Memory Virtualization:
Paging

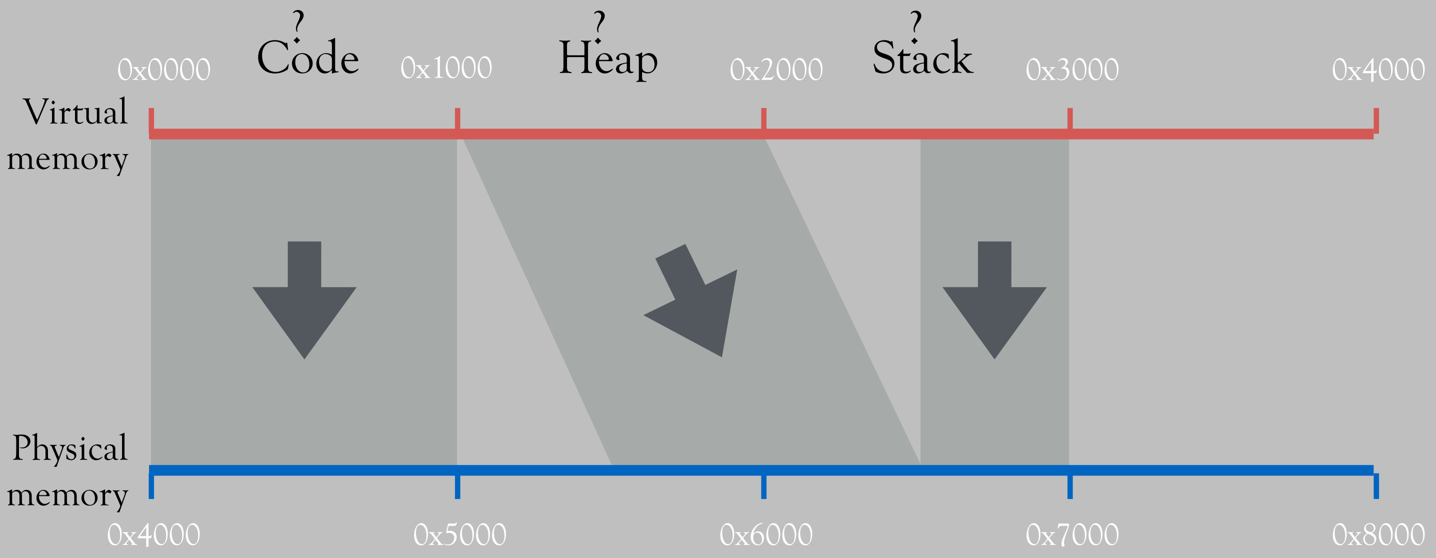
OSTEP Chapter 18:
http://pages.cs.wisc.edu/~remzi/OSTEP/vm-paging.pdf

Jan Reineke
Universität des Saarlandes

Review: Segmentation

Assumption: 14-bit virtual addresses, 2 most-significant bits determine segment

Segments:
0 => Code
1 => Heap
2 => Stack



Where does the segment table live?
→ Registers in MMU for active process
→ PCB otherwise

Segment	Base	Bounds
0	0x4000	0xfff
1	0x5800	0xfff
2	0x6800	0x7ff

Review: Memory accesses

```
0x0010:  movl 0x1100, %edi
0x0013:  addl $0x3, %edi
0x0019:  movl %edi, 0x1100
```

Seg	Base	Bounds
0	0x4000	0xfff
1	0x5800	0xfff
2	0x6800	0x7ff

Physical Memory Accesses?

1) Instruction Fetch, virtual address 0x0010
   - Physical addr.: 0x4010

   Load from virtual address 0x1100
   - Physical addr.: 0x5900

2) Instruction Fetch from virtual address 0x0013
   - Physical addr.: 0x4013

3) Instruction Fetch from virtual address 0x0019
   - Physical addr.: 0x4019

   Store to virtual address 0x1100
   - Physical addr.: 0x5900

Total of 5 memory accesses (3 instruction fetches, 2 load/stores).

Problem: External fragmentation

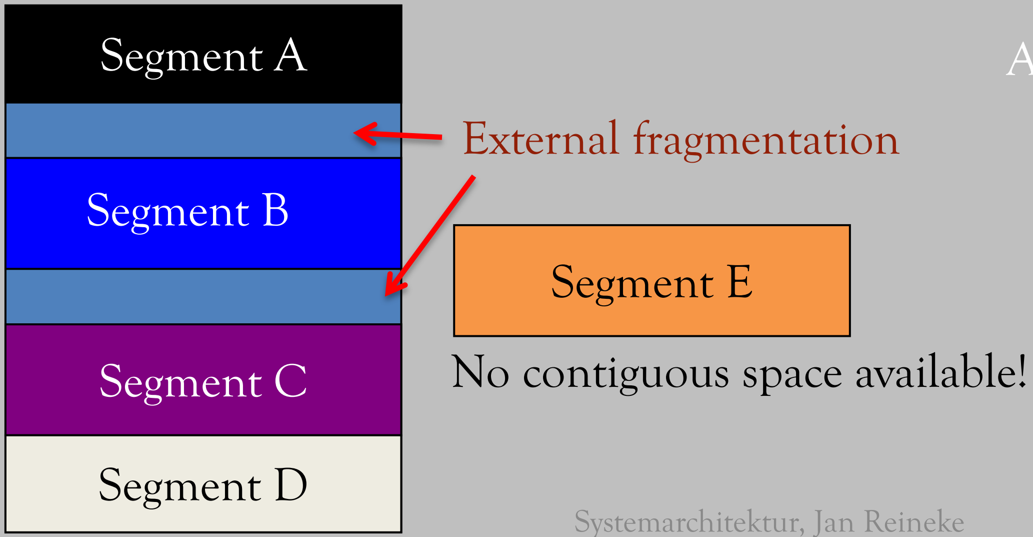
Definition (**Fragmentation**): Free memory that cannot be usefully allocated

Why?

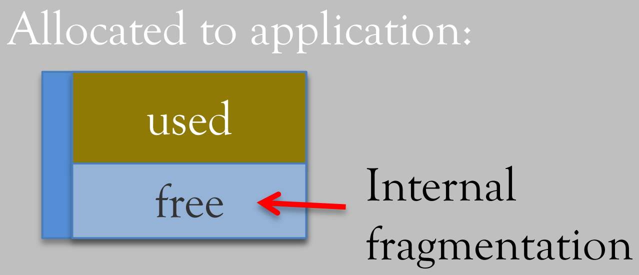
- Free memory (hole) is too small and scattered
- Rules for allocating memory prohibit using this free space

Types of fragmentation:

- **External**: visible to OS
- **Internal**: visible to application (e.g. if must allocate at some granularity)



No contiguous space available!



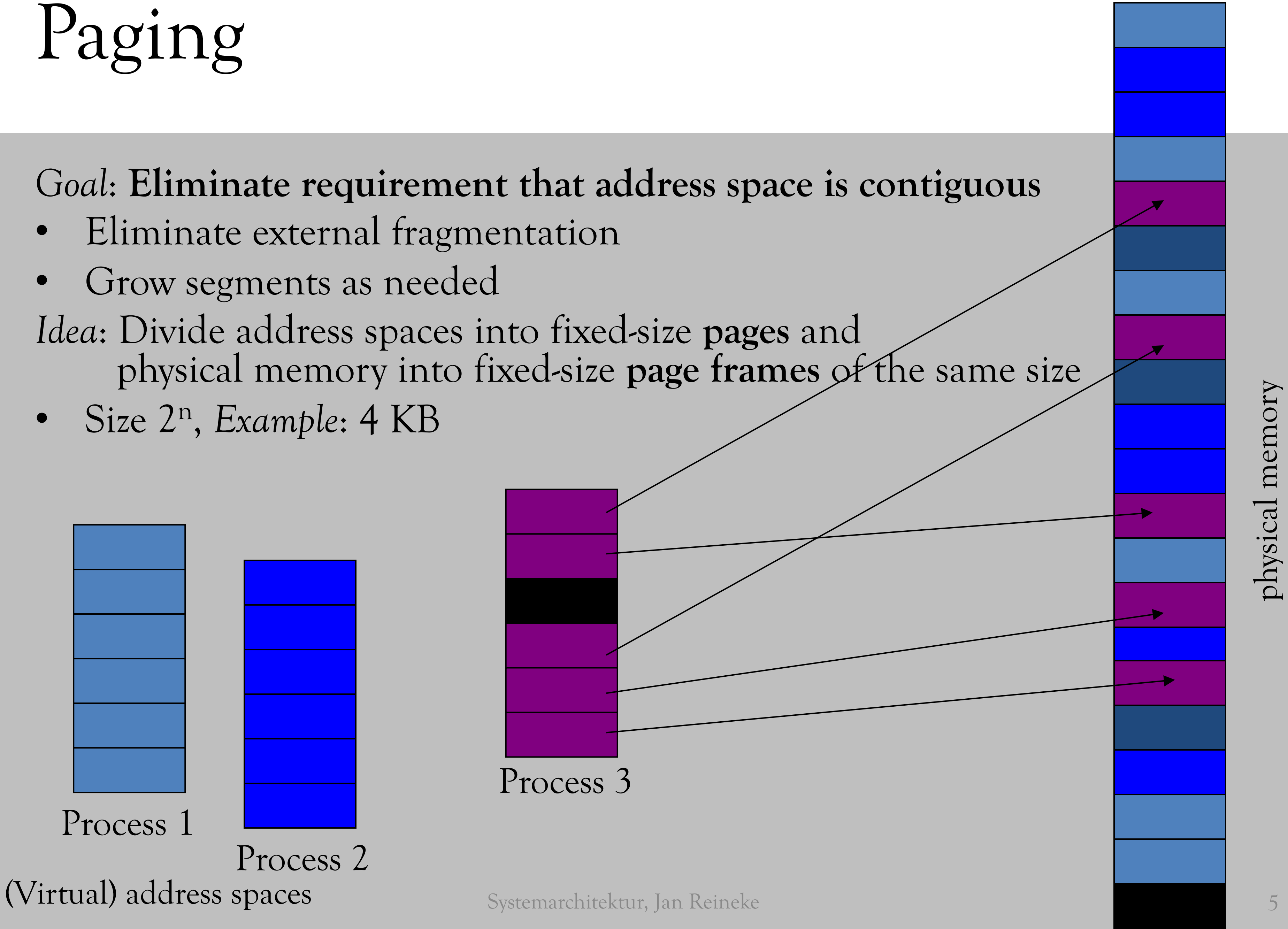
Paging

Goal: **Eliminate requirement that address space is contiguous**

- Eliminate external fragmentation
- Grow segments as needed

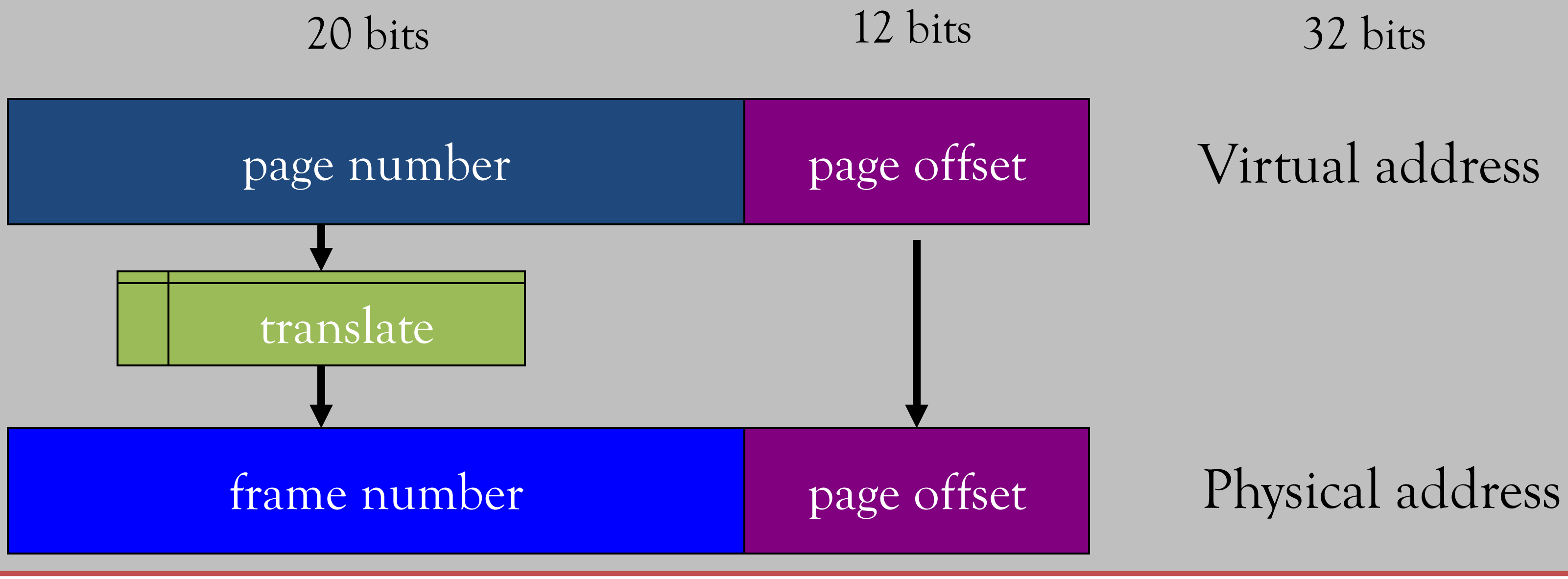
Idea: Divide address spaces into fixed-size **pages** and physical memory into fixed-size **page frames** of the same size

- Size 2^n, *Example*: 4 KB



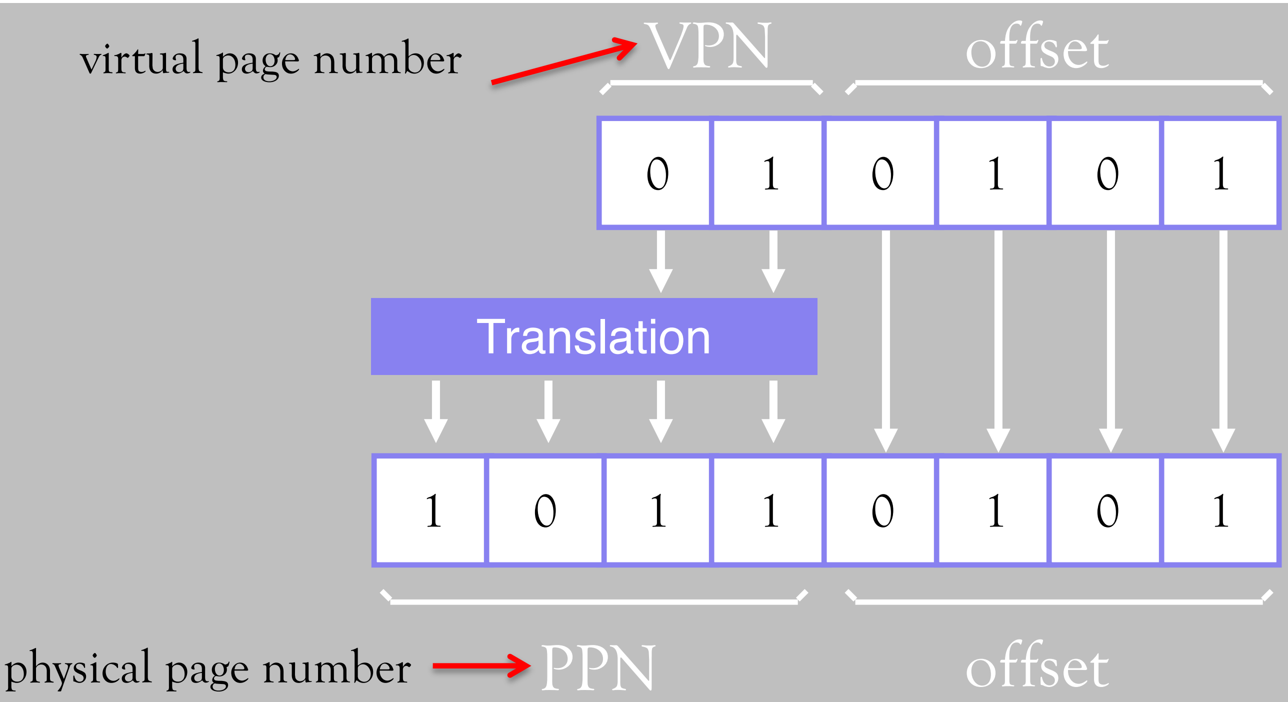
Address translation

- How to translate virtual addresses into physical addresses?
  - Most-significant bits determine page number
  - Least-signifiant bits determine offset within page



How does format of address space
determine number of pages and size of pages?

Address translation

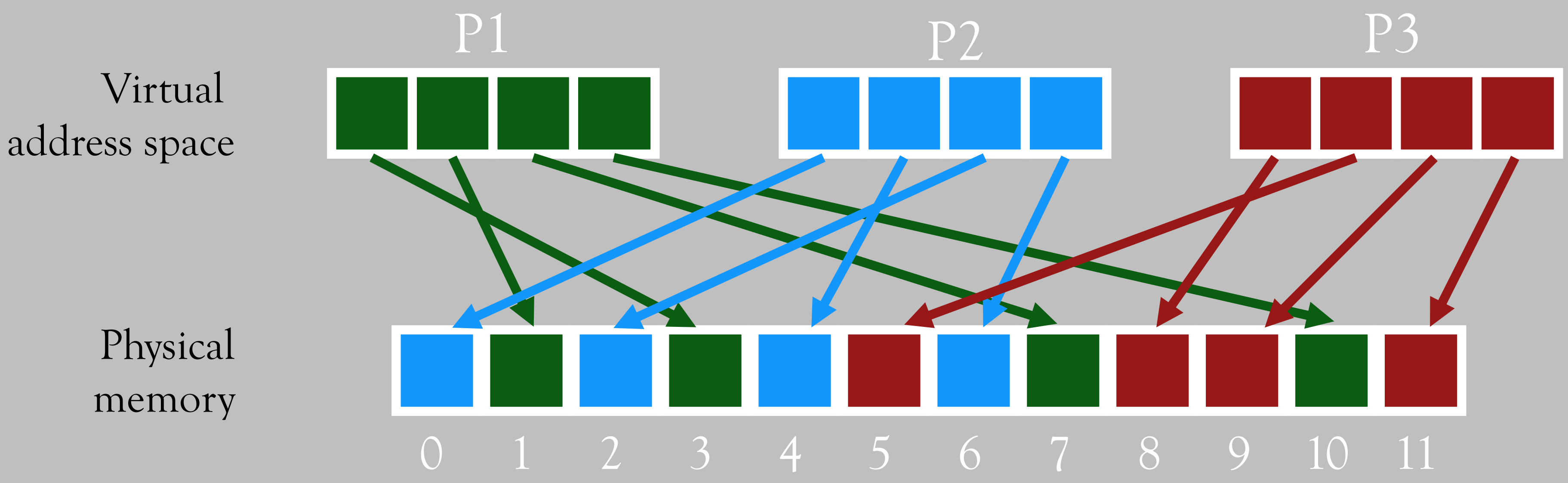


How should OS translate VPN to PPN?

What data structure is good?

Big array:
page table

Quiz: Page table



Page tables:

P1	P2	P3
3	0	8
1	4	5
7	2	9
10	6	11

Where are page tables stored?

How big is a typical page table?

- 32-bit address space
- 4 KB pages
- 32 Bit = 4 byte entries

Solution: $2 \wedge (32 - \log_2 (4*1024))*4$ Byte = **4 MB**

- Size of page table = Number of entries * Size of individual entries
- Number of entries = Number of pages = 2^(bits for page number)
- Bits for page number = 32 – bits for offset = 32 – $\log_2(4096)$ = 32-12 = 20
- Thus, 2^{20} 4-byte entries

Where are page tables stored?

Implication:
Store page table in memory, **not** in registers.
Hardware finds page table with page table base register.

What happens on a context switch:

- Save old page table base register in PCB of descheduled process
- Change contents of page table base register to newly scheduled process

Other page table info

What other info is in page table entries besides translation?

- `valid bit` = is the page allocated?
- `protection bits`, encode access rights
- `present bit`
- `reference bit`
- `dirty bit`

discussed later on

Memory accesses with paging

```
0x0010: movl 0x1100, %edi
0x0013: addl $0x3, %edi
0x0019: movl %edi, 0x1100
```

Page table is at phys. addr. 0x5000
Every entry is 4 bytes
4KB pages, i.e., 12 bits for offset

First 4 entries of the page table

2
0
80
99

Physical memory accesses with paging?

1) Instruction Fetch at virtual address 0x0010:

- VPN = 0, so access entry 0 of page table
- Access 1: 0x5000 to learn: PPN = 2
- Access 2: Instruction fetch at phys. addr. 0x2010

Load at virtual address 0x1100:

- VPN = 1, so access entry 1 of page table
- Access 3: 0x5004 to learn: PPN = 0
- Access 4: Load at phys. addr. 0x0100

Doubling number of memory accesses!

Advantages of paging

No external fragmentation:

Any page can be placed in any frame in physical memory

Fast to allocate and free:

Linked list of free pages

Simple to swap-out portions of memory to disk (later lecture)

- Page size = disk block size
- Can run process when some pages are on disk
  - Add `present bit` to page table entries

Disadvantages of paging

Internal fragmentation:

Wasted memory grows with larger pages

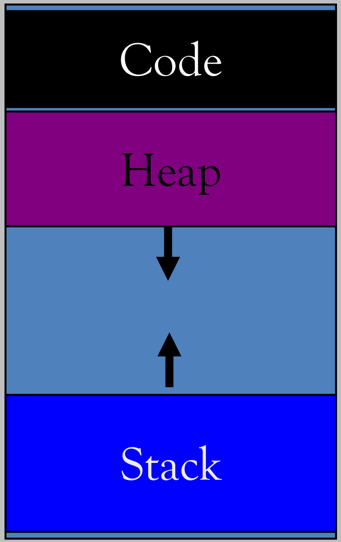
Additional memory access to page table upon every memory access

→ **extremely inefficient**

- Page table must be stored in memory
- MMU stores only base address of page table
- Solution: TLB = "page table cache"

Substantial storage for page tables

- Simple page table: one entry for each page in address space, even if not allocated


Code
Heap
Stack