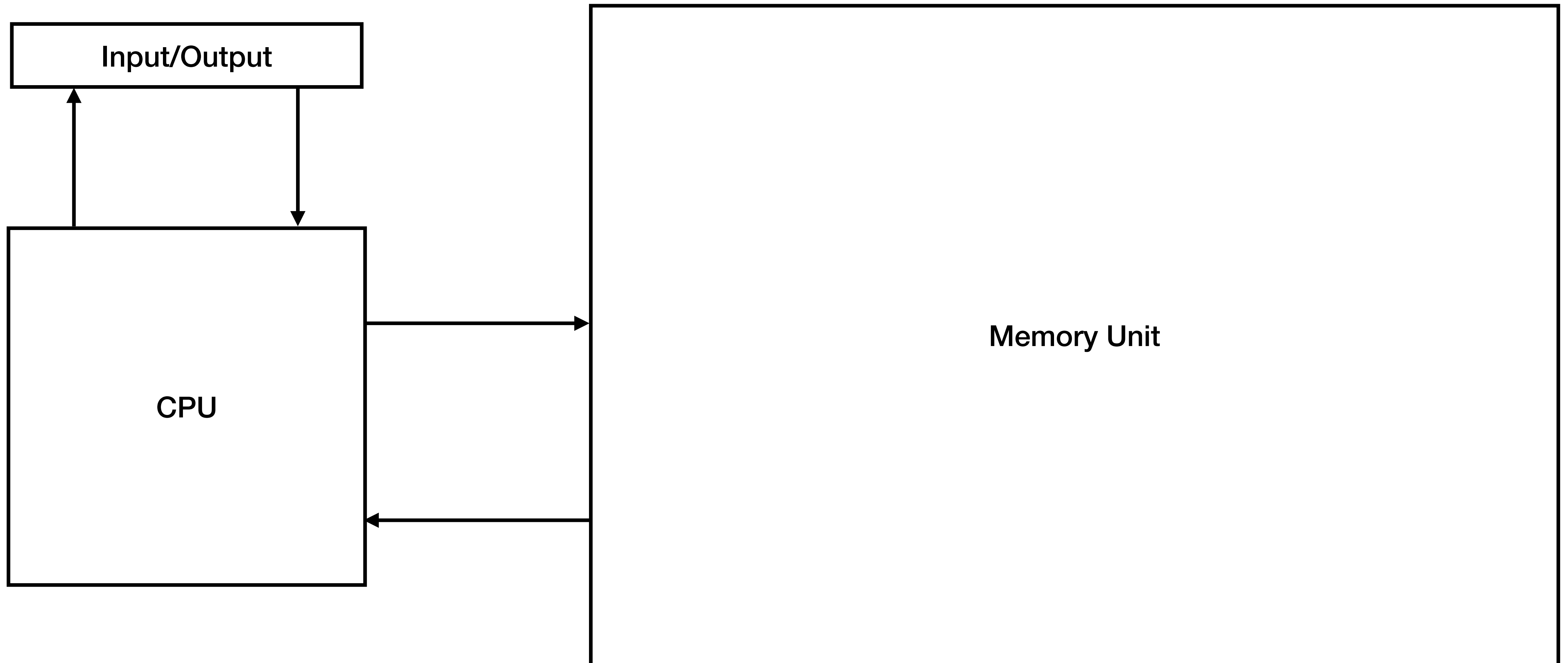


Machine Structure

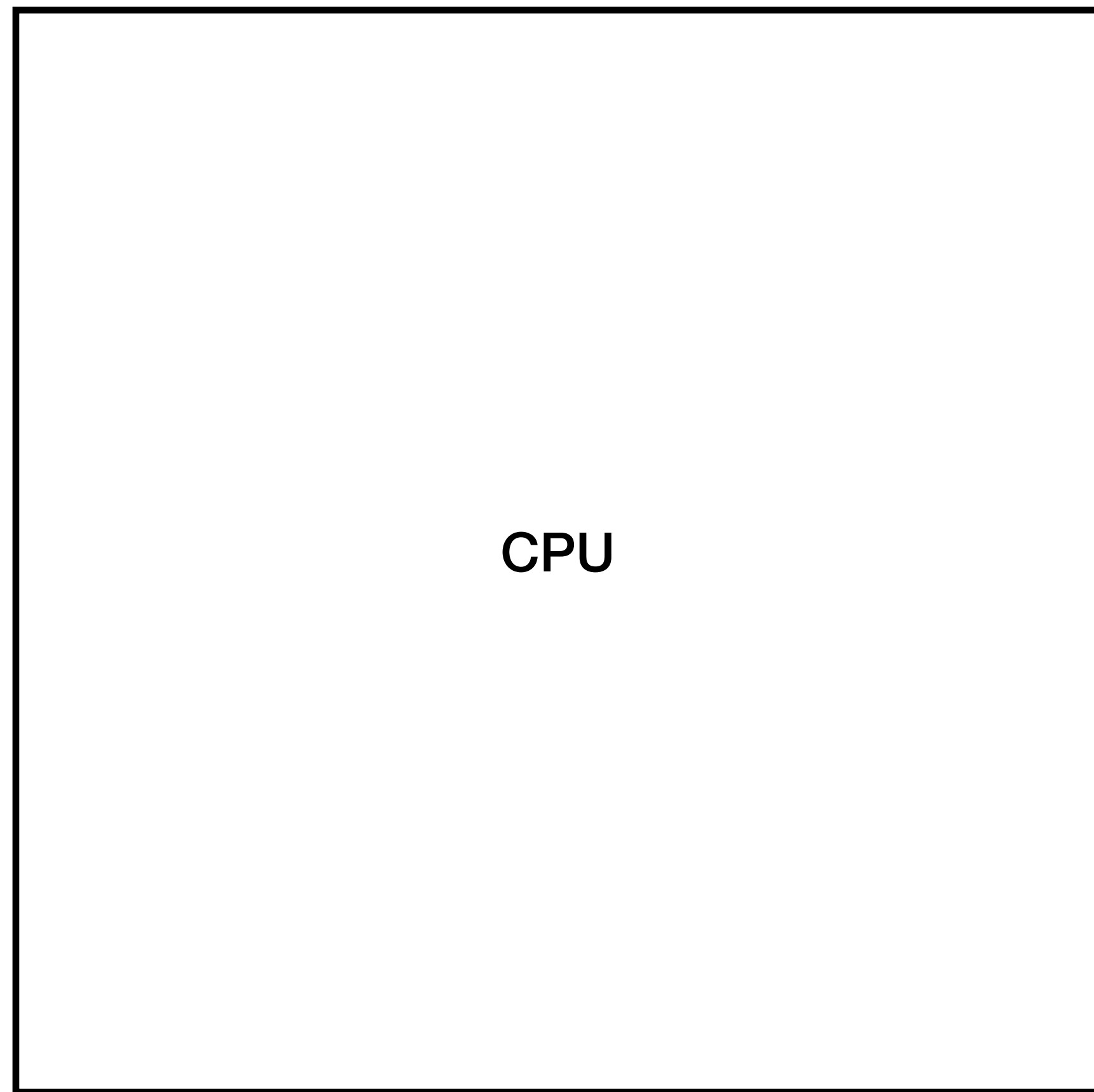
CS143: lecture 17

Konstantinos Ameranis, July 30

A Von Neumann Machine

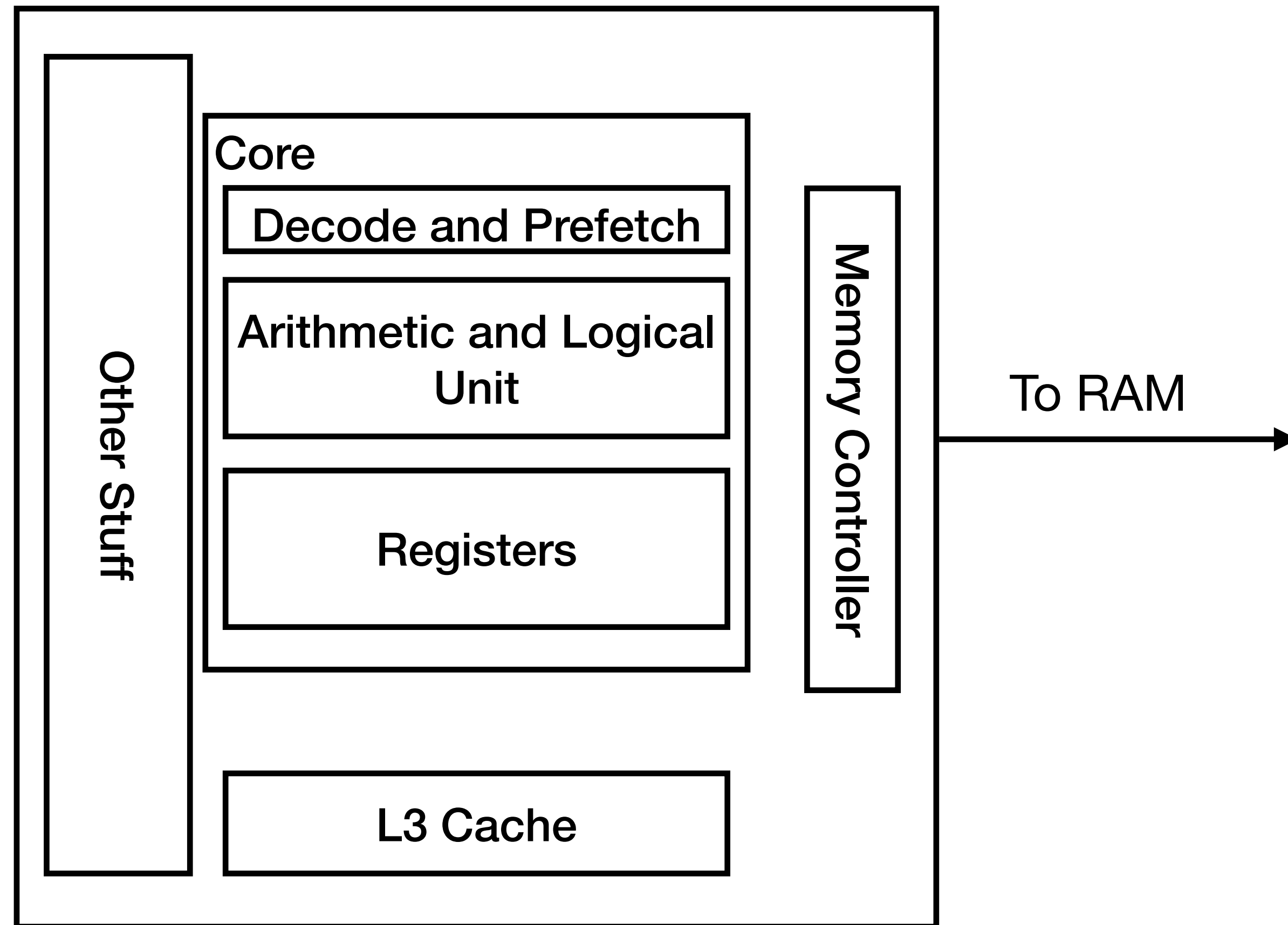


A Von Neumann Machine



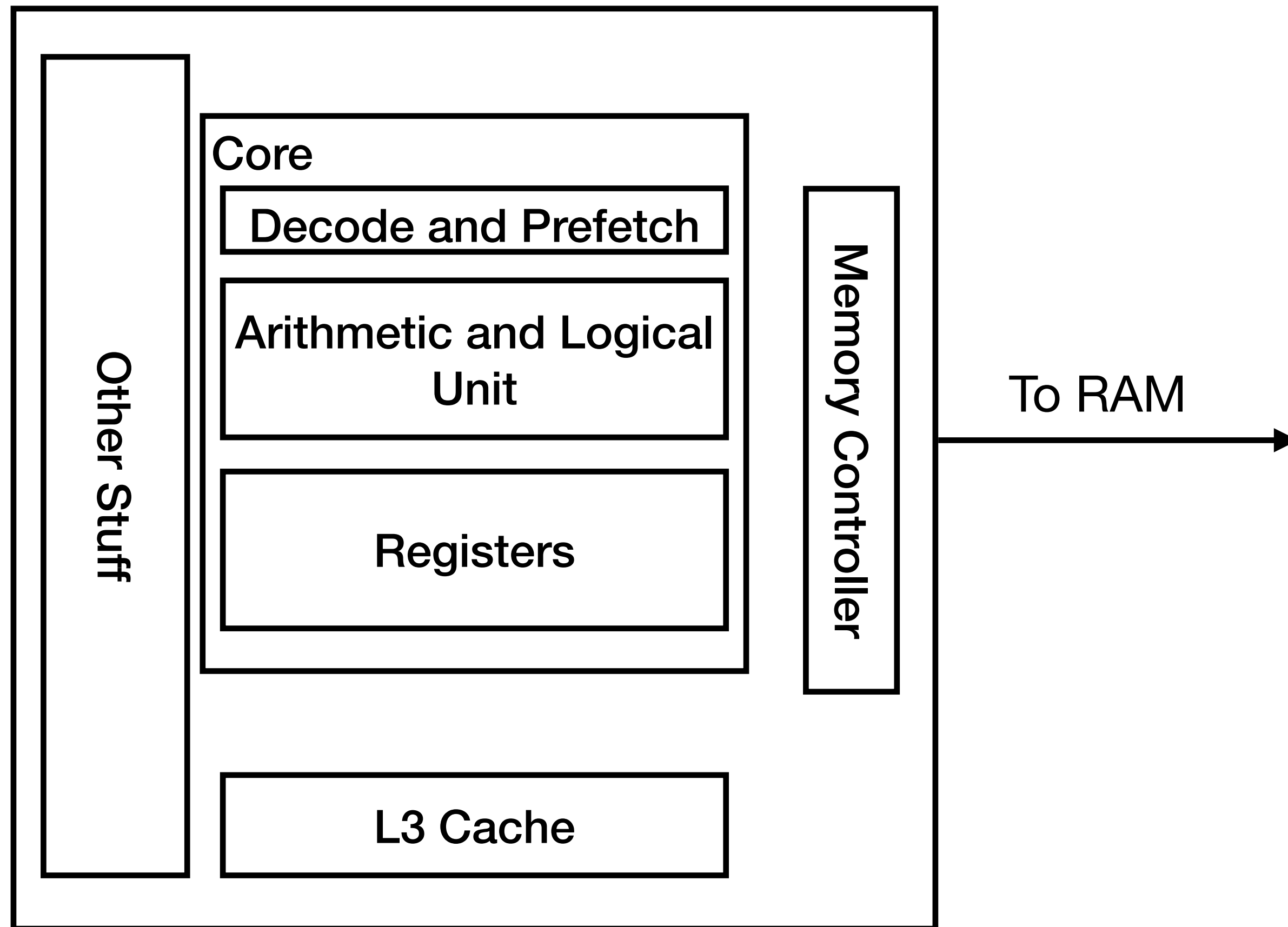
CPU

Review



CPU

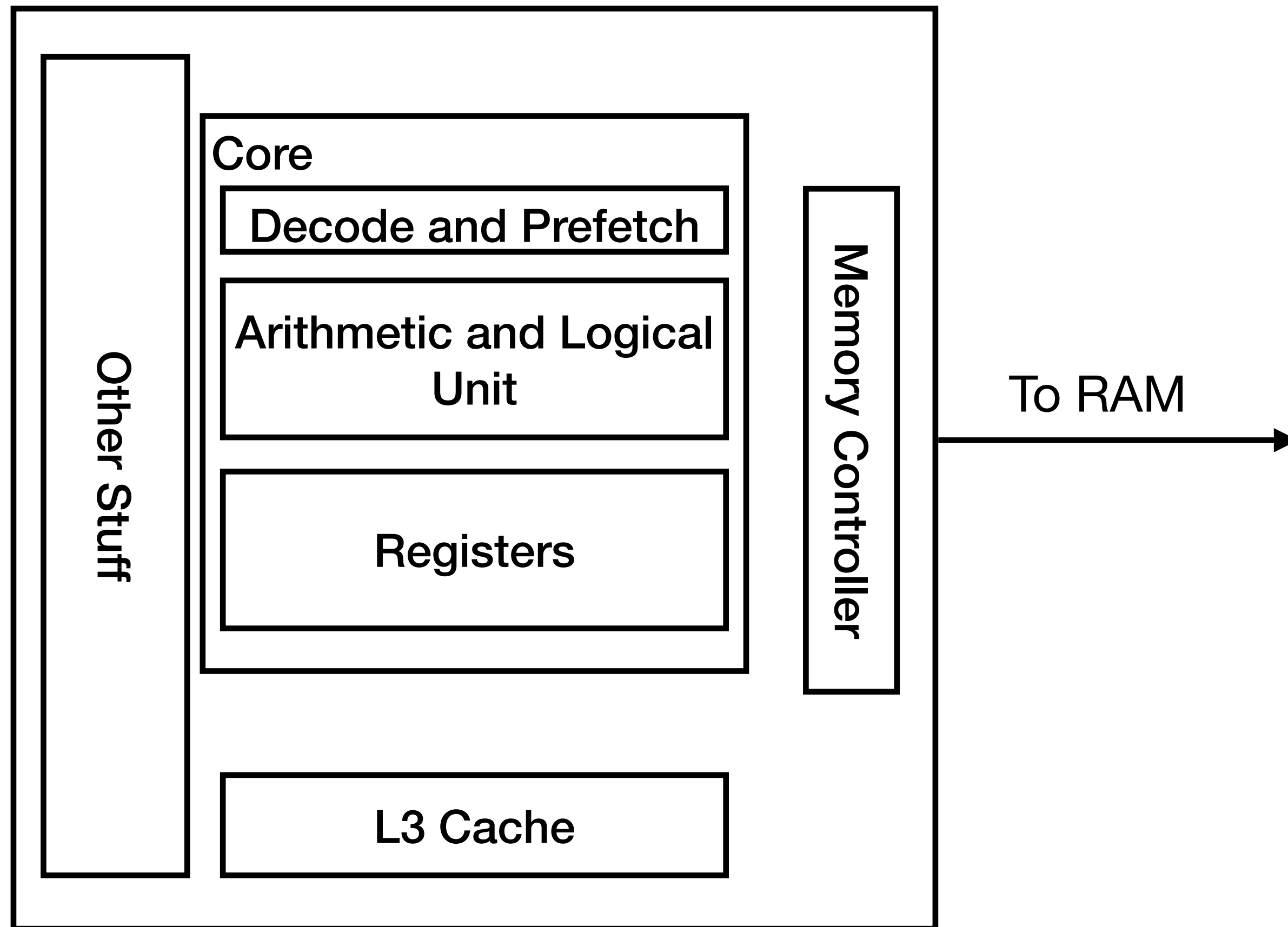
Review



- *Registers*: named locations storing 64-bit values. These registers can hold integers or addresses (pointers).

CPU

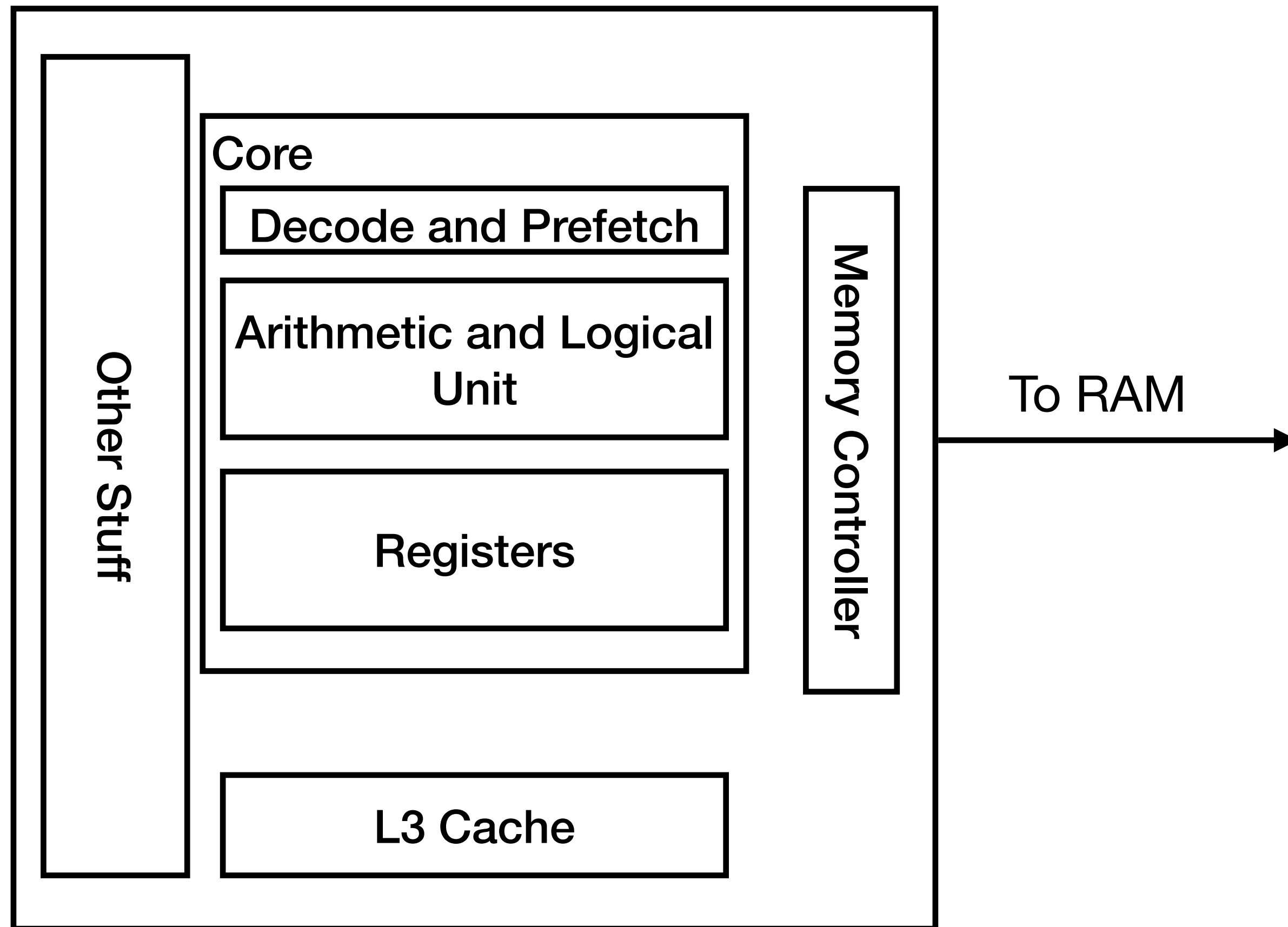
Review



- *Registers*: named locations storing 64-bit values. These registers can hold integers or addresses (pointers).
- Some registers keep track of program states; others hold temporary data, such as local variables

CPU

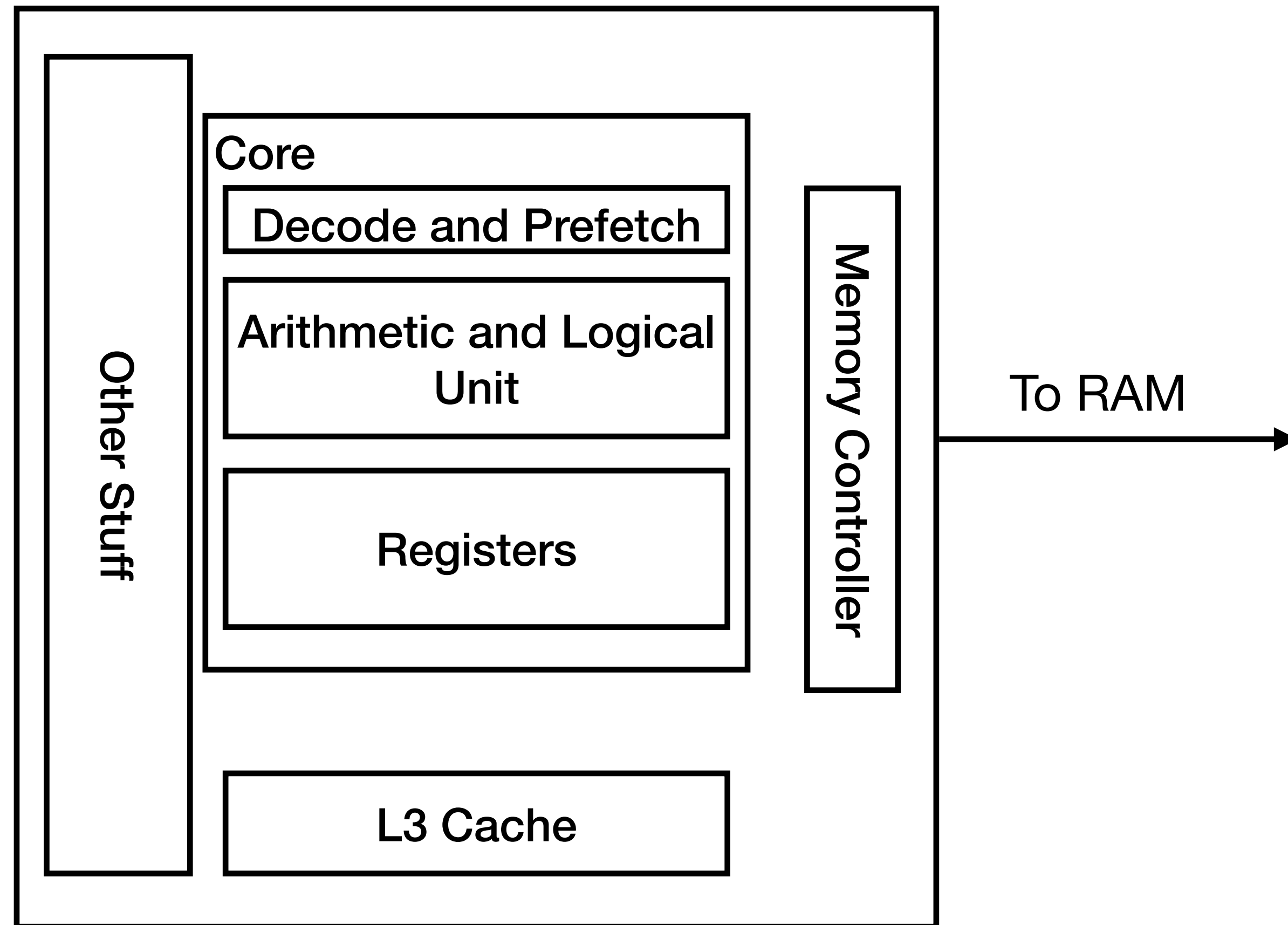
Review



- *Registers*: named locations storing 64-bit values. These registers can hold integers or addresses (pointers).
- Some registers keep track of program states; others hold temporary data, such as local variables
- *ALU*: reads from registers and perform calculations.

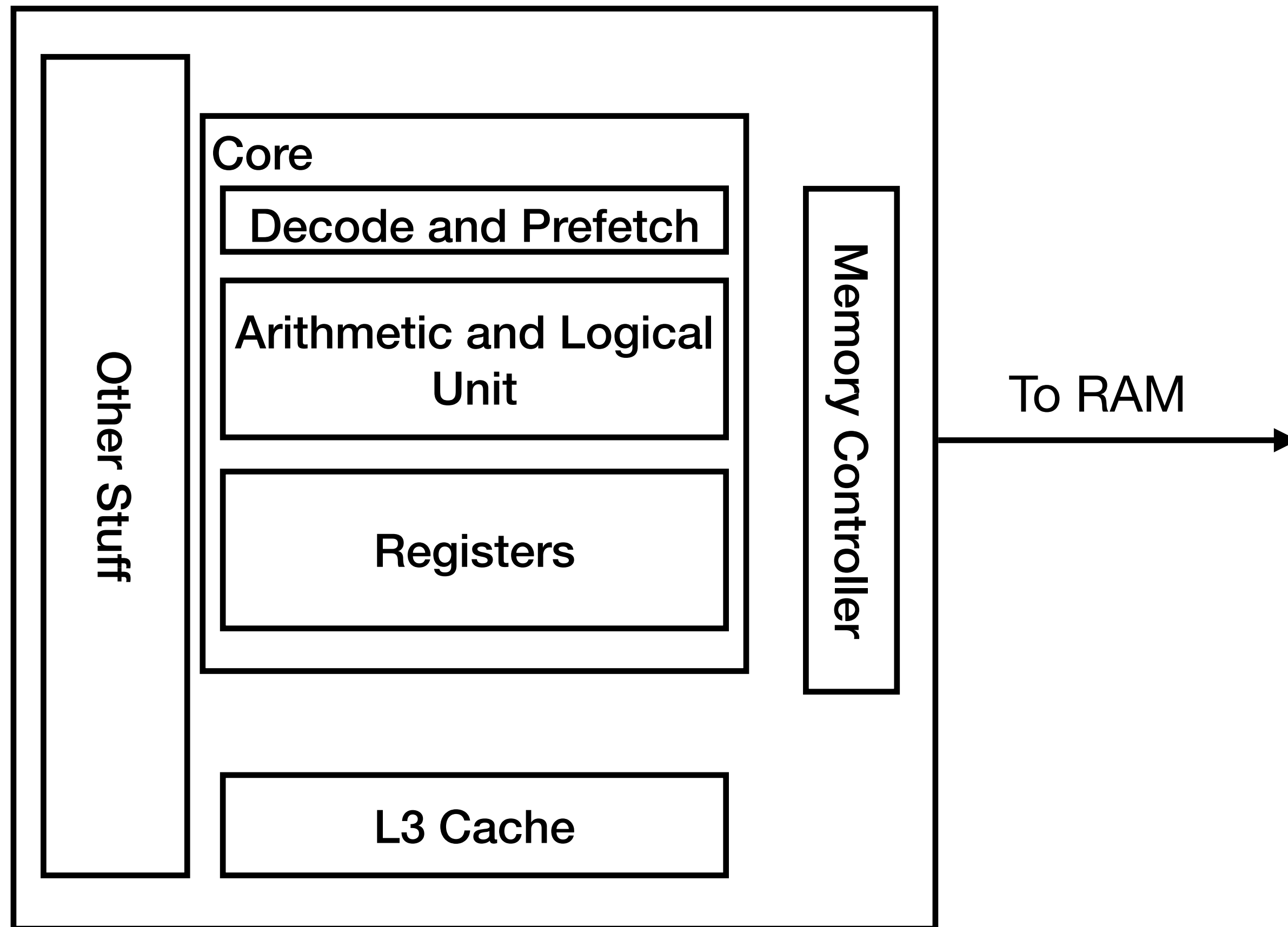
CPU

Review



CPU

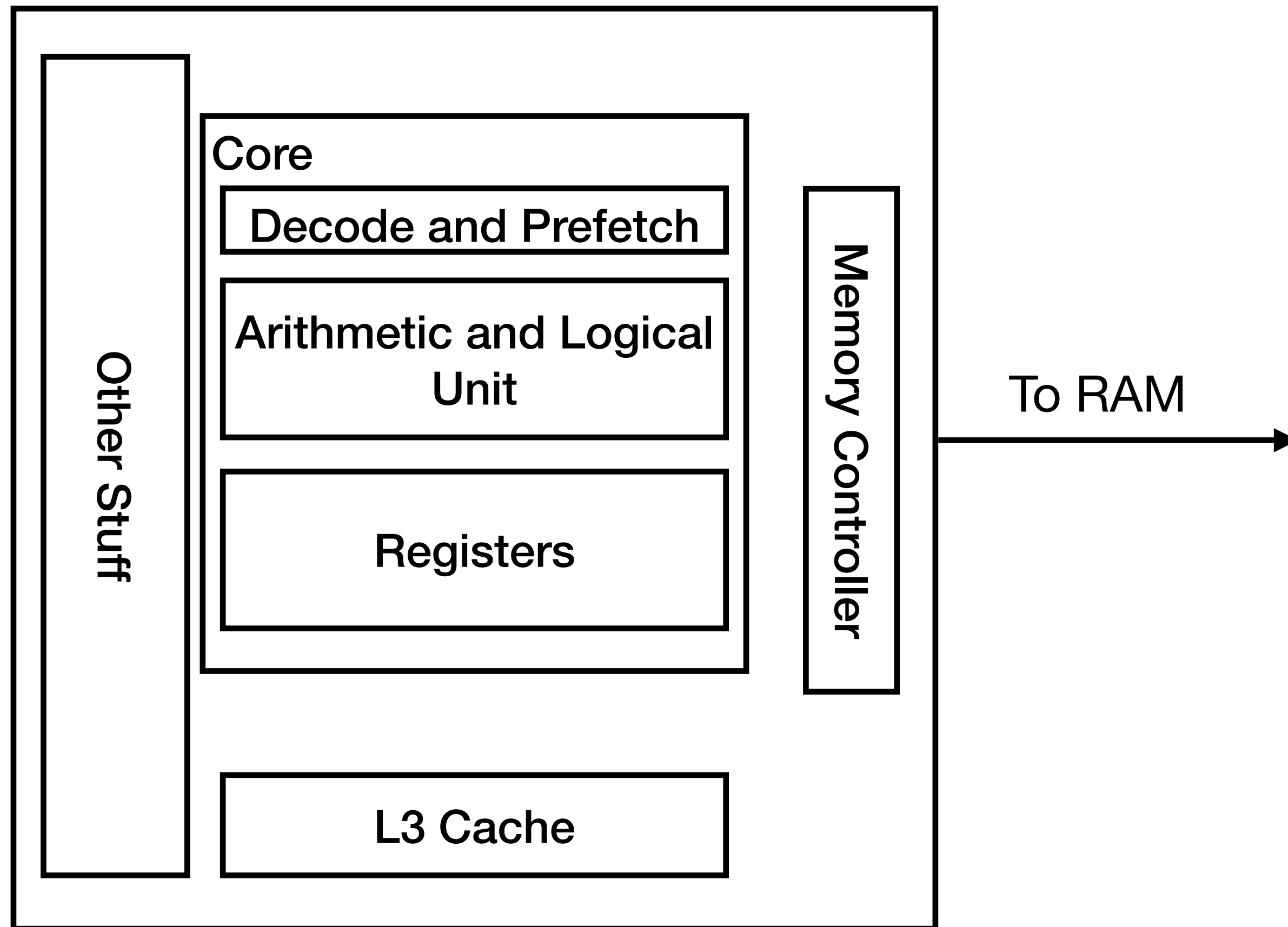
Review



- *Cache*: stores recently read memory to improve performance.

CPU

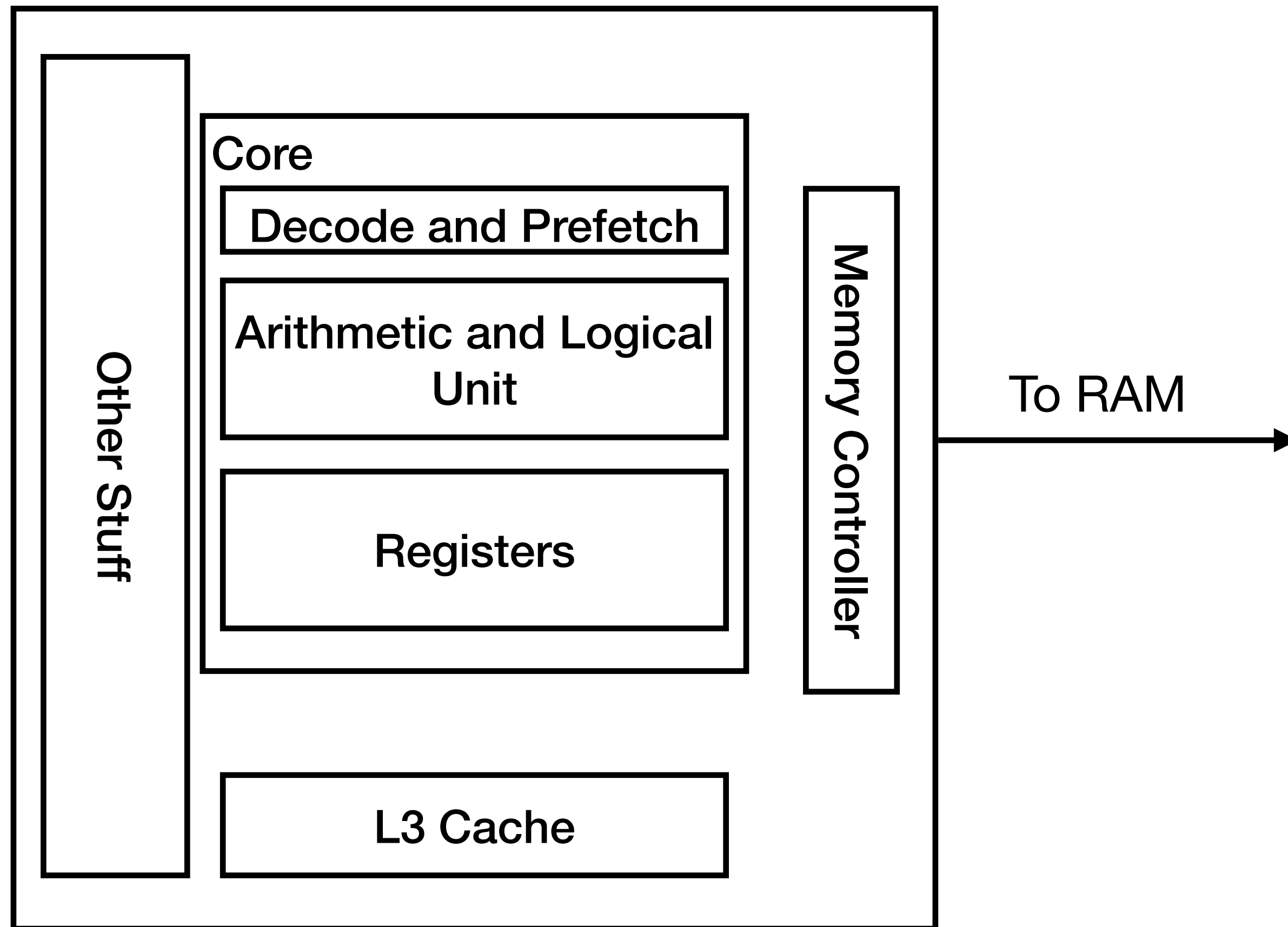
Review



- *Cache*: stores recently read memory to improve performance.
- 144!

CPU

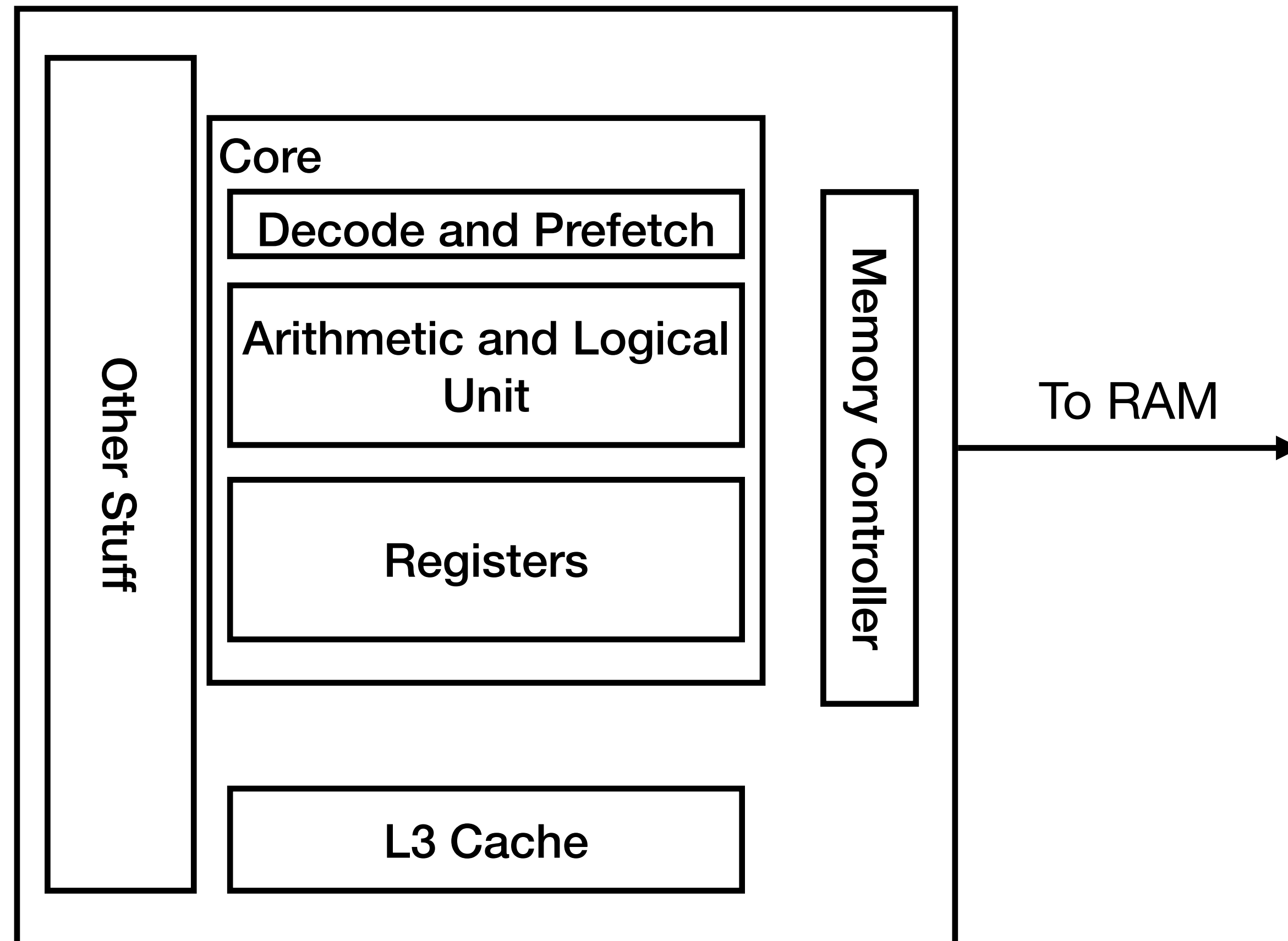
Review



- *Cache*: stores recently read memory to improve performance.
 - 144!
- *Program counter* (pc), or *instruction pointer* (ip), points at some instruction in memory.

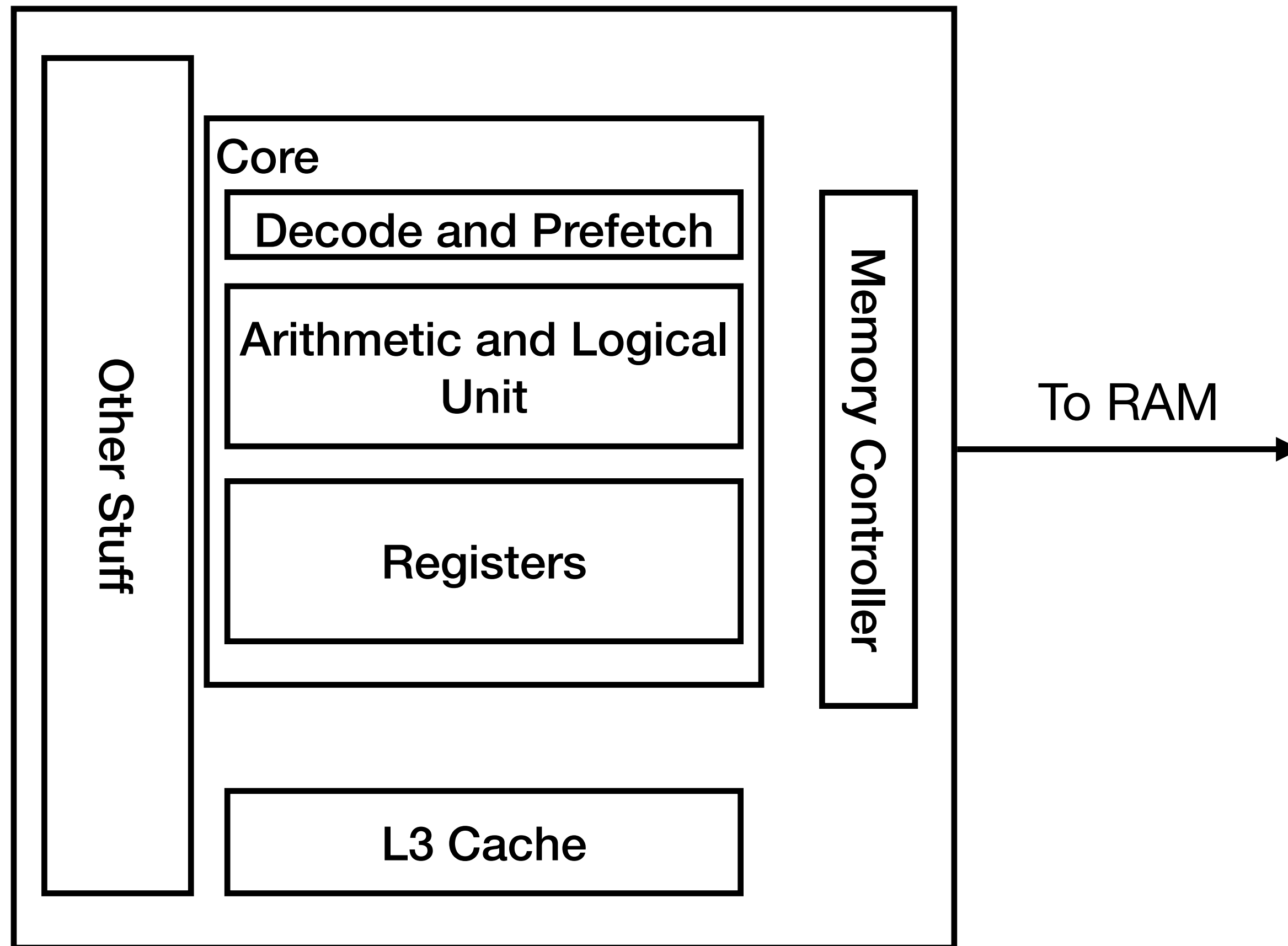
CPU

Review



CPU

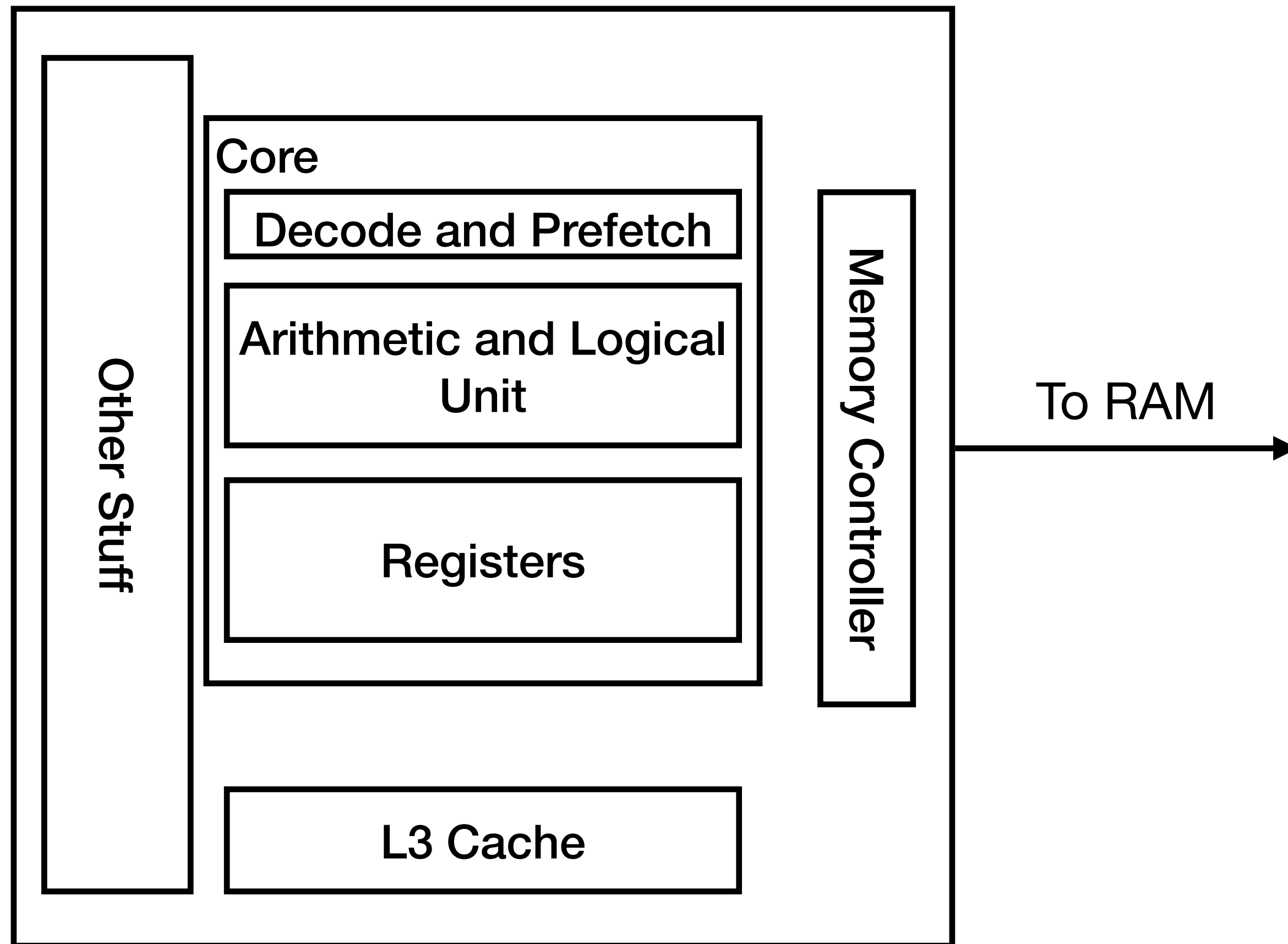
Review



- From the time power is applied to the system, until the power is shut off, CPU performs the same basic tasks repeatedly:

CPU

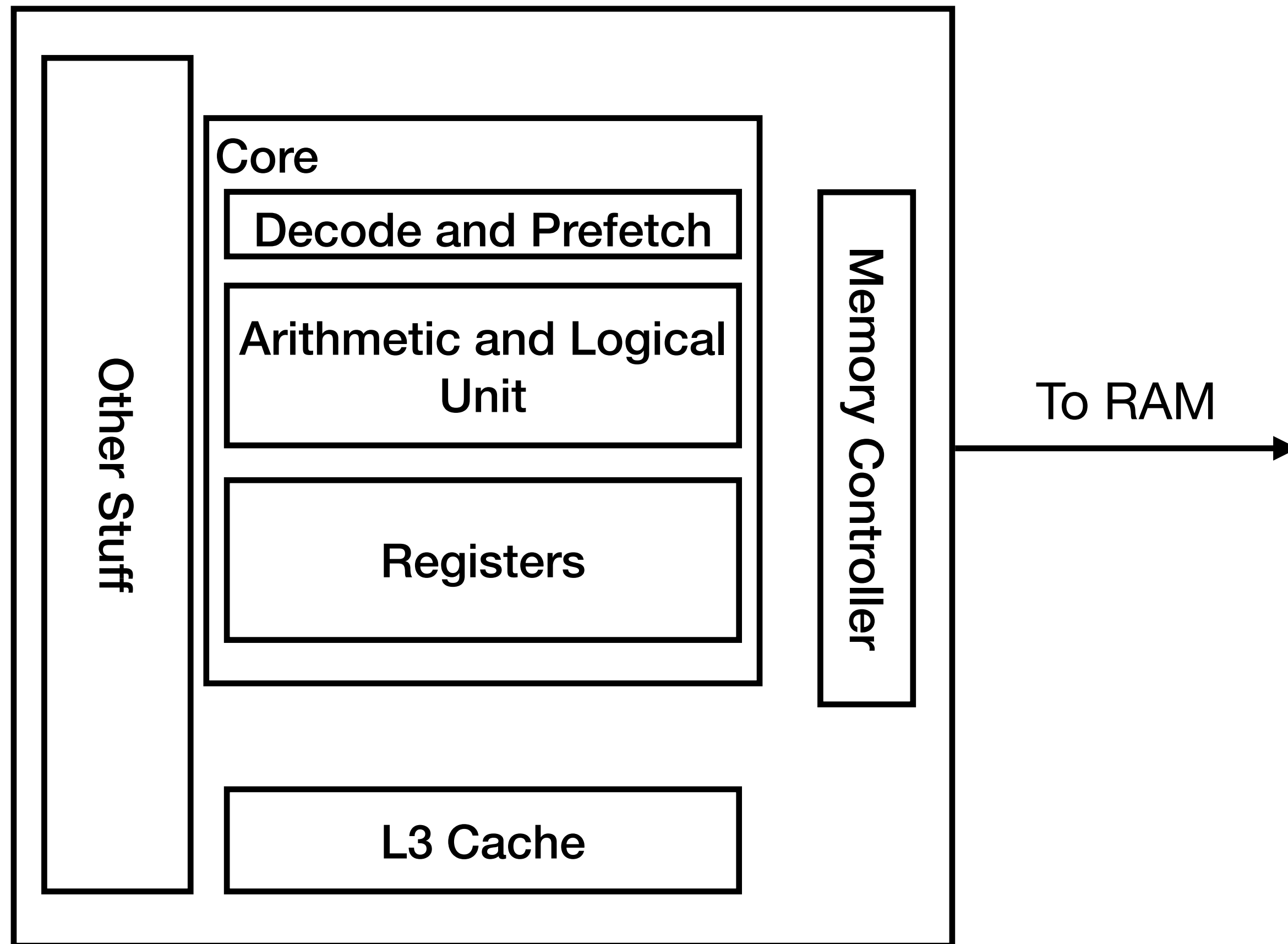
Review



- From the time power is applied to the system, until the power is shut off, CPU performs the same basic tasks repeatedly:
- Reads the instruction pointed by *pc*

CPU

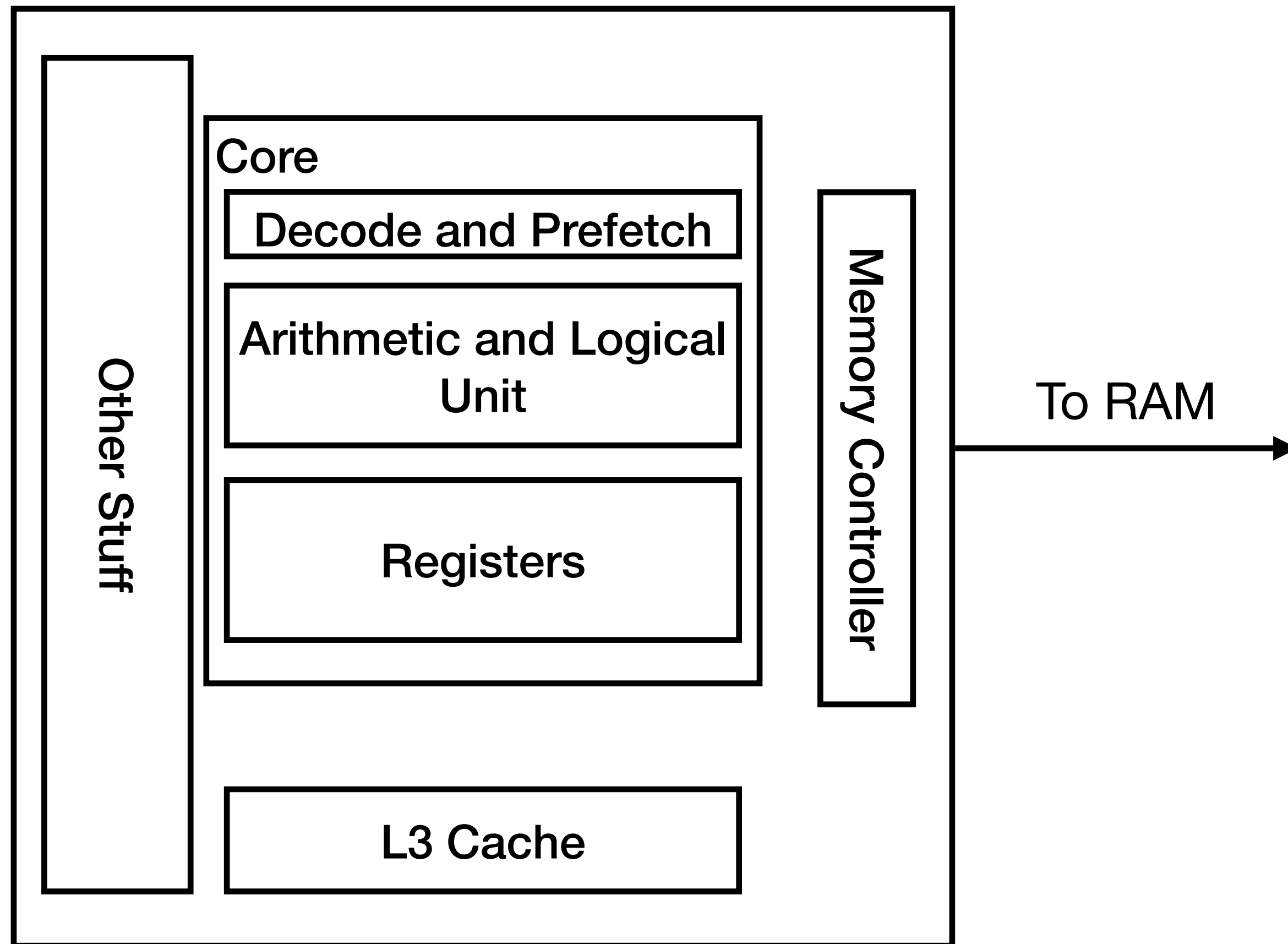
Review



- From the time power is applied to the system, until the power is shut off, CPU performs the same basic tasks repeatedly:
- Reads the instruction pointed by *pc*
- Interprets the bits in the instruction

CPU

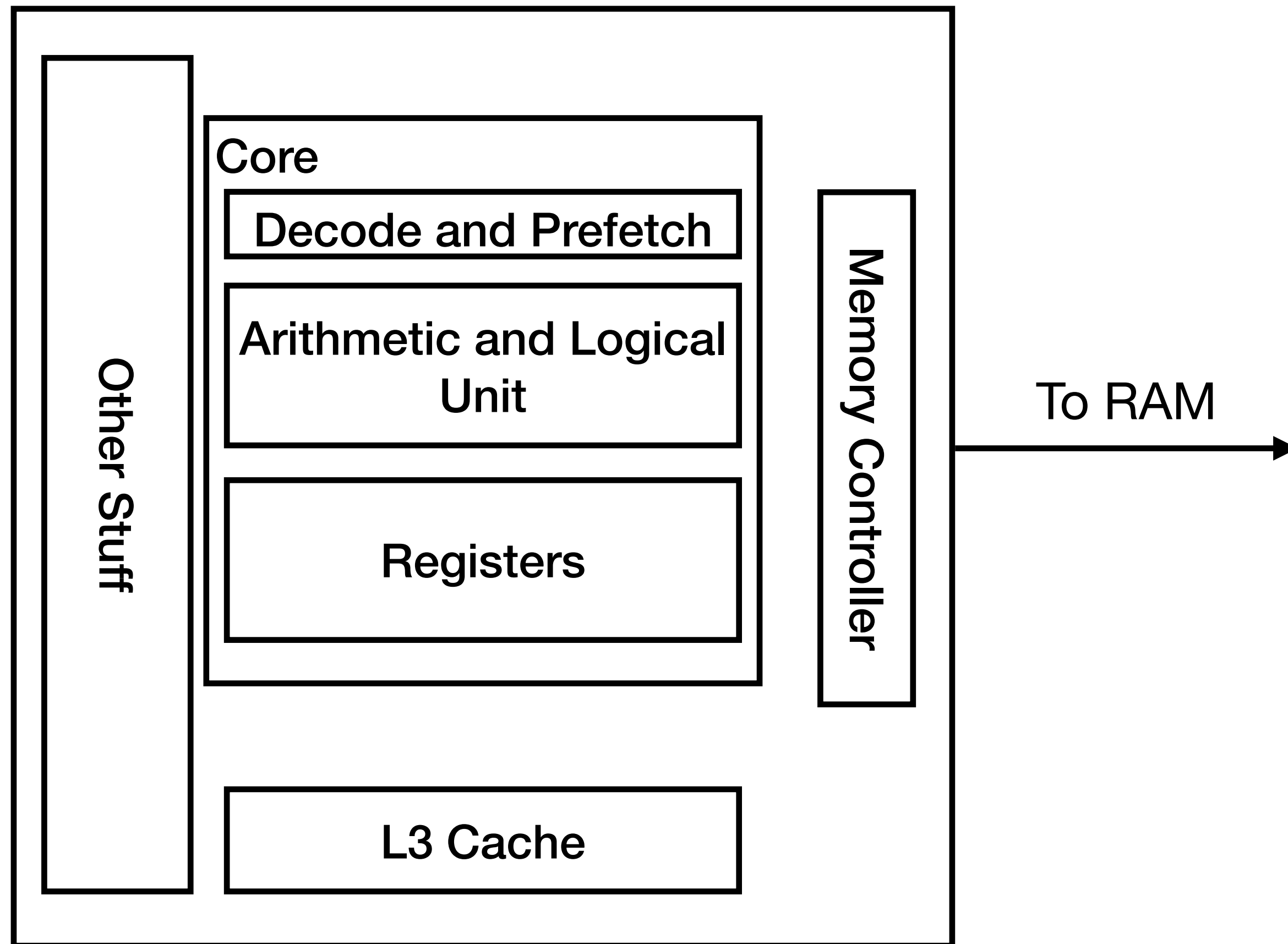
Review



- From the time power is applied to the system, until the power is shut off, CPU performs the same basic tasks repeatedly:
- Reads the instruction pointed by *pc*
- Interprets the bits in the instruction
- Performs some operations as instructed

CPU

Review



- From the time power is applied to the system, until the power is shut off, CPU performs the same basic tasks repeatedly:
- Reads the instruction pointed by *pc*
- Interprets the bits in the instruction
- Performs some operations as instructed
- Updates the *pc* to point to the *next* instruction

Instructions

Review

Instructions

Review

```
int accum = 0;

int sum(int x, int y)
{
    int t = x + y;
    accum += t;
    return t;
}
```

code.c

Instructions

Review

```
int accum = 0;
```

```
clang -O2 -c code.c
```

```
int sum(int x, int y)
{
    int t = x + y;
    accum += t;
    return t;
}
```

```
code.c
```

Instructions

Review

```
int accum = 0;

int sum(int x, int y)
{
    int t = x + y;
    accum += t;
    return t;
}
```

`code.c`

```
clang -O2 -c code.c
```

```
89 f3 01 f0 01 05 00 00 00 00 c3
```

Instructions

Review

```
int accum = 0;

int sum(int x, int y)
{
    int t = x + y;
    accum += t;
    return t;
}
```

`code.c`

```
clang -O2 -c code.c
```

```
89 f3 01 f0 01 05 00 00 00 00 c3
```

- This function is compiled to 11 bytes of instructions

Instructions

Review

```
int accum = 0;
```

```
int sum(int x, int y)
{
    int t = x + y;
    accum += t;
    return t;
}
```

code.c

```
$ objdump -d code.o
```

```
code.o:          file format elf64-x86-64
```

Disassembly of section .text:

```
0000000000000000 <sum>:
```

```
0:      89 f8
```

```
mov     %edi,%eax
```

```
2:      01 f0
```

```
add     %esi,%eax
```

```
4:      01 05 00 00 00 00
```

```
add     %eax,0x0(%rip)
```

```
a:      c3
```

```
retq
```

Objdump displays info about object files. -d "disassembles" machine code.

Instructions

Review

```
int accum = 0;

int sum(int x, int y)
{
    int t = x + y;
    accum += t;
    return t;
}
```

code.c

```
$ objdump -d code.o

code.o:          file format elf64-x86-64
```

Disassembly of section .text:

```
0000000000000000 <sum>:
0:      89 f8                mov     %edi,%eax
2:      01 f0                add     %esi,%eax
4:      01 05 00 00 00 00    add     %eax,0x0(%rip)
a:      c3                  retq
```

%xxx are
register names

Instructions

Review

```
int accum = 0;
```

```
int sum(int x, int y)
{
    int t = x + y;
    accum += t;
    return t;
}
```

code.c

```
$ objdump -d code.o
```

```
code.o:          file format elf64-x86-64
```

Disassembly of section .text:

```
0000000000000000 <sum>:
0:      89 f8
2:      01 f0
4:      01 05 00 00 00 00
a:      c3
```

By convention, %edi stores the first arg, %esi stores the second.

```
mov     %edi,%eax
add     %esi,%eax
add     %eax,0x0(%rip)
retq
```

Instructions

Review

```
int accum = 0;

int sum(int x, int y)
{
    int t = x + y;
    accum += t;
    return t;
}
```

code.c

```
$ objdump -d code.o
```

```
code.o:          file format elf64-x86-64
```

Disassembly of section .text:

```
0000000000000000 <sum>:
 0:      89 f8
 2:      01 f0
 4:      01 05 00 00 00 00
 a:      c3
```

%eax stores the return value.

```
mov    %edi,%eax
add    %esi,%eax
add    %eax,0x0(%rip)
retq
```

Instructions

Review

```
int accum = 0;

int sum(int x, int y)
{
    int t = x + y;
    accum += t;
    return t;
}
```

code.c

```
$ objdump -d code.o
```

```
code.o:          file format elf64-x86-64
```

Disassembly of section .text:

```
0000000000000000 <sum>:
```

0:	move the first argument	mov	%edi,%eax
2:	(edi) to the return	add	%esi,%eax
4:	value (eax)	add	%eax,0x0(%rip)
a:	c3	retq	

Instructions

Review

```
int accum = 0;

int sum(int x, int y)
{
    int t = x + y;
    accum += t;
    return t;
}
```

code.c

```
$ objdump -d code.o
```

```
code.o:          file format elf64-x86-64
```

Disassembly of section .text:

```
0000000000000000 <sum>:
```

0:

2:

4:

a:

add the second argument
(esi) to the return
value (eax)

c3

mov

add

add

retq

%edi,%eax

%esi,%eax

%eax,0x0(%rip)

Instructions

Review

```
int accum = 0;

int sum(int x, int y)
{
    int t = x + y;
    accum += t;
    return t;
}
```

code.c

```
$ objdump -d code.o
```

```
code.o:          file format elf64-x86-64
```

Disassembly of section .text:

```
0000000000000000 <sum>:
```

0:

2:

4:

a:

add eax to a location.

c3

mov

add

add

retq

%edi,%eax

%esi,%eax

%eax,0x0(%rip)

Instructions

Review

```
int accum = 0;

int sum(int x, int y)
{
    int t = x + y;
    accum += t;
    return t;
}
```

code.c

```
$ objdump -d code.o
```

```
code.o:          file format elf64-x86-64
```

Disassembly of section .text:

```
0000000000000000 <sum>:
```

0:		mov	%edi,%eax
2:	return	add	%esi,%eax
4:		add	%eax,0x0(%rip)
a:	c3	retq	

Instructions

Review

```
int accum = 0;

int sum(int x, int y)
{
    int t = x + y;
    accum += t;
    return t;
}
```

code.c

Instructions

Review

```
int accum = 0;
```

```
int sum(int x, int y)
{
    int t = x + y;
    accum += t;
    return t;
}
```

code.c

- Load: copy some bytes from memory to a register

Instructions

Review

```
int accum = 0;

int sum(int x, int y)
{
    int t = x + y;
    accum += t;
    return t;
}
```

`code.c`

- Load: copy some bytes from memory to a register
- Store: copy some bytes from a register to memory

Instructions

Review

```
int accum = 0;

int sum(int x, int y)
{
    int t = x + y;
    accum += t;
    return t;
}
```

`code.c`

- Load: copy some bytes from memory to a register
- Store: copy some bytes from a register to memory
- Update: copy the contents of two registers to the ALU, which does some calculation and stores the result in a register

Instructions

Review

```
int accum = 0;

int sum(int x, int y)
{
    int t = x + y;
    accum += t;
    return t;
}
```

`code.c`

- Load: copy some bytes from memory to a register
- Store: copy some bytes from a register to memory
- Update: copy the contents of two registers to the ALU, which does some calculation and stores the result in a register
- I/O operations: read/write from an I/O device into a register

Instructions

Review

```
int accum = 0;

int sum(int x, int y)
{
    int t = x + y;
    accum += t;
    return t;
}
```

code.c

- Load: copy some bytes from memory to a register
- Store: copy some bytes from a register to memory
- Update: copy the contents of two registers to the ALU, which does some calculation and stores the result in a register
- I/O operations: read/write from an I/O device into a register
- Jump: Set the *pc* to be some arbitrary value

Instructions

Review

```
int accum = 0;

int sum(int x, int y)
{
    int t = x + y;
    accum += t;
    return t;
}
```

code.c

```
int sum(int x, int y);

int main(void)
{
    return sum(1, 3);
}
```

main.c

Instructions

Review

```
int accum = 0;

int sum(int x, int y)
{
    int t = x + y;
    accum += t;
    return t;
}
```

`code.c`

```
int sum(int x, int y);

int main(void)
{
    return sum(1, 3);
}
```

`main.c`

```
clang -O2 -o prog code.o main.c
```

Instructions

Review

0000000000401110 <sum>:

401110:89 f8

401112:01 f0

401114:01 05 12 2f 00 00

40111a:c3

40111b:0f 1f 44 00 00

mov %edi,%eax

add %esi,%eax

add %eax,0x2f12(%rip) # 40402c <accum>

retq

nopl 0x0(%rax,%rax,1)

0000000000401120 <main>:

401120:bf 01 00 00 00

401125:be 03 00 00 00

40112a:e9 e1 ff ff ff

40112f:90

mov \$0x1,%edi

mov \$0x3,%esi

jmpq 401110 <sum>

nop

mov 1 to edi

Instructions

Review

0000000000401110 <sum>:

401110:89 f8

401112:01 f0

401114:01 05 12 2f 00 00

40111a:c3

40111b:0f 1f 44 00 00

mov %edi,%eax

add %esi,%eax

add %eax,0x2f12(%rip) # 40402c <accum>

retq

nopl 0x0(%rax,%rax,1)

0000000000401120 <main>:

401120:bf 01 00 00 00

401125:be 03 00 00 00

40112a:e9 e1 ff ff ff

40112f:90

mov \$0x1,%edi

mov \$0x3,%esi

jmpq 401110 <sum>

nop

mov 3 to esi

Instructions

Review

0000000000401110 <sum>:

401110: 89 f8

401112: 01 f0

401114: 01 05 12 2f 00 00

40111a: c3

40111b: 0f 1f 44 00 00

mov %edi,%eax

add %esi,%eax

add %eax,0x2f12(%rip) # 40402c <accum>

retq

nopl 0x0(%rax,%rax,1)

0000000000401120 <main>:

401120: bf 01 00 00 00

401125: be 03 00 00 00

40112a: e9 e1 ff ff ff

40112f: 90

mov \$0x1,%edi

mov \$0x3,%esi

jmpq 401110 <sum>

nop

jump to location 401110

Instructions

Review

0000000000401110 <sum>:

401110:89 f8

401112:01 f0

401114:01 05 12 2f 00 00

40111a:c3

40111b:0f 1f 44 00 00

0000000000401120 <main>:

401120:bf 01 00 00 00

401125:be 03 00 00 00

40112a:e9 e1 ff ff ff

40112f:90

mov %edi,%eax

add %esi,%eax

add %eax,0x2f12(%rip)

retq

nopl 0x0(%rax,%rax,1)

mov \$0x1,%edi

mov \$0x3,%esi

jmpq 401110 <sum>

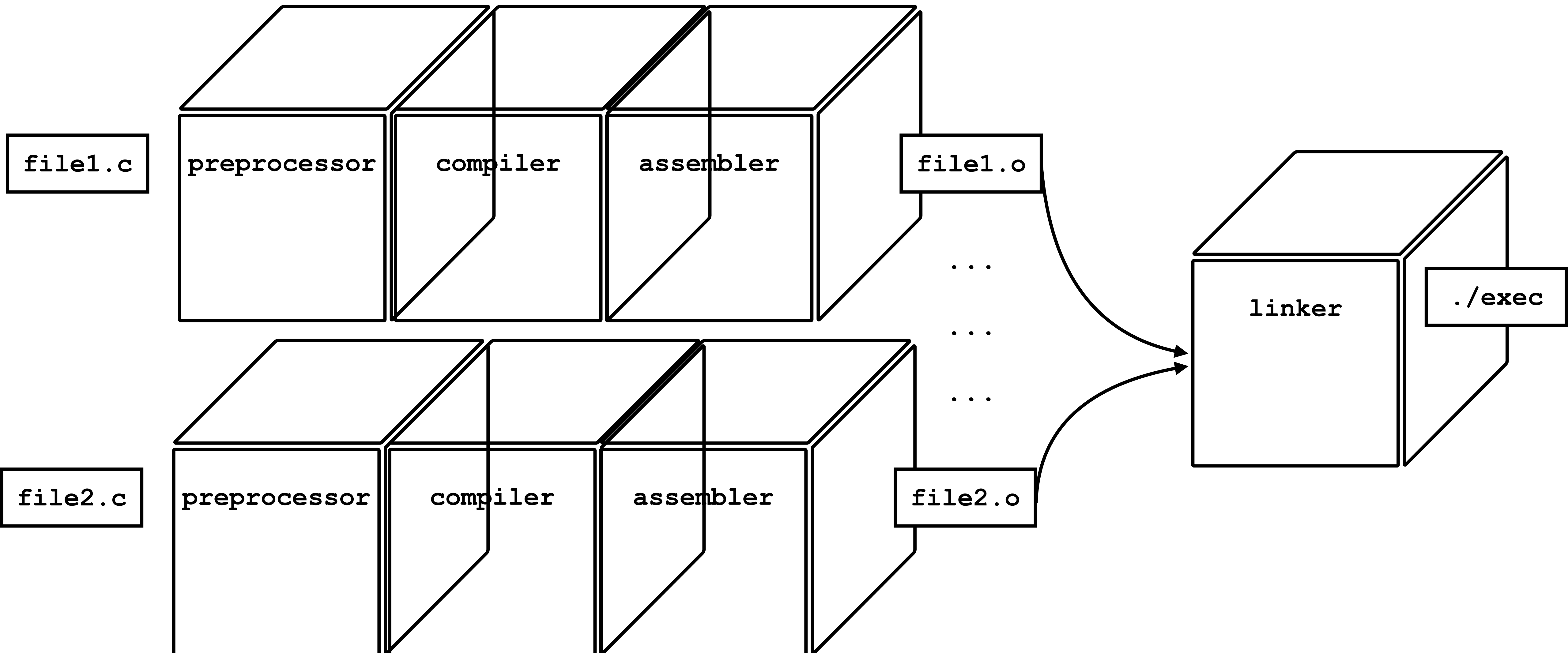
nop

40402c <accum>

The global variable has been assigned a location 40402c

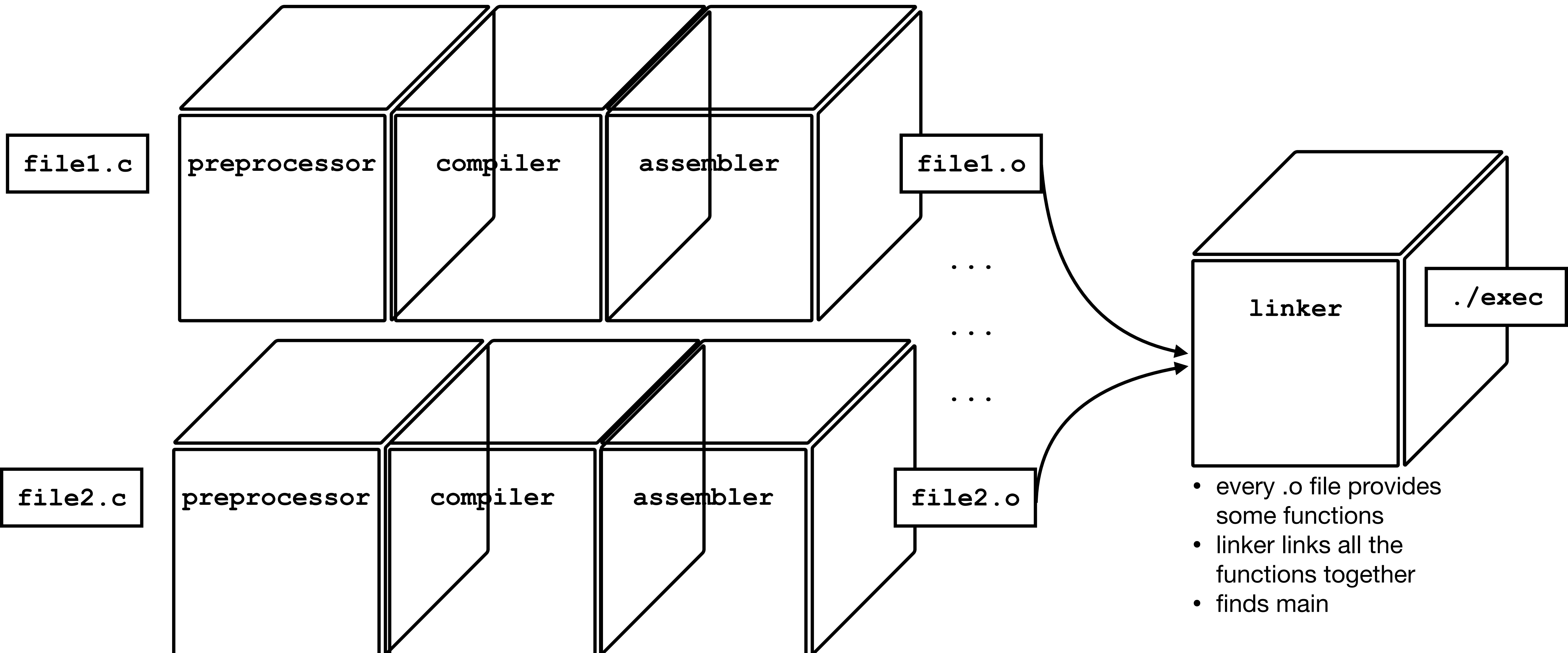
Separate Compilation

Review



Separate Compilation

Review



Instructions

Review

Instructions

Review

- C is compiled to a low-level language called *assembly* language.

Instructions

Review

- C is compiled to a low-level language called *assembly* language.
 - Assembly language expresses a sequence of CPU *instructions*.

Instructions

Review

- C is compiled to a low-level language called *assembly* language.
 - Assembly language expresses a sequence of CPU *instructions*.
- The *assembly* language is *assembled* by an *assembler* to machine code (byte sequences).

Instructions

Review

- C is compiled to a low-level language called *assembly* language.
 - Assembly language expresses a sequence of CPU *instructions*.
- The *assembly* language is *assembled* by an *assembler* to machine code (byte sequences).
- *Disassembler* does the opposite.

Instructions

Review

- C is compiled to a low-level language called *assembly* language.
 - Assembly language expresses a sequence of CPU *instructions*.
- The *assembly* language is *assembled* by an *assembler* to machine code (byte sequences).
- *Disassembler* does the opposite.
- CPU executes instructions in a loop from power-on to power-off

Instructions

Review

- C is compiled to a low-level language called *assembly* language.
 - Assembly language expresses a sequence of CPU *instructions*.
- The *assembly* language is *assembled* by an *assembler* to machine code (byte sequences).
- *Disassembler* does the opposite.
- CPU executes instructions in a loop from power-on to power-off
- A CPU core contains an ALU and a number of registers (and other stuff).

Process Memory

7FFFFFFFFFFFFFFF

argv, environments

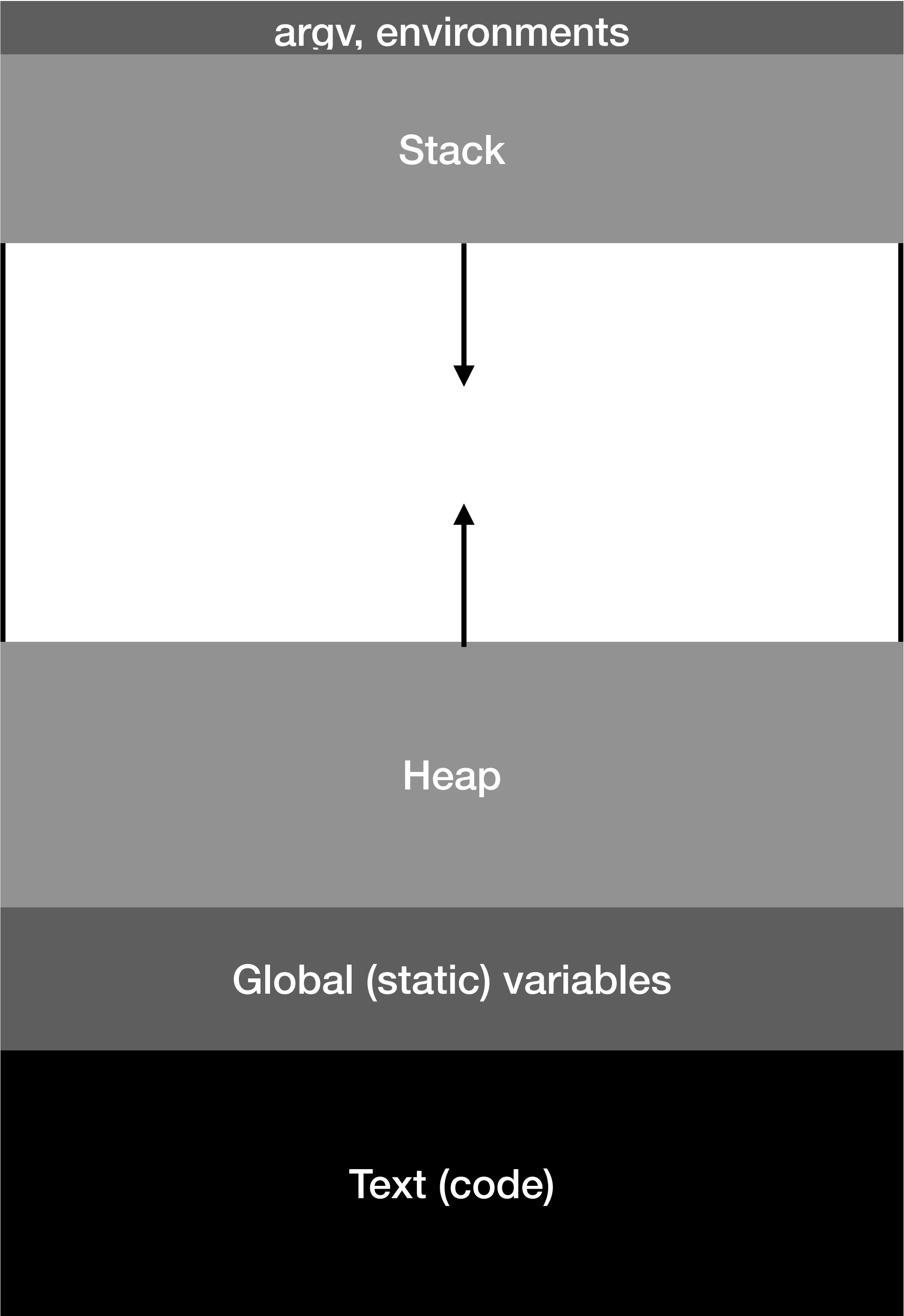
Stack

Heap

Global (static) variables

Text (code)

0000000000000000



Instructions

Instructions

- When you run `./prog arg1 arg2 arg3`, a *loader* puts the content of `prog` into memory, and:

Instructions

- When you run `./prog arg1 arg2 arg3`, a *loader* puts the content of `prog` into memory, and:
 - Moves the *pc* to the first instruction in `prog`

Instructions

- When you run `./prog arg1 arg2 arg3`, a *loader* puts the content of `prog` into memory, and:
 - Moves the *pc* to the first instruction in `prog`
 - Initializes the stack

Instructions

- When you run `./prog arg1 arg2 arg3`, a *loader* puts the content of `prog` into memory, and:
 - Moves the *pc* to the first instruction in `prog`
 - Initializes the stack
 - Copies the command-line arguments into memory

Instructions

- When you run `./prog arg1 arg2 arg3`, a *loader* puts the content of `prog` into memory, and:
 - Moves the *pc* to the first instruction in `prog`
 - Initializes the stack
 - Copies the command-line arguments into memory
 - ...

Machine

Your computer can do many things at the same time...



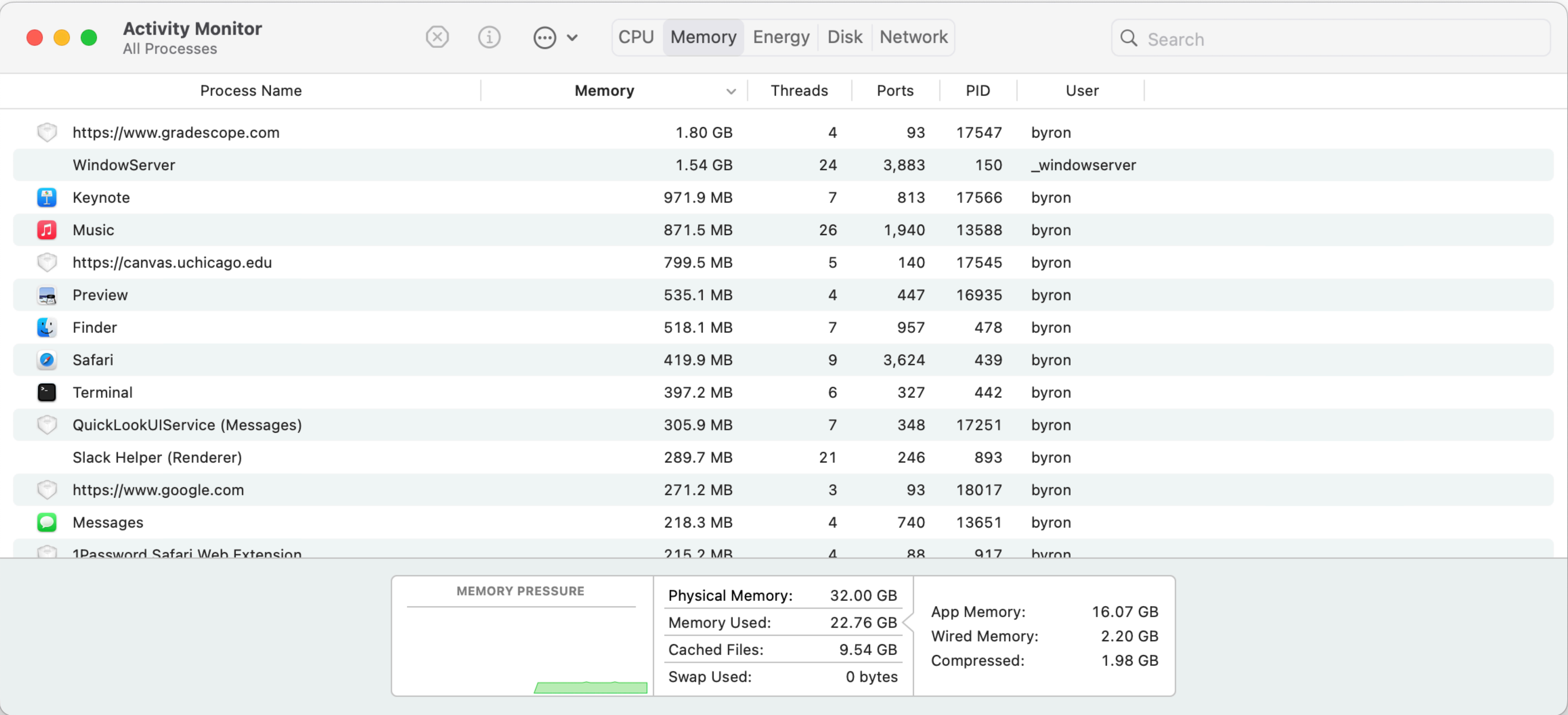
Machine

Your computer can do many things at the same time...



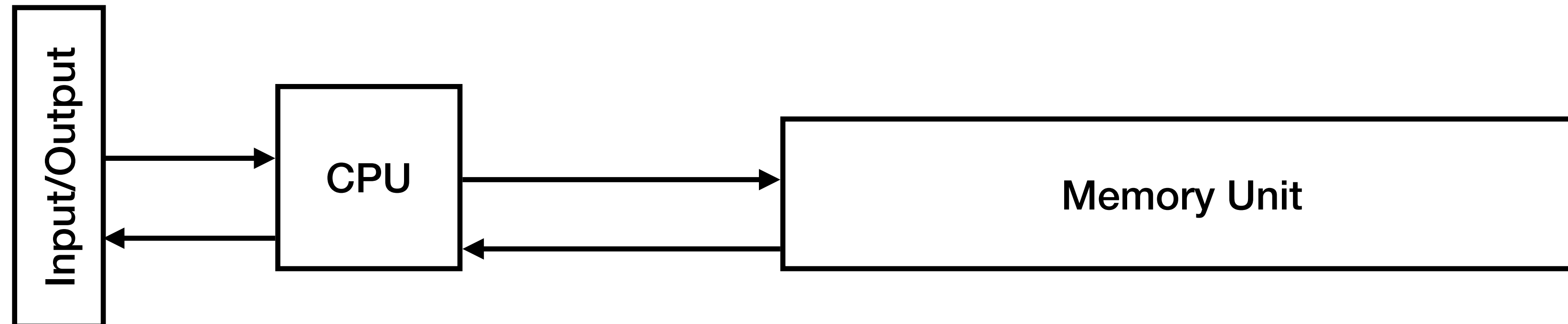
Machine

Your computer can do many things at the same time...



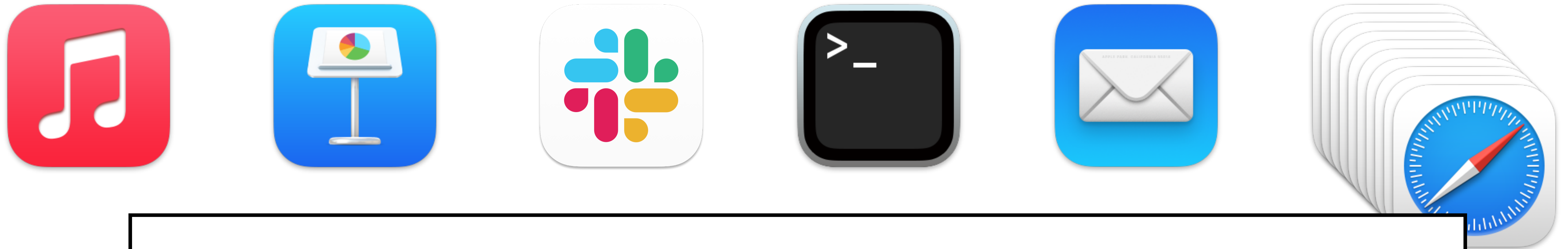
Machine

Your computer can do many things at the same time...

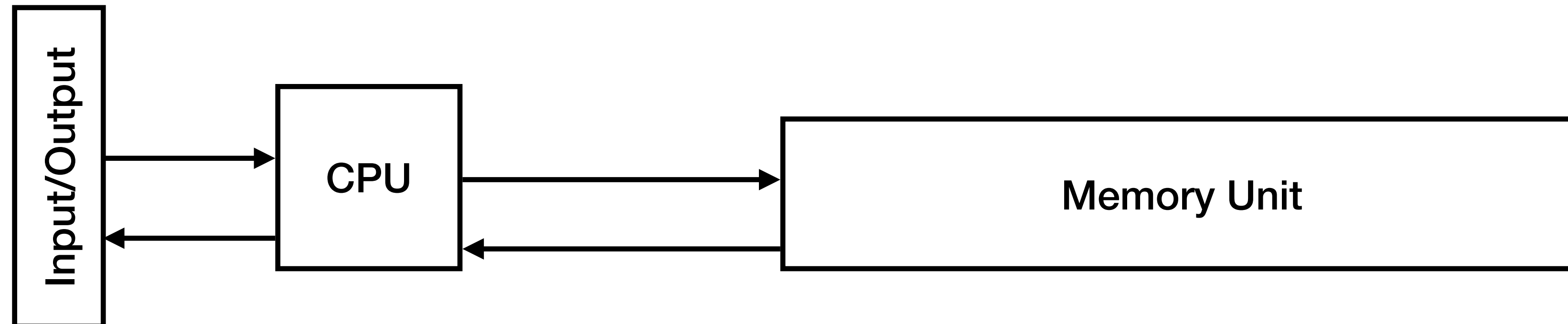


Machine

Your computer can do many things at the same time...

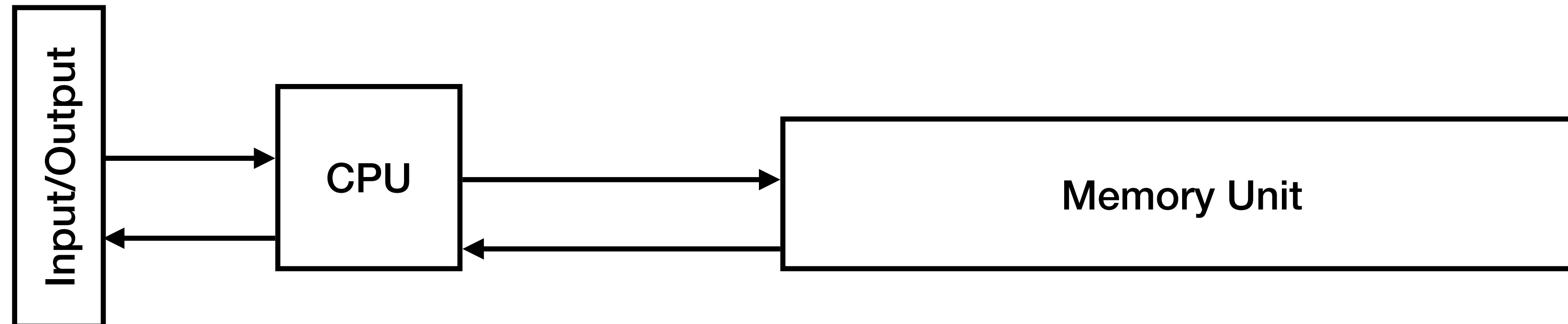


Operating System (OS)



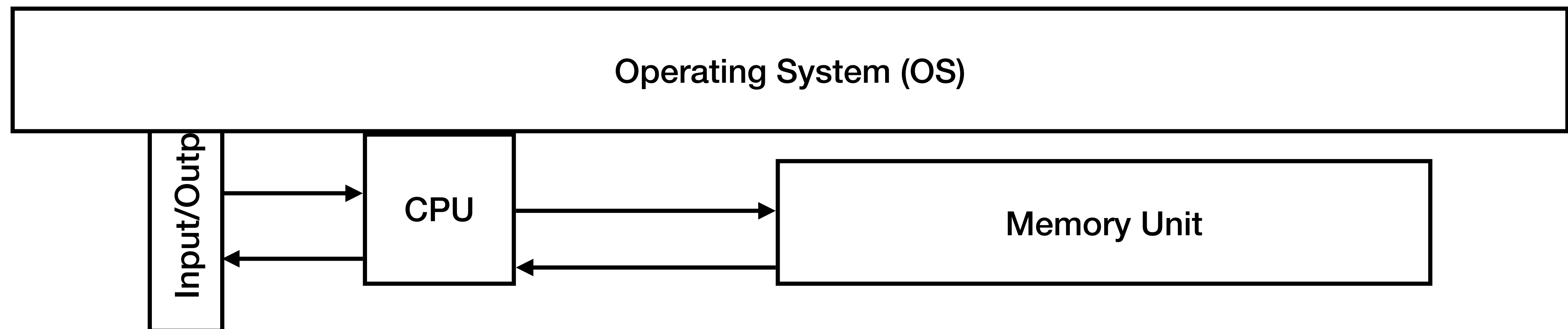
Machine

Your computer can do many things at the same time...



Machine

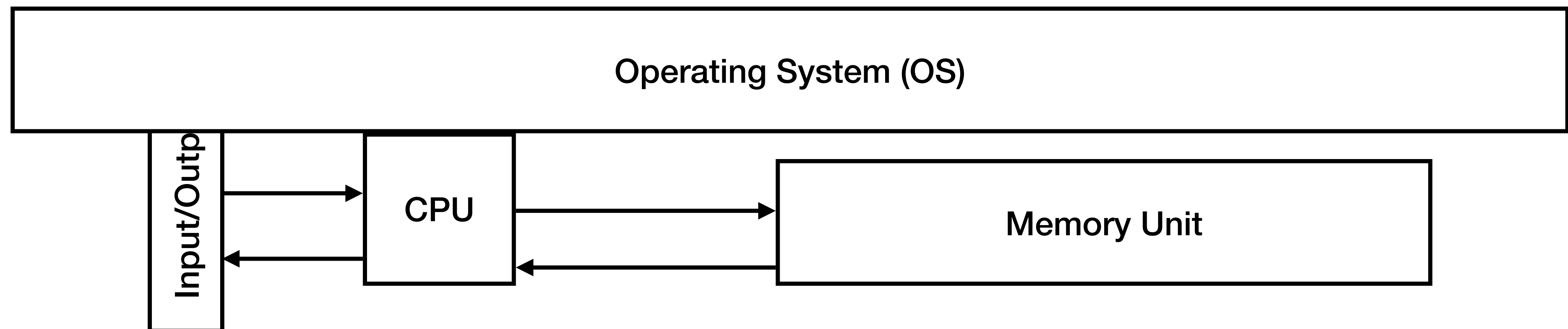
Your computer can do many things at the same time...



Machine

Your computer can do many things at the same time...

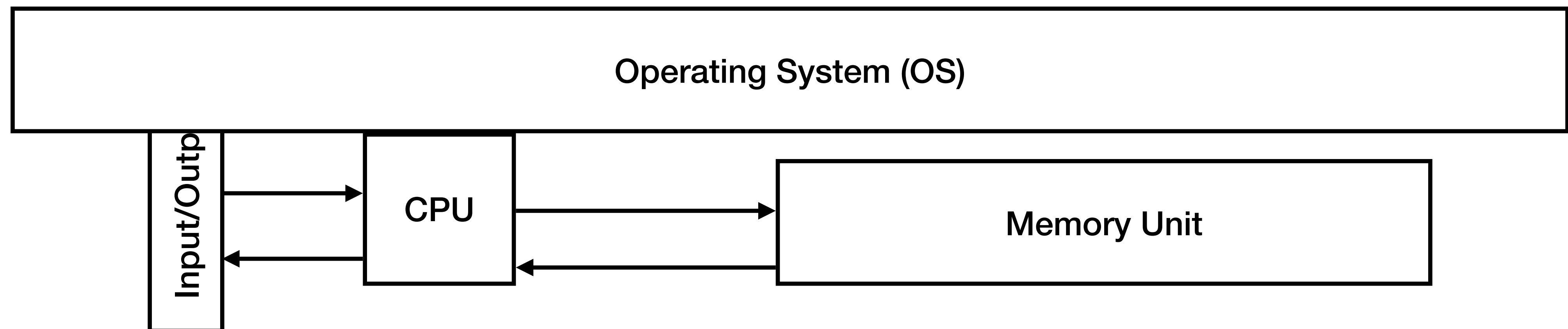
- The operating system creates an illusion that each process is running by itself by:



Machine

Your computer can do many things at the same time...

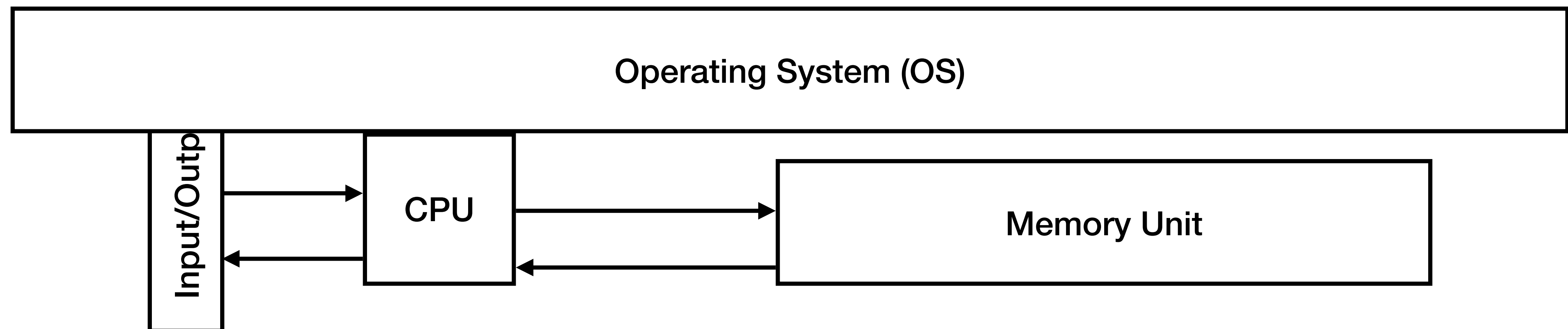
- The operating system creates an illusion that each process is running by itself by:
 - *Context switching* -- rapidly switching which process has control over the CPU



Machine

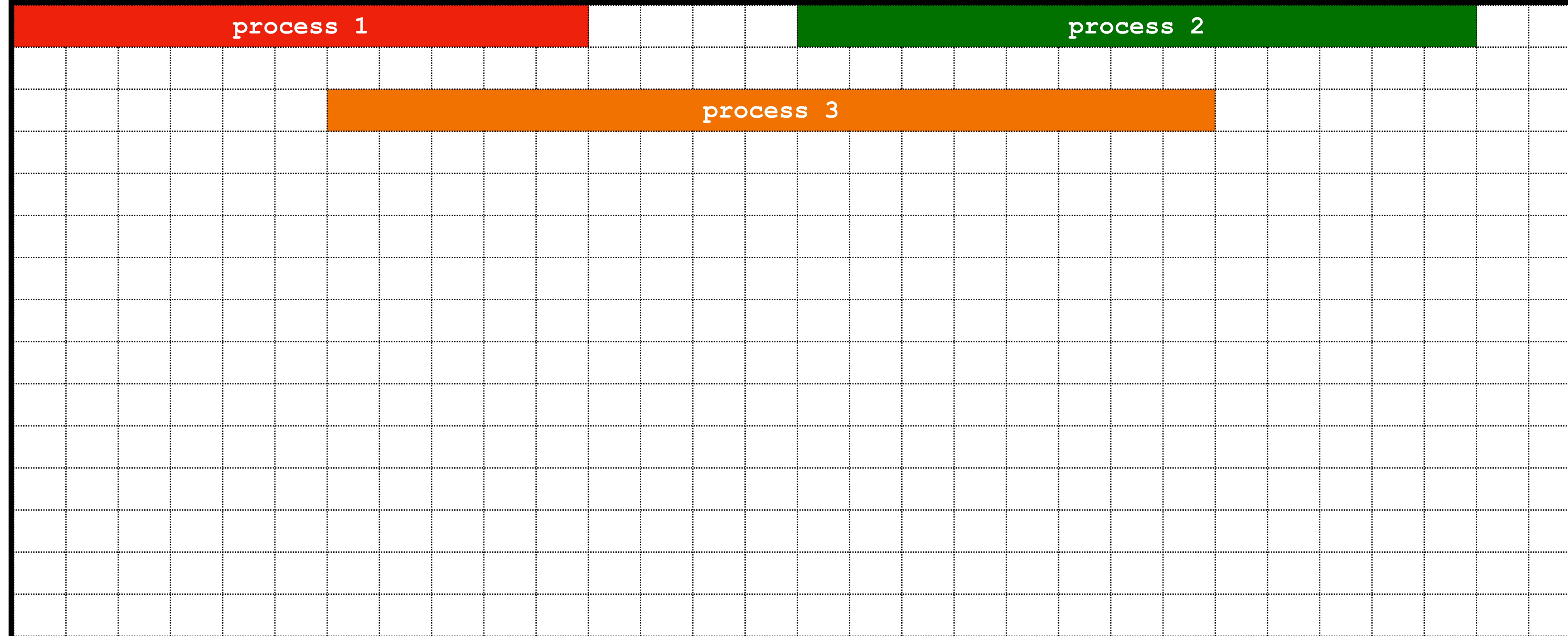
Your computer can do many things at the same time...

- The operating system creates an illusion that each process is running by itself by:
 - *Context switching* -- rapidly switching which process has control over the CPU
 - *Virtual memory* -- providing each process with its own address space



Virtual Memory

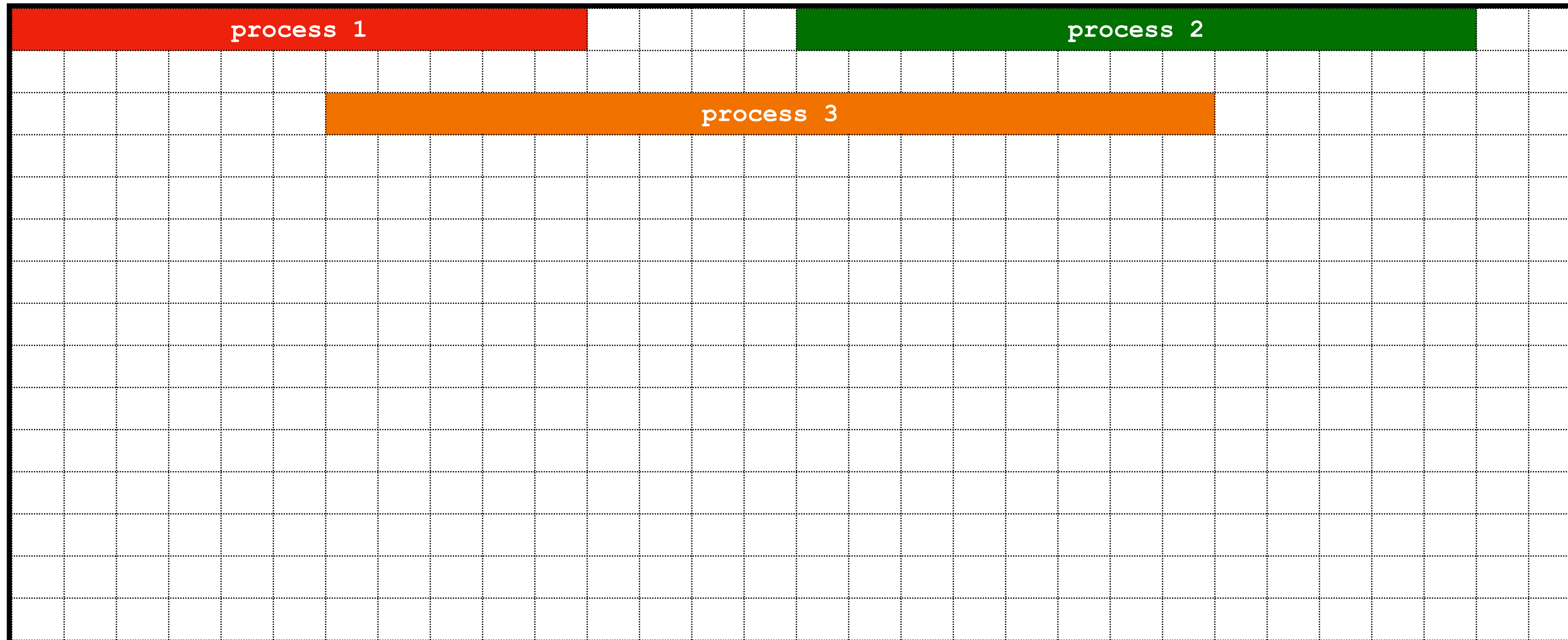
Physical memory



Virtual Memory

What if process 1 needs more memory?

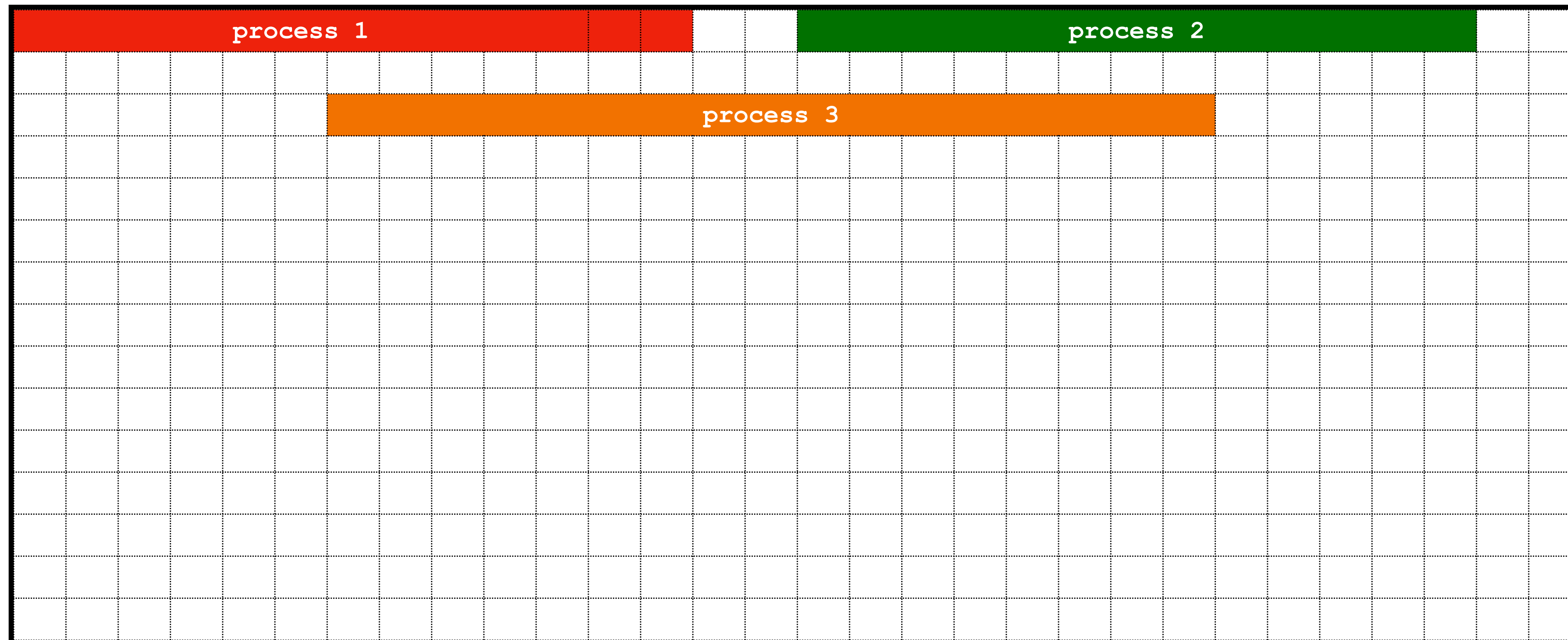
Physical memory



Virtual Memory

What if process 1 needs more memory?

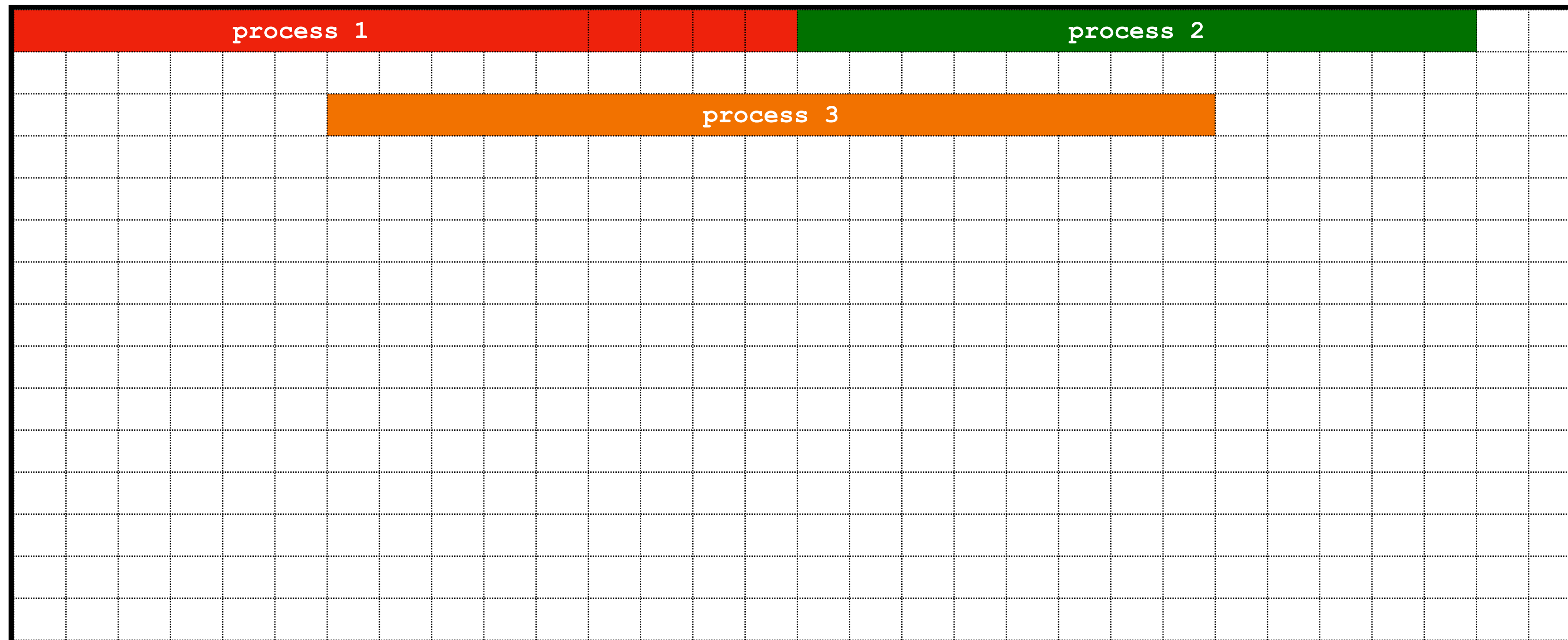
Physical memory



Virtual Memory

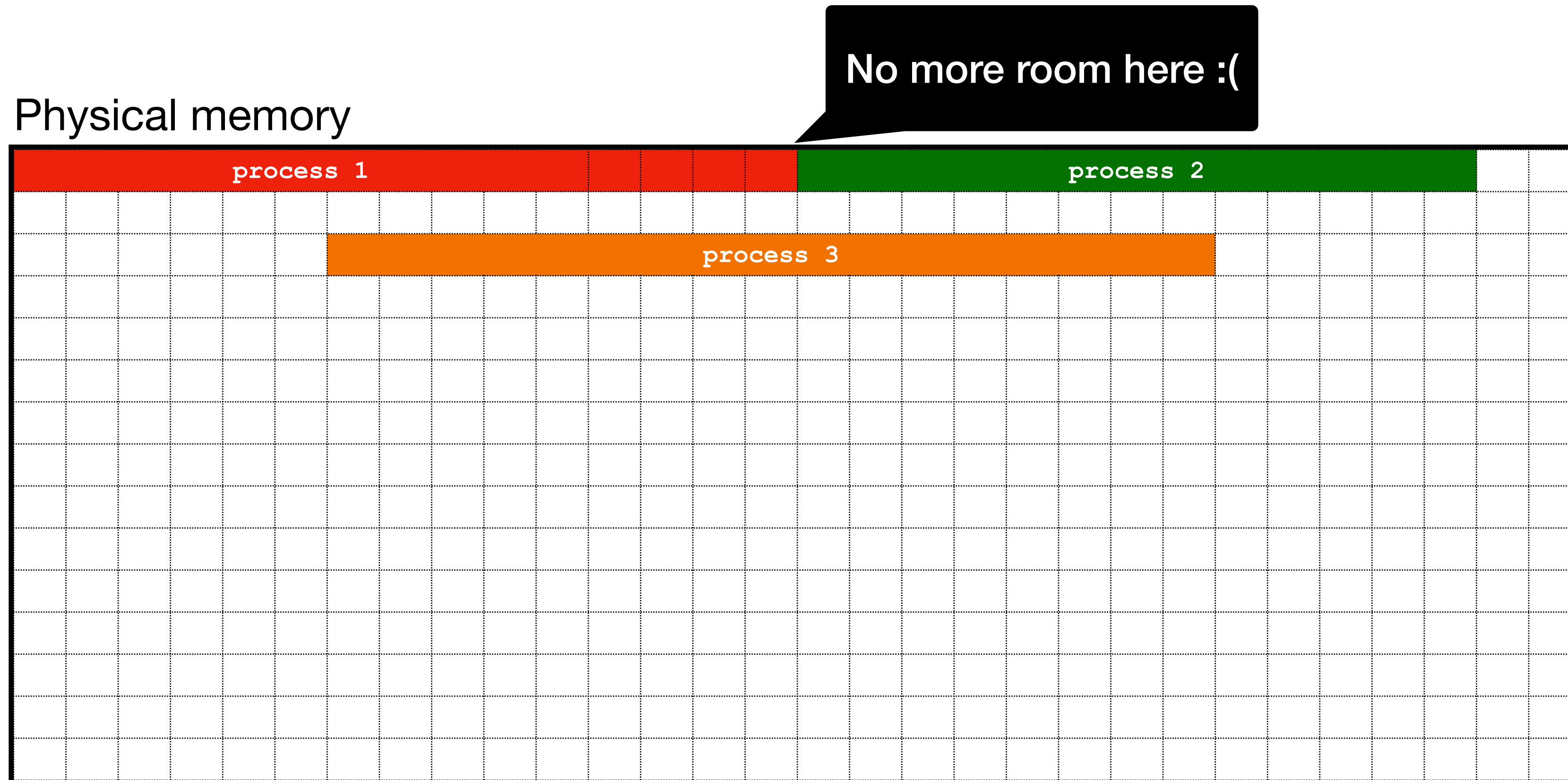
What if process 1 needs more memory?

Physical memory



Virtual Memory

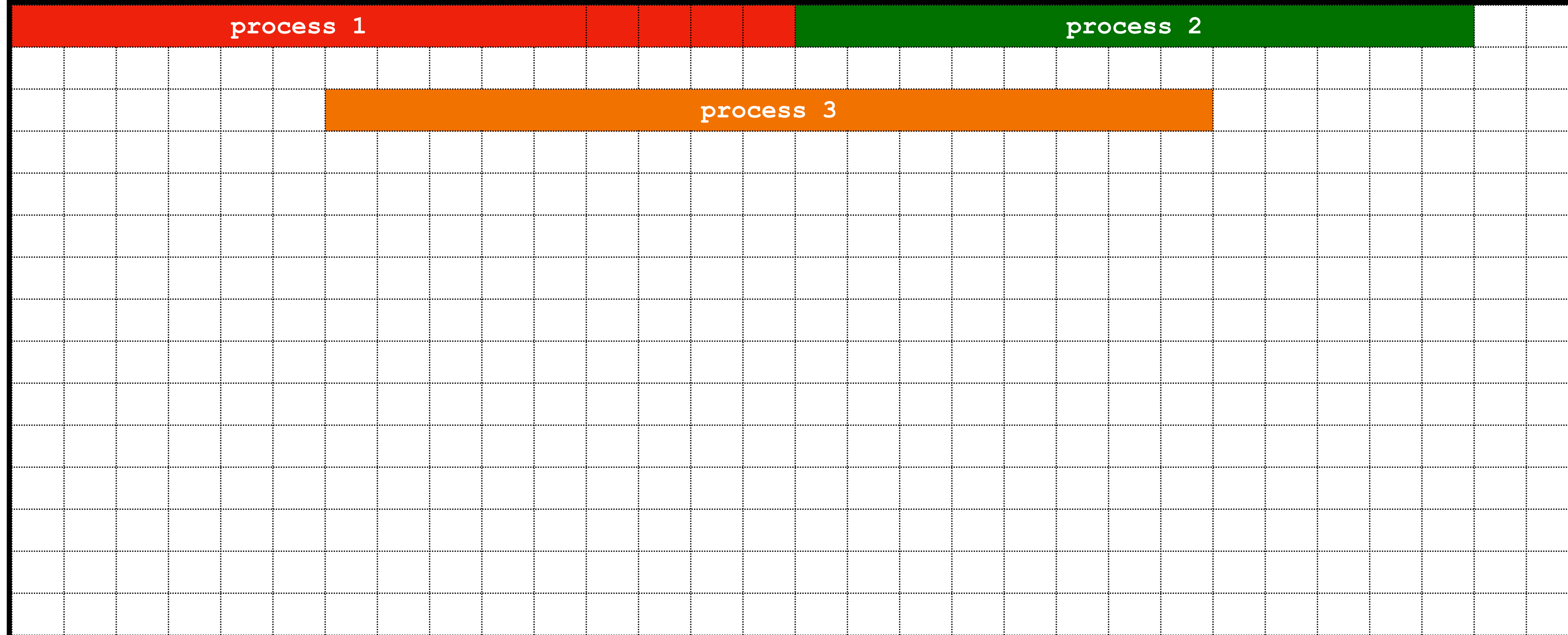
What if process 1 needs more memory?



Virtual Memory

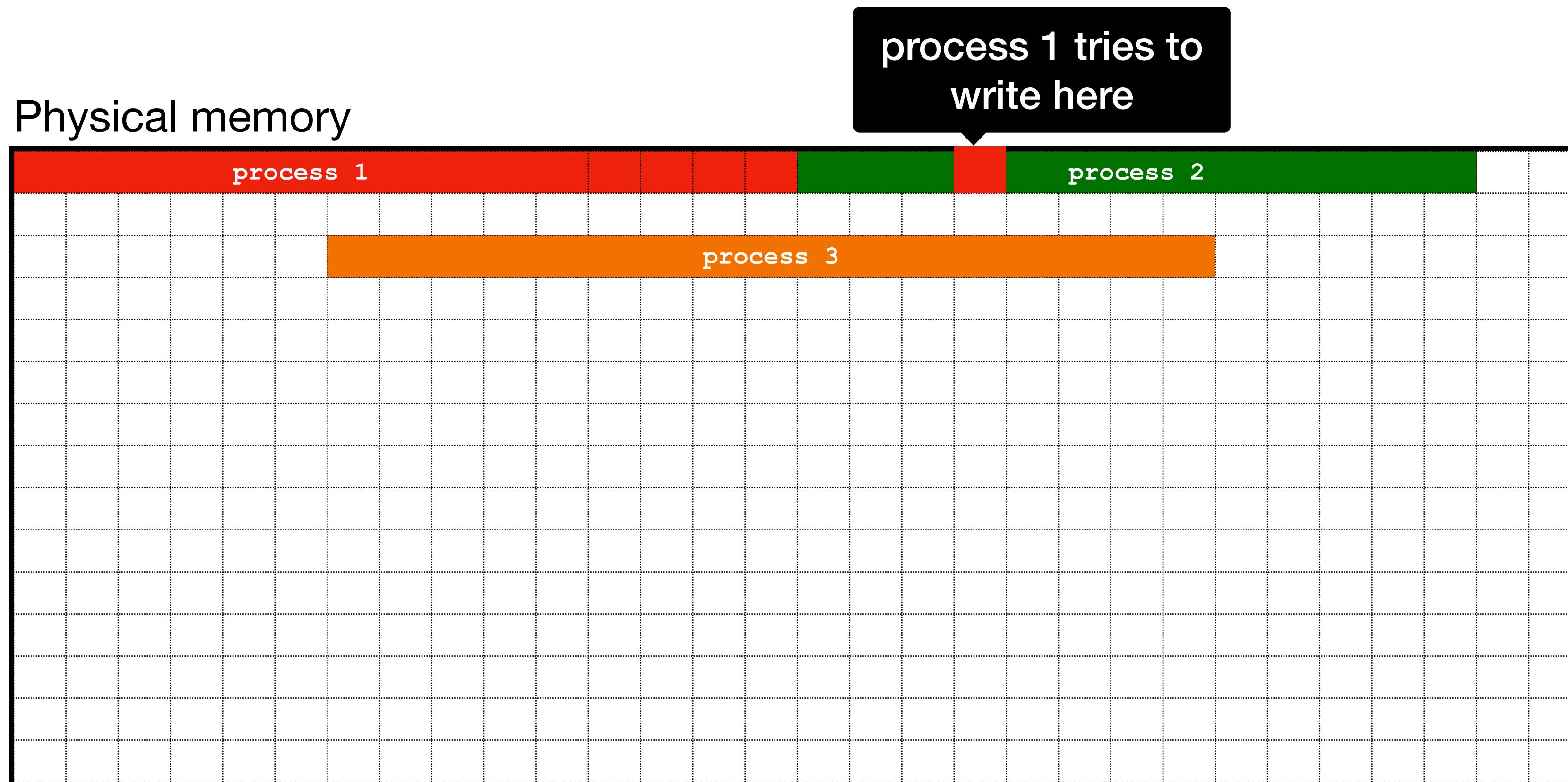
What if process 1 is buggy or malicious?

Physical memory

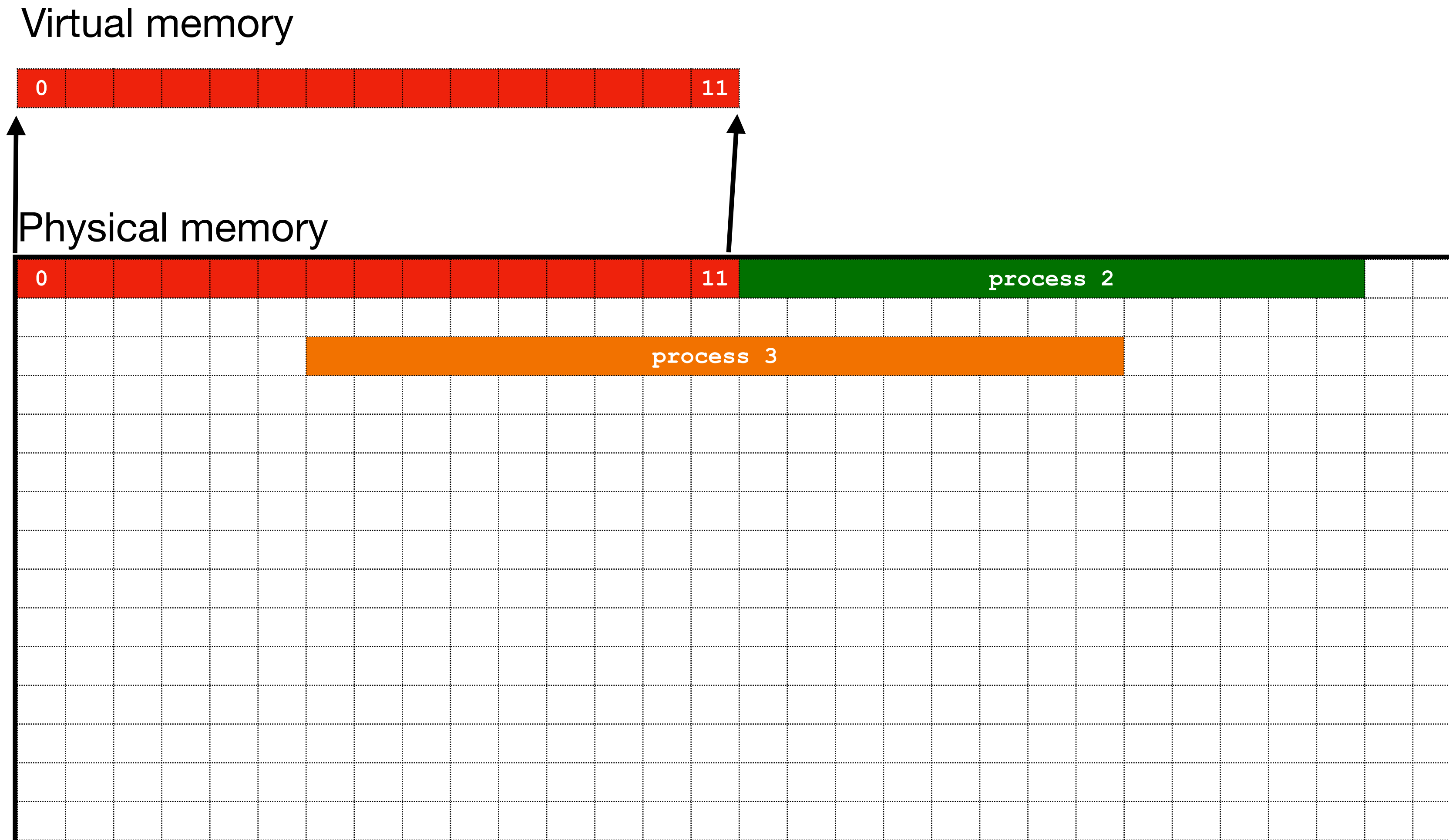


Virtual Memory

What if process 1 is buggy or malicious?



Virtual Memory

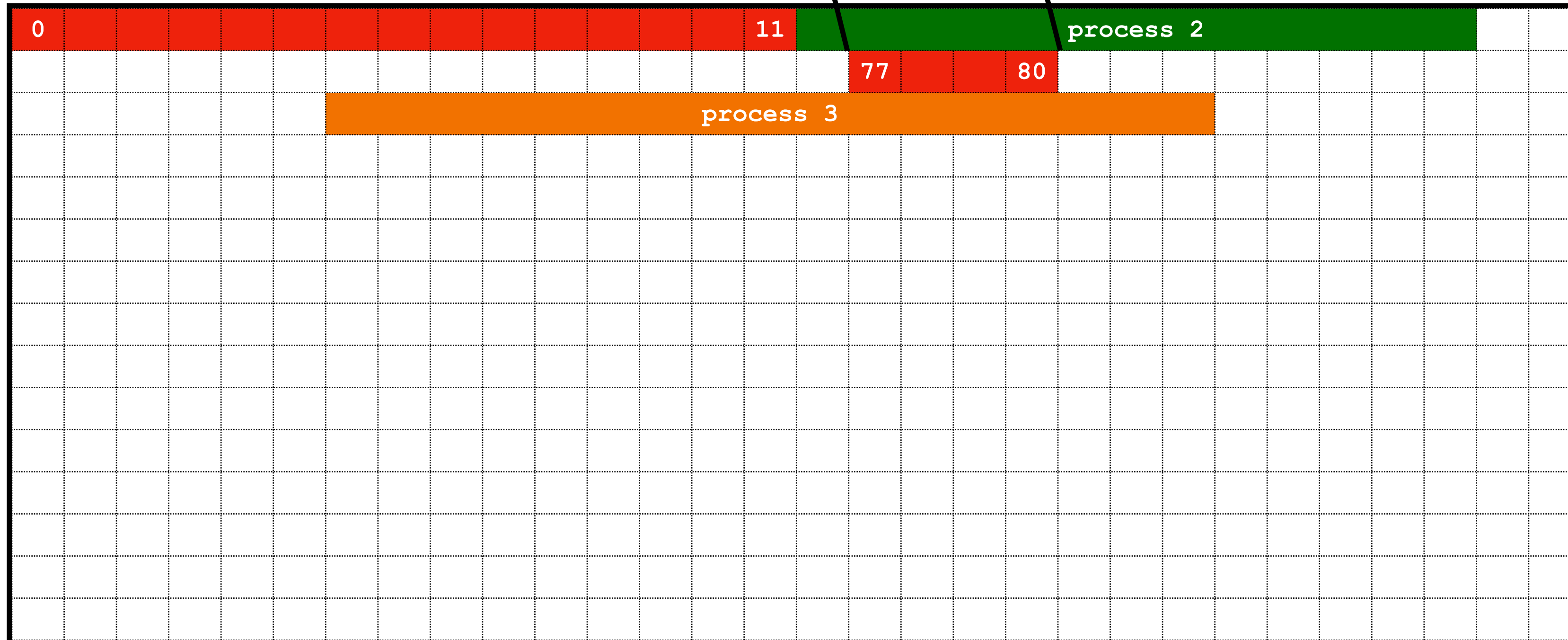


Virtual Memory

Virtual memory



Physical memory

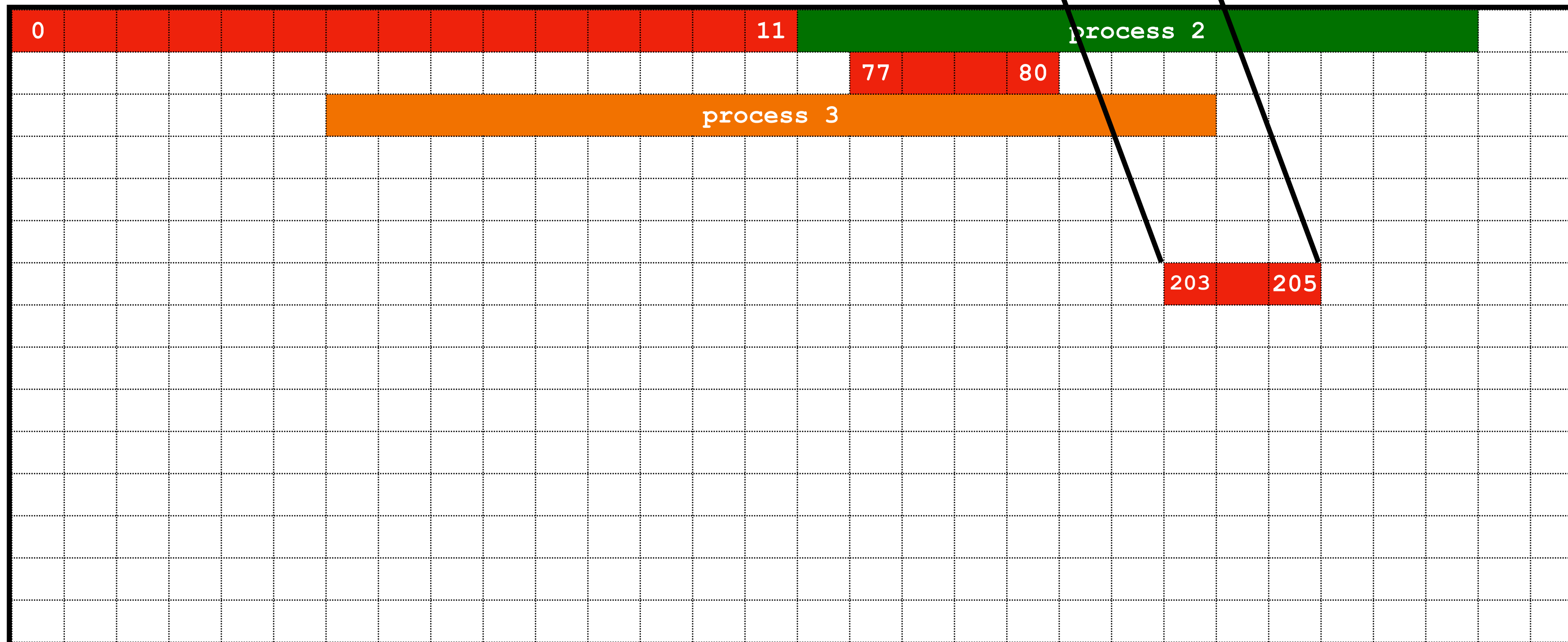


Virtual Memory

Virtual memory



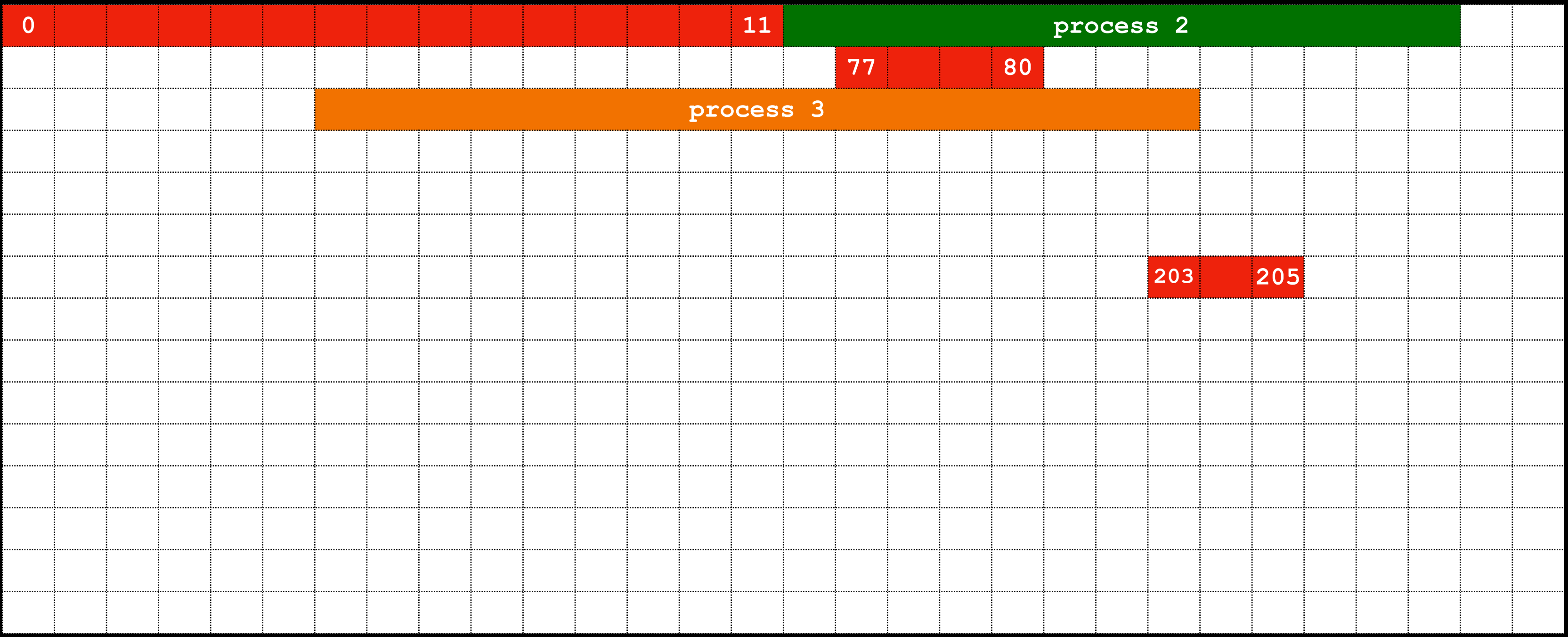
Physical memory



Virtual Memory



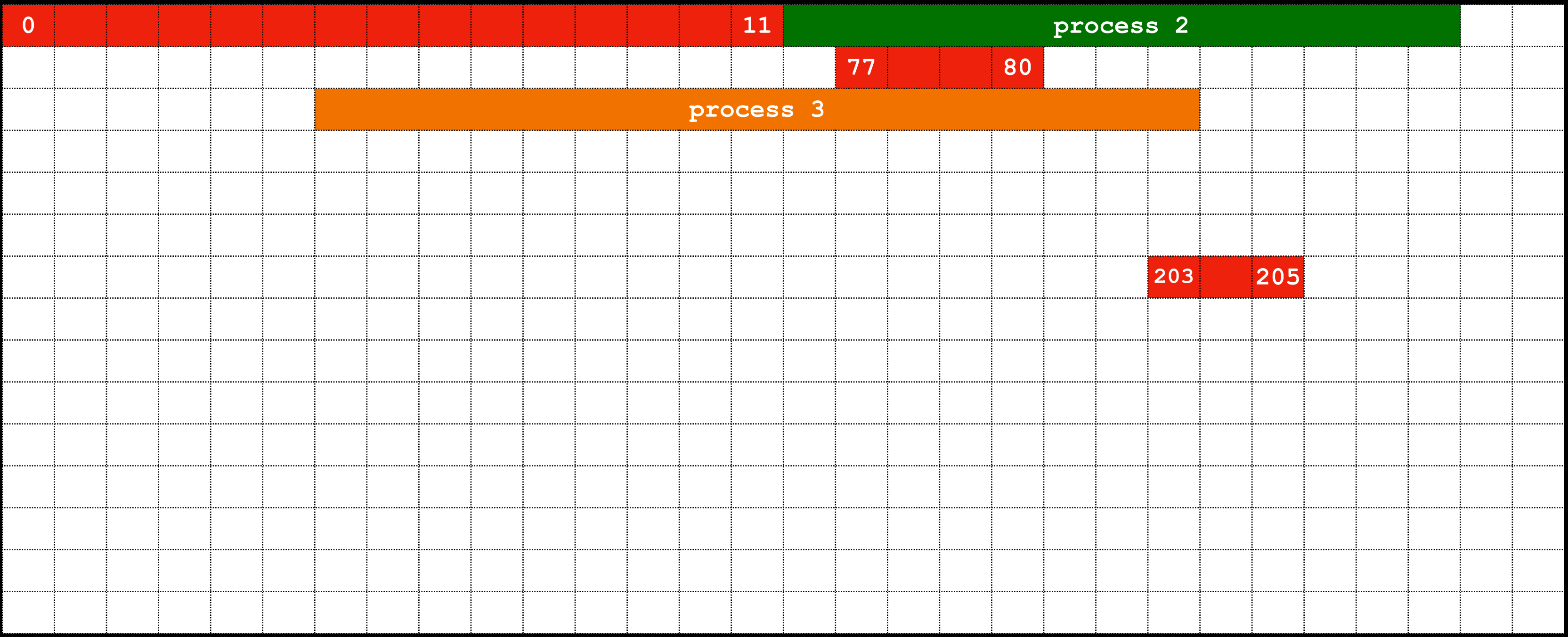
Physical memory



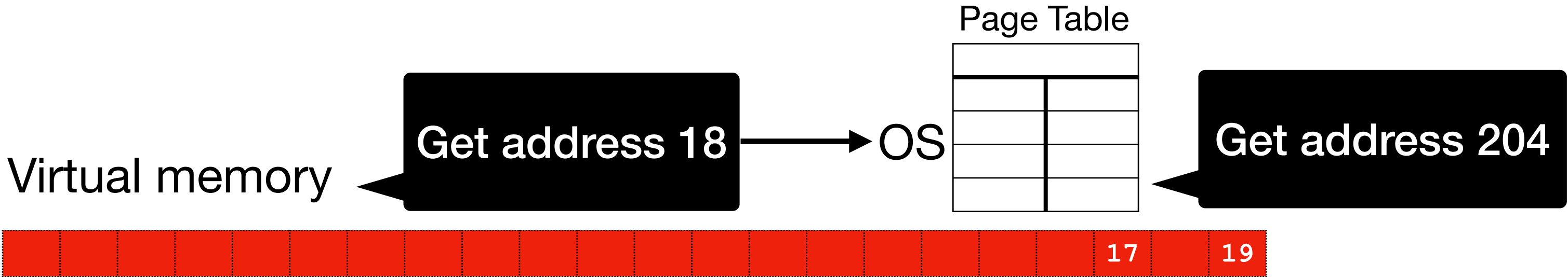
Virtual Memory



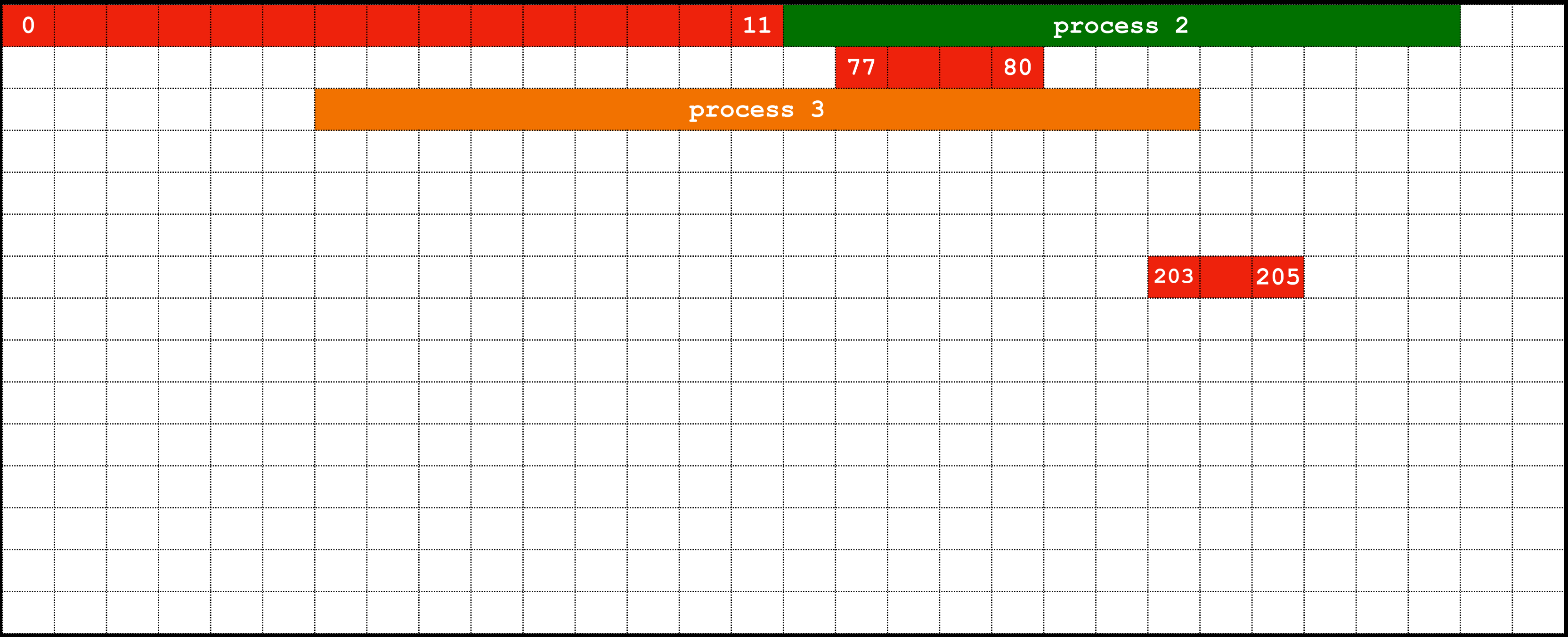
Physical memory



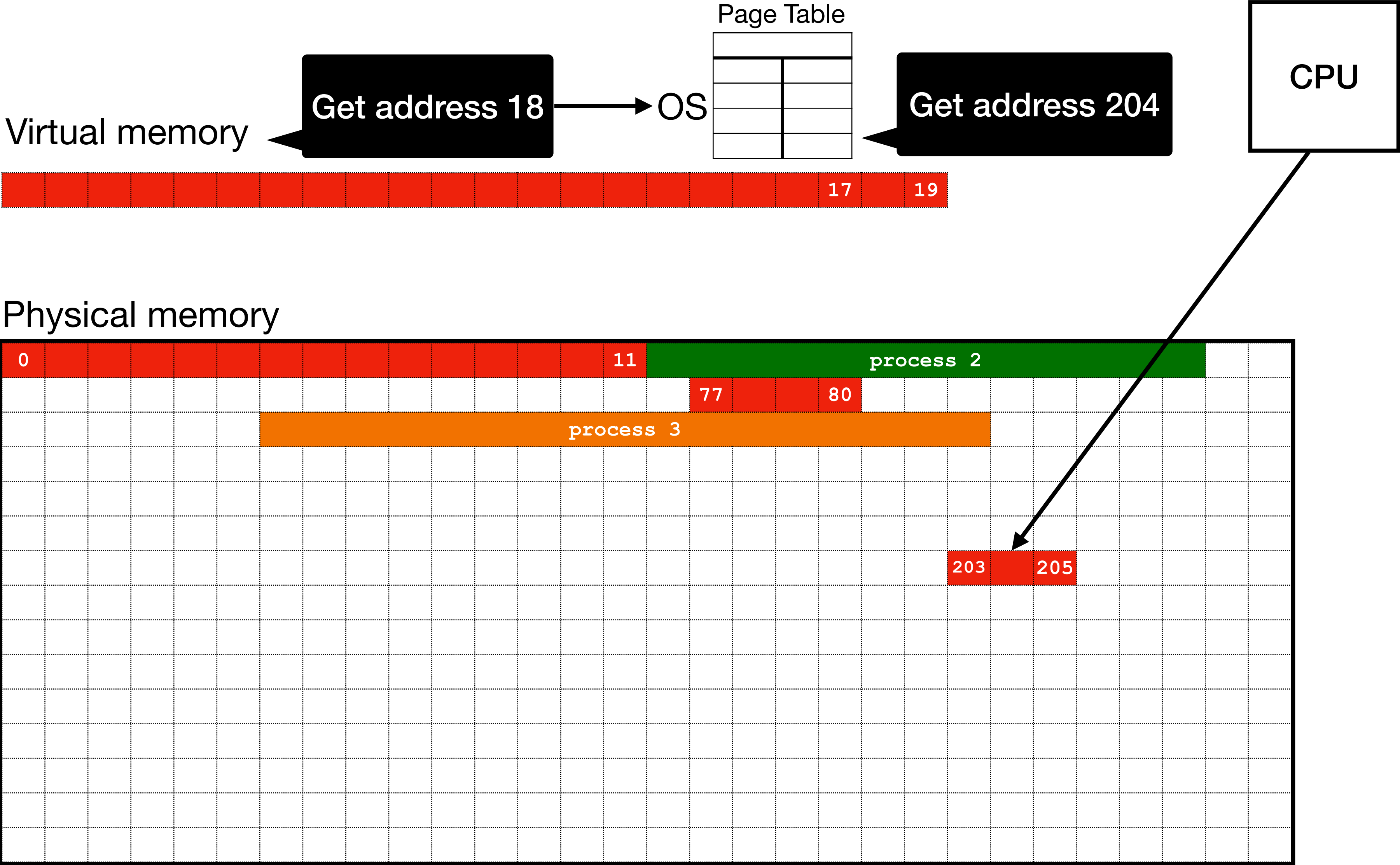
Virtual Memory



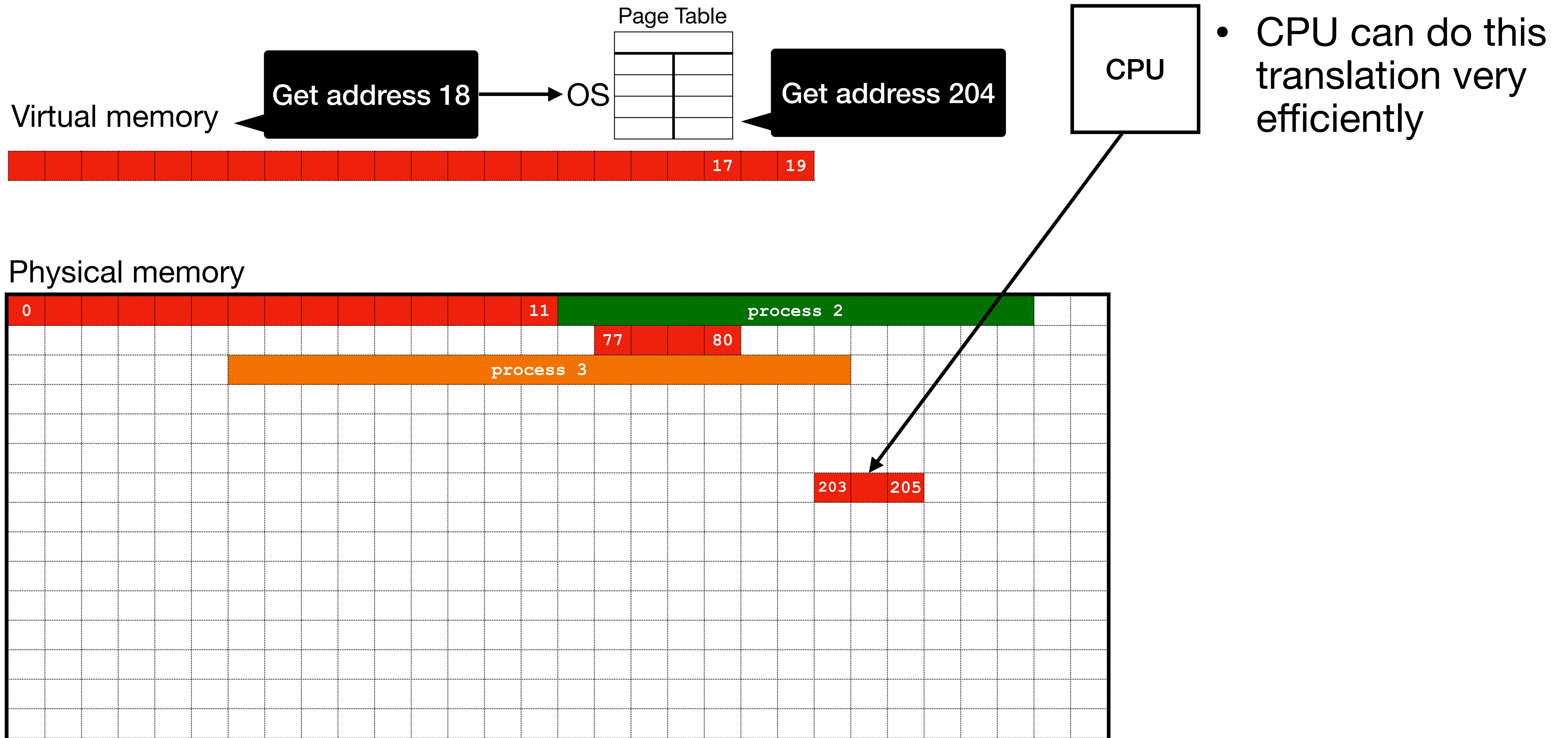
Physical memory



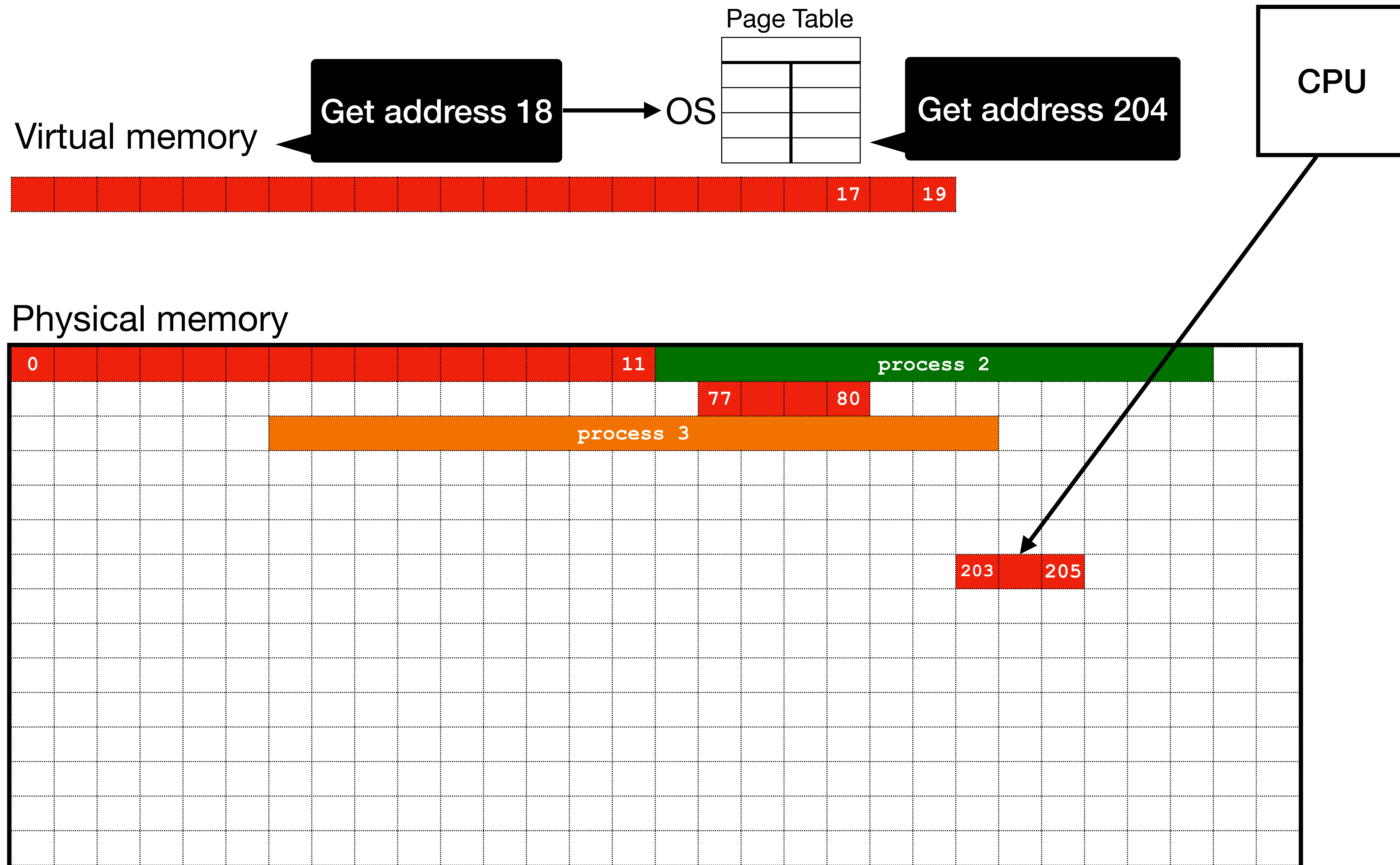
Virtual Memory



Virtual Memory

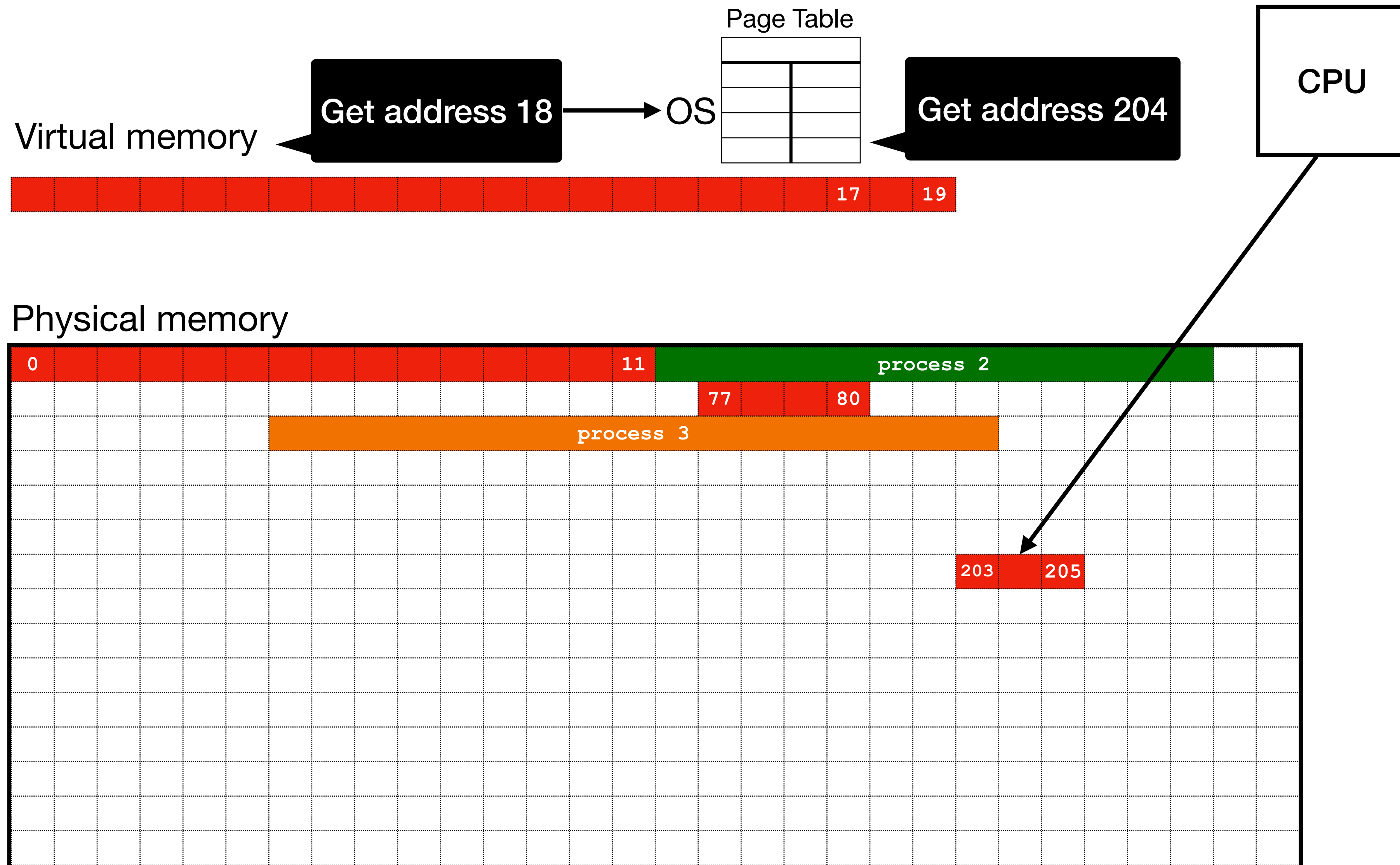


Virtual Memory



- CPU can do this translation very efficiently
- The chunks of memory used to be called *segments*.

Virtual Memory



- CPU can do this translation very efficiently
- The chunks of memory used to be called *segments*.
- segmentation fault!

Context Switching

Context Switching

- Each process has its own

Context Switching

- Each process has its own
 - Virtual memory

Context Switching

- Each process has its own
 - Virtual memory
 - Registers

Context Switching

- Each process has its own
 - Virtual memory
 - Registers
 - Program counter

Context Switching

- Each process has its own
 - Virtual memory
 - Registers
 - Program counter
 - ...

Context Switching

- Each process has its own
 - Virtual memory
 - Registers
 - Program counter
 - ...
- OS keeps track of these data in its internal data structure.

Threads

Activity Monitor

All Processes

CPU













Memory

Energy

Disk

Network

Search

Process Name	Memory	Threads	Ports	PID	User
 https://www.gradescope.com	1.80 GB	4	93	17547	byron
WindowServer	1.54 GB	24	3,883	150	_windowserver
 Keynote	971.9 MB	7	813	17566	byron
 Music	871.5 MB	26	1,940	13588	byron
 https://canvas.uchicago.edu	799.5 MB	5	140	17545	byron
 Preview	535.1 MB	4	447	16935	byron
 Finder	518.1 MB	7	957	478	byron
 Safari	419.9 MB	9	3,624	439	byron
 Terminal	397.2 MB	6	327	442	byron
 QuickLookUIService (Messages)	305.9 MB	7	348	17251	byron
Slack Helper (Renderer)	289.7 MB	21	246	893	byron
 https://www.google.com	271.2 MB	3	93	18017	byron
 Messages	218.3 MB	4	740	13651	byron
 1Password Safari Web Extension	215.2 MB	4	88	917	byron

MEMORY PRESSURE

Physical Memory:

32.00 GB

Memory Used:

22.76 GB

Cached Files:

9.54 GB

Swap Used:

0 bytes

App Memory:

16.07 GB

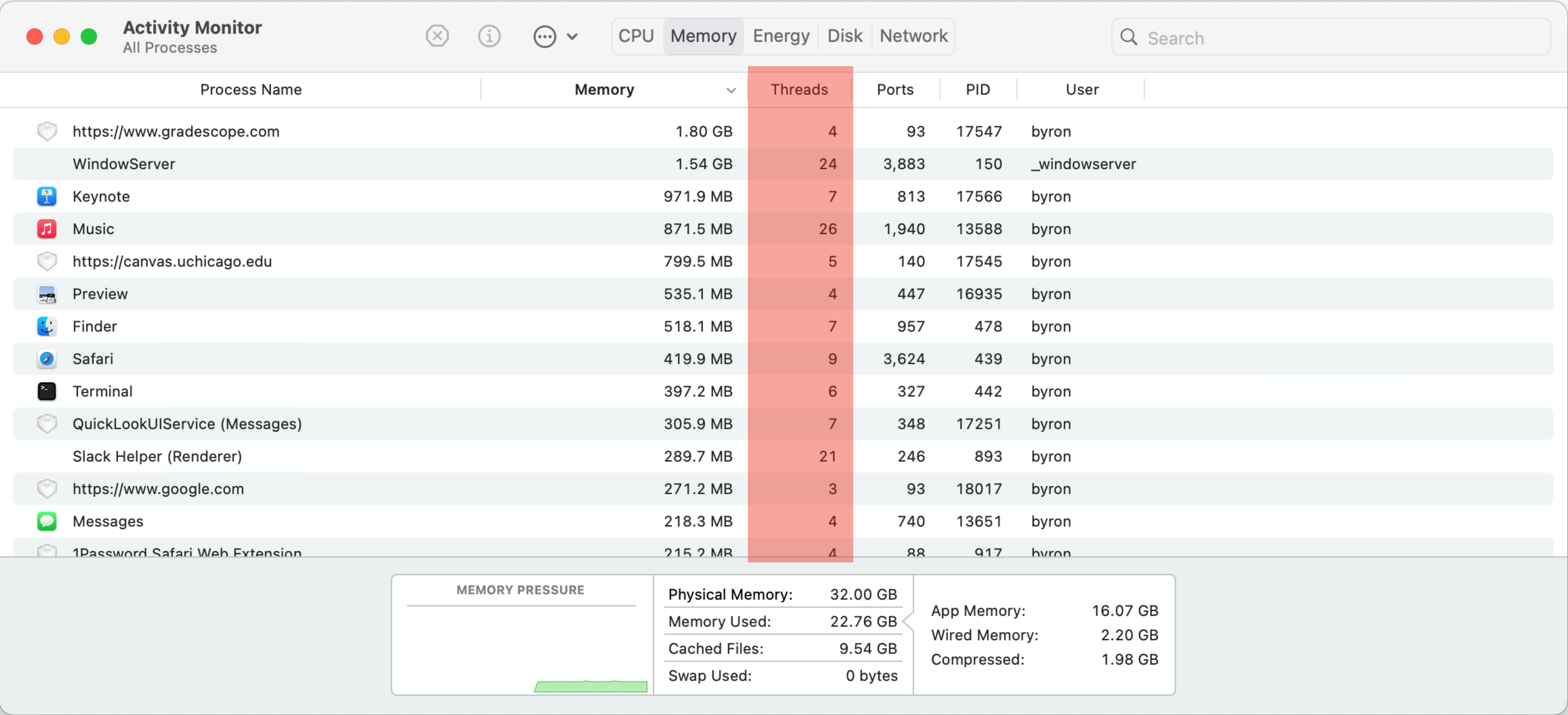
Wired Memory:

2.20 GB

Compressed:

1.98 GB

Threads



Threads

Threads

- A thread is a unit of execution. Each thread has its own:

Threads

- A thread is a unit of execution. Each thread has its own:
 - Thread ID

Threads

- A thread is a unit of execution. Each thread has its own:
 - Thread ID
 - Stack

Threads

- A thread is a unit of execution. Each thread has its own:
 - Thread ID
 - Stack
 - Program counter (pc)

Threads

- A thread is a unit of execution. Each thread has its own:
 - Thread ID
 - Stack
 - Program counter (pc)
 - Registers

Threads

- A thread is a unit of execution. Each thread has its own:
 - Thread ID
 - Stack
 - Program counter (pc)
 - Registers
- A process contains a number of threads. Threads within a process share:

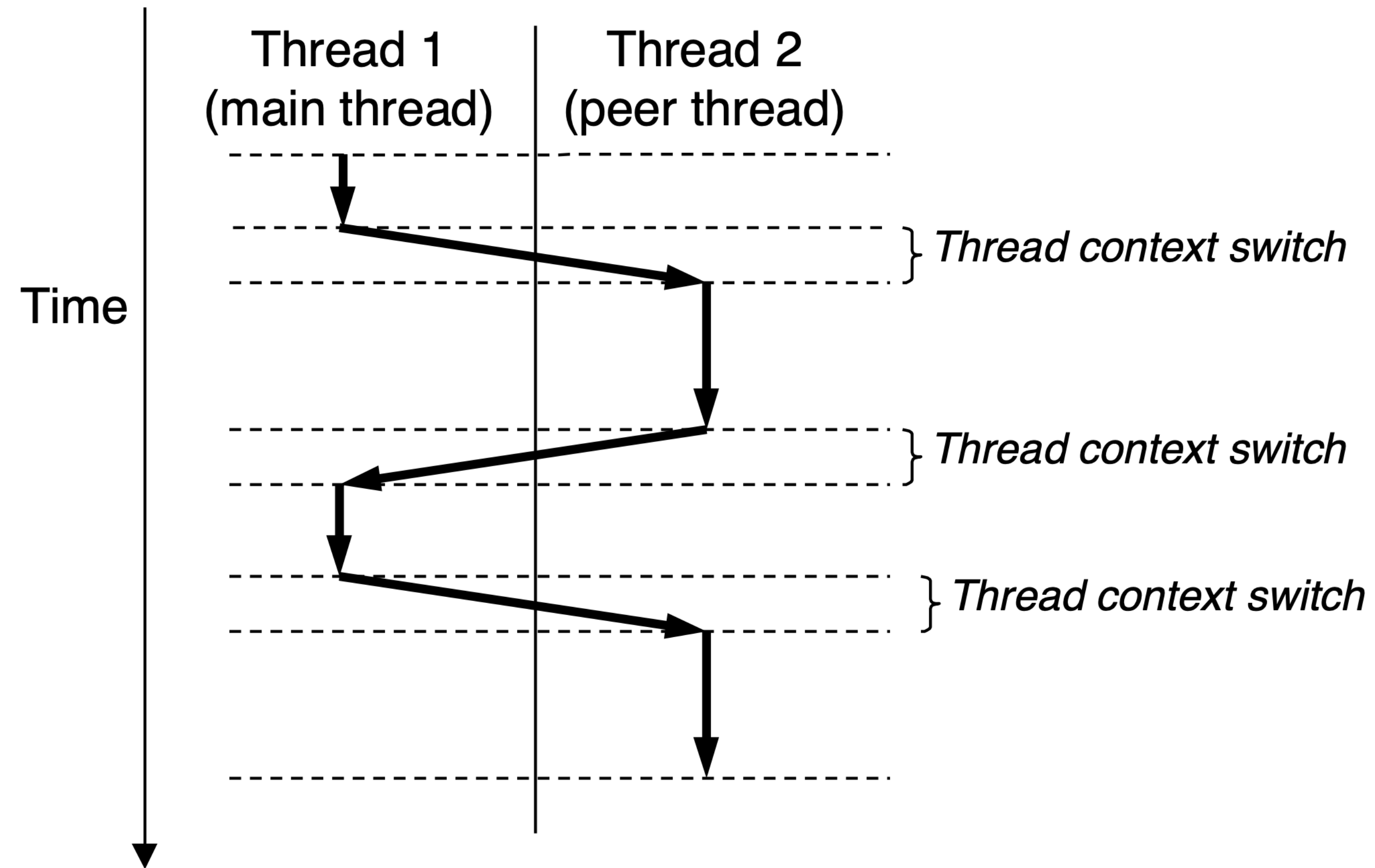
Threads

- A thread is a unit of execution. Each thread has its own:
 - Thread ID
 - Stack
 - Program counter (pc)
 - Registers
- A process contains a number of threads. Threads within a process share:
 - Code, data

Threads

- A thread is a unit of execution. Each thread has its own:
 - Thread ID
 - Stack
 - Program counter (pc)
 - Registers
- A process contains a number of threads. Threads within a process share:
 - Code, data
- Threads are executed *concurrently*.

Threads



Threads

- In C, threads are managed by a library called **pthread**

Threads

Creation

```
#include <pthread.h>
#include <stdio.h>

void *thread(void *data);

int main()
{
    pthread_t tid;

    pthread_create(&tid, NULL, thread, "hello, world!");
    pthread_join(tid, NULL);

    return 0;
}

void *thread(void *data)
{
    char *str = data;
    printf("%s\n", str);

    return NULL;
}
```

Threads

Creation

```
#include <pthread.h>
#include <stdio.h>
```

```
void *thread(void *data);
```

```
int main()
```

```
{
    pthread_t tid;
```

```
    pthread_create(&tid, NULL, thread, "hello, world!");
    pthread_join(tid, NULL);
```

```
    return 0;
```

```
}
```

```
void *thread(void *data)
```

```
{
```

```
    char *str = data;
    printf("%s\n", str);
```

```
    return NULL;
```

```
}
```

launches a new thread

Threads

Creation

```
#include <pthread.h>
#include <stdio.h>
```

```
void *thread(void *data);
```

```
int main()
{
```

```
    pthread_t tid;
```

```
    pthread_create(&tid, NULL, thread, "hello, world!");
    pthread_join(tid, NULL);
```

```
    return 0;
```

```
}
```

```
void *thread(void *data)
```

```
{
```

```
    char *str = data;
```

```
    printf("%s\n", str);
```

```
    return NULL;
```

```
}
```

waits for the thread to
terminate

Threads

Creation

```
#include <pthread.h>
#include <stdio.h>
```

```
void *thread(void *data);
```

```
int main()
{
    pthread_t tid;

    pthread_create(&tid, NULL, thread, "hello, world!");
    pthread_join(tid, NULL);

    return 0;
}
```

```
void *thread(void *data)
{
    char *str = data;
    printf("%s\n", str);

    return NULL;
}
```

function to run

Threads

Creation

```
int  
pthread_create(pthread_t *thread, const pthread_attr_t *attr,  
               void *(*start_routine) (void *),  
               void *arg);
```

Threads

Creation

`pthread_create`
creates a new thread
and immediately starts
running the thread.

```
int  
pthread_create(pthread_t *thread, const pthread_attr_t *attr,  
               void *(*start_routine) (void *),  
               void *arg);
```

Threads

Creation

```
int  
pthread_create(pthread_t *thread, const pthread_attr_t *attr,  
               void *(*start_routine) (void *),  
               void *arg);
```

Once created, each thread is assigned a thread ID by the OS.

Threads

Creation

```
int  
pthread_create(pthread_t *thread, const pthread_attr_t *attr,  
               void *(*start_routine)(void *),  
               void *arg);
```

Thread attribute.
Almost always just
NULL

Threads

Creation

```
int  
pthread_create(pthread_t *thread, const pthread_attr_t *attr,  
               void *(*start_routine) (void *),  
               void *arg);
```

The function to run in the thread. Arg: void *, Return: void *

Threads

Creation

```
int  
pthread_create(pthread_t *thread, const pthread_attr_t *attr,  
               void *(*start_routine)(void *),  
               void *arg);
```

The argument to the function. Similar to the data pointer in *_walk

Threads

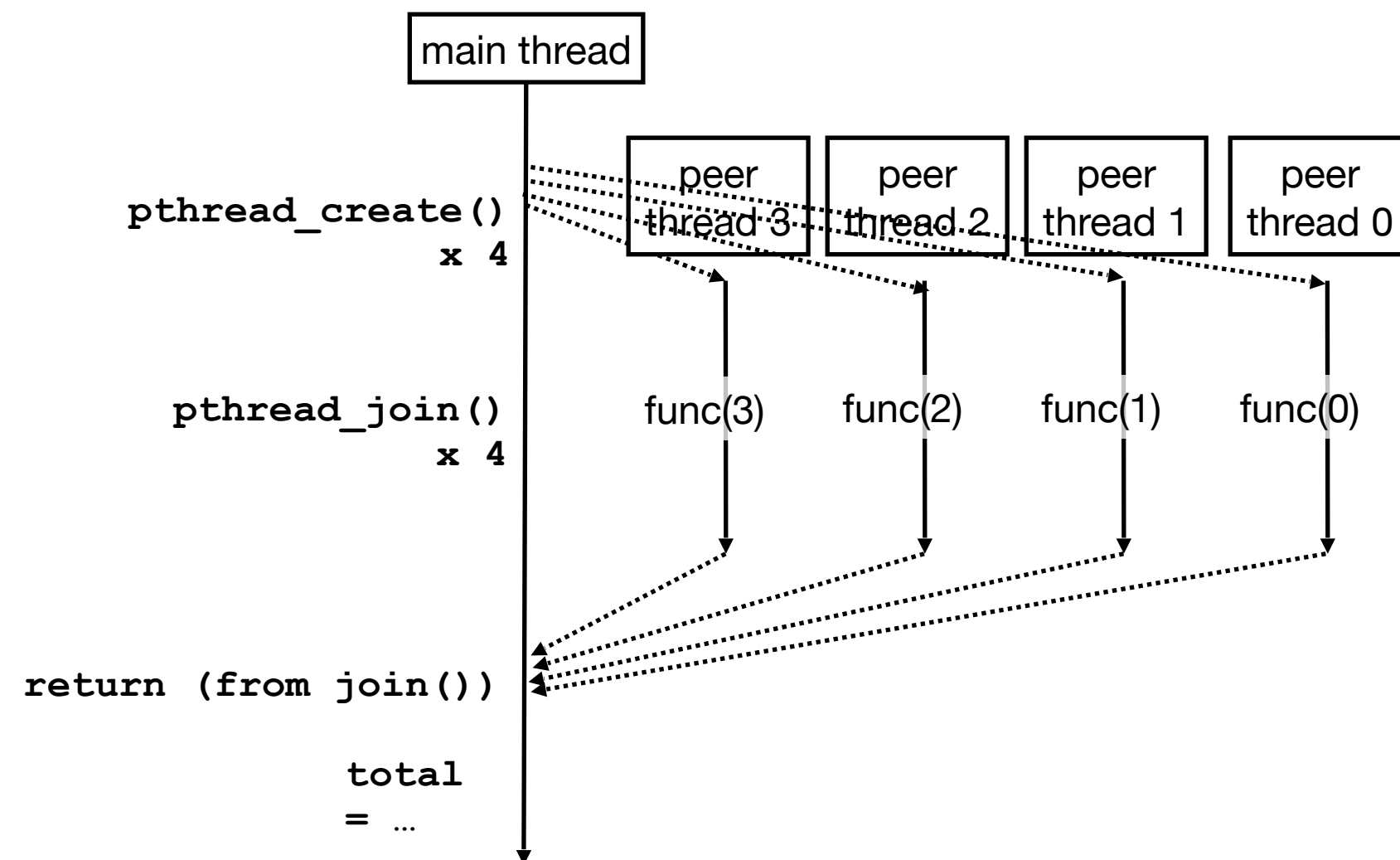
Creation

```
int  
pthread_join(pthread_t thread, void **value_ptr);
```

- Demo

Moral: `slowprimes.c`

- C is so much better than Python at number crunching
- If a problem can be broken down into disjoint subproblems, parallelism shortens the running time by a lot



Moral: `primes.c`

- The best kind of performance improvement is algorithmic improvement -- the one that improves the asymptotic running time.
- Thread creation has its own cost. This cost needs to be factored in when one decides whether to use concurrency.
- Multithreading loses some benefit when the problem cannot be cleanly sliced into subproblems
- Multithreading code is usually a lot more complicated than single-threaded code.

Moral: badcnt.c

- Concurrency bug can happen when threads share resources.
- Threads do not have its own virtual memory space like processes do
 - Memory access is not atomic -- $x += 1$ is three separate steps
 - Read x to a register (load)
 - Increment the register (update)
 - Write the register to memory (store)
- Memory updates need to be synchronized

Moral: `goodcnt.c`

- Synchronization can be done via `semaphores` (e.g. locks)
 - `pthread_mutex_lock` waits the value to be positive
 - `pthread_mutex_unlock` increments the value
- Synchronized code is not parallel :(

Moral: bomb . c

- Asynchronous I/O
- Semaphores can be used to indicate the readiness of a value

When should I use threads?

- Spot parallelizable code
 - Code that doesn't depend on other's result
 - E.g. loop where each iteration is independent
- Beware of concurrency bugs
 - Read-write conflict
 - Write-write conflict
 - Semaphore

Thread (cont.)

badcnt.c

```
/* shared variable */
unsigned int cnt = 0;
void *count(void *);

int main(void)
{
    pthread_t tid1, tid2;

    pthread_create(&tid1, NULL, count, NULL);
    pthread_create(&tid2, NULL, count, NULL);

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    if (cnt == N * 2) {
        printf("OK cnt=%u\n", cnt);
    } else {
        printf("BOOM cnt=%u\n", cnt);
    }

    return 0;
}
```

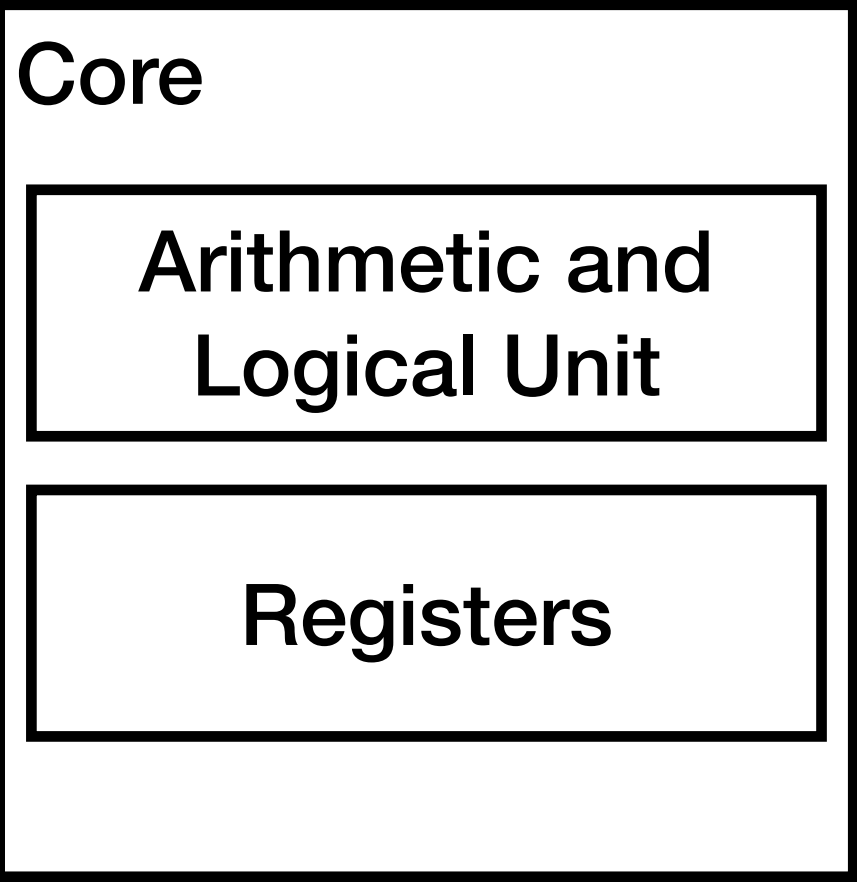
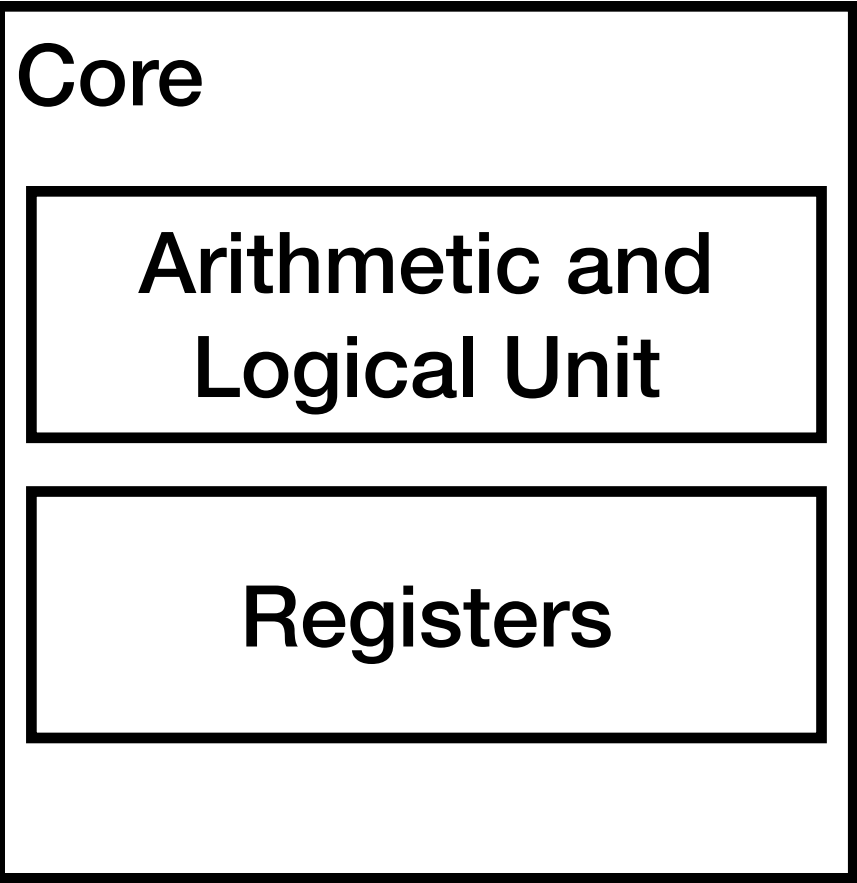
```
void *count(void *arg)
{
    (void) arg;

    for (unsigned int i = 0; i < N; i++) {
        cnt++;
    }

    return NULL;
}
```

Thread (cont.)

badcnt.c



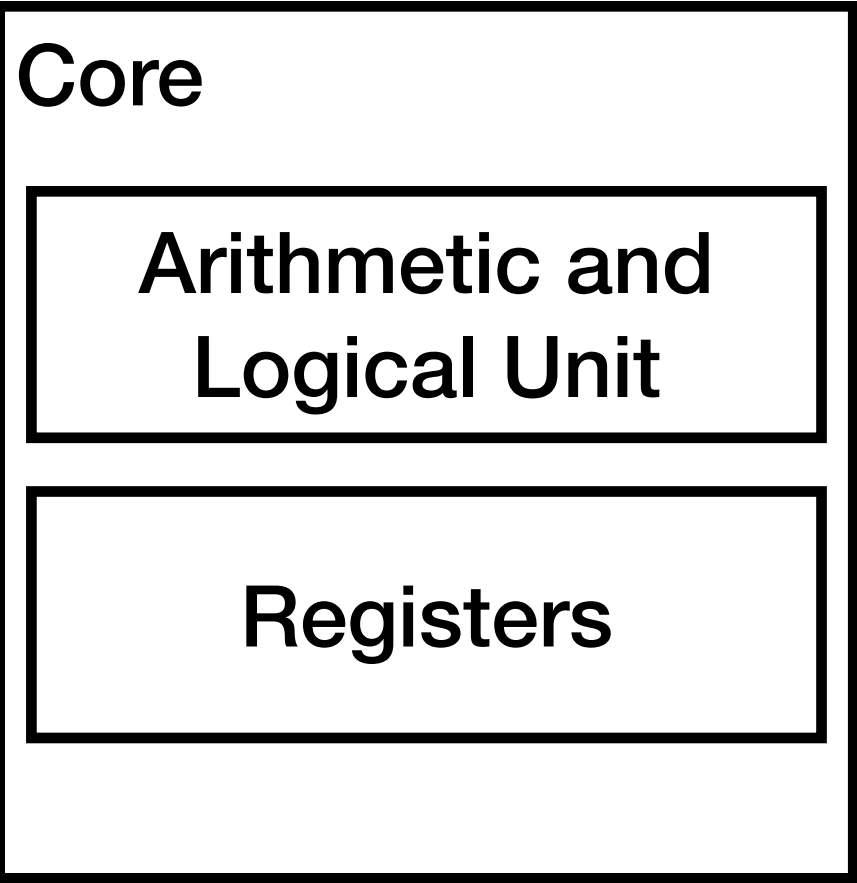
```
void *count(void *arg)
{
    (void) arg;

    for (unsigned int i = 0; i < N; i++) {
        cnt++;
    }

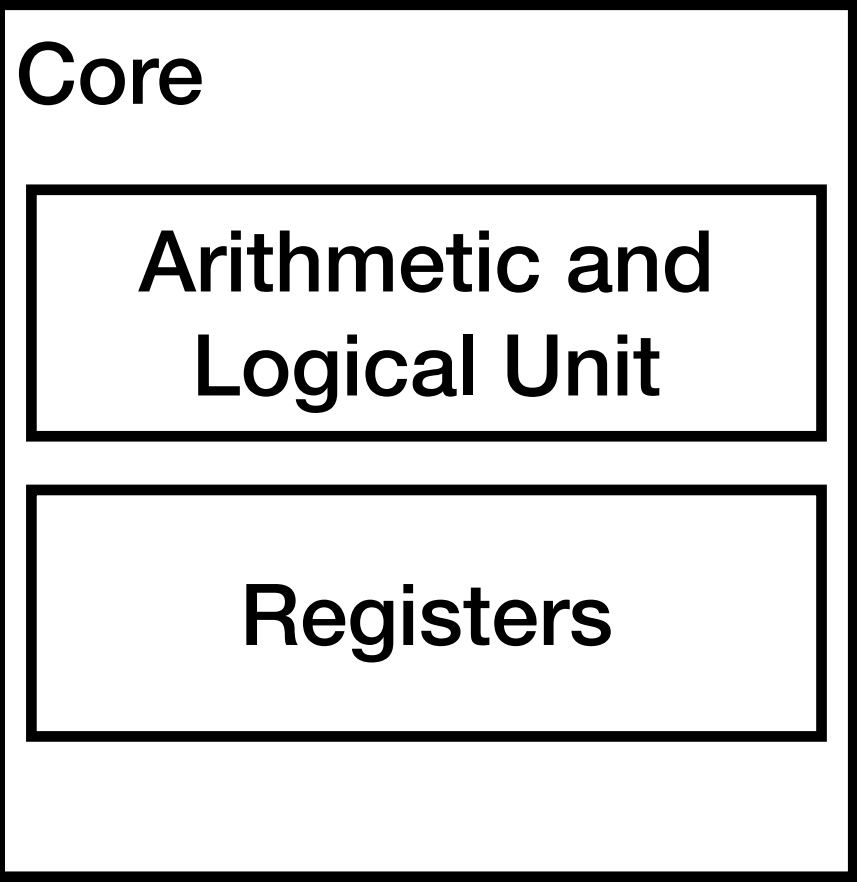
    return NULL;
}
```

Thread (cont.)

badcnt.c



0



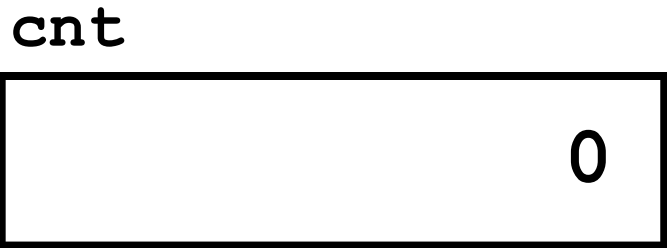
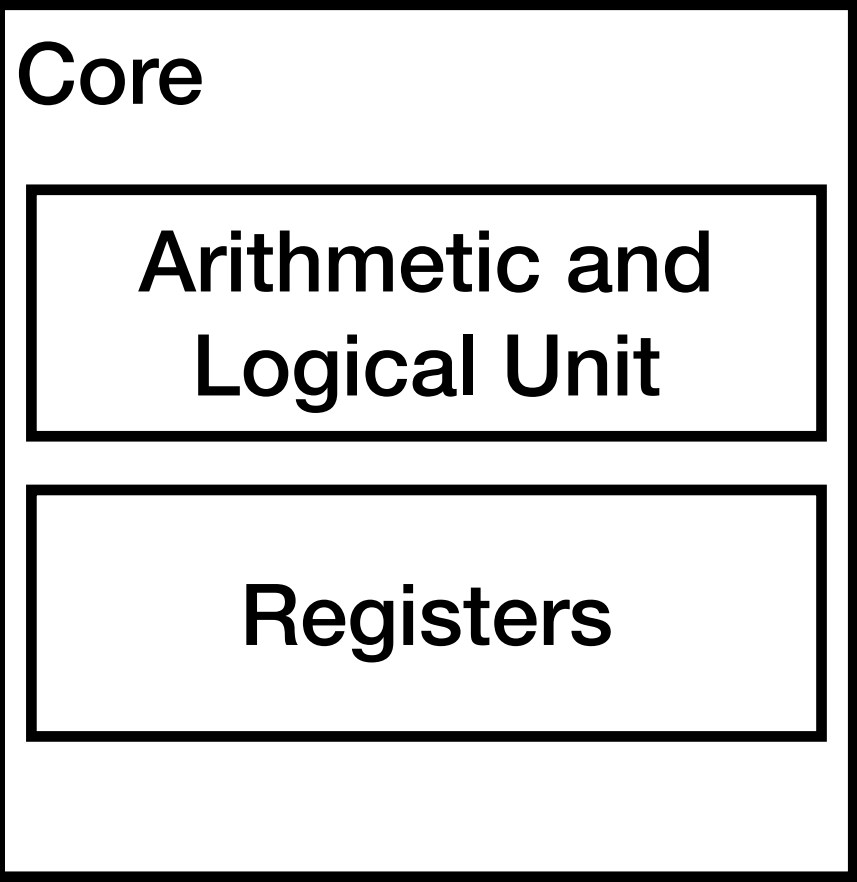
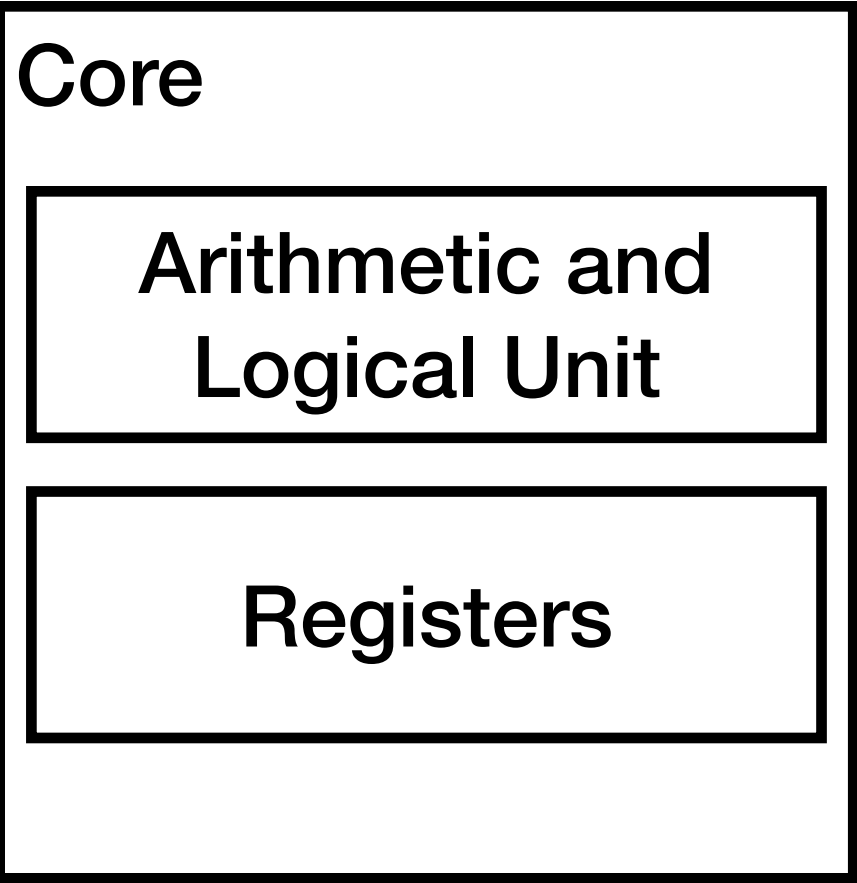
```
void *count(void *arg)
{
    (void) arg;

    for (unsigned int i = 0; i < N; i++) {
        cnt++;
    }

    return NULL;
}
```

Thread (cont.)

badcnt.c



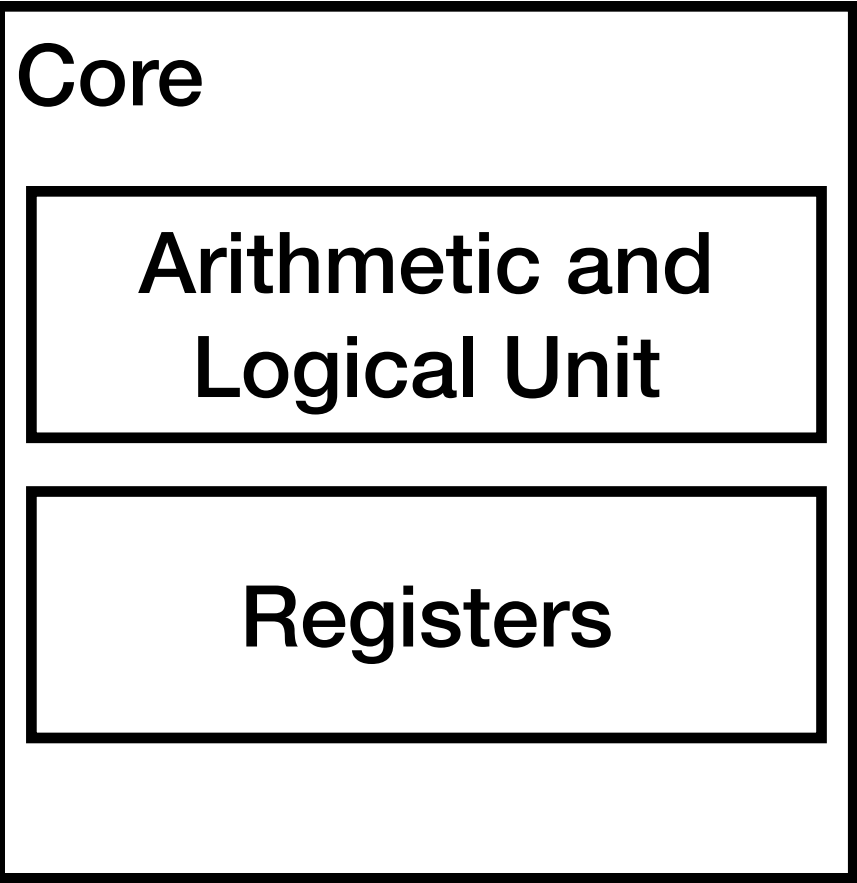
```
void *count(void *arg)
{
    (void) arg;

    for (unsigned int i = 0; i < N; i++) {
        cnt++;
    }

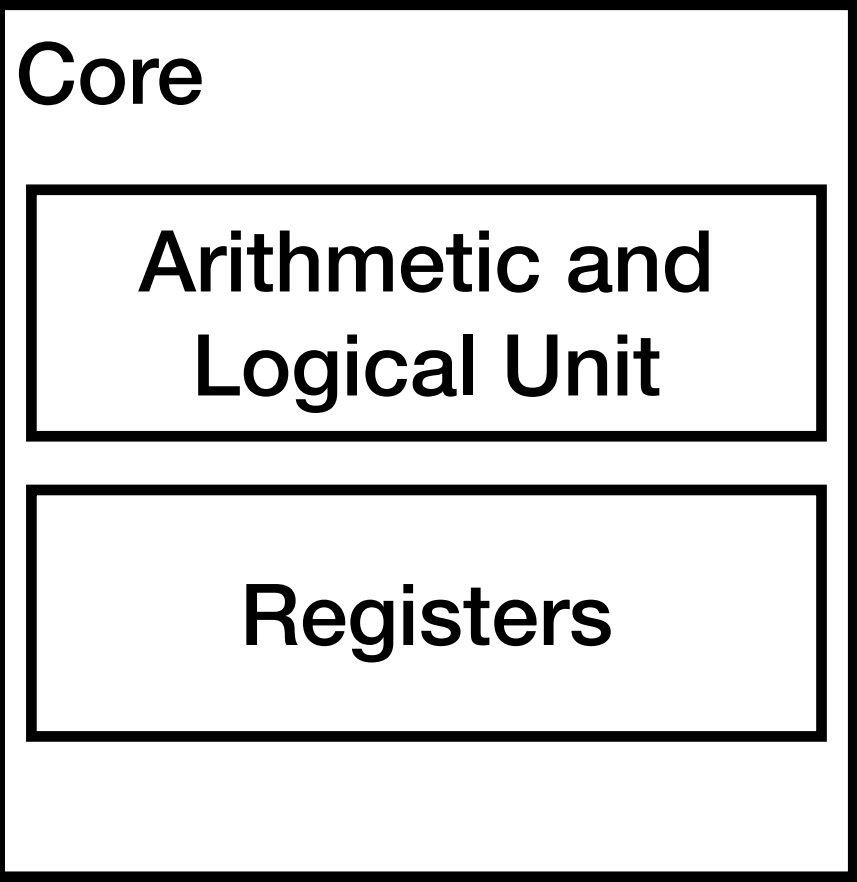
    return NULL;
}
```

Thread (cont.)

badcnt.c



1



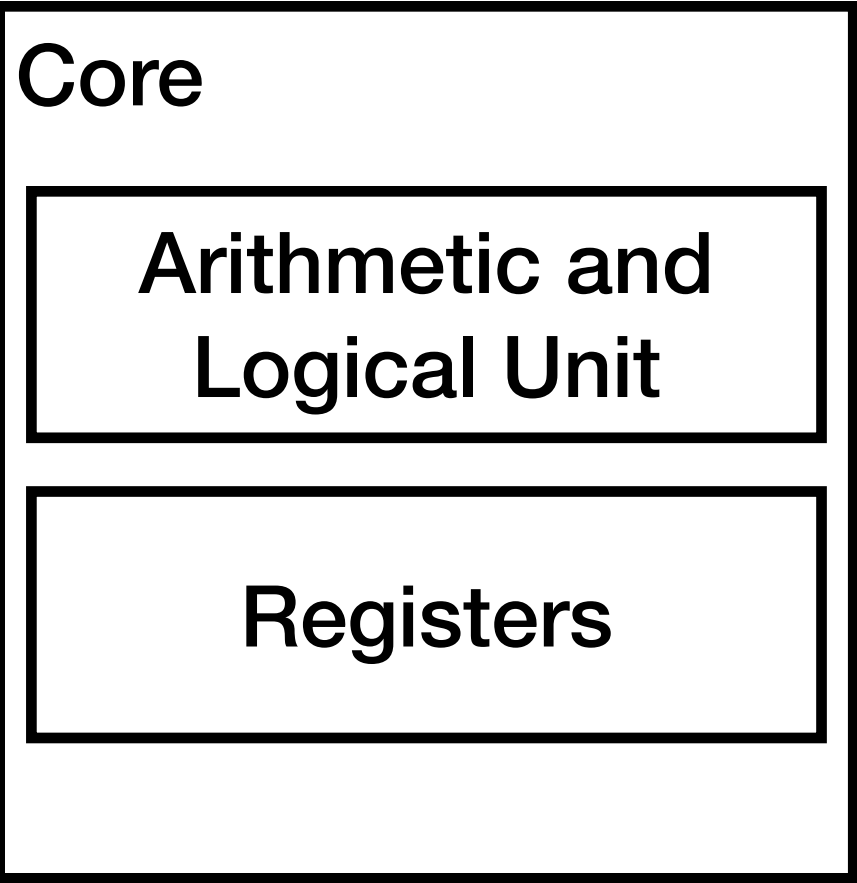
```
void *count(void *arg)
{
    (void) arg;

    for (unsigned int i = 0; i < N; i++) {
        cnt++;
    }

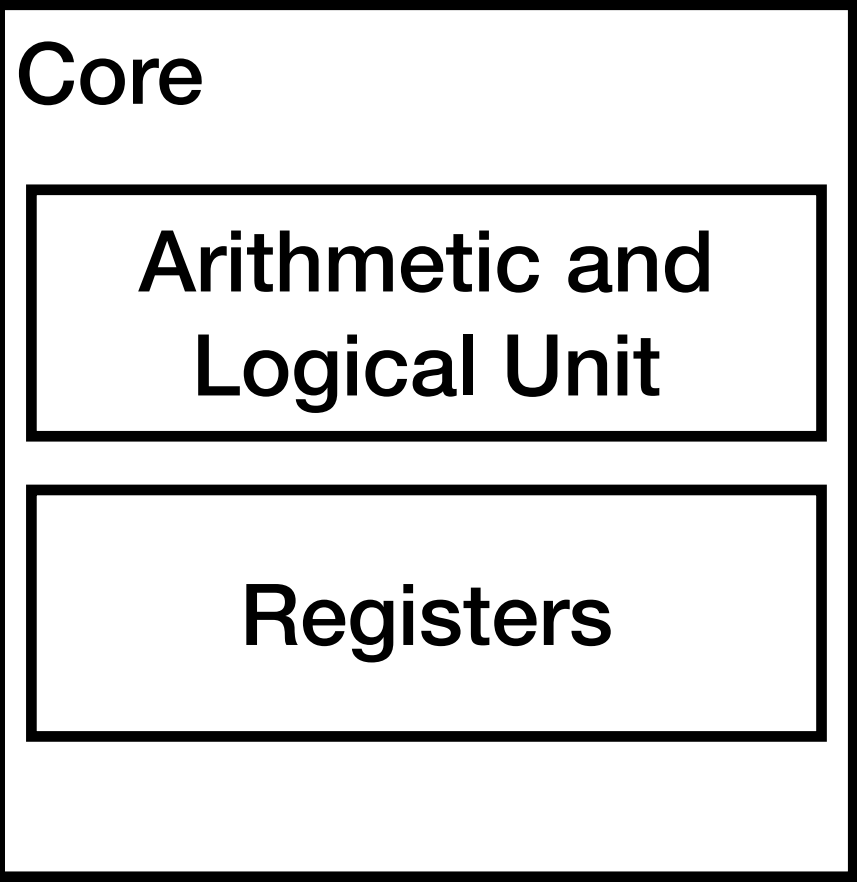
    return NULL;
}
```


Thread (cont.)

badcnt.c



1



1



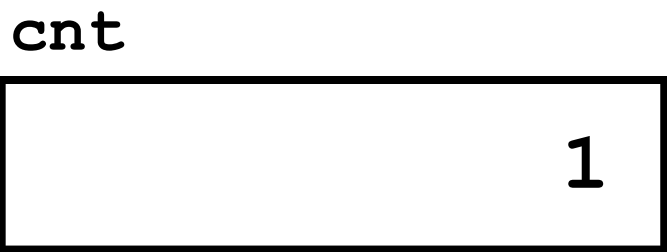
```
void *count(void *arg)
{
    (void) arg;

    for (unsigned int i = 0; i < N; i++) {
        cnt++;
    }

    return NULL;
}
```

Thread (cont.)

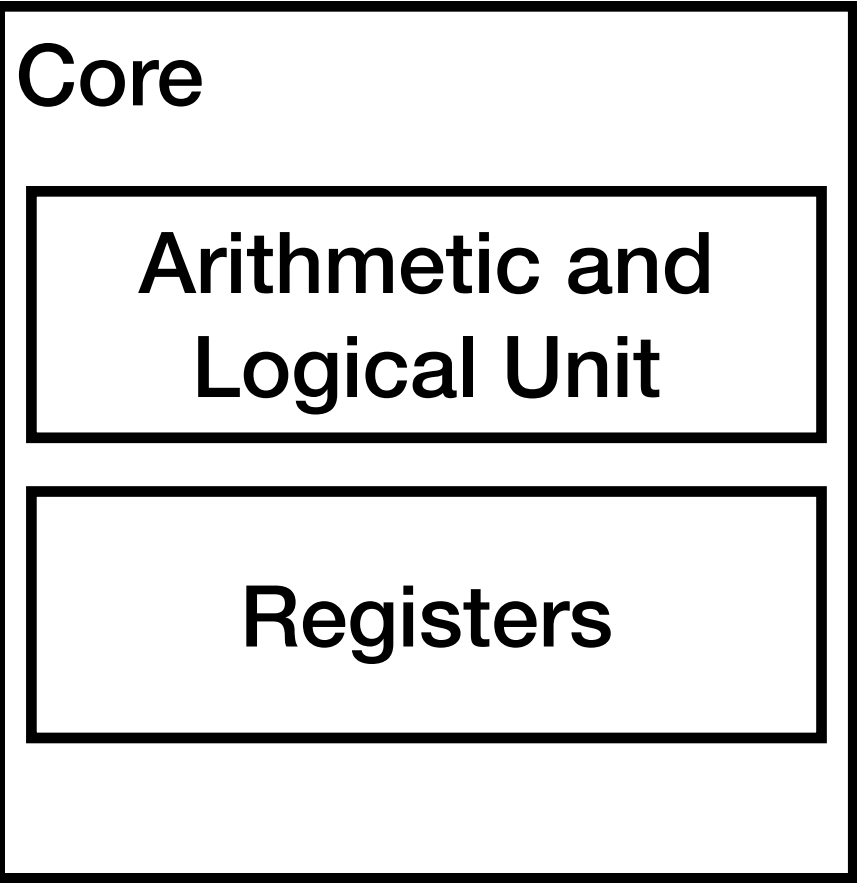
badcnt.c



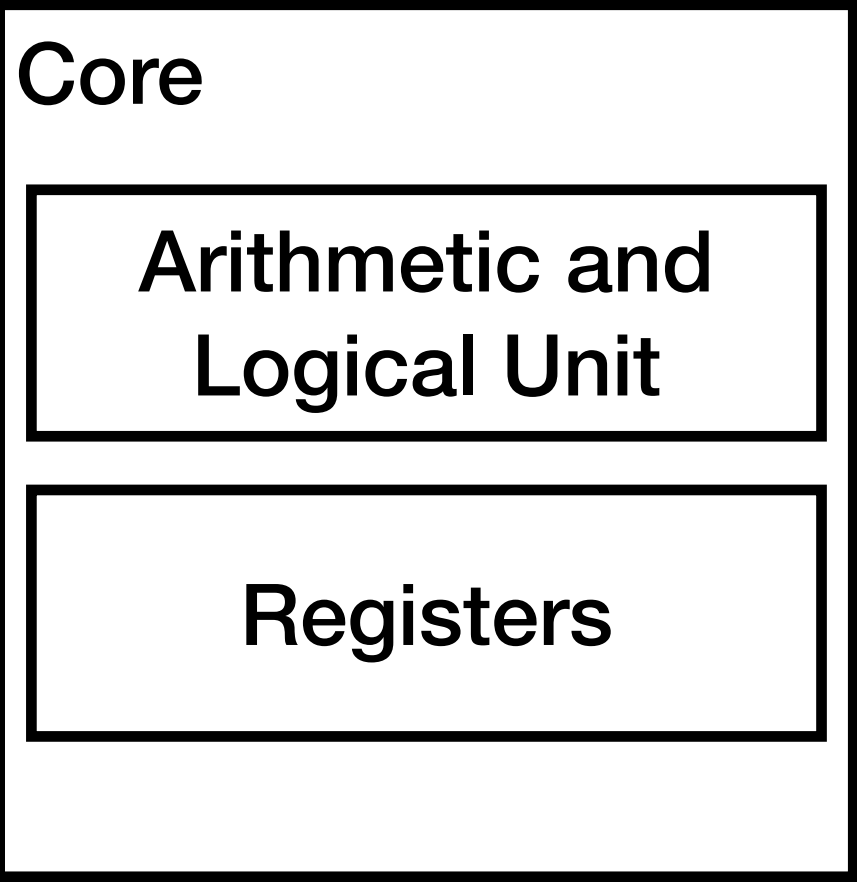
```
void *count(void *arg)
{
    (void) arg;

    for (unsigned int i = 0; i < N; i++) {
        cnt++;
    }

    return NULL;
}
```



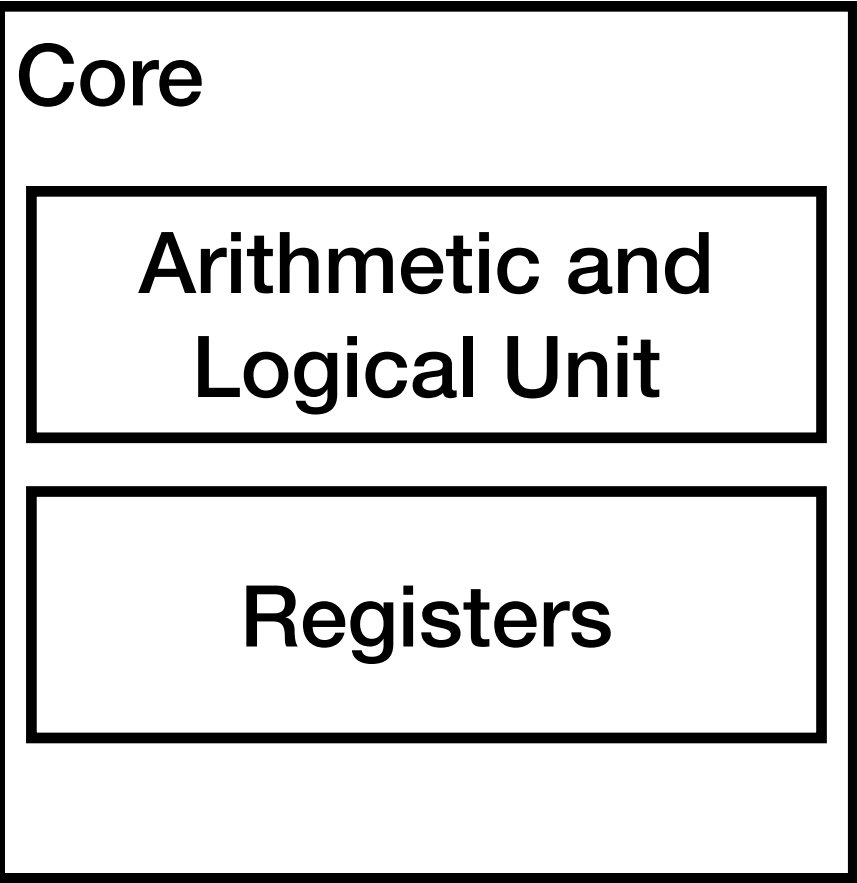
1



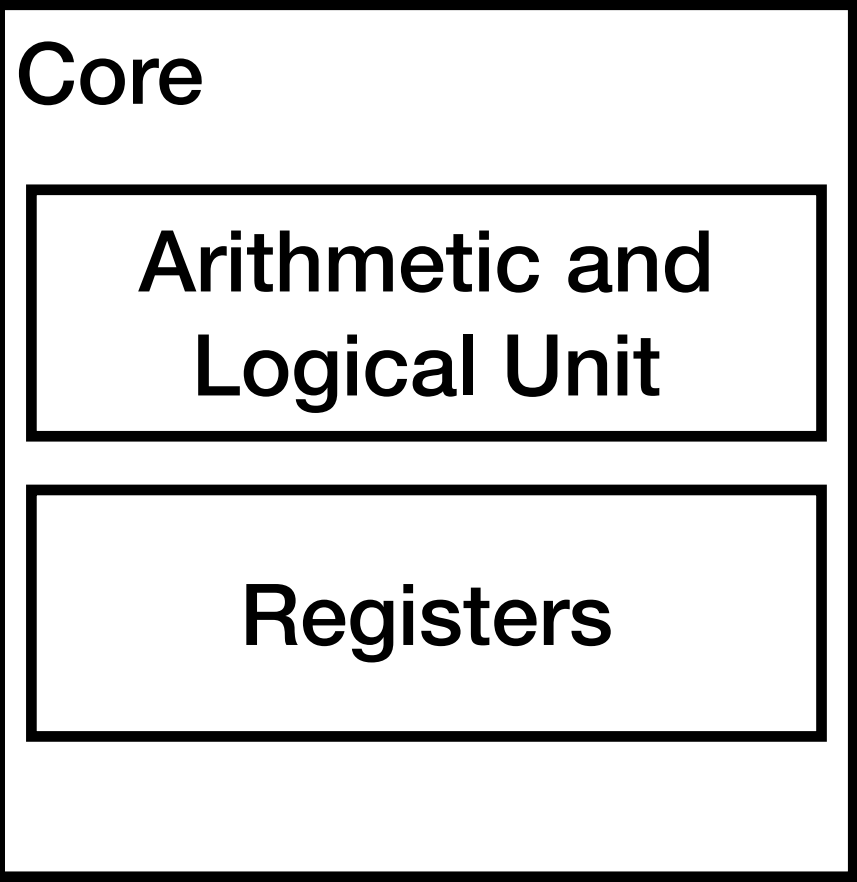
2

Thread (cont.)

badcnt.c



1



2



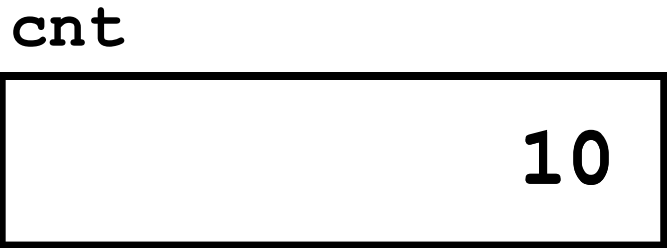
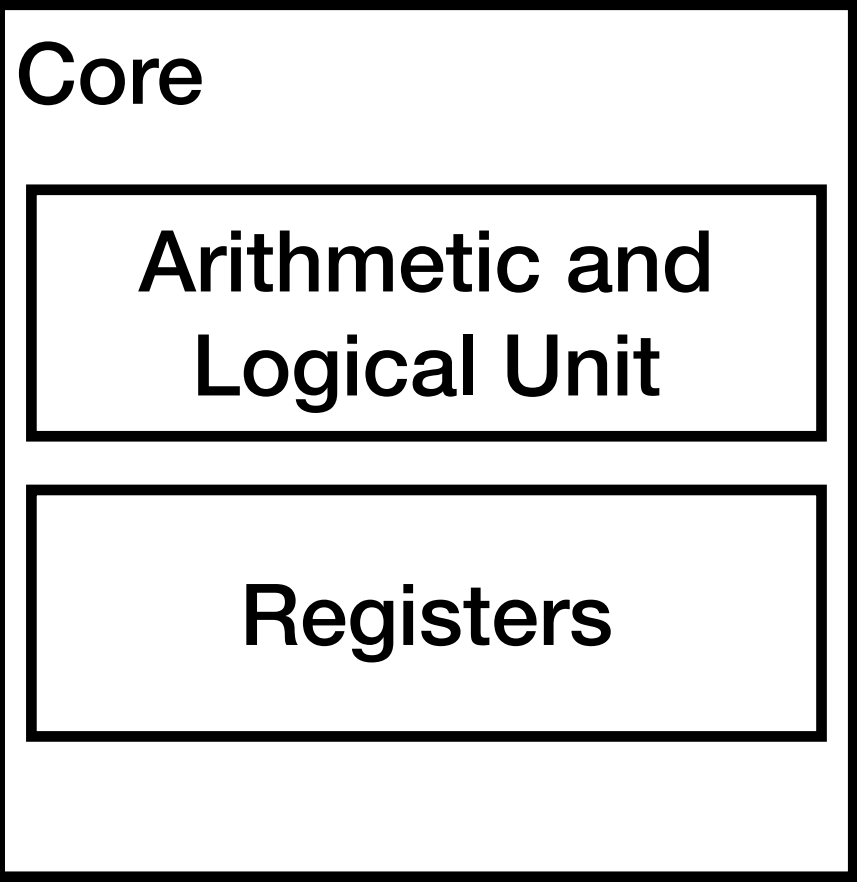
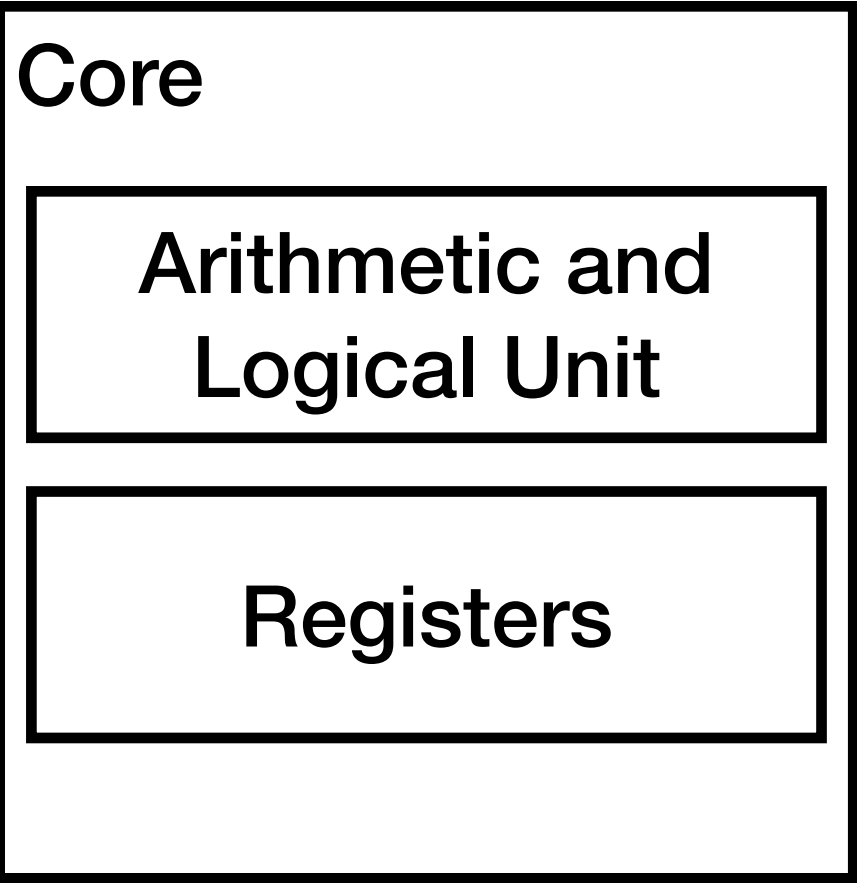
```
void *count(void *arg)
{
    (void) arg;

    for (unsigned int i = 0; i < N; i++) {
        cnt++;
    }

    return NULL;
}
```

Thread (cont.)

badcnt.c



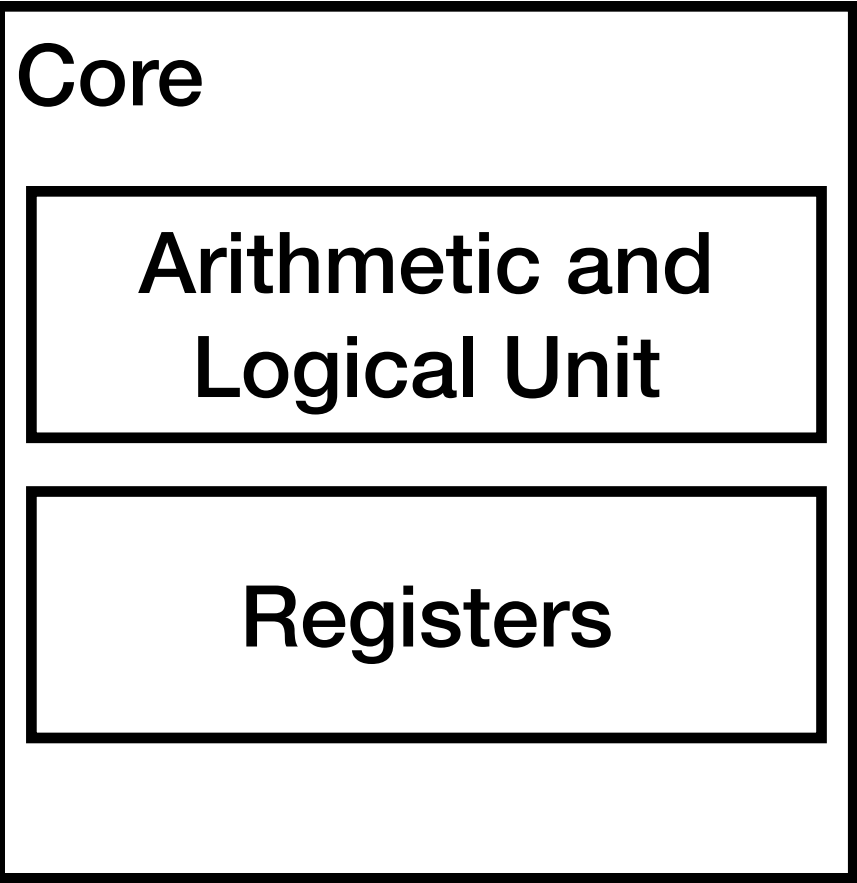
```
void *count(void *arg)
{
    (void) arg;

    for (unsigned int i = 0; i < N; i++) {
        cnt++;
    }

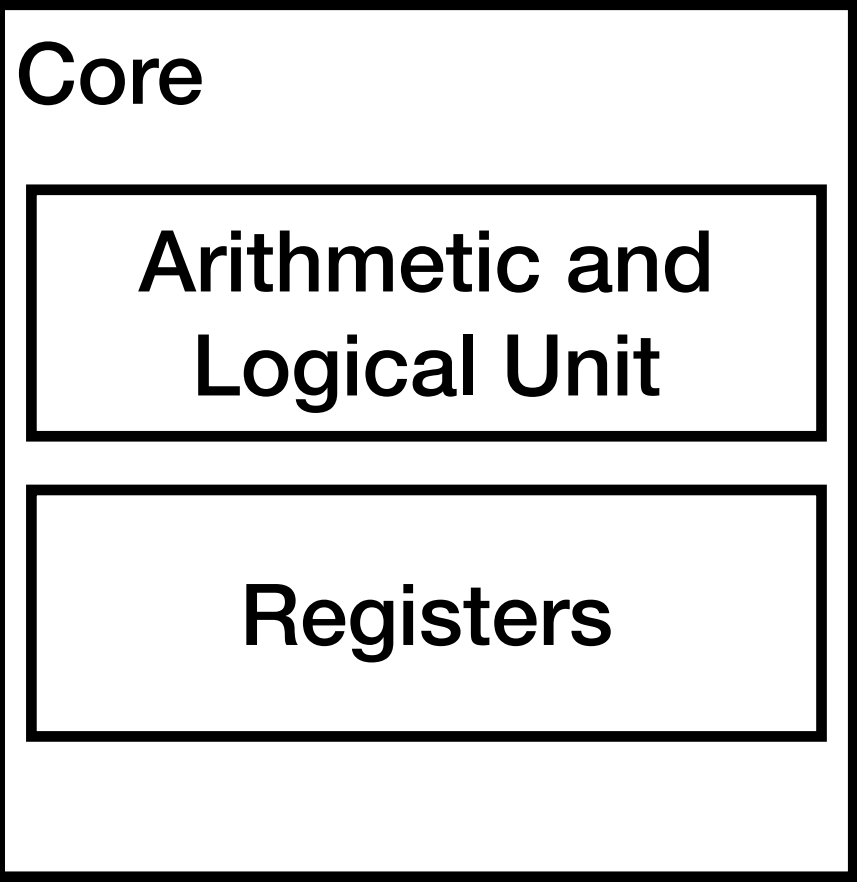
    return NULL;
}
```

Thread (cont.)

badcnt.c



10



10



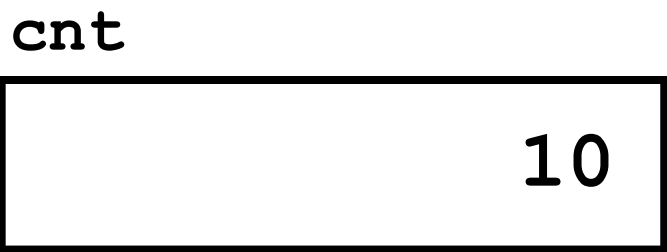
```
void *count(void *arg)
{
    (void) arg;

    for (unsigned int i = 0; i < N; i++) {
        cnt++;
    }

    return NULL;
}
```

Thread (cont.)

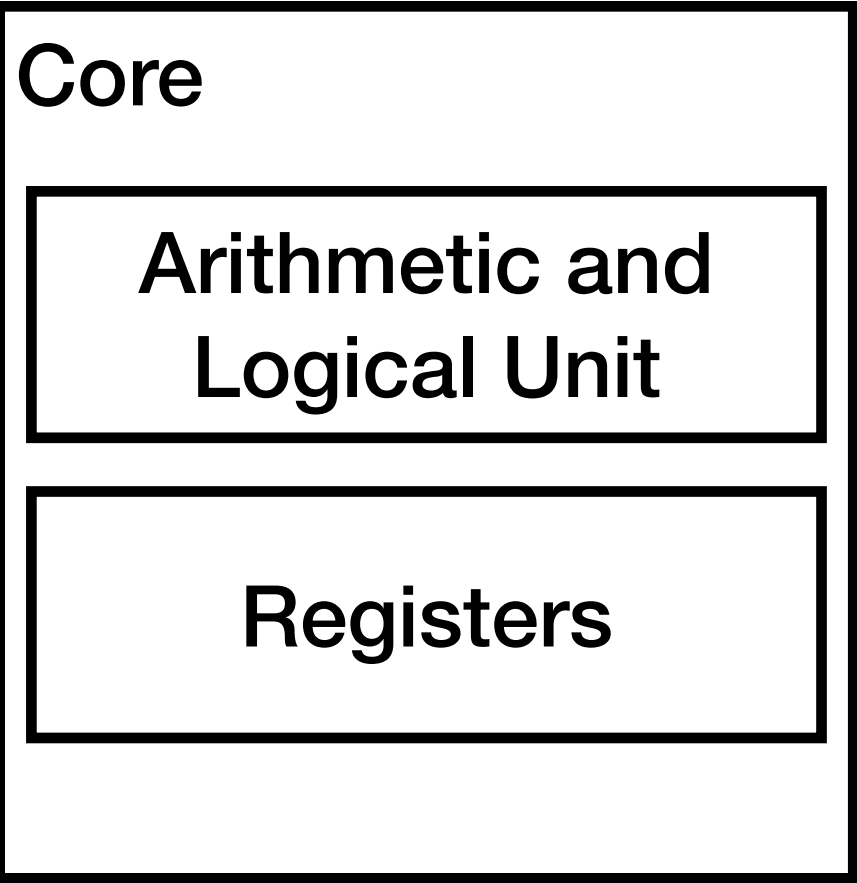
badcnt.c



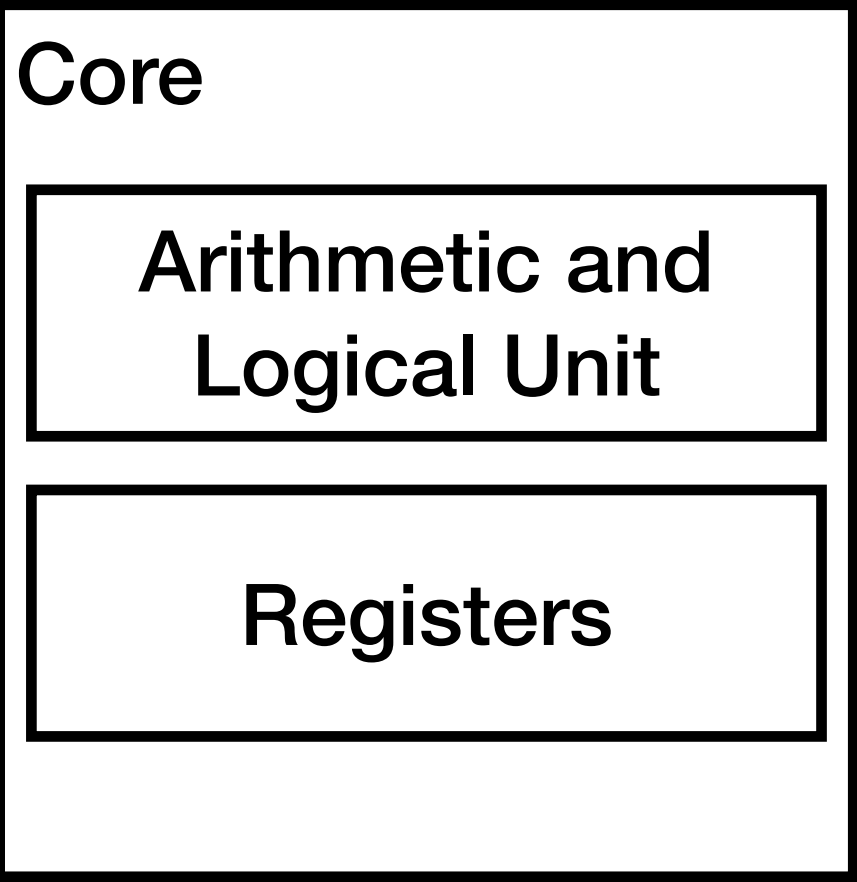
```
void *count(void *arg)
{
    (void) arg;

    for (unsigned int i = 0; i < N; i++) {
        cnt++;
    }

    return NULL;
}
```



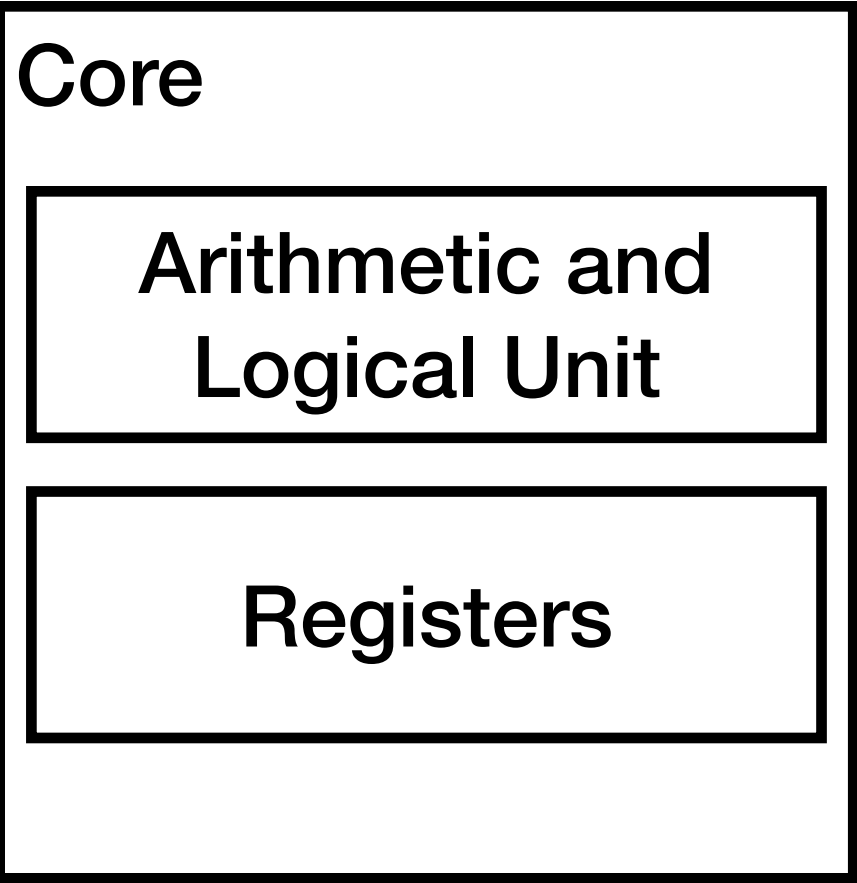
11



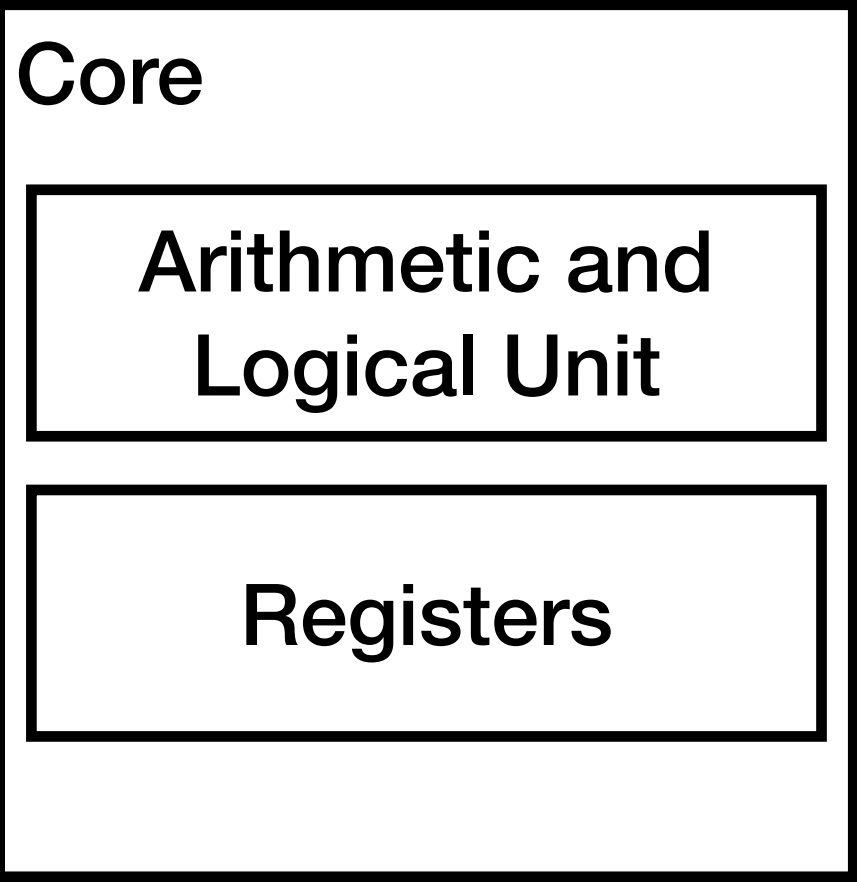
11

Thread (cont.)

badcnt.c



11



11



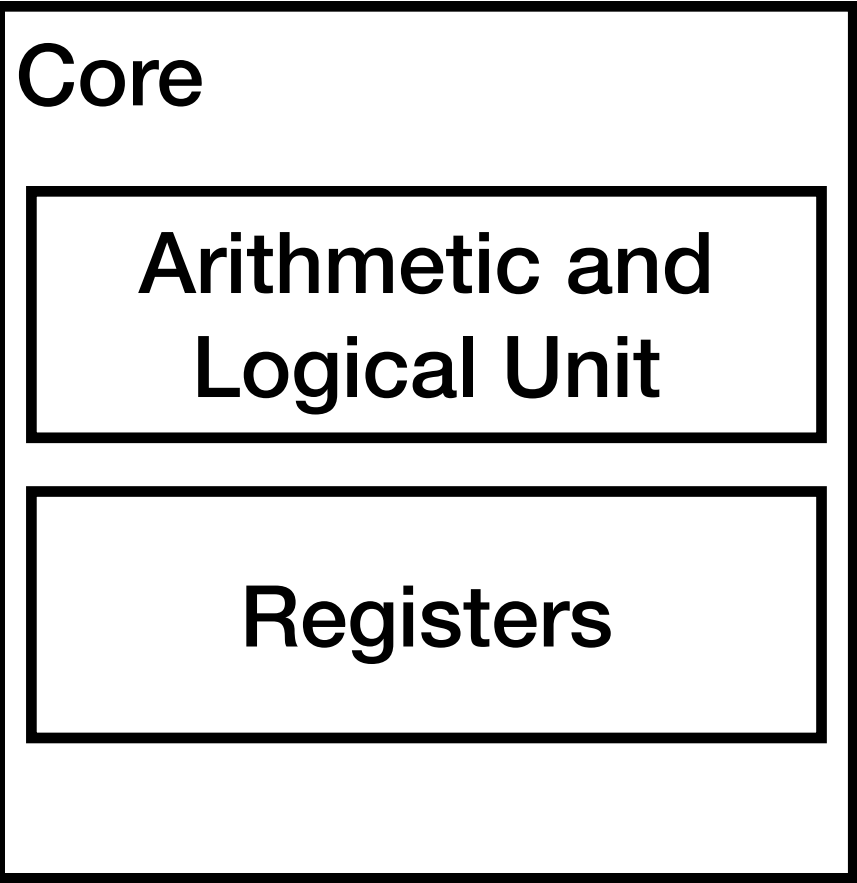
```
void *count(void *arg)
{
    (void) arg;

    for (unsigned int i = 0; i < N; i++) {
        cnt++;
    }

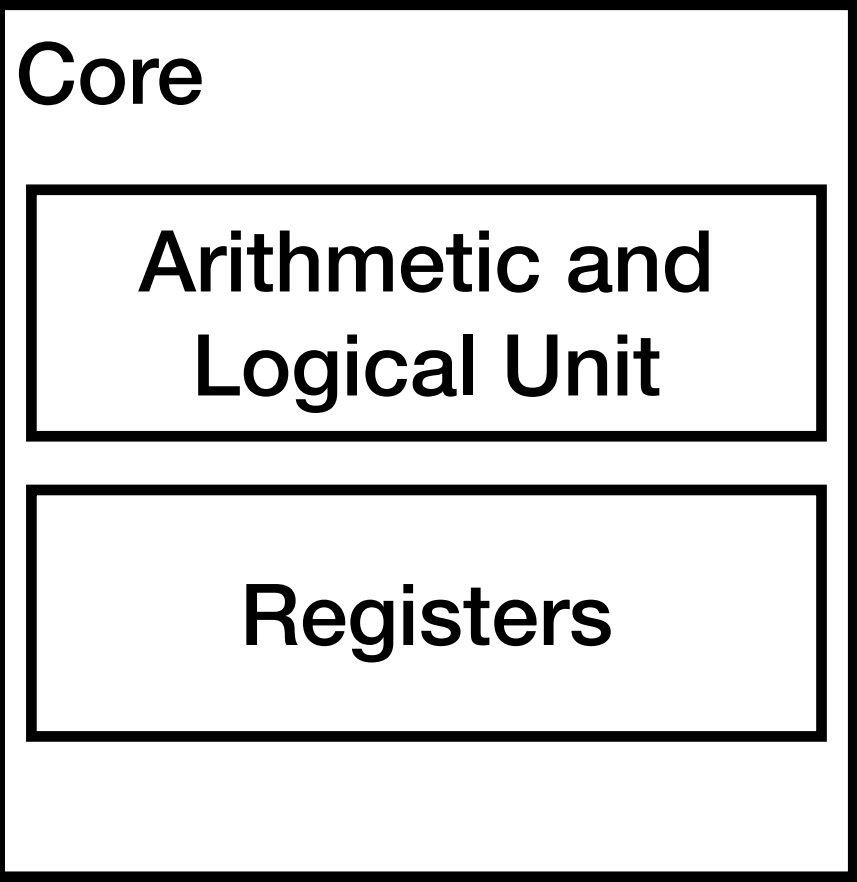
    return NULL;
}
```

Thread (cont.)

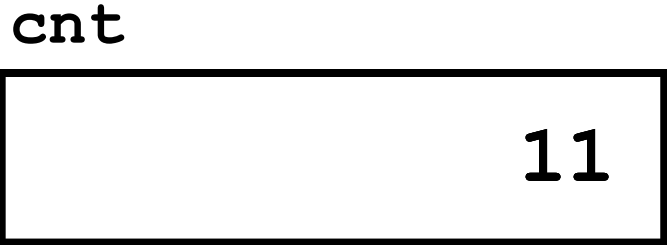
goodcnt.c



11



11



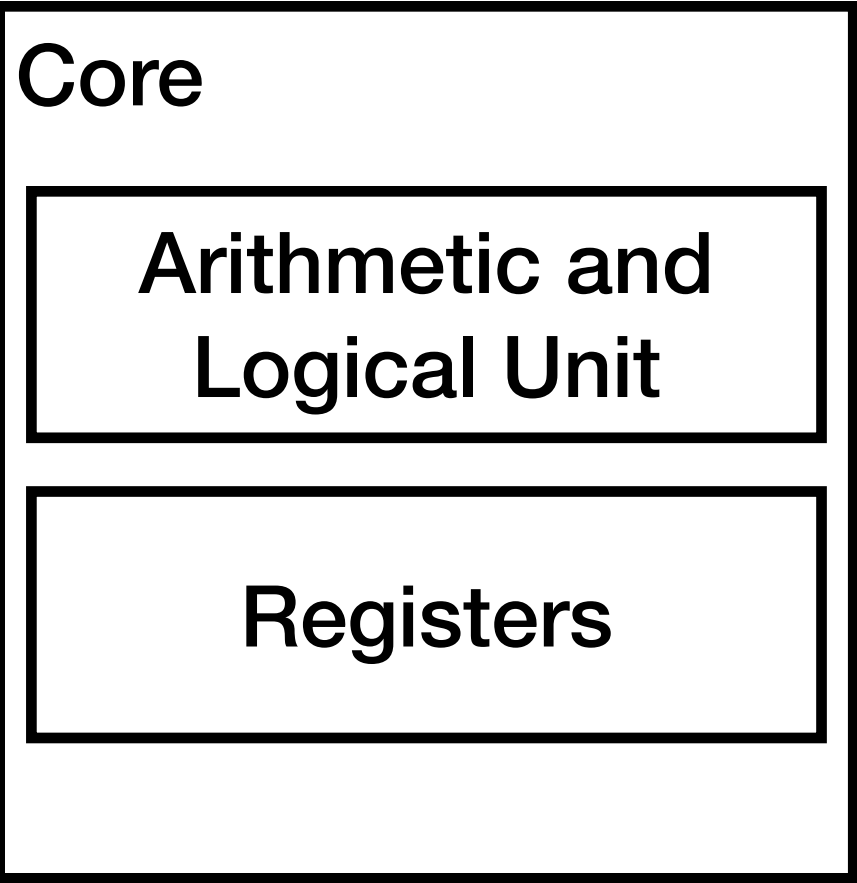
```
void *count(void *arg)
{
    (void) arg;

    for (unsigned int i = 0; i < N; i++) {
        pthread_mutex_lock(&sem);
        cnt++;
        pthread_mutex_unlock(&sem);
    }

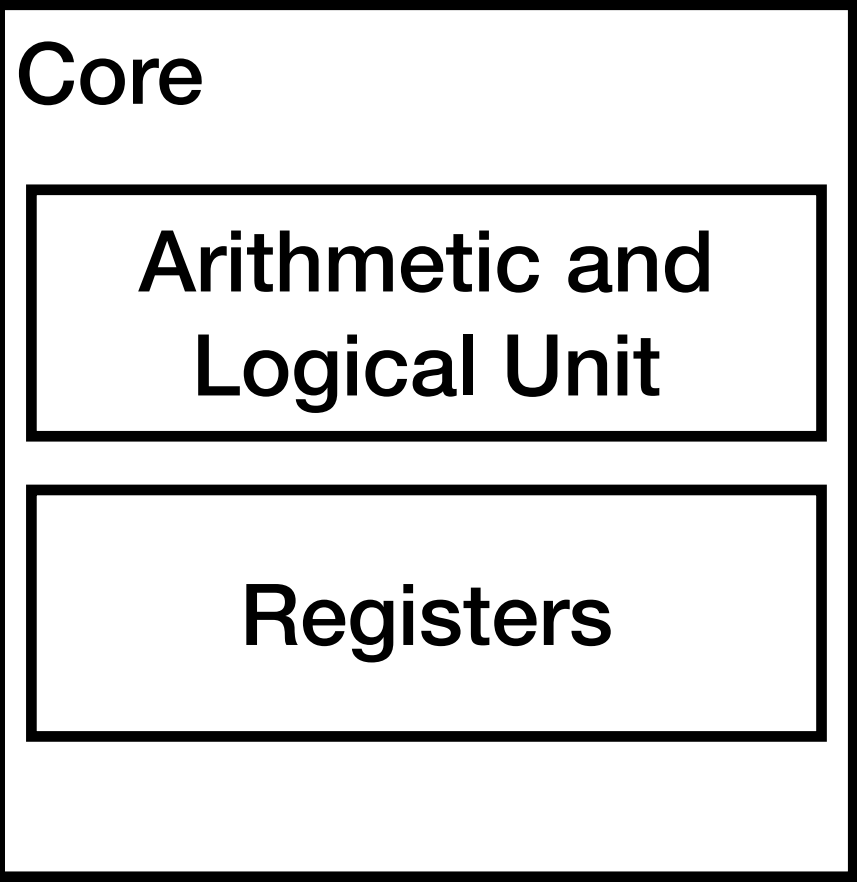
    return NULL;
}
```


Thread (cont.)

goodcnt.c



11



11



```
void *count(void *arg)
{
    (void) arg;

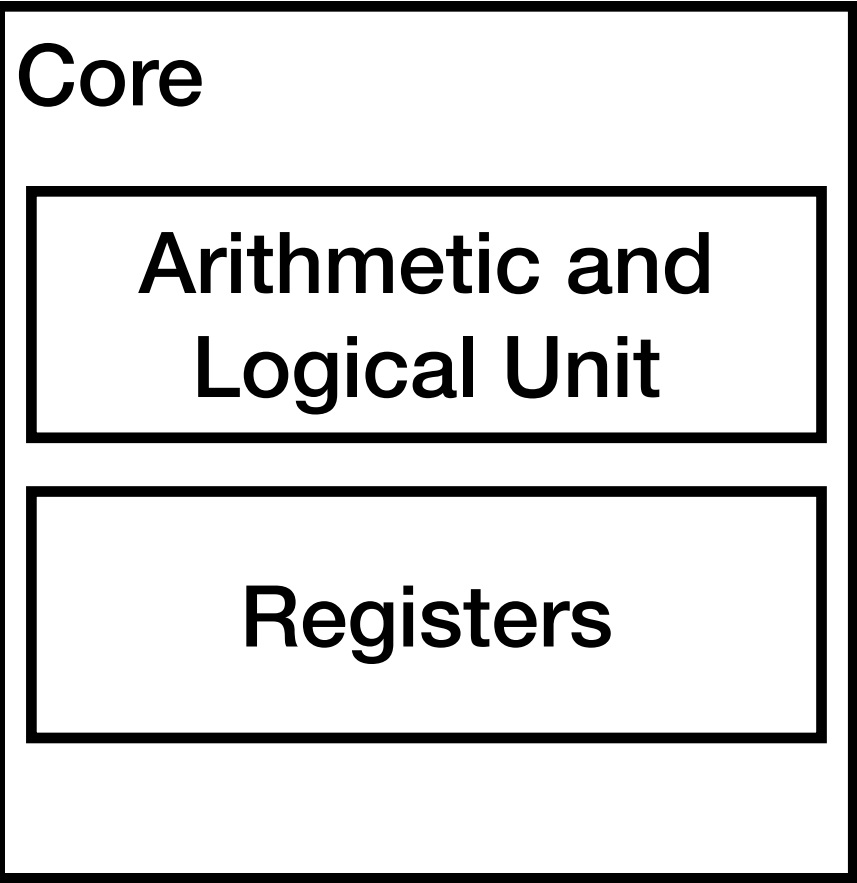
    for (unsigned int i = 0; i < N; i++) {
        pthread_mutex_lock(&sem);
        cnt++;
        pthread_mutex_unlock(&sem);
    }

    return NULL;
}
```

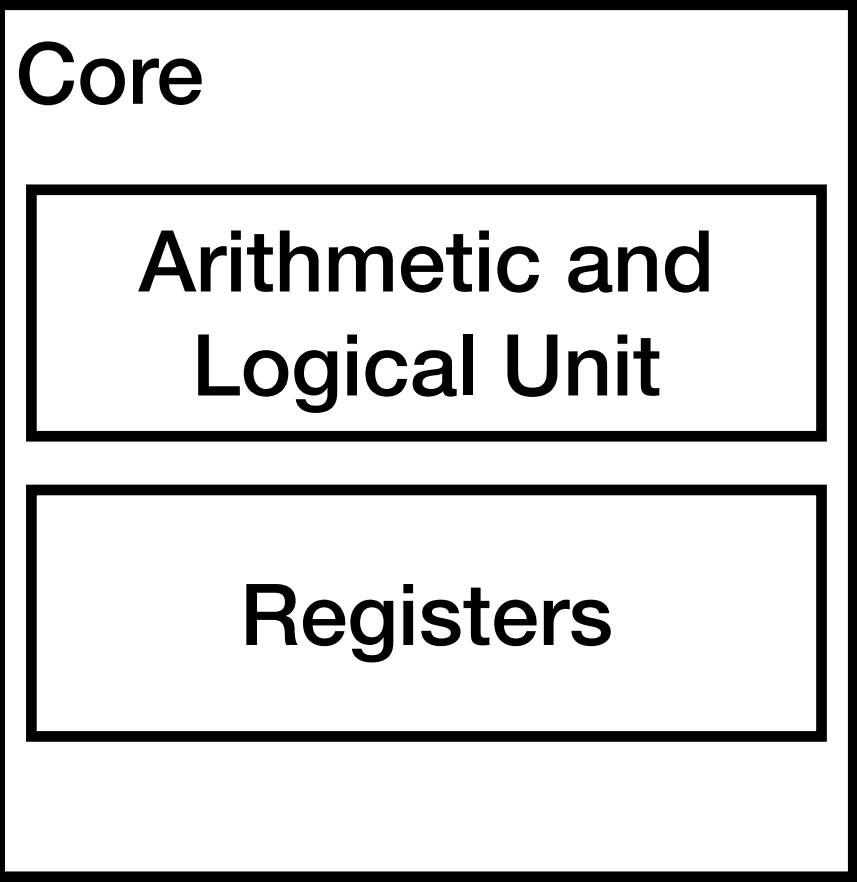
- pthread_mutex_lock "locks" cnt, if it is already locked, pthread_mutex_lock will wait

Thread (cont.)

goodcnt.c



11



11



```
void *count(void *arg)
{
    (void) arg;

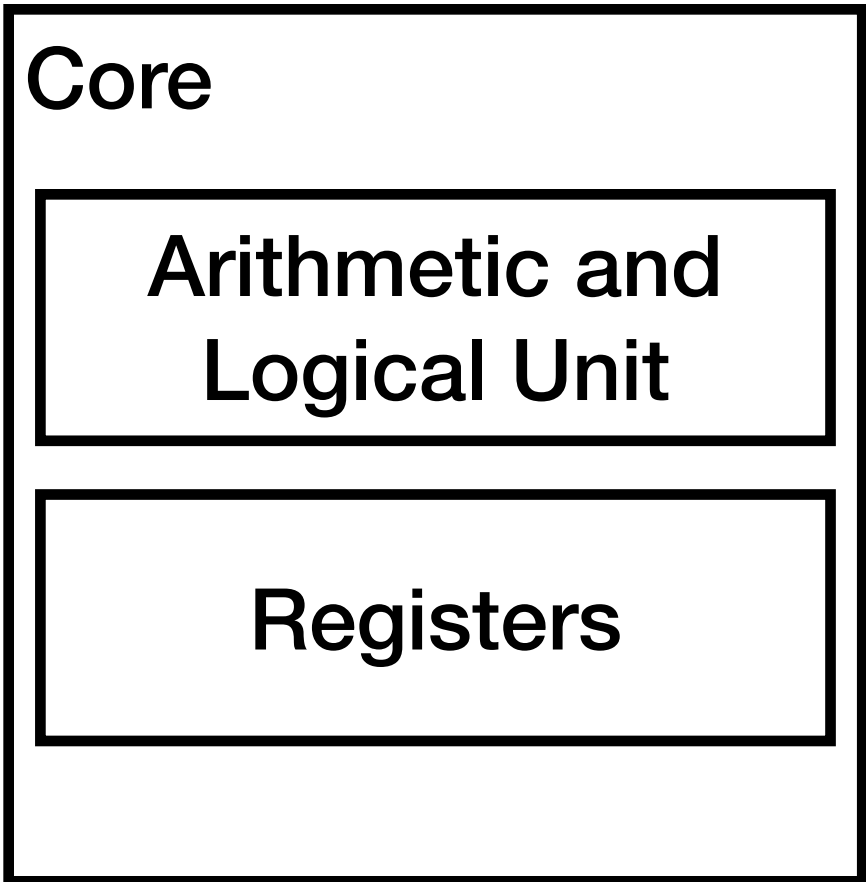
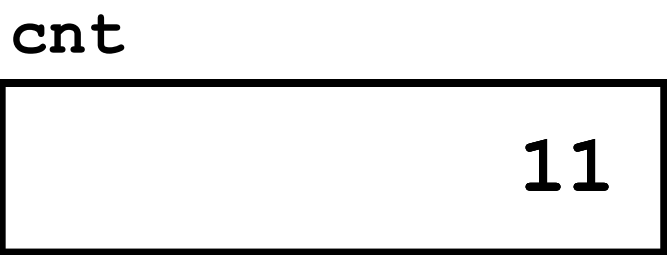
    for (unsigned int i = 0; i < N; i++) {
        pthread_mutex_lock(&sem);
        cnt++;
        pthread_mutex_unlock(&sem);
    }

    return NULL;
}
```

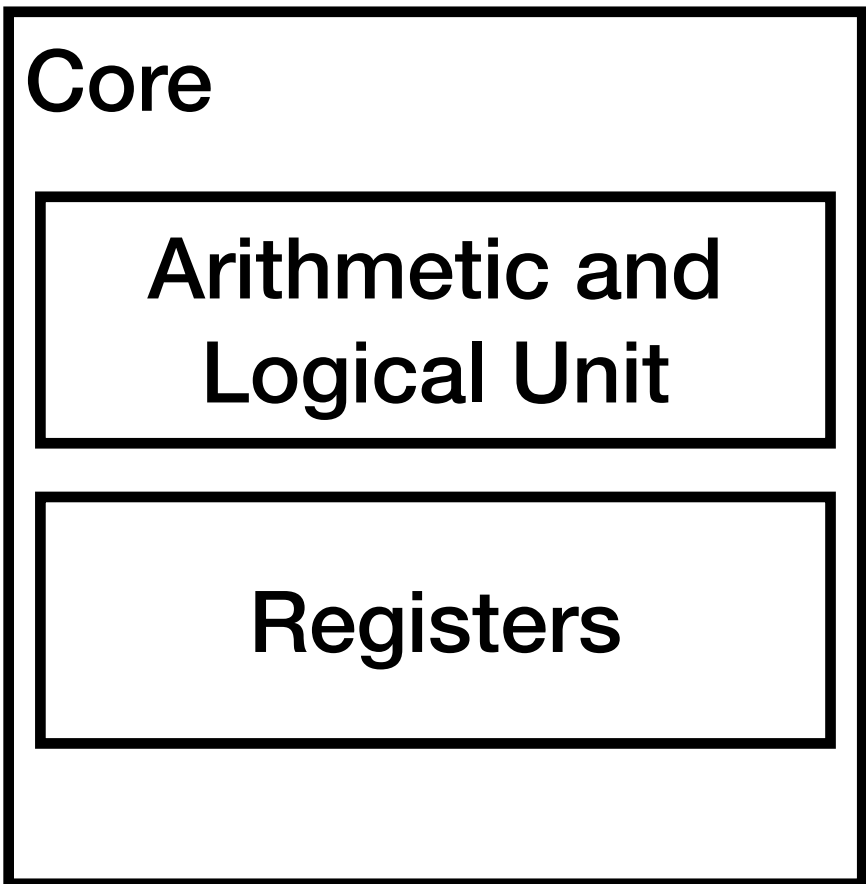
- pthread_mutex_lock "locks" cnt, if it is already locked, pthread_mutex_lock will wait
- pthread_mutex_unlock "unlocks" cnt

Thread (cont.)

goodcnt.c



11



11

```
void *count(void *arg)
{
    (void) arg;

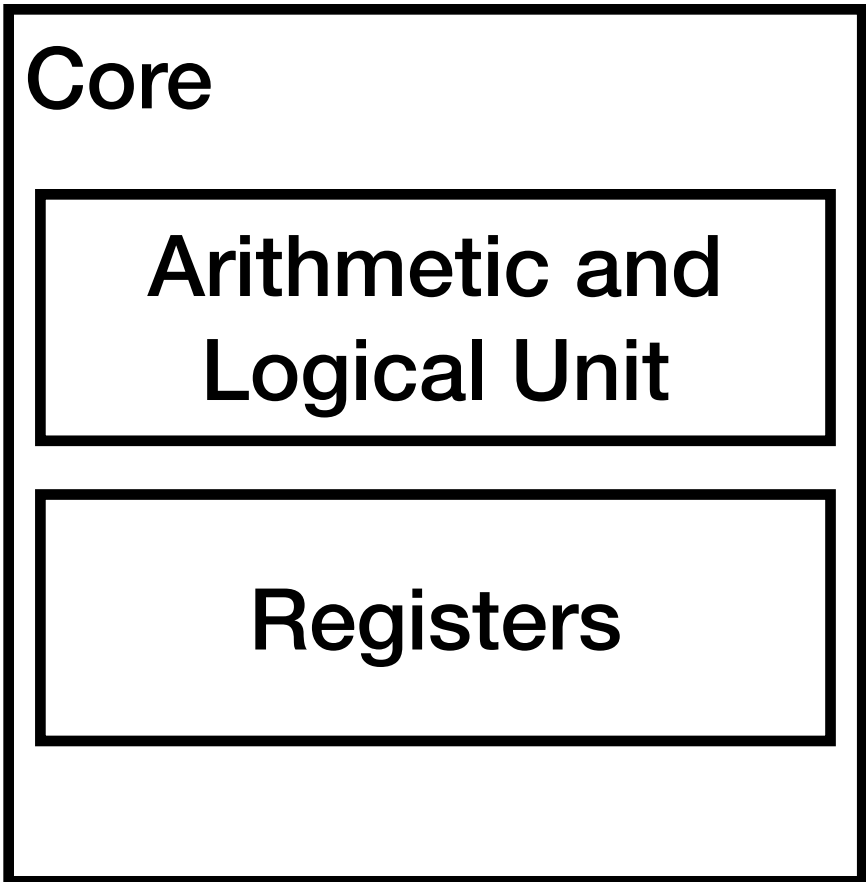
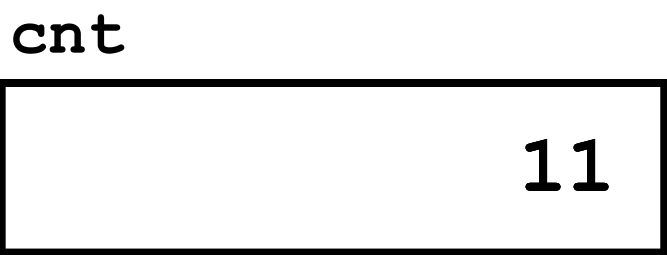
    for (unsigned int i = 0; i < N; i++) {
        pthread_mutex_lock(&sem);
        cnt++;
        pthread_mutex_unlock(&sem);
    }

    return NULL;
}
```

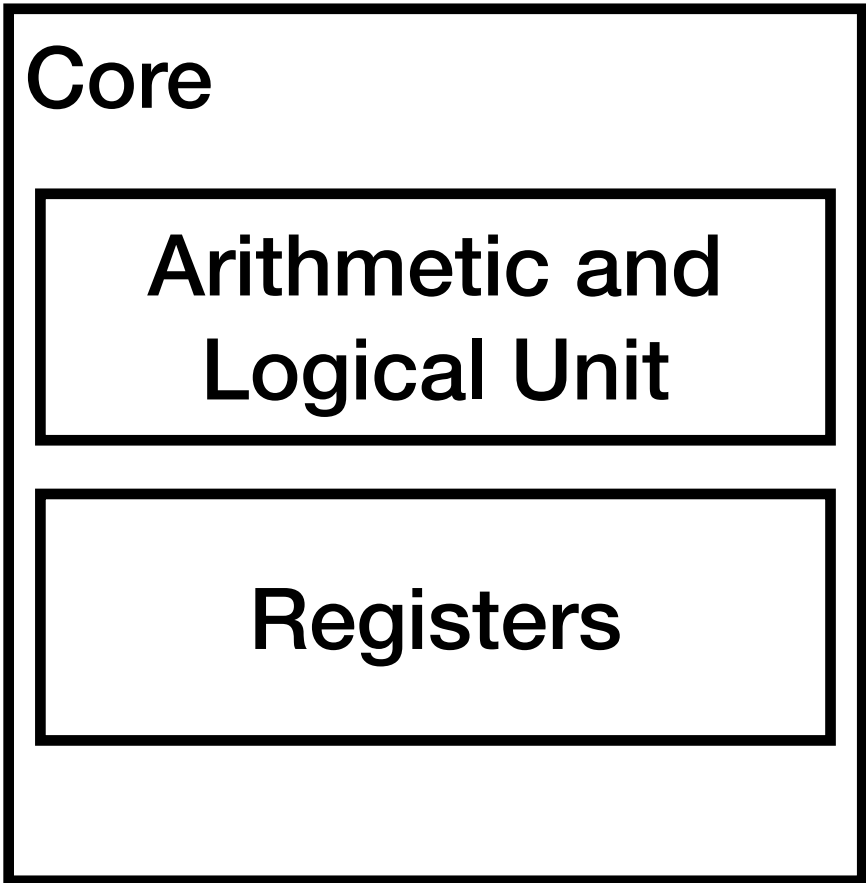
- pthread_mutex_lock "locks" cnt, if it is already locked, pthread_mutex_lock will wait
- pthread_mutex_unlock "unlocks" cnt
- Each loop now becomes

Thread (cont.)

goodcnt.c



11



11

```
void *count(void *arg)
{
    (void) arg;

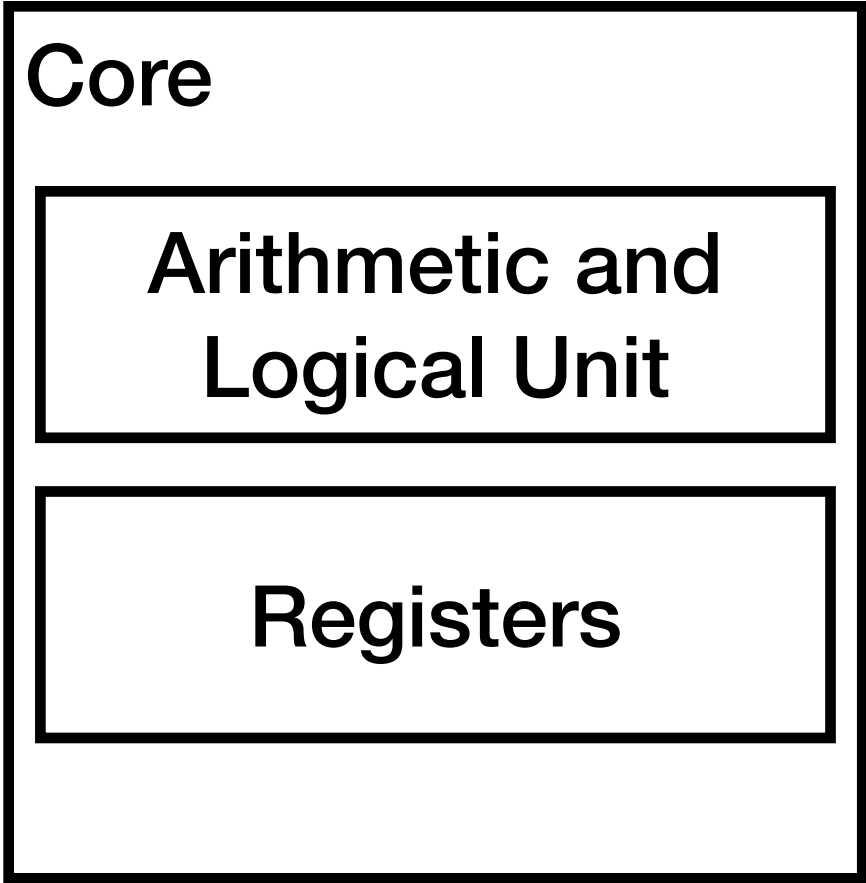
    for (unsigned int i = 0; i < N; i++) {
        pthread_mutex_lock(&sem);
        cnt++;
        pthread_mutex_unlock(&sem);
    }

    return NULL;
}
```

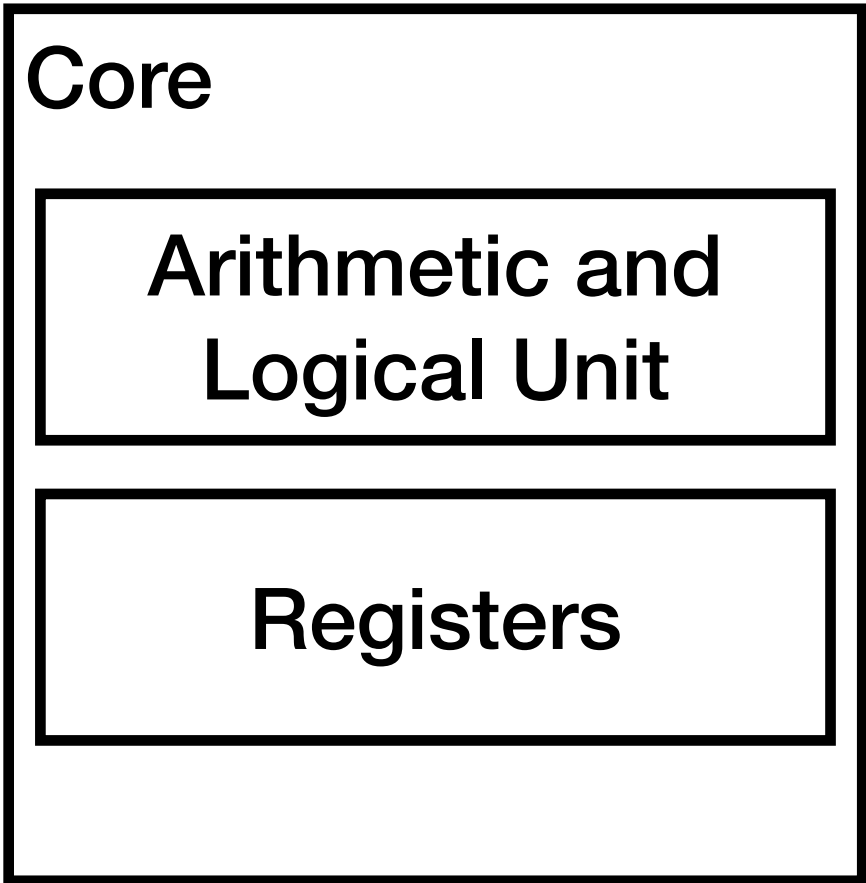
- pthread_mutex_lock "locks" cnt, if it is already locked, pthread_mutex_lock will wait
- pthread_mutex_unlock "unlocks" cnt
- Each loop now becomes
 - lock

Thread (cont.)

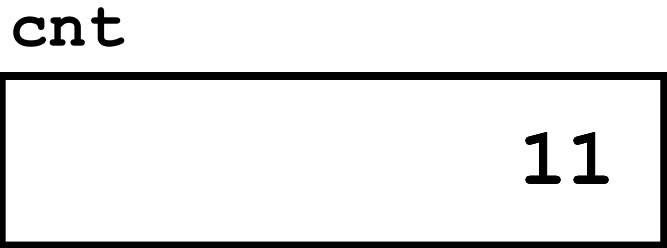
goodcnt.c



11



11



```
void *count(void *arg)
{
    (void) arg;

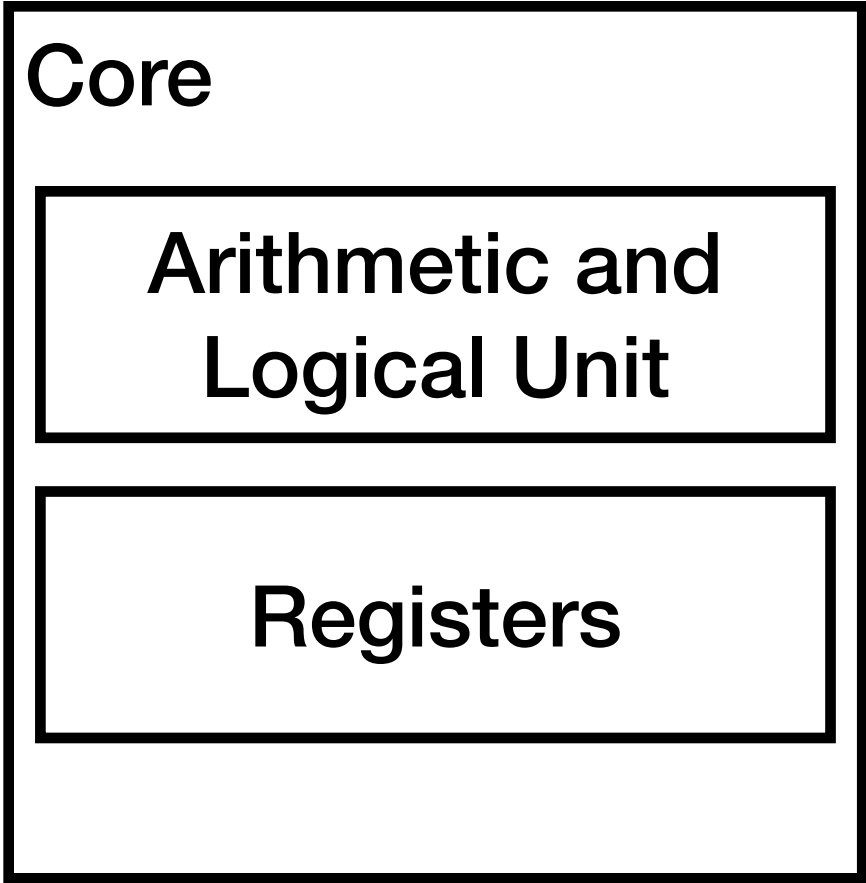
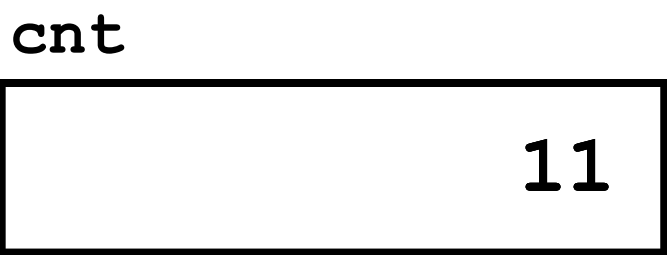
    for (unsigned int i = 0; i < N; i++) {
        pthread_mutex_lock(&sem);
        cnt++;
        pthread_mutex_unlock(&sem);
    }

    return NULL;
}
```

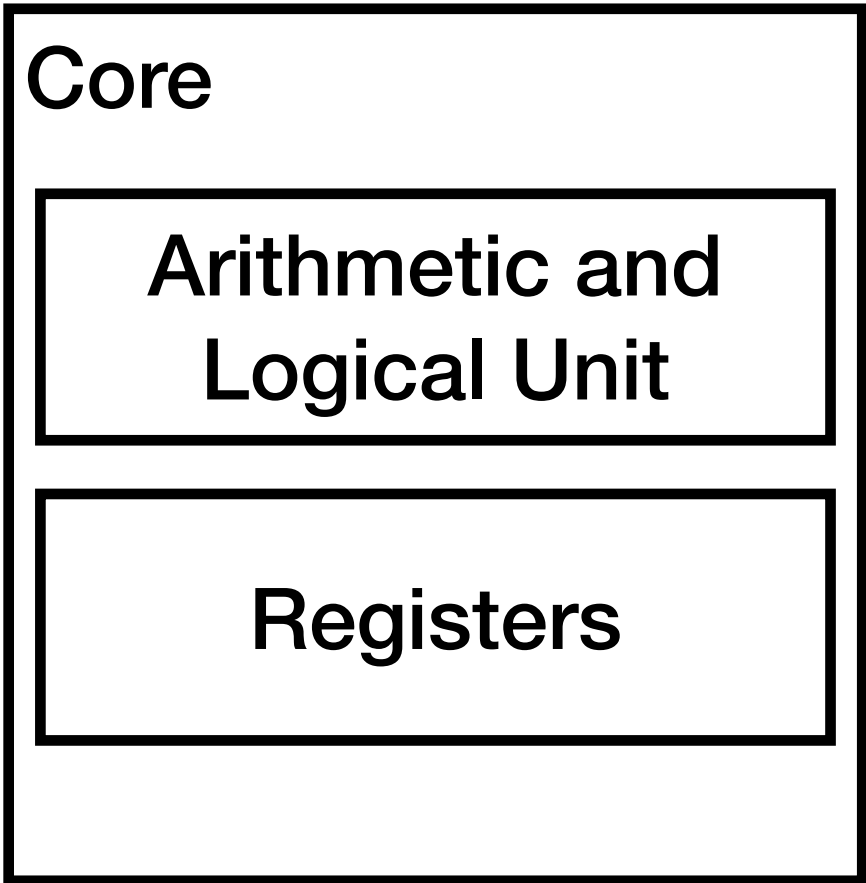
- pthread_mutex_lock "locks" cnt, if it is already locked, pthread_mutex_lock will wait
- pthread_mutex_unlock "unlocks" cnt
- Each loop now becomes
 - lock
 - read cnt

Thread (cont.)

goodcnt.c



11



11

```
void *count(void *arg)
{
    (void) arg;

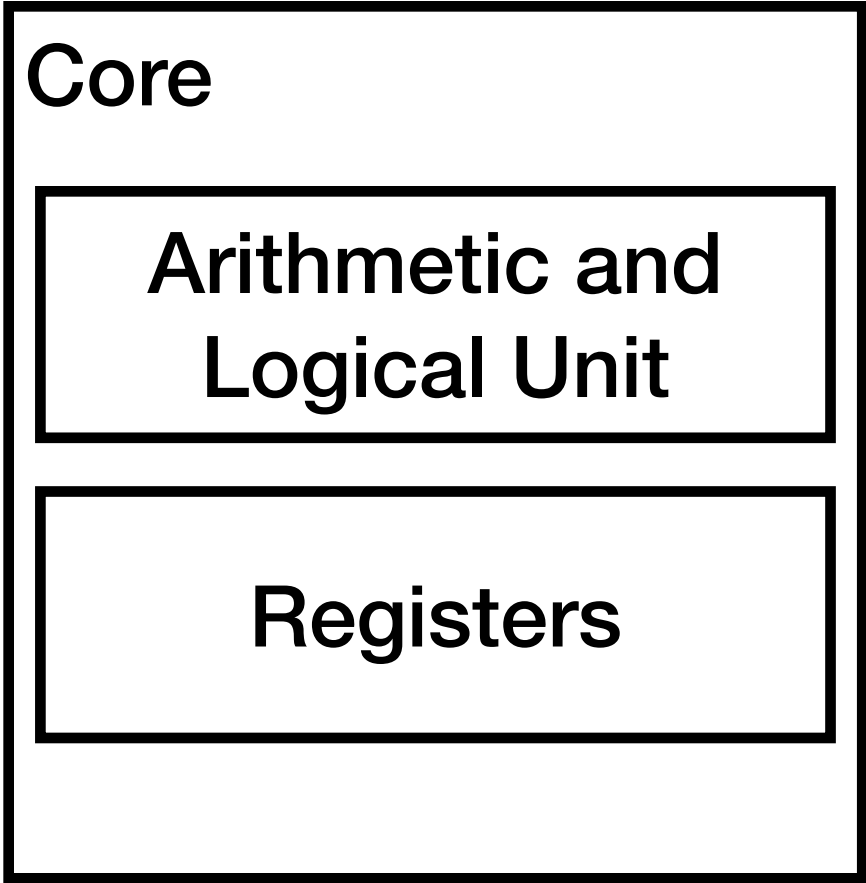
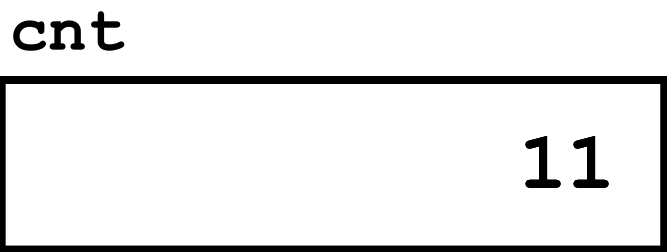
    for (unsigned int i = 0; i < N; i++) {
        pthread_mutex_lock(&sem);
        cnt++;
        pthread_mutex_unlock(&sem);
    }

    return NULL;
}
```

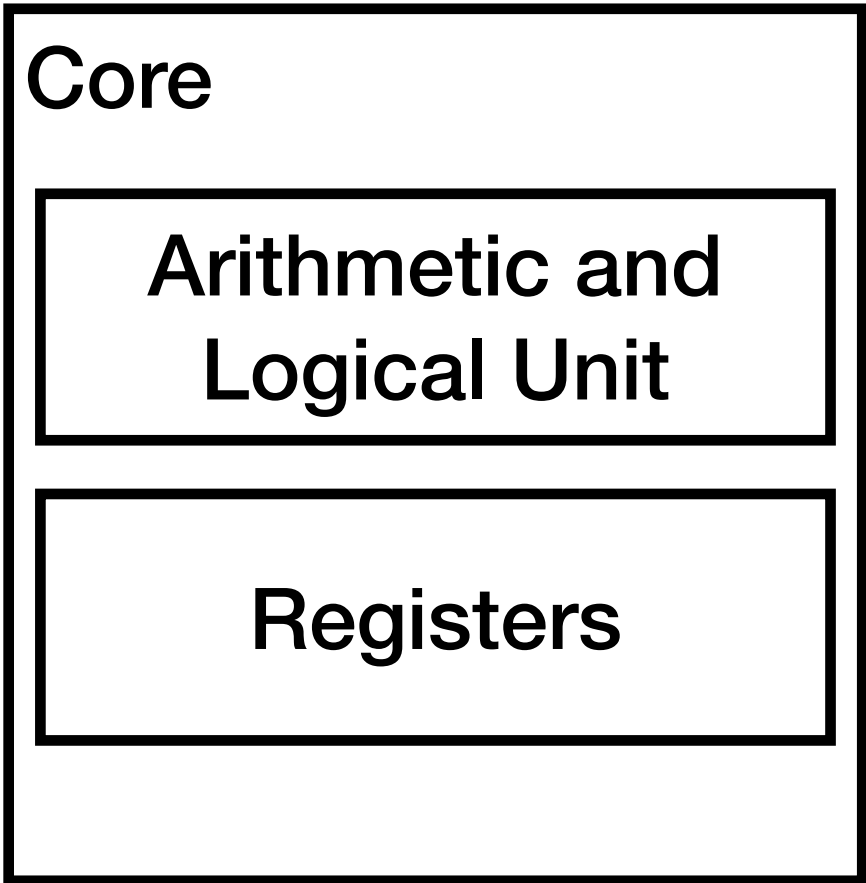
- pthread_mutex_lock "locks" cnt, if it is already locked, pthread_mutex_lock will wait
- pthread_mutex_unlock "unlocks" cnt
- Each loop now becomes
 - lock
 - read cnt
 - incr

Thread (cont.)

goodcnt.c



11



11

```
void *count(void *arg)
{
    (void) arg;

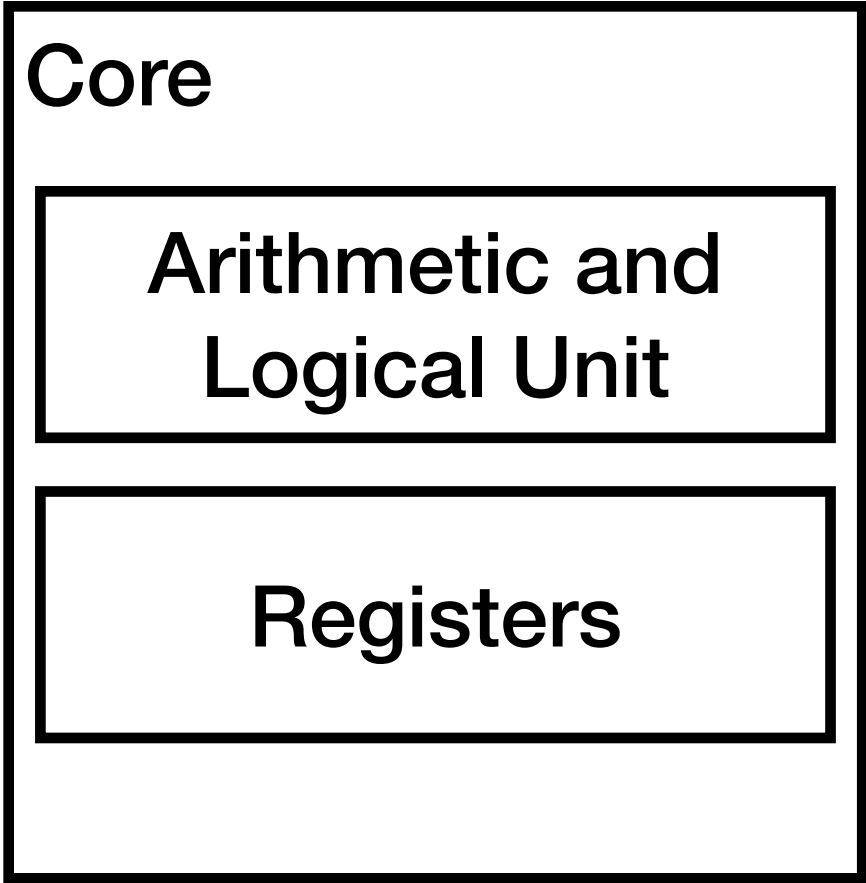
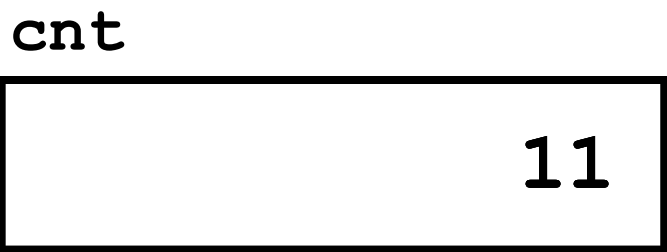
    for (unsigned int i = 0; i < N; i++) {
        pthread_mutex_lock(&sem);
        cnt++;
        pthread_mutex_unlock(&sem);
    }

    return NULL;
}
```

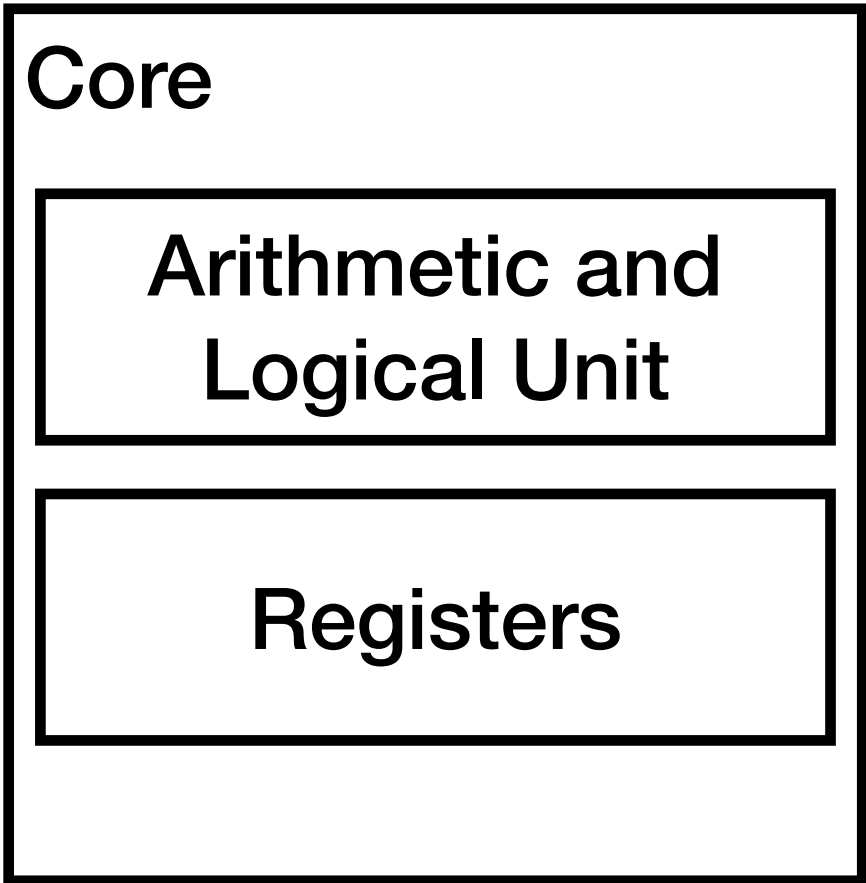
- pthread_mutex_lock "locks" cnt, if it is already locked, pthread_mutex_lock will wait
- pthread_mutex_unlock "unlocks" cnt
- Each loop now becomes
 - lock
 - read cnt
 - incr
 - write to cnt

Thread (cont.)

goodcnt.c



11



11

```
void *count(void *arg)
{
    (void) arg;

    for (unsigned int i = 0; i < N; i++) {
        pthread_mutex_lock(&sem);
        cnt++;
        pthread_mutex_unlock(&sem);
    }

    return NULL;
}
```

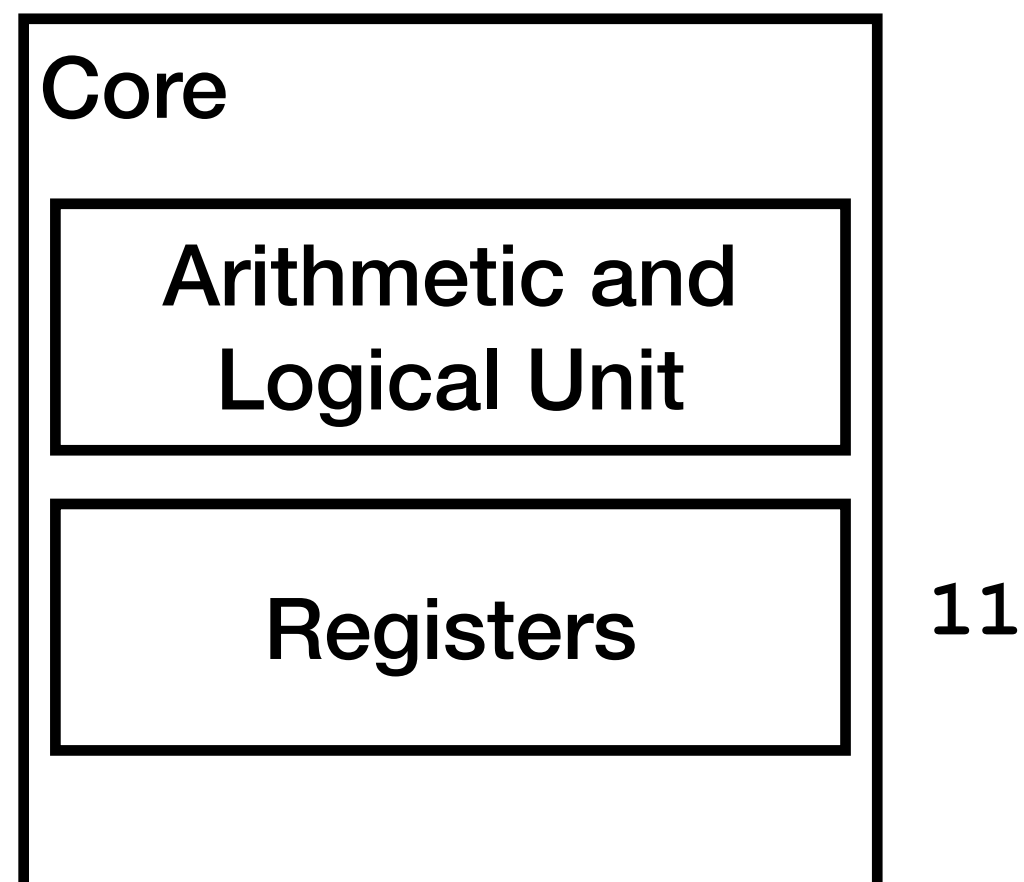
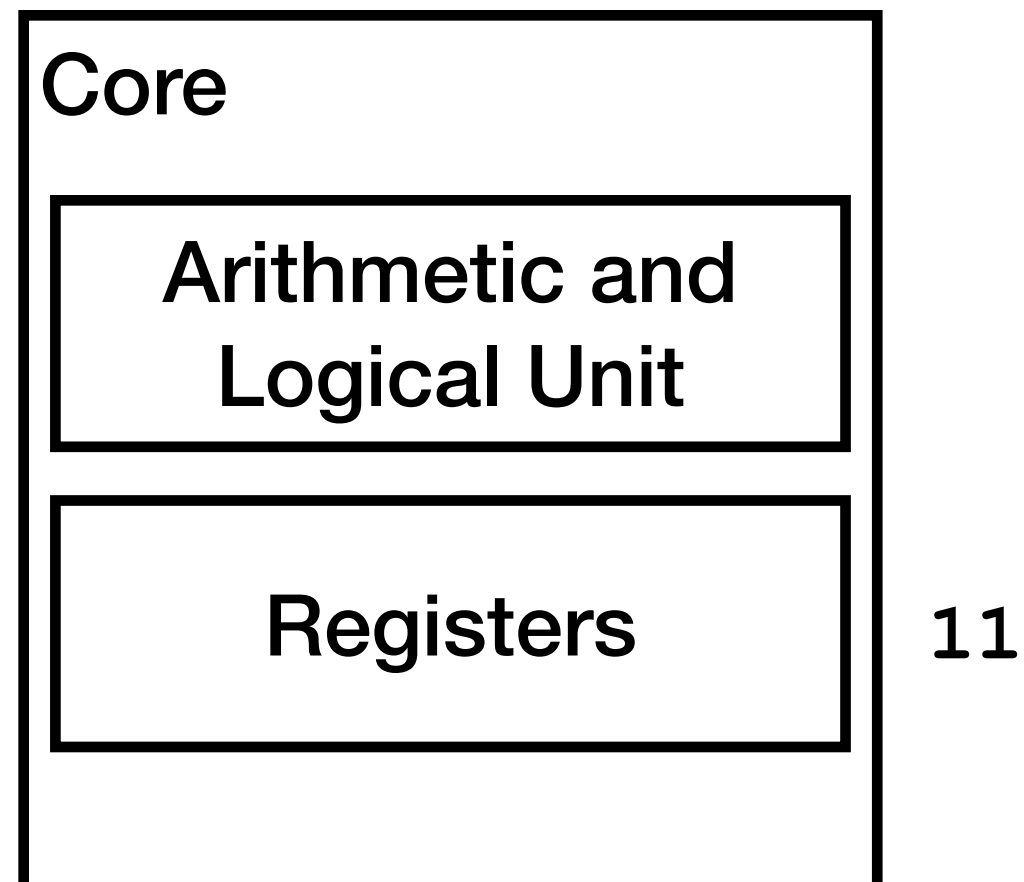
- pthread_mutex_lock "locks" cnt, if it is already locked, pthread_mutex_lock will wait
- pthread_mutex_unlock "unlocks" cnt
- Each loop now becomes
 - lock
 - read cnt
 - incr
 - write to cnt
 - unlock

Thread (cont.)

goodcnt.c

cnt

11



```
void *count(void *arg)
{
    (void) arg;

    for (unsigned int i = 0; i < N; i++) {
        pthread_mutex_lock(&sem);
        cnt++;
        pthread_mutex_unlock(&sem);
    }

    return NULL;
}
```

- pthread_mutex_lock "locks" cnt, if it is already locked, pthread_mutex_lock will wait
- pthread_mutex_unlock "unlocks" cnt
- Each loop now becomes
 - lock
 - read cnt
 - incr
 - write to cnt
 - unlock
- No other thread will touch the number between reading and writing