Operating Systems: Memory Management

Raymond Namyst

Dept. of Computer Science

University of Bordeaux, France

https://gforgeron.gitlab.io/se/

Back to good old times

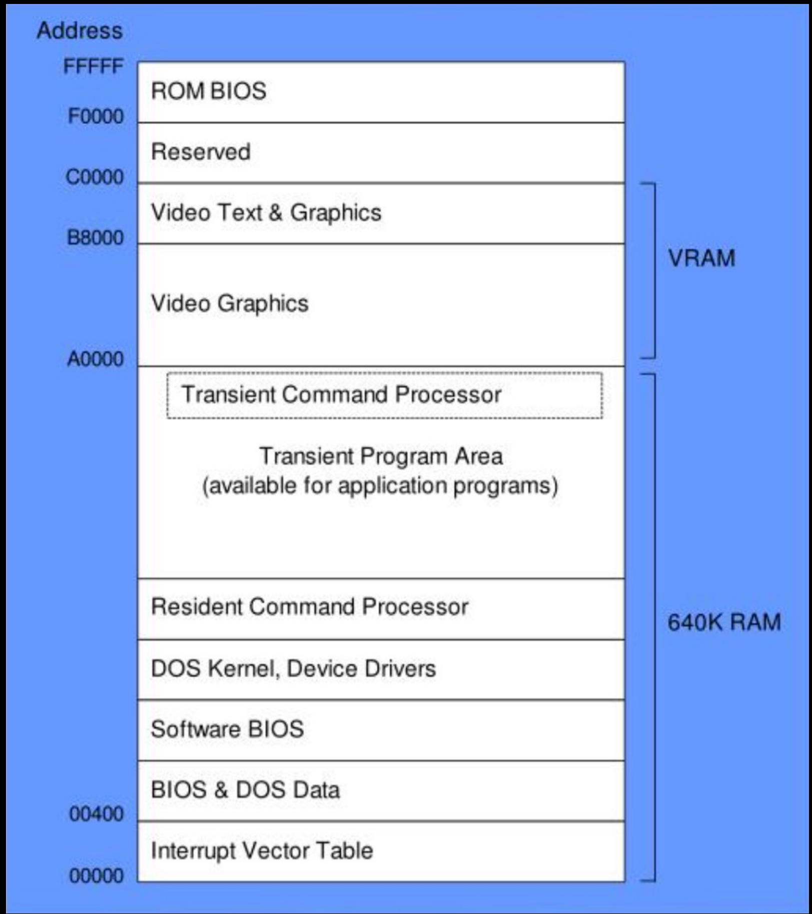
- Only one process at a time was loaded in memory
  - OS resides in a specific part of RAM
  - The other part can host a user process

  - No need for any sophisticated memory management on the OS side

  - Programs starting address is expected to be known at compile time
    - That's what your Computer Architecture teacher told you, uh? ☺

MS-DOS (ex IBM PC DOS)

- Single task OS
- Max 1 MB of RAM
- 16 bits “real” addressing
  - No protection
  - Even the interrupt Vector Table can be modified by user programs
  - Sounds weird that we can use more than 64KB… 🤔



MS-DOS (ex IBM PC DOS)

- Funnily enough…
  - ..OS routines were “portably” reached through interrupt multiplexers
    - int 08h — Timer interrupt
    - int 10h — Video services
    - Int 16h — Keyboard services
    - int 21h — MS-DOS services
- Example: PutChar (’A’)

MS-DOS (ex IBM PC DOS)

- Funnily enough...
  - ..OS routines were “portably” reached through interrupt multiplexers
    - int 08h — Timer interrupt
    - int 10h — Video services
    - Int 16h — Keyboard services
    - int 21h — MS-DOS services
- Example: PutChar (’A’)

```
mov ah, 02h ; SC_PutChar == 0x02
mov dl, 'A'
int 21h
```

MS-DOS (ex IBM PC DOS)

- Funnily enough…
  - ..OS routines were "portably" reached through interrupt multiplexers
    - int 08h — Timer interrupt
    - int 10h — Video services
    - Int 16h — Keyboard services
    - int 21h — MS-DOS services

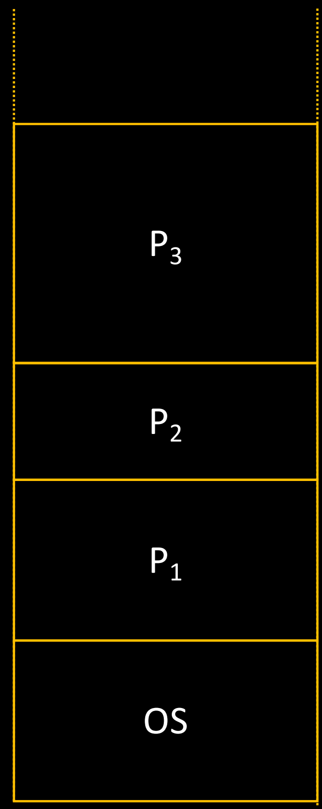
- Example: PutChar ('A')

  Exit (0)

```
mov ah, 02h ; SC_PutChar == 0x02
mov dl, 'A'
int 21h

mov ah, 4Ch ; SC_Exit == 0x4C
mov al, 0   ; EXIT_SUCCESS
int 21h
```

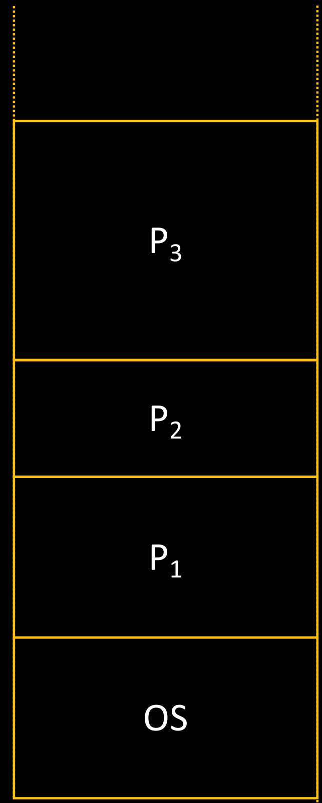
Towards Multiprogramming

- What was the reason for introducing multitasking in Operating Systems?
  - i.e. allowing multiple processes to simultaneously stay in memory



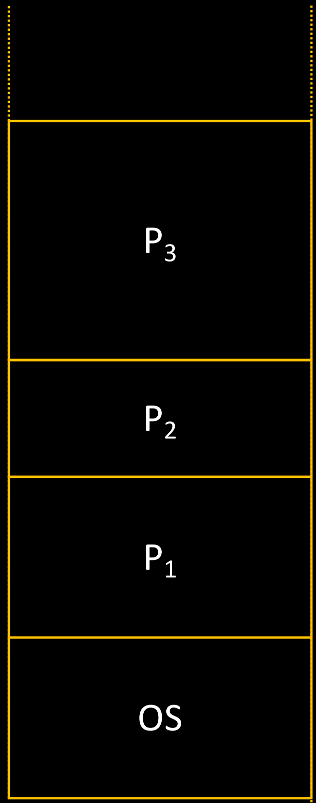
Towards Multiprogramming

- What was the reason for introducing multitasking in Operating Systems?
  - i.e. allowing multiple processes to simultaneously stay in memory
- Money!
  - Processes spend a significant time in I/O operations
    - With tape drives, it took time…
    - CPU idleness costs a lot
- Let P be the (average) ratio of I/O time
  - P = probability to be idle
  - By using n processes, the probability of the CPU being idle is $1 - P^n$



Towards Multiprogramming

- With great power comes great complications!
  - How to compile processes even if we don't know at which address they will be placed?
  - How to address memory fragmentation?
  - How to let processes grow?
  - How to enforce memory protection?



Towards Multiprogramming

- With great power comes great complications!
  - How to compile processes even if we don't know at which address they will be placed?

```
int a = 5;
int b = 31;
int c = 0;

  c = a + b;
```

Towards Multiprogramming

- With great power comes great complications!
  - How to compile processes even if we don't know at which address they will be placed?
    - Illustration with y86 code

https://dept-info.labri.fr/ENSEIGNEMENT/archi/js-y86/

```
                          |
0x0000:                   | .pos 0
0x0000:                   | Init:
0x0000: 30f570010000      |     irmovl Stack, %ebp
0x0006: 30f470010000      |     irmovl Stack, %esp
0x000c: 500864010000      |     mrmovl a, %eax
0x0012: 501868010000      |     mrmovl b, %ecx
0x0018: 6010              |     addl %ecx, %eax
0x001a: 40086c010000      |     rmmovl %eax, c
0x0020: 10                |     halt
0x0021:                   | .pos 356
0x0164:                   | a:
0x0164: 05000000          |     .long 5
0x0168:                   | b:
0x0168: 1f000000          |     .long 31
0x016c:                   | c:
0x016c: 00000000          |     .long 0
0x0170:                   | Stack:
                          |
```

Towards Multiprogramming

- With great power comes great complications!
  - How to compile processes even if we don't know at which address they will be placed?
    - Illustration with y86 code

```
                           |
0x0000:                    | .pos 0
0x0000:                    | Init:
0x0000: 30f570010000 |     irmovl Stack, %ebp
0x0006: 30f470010000 |     irmovl Stack, %esp
0x000c: 500864010000 |     mrmovl a, %eax
0x0012: 501868010000 |     mrmovl b, %ecx
0x0018: 6010               |     addl %ecx, %eax
0x001a: 40086c010000 |     rmmovl %eax, c
0x0020: 10                 |     halt
0x0021:                    | .pos 356
0x0164:                    | a:
0x0164: 05000000           |     .long 5
0x0168:                    | b:
0x0168: 1f000000           |     .long 31
0x016c:                    | c:
0x016c: 00000000           |     .long 0
0x0170:                    | Stack:
                           |
```

Towards Multiprogramming

- With great power comes great complications!
  - How to compile processes even if we don’t know at which address they will be placed?
    - Illustration with y86 code

```
                              |
0x0000:                       | .pos 0
0x0000:                       | Init:
0x0000: 30f570010000 |     irmovl Stack, %ebp
0x0006: 30f470010000 |     irmovl Stack, %esp
0x000c: 500864010000 |     mrmovl a, %eax
0x0012: 501868010000 |     mrmovl b, %ecx
0x0018: 6010                  |     addl %ecx, %eax
0x001a: 40086c010000 |     rmmovl %eax, c
0x0020: 10                    |     halt
0x0021:                       | .pos 356
0x0164:                       | a:
0x0164: 05000000          |     .long 5
0x0168:                       | b:
0x0168: 1f000000          |     .long 31
0x016c:                       | c:
0x016c: 00000000          |     .long 0
0x0170:                       | Stack:
                              |
```

Towards Multiprogramming

- With great power comes great complications!
  - How to compile processes even if we don't know at which address they will be placed?
    - Illustration with y86 code

    - This code assumes that it will be placed at address 0…
      - Otherwise, it wouldn't work

```
                        |
0x0000:                 | .pos 0
0x0000:                 | Init:
0x0000: 30f570010000 |     irmovl Stack, %ebp
0x0006: 30f470010000 |     irmovl Stack, %esp
0x000c: 500864010000 |     mrmovl a, %eax
0x0012: 501868010000 |     mrmovl b, %ecx
0x0018: 6010          |     addl %ecx, %eax
0x001a: 40086c010000 |     rmmovl %eax, c
0x0020: 10            |     halt
0x0021:               | .pos 356
0x0164:               | a:
0x0164: 05000000      |     .long 5
0x0168:               | b:
0x0168: 1f000000      |     .long 31
0x016c:               | c:
0x016c: 00000000      |     .long 0
0x0170:               | Stack:
                        |
```

Towards Multiprogramming

- So, what shall we do if the program is loaded at address 0x100 ?
  - At <u>load time</u>, we must change
    - 70 01 00 00 -> 70 02 00 00
      - At 2 places
    - 64 01 00 00 -> 64 02 00 00
    - 68 01 00 00 -> 68 02 00 00
    - 6c 01 00 00 -> 6c 02 00 00

```
                         |
0x0000:                  | .pos 0
0x0000:                  | Init:
0x0000: 30f570010000     |     irmovl Stack, %ebp
0x0006: 30f470010000     |     irmovl Stack, %esp
0x000c: 500864010000     |     mrmovl a, %eax
0x0012: 501868010000     |     mrmovl b, %ecx
0x0018: 6010             |     addl %ecx, %eax
0x001a: 40086c010000     |     rmmovl %eax, c
0x0020: 10               |     halt
0x0021:                  | .pos 356
0x0164:                  | a:
0x0164: 05000000         |     .long 5
0x0168:                  | b:
0x0168: 1f000000         |     .long 31
0x016c:                  | c:
0x016c: 00000000         |     .long 0
0x0170:                  | Stack:
                         |
```

Towards Multiprogramming

- So, what shall we do if the program is loaded at address 0x100 ?
  - At load time, we must change
    - `70 01 00 00 -> 70 02 00 00`
      - At 2 places
    - `64 01 00 00 -> 64 02 00 00`
    - `68 01 00 00 -> 68 02 00 00`
    - `6c 01 00 00 -> 6c 02 00 00`
  - So "*Find & Replace*" and that's it?

```
                          |
0x0000:                   | .pos 0
0x0000:                   | Init:
0x0000: 30f570010000      |     irmovl Stack, %ebp
0x0006: 30f470010000      |     irmovl Stack, %esp
0x000c: 500864010000      |     mrmovl a, %eax
0x0012: 501868010000      |     mrmovl b, %ecx
0x0018: 6010              |     addl %ecx, %eax
0x001a: 40086c010000      |     rmmovl %eax, c
0x0020: 10                |     halt
0x0021:                   | .pos 356
0x0164:                   | a:
0x0164: 05000000          |     .long 5
0x0168:                   | b:
0x0168: 1f000000          |     .long 31
0x016c:                   | c:
0x016c: 00000000          |     .long 0
0x0170:                   | Stack:
                          |
```

Towards Multiprogramming

- So, what shall we do if the program is loaded at address 0x100 ?
  - At <u>load time</u>, we must change
    - `70 01 00 00 -> 70 02 00 00`
      - At 2 places
    - `64 01 00 00 -> 64 02 00 00`
    - `68 01 00 00 -> 68 02 00 00`
    - `6c 01 00 00 -> 6c 02 00 00`
  - ~~So "*Find & Replace*" and that's it?~~

```
                            |
0x0000:                     | .pos 0
0x0000:                     | Init:
0x0000: 30f570010000 |     irmovl Stack, %ebp
0x0006: 30f470010000 |     irmovl Stack, %esp
0x000c: 500864010000 |     mrmovl a, %eax
0x0012: 501868010000 |     mrmovl b, %ecx
0x0018: 6010                |     addl %ecx, %eax
0x001a: 40086c010000 |     rmmovl %eax, c
0x0020: 10                  |     halt
0x0021:                     | .pos 356
0x0164:                     | a:
0x0164: 68010000            |     .long 360
0x0168:                     | b:
0x0168: 1f000000            |     .long 31
0x016c:                     | c:
0x016c: 00000000            |     .long 0
0x0170:                     | Stack:
                            |
```

Towards Multiprogramming

- The compiler generates "relative" references in the code
  - As if the code would start at 0x0
  - The list of these references is included in the binary
- If the program is loaded at 0x100, the loader has to perform
  - val = val + 0x100
    - At 0x0002, 0x0008, 0x000e, 0x0014 and 0x001c

```
                            |
0x0000:                     | .pos 0
0x0000:                     | Init:
0x0000: 30f570010000        |     irmovl Stack, %ebp
0x0006: 30f470010000        |     irmovl Stack, %esp
0x000c: 500864010000        |     mrmovl a, %eax
0x0012: 501868010000        |     mrmovl b, %ecx
0x0018: 6010                |     addl %ecx, %eax
0x001a: 40086c010000        |     rmmovl %eax, c
0x0020: 10                  |     halt
0x0021:                     | .pos 356
0x0164:                     | a:
0x0164: 68010000            |     .long 360
0x0168:                     | b:
0x0168: 1f000000            |     .long 31
0x016c:                     | c:
0x016c: 00000000            |     .long 0
0x0170:                     | Stack:
                            |
```

Code relocation

- For *different purposes*, code relocation also used by today's compilers
  - At compile time (≠ linking), final address of symbols is unknown
    - The compiler builds a list of *relocation entries* to be handled by the linker

```c
int i = 0;
int j = 31;

int f (int x, int y)
{
  return x + y;
}

int main (int argc, char *argv[])
{
  int a;

  if (argc > 1)
    i = atoi (argv[1]);

  a = f (i, j);
  printf ("Result : %d\n", a);

  return 0;
}
```

Code relocation

- For *different purposes*, code relocation also used by today's compilers
  - At compile time (≠ linking), final address of symbols is unknown
    - The compiler builds a list of *relocation entries* to be handled by the linker
  - Address Space Layout Randomization (ASLR)

```c
int i = 0;
int j = 31;

int f (int x, int y)
{
  return x + y;
}

int main (int argc, char *argv[])
{
  int a;

  if (argc > 1)
    i = atoi (argv[1]);

  a = f (i, j);
  printf ("Result : %d\n", a);

  return 0;
}
```

Code relocation (x64)

[mymachine] objdump -d prog.o

```
0000000000000014 <main>:
 14:  55                    push   %rbp
 15:  48 89 e5              mov    %rsp,%rbp
 [...]
 34:  48 89 c7              mov    %rax,%rdi
 37:  e8 00 00 00 00        callq  3c <main+0x28>
 3c:  89 05 00 00 00 00     mov    %eax,0x0(%rip)
 42:  8b 15 00 00 00 00     mov    0x0(%rip),%edx
 48:  8b 05 00 00 00 00     mov    0x0(%rip),%eax
 4e:  89 d6                 mov    %edx,%esi
 50:  89 c7                 mov    %eax,%edi
 52:  e8 00 00 00 00        callq  57 <main+0x43>
 57:  89 45 fc              mov    %eax,-0x4(%rbp)
 5a:  8b 45 fc              mov    -0x4(%rbp),%eax
 5d:  89 c6                 mov    %eax,%esi
 5f:  48 8d 3d 00 00 00 00  lea    0x0(%rip),%rdi
 66:  b8 00 00 00 00        mov    $0x0,%eax
 6b:  e8 00 00 00 00        callq  70 <main+0x5c>
 70:  b8 00 00 00 00        mov    $0x0,%eax
 75:  c9                    leaveq
 76:  c3                    retq
```

[mymachine] readelf -r prog.o

```
Section de réadressage '.rela.text' à l'adresse de décalage 0x2f8 contient 7 entrées:
 Décalage      Info          Type            Val.-symboles     Noms-symb.+ Addenda
000000000038  000e00000004 R_X86_64_PLT32  0000000000000000 atoi - 4
00000000003e  000900000002 R_X86_64_PC32   0000000000000000 i - 4
000000000044  000a00000002 R_X86_64_PC32   0000000000000000 j - 4
00000000004a  000900000002 R_X86_64_PC32   0000000000000000 i - 4
000000000053  000b00000004 R_X86_64_PLT32  0000000000000000 f - 4
000000000062  000500000002 R_X86_64_PC32   0000000000000000 .rodata - 4
00000000006c  000f00000004 R_X86_64_PLT32  0000000000000000 printf - 4
```

Code relocation (x64)

[mymachine] objdump -d prog.o

```
0000000000000014 <main>:
 14:   55                    push  %rbp
 15:   48 89 e5              mov   %rsp,%rbp
 [...]
 34:   48 89 c7              mov   %rax,%rdi
 37:   e8 00 00 00 00        callq 3c <main+0x28>
 3c:   89 05 00 00 00 00     mov   %eax,0x0(%rip)
 42:   8b 15 00 00 00 00     mov   0x0(%rip),%edx
 48:   8b 05 00 00 00 00     mov   0x0(%rip),%eax
 4e:   89 d6                 mov   %edx,%esi
 50:   89 c7                 mov   %eax,%edi
 52:   e8 00 00 00 00        callq 57 <main+0x43>
 57:   89 45 fc              mov   %eax,-0x4(%rbp)
 5a:   8b 45 fc              mov   -0x4(%rbp),%eax
 5d:   89 c6                 mov   %eax,%esi
 5f:   48 8d 3d 00 00 00 00  lea   0x0(%rip),%rdi
 66:   b8 00 00 00 00        mov   $0x0,%eax
 6b:   e8 00 00 00 00        callq 70 <main+0x5c>
 70:   b8 00 00 00 00        mov   $0x0,%eax
 75:   c9                    leaveq
 76:   c3                    retq
```

[mymachine] readelf -r prog.o

```
Section de réadressage '.rela.text' à l'adresse de décalage 0x2f8 contient 7 entrées:
 Décalage      Info          Type            Val.-symboles Noms-symb.+ Addenda
000000000038  000e00000004 R_X86_64_PLT32  0000000000000000 atoi - 4
00000000003e  000900000002 R_X86_64_PC32   0000000000000000 i - 4
000000000044  000a00000002 R_X86_64_PC32   0000000000000000 j - 4
00000000004a  000900000002 R_X86_64_PC32   0000000000000000 i - 4
000000000053  000b00000004 R_X86_64_PLT32  0000000000000000 f - 4
000000000062  000500000002 R_X86_64_PC32   0000000000000000 .rodata - 4
00000000006c  000f00000004 R_X86_64_PLT32  0000000000000000 printf - 4
```

We don’t know yet the address of ‘i’

Code relocation (x64)

[mymachine] objdump -d prog.o

```
0000000000000014 <main>:
 14:   55                   push  %rbp
 15:   48 89 e5             mov   %rsp,%rbp
 [...]
 34:   48 89 c7             mov   %rax,%rdi
 37:   e8 00 00 00 00       callq  3c <main+0x28>
 3c:   89 05 00 00 00 00    mov   %eax,0x0(%rip)
 42:   8b 15 00 00 00 00    mov   0x0(%rip),%edx
 48:   8b 05 00 00 00 00    mov   0x0(%rip),%eax
 4e:   89 d6                mov   %edx,%esi
 50:   89 c7                mov   %eax,%edi
 52:   e8 00 00 00 00       callq  57 <main+0x43>
 57:   89 45 fc             mov   %eax,-0x4(%rbp)
 5a:   8b 45 fc             mov   -0x4(%rbp),%eax
 5d:   89 c6                mov   %eax,%esi
 5f:   48 8d 3d 00 00 00 00           lea   0x0(%rip),%rdi
 66:   b8 00 00 00 00       mov   $0x0,%eax
 6b:   e8 00 00 00 00       callq  70 <main+0x5c>
 70:   b8 00 00 00 00       mov   $0x0,%eax
 75:   c9                   leaveq
 76:   c3                   retq
```

[mymachine] readelf -r prog.o

```
Section de réadressage '.rela.text' à l'adresse de décalage 0x2f8 contient 7 entrées:
 Décalage     Info      Type        Val.-symboles Noms-symb.+ Addenda
000000000038  000e00000004 R_X86_64_PLT32   0000000000000000 atoi - 4
00000000003e  000900000002 R_X86_64_PC32    0000000000000000 i - 4
000000000044  000a00000002 R_X86_64_PC32    0000000000000000 j - 4
00000000004a  000900000002 R_X86_64_PC32    0000000000000000 i - 4
000000000053  000b00000004 R_X86_64_PLT32   0000000000000000 f - 4
000000000062  000500000002 R_X86_64_PC32    0000000000000000 .rodata - 4
00000000006c  000f00000004 R_X86_64_PLT32   0000000000000000 printf - 4
```

We don't know yet the address of 'j'

Code relocation (x64)

[mymachine] objdump -d prog.o

```
0000000000000014 <main>:
 14:   55                    push   %rbp
 15:   48 89 e5              mov    %rsp,%rbp
 [...]
 34:   48 89 c7              mov    %rax,%rdi
 37:   e8 00 00 00 00        callq  3c <main+0x28>
 3c:   89 05 00 00 00 00     mov    %eax,0x0(%rip)
 42:   8b 15 00 00 00 00     mov    0x0(%rip),%edx
 48:   8b 05 00 00 00 00     mov    0x0(%rip),%eax
 4e:   89 d6                 mov    %edx,%esi
 50:   89 c7                 mov    %eax,%edi
 52:   e8 00 00 00 00        callq  57 <main+0x43>
 57:   89 45 fc              mov    %eax,-0x4(%rbp)
 5a:   8b 45 fc              mov    -0x4(%rbp),%eax
 5d:   89 c6                 mov    %eax,%esi
 5f:   48 8d 3d 00 00 00 00  lea    0x0(%rip),%rdi
 66:   b8 00 00 00 00        mov    $0x0,%eax
 6b:   e8 00 00 00 00        callq  70 <main+0x5c>
 70:   b8 00 00 00 00        mov    $0x0,%eax
 75:   c9                    leaveq
 76:   c3                    retq
```

We don’t even know the address of function ‘f’

[mymachine] readelf -r prog.o

```
Section de réadressage '.rela.text' à l'adresse de décalage 0x2f8 contient 7 entrées:
 Décalage      Info          Type           Val.-symboles    Noms-symb.+ Addenda
000000000038  000e00000004 R_X86_64_PLT32  0000000000000000 atoi - 4
00000000003e  000900000002 R_X86_64_PC32   0000000000000000 i - 4
000000000044  000a00000002 R_X86_64_PC32   0000000000000000 j - 4
00000000004a  000900000002 R_X86_64_PC32   0000000000000000 i - 4
000000000053  000b00000004 R_X86_64_PLT32  0000000000000000 f - 4
000000000062  000500000002 R_X86_64_PC32   0000000000000000 .rodata - 4
00000000006c  000f00000004 R_X86_64_PLT32  0000000000000000 printf - 4
```

Code relocation (x64)

[mymachine] objdump -d prog.o

```
0000000000000014 <main>:
 14:   55                   push   %rbp
 15:   48 89 e5             mov    %rsp,%rbp
 [...]
 34:   48 89 c7             mov    %rax,%rdi
 37:   e8 00 00 00 00       callq  3c <main+0x28>
 3c:   89 05 00 00 00 00    mov    %eax,0x0(%rip)
 42:   8b 15 00 00 00 00    mov    0x0(%rip),%edx
 48:   8b 05 00 00 00 00    mov    0x0(%rip),%eax
 4e:   89 d6                mov    %edx,%esi
 50:   89 c7                mov    %eax,%edi
 52:   e8 00 00 00 00       callq  57 <main+0x43>
 57:   89 45 fc             mov    %eax,-0x4(%rbp)
 5a:   8b 45 fc             mov    -0x4(%rbp),%eax
 5d:   89 c6                mov    %eax,%esi
 5f:   48 8d 3d 00 00 00 00          lea    0x0(%rip),%rdi
 66:   b8 00 00 00 00       mov    $0x0,%eax
 6b:   e8 00 00 00 00       callq  70 <main+0x5c>
 70:   b8 00 00 00 00       mov    $0x0,%eax
 75:   c9                   leaveq
 76:   c3                   retq
```

[mymachine] readelf -r prog.o

```
Section de réadressage '.rela.text' à l'adresse de décalage 0x2f8 contient 7 entrées:
 Décalage     Info         Type            Val.-symboles Noms-symb.+ Addenda
000000000038  000e00000004 R_X86_64_PLT32   0000000000000000 atoi - 4
00000000003e  000900000002 R_X86_64_PC32    0000000000000000 i - 4
000000000044  000a00000002 R_X86_64_PC32    0000000000000000 j - 4
00000000004a  000900000002 R_X86_64_PC32    0000000000000000 i - 4
000000000053  000b00000004 R_X86_64_PLT32   0000000000000000 f - 4
000000000062  000500000002 R_X86_64_PC32    0000000000000000 .rodata - 4
00000000006c  000f00000004 R_X86_64_PLT32   0000000000000000 printf - 4
```

So we explain how to fix the problem at loading time:

val = val + @i – 0x42 at 0x3e

val = val + @i – 0x4e at 0x4a

And also

val = val + @j – 0x48 at 0x44

val = val + @f – 0x57 at 0x53

Code relocation (x64)

[mymachine] objdump -d prog

```
0000000000001159 <main>:
  1159:    55                   push   %rbp
  115a:    48 89 e5             mov    %rsp,%rbp
  [...]
  1179:    48 89 c7             mov    %rax,%rdi
  117c:    e8 bf fe ff ff       callq  1040 <atoi@plt>
  1181:    89 05 b9 2e 00 00    mov    %eax,0x2eb9(%rip)
  1187:    8b 15 ab 2e 00 00    mov    0x2eab(%rip),%edx
  118d:    8b 05 ad 2e 00 00    mov    0x2ead(%rip),%eax
  1193:    89 d6                mov    %edx,%esi
  1195:    89 c7                mov    %eax,%edi
  1197:    e8 a9 ff ff ff       callq  1145 <f>
  119c:    89 45 fc             mov    %eax,-0x4(%rbp)
  [...]
  11ba:    c9                   leaveq
  11bb:    c3                   retq
0000000000004038 <j>:
  4038: 1f
 [...]
0000000000004040 <i>:
  4040: 0
```

[mymachine] readelf -r prog

```
Section de réadressage '.rela.dyn' à l'adresse de décalage 0x4b0 contient 8 entrées:
 Décalage     Info       Type       Val.-symboles Noms-symb.+ Addenda
[...]
000000003fe0  000300000006 R_X86_64_GLOB_DAT 0000000000000000 __libc_start_main@GLIBC_2.2.5 + 0
[...]

Section de réadressage '.rela.plt' à l'adresse de décalage 0x570 contient 2 entrées:
 Décalage     Info       Type       Val.-symboles Noms-symb.+ Addenda
000000004018  000200000007 R_X86_64_JUMP_SLO 0000000000000000 printf@GLIBC_2.2.5 + 0
000000004020  000500000007 R_X86_64_JUMP_SLO 0000000000000000 atoi@GLIBC_2.2.5 + 0
```

In the binary (after linking phase)
locations for i, j and f are known...

Code relocation (x64)

[mymachine] objdump -d prog

```
0000000000001159 <main>:
  1159:    55                push   %rbp
  115a:    48 89 e5          mov    %rsp,%rbp
  [...]
  1179:    48 89 c7          mov    %rax,%rdi
  117c:    e8 bf fe ff ff    callq  1040 <atoi@plt>
  1181:    89 05 b9 2e 00 00 mov    %eax,0x2eb9(%rip)
  1187:    8b 15 ab 2e 00 00 mov    0x2eab(%rip),%edx
  118d:    8b 05 ad 2e 00 00 mov    0x2ead(%rip),%eax
  1193:    89 d6             mov    %edx,%esi
  1195:    89 c7             mov    %eax,%edi
  1197:    e8 a9 ff ff ff    callq  1145 <f>
  119c:    89 45 fc          mov    %eax,-0x4(%rbp)
  [...]
  11ba:    c9                leaveq
  11bb:    c3                retq
0000000000004038 <j>:
  4038: 1f
 [...]
0000000000004040 <i>:
  4040: 0
```

[mymachine] readelf -r prog

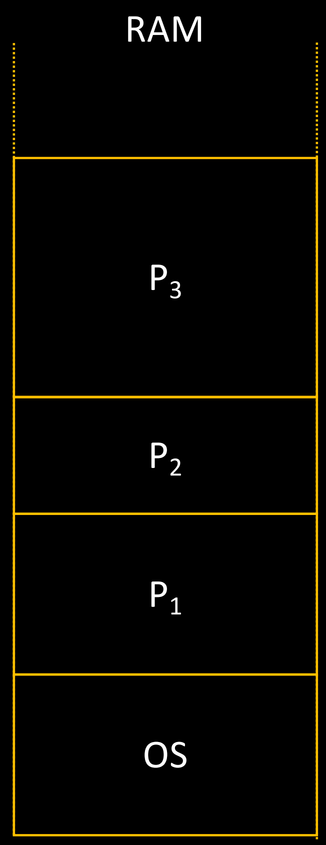
```
Section de réadressage '.rela.dyn' à l'adresse de décalage 0x4b0 contient 8 entrées:
 Décalage     Info      Type      Val.-symboles Noms-symb.+ Addenda
[...]
000000003fe0  000300000006 R_X86_64_GLOB_DAT 0000000000000000 __libc_start_main@GLIBC_2.2.5 + 0
[...]

Section de réadressage '.rela.plt' à l'adresse de décalage 0x570 contient 2 entrées:
 Décalage     Info      Type      Val.-symboles Noms-symb.+ Addenda
000000004018  000200000007 R_X86_64_JUMP_SLO 0000000000000000 printf@GLIBC_2.2.5 + 0
000000004020  000500000007 R_X86_64_JUMP_SLO 0000000000000000 atoi@GLIBC_2.2.5 + 0
```

rip-relative addressing

Resulting address: 0x2eab + %rip (0x118d) = 0x4038

Towards Multiprogramming

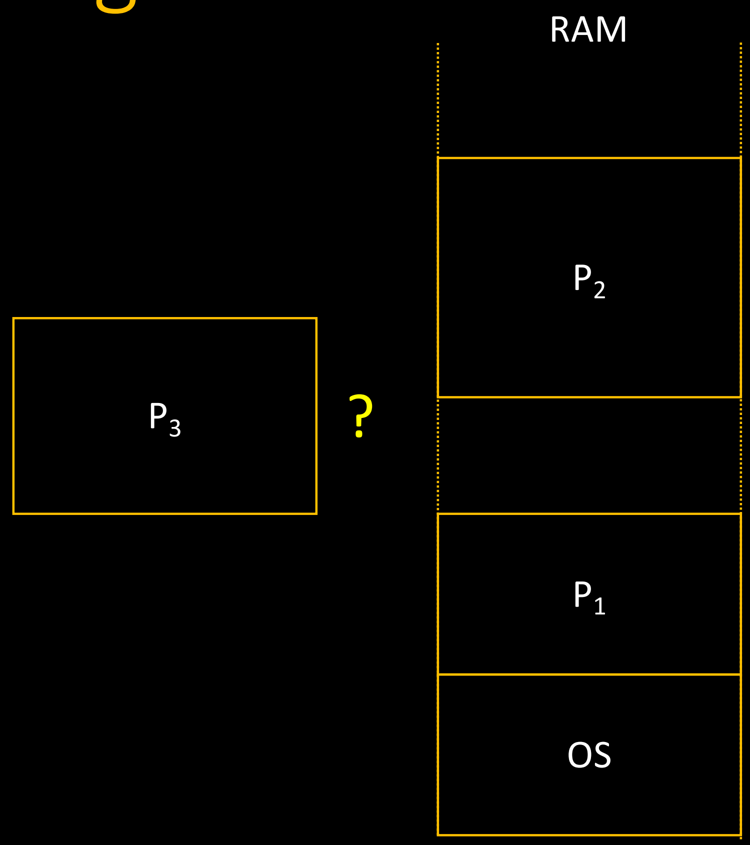
- Code relocation performed by compiler + loader
  - ✅ How to compile processes even if we don't know at which address they will be placed?
- Still to be addressed
  - How to address memory fragmentation?
  - How to let processes grow?
  - How to enforce memory protection?



Towards Multiprogramming

- Fragmentation and process expansion raise similar issues

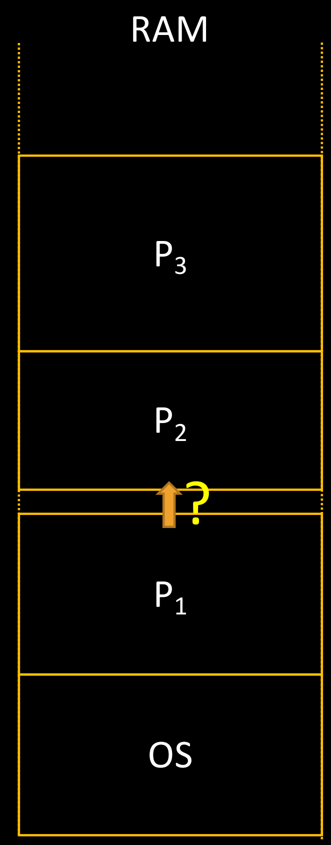
- Memory fragmentation
  - At some point, the OS has to collect and fuse free spaces by moving processes
    - Expensive memcpy + relocate phase
      - Relocation data must be kept!



Towards Multiprogramming

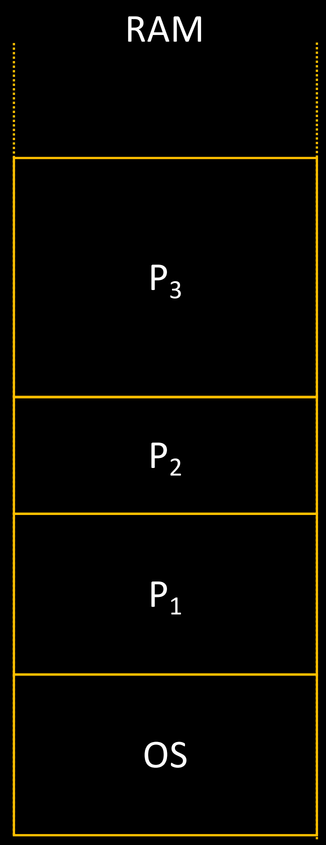
- Fragmentation and process expansion raise similar issues

- Process expansion
  - To make room for an unexpectedly large amount of malloc operations (for instance), the OS must
    - Either move away multiple processes
    - Or relocate current process elsewhere



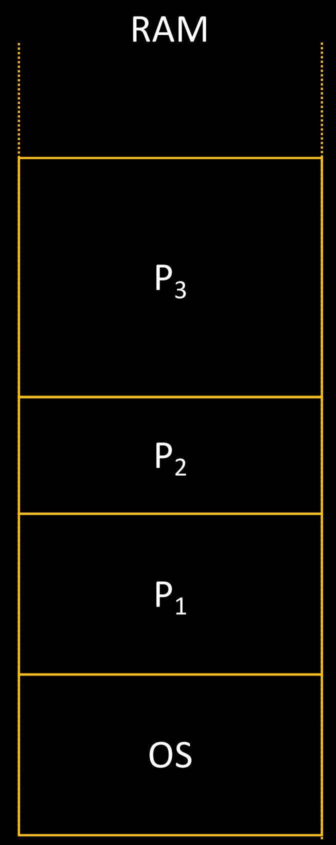
Towards Multiprogramming

- How to enforce memory protection?
  - Ask the compiler to perform checks at compile time?



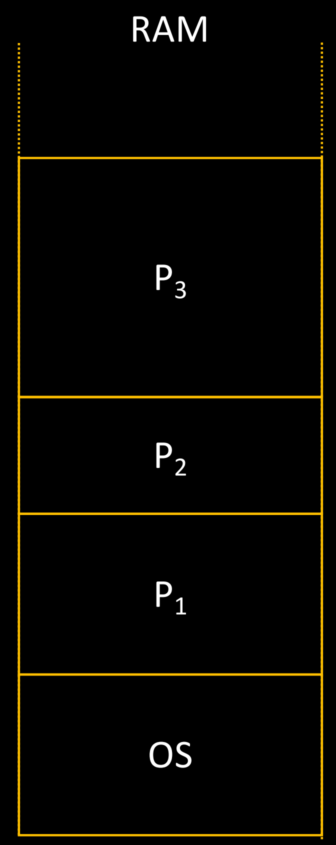
Towards Multiprogramming

- How to enforce memory protection?
  - Ask the compiler to perform checks at compile time?
    - Illusory
      - Think about indirect memory accesses
  - Ask the compiler to generate checks each time an address is about to be used?



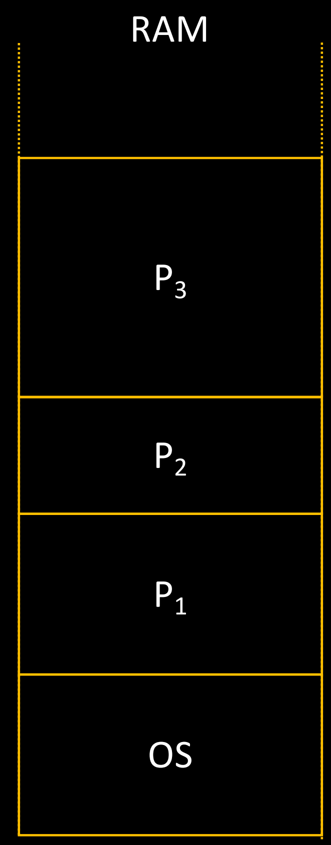
Towards Multiprogramming

- How to enforce memory protection?
  - Ask the compiler to perform checks at compile time?
    - Illusory
      - Think about indirect memory accesses
  - Ask the compiler to generate checks each time an address is about to be used?
    - Expensive… and illusory



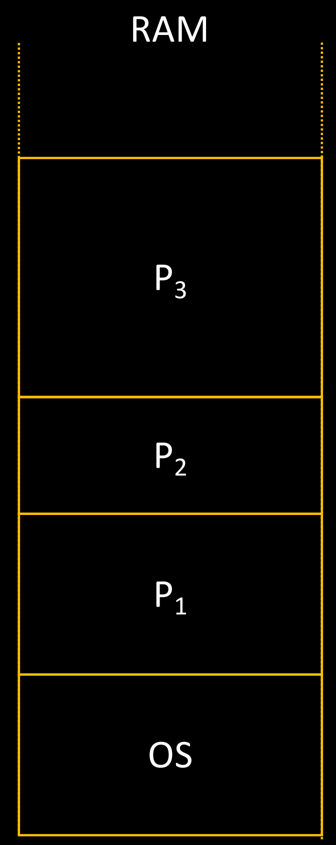
Towards Multiprogramming

- How to enforce memory protection?
  - Ask Computer Architects to add new functionalities to processors!
    - Memory access control
    - Efficient (free?) Relocation



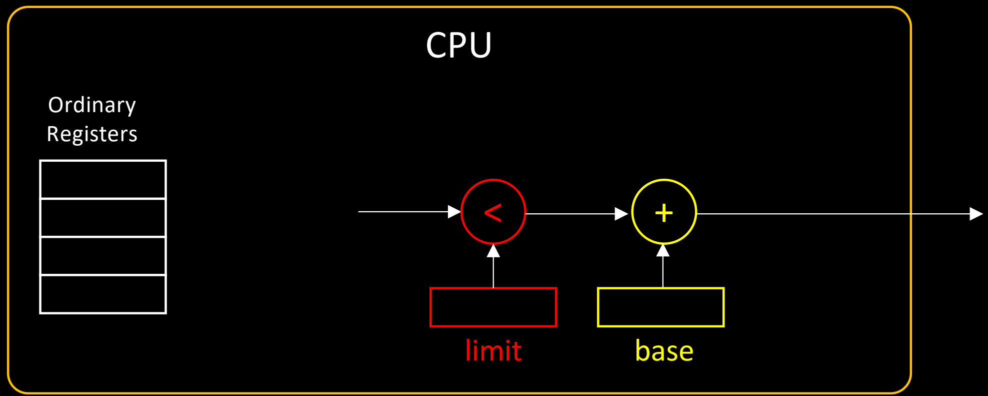
Towards Multiprogramming

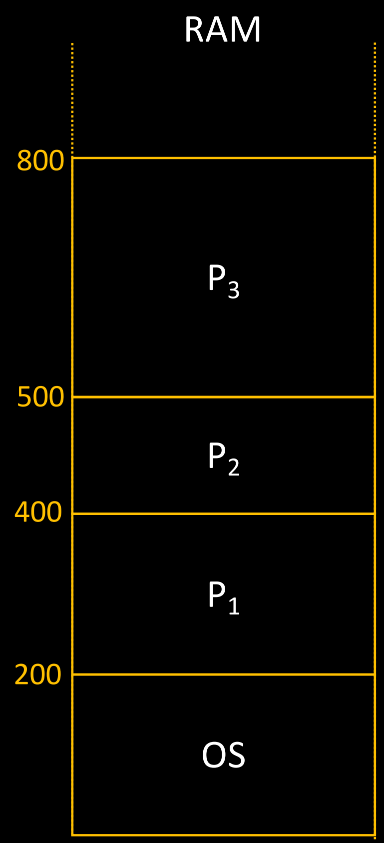
- How to enforce memory protection?
  - Ask Computer Architects to add new functionalities to processors!
    - Memory access control
    - Efficient (free?) Relocation
  - Computer Architects answered:
    "Ok guys, we'll add two registers…"



Towards Multiprogramming

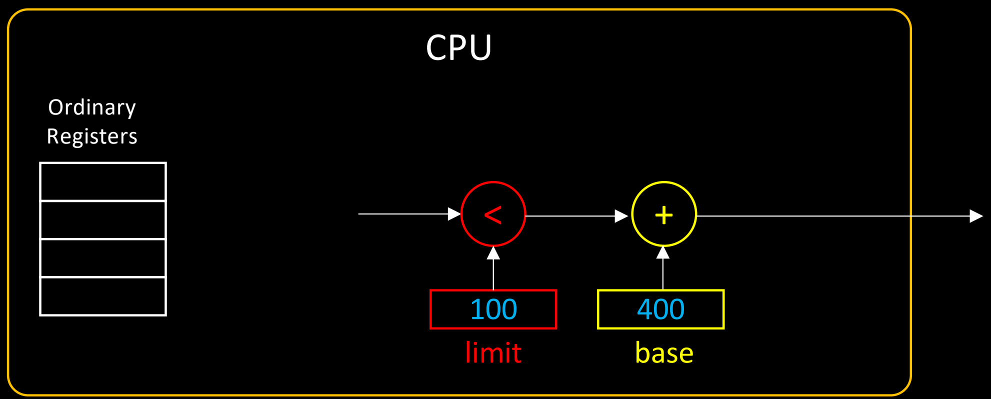
- Two special registers:
  - *Limit*: size of current process
  - *Base*: starting address

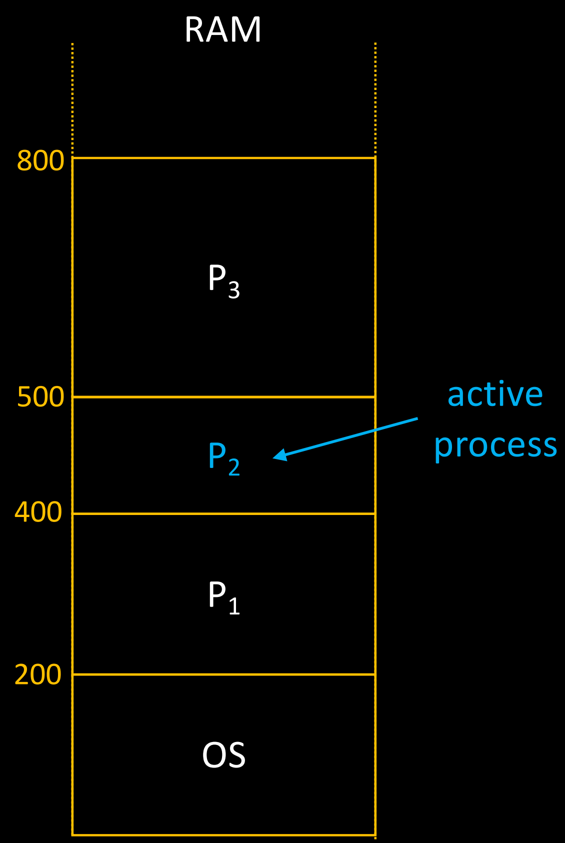




Towards Multiprogramming

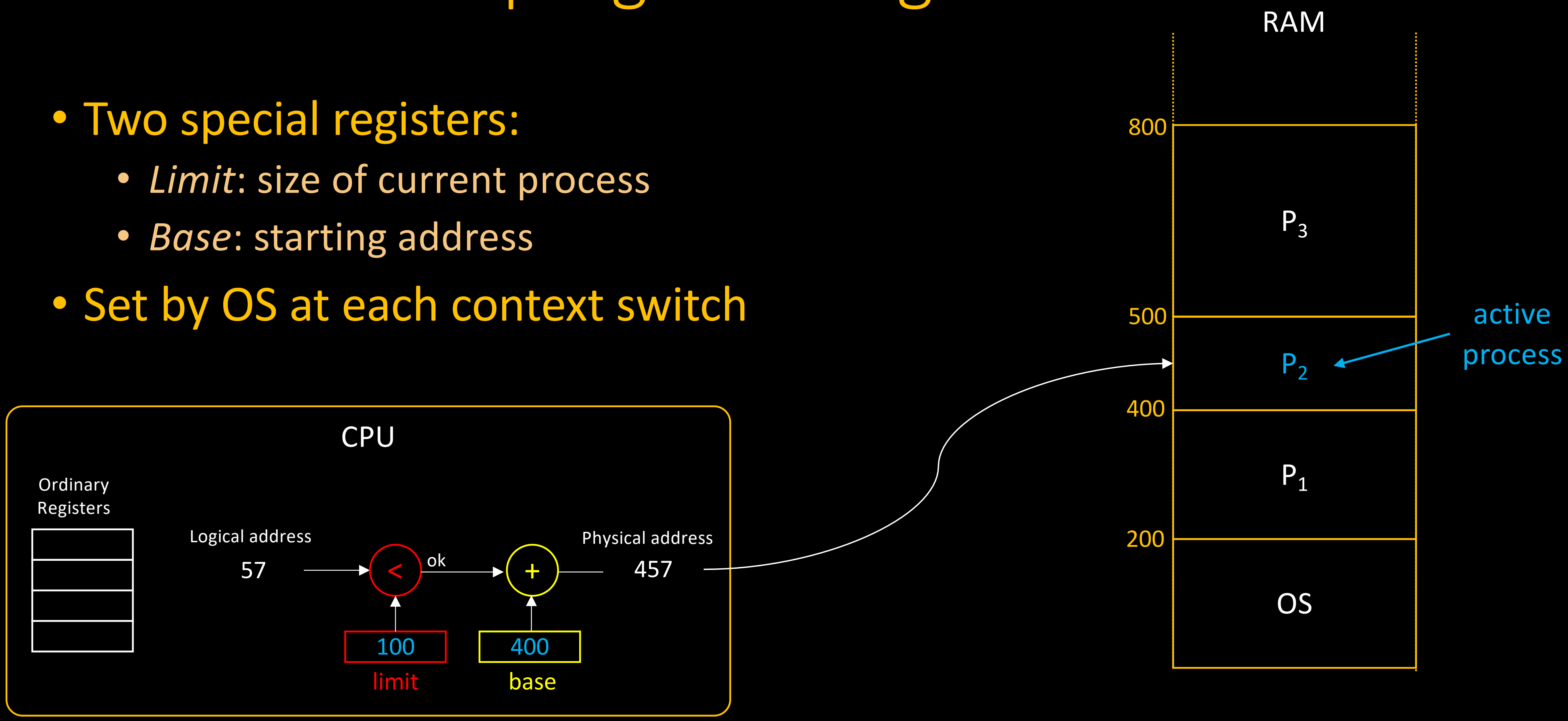
- Two special registers:
  - *Limit*: size of current process
  - *Base*: starting address
- Set by OS at each context switch





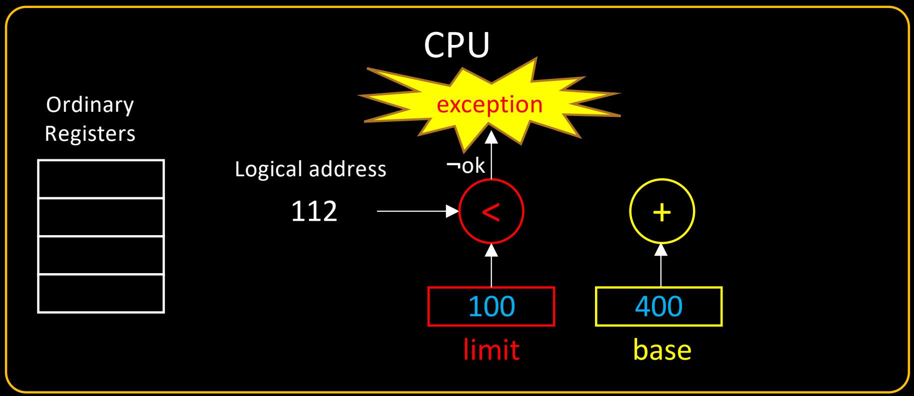
Towards Multiprogramming

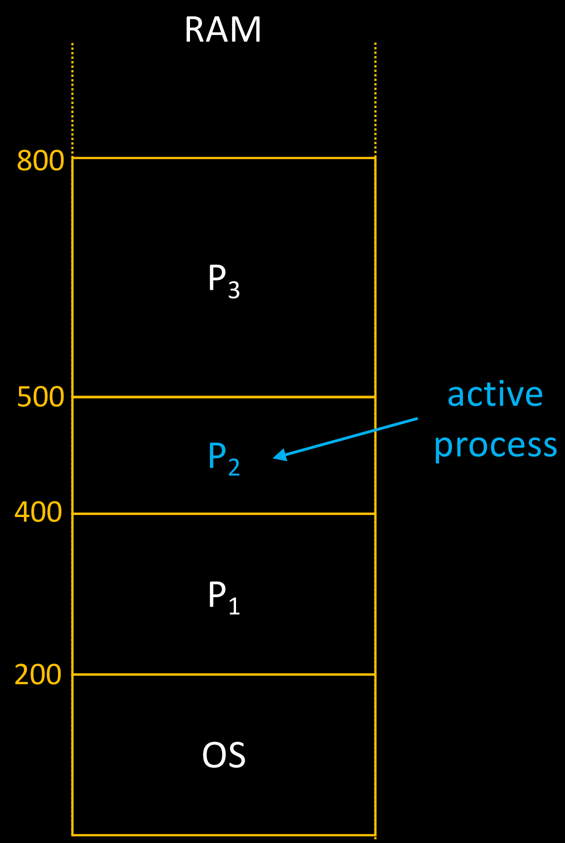
- Two special registers:
  - *Limit*: size of current process
  - *Base*: starting address
- Set by OS at each context switch



Towards Multiprogramming

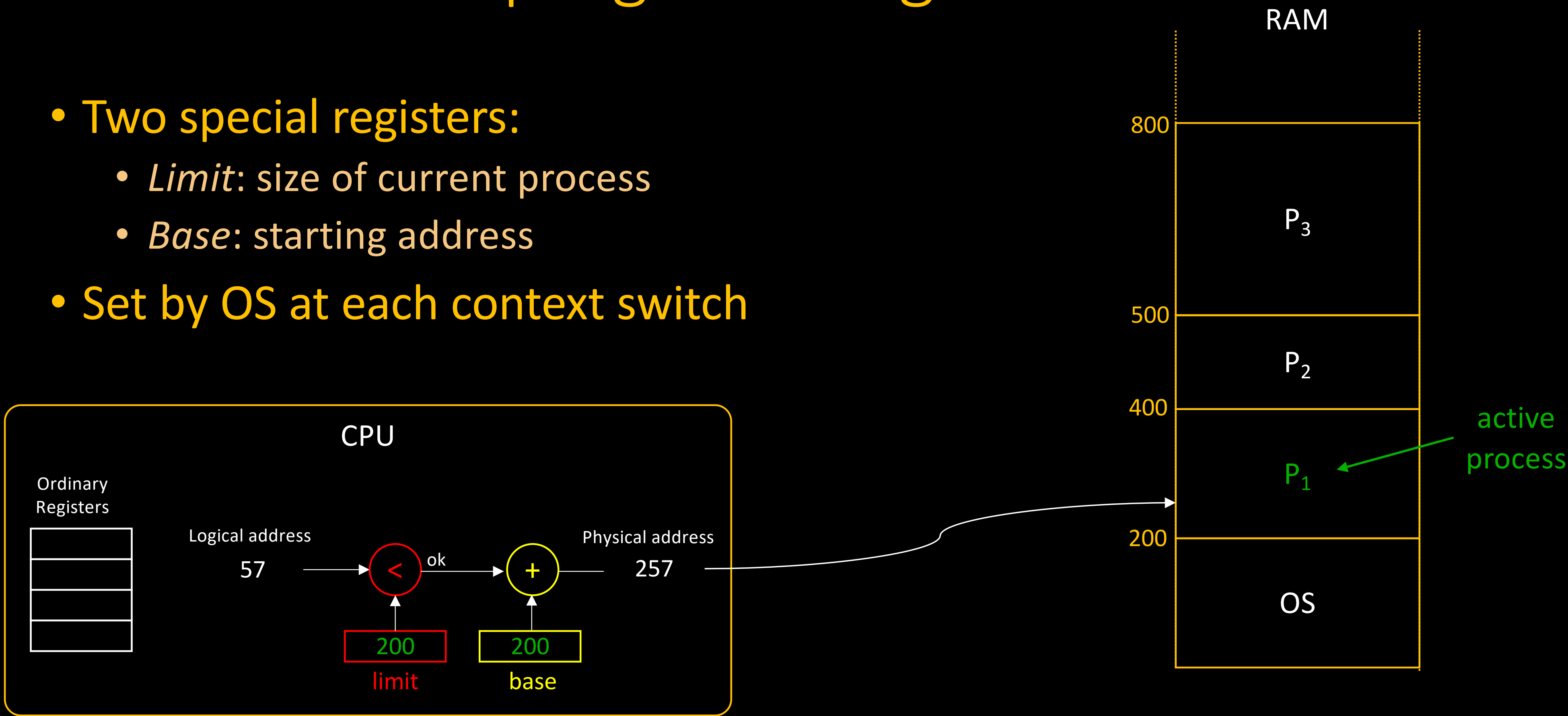
- Two special registers:
  - *Limit*: size of current process
  - *Base*: starting address
- Set by OS at each context switch





Towards Multiprogramming

- Two special registers:
  - *Limit*: size of current process
  - *Base*: starting address
- Set by OS at each context switch



Base + Limit registers

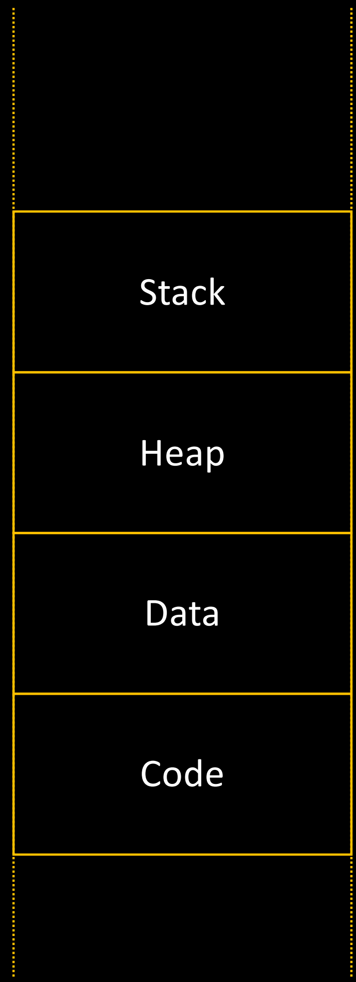
- Processes are now isolated from each other
  - Logical to physical conversion incurs almost no overhead
  - Moving a process to a new location = cost of memmove
  - Protection is guaranteed by hardware
    - No access allowed outside address space

Base + Limit registers

- Processes are now isolated from each other
  - Logical to physical conversion incurs almost no overhead
  - Moving a process to a new location = cost of memmove
  - Protection is guaranteed by hardware
    - No access allowed outside address space
- Well, maybe they’re too isolated
  - No direct data sharing between processes is possible
- Memory fragmentation is still *pain in the a*^H^H^H, hum, very annoying

Splitting address spaces

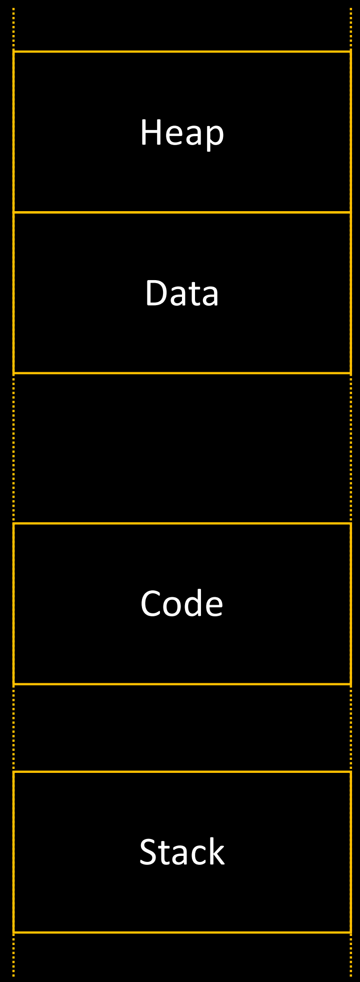
- Address spaces are composed of different regions
  - code, data, heap, stack



Splitting address spaces

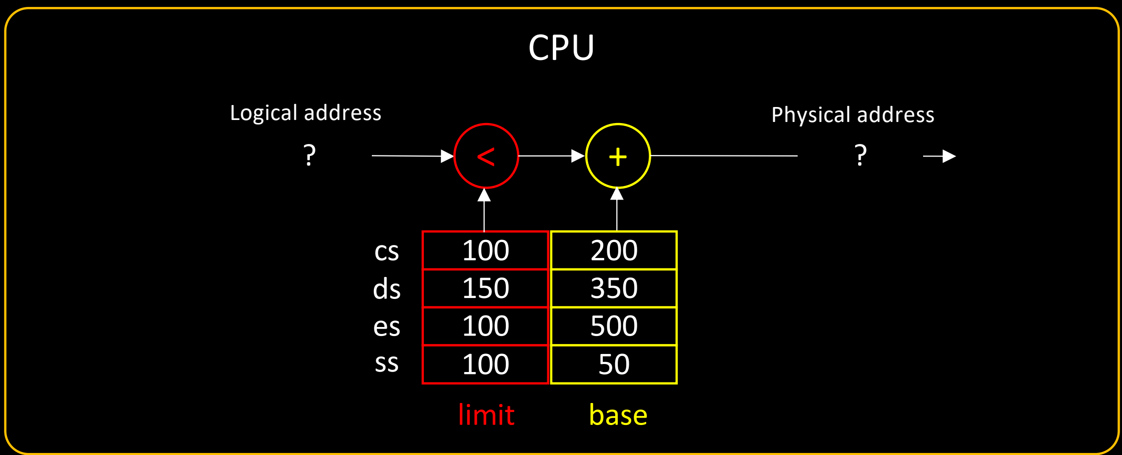
- Address spaces are composed of different regions
    - code, data, heap, stack

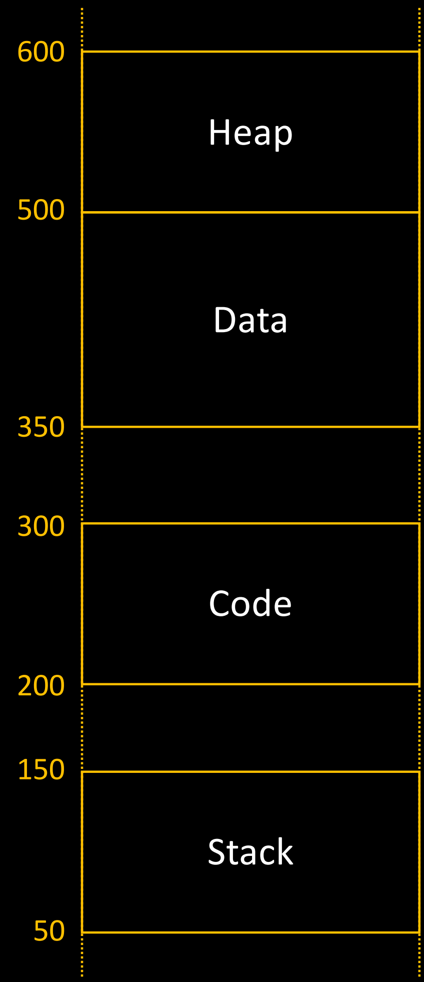
- There’s no reason why they should stick together
    - Having one separate (base,limit) per region would allow independent allocations



Segmentation

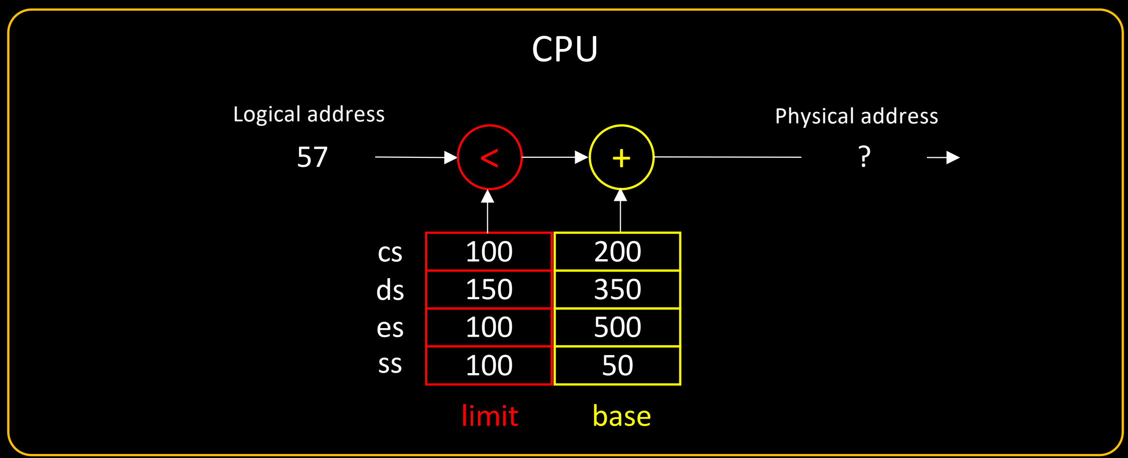
- Having one separate (base, limit) per *memory segment*
  - Array of (limit,base)

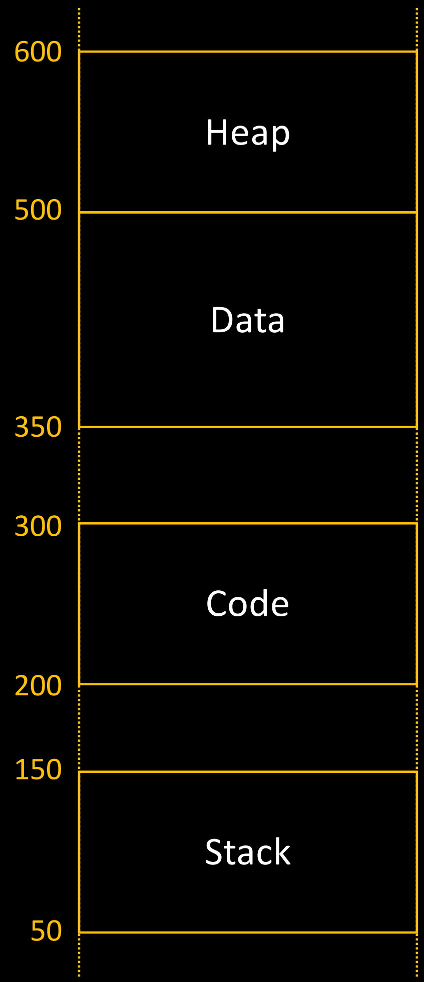




Segmentation

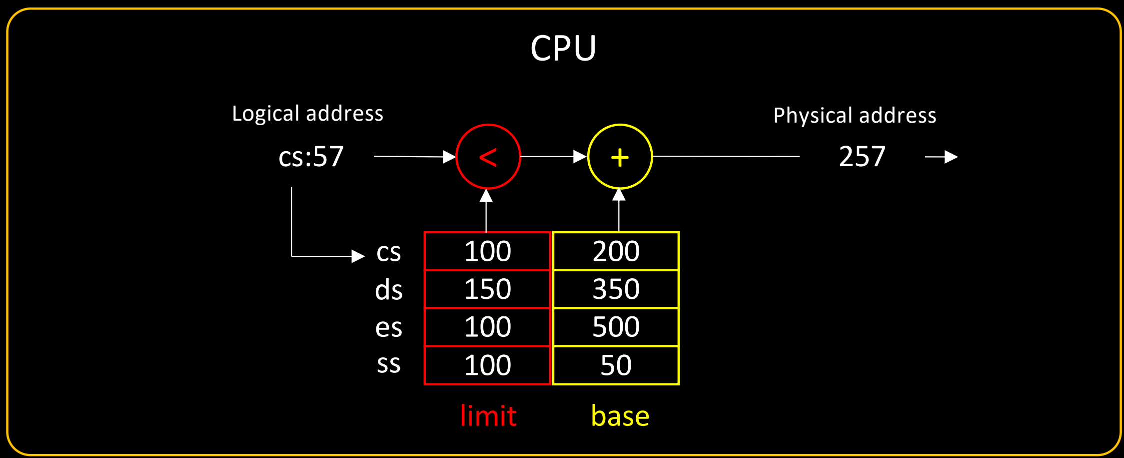
- How to determine which segment to use?
  - code, data, extra or stack?

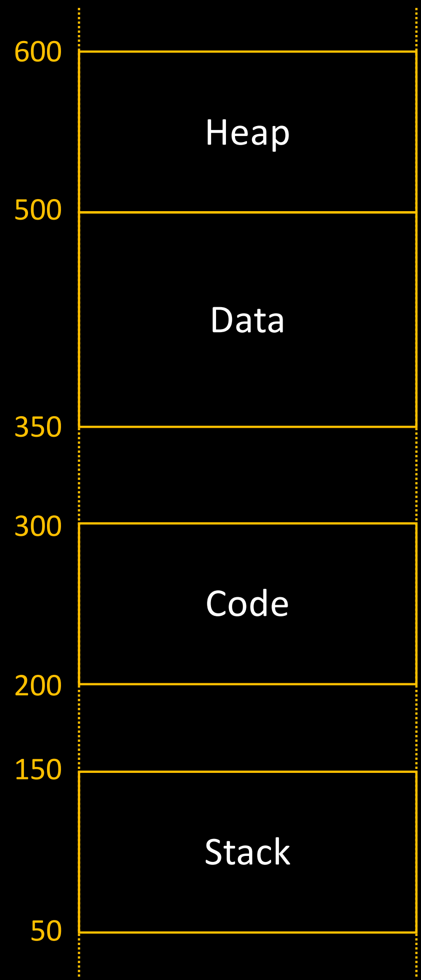




Segmentation

- Addresses = segment:offset
  - mov ds:[ax], bx
  - jmp cs:57
- Default segment is instruction-specific





Segmentation

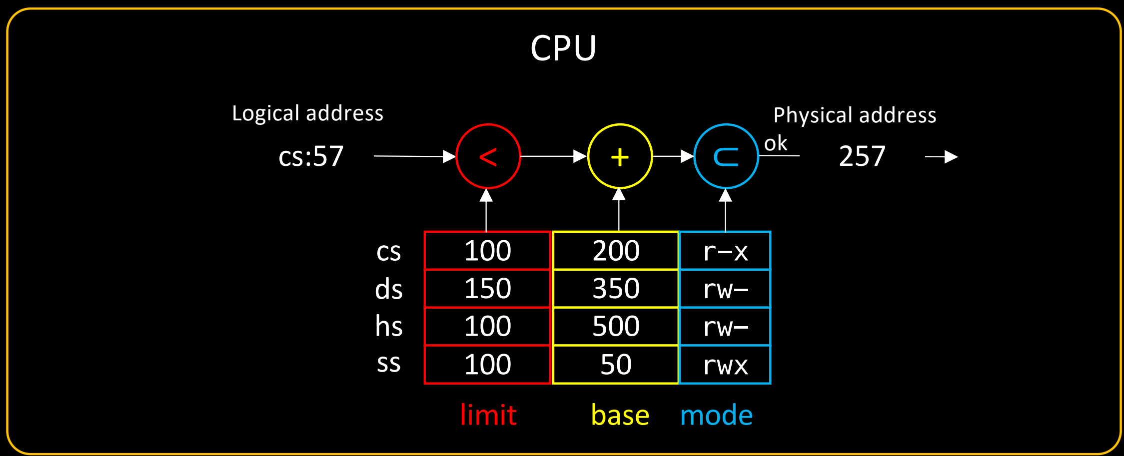
- Splitting address spaces in smaller chunks provides more allocation flexibility

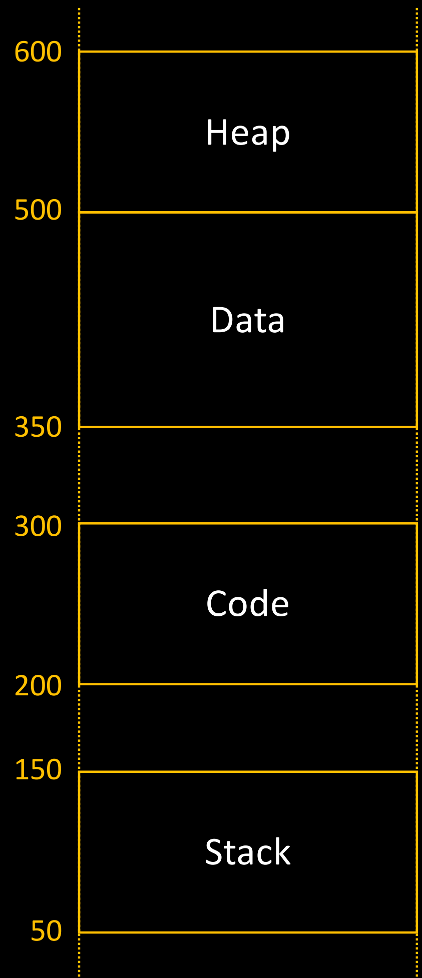
- Shared memory between processes is possible
  - Use the same (base, limit) for multiple processes

  - Sharing the code segments could save memory, for instance!
    - How about security?

Segmentation

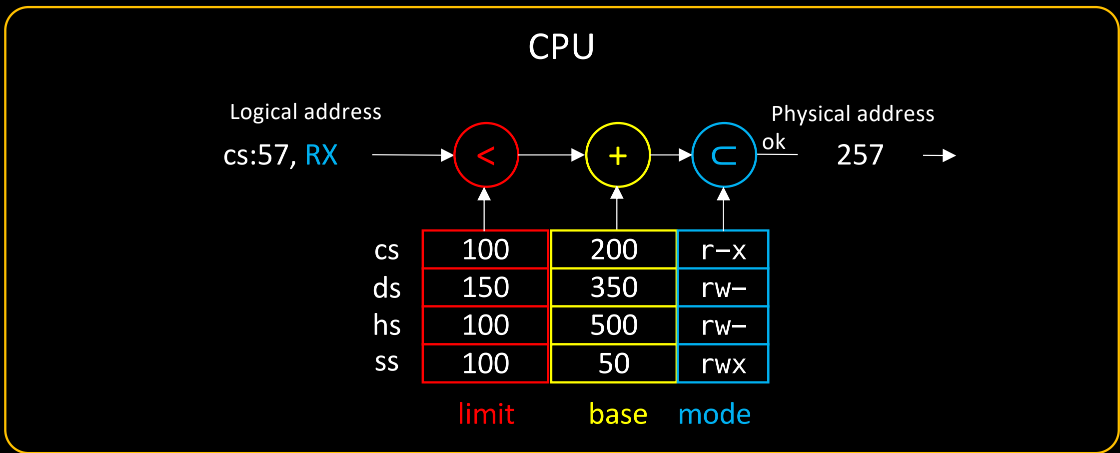
- Access rights can be specified in segment descriptors
  - *Read, Write, Execute*

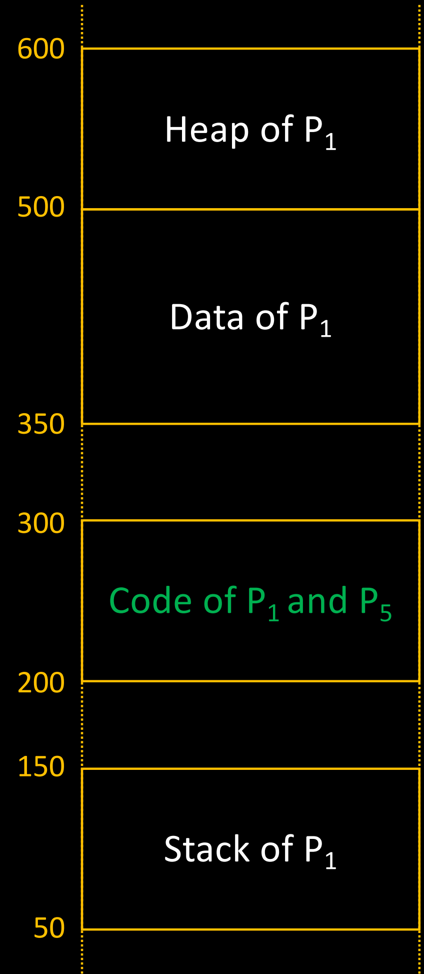




Segmentation

- Access rights can be specified in segment descriptors
  - *Read, Write, Execute*
- Access mode is provided by CPU





Quiz time

Segmentation

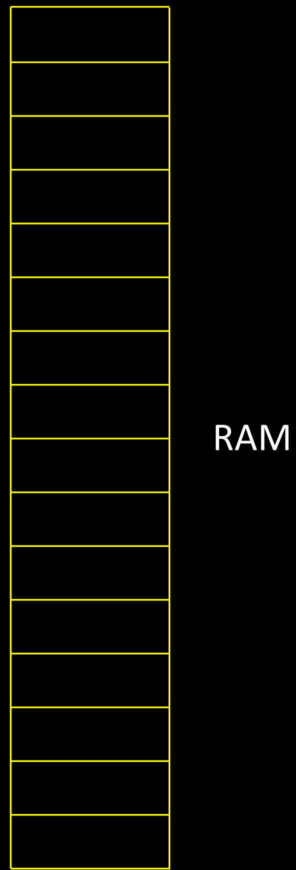
- Splitting address spaces in smaller chunks provides more allocation flexibility
- Shared memory between processes is possible
  - Use the same (base, limit) for multiple processes
- Memory accesses are controlled on a per-segment basis

Segmentation

- Splitting address spaces in smaller chunks provides more allocation flexibility
- Shared memory between processes is possible
  - Use the same (base, limit) for multiple processes
- Memory accesses are controlled on a per-segment basis
- But fragmentation is still a problem for the OS

Towards no fragmentation on the OS side

- To get rid of small chunks of free memory…
  …let’s enforce a single chunk size!
  - Called Page (aka Frame)

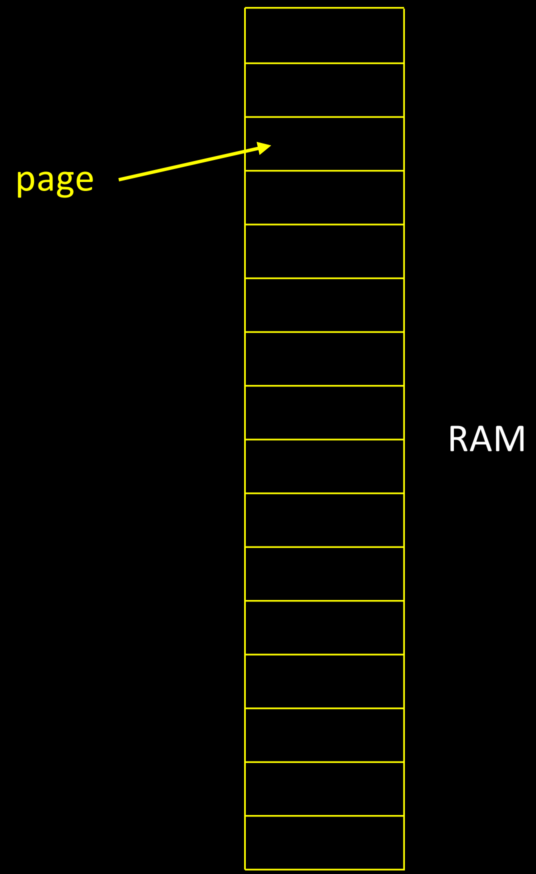


Towards no fragmentation on the OS side

- To get rid of small chunks of free memory…
  …let's enforce a single chunk size!
  - Called Page (aka Frame)

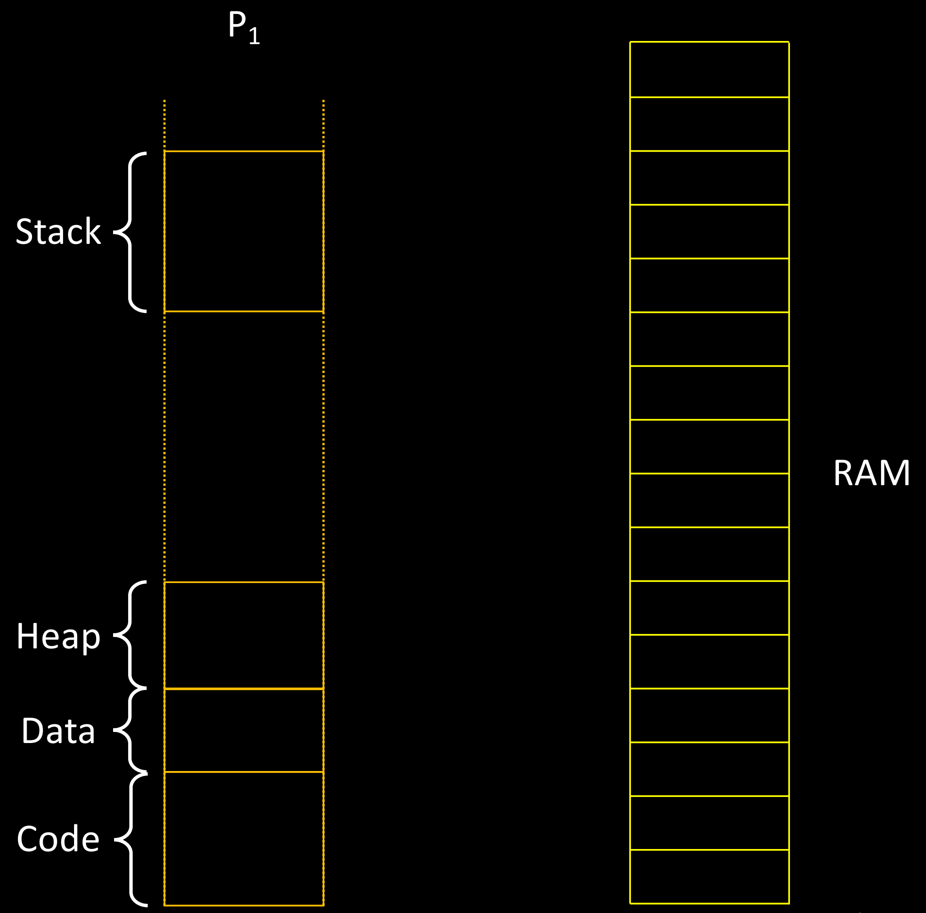
- Physical memory is virtually divided in pages of the same size
  - Typically 4KB on x86 architectures

- A page is either
  - Allocated (e.g. to a process)
  - Free



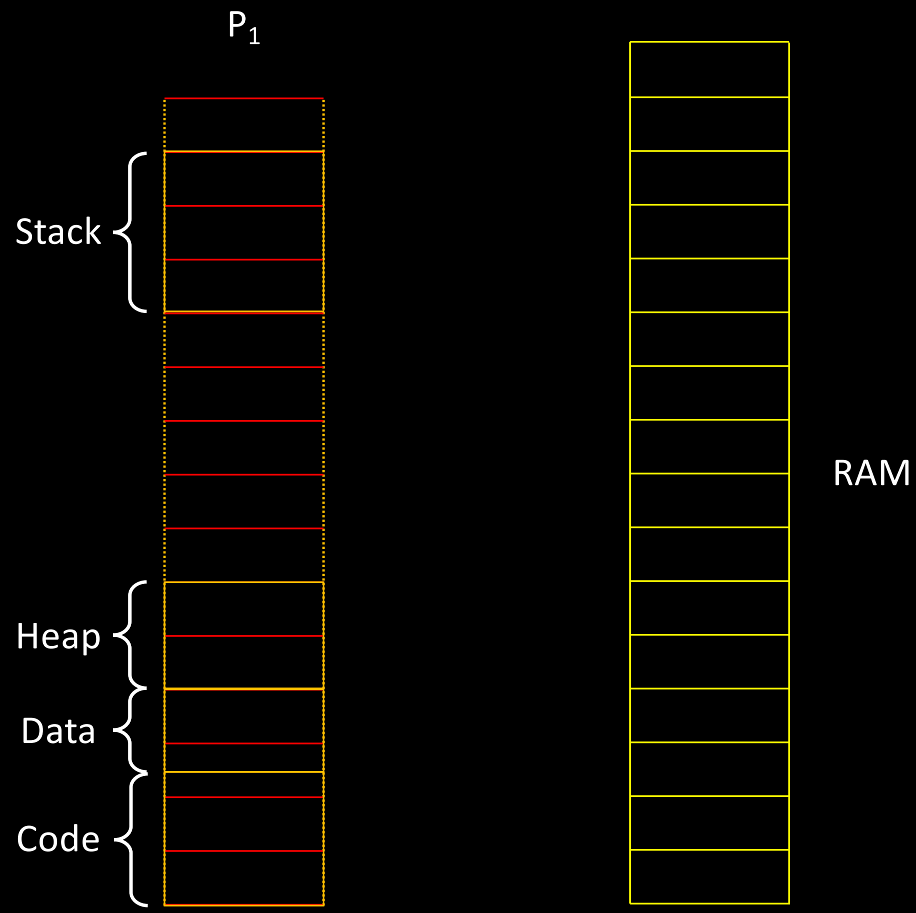
Memory Paging

- Processes' address spaces are also (virtually) divided in pages



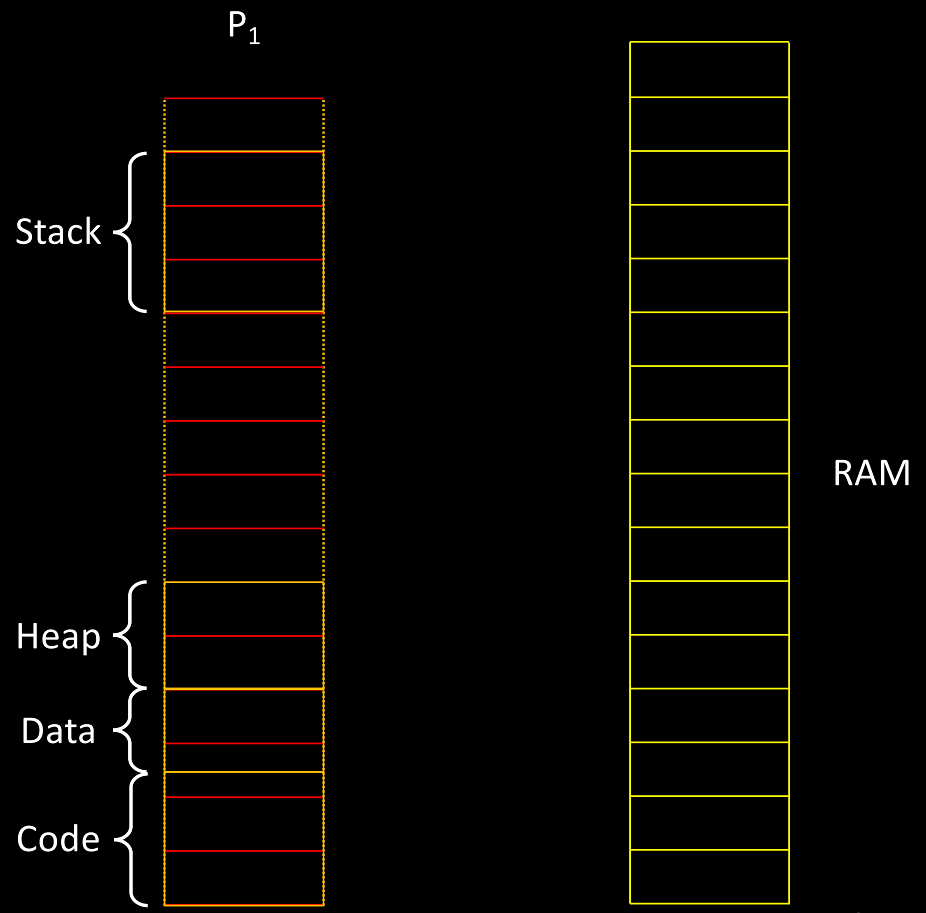
Memory Paging

- Processes' address spaces are also (virtually) divided in pages
  - Page is the unique allocation unit



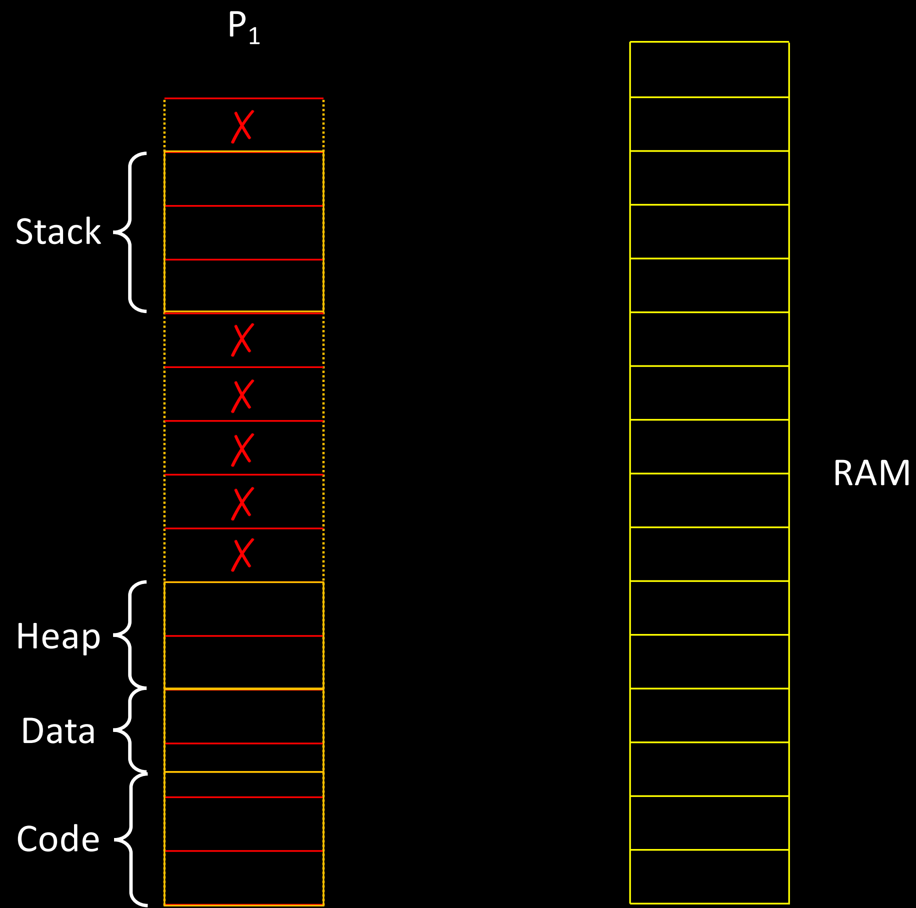
Memory Paging

- Processes' address spaces are also (virtually) divided in pages
  - Space reclaimed by processes must be rounded to a multiple of Page Size
  - And aligned on a multiple of Page Size as well



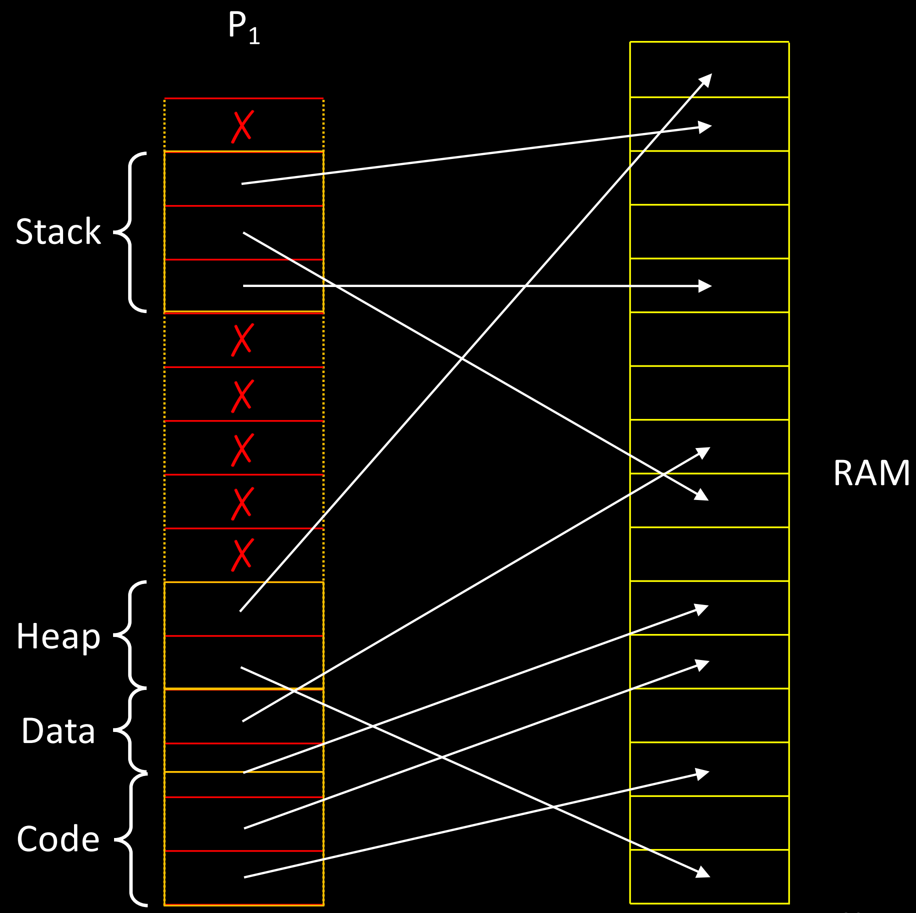
Memory Paging

- Processes' address spaces are also (virtually) divided in pages
  - Not all pages are allocated



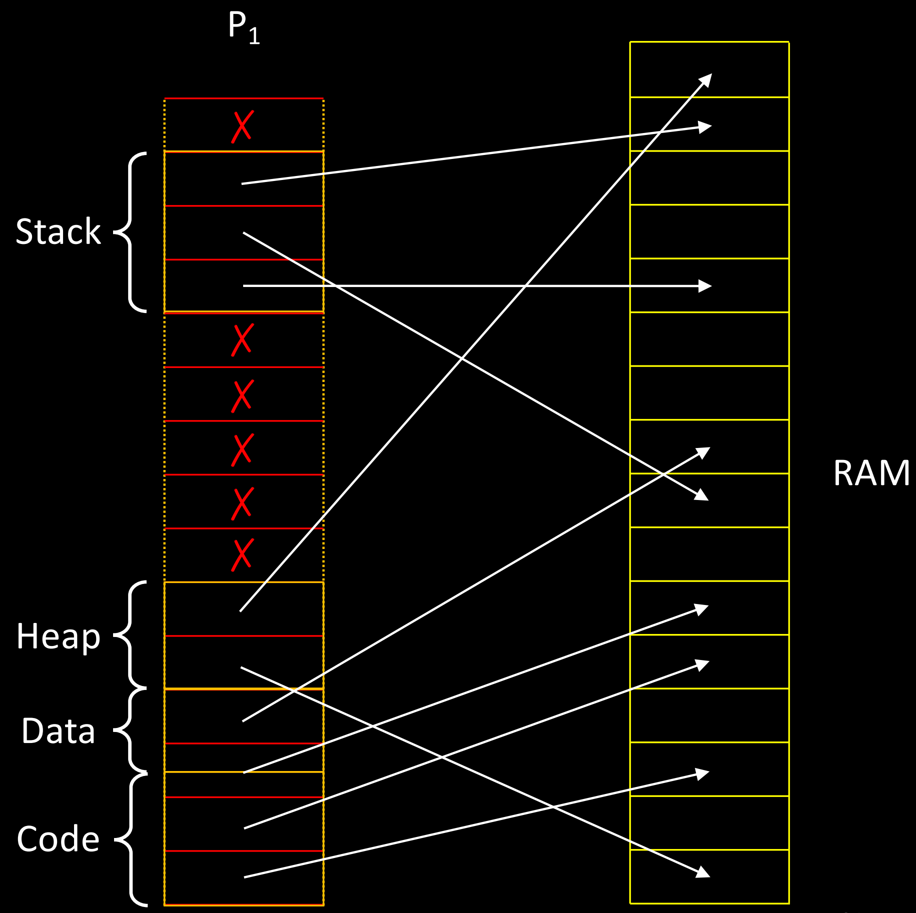
Memory Paging

- Processes' address spaces are also (virtually) divided in pages
  - Pages are dynamically allocated on-the-fly
    - No guarantee that contiguous *virtual pages* are allocated contiguously in physical memory



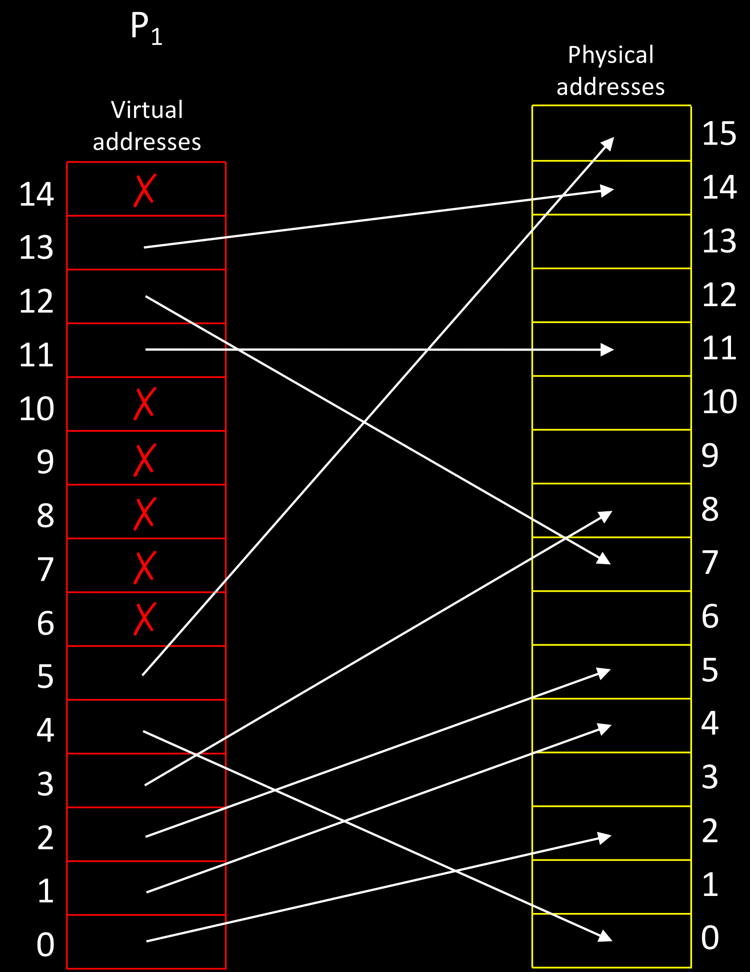
Memory Paging

- Processes' address spaces are also (virtually) divided in pages
  - Pages are dynamically allocated on-the-fly
    - No guarantee that contiguous *virtual pages* are allocated contiguously in physical memory
    - Very efficient allocator on the OS side!
      - get_free_page() could be implemented using O(1) algorithms



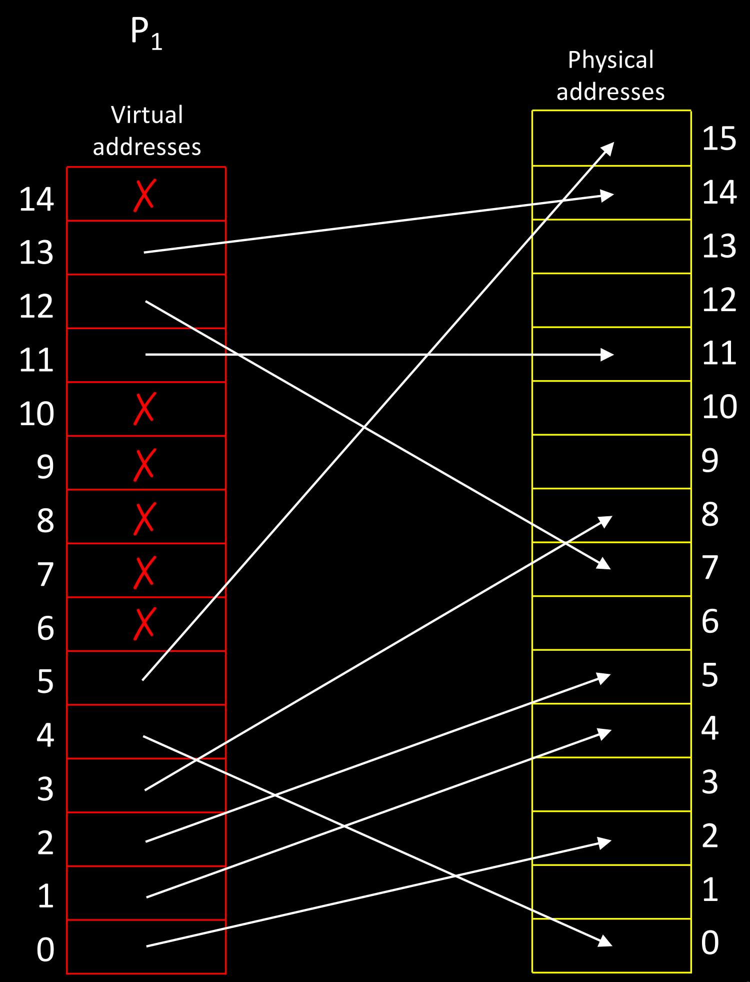
Memory Paging

- Virtual to Physical address translation
  - When the CPU executes user-level code, it sees virtual addresses
  - As in Segmented Systems, we must translate such addresses into physical addresses in RAM
  - Problem: the mapping is irregular!



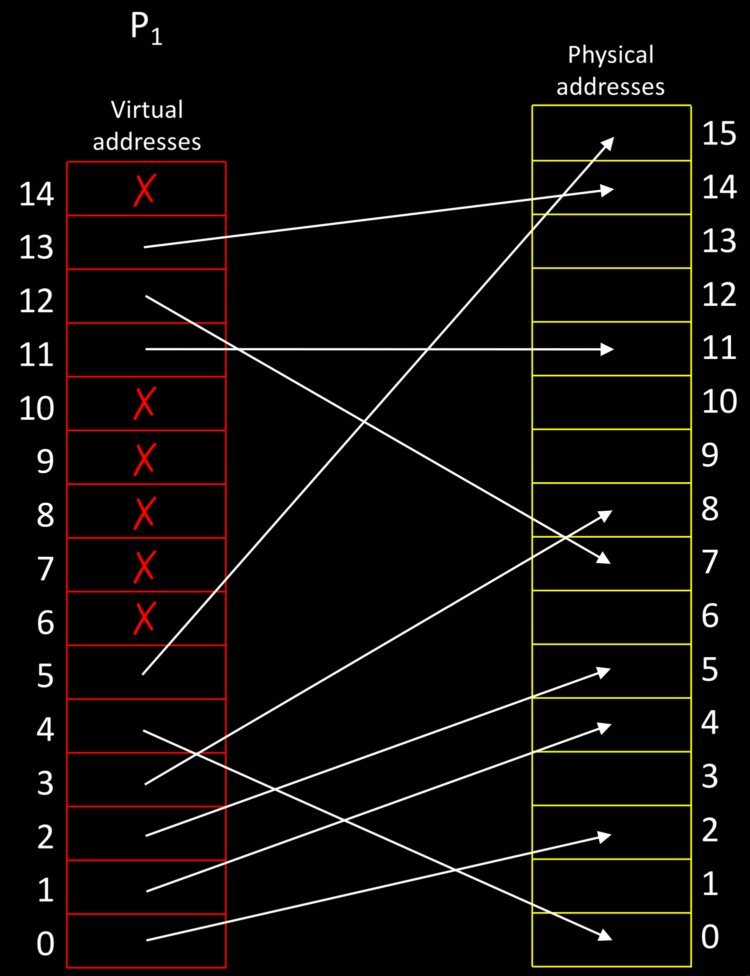
Memory Paging

- Virtual to Physical address translation
  - Assuming page size is 4KB (2^{12})
  - Say a variable 'i' in the data segment has the following virtual address:
    - &i = 12436
    - Where is 'i' located?



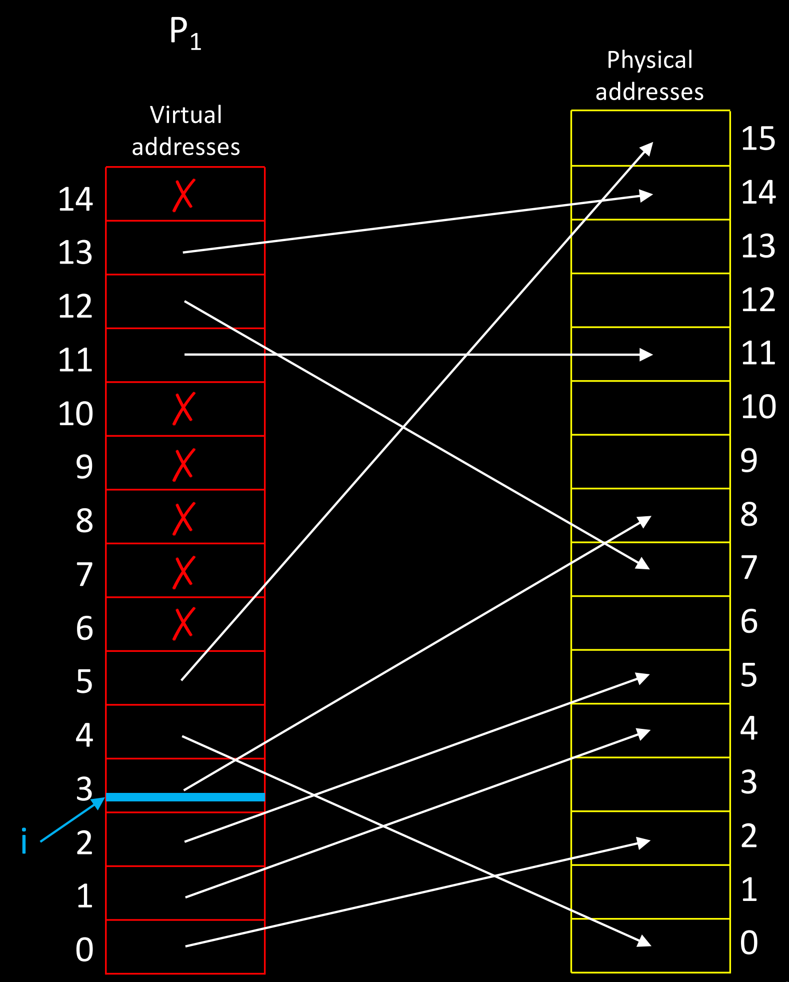
Memory Paging

- Virtual to Physical address translation
  - Assuming page size is 4KB (2^{12})
  - Say a variable 'i' in the data segment has the following virtual address:
    - &i = 12436
    - 12436 = 3 * 4096 + 148



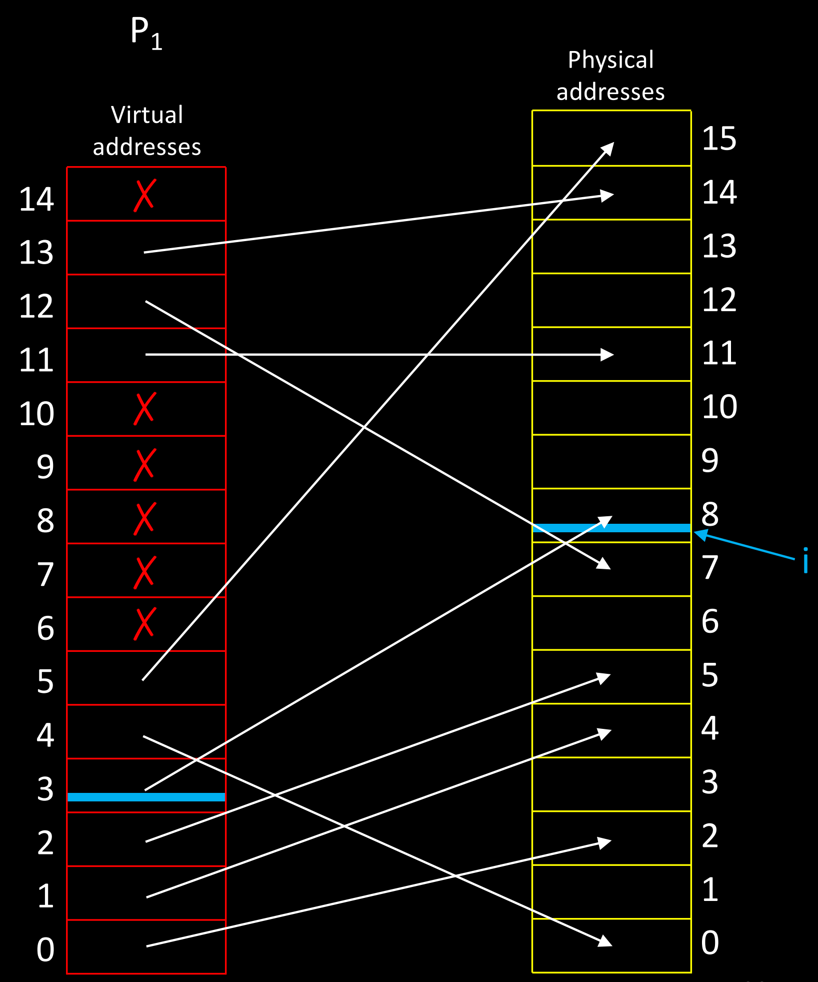
Memory Paging

- Virtual to Physical address translation
  - Assuming page size is 4KB (2^{12})
  - Say a variable 'i' in the data segment has the following virtual address:
    - &i = 12436
    - 12436 = 3 * 4096 + 148
    - 'i' is located inside virtual page 3, at offset 148



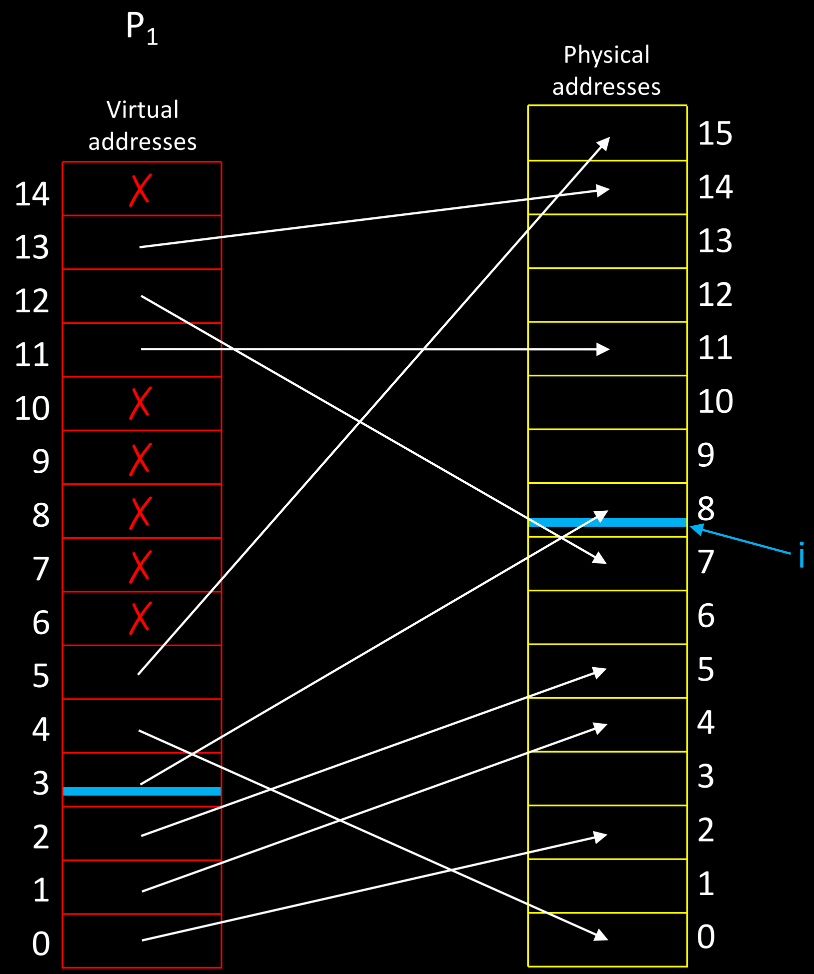
Memory Paging

- Virtual to Physical address translation
  - Assuming page size is 4KB (2^{12})
  - Say a variable 'i' in the data segment has the following virtual address:
    - &i = 12436
    - 12436 = 3 * 4096 + 148
    - 'i' is located inside virtual page 3, at offset 148
    - Its physical address is 8 * 4096 + 148



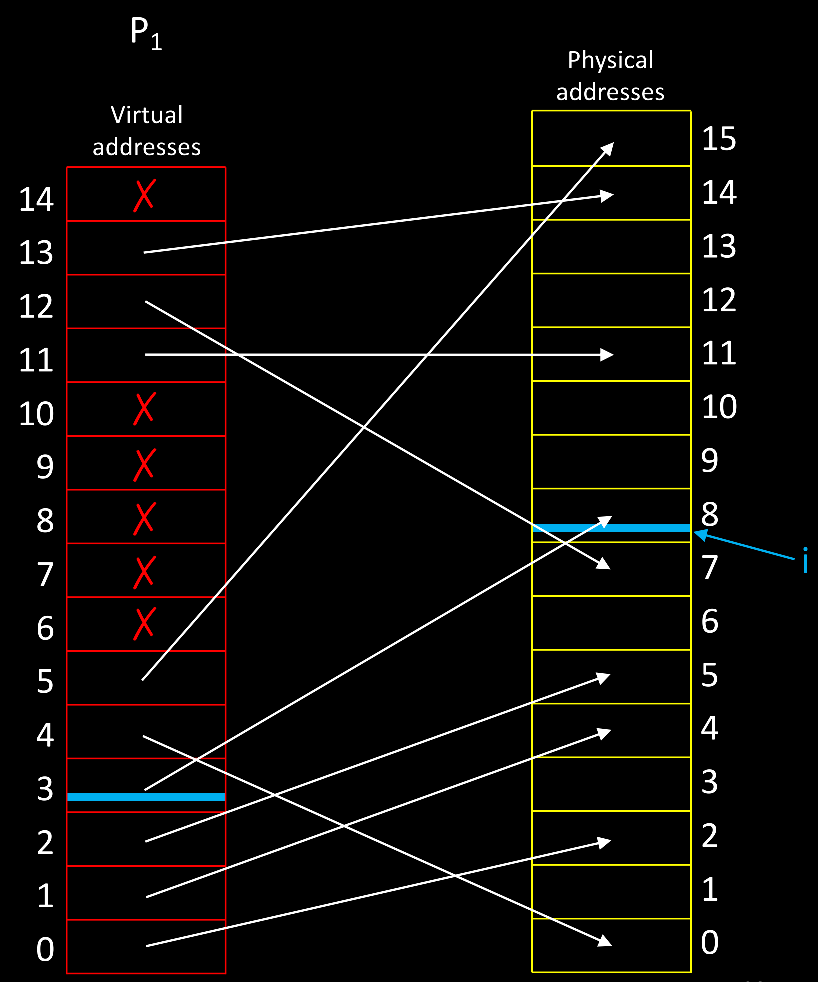
Memory Paging

- Virtual to Physical address translation
  - Binary representations of 32-bit addresses
    - v@i = 3 * 4096 + 148
    - v@i = $\overbrace{000000\ldots00011}^{\text{20 bits}}\ \overbrace{000010010100}^{\text{12 bits}}$
    - p@i =



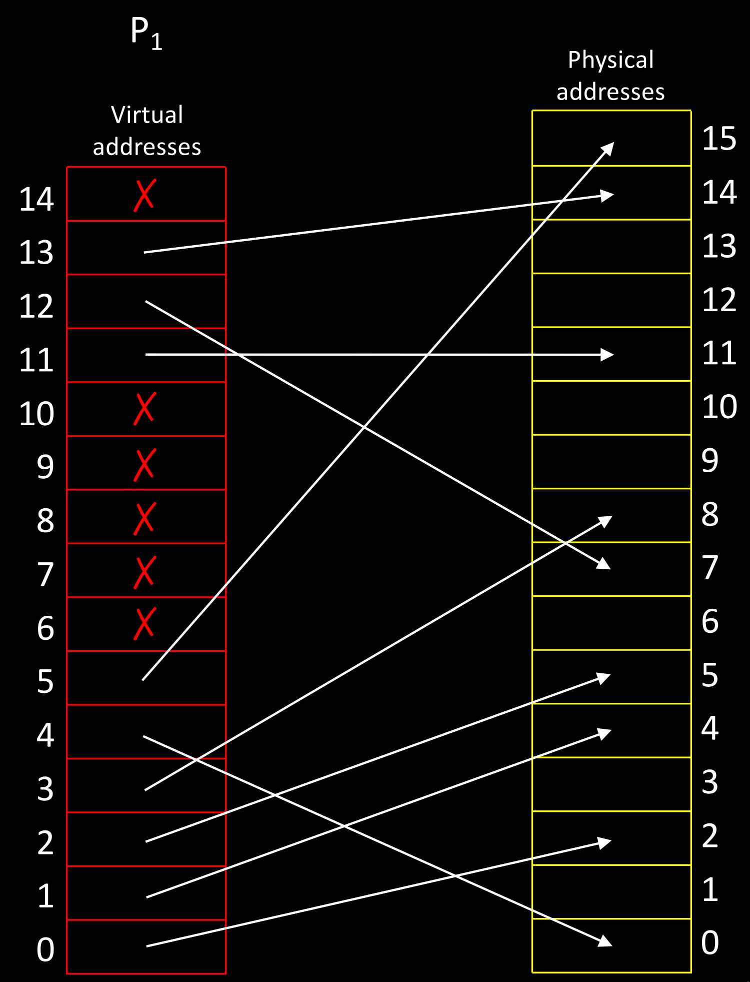
Memory Paging

- Virtual to Physical address translation
  - Binary representations of 32-bit addresses
    - v@i = 3 * 4096 + 148
    - v@i = $\overbrace{\text{000000...00011}}^{\text{20 bits}}\ \overbrace{\text{000010010100}}^{\text{12 bits}}$
    - p@i = 000000...01000 000010010100



Memory Paging

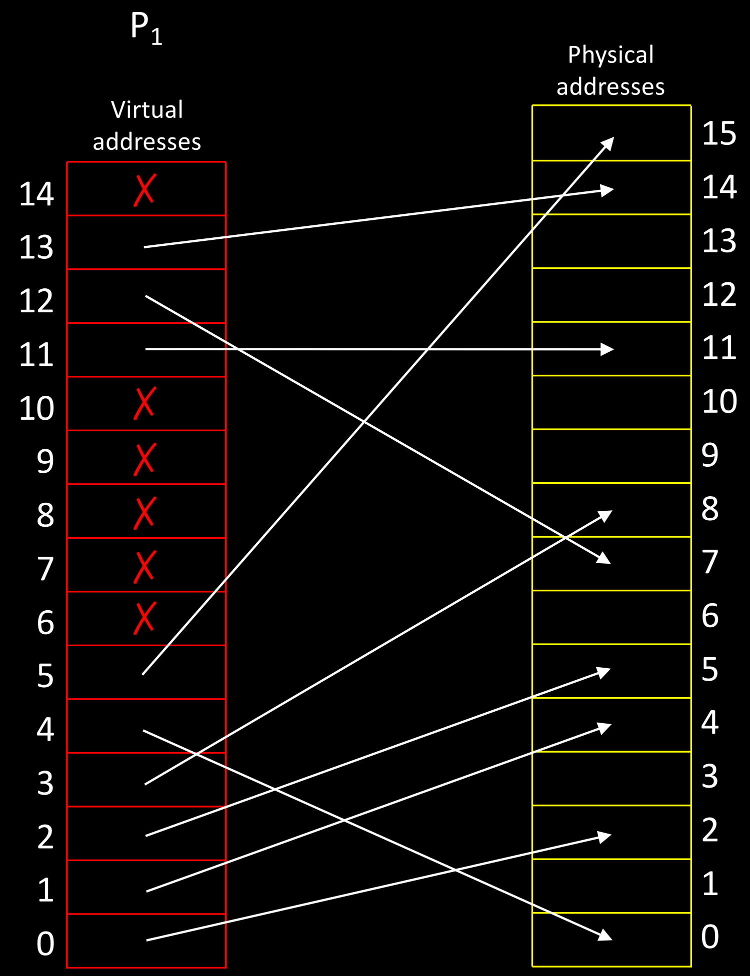
- Virtual to Physical address translation
  - We "just" need to convert virtual pages (VP) to physical pages (PP)
    - For each process
  - Use a table?



Memory Paging

Page table of P_1

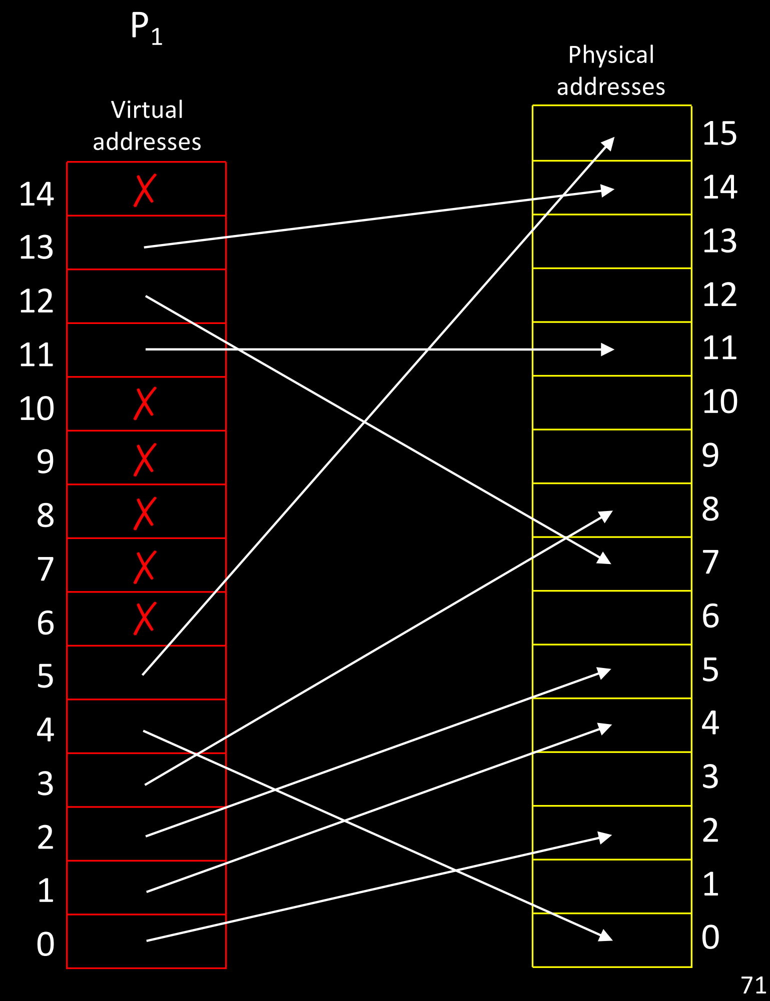
	Phys. Page	Valid
0		
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		
13		
14		



Memory Paging

Page table of P_1

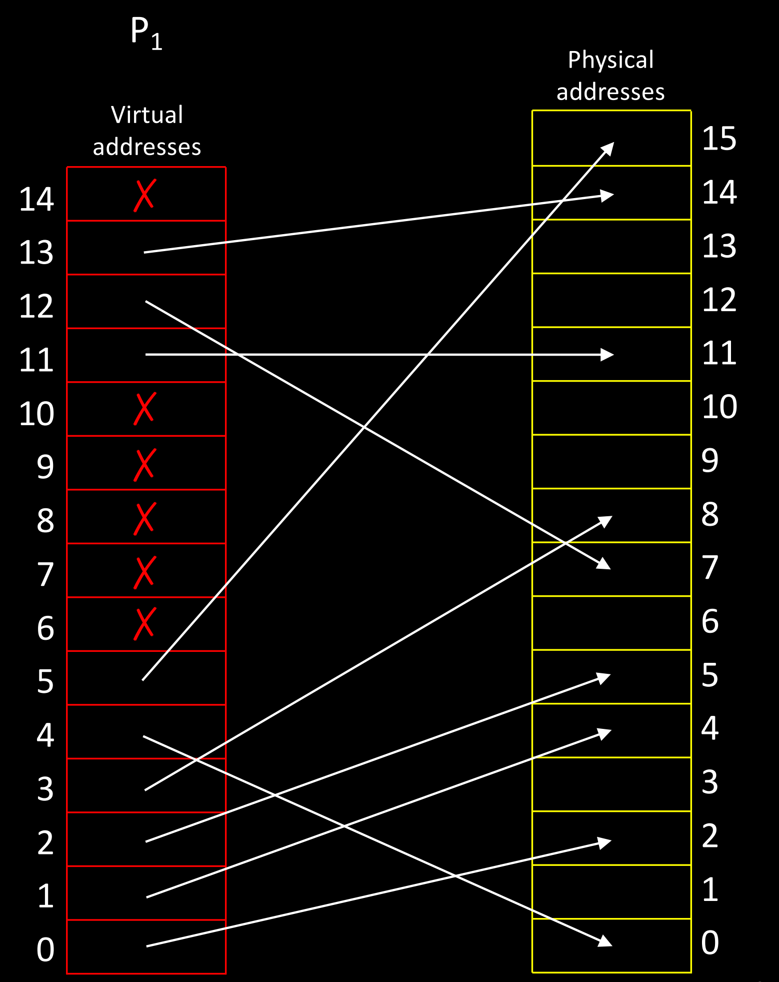
	Phys. Page	Valid
0	2	1
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		
13		
14		



Memory Paging

Page table of P_1

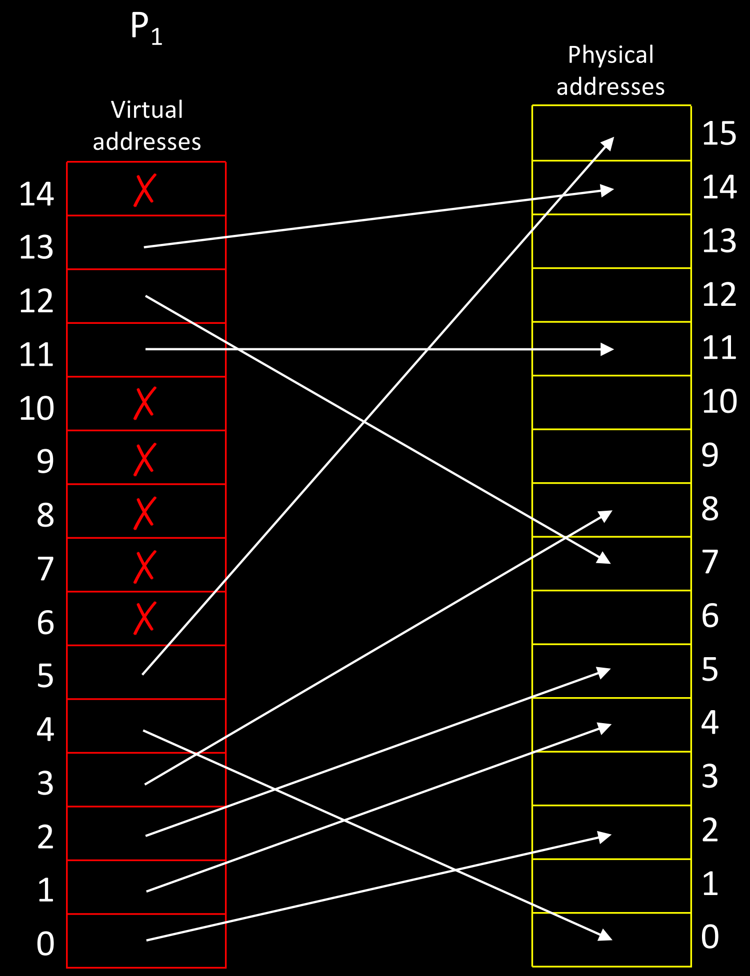
	Phys. Page	Valid
0	2	1
1	4	1
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		
13		
14		



Memory Paging

Page table of P_1

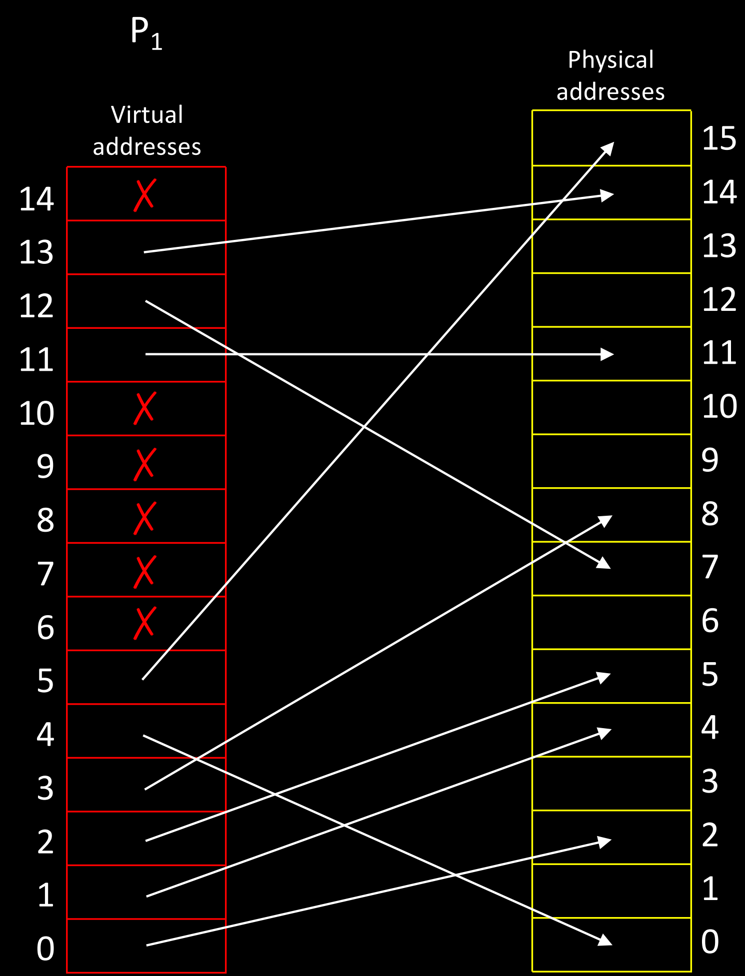
	Phys. Page	Valid
0	2	1
1	4	1
2	5	1
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		
13		
14		



Memory Paging

Page table of P_1

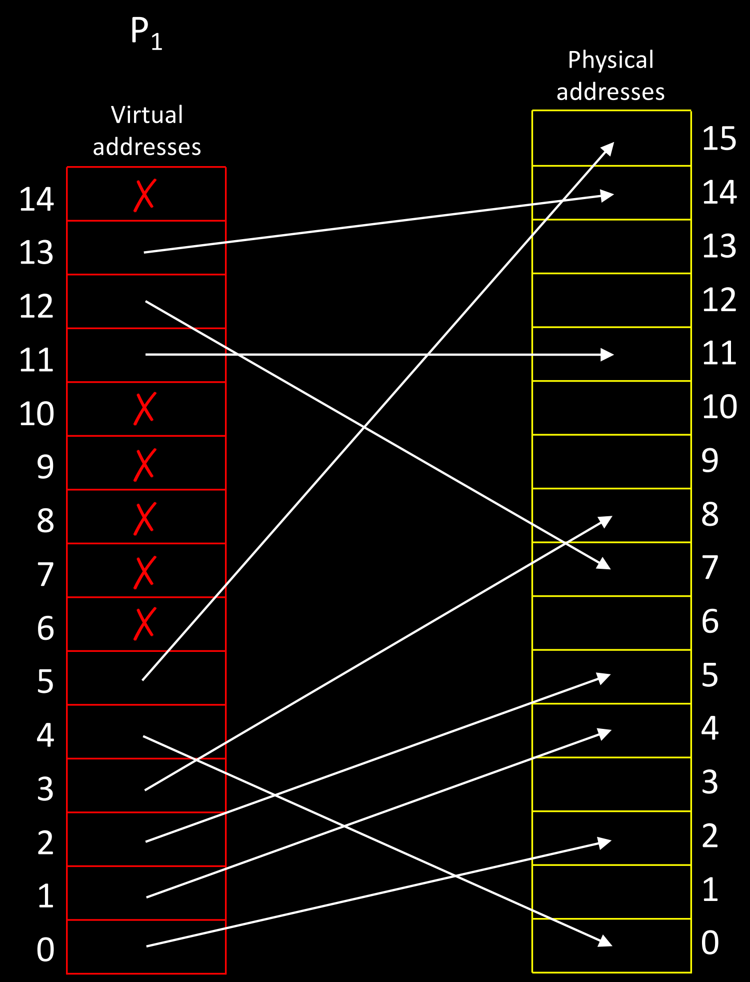
	Phys. Page	Valid
0	2	1
1	4	1
2	5	1
3	8	1
4	0	1
5	15	1
6		
7		
8		
9		
10		
11		
12		
13		
14		



Memory Paging

Page table of P_1

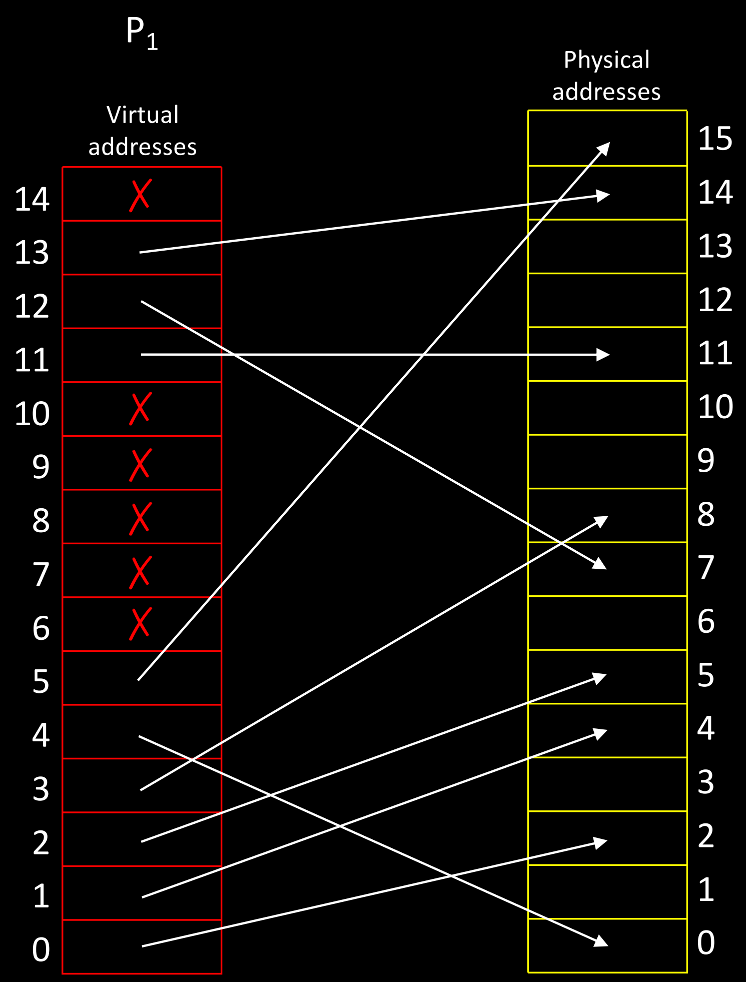
	Phys. Page	Valid
0	2	1
1	4	1
2	5	1
3	8	1
4	0	1
5	15	1
6		0
7		
8		
9		
10		
11		
12		
13		
14		



Memory Paging

Page table of P_1

	Phys. Page	Valid
0	2	1
1	4	1
2	5	1
3	8	1
4	0	1
5	15	1
6		0
7		0
8		0
9		0
10		0
11	11	1
12	7	1
13	14	1
14		0



Memory Paging

- Virtual to Physical address translation
  - We "just" need to convert virtual pages (VP) to physical pages (PP)
    - For each process
  - Use a table?

- How many virtual pages per process?

	Phys. Page	Valid
0	2	1
1	4	1
2	5	1
3	8	1
4	0	1
5	15	1
6		0
7		0
8		0
9		0
10		0
11	11	1
12	7	1
13	14	1
14		0

Memory Paging

- Virtual to Physical address translation
  - We “just” need to convert virtual pages (VP) to physical pages (PP)
    - For each process
  - Use a table?

- How many virtual pages per process?
  - 2^{20} entries in the table (~ 1 million)

	Phys. Page	Valid
0	2	1
1	4	1
2	5	1
3	8	1
4	0	1
5	15	1
6		0
7		0
8		0
9		0
10		0
11	11	1
12	7	1
13	14	1
14		0

Memory Paging

- Virtual to Physical address translation
  - We “just” need to convert virtual pages (VP) to physical pages (PP)
    - For each process
  - Use a table?

- How many virtual pages per process?
  - 2^{20} entries in the table (~ 1 million)
  - Each entry occupies 20 bits
    - Rounded to 32 bits = 4 bytes

	Phys. Page	Valid
0	2	1
1	4	1
2	5	1
3	8	1
4	0	1
5	15	1
6		0
7		0
8		0
9		0
10		0
11	11	1
12	7	1
13	14	1
14		0

Memory Paging

- Virtual to Physical address translation
  - We “just” need to convert virtual pages (VP) to physical pages (PP)
    - For each process
  - Use a table?

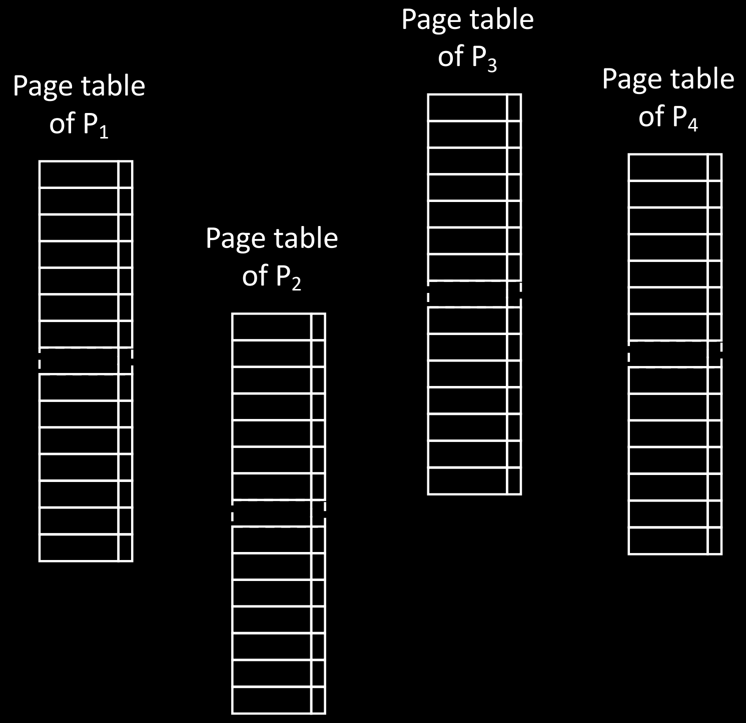
- How many virtual pages per process?
  - 2^{20} entries in the table (~ 1 million)
  - Each entry occupies 20 bits
    - Rounded to 32 bits = 4 bytes
  - 4 MB per process! Ouch!

	Phys. Page	Valid
0	2	1
1	4	1
2	5	1
3	8	1
4	0	1
5	15	1
6		0
7		0
8		0
9		0
10		0
11	11	1
12	7	1
13	14	1
14		0

Memory Paging

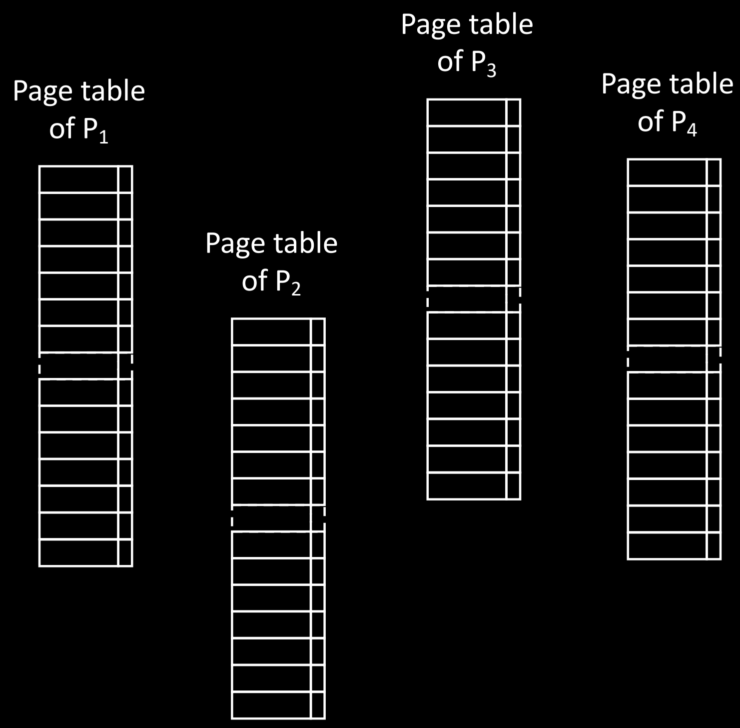
- Ok, now we have one 4MB-table per process
  - We'll see if we can reduce our memory footprint later

- Where are the tables stored?



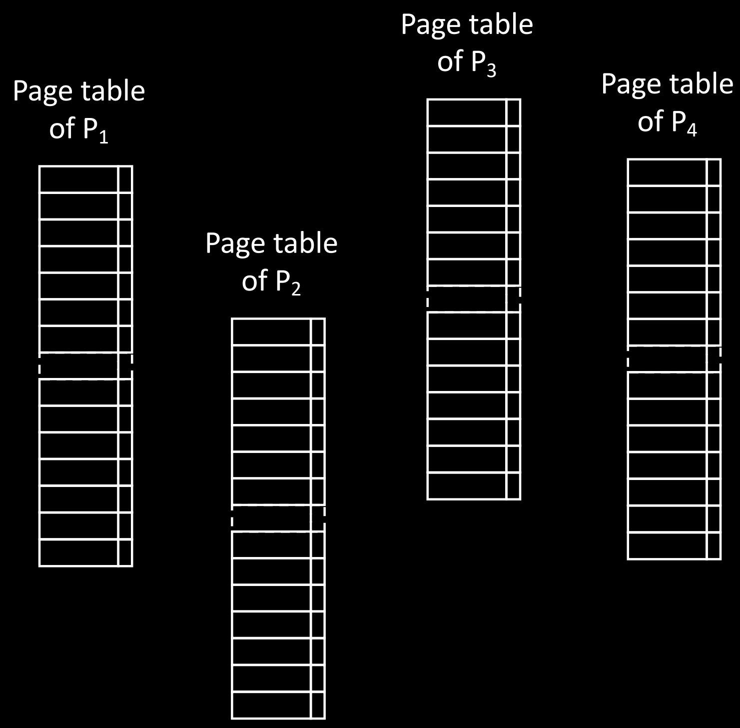
Memory Paging

- Ok, now we have one 4MB-table per process
  - We'll see if we can reduce our memory footprint later
- Where are the tables stored?
  - In RAM
    - Where else??



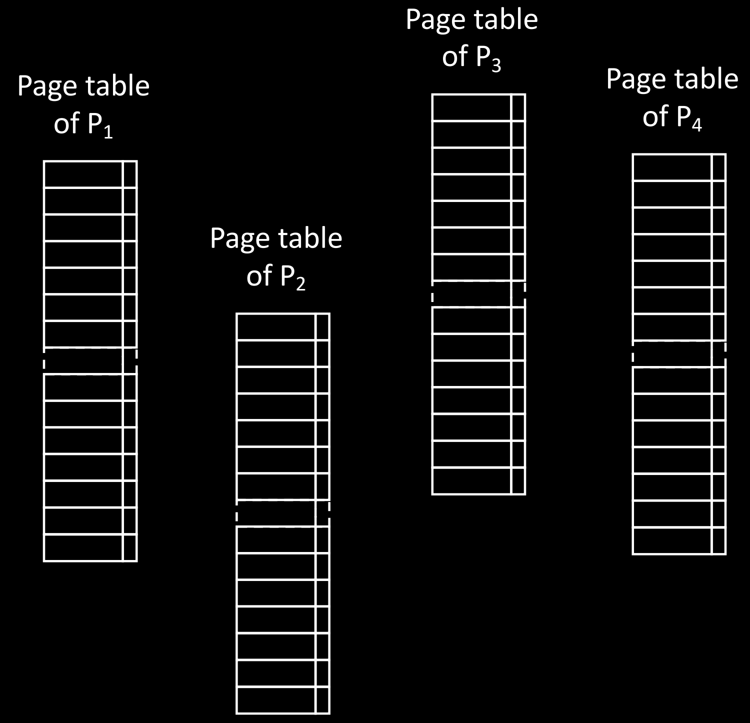
Address translation

- As usual, virtual to physical address translation must happen inside the CPU
  - The CPU thus needs to know which table should be used



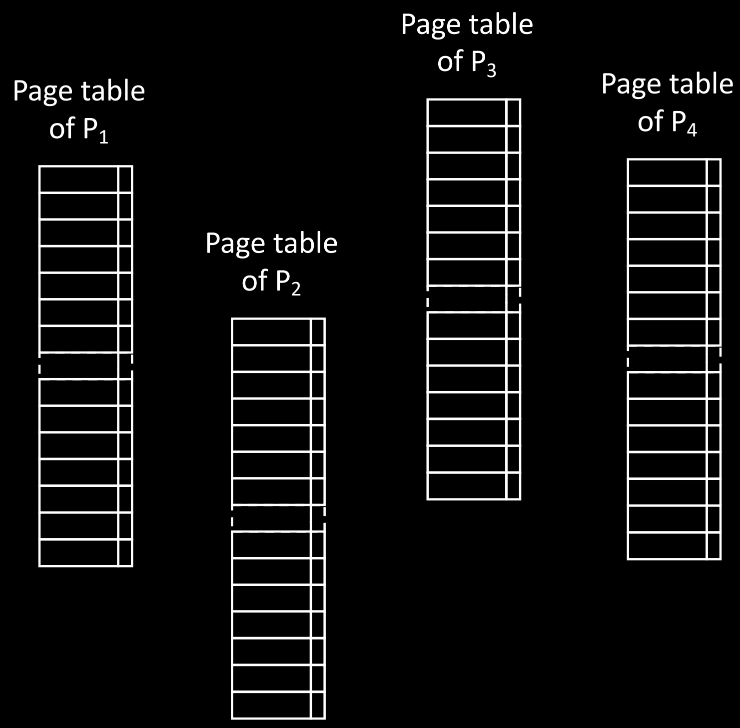
Address translation

- As usual, virtual to physical address translation must happen inside the CPU
  - The CPU thus needs to know which table should be used
  - The address of the "*current*" page table must be stored in a special register

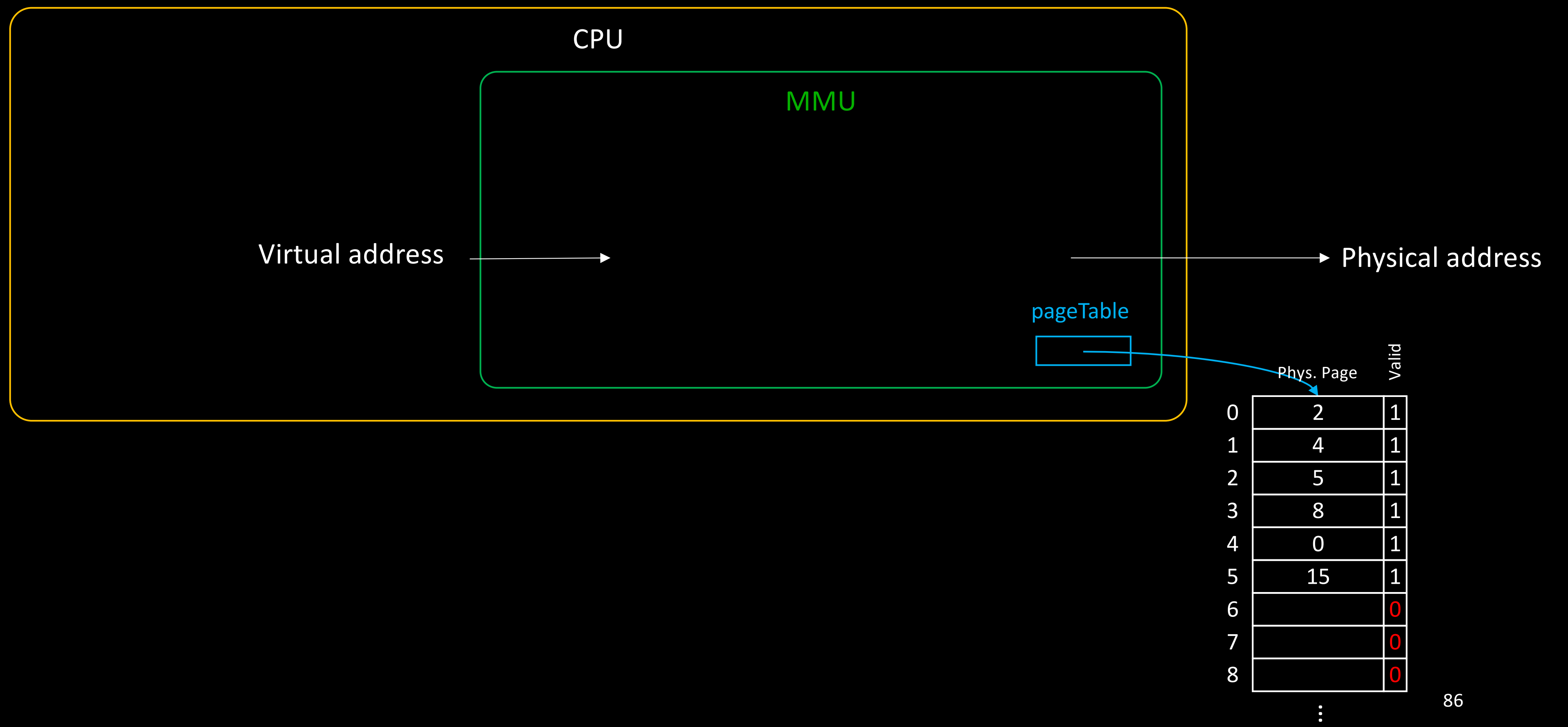


Address translation

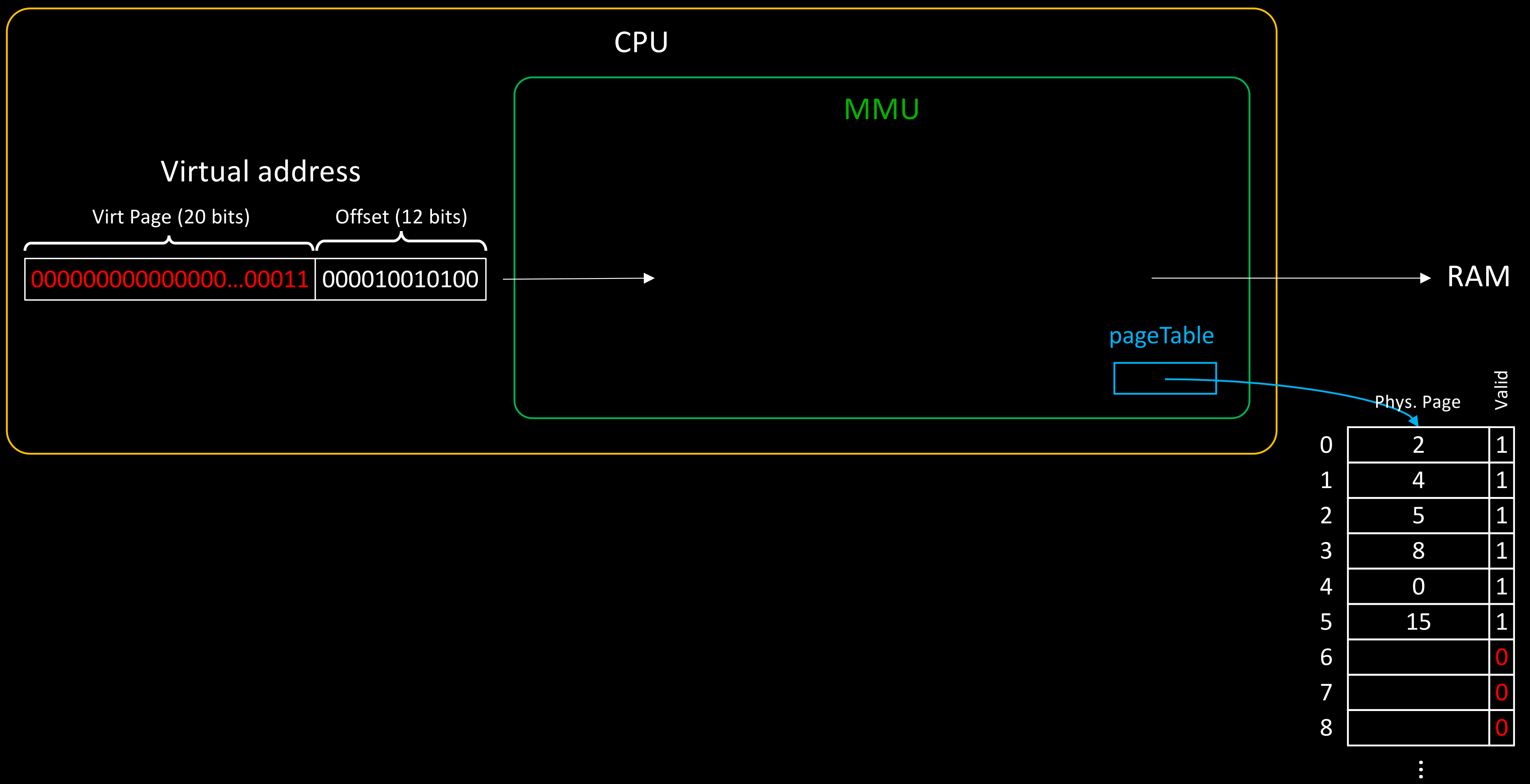
- As usual, virtual to physical address translation must happen inside the CPU
  - The CPU thus needs to know which table should be used
  - The address of the "*current*" page table must be stored in a special register
    - Updated at each context switch…
      …that changes current address space



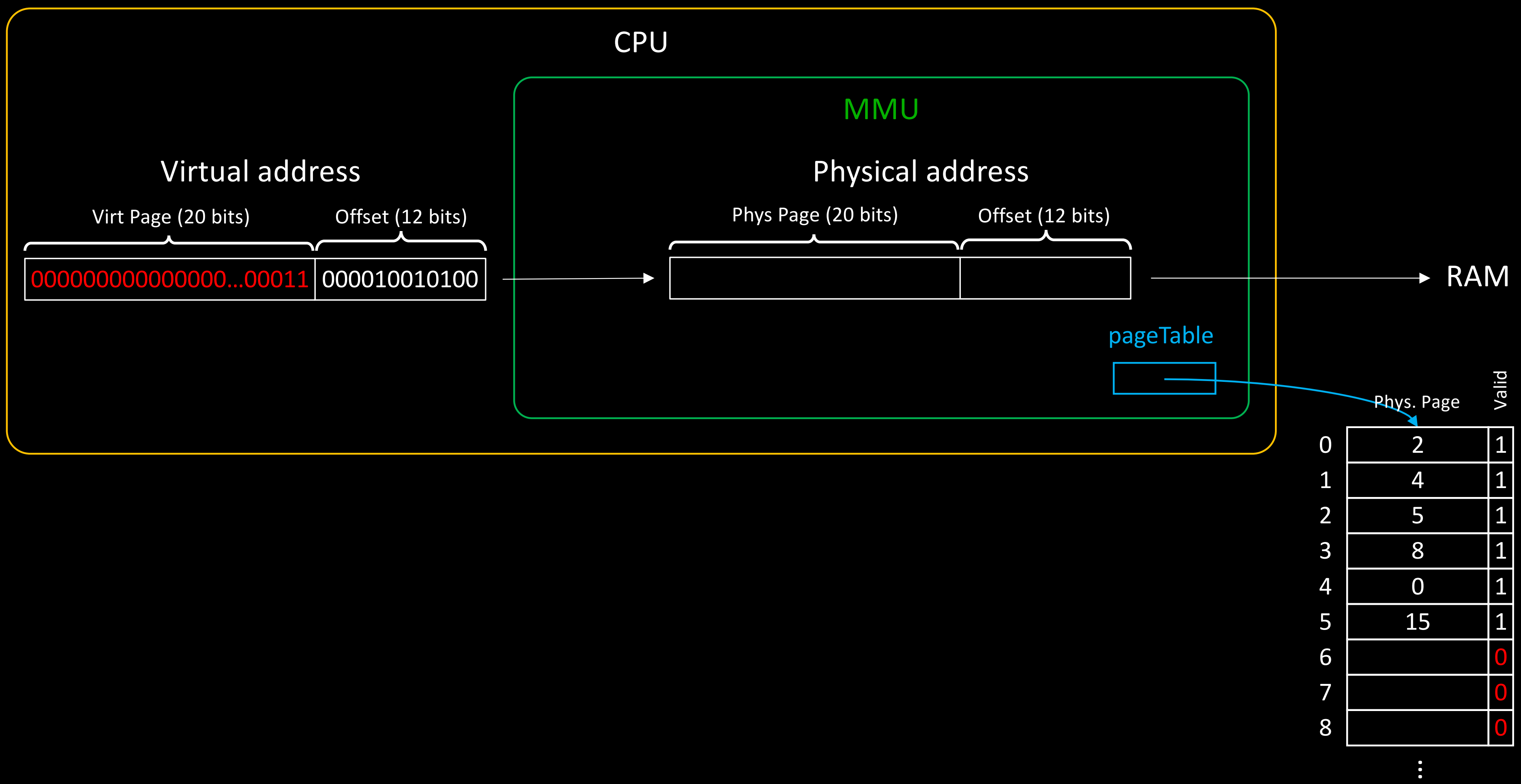
The Memory Management Unit (MMU)



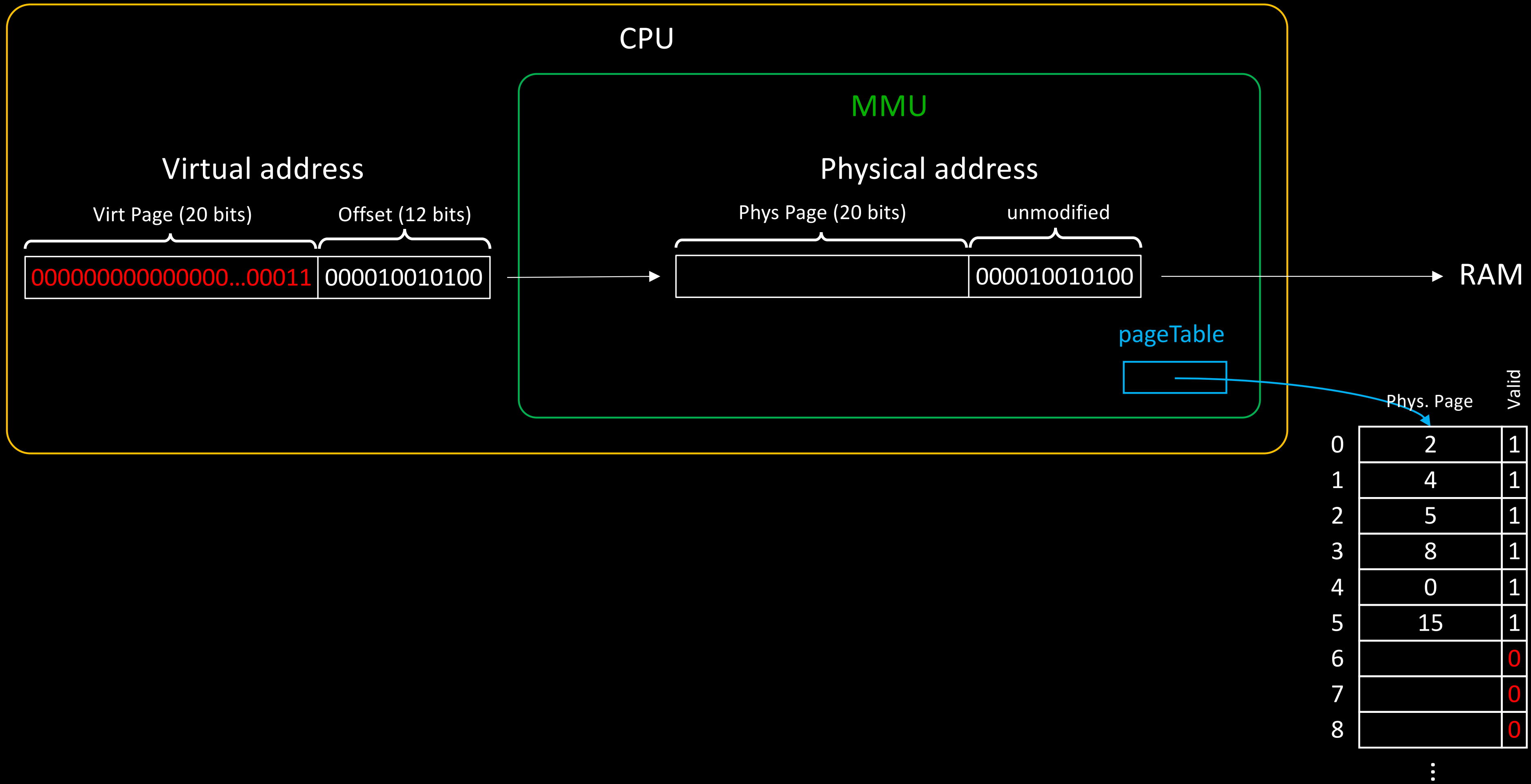
The Memory Management Unit (MMU)



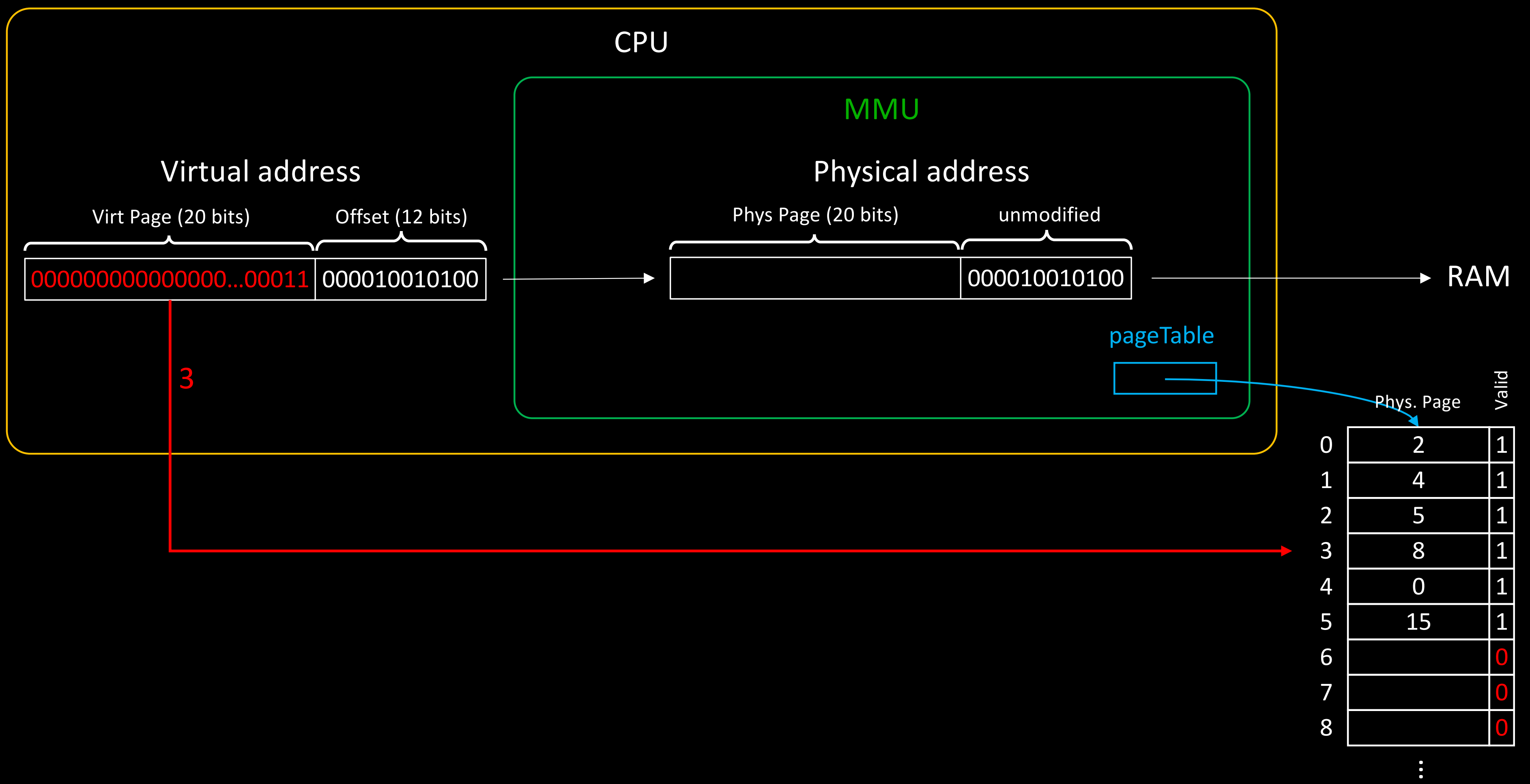
The Memory Management Unit (MMU)



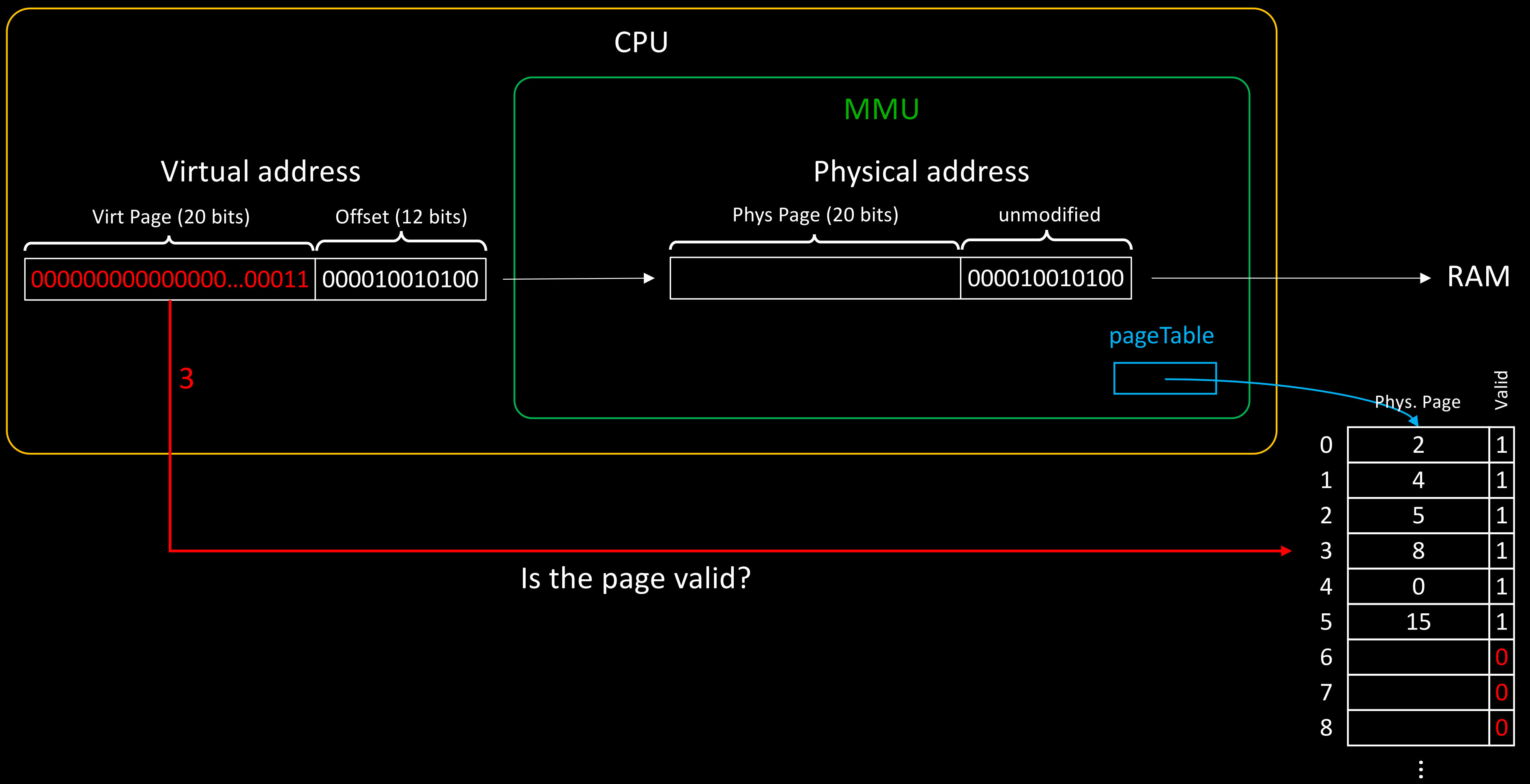
The Memory Management Unit (MMU)



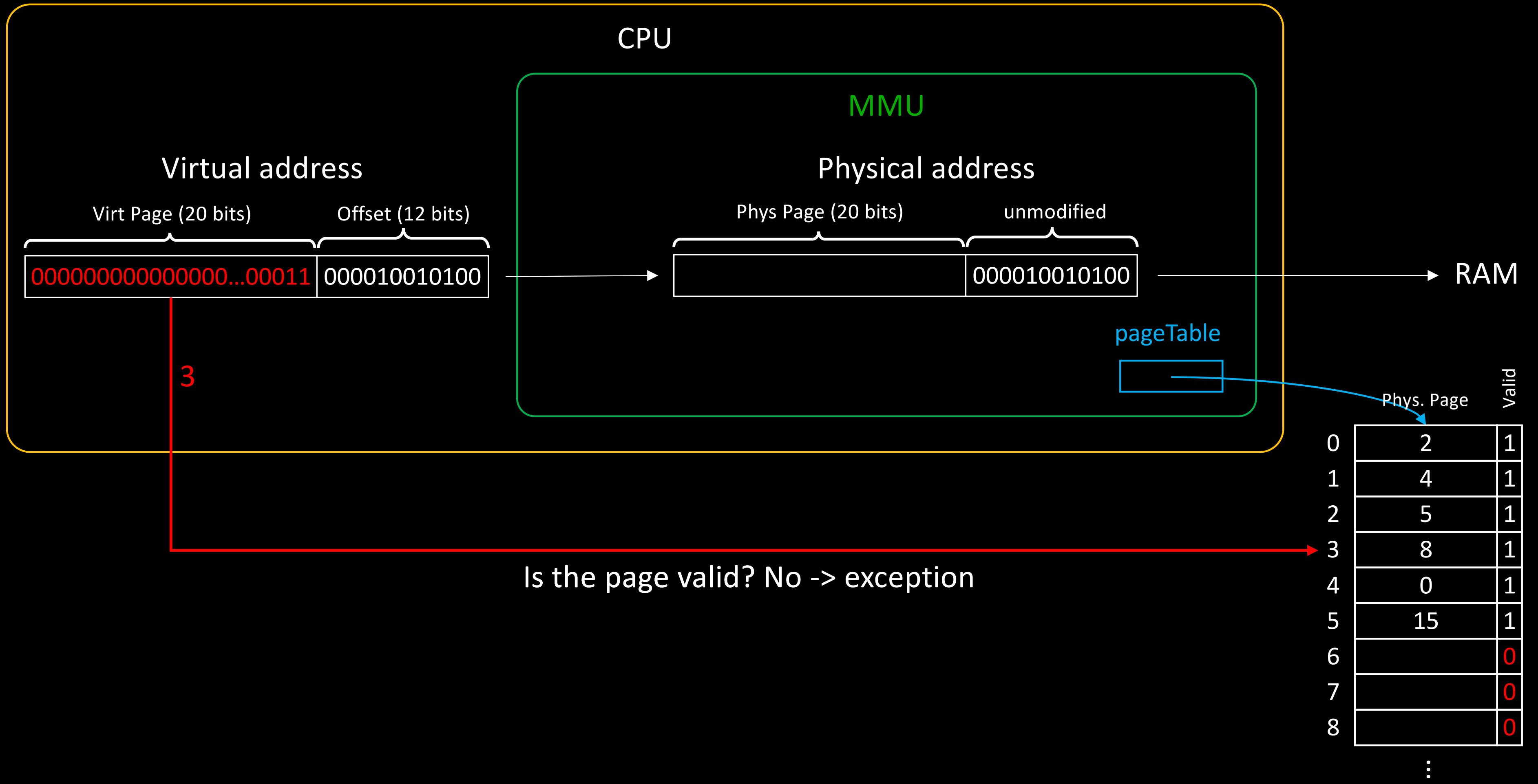
The Memory Management Unit (MMU)



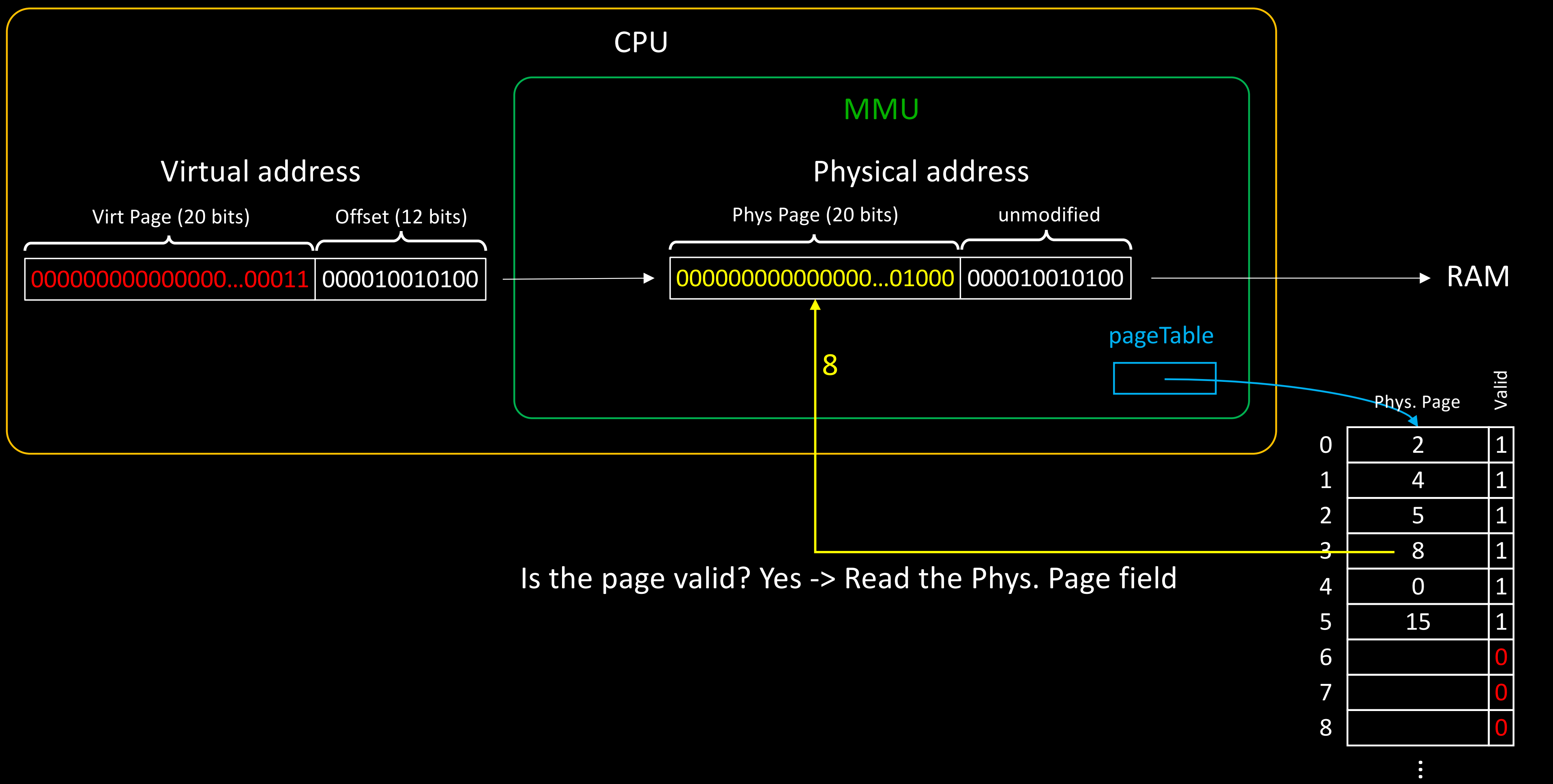
The Memory Management Unit (MMU)



The Memory Management Unit (MMU)



The Memory Management Unit (MMU)



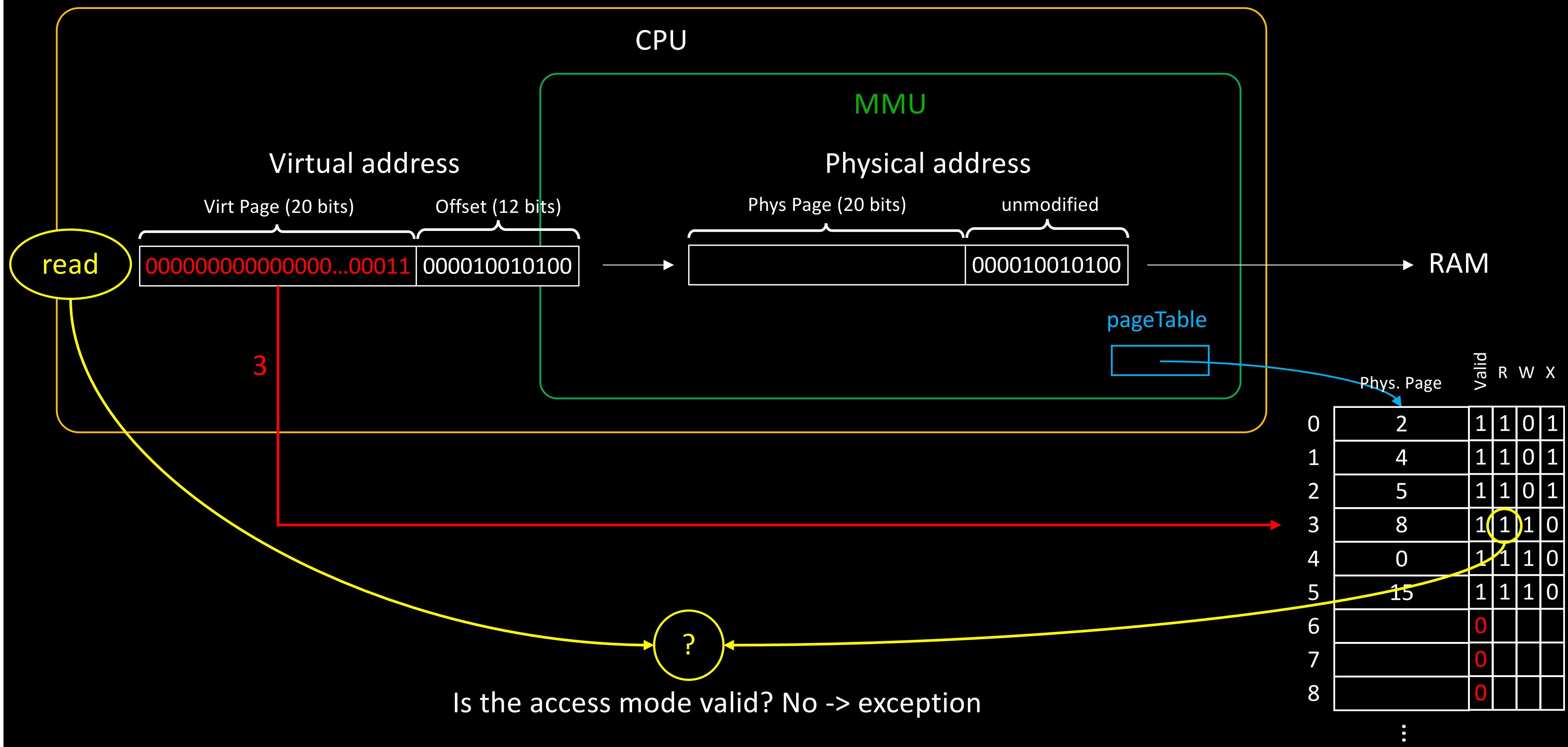
Is the page valid? Yes -> Read the Phys. Page field

The Memory Management Unit (MMU)

- Actually, page table entries feature additional access mode bits
  - R, W, X

	Phys. Page	Valid	R	W	X	
0	2	1	1	0	1	
1	4	1	1	0	1	RX
2	5	1	1	0	1	
3	8	1	1	1	0	
4	0	1	1	1	0	RW
5	15	1	1	1	0	
6		0				
7		0				
8		0				
9		0				
10		0				
11	11	1	1	1	0	
12	7	1	1	1	0	RW
13	14	1	1	1	0	
14		0				

The Memory Management Unit (MMU)



Is the access mode valid? No -> exception

The Memory Management Unit (MMU)

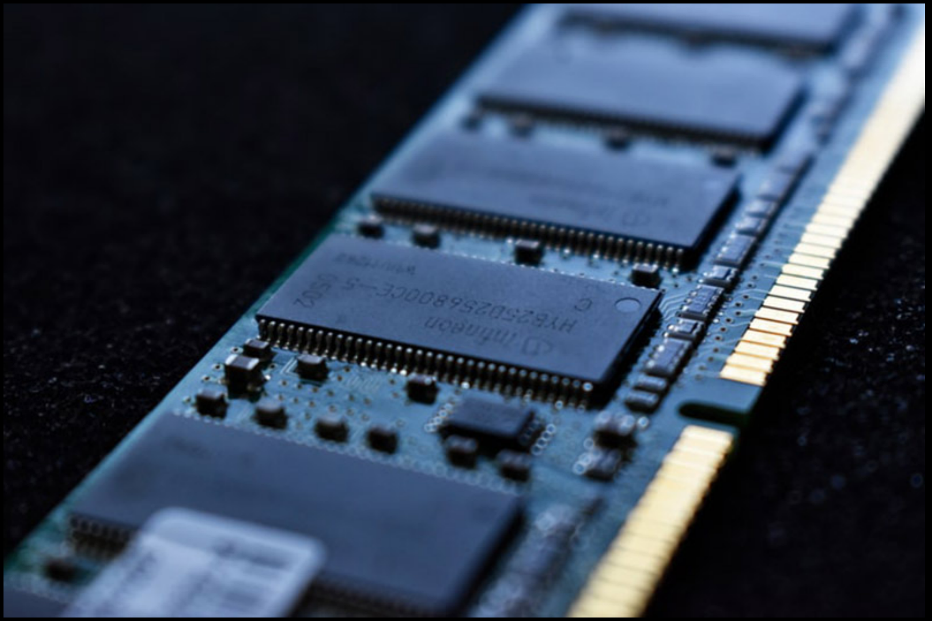
- Address translation is costly
  - MMU is a hardware circuit but…

The Memory Management Unit (MMU)

- Address translation is costly
  - MMU is a hardware circuit but…
  - Each memory access involves an implicit extra memory access!

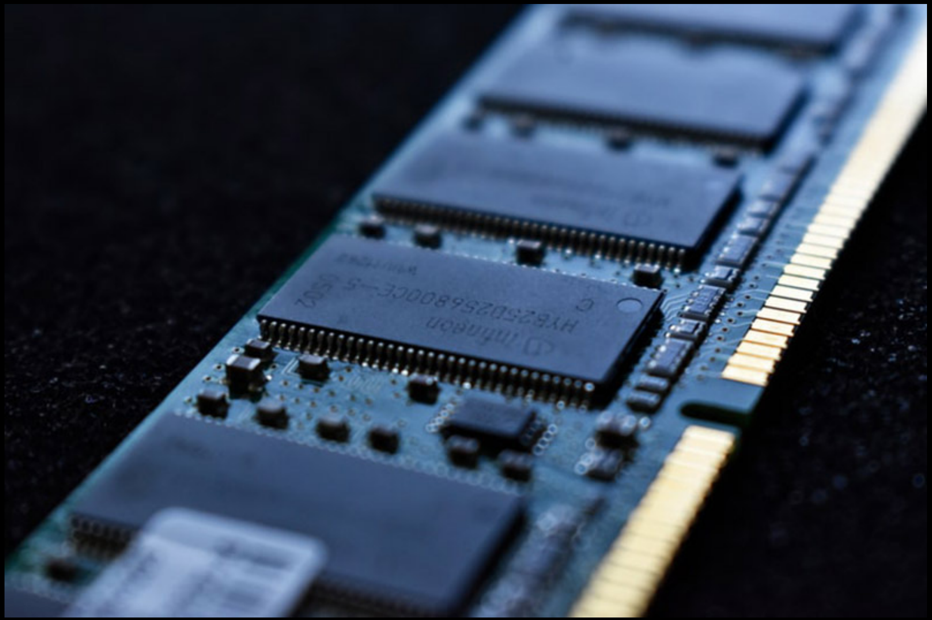
The Memory Management Unit (MMU)

- Address translation is costly
  - MMU is a hardware circuit but…
  - Each memory access involves an implicit extra memory access!
    - DDR RAM at 48 GB/s seems to run at 24 GB/s!



The Memory Management Unit (MMU)

- Address translation is costly
  - MMU is a hardware circuit but…
  - Each memory access involves an implicit extra memory access!
    - DDR RAM at 48 GB/s seems to run at 24 GB/s!
    - Under MS-DOS, we would get the raw performance ☺



Memory Paging

- So we have two serious problems
  - Memory footprint of page tables
  - Overhead of page table accesses

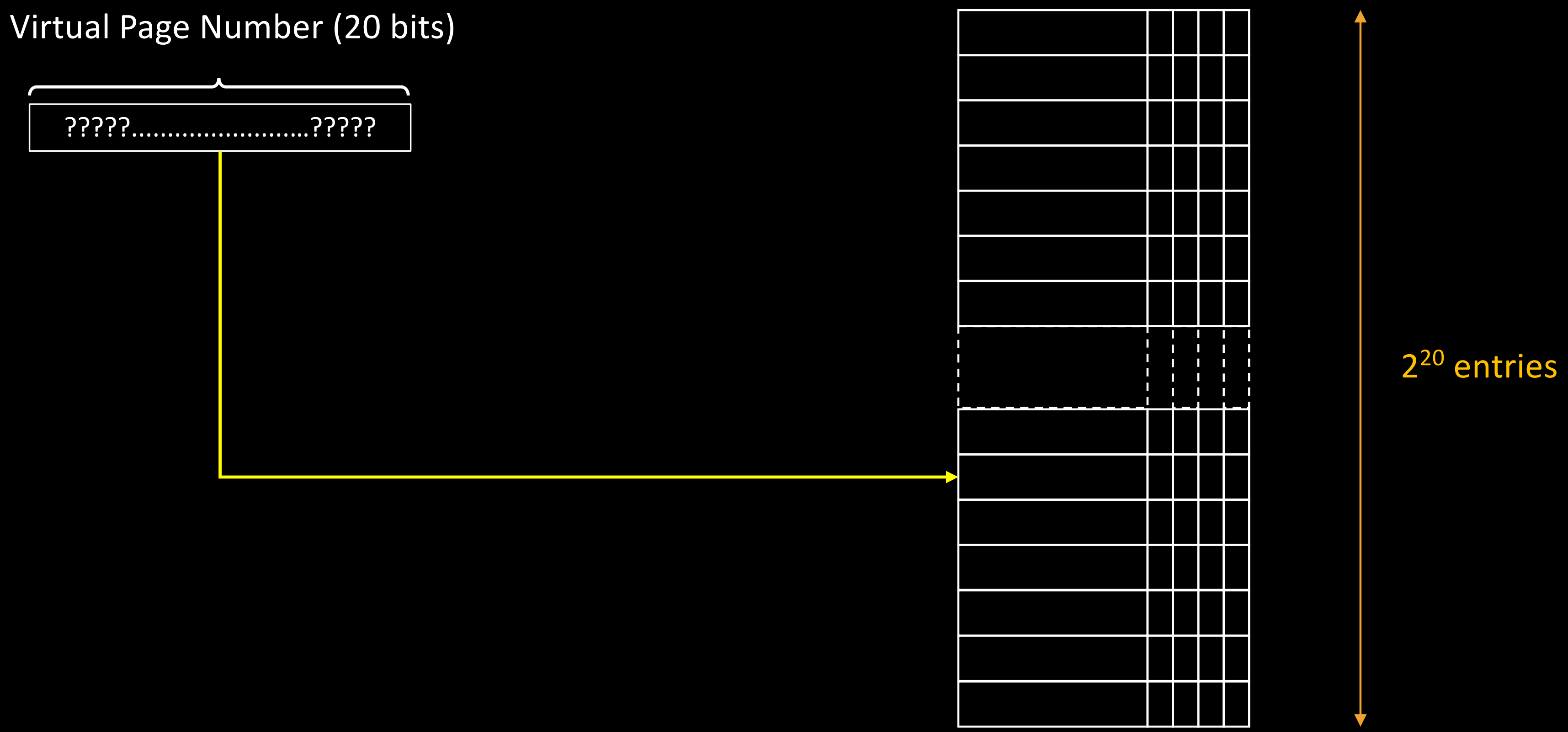
Reducing Memory Footprint

- Fact
  - Page tables contain plenty of invalid pages
    - Large contiguous series of invalid pages

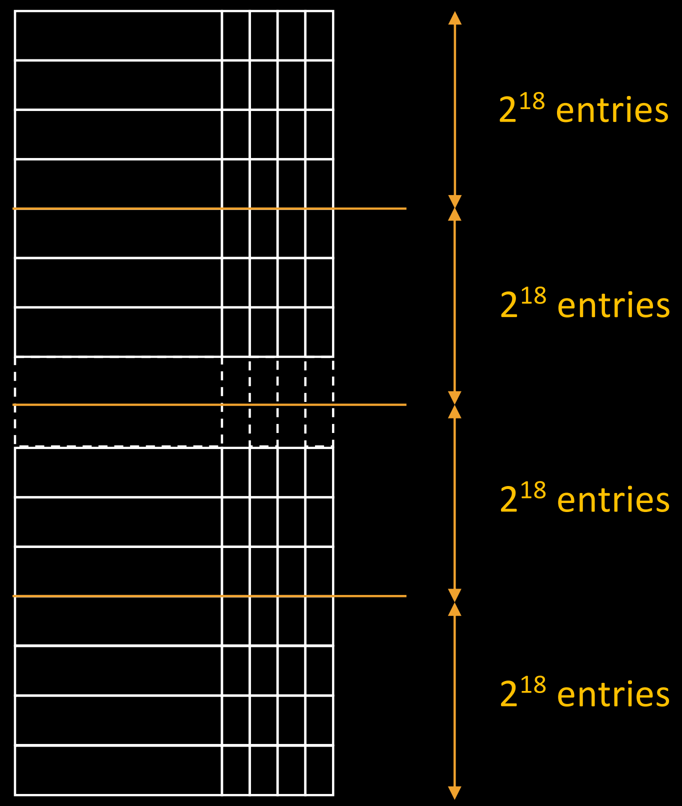
Reducing Memory Footprint

- Fact
  - Page tables contain plenty of invalid pages
    - Large contiguous series of invalid pages
- Idea
  - Compress invalid chunks?
    - How to do that without loosing the “array indexing” property?

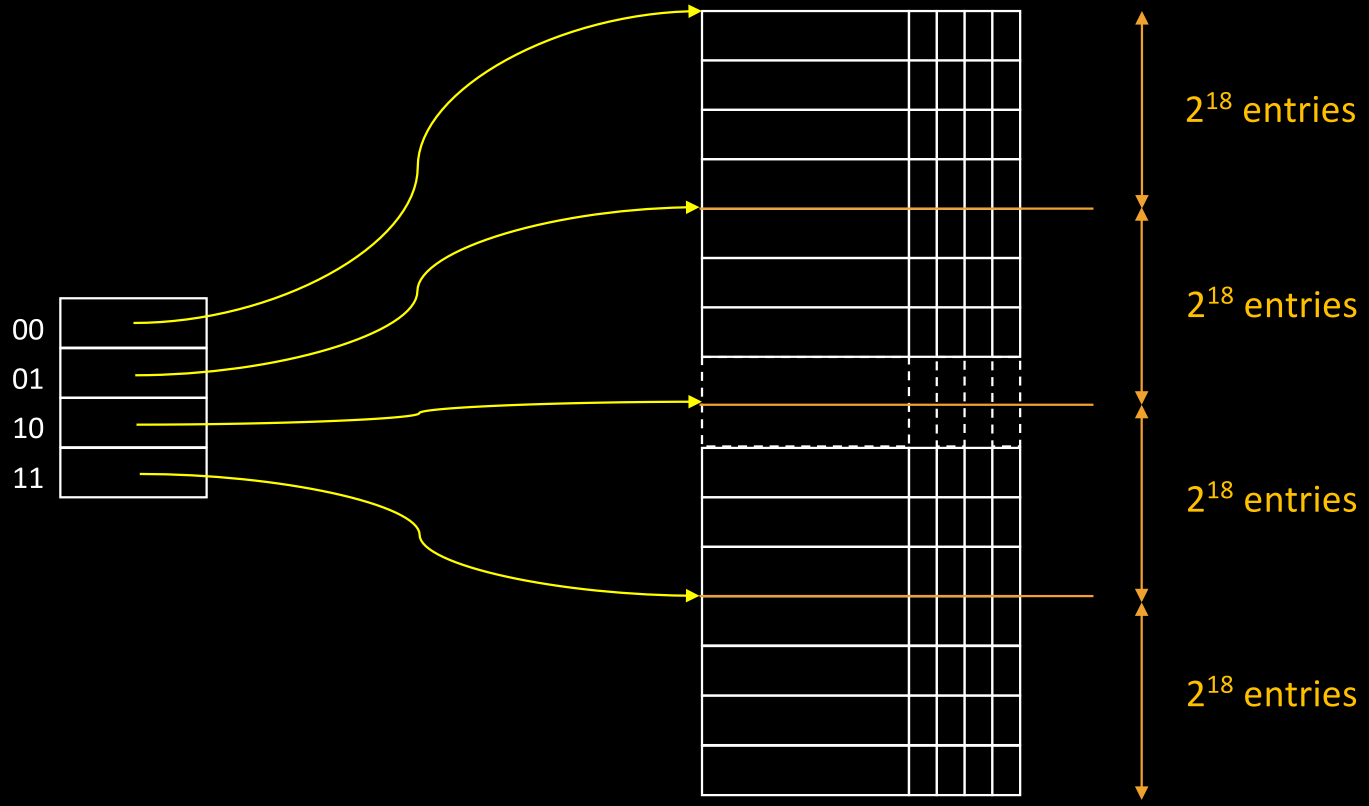
Reducing Memory Footprint



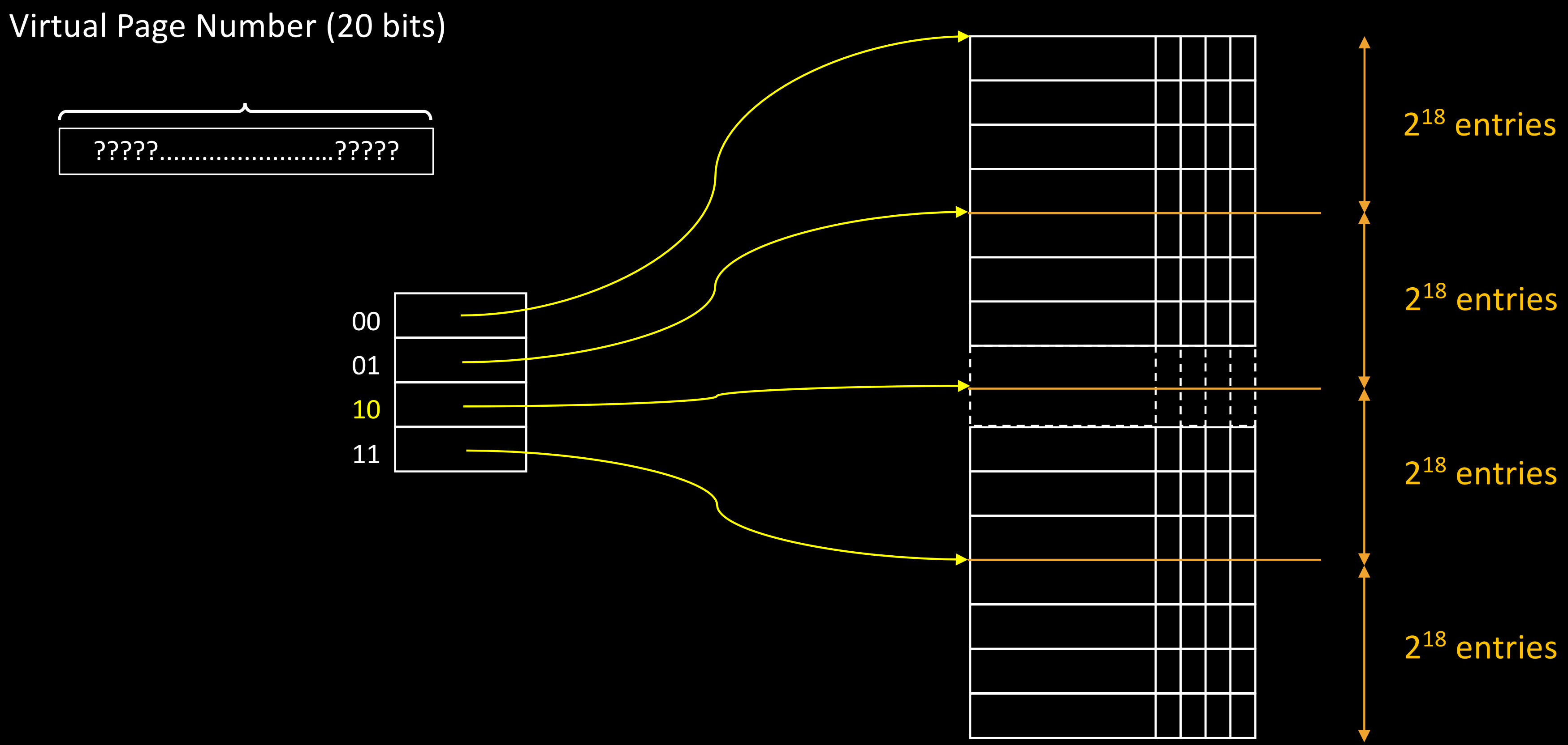
Reducing Memory Footprint



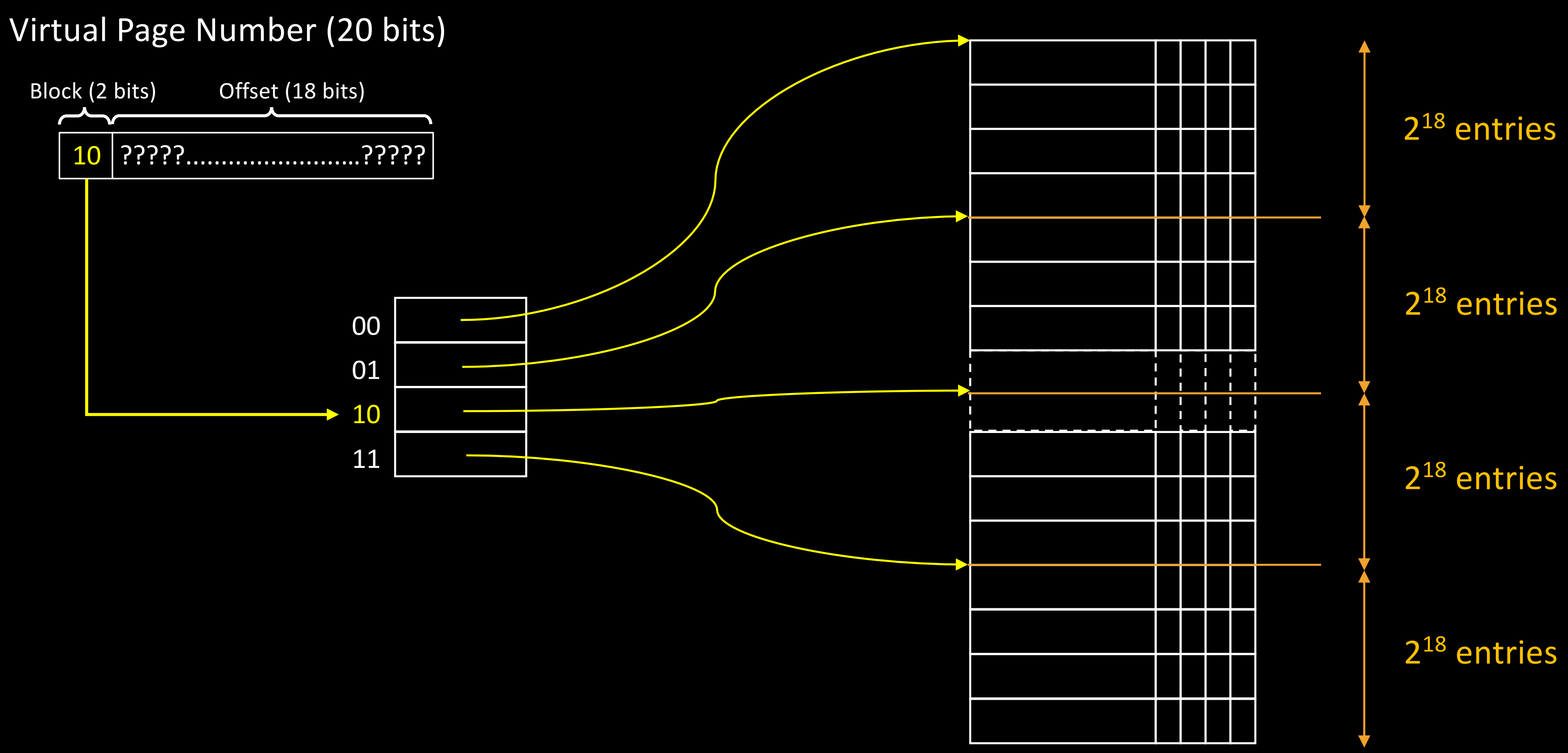
Reducing Memory Footprint



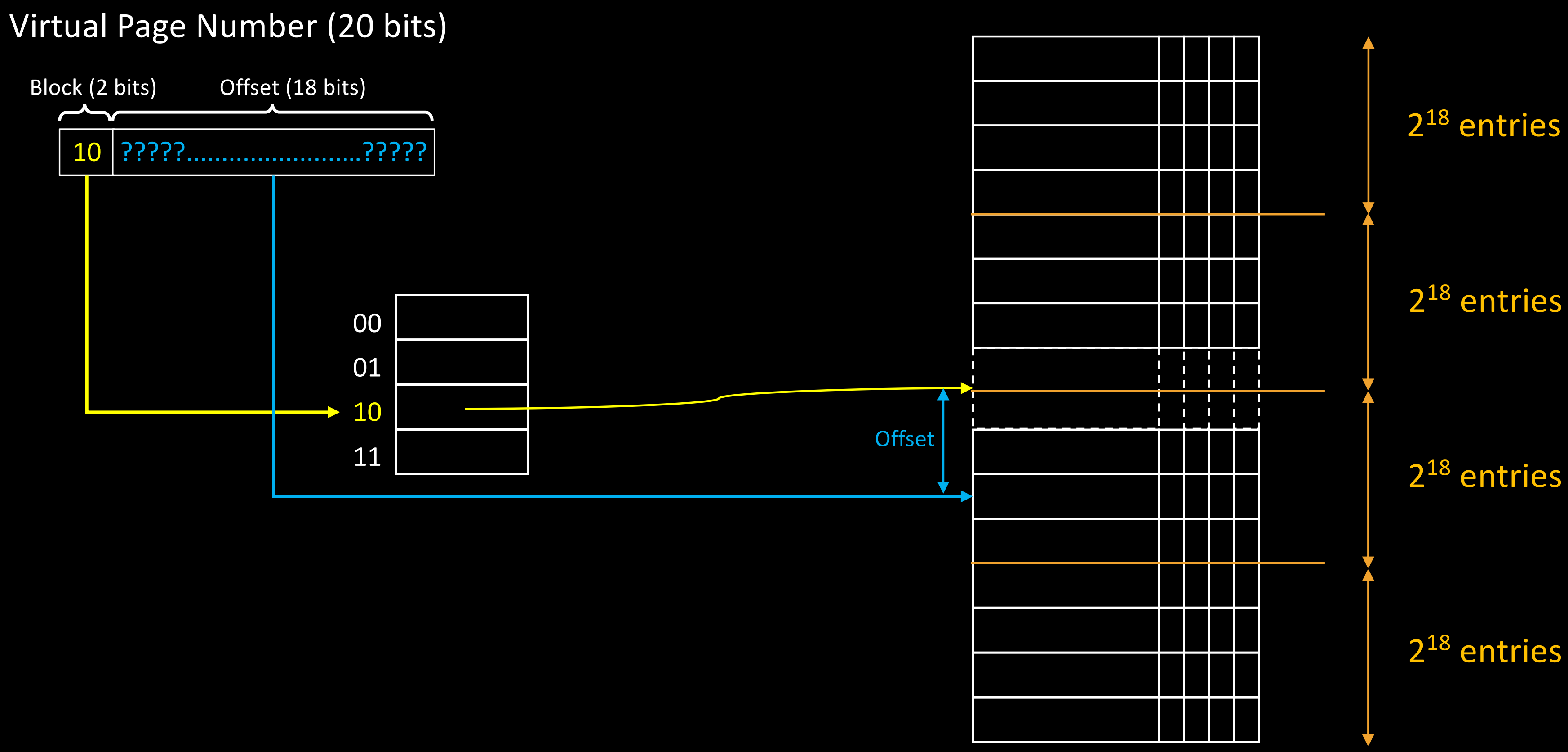
Reducing Memory Footprint



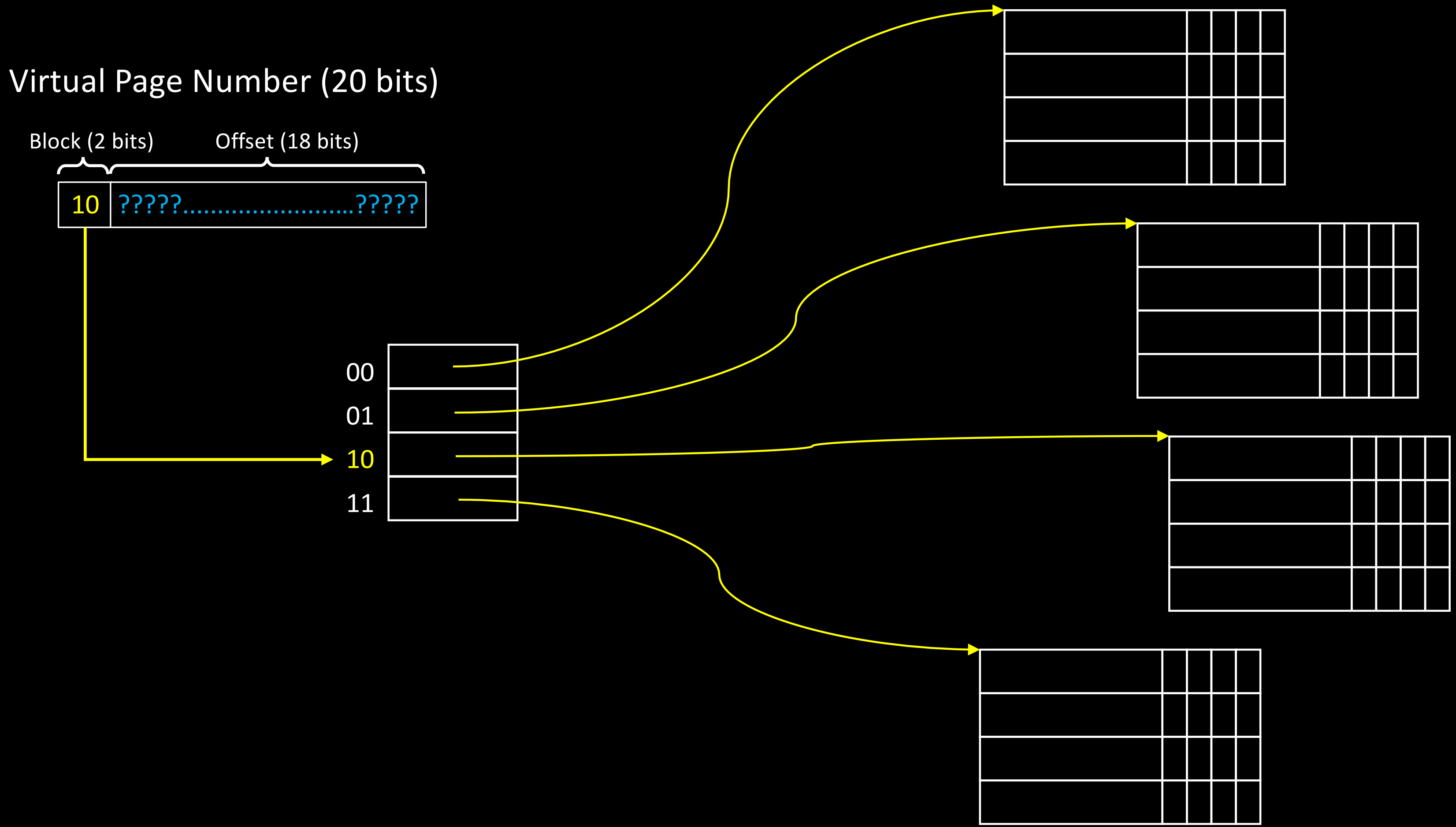
Reducing Memory Footprint



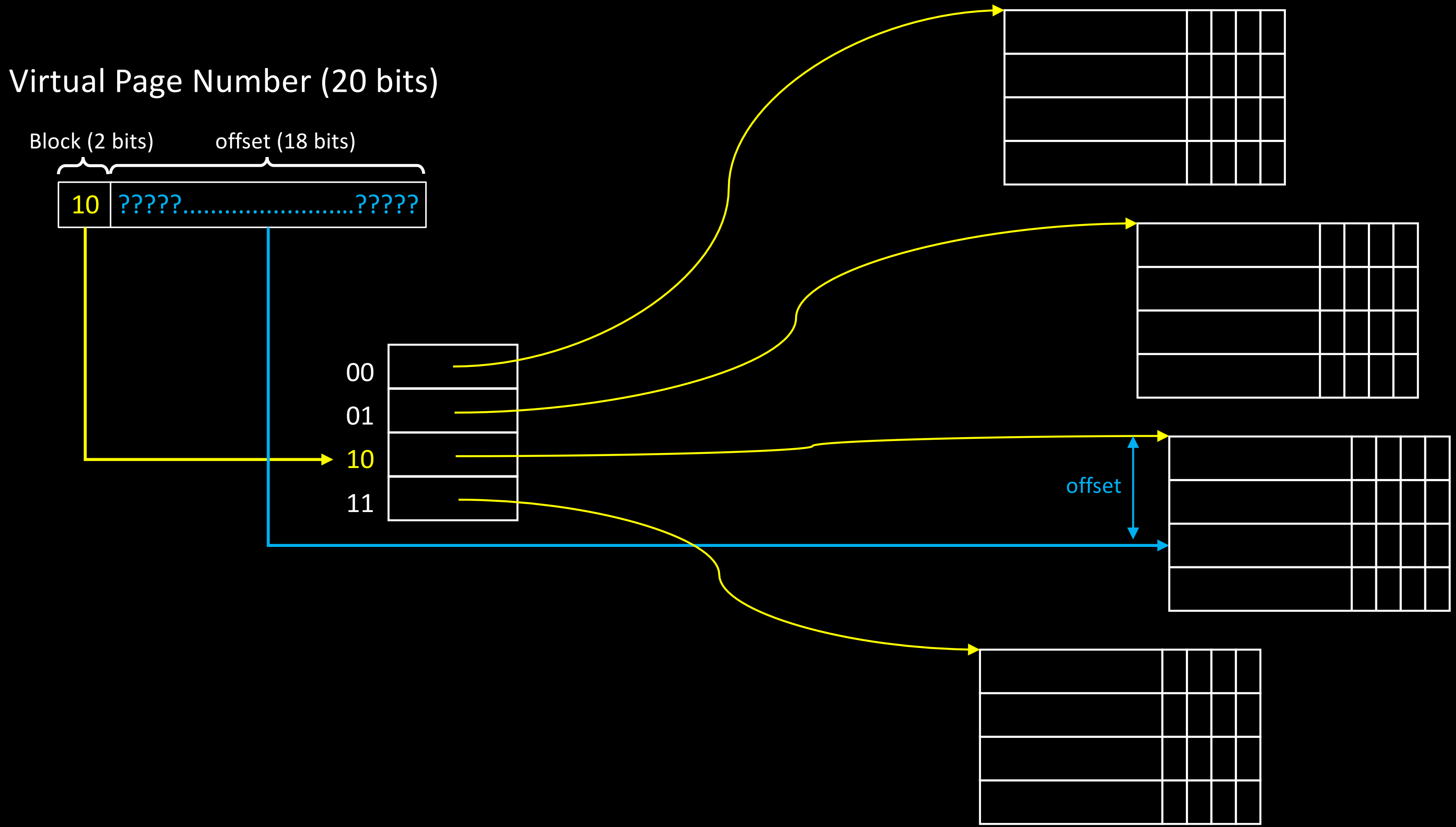
Reducing Memory Footprint



Reducing Memory Footprint

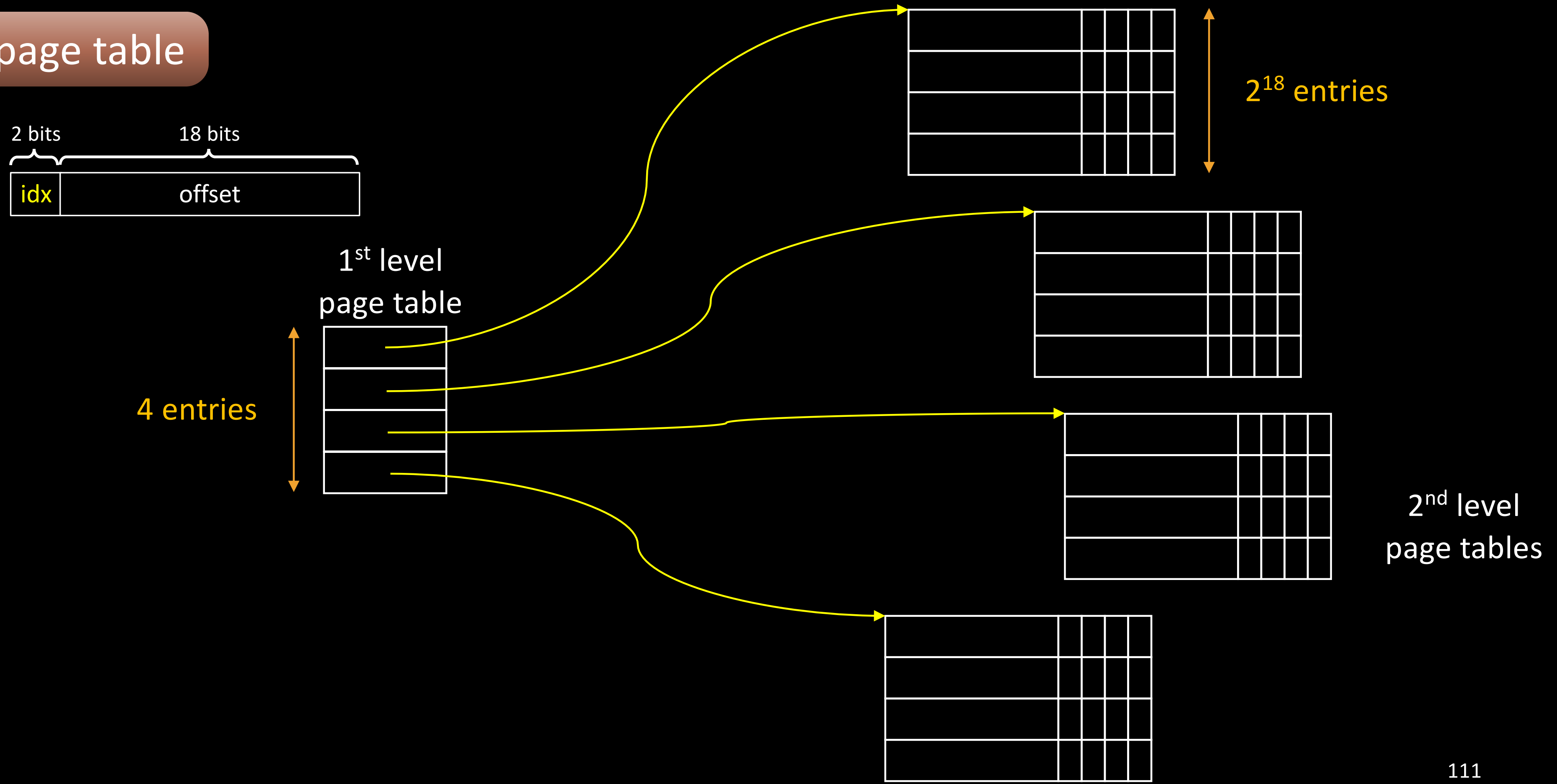


Reducing Memory Footprint



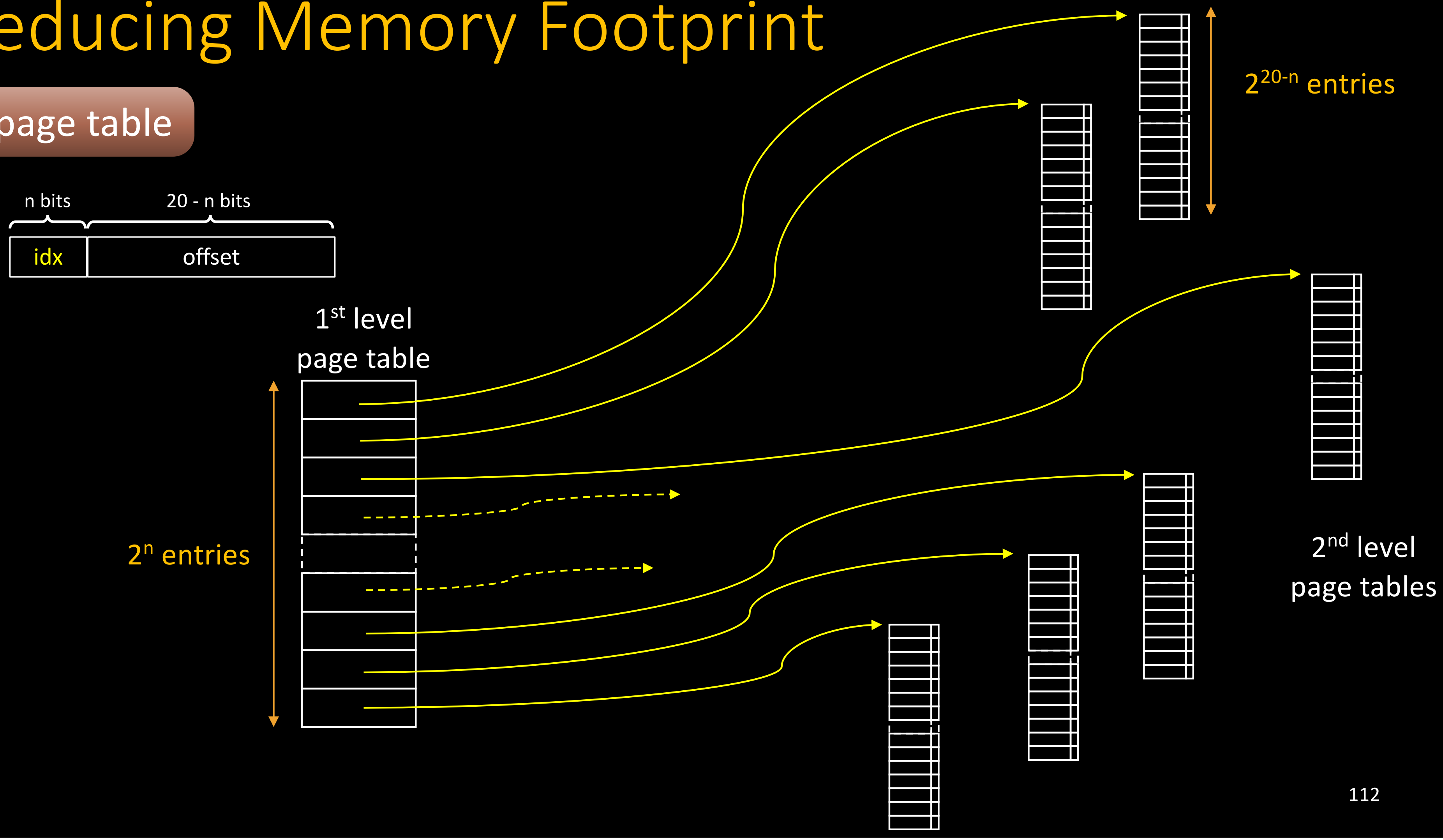
Reducing Memory Footprint

2-level page table

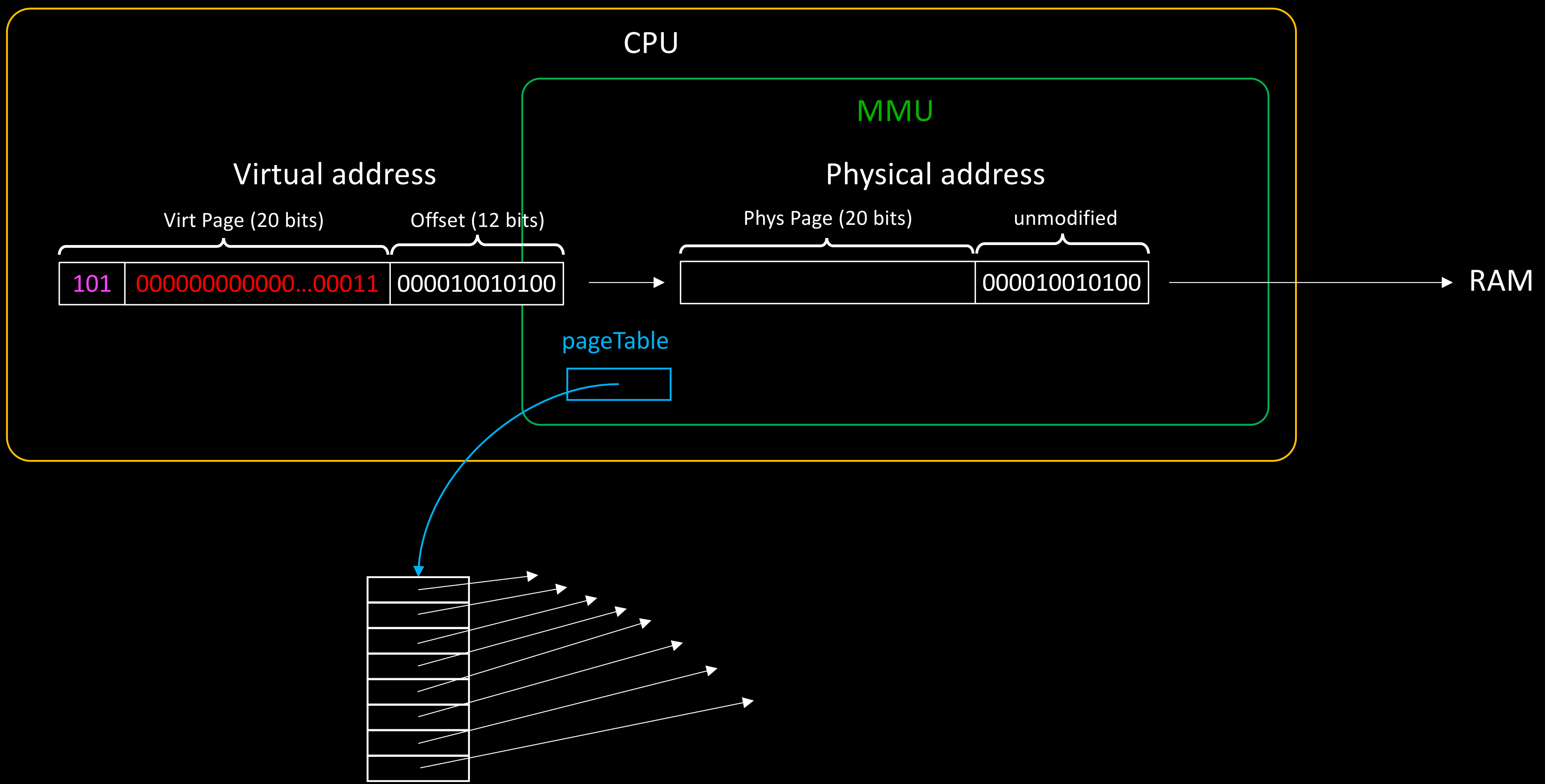


Reducing Memory Footprint

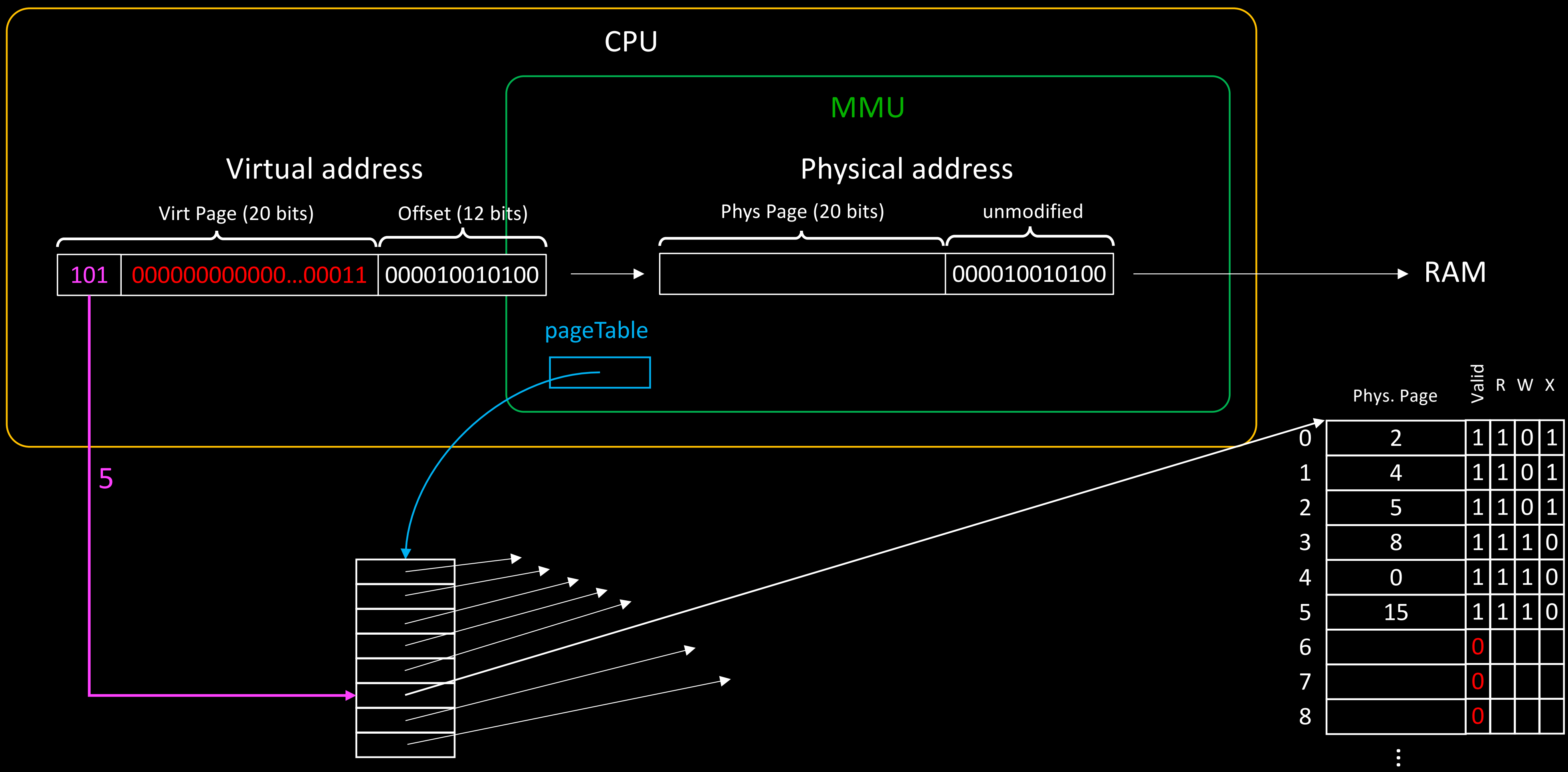
2-level page table



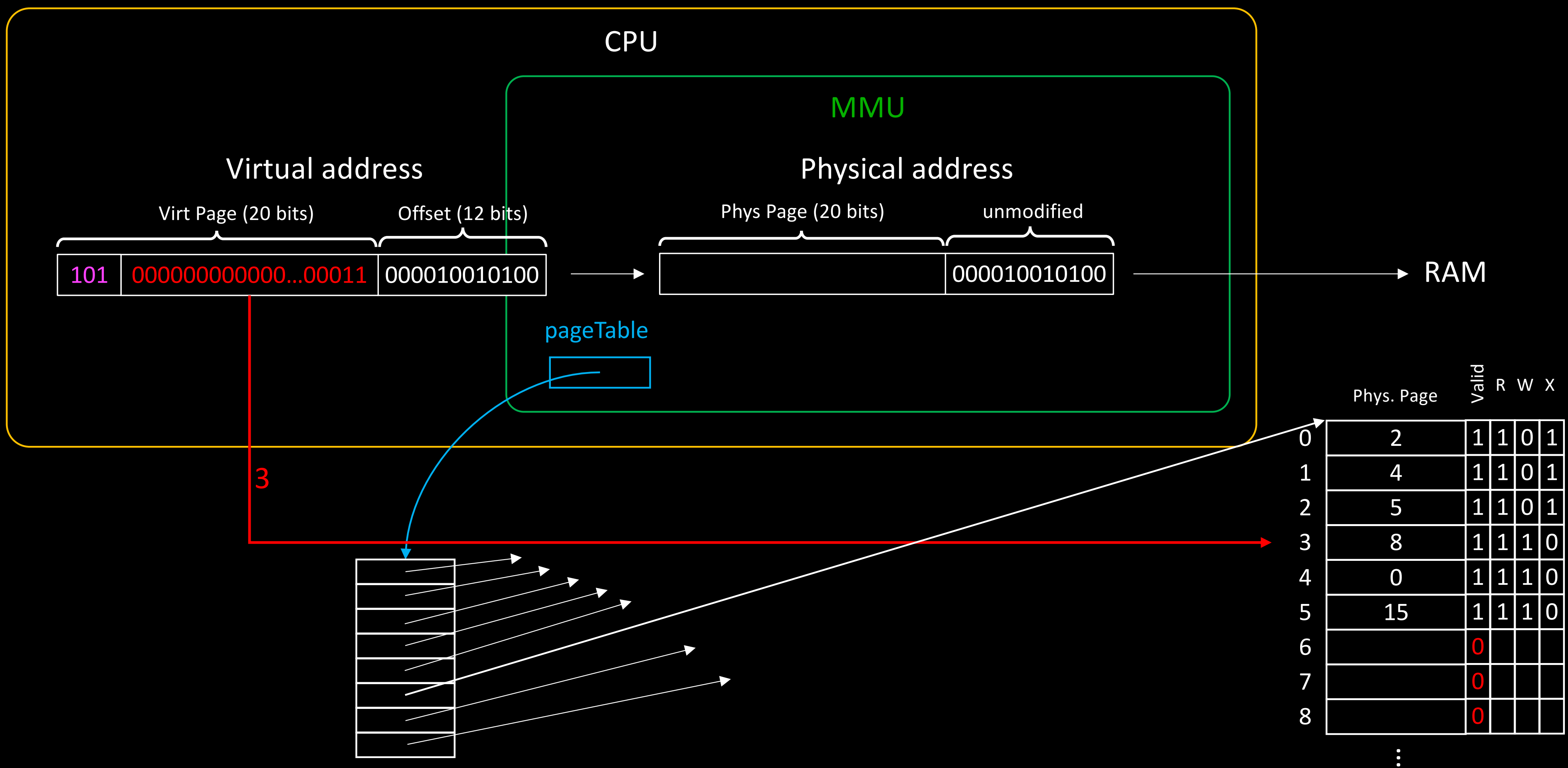
MMU and 2-level tables



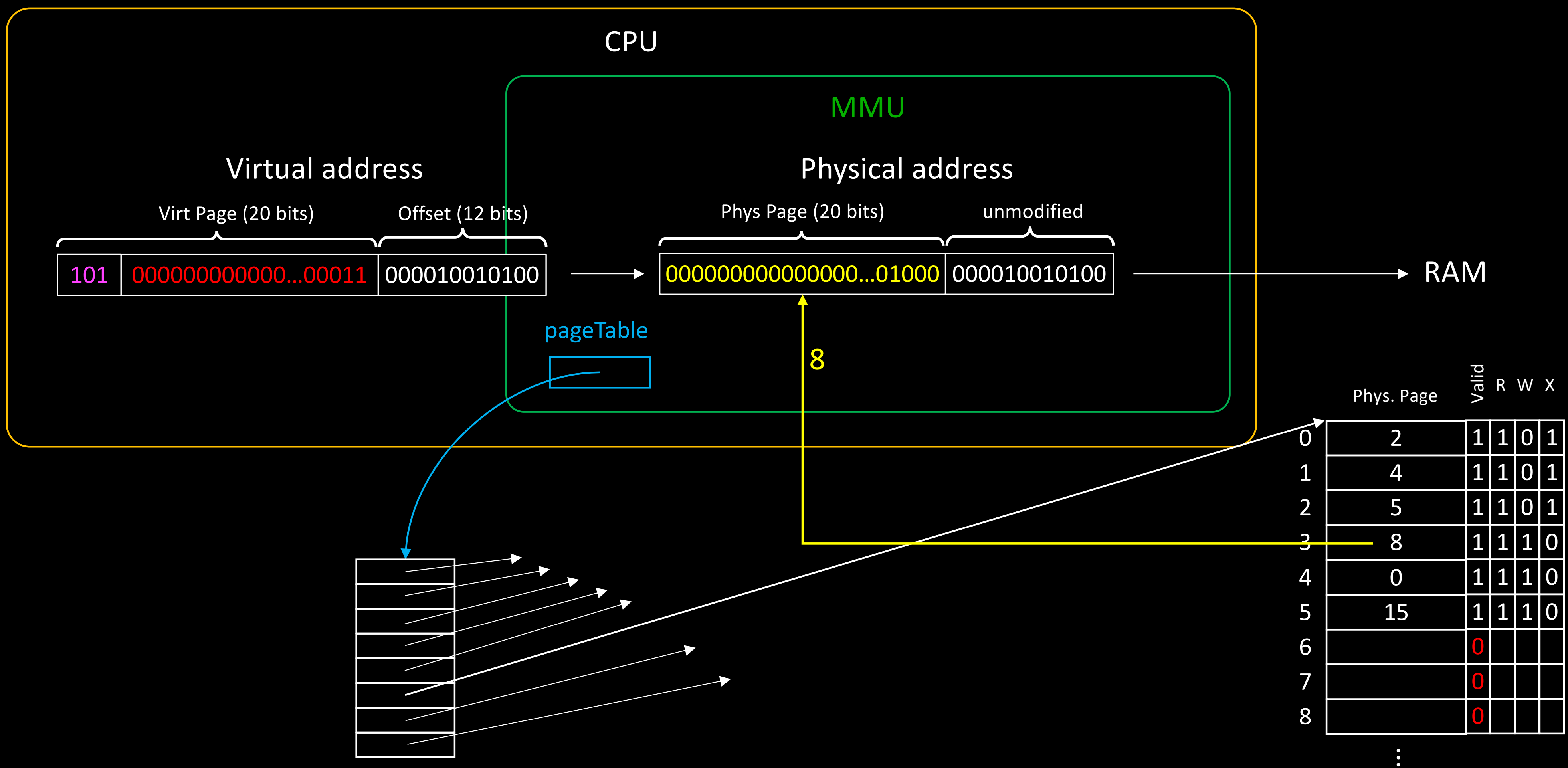
MMU and 2-level tables



MMU and 2-level tables



MMU and 2-level tables

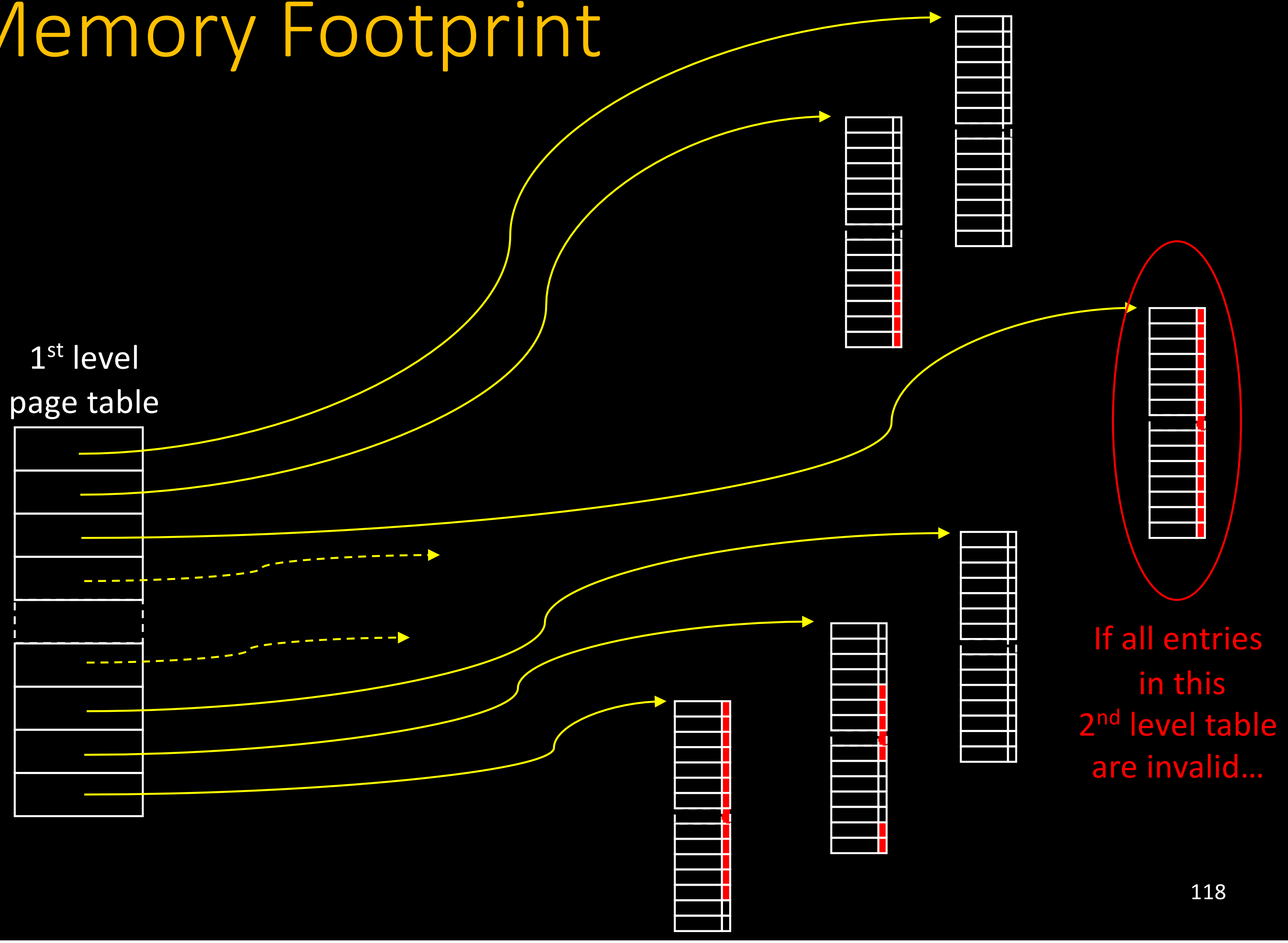


What’s the point of doing that?

The memory footprint is even worse!

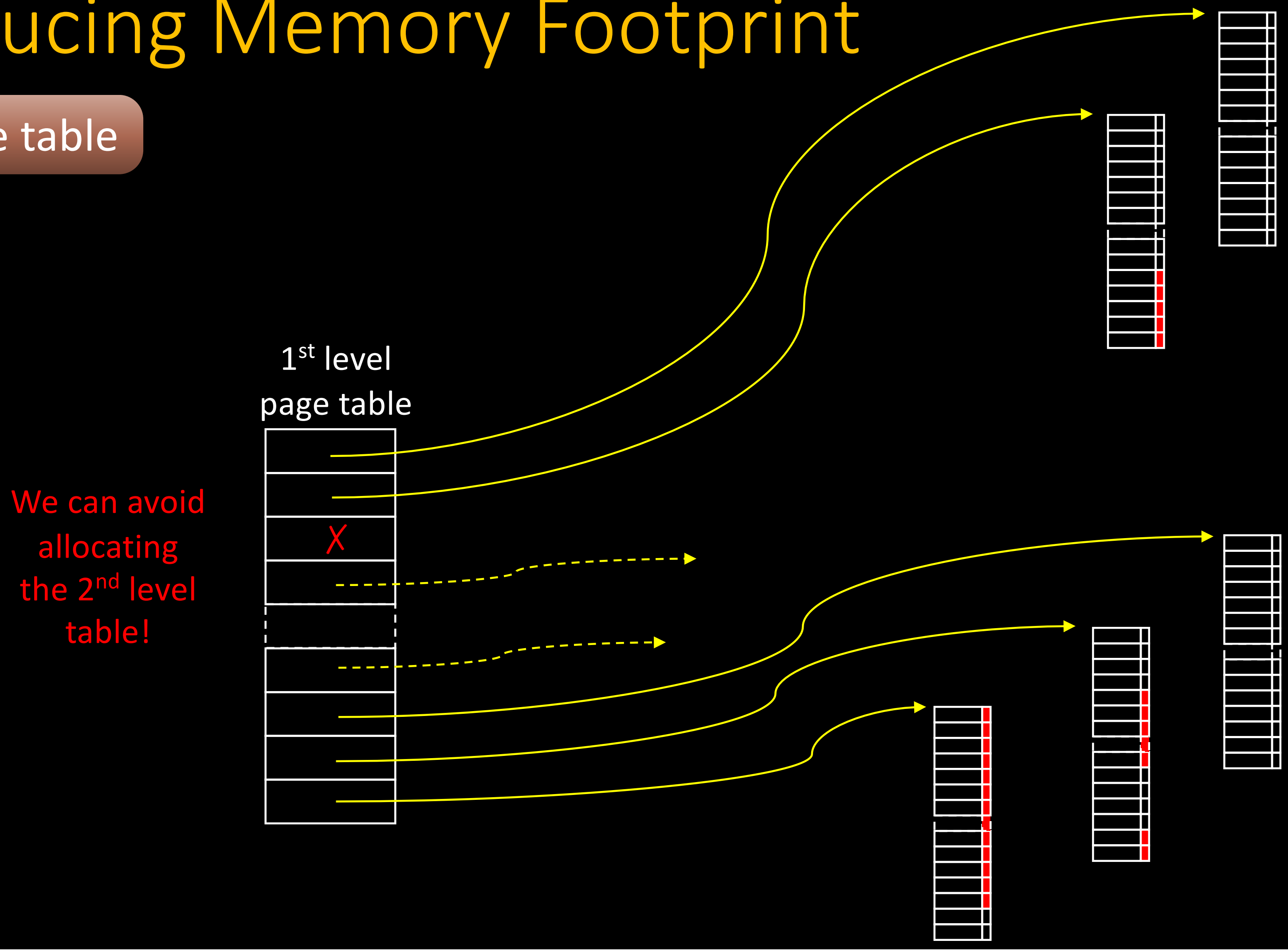
Reducing Memory Footprint

2-level page table



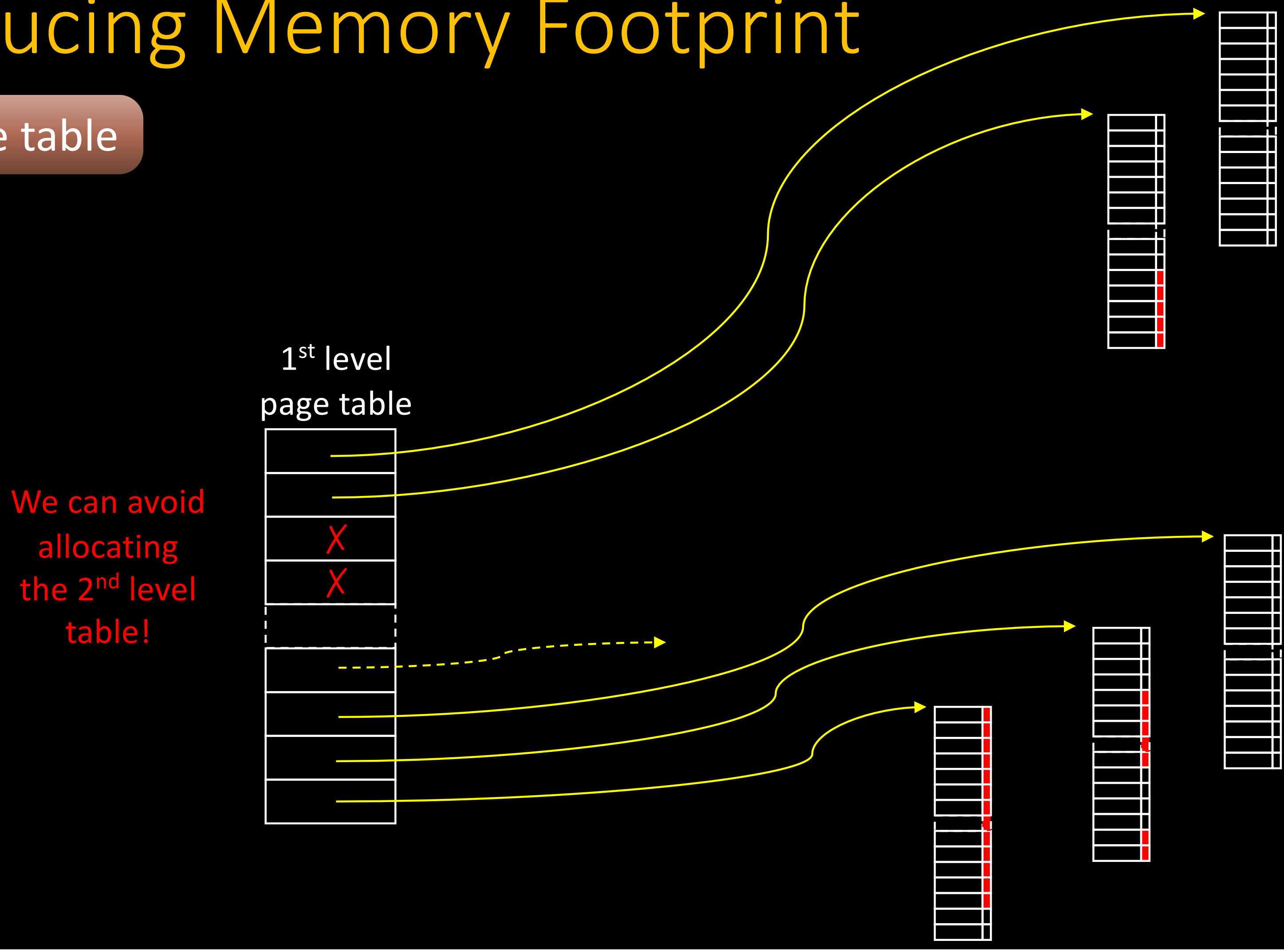
Reducing Memory Footprint

2-level page table



Reducing Memory Footprint

2-level page table



Reducing Memory Footprint

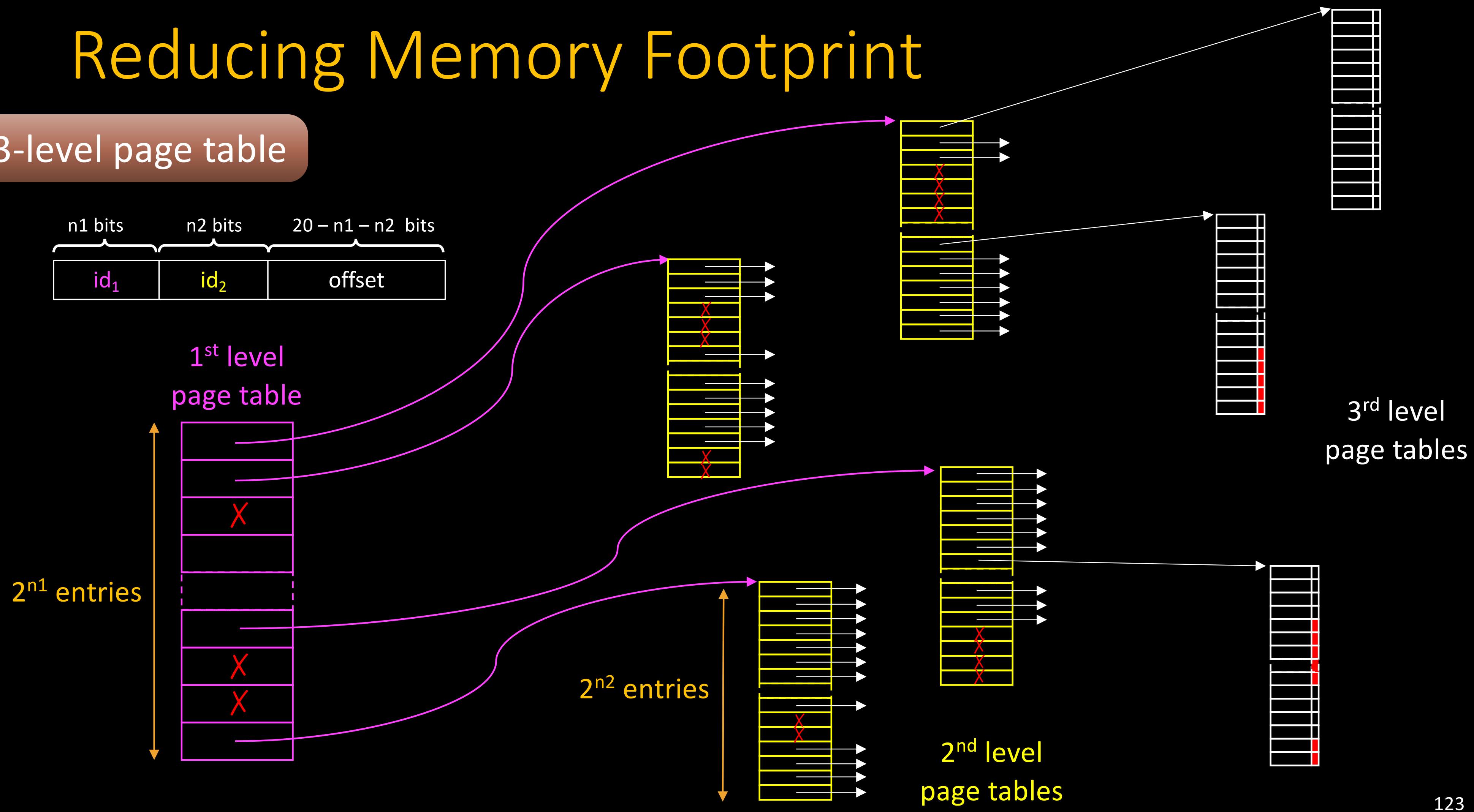
- 2-level page tables can save space!
  - They add a 1st level table…
    …but the gain comes from the non-allocation some 2nd level tables
  - Page tables are built incrementally
    - So unnecessary tables are never allocated
  - It works if invalid memory regions are sufficiently
    - Big
    - Well aligned

Reducing Memory Footprint

- 2-level page tables can save space!
  - They add a 1^{st} level table…
    …but the gain comes from the non-allocation some 2^{nd} level tables
  - It works if invalid memory regions are sufficiently
    - Big
    - Well aligned
- For more flexibility, we can increase the # of levels
  - Current CPUs support 3, 4 or 5 levels

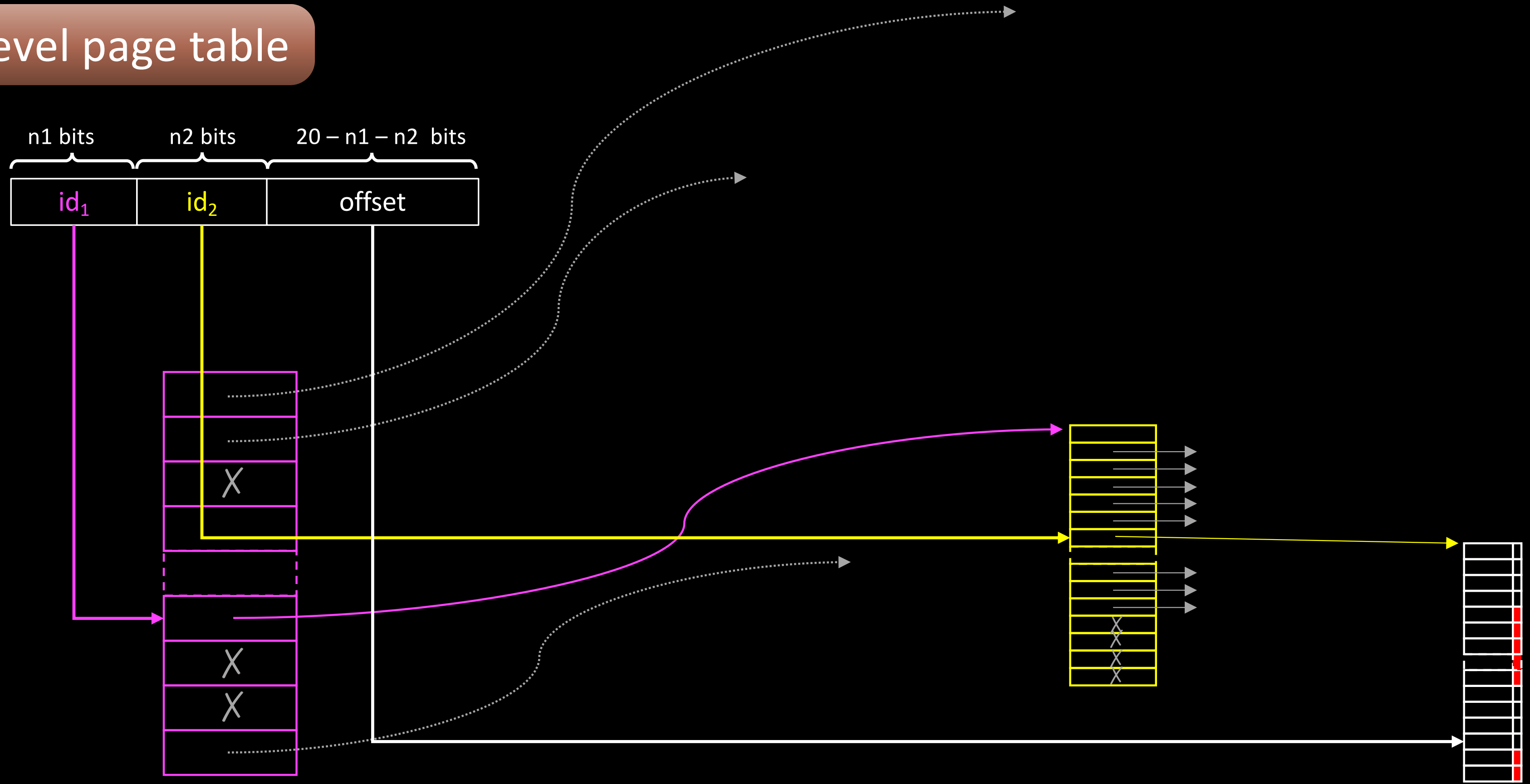
Reducing Memory Footprint

3-level page table



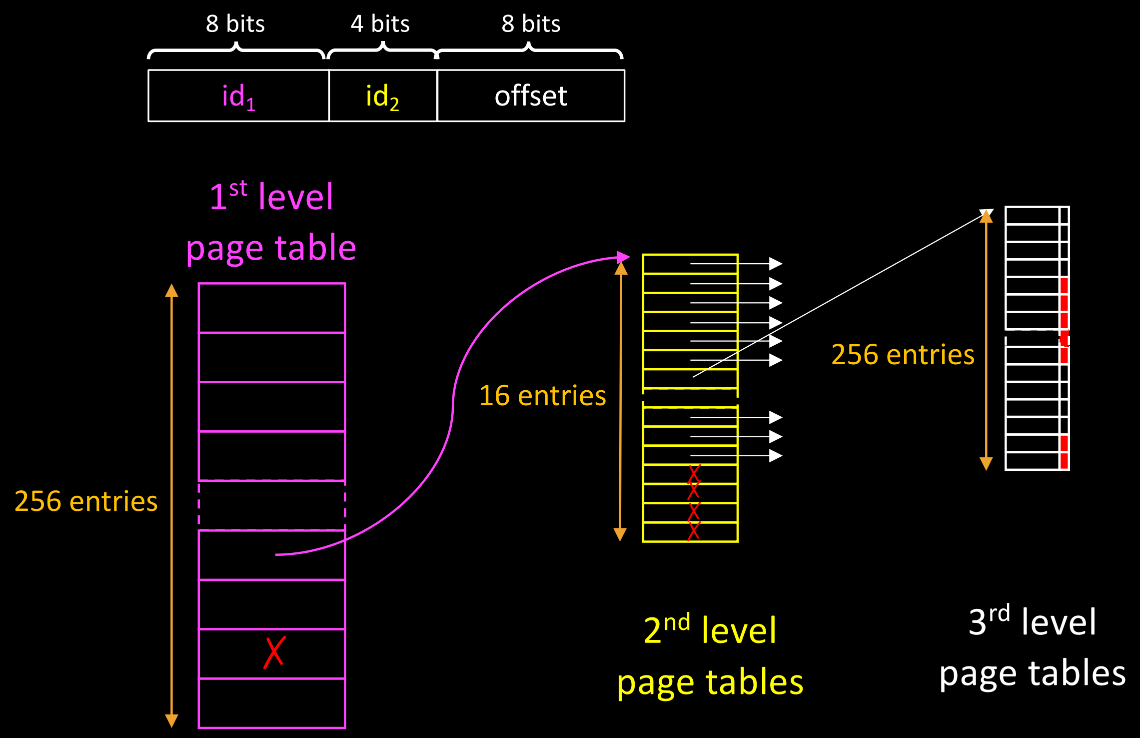
Reducing Memory Footprint

3-level page table



Reducing Memory Footprint: example

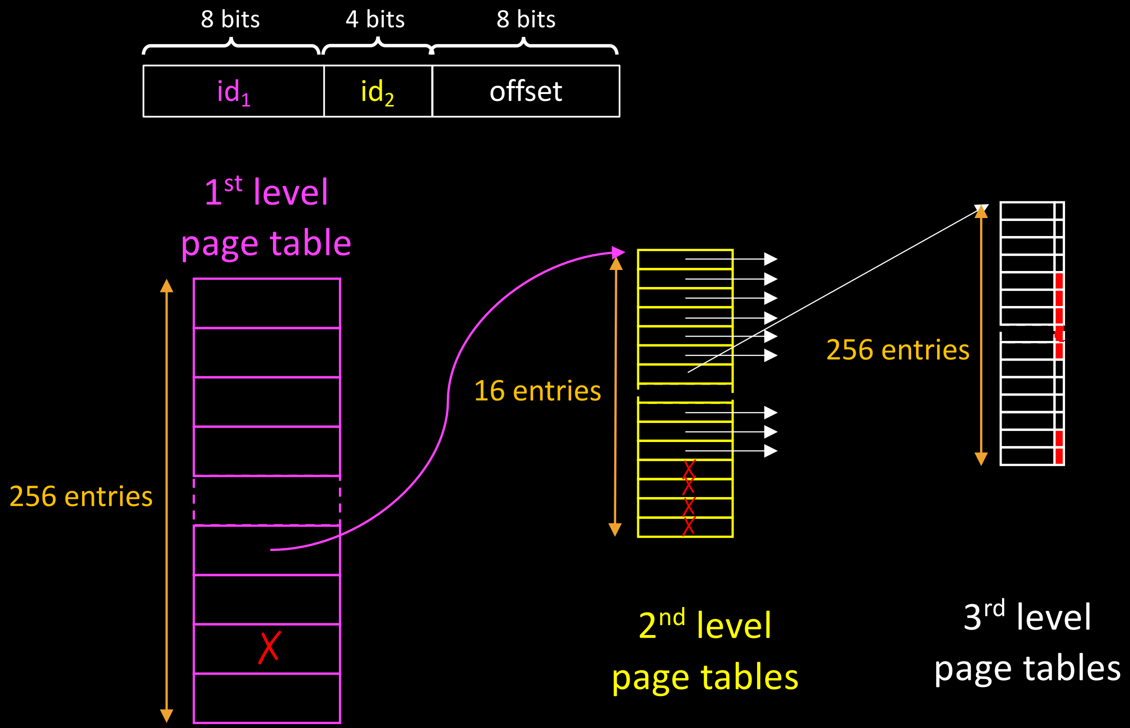
3-level page table



- Overhead compared to monolithic table
  - 1st level:

Reducing Memory Footprint: example

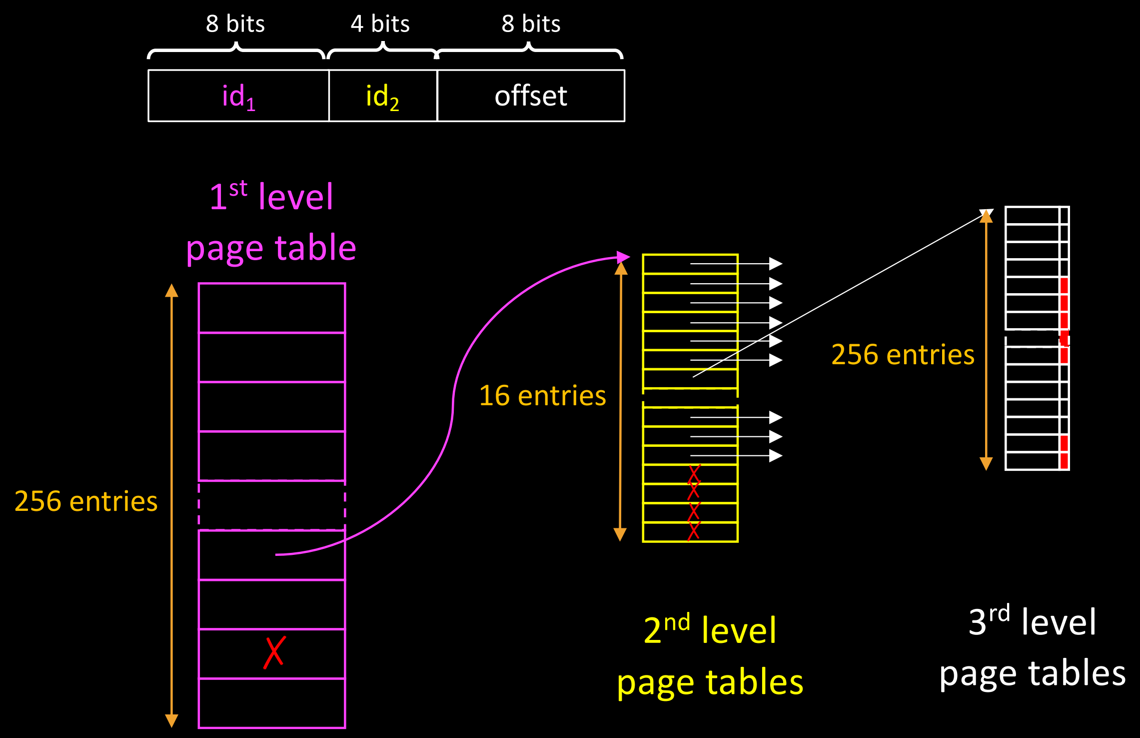
3-level page table



- Overhead compared to monolithic table
  - 1st level: 256 x 4 = 1KB
  - 2nd level:

Reducing Memory Footprint: example

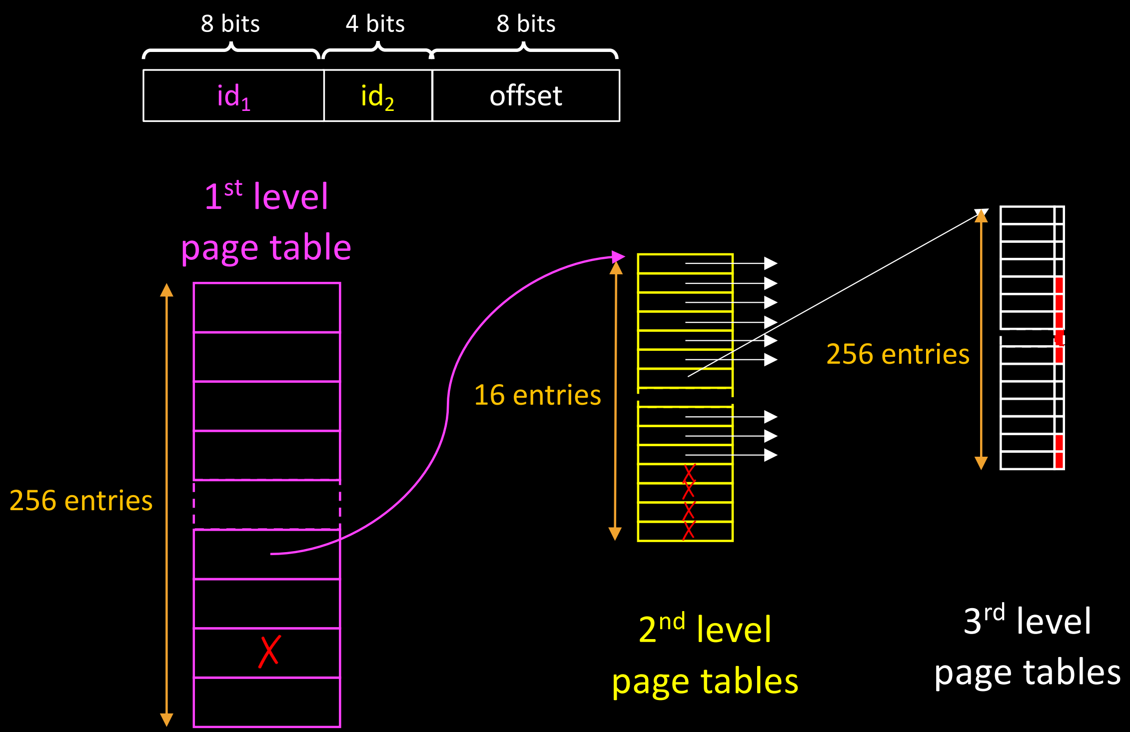
3-level page table



- Overhead compared to monolithic table
  - 1st level: 256 x 4 = 1KB
  - 2nd level: 256 x 16 x 4 = 16 KB

Reducing Memory Footprint: example

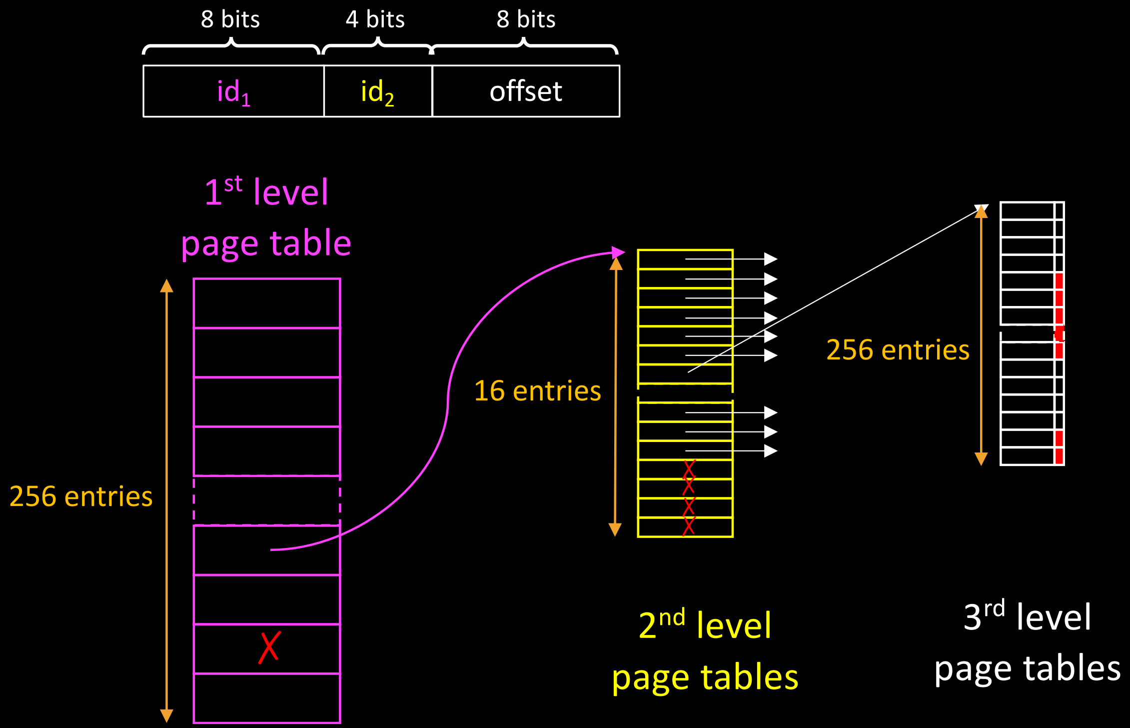
3-level page table



- Overhead compared to monolithic table
  - 1st level: 256 x 4 = 1KB
  - 2nd level: 256 x 16 x 4 = 16 KB
- To save a 3rd level table, we need a well-aligned hole of

Reducing Memory Footprint: example

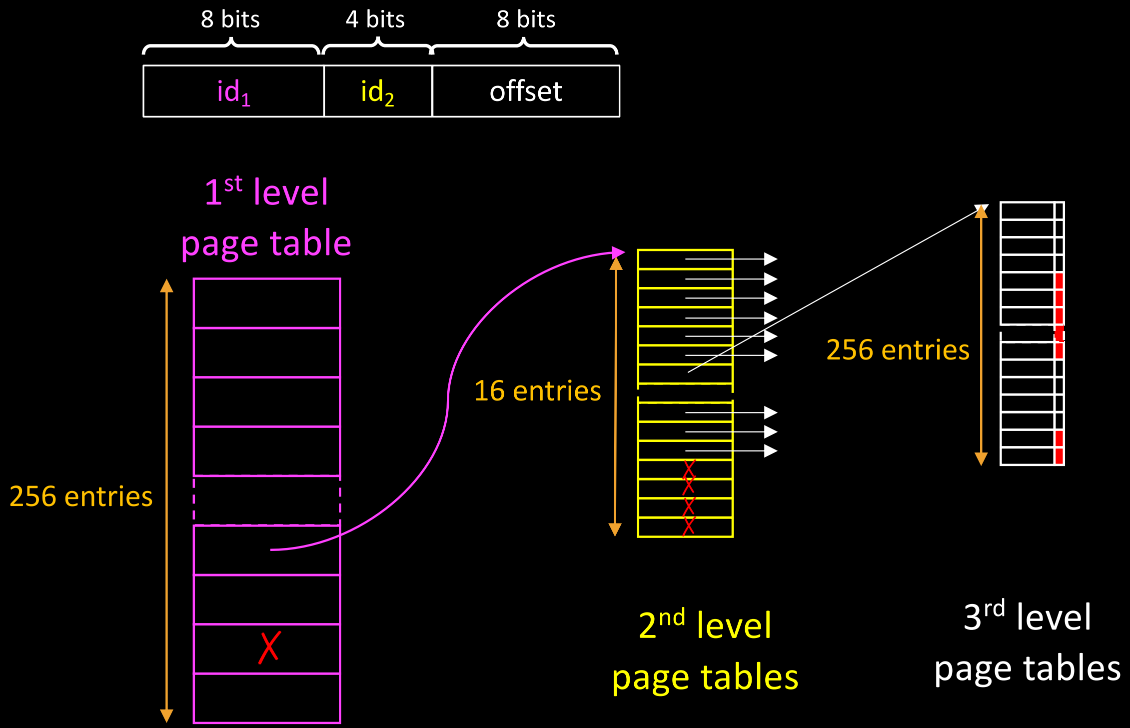
3-level page table



- Overhead compared to monolithic table
  - 1st level: 256 x 4 = 1KB
  - 2nd level: 256 x 16 x 4 = 16 KB
- To save a 3rd level table, we need a well-aligned hole of
  - 256 x 4 KB = 1 MB
  - The gain is:

Reducing Memory Footprint: example

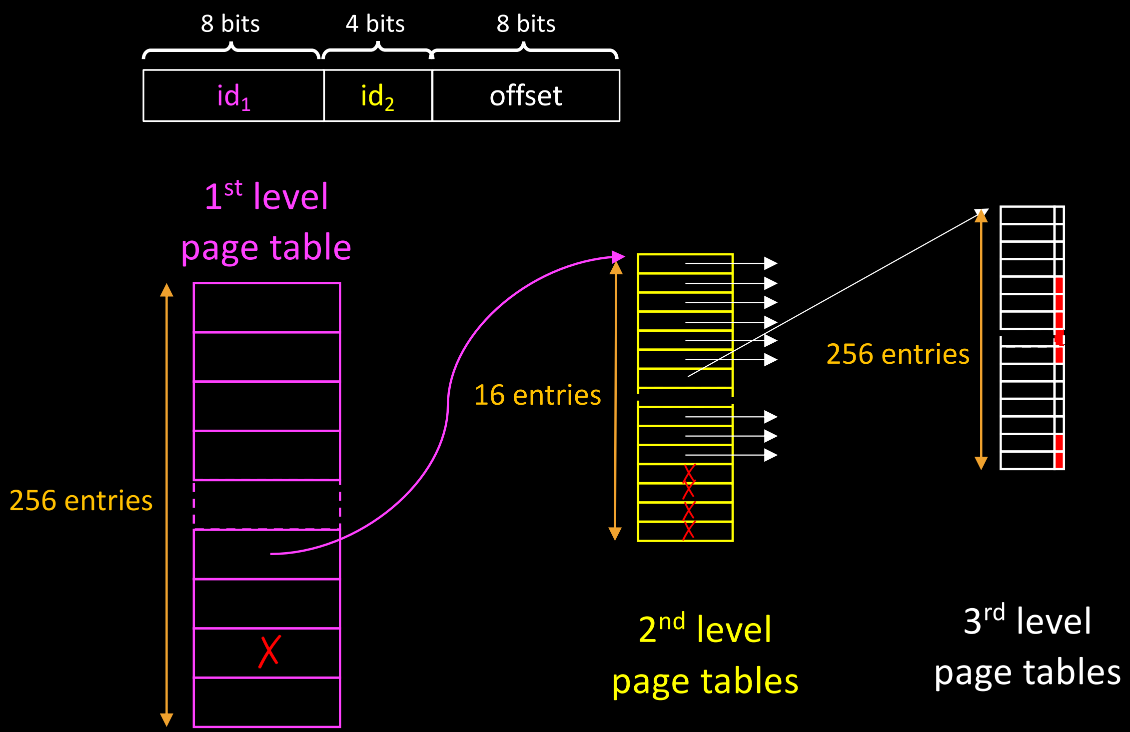
3-level page table



- Overhead compared to monolithic table
  - 1st level: 256 x 4 = 1KB
  - 2nd level: 256 x 16 x 4 = 16 KB
- To save a 3rd level table, we need a well-aligned hole of
  - 256 x 4 KB = 1 MB
  - The gain is: 256 x 4 = 1KB
- To save a 2nd level table, we need a well-aligned hole of

Reducing Memory Footprint: example

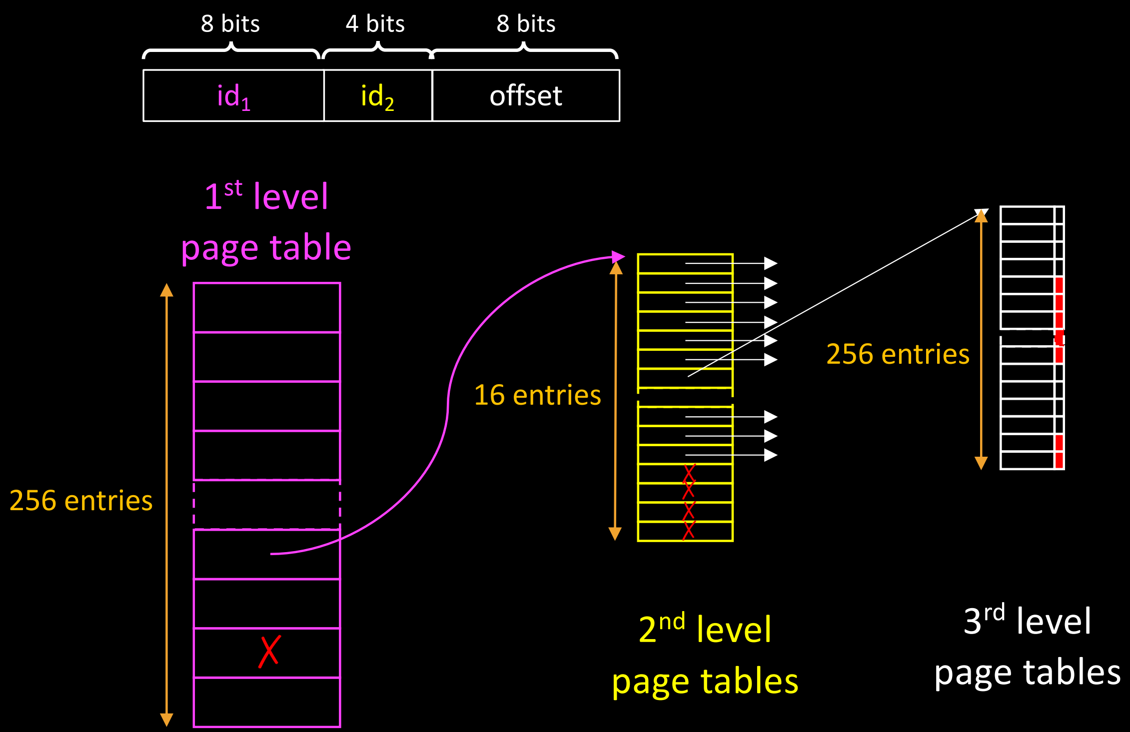
3-level page table



- Overhead compared to monolithic table
  - 1st level: 256 x 4 = 1KB
  - 2nd level: 256 x 16 x 4 = 16 KB
- To save a 3rd level table, we need a well-aligned hole of
  - 256 x 4 KB = 1 MB
  - The gain is: 256 x 4 = 1KB
- To save a 2nd level table, we need a well-aligned hole of
  - 16 x 1MB = 16 MB
  - The gain is:

Reducing Memory Footprint: example

3-level page table



- Overhead compared to monolithic table
  - 1st level: 256 x 4 = 1KB
  - 2nd level: 256 x 16 x 4 = 16 KB
- To save a 3rd level table, we need a well-aligned hole of
  - 256 x 4 KB = 1 MB
  - The gain is: 256 x 4 = 1KB
- To save a 2nd level table, we need a well-aligned hole of
  - 16 x 1MB = 16 MB
  - The gain is: 16 KB + 64 B

Reducing Memory Footprint

- 3-level page tables can really save a lot of space
  - But it adds 3 extra memory accesses on the path to RAM
- The memory appears 4 times slower than expected!
- Improving memory footprint made things go worse

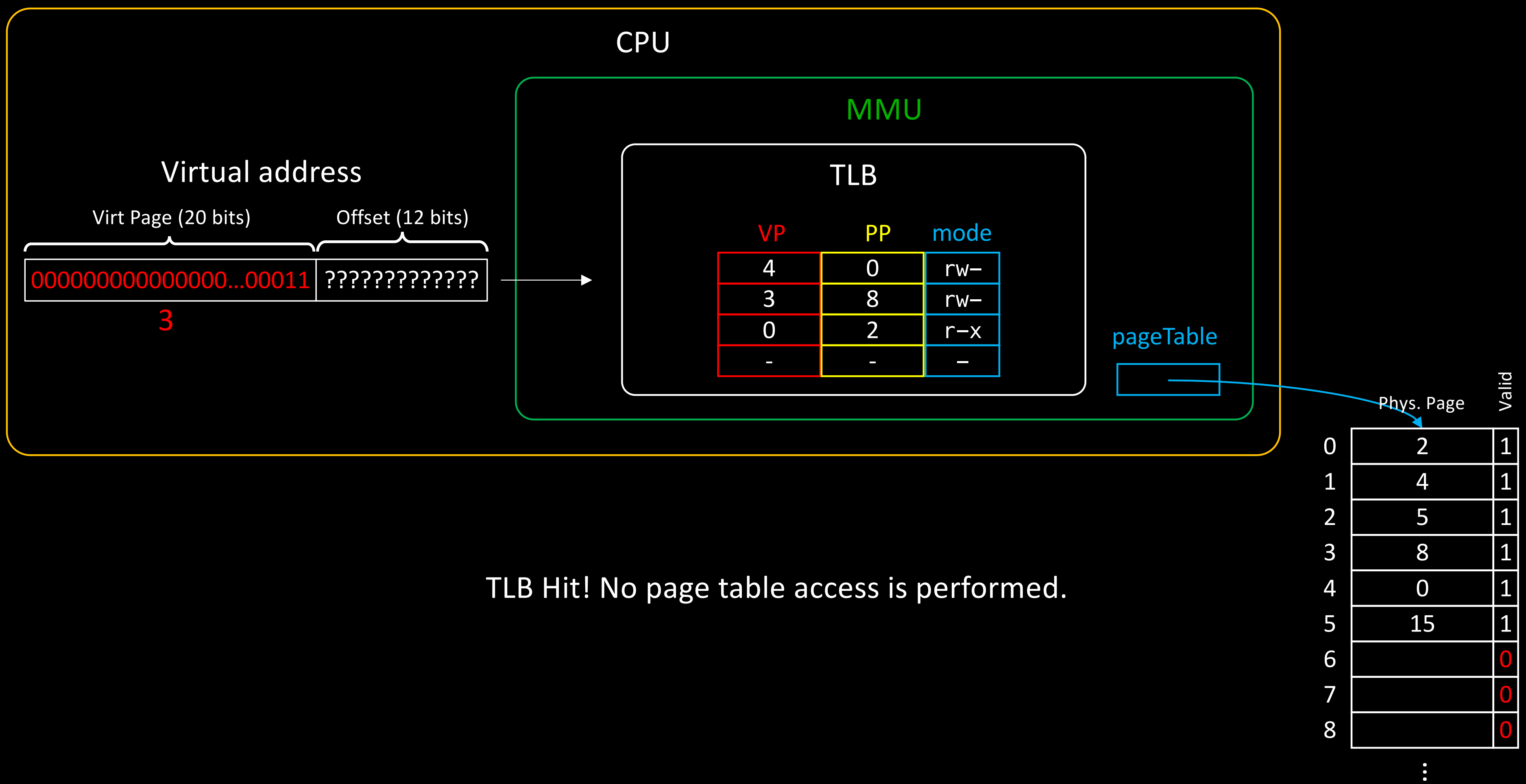
Improving translation performance

- As usual, each time we complain about memory

Improving translation performance

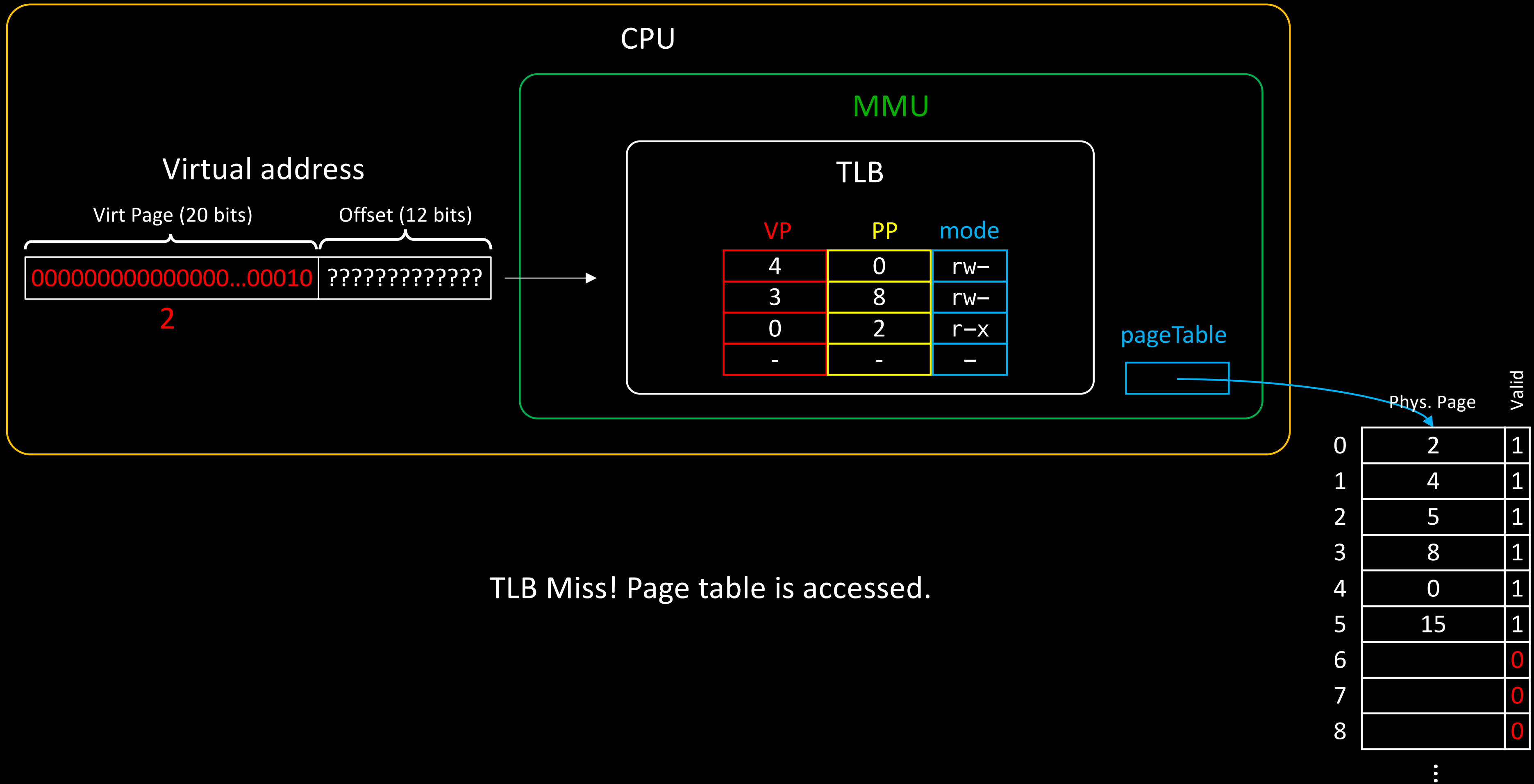
- As usual, each time we complain about memory
  - We introduce a cache…
- Idea: use a cache inside MMU to speed up ”most useful translations”
  - Keep tuples <virtual page #, phys page #, access modes>
- This cache is called *Translation Lookaside Buffer* (TLB)

The Translation Lookaside Buffer (TLB)



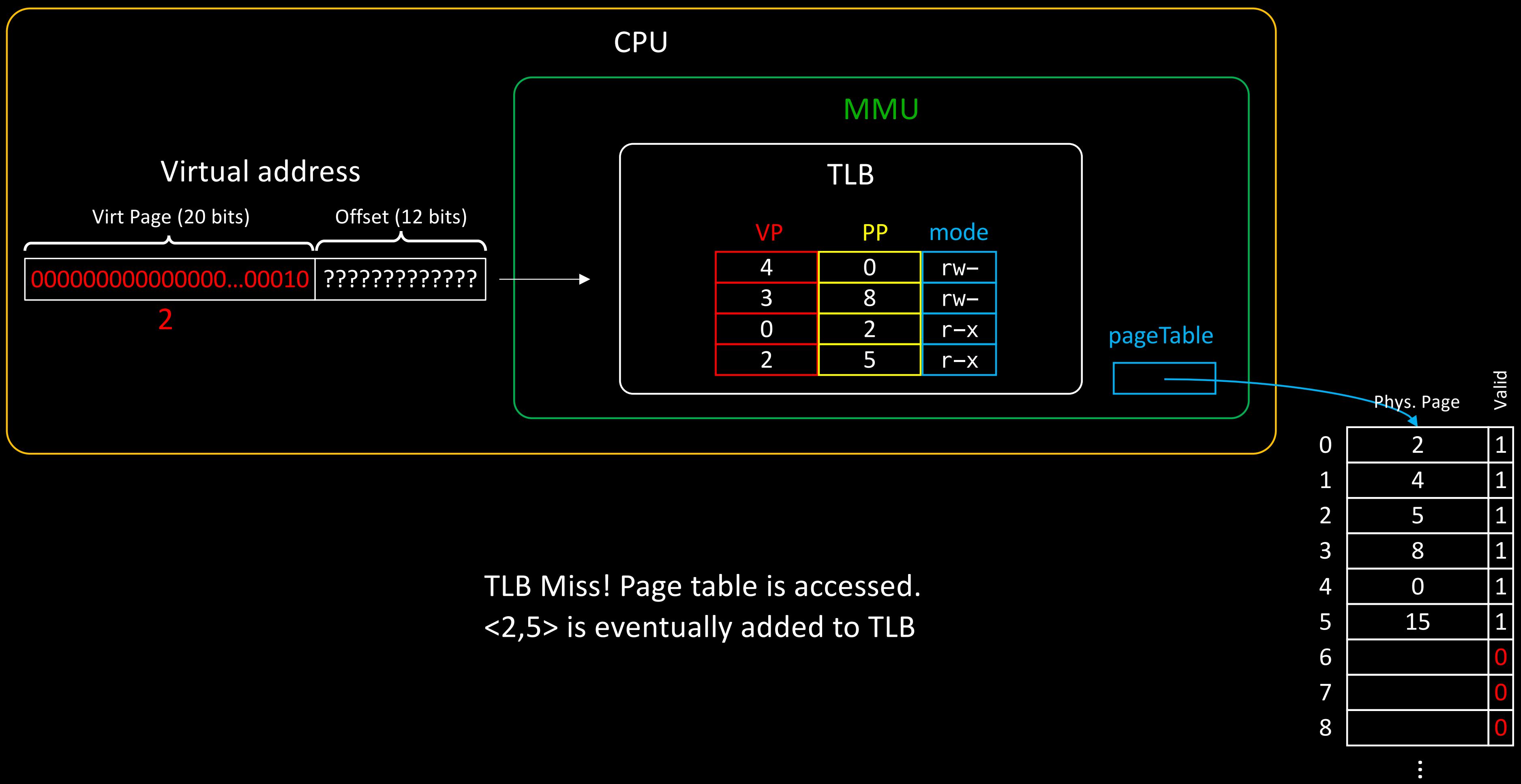
TLB Hit! No page table access is performed.

The Translation Lookaside Buffer (TLB)



TLB Miss! Page table is accessed.

The Translation Lookaside Buffer (TLB)



TLB Miss! Page table is accessed.
<2,5> is eventually added to TLB

The Translation Lookaside Buffer (TLB)

- When TLB is full, which entry gets evicted?

The Translation Lookaside Buffer (TLB)

- When TLB is full, which entry gets evicted?
  - Last Recently Used (LRU) policy

The Translation Lookaside Buffer (TLB)

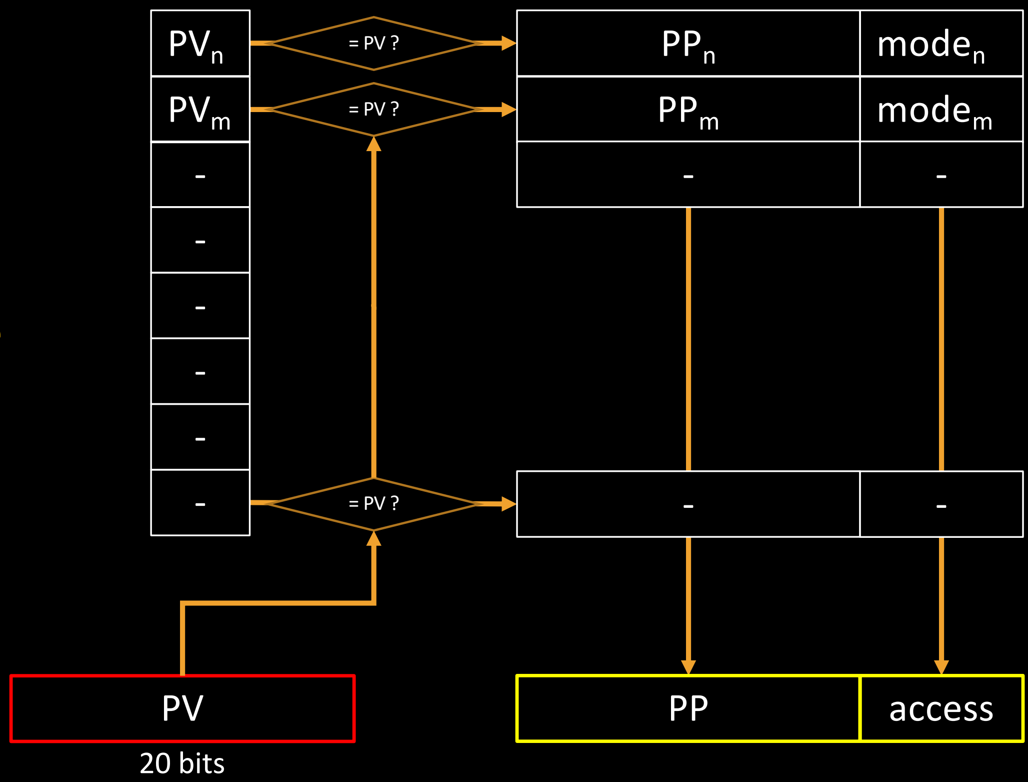
- When TLB is full, which entry gets evicted?
  - Last Recently Used (LRU) policy

- TLB is a fully-associative cache
  - Fast

The Translation Lookaside Buffer (TLB)

- When TLB is full, which entry gets evicted?
  - Last Recently Used (LRU) policy

- TLB is a fully-associative cache
  - Fast
  - Expensive
    - Typically 32 or 64 entries
      - Is that effective?



The Translation Lookaside Buffer (TLB)

- When the OS scheduler switches from P_1 to P_2
  - The TLB contains entries from the page table of P_1
  - We must make sure P_2 won’t use these values
- TLB flush
  - Should we backup its content to RAM?

TLB

VP	PP	mode
4	0	rw–
3	8	rw–
-	-	–
-	-	–

The Translation Lookaside Buffer (TLB)

- Using TAGS to avoid flushes

TLB

tag	VP	PP	mode
P_1	4	0	rw–
P_1	3	8	rw–
P_2	4	21	r–x
-	-	-	–

The Translation Lookaside Buffer (TLB)

- Using TAGS to avoid flushes

- The MMU needs to know who is the current process

TLB

tag	VP	PP	mode
P_1	4	0	rw–
P_1	3	8	rw–
P_2	4	21	r–x
-	-	-	–

The Translation Lookaside Buffer (TLB)

- Using TAGS to avoid flushes

- The MMU needs to know who is the current process
  - @pageTable is usually used instead of PID

TLB

tag	VP	PP	mode
$pgtable_1$	4	0	rw–
$pgtable_1$	3	8	rw–
$pgtable_2$	4	21	r–x
-	-	-	–

The Translation Lookaside Buffer (TLB)

- Modern CPU generally have two separate TLBs
  - Instruction TLB: iTLB
  - Data TLB: dTLB
  - dTLB misses >> iTLB misses
- They also feature several levels
  - L1 private TLB, L2 shared TLB
  - Not all TLB are fully associative
    - Cache associativity will be further explored in other Master Courses

Memory Paging

- The big picture
  - Virtual address spaces and RAM are divided into pages
    - Memory allocation is made on a page basis
  - Page tables, allocated for each process, allow VP to PP conversions
    - To save space, systems use multi-level page tables
    - To speed up conversions, the TLB cache keeps the more recent conversions

Quizz time

https://www.wooclap.com/SEFOREVER

Memory Paging

- So far, we've mostly talked about (user-space) processes

- How is memory accessed on the Kernel side ?
  - In particular, what happens during a system call?
    - Recall that the kernel needs to access user-space memory
      - E.g. `read (fd, buffer, size)`

    - In other words, kernel must access both kernel- and user-space…

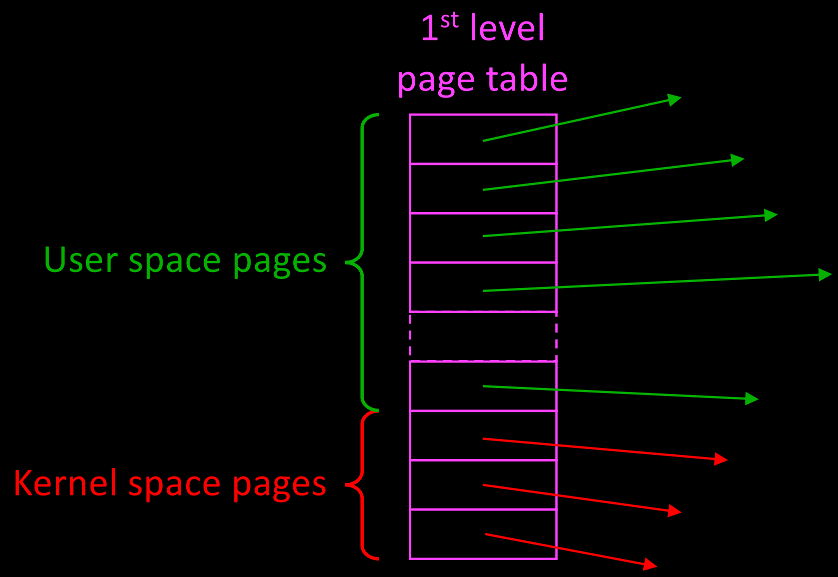
User/kernel Paging

- *Page Table Entries* feature a "user" bit
    - 0 = page only accessible in kernel mode
    - 1 = page accessible in both modes
- The upper part of the table is dedicated to kernel pages
- In some sense, current process' page table *grows* when entering the kernel

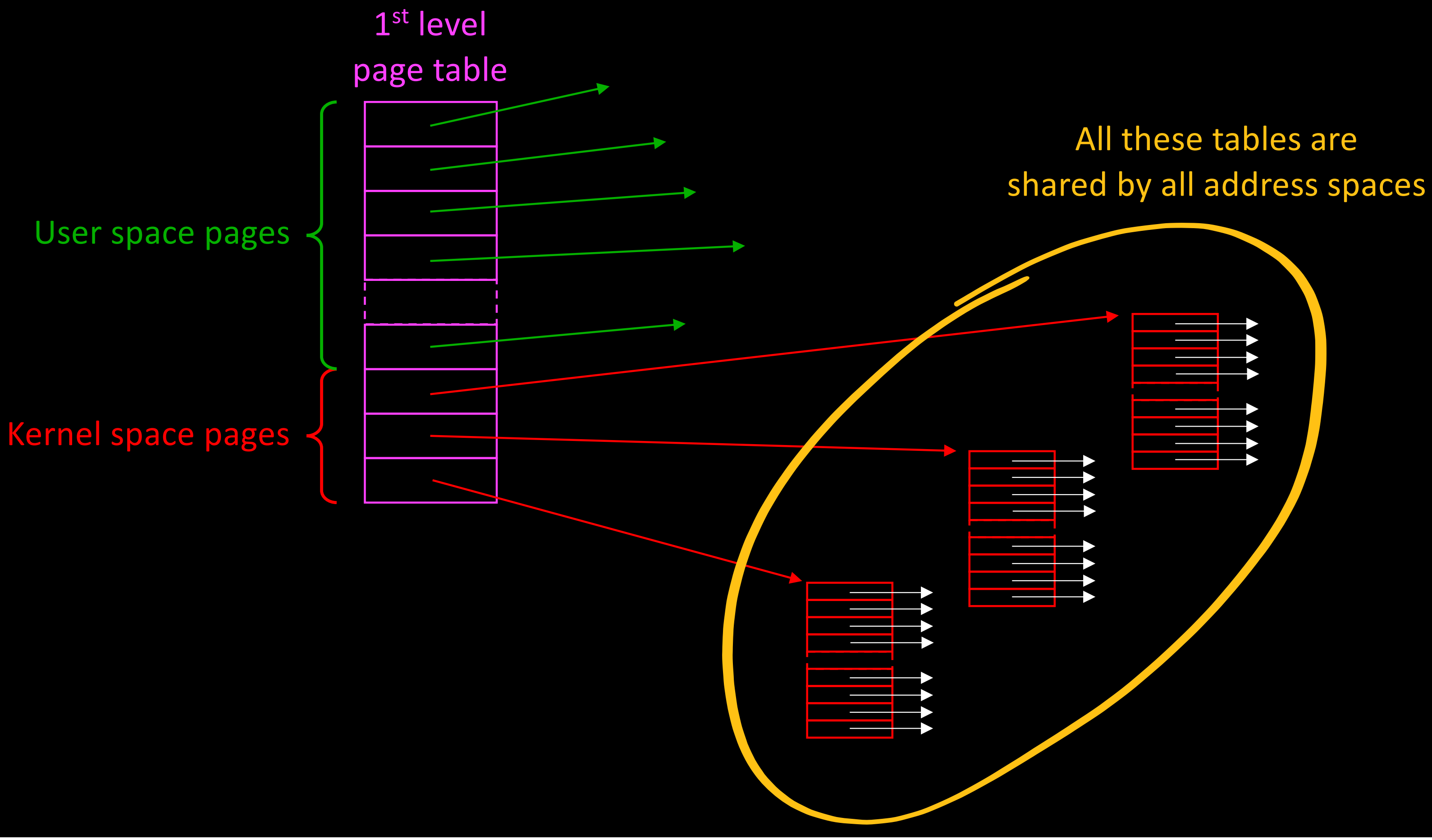
	Phys. Page	Valid	R	W	X	User	
0	2	1	1	0	1	1	
1	4	1	1	0	1	1	
2	5	1	1	0	1	1	
3	8	1	1	1	0	1	
4	0	1	1	1	0	1	
5	15	1	1	1	0	1	User
6		0					
7		0					
8		0					
9	7	1	1	1		1	
10	14	1	1	1		1	
11	6	1	1	1	0	0	
12	3	1	1	1	0	0	Kernel
13	10	1	1	1	0	0	
14		0					

User/kernel Paging

- *Page Table Entries* feature a "user" bit
  - 0 = page only accessible in kernel mode
  - 1 = page accessible in both modes
- The upper part of the table is dedicated to kernel pages
- In some sense, current process' page table *grows* when entering the kernel



User/kernel Paging



User/kernel Paging

- So every process ”sees” the same set of kernel pages (when in kernel mode)

- In Linux 32bits
  - 3 GB virtual memory for user-space
  - 1 GB for kernel usage

- In Linux 64bits
  - The whole physical memory is mapped in kernel virtual space

- Syscalls can directly access virtual addresses passed as parameters
  - E.g. `write (1, “Hello”, 5);`

The Meltdown hardware vulnerability (2017)

- Lipp, Moritz & Schwarz, Michael & Gruss, Daniel & Prescher, Thomas & Haas, Werner & Mangard, Stefan & Kocher, Paul & Genkin, Daniel & Yarom, Yuval & Hamburg, Mike. (2018). Meltdown.
- The Meltdown vulnerability can be exploited to gain access to physical memory

MELTDOWN

The Meltdown hardware vulnerability (2017)

- Lipp, Moritz & Schwarz, Michael & Gruss, Daniel & Prescher, Thomas & Haas, Werner & Mangard, Stefan & Kocher, Paul & Genkin, Daniel & Yarom, Yuval & Hamburg, Mike. (2018). Meltdown.

- The Meltdown vulnerability can be exploited to gain access to physical memory
  - Idea:
    - Exploit (unfortunate) race condition in modern CPU pipelines
    - Use a cache side-channel attack to deduce contents of kernel memory

MELTDOWN

The Meltdown hardware vulnerability (2017)

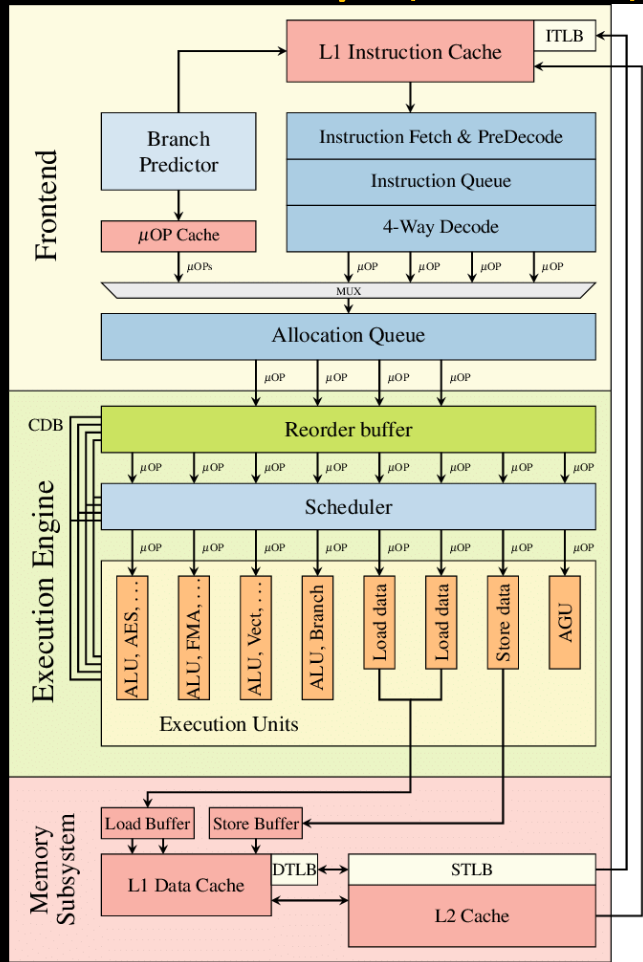
- Lipp, Moritz & Schwarz, Michael & Gruss, Daniel & Prescher, Thomas & Haas, Werner & Mangard, Stefan & Kocher, Paul & Genkin, Daniel & Yarom, Yuval & Hamburg, Mike. (2018). Meltdown.
- The Meltdown vulnerability can be exploited to gain access to physical memory
  - Idea:
    - Exploit (unfortunate) race condition in modern CPU pipelines
    - Use a cache side-channel attack to deduce contents of kernel memory
- Affected hardware
  - Intel x86, IBM POWER, some ARMs

MELTDOWN

The Meltdown hardware vulnerability (2017)

- Modern CPU pipelines
  - Out-of-order and speculative execution
    - To avoid pipeline stalls, instructions can be
      - Reordered
        - False dependencies removal
          - E.g. register renaming

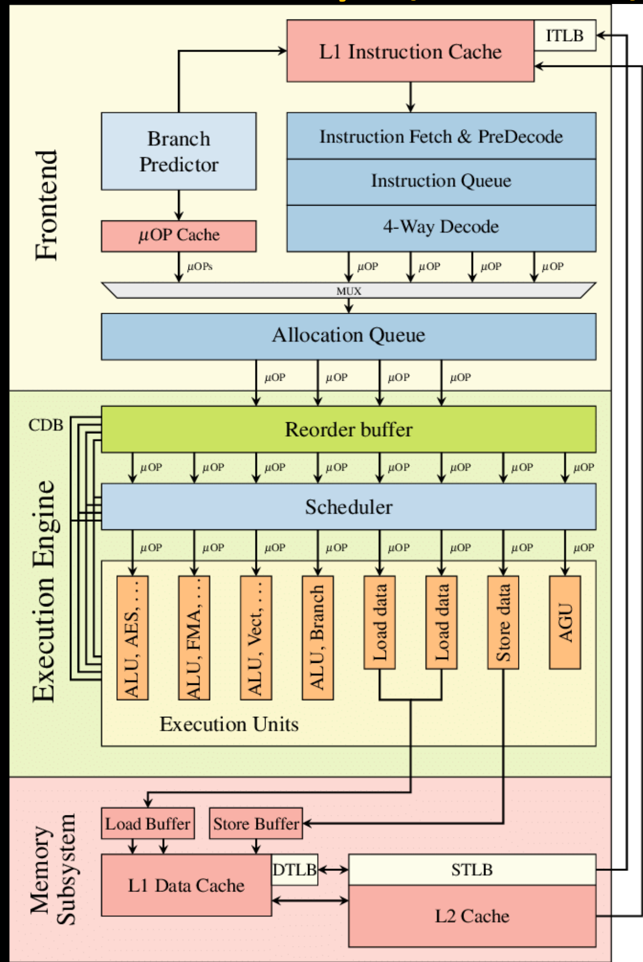
```
movq _var_a, %rax
addq %rax, %rbx
movq _var_b, %rax
mulq %rax, %rcx
```



The Meltdown hardware vulnerability (2017)

- Modern CPU pipelines
  - Out-of-order and speculative execution
    - To avoid pipeline stalls, instructions can be
      - Reordered
        - False dependencies removal
          - E.g. register renaming

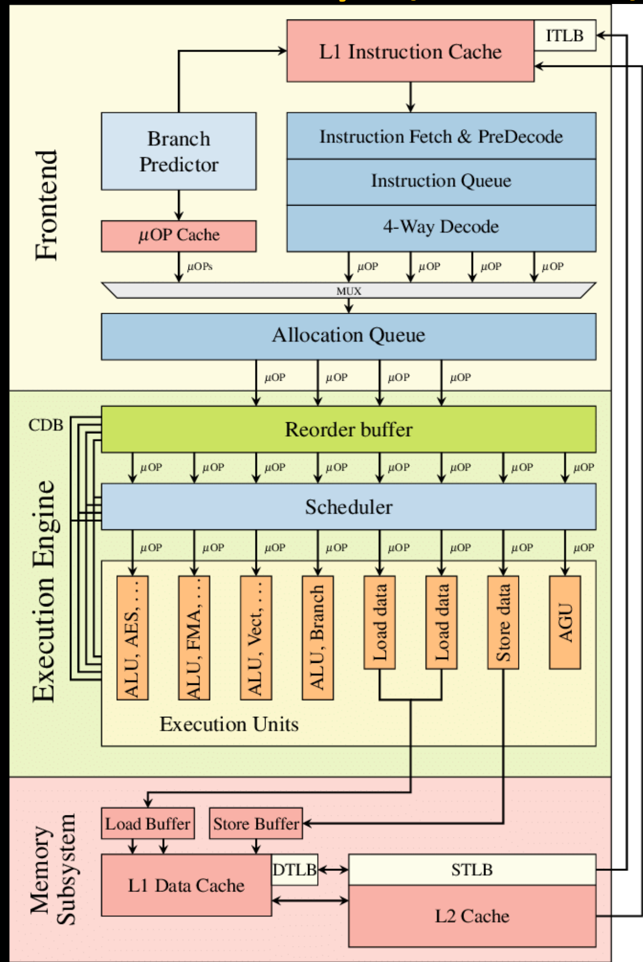
```
movq _var_a, %rax
addq %rax, %rbx
movq _var_b, %rax'
mulq %rax', %rcx
```



The Meltdown hardware vulnerability (2017)

- Modern CPU pipelines
  - Out-of-order and speculative execution
    - To avoid pipeline stalls, instructions can be
      - Executed although we're not 100% certain they should be
        - Speculative execution
          - E.g. Branch prediction

```
if (x > 0)
  y = f();
else
  y = g();
```

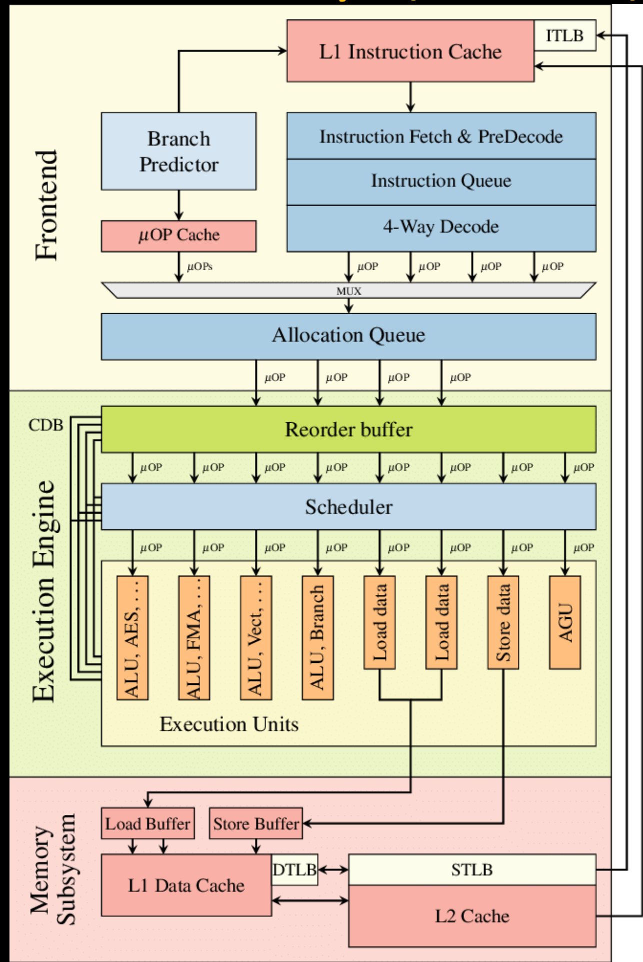


The Meltdown hardware vulnerability (2017)

- Modern CPU pipelines
  - Out-of-order and speculative execution
    - To avoid pipeline stalls, instructions can be
      - Executed although we’re not 100% certain they should be
        - Speculative execution
          - E.g. Branch prediction

```
if (x > 0)
  y = f();
else
  y = g();
```

        - Instructions should be NOT BE COMMITTED in case of misprediction
          - No side effect should be observed outside CPU

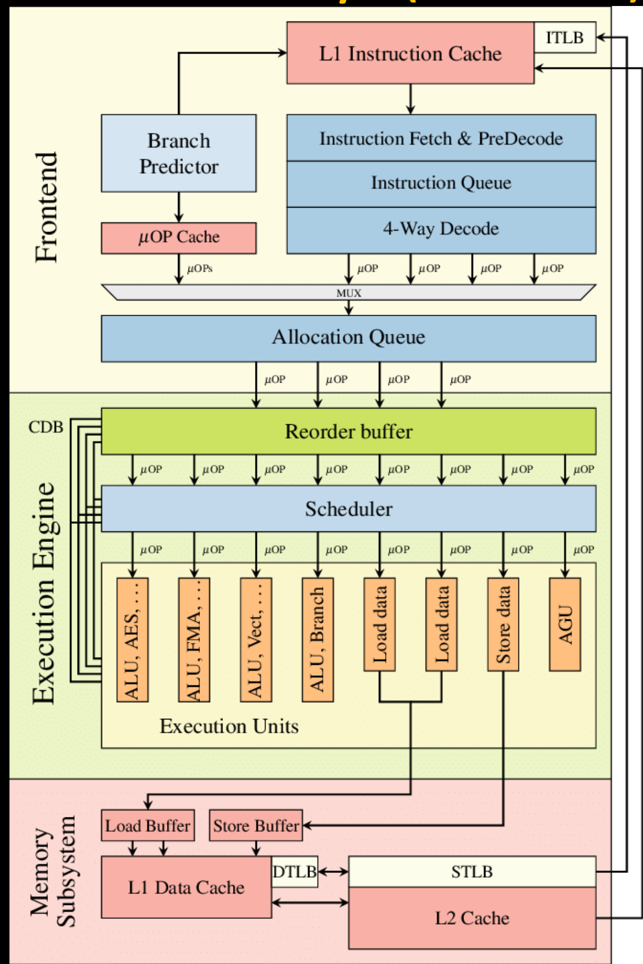


The Meltdown hardware vulnerability (2017)

- Modern CPU pipelines
  - Out-of-order and speculative execution
    - To avoid pipeline stalls, instructions can be
      - Executed although we're not 100% certain they should be
        - Speculative execution
          - E.g. Branch prediction

```
if (x > 0)
  y = f();
else
  y = g();
```

        - Instructions should be NOT BE COMMITTED in case of misprediction
          - No side effect should be observed outside CPU
            - None? Well, we will see…



The Meltdown hardware vulnerability (2017)

- Exceptions and speculative execution
  - The first instructions raises an exception
    - Trap into the kernel

```
char array [N * 4096];
int data = …;
char c;

*((int *)NULL) = 12;
c = array [data * 4096];
```

The Meltdown hardware vulnerability (2017)

- Exceptions and speculative execution
  - The first instructions raises an exception
    - Trap into the kernel
  - However, the second instruction gets executed before the exception actually traps…
    - The reorder buffer is cleared to cancel the instruction
      - c is not modified
    - But there is a side-effect…

```
char array [N * 4096];
int data = …;
char c;

*((int *)NULL) = 12;
c = array [data * 4096];
```

The Meltdown hardware vulnerability (2017)

- Side effect
  - Memory content at
    - array [data * 4096]

    has been accessed and kept into cache(s)

- Cache timing attack

```
char array [N * 4096];
int data = …;
char c;

*((int *)NULL) = 12;
c = array [data * 4096];
```

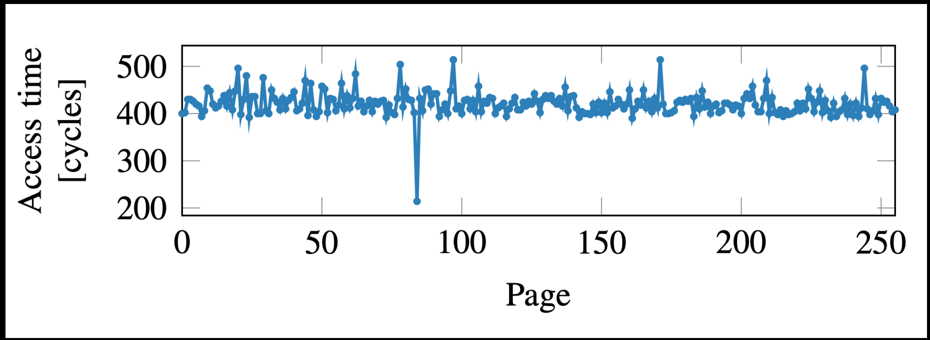
The Meltdown hardware vulnerability (2017)

- Side effect
  - Memory content at
    - array [data * 4096]

    has been accessed and kept into cache(s)

- Cache timing attack
  - If we now measure the access time to every [i * 4096] element
    - We guess the value of data!
      - 84 in this example

- So what? Big deal?



The Meltdown hardware vulnerability (2017)

- Our goal is to read a normally inaccessible byte from kernel memory
  - Byte address is in rcx
  - mov al, byte [rcx] will raise an exception

```
; rcx = kernel address
; rbx = array base address
retry:
  mov al, byte [rcx]
  shl rax, 0xc
  jz retry
  mov rbx, qword [rbx + rax]
```

The Meltdown hardware vulnerability (2017)

- Our goal is to read a normally inaccessible byte from kernel memory
  - Byte address is in rcx
  - mov al, byte [rcx] will raise an exception
    - HOWEVER, the exception will be scheduled in parallel with the transient instructions
      - Race condition
      - Yeah, that's incredible…

```
; rcx = kernel address
; rbx = array base address
retry:
  mov al, byte [rcx]
  shl rax, 0xc
  jz retry
  mov rbx, qword [rbx + rax]
```

The Meltdown hardware vulnerability (2017)

- Our goal is to read a normally inaccessible byte from kernel memory
  - Byte address is in rcx
  - mov al, byte [rcx] will raise an exception
    - HOWEVER, the exception will be scheduled in parallel with the transient instructions
      - Race condition
      - Yeah, that's incredible…
    - So mov rbx, qword [rbx + rax] will be executed, then cancelled…
      - But the cache will be loaded

```
; rcx = kernel address
; rbx = array base address
retry:
  mov al, byte [rcx]
  shl rax, 0xc
  jz retry
  mov rbx, qword [rbx + rax]
```

The Meltdown hardware vulnerability (2017)

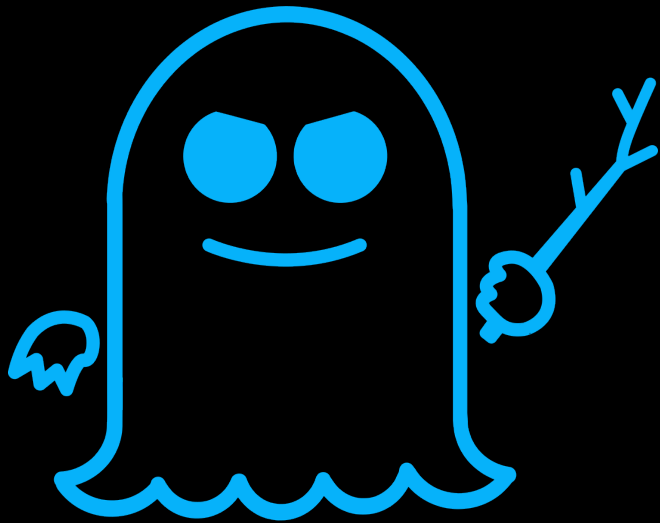
- Repeating this process for all kernel-space address, we can read the whole physical memory!
  - Direct-physical map started at address 0xffff 8800 0000 0000 on Linux systems ☺
    - Without Kernel Address Space Layout Randomization (KASLR)
- Authors report a 503 KB/s rate

```
; rcx = kernel address
; rbx = array base address
retry:
  mov al, byte [rcx]
  shl rax, 0xc
  jz retry
  mov rbx, qword [rbx + rax]
```

The Spectre hardware vulnerability (2017)

- Kocher, Paul; Genkin, Daniel; Gruss, Daniel; Haas, Werner; Hamburg, Mike; Lipp, Moritz; Mangard, Stefan; Prescher, Thomas; Schwarz, Michael; Yarom, Yuval (2018). Spectre Attacks: Exploiting Speculative Execution

- The Spectre vulnerability exploit branch prediction + speculative execution
  - Nice blog post from Emile Josso (former CSI Master Student)
    - https://blog.amossys.fr/spectre-v1-userland.html



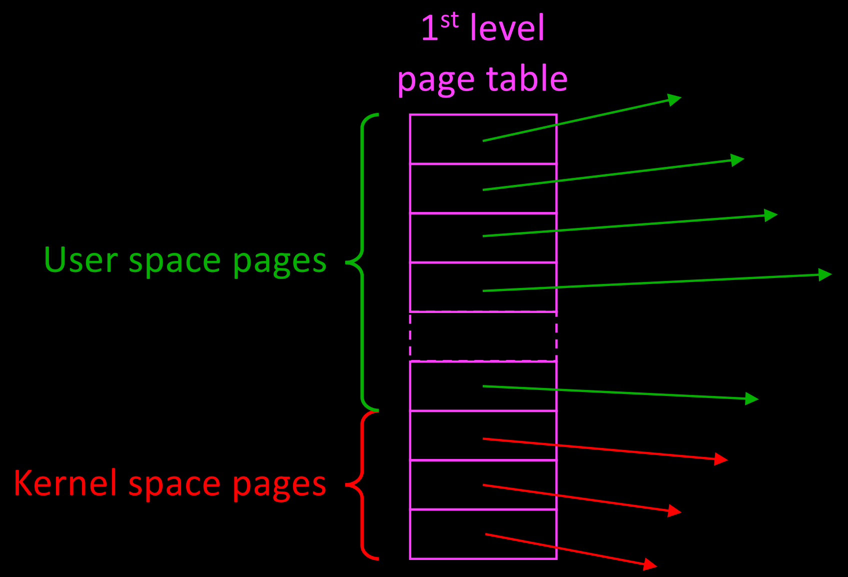
SPECTRE

Is there a Fix to Meltdown?

- Solving the problem on the hardware is tough
  - Without all these aggressive optimizations, CPUs would be *much* slower!

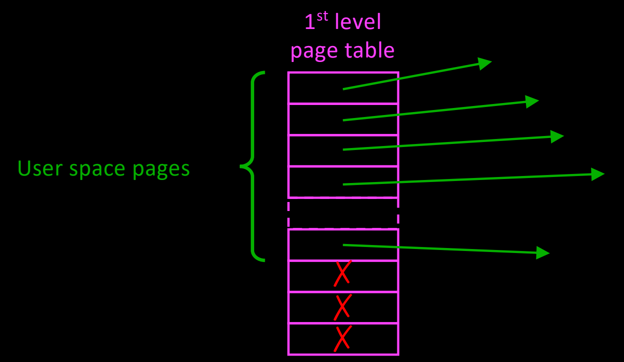
Is there a Fix to Meltdown?

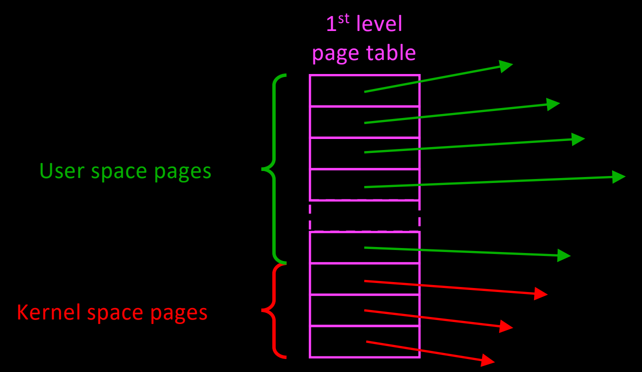
- Solving the problem on the hardware is tough
  - Without all these aggressive optimizations, CPUs would be *much* slower!
- What can we do on the software side?
  - The problem comes from the fact that kernel space is part of the page table…



Is there a Fix to Meltdown?

- Kernel Page Table Isolation (KPTI)
  - Formerly KAISER
    - Kernel Address Isolation to have Side-channels Efficiently Removed
- Idea: two pages tables per process (!)
  - The full one is used inside kernel
  - The second one only covers user space addresses

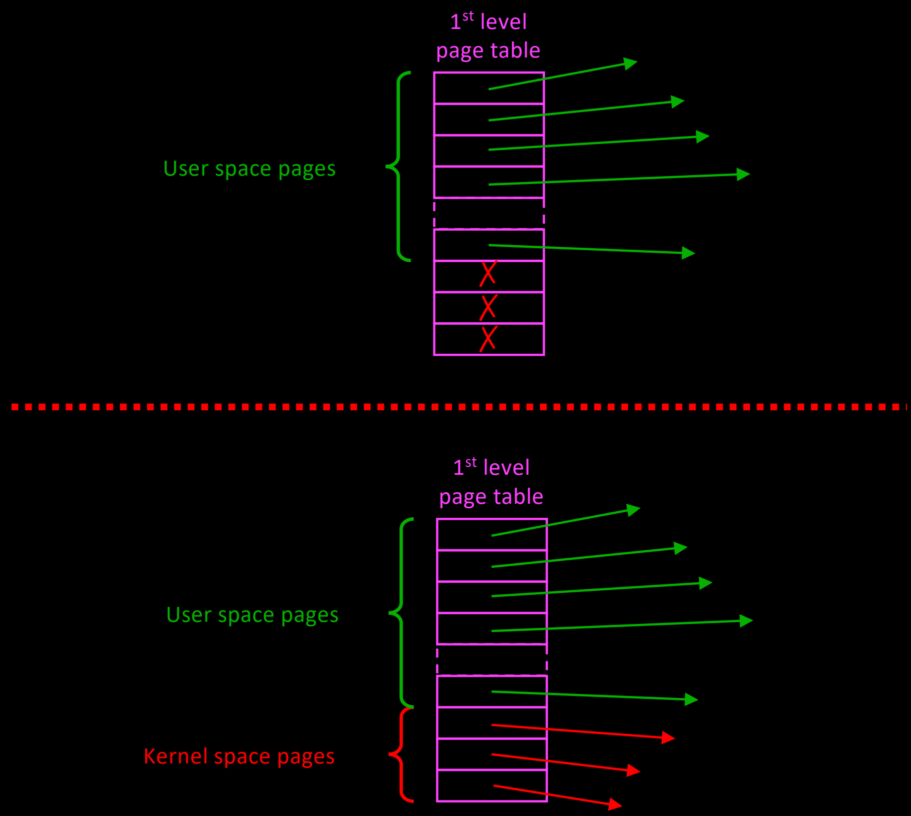




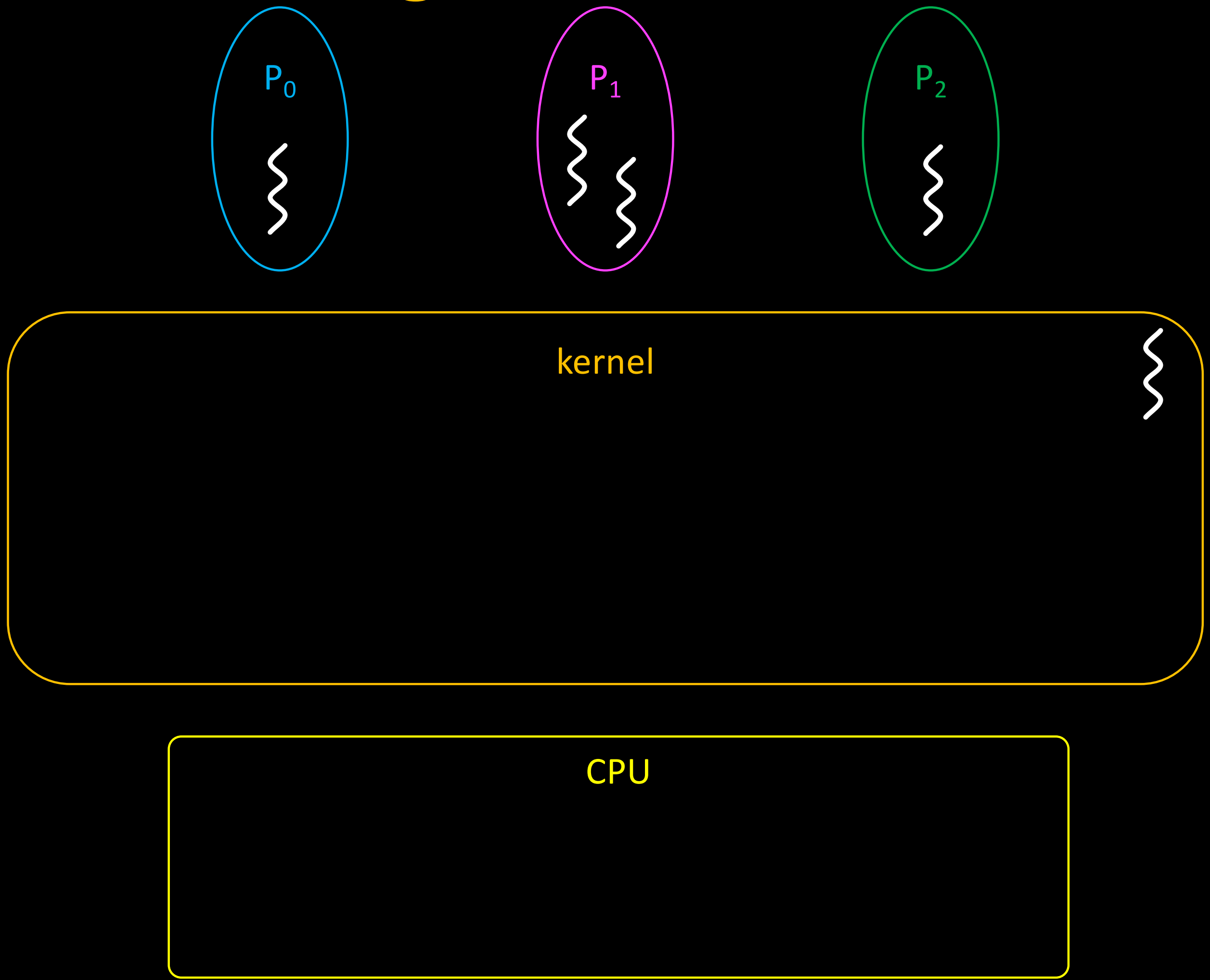
Is there a Fix to Meltdown?

- Kernel Page Table Isolation (KPTI)
  - Formerly KAISER
    - Kernel Address Isolation to have Side-channels Efficiently Removed

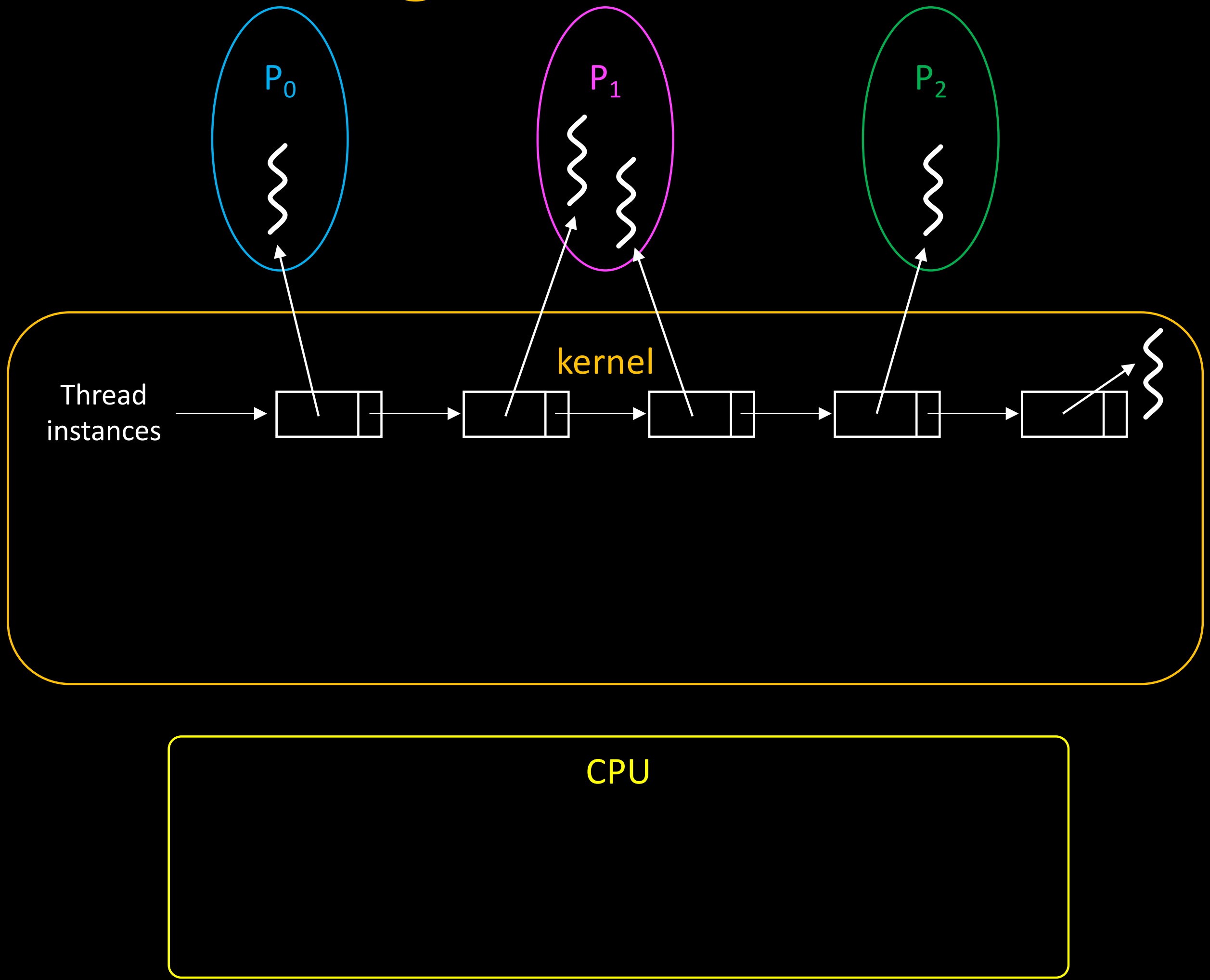
- Overhead
  - 5% to 25% slowdown reported on Haswell/Skylake architectures
    - Ouch!



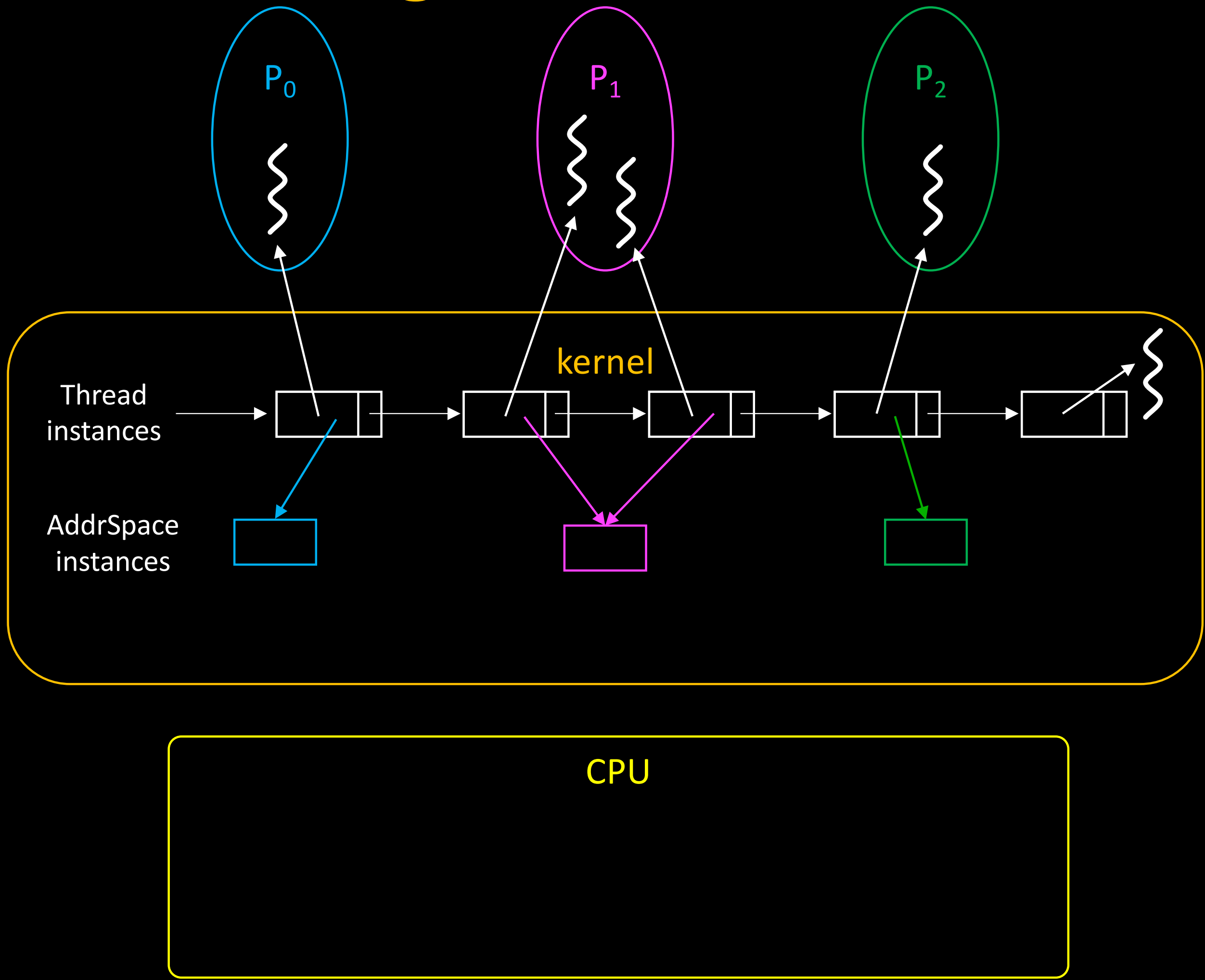
Pagination: the Big Picture



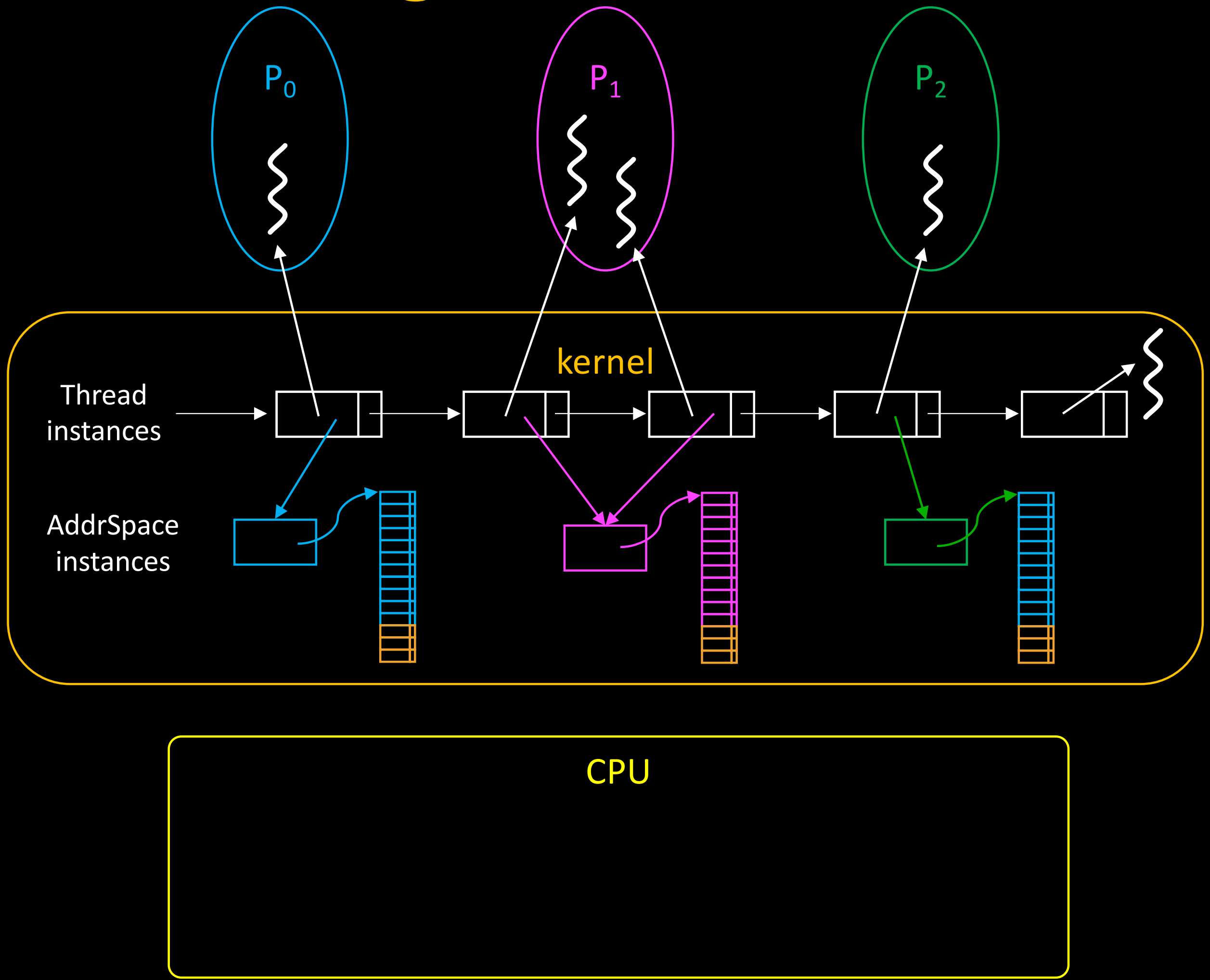
Pagination: the Big Picture



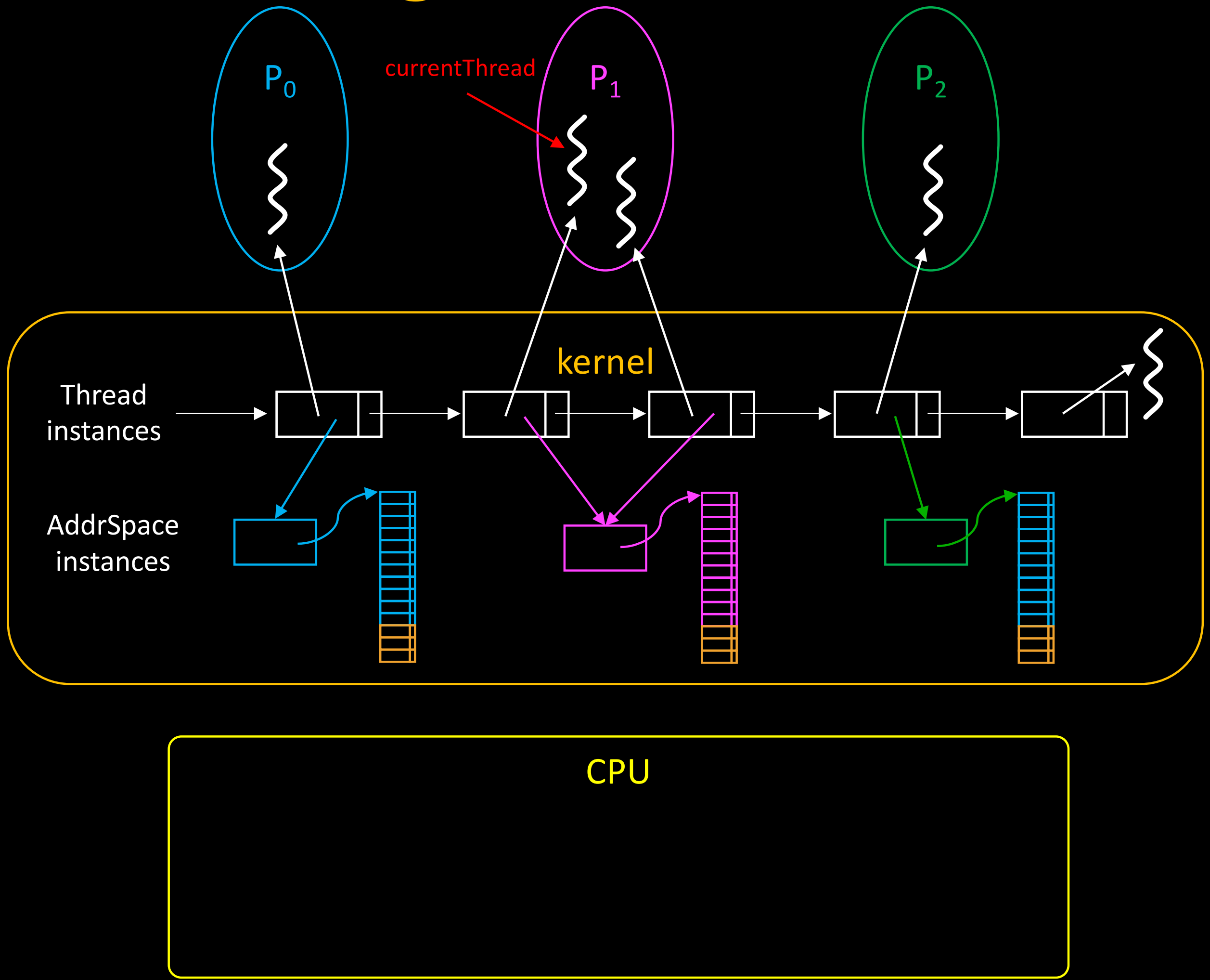
Pagination: the Big Picture



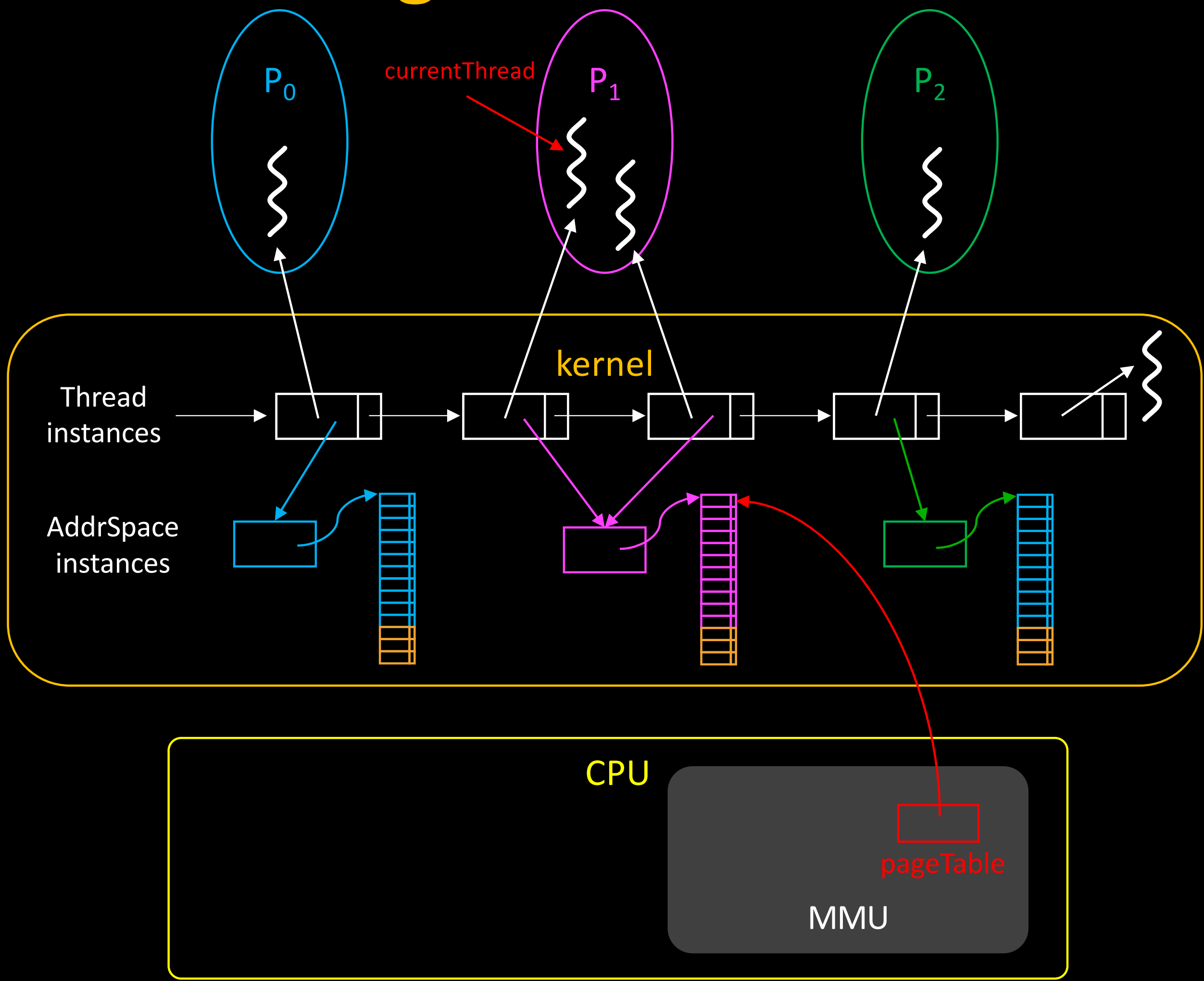
Pagination: the Big Picture



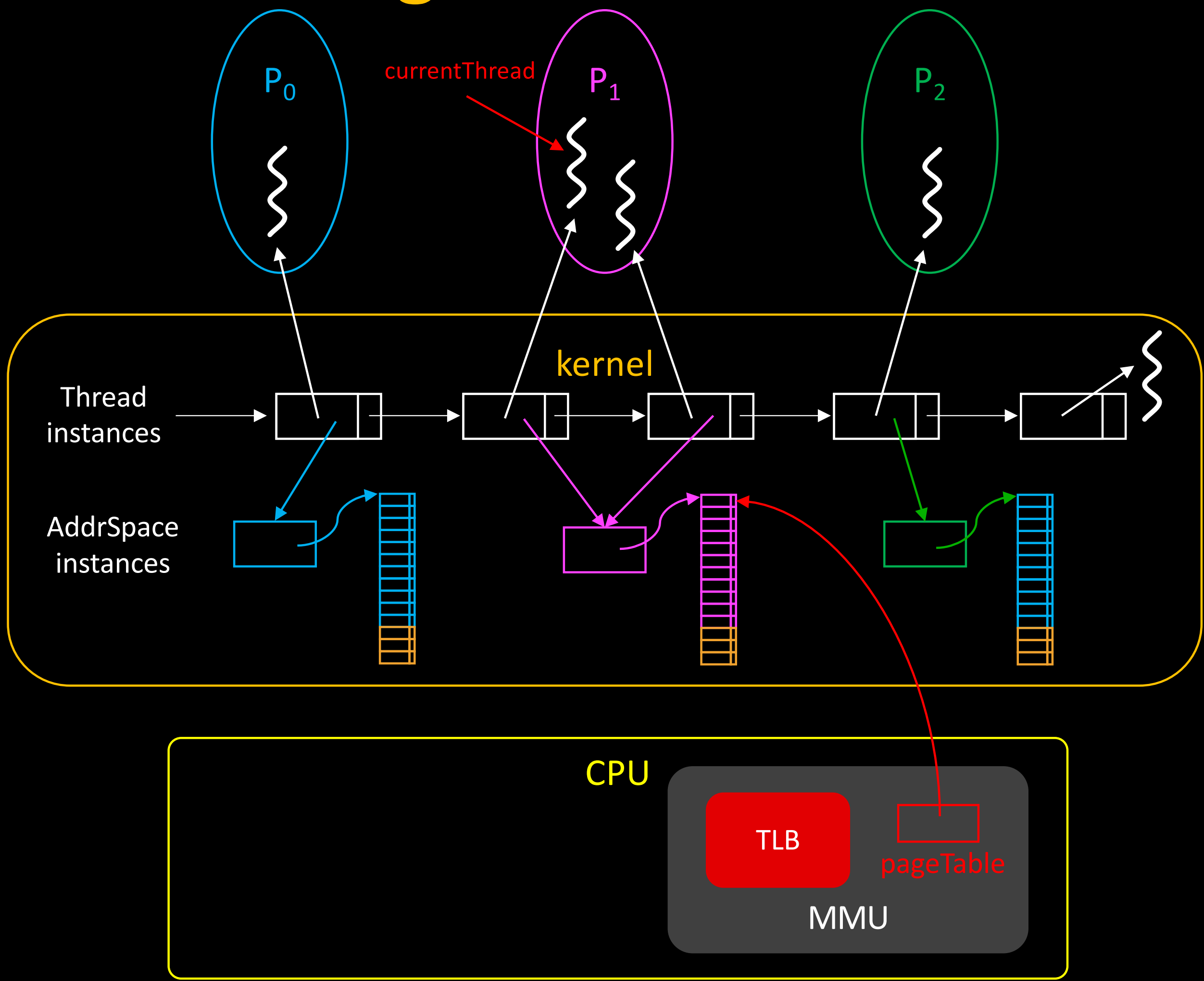
Pagination: the Big Picture



Pagination: the Big Picture



Pagination: the Big Picture



Optimizing Pagination

- To save memory and to speed up overall process performance, OS kernels use several aggressive optimizations

- Based on laziness
  - Processes ask services
  - Kernel says: “Sure!”
    - But does not process it immediately
    - Later on, WHEN ABSOLUTELY NEEDED, it will be done

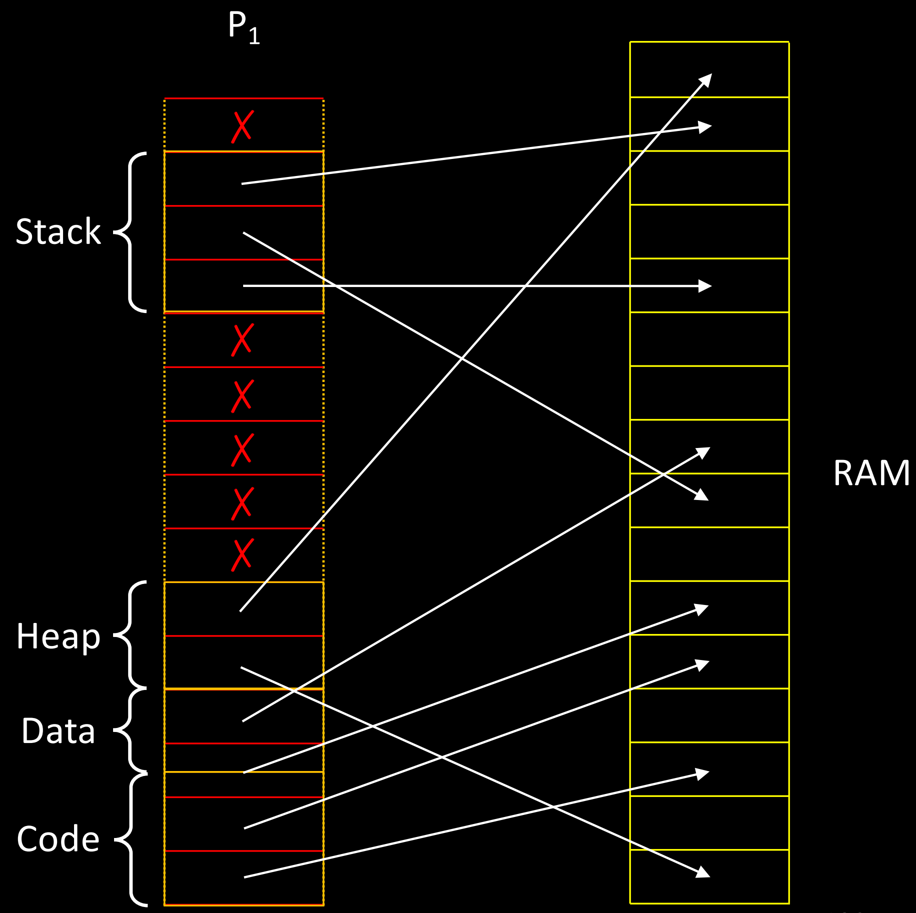
- We’ll explore two of such optimizations
  - First-touch memory allocation (aka Lazy allocation)
  - Copy-on-Write

First-touch memory allocation

- Idea
  - Upon process creation, only a subset of its address space is allocated
    - A few virtual pages are allocated right from the start
    - The allocation of most pages is postponed
  - Pages will be allocated *“on demand”*
    - i.e. when the CPU will access them for the first time
- Benefits
  - If a page is never accessed, it will never be allocated
    - Better memory utilization!

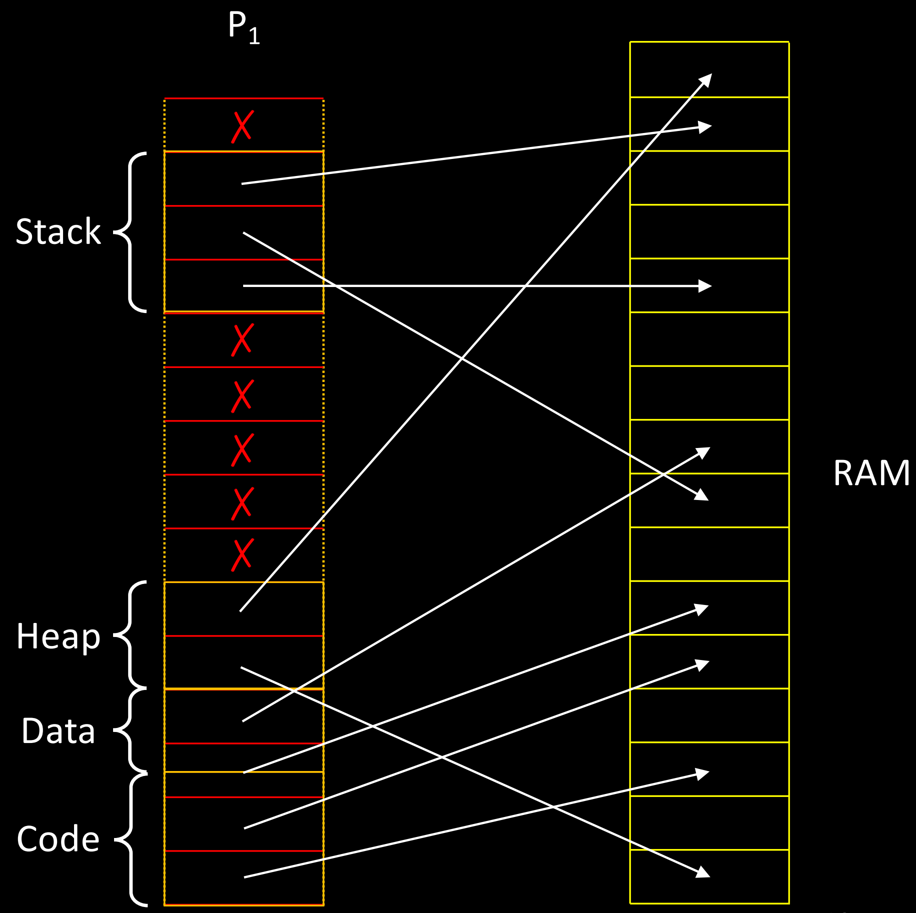
First-touch memory allocation

- Seriously? Are there some processes which do not use their entire code, data, heap or stack area?



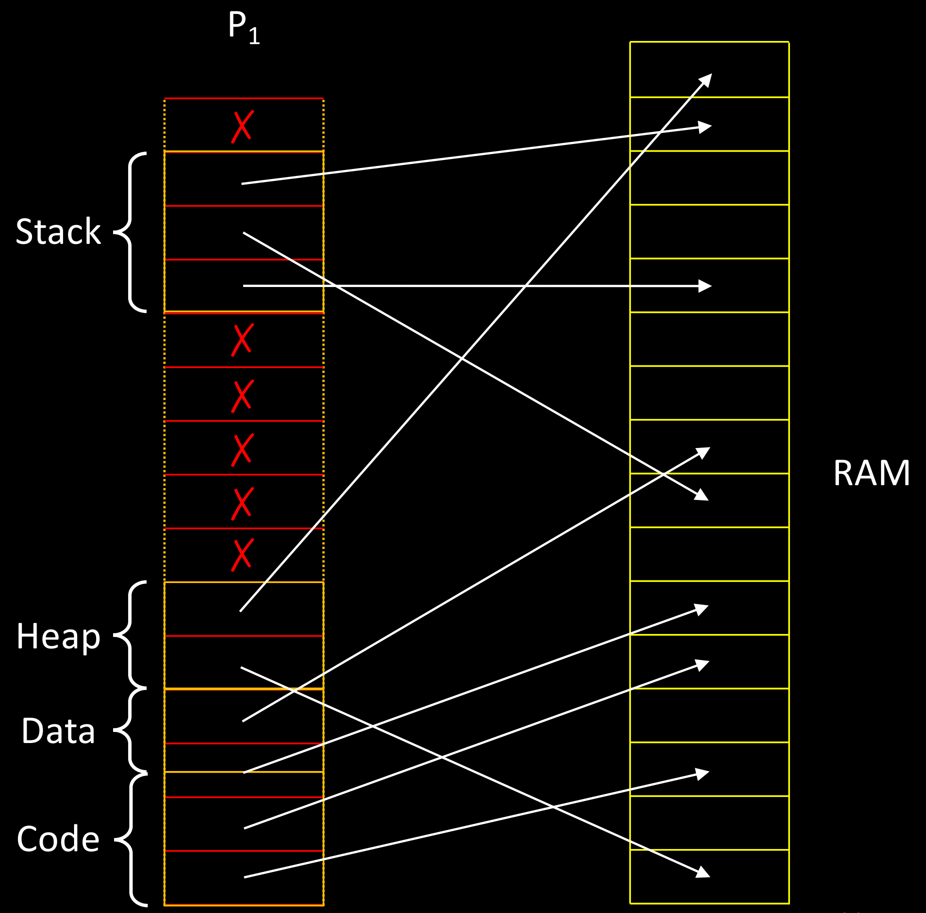
First-touch memory allocation

- Seriously? Are there some processes which do not use their entire code, data, heap or stack area?
  - Almost every process!



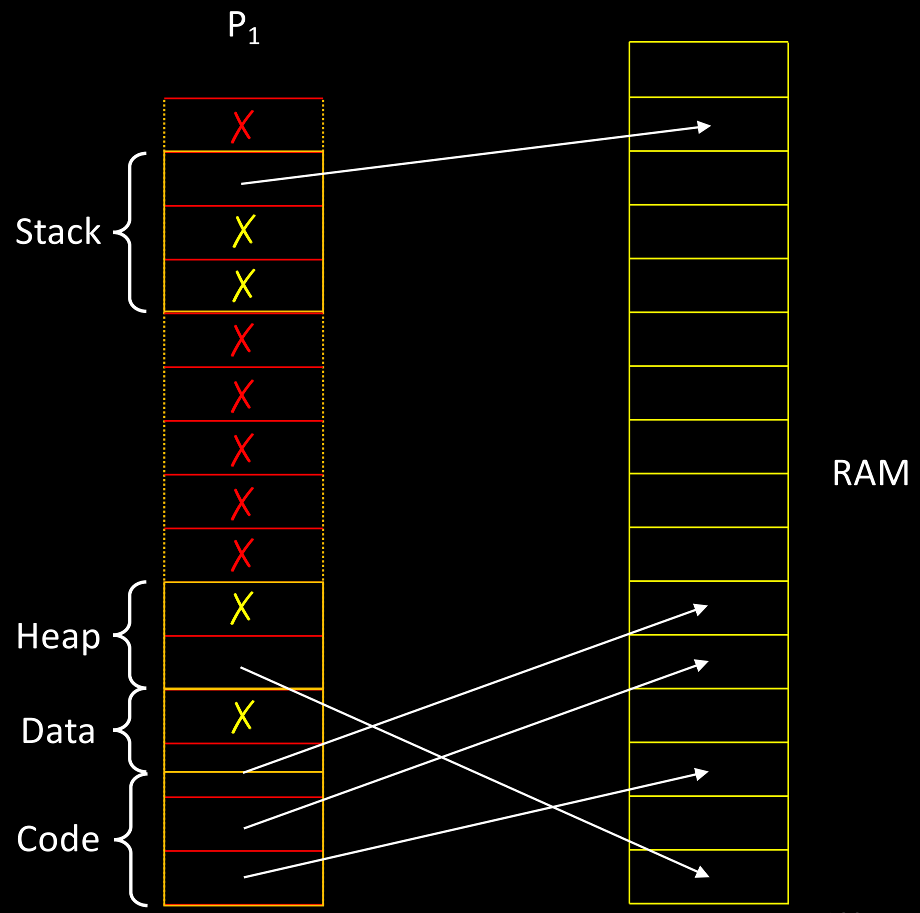
First-touch memory allocation

- Seriously? Are there some processes which do not use their entire code, data, heap or stack area?
  - Almost every process!
    - Maximum Stack Size is 8MB by default
      - Processes only need a fraction of it
    - Code is plenty of functions which will never be called
    - Some static arrays won't be entirely accessed



First-touch memory allocation

- Seriously? Are there some processes which do not use their entire code, data, heap or stack area?
  - Almost every process!
    - Some pages will be allocated on demand (X)
  - Let's see if we can observe address space growth on Linux…



First-touch memory allocation

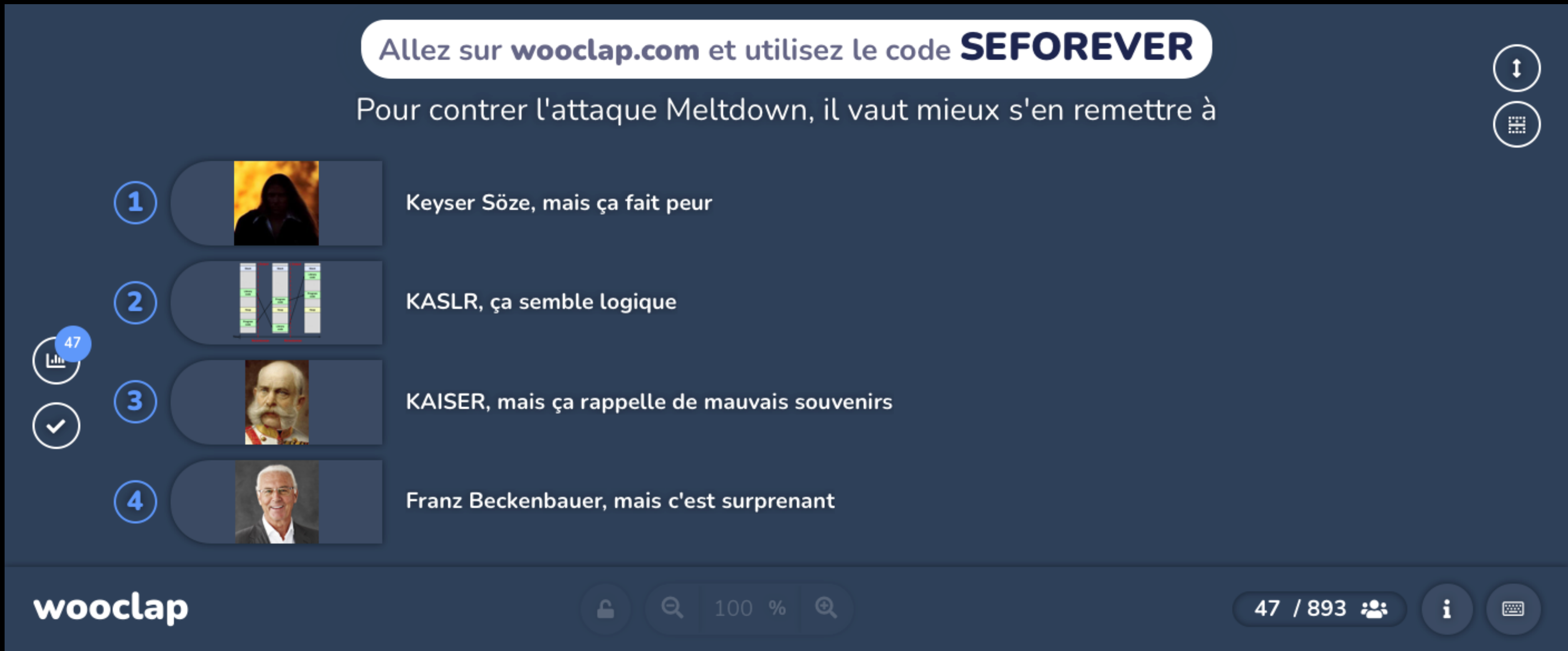
- Let’s see if we can observe address space growth on Linux…
  - Access to [sp – 1 * 4096]
  - Access to [sp – 2 * 4096]
  - Access to [sp – 3 * 4096]
  - …

- Check if #phys_pages increases

```c
int main (int argc, char *argv[])
{
  int i;

  for (int p = 1; p <= 50; p++) {
    int *approx_sp = &i - 1024 * p;
    *approx_sp = 12; // Weird, isn't it?

    printf ("Wrote at address %p (iteration %d)\n",
            approx_sp, p);

    show_nb_phys_pages ();
  }

  return 0;
}
```

Quizz time

https://www.wooclap.com/SEFOREVER



For the record

- KAISER
  - kernel address isolation to have side-channels efficiently removed

First-touch memory allocation

- How does it work?
  - A virtual page which is not allocated is necessarily marked *invalid* in the page table!
    - Otherwise, the MMU would proceed to incorrect translation

	Phys. Page	Valid
0	2	1
1	4	1
2	5	1
3	X	0
4	0	1
5	X	0
6		0
7		0
8		0
9		0
10		0
11	11	1
12	X	0
13	X	0
14		0

First-touch memory allocation

- How does it work?
  - A virtual page which is not allocated is necessarily marked *invalid* in the page table!
    - Otherwise, the MMU would proceed to incorrect translation
  - Access to an invalid page -> page fault
    - How can the kernel distinguish between
      - Lazy allocation

      and
      - Genuine Segmentation Fault?

	Phys. Page	Valid
0	2	1
1	4	1
2	5	1
3	X	0
4	0	1
5	X	0
6		0
7		0
8		0
9		0
10		0
11	11	1
12	X	0
13	X	0
14		0

First-touch memory allocation

- Kernel must keep information about lazy allocations
  - Stored in the page table?

	Phys. Page	Valid	R	W	X
0	2	1	1	0	1
1	4	1	1	0	1
2	5	1	1	0	1
3	X	0			
4	0	1	1	1	0
5	X	0			
6		0			
7		0			
8		0			
9		0			
10		0			
11	11	1	1	1	0
12	X	0			
13	X	0			
14		0			

First-touch memory allocation

- Kernel must keep information about lazy allocations
  - Stored in the page table?
    - Would lead to allocate unnecessary 2nd and 3rd level tables…
  - In a separate kernel data structure
    - In theory, for each page, the kernel must keep
      - “should it be allocated on first touch?”
      - “if so, what rights should be set?”

	Phys. Page	Valid	R	W	X
0	2	1	1	0	1
1	4	1	1	0	1
2	5	1	1	0	1
3	X	0			
4	0	1	1	1	0
5	X	0			
6		0			
7		0			
8		0			
9		0			
10		0			
11	11	1	1	1	0
12	X	0			
13	X	0			
14		0			

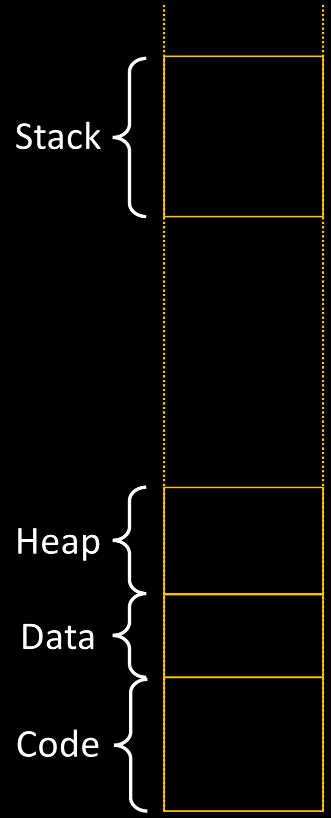
First-touch memory allocation

- Kernel must keep information about lazy allocations
  - Stored in the page table?
    - Would lead to allocate unnecessary 2nd and 3rd level tables…
  - In a separate kernel data structure

	Phys. Page	Valid	R	W	X
0	2	1	1	0	1
1	4	1	1	0	1
2	5	1	1	0	1
3	X	0			
4	0	1	1	1	0
5	X	0			
6		0			
7		0			
8		0			
9		0			
10		0			
11	11	1	1	1	0
12	X	0			
13	X	0			
14		0			

First-touch memory allocation

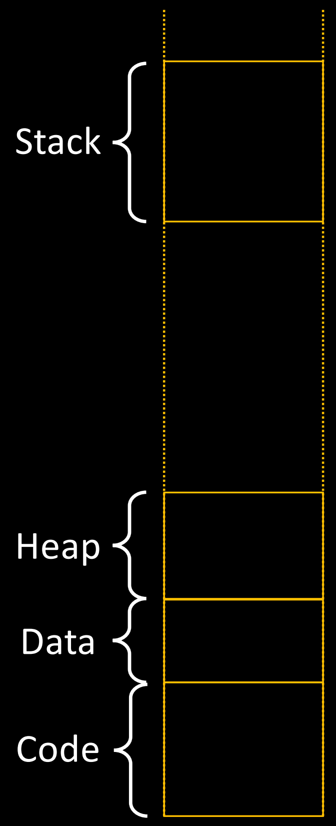
- A more compact data structure is the list of *Virtual Memory Areas (VMA)*
  - Contiguous series of virtual pages sharing the same characteristics
    - Typically a few dozens of areas
  - A list of VMAs is kept for each process



First-touch memory allocation

- Actually, a more compact information is the list of *Virtual Memory Areas (VMA)*

```
struct vm_area_struct {
  unsigned long vm_start;
  unsigned long vm_end;
  pgprot_t vm_page_prot;
  unsigned short vm_flags;
  struct file * vm_file;
  …
};
```

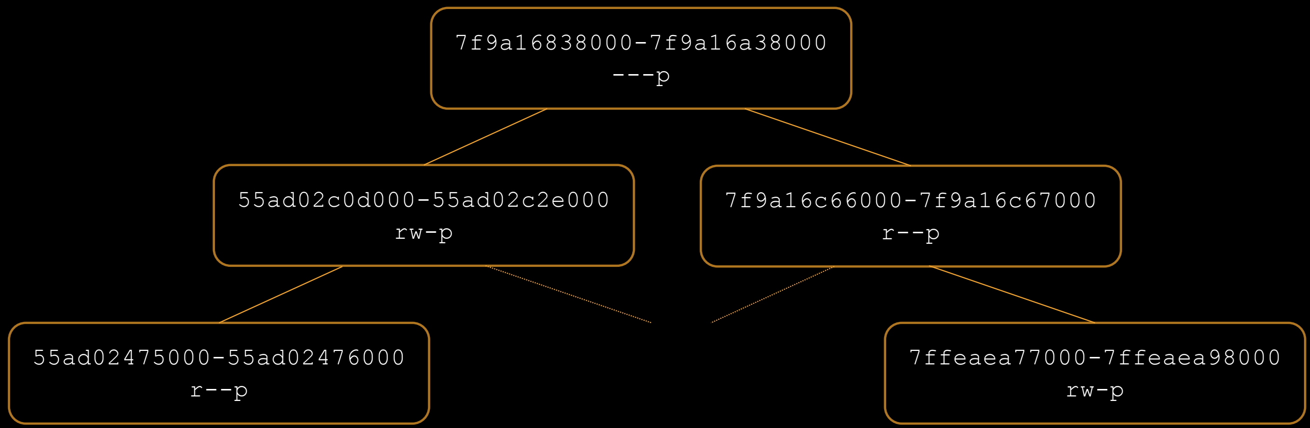


Virtual Memory Areas of a process

```
[jolicoeur] cat /proc/self/maps
    55ad0226e000-55ad02276000 r-xp 00000000 08:01 1573289                    /bin/cat
    55ad02475000-55ad02476000 r--p 00007000 08:01 1573289                    /bin/cat
    55ad02476000-55ad02477000 rw-p 00008000 08:01 1573289                    /bin/cat
    55ad02c0d000-55ad02c2e000 rw-p 00000000 00:00 0                          [heap]
    7f9a1646b000-7f9a1669e000 r--p 00000000 08:01 7079259                    /usr/lib/locale/locale-archive
    7f9a166a3000-7f9a16838000 r-xp 00000000 08:01 8131225                    /lib/x86_64-linux-gnu/libc-2.24.so
    7f9a16838000-7f9a16a38000 ---p 00195000 08:01 8131225                    /lib/x86_64-linux-gnu/libc-2.24.so
    7f9a16a38000-7f9a16a3c000 r--p 00195000 08:01 8131225                    /lib/x86_64-linux-gnu/libc-2.24.so
    7f9a16a3c000-7f9a16a3e000 rw-p 00199000 08:01 8131225                    /lib/x86_64-linux-gnu/libc-2.24.so
    7f9a16a43000-7f9a16a66000 r-xp 00000000 08:01 8128192                    /lib/x86_64-linux-gnu/ld-2.24.so
    7f9a16c66000-7f9a16c67000 r--p 00023000 08:01 8128192                    /lib/x86_64-linux-gnu/ld-2.24.so
    7f9a16c67000-7f9a16c68000 rw-p 00024000 08:01 8128192                    /lib/x86_64-linux-gnu/ld-2.24.so
    7ffeaea77000-7ffeaea98000 rw-p 00000000 00:00 0                          [stack]
```

First-touch memory allocation

- VMAs of a process are stored in an AVL tree
  - Self-balancing, binary search tree, O(log(n))



First-touch memory allocation

- When a page fault exception occurs
  - The MMU keeps the faulty virtual address in a special register
    - E.g. CR2 register on Intel X86
  - The kernel searches if the corresponding virtual page belongs to an existing VMA

	Phys. Page	Valid	R	W	X
0	2	1	1	0	1
1	4	1	1	0	1
2	5	1	1	0	1
3	X	0			
4	0	1	1	1	0
5	X	0			
6		0			
7		0			
8		0			
9		0			
10		0			
11	11	1	1	1	0
12	X	0			
13	X	0			
14		0			

First-touch memory allocation

- When a page fault exception occurs
  - The MMU keeps the faulty virtual address in a special register
    - E.g. CR2 register on Intel X86
  - The kernel searches if the corresponding virtual page belongs to an existing VMA
    - Yes -> it's a first touch allocation
      - get_free_page () and
        fix the page table entry

	Phys. Page	Valid	R	W	X
0	2	1	1	0	1
1	4	1	1	0	1
2	5	1	1	0	1
3	X	0			
4	0	1	1	1	0
5	20	1	1	1	0
6		0			
7		0			
8		0			
9		0			
10		0			
11	11	1	1	1	0
12	X	0			
13	X	0			
14		0			

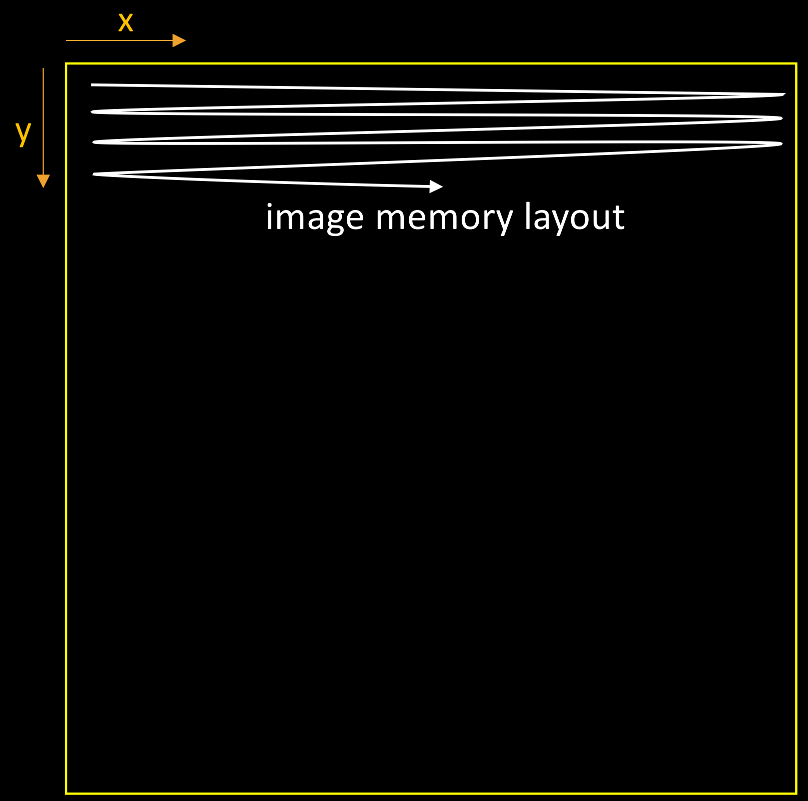
First-touch memory allocation

- When a page fault exception occurs
  - The MMU keeps the faulty virtual address in a special register
    - E.g. CR2 register on Intel X86
  - The kernel searches if the corresponding virtual page belongs to an existing VMA
    - No -> It's a Segmentation Fault
      - No mercy!
        Send SIGSEGV to process

	Phys. Page	Valid	R	W	X
0	2	1	1	0	1
1	4	1	1	0	1
2	5	1	1	0	1
3	X	0			
4	0	1	1	1	0
5	X	0			
6		0			
7		0			
8		0			
9		0			
10		0			
11	11	1	1	1	0
12	X	0			
13	X	0			
14		0			

First-touch memory allocation

- Consequences in everyday life
  - Large, uninitialized data structures are allocated one-page-at-a-time
    - Significant access time variability when crossing page boundaries
    - Example
      - #define DIM 2048
      - unsigned image[DIM][DIM];
        - (pixels format: RGBA8888)



First-touch memory allocation

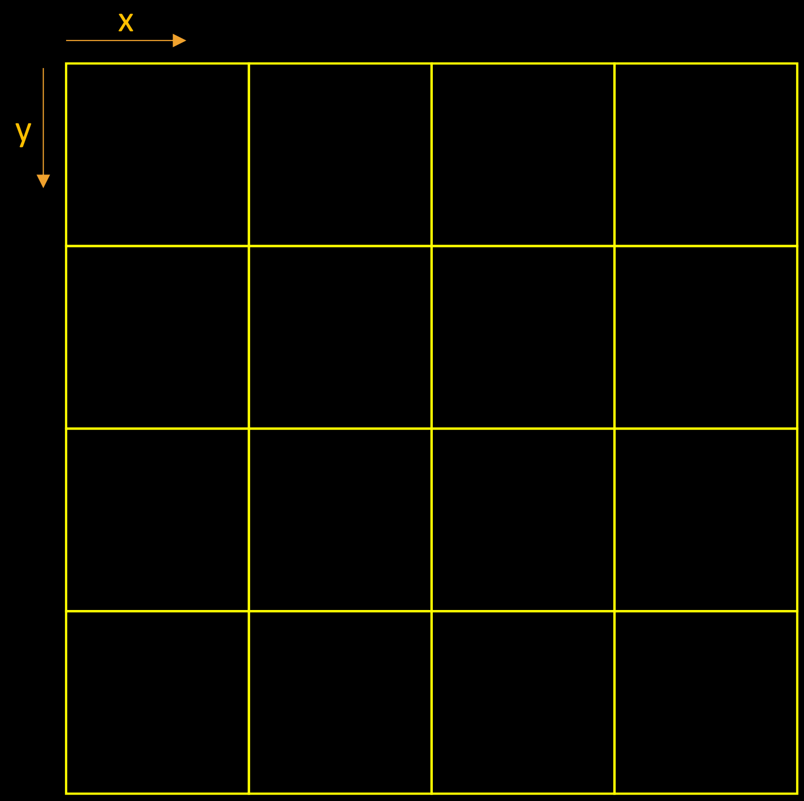
- Example
  - #define DIM 2048
  - unsigned image[DIM][DIM];
    - (pixels format: RGBA8888)

- “invert” iterative computation
  - Compute negative of previous image
    - image[i][j] ^= 0xFFFFFF00;

First-touch memory allocation

- Tiled computation scheme

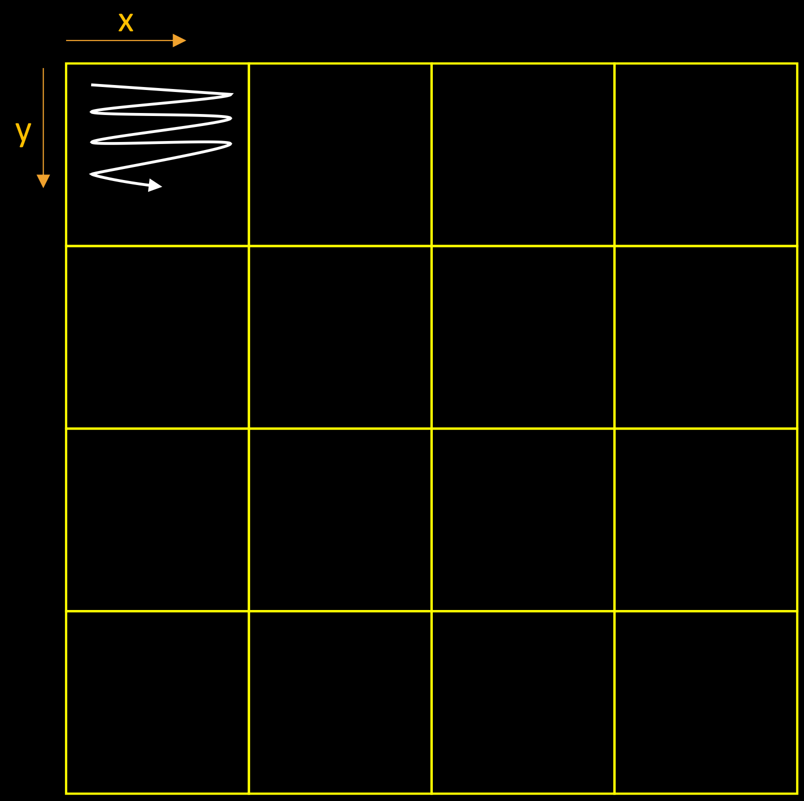
```
void do_tile (int x, int y, int width, int height)
{
  for (int i = y; i < y + height; i++)
    for (int j = x; j < x + width; j++)
      image (i, j) = ^0xFFFFFF00;
}

for (int y = 0; y < DIM; y += TILE_SIZE)
  for (int x = 0; x < DIM; x += TILE_SIZE)
    do_tile (x, y, TILE_SIZE, TILE_SIZE);
```



First-touch memory allocation

- Tiled computation scheme

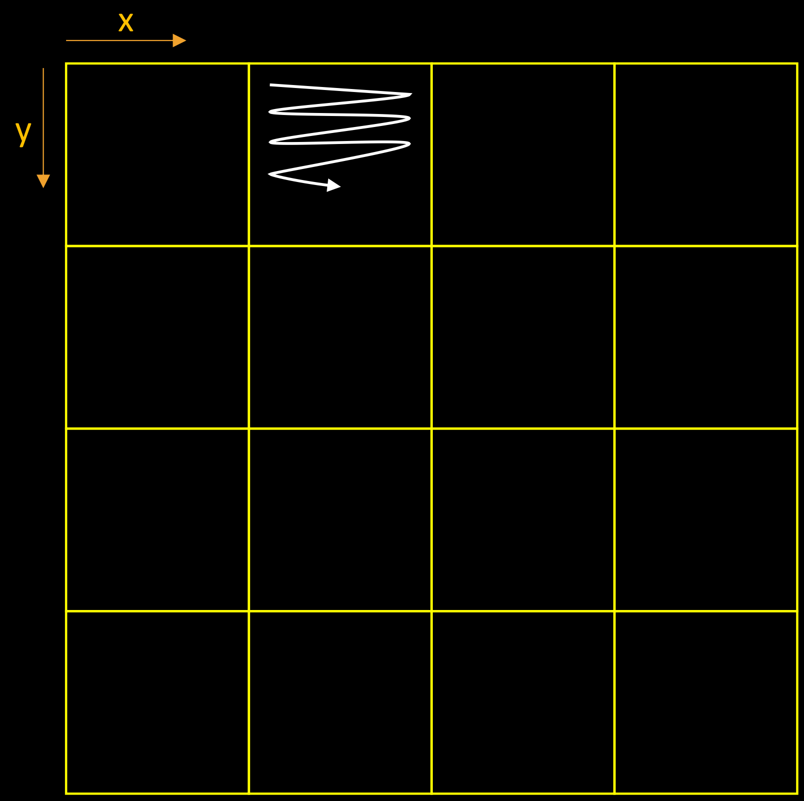
```
void do_tile (int x, int y, int width, int height)
{
  for (int i = y; i < y + height; i++)
    for (int j = x; j < x + width; j++)
      image (i, j) = ^0xFFFFFF00;
}

for (int y = 0; y < DIM; y += TILE_SIZE)
  for (int x = 0; x < DIM; x += TILE_SIZE)
    do_tile (x, y, TILE_SIZE, TILE_SIZE);
```



First-touch memory allocation

- Tiled computation scheme

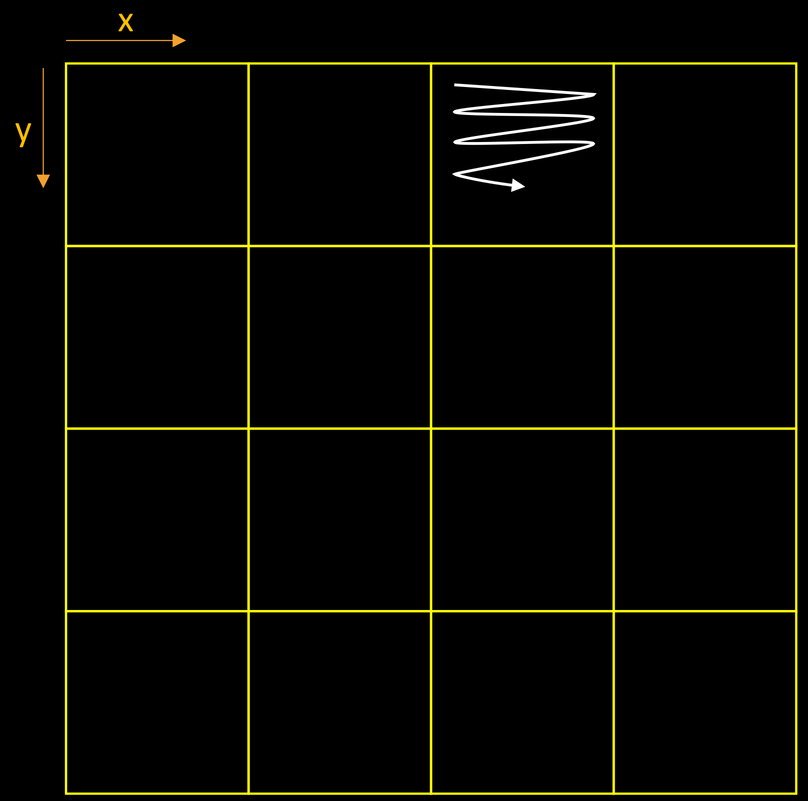
```
void do_tile (int x, int y, int width, int height)
{
  for (int i = y; i < y + height; i++)
    for (int j = x; j < x + width; j++)
      image (i, j) = ^0xFFFFFF00;
}

for (int y = 0; y < DIM; y += TILE_SIZE)
  for (int x = 0; x < DIM; x += TILE_SIZE)
    do_tile (x, y, TILE_SIZE, TILE_SIZE);
```



First-touch memory allocation

- Tiled computation scheme

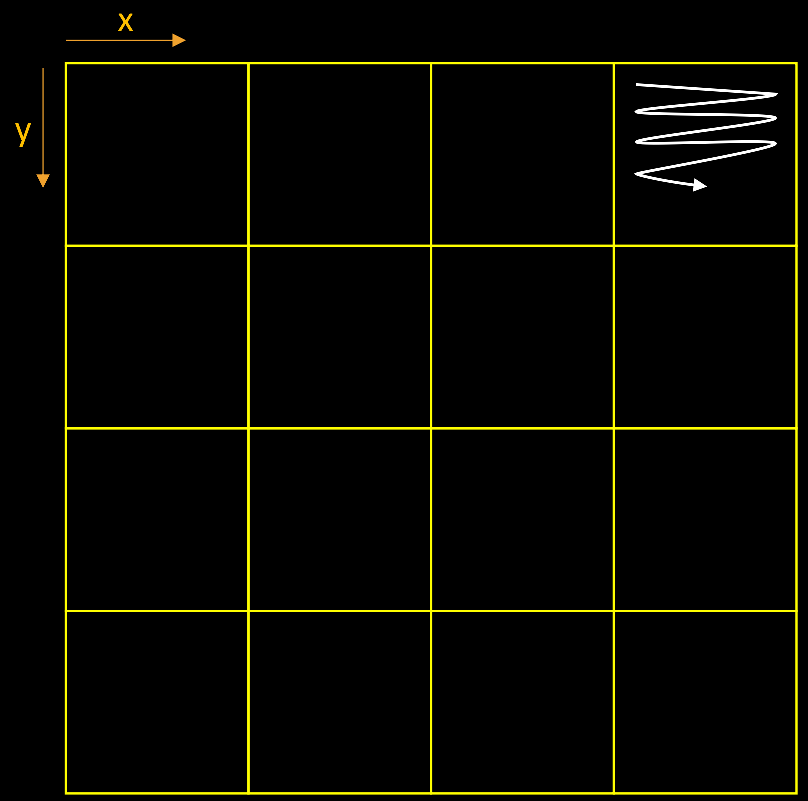
```
void do_tile (int x, int y, int width, int height)
{
  for (int i = y; i < y + height; i++)
    for (int j = x; j < x + width; j++)
      image (i, j) = ^0xFFFFFF00;
}

for (int y = 0; y < DIM; y += TILE_SIZE)
  for (int x = 0; x < DIM; x += TILE_SIZE)
    do_tile (x, y, TILE_SIZE, TILE_SIZE);
```



First-touch memory allocation

- Tiled computation scheme

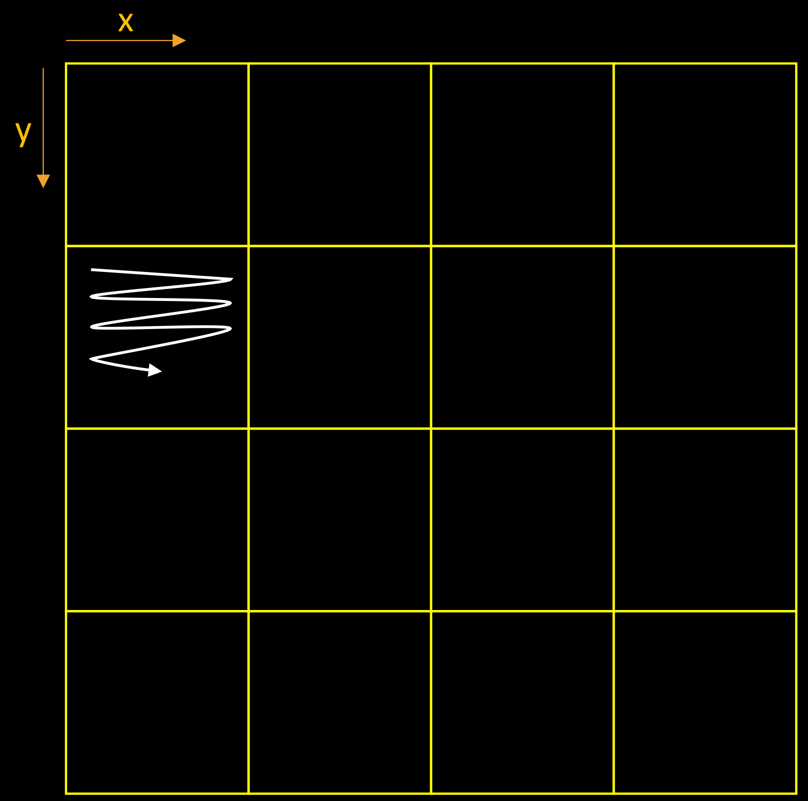
```
void do_tile (int x, int y, int width, int height)
{
  for (int i = y; i < y + height; i++)
    for (int j = x; j < x + width; j++)
      image (i, j) = ^0xFFFFFF00;
}

for (int y = 0; y < DIM; y += TILE_SIZE)
  for (int x = 0; x < DIM; x += TILE_SIZE)
    do_tile (x, y, TILE_SIZE, TILE_SIZE);
```



First-touch memory allocation

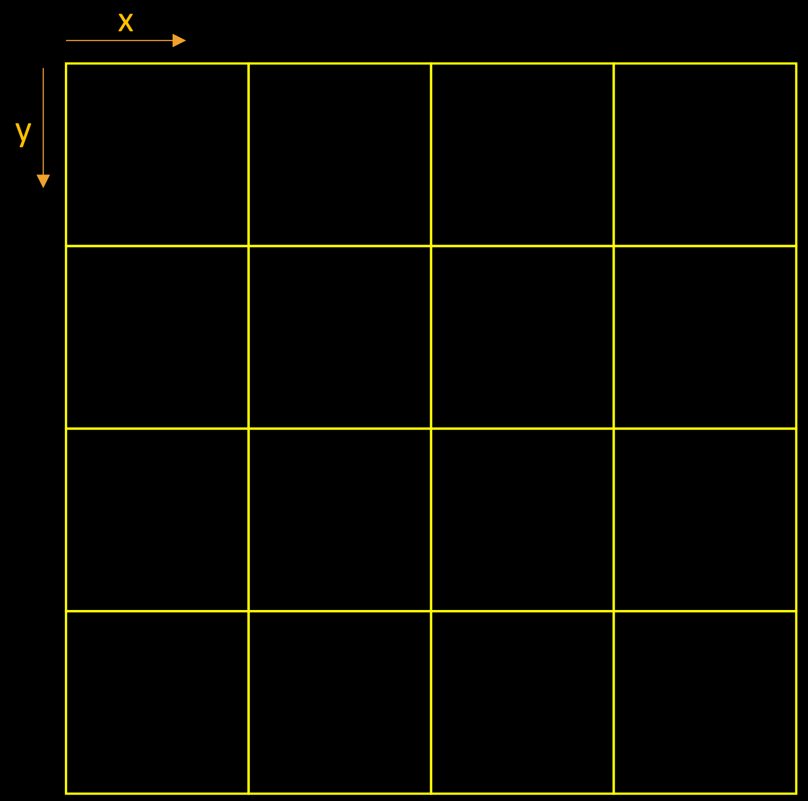
- Tiled computation scheme

```
void do_tile (int x, int y, int width, int height)
{
  for (int i = y; i < y + height; i++)
    for (int j = x; j < x + width; j++)
      image (i, j) = ^0xFFFFFF00;
}

for (int y = 0; y < DIM; y += TILE_SIZE)
  for (int x = 0; x < DIM; x += TILE_SIZE)
    do_tile (x, y, TILE_SIZE, TILE_SIZE);
```



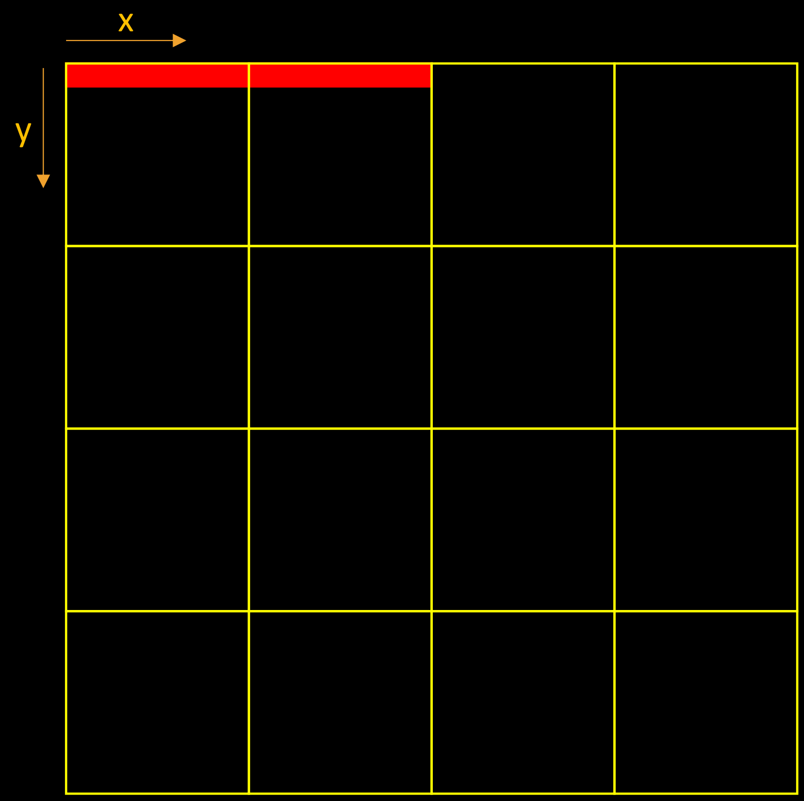
First-touch memory allocation

- Where are the pages?
  - 1024 pixels = 4KB = 1 page



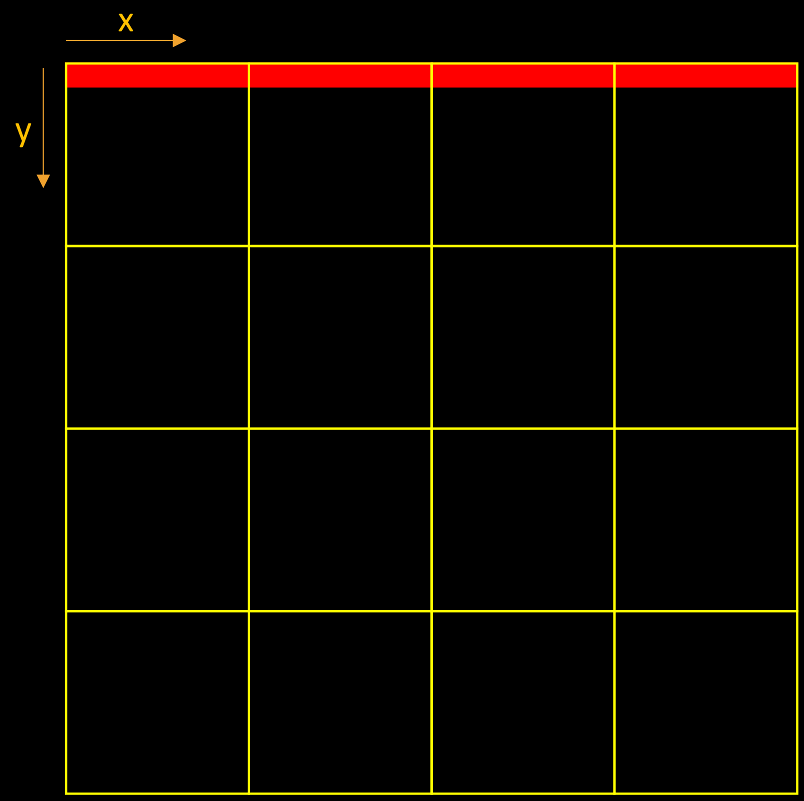
First-touch memory allocation

- Where are the pages?
  - 1024 pixels = 4KB = 1 page



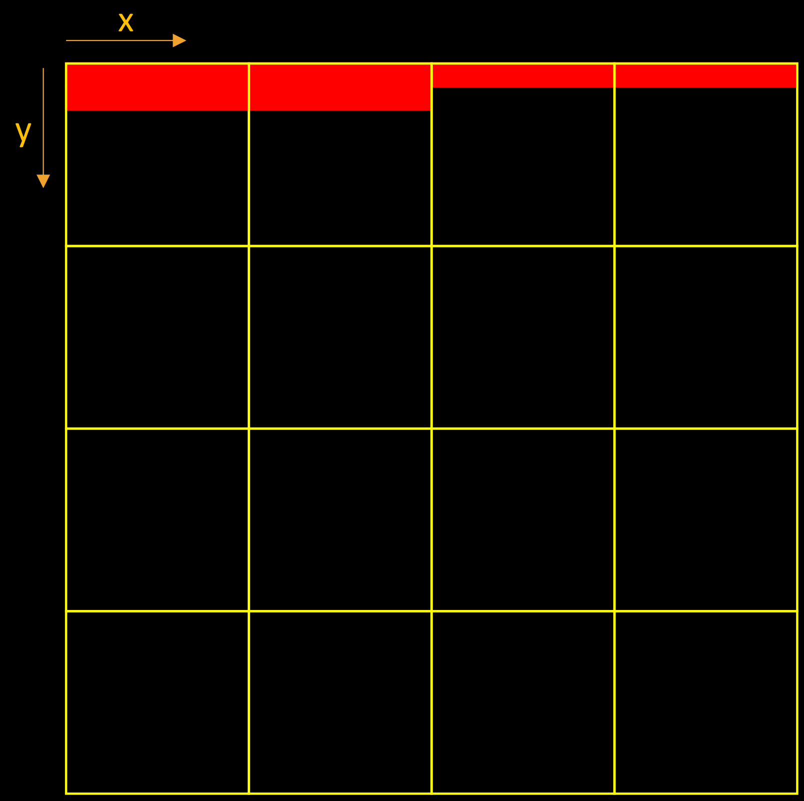
First-touch memory allocation

- Where are the pages?
  - 1024 pixels = 4KB = 1 page



First-touch memory allocation

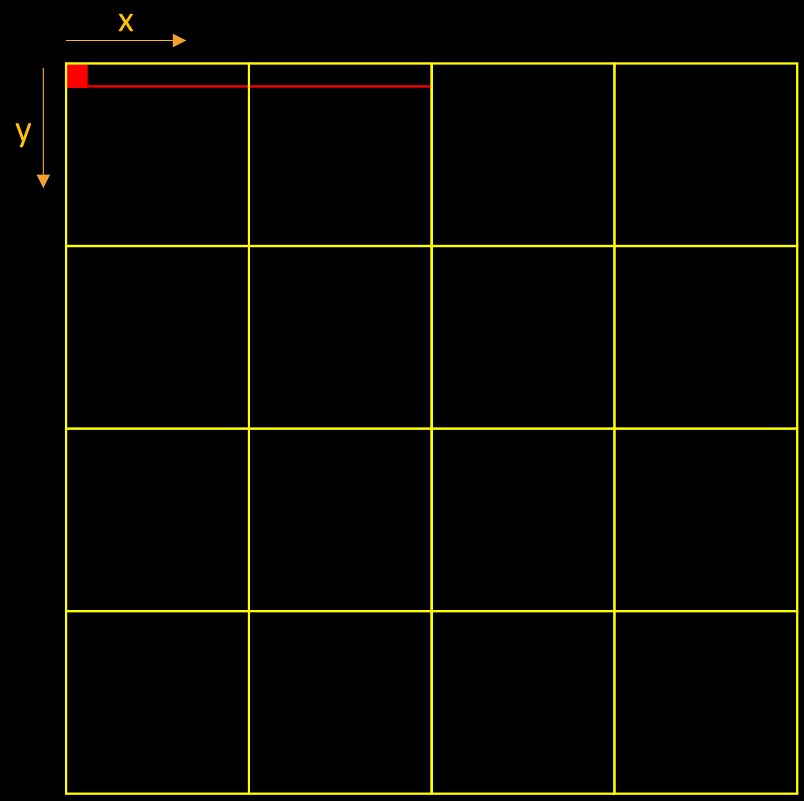
- Where are the pages?
  - 1024 pixels = 4KB = 1 page



First-touch memory allocation

- Where are the pages?
  - 1024 pixels = 4KB = 1 page

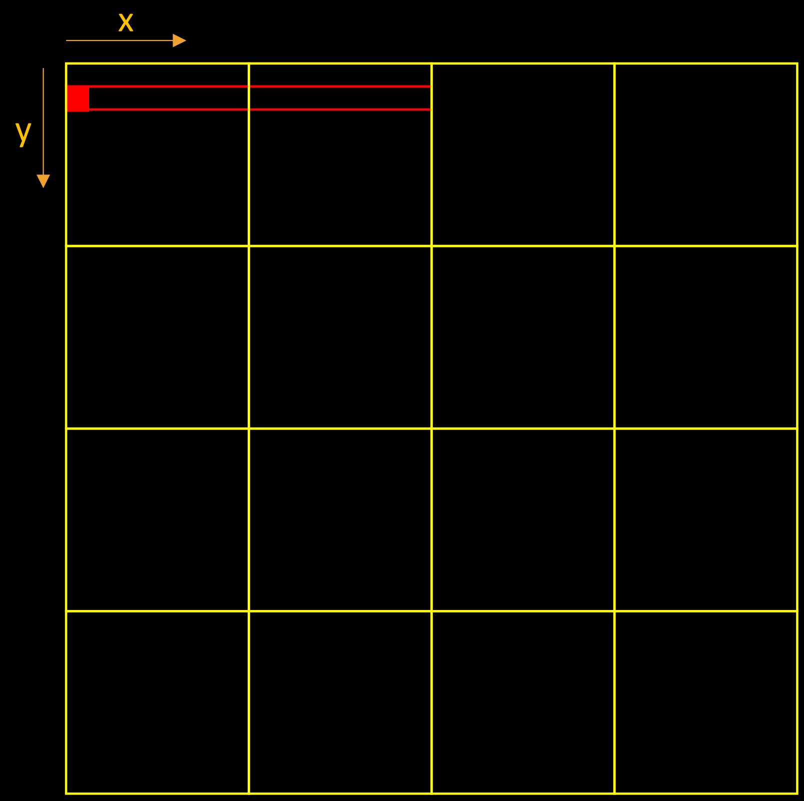
- In this example, the first tile causes 512 *page faults*



First-touch memory allocation

- Where are the pages?
  - 1024 pixels = 4KB = 1 page

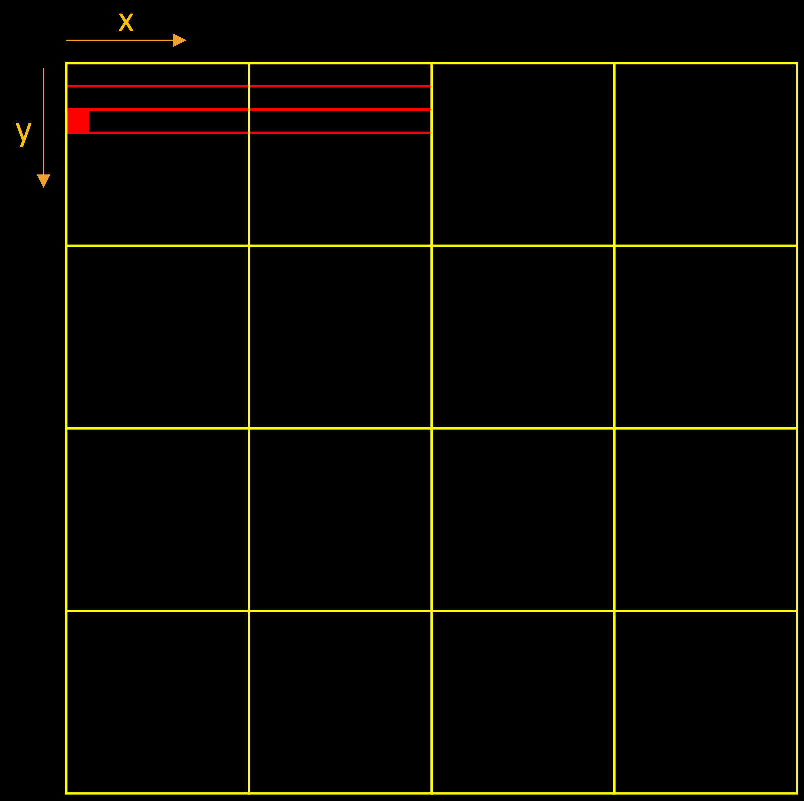
- In this example, the first tile causes 512 *page faults*



First-touch memory allocation

- Where are the pages?
  - 1024 pixels = 4KB = 1 page

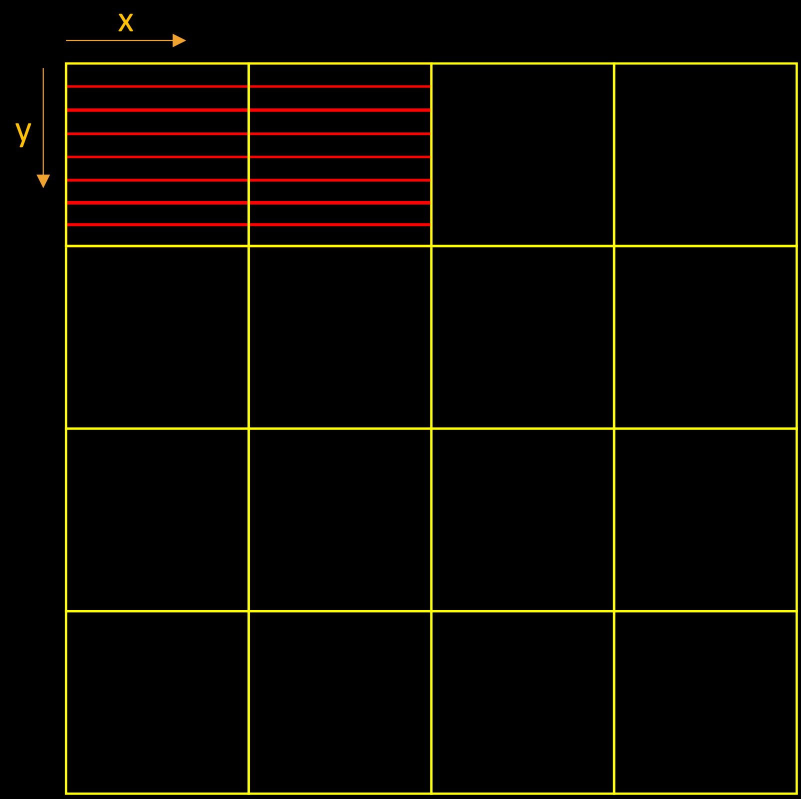
- In this example, the first tile causes 512 *page faults*



First-touch memory allocation

- Where are the pages?
  - 1024 pixels = 4KB = 1 page

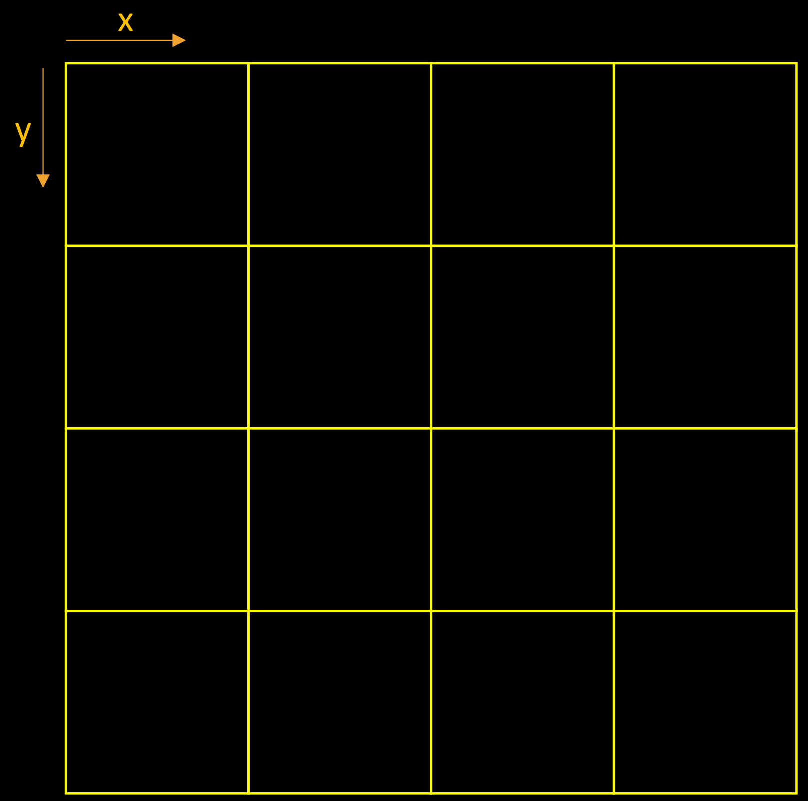
- In this example, the first tile causes 512 *page faults*
  - The second tile involves none



First-touch memory allocation

- Let us collect profiling data

```
void do_tile (int x, int y, int width, int height)
{
  tile_start (…);

  for (int i = y; i < y + height; i++)
    for (int j = x; j < x + width; j++)
      image (i, j) = ^0xFFFFFF00;

  tile_stop (…);
}
```



First-touch memory allocation

- Consequences in everyday life
  - Large, uninitialized data structures are allocated one-page-at-a-time
    - Significant access time variability when crossing page boundaries
  - In an upcoming course, we'll see that the core responsible for the first-touch access really matters

- By the way
  - calloc ≠ malloc + bzero
    - calloc can efficiently reserve a pool of (blank) virtual pages
    - malloc can do as well…
      but bzero will immediately trigger allocations

Allez sur **wooclap.com** et utilisez le code **SEFOREVER**

Dans la série "Ces acronymes qui nous pourrissent la vie", quelles affirmations sont vraies ?

1. Une VMA est une zone de l'espace d'adressage à laquelle on ne peut pas accéder
2. Le TLB est une mémoire accueillant la table de 1er niveau du processus en cours
3. La MMU est un circuit matériel faisant partie du CPU
4. SEFOREVER traduit le fait que chaque année, vous tenterez de valider l'UE de SE, en vain...

51

wooclap

100 %

51 / 893

The Copy-on-Write mechanism

- Motivation
  - Unix Process creation is historically done in two steps
    - fork () & exec ()

The Copy-on-Write mechanism

- Motivation
  - Unix Process creation is historically done in two steps
    - fork () & exec ()
  - fork creates a clone
    - Child obtains a duplicate of Parent’s address space

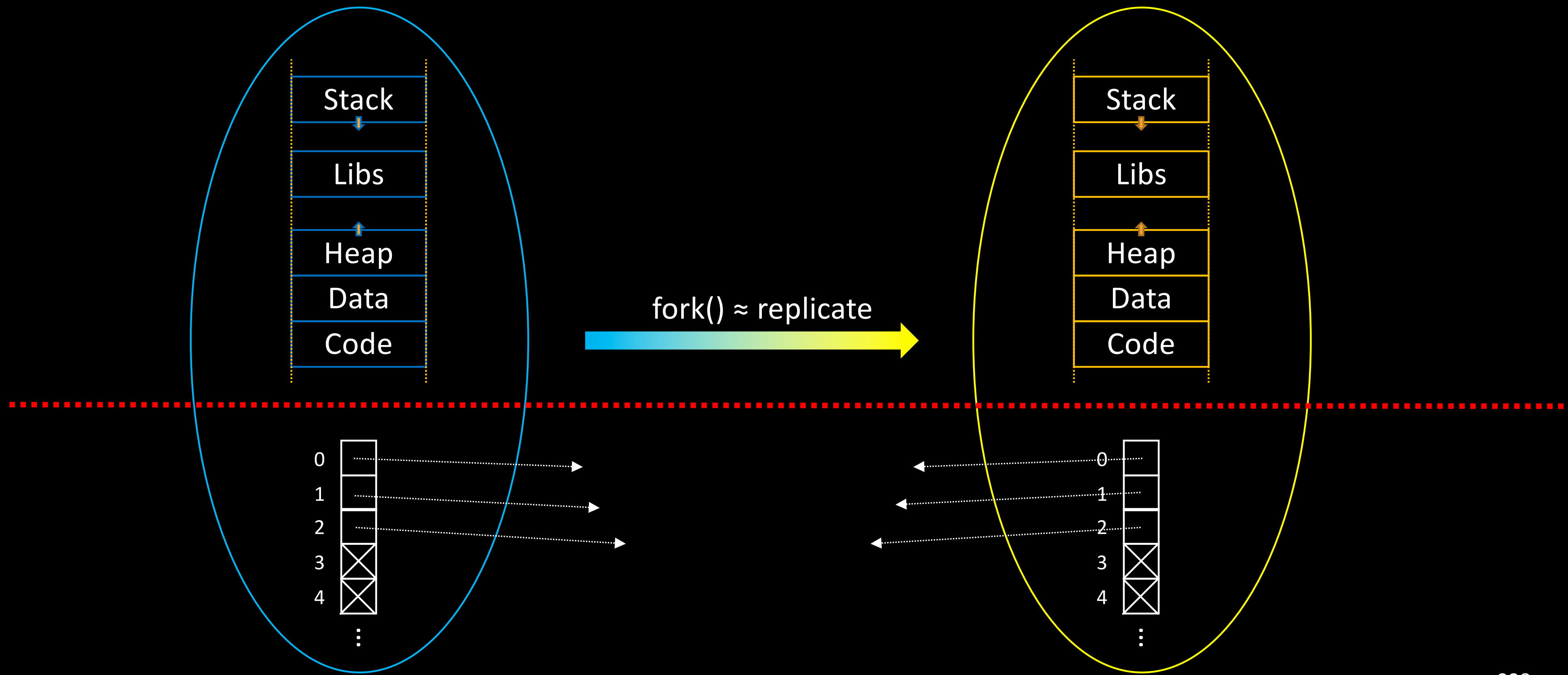
```
int main (int argc, char *argv[])
{
  pid_t pid = fork ();
  if (pid) { // Parent

  } else { // Child

  }
  return 0;
}
```

The Copy-on-Write mechanism

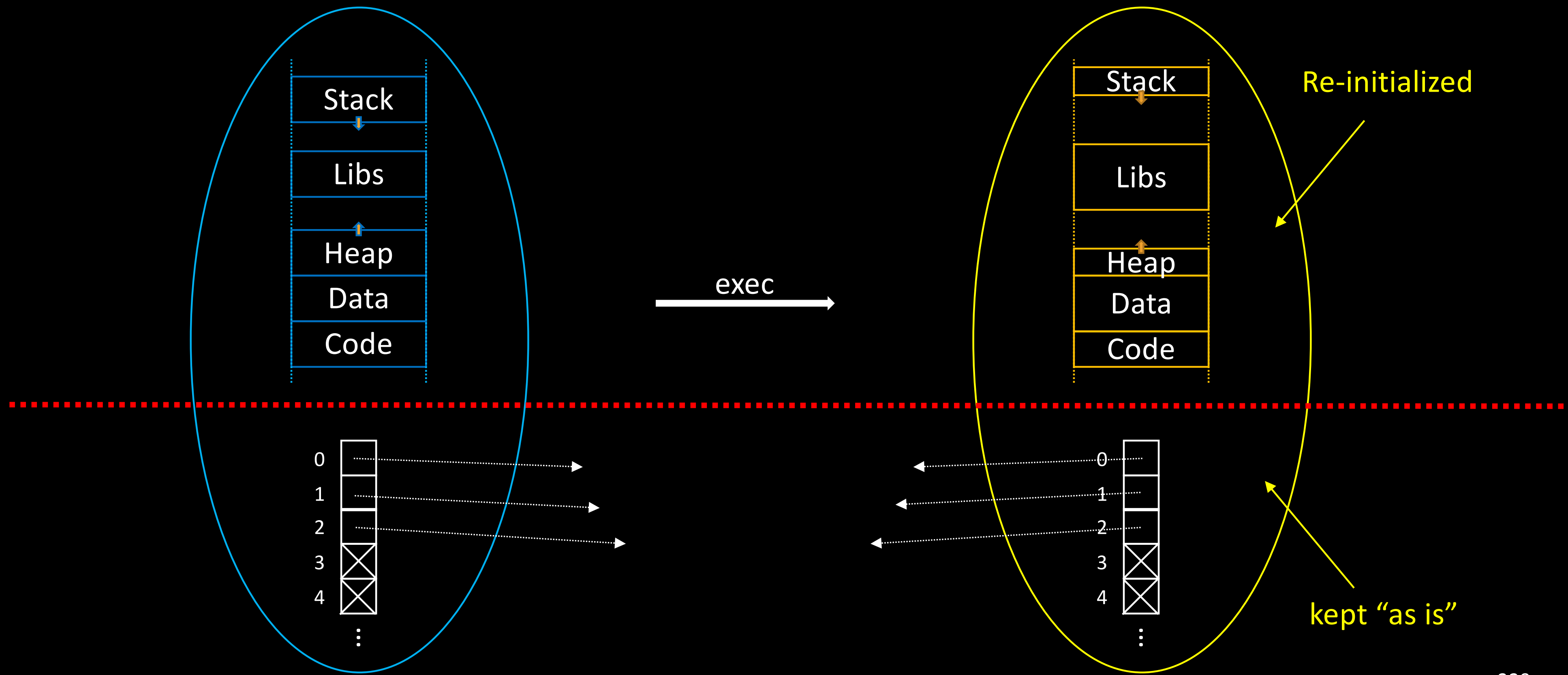
- Motivation
  - Unix Process creation is historically done in two steps
    - fork () & exec ()
  - fork creates a clone
    - Child obtains a duplicate of Parent's address space
  - exec loads a new program
    - User-space part of the child's address space is reset

```
int main (int argc, char *argv[])
{
  pid_t pid = fork ();
  if (pid) { // Parent
    wait (NULL);
  } else { // Child
    execl ("/bin/ls", "ls", "-l", NULL);
    perror ("ls");
    exit (EXIT_FAILURE);
  }
  return 0;
}
```

Process Creation:
fork() replicates the whole bubble!



Exec resets user-space content



The Copy-on-Write mechanism

- Intrinsically inefficient
  - Most of a time, all the pages copied during fork are dropped by exec!
- Parent and Child cannot share the same address space
  - If child doesn’t call exec, both address spaces must evolve independently
- Idea
  - Duplicate Parent’s page table
    - Both processes share the same physical pages
  - Set pages as read-only pages on both sides

The Copy-on-Write mechanism

P_1

	Phys. Page	Valid	R	W	X
0	2	1	1	0	1
1	4	1	1	0	1
2	5	1	1	0	1
3	8	1	1	0	0
4	0	1	1	0	0
5	15	1	1	0	0
6		0			
7		0			
8		0			
9		0			
10		0			
11	11	1	1	0	0
12	7	1	1	0	0
13	14	1	1	0	0
14		0			

Both processes share the same set of pages, but nobody can modify a page…

P_2

	Phys. Page	Valid	R	W	X
0	2	1	1	0	1
1	4	1	1	0	1
2	5	1	1	0	1
3	8	1	1	0	0
4	0	1	1	0	0
5	15	1	1	0	0
6		0			
7		0			
8		0			
9		0			
10		0			
11	11	1	1	0	0
12	7	1	1	0	0
13	14	1	1	0	0
14		0			

The Copy-on-Write mechanism

- This is where CoW comes into play!
  - When one process tries to write to a page
    - We give him a private copy!

P_1

	Phys. Page	Valid	R	W	X
0	2	1	1	0	1
1	4	1	1	0	1
2	5	1	1	0	1
3	8	1	1	0	0
4	0	1	1	0	0
5	15	1	1	0	0
6		0			
7		0			
8		0			
9		0			
10		0			
11	11	1	1	0	0
12	7	1	1	0	0
13	14	1	1	0	0
14		0			

The Copy-on-Write mechanism

- This is where CoW comes into play!
  - When one process tries to write to a page
    - We give him a private copy!
    - How do we make sure it is not a bad access?

P_1

	Phys. Page	Valid	R	W	X
0	2	1	1	0	1
1	4	1	1	0	1
2	5	1	1	0	1
3	8	1	1	0	0
4	0	1	1	0	0
5	15	1	1	0	0
6		0			
7		0			
8		0			
9		0			
10		0			
11	11	1	1	0	0
12	7	1	1	0	0
13	14	1	1	0	0
14		0			

The Copy-on-Write mechanism

- This is where CoW comes into play!
  - When one process tries to write to a page
    - We give him a private copy!
    - How do we make sure it is not a bad access?
      - Was the “write” flag set before we decided to make all pages read-only?

P_1

	Phys. Page	Valid	R	W	X
0	2	1	1	0	1
1	4	1	1	0	1
2	5	1	1	0	1
3	8	1	1	0	0
4	0	1	1	0	0
5	15	1	1	0	0
6		0			
7		0			
8		0			
9		0			
10		0			
11	11	1	1	0	0
12	7	1	1	0	0
13	14	1	1	0	0
14		0			

The Copy-on-Write mechanism

- This is where CoW comes into play!
  - When one process tries to write to a page
    - We give him a private copy!
    - How do we make sure it is not a bad access?
      - Was the “write” flag set before we decided to make all pages read-only?

    - Again, the list of VMAs is our friend!

P_1

	Phys. Page	Valid	R	W	X
0	2	1	1	0	1
1	4	1	1	0	1
2	5	1	1	0	1
3	8	1	1	0	0
4	0	1	1	0	0
5	15	1	1	0	0
6		0			
7		0			
8		0			
9		0			
10		0			
11	11	1	1	0	0
12	7	1	1	0	0
13	14	1	1	0	0
14		0			

The Copy-on-Write mechanism

- This is where CoW comes into play!
  - When one process tries to write to a page
    - The kernel checks the rights stored in the VMA that the faulty page belongs to
      - “write” flag off?
        - Segmentation Fault
      - “write” flag on?
        - We perform a Cow

P_1

	Phys. Page	Valid	R	W	X
0	2	1	1	0	1
1	4	1	1	0	1
2	5	1	1	0	1
3	8	1	1	0	0
4	0	1	1	0	0
5	15	1	1	0	0
6		0			
7		0			
8		0			
9		0			
10		0			
11	11	1	1	0	0
12	7	1	1	0	0
13	14	1	1	0	0
14		0			

The Copy-on-Write mechanism

- Copy-on-Write
  - Allocate a new physical page
    - get_free_page() -> 10

P_1

	Phys. Page	Valid	R	W	X
0	2	1	1	0	1
1	4	1	1	0	1
2	5	1	1	0	1
3	8	1	1	0	0
4	0	1	1	0	0
5	15	1	1	0	0
6		0			
7		0			
8		0			
9		0			
10		0			
11	11	1	1	0	0
12	7	1	1	0	0
13	14	1	1	0	0
14		0			

The Copy-on-Write mechanism

- Copy-on-Write
  - Allocate a new physical page
    - get_free_page() -> 10
  - Copy contents of pp #15 to pp #10
    - memcpy

P_1

	Phys. Page	Valid	R	W	X
0	2	1	1	0	1
1	4	1	1	0	1
2	5	1	1	0	1
3	8	1	1	0	0
4	0	1	1	0	0
5	15	1	1	0	0
6		0			
7		0			
8		0			
9		0			
10		0			
11	11	1	1	0	0
12	7	1	1	0	0
13	14	1	1	0	0
14		0			

The Copy-on-Write mechanism

- Copy-on-Write
  - Allocate a new physical page
    - get_free_page() -> 10
  - Copy contents of pp #15 to pp #10
    - memcpy
  - Fix the page table
    - New physical page number
    - Rights from VMA

P_1

	Phys. Page	Valid	R	W	X
0	2	1	1	0	1
1	4	1	1	0	1
2	5	1	1	0	1
3	8	1	1	0	0
4	0	1	1	0	0
5	10	1	1	1	0
6		0			
7		0			
8		0			
9		0			
10		0			
11	11	1	1	0	0
12	7	1	1	0	0
13	14	1	1	0	0
14		0			

The Copy-on-Write mechanism

- Copy-on-Write
  - Allocate a new physical page
    - get_free_page() -> 10
  - Copy contents of pp #15 to pp #30
    - memcpy
  - Fix the page table
    - New physical page number
    - Rights from VMA

  - Physical page #15 is no longer shared by current process

P1

	Phys. Page	Valid	R	W	X
0	2	1	1	0	1
1	4	1	1	0	1
2	5	1	1	0	1
3	8	1	1	0	0
4	0	1	1	0	0
5	10	1	1	1	0
6		0			
7		0			
8		0			
9		0			
10		0			
11	11	1	1	0	0
12	7	1	1	0	0
13	14	1	1	0	0
14		0			

The Copy-on-Write mechanism

P_1

	Phys. Page	Valid	R	W	X
0	2	1	1	0	1
1	4	1	1	0	1
2	5	1	1	0	1
3	8	1	1	0	0
4	0	1	1	0	0
5	10	1	1	1	0
6		0			
7		0			
8		0			
9		0			
10		0			
11	11	1	1	0	0
12	7	1	1	0	0
13	14	1	1	0	0
14		0			

Situation after CoW…

P_2

	Phys. Page	Valid	R	W	X
0	2	1	1	0	1
1	4	1	1	0	1
2	5	1	1	0	1
3	8	1	1	0	0
4	0	1	1	0	0
5	15	1	1	0	0
6		0			
7		0			
8		0			
9		0			
10		0			
11	11	1	1	0	0
12	7	1	1	0	0
13	14	1	1	0	0
14		0			

The Copy-on-Write mechanism

P_1

	Phys. Page	Valid	R	W	X
0	2	1	1	0	1
1	4	1	1	0	1
2	5	1	1	0	1
3	8	1	1	0	0
4	0	1	1	0	0
5	10	1	1	1	0
6		0			
7		0			
8		0			
9		0			
10		0			
11	11	1	1	0	0
12	7	1	1	0	0
13	14	1	1	0	0
14		0			

Situation after CoW…

Shall we do something for P_2?

P_2

	Phys. Page	Valid	R	W	X
0	2	1	1	0	1
1	4	1	1	0	1
2	5	1	1	0	1
3	8	1	1	0	0
4	0	1	1	0	0
5	15	1	1	0	0
6		0			
7		0			
8		0			
9		0			
10		0			
11	11	1	1	0	0
12	7	1	1	0	0
13	14	1	1	0	0
14		0			

The Copy-on-Write mechanism

- Copy-on-Write
  - After P_1's page table is fixed, we feel like we should also fix page table of P_2
    - Otherwise, if P_2 attempts to write to virtual page #5, we'll perform a silly copy-on-write!

P_1

	Phys. Page	Valid	R	W	X
0	2	1	1	0	1
1	4	1	1	0	1
2	5	1	1	0	1
3	8	1	1	0	0
4	0	1	1	0	0
5	10	1	1	1	0
6		0			
7		0			
8		0			
9		0			
10		0			
11	11	1	1	0	0
12	7	1	1	0	0
13	14	1	1	0	0
14		0			

The Copy-on-Write mechanism

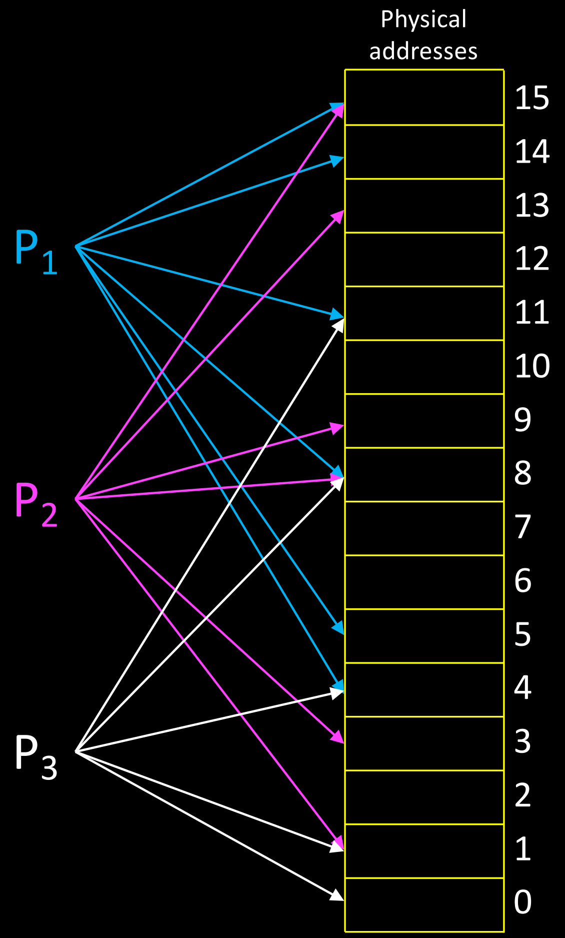
- Copy-on-Write
  - After P_1's page table is fixed, we feel like we should also fix page table of P_2
    - Otherwise, if P_2 attempts to write to virtual page #5, we'll perform a silly copy-on-write!
    - OK, but how do we know the list of processes sharing a physical page?
      - Indeed, there can be many processes sharing a single page
        - fork() cascade…

P_1

	Phys. Page	Valid	R	W	X
0	2	1	1	0	1
1	4	1	1	0	1
2	5	1	1	0	1
3	8	1	1	0	0
4	0	1	1	0	0
5	10	1	1	1	0
6		0			
7		0			
8		0			
9		0			
10		0			
11	11	1	1	0	0
12	7	1	1	0	0
13	14	1	1	0	0
14		0			

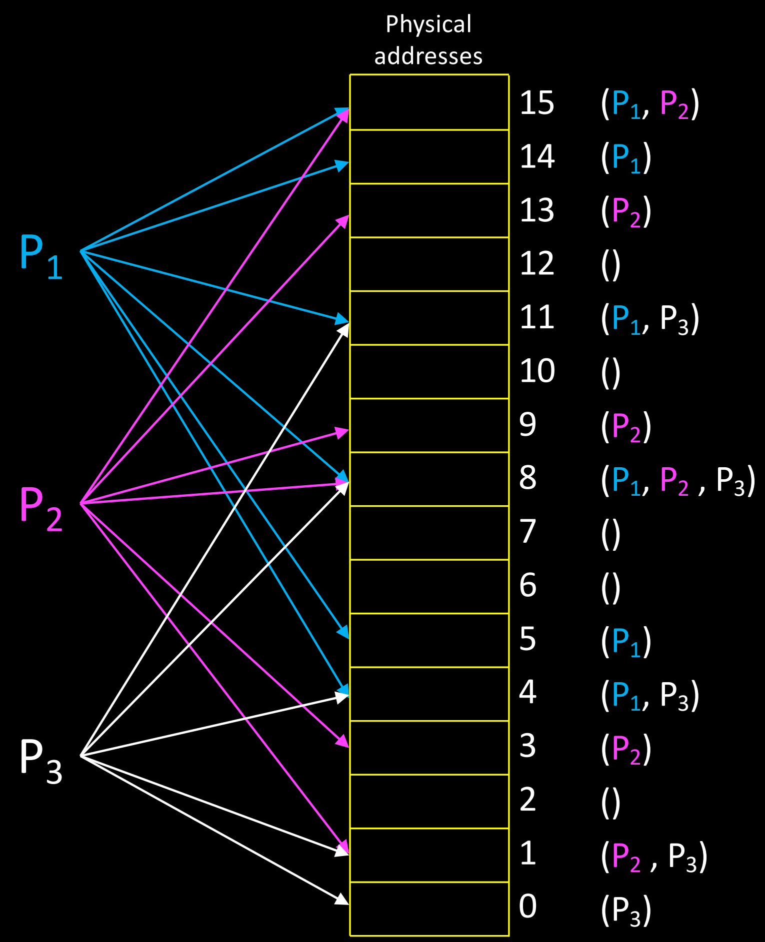
The Copy-on-Write mechanism

- Copy-on-Write
  - After P_1's page table is fixed, we feel like we should also fix page table of P_2
    - Otherwise, if P_2 attempts to write to virtual page #5, we'll perform a silly copy-on-write!!
    - OK, but how do we know the list of processes sharing a physical page?
      - Indeed, there can be many processes sharing a single page
        - fork() cascade…



The Copy-on-Write mechanism

- Shall we keep, for each physical page, a list of owners?
  - This way, after a CoW, we can fix the table of the lonely owner of a page if needed
  - But maintaining lists of processes is costly



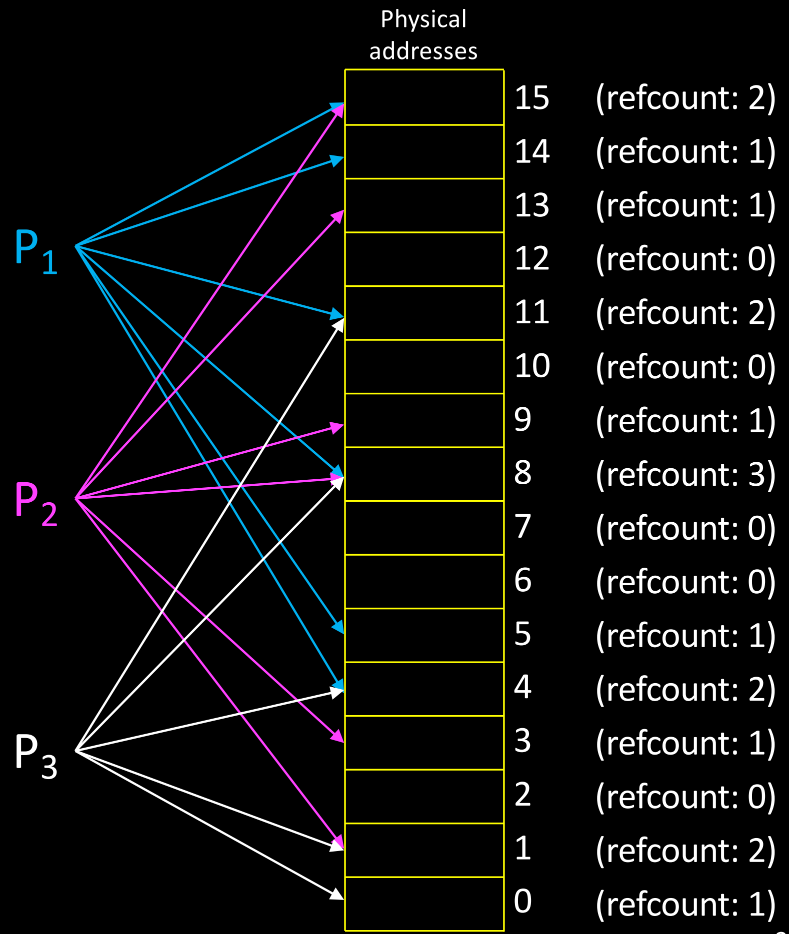
The Copy-on-Write mechanism

- We can maintain a simple reference counter instead

  ```
  unsigned refcount[#PhysPages];
  ```

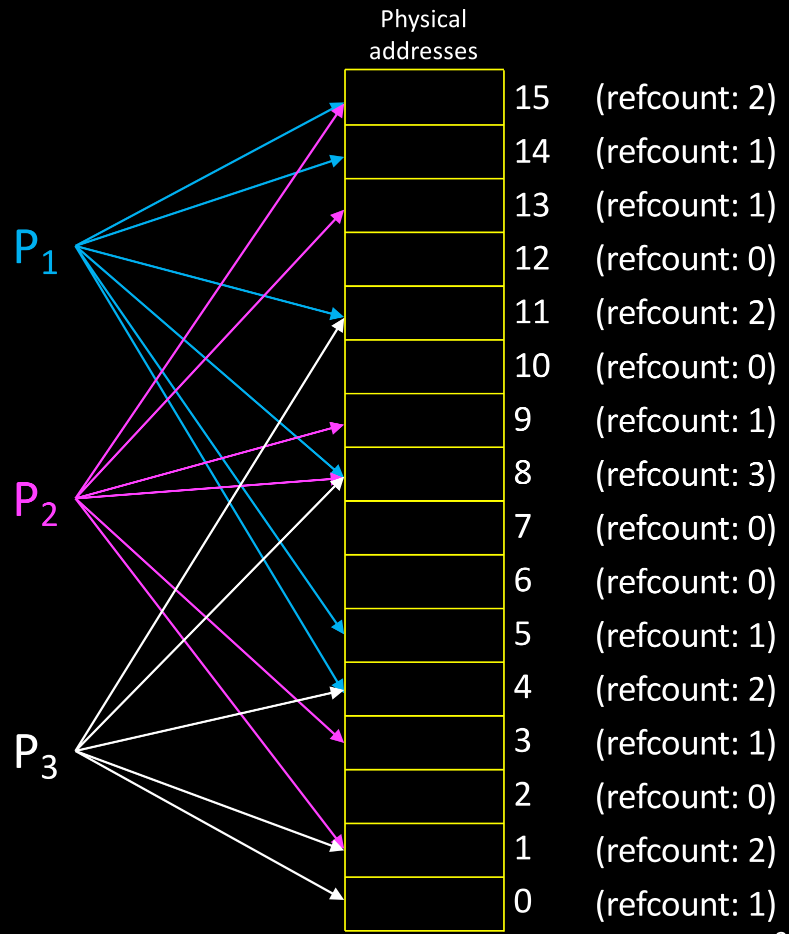
  - Increased by fork ()
  - Decreased by CoW

- We can no longer fix the pageTable of the last owner…
  - We can't avoid the disgrace of an extra page fault 😱



The Copy-on-Write mechanism

- We can no longer fix the pageTable of the last owner…
  - However, when a page fault occurs, the last owner sees refcount = 1
    - He can avoid a silly CoW and just fix his table



The Copy-on-Write mechanism

- Wrap-up
  - The CoW mechanism allows multiple processes to share pages as long as they do not attempt to modify them
  - It’s incredibly effective given that fork() is usually followed by exec()…
  - It’s also useful with shared memory-mapped files
    - (to be explored in next chapter)

Allez sur **wooclap.com** et utilisez le code **SEFOREVER**

Le mécanisme d'allocation paresseuse

1. Permet aux processus de s'assoupir lors d'un malloc
2. Permet au noyau de faire son radin quand on lui demande des pages
3. Permet de différer l'allocation des pages jusqu'à la première tentative d'accès
4. Permet de différer l'allocation des pages juqu'à la première tentative d'écriture

48

wooclap

100 %

48 / 893

Additional resources available on http://gforgeron.gitlab.io/se/