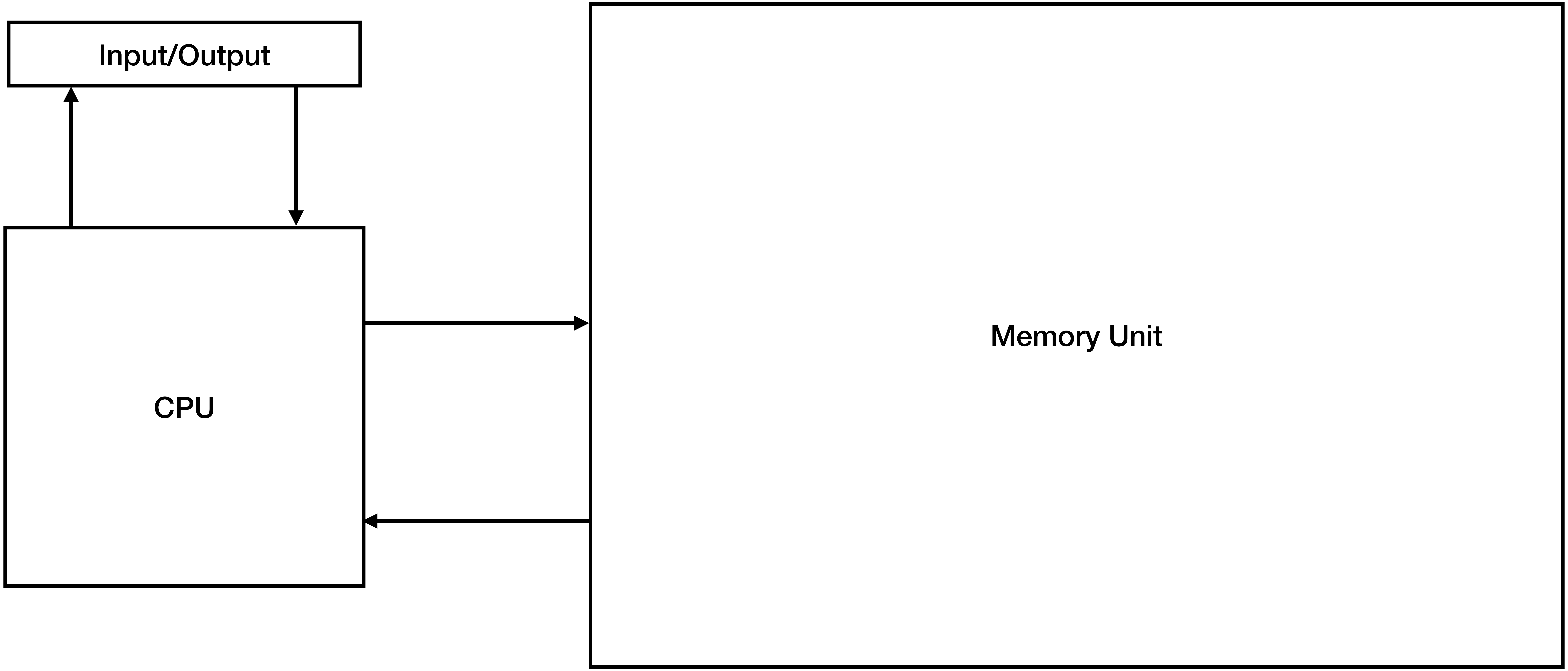
What is code?

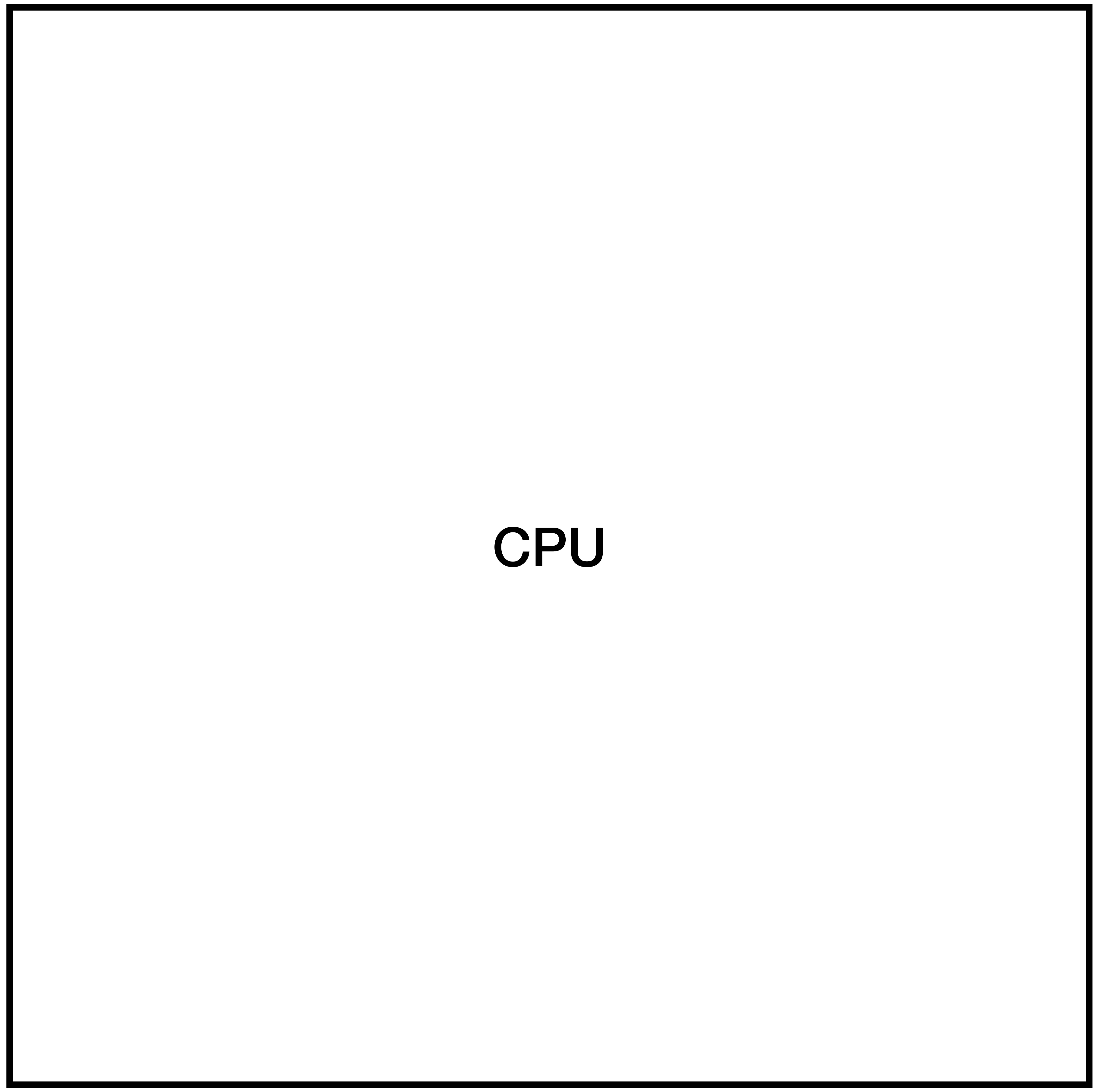
CS143: lecture 16

Byron Zhong, July 24

A Von Neumann Machine

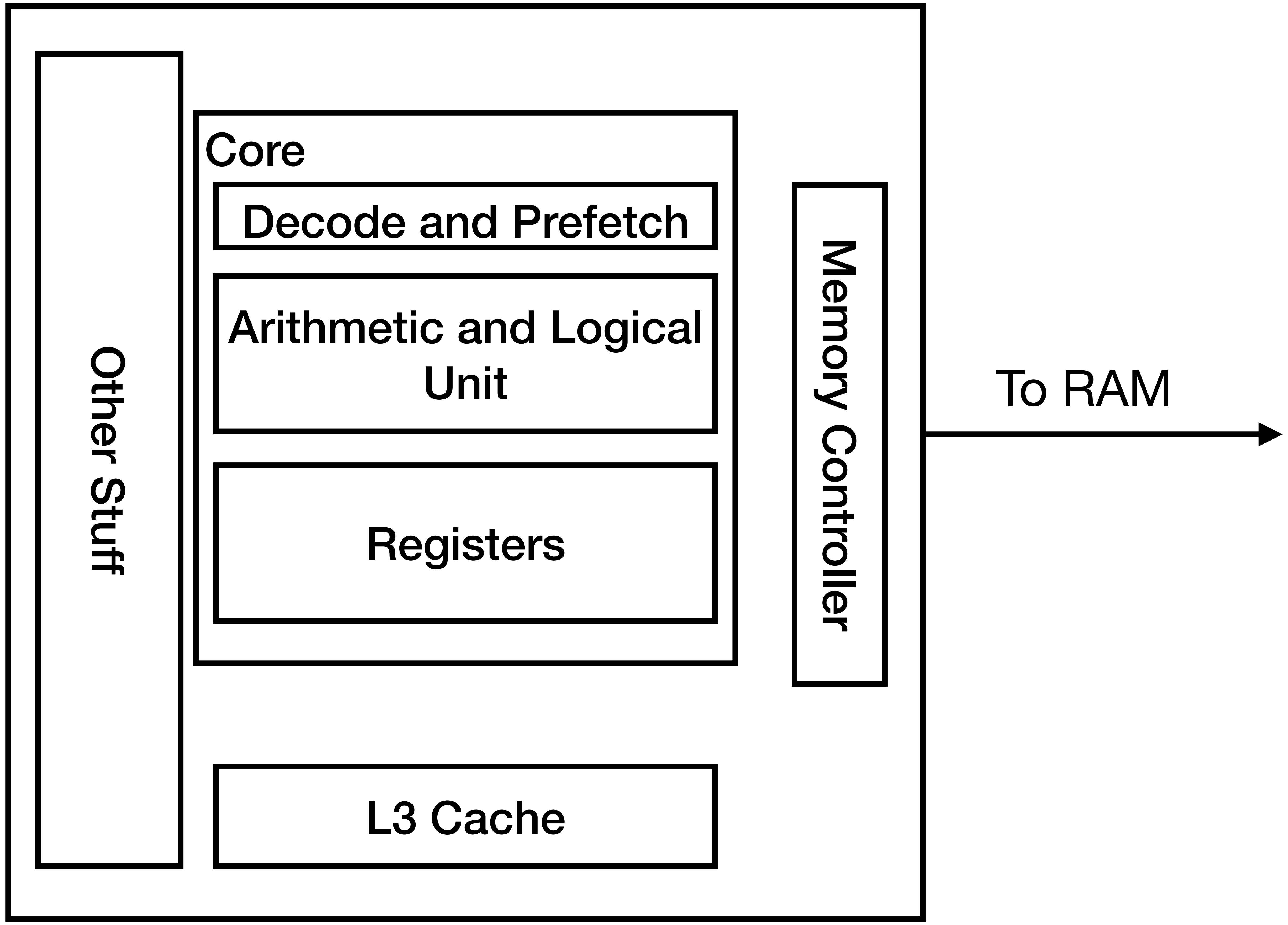


A Von Neumann Machine



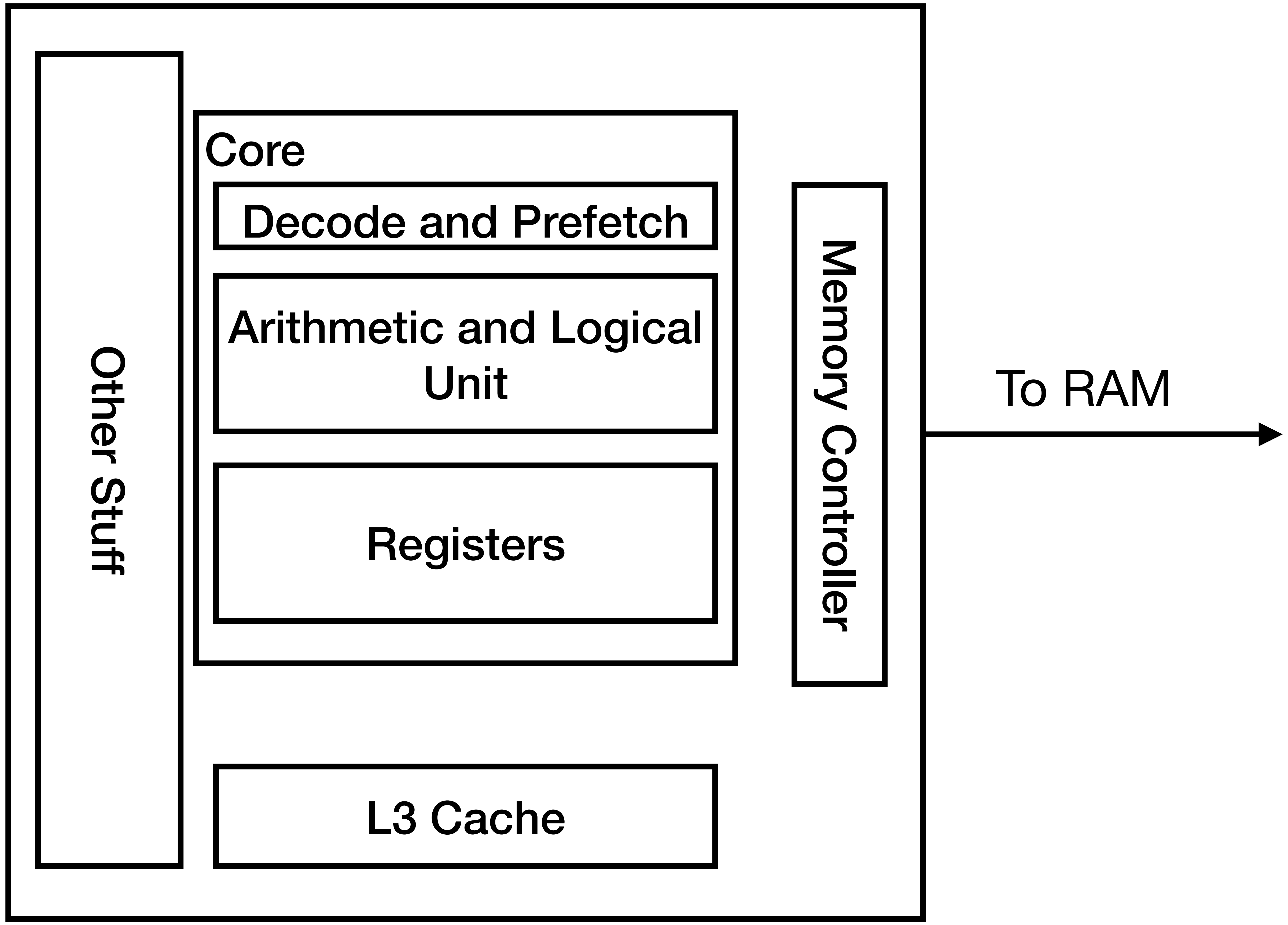
CPU

CPU



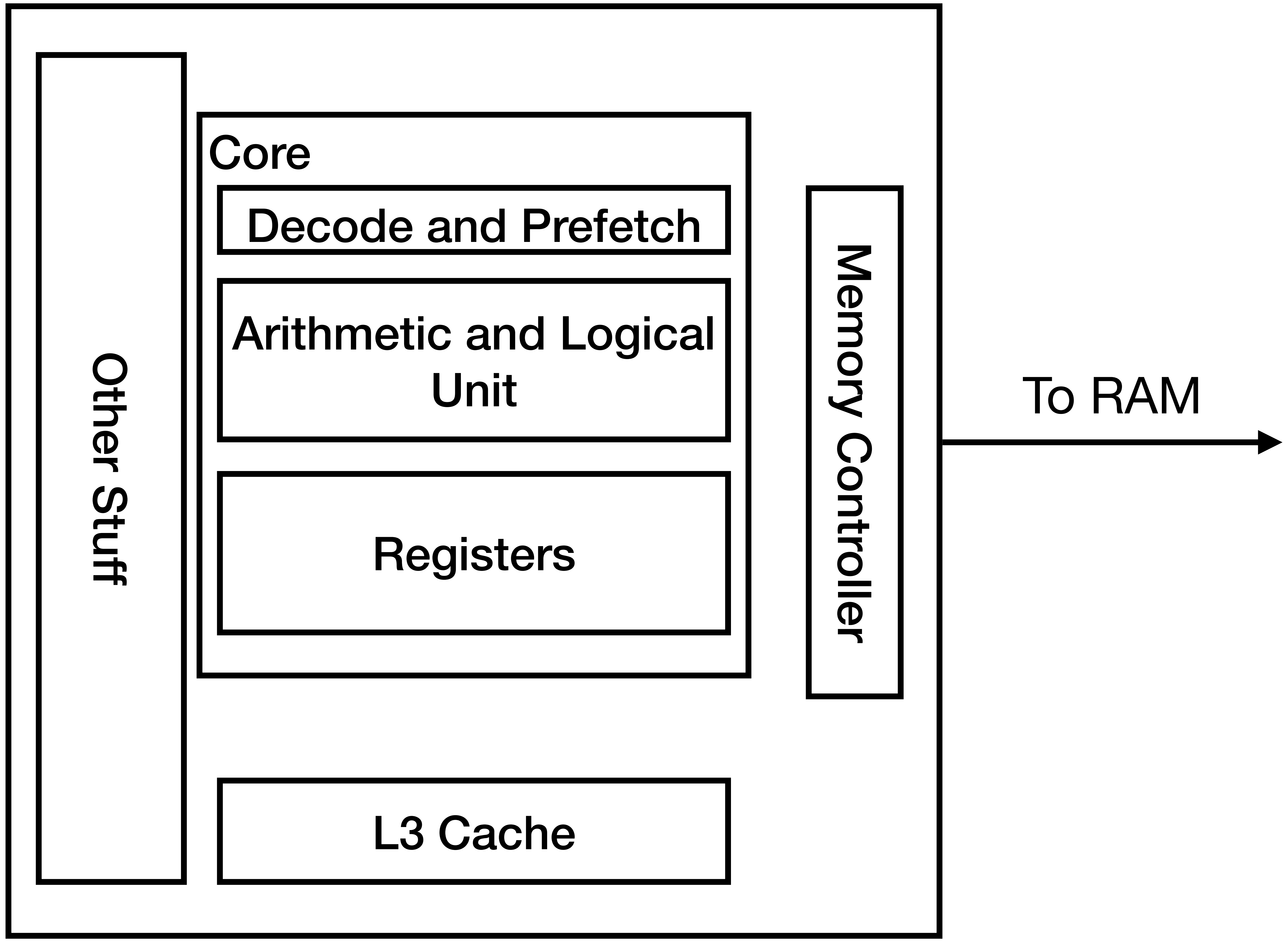
- *Registers*: named locations storing 64-bit values. These registers can hold integers or addresses (pointers).
    - Some registers keep track of program states; others hold temporary data, such as local variables
- *ALU*: reads from registers and perform calculations.

CPU



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

CPU



- 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

```c
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

```c
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

```c
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

```
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
```

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

Instructions

```
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
```

%eax stores the return value.

Instructions

```c
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:                             add    %esi,%eax
   4:                             add    %eax,0x0(%rip)
   a:   c3                        retq
```

move the first argument
(edi) to the return
value (eax)

Instructions

```
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:                                add    %esi,%eax
   4:                                add    %eax,0x0(%rip)
   a:   c3                           retq
```

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

Instructions

```
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:                                    add    %esi,%eax
   4:                                    add    %eax,0x0(%rip)
   a:   c3                               retq
```

add eax to a location.

Instructions

```
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:                                   add    %esi,%eax
   4:                                   add    %eax,0x0(%rip)
   a:   c3                              retq
```

return

Instructions

```c
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

```c
int accum = 0;

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

code.c

```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

```
0000000000401110 <sum>:
  401110:89 f8                  mov    %edi,%eax
  401112:01 f0                  add    %esi,%eax
  401114:01 05 12 2f 00 00      add    %eax,0x2f12(%rip)       # 40402c <accum>
  40111a:c3                     retq
  40111b:0f 1f 44 00 00         nopl   0x0(%rax,%rax,1)

0000000000401120 <main>:
  401120:bf 01 00 00 00         mov    $0x1,%edi
  401125:be 03 00 00 00         mov    $0x3,%esi
  40112a:e9 e1 ff ff ff         jmpq   401110 <sum>
  40112f:90                     nop
```

mov 1 to edi

Instructions

```
0000000000401110 <sum>:
  401110:89 f8                    mov    %edi,%eax
  401112:01 f0                    add    %esi,%eax
  401114:01 05 12 2f 00 00        add    %eax,0x2f12(%rip)      # 40402c <accum>
  40111a:c3                       retq
  40111b:0f 1f 44 00 00           nopl   0x0(%rax,%rax,1)

0000000000401120 <main>:
  401120:bf 01 00 00 00           mov    $0x1,%edi
  401125:be 03 00 00 00           mov    $0x3,%esi
  40112a:e9 e1 ff ff ff           jmpq   401110 <sum>
  40112f:90                       nop
```

mov 3 to esi

Instructions

```
0000000000401110 <sum>:
  401110:89 f8                       mov    %edi,%eax
  401112:01 f0                       add    %esi,%eax
  401114:01 05 12 2f 00 00           add    %eax,0x2f12(%rip)        # 40402c <accum>
  40111a:c3                          retq
  40111b:0f 1f 44 00 00              nopl   0x0(%rax,%rax,1)

0000000000401120 <main>:
  401120:bf 01 00 00 00              mov    $0x1,%edi
  401125:be 03 00 00 00              mov    $0x3,%esi
  40112a:e9 e1 ff ff ff              jmpq   401110 <sum>
  40112f:90                          nop
```

jump to location 401110

Instructions

```
0000000000401110 <sum>:
  401110:89 f8                  mov    %edi,%eax
  401112:01 f0                  add    %esi,%eax
  401114:01 05 12 2f 00 00      add    %eax,0x2f12(%rip)      # 40402c <accum>
  40111a:c3                     retq
  40111b:0f 1f 44 00 00         nopl   0x0(%rax,%rax,1)

0000000000401120 <main>:
  401120:bf 01 00 00 00         mov    $0x1,%edi
  401125:be 03 00 00 00         mov    $0x3,%esi
  40112a:e9 e1 ff ff ff         jmpq   401110 <sum>
  40112f:90                     nop
```

The global variable has been assigned a location 40402c

Separate Compilation

How C is actually compiled

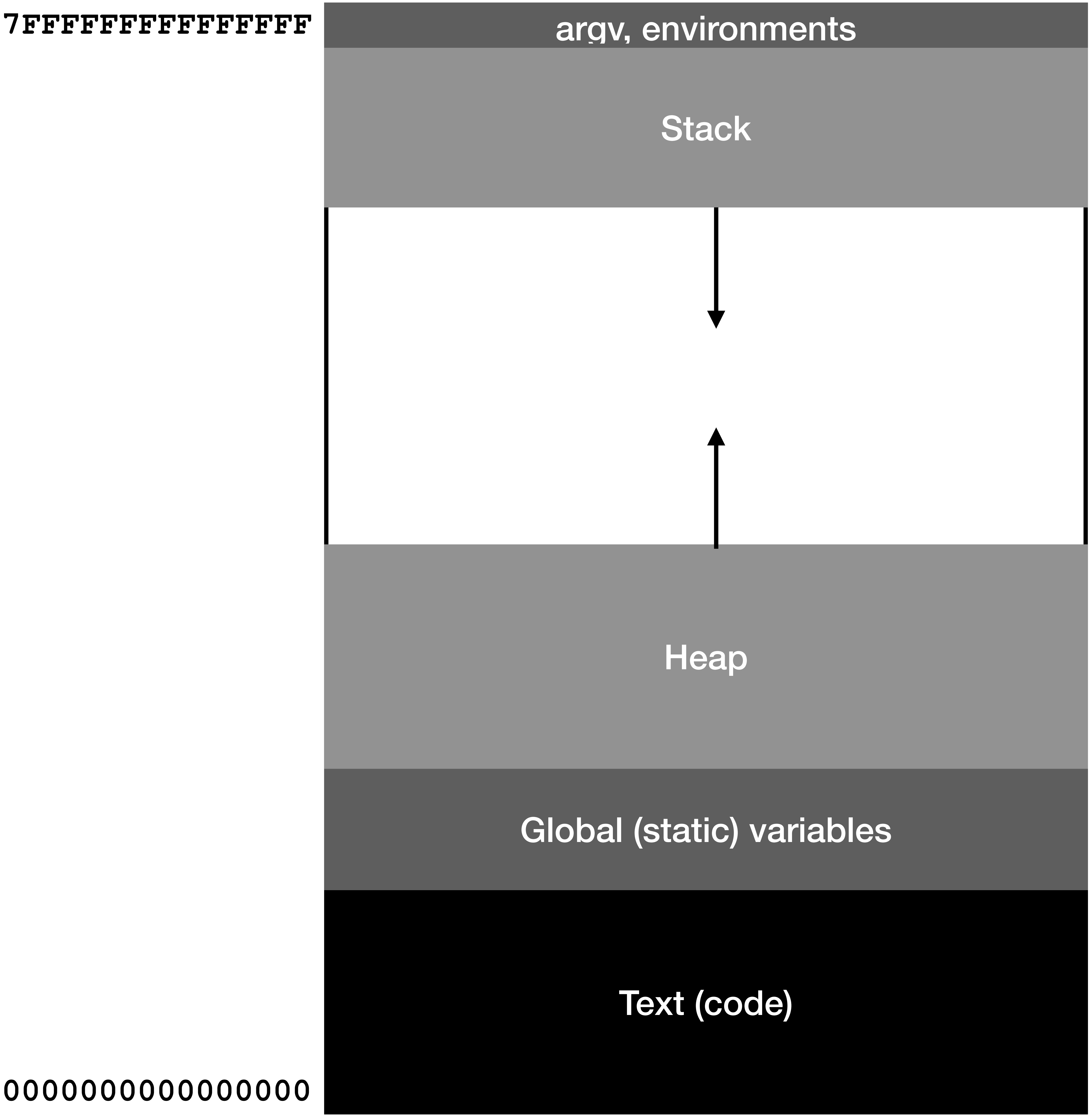
file1.c
preprocessor
compiler
assembler
file1.o
file2.c
preprocessor
compiler
assembler
file2.o
. . .
. . .
. . .
linker
./exec

- every .o file provides some functions
- linker links all the functions together
- finds main

Instructions

- 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

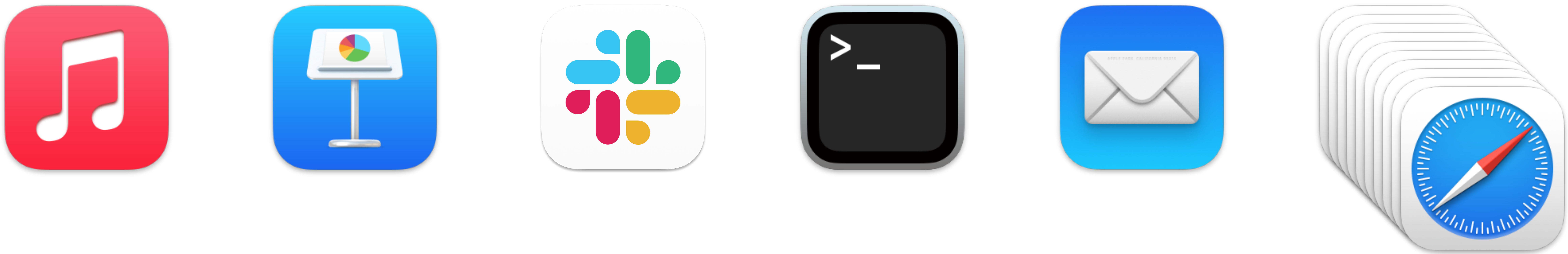


Instructions

- 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
  - ...

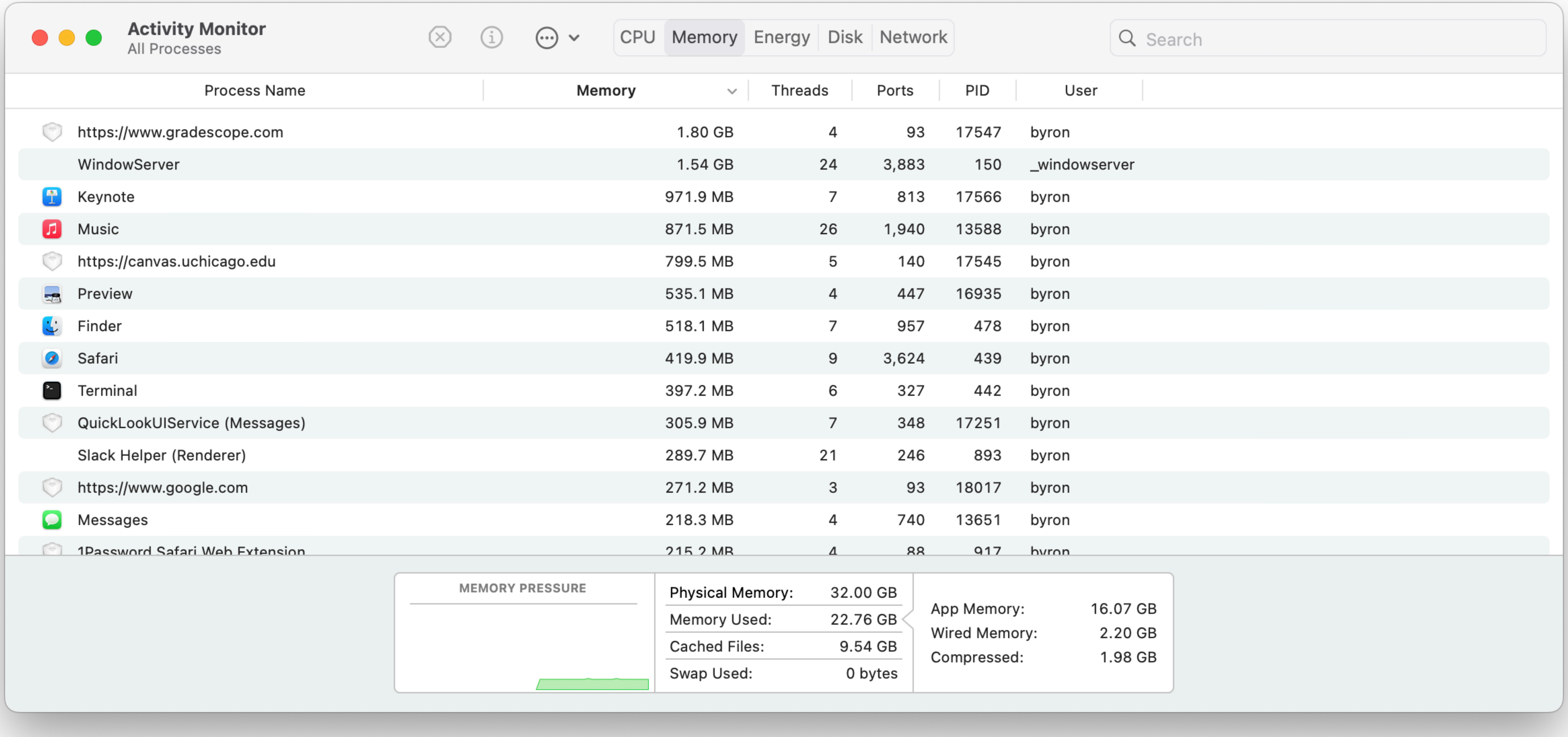
Machine

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



Machine

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



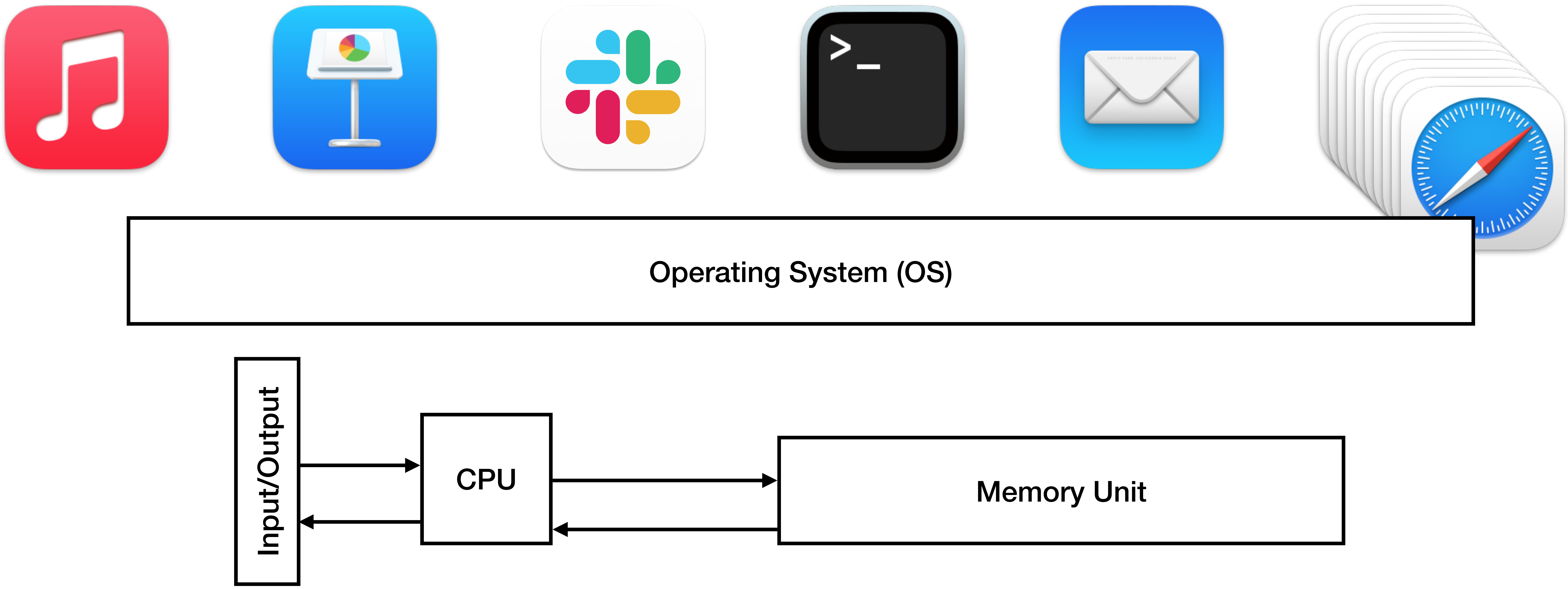
Process Name	Memory	Threads	Ports	PID	User
https://www.gradescope.com	1.80 GB	4	93	17547	byron
WindowServer	1.54 GB	24	3,883	150	_windowserver
Keynote	971.9 MB	7	813	17566	byron
Music	871.5 MB	26	1,940	13588	byron
https://canvas.uchicago.edu	799.5 MB	5	140	17545	byron
Preview	535.1 MB	4	447	16935	byron
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

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

Machine

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



Operating System (OS)

Input/Output

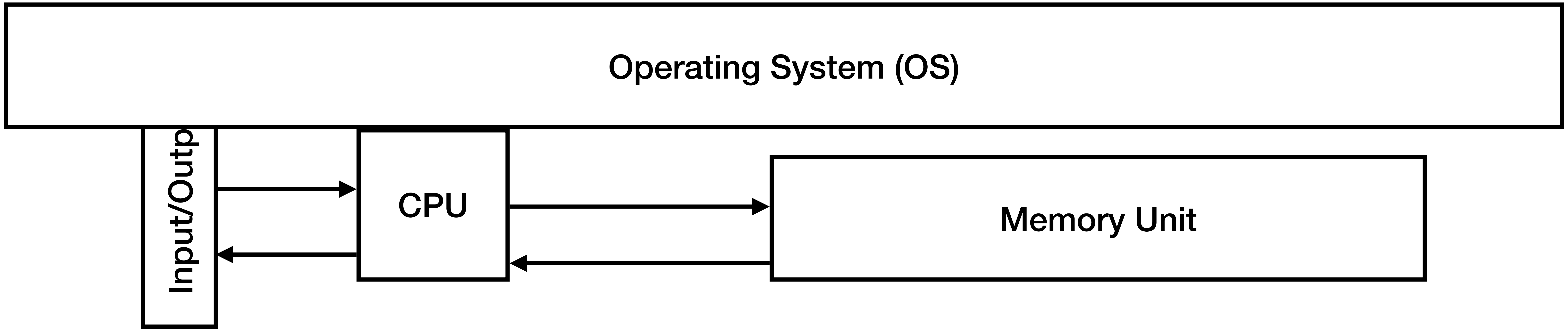
CPU

Memory Unit

Machine

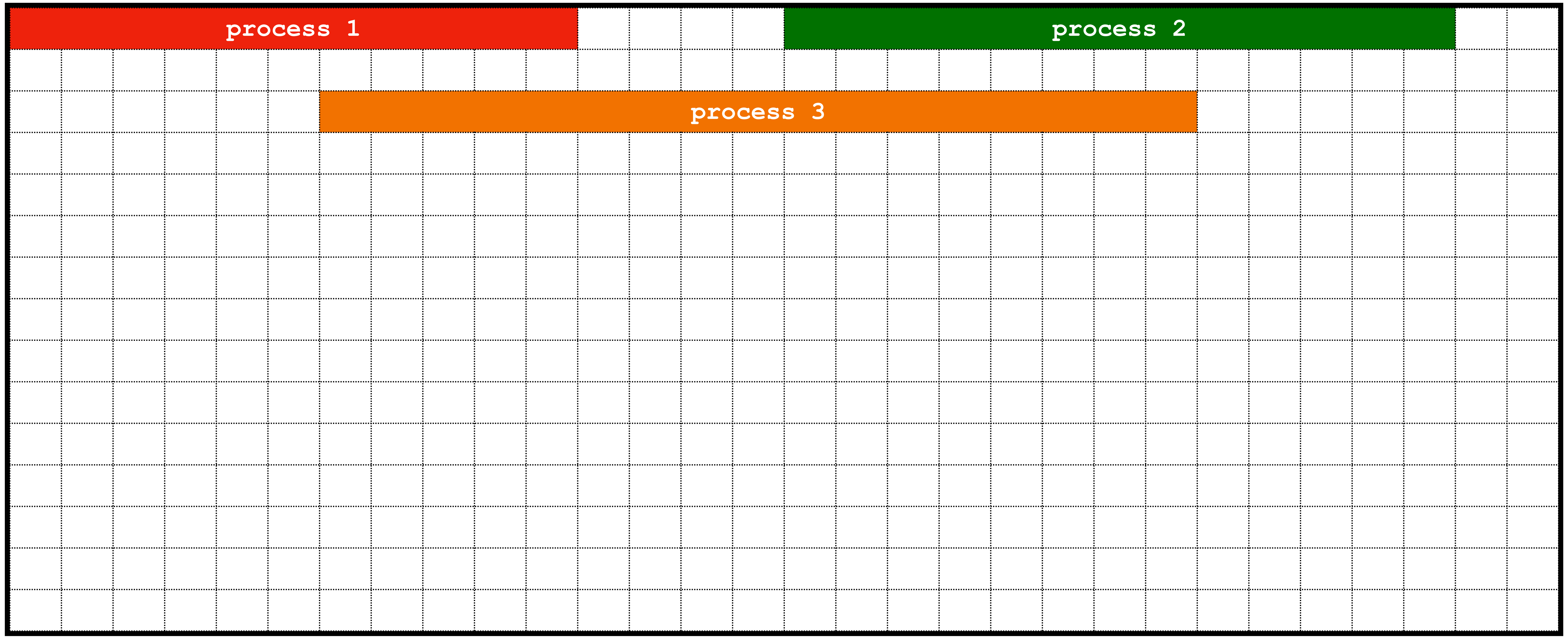
Your computer can do many things at the same time...

- 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

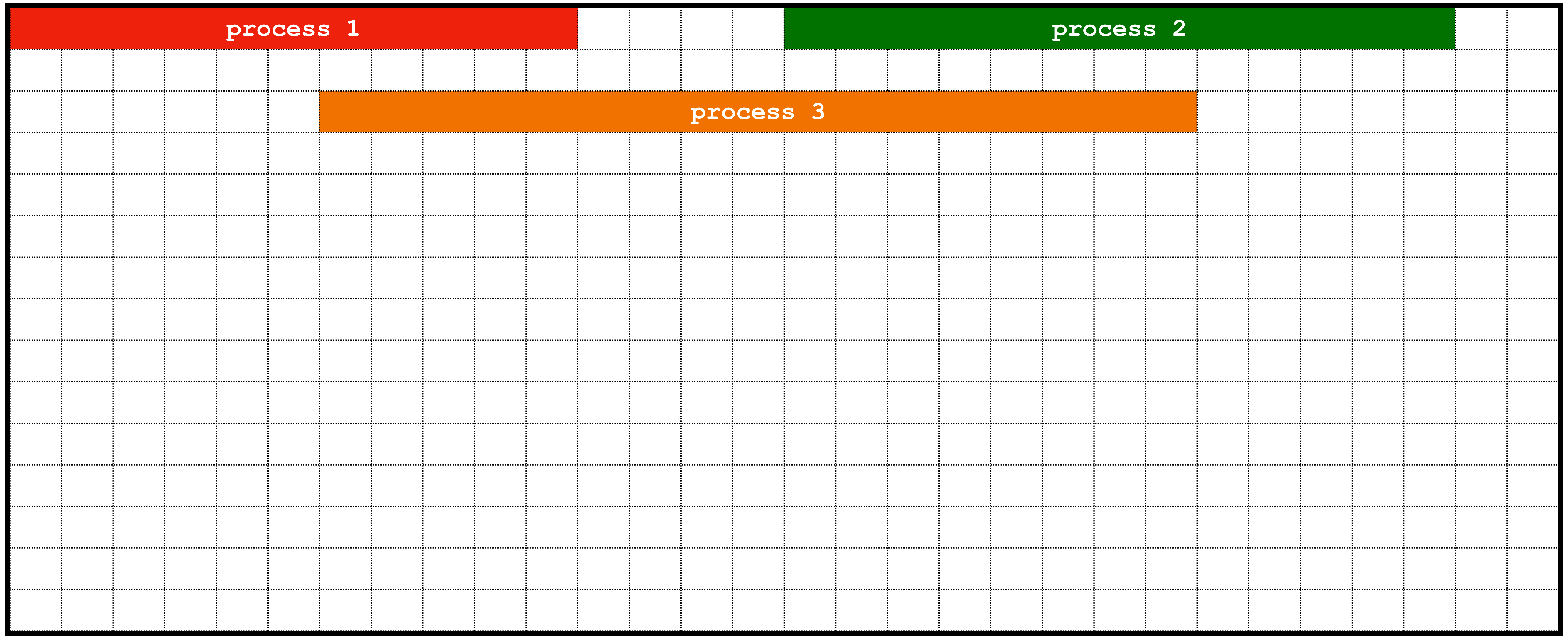
Physical memory



Virtual Memory

What if process 1 needs more memory?

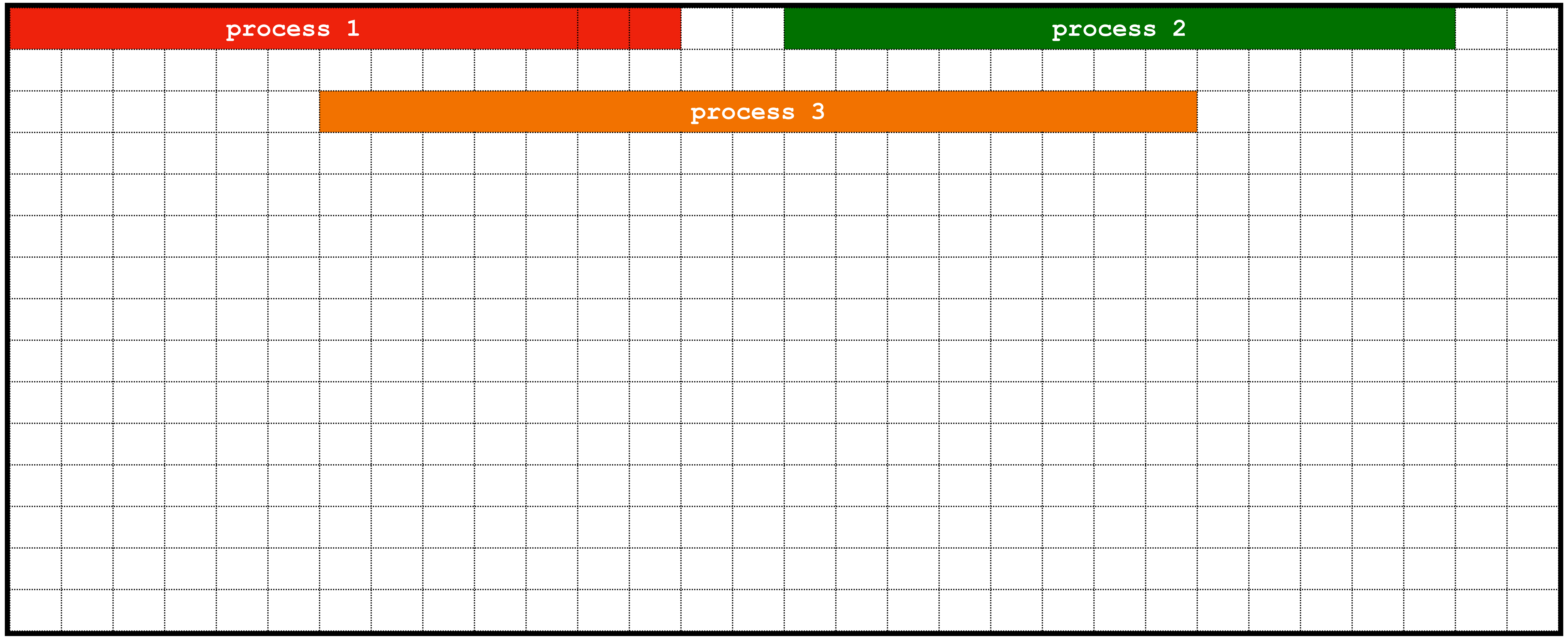
Physical memory



Virtual Memory

What if process 1 needs more memory?

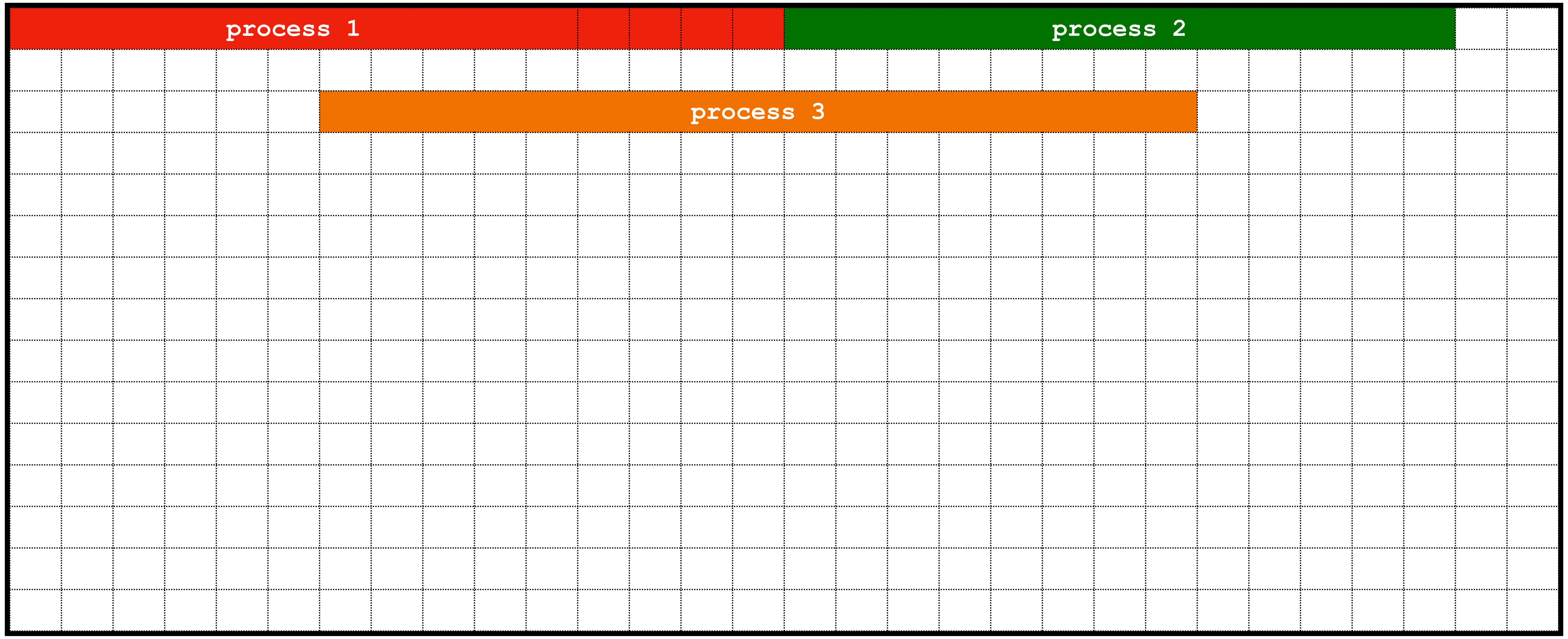
Physical memory



Virtual Memory

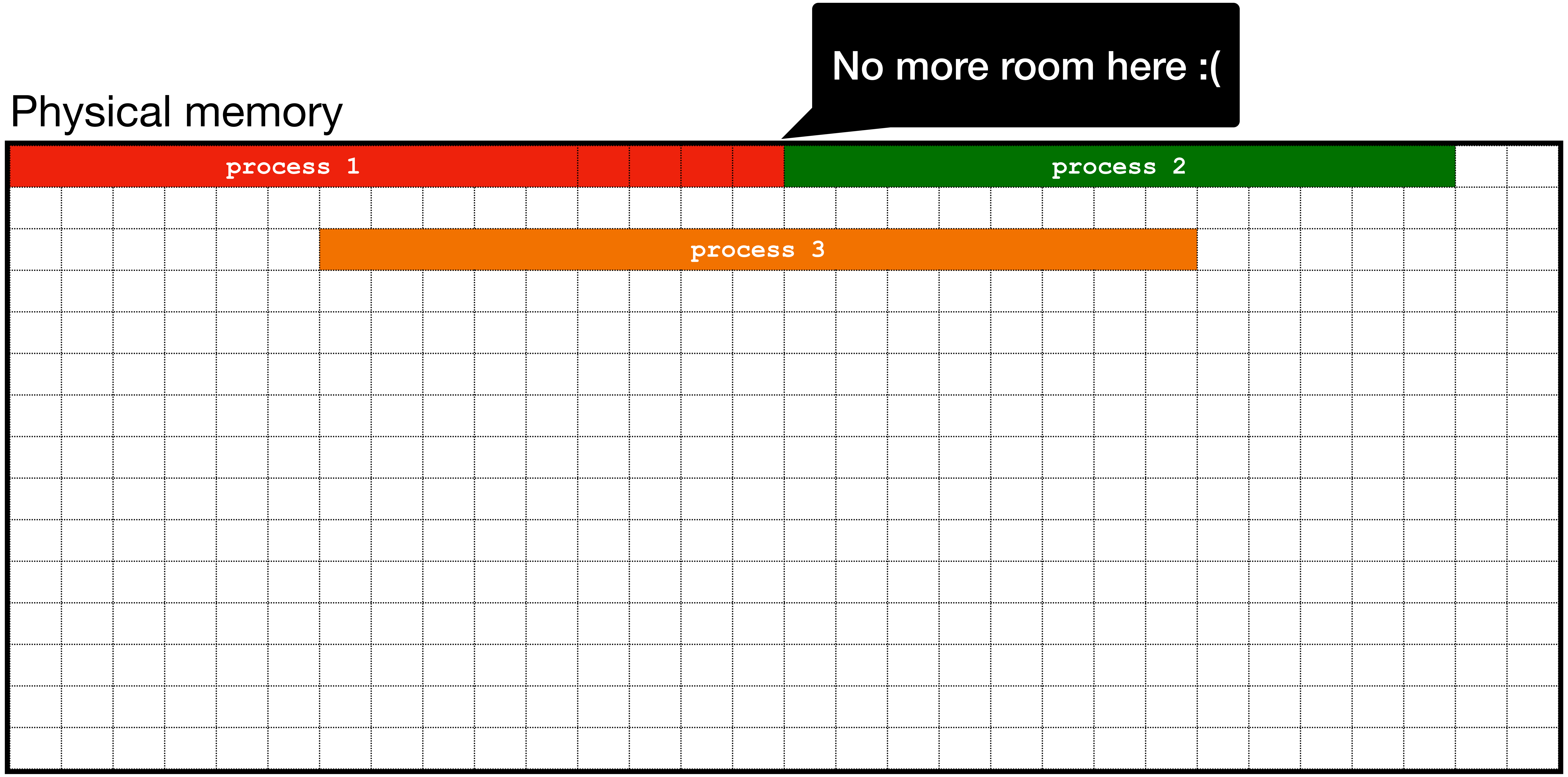
What if process 1 needs more memory?

Physical memory



Virtual Memory

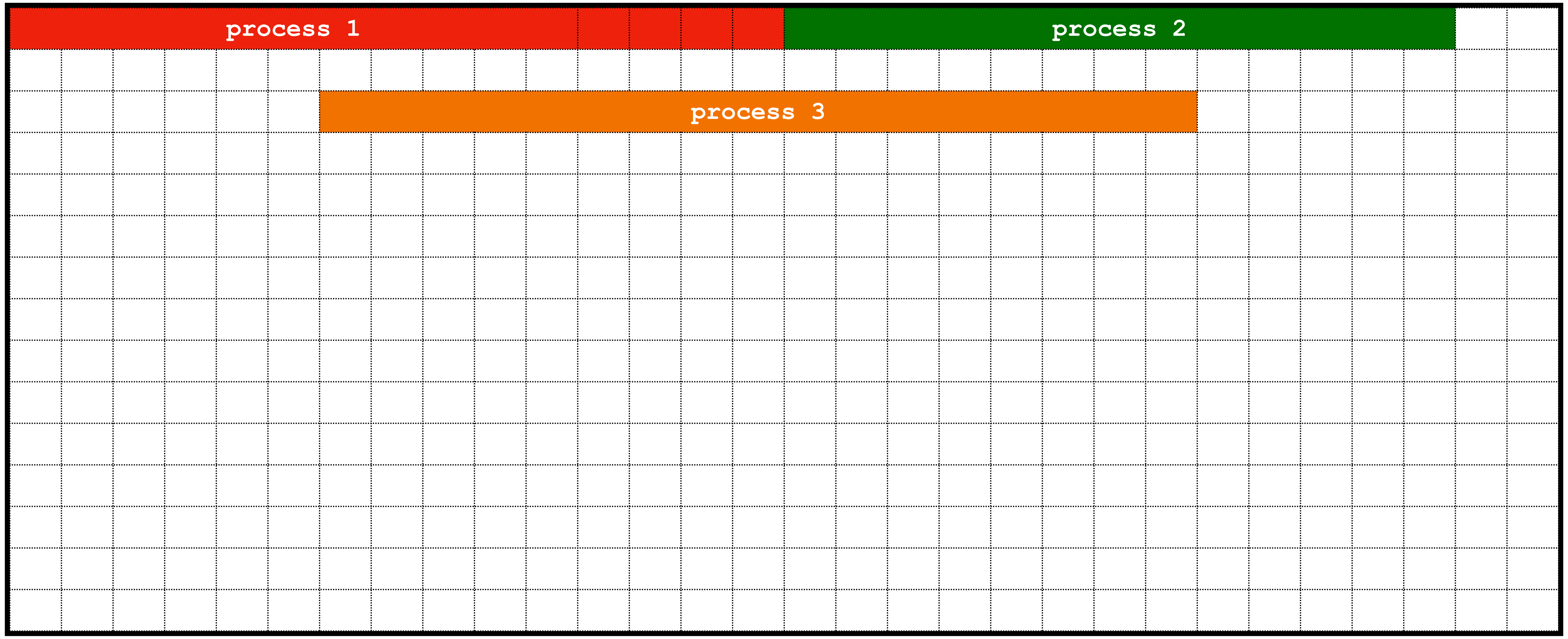
What if process 1 needs more memory?



Virtual Memory

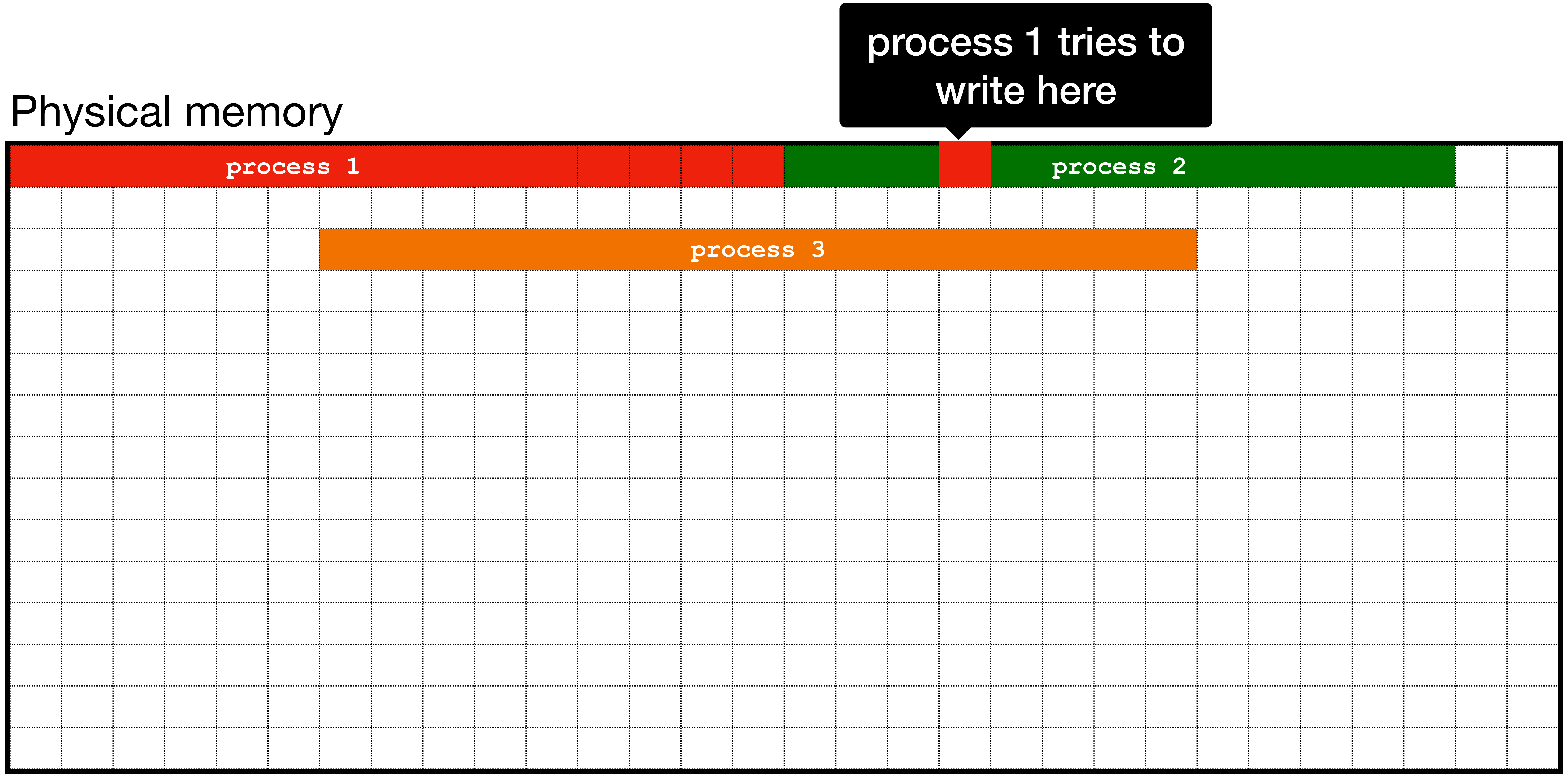
What if process 1 is buggy or malicious?

Physical memory

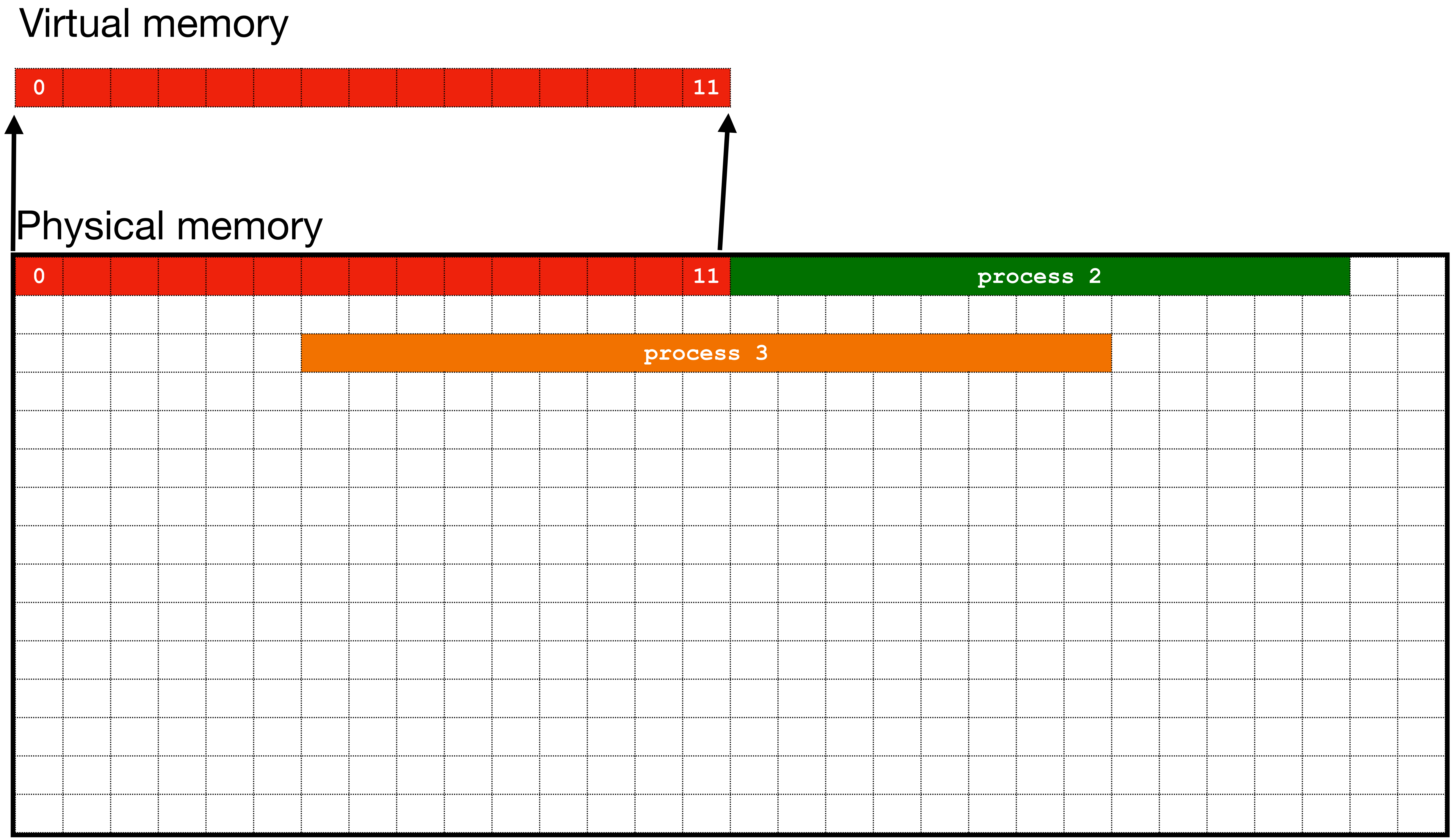


Virtual Memory

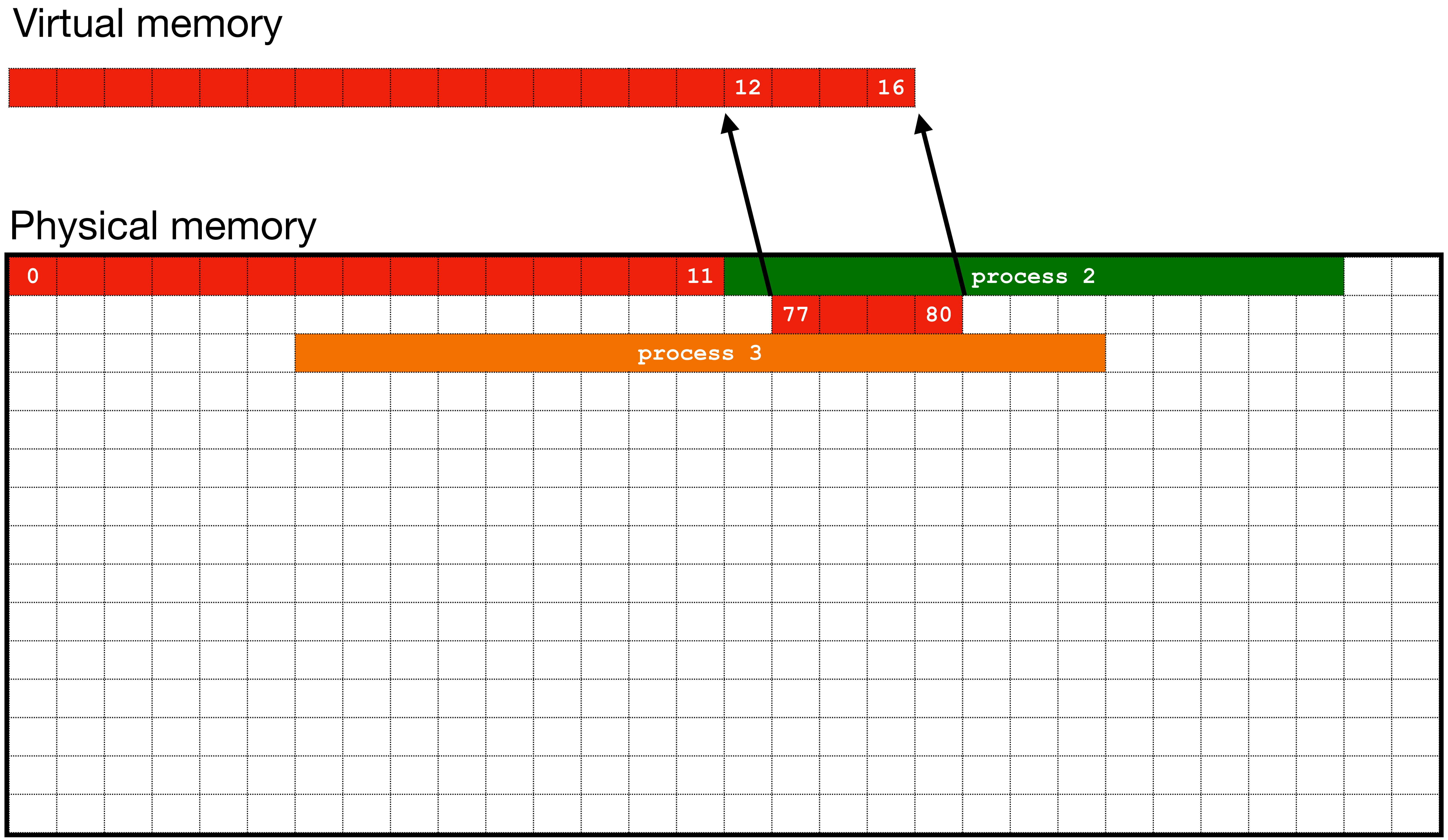
What if process 1 is buggy or malicious?



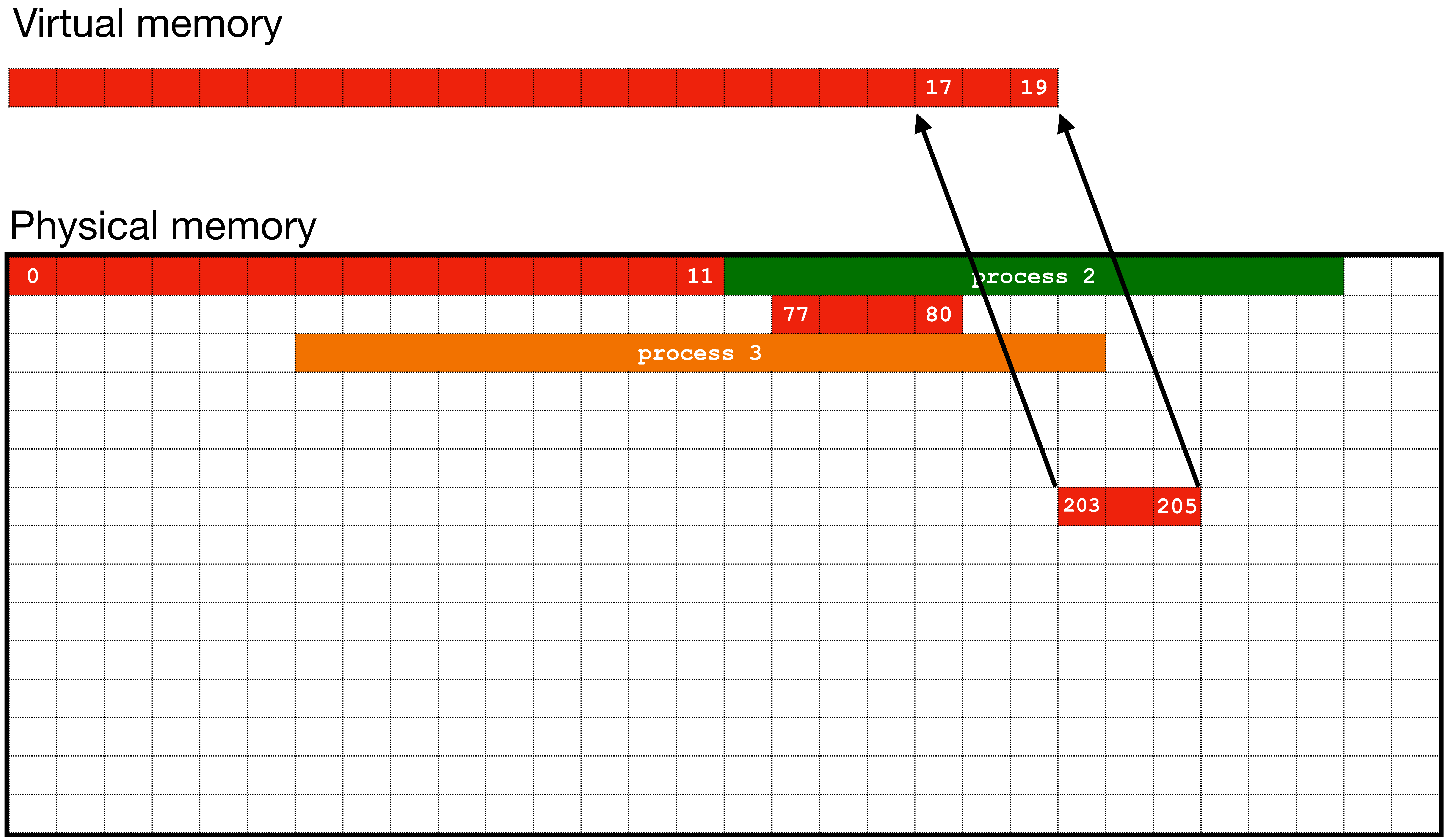
Virtual Memory



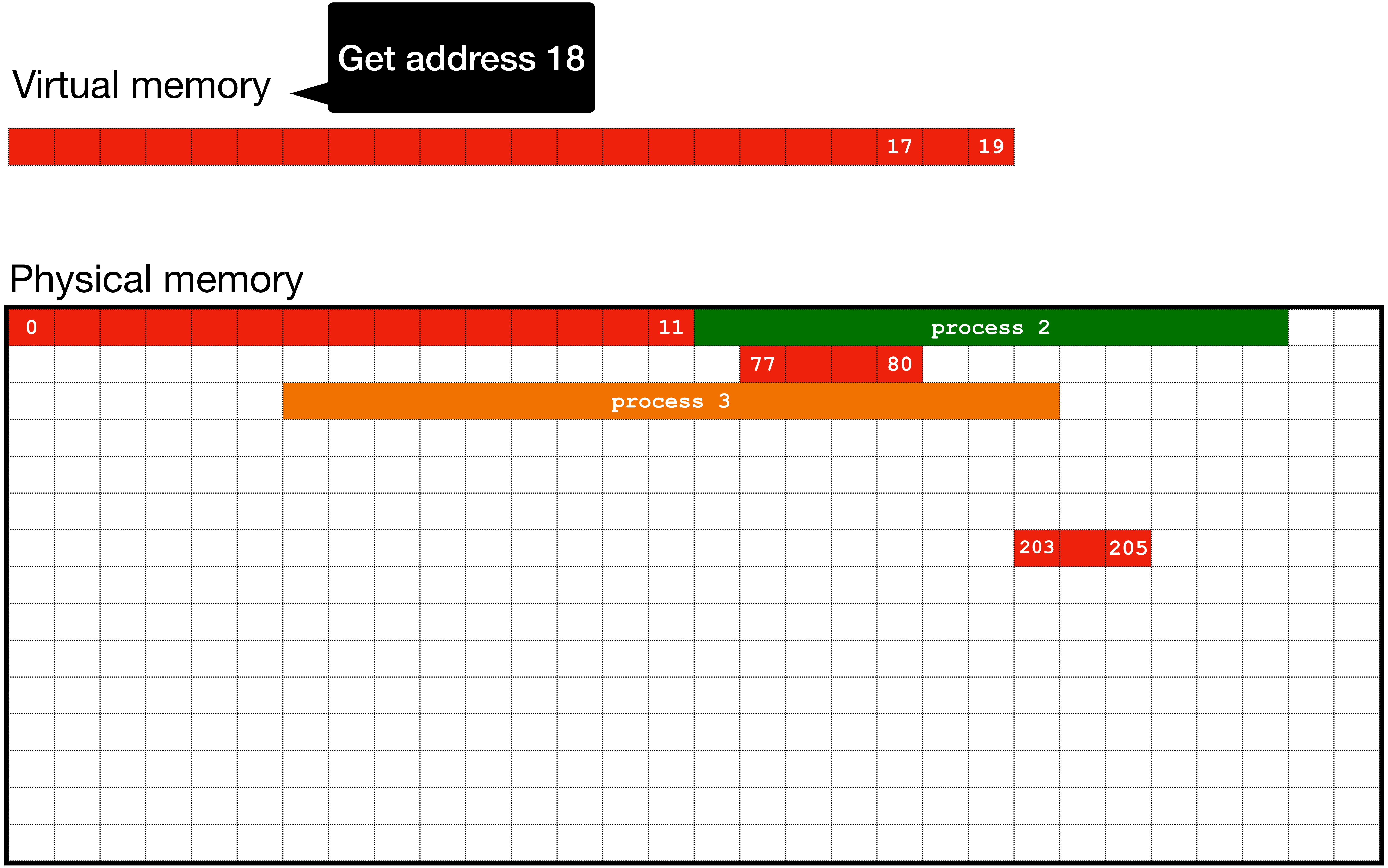
Virtual Memory



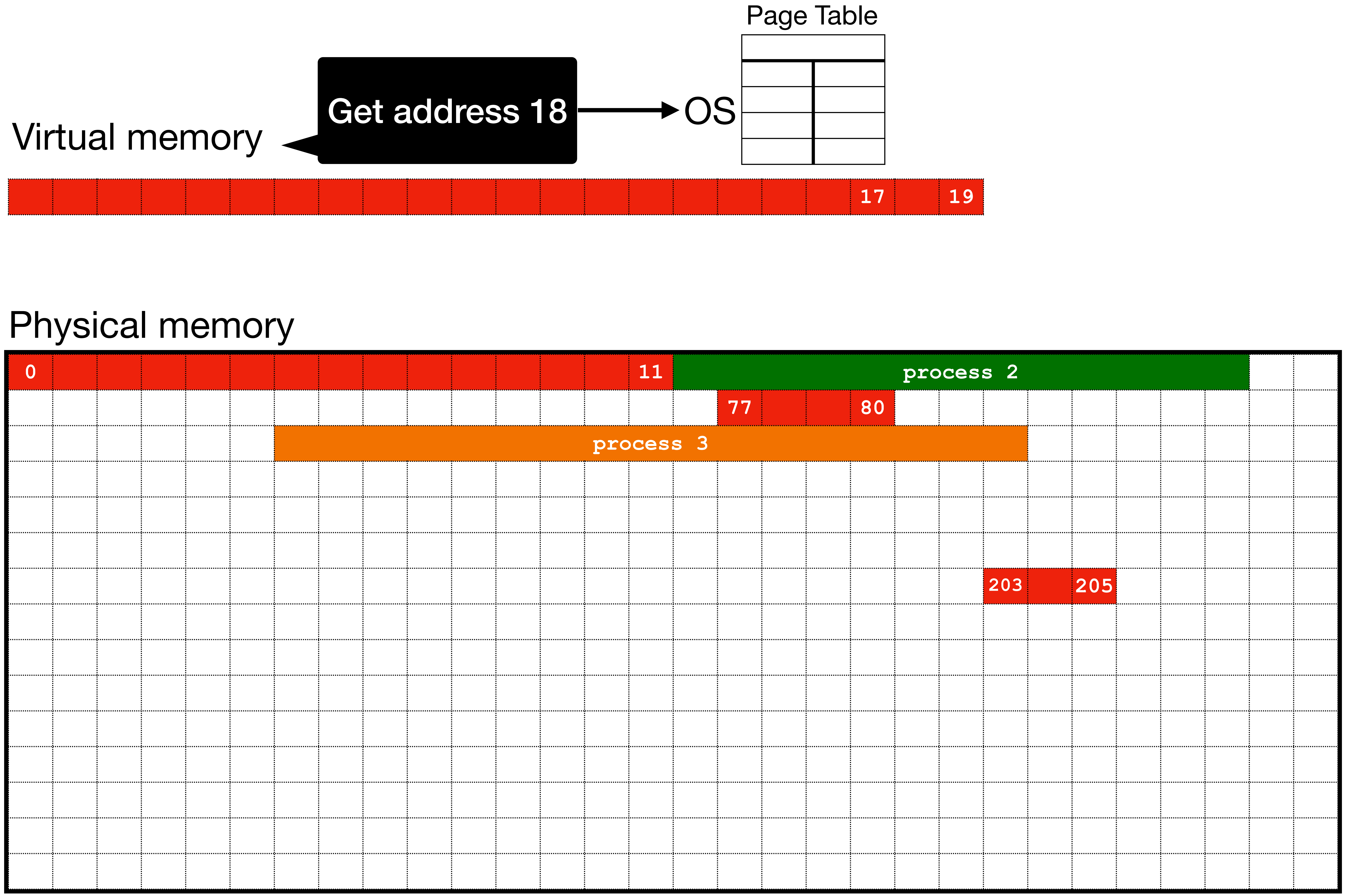
Virtual Memory



Virtual Memory



Virtual Memory



Page Table

Get address 18

OS

Virtual memory

17

19

Physical memory

0

11

process 2

77

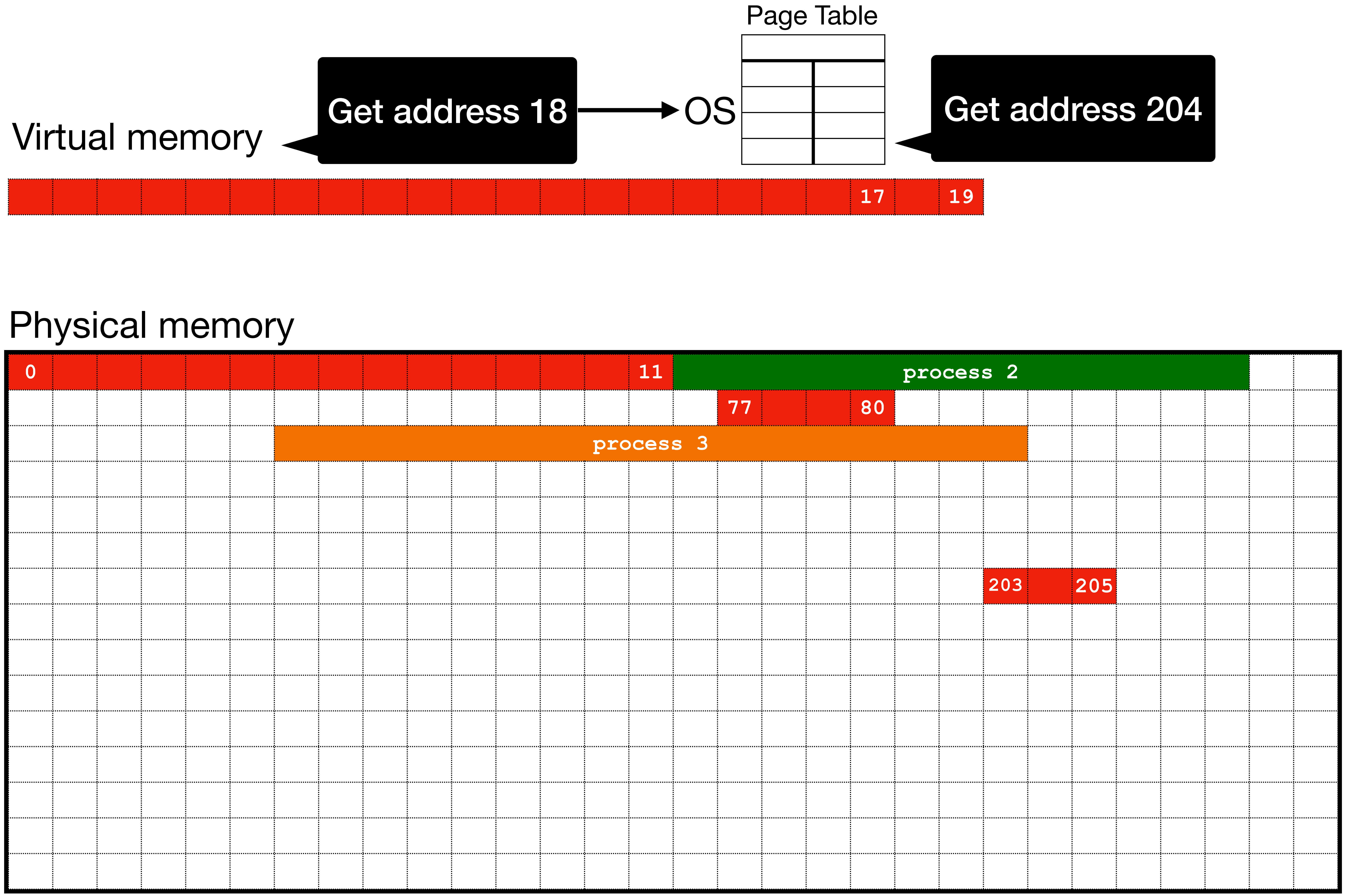
80

process 3

203

205

Virtual Memory



Page Table

Get address 18

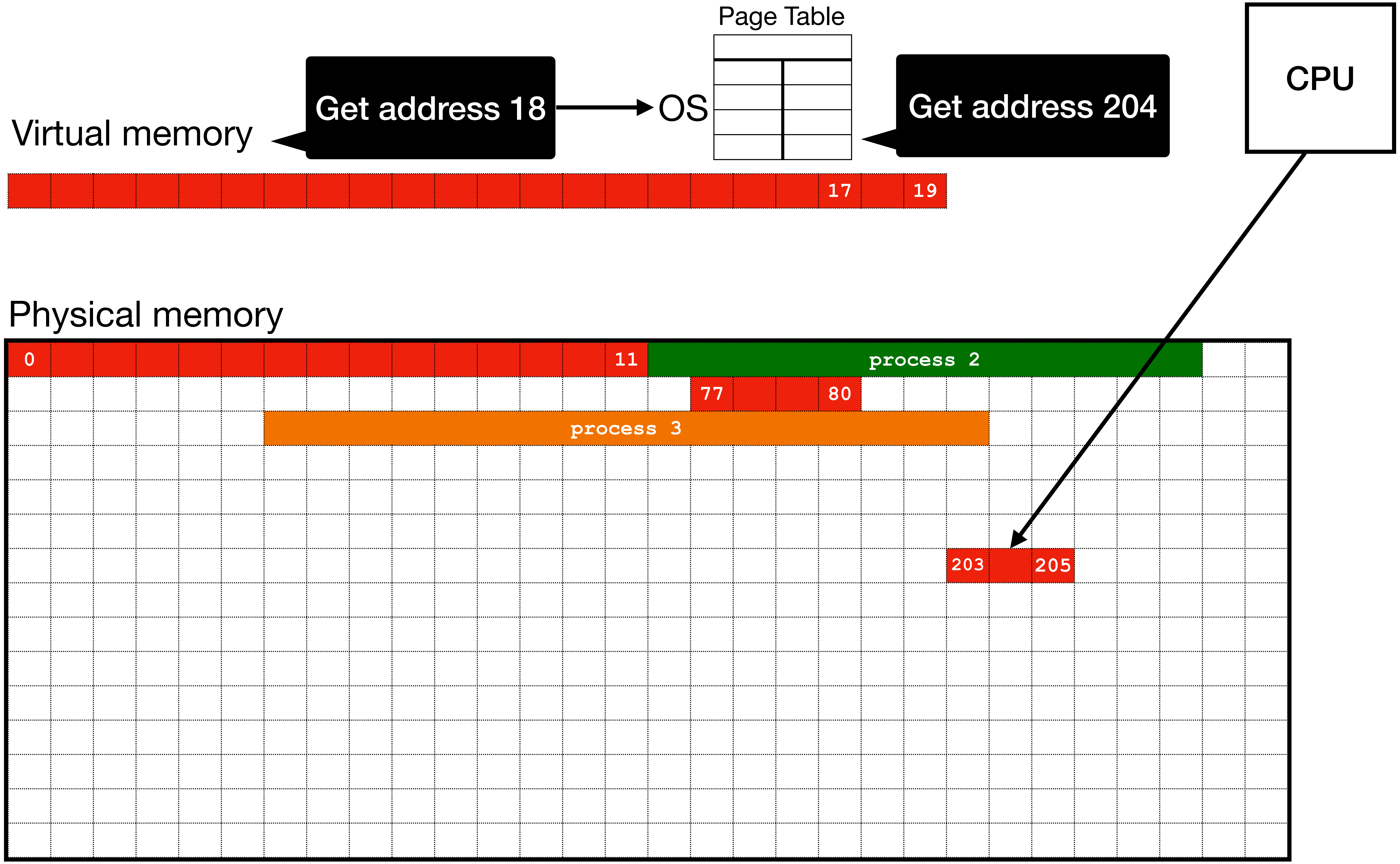
OS

Get address 204

Virtual memory

Physical memory

Virtual Memory

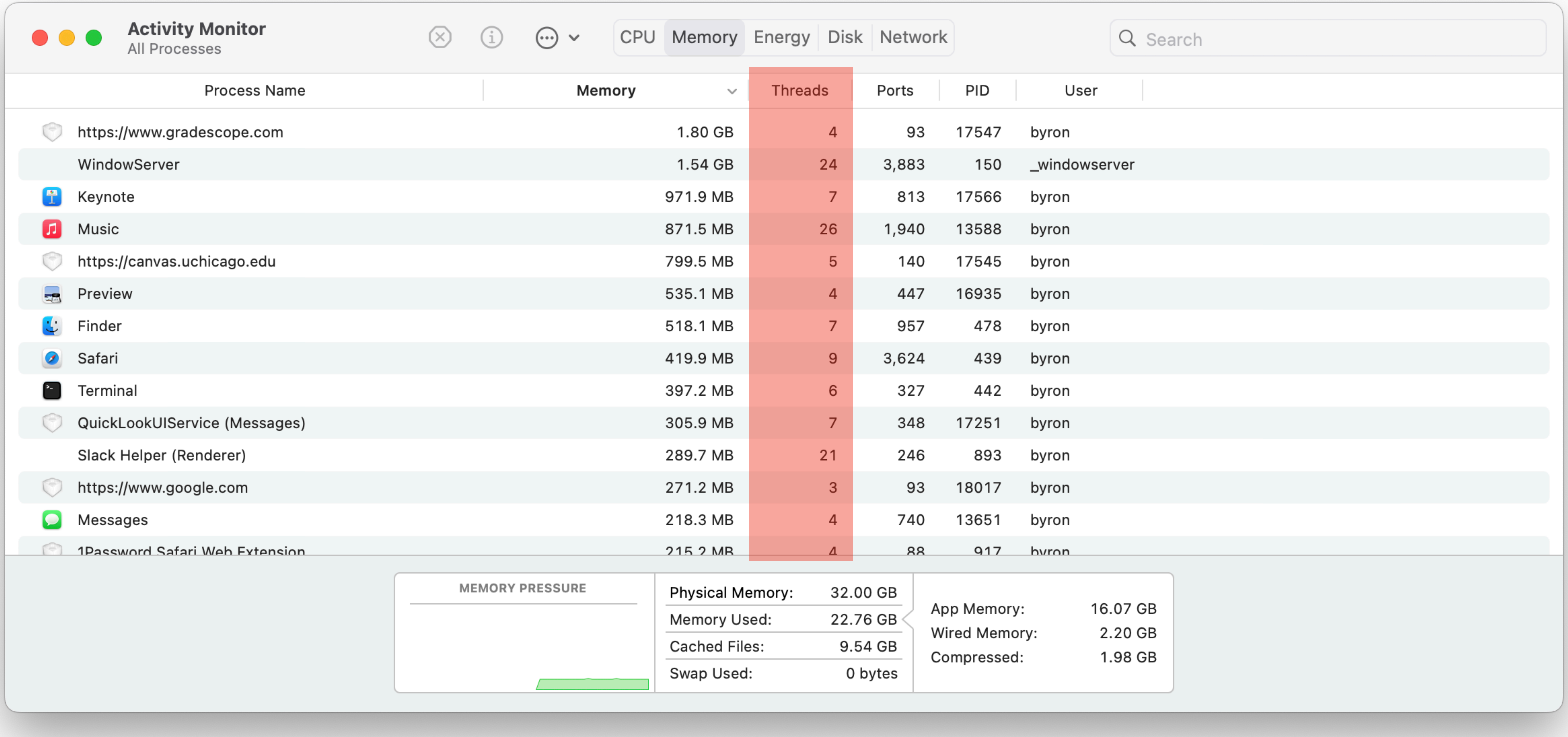


- CPU can do this translation very efficiently
- The chunks of memory used to be called *segments*.
- segmentation fault!

Context Switching

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

Threads



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

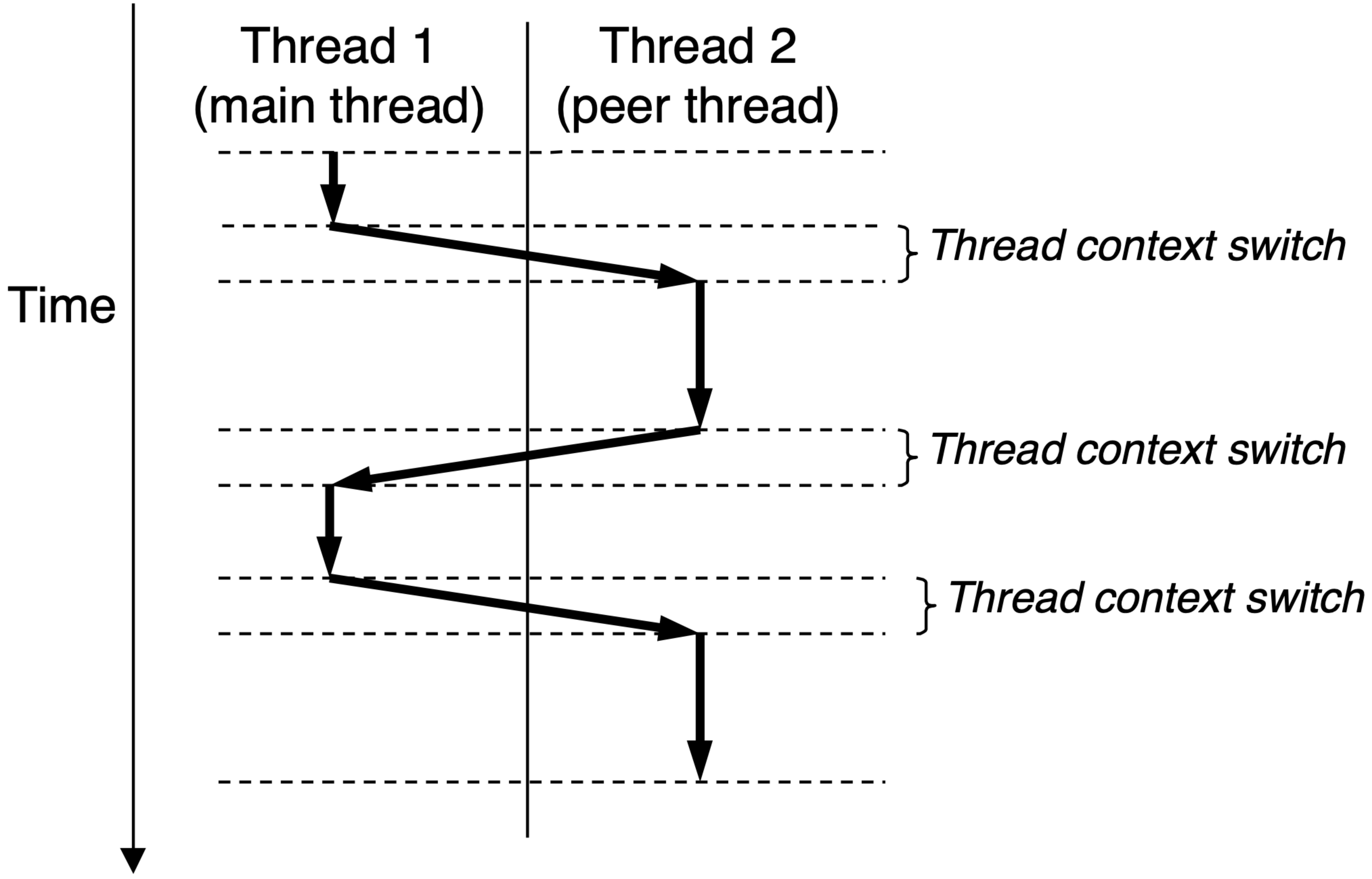
Physical Memory:	32.00 GB
Memory Used:	22.76 GB
Cached Files:	9.54 GB
Swap Used:	0 bytes

App Memory:	16.07 GB
Wired Memory:	2.20 GB
Compressed:	1.98 GB

Threads

- 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


Thread 1
(main thread)
Thread 2
(peer thread)
Time
Thread context switch
Thread context switch
Thread context switch

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

```c
#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

```c
#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

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

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

Threads

Creation

Thread attribute.
Almost always just
`NULL`

```
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);
```

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);
```