

Sistemi Operativi I

Corso di Laurea in Informatica
2023-2024

Gabriele Tolomei

Dipartimento di Informatica
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

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

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

Contiguous Memory Allocation

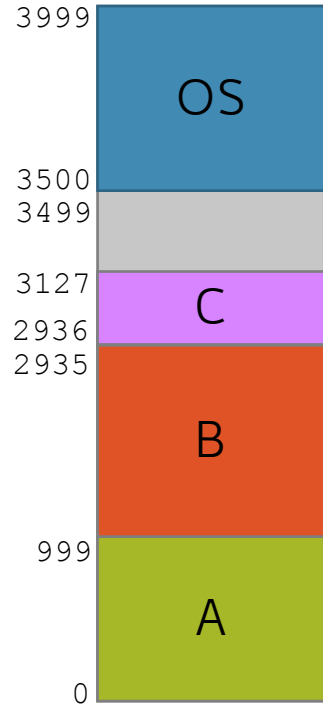
- So far, we have assumed each process is allocated into a contiguous space of physical memory
- One simple method is to divide upfront all available memory dedicated to user processes into **equally-sized** segments/partitions
 - Assign each process to a segment
 - Implicitly restricts the grade of 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

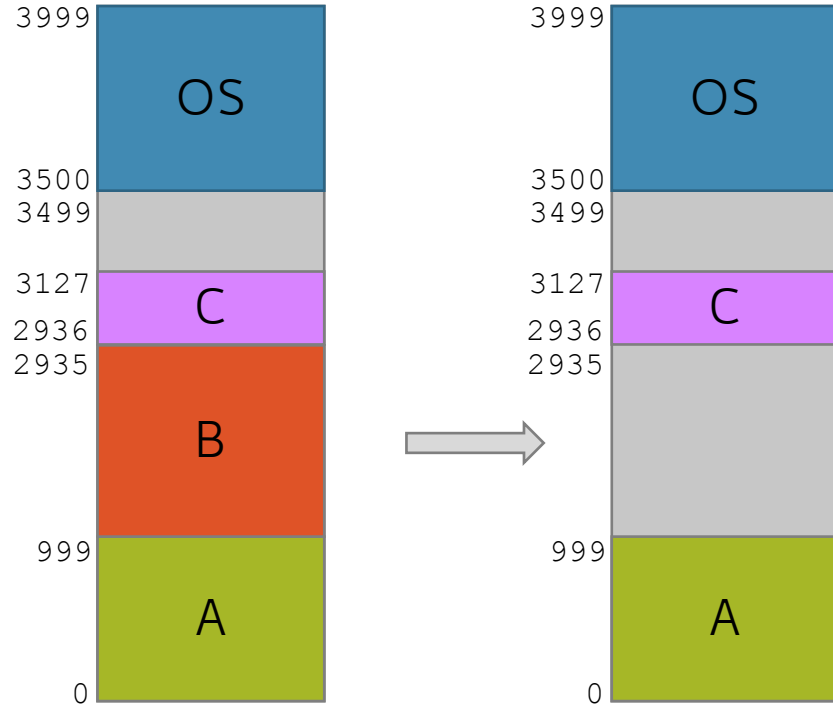
Contiguous Memory Allocation

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

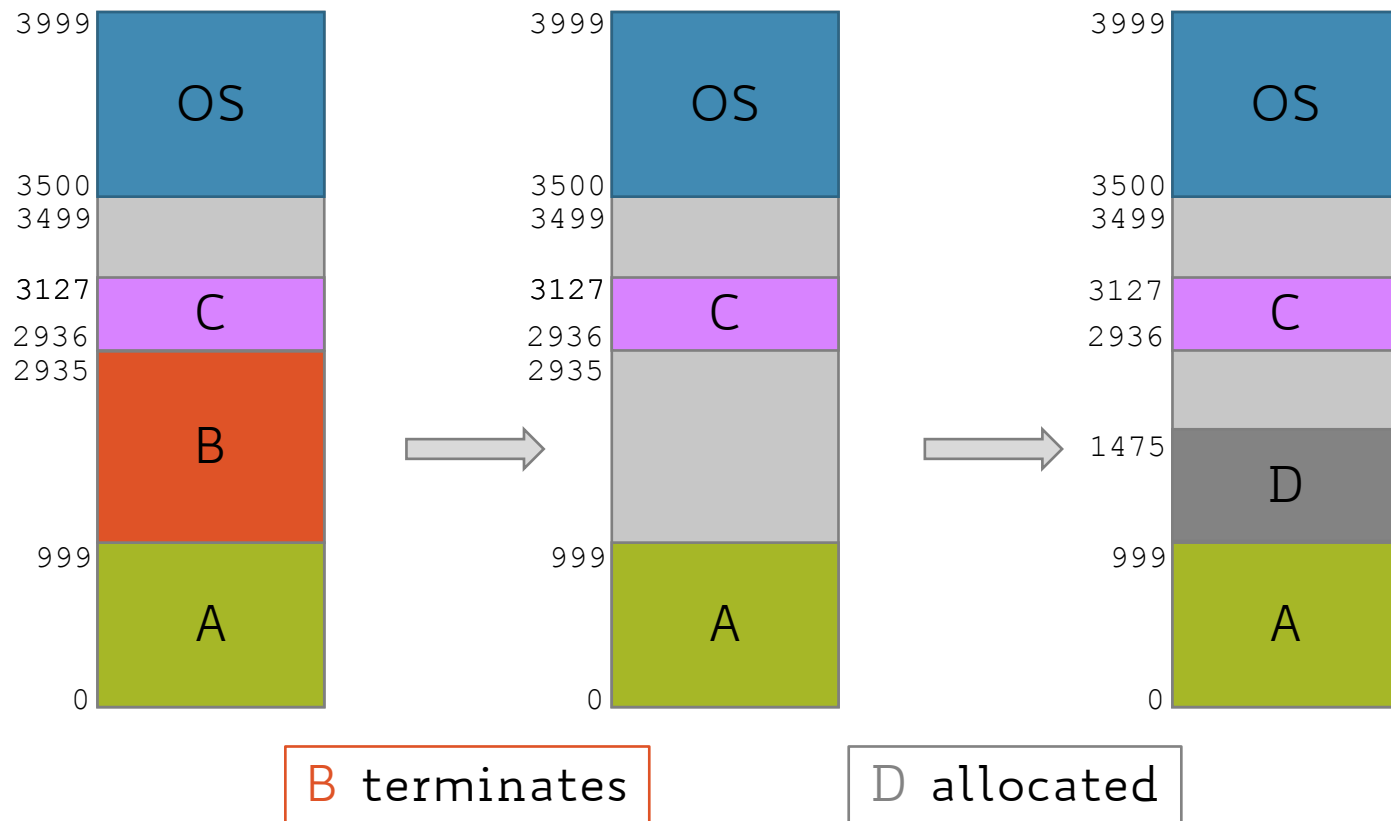
An alternative approach is for the OS to keep track of **free** (unused) memory segments, as processes enter the system, grow, and terminate



B terminates

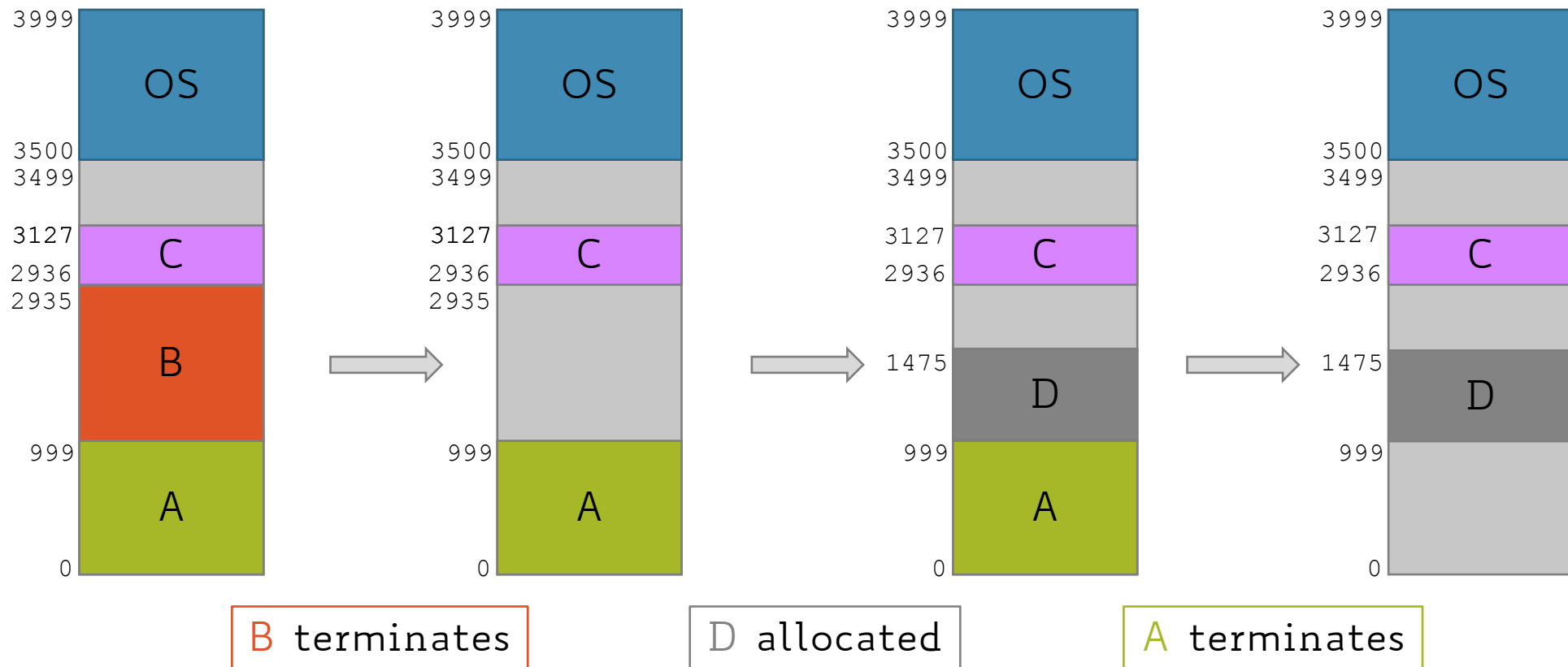
Contiguous Memory Allocation

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

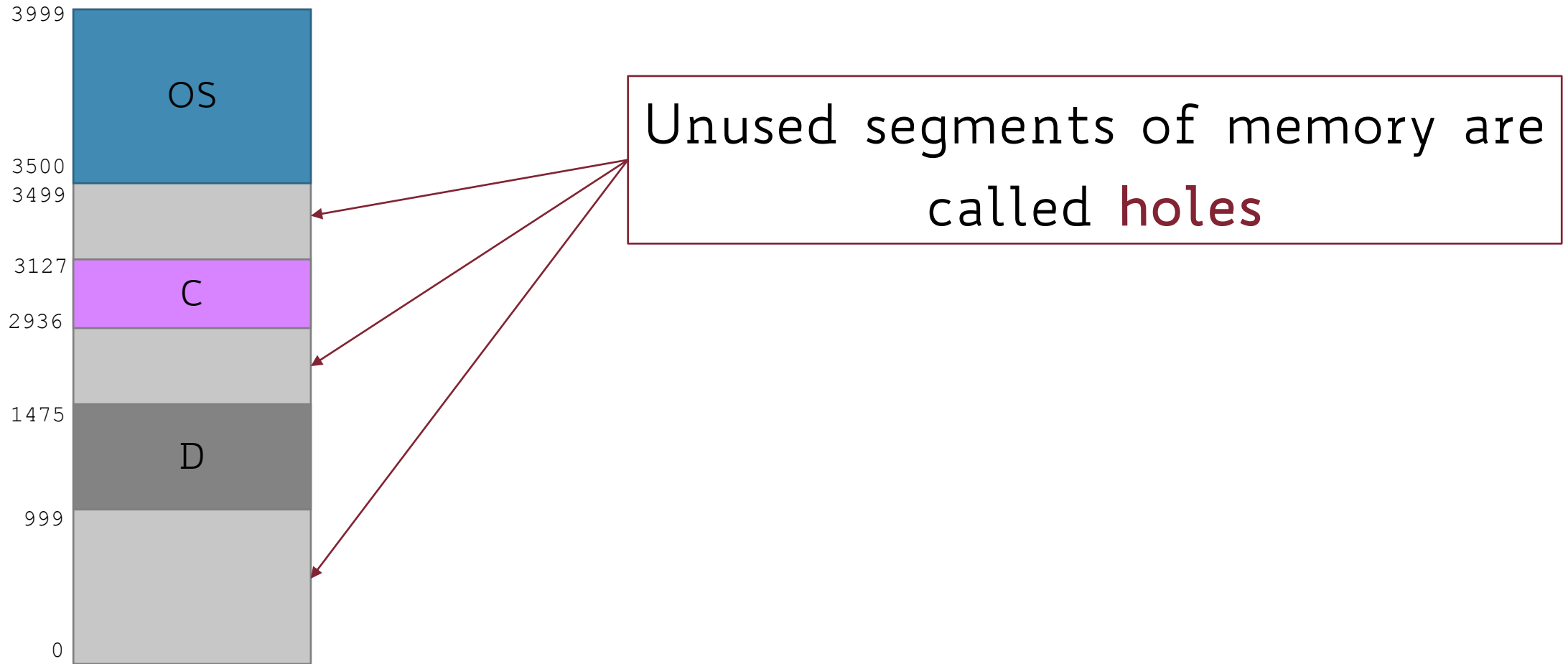


Contiguous Memory Allocation

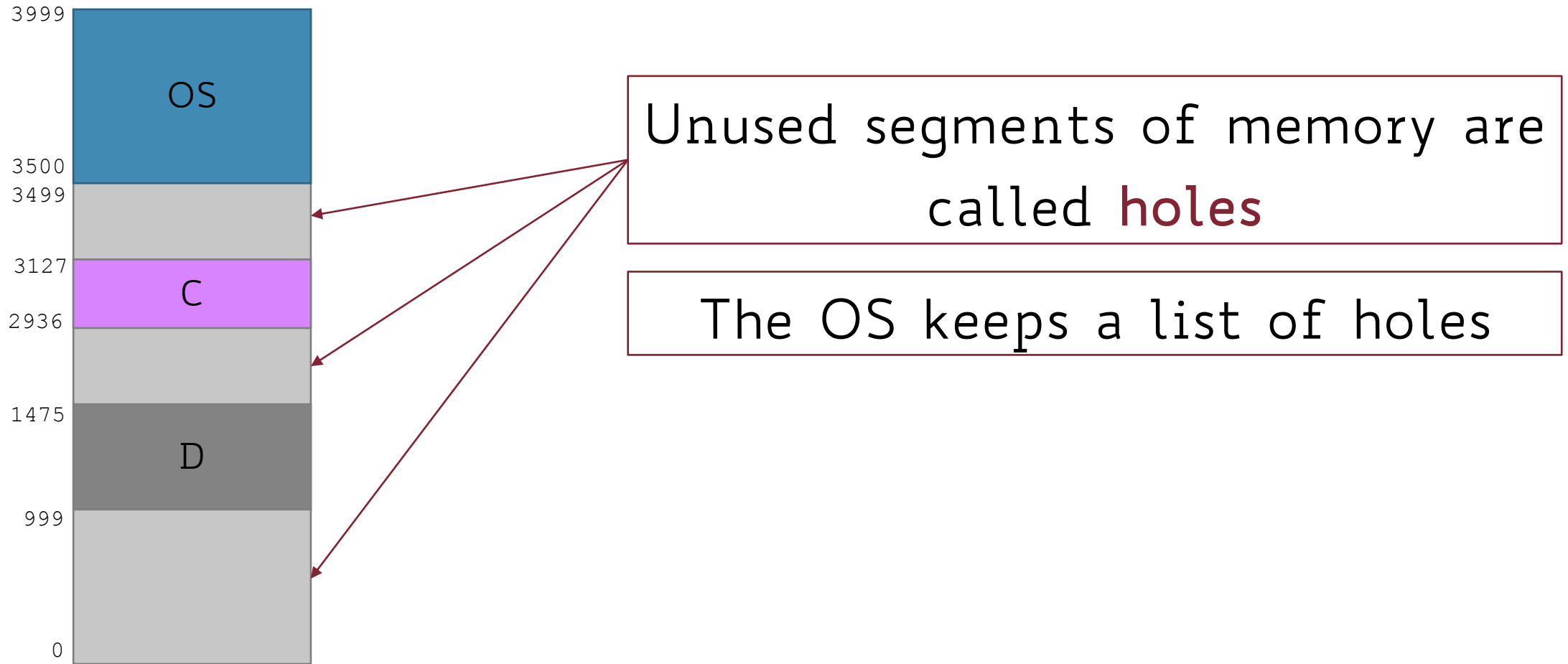
An alternative approach is for the OS to keep track of **free** (unused) memory segments, as processes enter the system, grow, and terminate



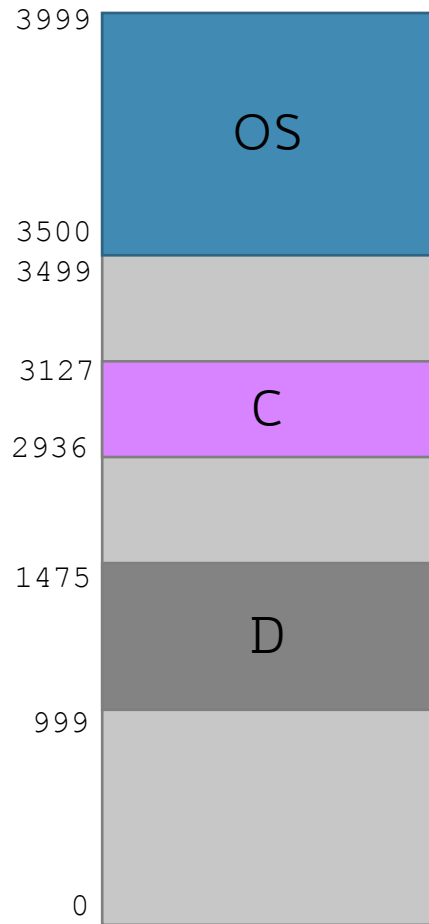
Contiguous Memory Allocation



Contiguous Memory Allocation



Contiguous Memory Allocation

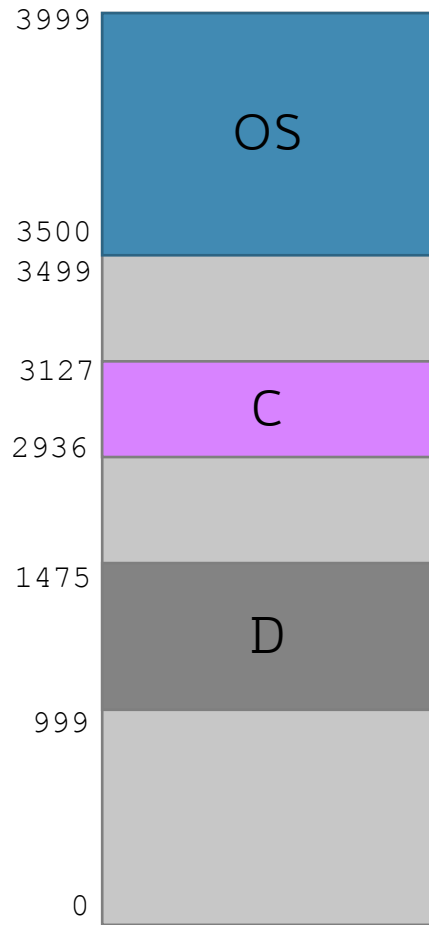


Unused segments of memory are called **holes**

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

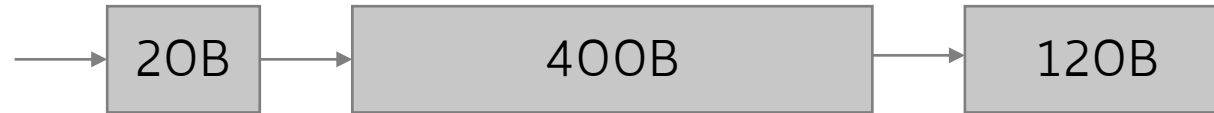
- Linearly scan the list of holes until one is found that is big enough to satisfy the request
- Subsequent requests may either start from the beginning of the list or from the end of previous search

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

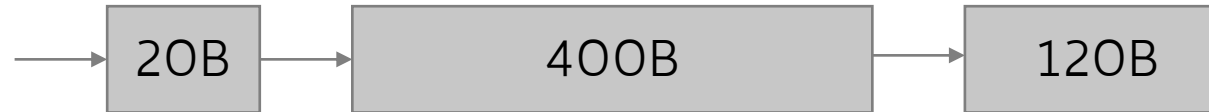
Memory Allocation Policies: First-Fit

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



Memory Allocation Policies: First-Fit

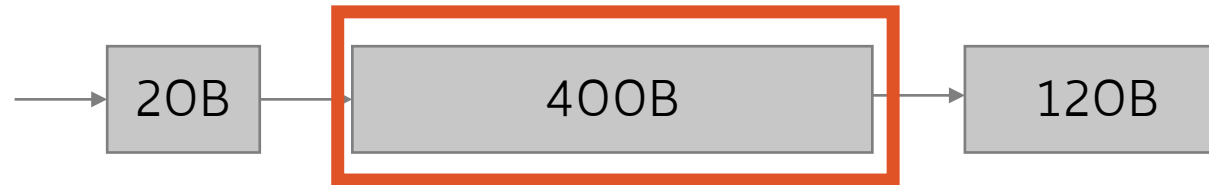
Suppose process **X** needs 100B of memory to be loaded, and the list of holes is as follows:



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

Memory Allocation Policies: First-Fit

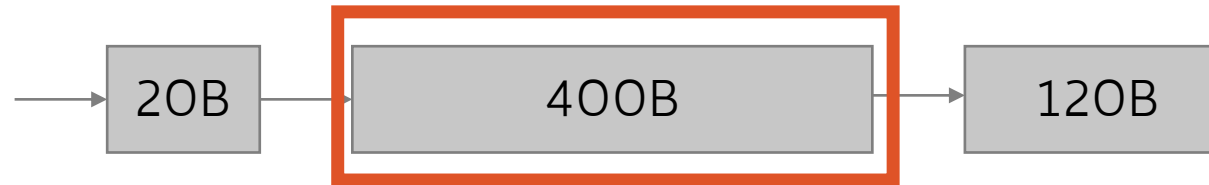
Suppose process **X** needs 100B of memory to be loaded, and the list of holes is as follows:



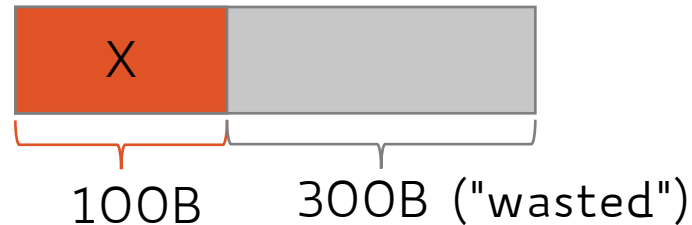
Which segment will be picked by the OS using first-fit?

Memory Allocation Policies: First-Fit

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

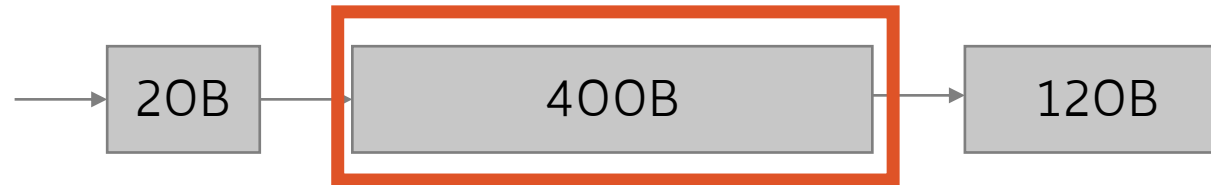


Which segment will be picked by the OS using first-fit?



Memory Allocation Policies: First-Fit

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



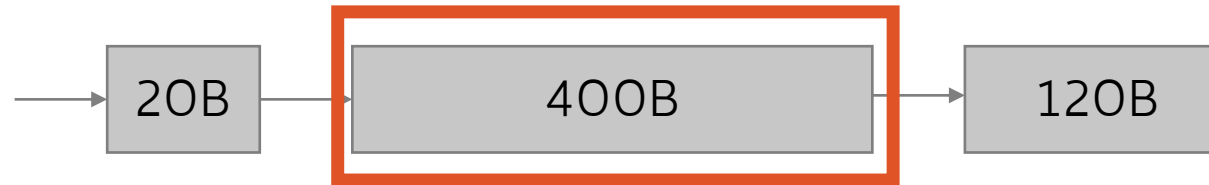
Which segment will be picked by the OS using first-fit?



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

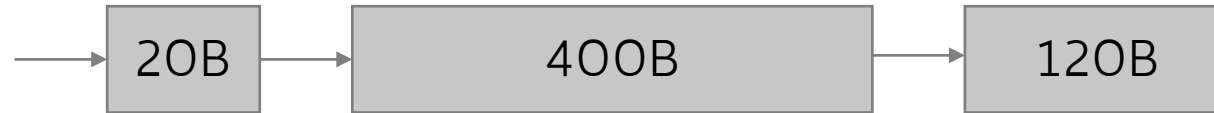
- Allocate the smallest hole that is big enough to satisfy the request
- This saves large holes for other process requests that may need them
- However, the resulting unused portions of holes may be too small to be of any use, and will therefore be wasted
- **Complexity:** still $O(n)$ but can be $O(\log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

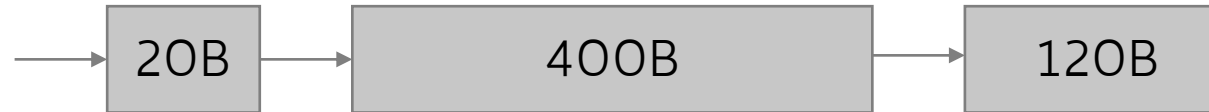
Memory Allocation Policies: Best-Fit

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



Memory Allocation Policies: Best-Fit

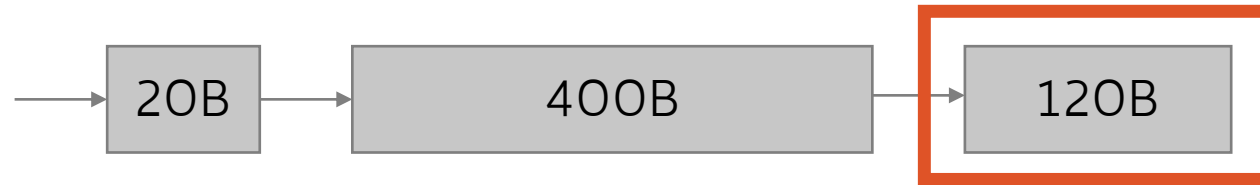
Suppose process **X** needs 100B of memory to be loaded, and the list of holes is as follows:



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

Memory Allocation Policies: Best-Fit

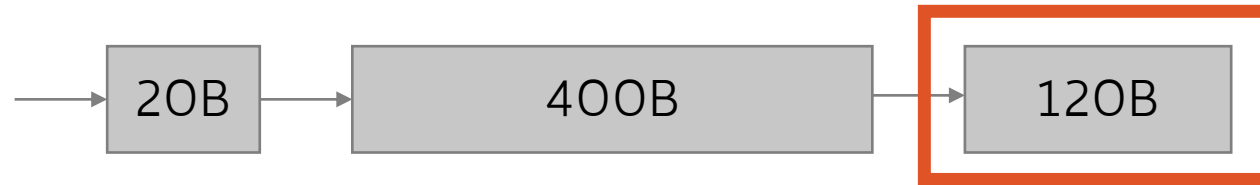
Suppose process **X** needs 100B of memory to be loaded, and the list of holes is as follows:



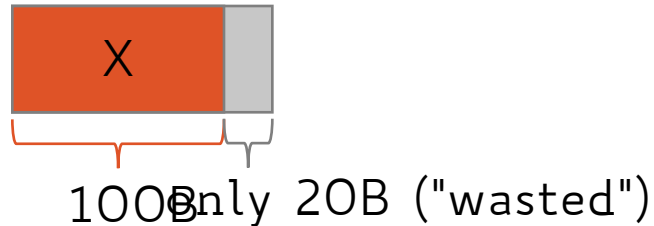
Which segment will be picked by the OS using best-fit?

Memory Allocation Policies: Best-Fit

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

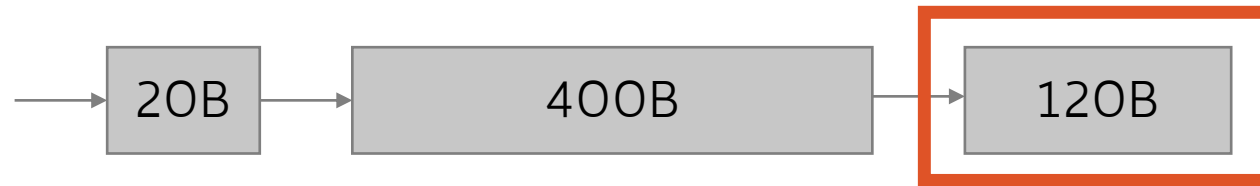


Which segment will be picked by the OS using best-fit?

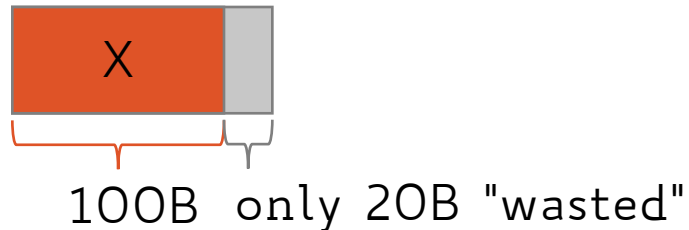


Memory Allocation Policies: Best-Fit

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



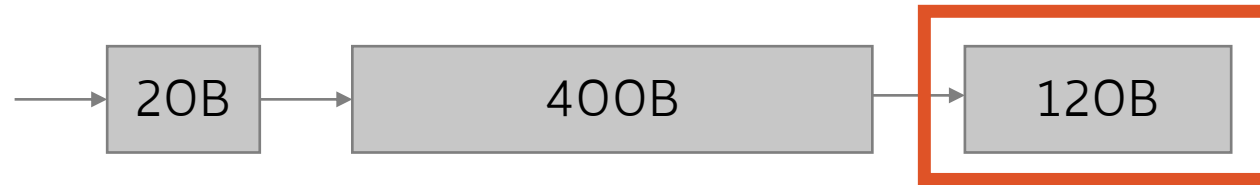
Which segment will be picked by the OS using best-fit?



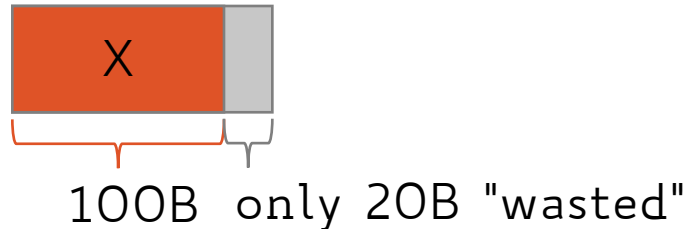
What if afterwards process **Y** requires 350B?

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

Problem

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

```
graph TD; A[Problem] --> B[External Fragmentation]; A --> C[Internal Fragmentation];
```

External
Fragmentation

Internal
Fragmentation

External Fragmentation

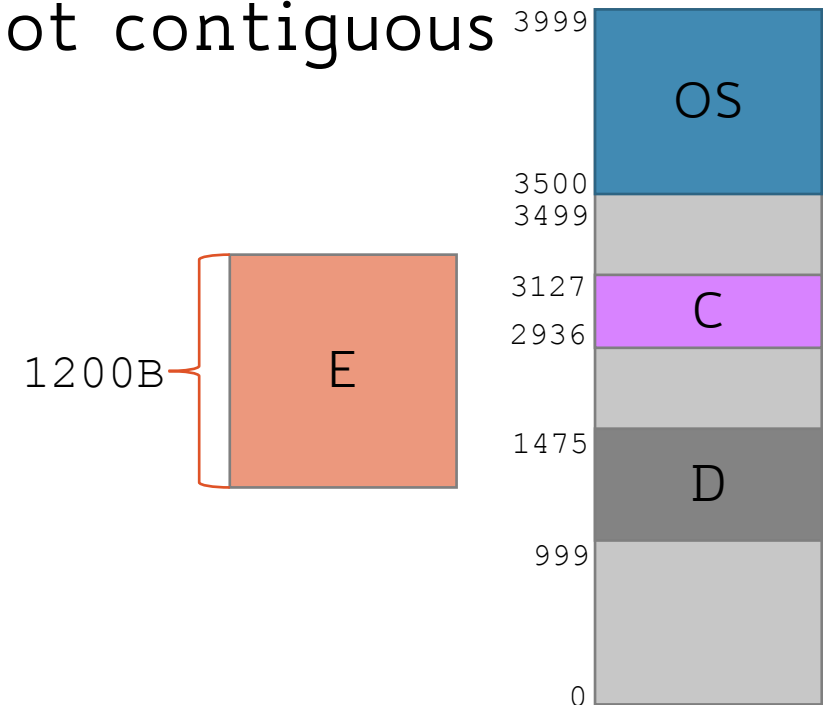
- Frequent loading and unloading processes causes holes to be broken into small (i.e., unusable) chunks

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

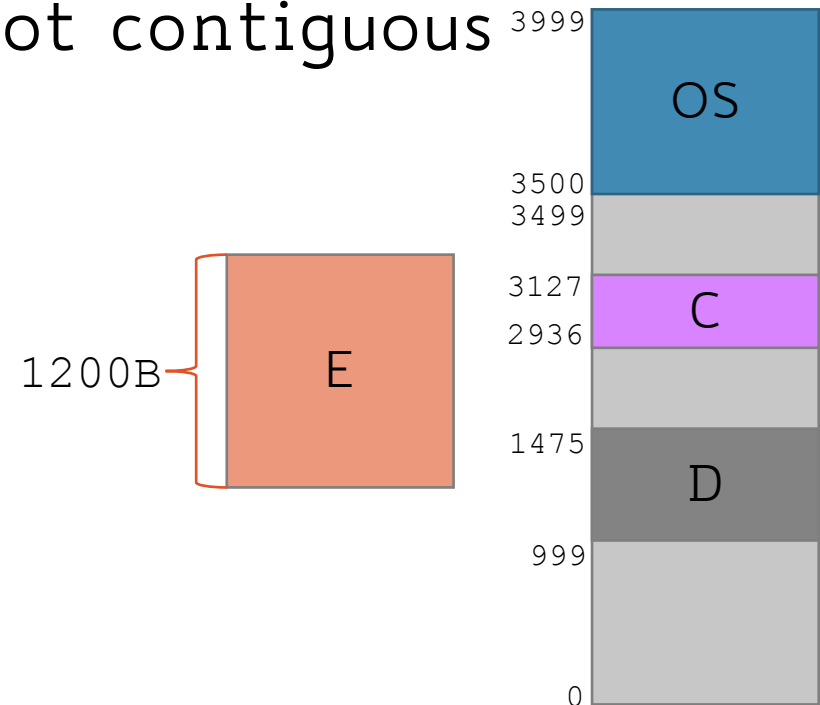


External Fragmentation

- 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

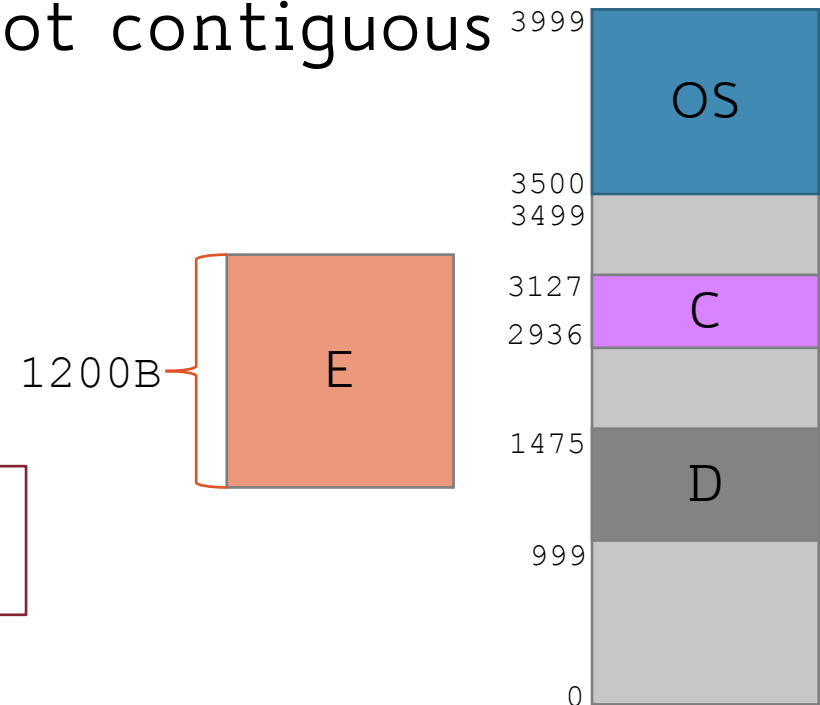
- Frequent loading and unloading processes causes holes to be broken into small (i.e., unusable) chunks
- It happens when there is enough memory to load a process in memory but space is not contiguous

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

1/3 of memory space is wasted on average

Goal:

Allocation policy that minimizes wasted space!



Internal Fragmentation

- It happens when memory internal to a segment is wasted

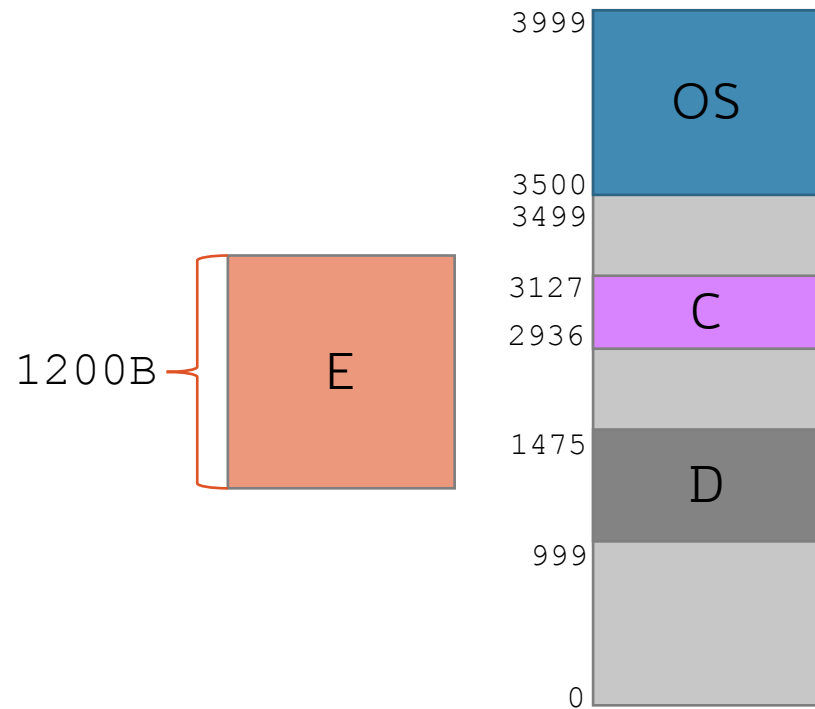
Internal Fragmentation

- 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

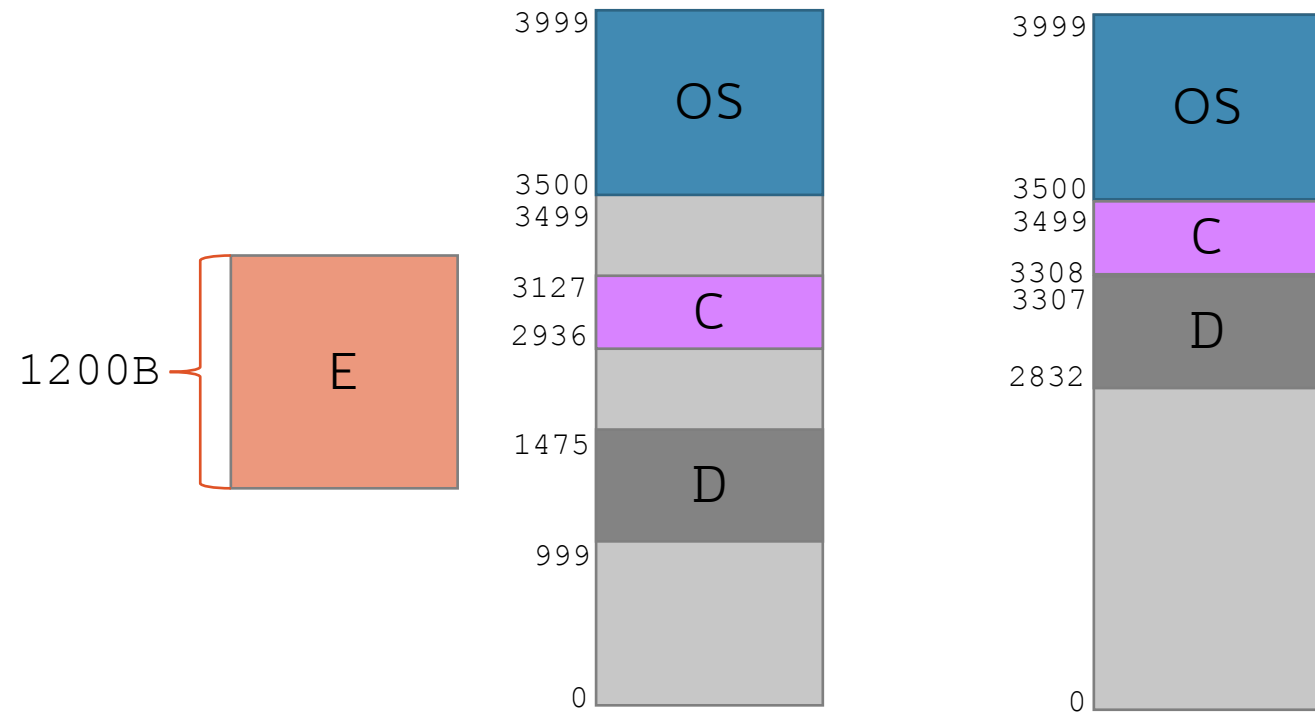
Internal Fragmentation

- 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

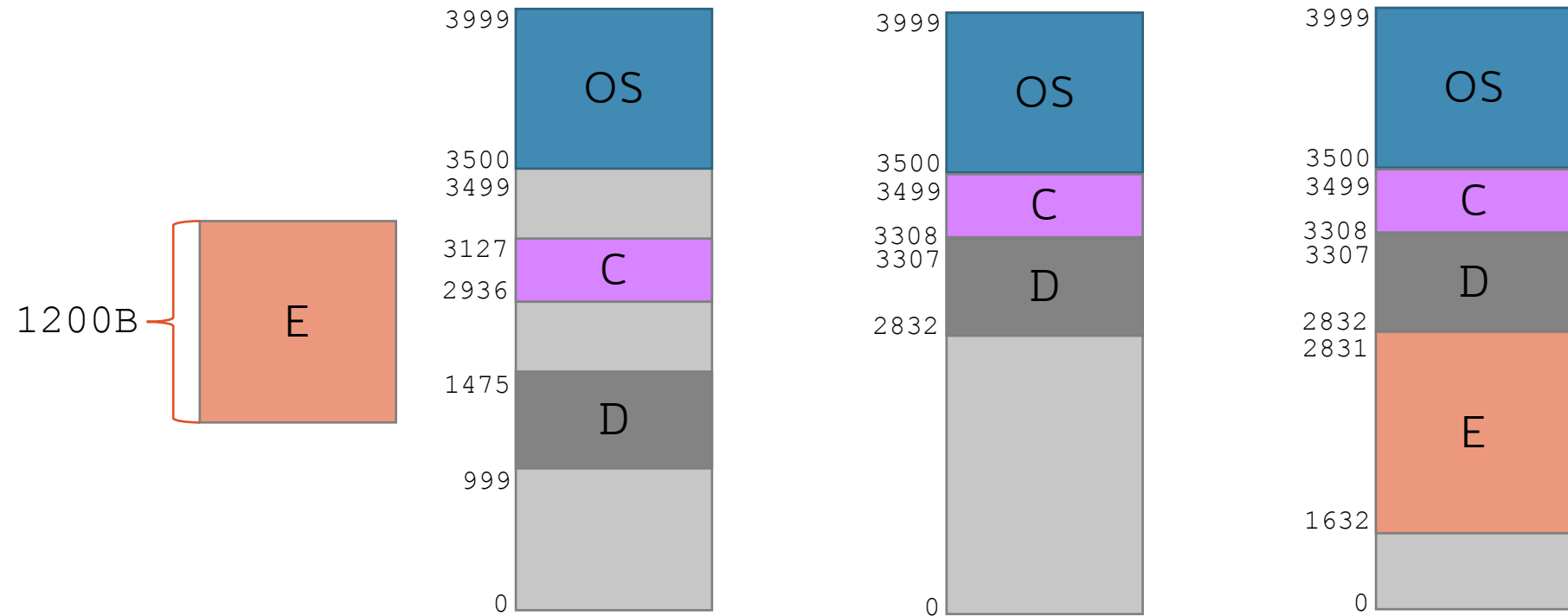
Solution to Fragmentation: Full Compaction



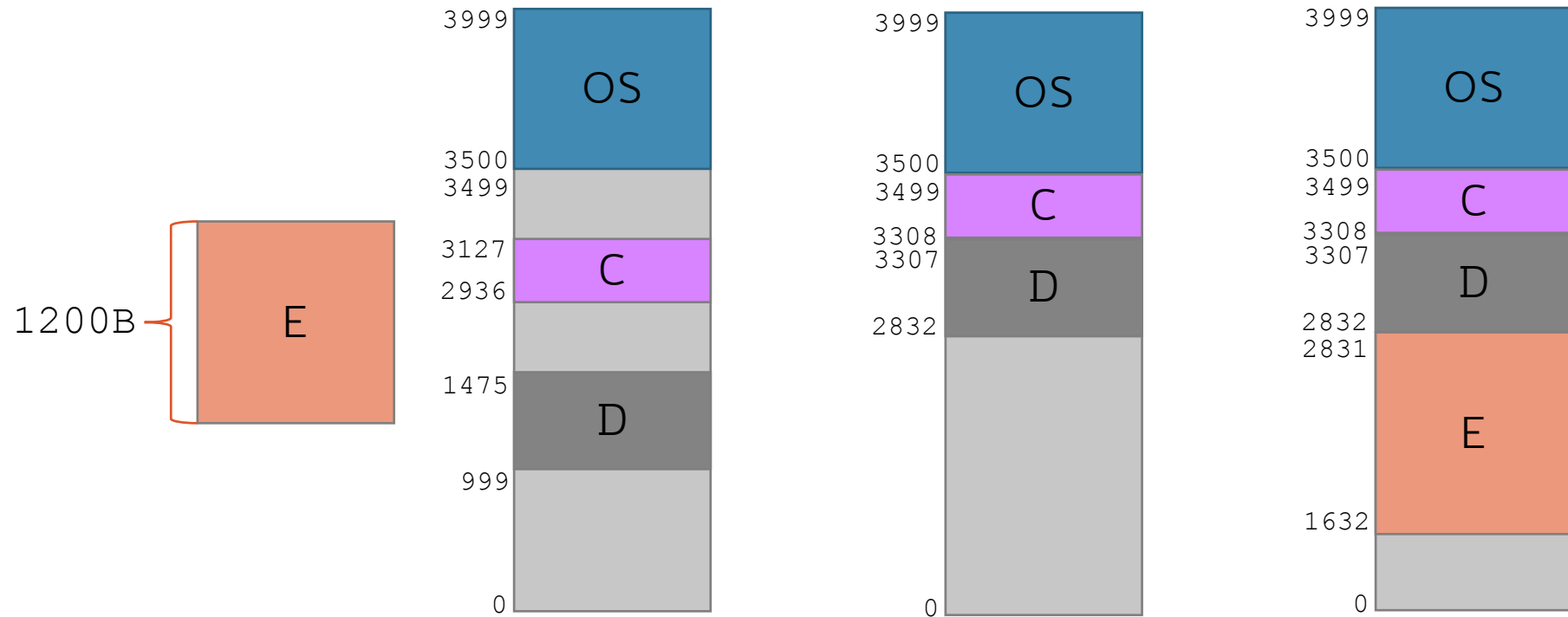
Solution to Fragmentation: Full Compaction



Solution to Fragmentation: Full Compaction

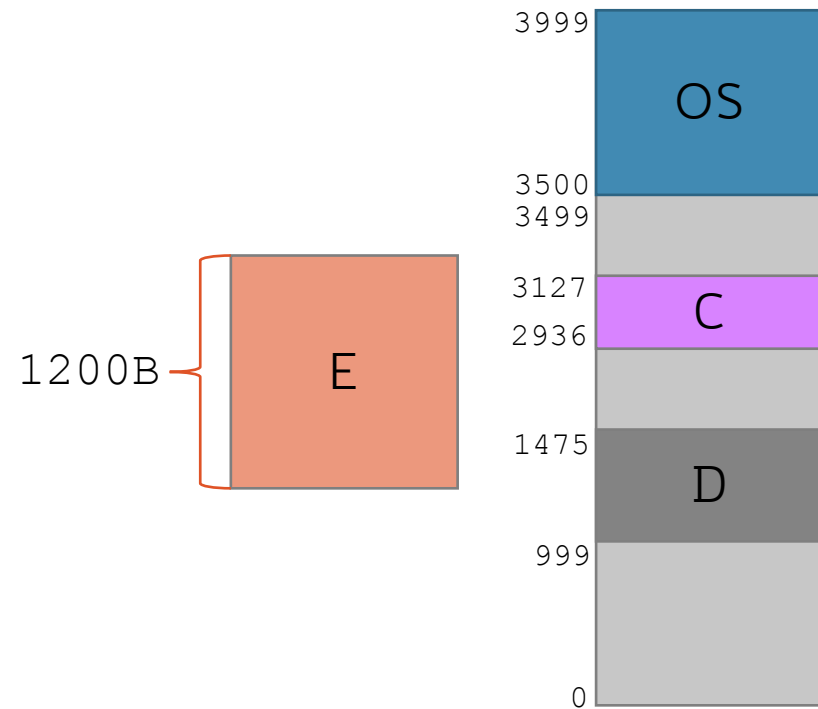


Solution to Fragmentation: Full Compaction

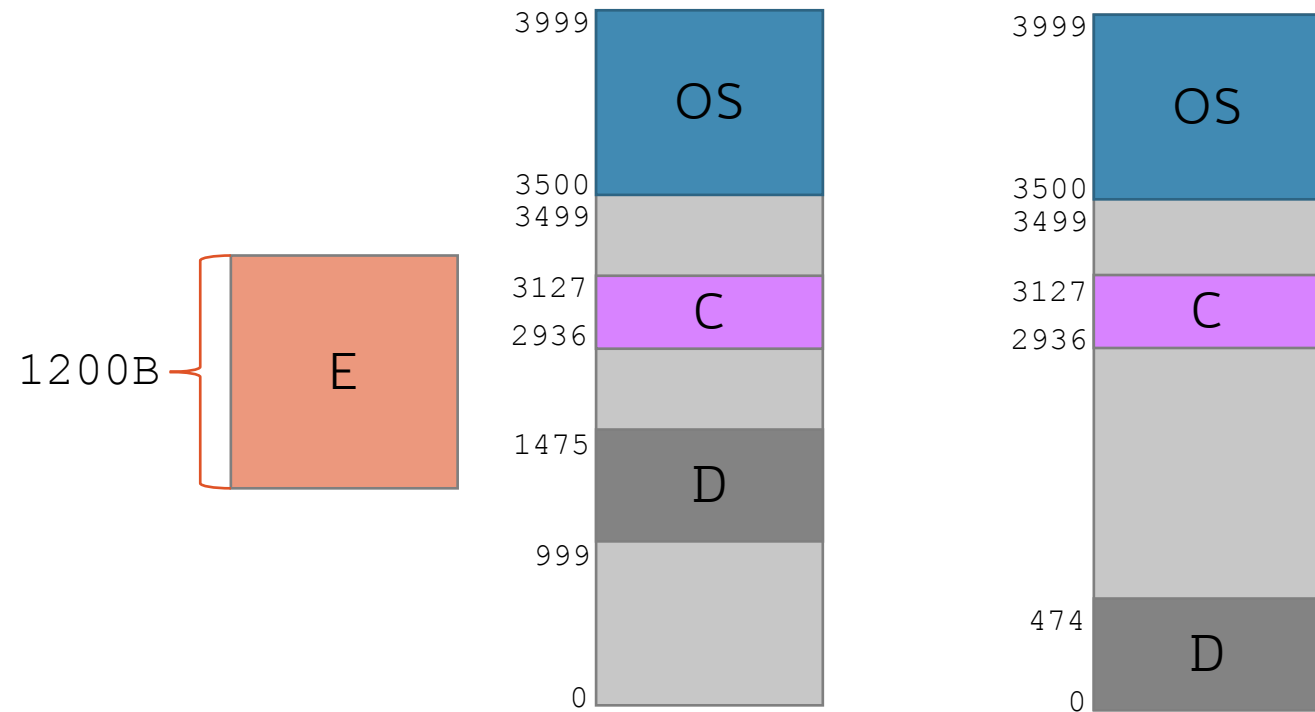


Only one hole is left but two processes need to be moved (C and D)

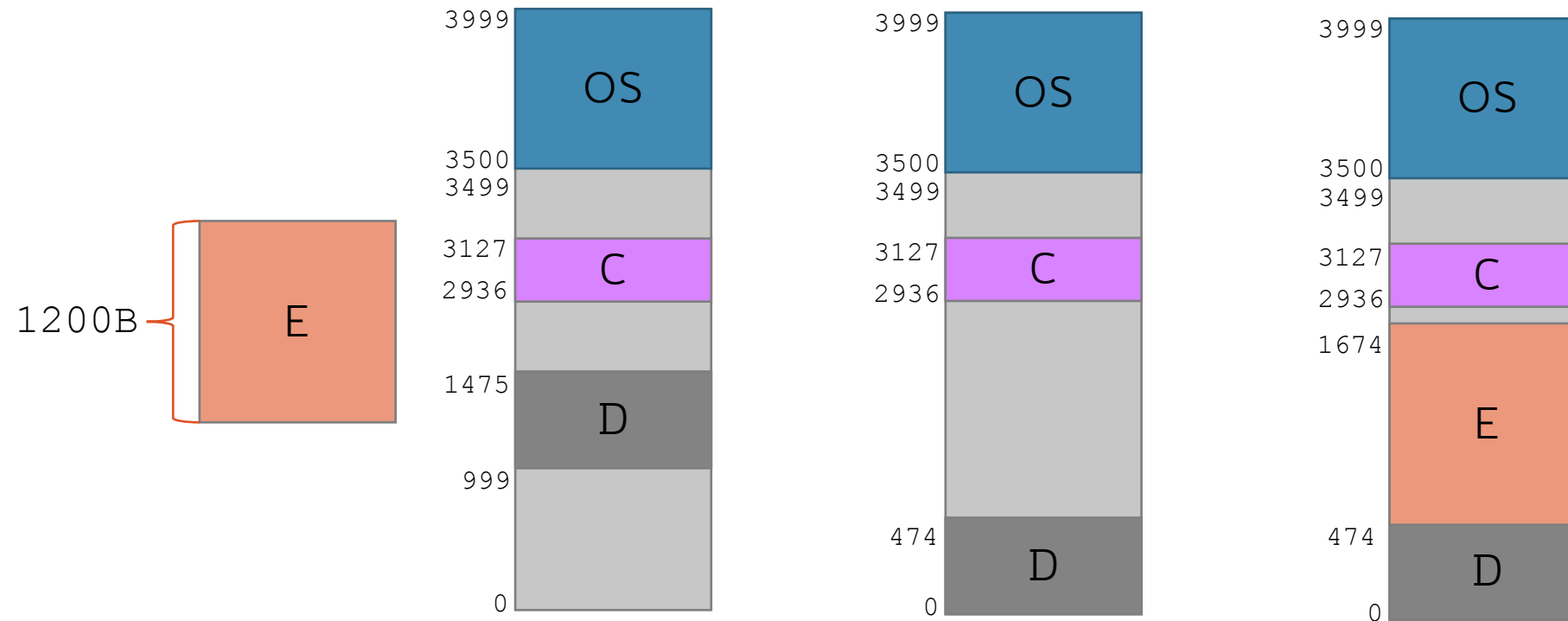
Solution to Fragmentation: Partial Compaction



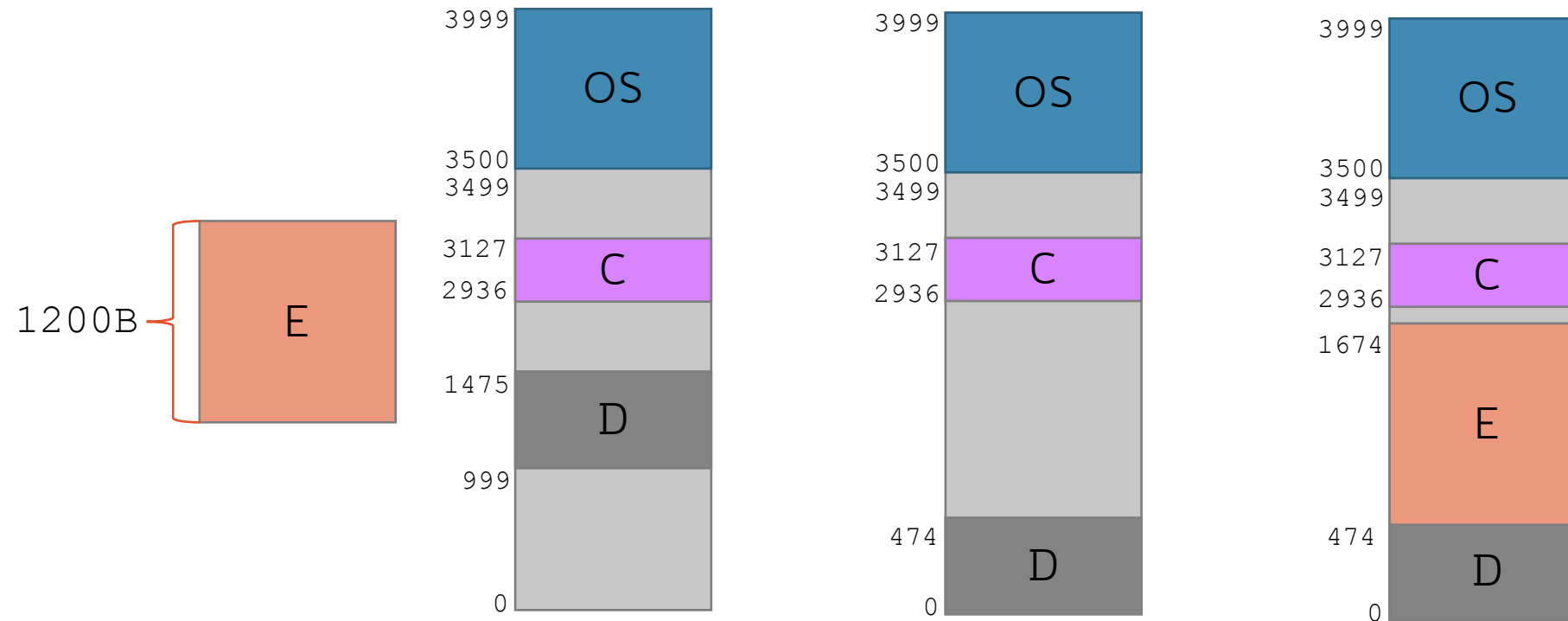
Solution to Fragmentation: Partial Compaction



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

Swapping: Example

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

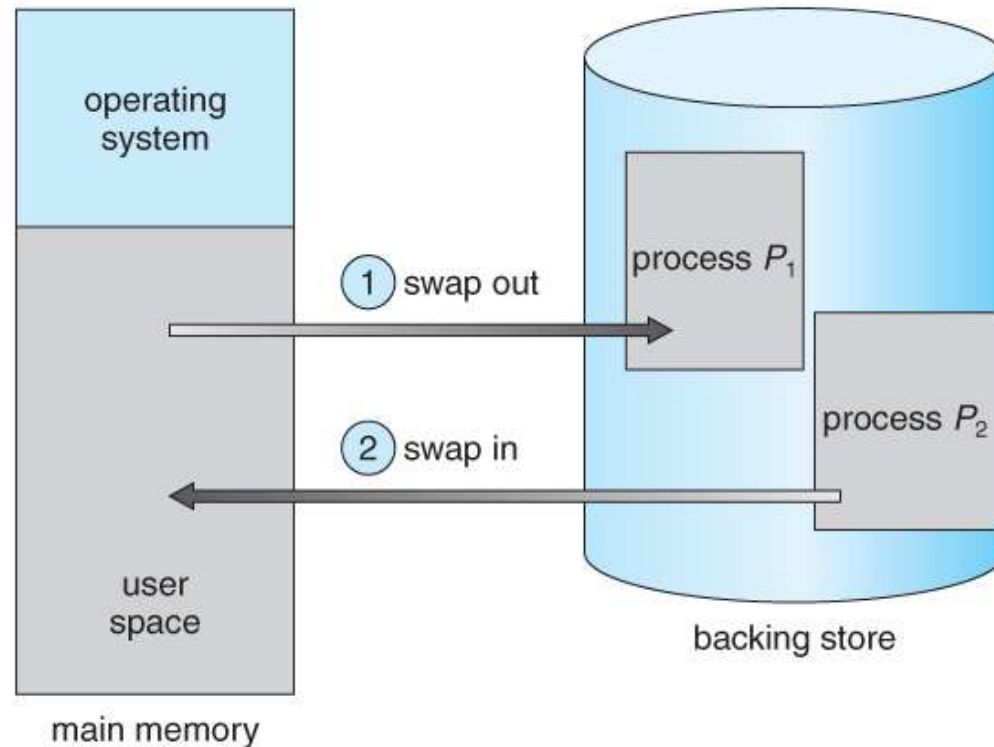
Swapping: Example

- Swapping is a very slow process compared to other operations due to the interaction with hard disk
- 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
- 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

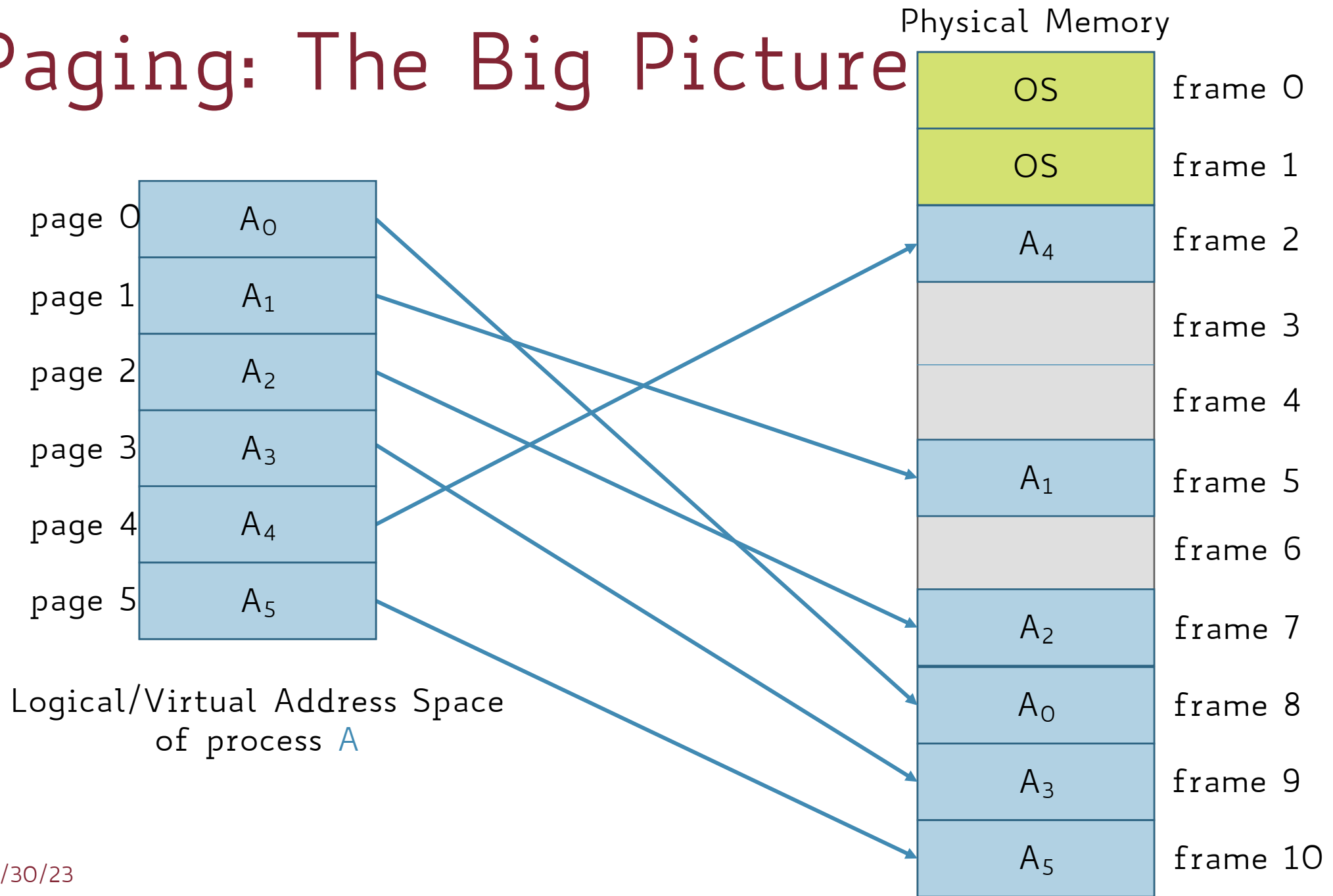
Processes spend **90%** of their time accessing only **10%** of their allocated memory space

Paging: The Big Picture

page 0	A_0
page 1	A_1
page 2	A_2
page 3	A_3
page 4	A_4
page 5	A_5

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

0	A ₀
1	A ₁
2	A ₂
3	A ₃
4	A ₄
5	A ₅

OS	0
OS	1
A ₄	2
	3
	4
A ₁	5
	6
A ₂	7
A ₀	8
A ₃	9
A ₅	10

Page Table: Mapping Pages to Frames

Lookup table to retrieve what frame a page is stored in

0	A ₀
1	A ₁
2	A ₂
3	A ₃
4	A ₄
5	A ₅

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

OS	0
OS	1
A ₄	2
	3
	4
A ₁	5
	6
A ₂	7
A ₀	8
A ₃	9
A ₅	10

Page Table: Mapping Pages to Frames

Lookup table to retrieve what frame a page is stored in

0	A ₀	Page	Frame	OS	0
1	A ₁	0	8	OS	1
2	A ₂	1	5	A ₄	2
3	A ₃	2	7		3
4	A ₄	3	9		4
5	A ₅	4	2	A ₁	5
		5	10		6
				A ₂	7
				A ₀	8
				A ₃	9
				A ₅	10

We have assumed **all** pages of a process are mapped to physical frames, but this is not always the case

Summary

- Contiguous memory allocation may cause fragmentation
 - External vs. Internal Fragmentation

Summary

- Contiguous memory allocation may cause fragmentation
 - External vs. Internal Fragmentation
- Existing countermeasures (compaction) exist but they are costly

Summary

- Contiguous memory allocation may cause fragmentation
 - External vs. Internal Fragmentation
- Existing countermeasures (compaction) exist but they are costly
- We may want to relax the constraint on having an entire process loaded in main memory

Summary

- Contiguous memory allocation may cause fragmentation
 - External vs. Internal Fragmentation
- Existing countermeasures (compaction) exist but they are costly
- We may want to relax the constraint on having an entire process loaded in main memory
- Paging solves all these issues!