DCS216 Operating Systems

Lecture 18
Memory (1)

May 6th, 2024

Instructor: Xiaoxi Zhang

Sun Yat-sen University

Content

Content

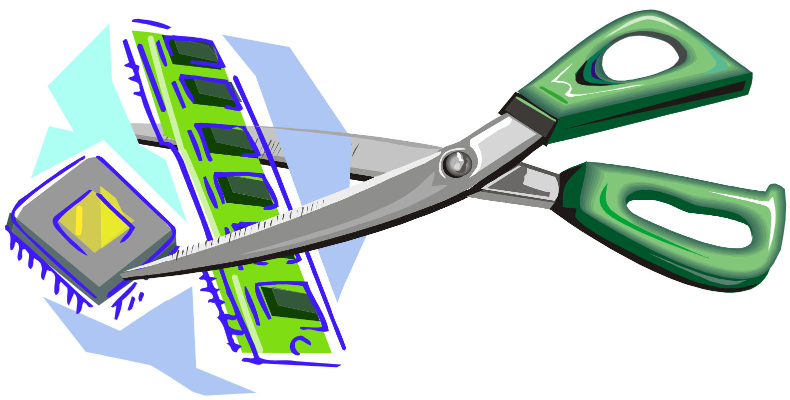
- Basic Concepts
  - Main Memory
  - Hardware Memory Protection
  - Address Binding
  - Logical Address vs. Physical Address
  - Static Linking vs. Dynamic Linking vs. Dynamic Loading
- Swapping
- Memory Partition
  - Fixed Partition $\Rightarrow$ *internal* fragmentation
  - Variable Partition $\Rightarrow$ *external* fragmentation
- Segmentation (分段)

Virtualizing Resources

- Physical Reality:

  Different processes/threads sharing the same hardware.

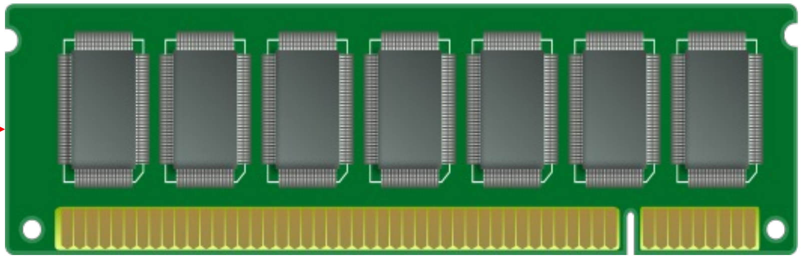
  - Need to multiplex **CPU** (just finished: processes/threads, scheduling)
  - Need to multiplex **Memory** (starting today)
  - Need to multiplex **Disk** and **I/O devices** (later...)



Background

- The main purpose of a computer system is to **execute programs**.

- During execution, these programs must be brought (from disk) into main memory (at least *partially*) in order to run.



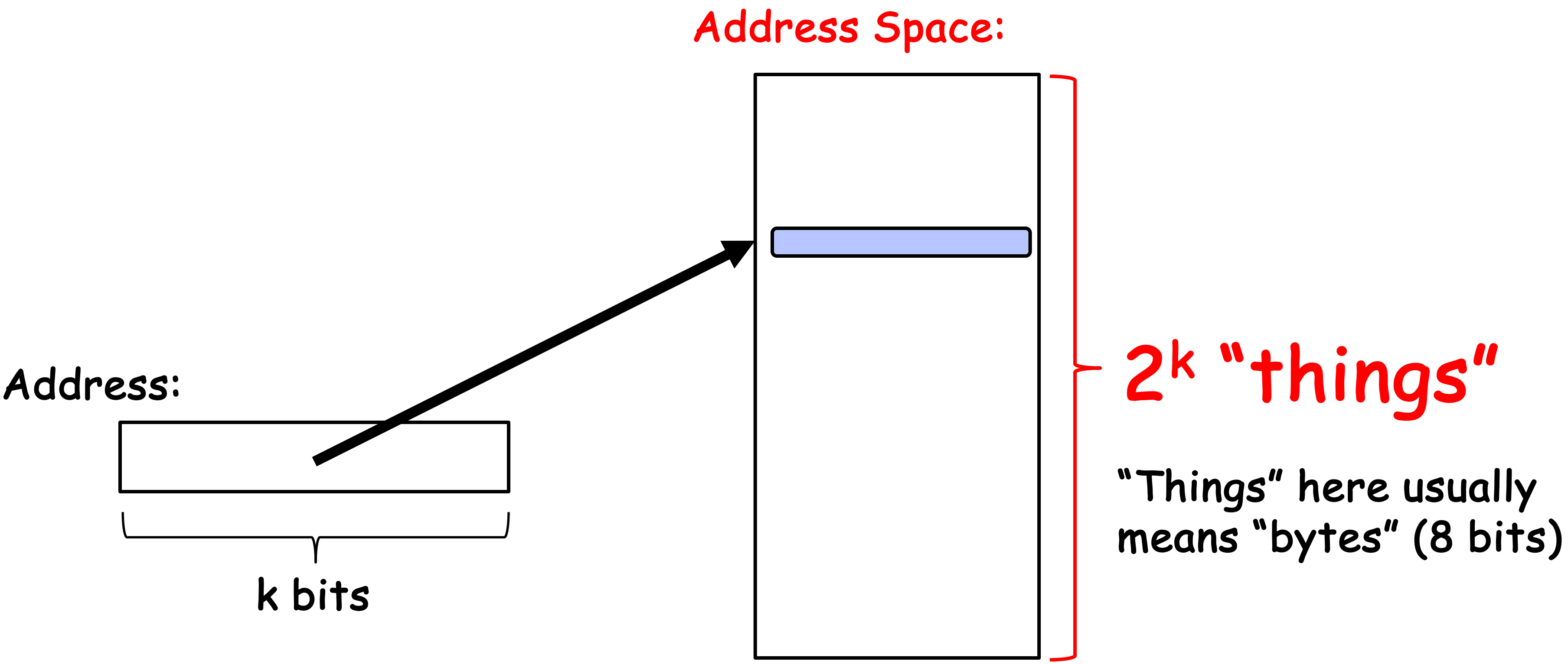
Main memory, or RAM module

- The CPU can directly access **only**
  - Registers
  - Main Memory

Background

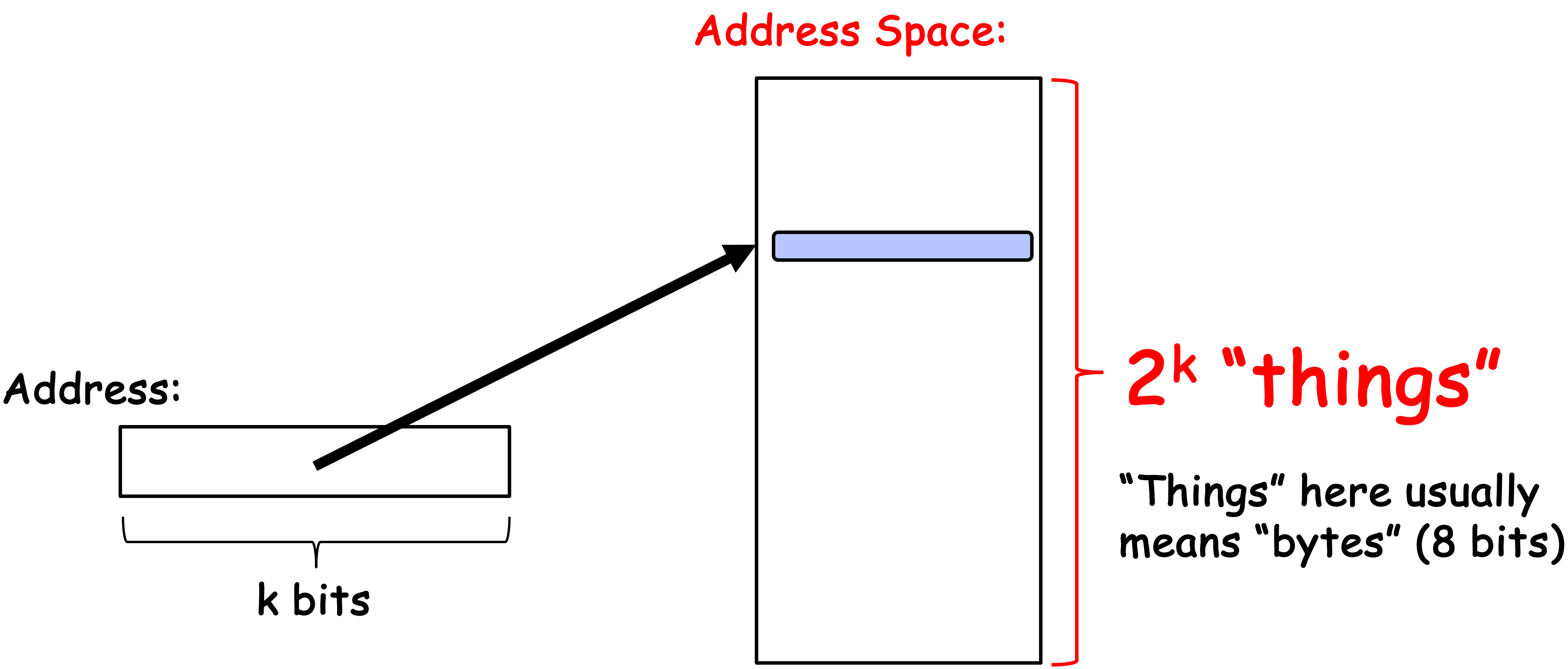
- The main purpose of a computer system is to **execute programs**.
- During execution, these programs must be brought (from disk) into main memory (at least *partially*) in order to run.
- Memory consists of a large array of words or bytes, each with its own **address**
- The CPU fetches instructions from memory according to the value of the **PC** (program counter, or instruction pointer)
- These instructions may cause additional **loading from** and **storing to** specific memory addresses.

- **Address vs. Address Space**



Address vs. Address Space

- For example, in x86, each (virtual) address has 32 bits, thus the address space is $2^{32} \approx 4\ billion\ bytes$



Basic Hardware

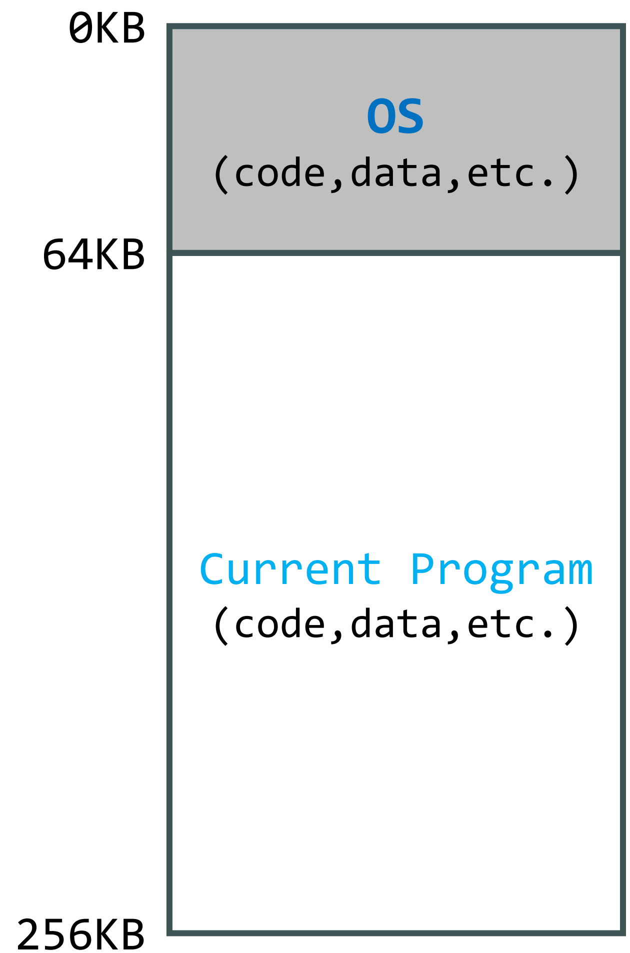
- The CPU can directly access **only**
  - Registers
  - Main Memory
- There are machine instructions that take **memory addresses** as arguments, but none that take **disk addresses**.
- So, any instructions in execution, and any data being used by the instructions, must be in one of these direct-access storage devices.
- If the data are not in memory, they must be moved there before the CPU can operate on them.

CPU Cycle Time for accessing memories

- Registers: Accessible within **one** cycle of the CPU clock
  - Most CPUs can decode instructions and perform simple operations on register contents at the rate of one or more operations per clock tick.
- Main Memory: Access may take **many** cycles of the CPU clock
  - In this case, the processor normally needs to stall, since it does not have the data required to complete the instruction that it is executing.

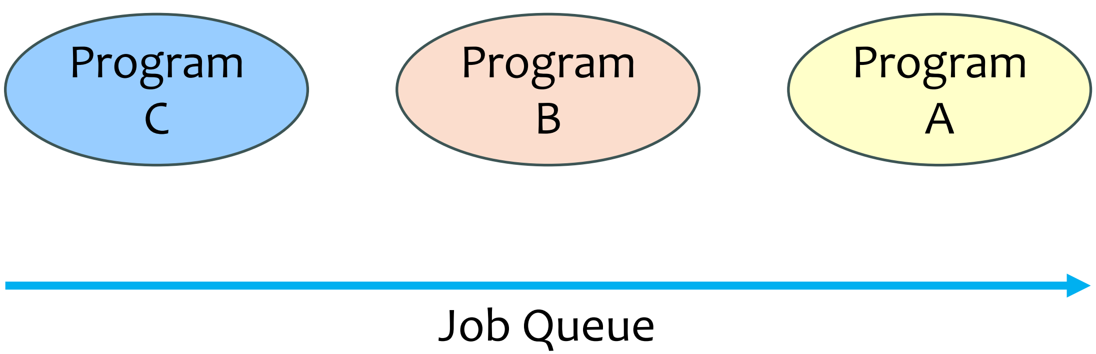
L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	25 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	3,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from disk	20,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

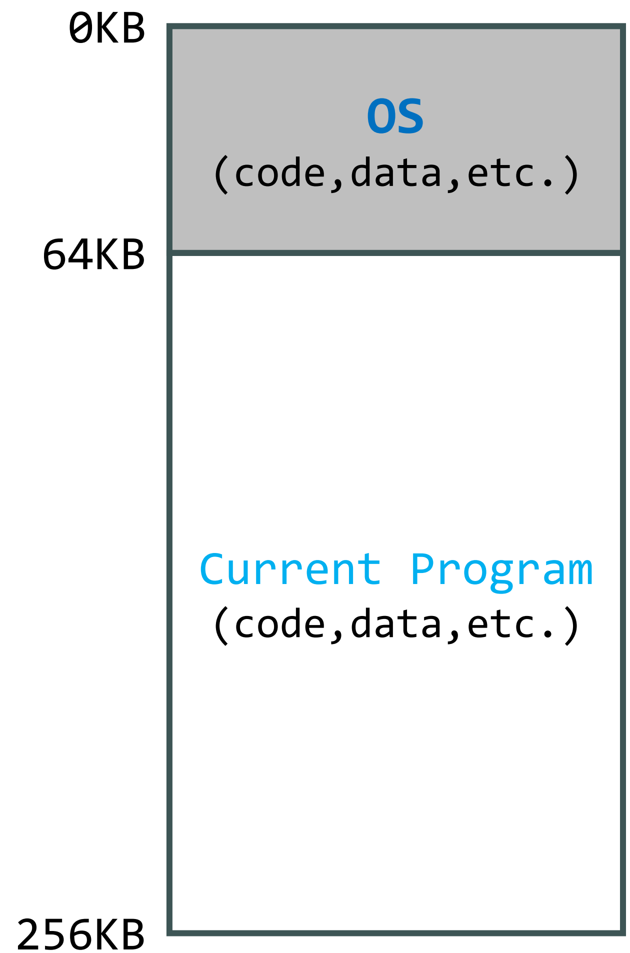
- **Early Systems (Uniprogramming)**
  - E.g., Simple Batch Processing System (单道批处理系统)
  - The OS is merely a set of routines (a library)
    - starting at address 0KB, for example
  - There can be only **ONE** running program
    - starting at address 64KB, for example
    - ...and occupying the **rest** of memory



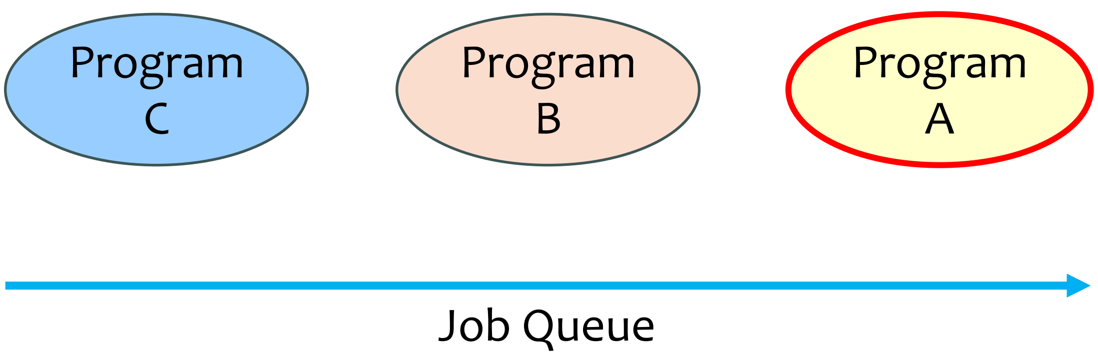
Early Systems (Uniprogramming)

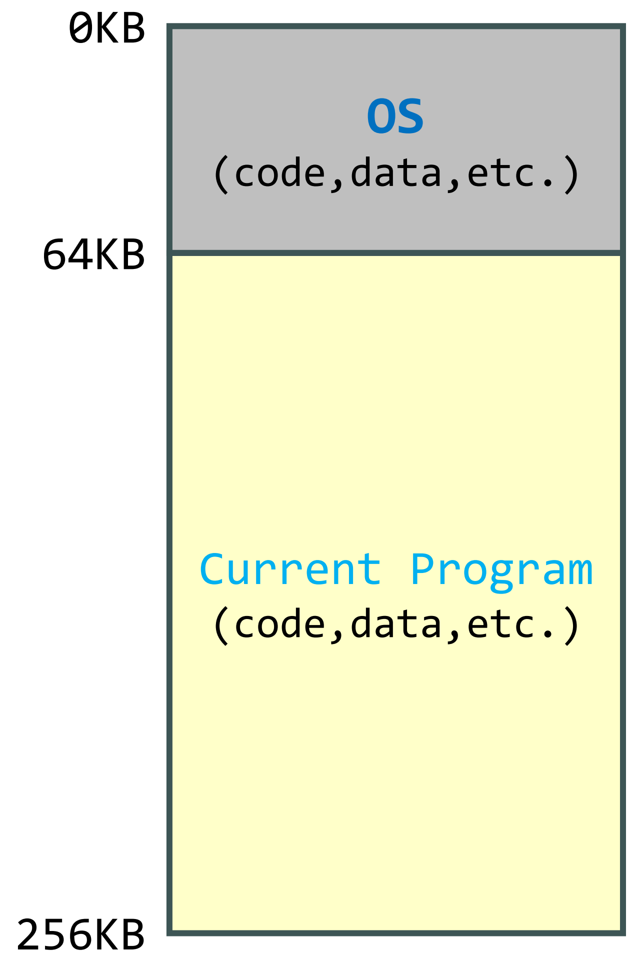
- E.g., Simple Batch Processing System (单道批处理系统)
- The OS is merely a set of routines (a library)
  - starting at address 0KB, for example
- There can be only **ONE** running program
  - starting at address 64KB, for example
  - ...and occupying the **rest** of memory
- Programs executed sequentially
  - one after another





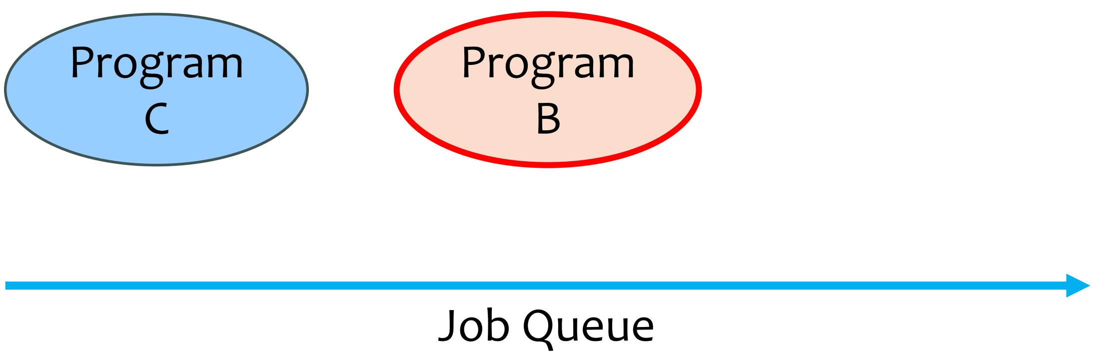
- **Early Systems (Uniprogramming)**
  - E.g., Simple Batch Processing System (单道批处理系统)
  - The OS is merely a set of routines (a library)
    - starting at address 0KB, for example
  - There can be only **ONE** running program
    - starting at address 64KB, for example
    - ...and occupying the **rest** of memory
  - Programs executed sequentially
    - one after another

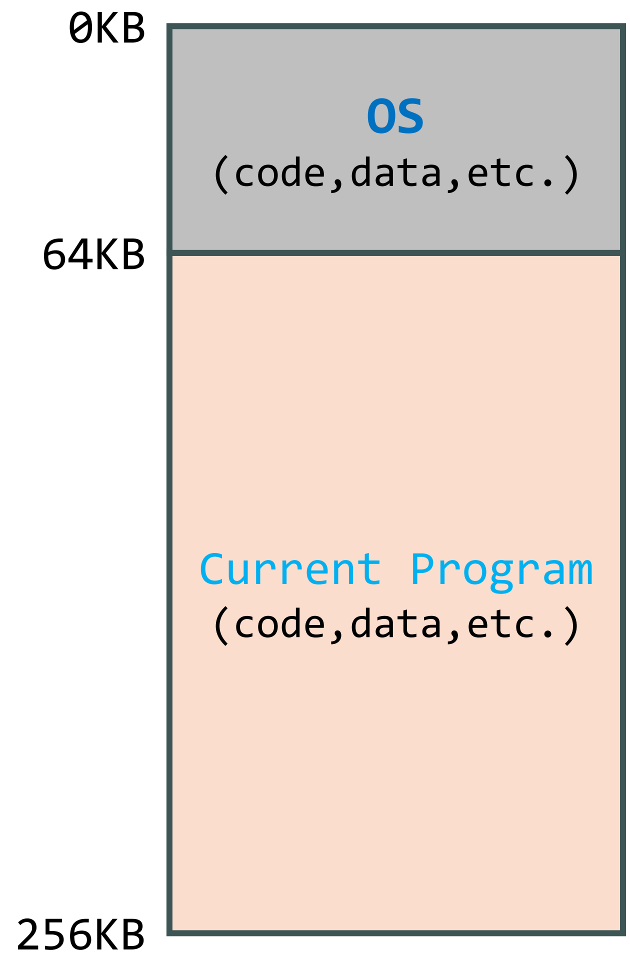




Basic Concepts

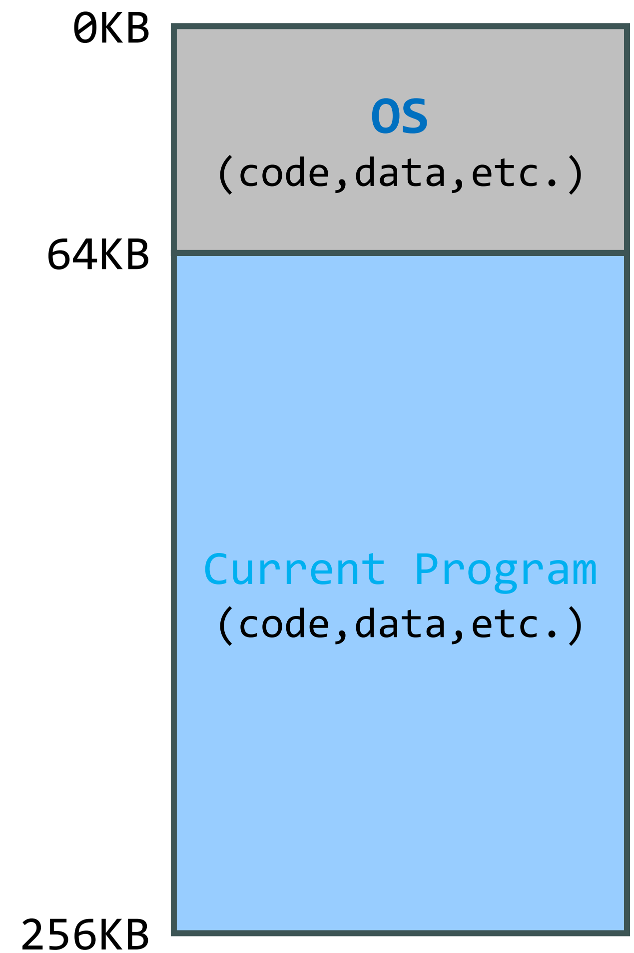
- **Early Systems (Uniprogramming)**
  - E.g., Simple Batch Processing System (单道批处理系统)
  - The OS is merely a set of routines (a library)
    - starting at address 0KB, for example
  - There can be only **ONE** running program
    - starting at address 64KB, for example
    - ...and occupying the **rest** of memory
  - Programs executed sequentially
    - one after another

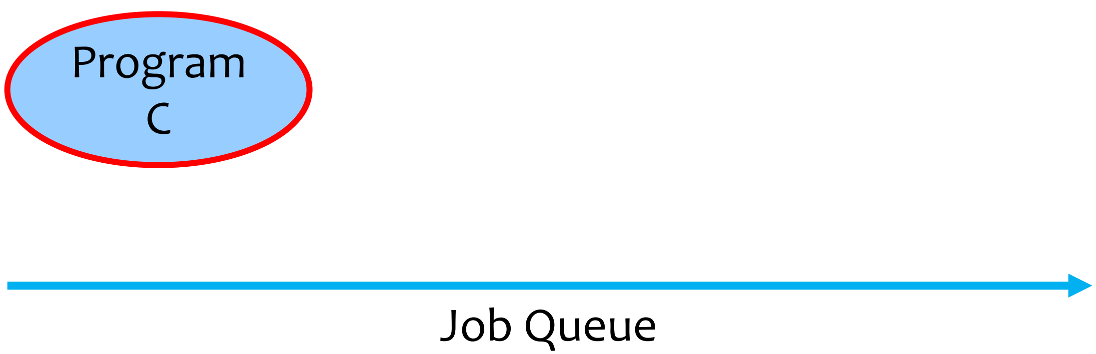




Early Systems (Uniprogramming)

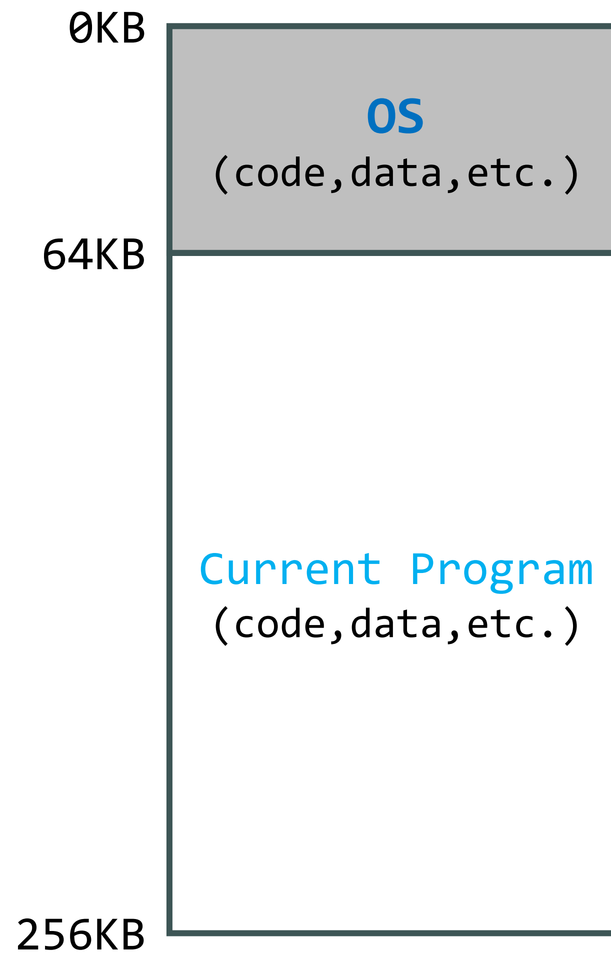
- E.g., Simple Batch Processing System (单道批处理系统)
- The OS is merely a set of routines (a library)
  - starting at address 0KB, for example
- There can be only **ONE** running program
  - starting at address 64KB, for example
  - ...and occupying the **rest** of memory
- Programs executed sequentially
  - one after another

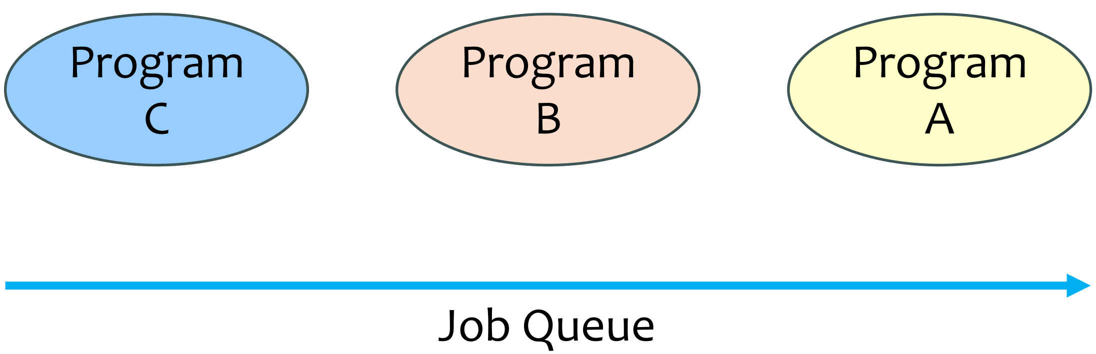




Early Systems (Uniprogramming)

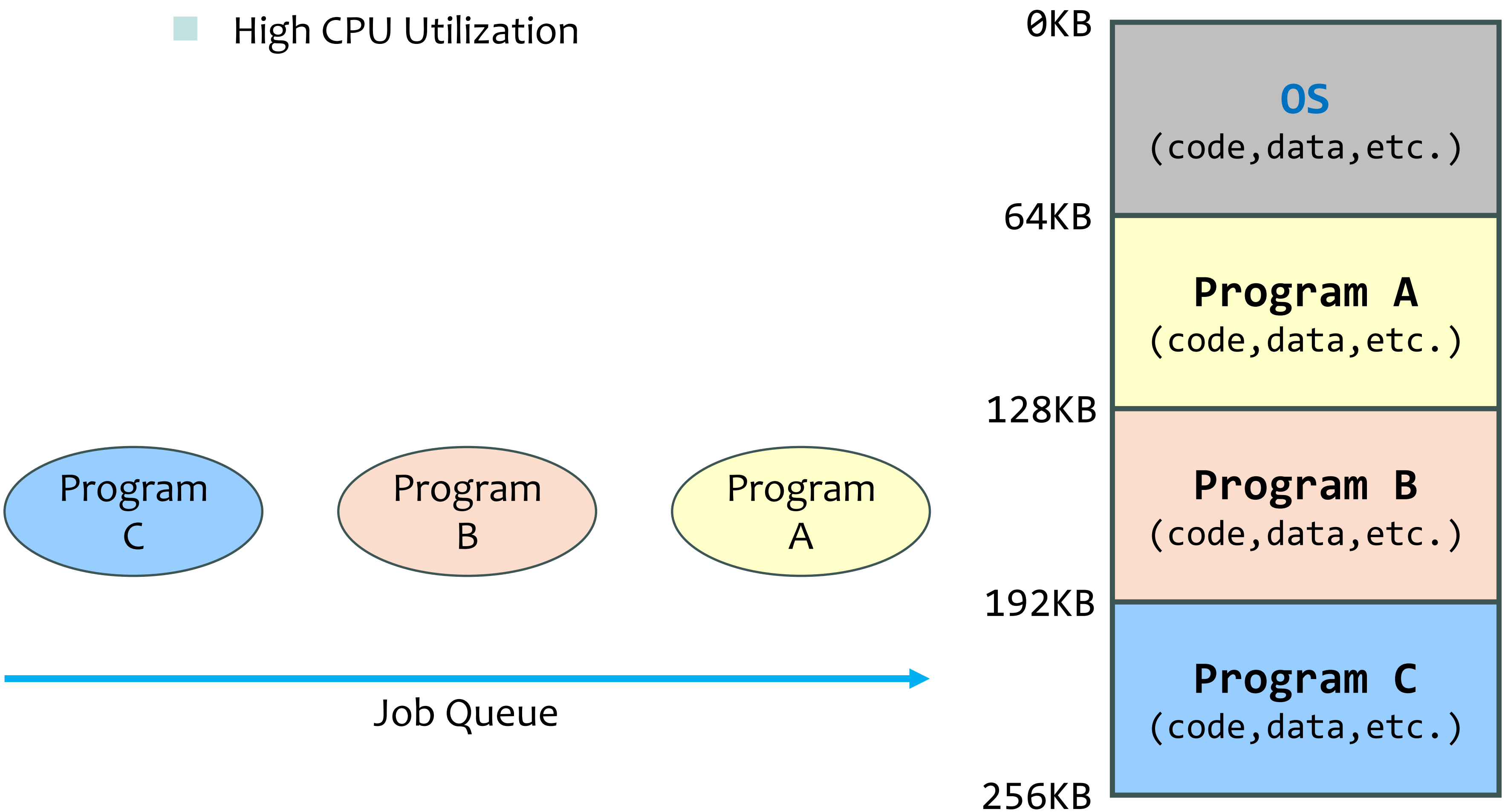
- E.g., Simple Batch Processing System (单道批处理系统)
- Limitations:
  - Low CPU Utilization
  - For example, if A performs I/O
    - the CPU is idle.
    - B and C cannot execute
    - because A is not finished.





Multiprogramming

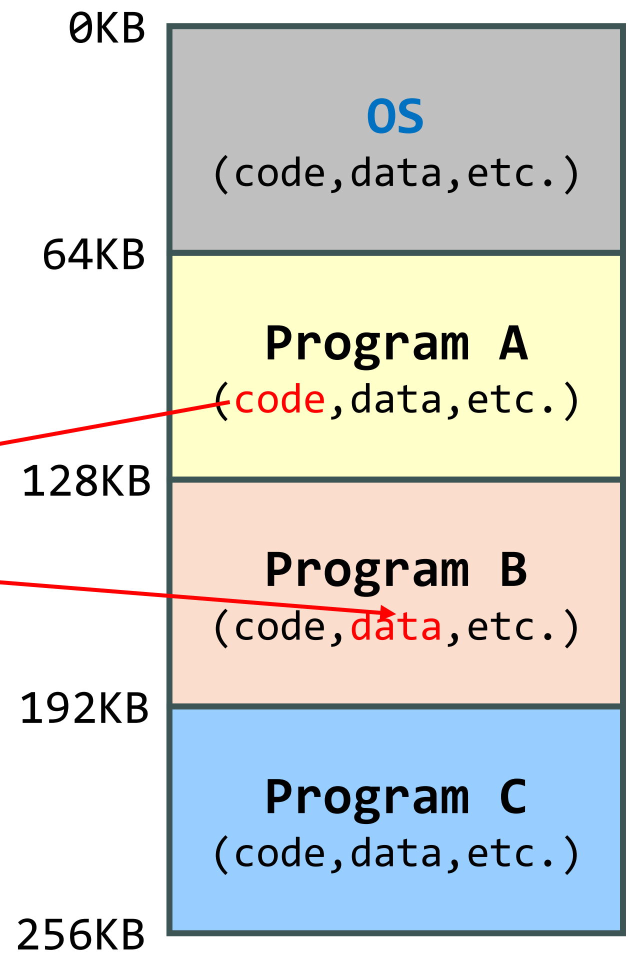
- Multiple programs loaded in memory (assume enough space)
  - The OS would switch between them.
  - High CPU Utilization



Multiprogramming

- Multiple programs loaded in memory (assume enough space)
  - The OS would switch between them.
  - High CPU Utilization
- Introduces another problem:
  - No protection!
  - E.g., **A** can modify the data in **B**'s address space

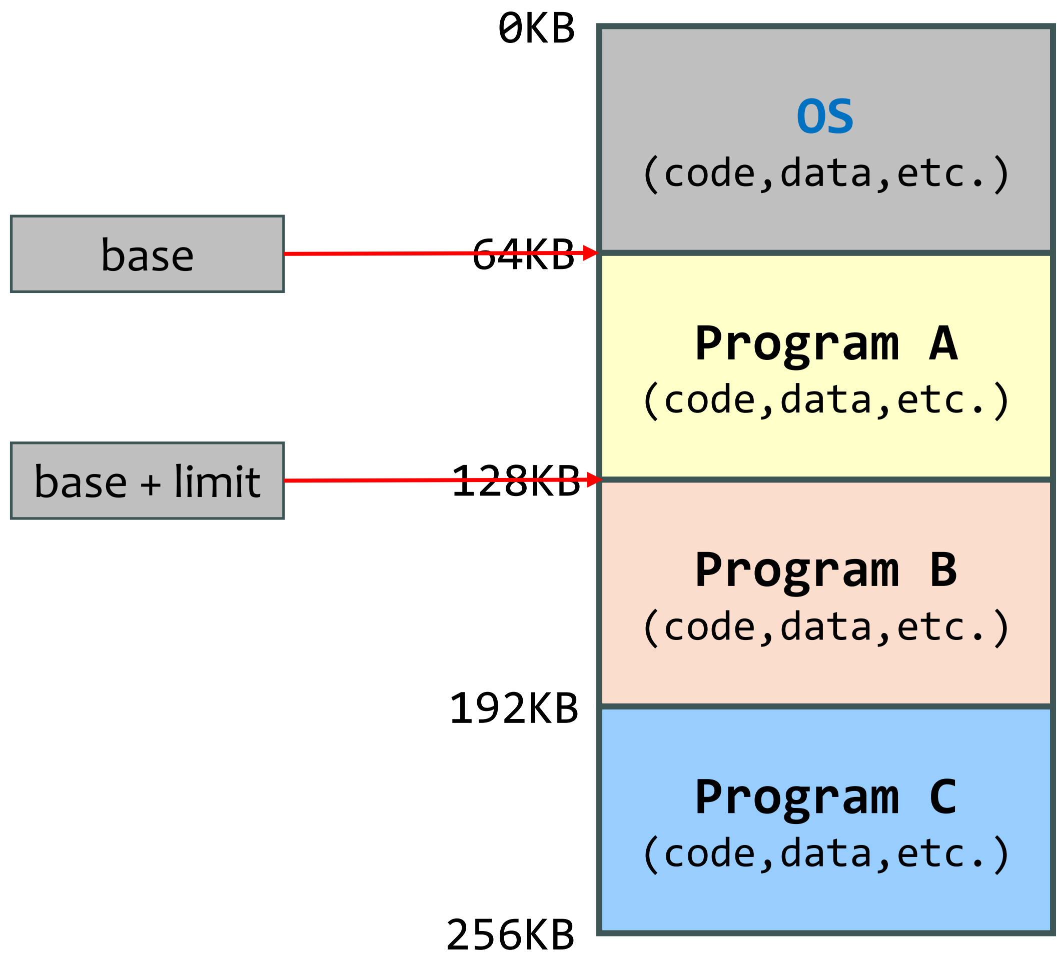
```
0x10200: movl 0x21000, %eax
0x10201: addl $0x1, %eax
0x10202: movl %eax, 0x21000
```



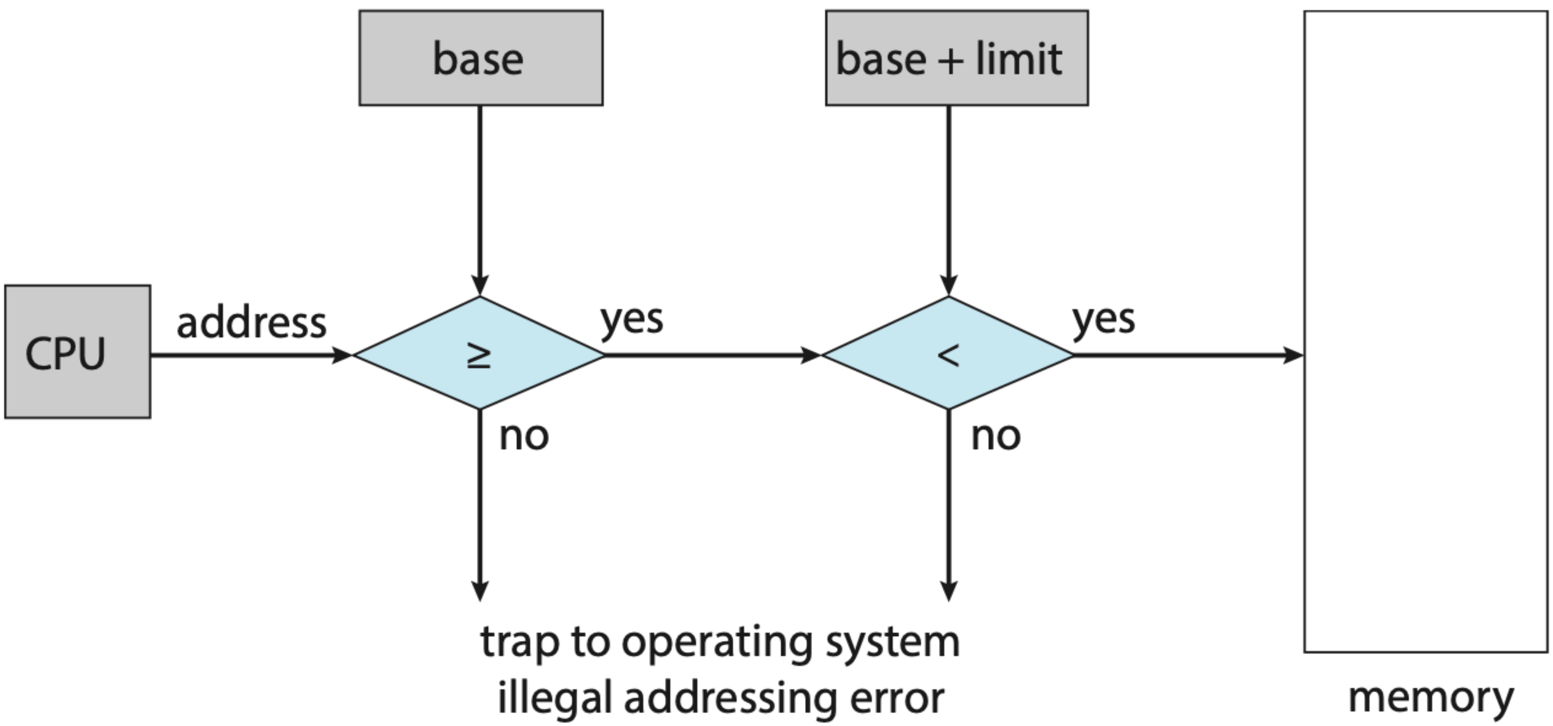
- **Hardware Address Protection**
  - The OS has to be protected from user processes
  - In addition, user processes must be protected from one another
  - This protection must be provided by the **hardware.**
    - Efficiency and speed
    - Security
    - Reliability
    - Simplicity and Transparency

Hardware Address Protection (Base & Bounds)

- A pair of **base** register and **limit** register (also called **bounds**) define the logical address space for a process

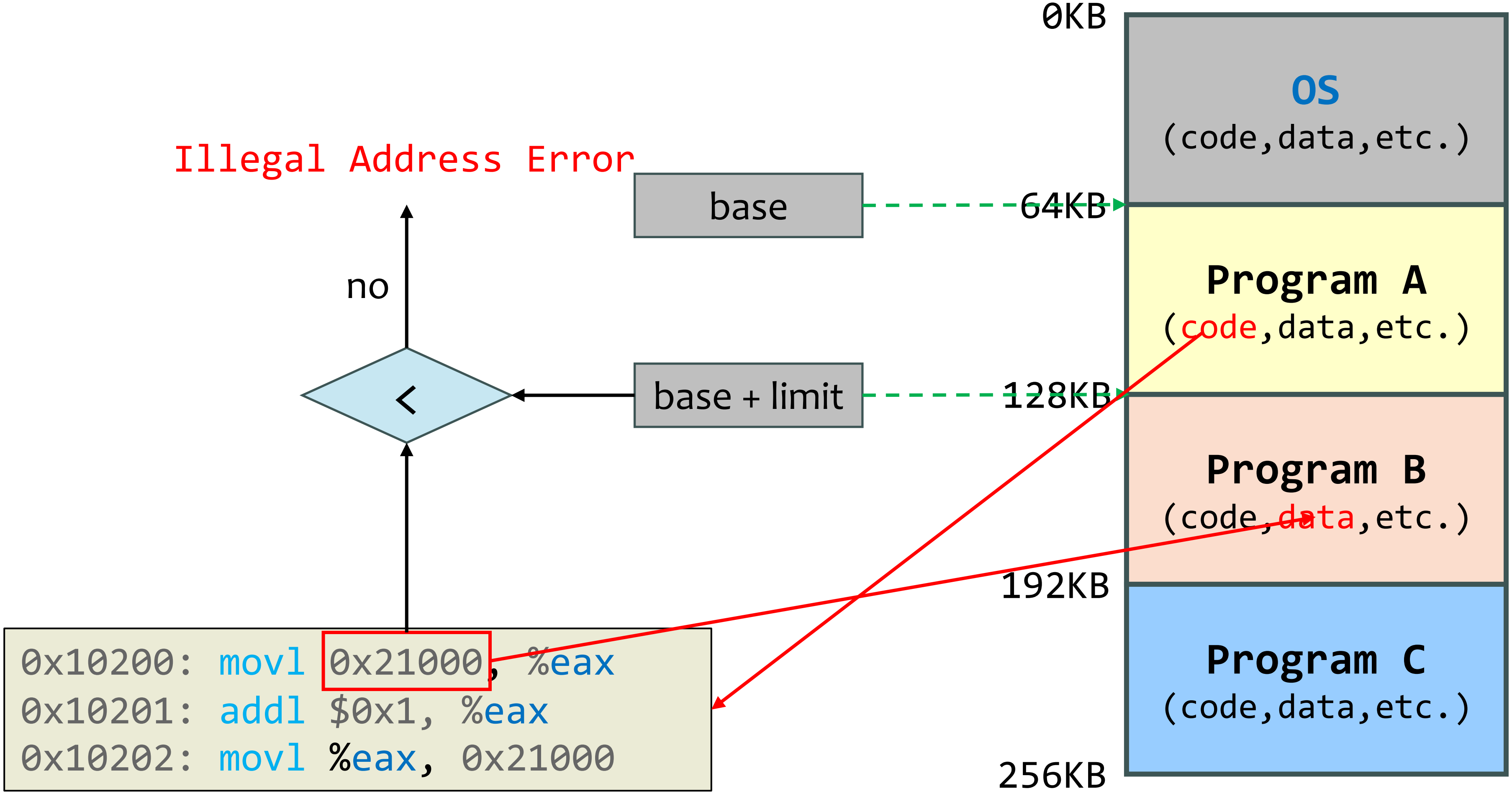


- **Hardware Address Protection (Base & Bounds)**
  - CPU must check **every** memory access generated in user mode to make sure it is between **base** and **limit** for that process



Multiprogramming (with Base & Bounds)

- CPU must check **every** memory access generated in user mode to make sure it is between **base** and **limit** for that process



Address Binding

- Usually, a program resides on a disk as a binary executable file.
- To be executed, the program must be brought into memory and placed within a process.
- Addresses represented in different ways at different stages of a program's life:
  - Source code addresses usually **symbolic**.
    - `int add(int a, int b)`
  - **Compiler** binds **symbolic** addresses to **relocatable** addresses
    - `0x000000`
  - **Linker** or **Loader** binds **relocatable** addresses to **absolute** addresses
    - `0x401792`
  - Each binding maps from one address space to another.

Address Binding

```c
/* main.c */
#include <stdio.h>
#include "utils.h"

int main() {
    int res = add(3, 4);
    printf("add(3, 4): %d\n", res);
    printf("&add(): %p\n", &add);
    return 0;
}
```

```c
/* utils.c */
#include "utils.h"

int add(int a, int b) {
    return a + b;
}
```

```c
/* utils.h */
#ifndef UTILS_H
#define UTILS_H

int add(int a, int b);

#endif
```

```
$ gcc -c -o main.o main.c
$ gcc -c -o utils.o utils.c
$ gcc -o main -static main.o utils.o
```

Address Binding

```c
/* main.c */
#include <stdio.h>
#include "utils.h"

int main() {
    int res = add(3, 4);
    printf("add(3, 4): %d\n", res);
    printf("&add(): %p\n", &add);
    return 0;
}
```

```c
/* utils.c */
#include "utils.h"

int add(int a, int b) {
    return a + b;
}
```

```c
/* utils.h */
#ifndef UTILS_H
#define UTILS_H

int add(int a, int b);

#endif
```

```
$ gcc -c -o main.o main.c
$ gcc -c -o utils.o utils.c
$ gcc -o main -static main.o utils.o
$ objdump -d --disassemble=add utils.o

utils.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <add>:
   0:   f3 0f 1e fa             endbr64
   4:   55                      push   %rbp
   5:   48 89 e5                mov    %rsp,%rbp
   8:   89 7d fc                mov    %edi,-0x4(%rbp)
   b:   89 75 f8                mov    %esi,-0x8(%rbp)
   e:   8b 55 fc                mov    -0x4(%rbp),%edx
  11:   8b 45 f8                mov    -0x8(%rbp),%eax
  14:   01 d0                   add    %edx,%eax
  16:   5d                      pop    %rbp
  17:   c3                      ret
```

object file

Relocatable address, or placeholder address

Address Binding

```c
/* main.c */
#include <stdio.h>
#include "utils.h"

int main() {
    int res = add(3, 4);
    printf("add(3, 4): %d\n", res);
    printf("&add(): %p\n", &add);
    return 0;
}
```

```c
/* utils.c */
#include "utils.h"

int add(int a, int b) {
    return a + b;
}
```

```c
/* utils.h */
#ifndef UTILS_H
#define UTILS_H

int add(int a, int b);

#endif
```

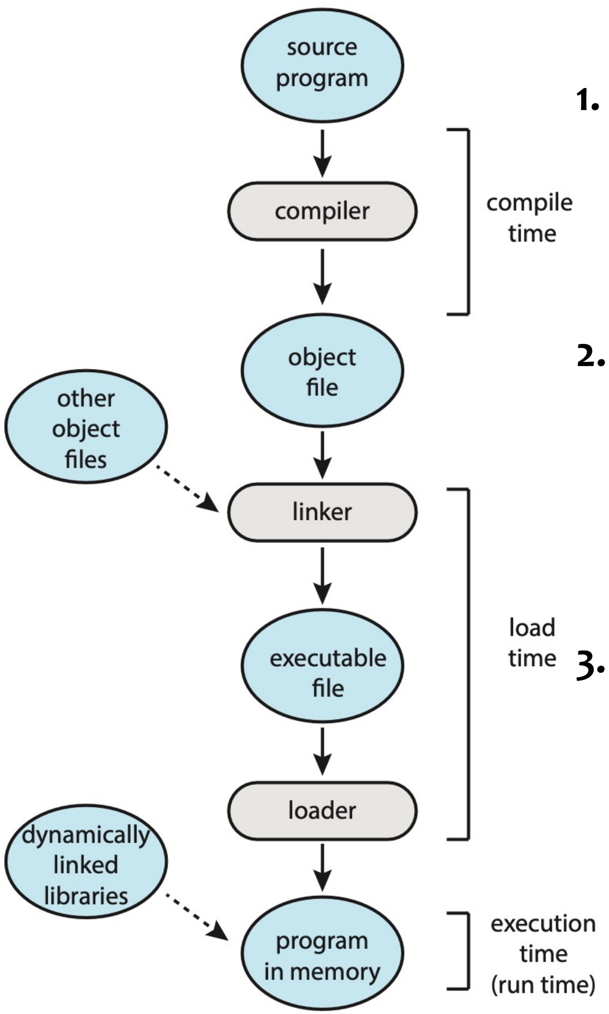
```
$ gcc -c -o main.o main.c
$ gcc -c -o utils.o utils.c
$ gcc -o main -static main.o utils.o
$ objdump -d --disassemble=add main

main:     file format elf64-x86-64


Disassembly of section .text:

0000000000401792 <add>:
  401792:   f3 0f 1e fa     endbr64
  401796:   55              push   %rbp
  401797:   48 89 e5        mov    %rsp,%rbp
  40179a:   89 7d fc        mov    %edi,-0x4(%rbp)
  40179d:   89 75 f8        mov    %esi,-0x8(%rbp)
  4017a0:   8b 55 fc        mov    -0x4(%rbp),%edx
  4017a3:   8b 45 f8        mov    -0x8(%rbp),%eax
  4017a6:   01 d0           add    %edx,%eax
  4017a8:   5d              pop    %rbp
  4017a9:   c3              ret
```

executable file

Relocated address.

Address Binding

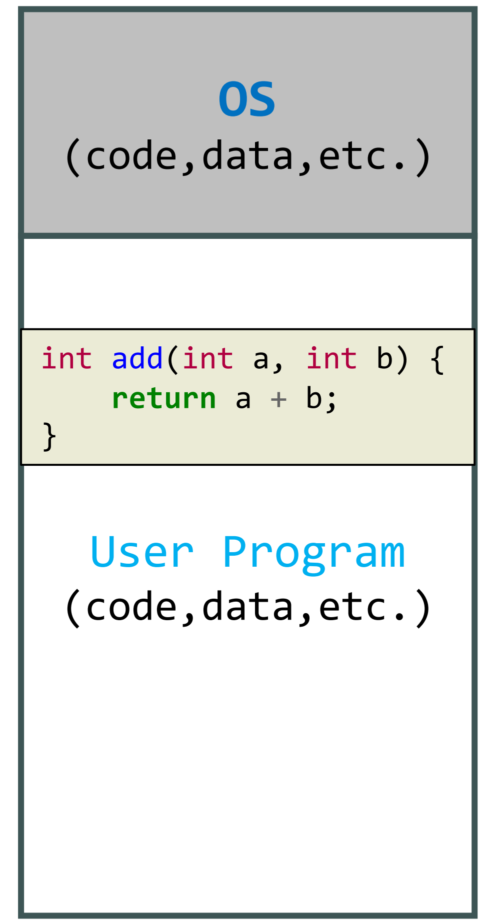
- Address binding of **instructions and data** to (**physical**) **memory addresses** can happen at **three** different stages:



1. **Compile Time**: If we know at compile time where the process will reside in memory, then absolute addresses can be generated.

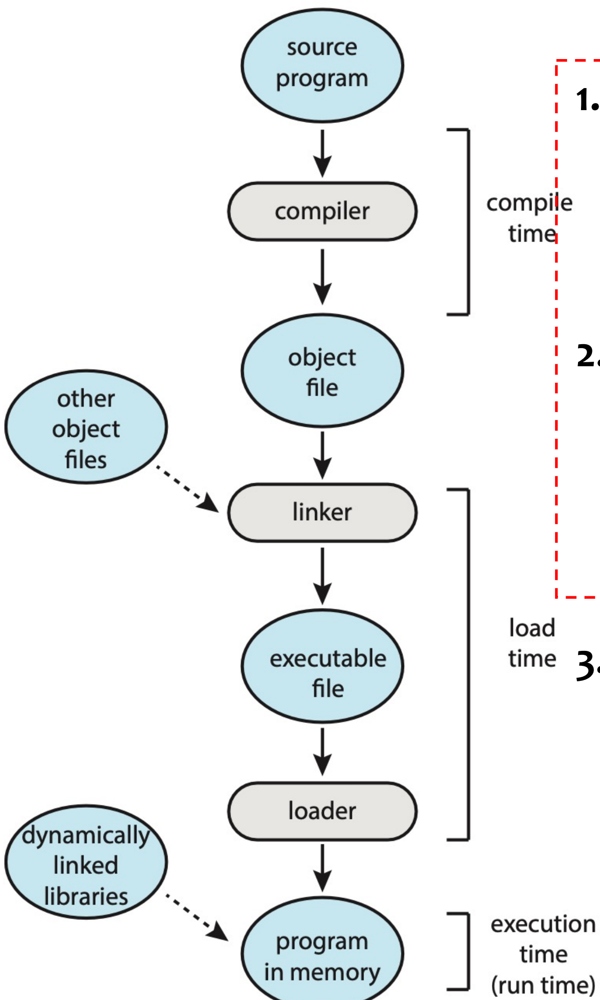
2. **Load Time**: If it is not known at compile time, then the compiler generates relocatable code. The **Loader rewrites** (updates) the addresses. This technique is also called **static relocation**.

3. **Execution Time**: If the process can be moved during its execution from one memory location to another, then binding must be delayed until run time. Also called **dynamic relocation**. Most general-purpose OSes use this method (via **MMU**).



Address Binding

- Address binding of **instructions and data** to (**physical**) **memory addresses** can happen at **three** different stages:



Logical Addr == Physical Addr

1. **Compile Time:** If we know at compile time where the process will reside in memory, then absolute addresses can be generated. ⇒ Uniprogramming

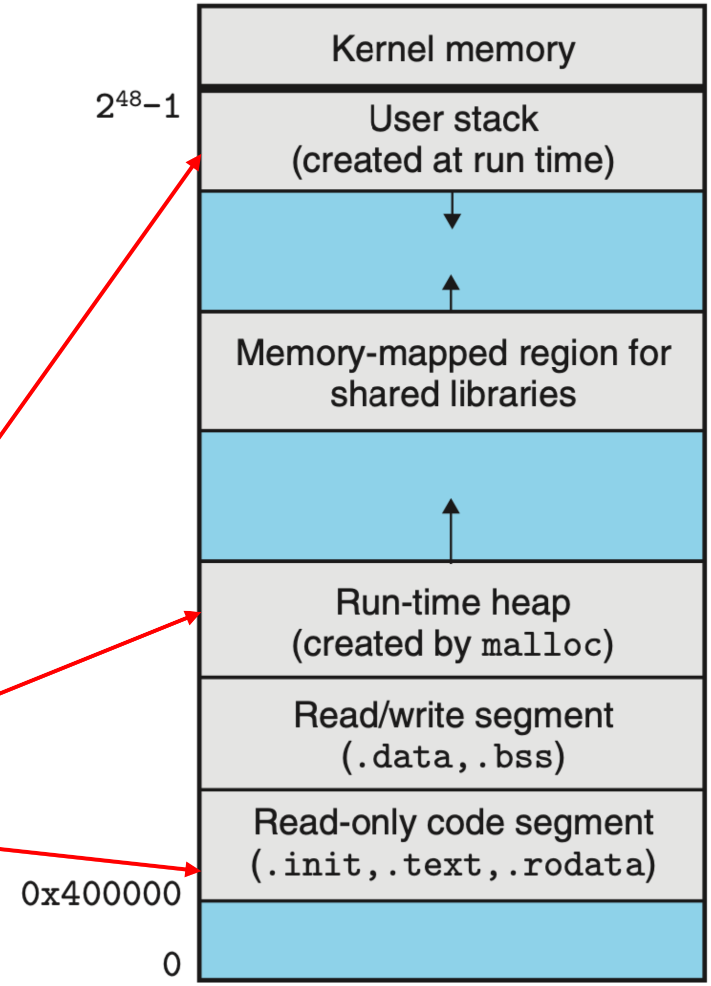
2. **Load Time:** If it is not known at compile time, then the compiler generates relocatable code. The **Loader rewrites** (updates) the addresses. This technique is also called **static relocation.** ⇒ Multiprogramming **without** translation

3. **Execution Time:** If the process can be moved during its execution from one memory location to another, then binding must be delayed until run time. Also called **dynamic relocation.** Most general-purpose OSes use this method (via **MMU**). ⇒ Multiprogramming **with** translation

"Every Address You See is Virtual"

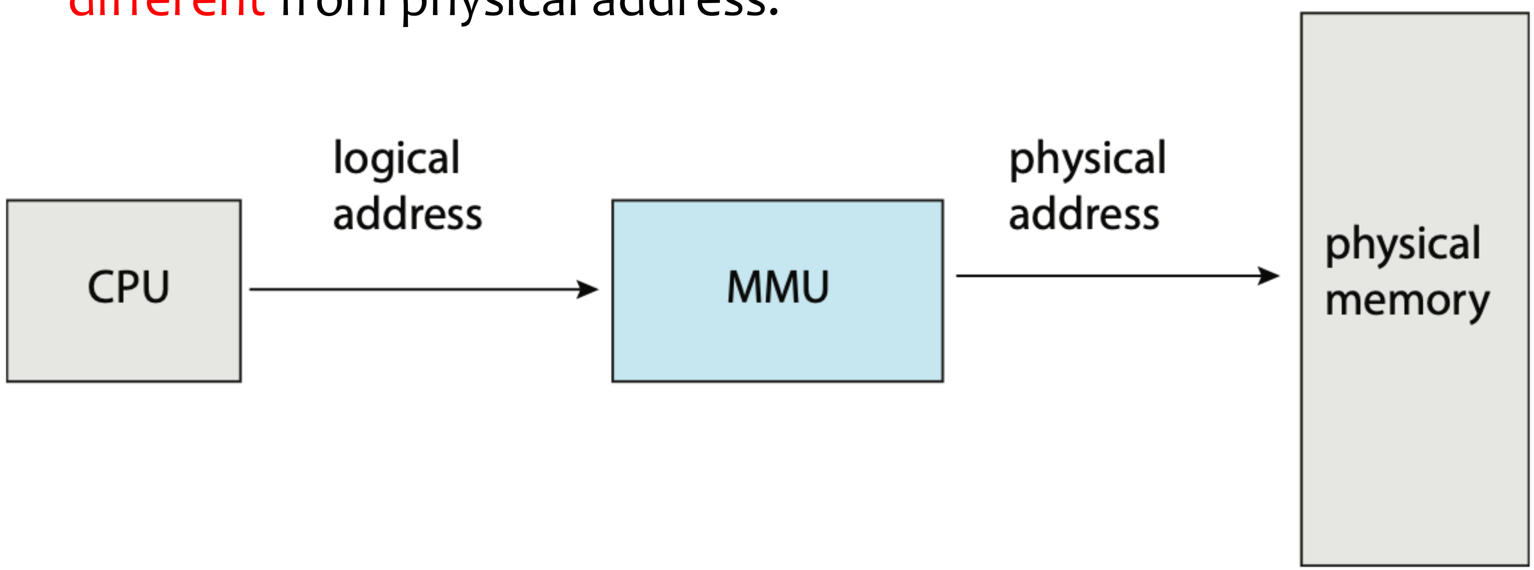
```c
/* va.c */
#include <stdio.h>
#include <stdlib.h>

int main() {
    printf("Address of CODE: %p\n", main);
    printf("Address of HEAP: %p\n",
           malloc(10e6));
    int x = 42;
    printf("Address of STACK: %p\n", &x);
    return 0;
}
```

```
$ gcc -static -o va va.c
$ ./va
Address of CODE: 0x401745
Address of HEAP: 0x781c4932b010
Address of STACK: 0x7ffdec62d214
```



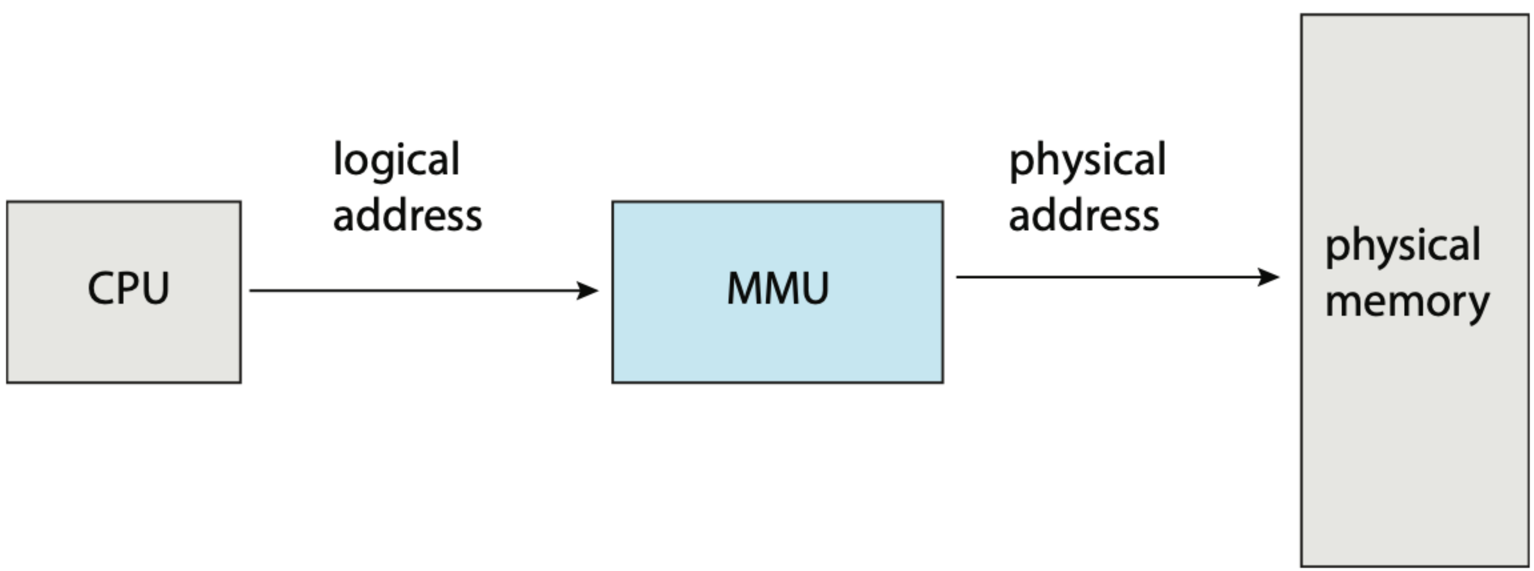
Logical Address Space vs. Physical Address Space

- **Logical Address** ⇒ An address generated by the CPU
  - also referred to as **virtual address**.
- **Physical Address** ⇒ An address seen by the memory-unit, that is, the one loaded into the memory-address register of the memory
- **Compile-Time** and **Load-Time** address-binding methods generate *identical* **logical** and **physical** addresses.
- In **Execution-Time** address-binding scheme, logical address is different from physical address.

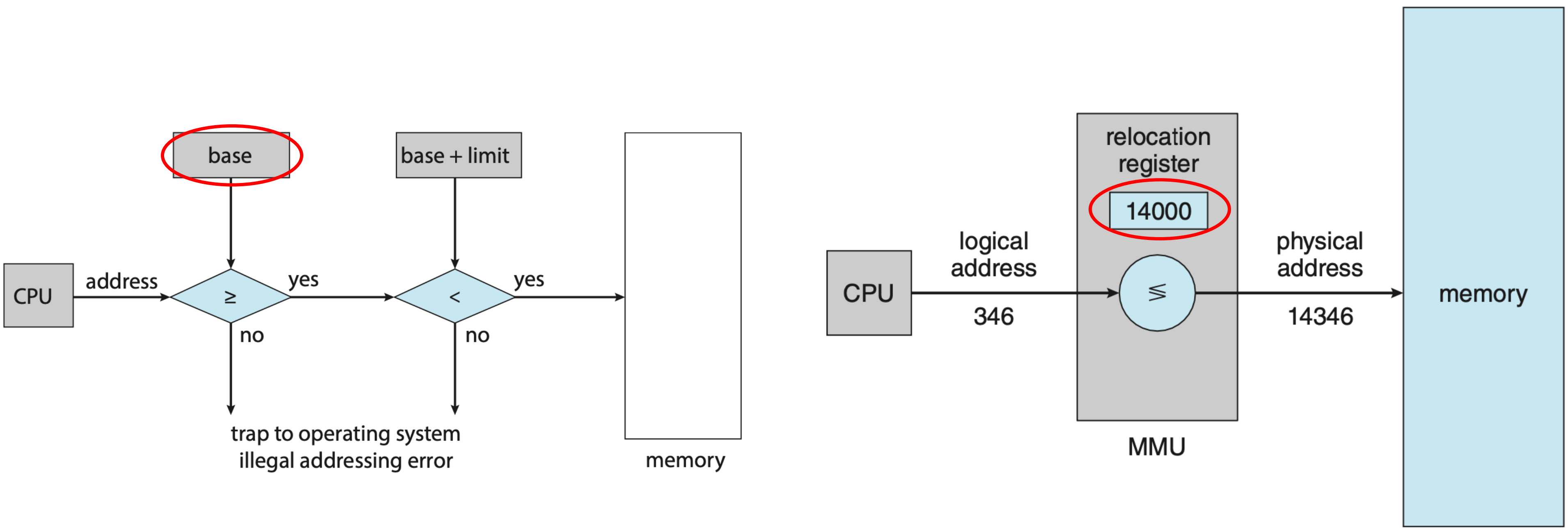


Basic Concepts

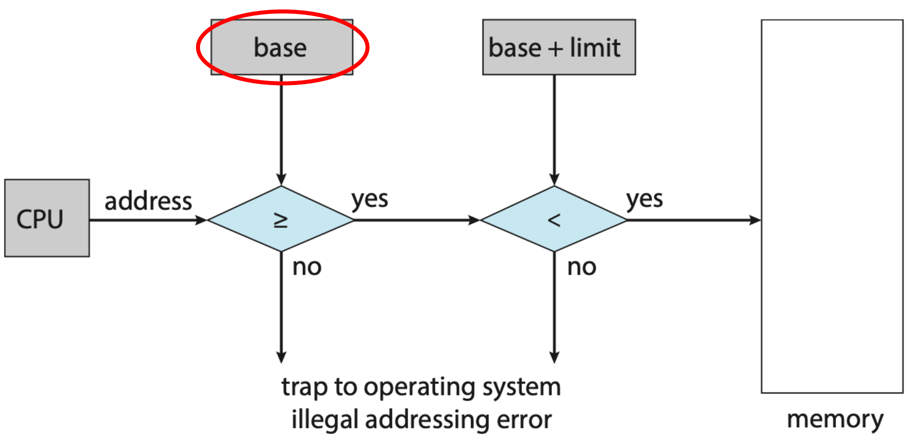
- **Memory-Management Unit (MMU)**
  - **Memory-Management Unit (MMU)** is a hardware device that maps virtual addresses to physical addresses **at run time**.
    - Many different methods are possible to accomplish such mapping
      - Contiguous Memory Allocation
      - Segmentation
      - Paging
      - ...

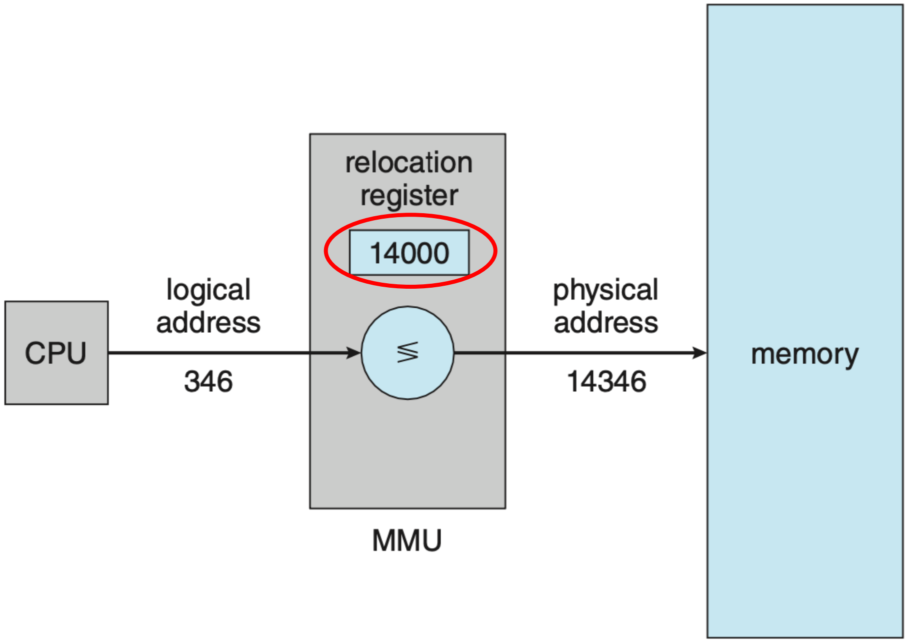


- **Memory-Management Unit (MMU)**
  - Consider a simple scheme, which is a generalization of the **base** and **limit** registers scheme.
  - The **base register** is now called **relocation register**.
  - The value in the relocation register is added to every address generated by a user process at the time it is sent to memory.



- **Memory-Management Unit (MMU)**
  - The user program deals with **logical** (virtual) addresses; it never sees the *real* **physical** addresses.
    - Executime-Time address-binding occurs when reference is made to location in memory.
    - Logical addresses are bound to physical addresses.

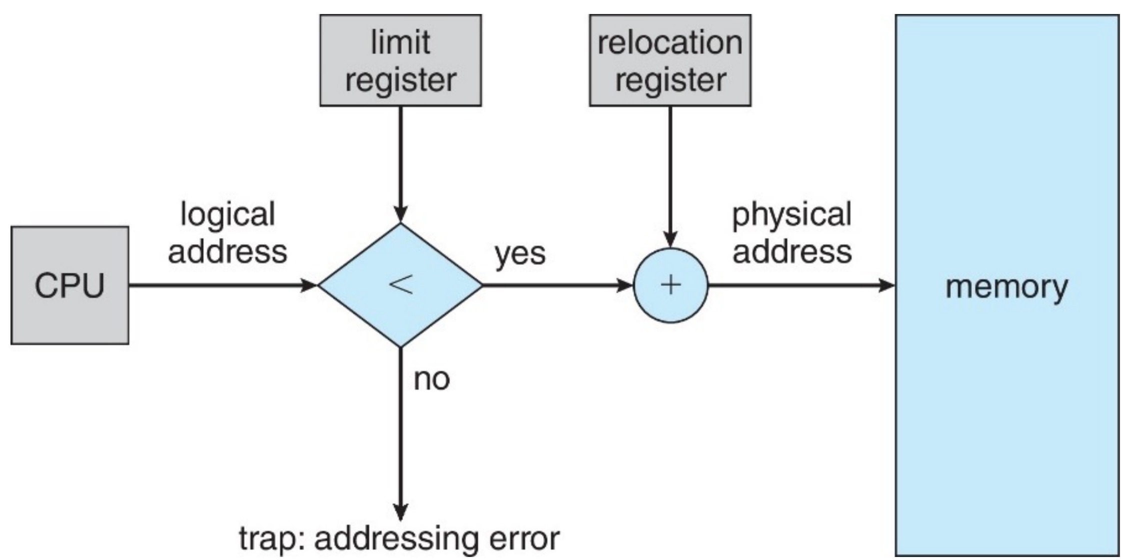




Hardware Support for Dynamic Relocation

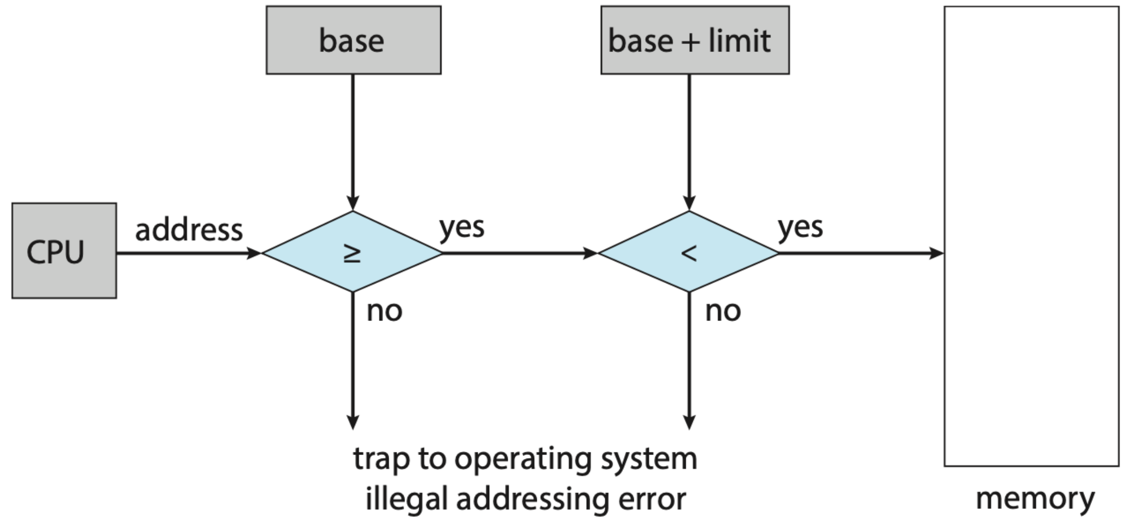
- When a process is assigned to the RUNNING state, a relocation/base register gets loaded with the starting physical address of the process.
- A limit/bounds register gets loaded with the process's ending physical address
- When a logical address is encountered, it is added with the content of the relocation register to obtain the physical address.
- **Protection:** each process can only access memory within its range.

Base&Bound **with** translation

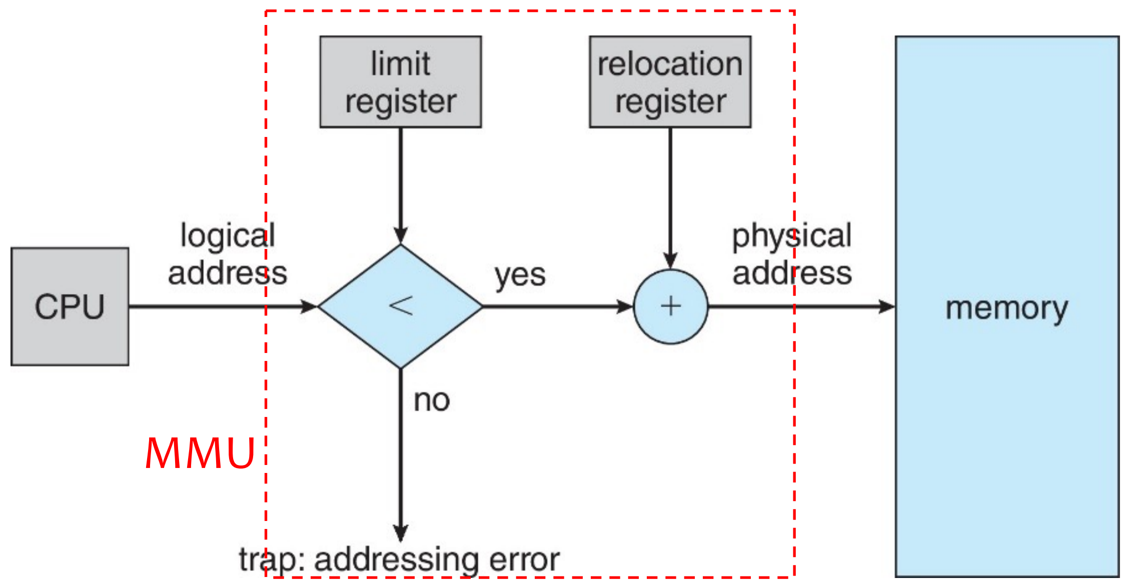


Hardware Support for Dynamic Relocation

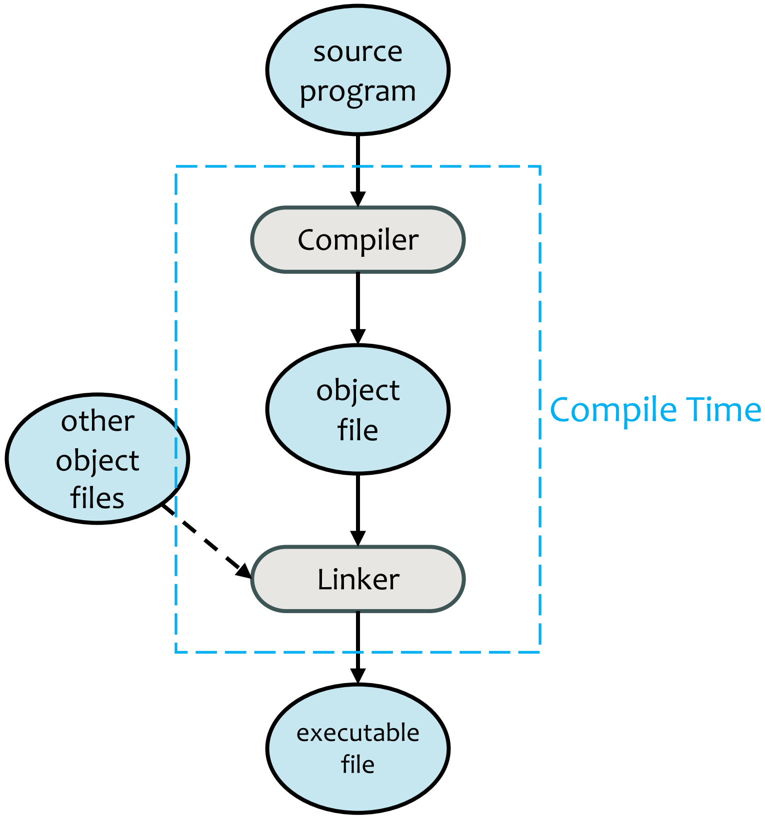
Base&Bound
without translation



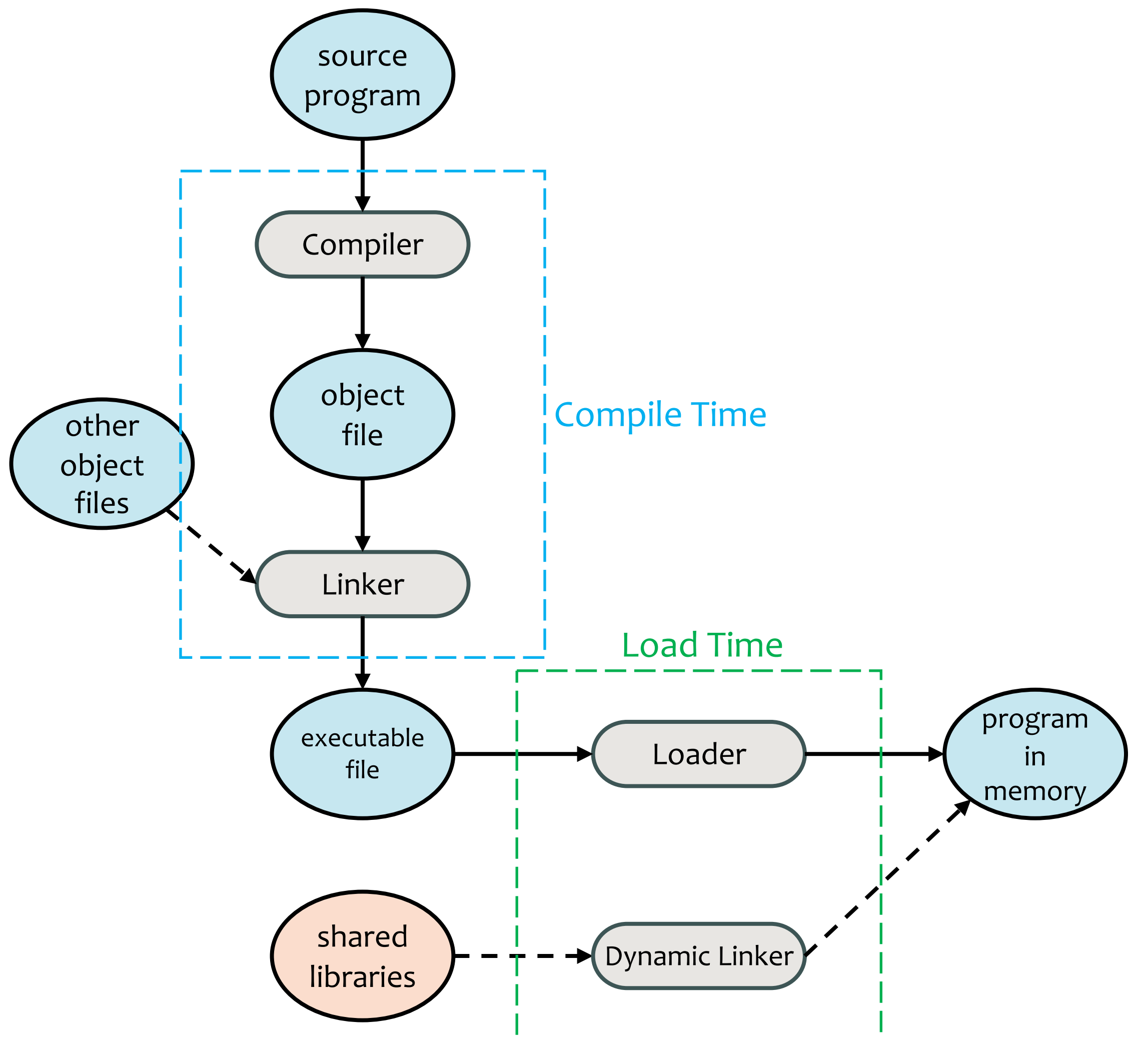
Base&Bound **with** translation



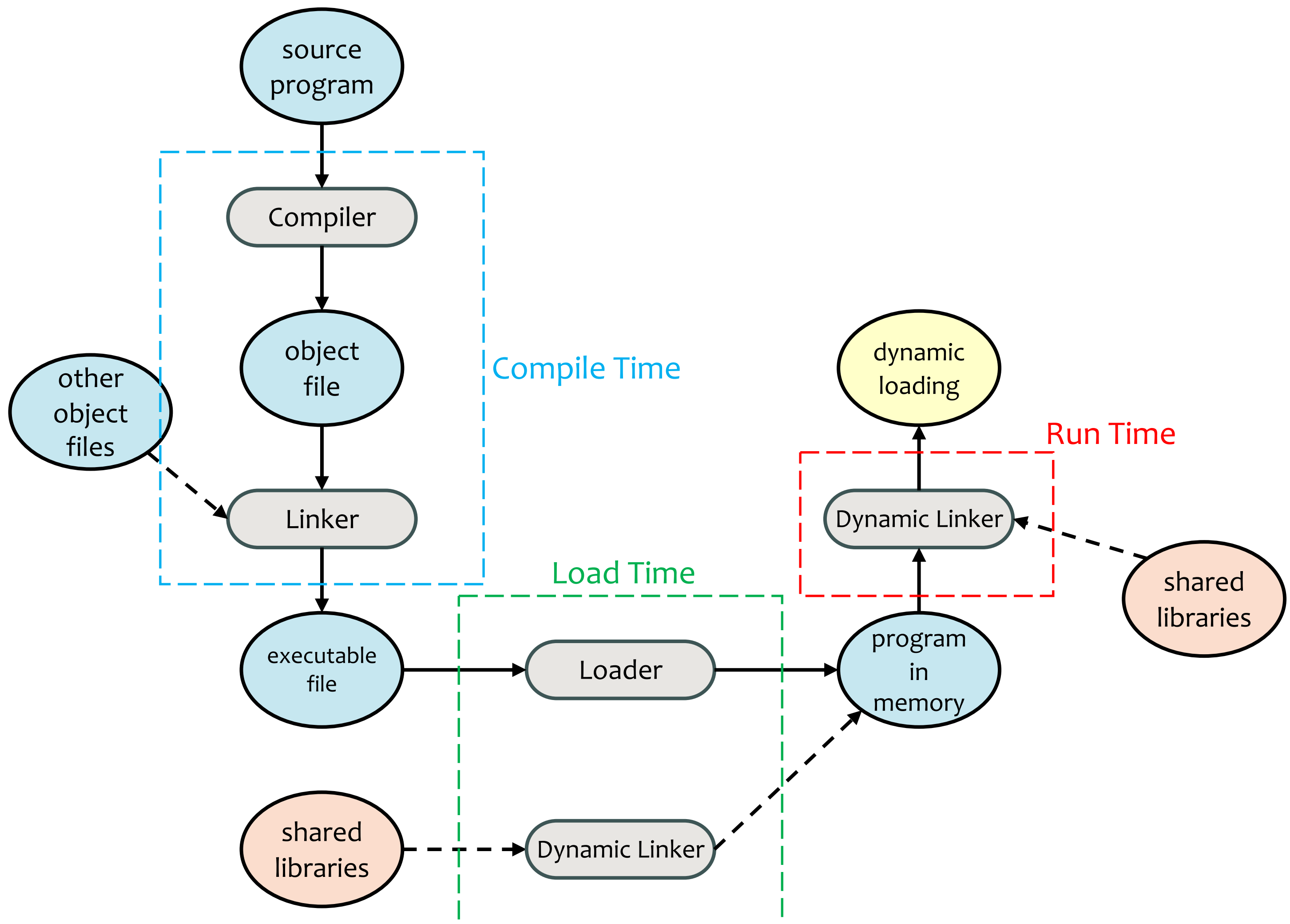
Static Linking



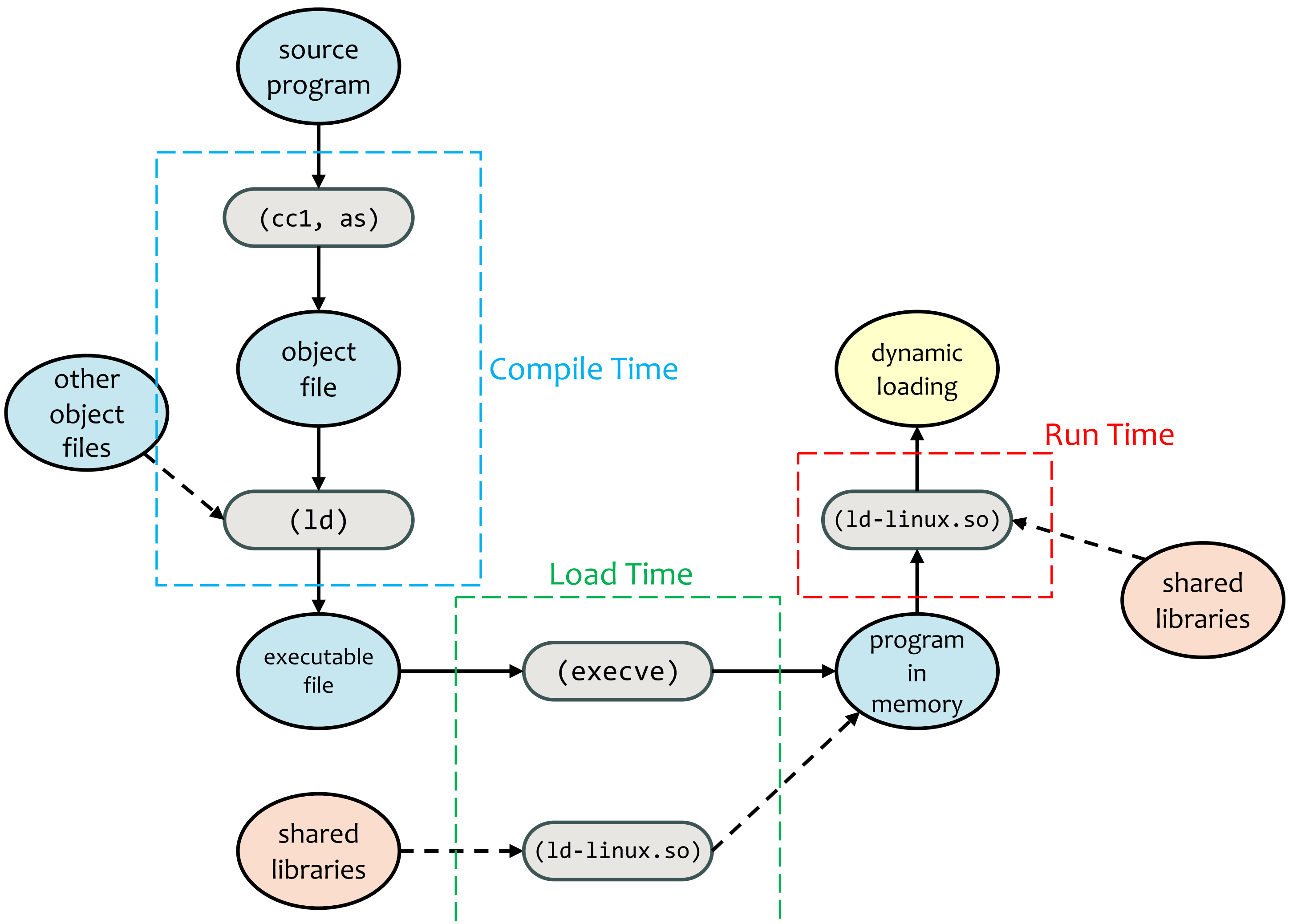
Dynamic Linking



Dynamic Loading



Address Binding



Static Linking

```c
/* main.c */
#include <stdio.h>
#include "utils.h"

int main() {
    int res = add(3, 4);
    printf("add(3, 4): %d\n", res);
    printf("&add(): %p\n", &add);
    return 0;
}
```

```c
/* utils.c */
#include "utils.h"

int add(int a, int b) {
    return a + b;
}
```

```c
/* utils.h */
#ifndef UTILS_H
#define UTILS_H

int add(int a, int b);

#endif
```

```
$ gcc -c -o main.o main.c
$ gcc -c -o utils.o utils.c
$ gcc -o main -static main.o utils.o
$ ./main
add(3, 4): 7
&add(): 0x401792
```

The executable file `main` contains the code for `add()`, which is located in the **code segment**. If we execute multiple copies of `main`:

```
$ for i in {1..100}; do (./main &); done
```

...then 100 copies of the same code for `add()` will reside in memory ⇒ **waste** of precious memory space!

Dynamic Linking

```c
/* main.c */
#include <stdio.h>
#include "utils.h"

int main() {
    int res = add(3, 4);
    printf("add(3, 4): %d\n", res);
    printf("&add(): %p\n", &add);
    return 0;
}
```

```c
/* utils.c */
#include "utils.h"

int add(int a, int b) {
    return a + b;
}
```

```c
/* utils.h */
#ifndef UTILS_H
#define UTILS_H

int add(int a, int b);

#endif
```

```
$ gcc -c -o main.o main.c
$ gcc -fPIC -shared utils.c -o libutils.so
```

compile into a shared library, or dynamically linked library `libutils.so`.

```
$ gcc main.o -L. -lutils -o main2 -Wl,-rpath=.
```

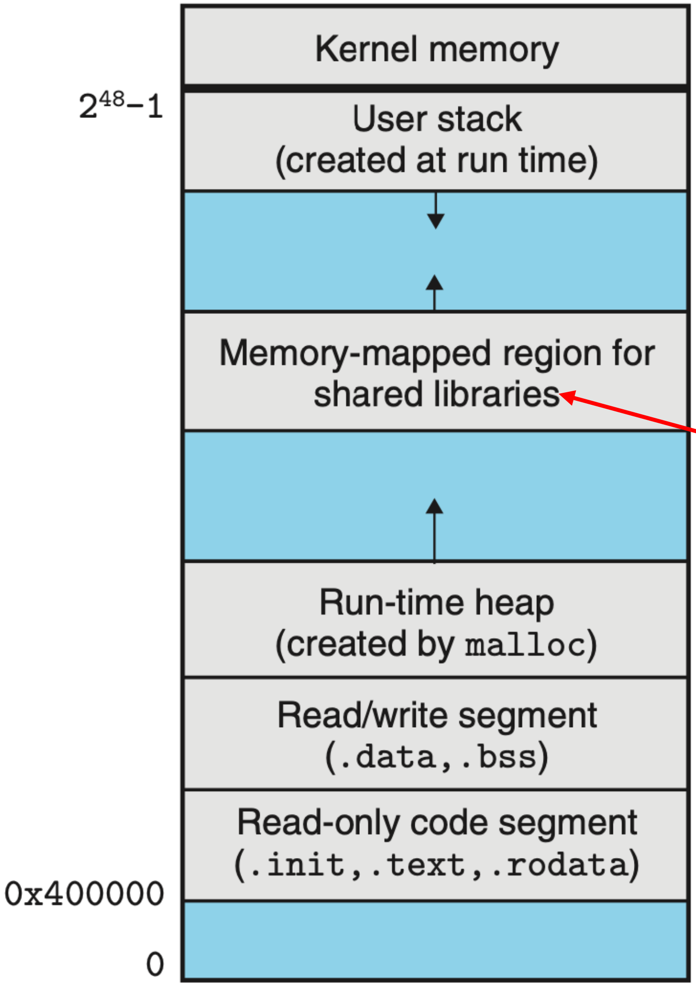
compile the `main` program and link it dynamically with the shared library `libutils.so`:

- `-L.`: tells the **compiler** to look for libraries in the current directory
- `-lutils`: tell the **dynamic linker** to link against `libutils.so`.
- `-Wl,-rpath=.`: tells the **loader** to look for shared libraries in the current directory at runtime

```
$ for i in {1..100}; do (./main &); done
```

...running multiple copies of the same program will only have **ONE** copy of the code for `add()`

Dynamic Linking




```
$ gcc -c -o main.o main.c
$ gcc -fPIC -shared utils.c -o libutils.so
```

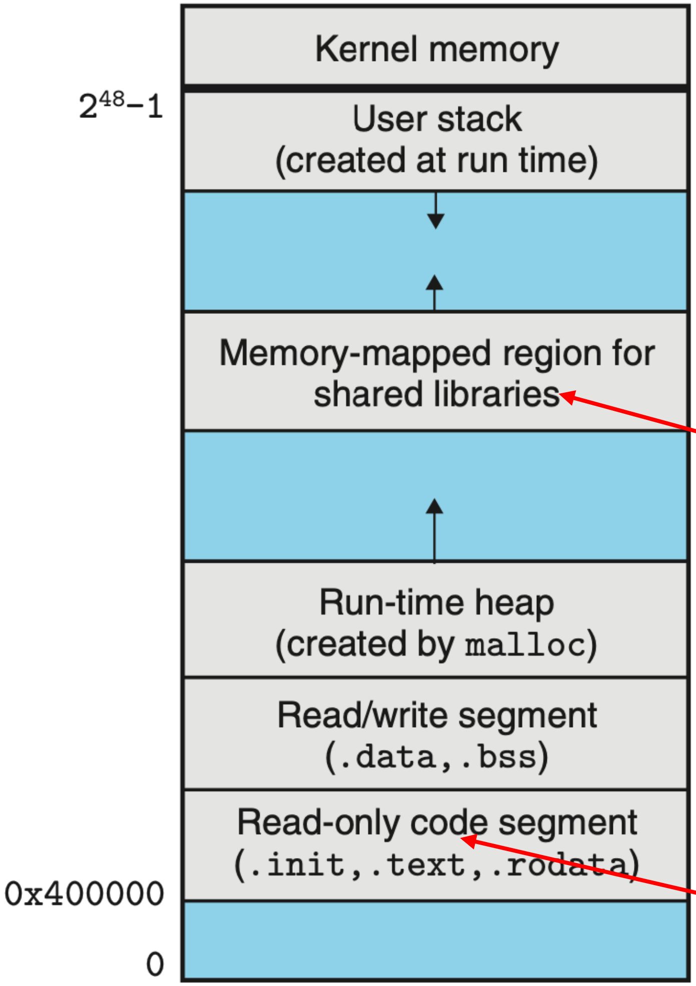
compile into a shared library, or dynamically linked library `libutils.so`.

```
$ gcc main.o -L. -lutils -o main2 -Wl,-rpath=.
```

```
$ ./main2
add(3, 4): 7
&add(): 0x7b68193160f9
```

```
$ ldd ./main2
    linux-vdso.so.1 (0x00007fff4a5f9000)
    libutils.so => ./libutils.so
(0x0000798be6b5e000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6
(0x0000798be6800000)
    /lib64/ld-linux-x86-64.so.2
(0x0000798be6b6a000)
```

Dynamic Linking




```
$ gcc -c -o main.o main.c
$ gcc -fPIC -shared utils.c -o libutils.so
```

compile into a shared library, or dynamically linked library `libutils.so`.

```
$ gcc main.o -L. -lutils -o main2 -Wl,-rpath=.
```

```
$ ./main2
add(3, 4): 7
&add(): 0x7b68193160f9
```

```
$ gcc -c -o main.o main.c
$ gcc -c -o utils.o utils.c
$ gcc -o main -static main.o utils.o
$ ./main
add(3, 4): 7
&add(): 0x401792
```

Dynamic Loading

- It is also possible for a process to request the dynamic linker to load and link arbitrary shared libraries at run time (after it executes, during, while it is running). This technique is called **Dynamic Loading**.

Dynamic Loading

```c
/* main_dl.c */
#include <stdio.h>
#include <dlfcn.h>

int main() {
    void *handle;
    int (*add)(int, int);

    handle = dlopen("./libutils.so",
                    RTLD_LAZY);

    add = dlsym(handle, "add");

    int res = add(3, 4);
    printf("add(3, 4): %d\n", res);
    printf("&add(): %p\n", &add);

    dlclose(handle);
    return 0;
}
```

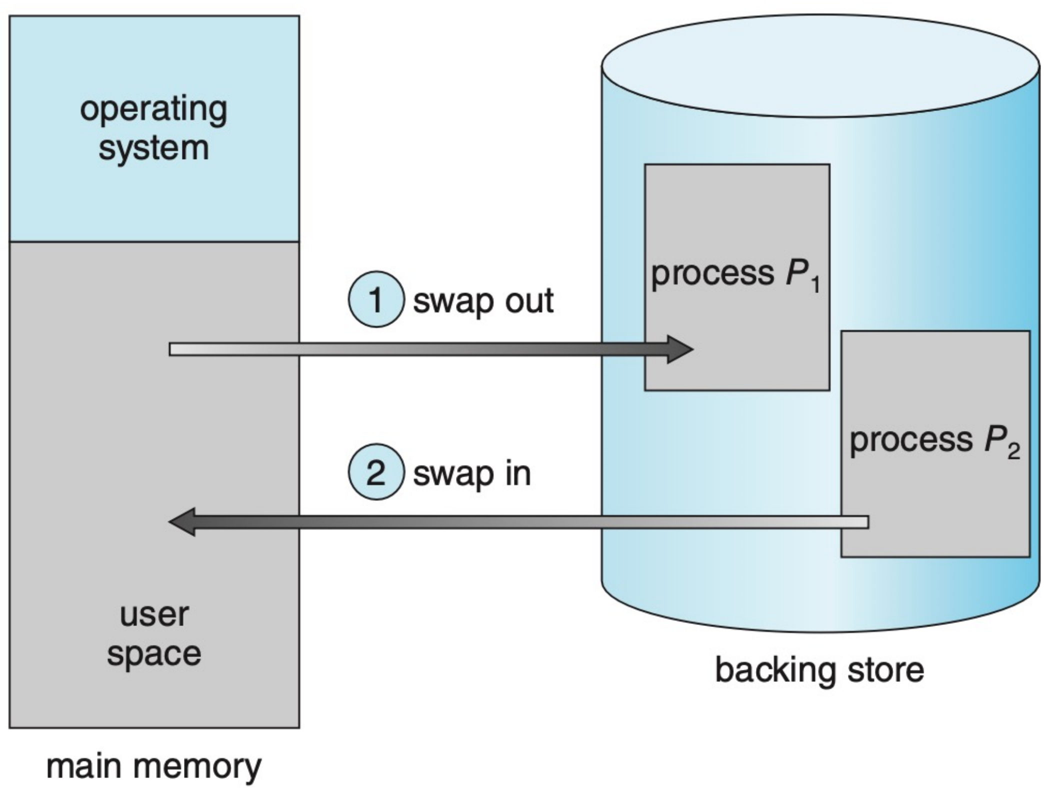
```
$ gcc -fPIC -shared utils.c -o libutils.so
$ gcc -o main_dl main_dl.c -ldl
$ ./main_dl
add(3, 4): 7
&add(): 0x7ffedb9b3ef8
```

- `dlopen` is used to load the shared library into memory at runtime. The library's memory address isn't fixed until this function is called.
- `dlsym` is used to look up the address of the `add` function within the shared library after it has been loaded. This address is resolved at runtime when `dlsym` is called, not before.

- Routine (e.g., `add()`) is not loaded until it is explicitly called ⇒ better memory space utilization
- Useful when large amounts of code are needed to handle infrequently occurring cases
- No special support from the OS is required
    - Implemented through program design
    - OS can help by providing libraries of dynamic loading (e..g, dlopen, dlsym, dlclose)

Swapping

- A process must be **in memory** to be executed.
- However, it can be temporarily **swapped** (**交换**) **out** of memory to a backing store (后备存储), and then brought back into memory for continued execution.



- **Swapping**
  - A process must be **in memory** to be executed.
  - However, it can be temporarily **swapped (交换) out** of memory to a backing store (后备存储), and then brought back into memory for continued execution.
  - Swapping makes it possible for the total physical address space of all processes to **exceed** the real physical memory of the system.
    - thus increasing the degree of multiprogramming.

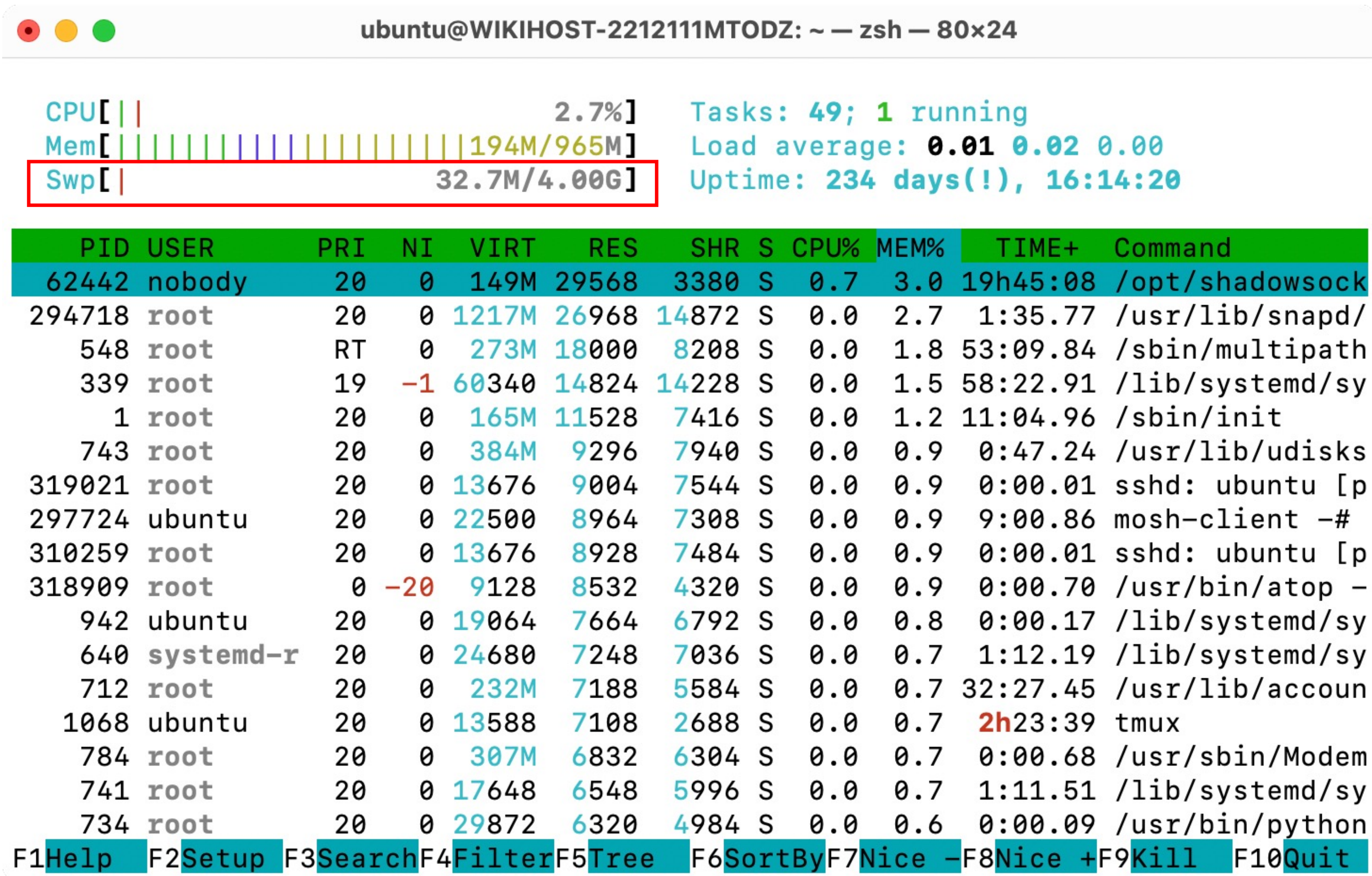
Swapping

Context Switch Time and Swapping

- If next process to be put on CPU is not in memory, then the OS need to swap out a process and swap in the target process.
- Context switch time can be very high.
- 100MB process swapping to disk with transfer rate of 50MB/sec
  - Swap out time: $\frac{100MB}{50MB/s} = 2000ms$
  - Plus swap in of same sized process
  - Total context switch swapping component time of 4000ms
- Other constraints on swapping:
  - Pending I/O – can't swap out as I/O would occur to wrong process.
  - Or always transfer I/O to kernel space, then to I/O device
    - Known as double buffering $\Rightarrow$ more overhead
- Standard swapping not being used in modern OSes
  - Swap only when free memory extremely low.

Context Switch Time and Swapping

- Standard swapping not being used in modern OSes
  - Swap only when free memory **extremely low**.



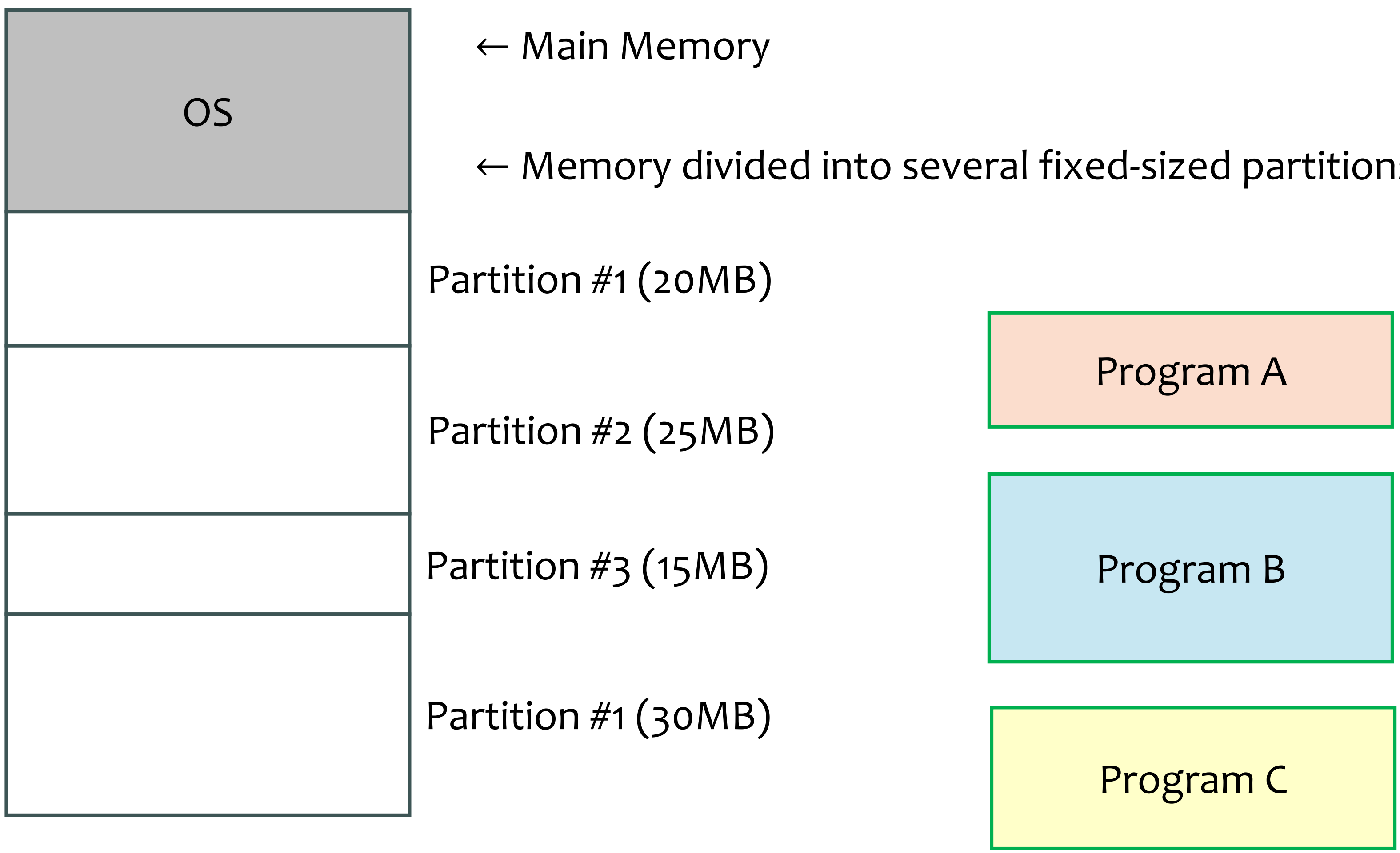
Memory Allocation

- Although the following simple (basic) memory management techniques are no longer used in modern OSes, they lay the ground for a proper discussion of *virtual memory*:
  - Contiguous Memory Allocation
    - **Fixed** (Static) Partitioning （**固定**分区）
    - **Variable** (Dynamic) Partitioning （**可变**分区）
  - Simple **Segmentation** （简单**分段**）
  - Simple **Paging** （简单**分页**）

Memory Allocation

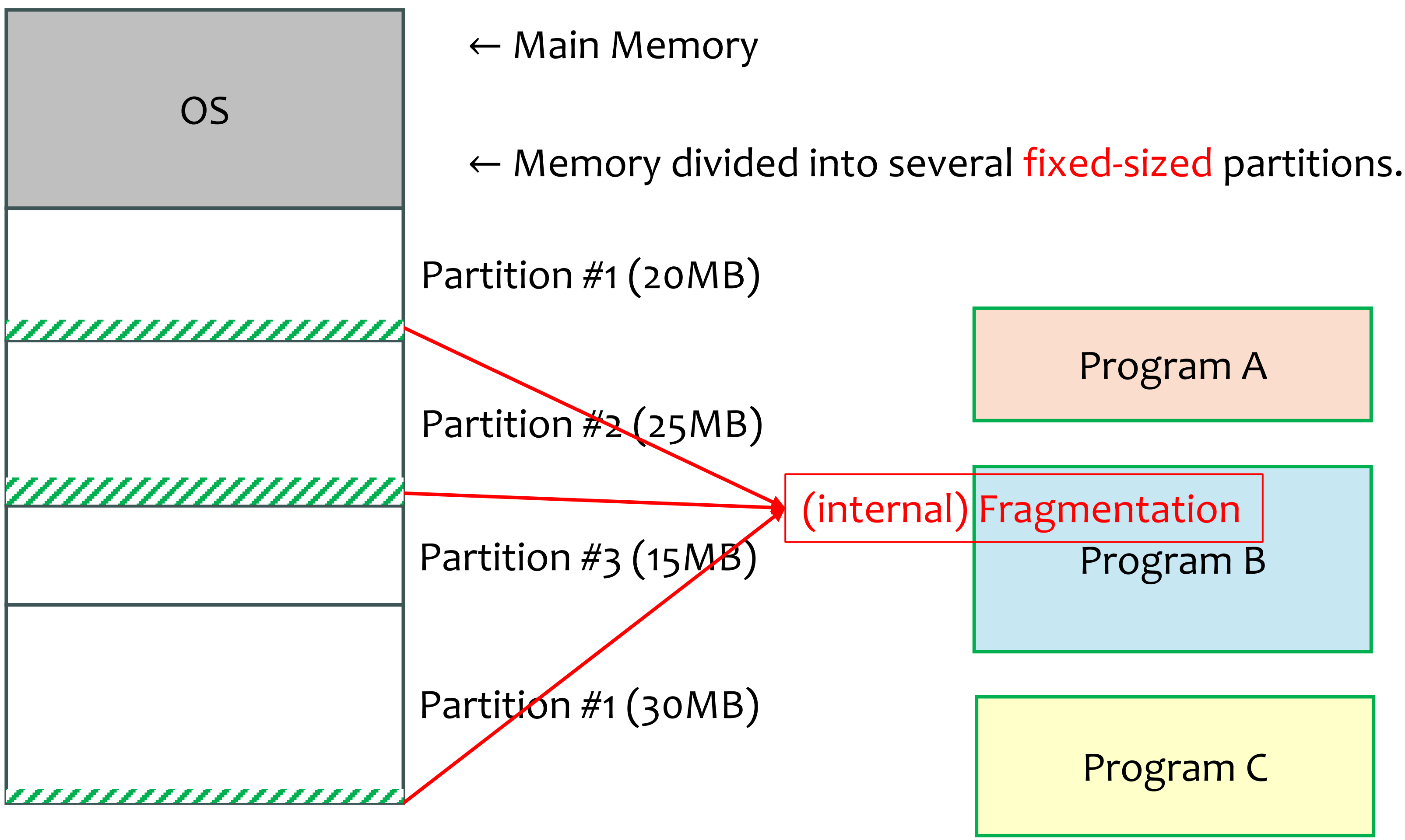
- **Fixed-sized Partitions**
  - One of the simplest methods for allocating memory.
  - Divide memory into several **fixed-sized** partitions
    - The size of the fixed-sized partitions can be **equal**, or **unequal**
  - Each partition may contain **exactly one** process.
  - When a partition is free, a process is selected from the input queue and is loaded into the free partition.
  - When the process terminates, the partition becomes available for another process.

Fixed-sized Partitions



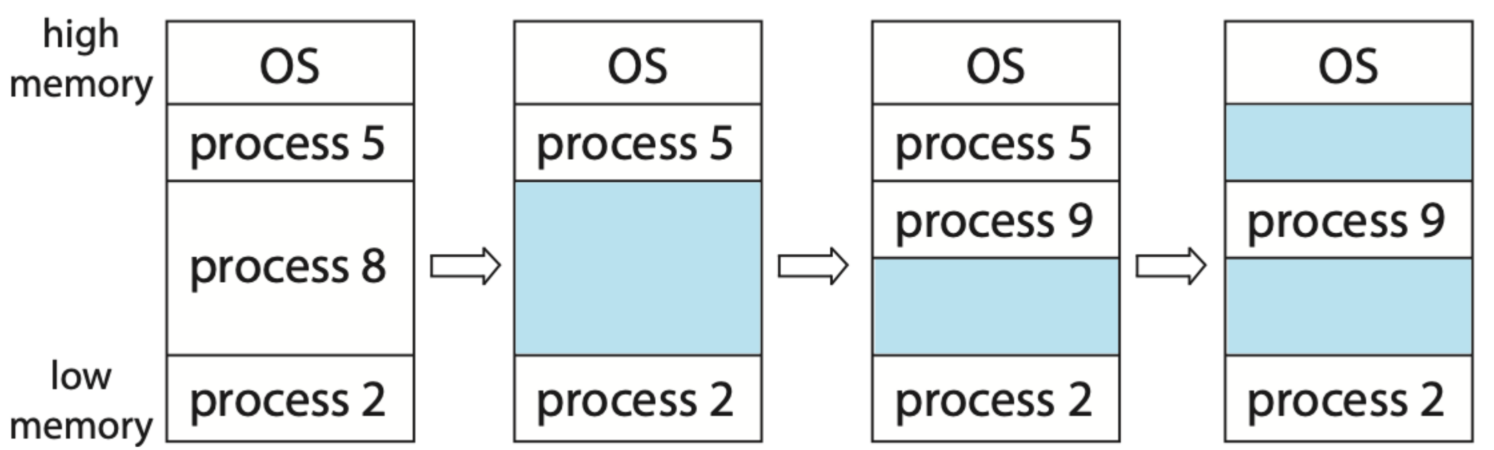
Memory Allocation

Fixed-sized Partitions

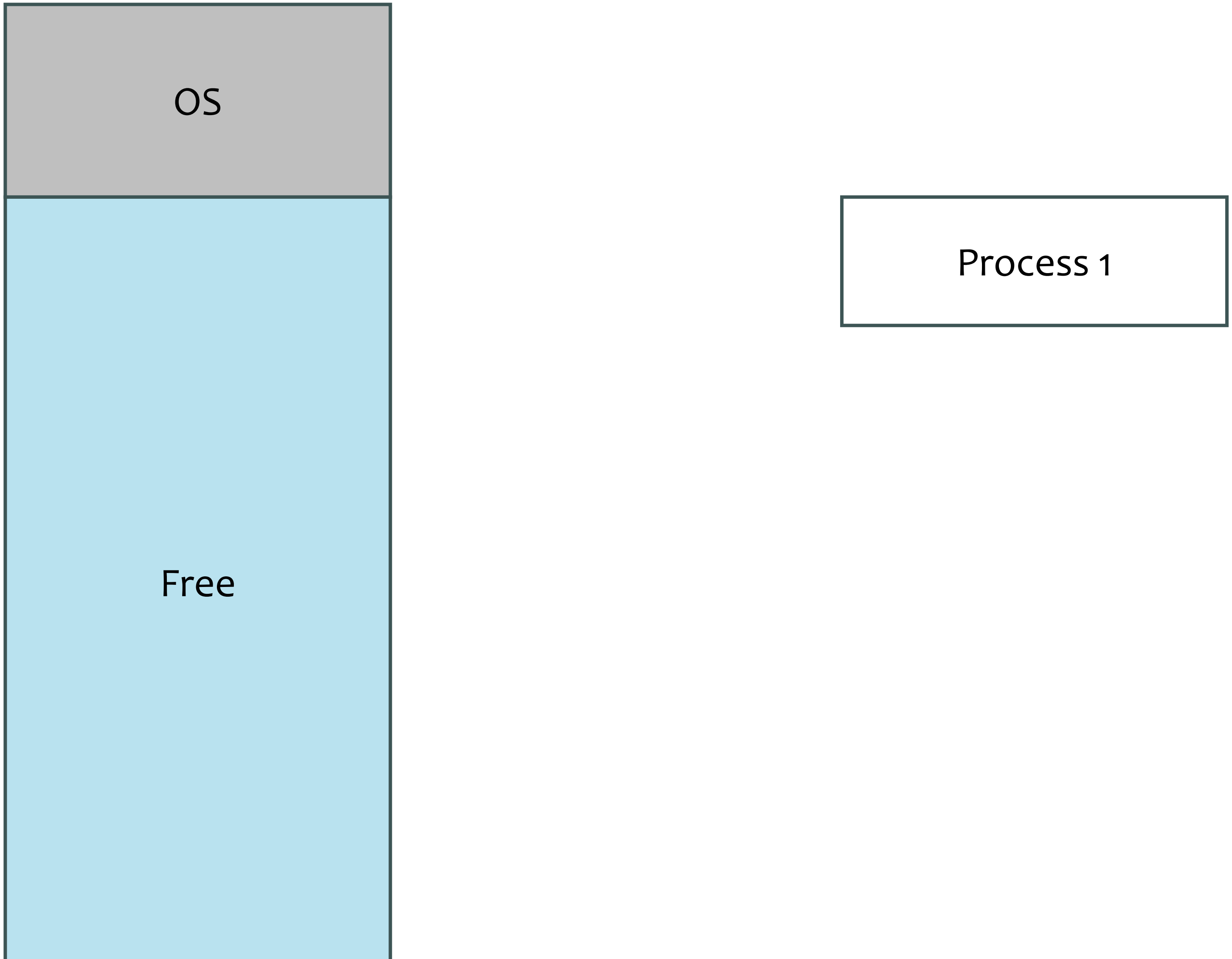


Variable Partitions

- Variable-partition sizes for efficiency (sized to the process's needs)
  - reduce **internal fragmentation**
- **Hole**: block of available memory, also called **free partitions.**
  - holes of various sizes are scattered throughout memory
- When a process arrives, it is allocated memory from a hole large enough to acommodate it.
- When a process exits, its partition is freed
  - adjacent free partitions combined
- The OS maintains info about: 1) allocated partitions; 2) free partitions

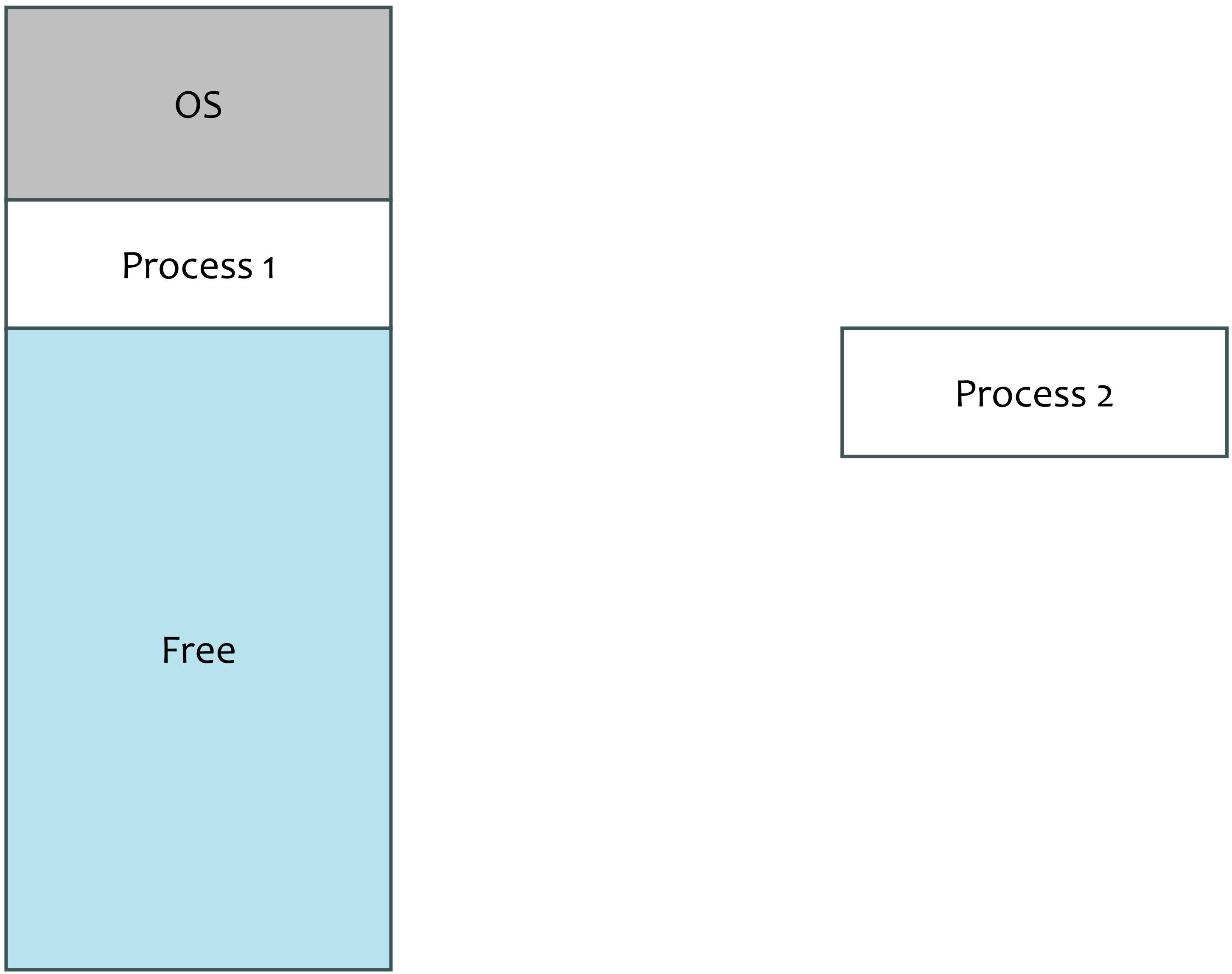


Variable Partitions Example

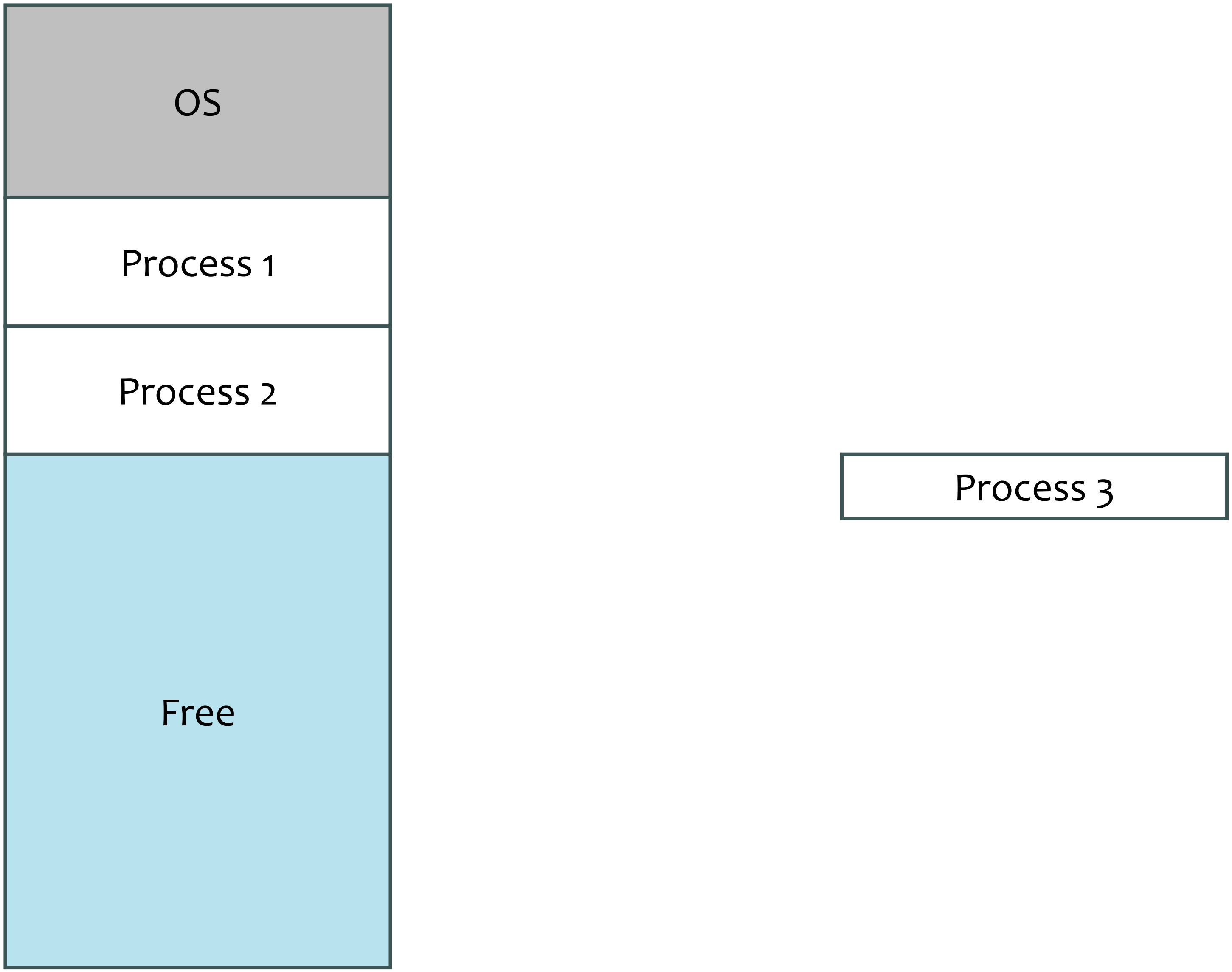


Memory Allocation

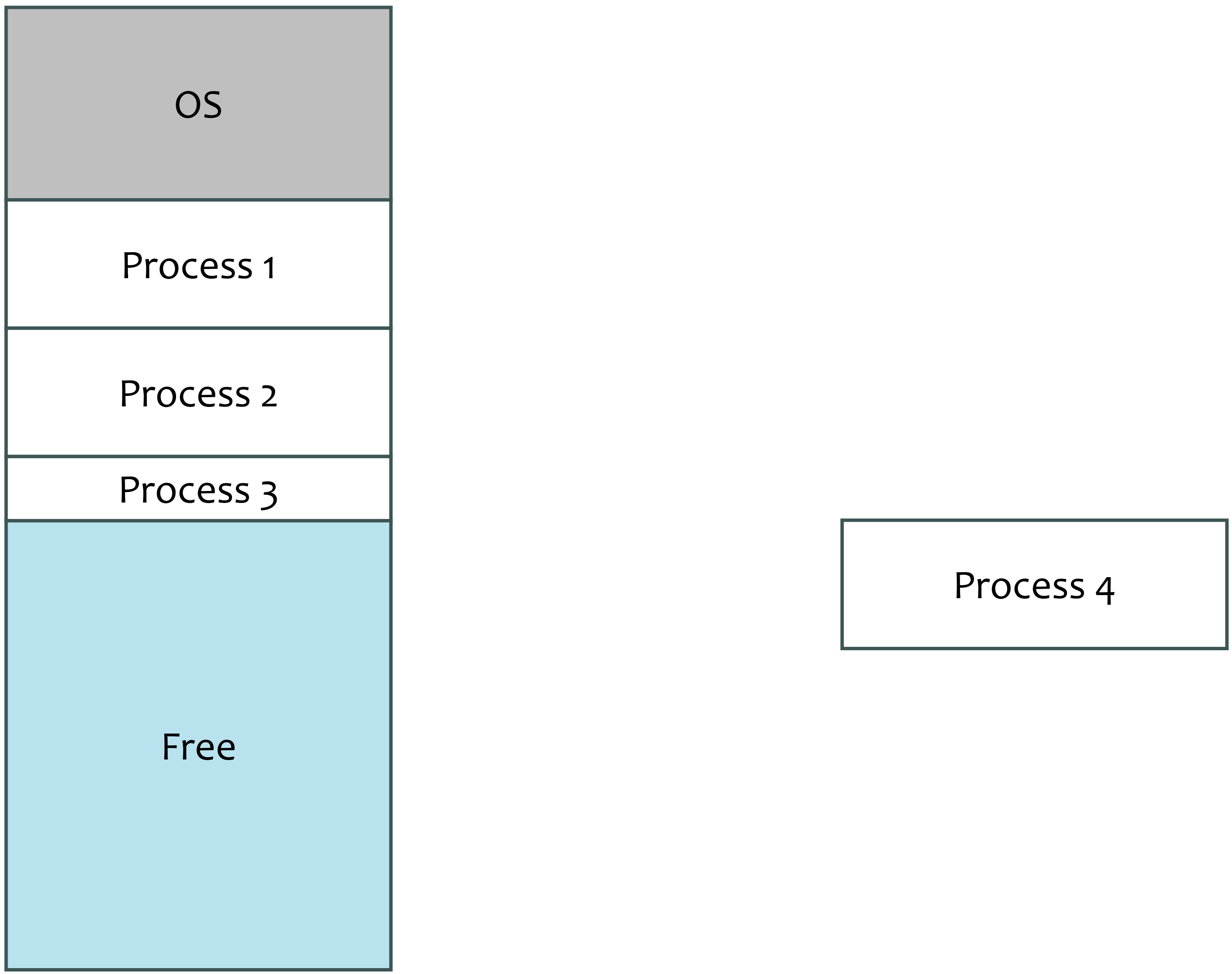
Variable Partitions Example



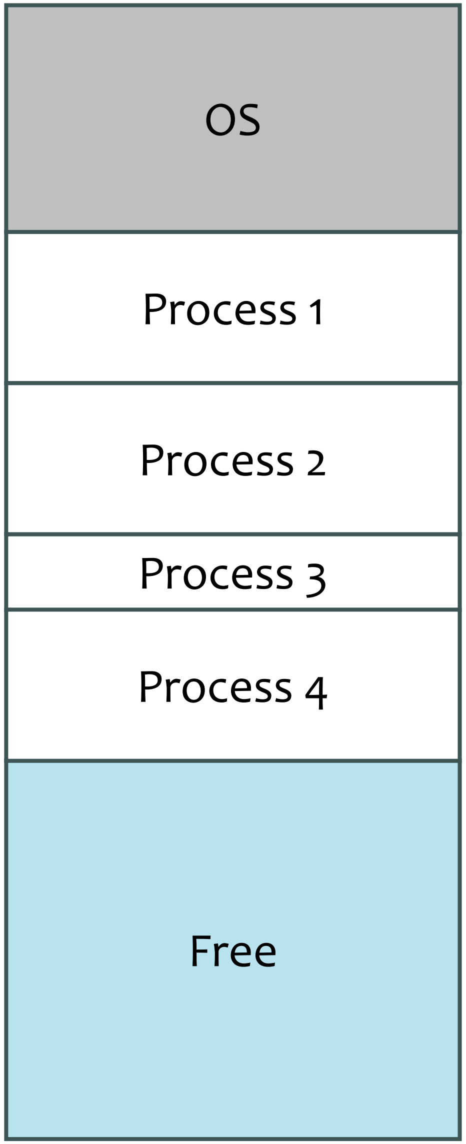
Variable Partitions Example



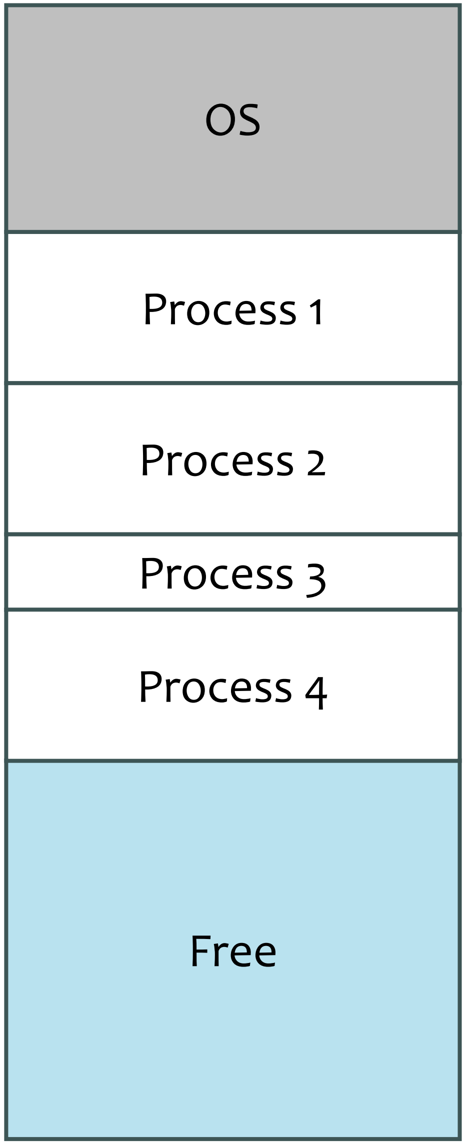
Variable Partitions Example



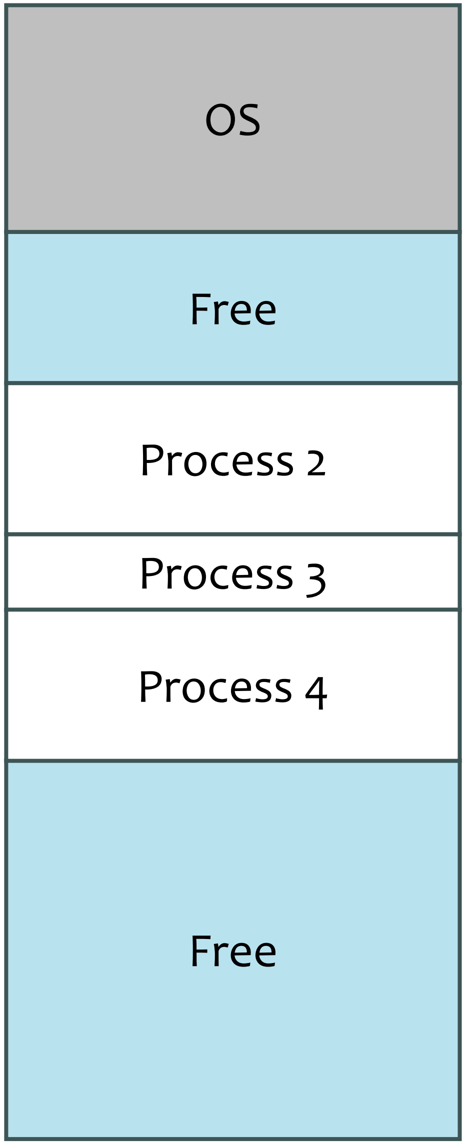
Variable Partitions Example



Variable Partitions Example



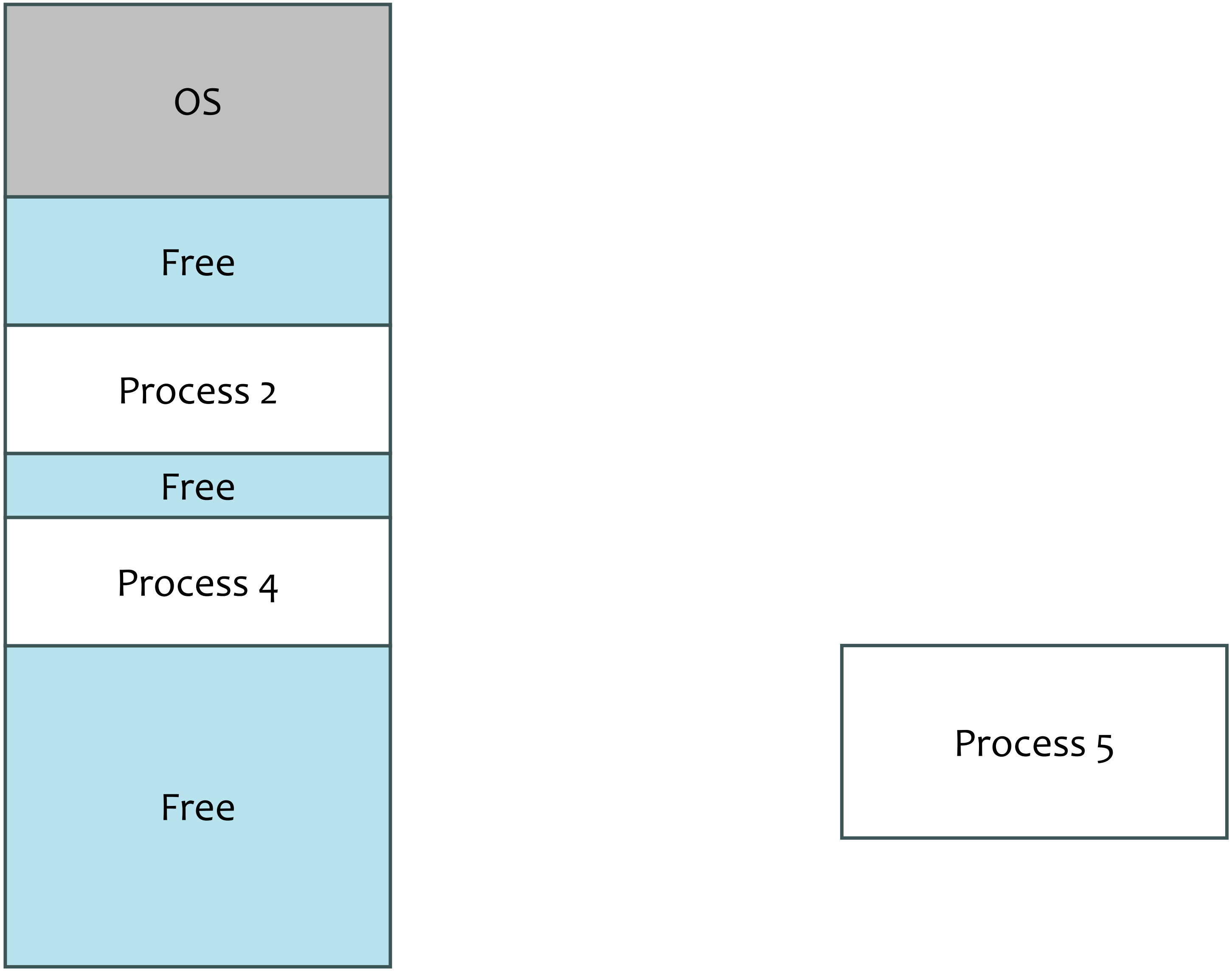
Variable Partitions Example



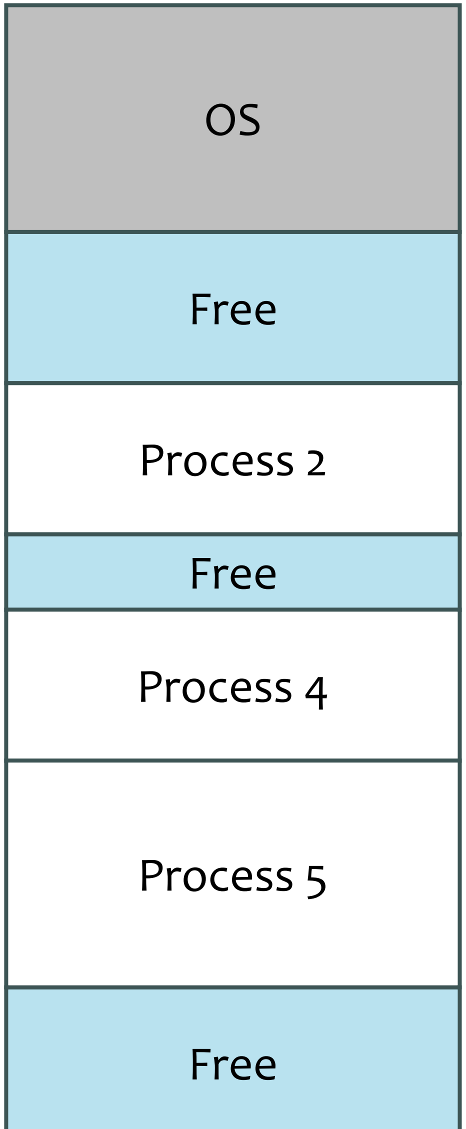
Variable Partitions Example

Memory Allocation

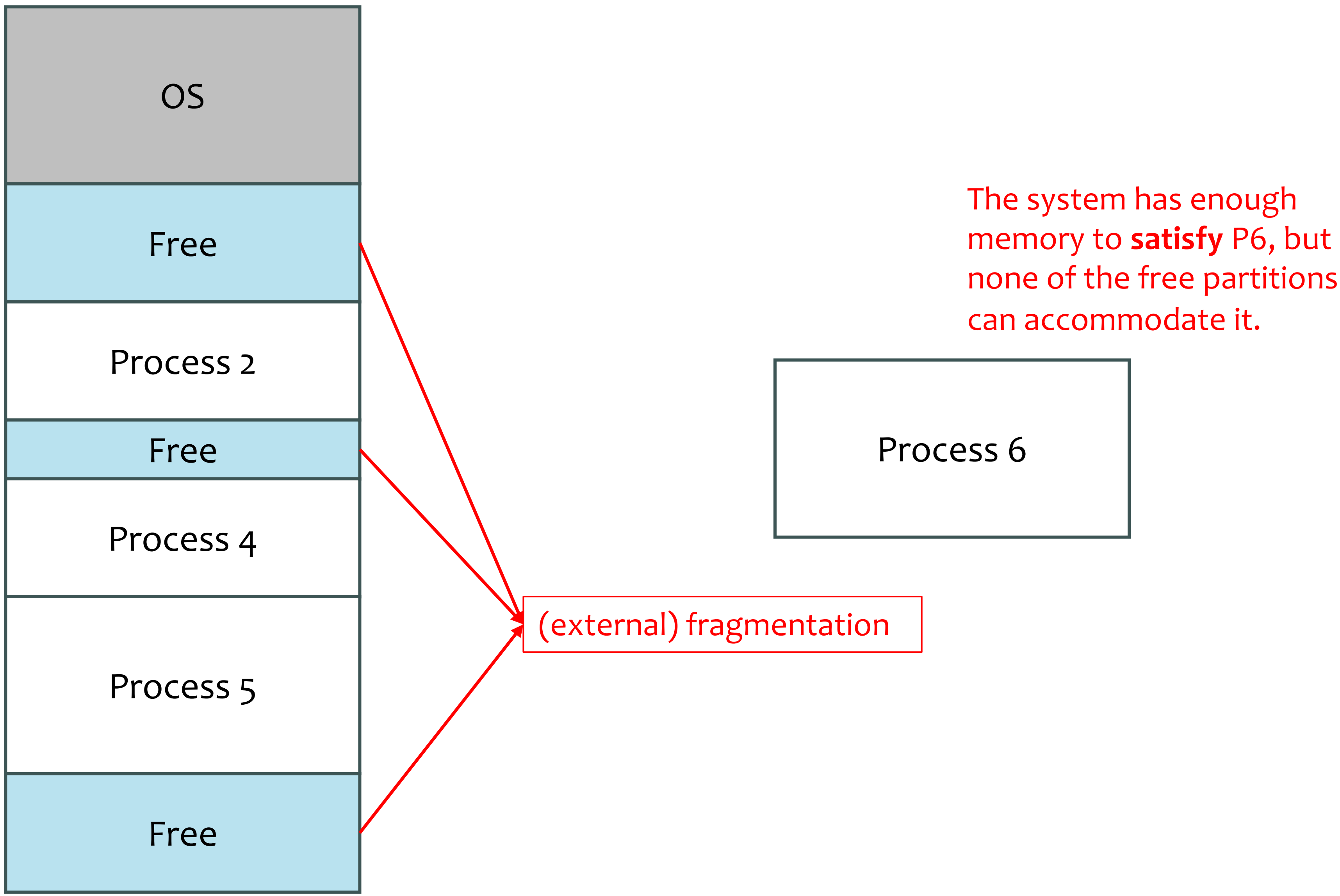
Variable Partitions Example



Variable Partitions Example

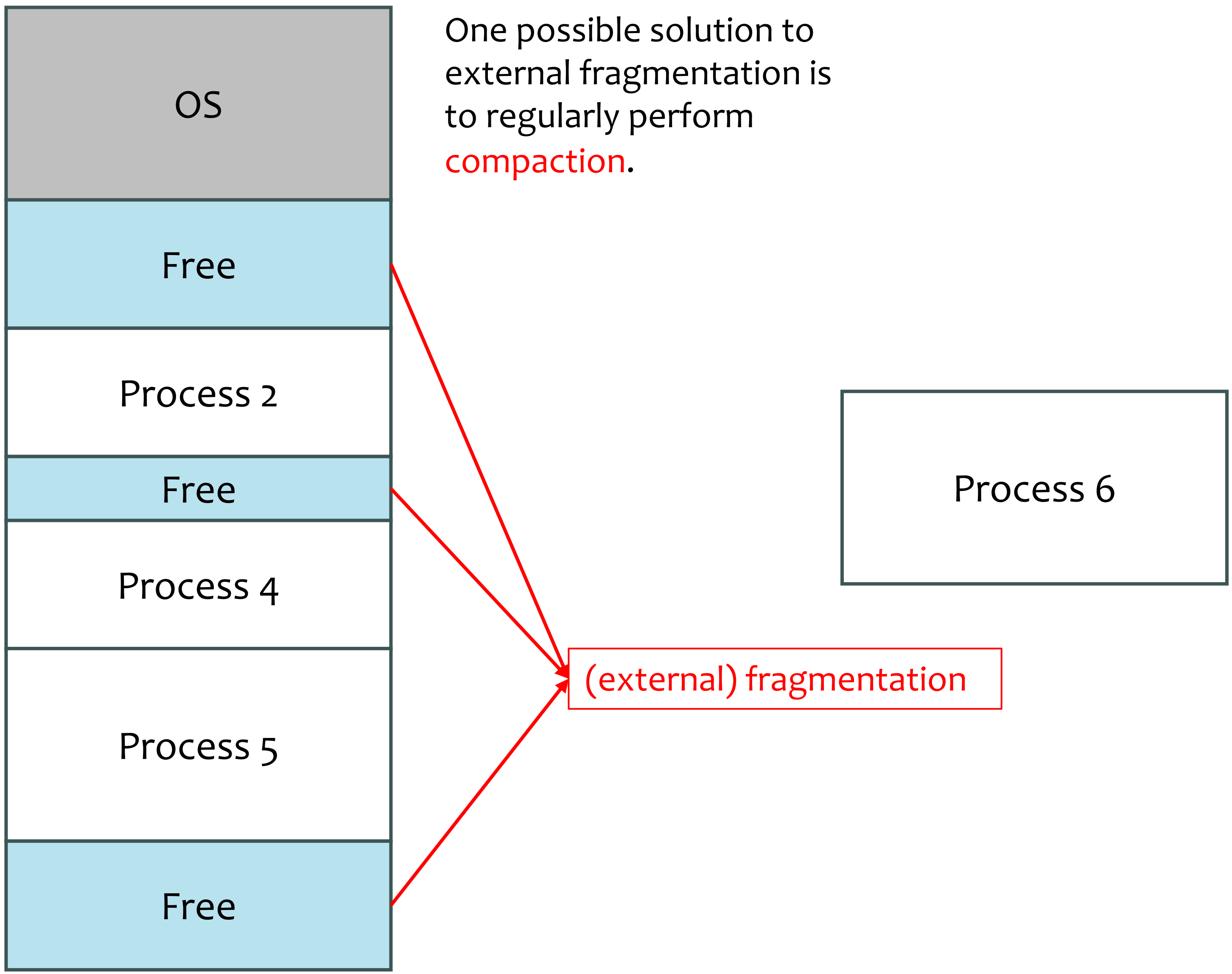


Variable Partitions Example

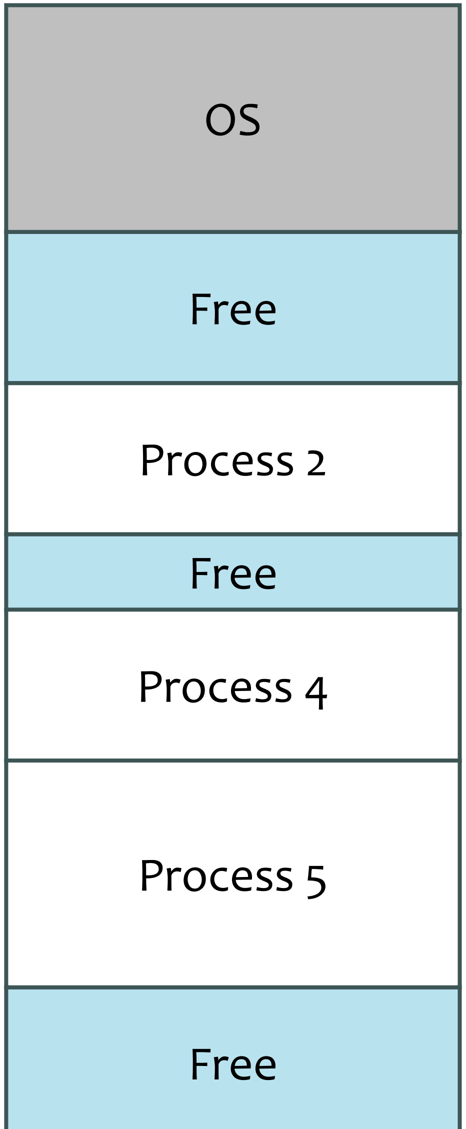


Variable Partitions Example

One possible solution to external fragmentation is to regularly perform compaction.



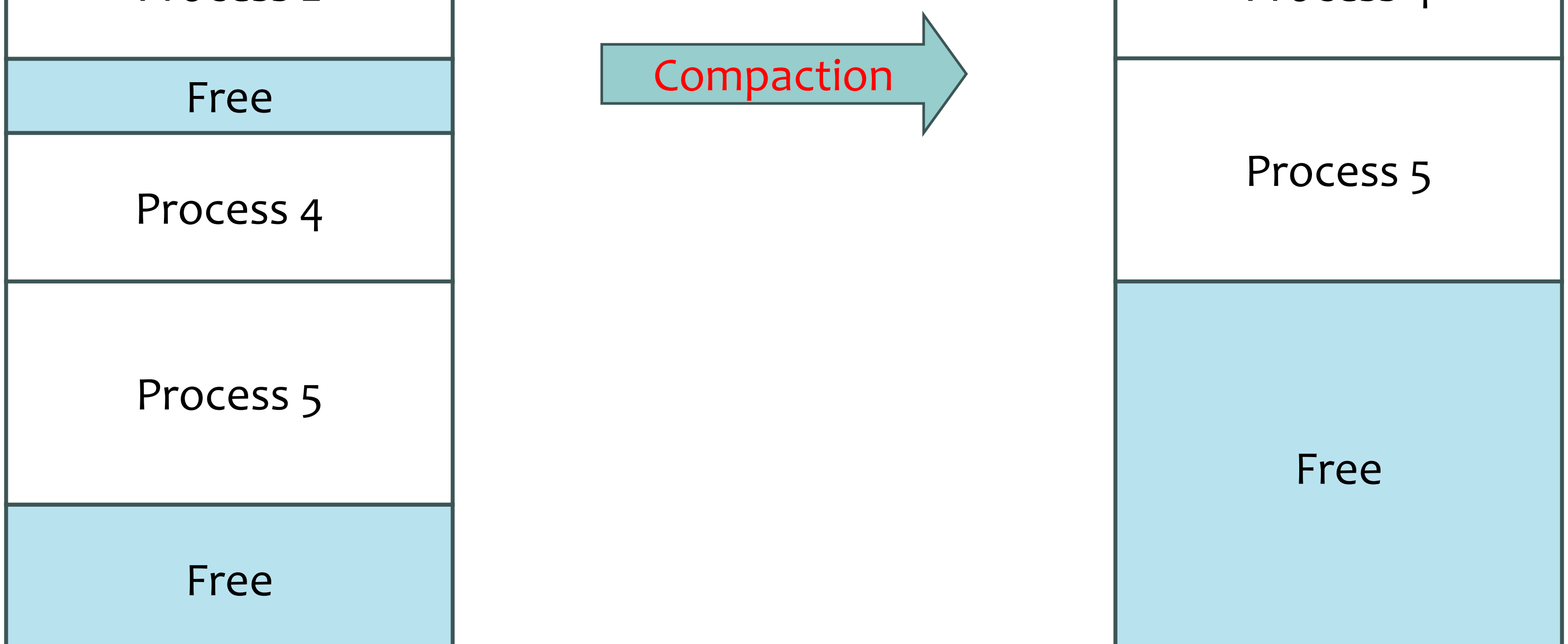
Variable Partitions Example



One possible solution to external fragmentation is to regularly perform compaction.

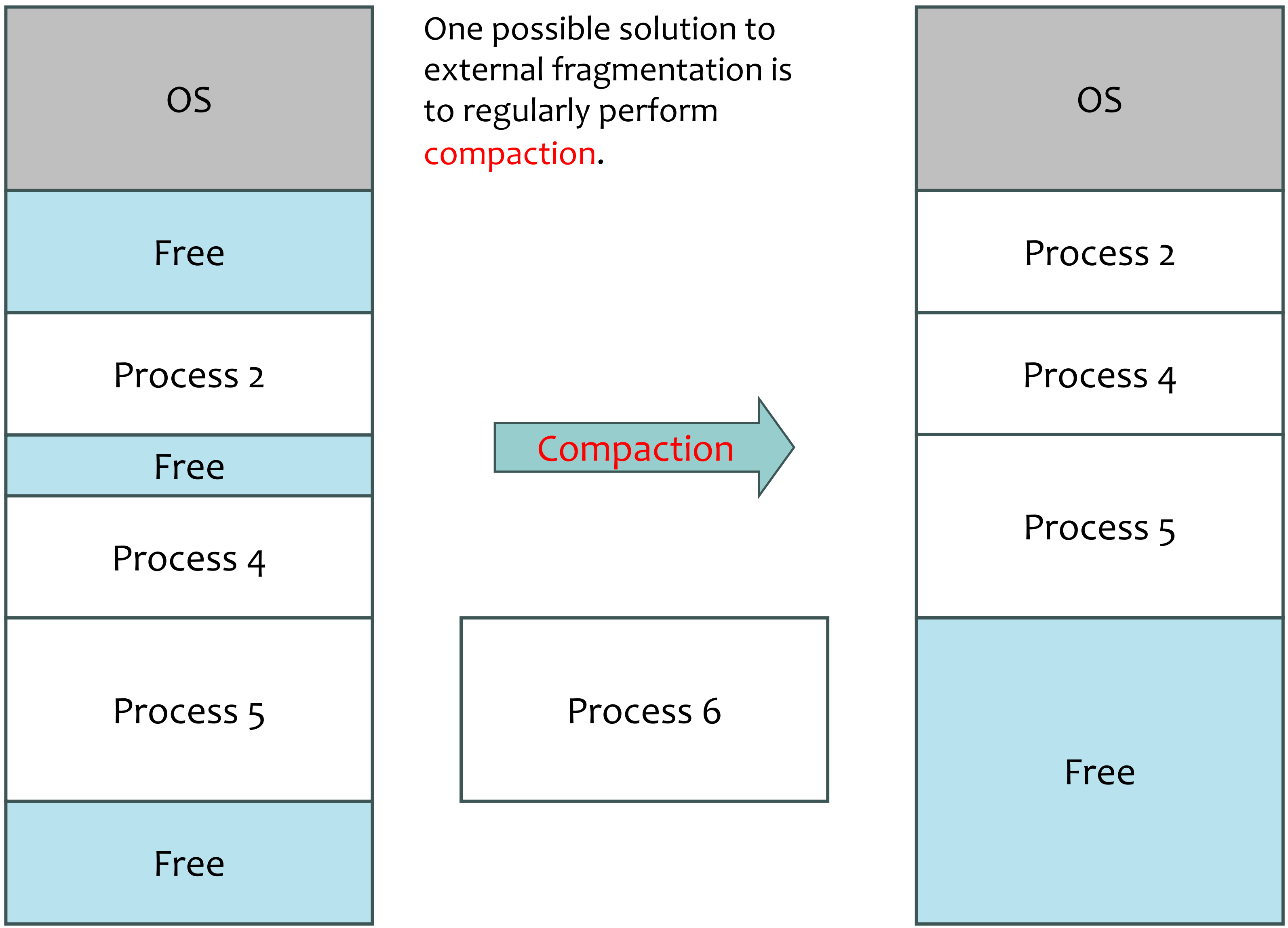
Variable Partitions Example

One possible solution to external fragmentation is to regularly perform compaction.



Variable Partitions Example

One possible solution to external fragmentation is to regularly perform compaction.



- **Contiguous Memory Allocation**
  - **Fixed** Partitions
    - Memory usage is inefficient.
    - Any program, no matter how small, occupies an entire partition
      - leads to serious **internal** **fragmentation**.
  - **Variable** Partitions
    - Partitions are of variable length
    - Number of partitions are variable
    - Each program is allocated exactly as much memory as requested
    - Eventually holes (free partitions) are formed.
      - leads to **external** **fragmentation**.
      - Compaction can be used to combine different (small) holes into one big trunk of free partition.
        - However, compaction is **not** always possible.
        - It is also very **expensive**, since memory operations consume many CPU cycles.

Dynamic Storage-Allocation Problem

- How to satisfy a request of size **n** from a list of holes (free partitions)?
  - First-fit: Allocate the **first** hole that is big enough
  - Best-fit: Allocate the **smallest** hole that is big enough
    - must search entire list of holes, unless ordered by size
  - Worst-fit: Allocate the **largest** hole
    - must also search entire list
    - Produces the largest leftover hole
- Simulations have shown that both first-fit and best-fit are better than worst-fit in terms of speed and storage utilization.

- **Fragmentation**
  - External **Fragmentation**: total memory space exists to satisfy a request, but it is not contiguous
  - Internal **Fragmentation**: allocated memory may be slightly larger than requested memory; the size difference is memory internal to a partition, but not being used.
  - First-fit analysis reveals that given N blocks allocated, 0.5N blocks might be lost to fragmentation
    - 1/3 may be unusable ⇒ 50-percent rule.

Thank you!