INF113: Introduction to Memory

Kirill Simonov

01.10.2025

Program address space

How does the memory of a process look like?

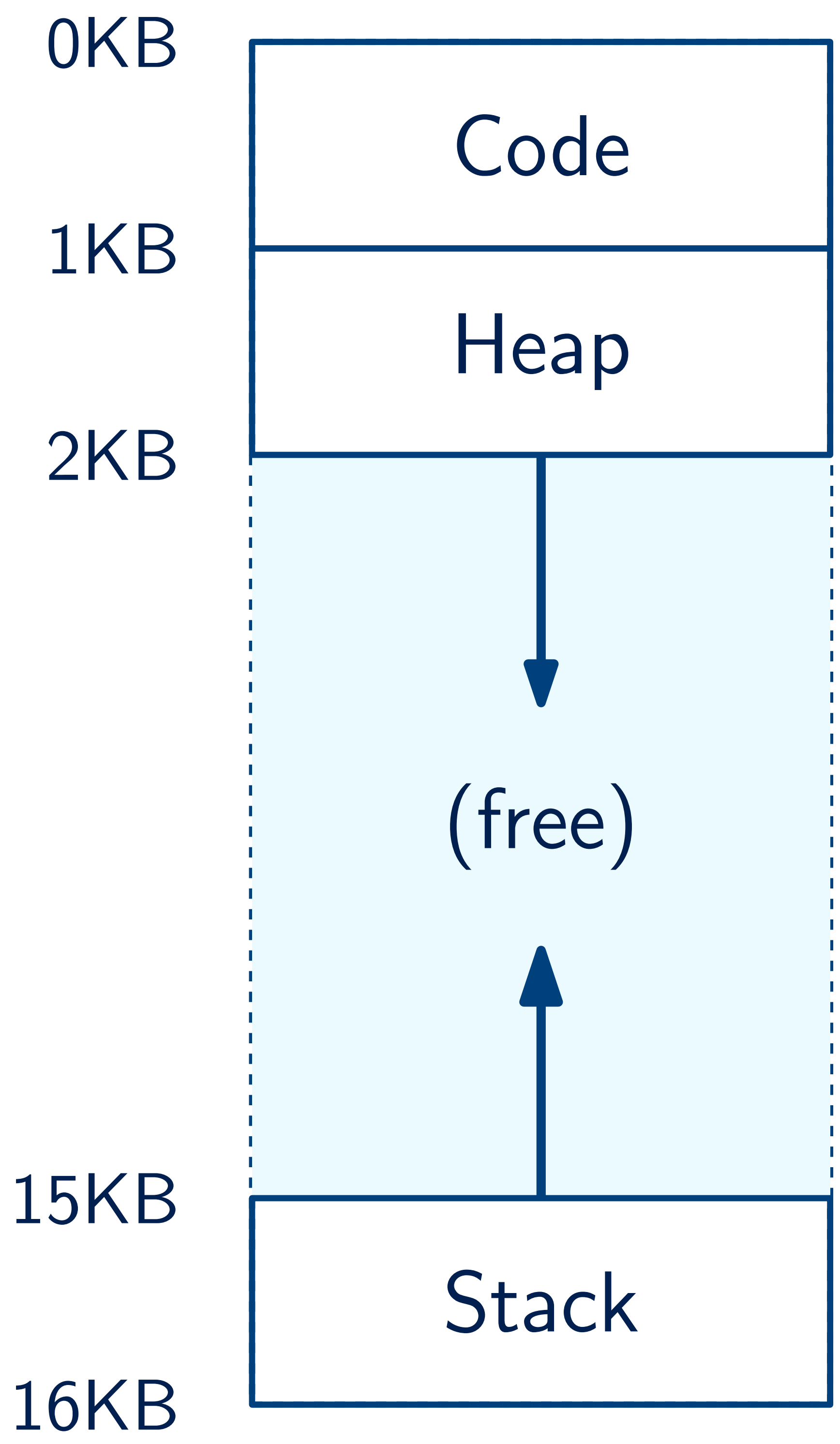
- First **static** data: code, global vars
- Then the **heap**: dynamic allocation

```
int* a = malloc(n * sizeof(int));
```

- Free space afterwards
- At the end, the **stack**: local variables

```
void rec(int d) {
  ...
}
```

- Both heap and stack will grow dynamically



Memory issues

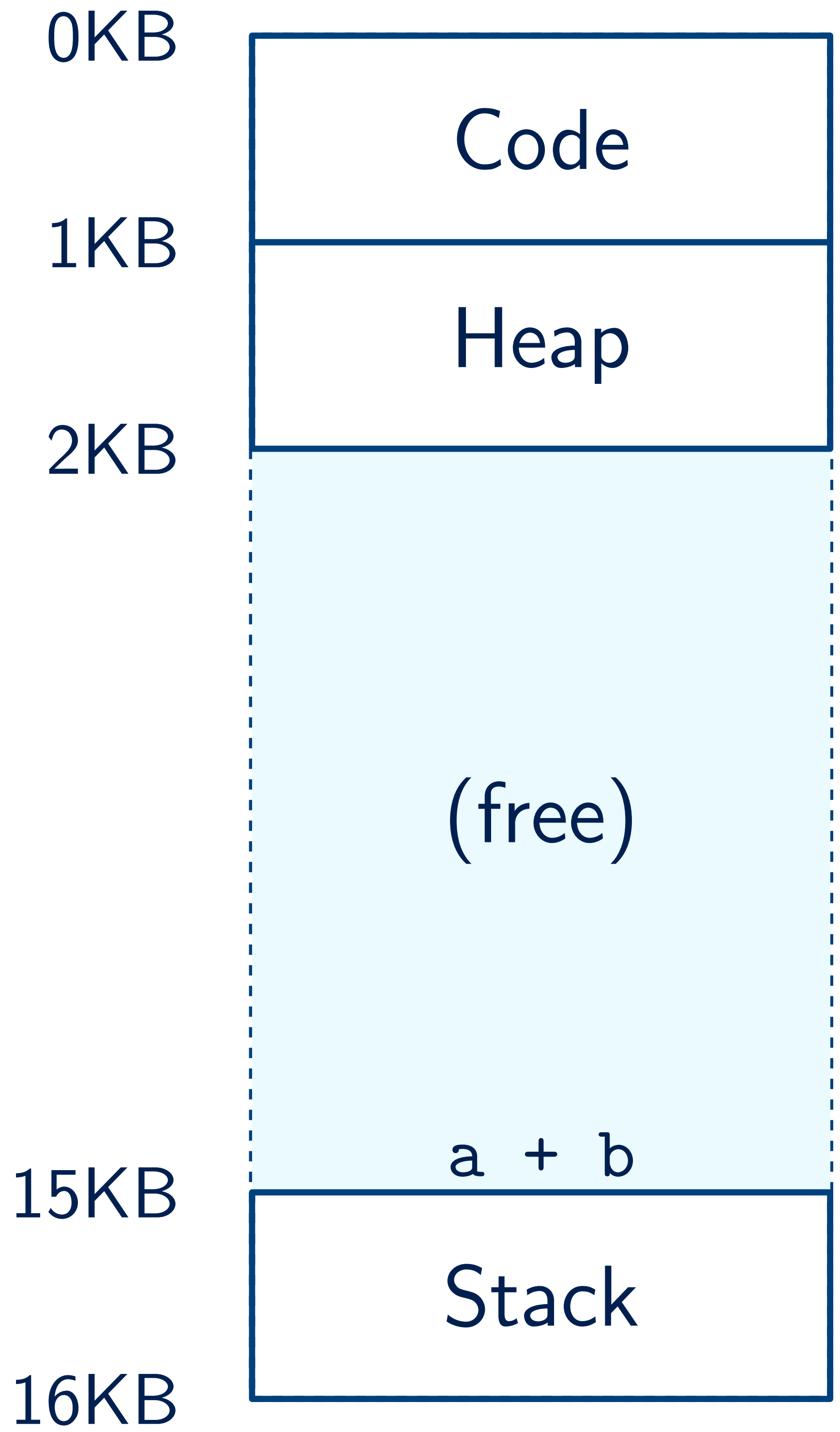
- Memory leaks: explicitly free allocated memory

```c
int* a = malloc(n * sizeof(int));
...
free(a);
```

- Even in higher-level languages:
  - Automatic deallocation via garbage collector
  - But an object will not be deleted while there is still a reference to it
- Dangling pointers: make sure the address is still available

```c
int* add(int a, int b) {
  int sum = a + b;
  return &sum;
}
```

- Addressing invalid memory might not produce any error!



Valgrind

- Install `sudo apt install valgrind`
- Compile your C program with `-g` `gcc -g -o program program.c`
- Run within `valgrind` `valgrind ./program`
- Look for leaks and errors:

```
==95170== Invalid write of size 4
==95170==    at 0x109199: main (merror2.c:7)
==95170==  Address 0x4a87068 is 0 bytes after a block of size 40
==95170==    at 0x4846828: malloc (in /usr/...)
==95170==    by 0x10917E: main (merror2.c:5)
```

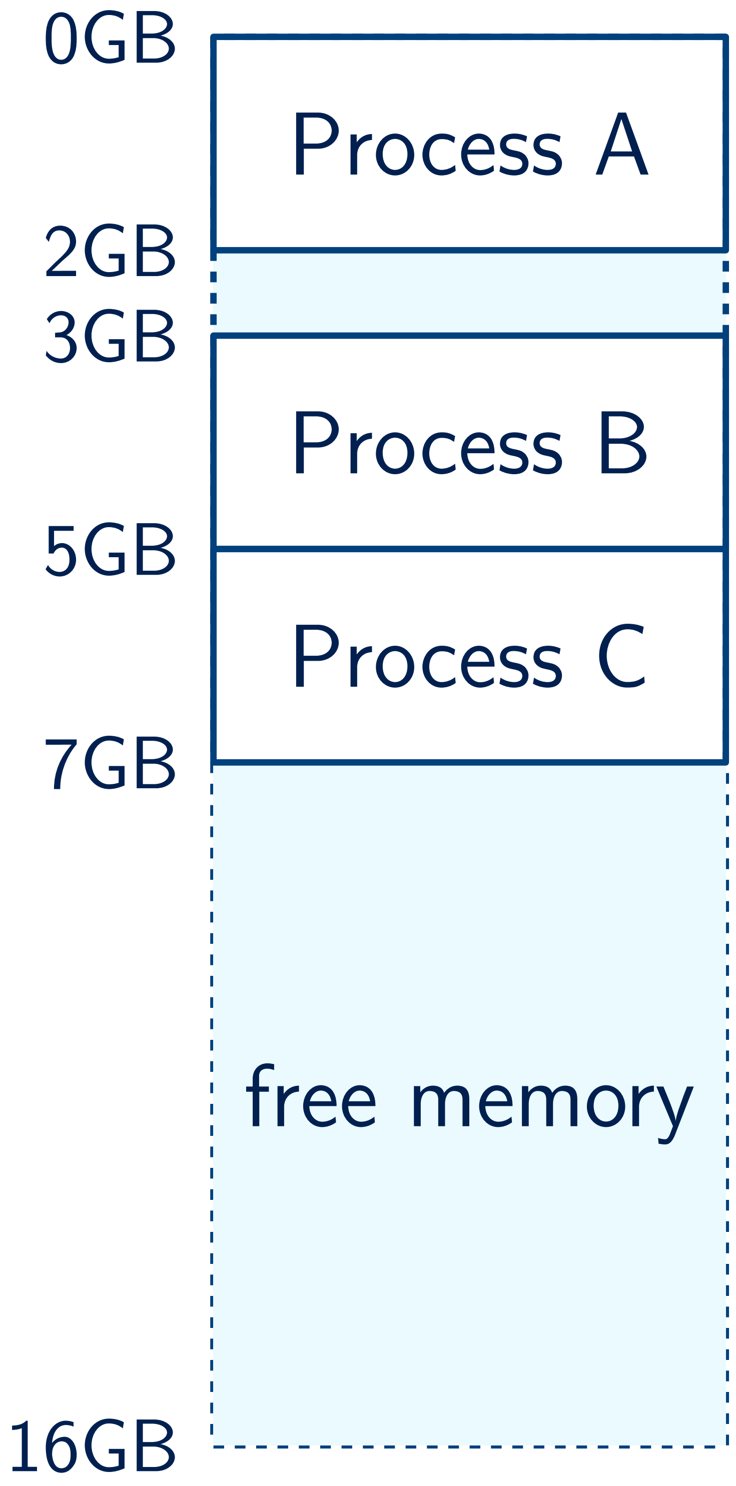
```
==95092== LEAK SUMMARY:
==95092==    definitely lost: 40 bytes in 1 blocks
```

```
==94931== Conditional jump or move depends on uninitialised value(s)
```

Memory virtualization

OS provides a virtual memory address space to each process

- **Abstraction:** The process can access virtual addresses as if it had direct control over the physical memory
- **Protection:** The process cannot access memory of other processes/OS
- **Efficiency:**
  - Transitioning from a virtual address to the physical address should be very fast
  - Avoid moving large chunks of memory
  - Avoid allocating much more memory to a process than it needs
  - ...



Address translation

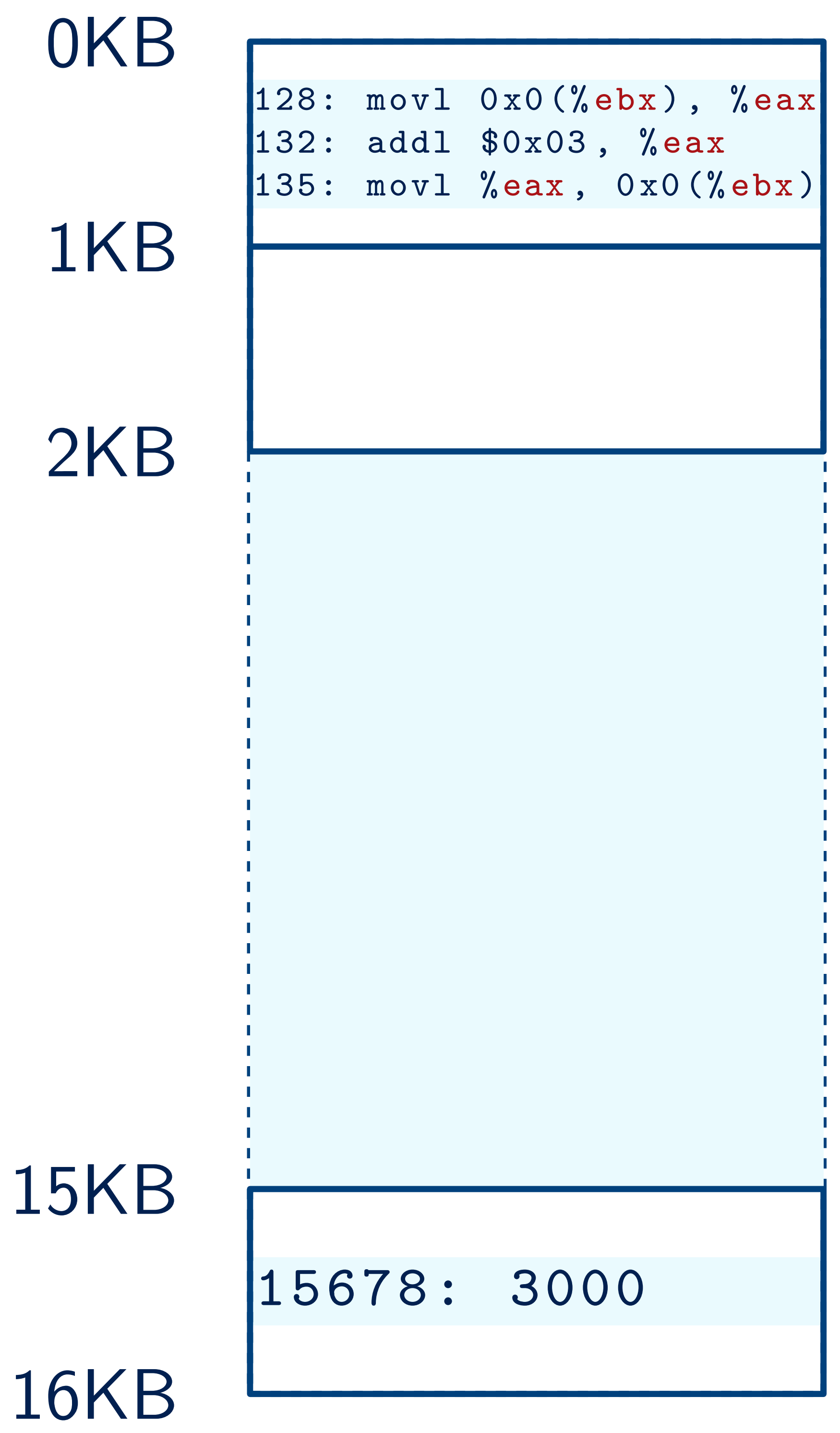
```c
int main() {
  int x = 3000;
  x = x + 3;
}
```

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

CPU cycle:

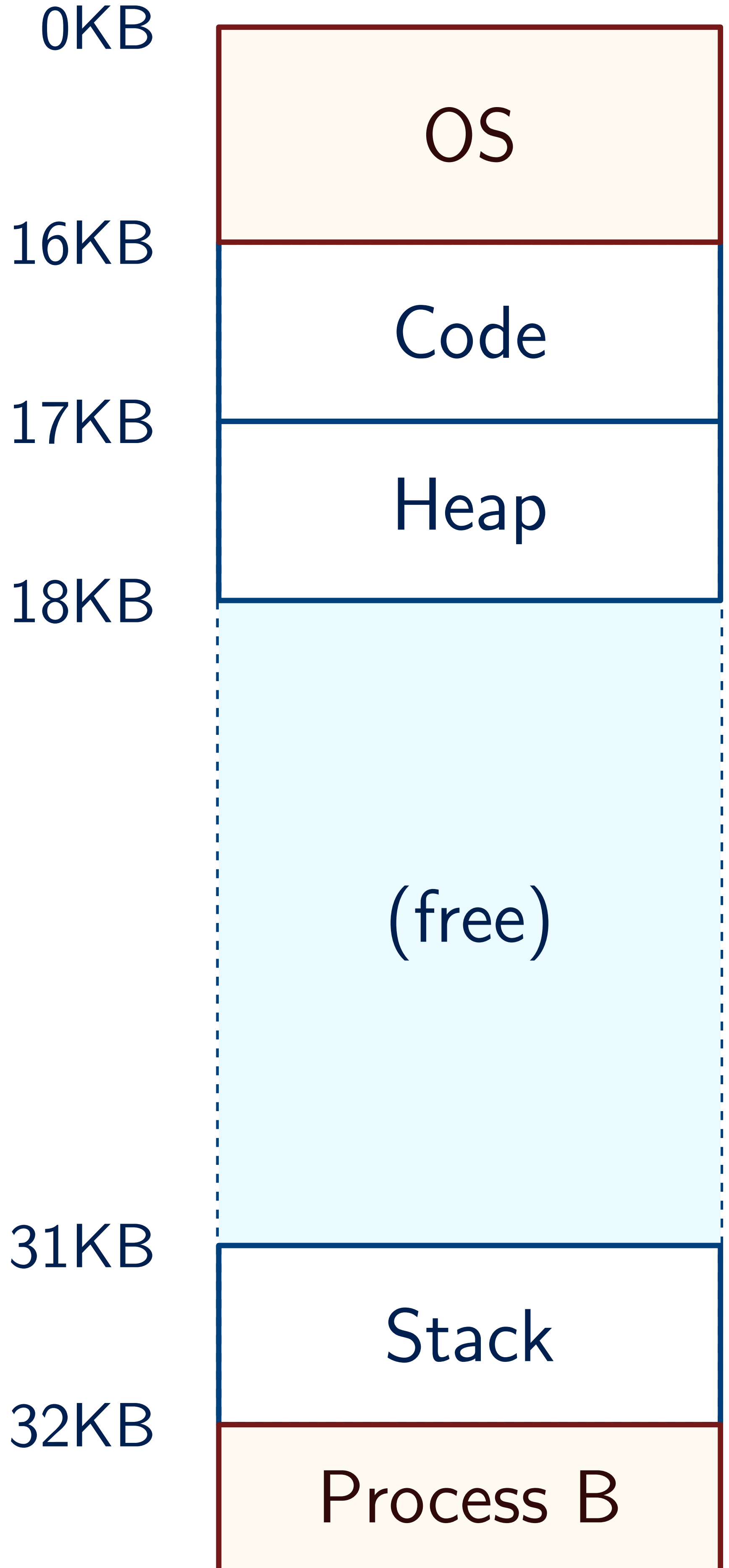
1. Fetch instruction at 128
2. Execute: Move from 15678 to eax
3. Fetch instruction at 132
4. Execute: Add to eax
5. Fetch instruction at 135
6. Execute: Move eax to 15678

program address space



Address translation

physical memory:



The program is not actually at address 0!

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

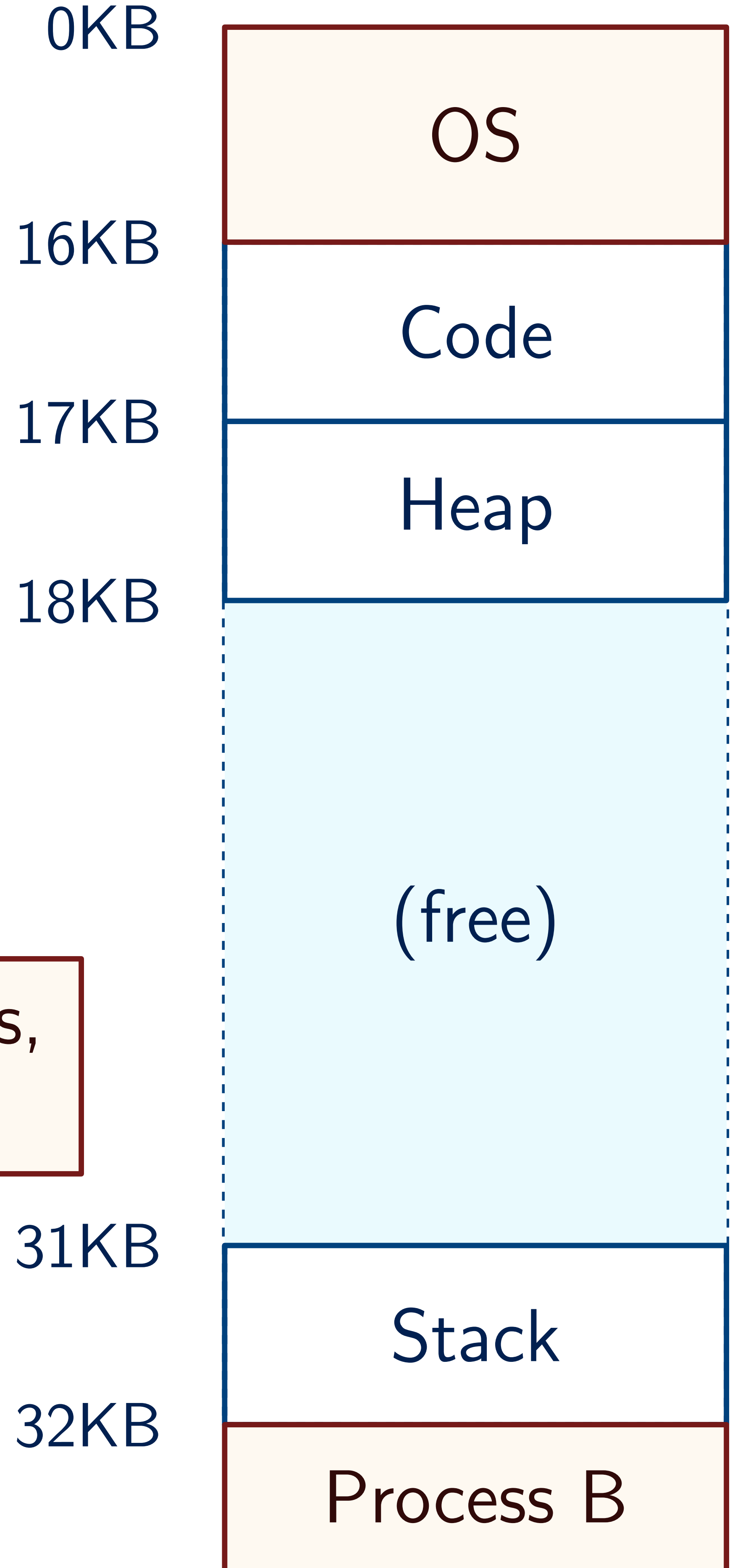
```
128: movl 0x0(%ebx), %eax
132: addl $0x03, %eax
135: movl %eax, 0x0(%ebx)
```

CPU cycle: (intended)

1. Fetch instruction at **16128**
2. Execute: Move from **31678** to eax
3. Fetch instruction at **16132**
4. Execute: Add to eax
5. Fetch instruction at **16135**
6. Execute: Move eax to **31678**

Address translation

physical memory:




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

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

The program is not actually at address 0!

How does the program/CPU know how to adjust instructions, depending on where the program is loaded in memory?

3. Fetch instruction at **16132**
4. Execute: Add to eax
5. Fetch instruction at **16135**
6. Execute: Move eax to **31678**

Attempt 1: Loader

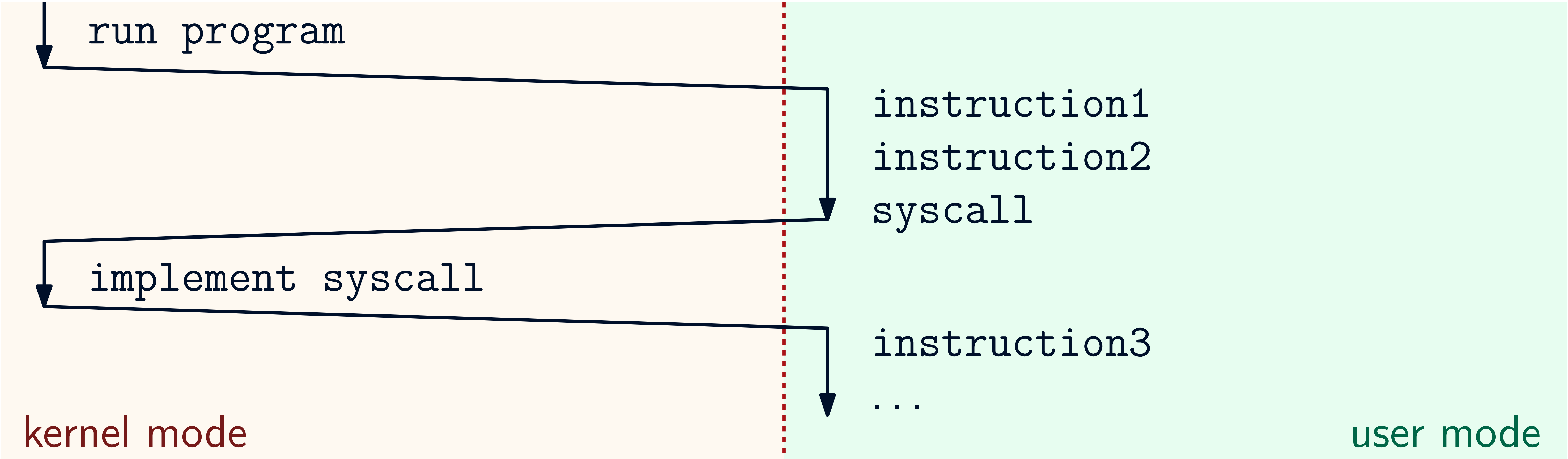
- OS can run the **loader** once it knows the target memory location
- Loader will increment all addresses in the program code by the offset in the memory
- Pros:
  - No additional hardware requirements
  - Small overhead after loading
- Cons:
  - Complicated to implement
  - Costly preprocessing
  - **No protection!**

```
movl 0x0(%ebx), %eax
```

```
addl $0x1000, %ebx
movl 0x0(%ebx), %eax
subl $0x1000, %ebx
```

Attempt 2: Context switch

- We already have **limited direct execution**: pass control to the OS when needed
- OS knows offsets of all address spaces in memory, so it can both
  - Adjust the address from virtual to physical
  - Check that the address is within the program's space
- Cons: context switch is **100x** slower than addressing memory...



Base and bound

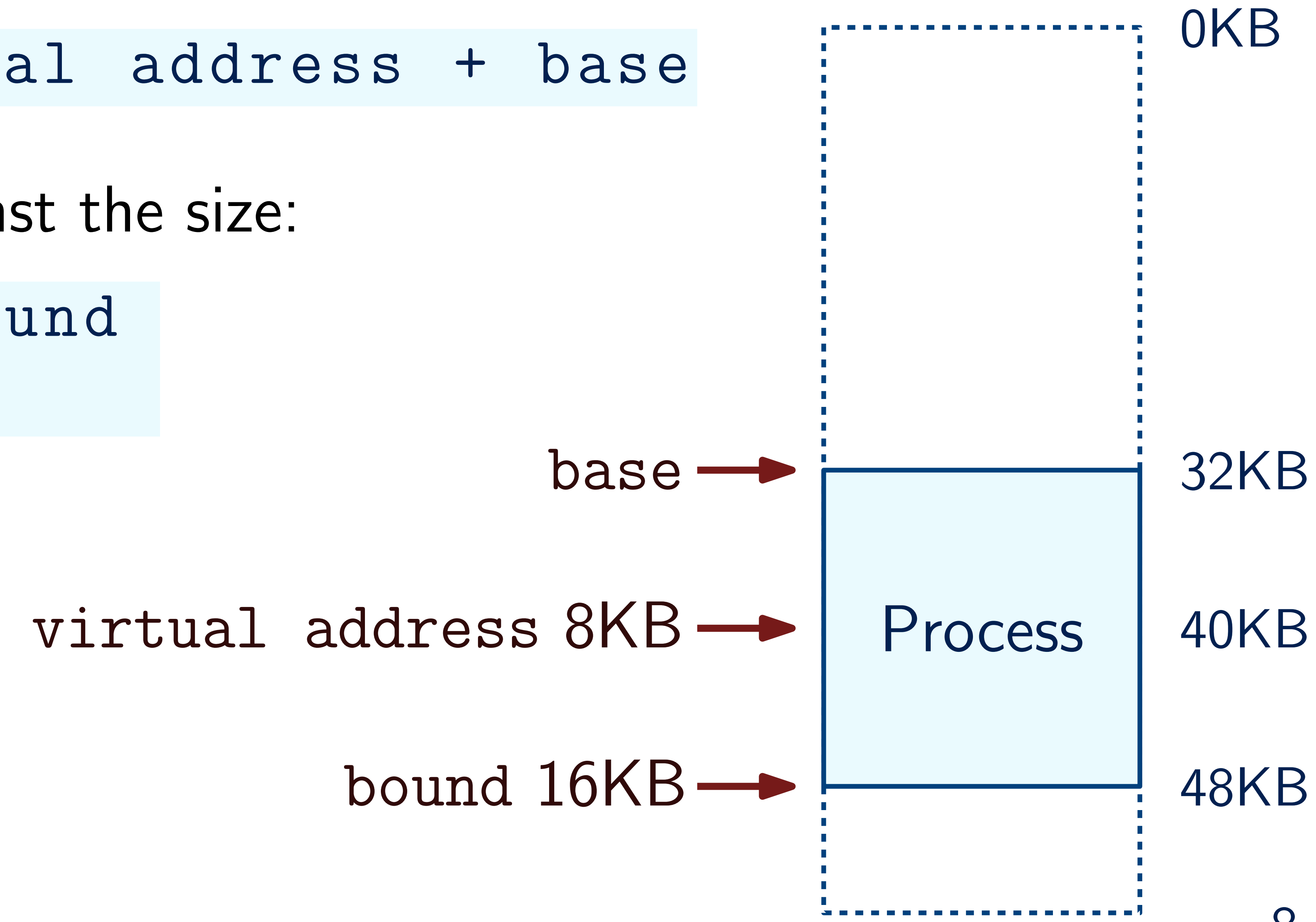
- **Assume:** Each process occupies a contiguous, dedicated part of the memory
- Translating addresses is fairly simple:

```
physical address = virtual address + base
```

- To check validity, just compare against the size:

```
if virtual address >= bound
  fail
```

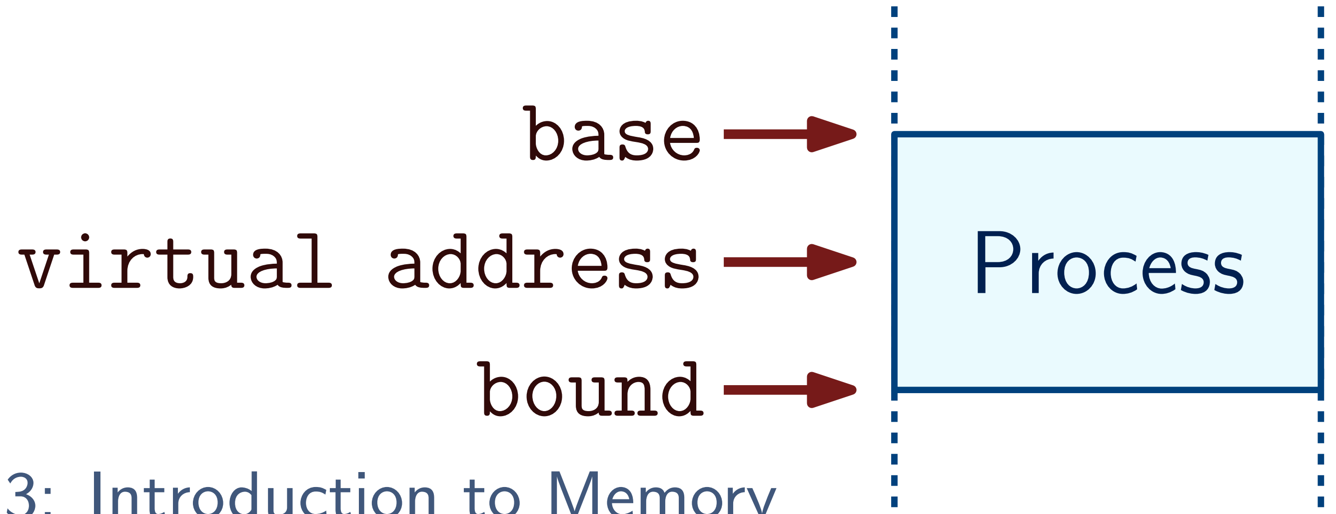
- base and bound set in hardware
  $\Rightarrow$ automatic address translation and protection!

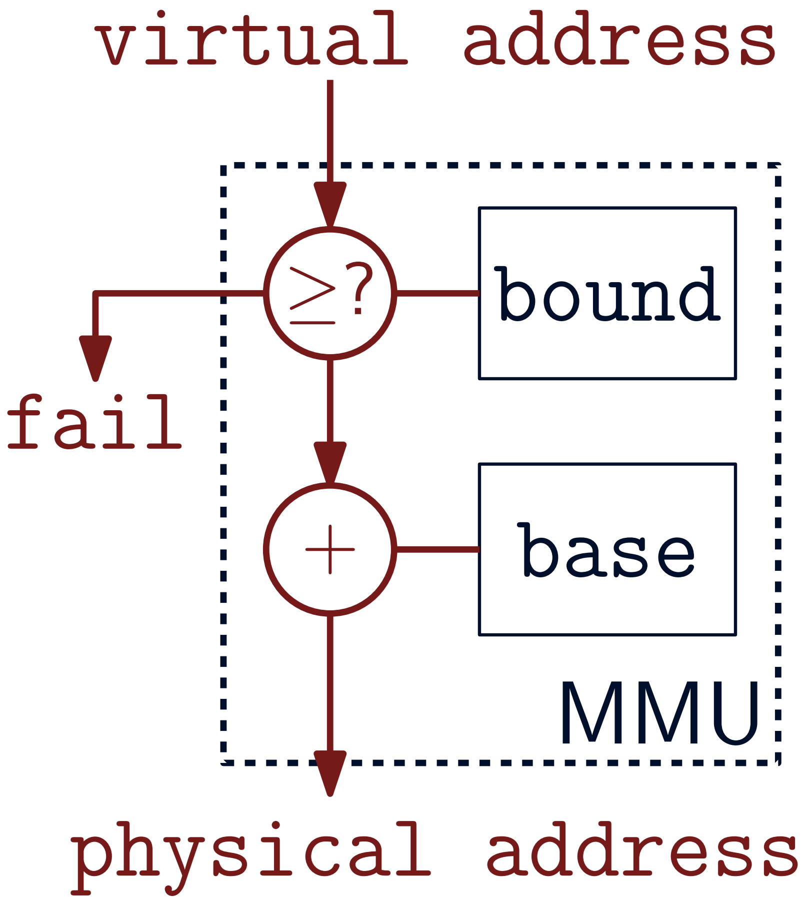


Base and bound: Hardware

- Memory management unit (MMU) on the CPU has `base` and `bound` registers
- When OS loads a program, it sets the registers to the program address space bounds
- When a program requests address, MMU converts virtual address into physical and checks validity

```
if virtual address >= bound
  fail
physical address = virtual address + base
```

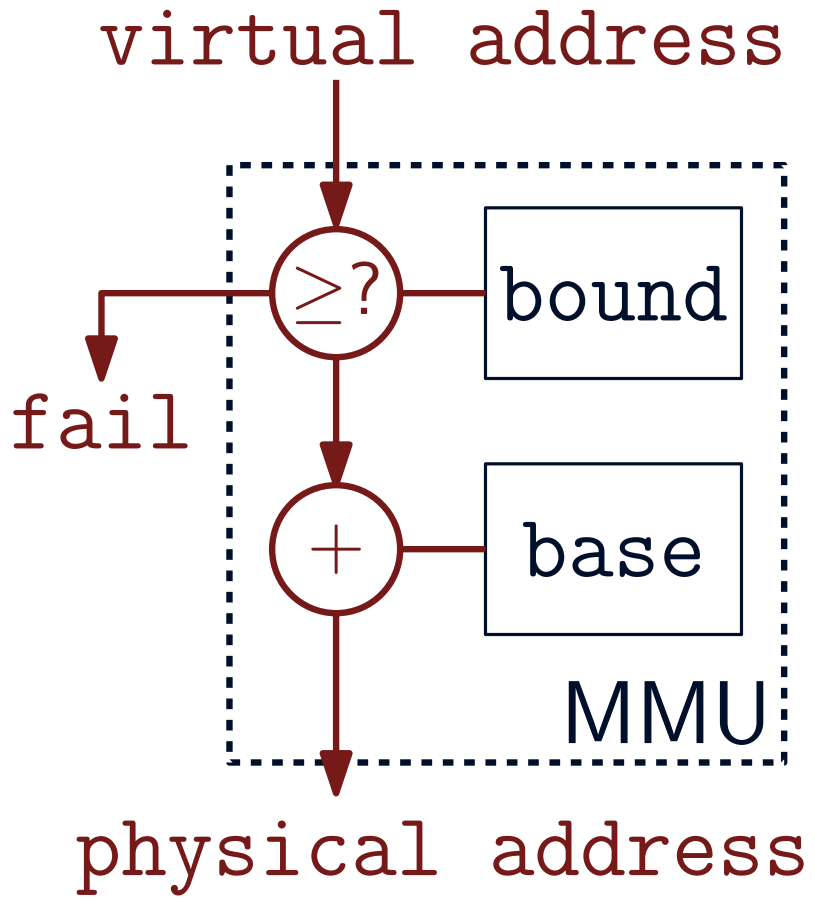




Base and bound: Hardware

- Memory management unit (MMU) on the CPU has `base` and `bound` registers
- When OS loads a program, it sets the registers to the program address space bounds
- When a program requests address, MMU converts virtual address into physical and checks validity

```
if virtual address >= bound
  fail
physical address = virtual address + base
```

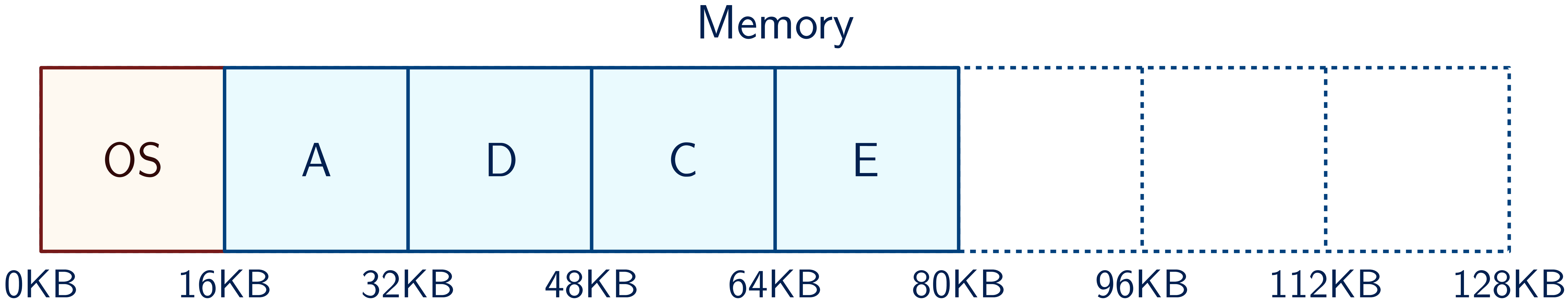


- Upon a context switch, `base` and `bound` need to be stored too
- Easy to relocate program: just update the registers

Simple VM

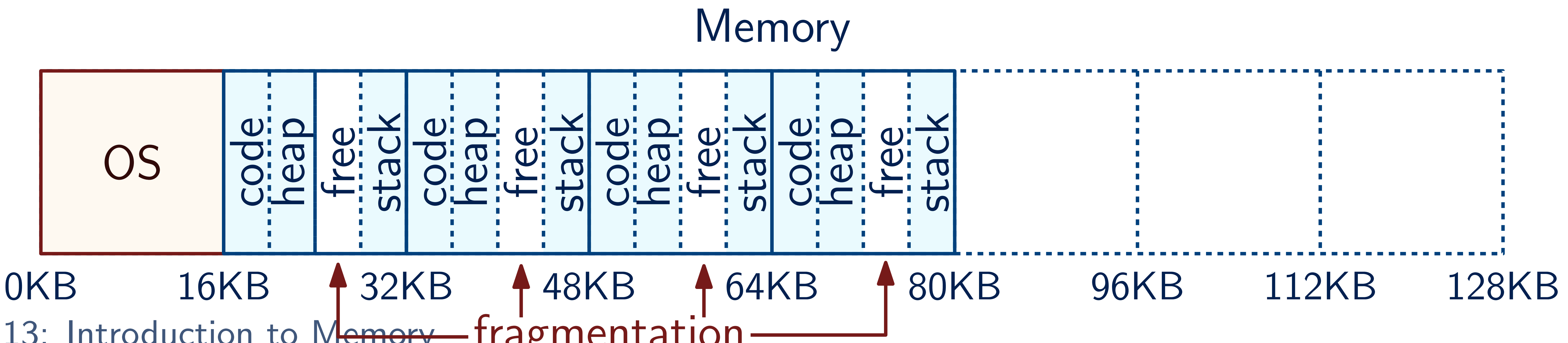
- **Assume:** Each process occupies a same-sized contiguous part of the memory
- OS can keep a **free list**: list of free slots in the memory
  - New process: Take first slot, remove from free list
  - Process over: Add its slot to the free list

Free list: $80 \rightarrow 96 \rightarrow 112$



Simple VM

- **Assume:** Each process occupies a same-sized contiguous part of the memory
- OS can keep a **free list**: list of free slots in the memory
  - New process: Take first slot, remove from free list
  - Process over: Add its slot to the free list
- Issues so far:
  - Processes may have very different memory requirements
  - Lots of memory within process slots is wasted



Summary

- Memory organization within a single process: code, heap, stack
- Using valgrind to detect memory issues
- The goals of memory virtualization
- Base and bound registers for address translation, and protection
- Simple VM based on free list
- **Next time:** Make it more efficient