Today: what you need to know for Project 2

Compiling, Linking, Loading

- Where are my variables?
- How are programs loaded into memory?

Virtual Memory Basics

- How do addresses get checked and how do they get mapped?
- What happens on a context-switch?

Accessing Information

All information a program reads/writes is stored somewhere in memory

Programmer uses symbolic names:

- local variables, global variables, assembly constants

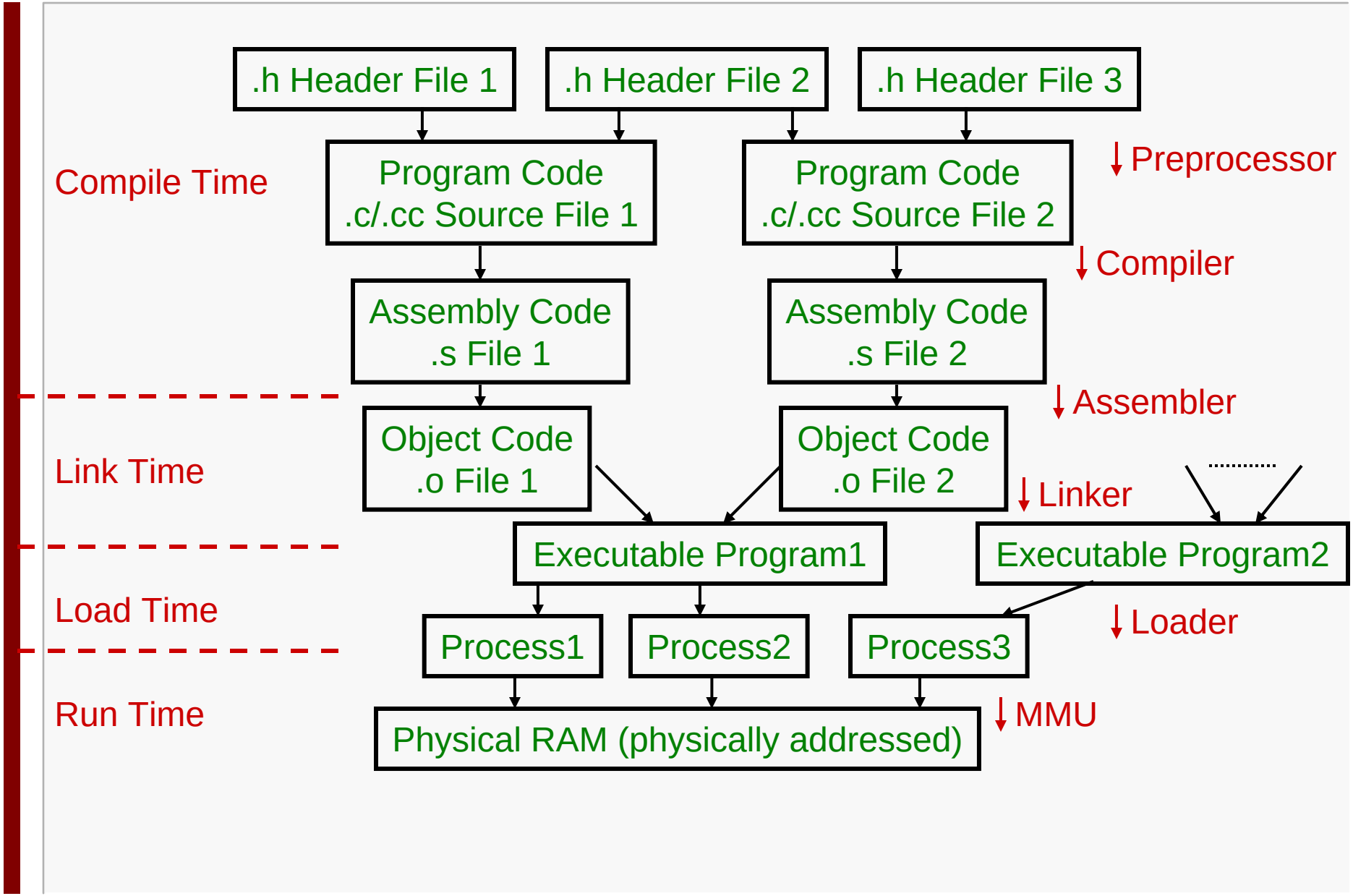
CPU instructions use virtual addresses:

- absolute addresses (at 0xC0000024)
- relative addresses (at $esp – 16)

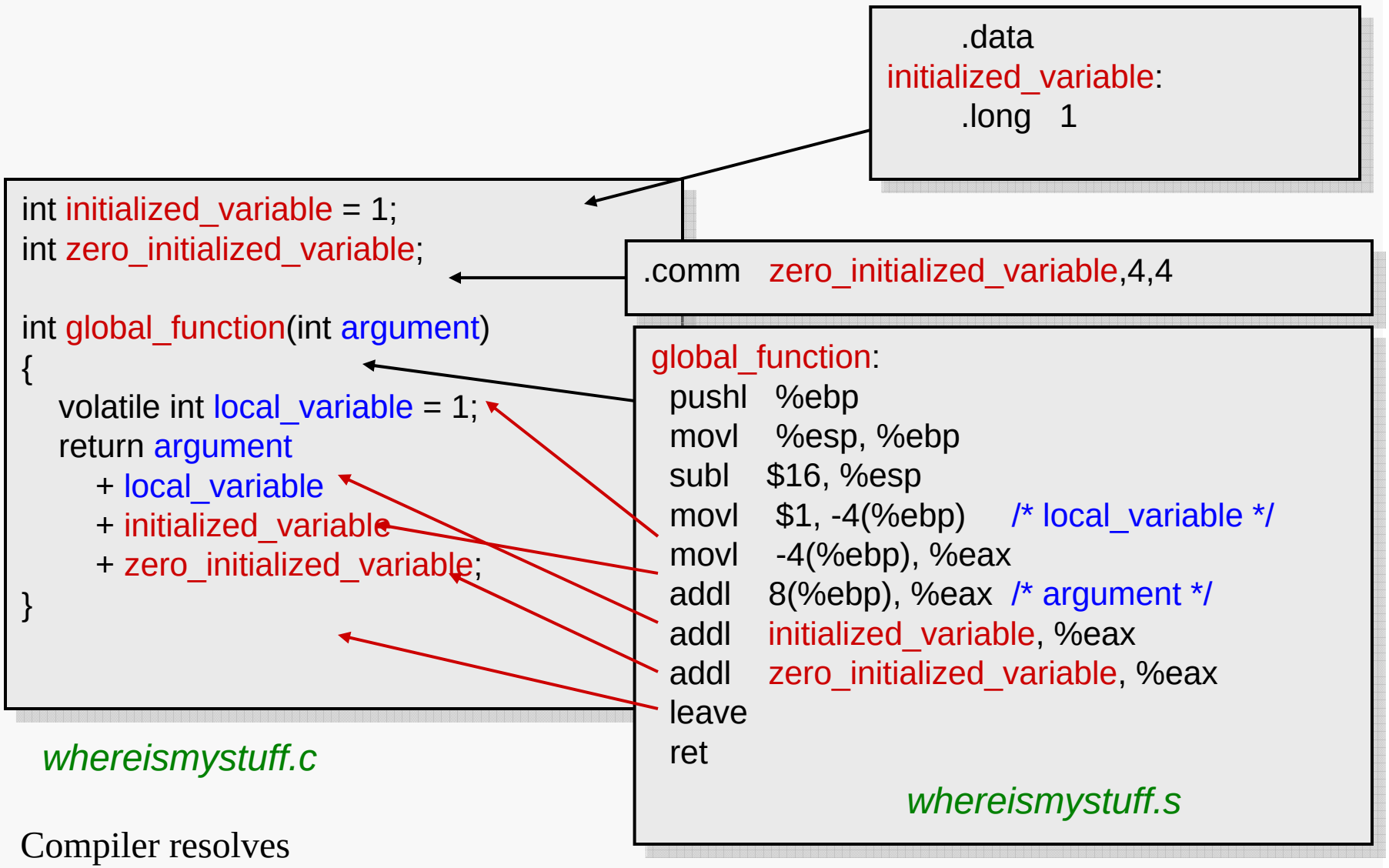
Actual memory uses physical addresses:

Big Question: who does the translation & when?

The Big Picture



Step 1: Compilation



Compiler resolves

local variable names (and struct field offsets!)

Step 2: Assembly

```
global_function:
  pushl   %ebp
  movl    %esp, %ebp
  subl    $16, %esp
  movl    $1, -4(%ebp)      /* local_varia
  movl    -4(%ebp), %eax
  addl    8(%ebp), %eax  /* argument
  addl    initialized_variable, %eax
```

```
RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
00000015 R_386_32          initialized_variable
0000001b R_386_32          zero_initialized_variable

Contents of section .data:
 0000 01000000                             ....
```

```
00000000 <global_function>:
   0:   55                      push %ebp
   1:   89 e5                   mov  %esp,%ebp
   3:   83 ec 10                sub  $0x10,%esp
   6:   c7 45 fc 01 00 00 00    movl $0x1,0xfffffffc(%ebp)
   d:   8b 45 fc                mov  0xfffffffc(%ebp),%eax
  10:   03 45 08                add  0x8(%ebp),%eax
  13:   03 05 00 00 00 00       add  0x0,%eax  // initialized_variable
  19:   03 05 00 00 00 00       add  0x0,%eax  // zero_initialized_variable
  1f:   c9                      leave
  20:   c3                      ret
```

whereismystuff.o

Step 3: Linking

```
0804837c <global_function>:
 804837c: 55                            push   %ebp
 804837d: 89 e5                         mov    %esp,%ebp
 804837f: 83 ec 10                      sub    $0x10,%esp
 8048382: c7 45 fc 01 00 00 00          movl   $0x1,0xfffffffc(%ebp)
 8048389: 8b 45 fc                      mov    0xfffffffc(%ebp),%eax
 804838c: 03 45 08                      add    0x8(%ebp),%eax
 804838f: 03 05 80 95 04 08             add    0x8049580,%eax
 8048395: 03 05 88 95 04 08             add    0x8049588,%eax
 804839b: c9                            leave
 804839c: c3                            ret
```

whereismystuff (.exe)

Linker resolves global addresses, stores instructions for loader in executable

- Key: linker links multiple, independently assembled .o files into executable
- Must decide on layout & then assign addresses & relocate

Step 4: Loading (Conceptual)

ELF Header:
-start address 0x080482d8

BSS:
- size & start of zero initialized data

Data:

```
Contents of section .data:
 8049574 00000000 00000000 88940408 01000000  ................
```

Code:

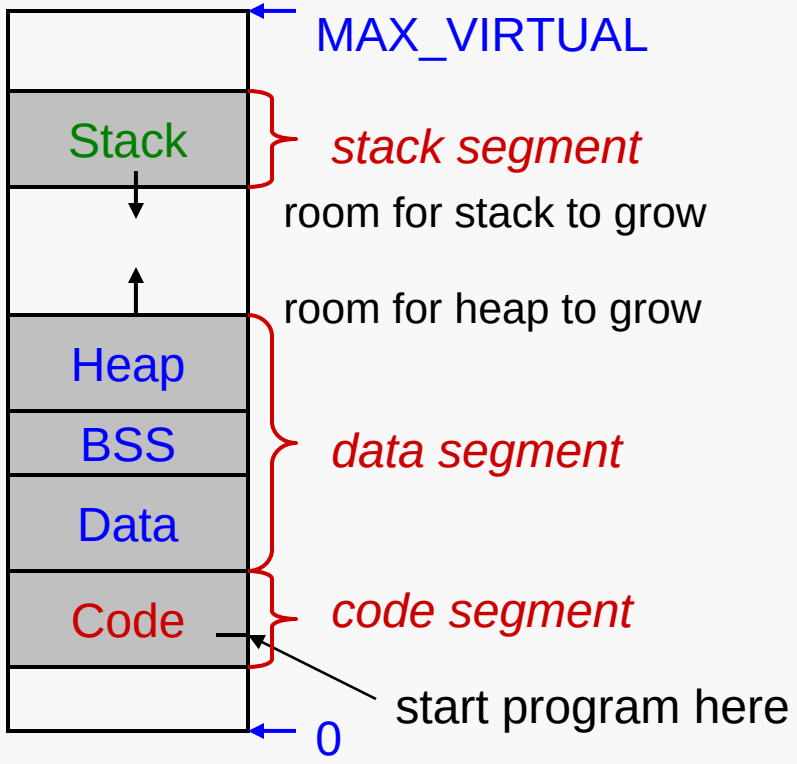
```
0804837c <global_function>:
 804837c: 55                    push   %ebp
 804837d: 89 e5                 mov    %esp,%ebp
 804837f: 83 ec 10              sub    $0x10,%esp
 8048382: c7 45 fc 01 00 00 00  movl   $0x1,0xfffffffc(%ebp)
 8048389: 8b 45 fc              mov    0xfffffffc(%ebp),%eax
 804838c: 03 45 08              add    0x8(%ebp),%eax
 804838f: 03 05 80 95 04 08     add    0x8049580,%eax
 8048395: 03 05 88 95 04 08     add    0x8049588,%eax
 804839b: c9                    leave
 804839c: c3                    ret
```

Executable = set of instructions stored on disk for loader

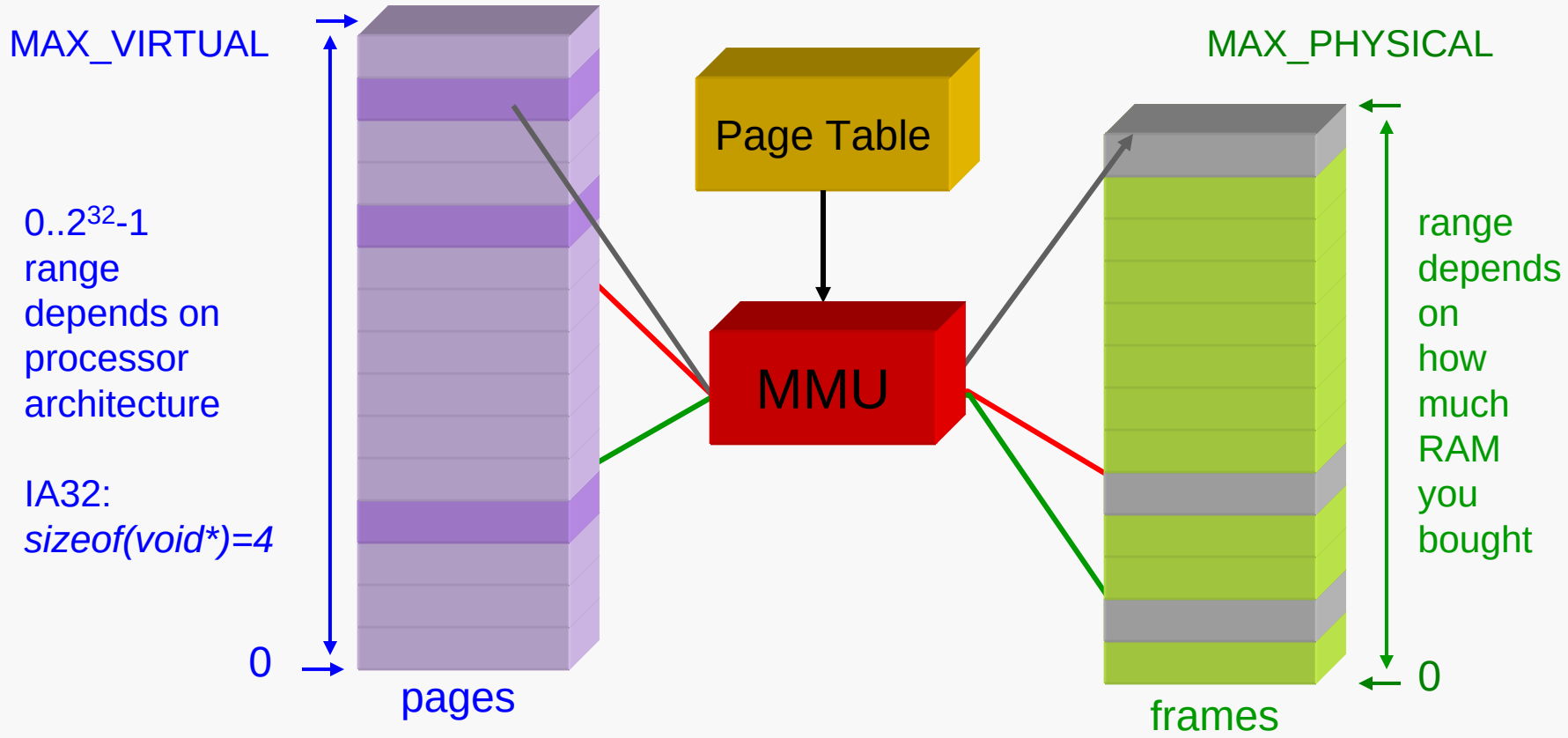
- consists of sections

Picture compiler & linker had in mind when building executable

- sections become segments



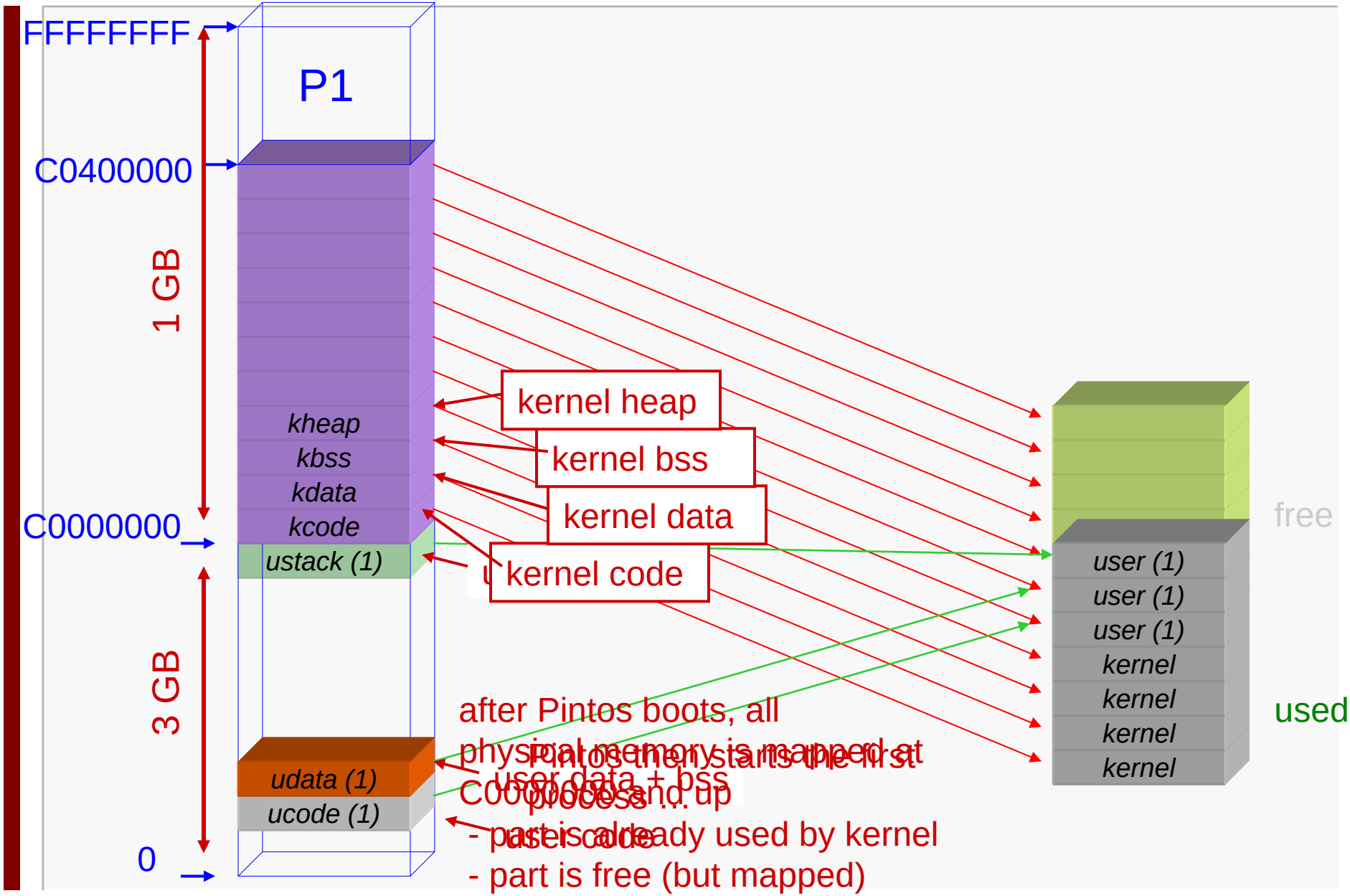
Virtual Memory & Paging (Simplified)



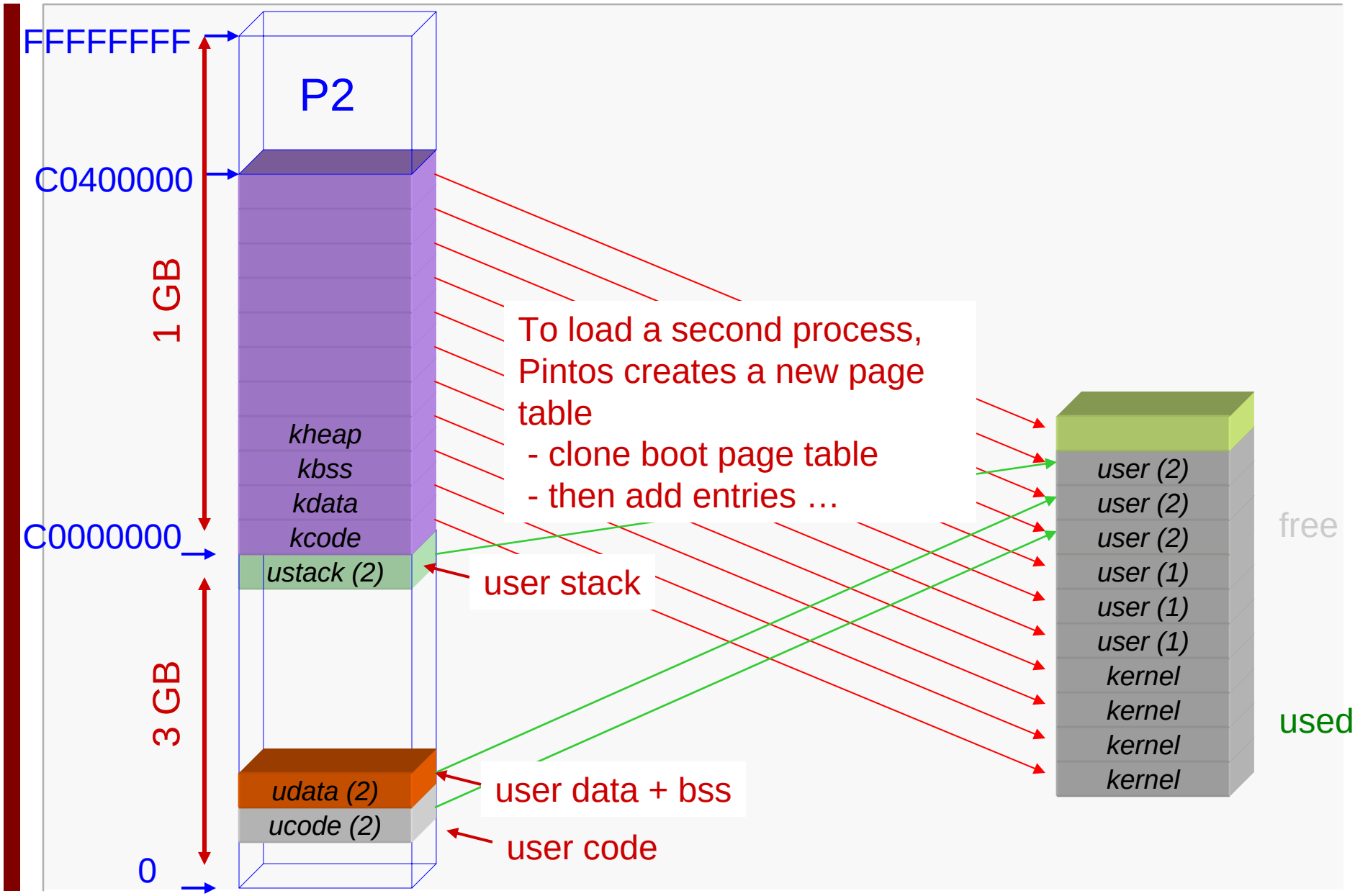
Maps virtual addresses to physical addresses (indirection)

- x86: page table is called *page directory* (one per process)
- mapping at page granularity (x86: 1 page = 4 KB = 4,096 bytes)

Step 5: Loading (For Real)



Step 5: Loading (2nd Process)



Context Switching

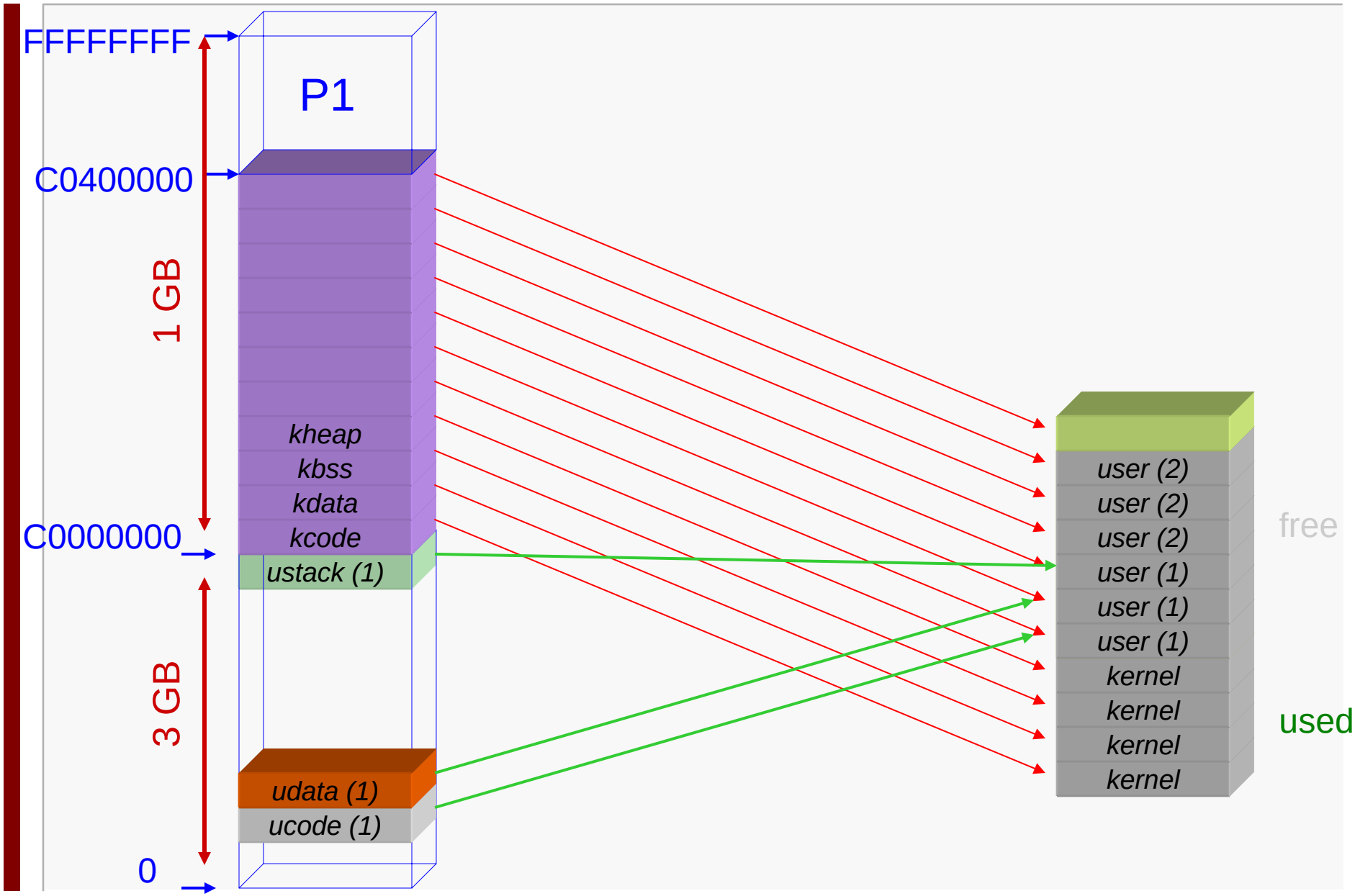
Each process has its own *address space*

- This means that the meaning of addresses (say 0x08c4000) may be different depending on which process is active
- Maps to different page in physical memory

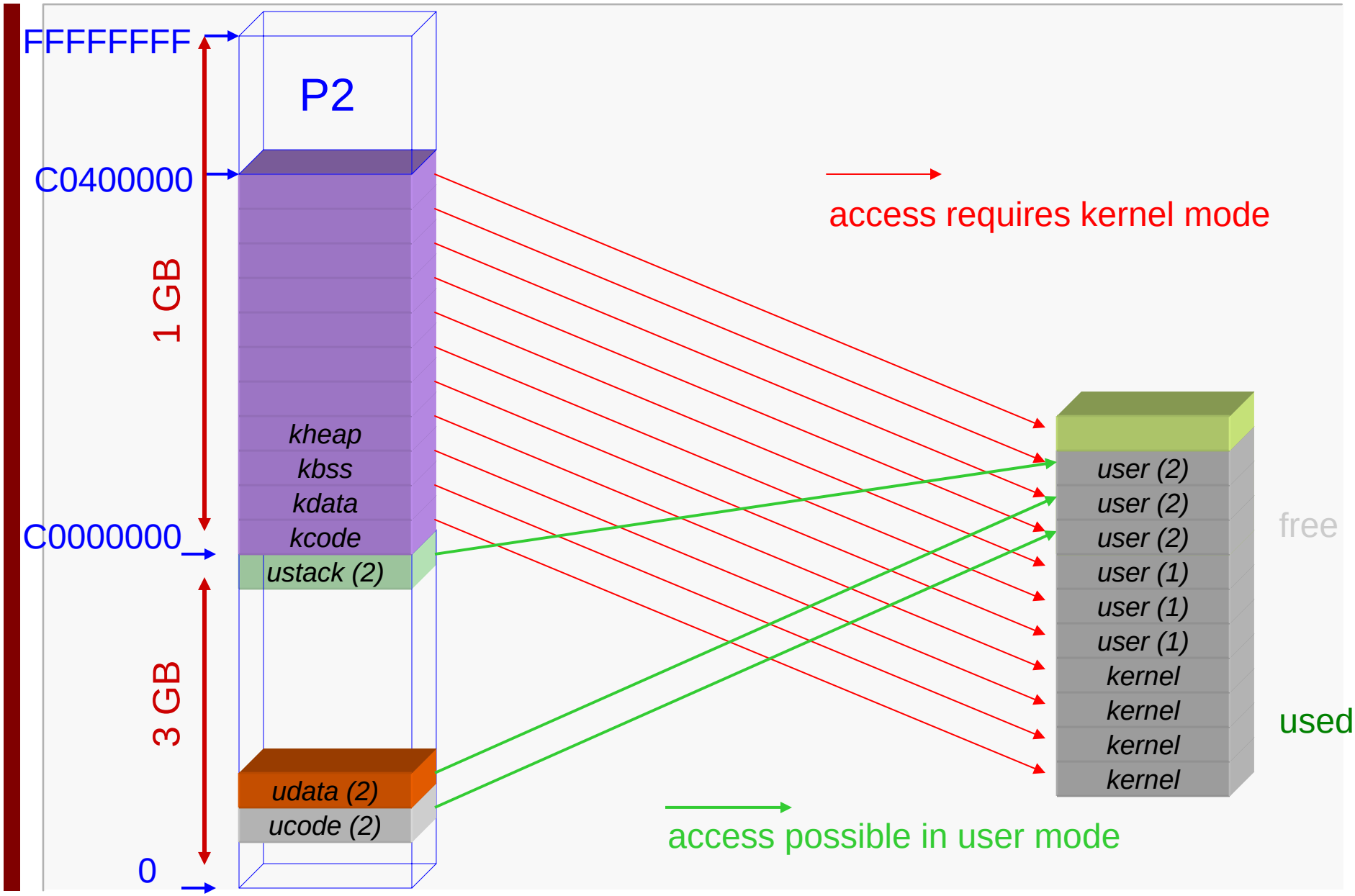
When processes switch, address spaces must switch

- MMU must be reprogrammed

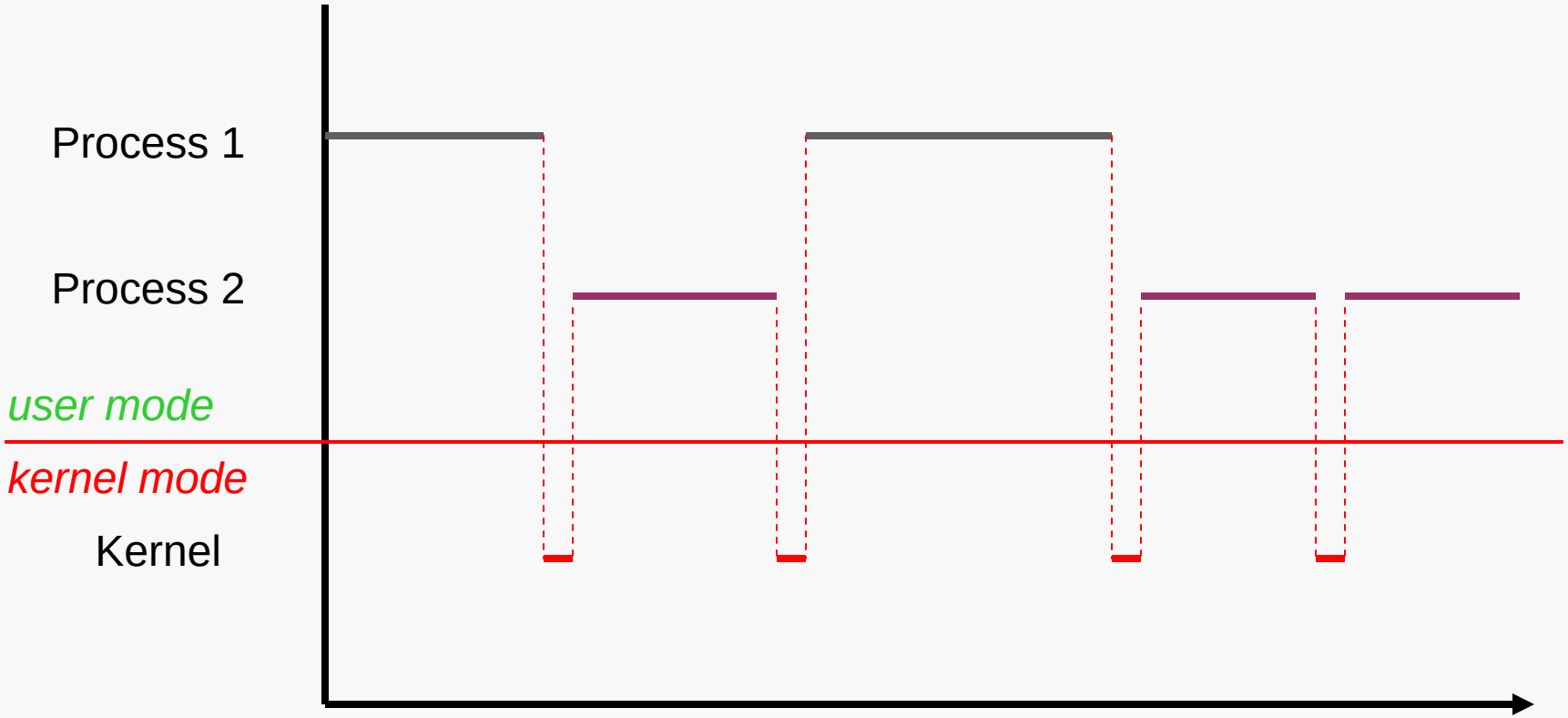
Process 1 Active



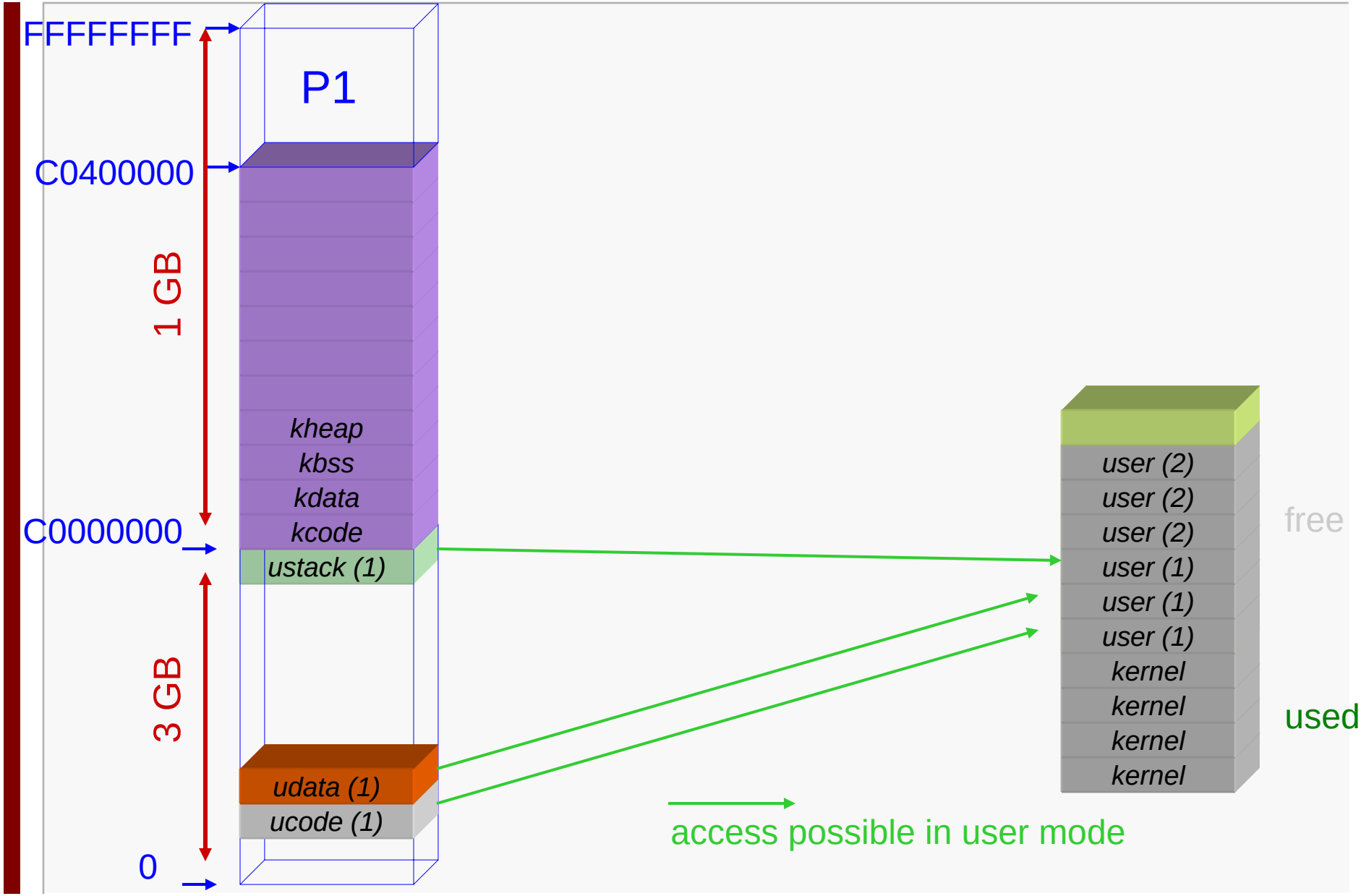
Process 2 Active



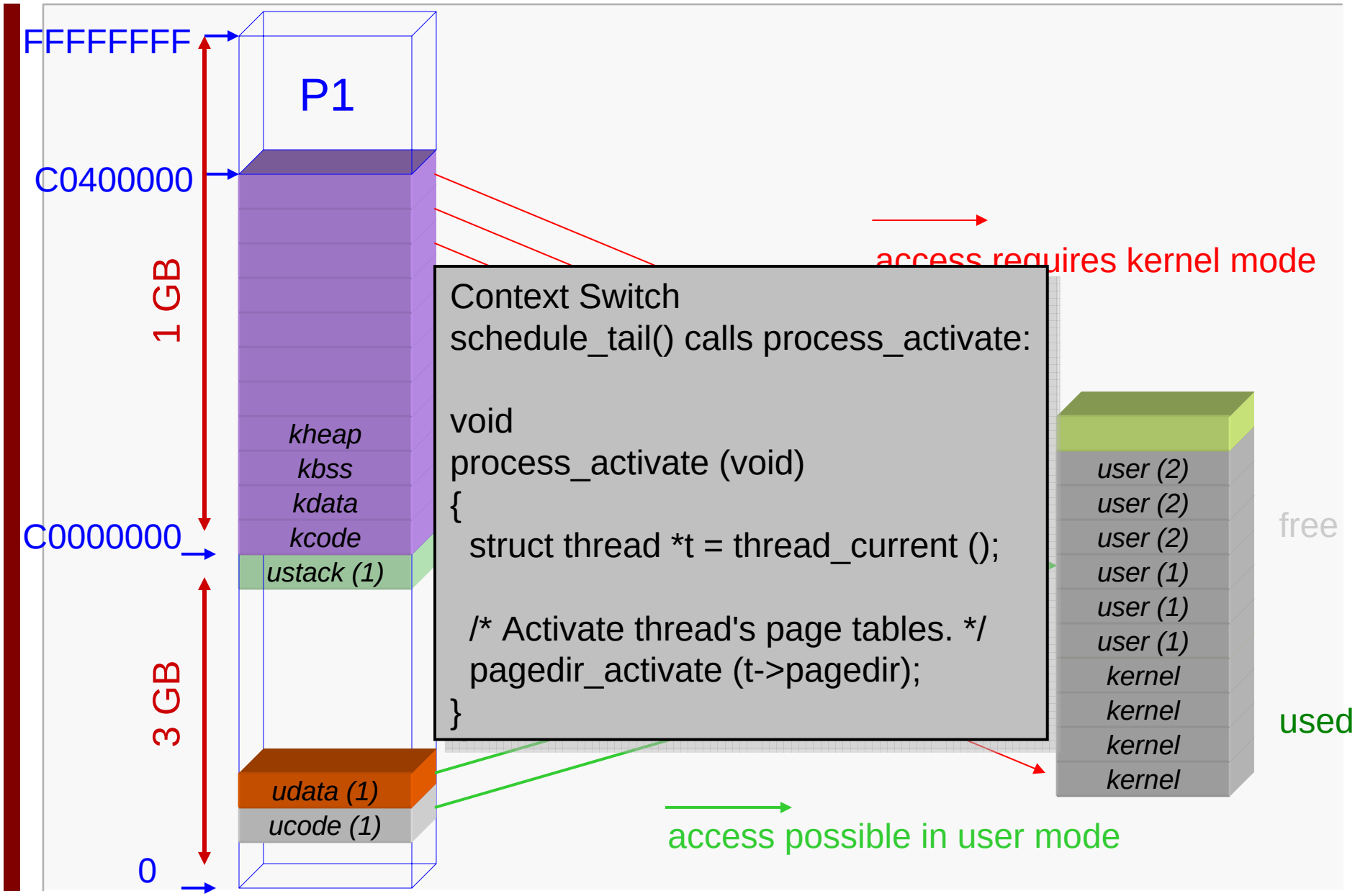
Context Switching



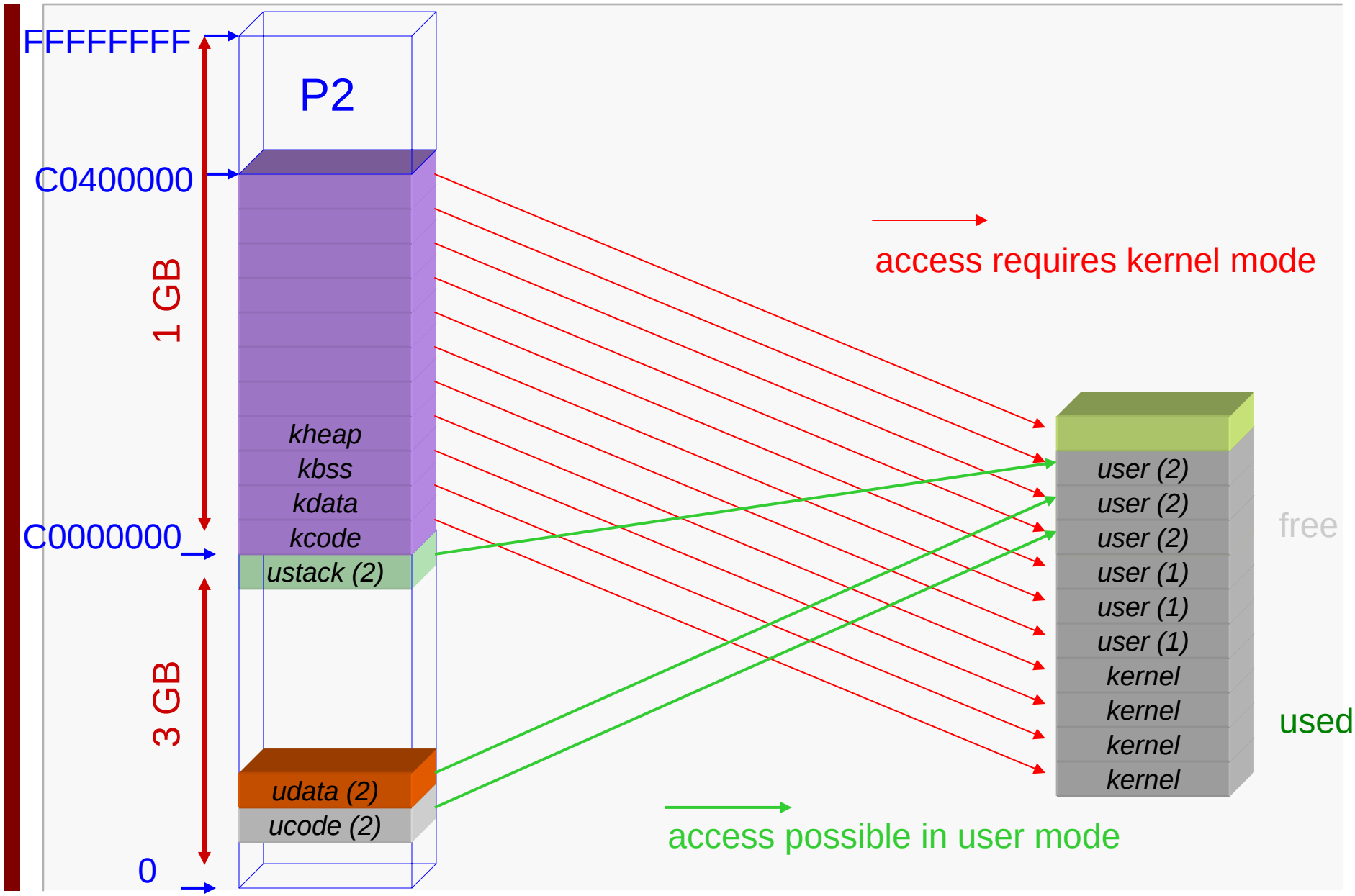
Process 1 Active in user mode



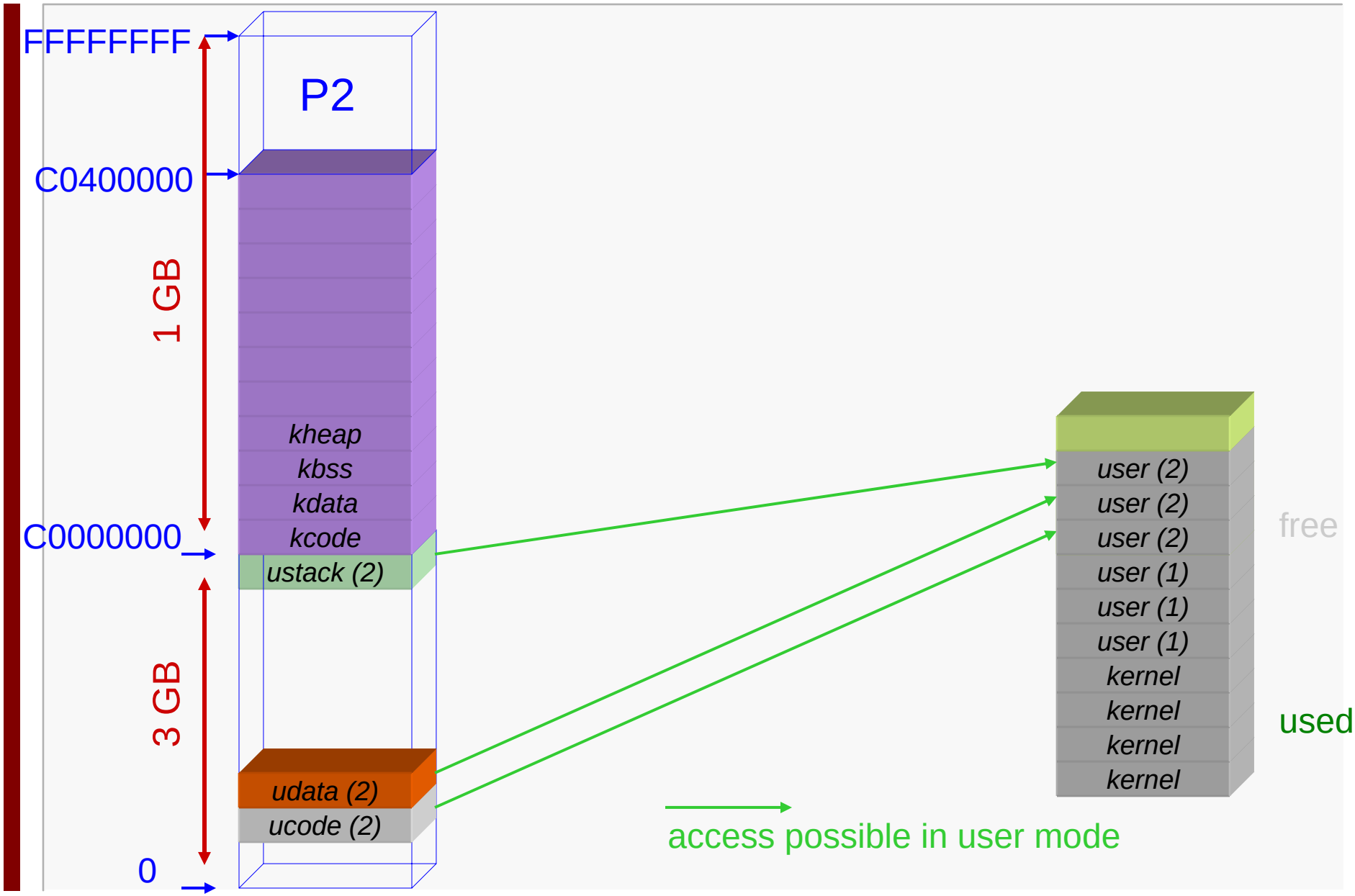
Process 1 Active in kernel mode



Process 2 Active in kernel mode



Process 2 Active in user mode



Summary

All you have to do in Project 2/3/4 is:

- Be aware of what memory is currently accessible
  - Your kernel executes in kernel mode, so you can access all memory (if you use >C000000 addresses)
  - But you must set it up such that your programs can run with the memory you allow them to access (at <C0000000)

Don't let user programs fool you into accessing other processes' (or arbitrary kernel) memory

- Kill them if they try

Keep track of where *stuff* is

- Virtually (toolchain's view):
  - below C0000000 (PHYS_BASE) user space
  - above C0000000 (PHYS_BASE) kernel space
- Physically (note: "physical" addresses are represented as 0xC0000000 + phys_addr in Pintos b/c of permanent mapping)