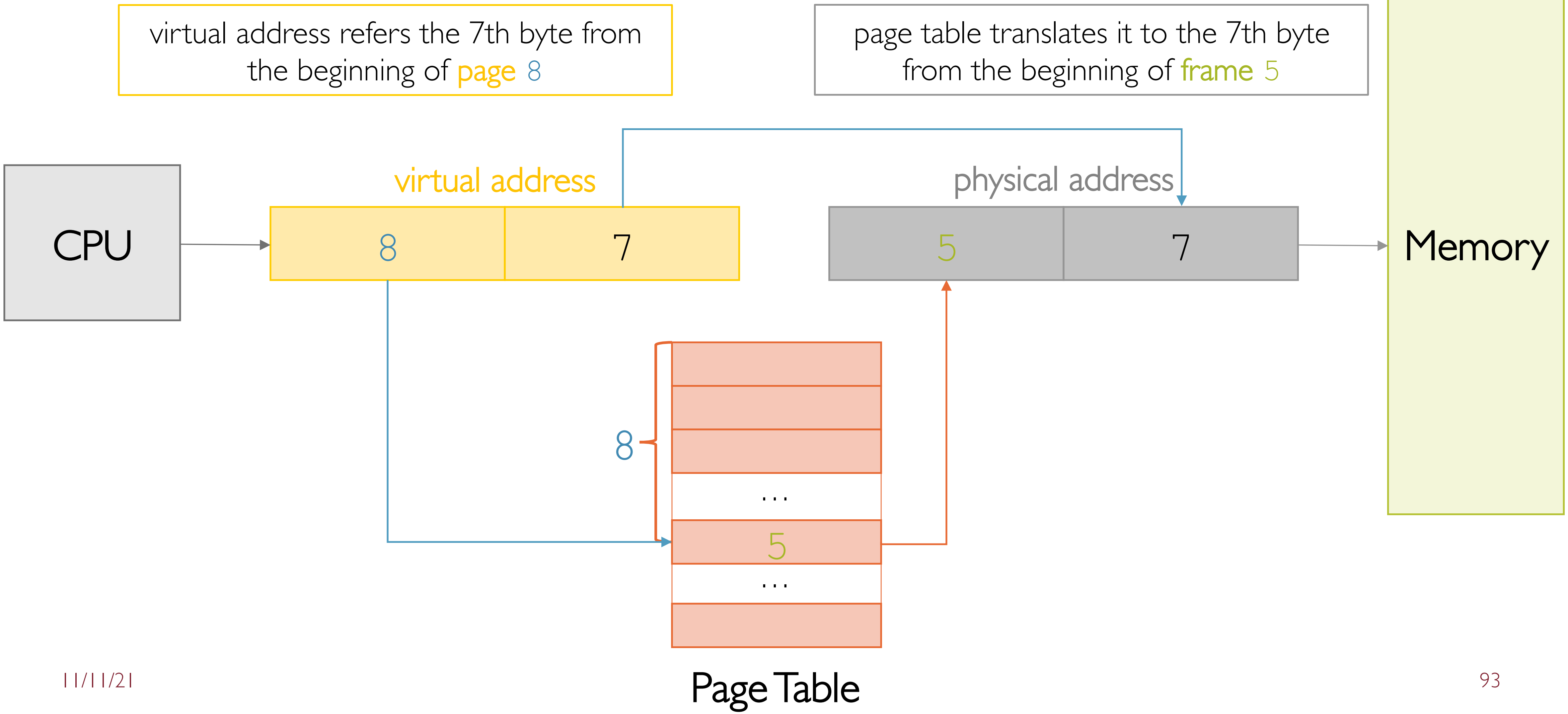
Page Table: Virtual to Physical Address

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 as Dynamic Relocation

- Paging is a form of dynamic relocation
- Each virtual address is bound by the 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: Steps

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

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

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

Paging: 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: 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: 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: 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: 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: 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`

Paging: 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`

Paging: 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: Implementation Details

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

Paging: 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: 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: 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: Why Power of 2?

- Virtual address is made of m bits

Paging: 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: 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: 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

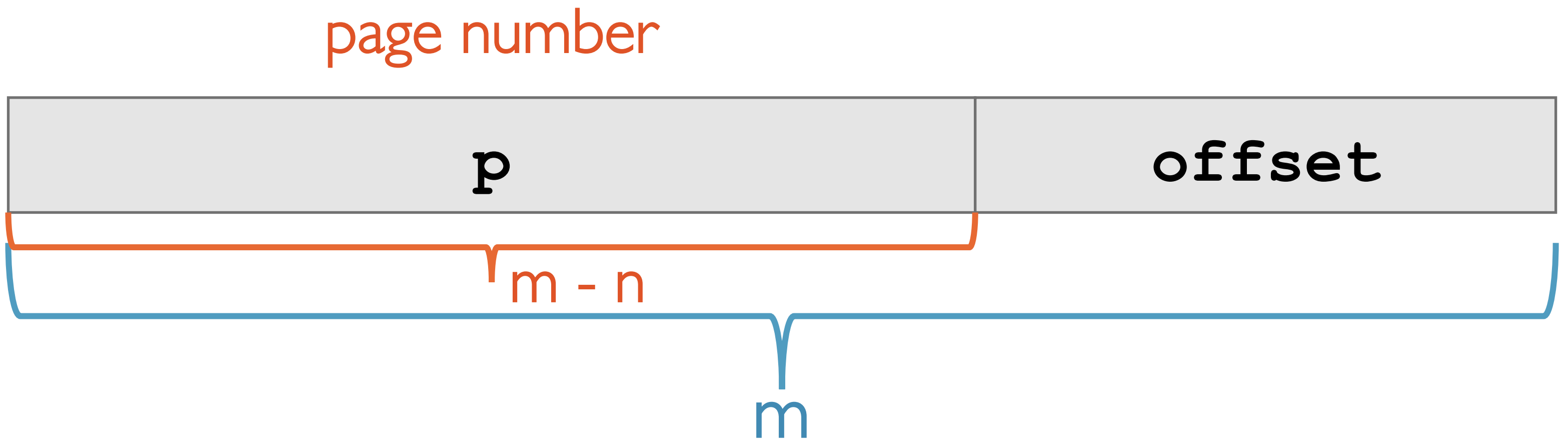
Paging: Why Power of 2?

- 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

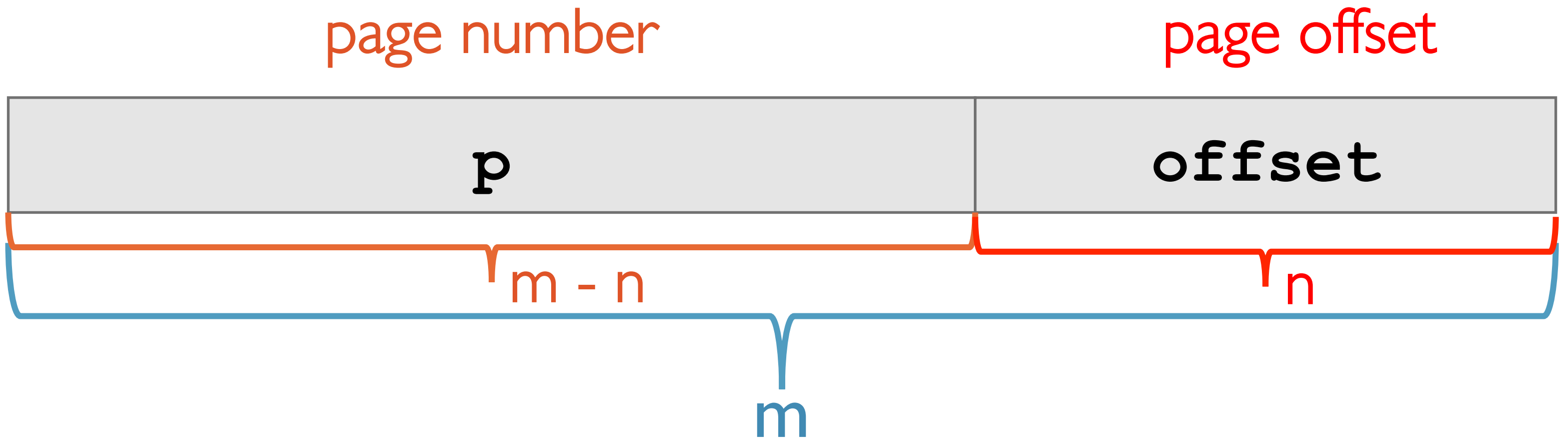
Paging: Why Power of 2?

$$\underbrace{\boxed{\texttt{p}}\boxed{\texttt{offset}}}_{m}$$

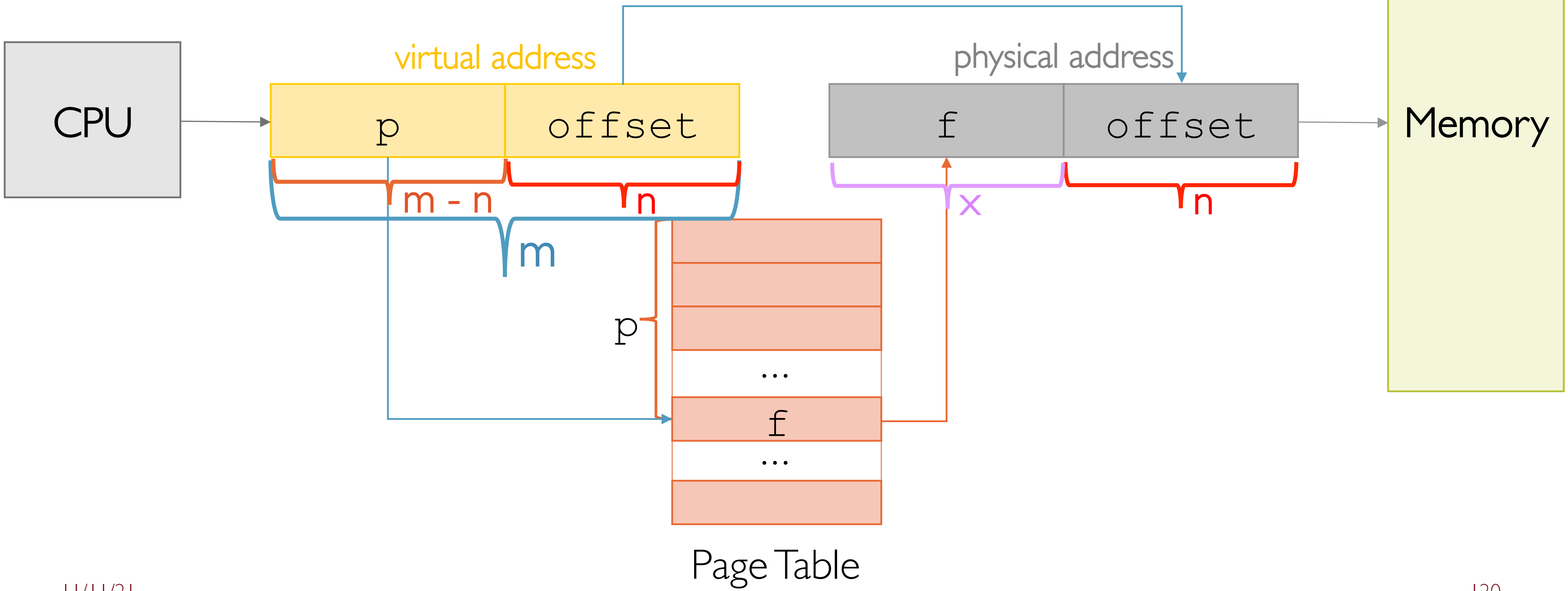
Paging: Why Power of 2?



Paging: Why Power of 2?

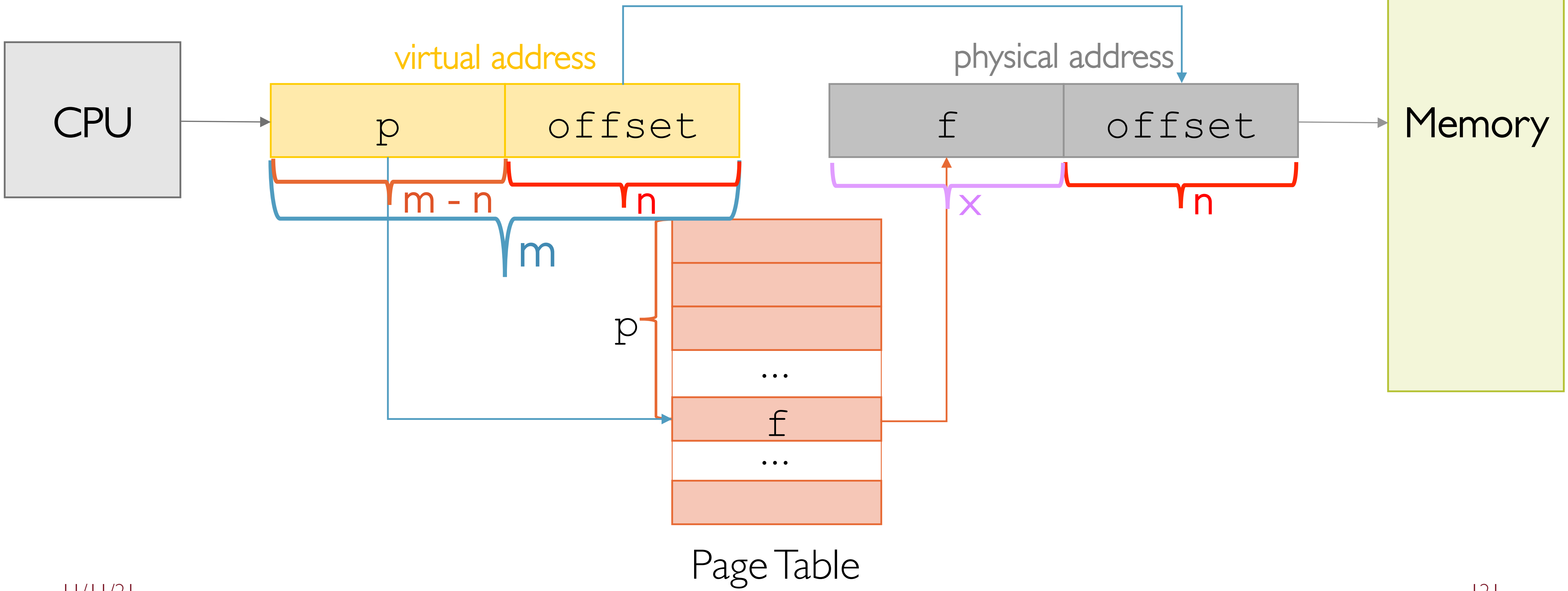


Paging: Practical Details



Paging: Practical Details

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



Paging: 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: 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: 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: 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: 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: 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: 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: Practical Example

Q3

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

Paging: 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: 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: 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: 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

$\mathbf{p} = x \;\mathbf{div}\; S = 42 \;\mathbf{div}\; 16 = \mathbf{2}$

Paging: 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

$\mathbf{p} = x \ \mathbf{div} \ S = 42 \ \mathbf{div} \ 16 = \mathbf{2}$

$\mathbf{offset} = x \ \mathbf{mod} \ S = 42 \ \mathbf{mod} \ 16 = \mathbf{10}$

10th byte from the beginning of frame 37

Paging: 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: 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: 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: 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: Practical Example 2

Q3

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

Paging: 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: 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: 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

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

Paging: 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

$\mathbf{p} = x \ \mathbf{div} \ S = 7 \ \mathbf{div} \ 4 = \mathbf{1}$

Paging: 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)

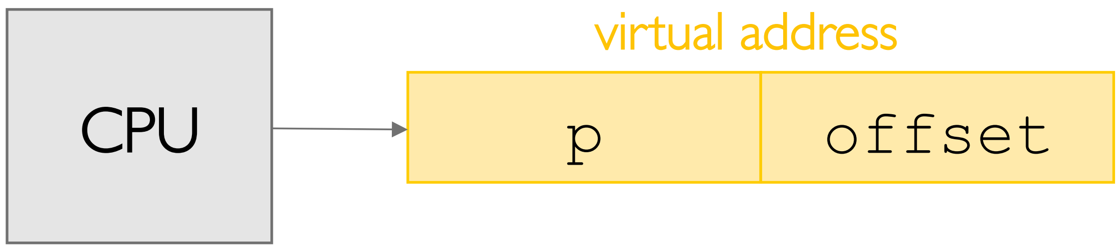
Translation Look-aside Buffer (TLB)

- 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

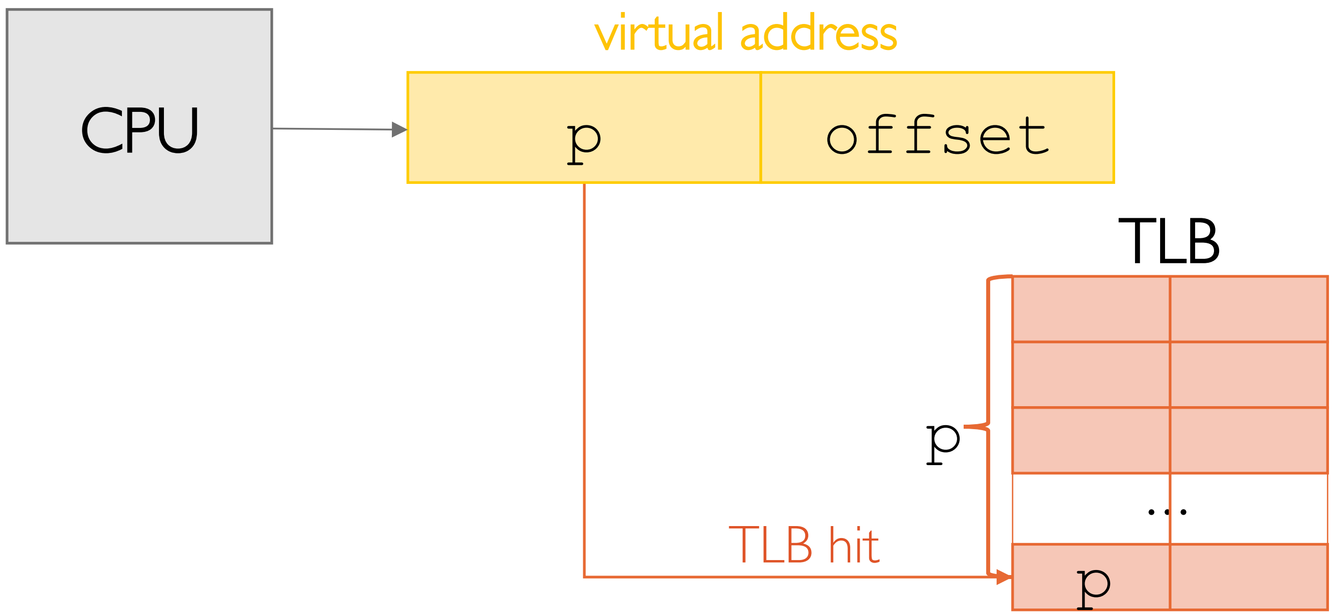
Translation Look-aside Buffer (TLB)

- 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

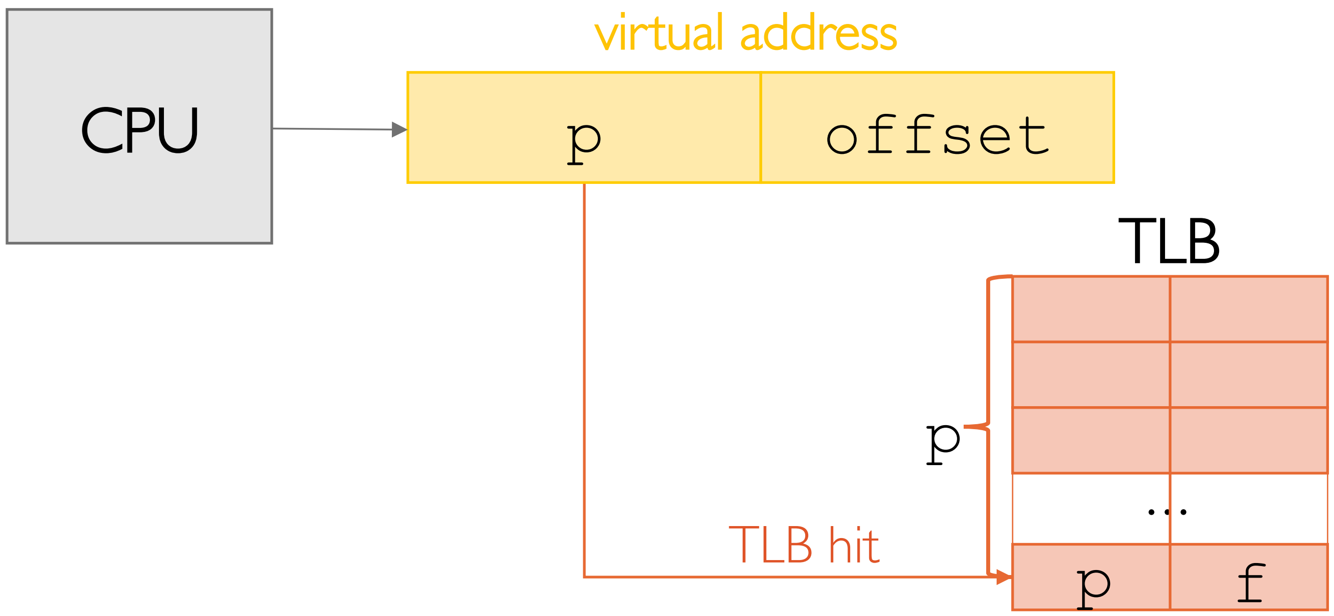
Translation Look-aside Buffer (TLB)



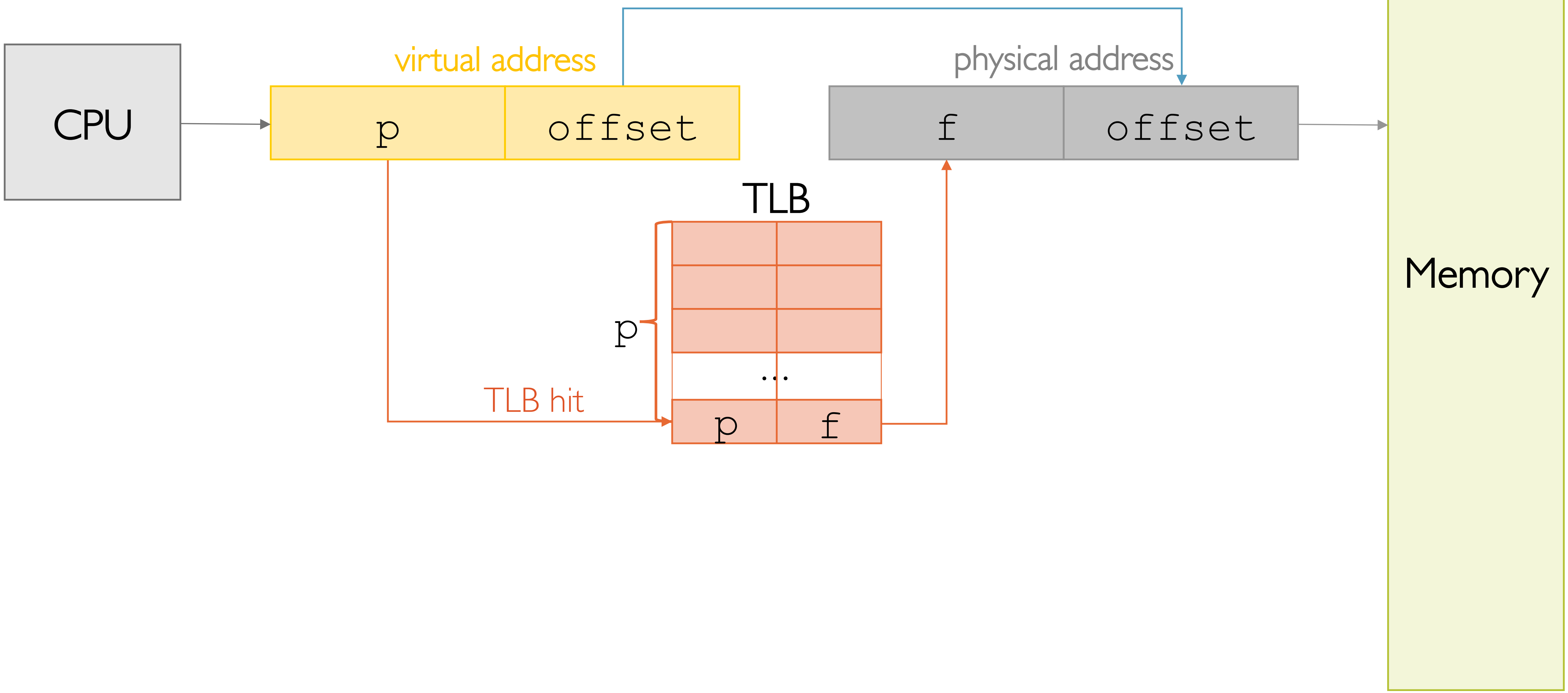
Translation Look-aside Buffer (TLB)



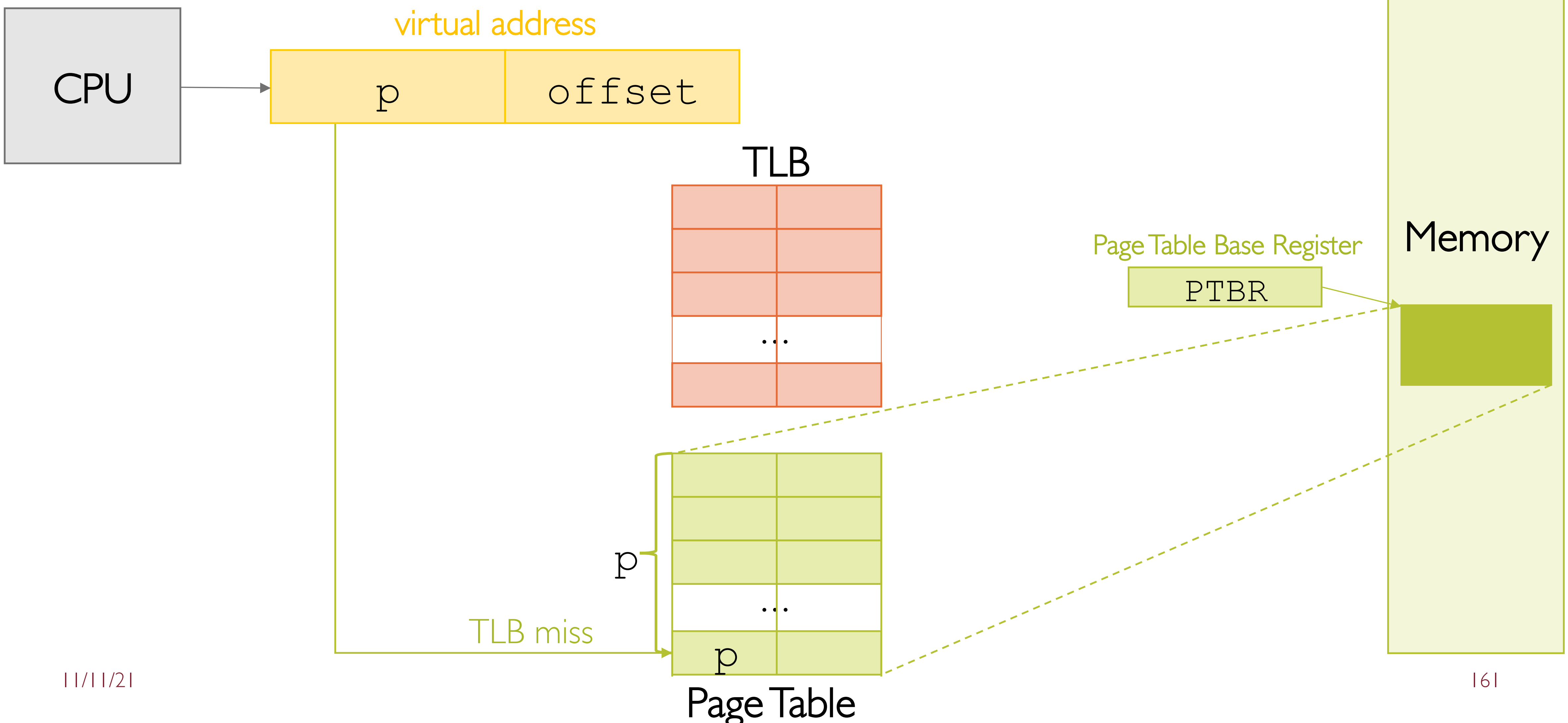
Translation Look-aside Buffer (TLB)



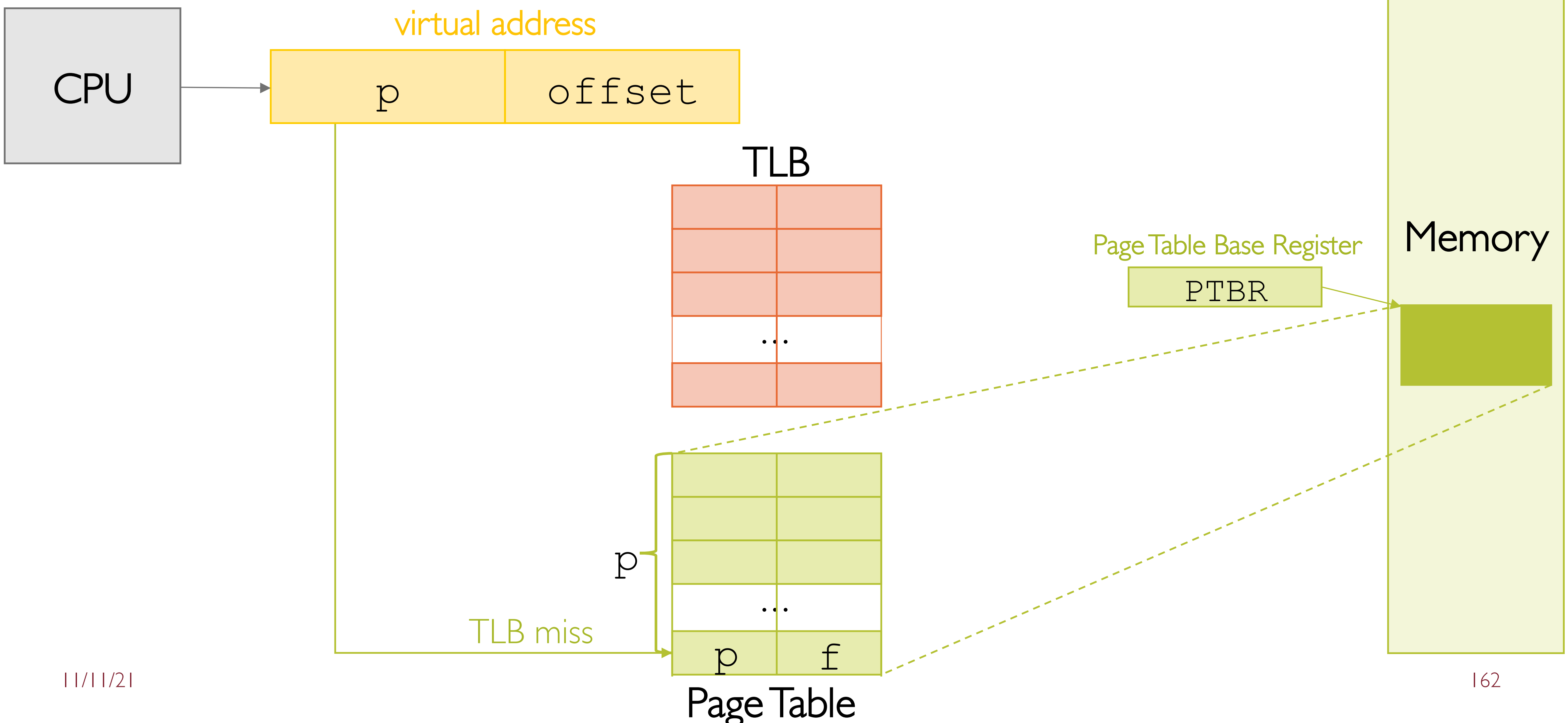
Translation Look-aside Buffer (TLB)



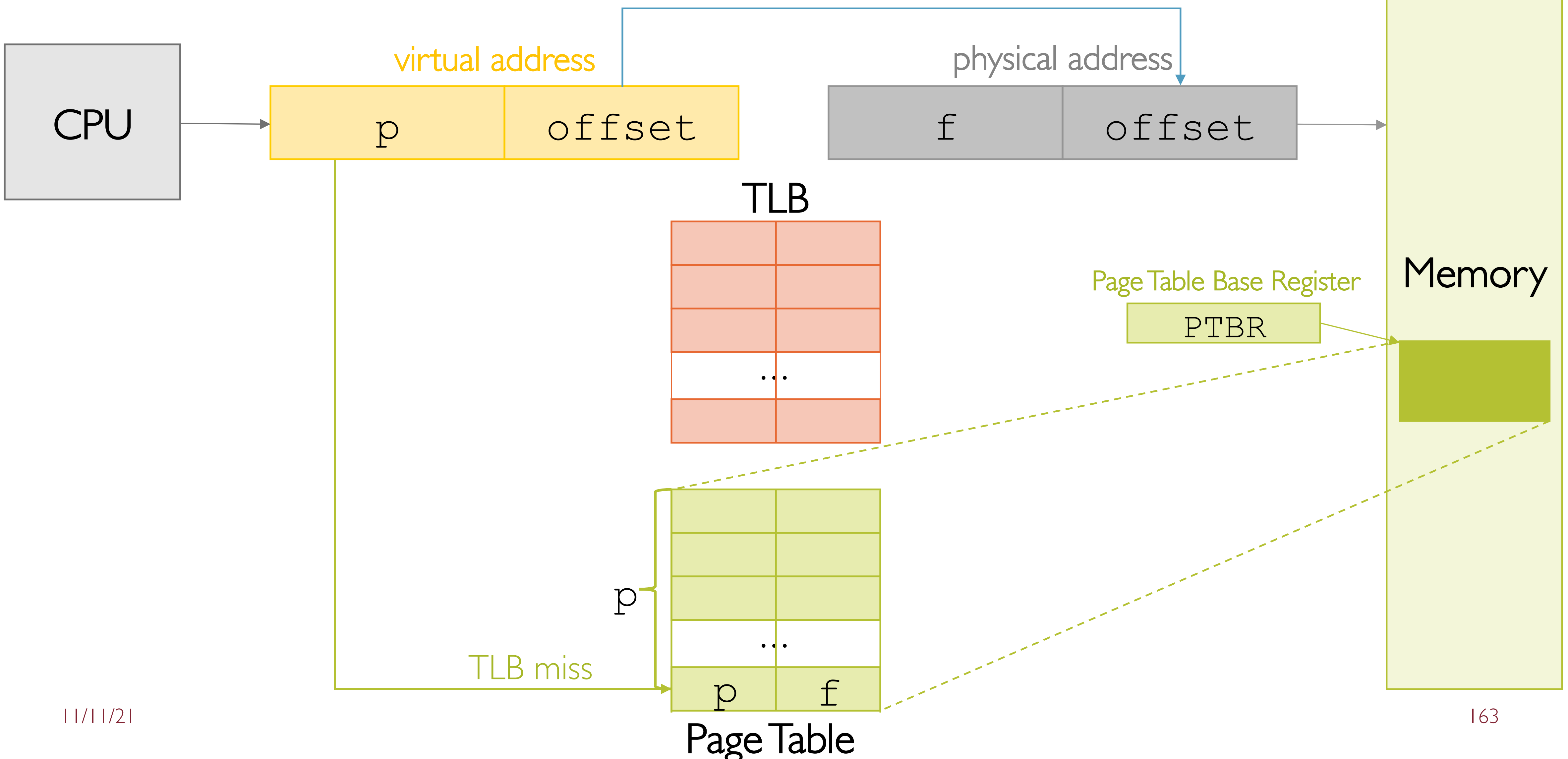
Translation Look-aside Buffer (TLB)



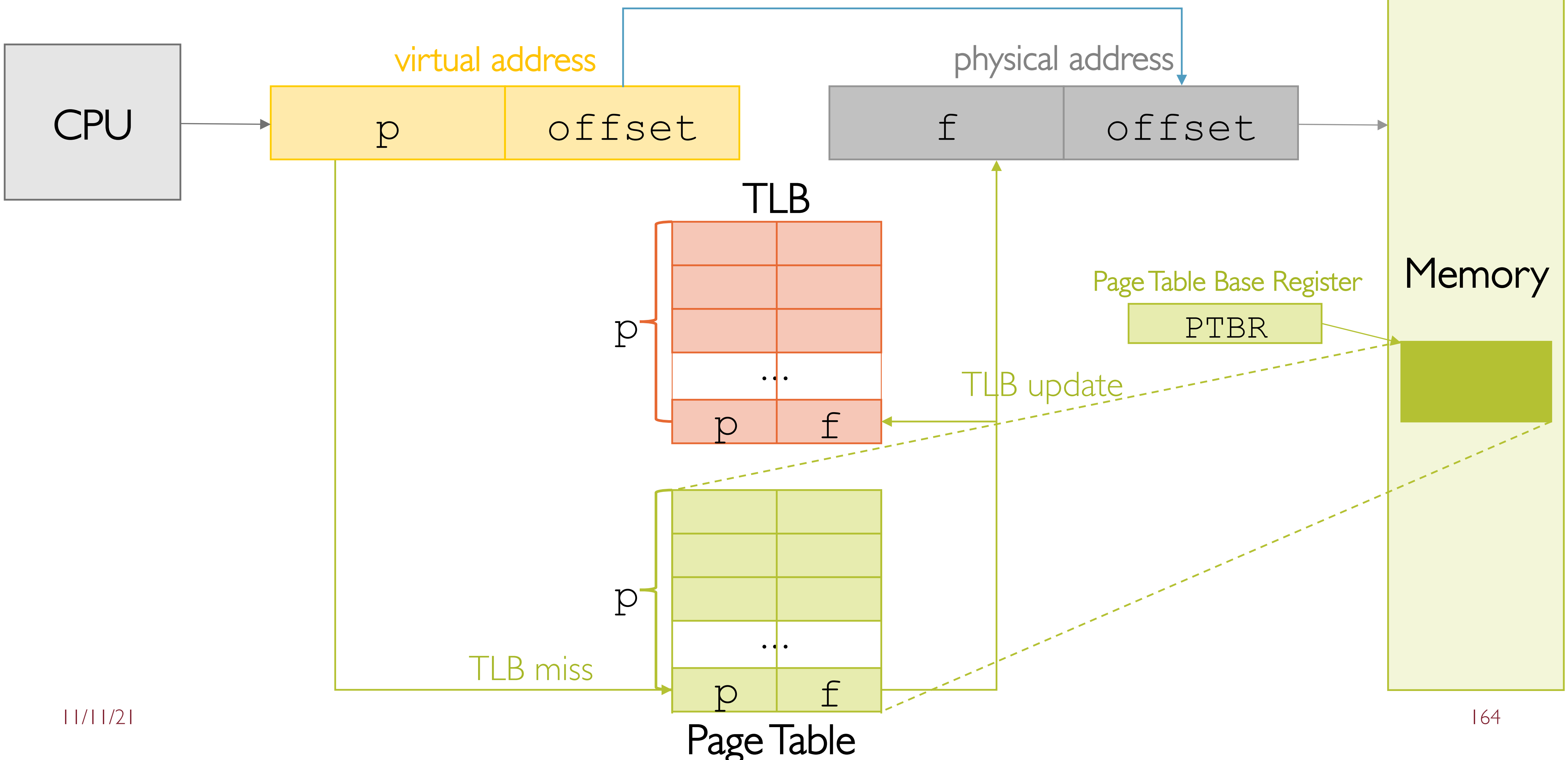
Translation Look-aside Buffer (TLB)



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

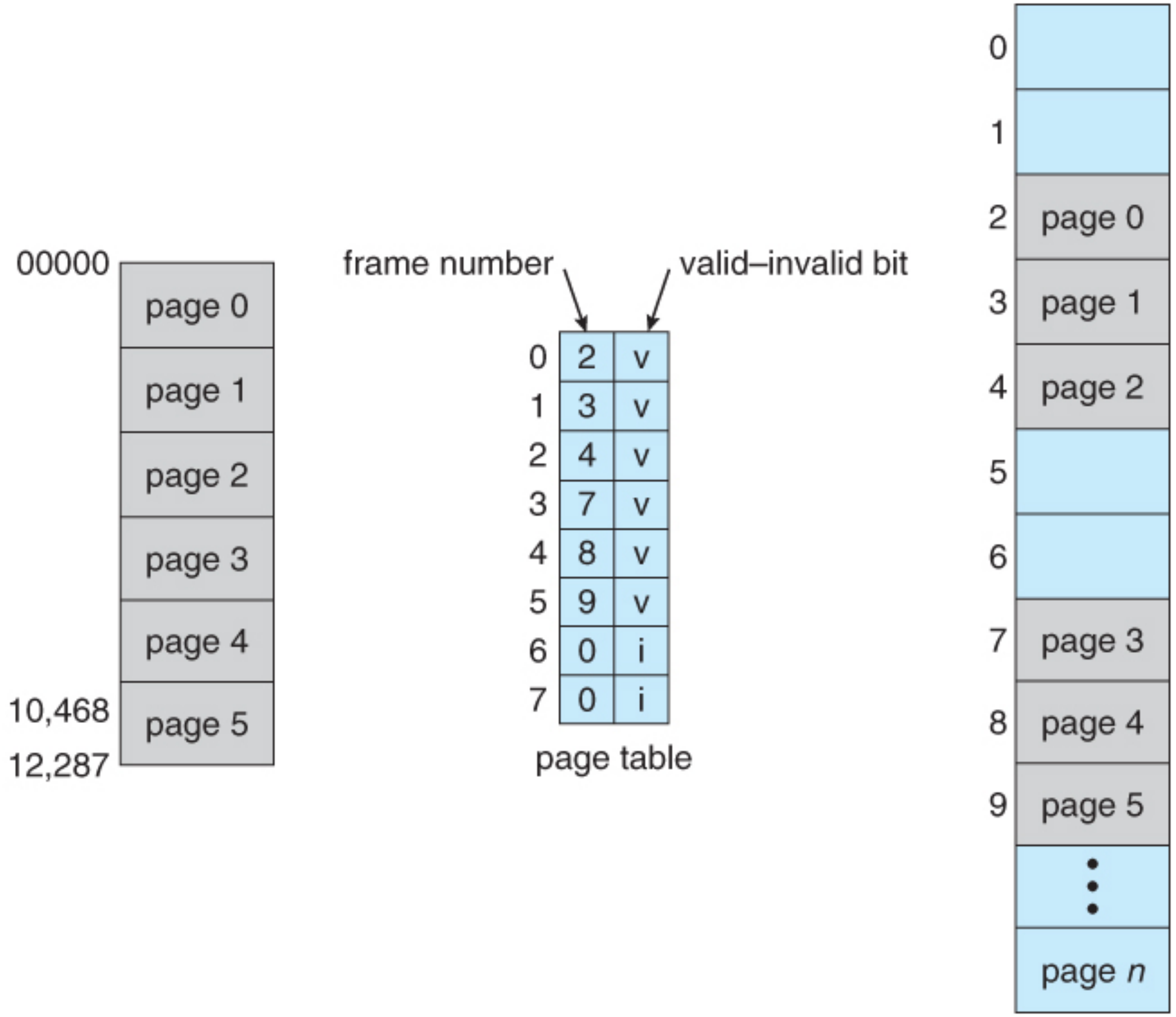
Additional Protection

- 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

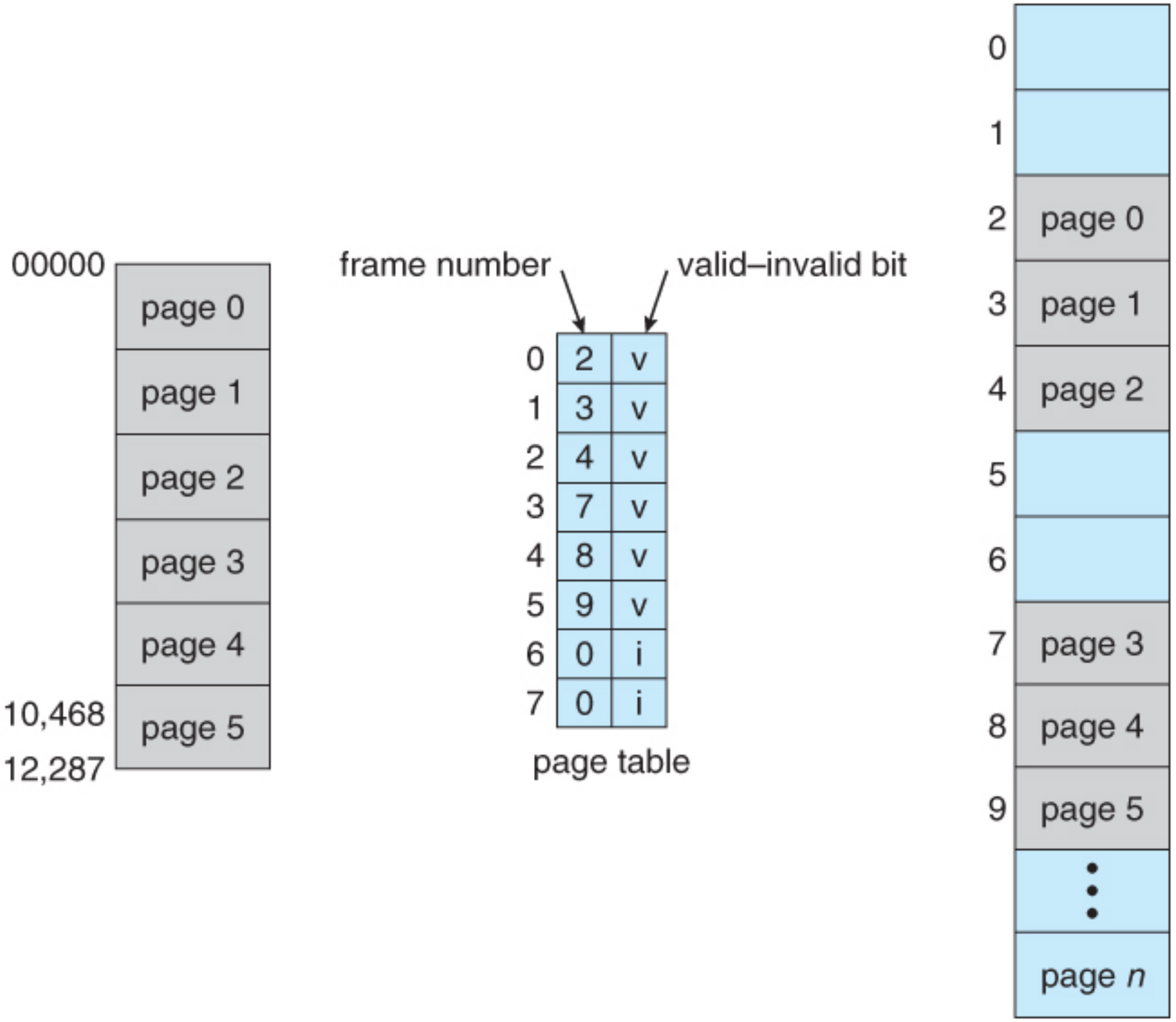
- valid/invalid bits cannot block all illegal memory accesses, due to the internal fragmentation
- Many processes do not use all of the page table entries available, particularly in modern systems with very large potential page tables
- Some systems use a page-table length register (`PTLR`) to specify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

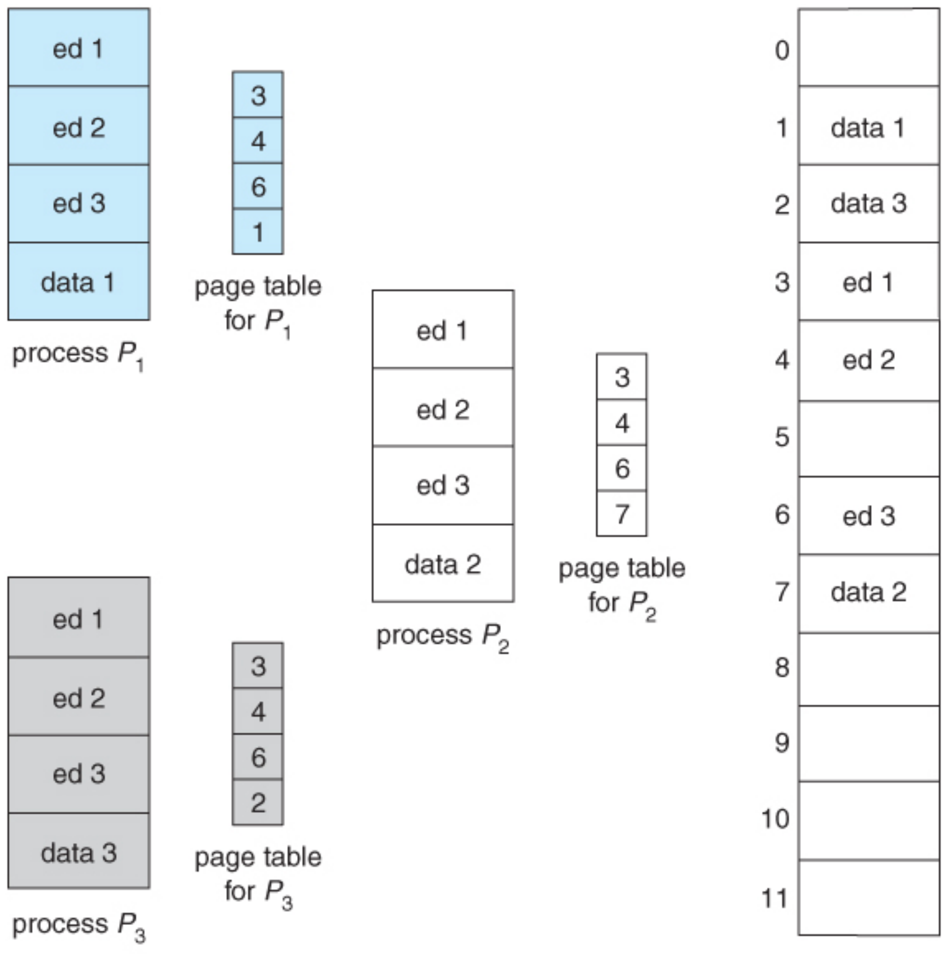
Sharing Pages

- 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

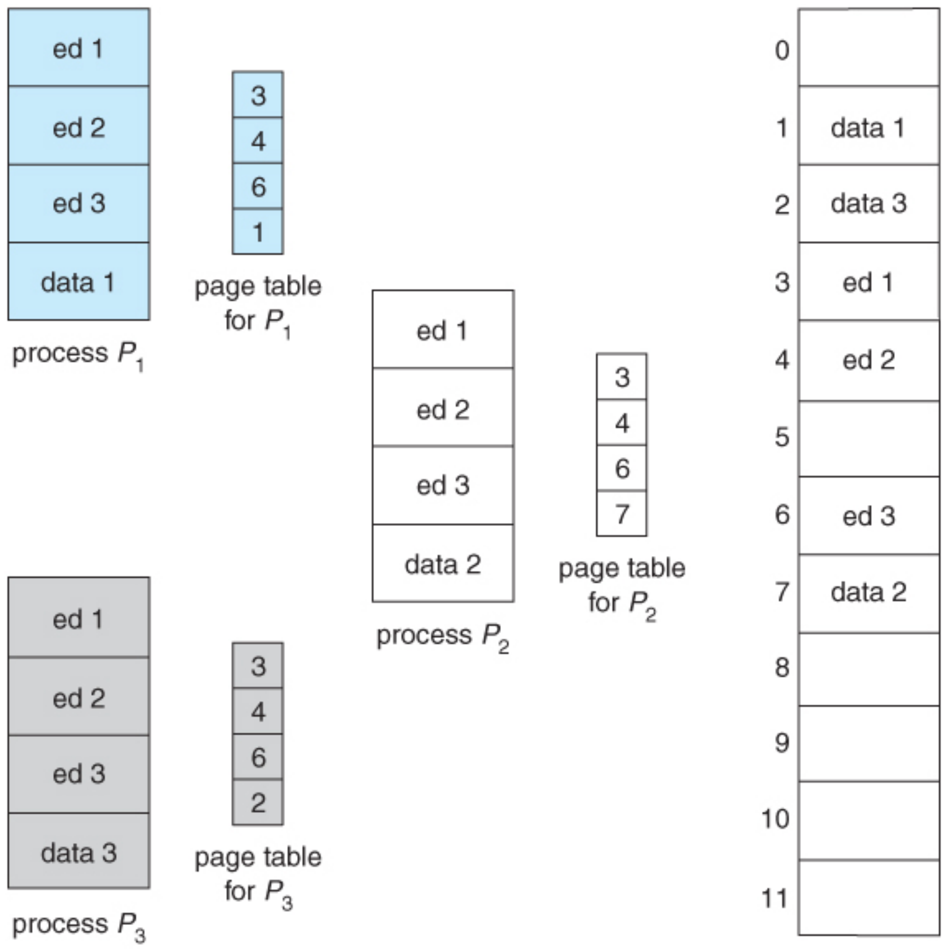
- Paging systems can make it very easy to share blocks of memory, since memory doesn't have to be contiguous anymore
- This can be done by simply duplicating page entries of different processes to the same page frames (both for code and data)
- Only if code is reentrant:
  - it does not write to or change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