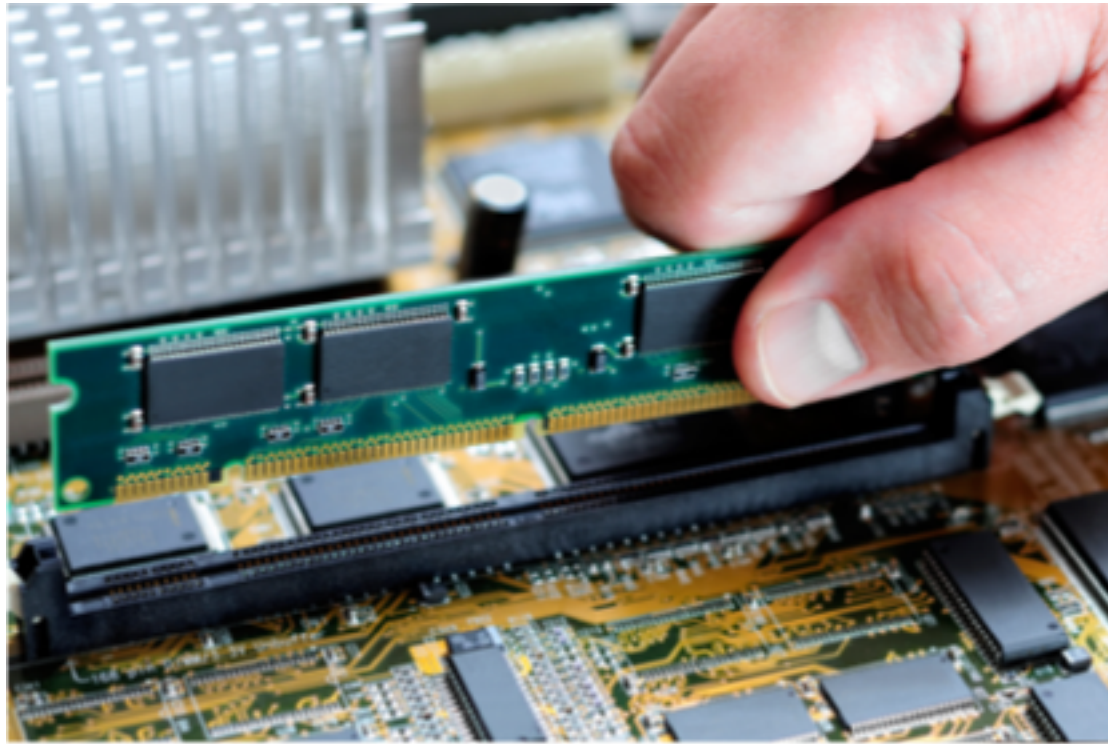


# CPE 460 Operating System Design

## *Chapter 7: Main Memory*

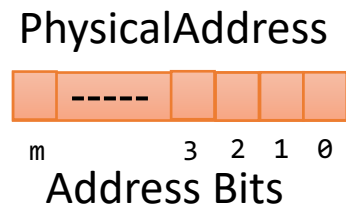
Ahmed Tamrawi

# What is a memory?



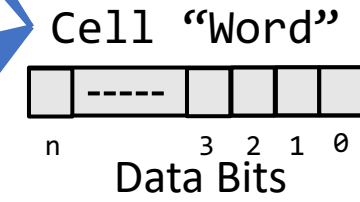
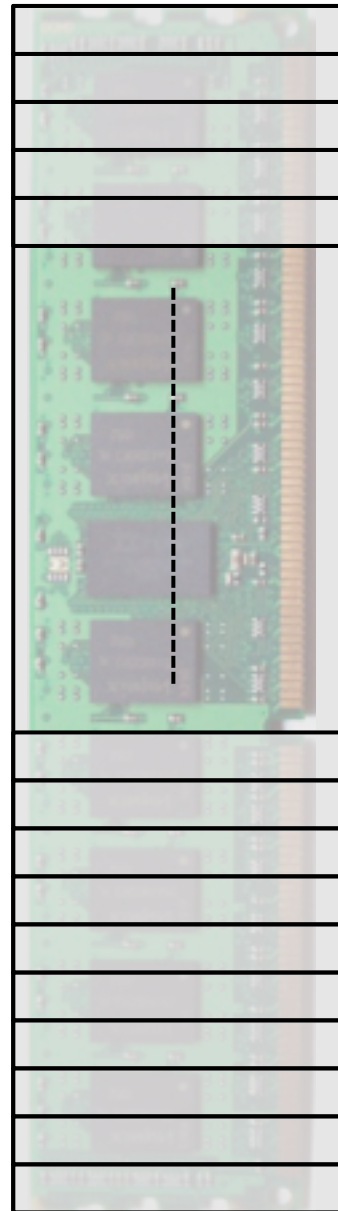
Hardware used for storing and retrieving stored data

Memory unit only sees a **stream of addresses + read requests, or address + data and write requests**



$2^m - 1$   
 $2^m - 2$   
 $2^m - 3$   
 $2^m - 4$   
 $2^m - 5$   
...

9  
8  
7  
6  
5  
4  
3  
2  
1  
0

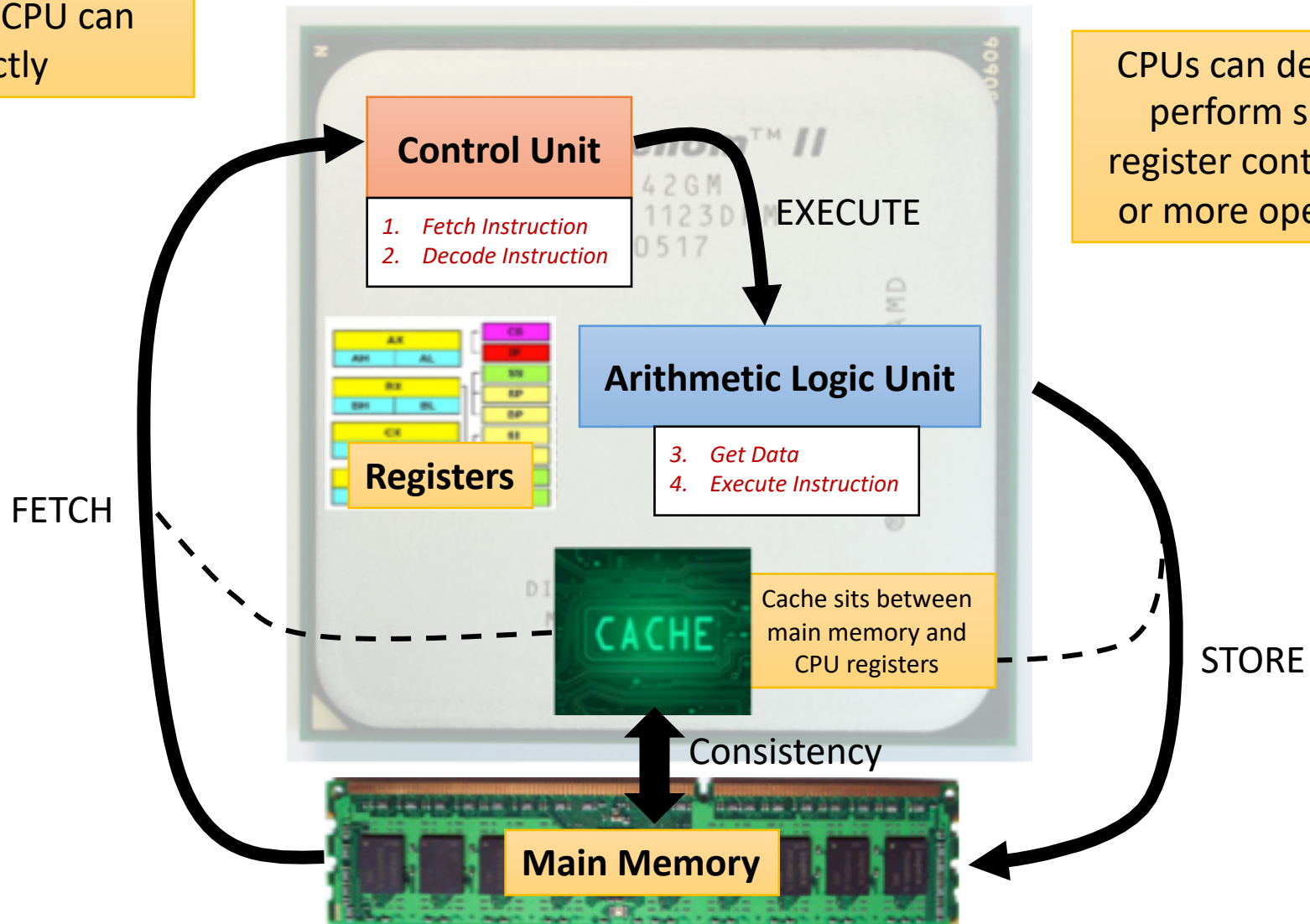


Main memory can take many CPU cycles, causing a **Memory Stall**

# Fetch – Execute Cycle

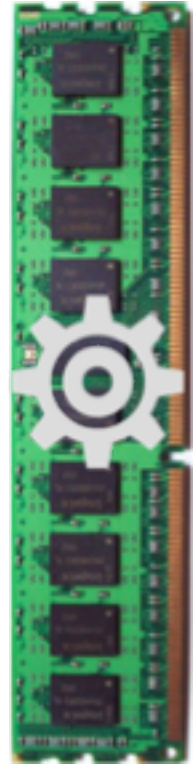
Main memory and registers are **only storage** CPU can access directly


CPUs can decode instructions and perform simple operations on register contents at the rate of one or more operations per clock tick.





Any program to run **must** be loaded in memory





```
// File: test.c
#include <stdio.h>
```

```
int main() {
    int x = 3000;
    x = x + 3;
    return 0;
}
```


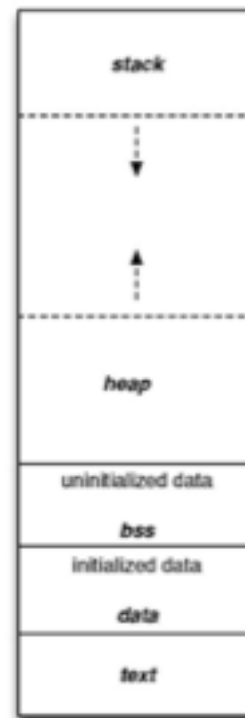
```
gcc -o test test.c
```



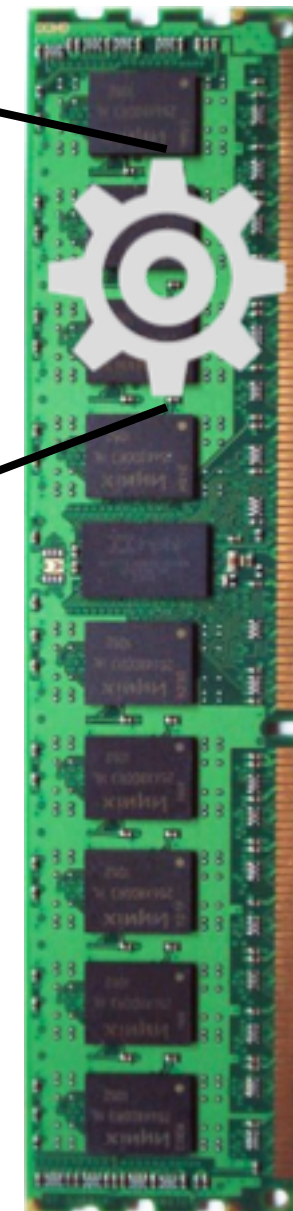
101  
011

Object file

Process Address Space



After **linking**, the OS loads  
the program into a  
process in memory



# Where should the OS load it?

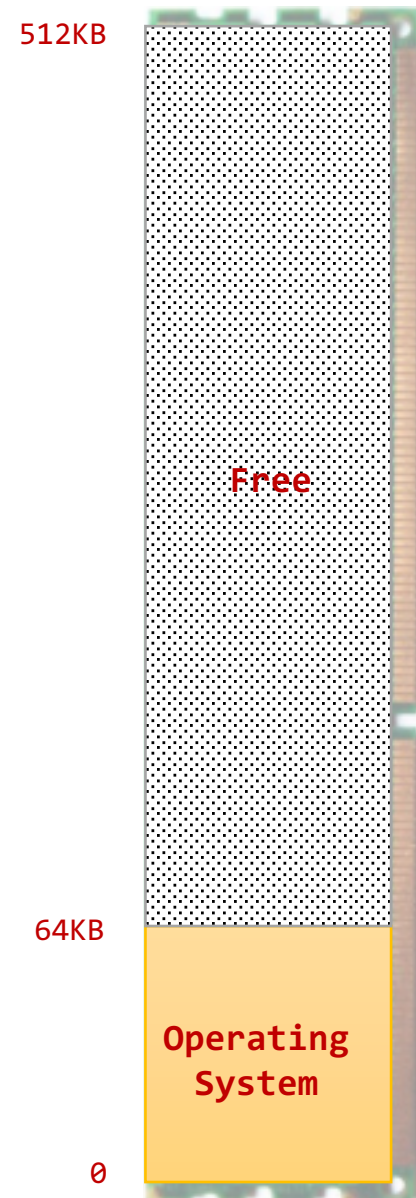


512KB

Free

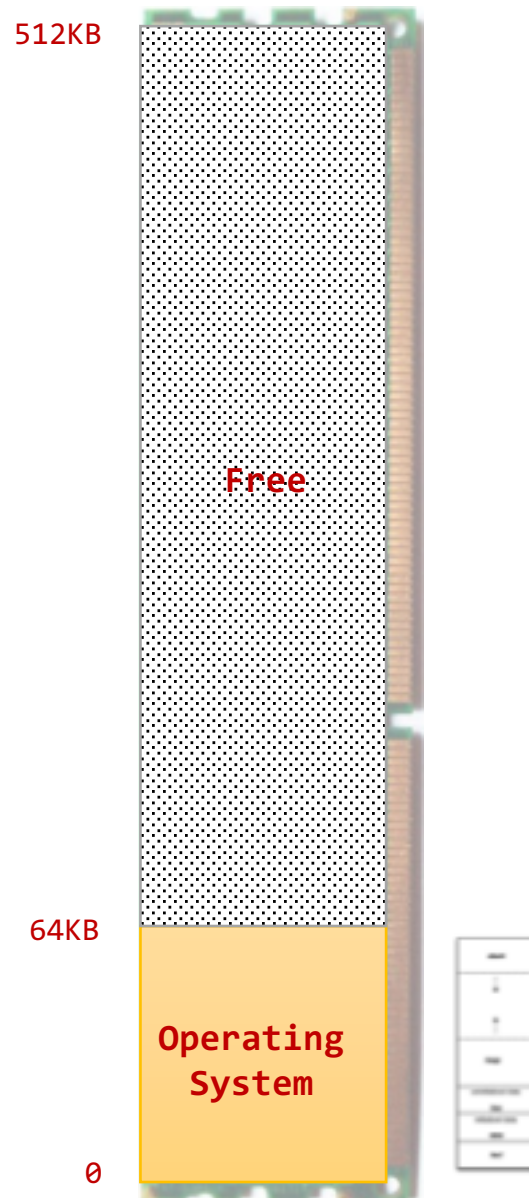
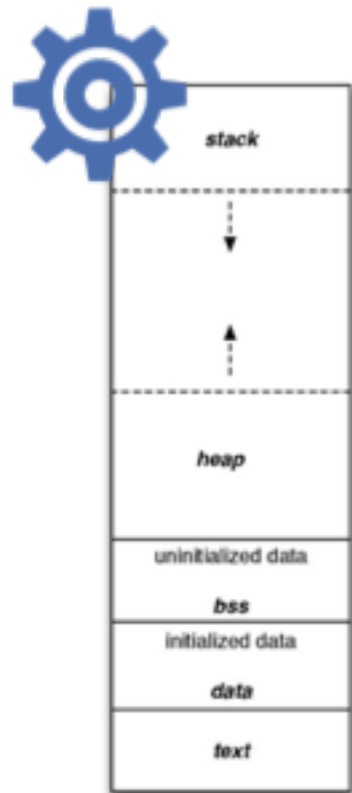
0



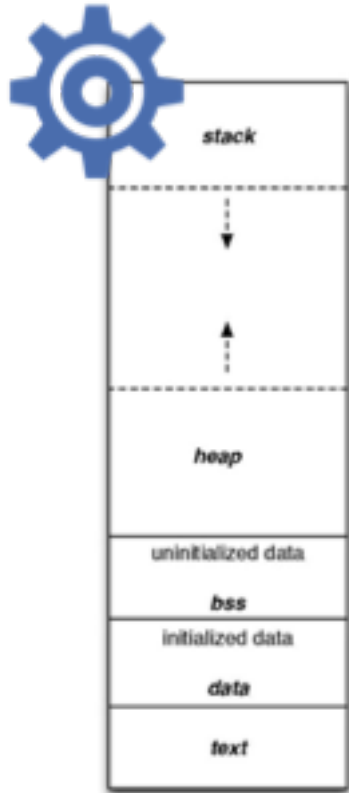


RAM
↓
↑
ROM
↓
ROM
↓
ROM

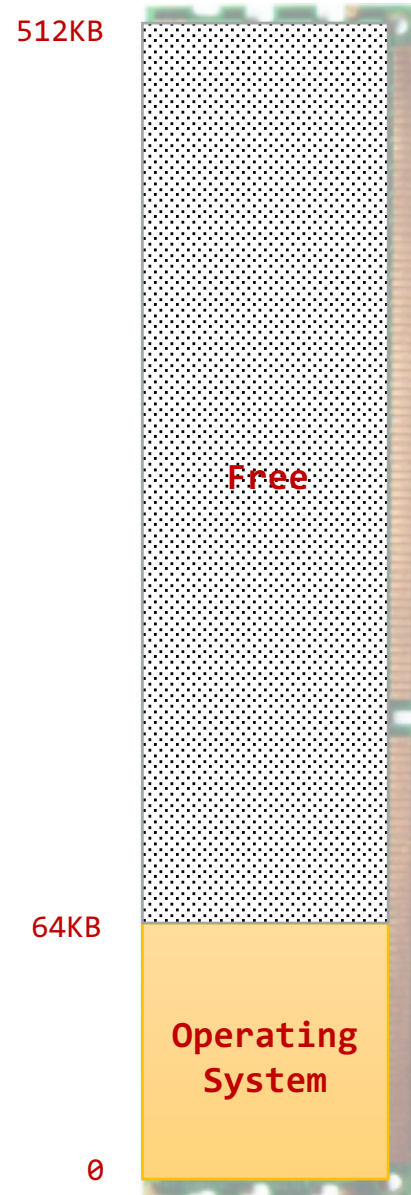
*OS occupies the lower set of addresses*



*OS occupies the lower set of addresses*

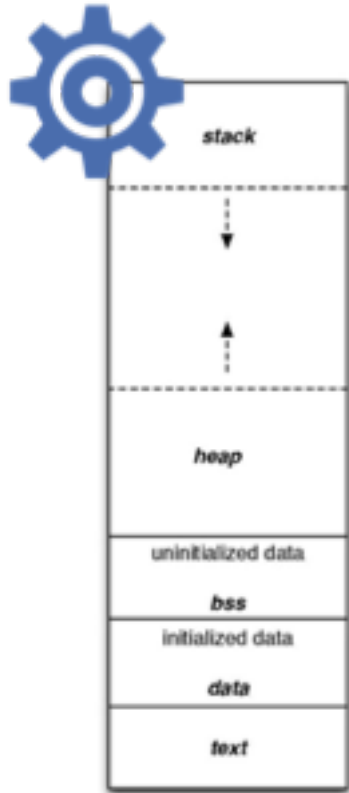


*Find the first free portion in the memory and put it there*



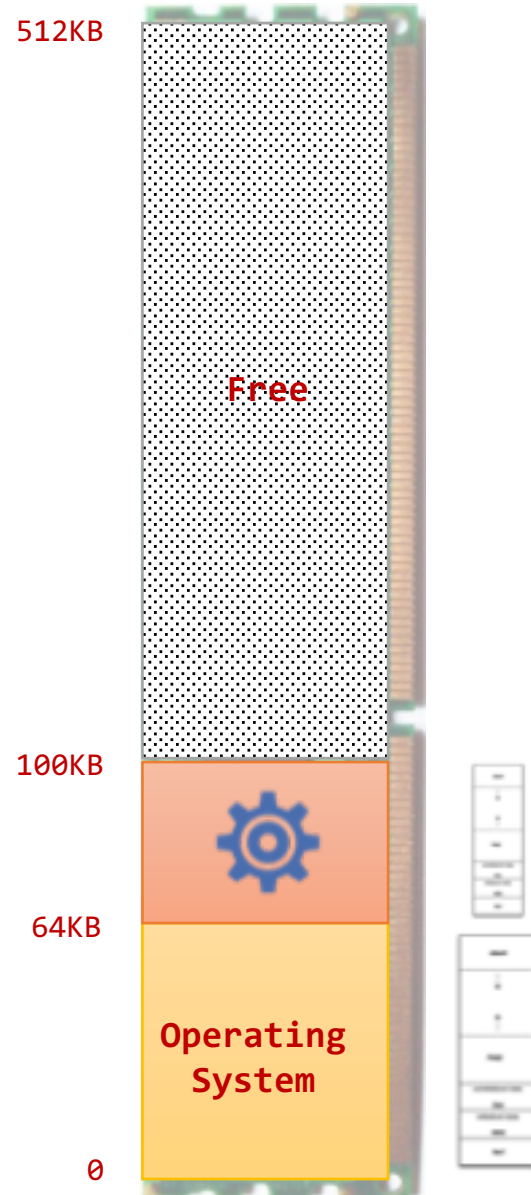
*OS occupies the lower set of addresses*



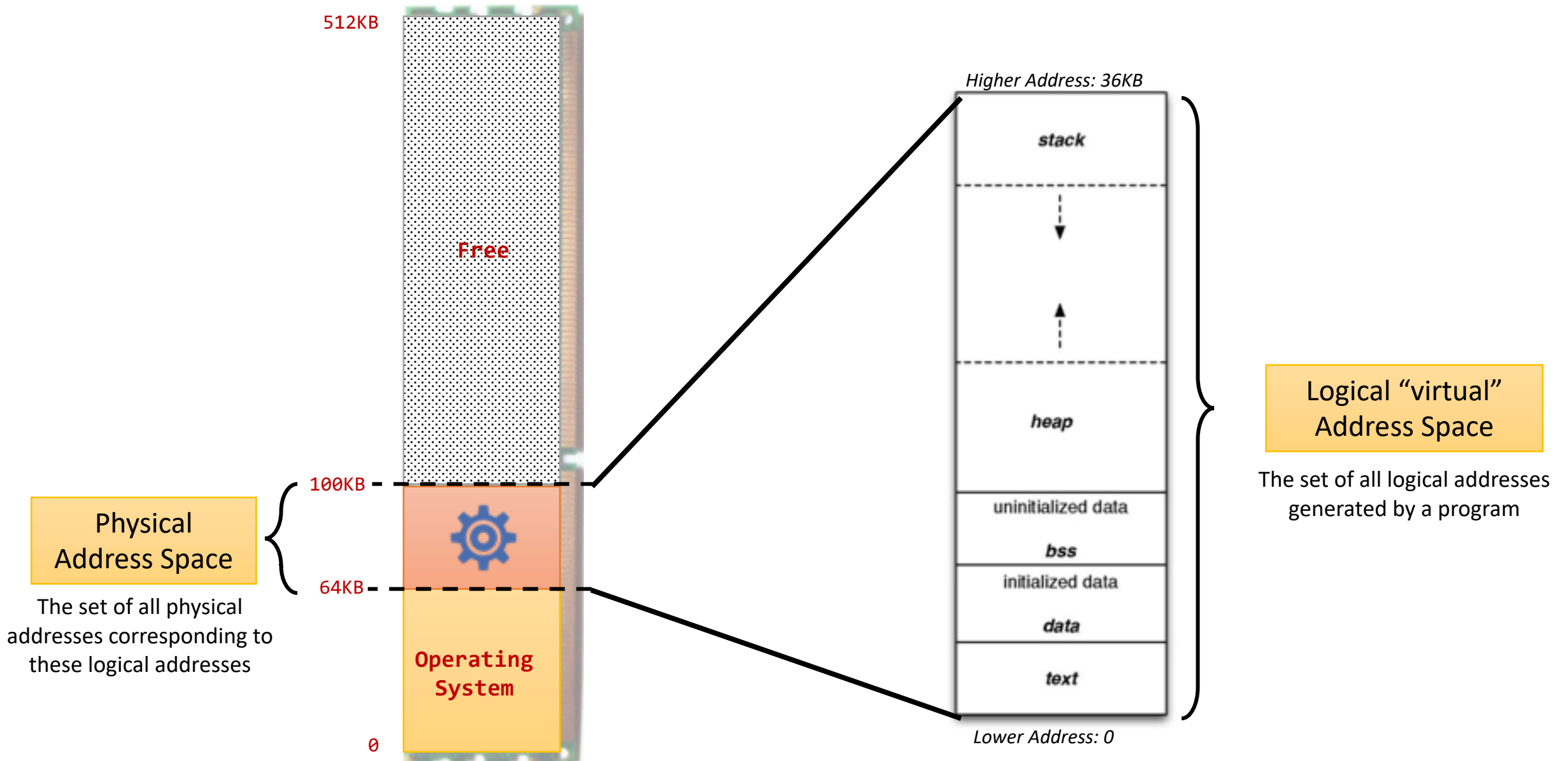



*Find the first free portion in the memory and put it there*

*There are many other strategies, we will discuss later*



# Process does not know about the memory





```
// File: test.c
#include <stdio.h>
```

```
int main() {
    int x = 3000;
    x = x + 3;
    return 0;
}
```

```
gcc -o test test.c
```



101  
011

Object file

```
objdump -d test
```

```
Disassembly of section __TEXT,__text:
__text:
100000f50: 55      pushq  %rbp
100000f51: 48 89 e5  movq  %rsp, %rbp
```

```
128: movl 0x0(%ebx), %eax ;load 0+ebx into eax
132: addl $0x03, %eax      ;add 3 to eax register
135: movl %eax, 0x0(%ebx) ;store eax back to mem
```

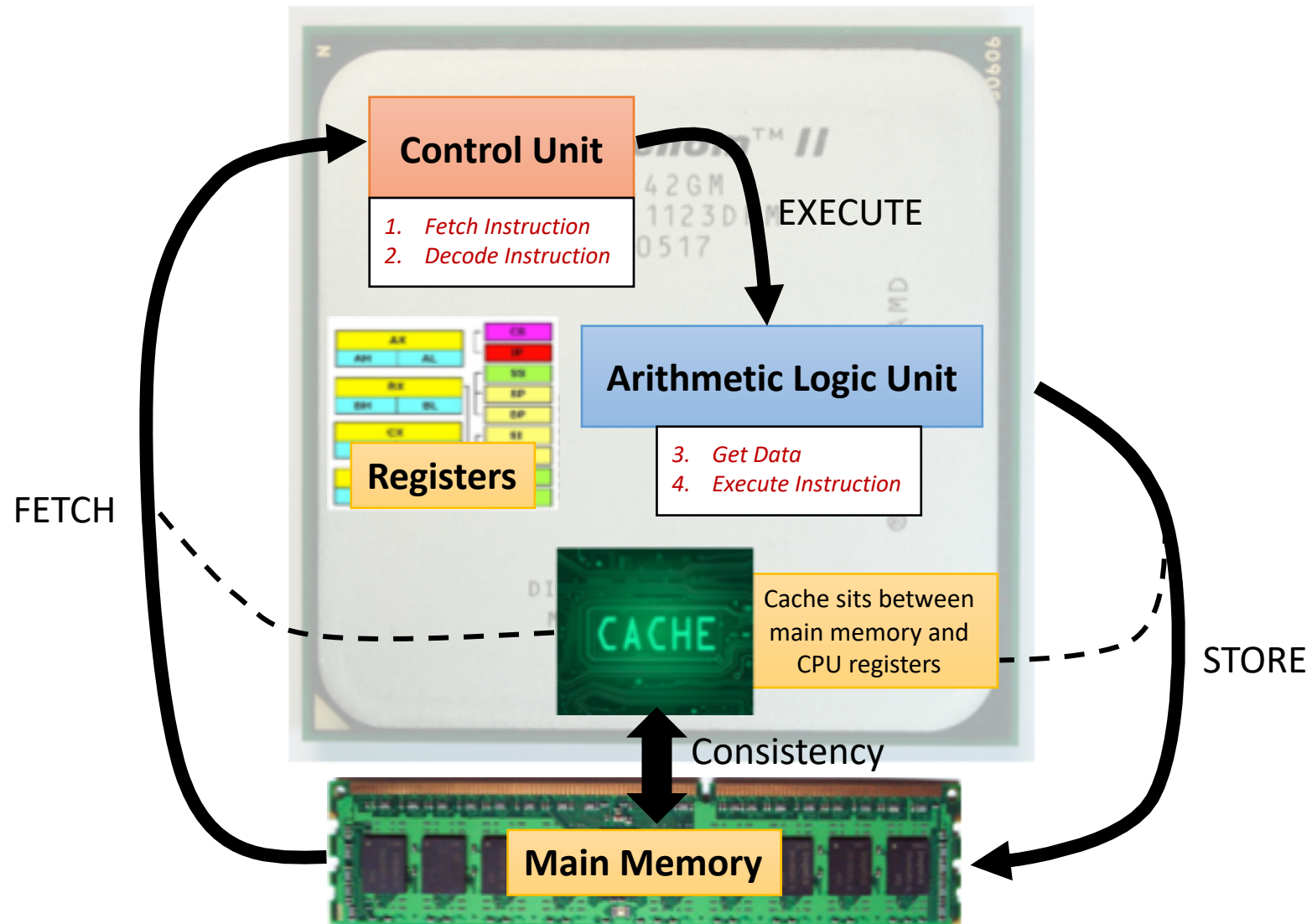
```
100000f74: 48 83 c4 18  subq  $16, %rsp
100000f78: 5d      popq   %rbp
100000f79: c3      retq

_main:
100000f50: 55      pushq  %rbp
100000f51: 48 89 e5  movq  %rsp, %rbp
100000f54: 48 83 ec 18  subq  $16, %rsp
100000f58: 48 8d 3d 3b 00 00 00  leaq  59(%rip), %rdi
100000f5f: c7 45 fc 00 00 00 00  movl  $0, -4(%rbp)
100000f66: b8 00 00 00 00  movb  $0, %al
100000f68: e8 0d 00 00 00  callq 13
100000f6d: 31 c9  xorl  %ecx, %ecx
100000f6f: 89 45 f8  movl  %eax, -8(%rbp)
100000f72: 89 c8  movl  %ecx, %eax
100000f74: 48 83 c4 18  addq  $16, %rsp
100000f78: 5d      popq   %rbp
100000f79: c3      retq

Disassembly of section __TEXT,__stubs:
__stubs:
100000f7a: ff 25 98 00 00 00  jmpq  *144(%rip)
Disassembly of section __TEXT,__stub_helper:
__stub_helper:
100000f80: 4c 8d 1d 81 00 00 00  leaq  129(%rip), %r11
100000f87: 41 53  pushq  %r11
100000f89: ff 25 71 00 00 00  jmpq  *113(%rip)
100000f8f: 90      nop
100000f90: 66 00 00 00 00  pushq  $0
100000f95: e9 e6 ff ff  jmp  -26 <__stub_helper>
```

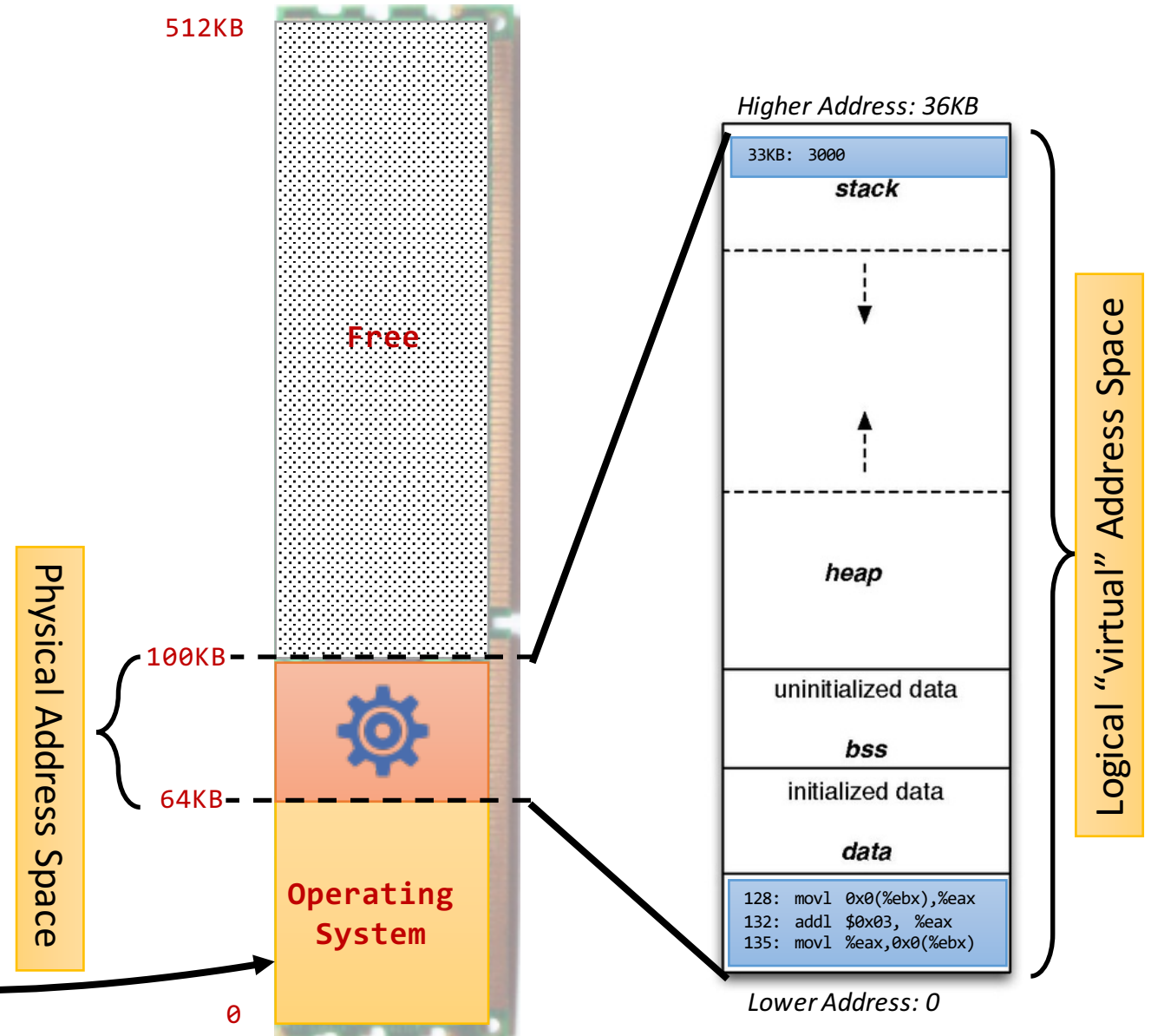
[http://www.thegeekstuff.com/2012/09/objdump-examples/?utm\\_source=feedburner](http://www.thegeekstuff.com/2012/09/objdump-examples/?utm_source=feedburner)  
<https://jvns.ca/blog/2014/09/06/how-to-read-an-executable/>

# Fetch – Execute Cycle



```
128: movl 0x0(%ebx),%eax
132: addl $0x03, %eax
135: movl %eax,0x0(%ebx)
```

Fetch instruction at address 128 from memory



```
128: movl 0x0(%ebx),%eax
132: addl $0x03, %eax
135: movl %eax,0x0(%ebx)
```

Fetch instruction at address 128 from memory



Physical Address Space

512KB

Free

100KB

64KB

Operating System

0

Higher Address: 36KB

33KB: 3000

**stack**



**heap**

uninitialized data

**bss**

initialized data

**data**

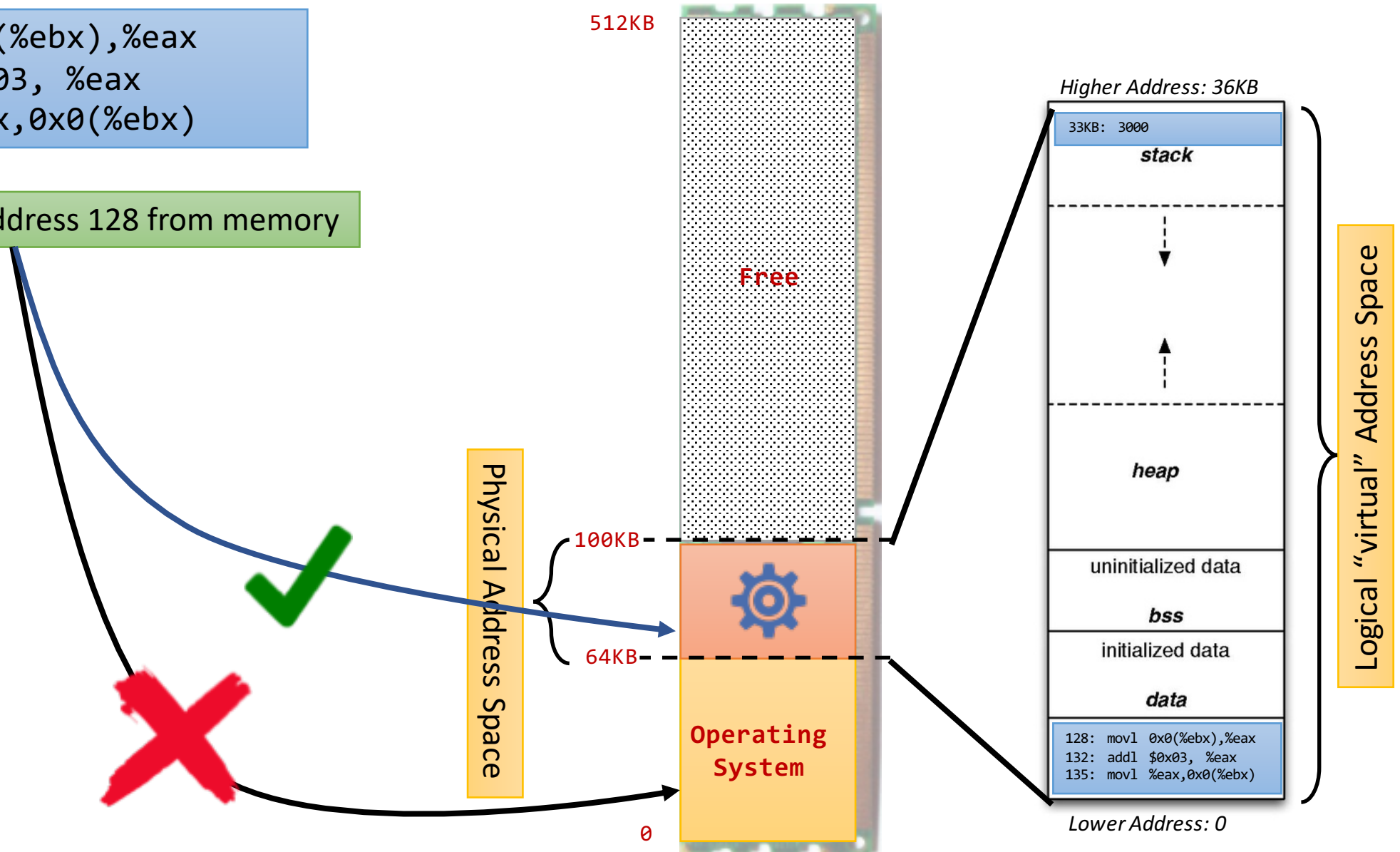
```
128: movl 0x0(%ebx),%eax
132: addl $0x03, %eax
135: movl %eax,0x0(%ebx)
```

Lower Address: 0

Logical "virtual" Address Space

```
128: movl 0x0(%ebx),%eax
132: addl $0x03, %eax
135: movl %eax,0x0(%ebx)
```

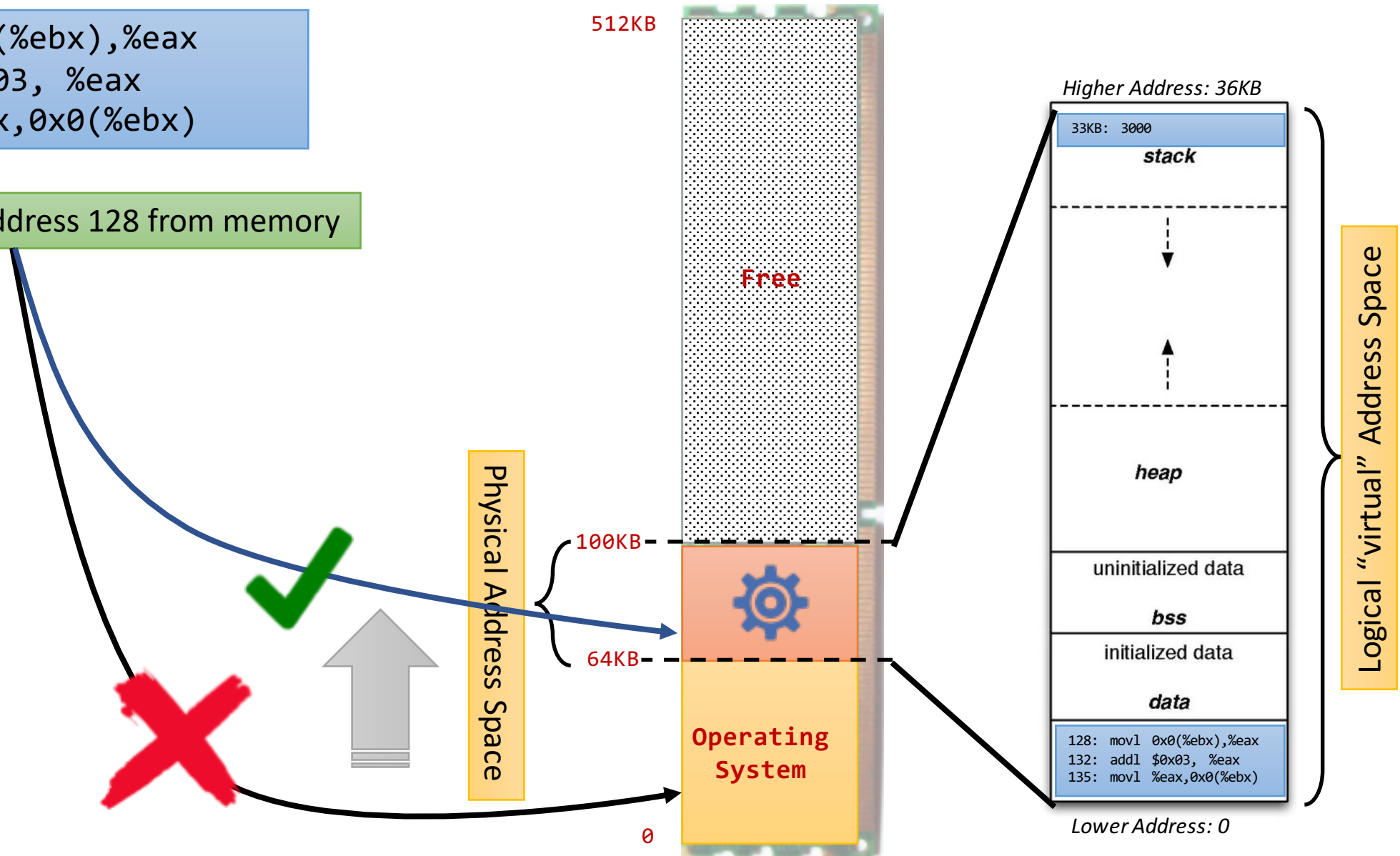
Fetch instruction at address 128 from memory





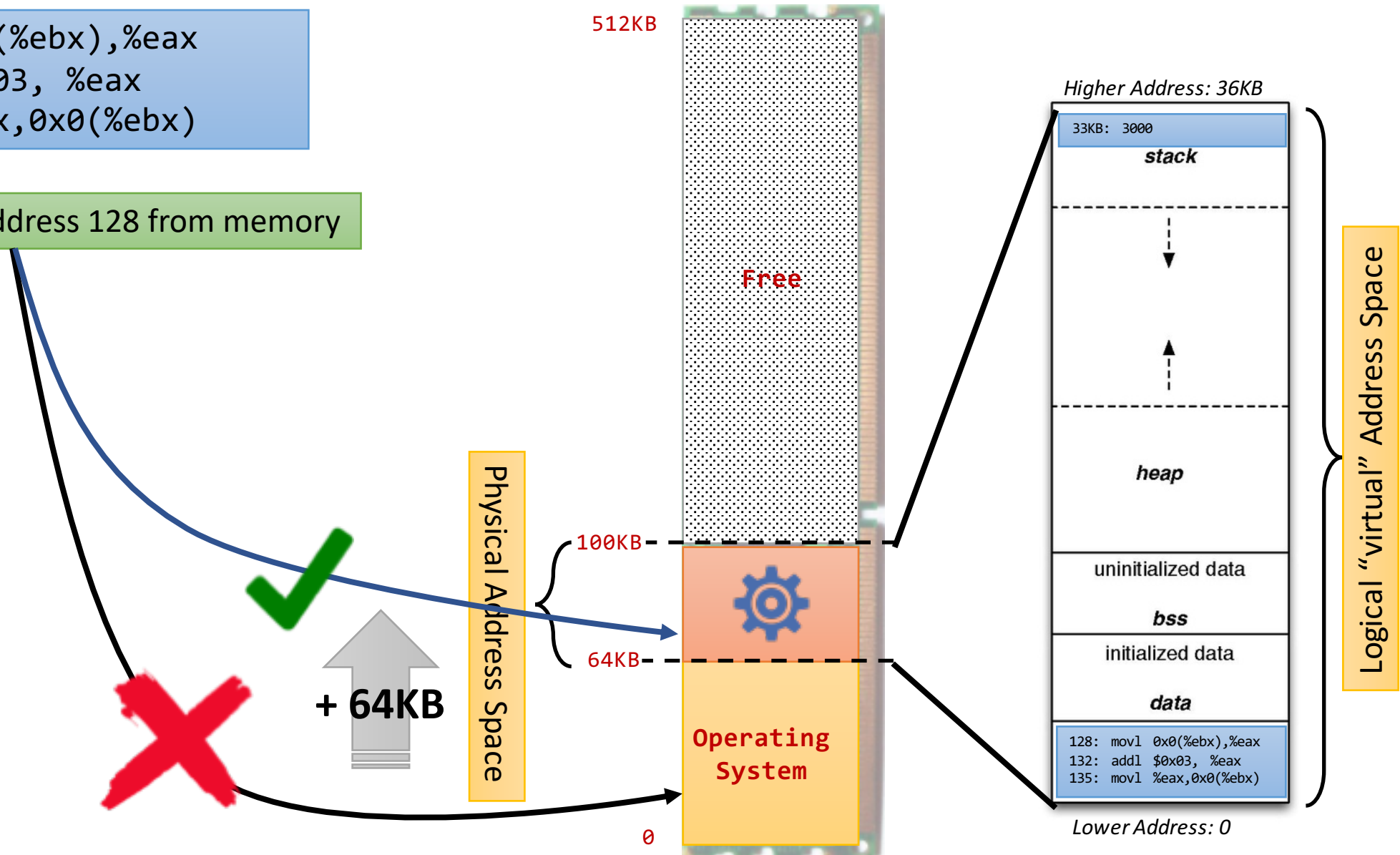
```
128: movl 0x0(%ebx),%eax
132: addl $0x03, %eax
135: movl %eax,0x0(%ebx)
```

Fetch instruction at address 128 from memory



```
128: movl 0x0(%ebx),%eax
132: addl $0x03, %eax
135: movl %eax,0x0(%ebx)
```

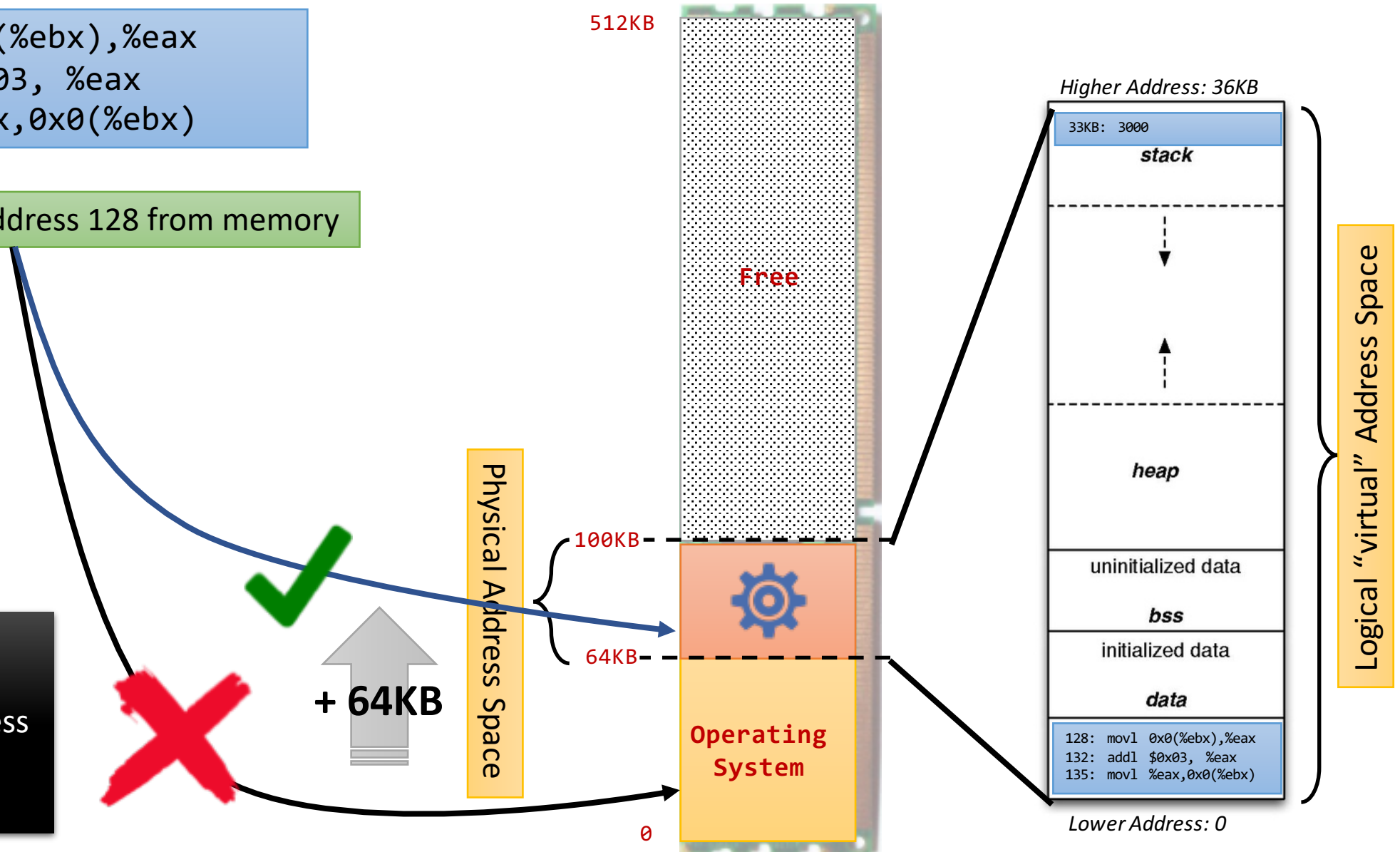
Fetch instruction at address 128 from memory



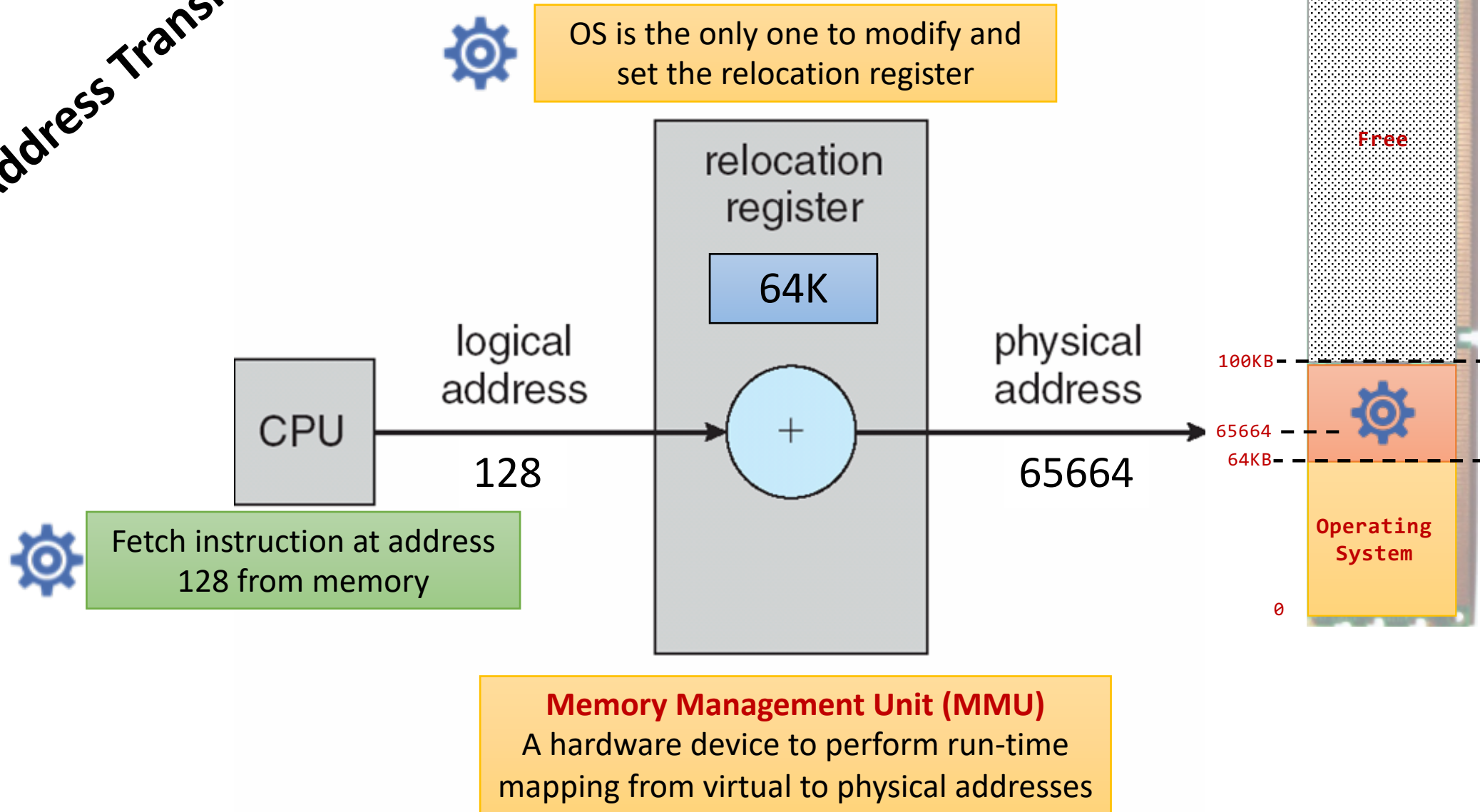
```
128: movl 0x0(%ebx),%eax
132: addl $0x03, %eax
135: movl %eax,0x0(%ebx)
```

Fetch instruction at address 128 from memory

Physical Address  
=  
Logical "Virtual" Address  
+  
Relocation Register



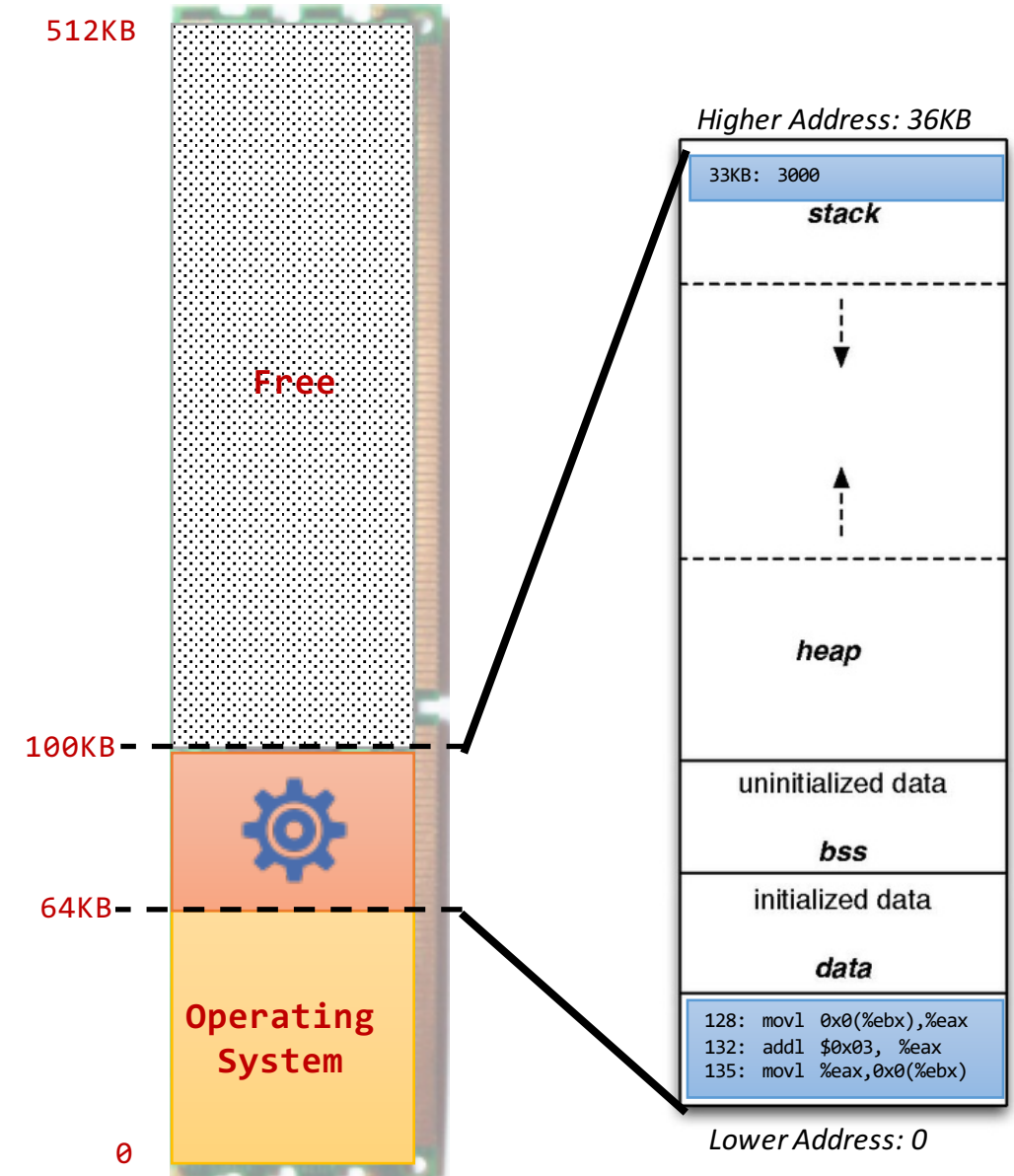
# Address Translation



## Registers

eax	ebx
0	33KB

```
128: movl 0x0(%ebx),%eax
132: addl $0x03, %eax
135: movl %eax,0x0(%ebx)
```

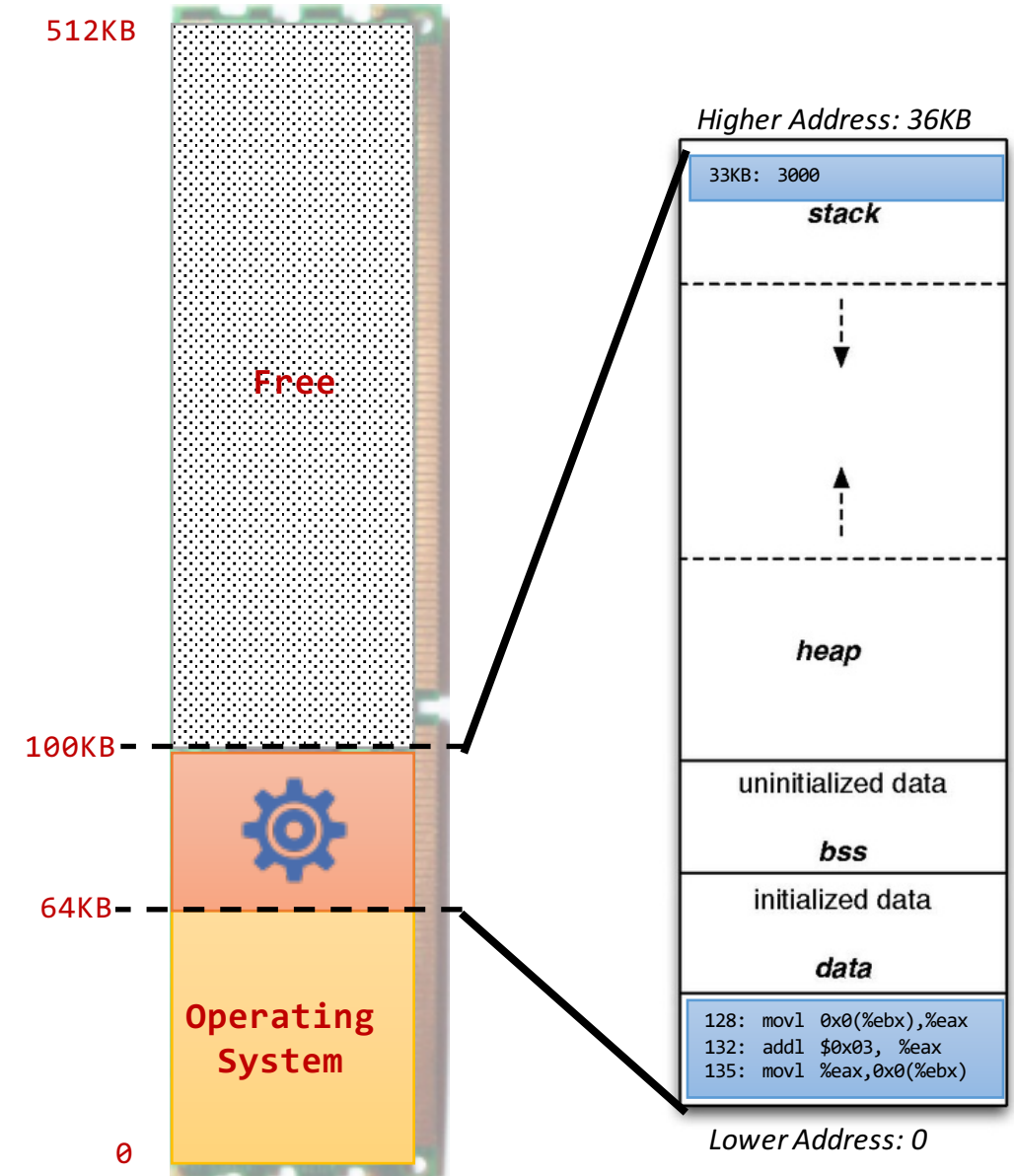


## Registers

eax	ebx
0	33KB

```
128: movl 0x0(%ebx),%eax
132: addl $0x03, %eax
135: movl %eax,0x0(%ebx)
```

Fetch instruction at address 128



## Registers

eax	ebx
0	33KB

```
128: movl 0x0(%ebx),%eax
132: addl $0x03, %eax
135: movl %eax,0x0(%ebx)
```

Fetch instruction at address 128

128

MMU

+ 64KB

65664

512KB

Free

100KB

64KB

Operating  
System

0

Higher Address: 36KB

33KB: 3000

*stack*



*heap*

uninitialized data

*bss*

initialized data

*data*

```
128: movl 0x0(%ebx),%eax
132: addl $0x03, %eax
135: movl %eax,0x0(%ebx)
```

Lower Address: 0



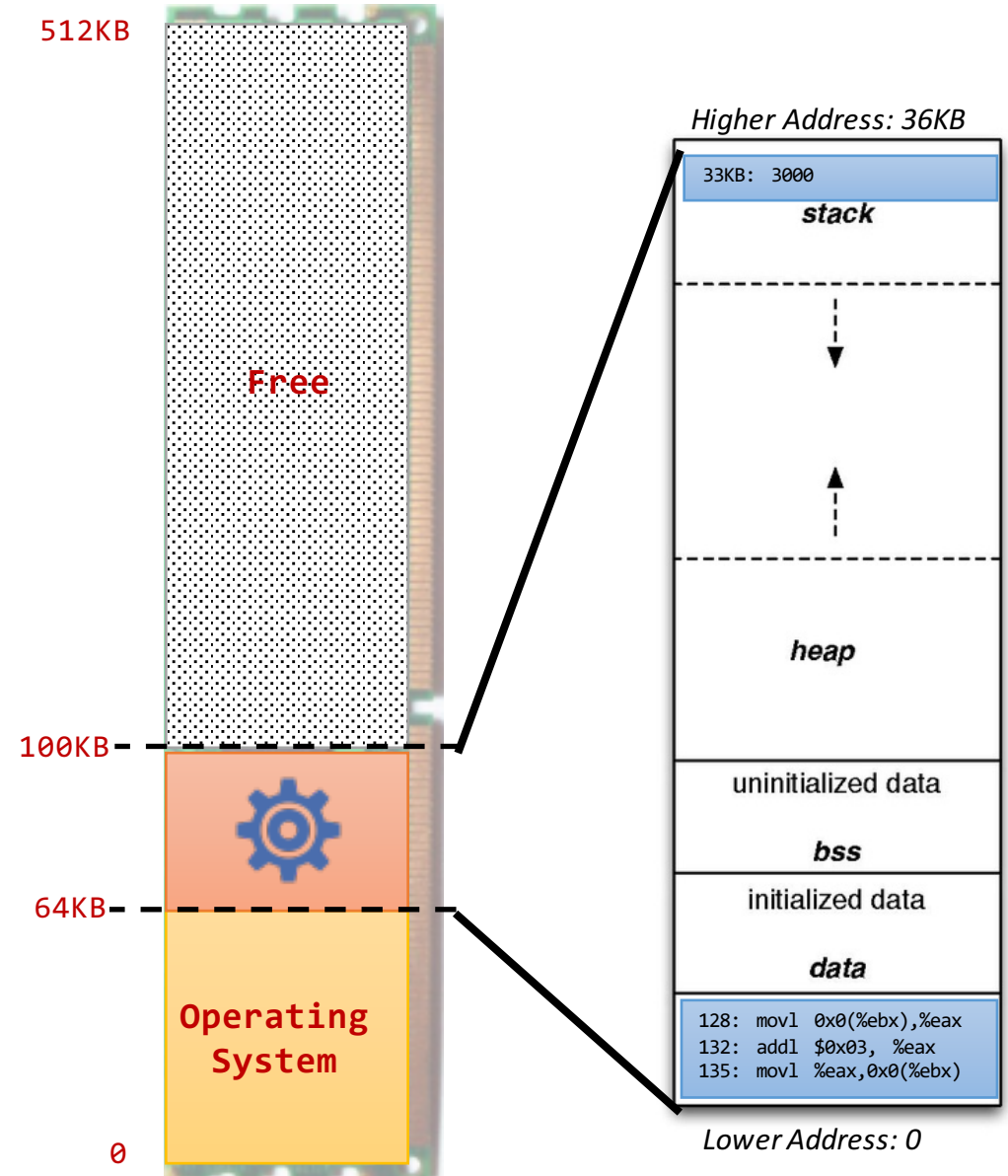
## Registers

eax	ebx
0	33KB

```
128: movl 0x0(%ebx),%eax
132: addl $0x03, %eax
135: movl %eax,0x0(%ebx)
```

Fetch instruction at address 128

Execute this instruction (load from address 33 KB)



## Registers

eax	ebx
3000	33KB

```
128: movl 0x0(%ebx),%eax
132: addl $0x03, %eax
135: movl %eax,0x0(%ebx)
```

Fetch instruction at address 128

Execute this instruction (load from address 33 KB)

33KB

MMU  
+ 64KB

96KB

100KB

64KB

Operating  
System

0

512KB

Free

Higher Address: 36KB

33KB: 3000

*stack*



*heap*

uninitialized data

*bss*

initialized data

*data*

```
128: movl 0x0(%ebx),%eax
132: addl $0x03, %eax
135: movl %eax,0x0(%ebx)
```

Lower Address: 0

## Registers

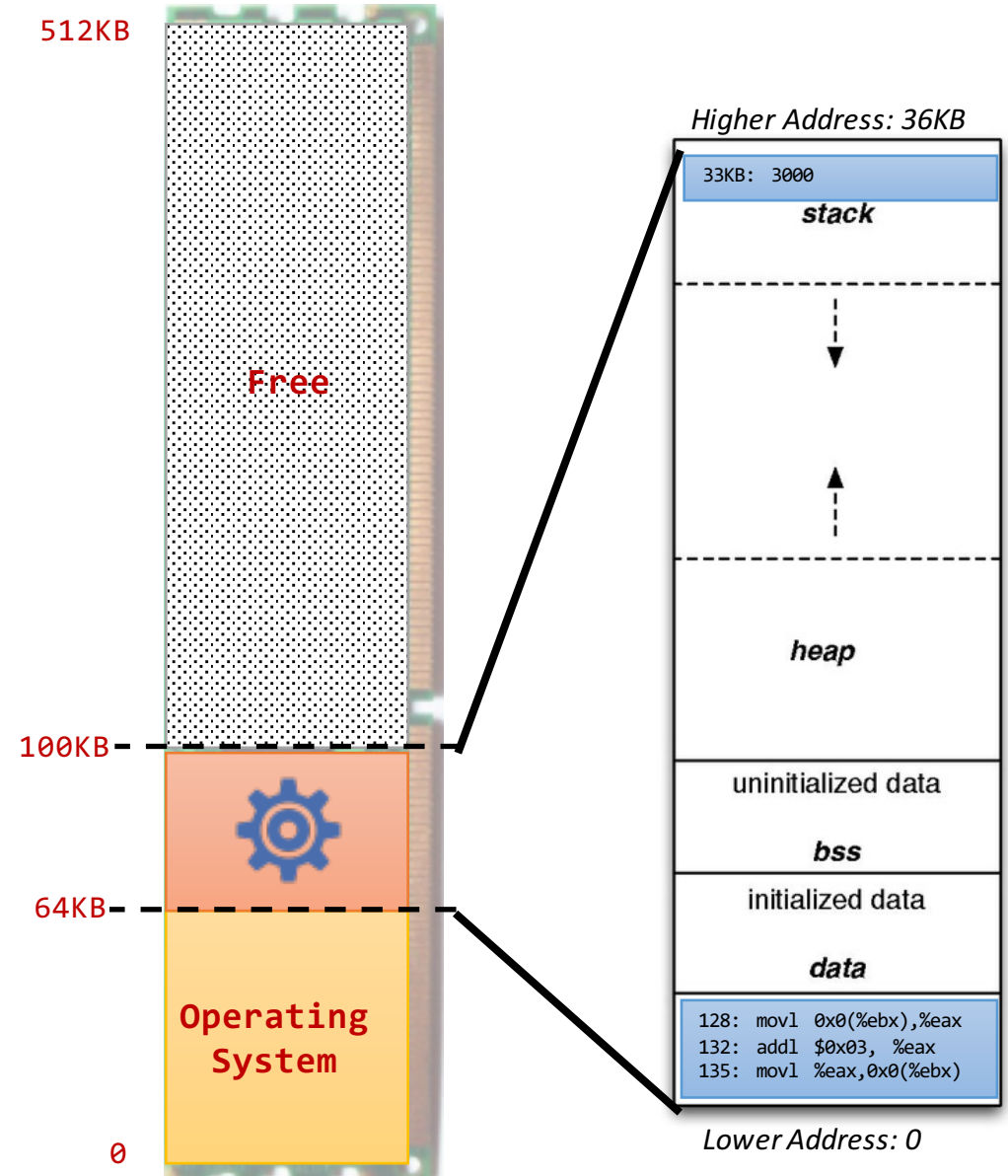
eax	ebx
3000	33KB

```
128: movl 0x0(%ebx),%eax
132: addl $0x03, %eax
135: movl %eax,0x0(%ebx)
```

Fetch instruction at address 128

Execute this instruction (load from address 15 KB)

Fetch instruction at address 132



## Registers

eax	ebx
3000	33KB

```
128: movl 0x0(%ebx),%eax
132: addl $0x03, %eax
135: movl %eax,0x0(%ebx)
```

Fetch instruction at address 128

Execute this instruction (load from address 15 KB)

Fetch instruction at address 132

132

MMU

+ 64KB

65668

64KB

100KB

512KB

0



Operating  
System

Higher Address: 36KB

33KB: 3000

*stack*



*heap*

uninitialized data

*bss*

initialized data

*data*

```
128: movl 0x0(%ebx),%eax
132: addl $0x03, %eax
135: movl %eax,0x0(%ebx)
```

Lower Address: 0

## Registers

eax	ebx
3003	33KB

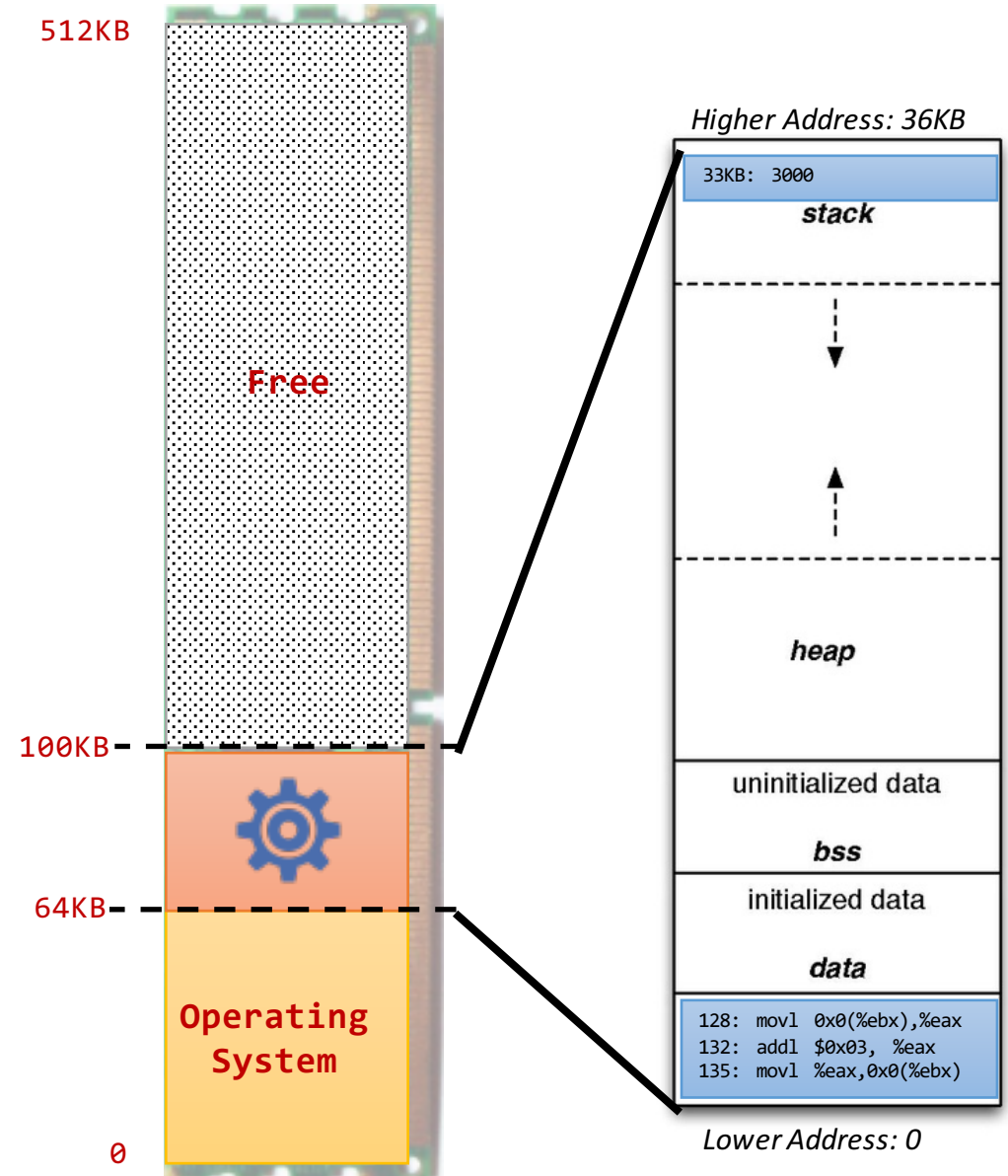
```
128: movl 0x0(%ebx),%eax
132: addl $0x03, %eax
135: movl %eax,0x0(%ebx)
```

Fetch instruction at address 128

Execute this instruction (load from address 15 KB)

Fetch instruction at address 132

Execute this instruction (no memory reference)



## Registers

eax	ebx
3003	33KB

```
128: movl 0x0(%ebx),%eax
132: addl $0x03, %eax
135: movl %eax,0x0(%ebx)
```

Fetch instruction at address 128

Execute this instruction (load from address 15 KB)

Fetch instruction at address 132

Execute this instruction (no memory reference)

Fetch the instruction at address 135

MMU

+ 64KB

135

512KB

Free

100KB

64KB

65671

0

Operating System

Higher Address: 36KB

33KB: 3000

stack

heap

uninitialized data

bss

initialized data

data

```
128: movl 0x0(%ebx),%eax
132: addl $0x03, %eax
135: movl %eax,0x0(%ebx)
```

Lower Address: 0

## Registers

eax	ebx
3003	33KB

```
128: movl 0x0(%ebx),%eax
132: addl $0x03, %eax
135: movl %eax,0x0(%ebx)
```

## Fetch instruction at address 128

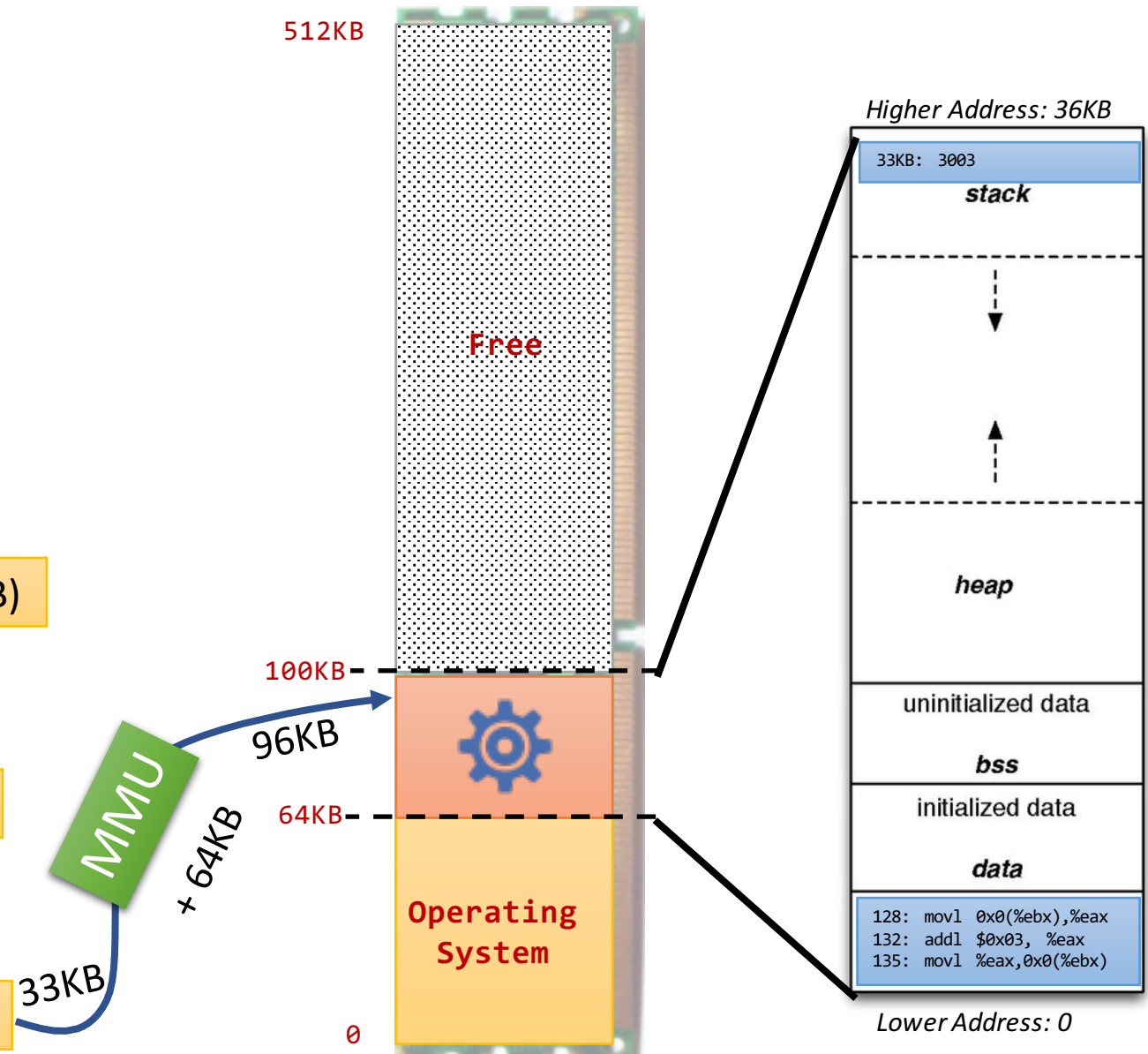
Execute this instruction (load from address 33 KB)

## Fetch instruction at address 132

Execute this instruction (no memory reference)

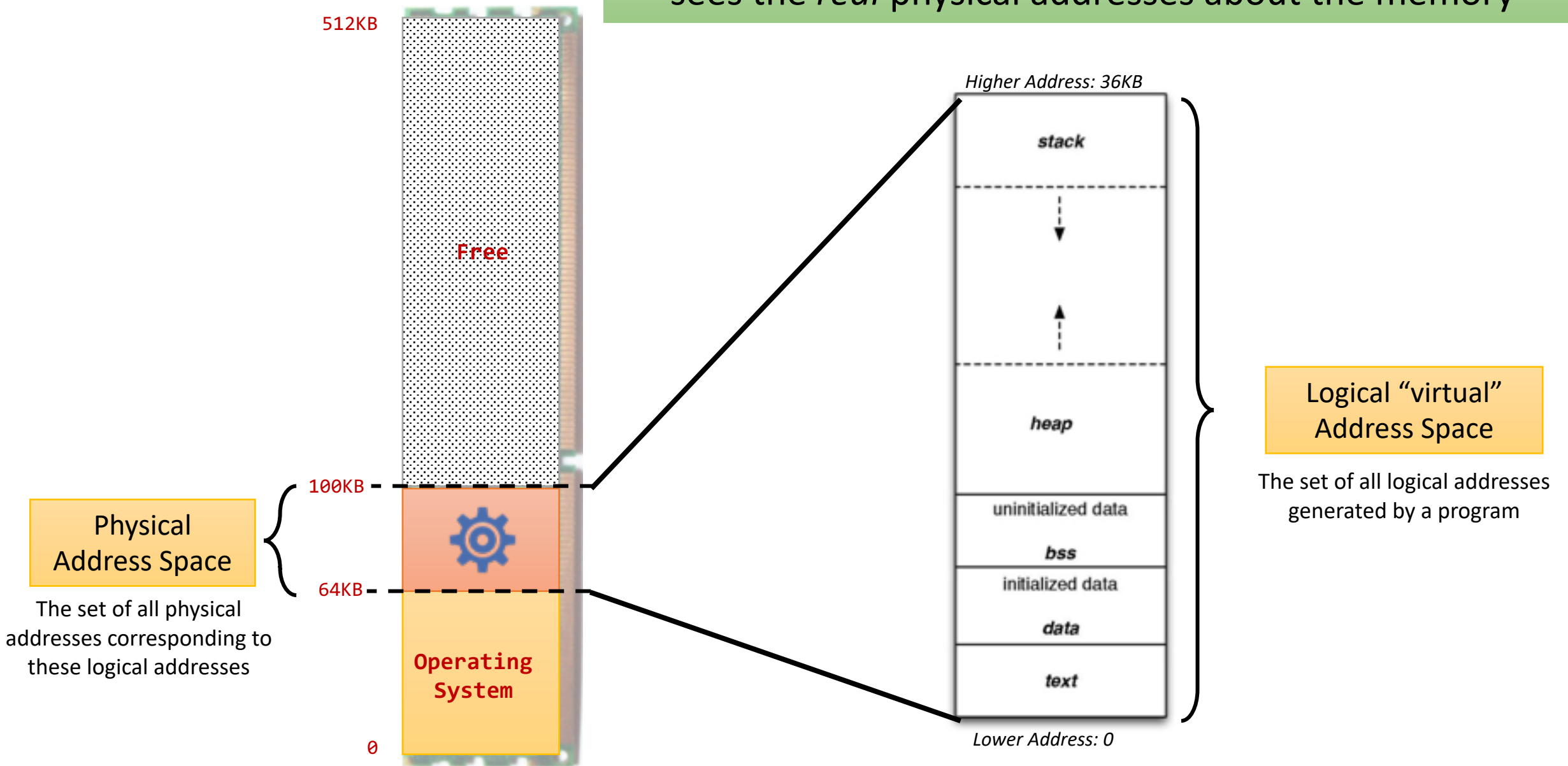
## Fetch the instruction at address 135

Execute this instruction (store to address 33 KB)

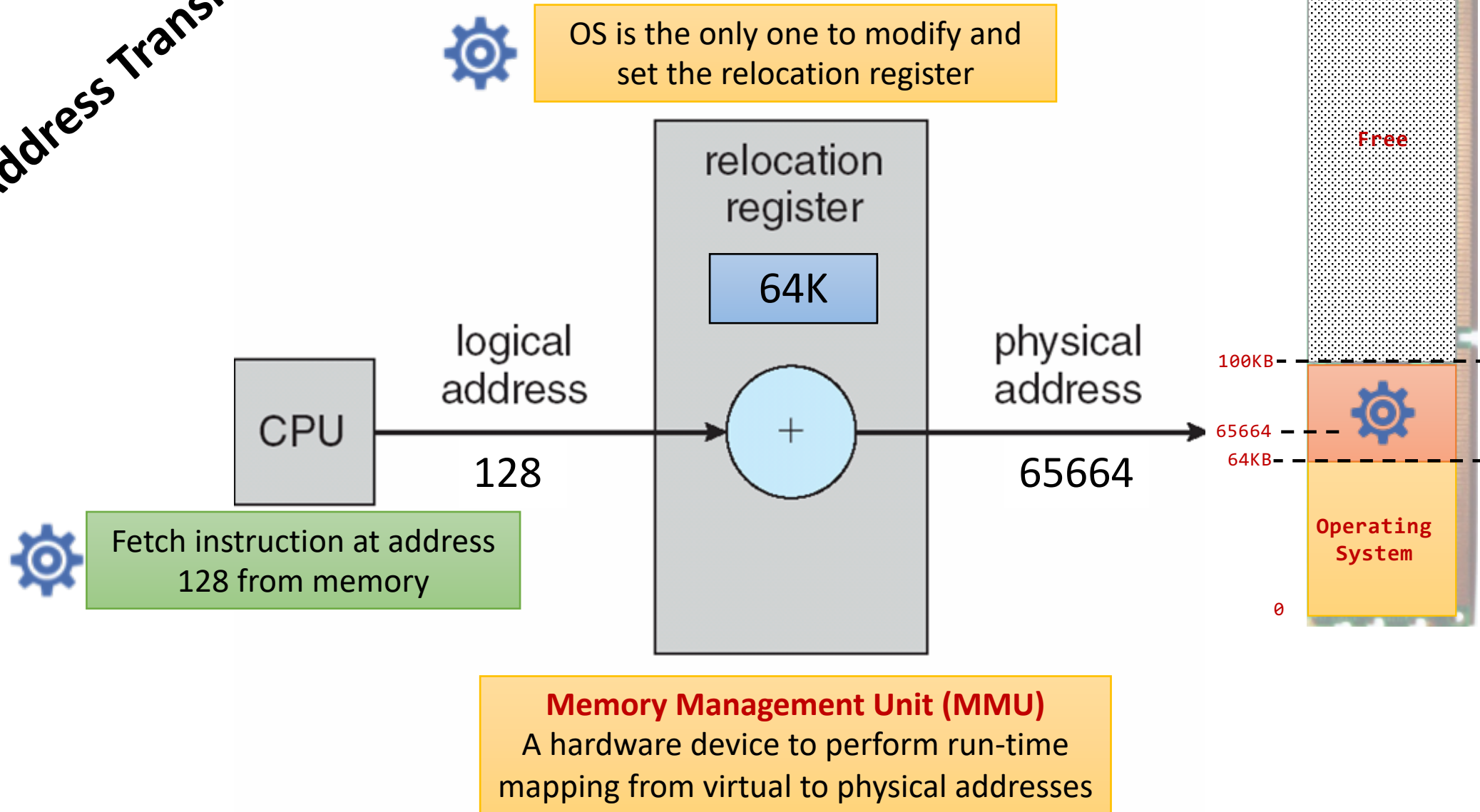




The user program deals with *logical* addresses; it never sees the *real* physical addresses about the memory

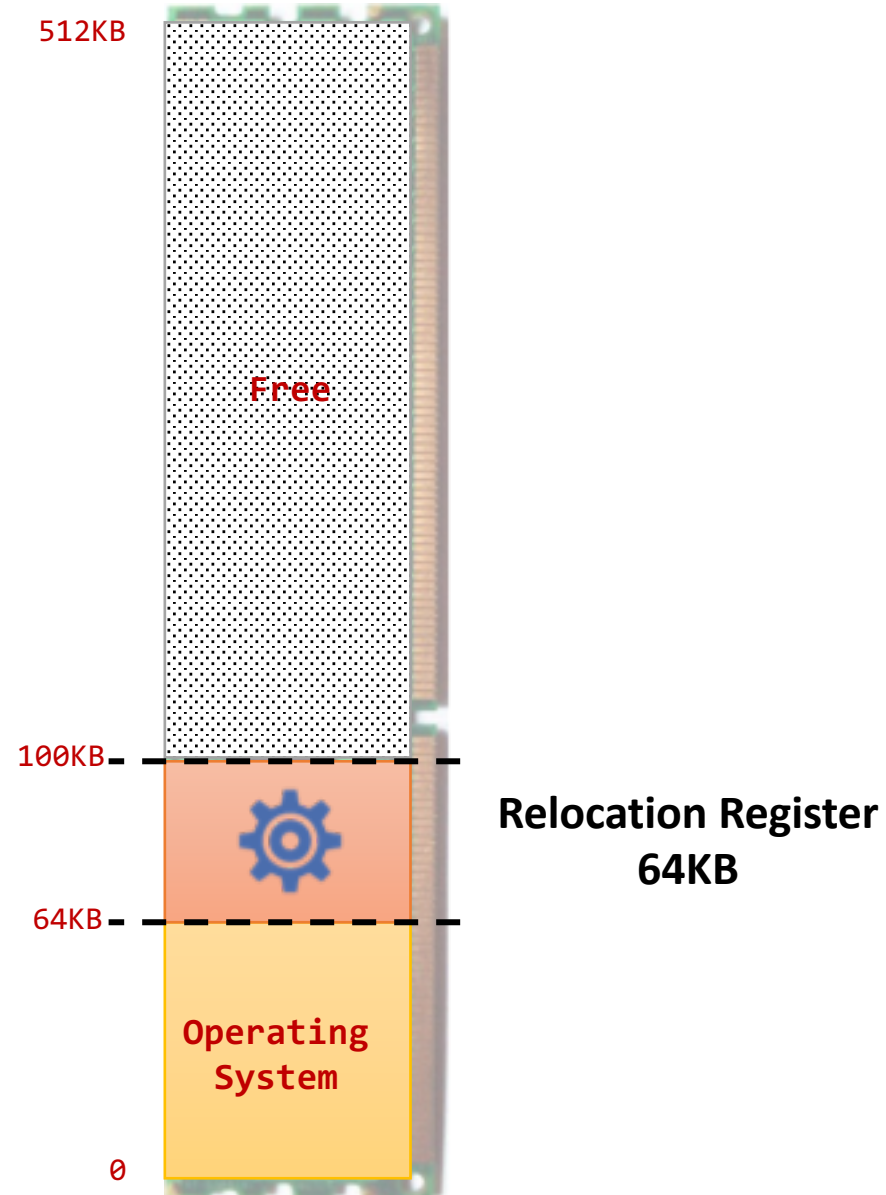


# Address Translation



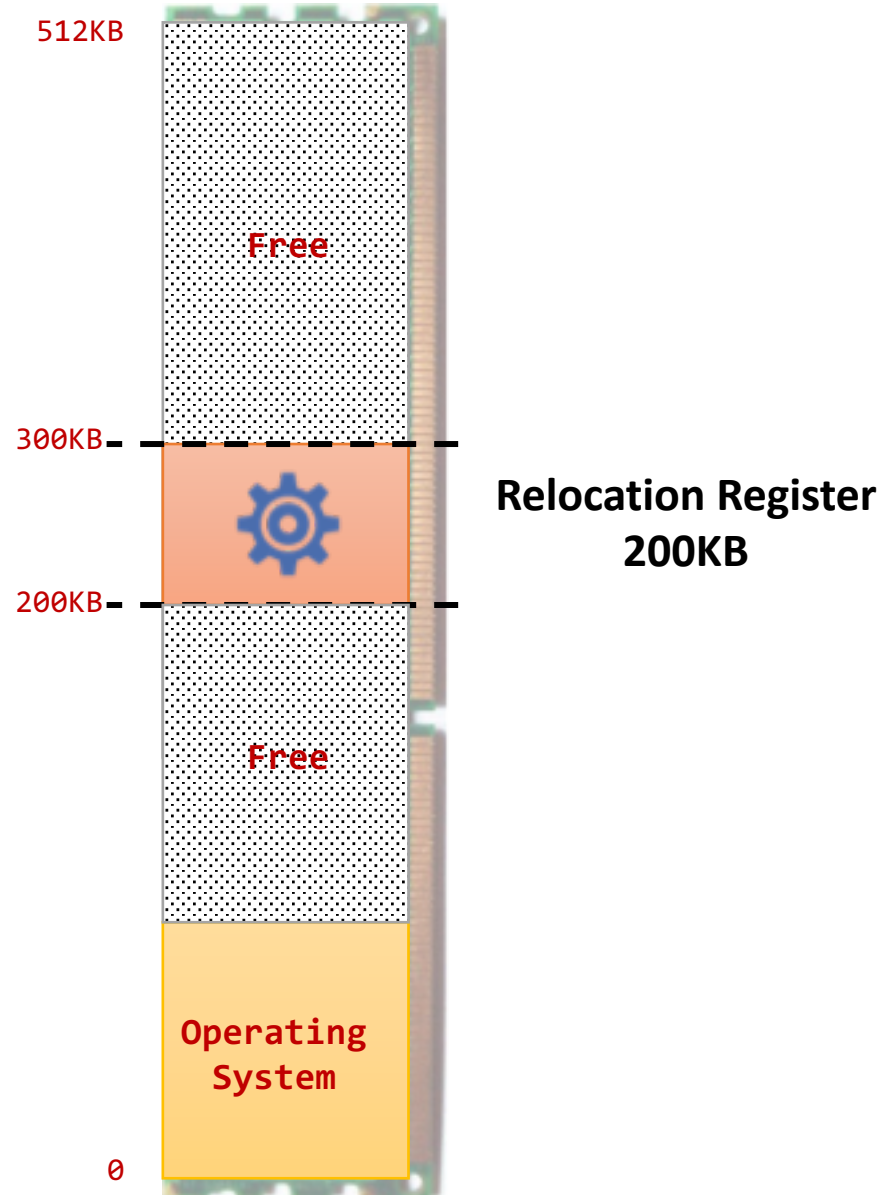
The relocation register enables the OS to simply move the process anywhere in the memory by changing the relocation register

Can then allow actions such as kernel code being transient and kernel changing size



The relocation register enables the OS to simply move the process anywhere in the memory by changing the relocation register

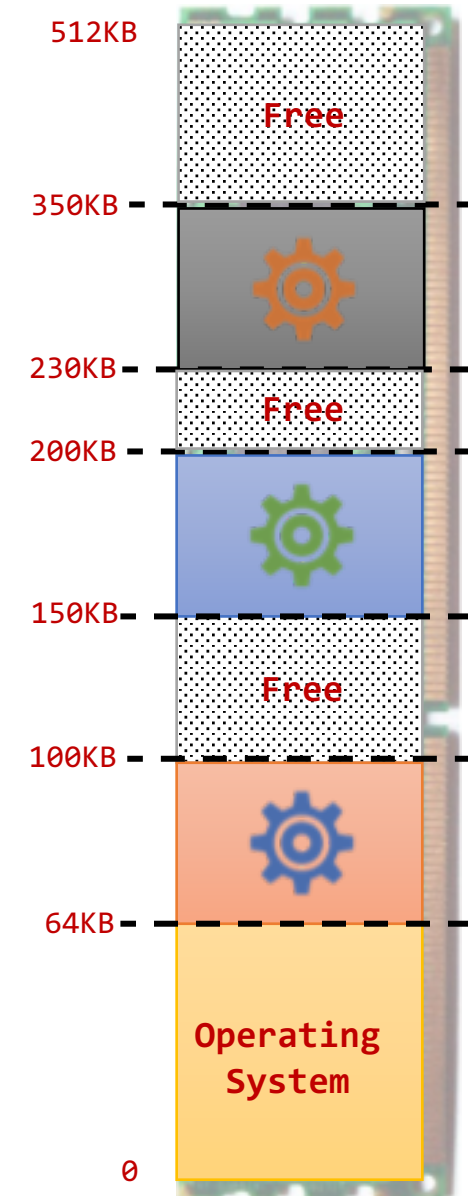
Can then allow actions such as kernel code being transient and kernel changing size





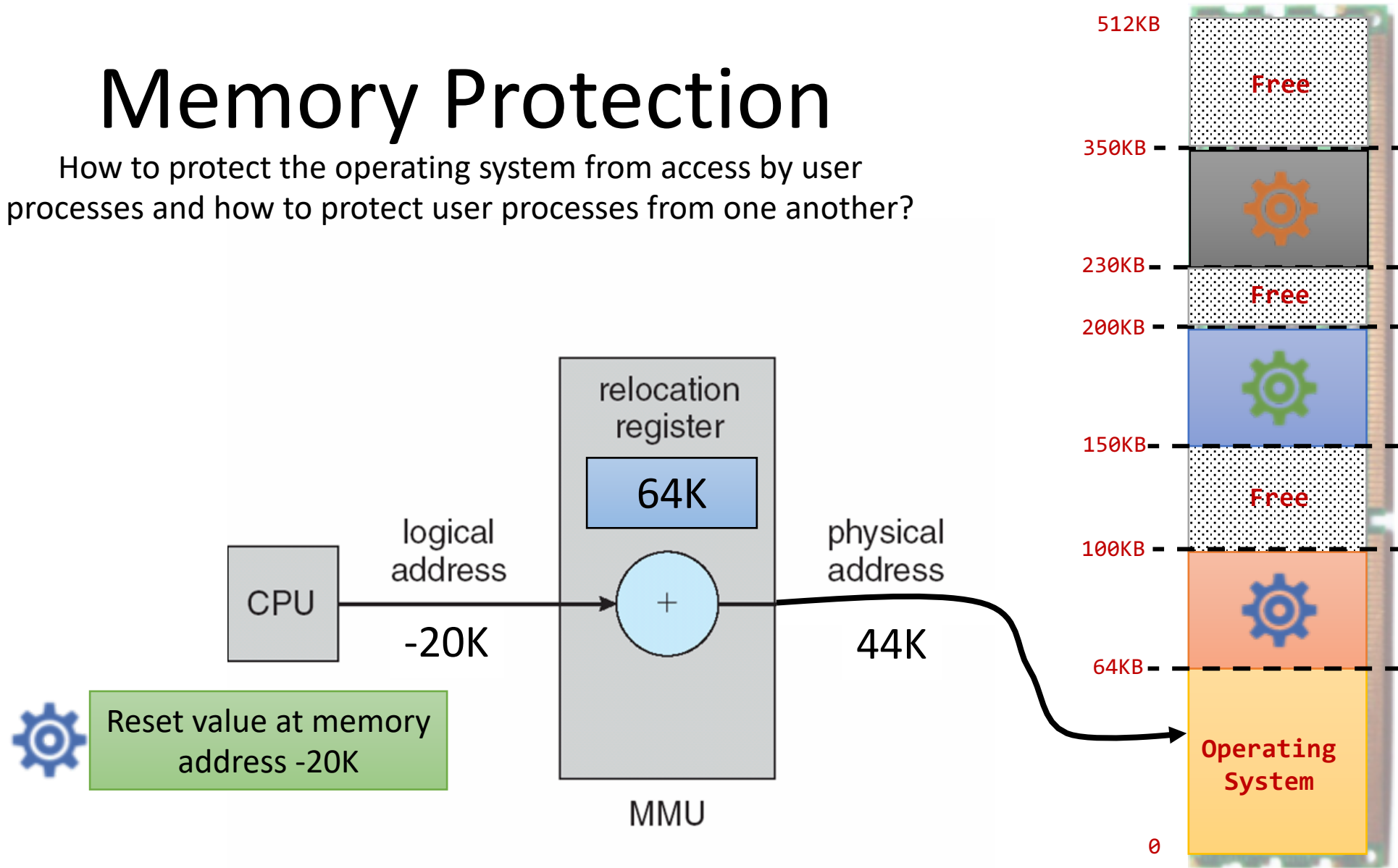
# Memory Protection

How to protect the operating system from access by user processes and how to protect user processes from one another?



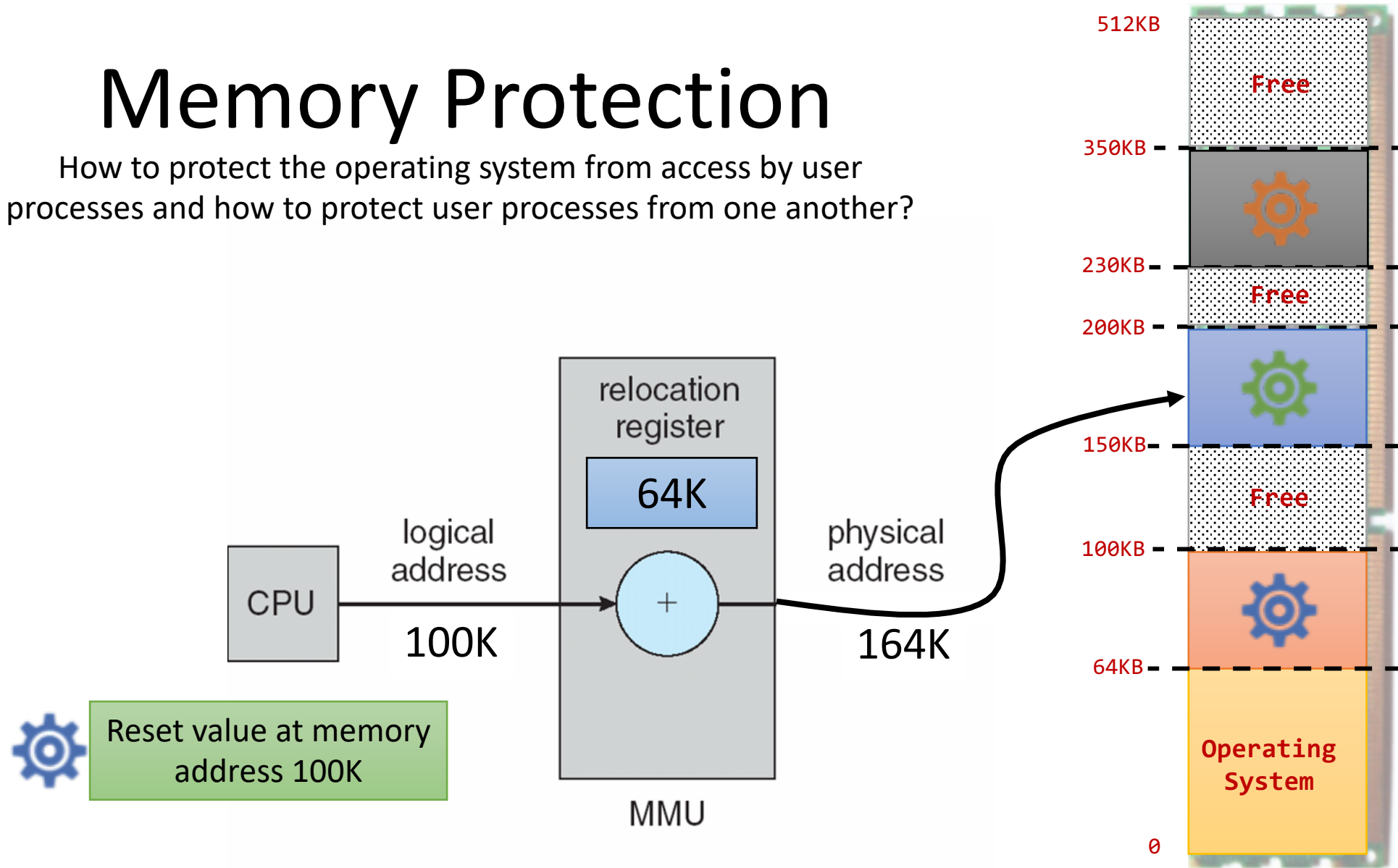
# Memory Protection

How to protect the operating system from access by user processes and how to protect user processes from one another?



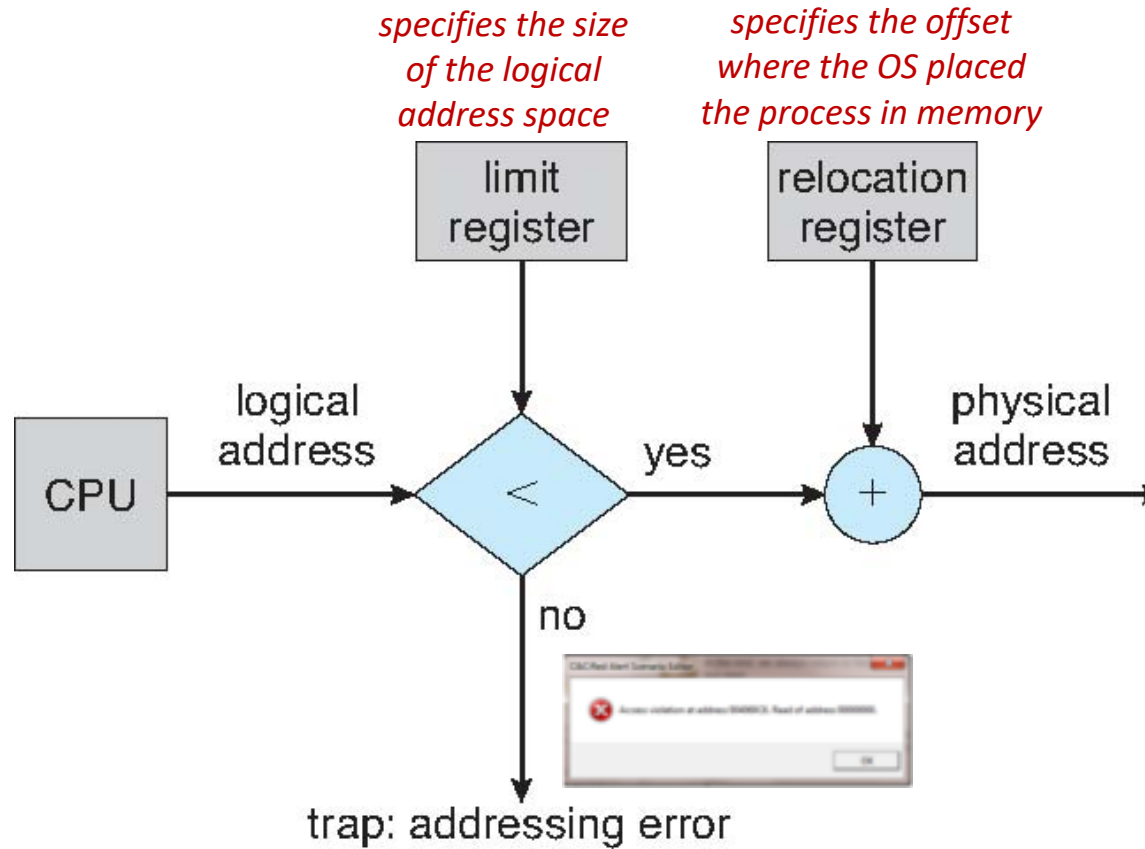
# Memory Protection

How to protect the operating system from access by user processes and how to protect user processes from one another?

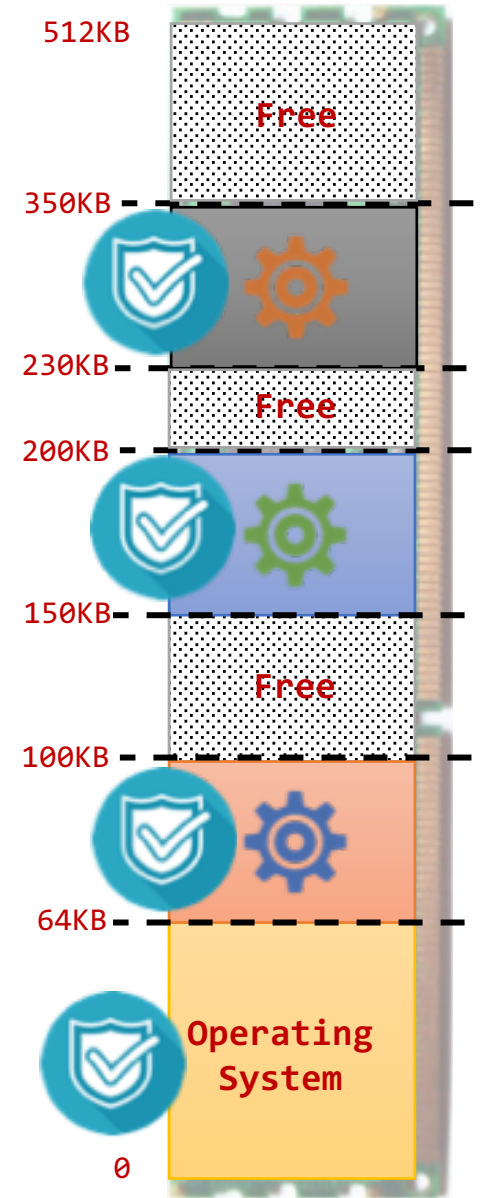


# Memory Protection

How to protect the operating system from access by user processes and how to protect user processes from one another?



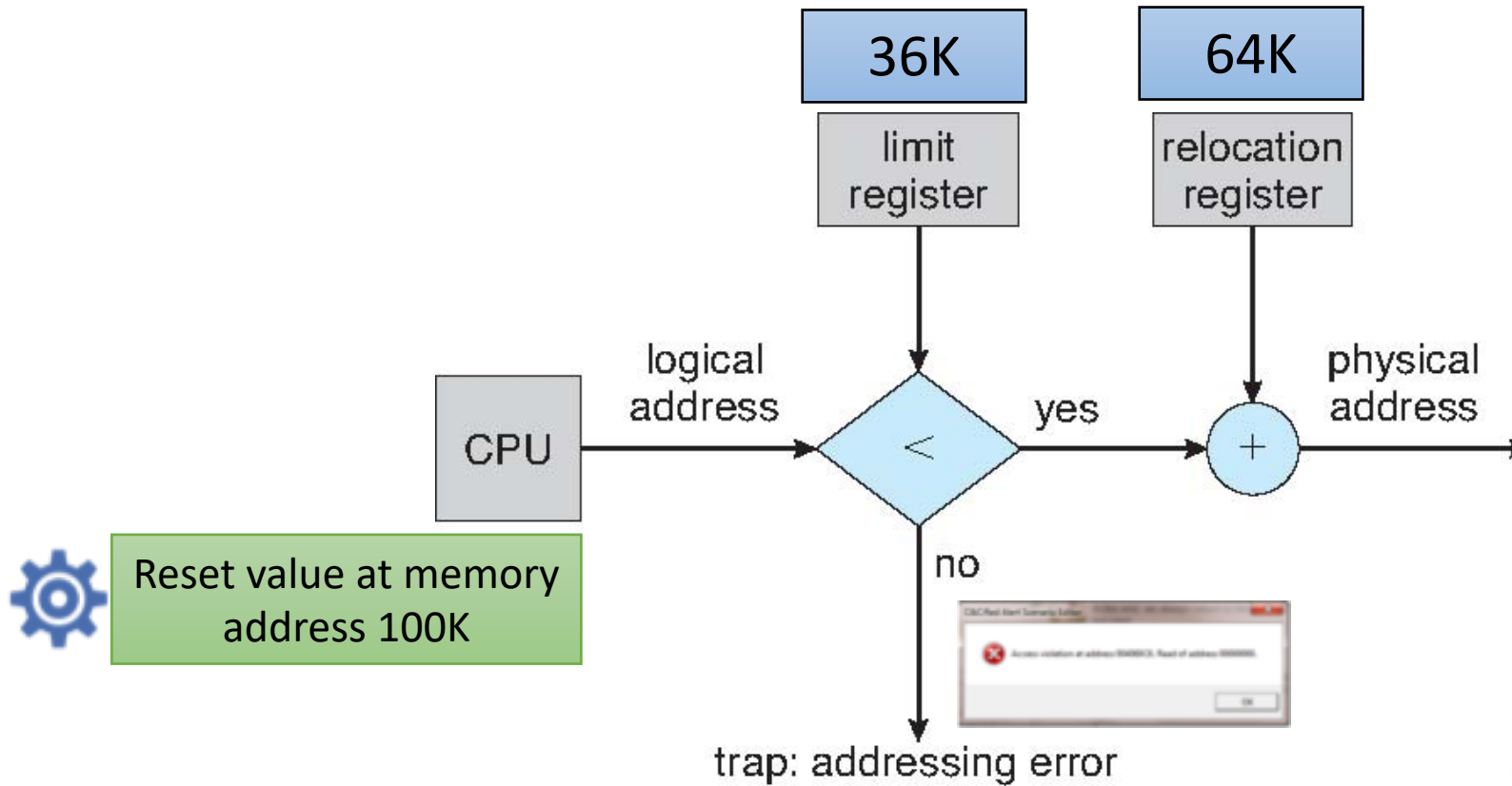
Any attempt by a program executing in user mode to access OS memory or other users' memory results in a trap to the operating system, which treats the attempt as a fatal error



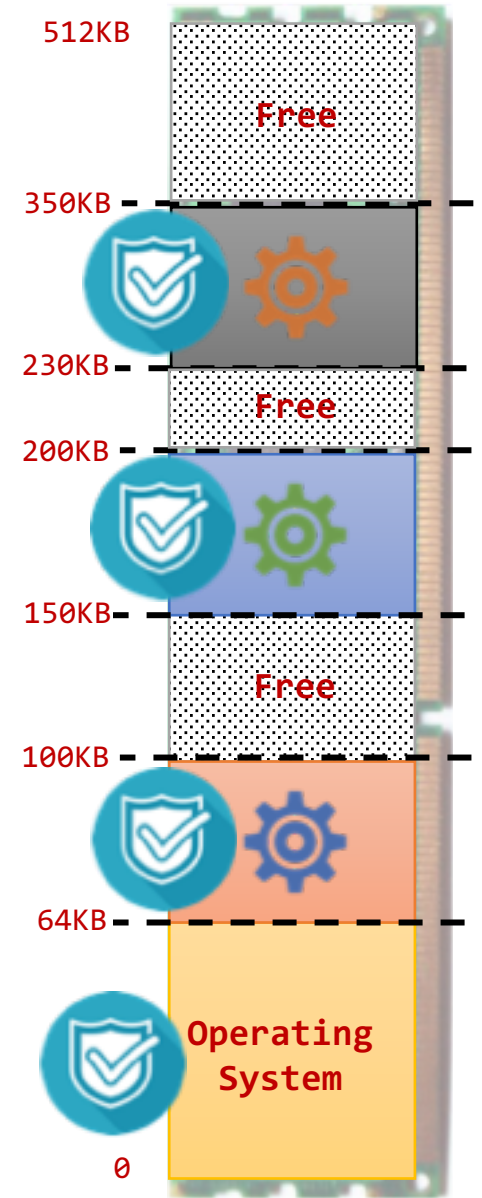


# Memory Protection

How to protect the operating system from access by user processes and how to protect user processes from one another?



*Any attempt by a program executing in user mode to access OS memory or other users' memory results in a trap to the operating system, which treats the attempt as a fatal error*

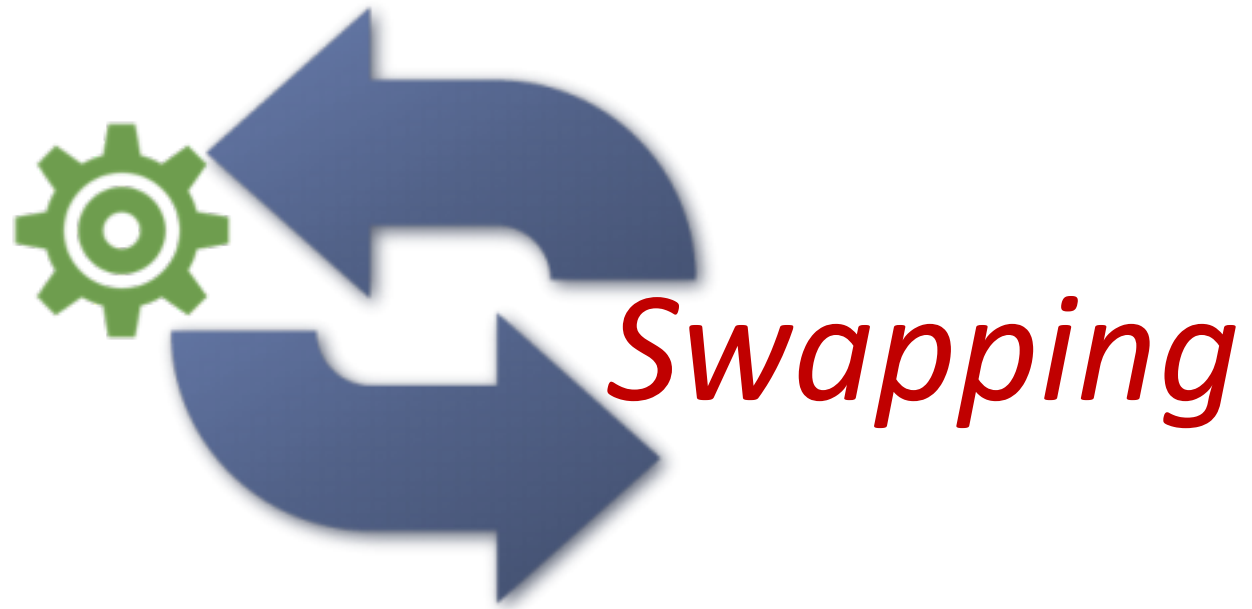


The relocation and limit registers is loaded only by the OS through a **special privileged instruction** only in **kernel mode**.

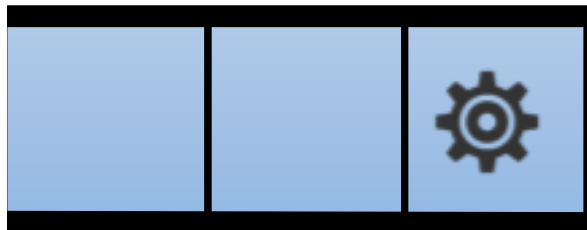
*This scheme allows the operating system to change the value of the registers but prevents user programs from changing the registers' contents.*

The OS is given unrestricted access to both OS memory and users' memory

This provision allows OS to load users' programs into users' memory, to dump out those programs in case of errors, to access and modify parameters of system calls, to perform I/O to and from user memory, and to provide many other services.

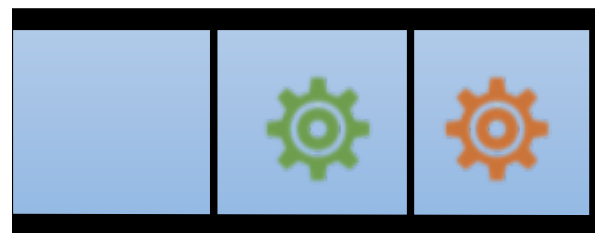
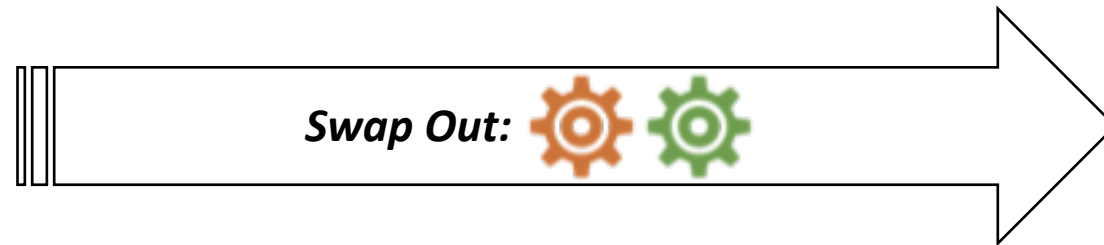
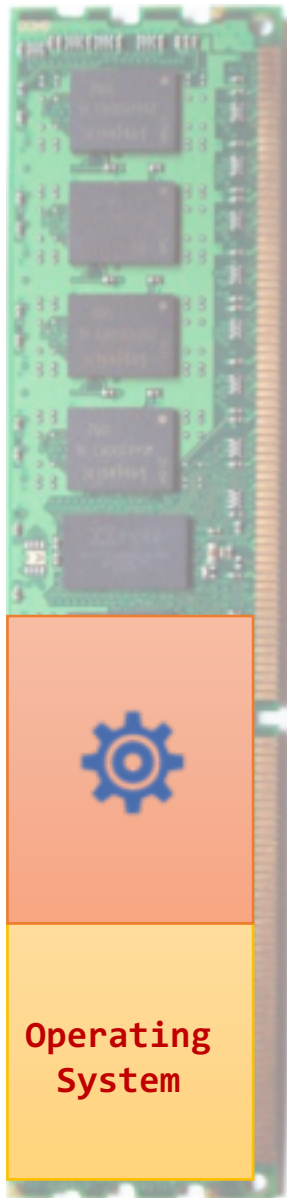


What if the total physical address space of all processes to exceed the real physical memory?



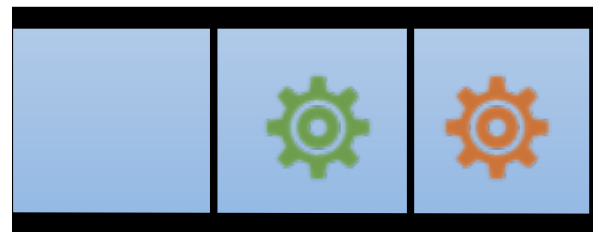
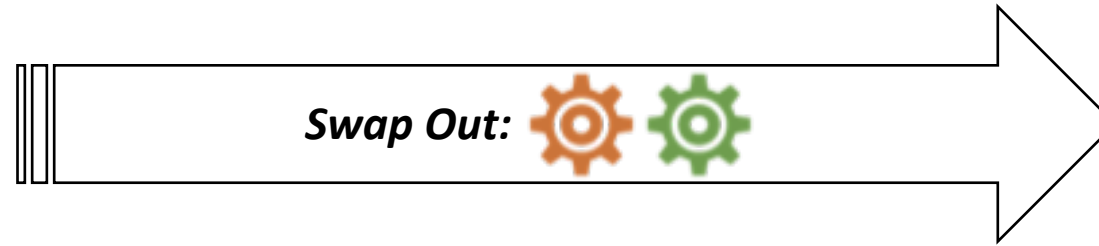
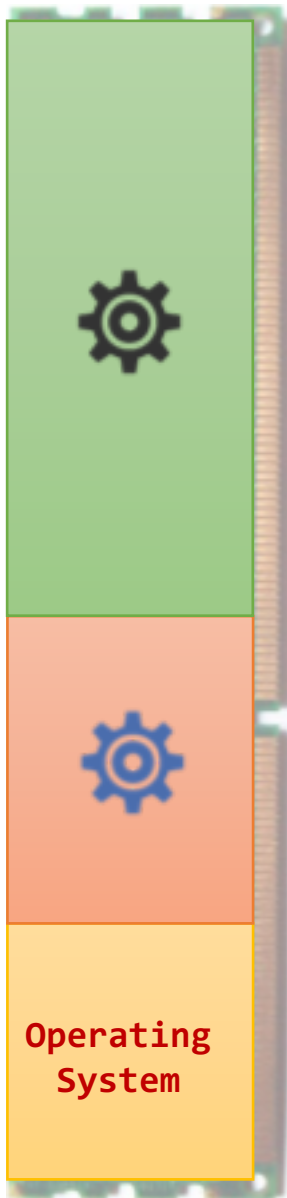
*Ready Queue*



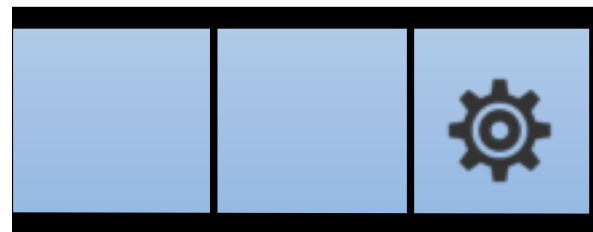
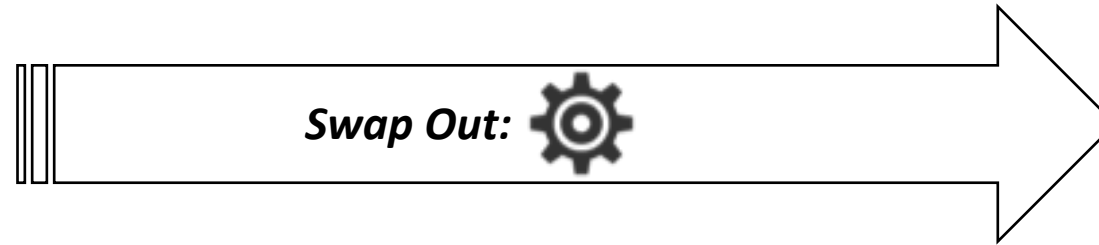
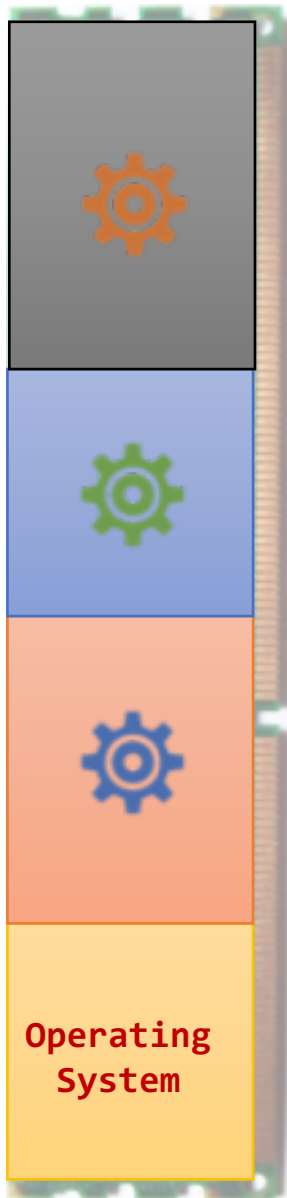


*Ready Queue*



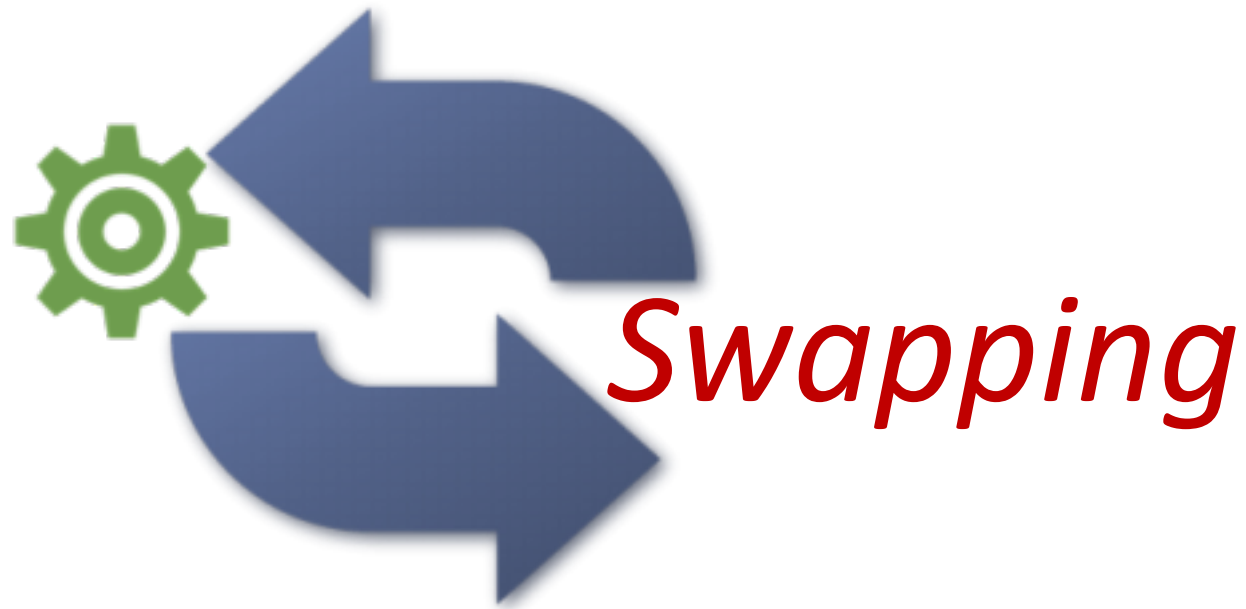


Ready Queue



*Ready Queue*



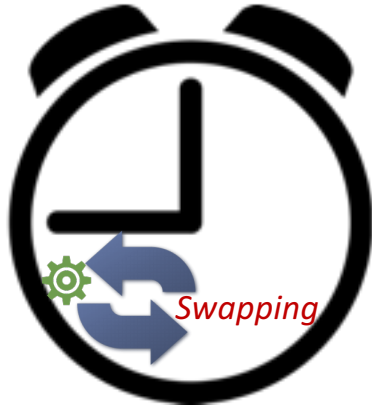


*A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution*

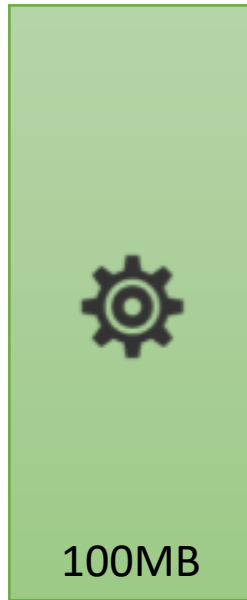
Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows) but its normally disabled

*Swap only when free memory extremely low*

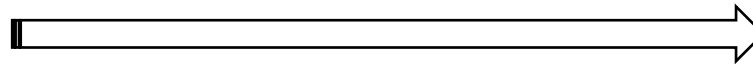




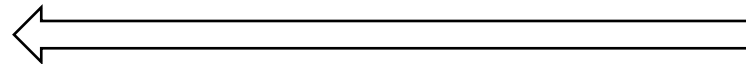
Major part of swap time is **transfer time**;  
total transfer time is directly proportional  
to the amount of memory swapped



**2 Seconds** + Context Switch Time

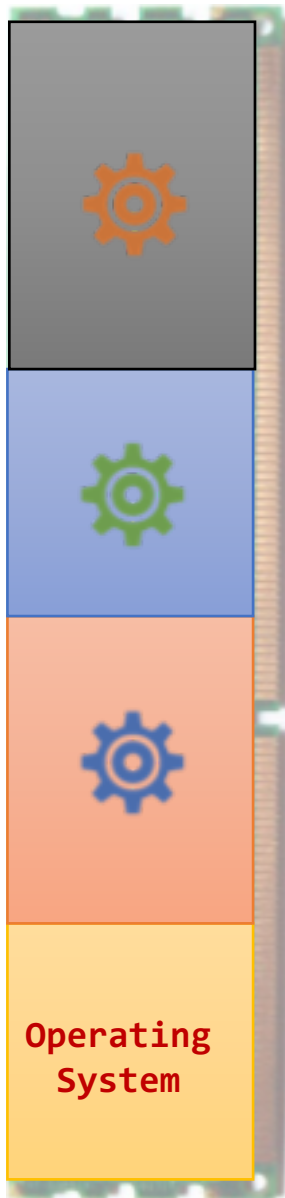


*Transfer Rate: 50MB/second*



**2 Seconds** + Context Switch Time





*is waiting for I/O Operation*

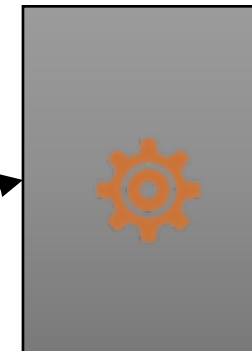
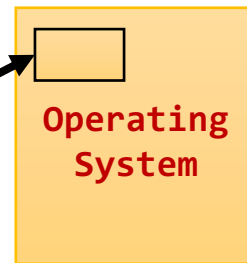


## What should the OS do?

(1) Don't swap out a process with pending I/O as the I/O would occur to wrong process

(2) Do the swapping, but perform **double buffering**

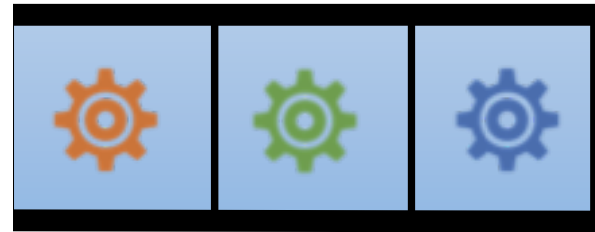
*Execute I/O operations only into OS buffers instead of process memory. After the process is **swapped in**, transfer between OS buffers and process memory*





The main memory must accommodate both the operating system and the various user processes

*We therefore need to allocate main memory in the most efficient way possible.*

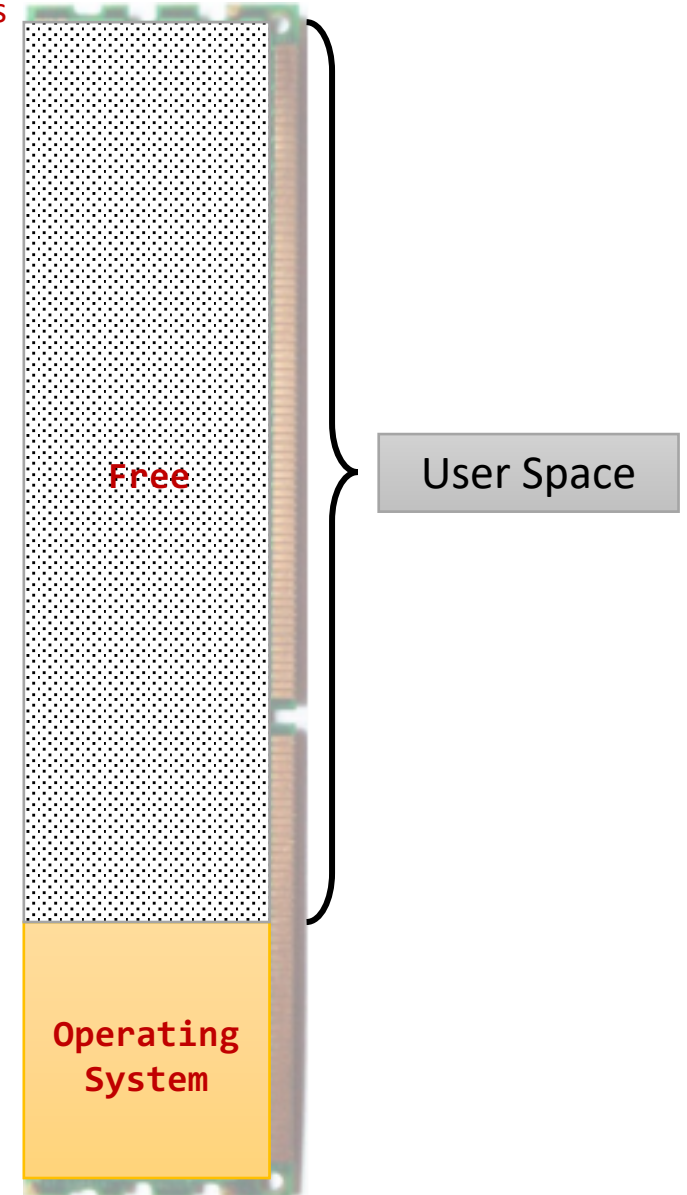


*Input Queue*

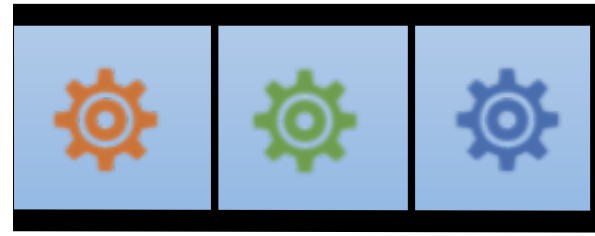
*The collection of processes on the disk waiting to be loaded into memory for execution*

Higher Address

Lower Address



# How does the OS **allocate** free space to processes to be loaded in memory?

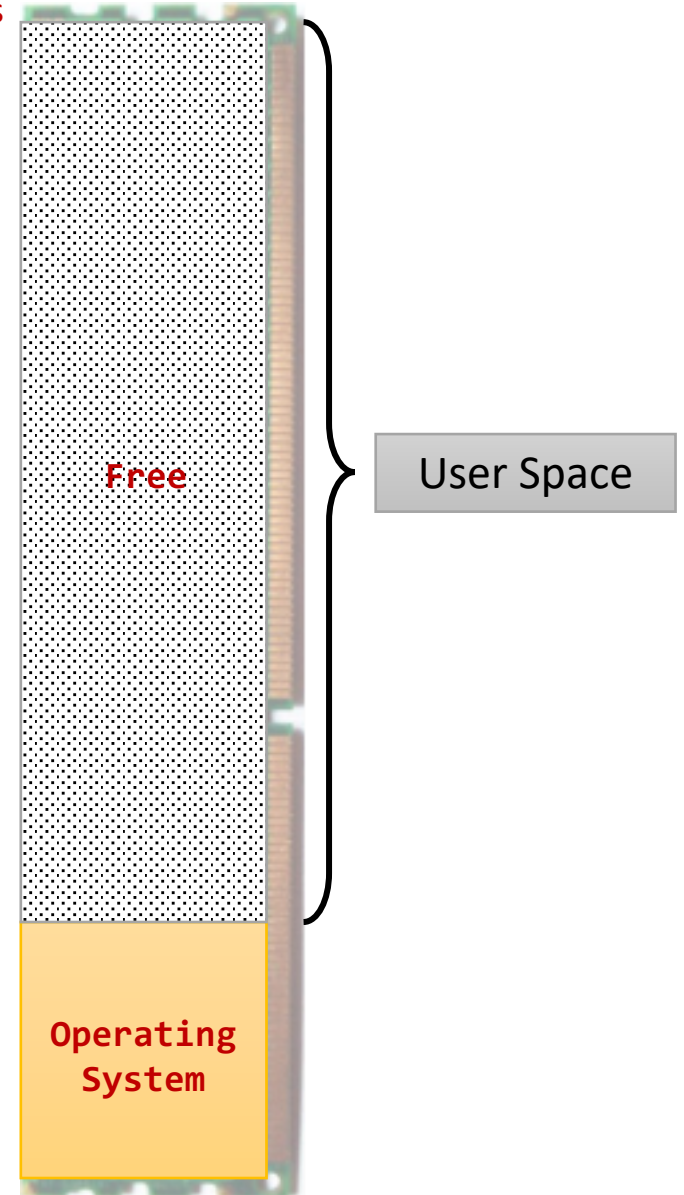


*Input Queue*

*The collection of processes on the disk waiting to be loaded into memory for execution*

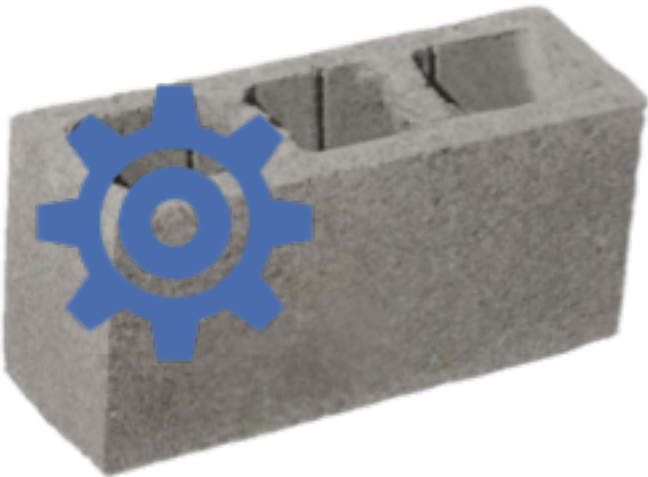
Higher Address

Lower Address



# How does the OS **allocate** free space to processes to be loaded in memory?

“Memory-Management Schemes”



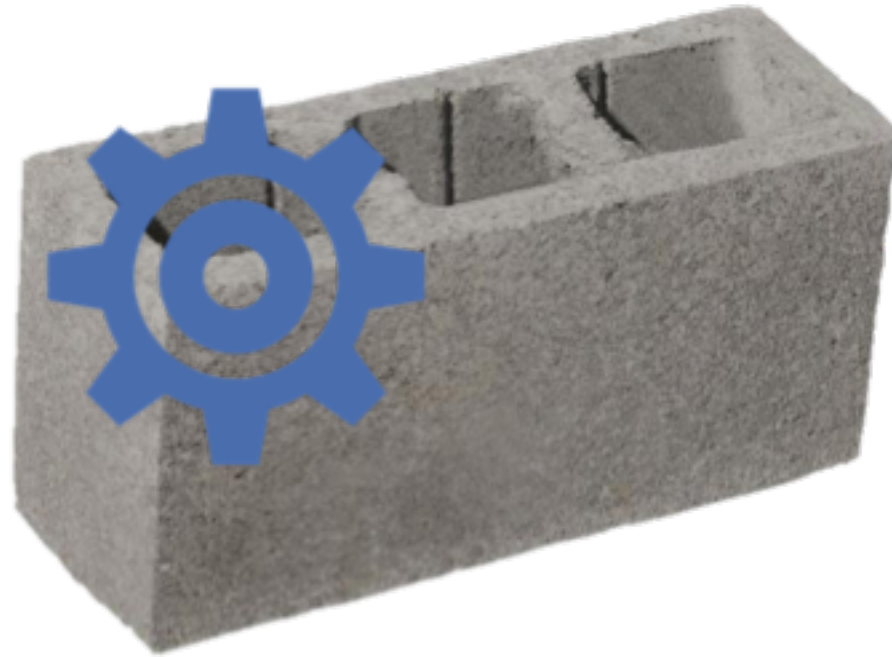
Contagious



Segmentation



Paging



# Contagious Memory Allocation

*Each process like a brick cannot be segmented or broken into pieces and must be allocated a single section of memory that is contiguous*

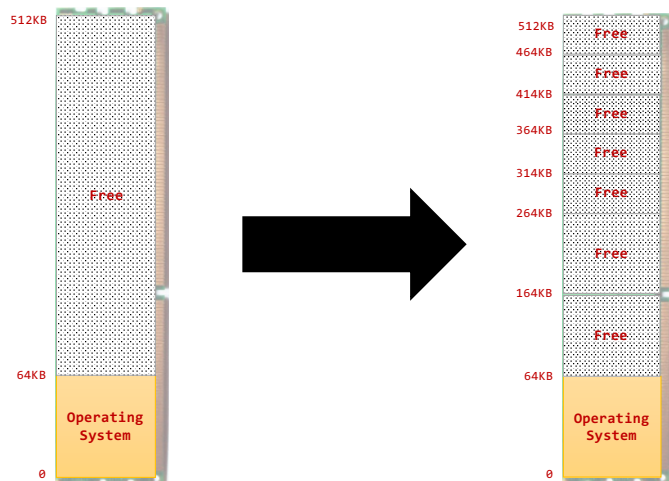




## Contagious Memory Allocation

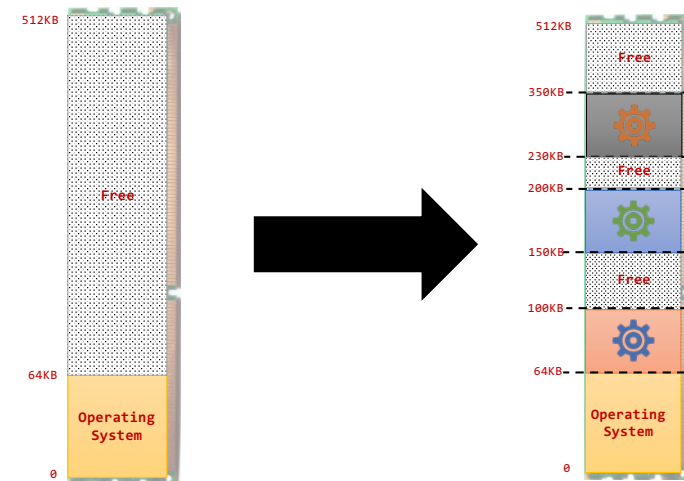
### Fixed-Size Partitioning

Divide memory into **fixed-size partitions** “segments”.  
Each segment contains only one process

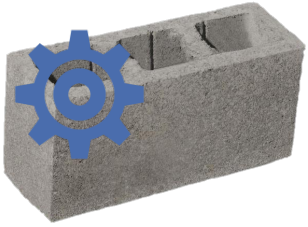


### Variable-Size Partitioning

Create partitions “segment” **variable-sized** to a given process’ needs



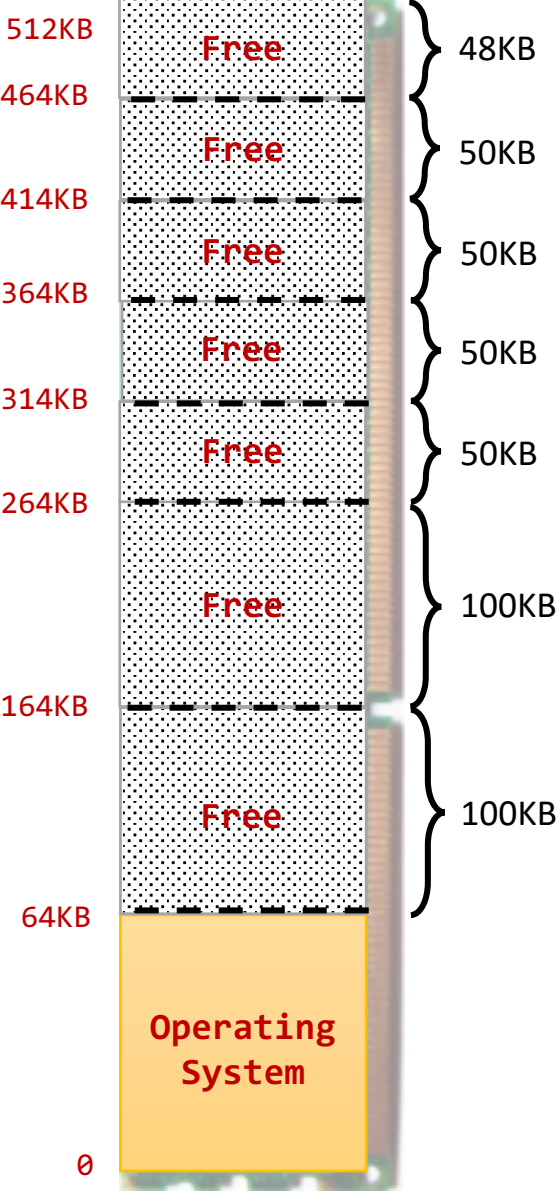


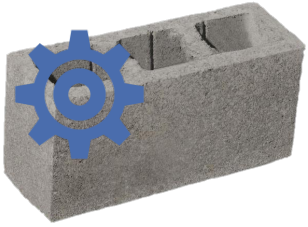


Contagious Memory Allocation  
*Fixed-Size Partitioning*



Divide memory into fixed-size partitions (segments)  
*Partitions don't have to be the same size*



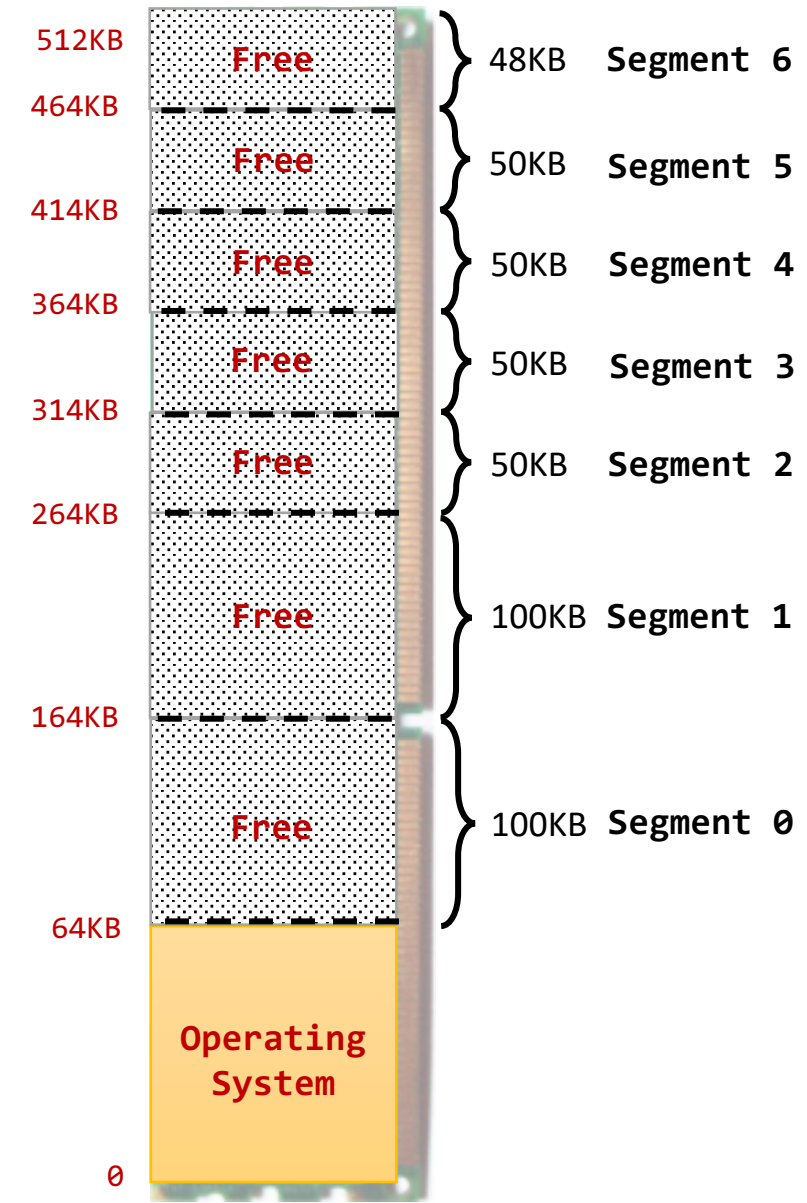
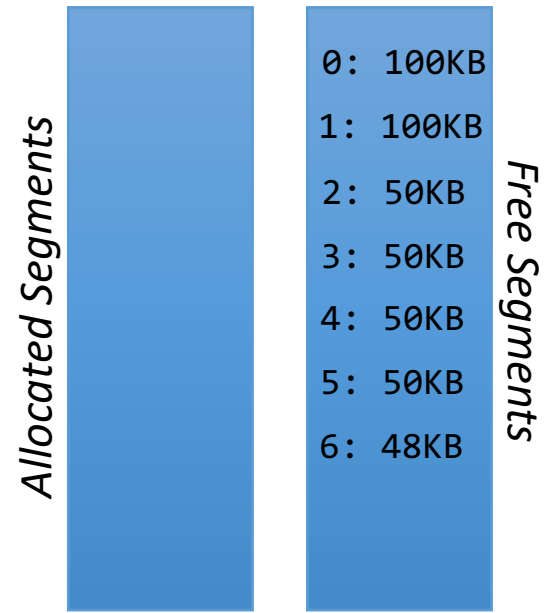


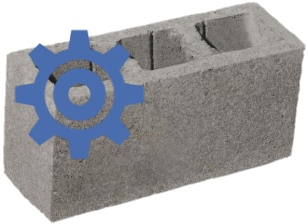
# Contagious Memory Allocation

*Fixed-Size Partitioning*

**Operating system maintains information about**

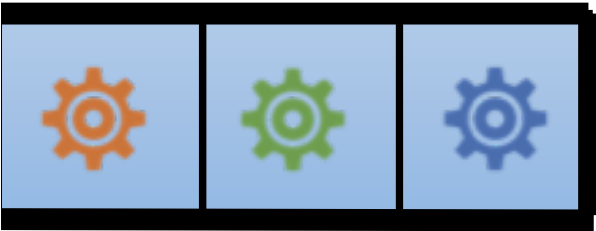
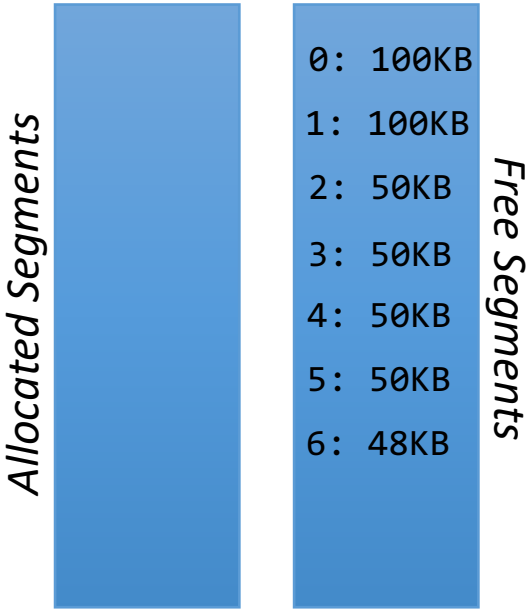
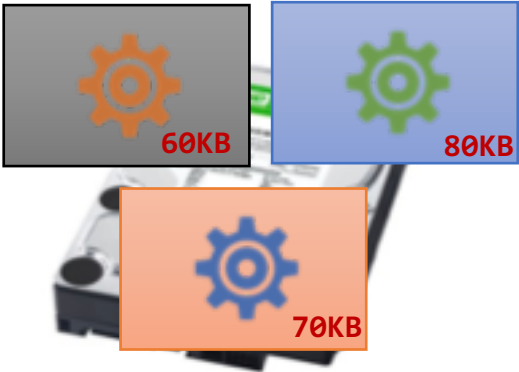
- a) Allocated Partitions (Segments)    b) Free Partitions (Holes or Segments)



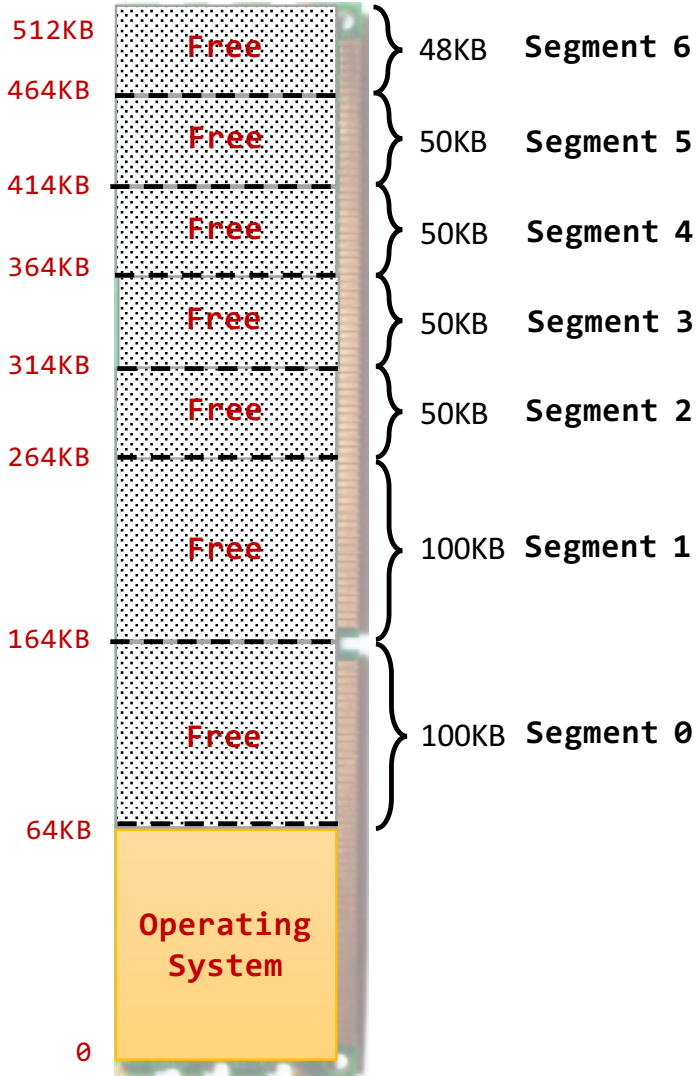


Contiguous Memory Allocation

*Fixed-Size Partitioning*

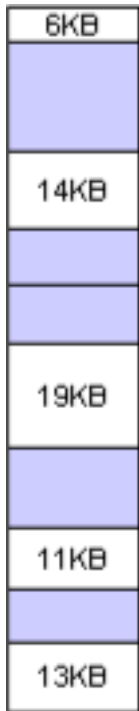


*Input Queue*

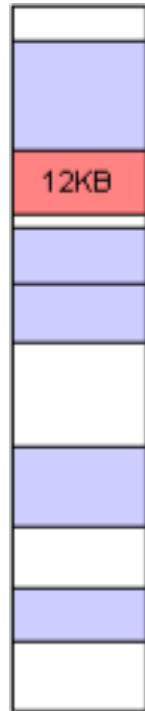


# Which Partition “Segment” to choose?

<https://www.youtube.com/watch?v=TnBQkzBsOe8>



12KB  
Process



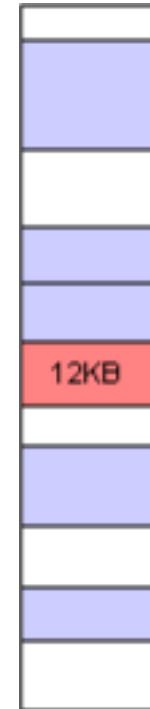
First-Fit

Allocate the **first** segment that is big enough



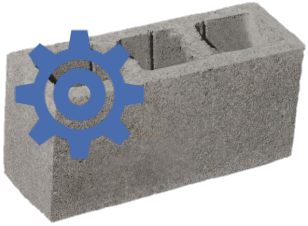
Best-Fit

Allocate the **smallest** segment that is big enough  
*search entire list, unless ordered by size*



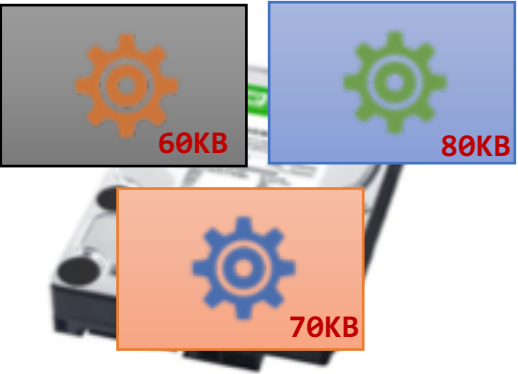
Worst-Fit

Allocate the **largest** segment  
*search entire list, unless ordered by size*



Contiguous Memory Allocation

*Fixed-Size Partitioning*



Allocated Segments

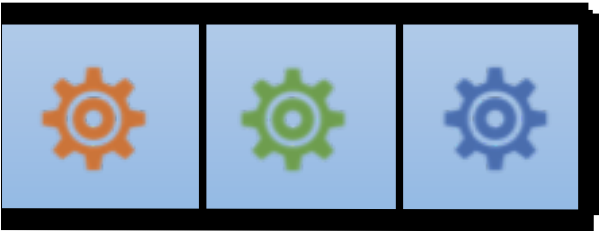


0: 100KB  
1: 100KB  
2: 50KB  
3: 50KB  
4: 50KB  
5: 50KB  
6: 48KB

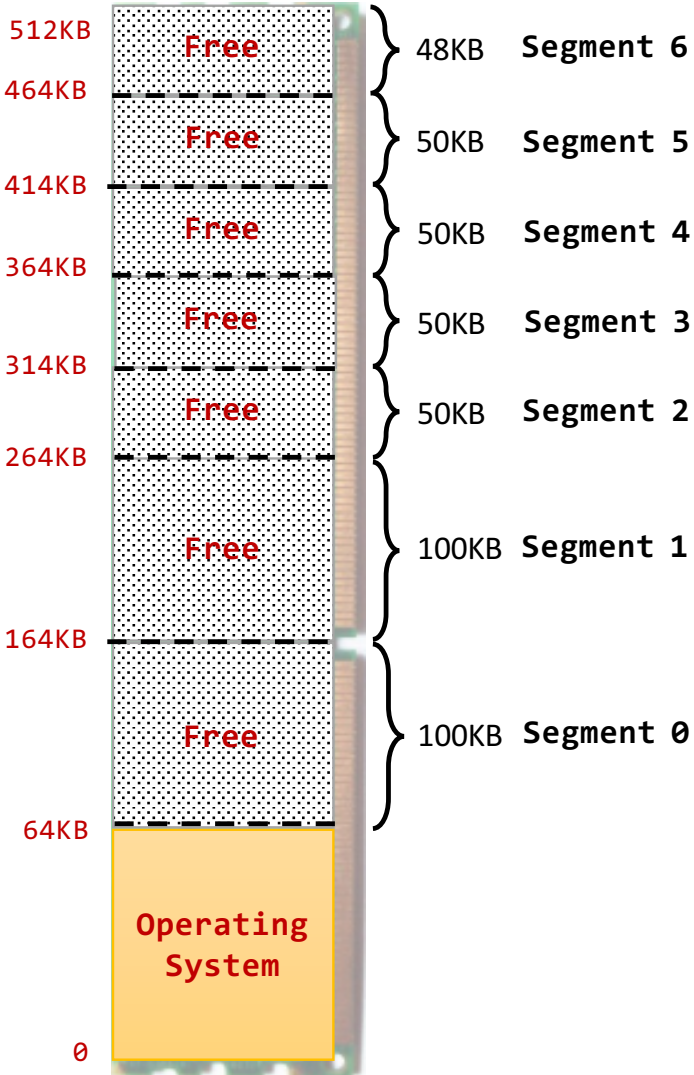
Free Segments

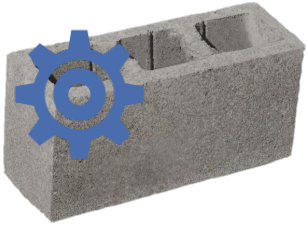
# First-Fit

Allocate the *first* segment that is big enough



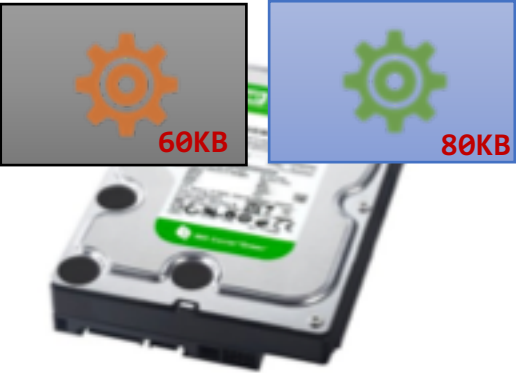
*Input Queue*





Contiguous Memory Allocation

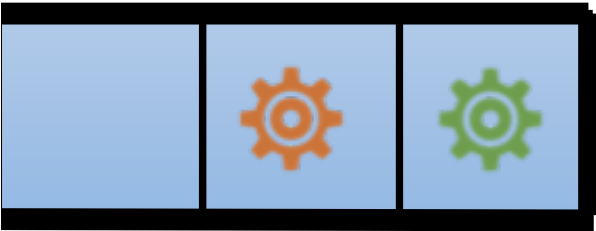
*Fixed-Size Partitioning*



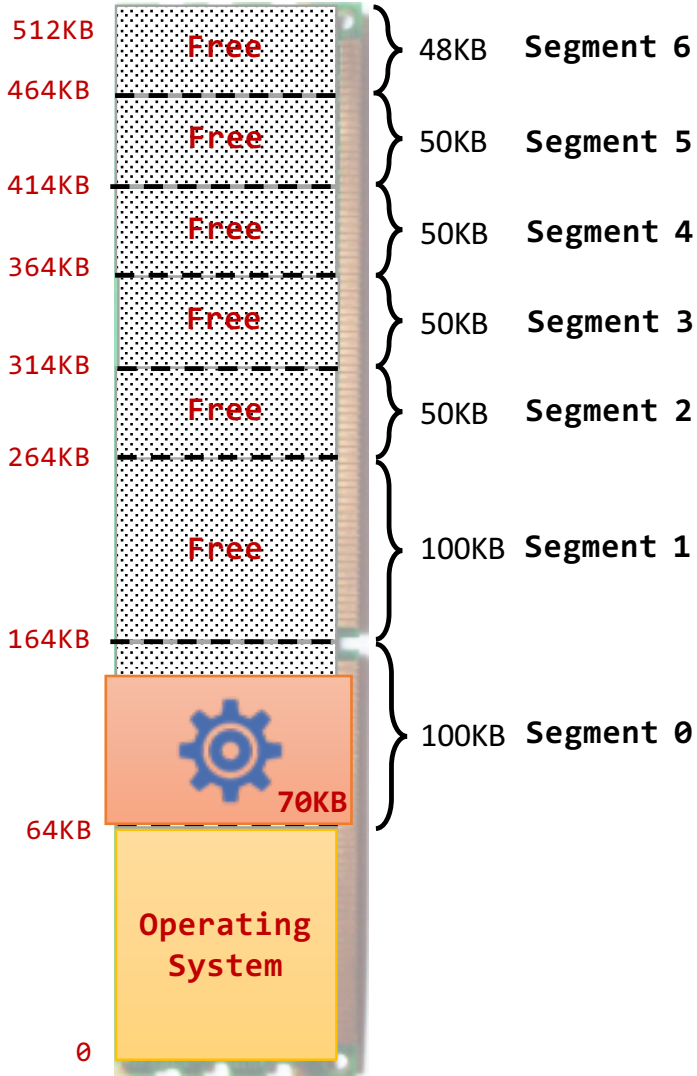
Allocated Segments	0: 100KB	1: 100KB	Free Segments
		2: 50KB	
		3: 50KB	
		4: 50KB	
		5: 50KB	
		6: 48KB	

First-Fit

Allocate the *first* segment that is big enough



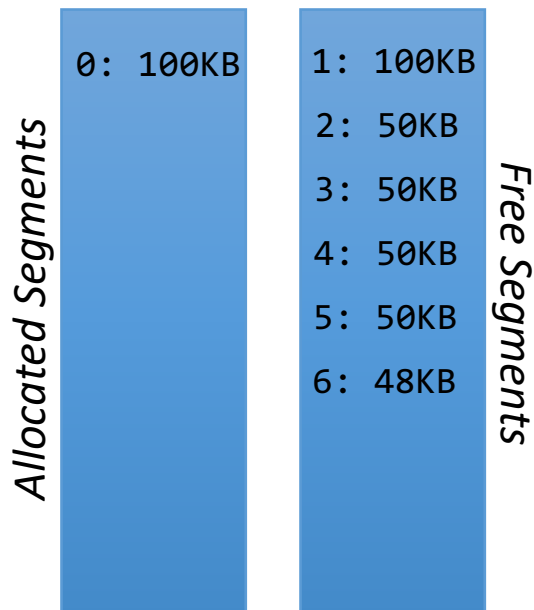
*Input Queue*





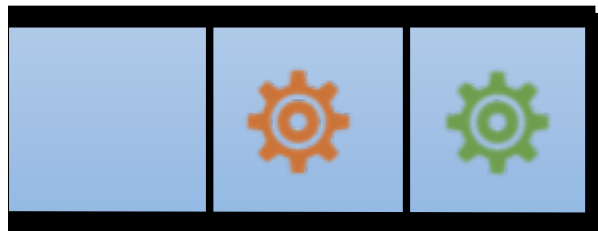
## Contiguous Memory Allocation

*Fixed-Size Partitioning*



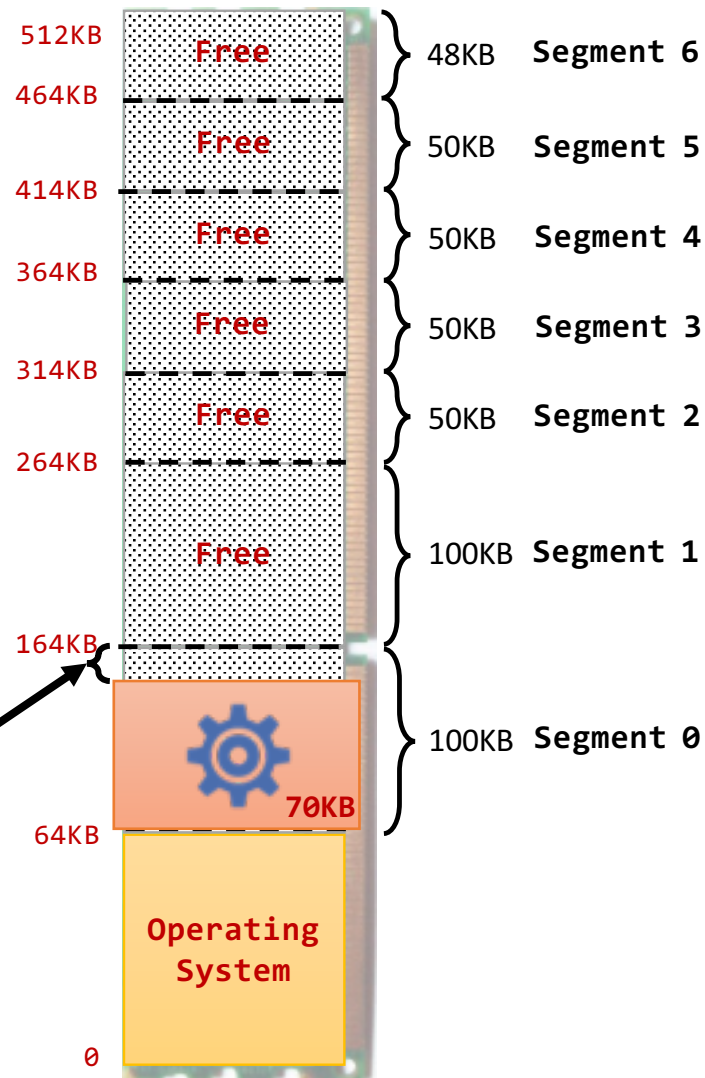
# First-Fit

Allocate the *first* segment that is big enough



*Input Queue*

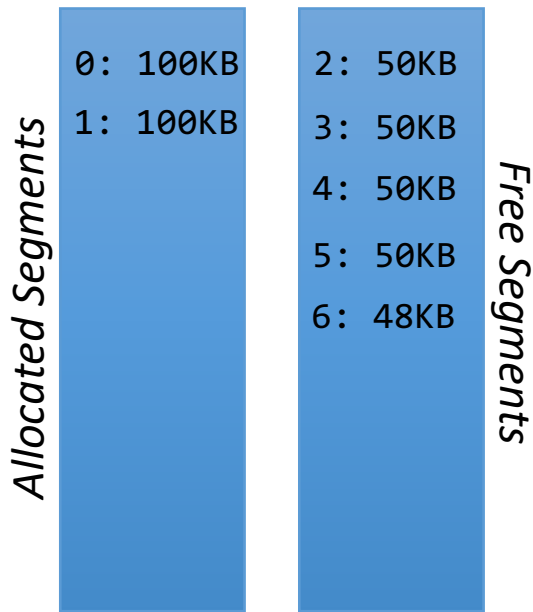
**Internal Fragmentation**  
Allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used





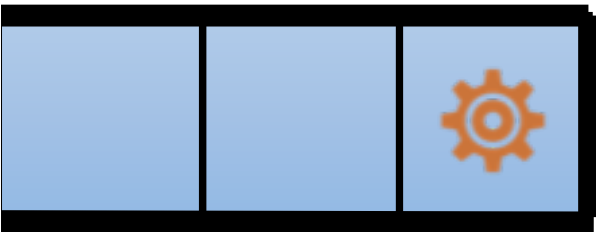
Contiguous Memory Allocation

*Fixed-Size Partitioning*

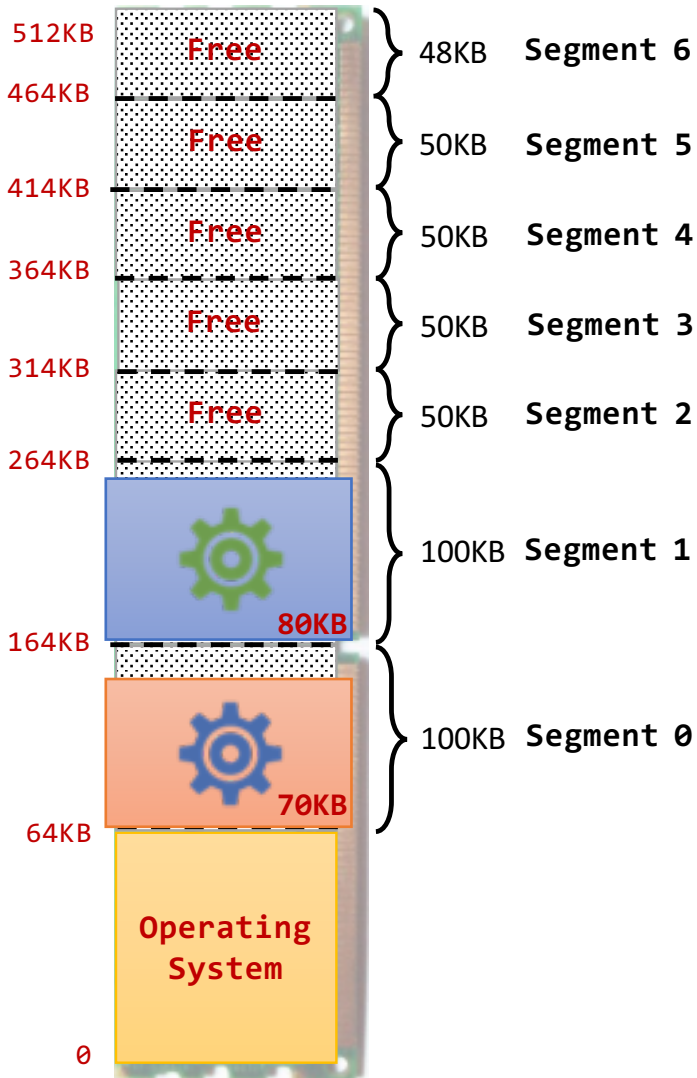


# First-Fit

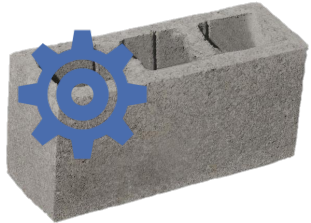
Allocate the *first* segment that is big enough



*Input Queue*

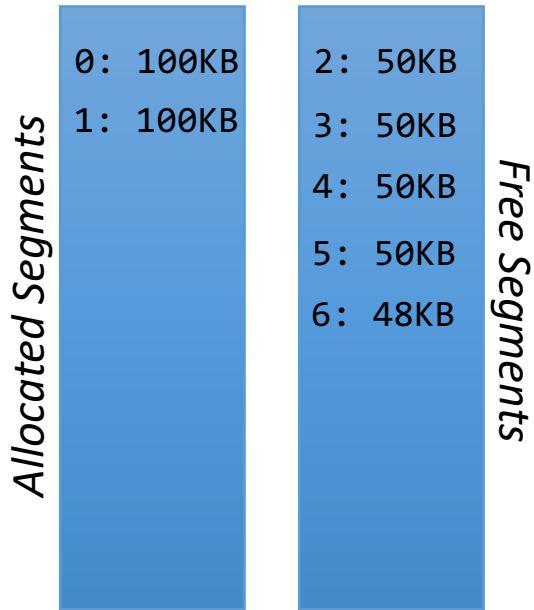






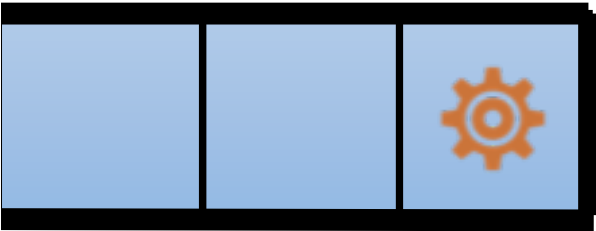
Contiguous Memory Allocation

*Fixed-Size Partitioning*

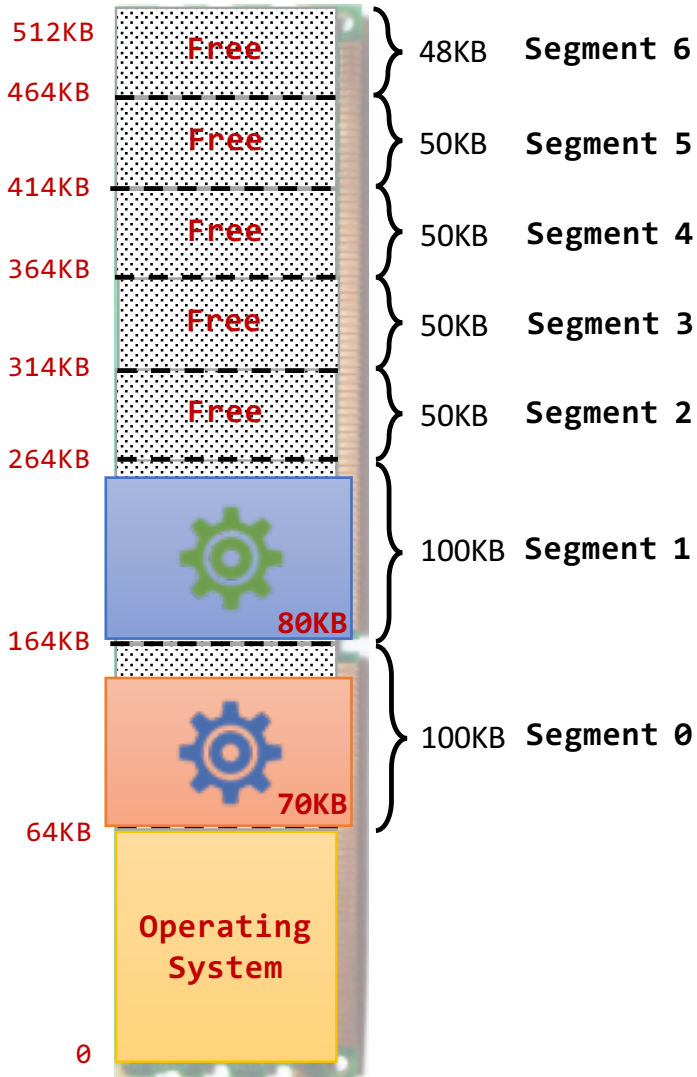


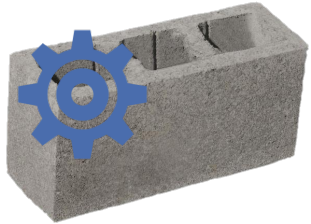
# First-Fit

Allocate the *first* segment that is big enough



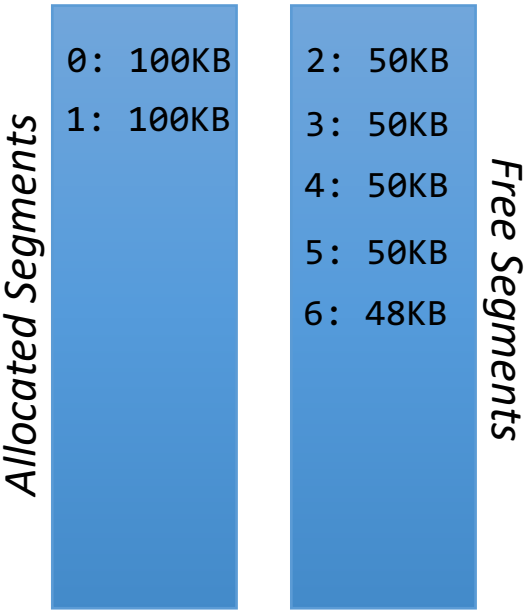
*Input Queue*





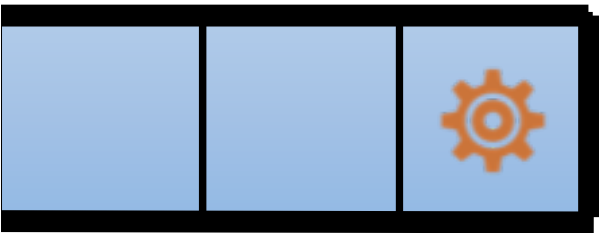
Contiguous Memory Allocation

*Fixed-Size Partitioning*



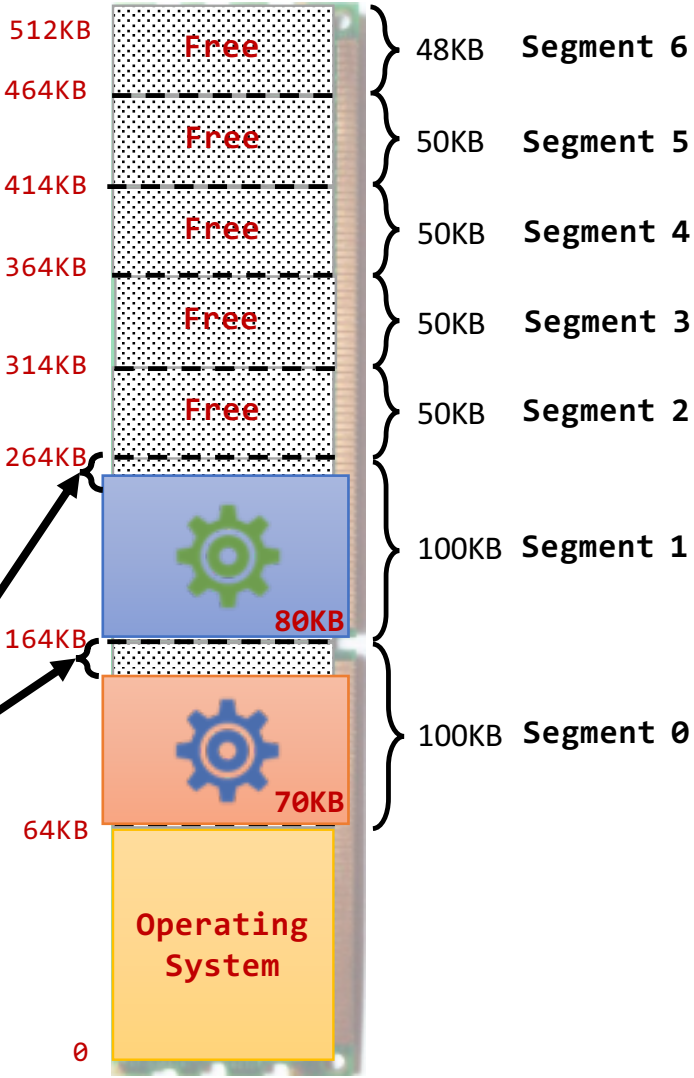
# First-Fit

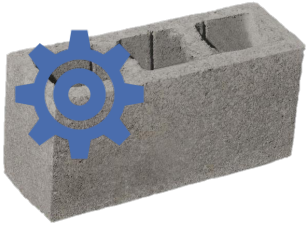
Allocate the *first* segment that is big enough



*Input Queue*

**Internal Fragmentation**  
Allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used





Contiguous Memory Allocation  
*Fixed-Size Partitioning*



Allocated Segments

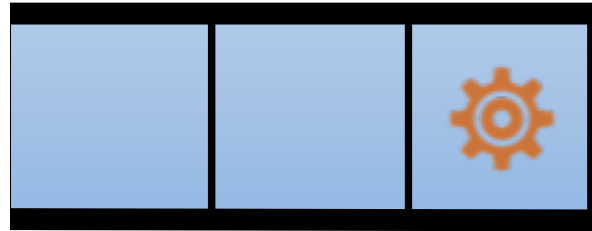
0: 100KB  
1: 100KB

2: 50KB  
3: 50KB  
4: 50KB  
5: 50KB  
6: 48KB

Free Segments

# First-Fit

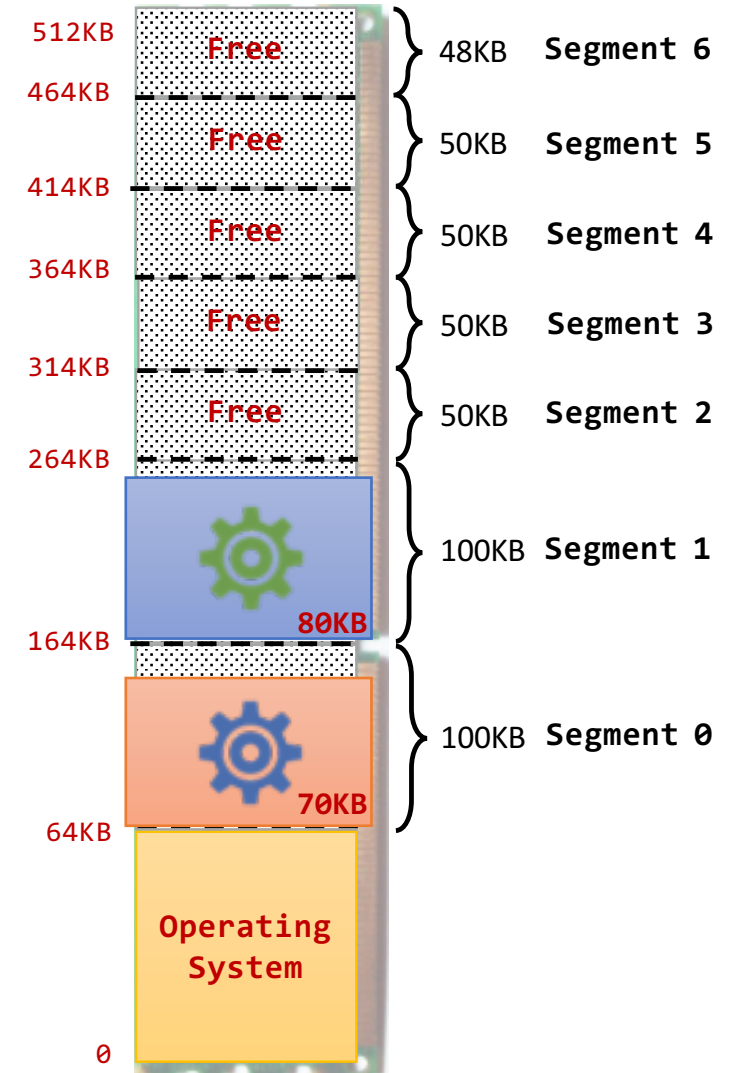
Allocate the *first* segment that is big enough

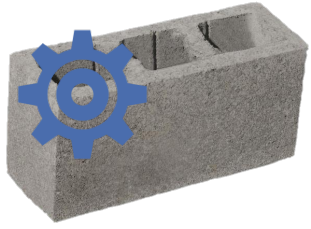


*Input Queue*

## External Fragmentation

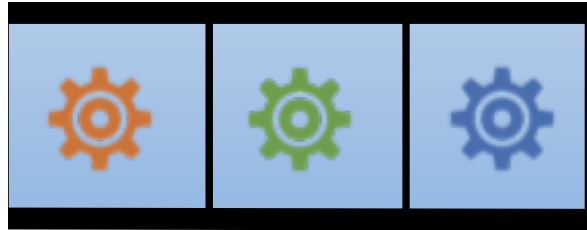
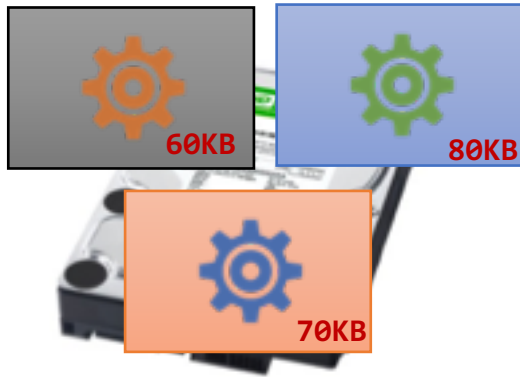
Total memory space exists to satisfy a request, but it is not contiguous





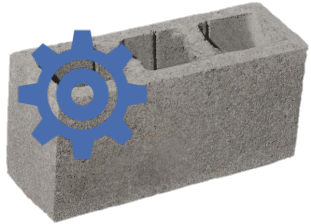
## Contagious Memory Allocation

*Variable-Size Partitioning*



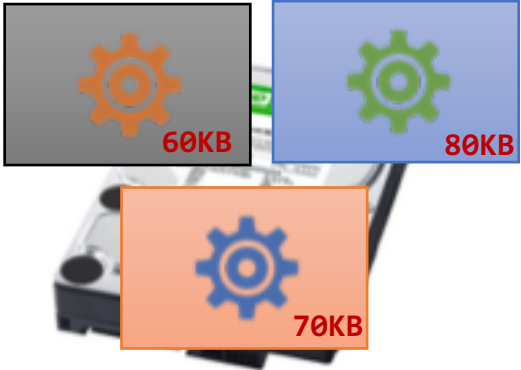
*Input Queue*





Contagious Memory Allocation

*Variable-Size Partitioning*



*Allocated Space*

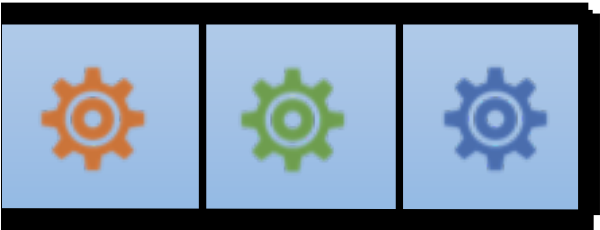
64KB

*Free Space*

448KB

# Best-Fit

Allocate the ***smallest*** segment that is big enough



*Input Queue*

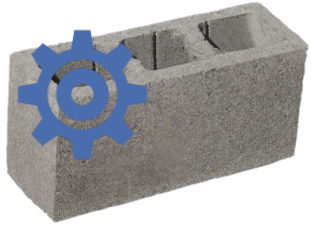
512KB

Free

64KB

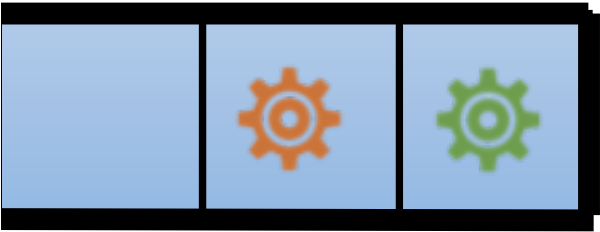
Operating  
System

0



# Best-Fit

Contagious Memory Allocation  
*Variable-Size Partitioning*



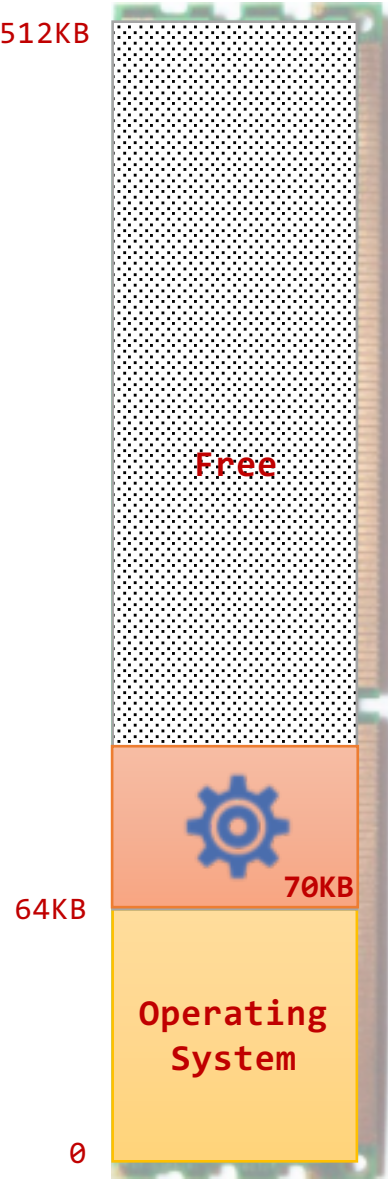
*Input Queue*

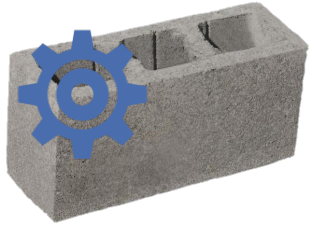
*Allocated Space*

134KB

*Free Space*

378KB





Contagious Memory Allocation  
*Variable-Size Partitioning*



*Allocated Space*

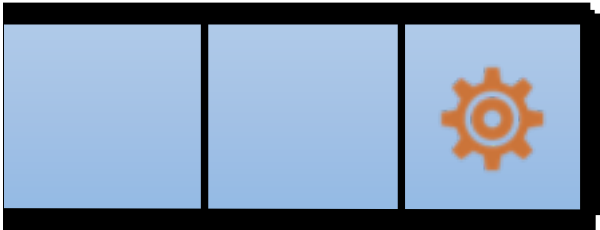
214KB

*Free Space*

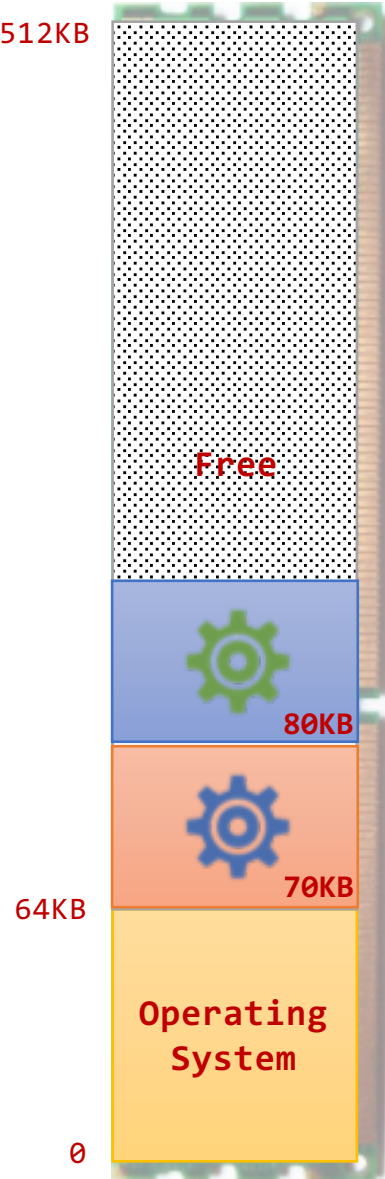
298KB

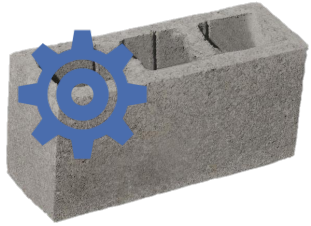
# Best-Fit

Allocate the ***smallest*** segment that is big enough



*Input Queue*





# Best-Fit

Contagious Memory Allocation  
*Variable-Size Partitioning*



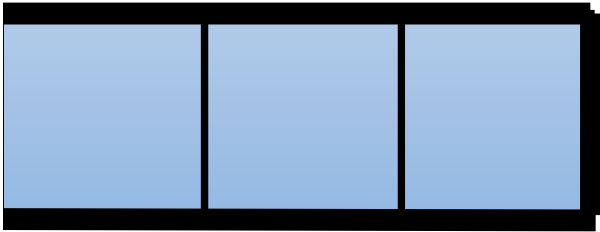
*Allocated Space*

274KB

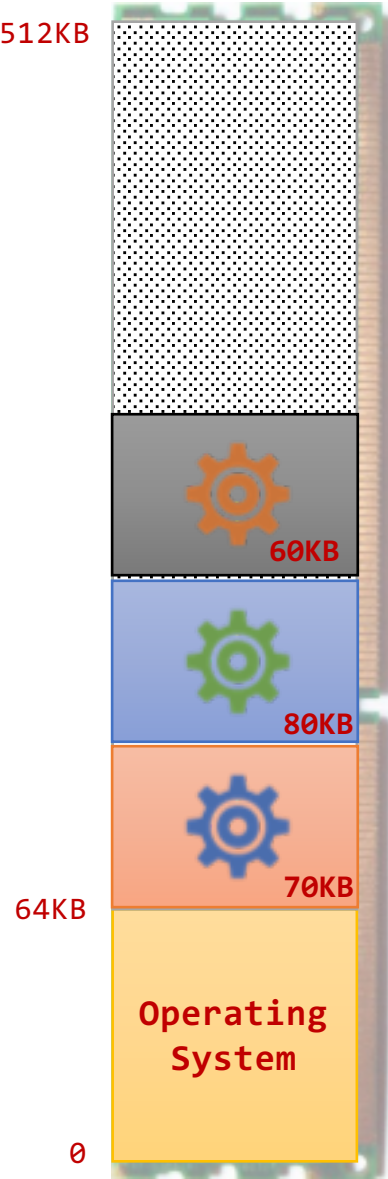
*Free Space*

238KB

Allocate the ***smallest*** segment that is big enough



*Input Queue*







# Best-Fit

Contagious Memory Allocation  
*Variable-Size Partitioning*



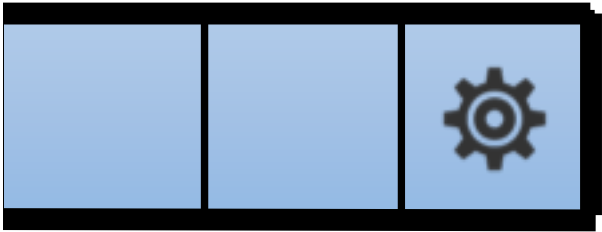
*Allocated Space*

274KB

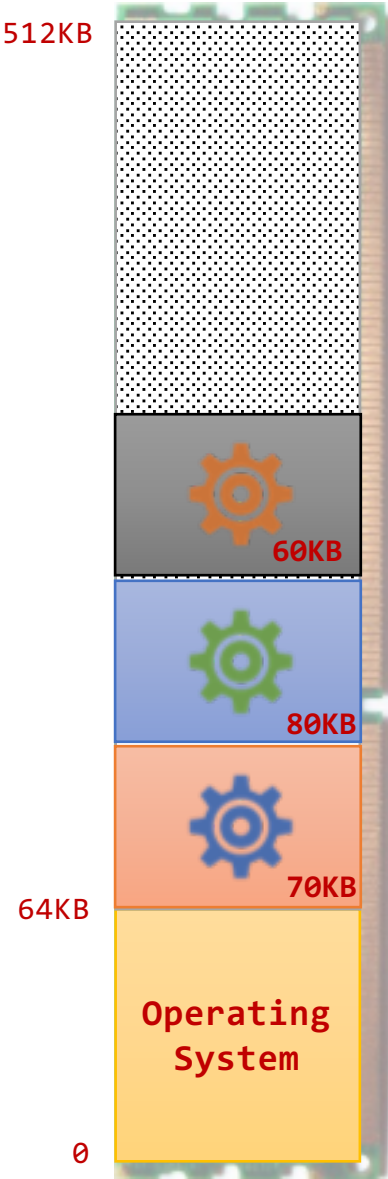
*Free Space*

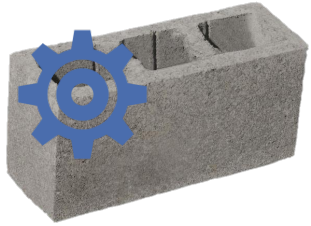
238KB

Allocate the ***smallest*** segment that is big enough



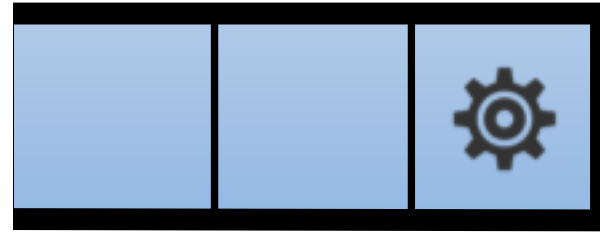
*Input Queue*





# Best-Fit

Allocate the ***smallest*** segment that is big enough



*Input Queue*

Contagious Memory Allocation  
*Variable-Size Partitioning*



*Allocated Space*

194KB

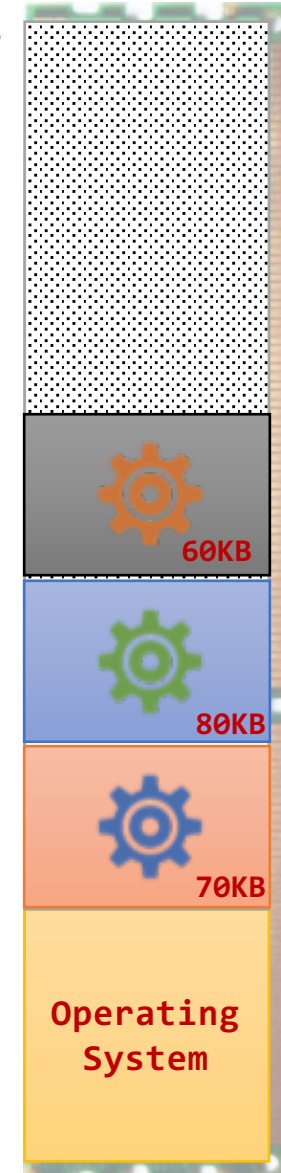
*Free Space*

318KB

## External Fragmentation

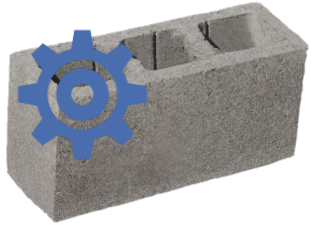
Total memory space exists to satisfy a request, but it is not contiguous

512KB



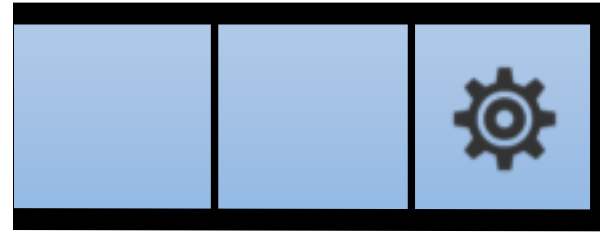
512KB





# Best-Fit

Allocate the ***smallest*** segment that is big enough



*Input Queue*

Contiguous Memory Allocation  
*Variable-Size Partitioning*



*Allocated Space*

194KB

*Free Space*

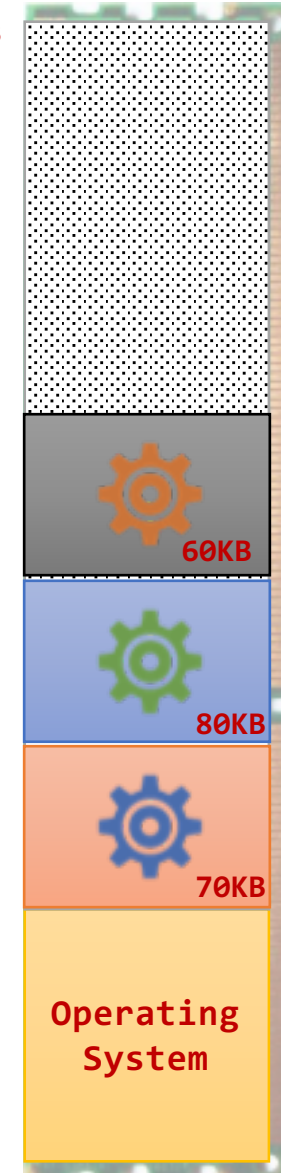
318KB

## External Fragmentation

Total memory space exists to satisfy a request, but it is not contiguous

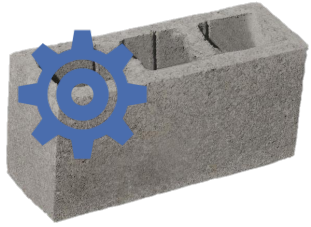
**Memory Compaction:** Shuffle memory contents to place all free memory together in one large block

512KB



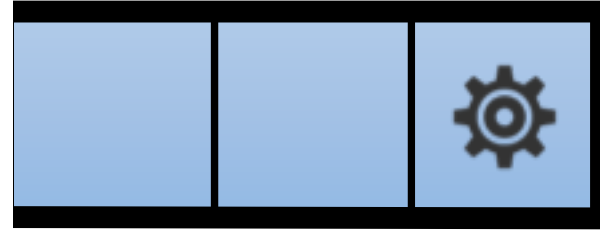
512KB





# Best-Fit

Allocate the ***smallest*** segment that is big enough



*Input Queue*

Contiguous Memory Allocation  
*Variable-Size Partitioning*



*Allocated Space*

194KB

*Free Space*

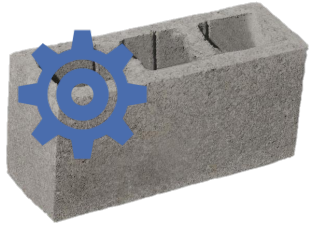
318KB

## External Fragmentation

Total memory space exists to satisfy a request, but it is not contiguous

**Memory Compaction:** Shuffle memory contents to place all free memory together in one large block





Contagious Memory Allocation  
*Variable-Size Partitioning*



*Allocated Space*

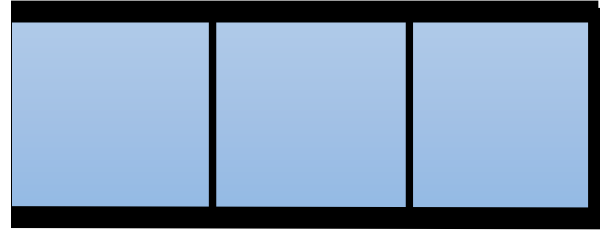
494KB

*Free Space*

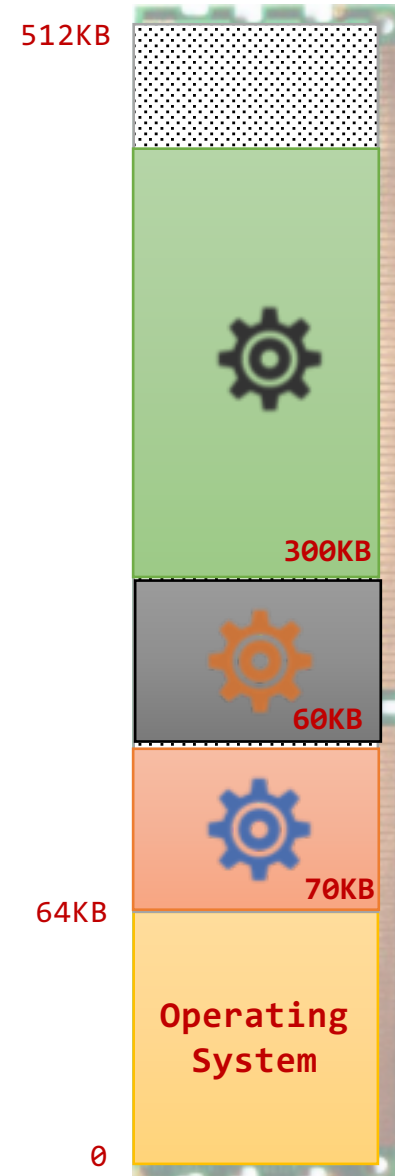
18KB

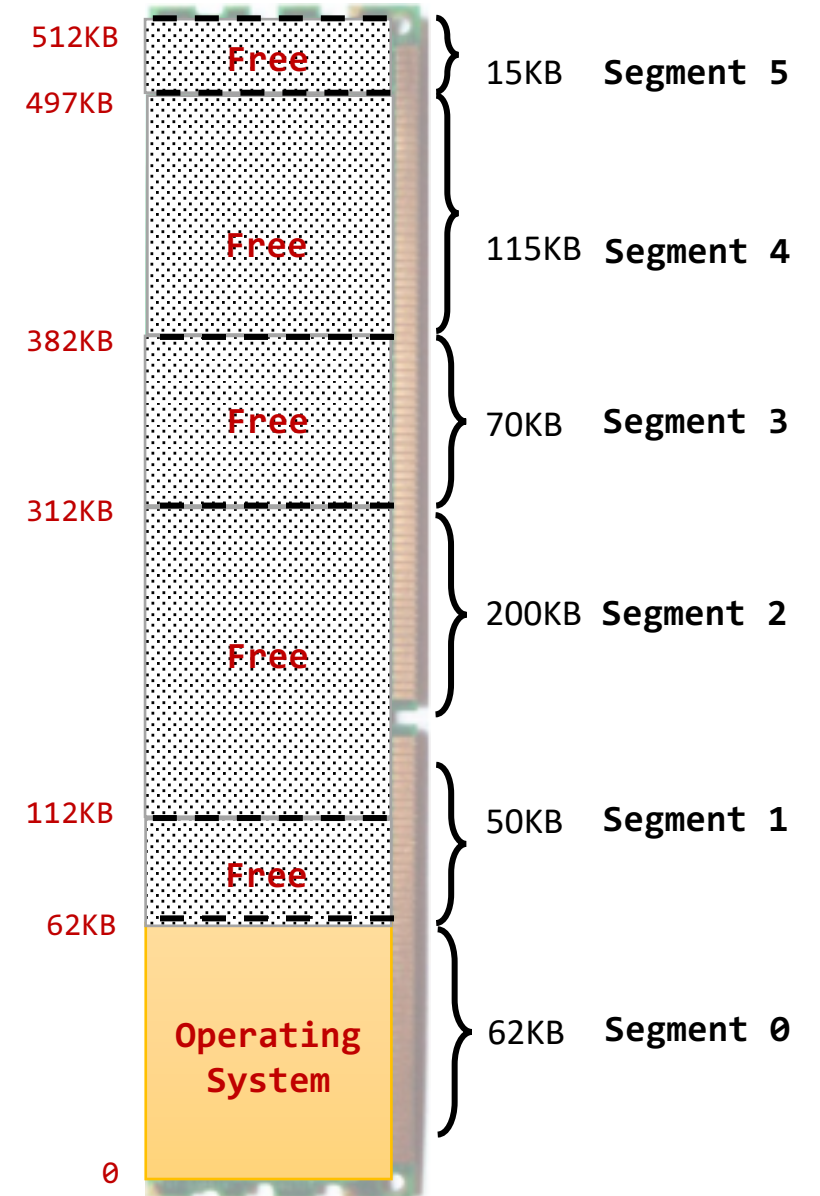
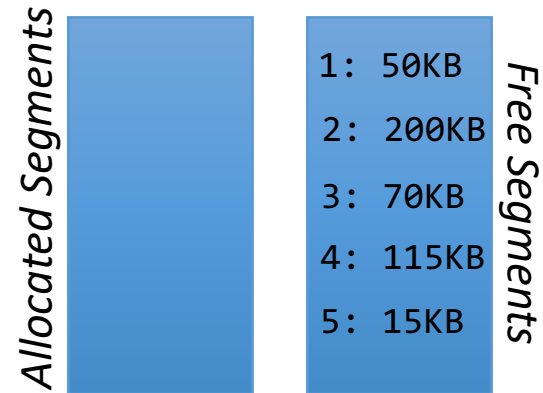
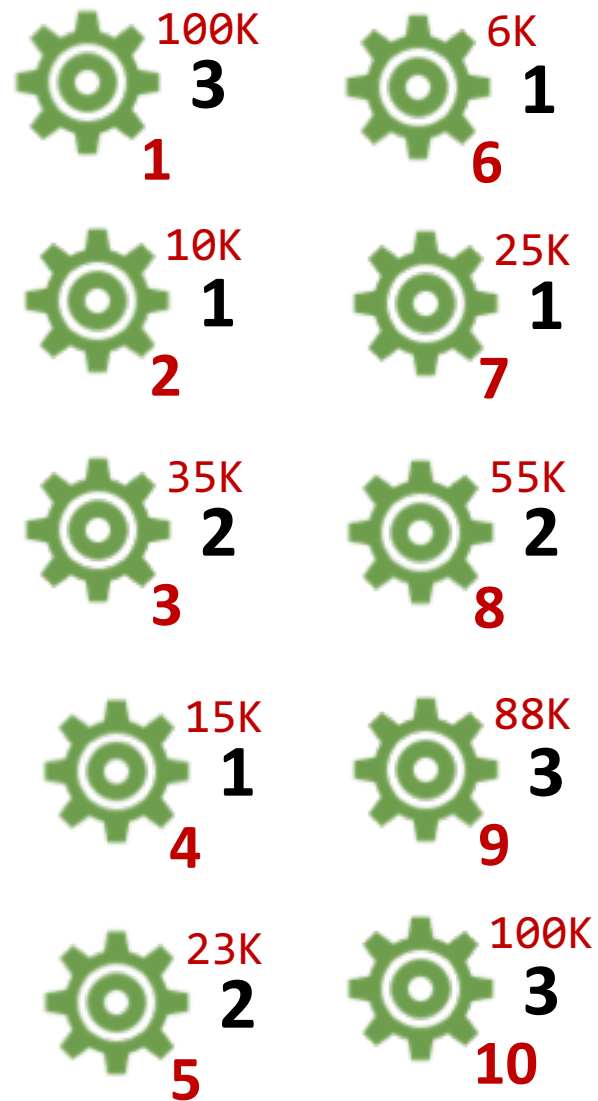
## Best-Fit

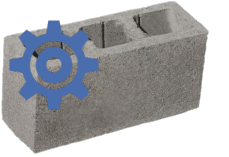
Allocate the ***smallest*** segment that is big enough



*Input Queue*

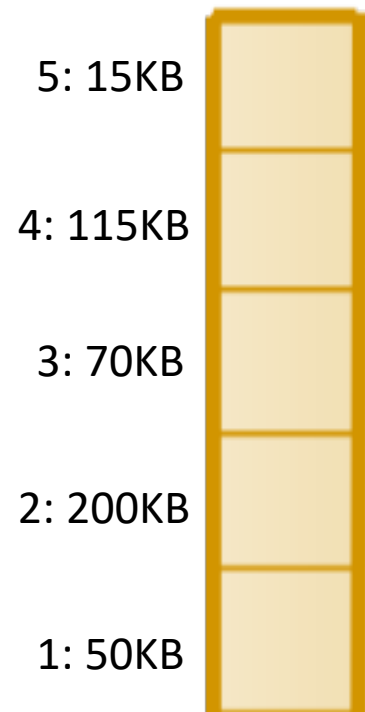
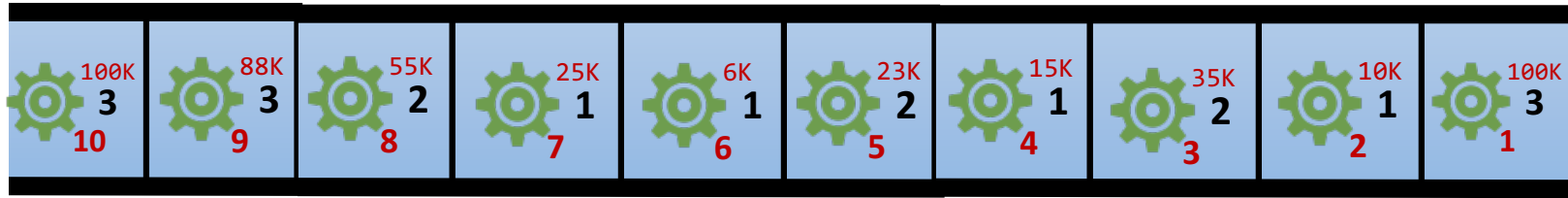




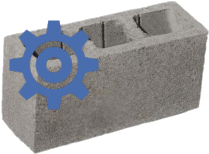


Contiguous Memory Allocation  
*Fixed-Size Partitioning*

*Input Queue*

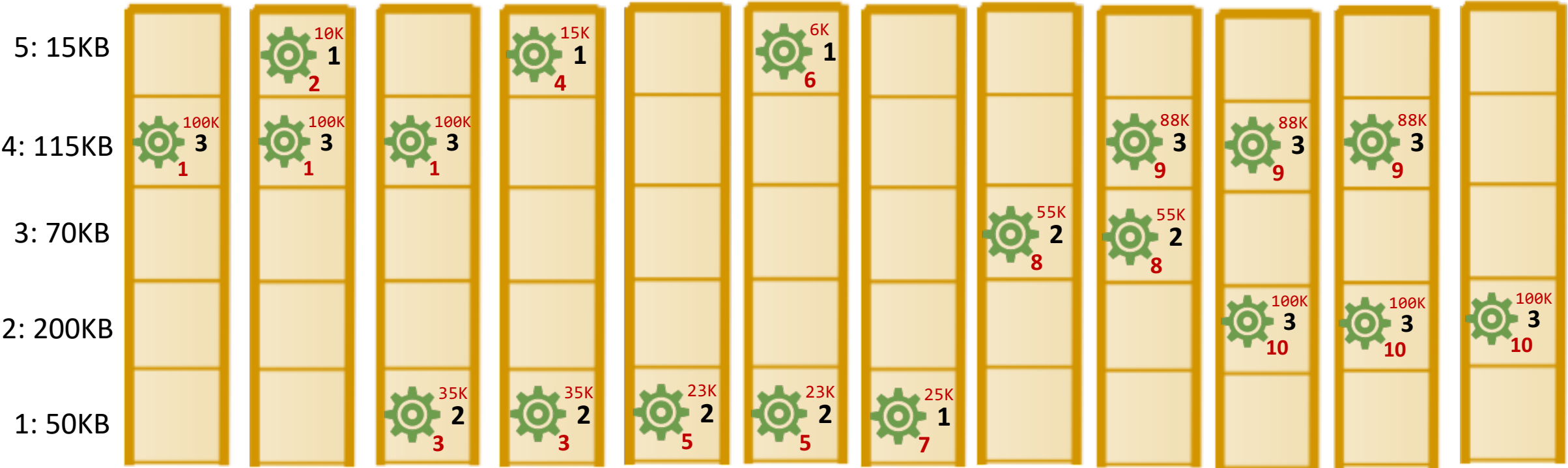
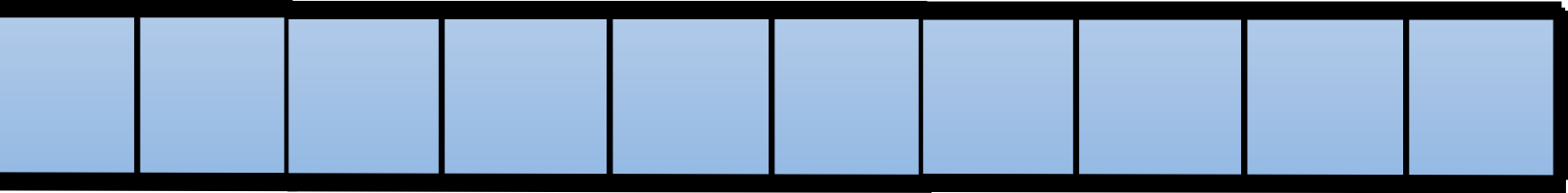


# Best-Fit



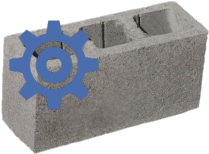
Contiguous Memory Allocation  
*Fixed-Size Partitioning*

*Input Queue*



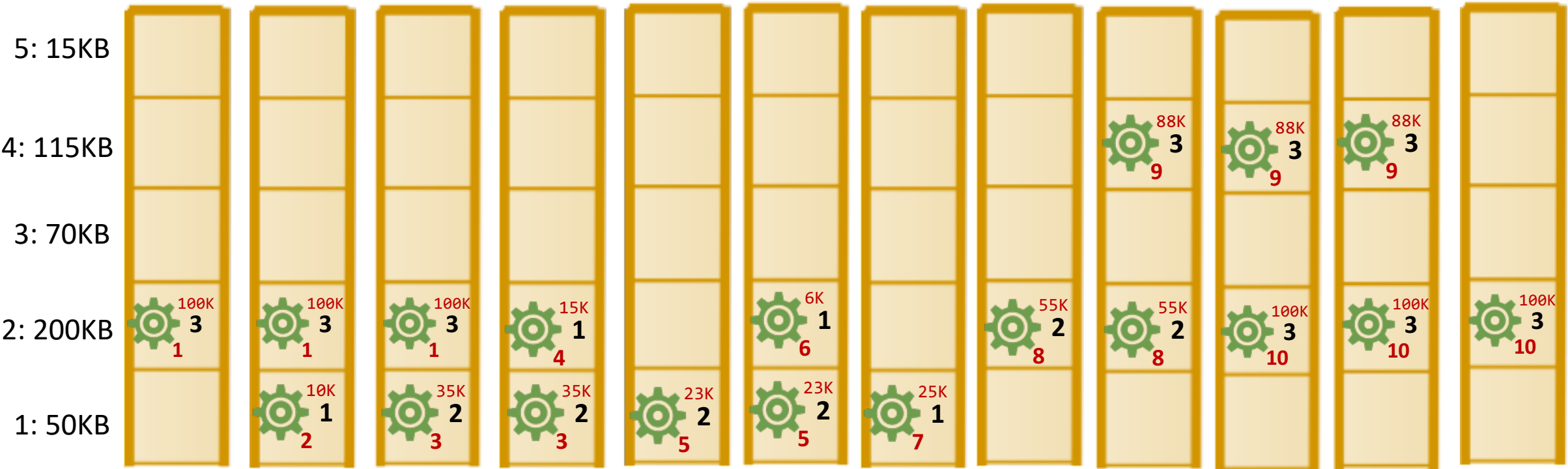
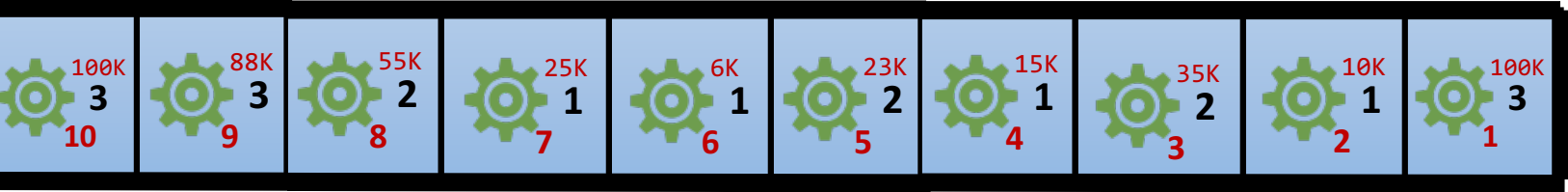


# First-Fit

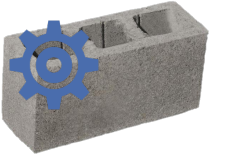


Contiguous Memory Allocation  
*Fixed-Size Partitioning*

*Input Queue*

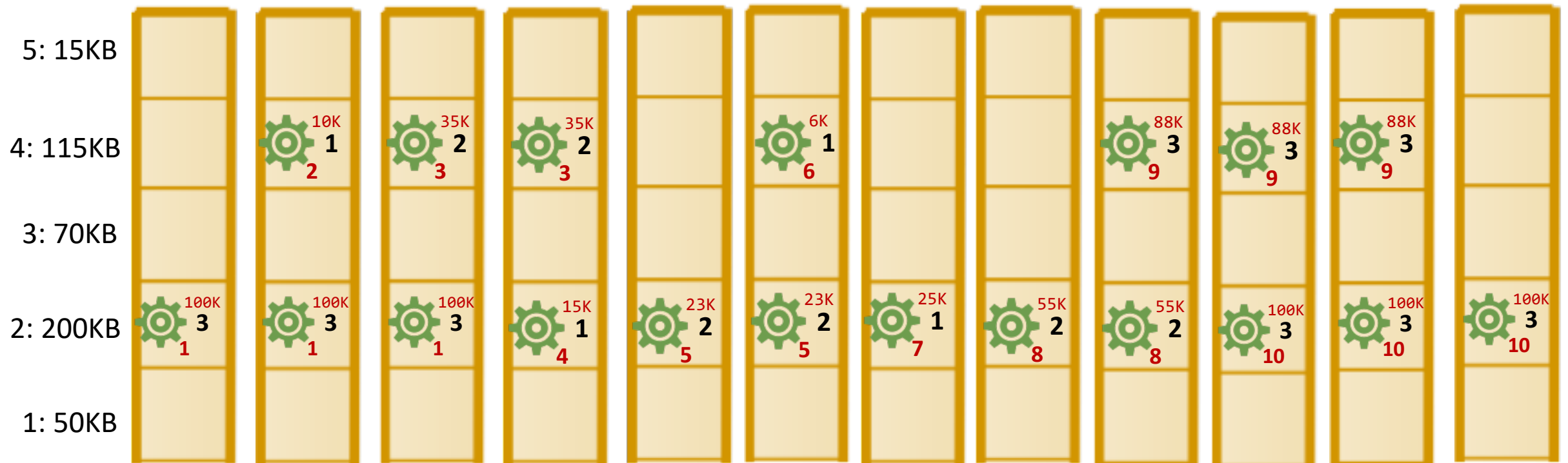
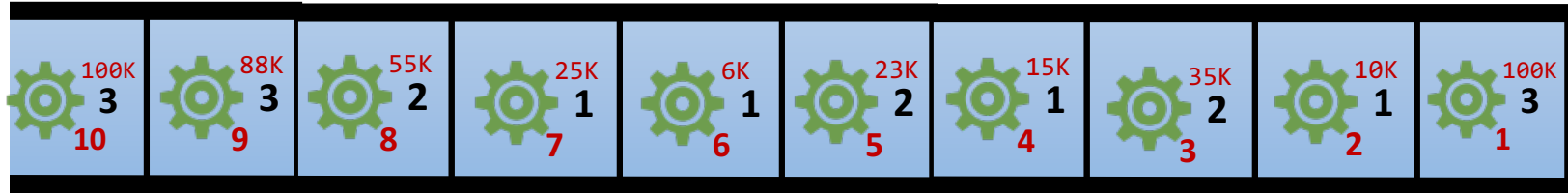


# Worst-Fit



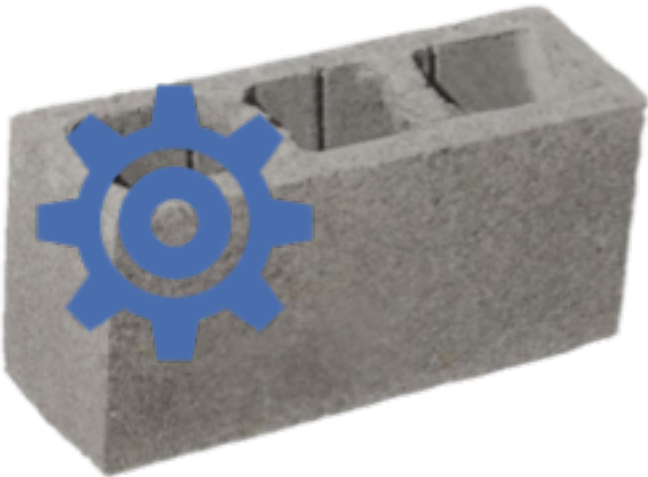
Contiguous Memory Allocation  
*Fixed-Size Partitioning*

*Input Queue*



# How does the OS **allocate** free space to processes to be loaded in memory?

“Memory-Management Schemes”



Contagious



Segmentation

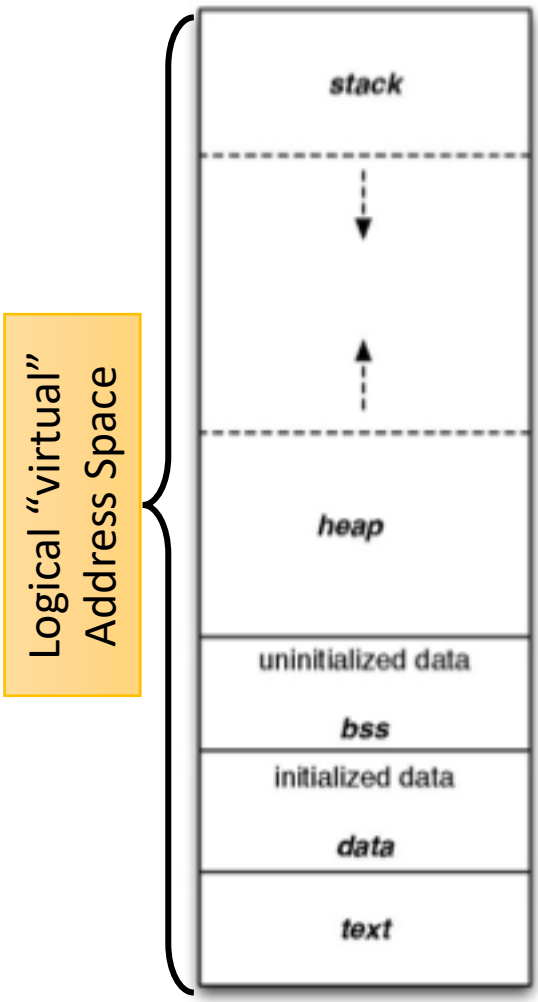


Paging

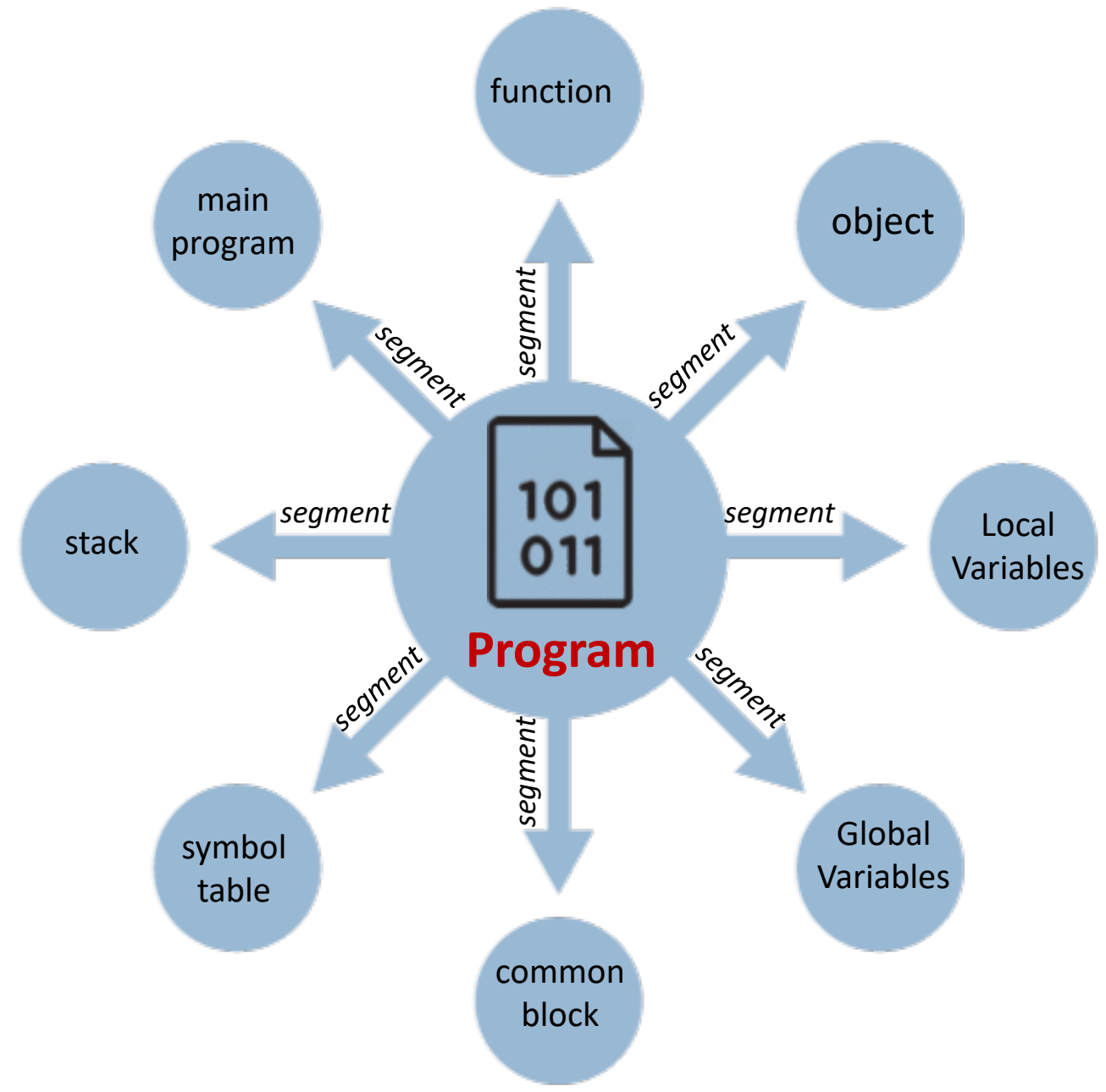


# Segmentation “Discontiguous”

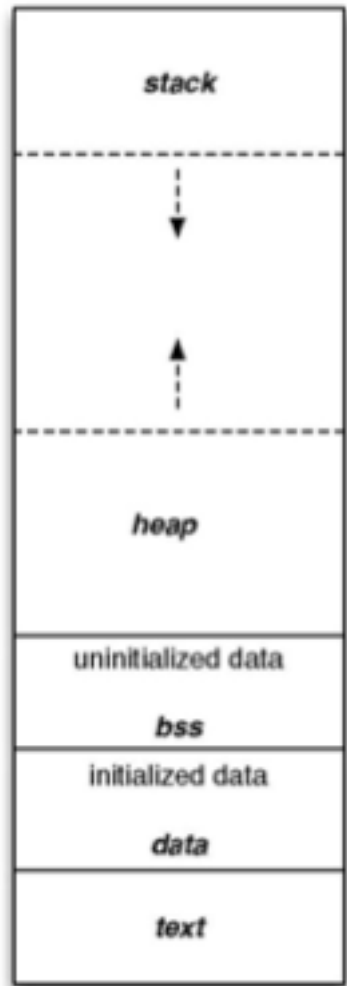
*Divide a process into segments (i.e., Code segment, data segment, stack segment, etc.) and place each segment into a partition of memory*



*Divide process into segments*

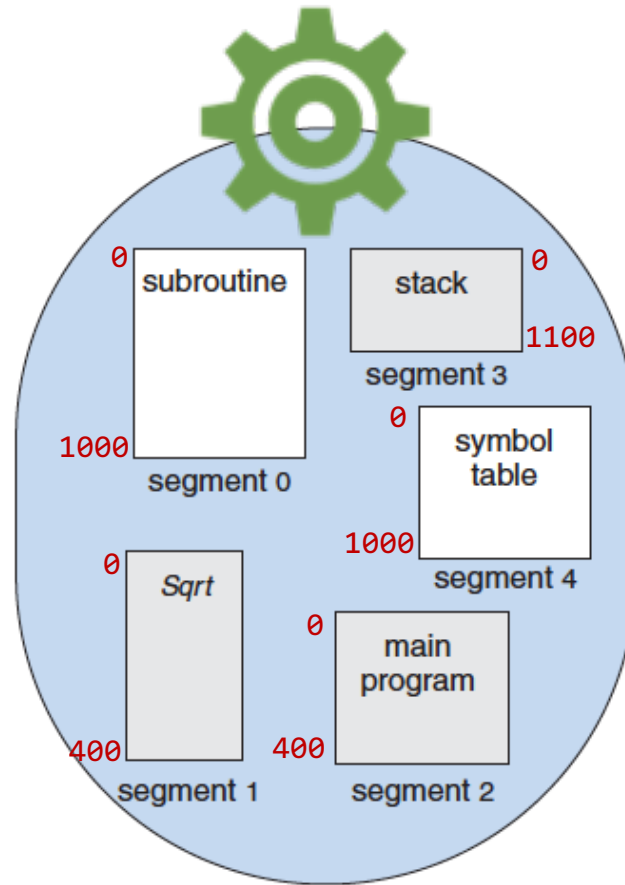


3900

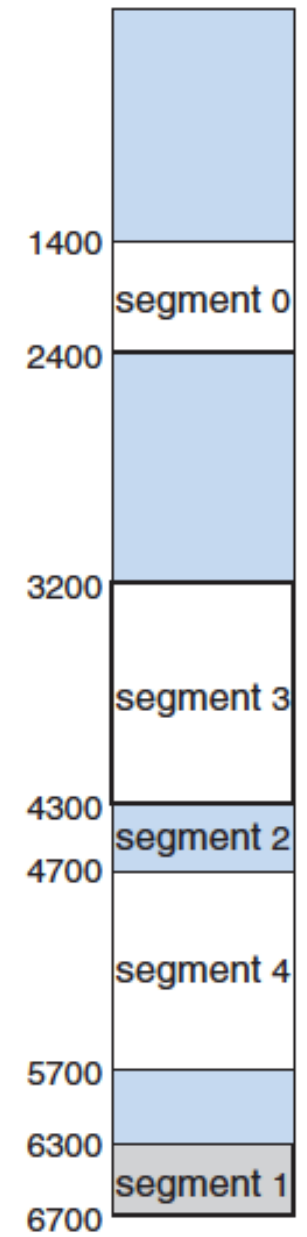


0

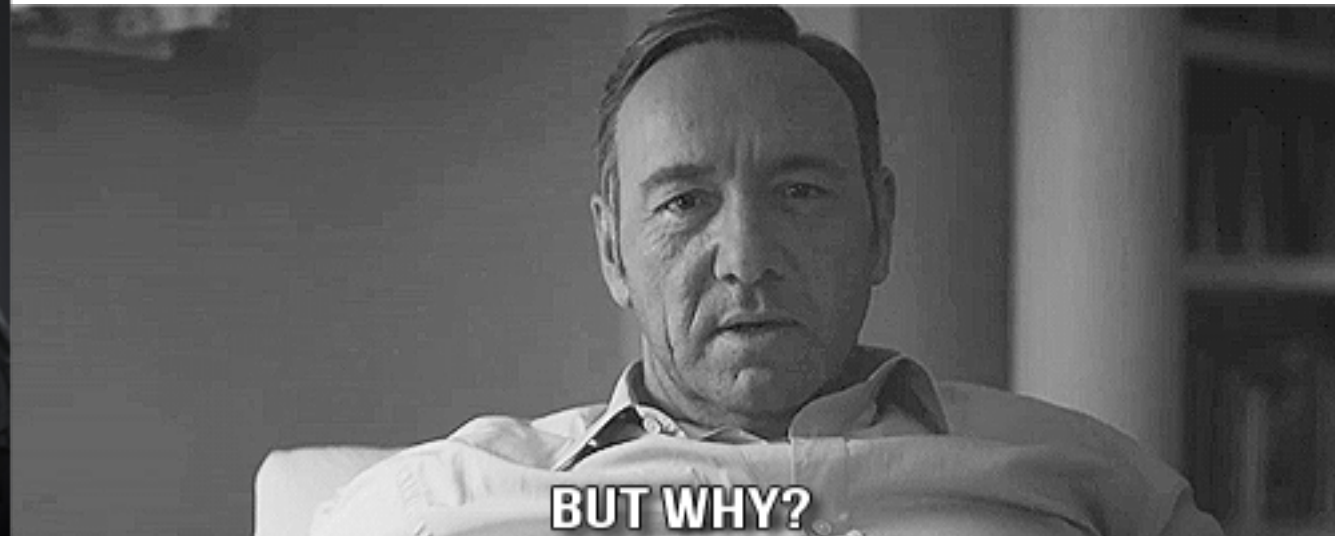
*Split into Segments*



*Place each segment in available partition*

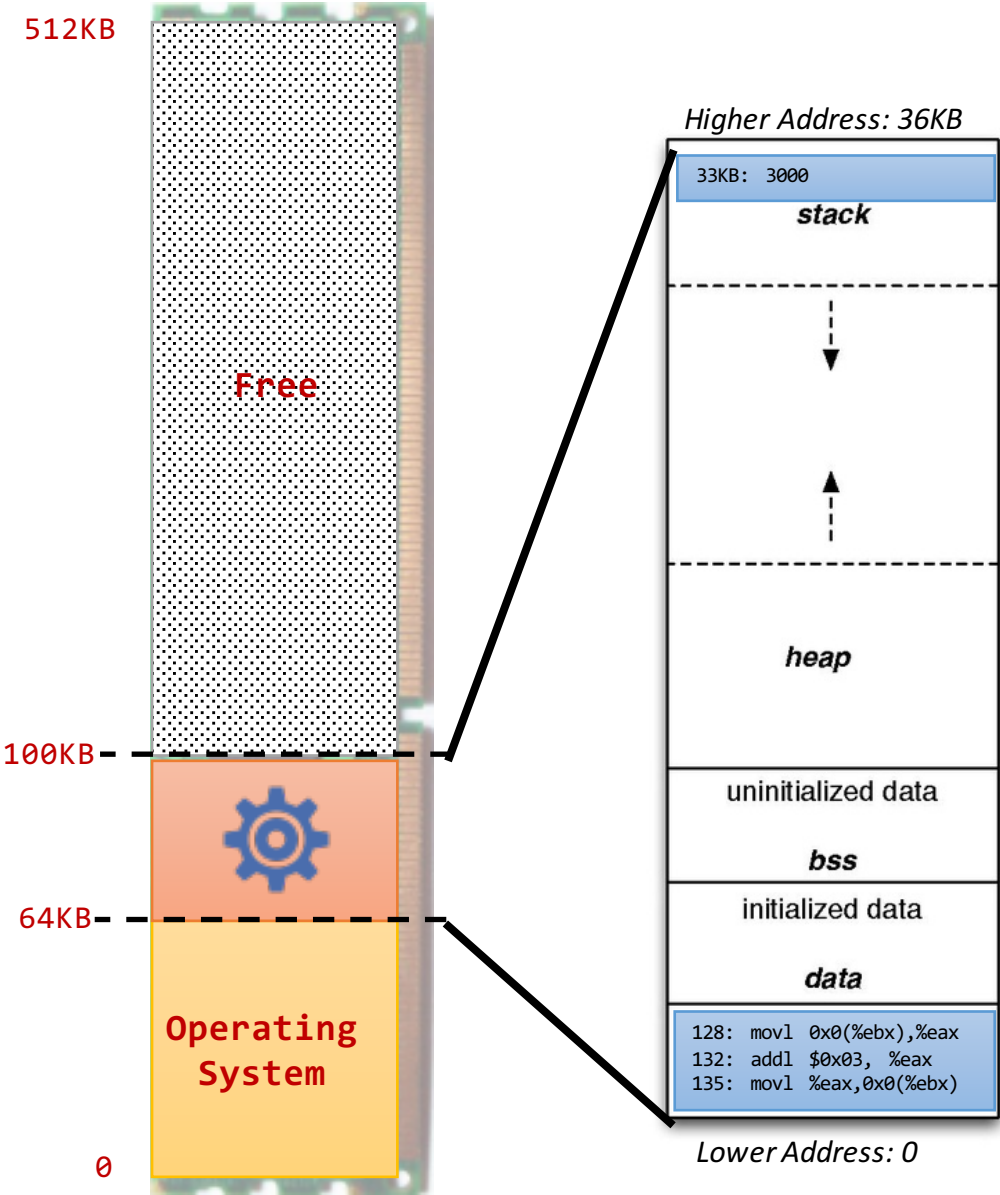


We need to track segments of the segmented process!



# Without Segmentation

*A process is loaded in contiguous memory section*





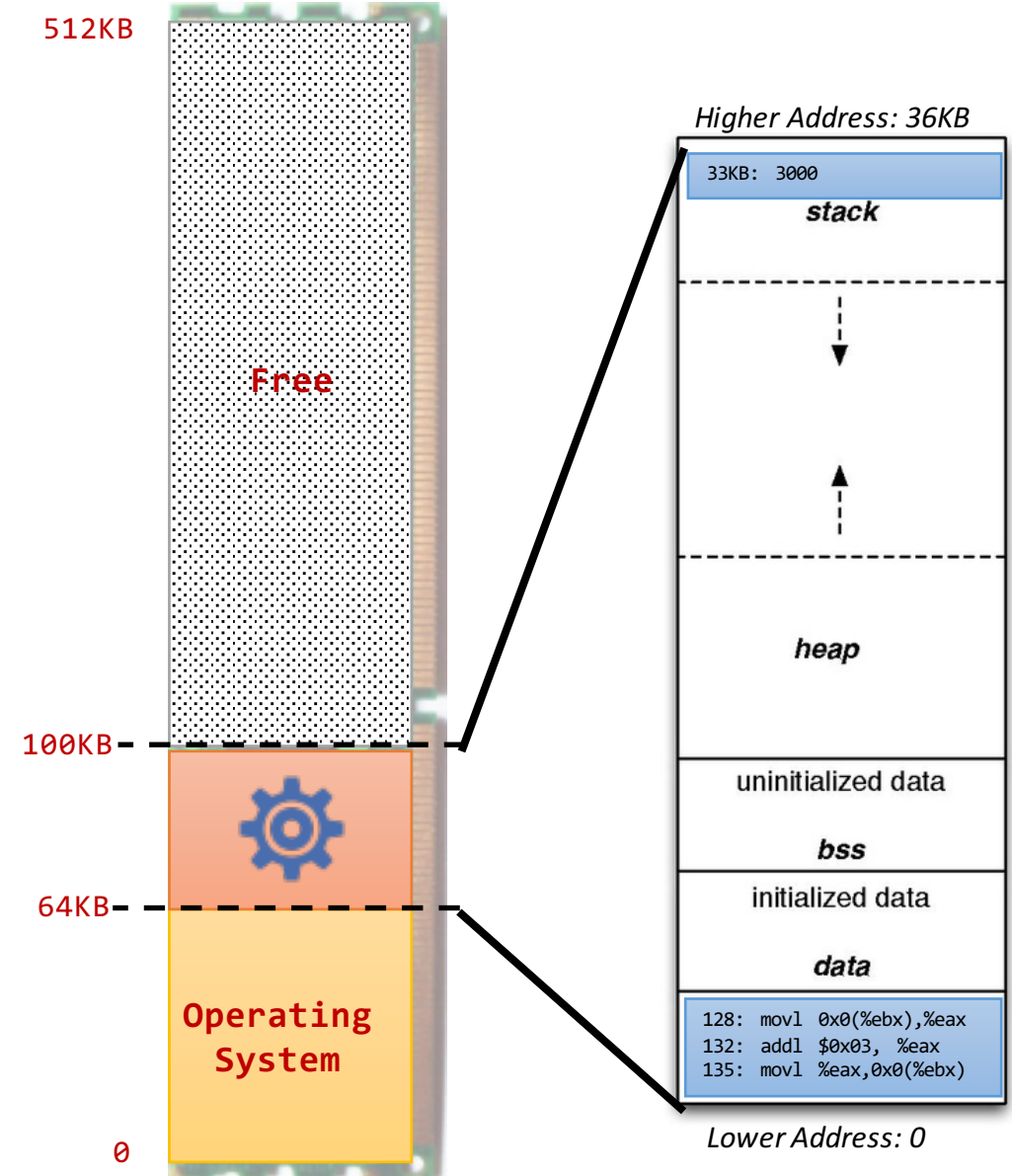
# Without Segmentation

A process is loaded in contiguous memory section

## Registers

eax	ebx
0	33KB

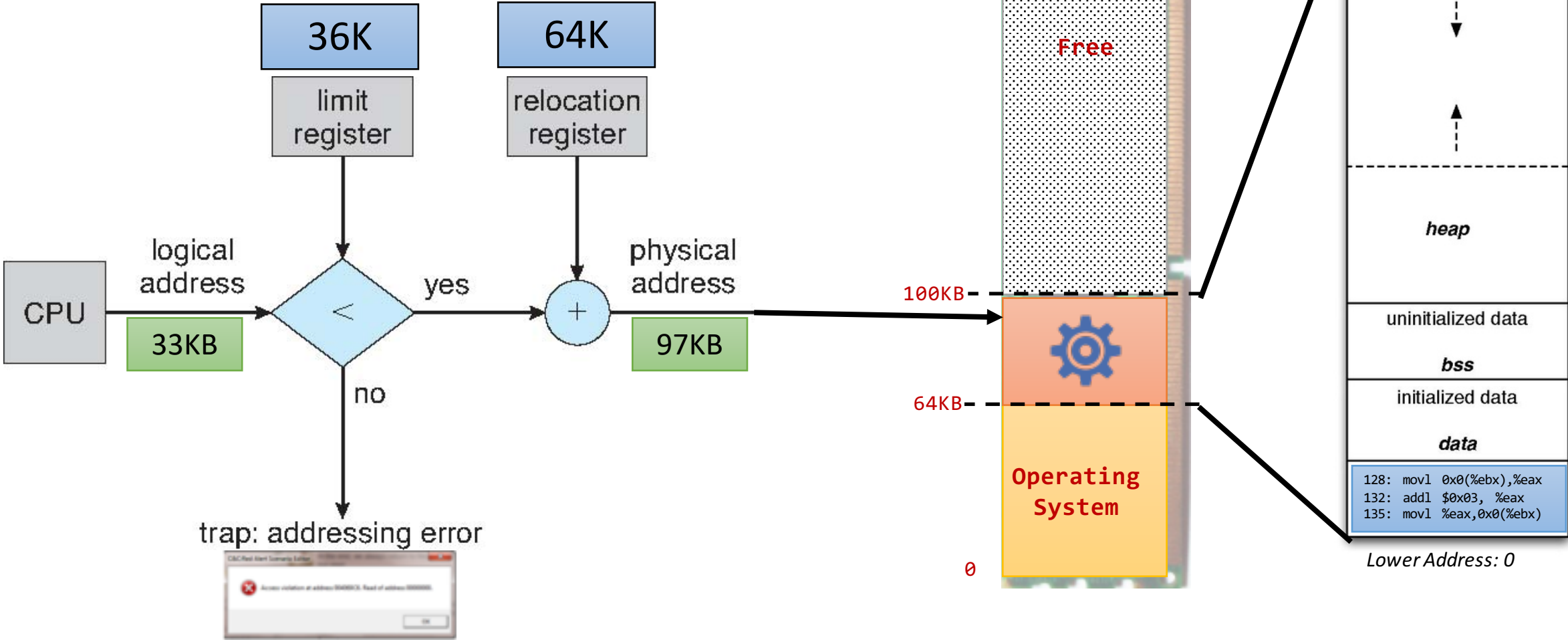
```
128: movl 0x0(%ebx),%eax
132: addl $0x03, %eax
135: movl %eax,0x0(%ebx)
```



# Without Segmentation

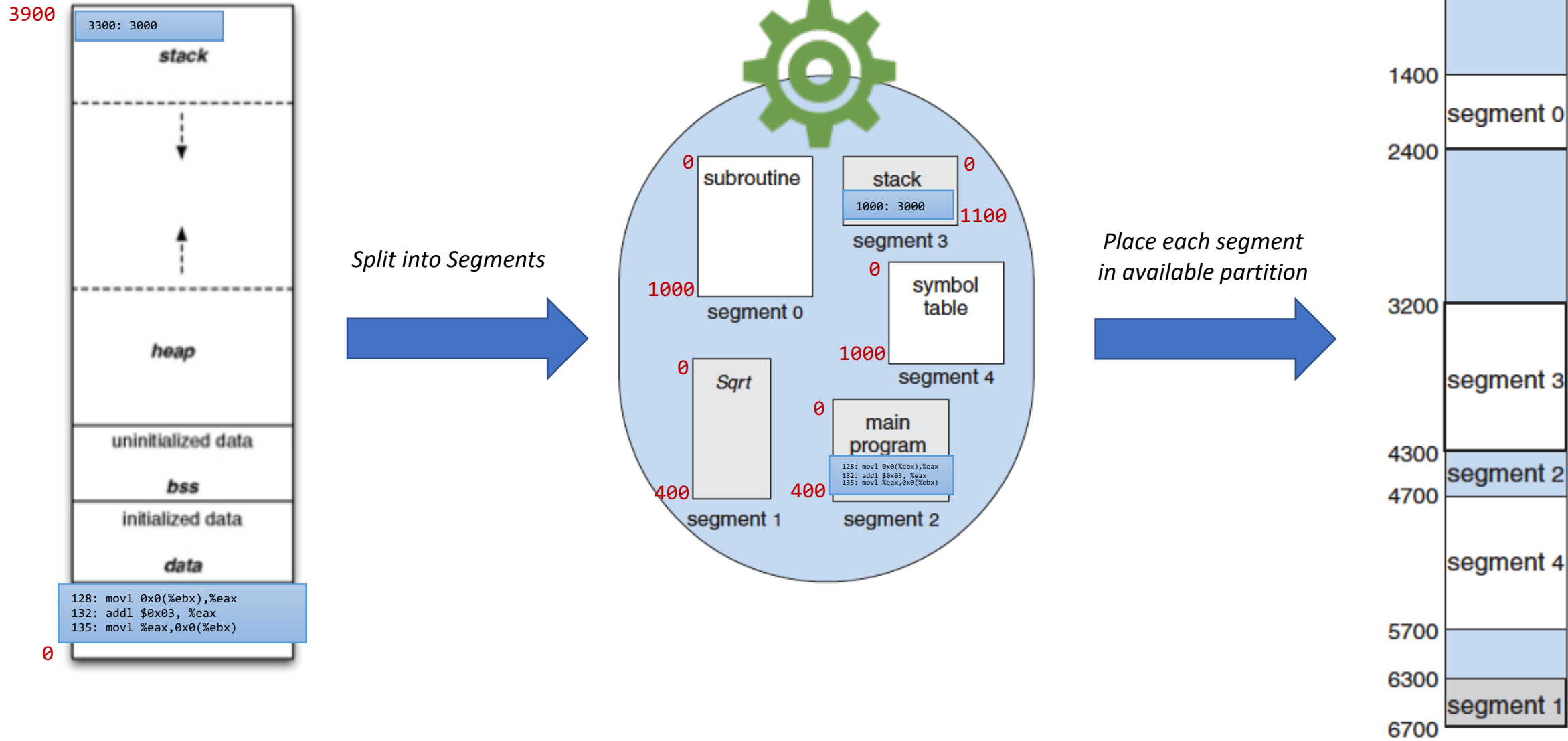
```
128: movl 0x0(%ebx),%eax
132: addl $0x03, %eax
135: movl %eax,0x0(%ebx)
```

Registers	
eax	ebx
0	33KB

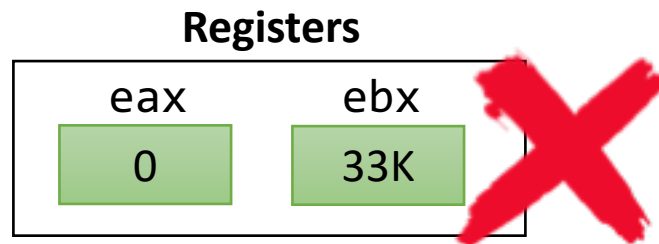


# With Segmentation

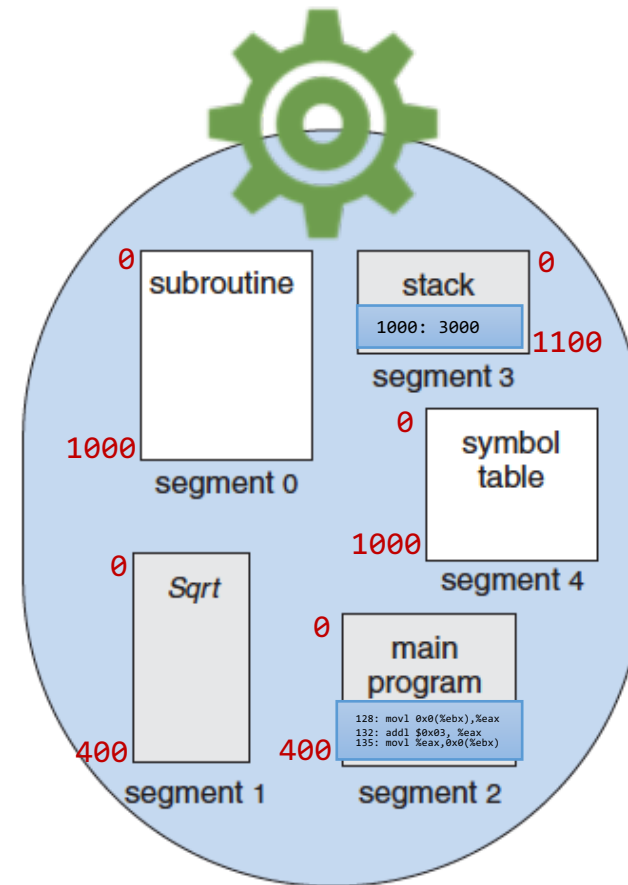
*A process is segmented and each segment is loaded in a separate partition*



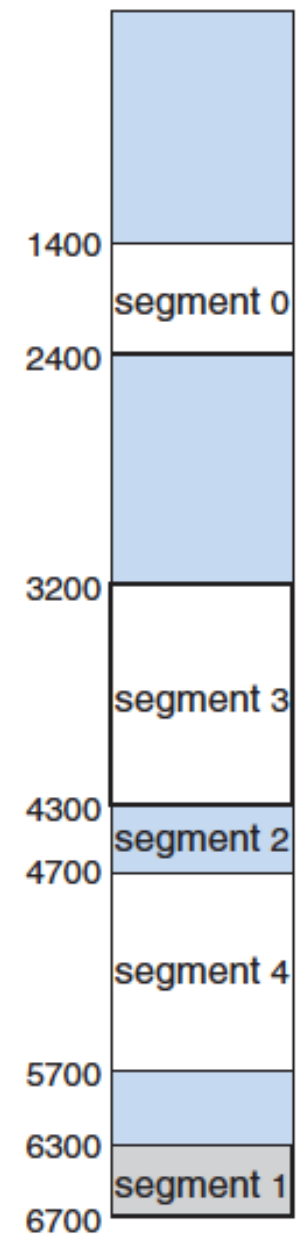
# With Segmentation



```
128: movl 0x0(%ebx),%eax
132: addl $0x03, %eax
135: movl %eax,0x0(%ebx)
```



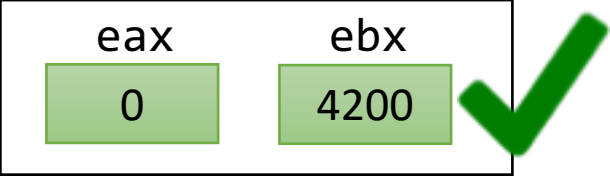
Logical Address Space



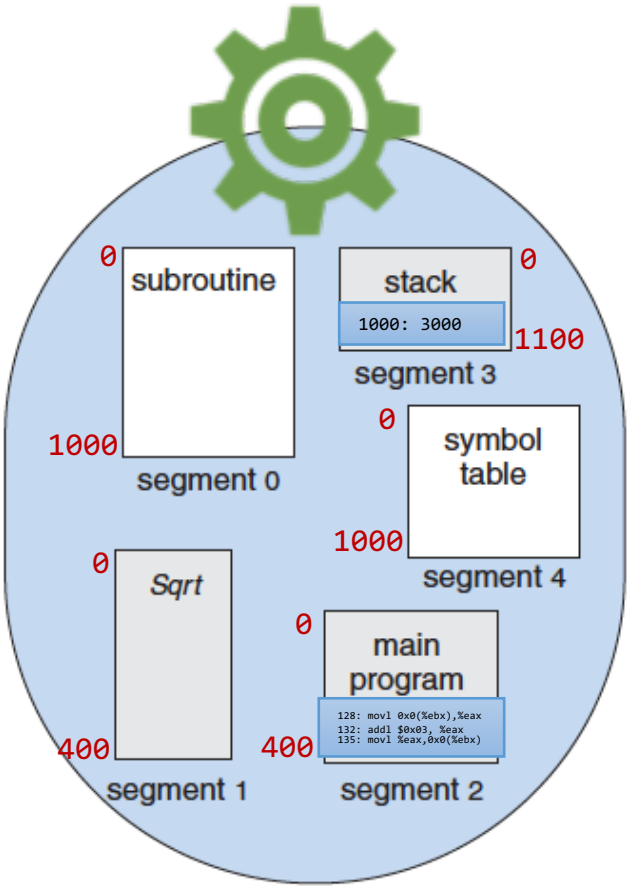
Physical Address Space

# With Segmentation

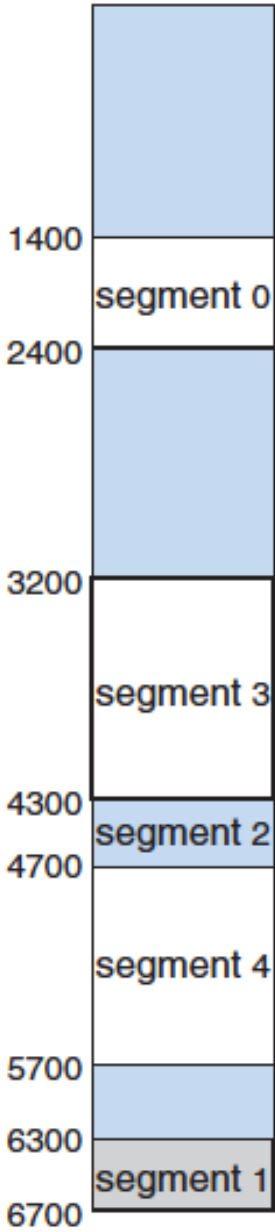
## Registers



```
128: movl 0x0(%ebx),%eax
132: addl $0x03, %eax
135: movl %eax,0x0(%ebx)
```



Logical Address Space



Physical Address Space

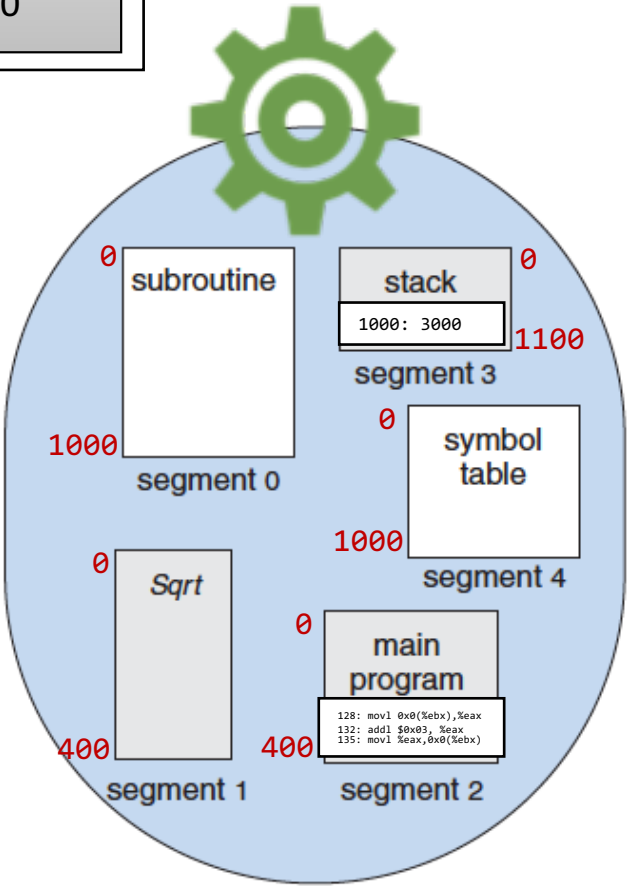
# With Segmentation

```
128: movl 0x0(%ebx),%eax
132: addl $0x03, %eax
135: movl %eax,0x0(%ebx)
```

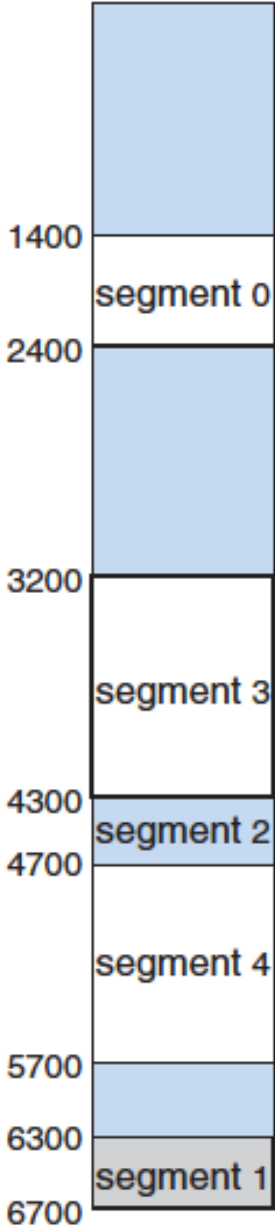
Registers		
eax	ebx	
0	3	1000

Segment Number	Offset Within Segment
3	1000

New Logical address

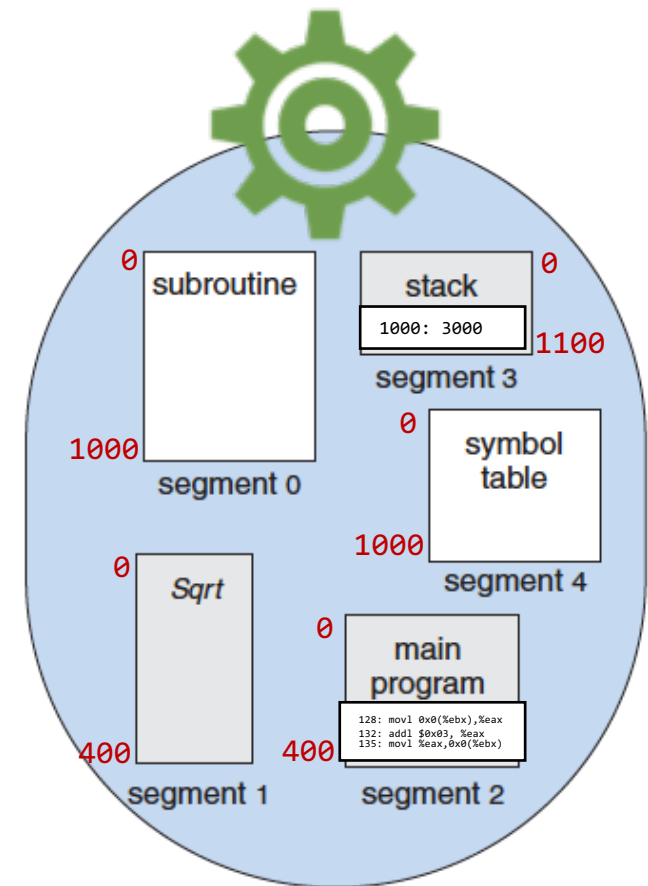
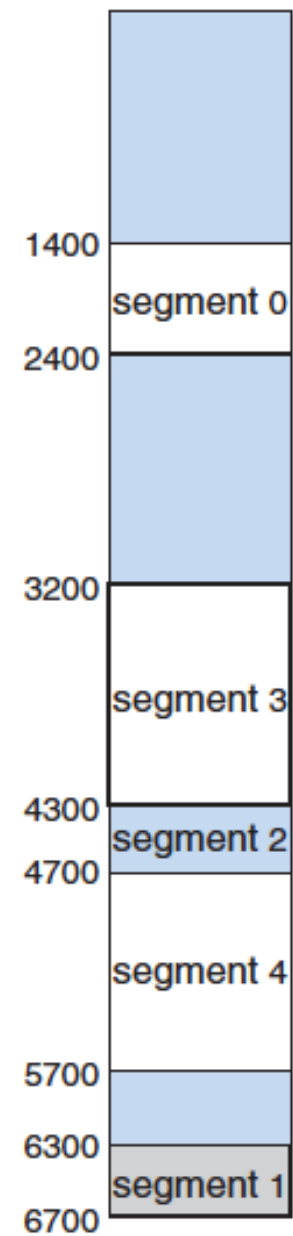
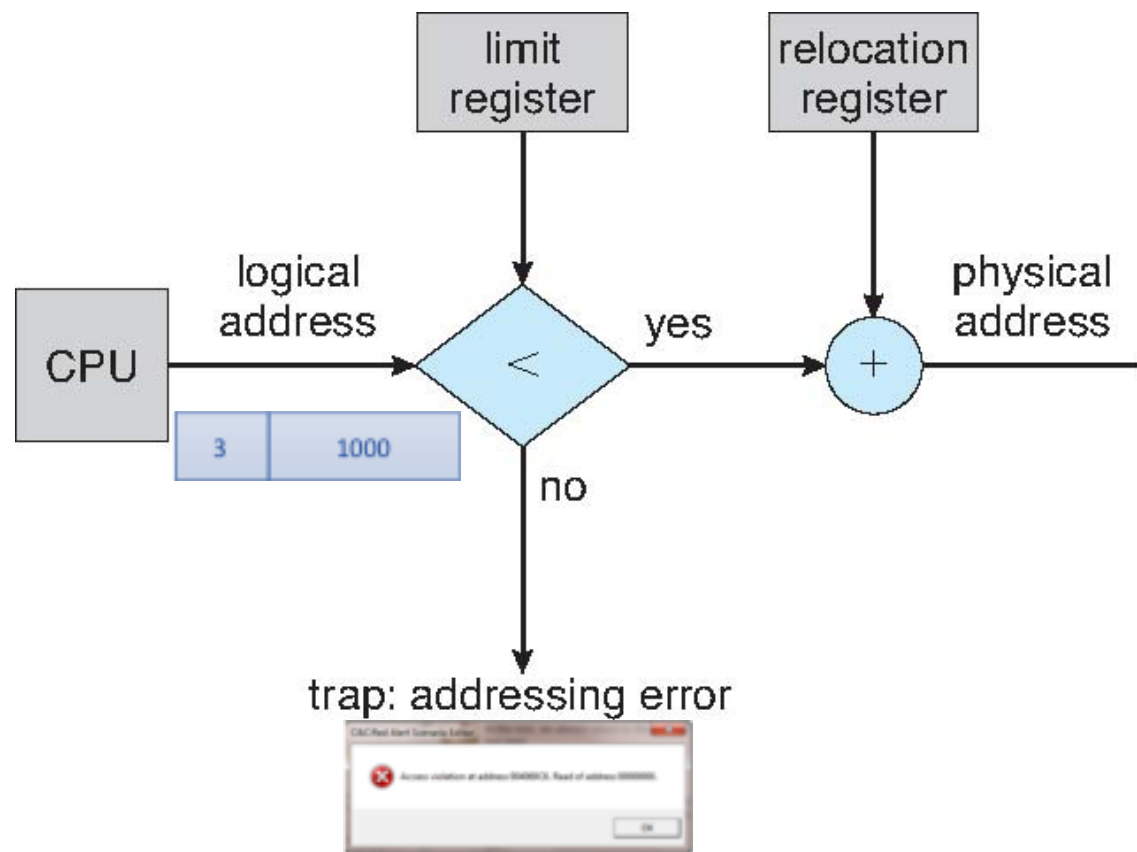


Logical Address Space



Physical Address Space

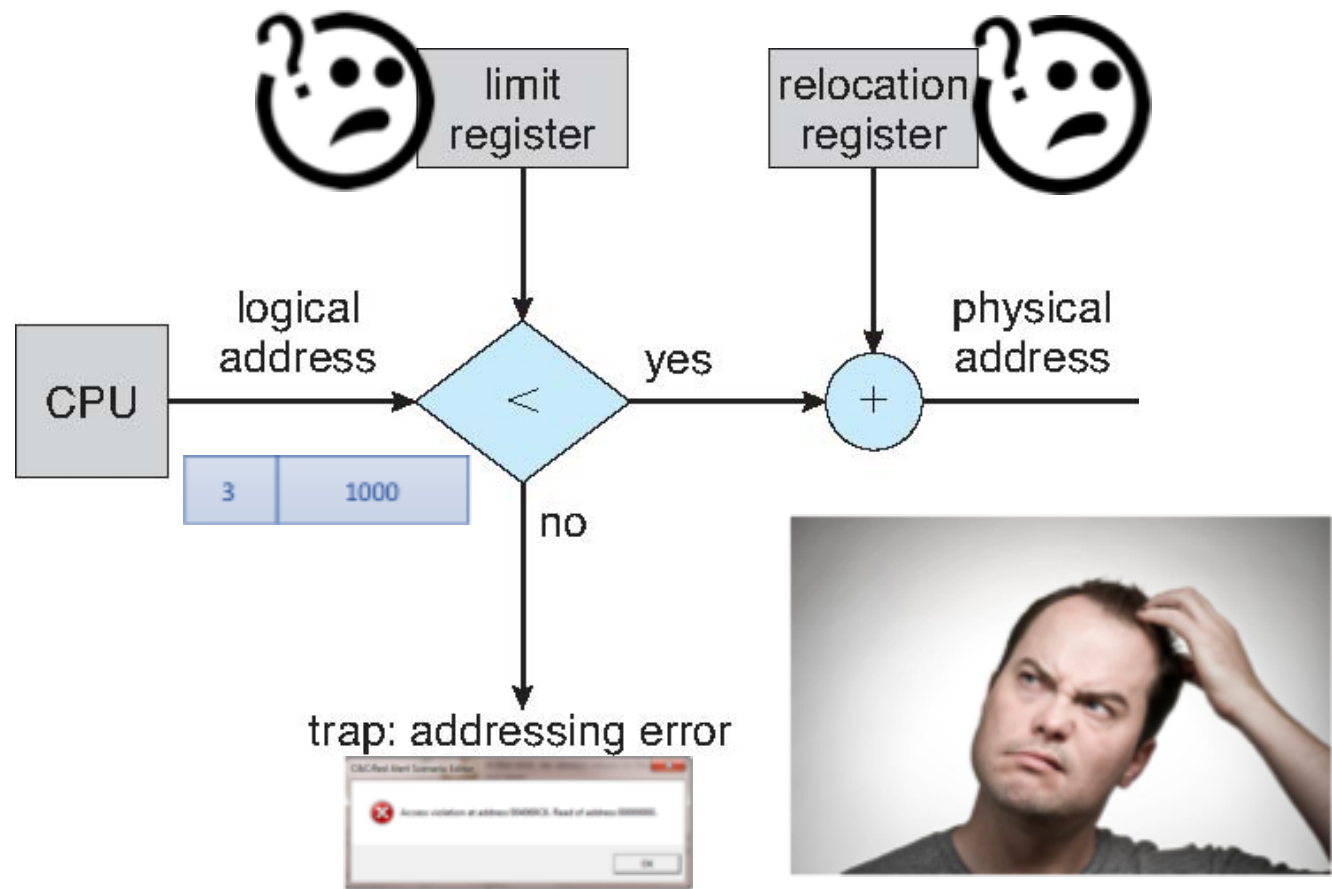
# With Segmentation



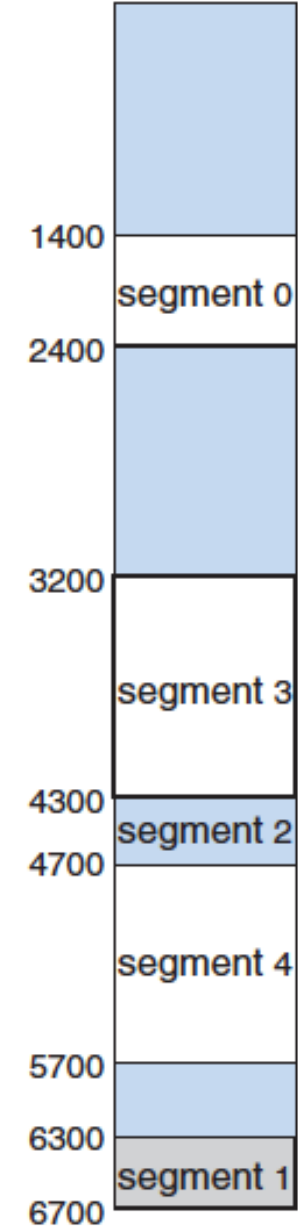
Logical Address Space

Physical Address Space

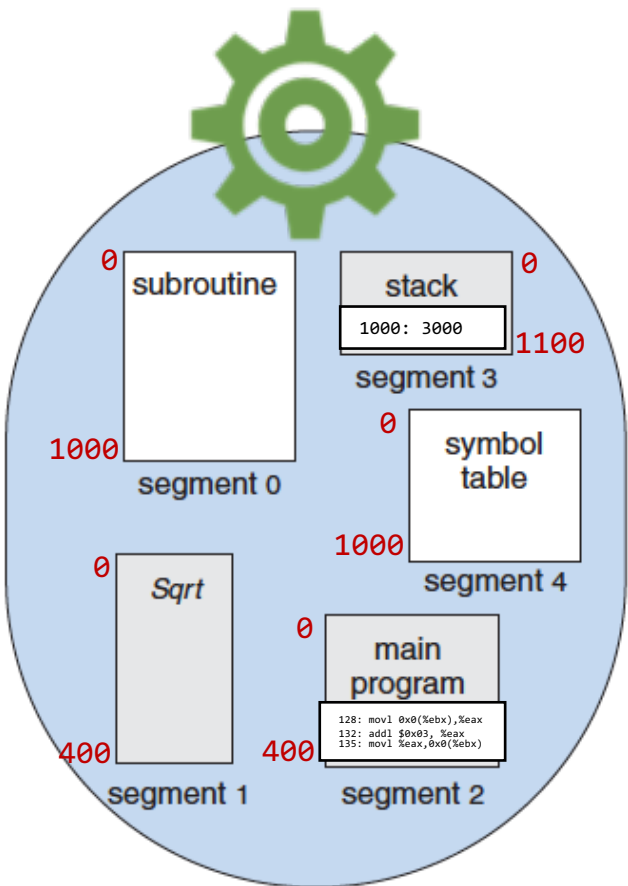
# With Segmentation



*Need a new Hardware Support*



Physical Address Space



Logical Address Space



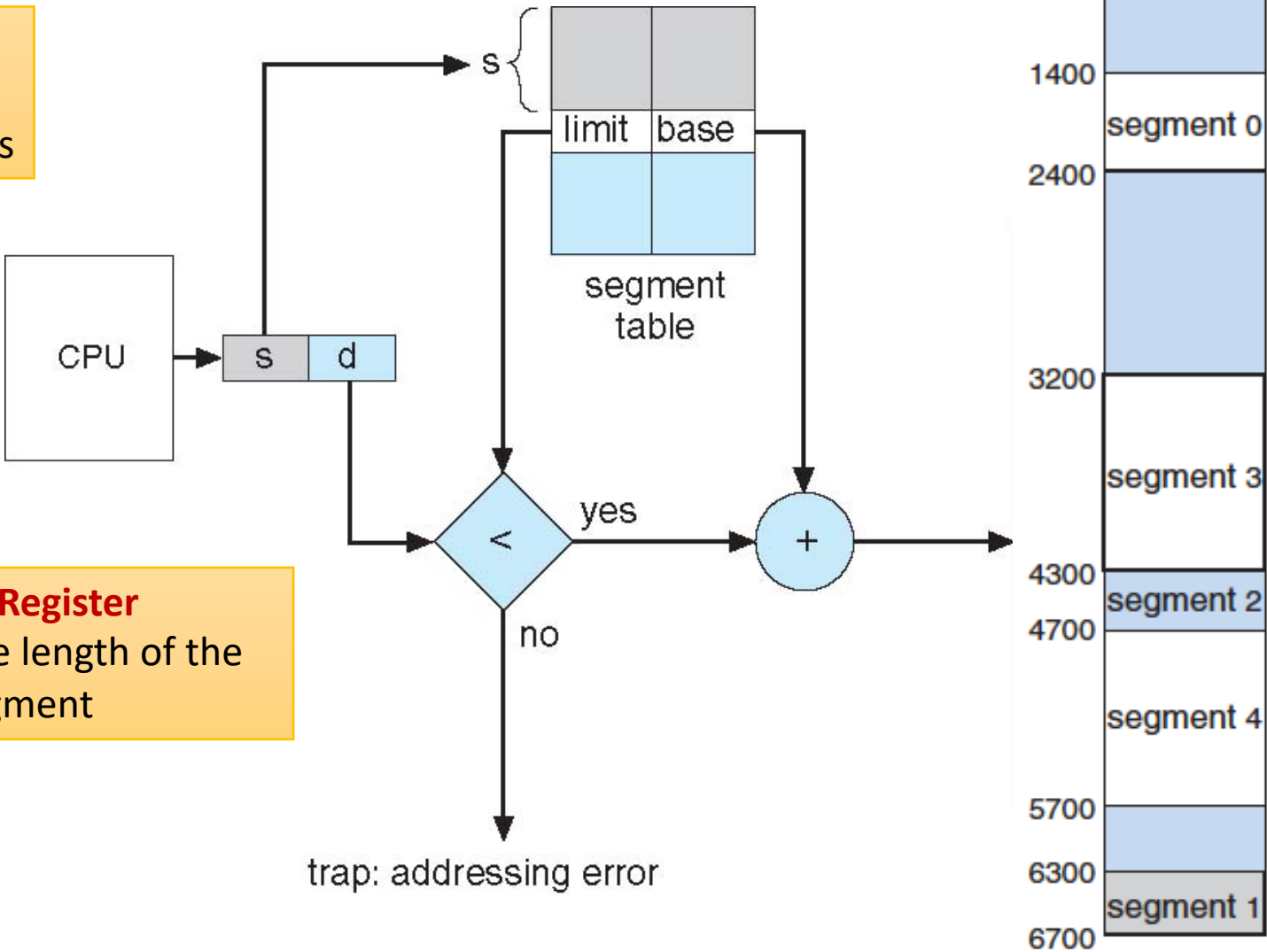
# Segmentation Hardware

**Segment Table**

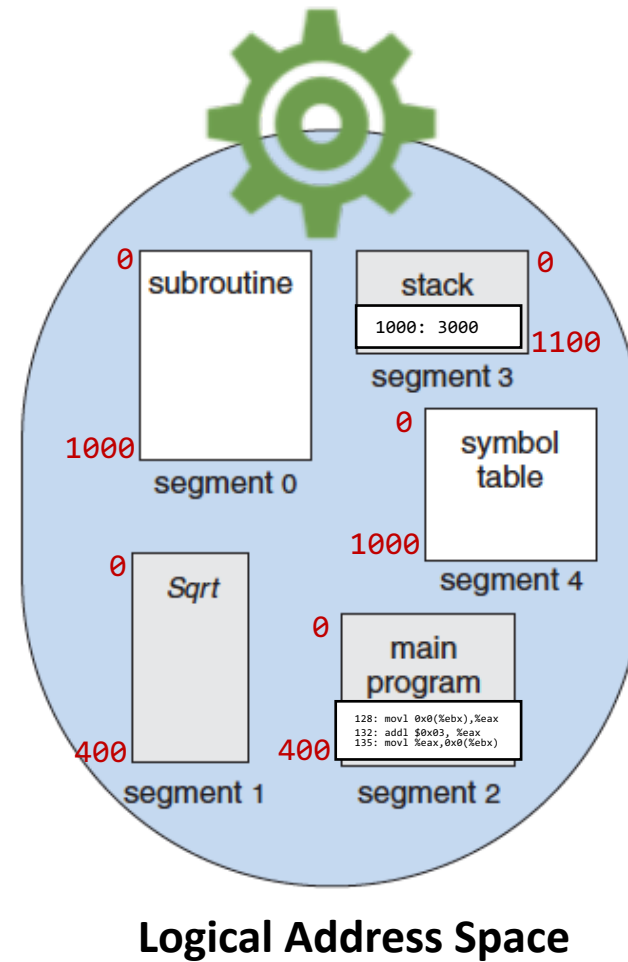
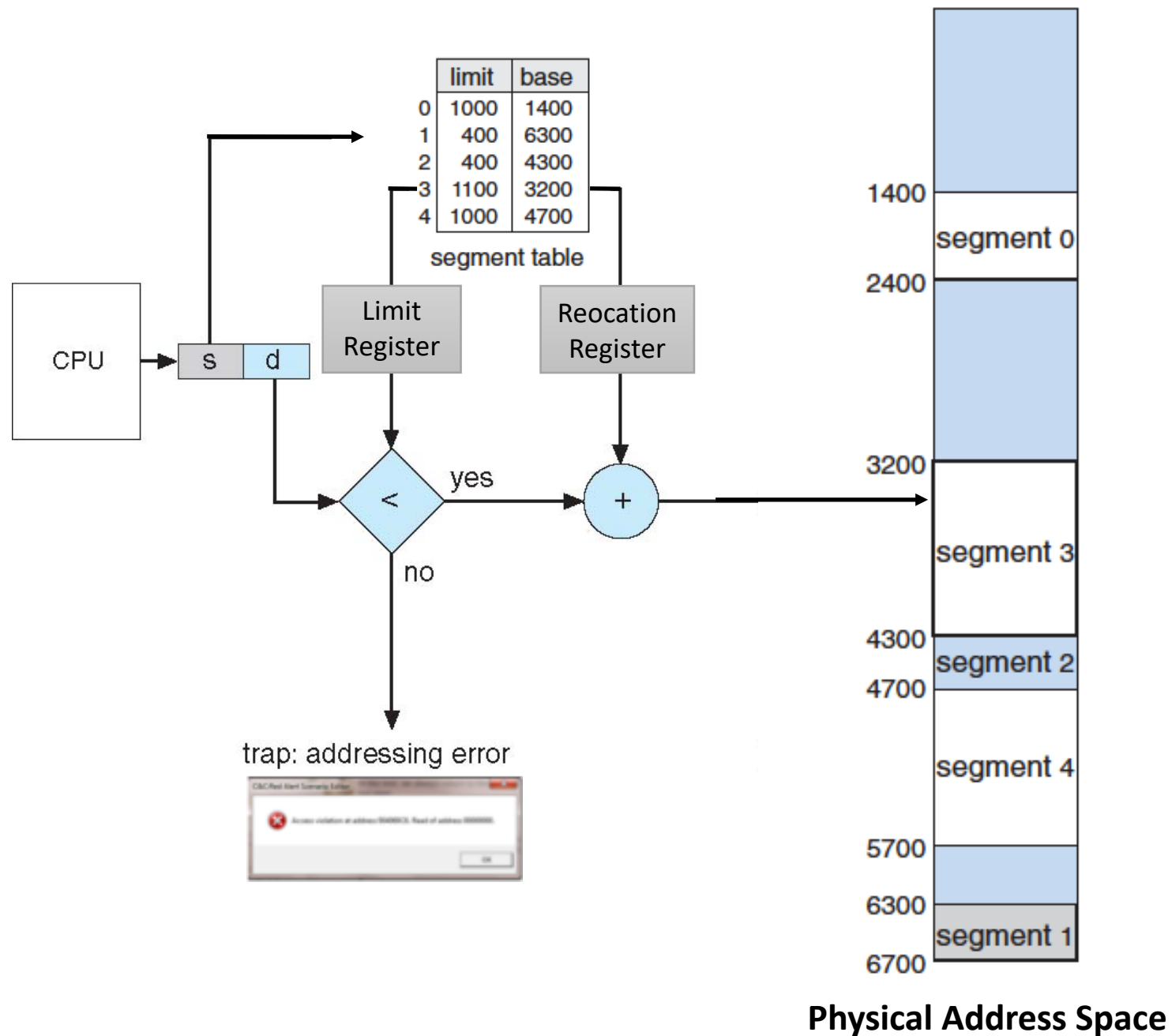
Maps two-dimensional programmer-defined addresses into one-dimensional physical address

**Base “Relocation” Register**  
*contains the starting physical address where the segments reside in memory*

**Limit Register**  
specifies the length of the segment



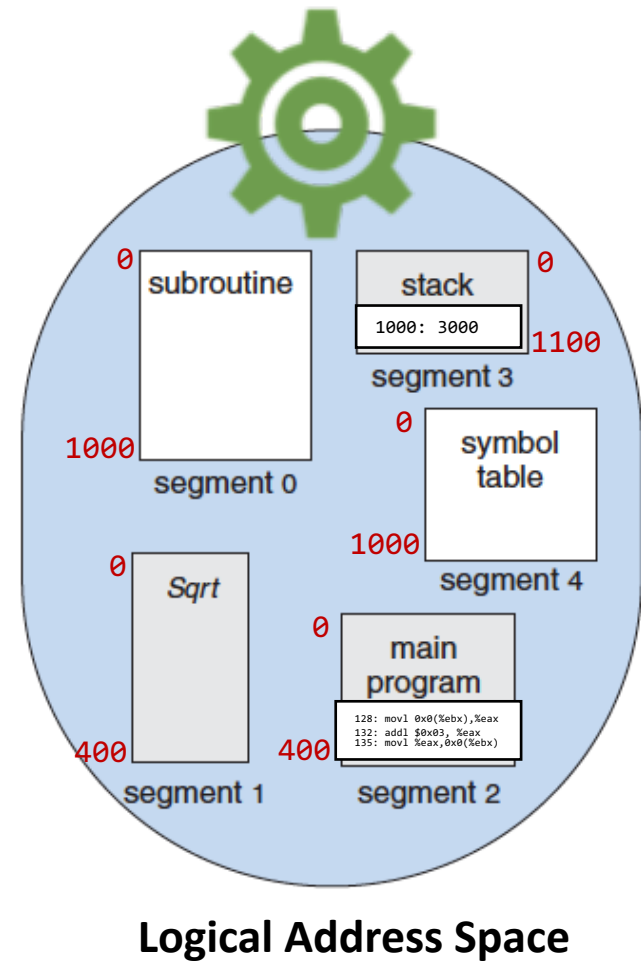
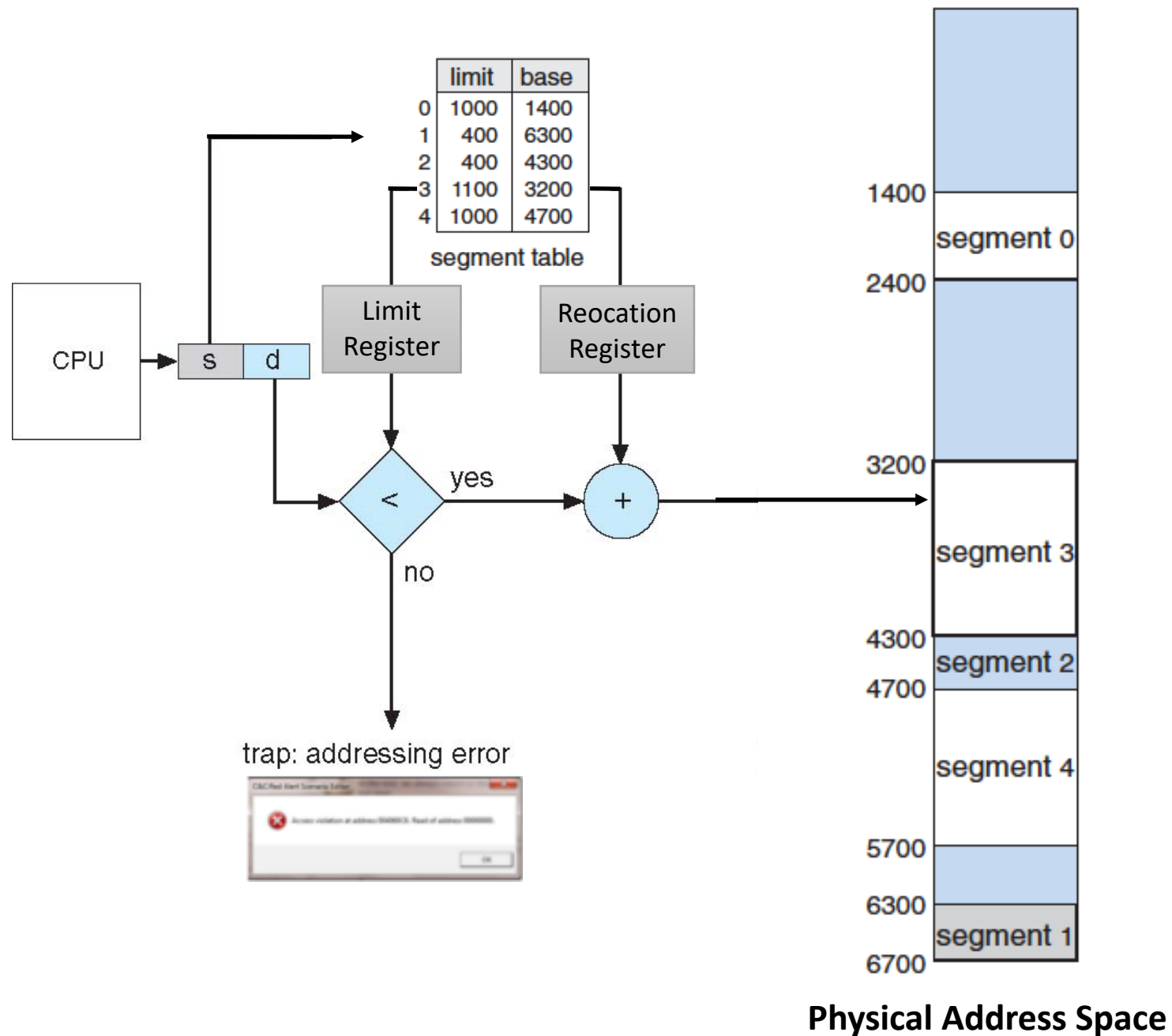
# With Segmentation





Is the Memory protected from unintended accesses?

# With Segmentation



# Can we have external fragmentation?

*What is the solution?*



## Process 1

Logical "Virtual" Address	Memory Content
0	a
1	b
2	c
3	d
4	e
5	f
6	g
7	h
8	i
9	j
10	k
11	l
12	m
13	n
14	o
15	p



## Process 1

Segment	Logical "Virtual" Address	Memory Content
0	0	a
	1	b
	2	c
	3	d
1	4	e
	5	f
	6	g
	7	h
2	8	i
	9	j
	10	k
	11	l
3	12	m
	13	n
	14	o
	15	p

Segment	Limit Register	Base Register
0	4	4
1	4	0
2	4	28
3	4	12



## Process 2

Logical "Virtual" Address	Memory Content
0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H
8	I
9	J
10	K
11	L



## Process 2

Segment	Logical "Virtual" Address	Memory Content
0	0	A
	1	B
	2	C
	3	D
1	4	E
	5	F
	6	G
	7	H
2	8	I
	9	J
	10	K
	11	L

Segment	Limit Register	Base Register
0	4	16
1	4	8
2	4	20



## Process 1

Segment	Logical "Virtual" Address	Memory Content
0	0	a
	1	b
	2	c
	3	d
1	4	e
	5	f
	6	g
	7	h
2	8	i
	9	j
	10	k
	11	l
3	12	m
	13	n
	14	o
	15	p

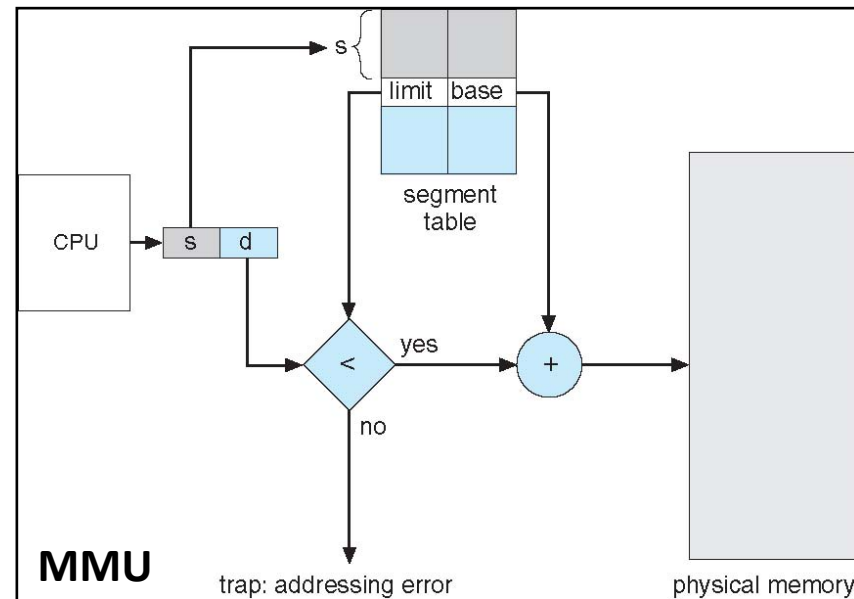
Segment	Limit Register	Base Register
0	4	4
1	4	0
2	4	28
3	4	12



## Process 2

Segment	Logical "Virtual" Address	Memory Content
0	0	A
	1	B
	2	C
	3	D
1	4	E
	5	F
	6	G
	7	H
2	8	I
	9	J
	10	K
	11	L

Segment	Limit Register	Base Register
0	4	16
1	4	8
2	4	20



0	
4	
8	
12	
16	
20	
24	
28	





## Process 1

Segment	Logical "Virtual" Address	Memory Content
0	0	a
	1	b
	2	c
	3	d
1	4	e
	5	f
	6	g
	7	h
2	8	i
	9	j
	10	k
	11	l
3	12	m
	13	n
	14	o
	15	p

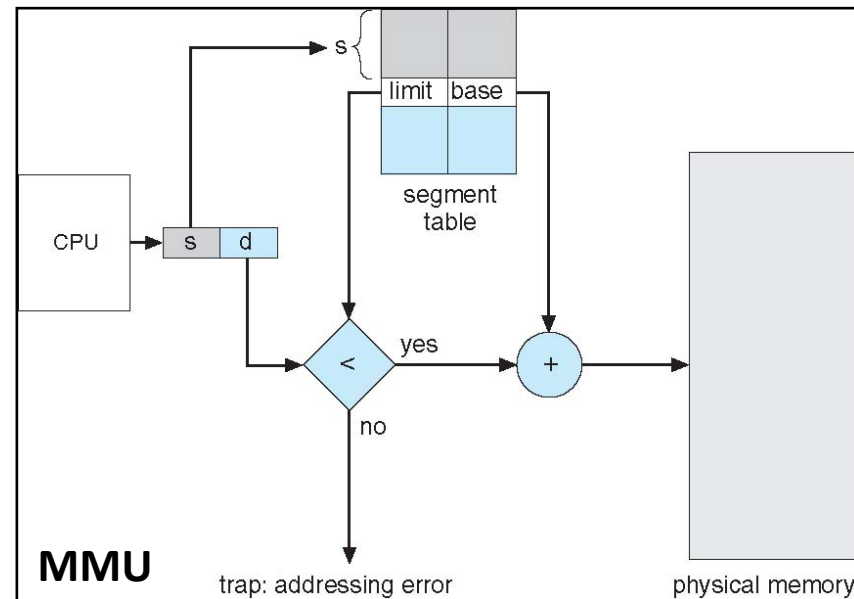
Segment	Limit Register	Base Register
0	4	4
1	4	0
2	4	28
3	4	12



## Process 2

Segment	Logical "Virtual" Address	Memory Content
0	0	A
	1	B
	2	C
	3	D
1	4	E
	5	F
	6	G
	7	H
2	8	I
	9	J
	10	K
	11	L

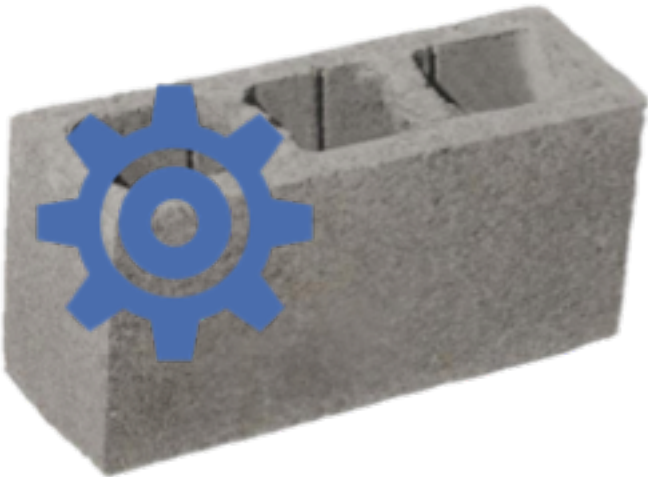
Segment	Limit Register	Base Register
0	4	16
1	4	8
2	4	20



0	e f g h
4	a b c d
8	E F G H
12	m n o p
16	A B C D
20	I J K L
24	
28	j k l

# How does the OS **allocate** free space to processes to be loaded in memory?

“Memory-Management Schemes”



Contagious



Segmentation



Paging



つづく