INF113: Multi-level Page Tables and Swap

Kirill Simonov

10.10.2025

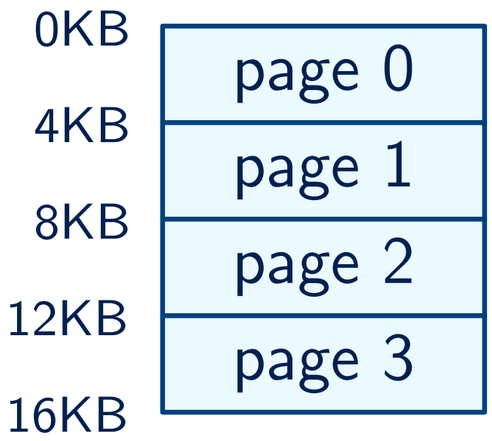
Mandatory Assignment 2

- Starts today afternoon
- Deadline in two weeks, Friday Oct 23
- You will experiment with memory management
- In case of issues, ask on Discord or attend group session
- Rule clarification: 10% of final grade, but no pass limit
- New: declaring usage of LLMs required, following the faculty policy

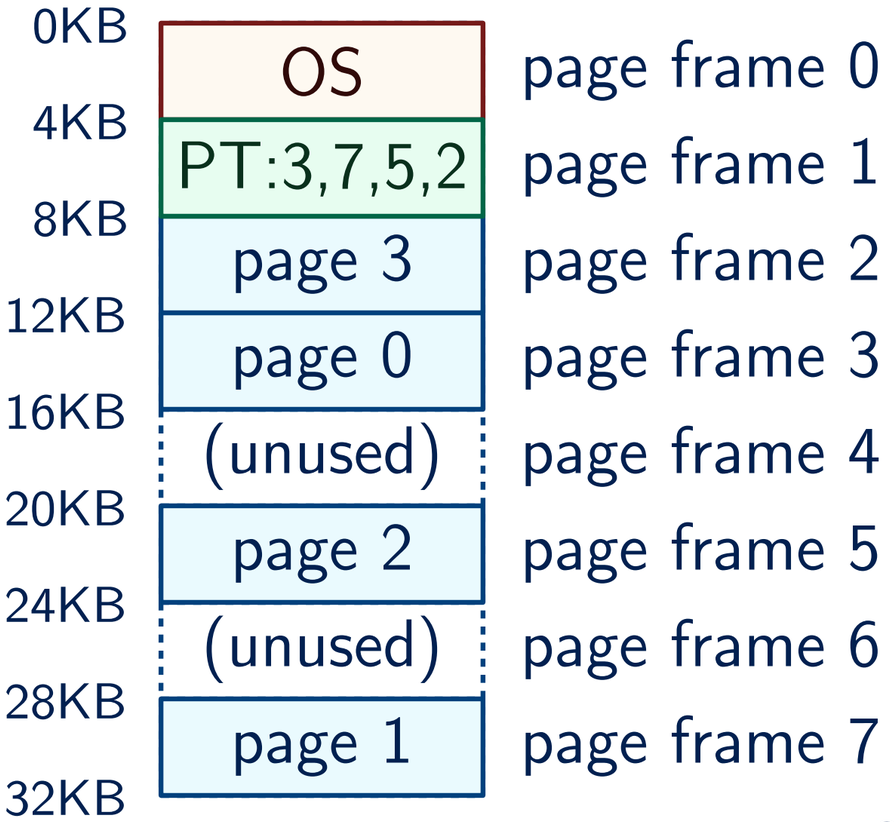
Paging: Reminder

- Address space is partitioned into 4KB pages, and memory into 4KB frames
  Pages are arbitrarily mapped into frames
- Mapping is stored in the memory as the page table
- Issues:
  - Extra memory accesses—solved by TLB ✓
  - Page tables take lots of memory ?

process address space

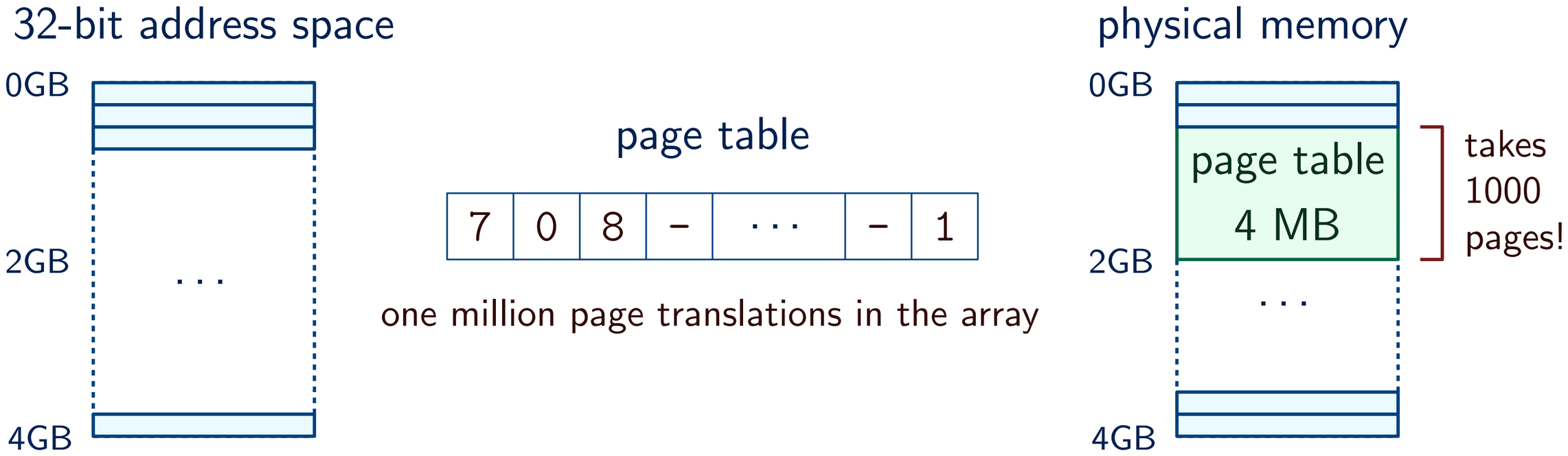


physical memory



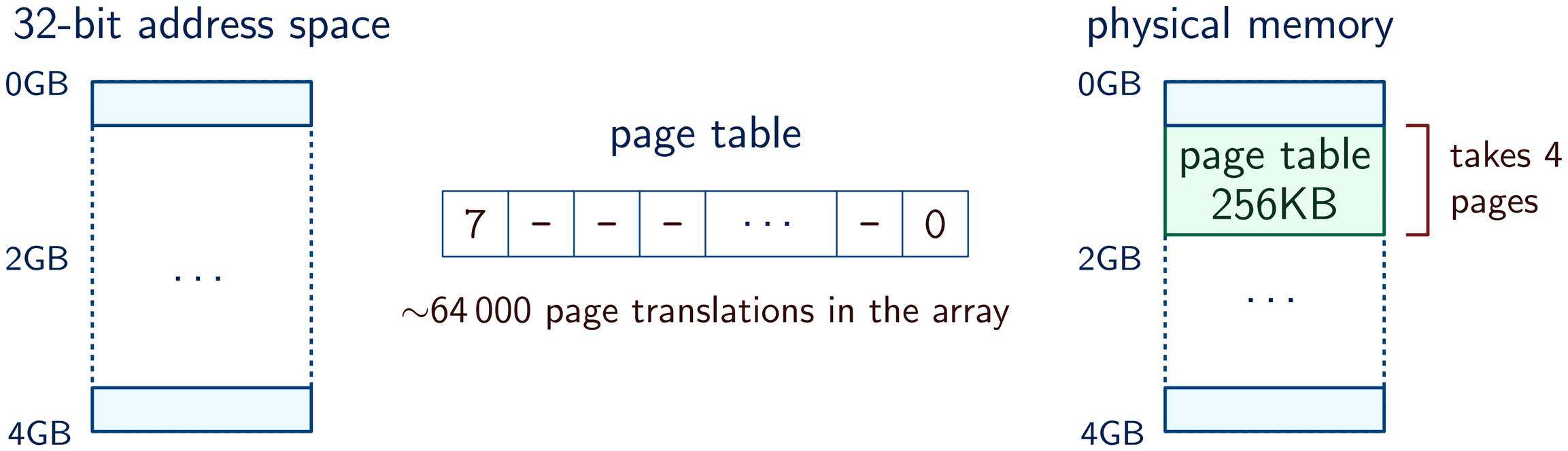
Linear page tables in memory

- **Issue 1: size**
  4GB memory, 4KB pages, 4B PTE, 100 processes $\mapsto$ 400MB in page tables
- **Issue 2: allocating memory for a page table**
  - Single page table takes 4MB, has to be consecutive in memory
  - Page table itself is not paged—bad for memory management



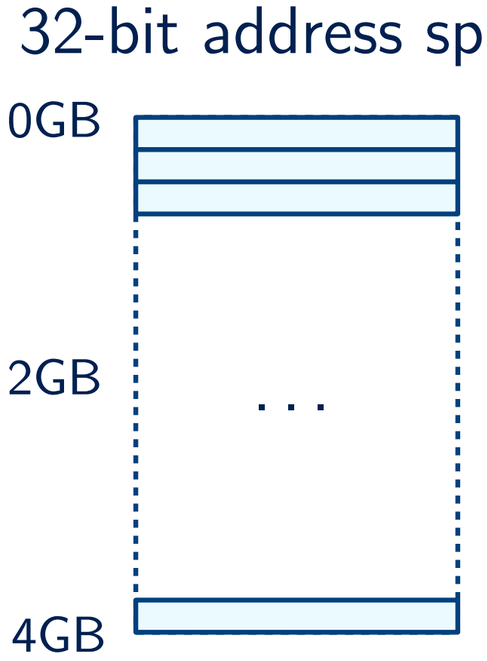
Attempt 1: Larger pages

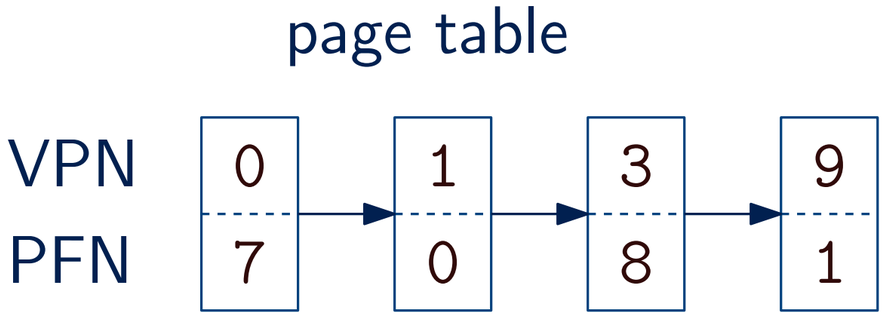
- Let's make the page 16x larger: 64KB
  - 4MB/16 = 256KB for a single page table = 4 pages, 25MB for all PTs
- Also increases fragmentation...
  - Now every memory allocation potentially wastes 64KB
  - 4 pages per process, 100 processes = 25MB



Attempt 2: Data structures

- We know lots of pages may be empty—maybe use a sparser DS?
  - Linked list?
  - Binary search tree?





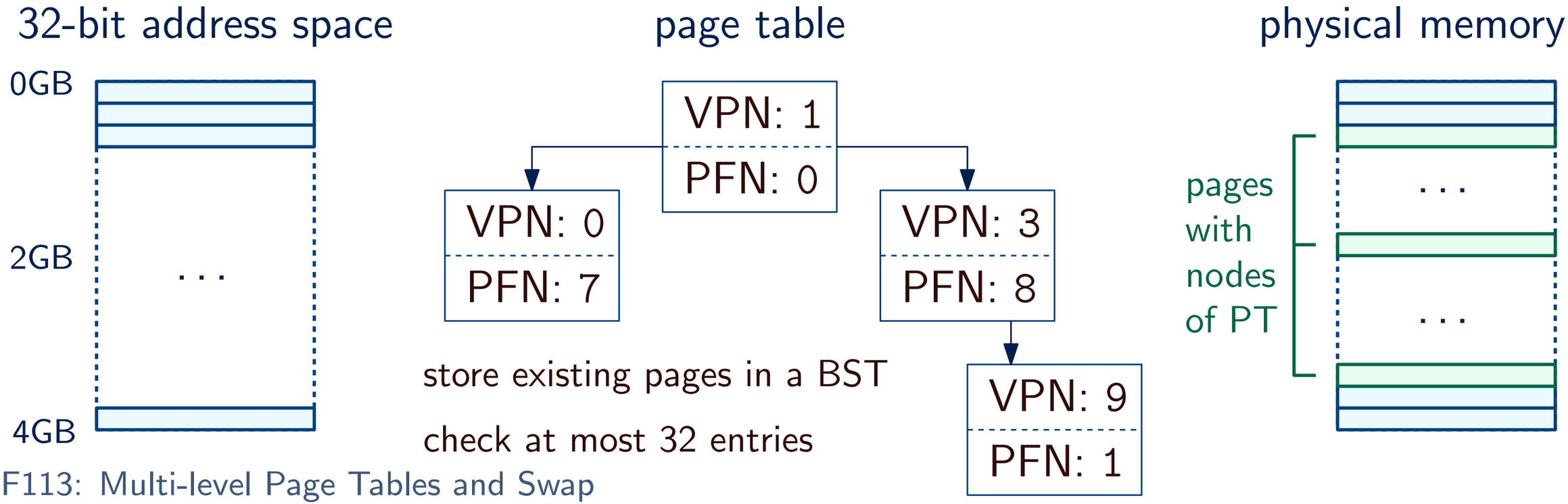
linked list only stores existing pages

finding PFN requires cycling through all entries!

Attempt 2: Data structures

- We know lots of pages may be empty—maybe use a sparser DS?
  - Linked list?
  - Binary search tree?

- Fixes space issues, but overhead for a memory access is terrible



Multi-level page table

- Let's split the page table into page-sized chunks!

do not store
empty PT pages

store PT page
info in PD

PT	PFN	valid
0	3	1
1	–	0
2	–	0
3	6	1

page directory

VPN	PFN	valid
0	4	1
1	1	1

PT page 0

2	–	0
3	–	0

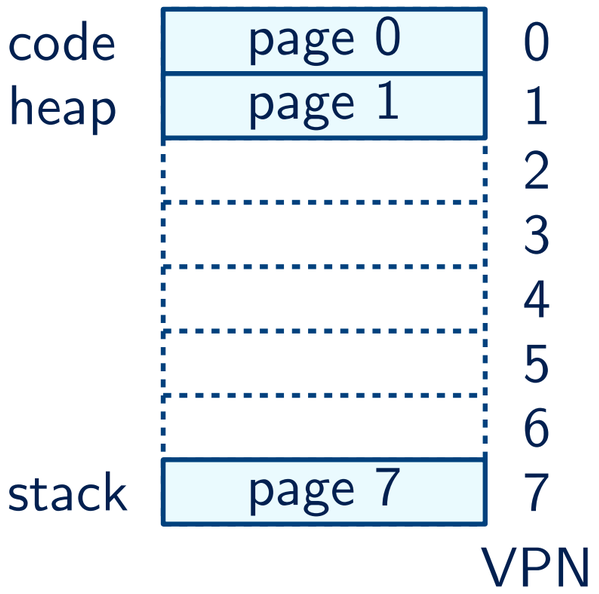
PT page 1

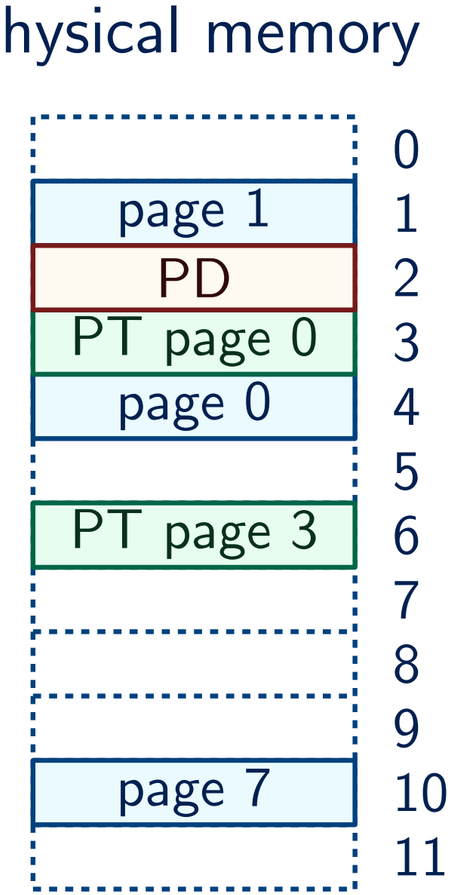
4	–	0
5	–	0

PT page 2

6	–	0
7	10	1

PT page 3





Addess translation: MLPT

- Compute page index in PT and fetch its location

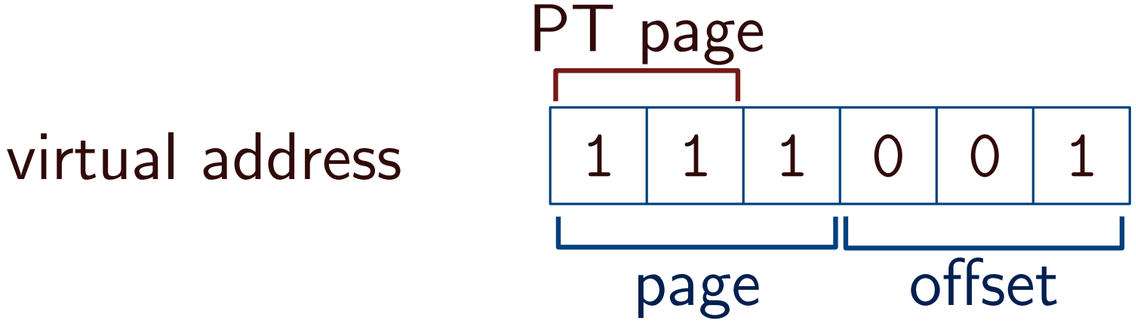
```
PD_index = virt_addr & 0x30
PT = PD[PD_index]
```

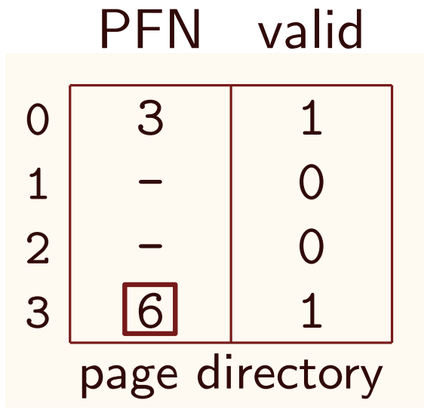
- Find target PFN by index within PT page

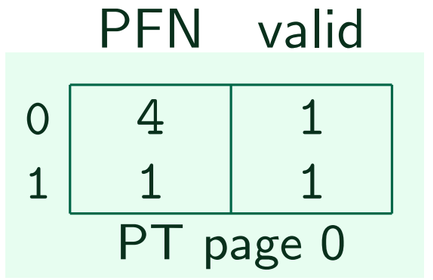
```
PT_index = virt_addr & 0x08
PFN = PT[PT_index]
```

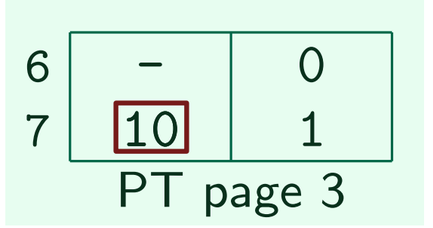
- Add offset to obtain final address

```
offset = virt_addr & 0x07
phys_addr = (PFN << 3) | offset
```

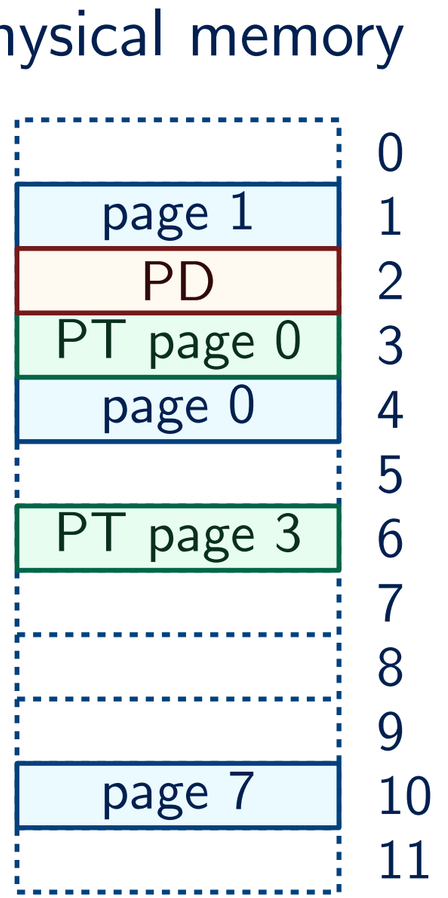






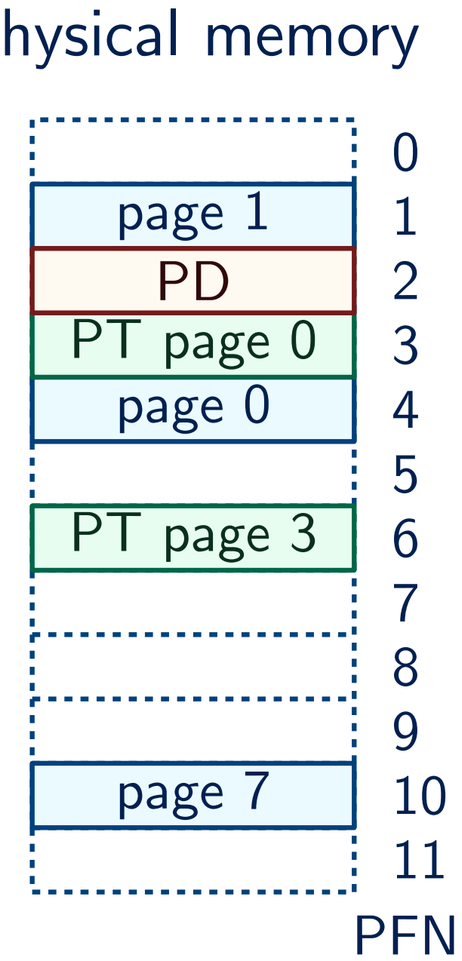


physical address is 1010001



Benefits of MLPT

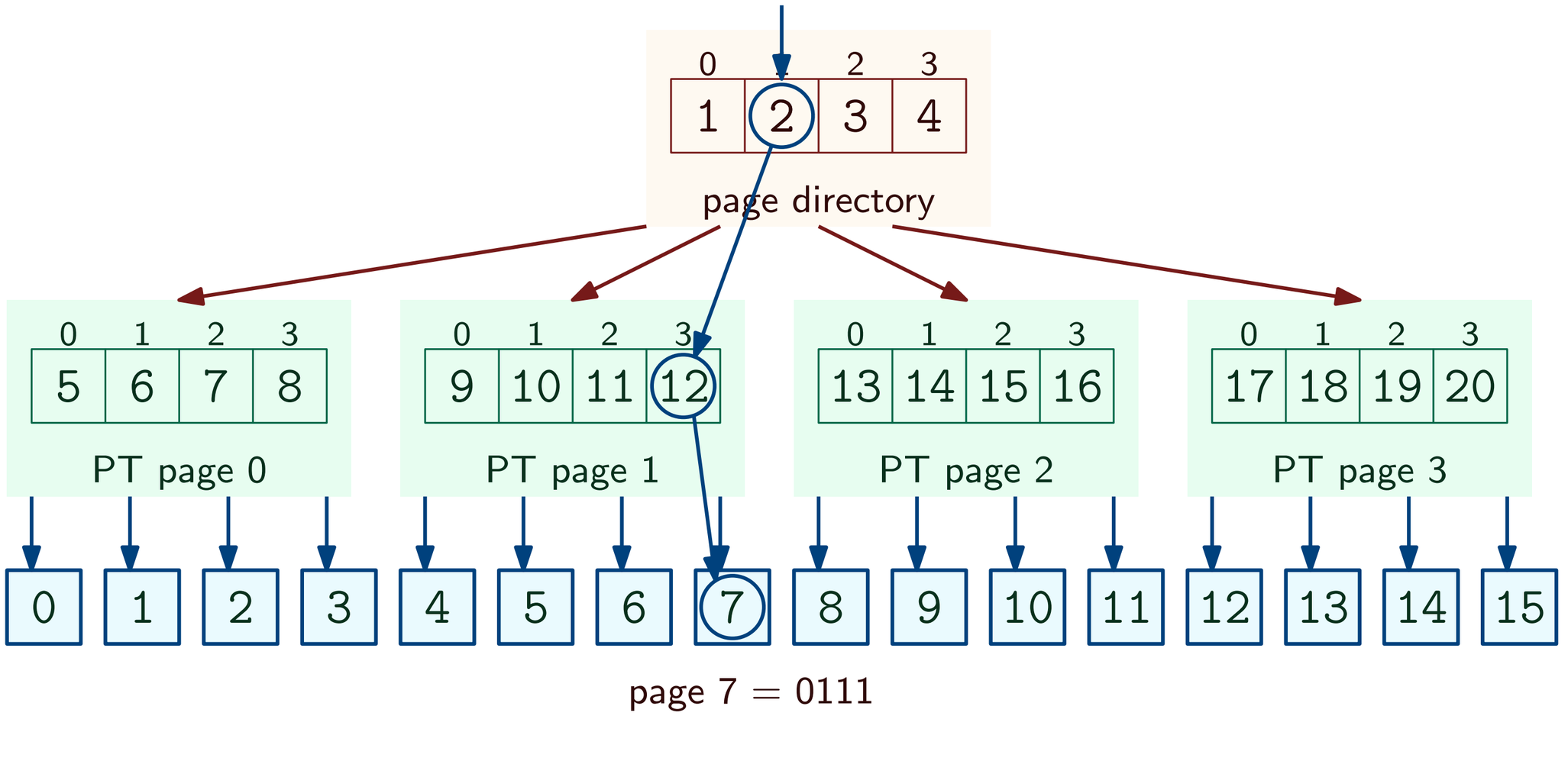
- Size of page table scales with used memory pages
  - Page table size is 4KB (PD) + 4KB for each used PT page
  - Each PT page handles a thousand virtual pages
- Page table is also stored in pages
  - PD and individual PT pages can be stored at any frame
  - PD keeps references to where each individual PT page is stored
- Issue: more memory reads
  - Each memory access turns into three
  - However, TLB still helps
- Issue: page directory size
  - PD might not always fit in a single page
  - Example: 32-bit addresses, 4KB page, 4B per PTE/PDE
    20-bit page index, 1024 PTE in a PT page, 1024 PT pages described in PD



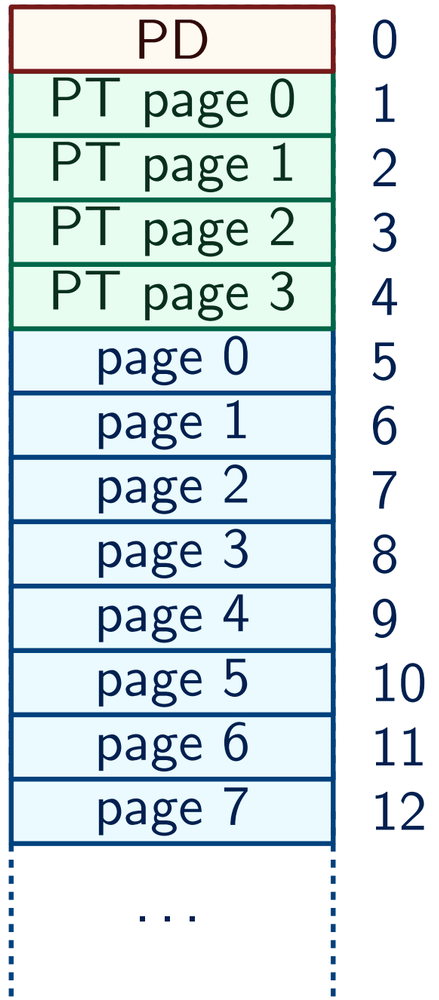
more memory/larger PTE size/smaller page → overflow!

MLPT as a tree

- Two-level page table is a search tree of depth 2

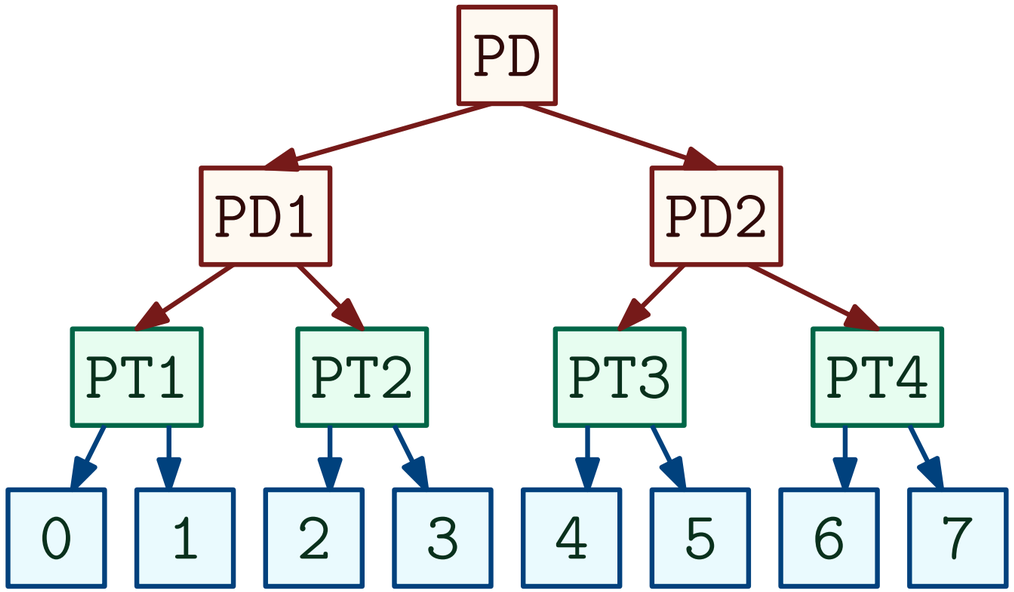


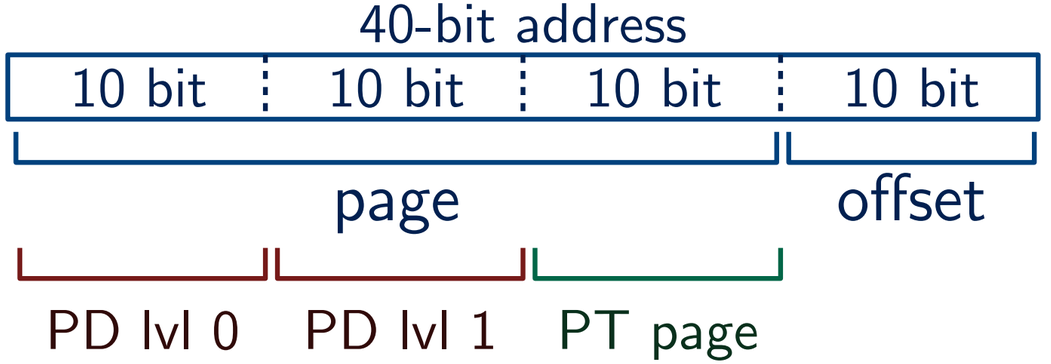
physical memory



More levels

- MLPT may also have, e.g., 3 levels

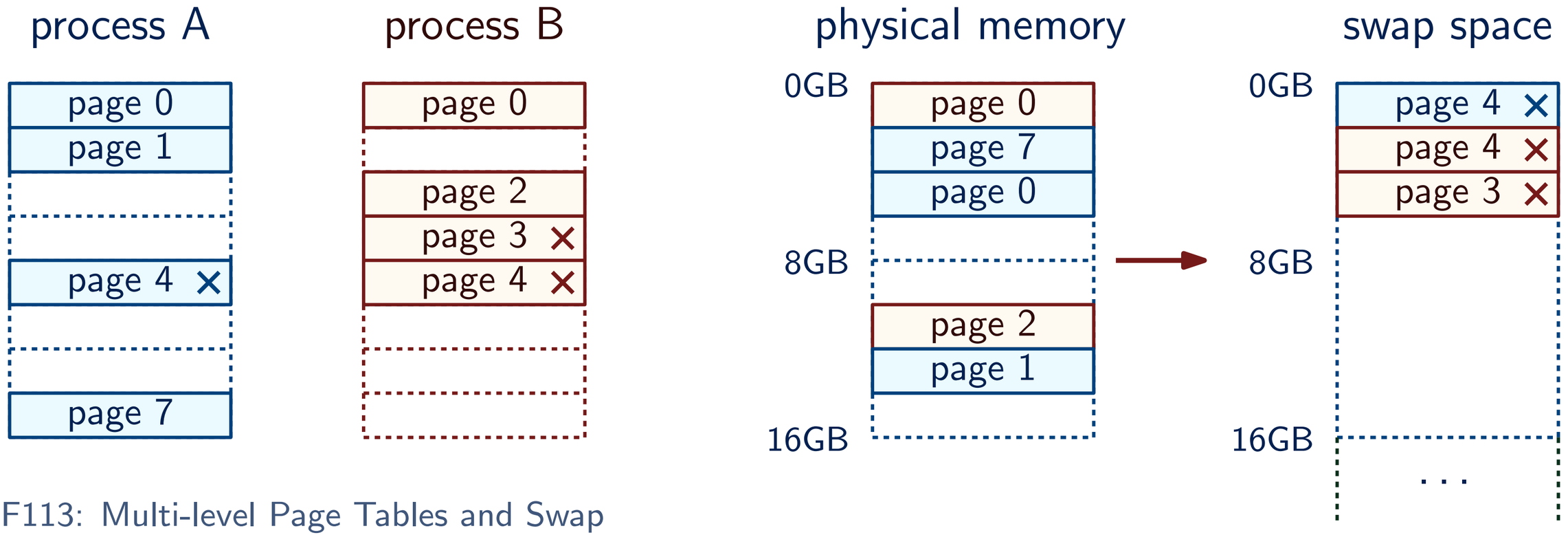




- Handles larger memory space and/or smaller page size
- Downside: one extra memory access per level

Beyond physical memory

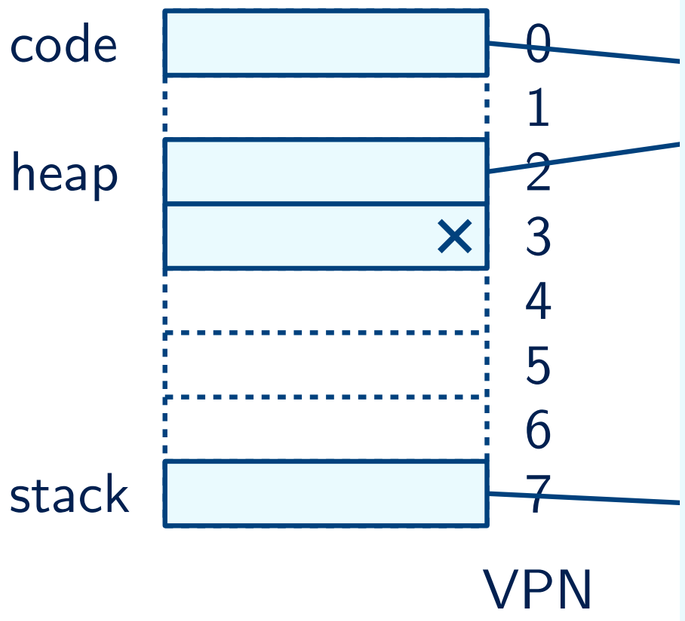
- It may be too generous to keep all pages of all processes in memory at all times
  - Some processes may be sleeping for a while
  - Some pages may be almost never in use
- **Swap space**: portion of the disk where memory pages can be stored
  - With paging, we can swap any page out, without affecting the rest



Present bit

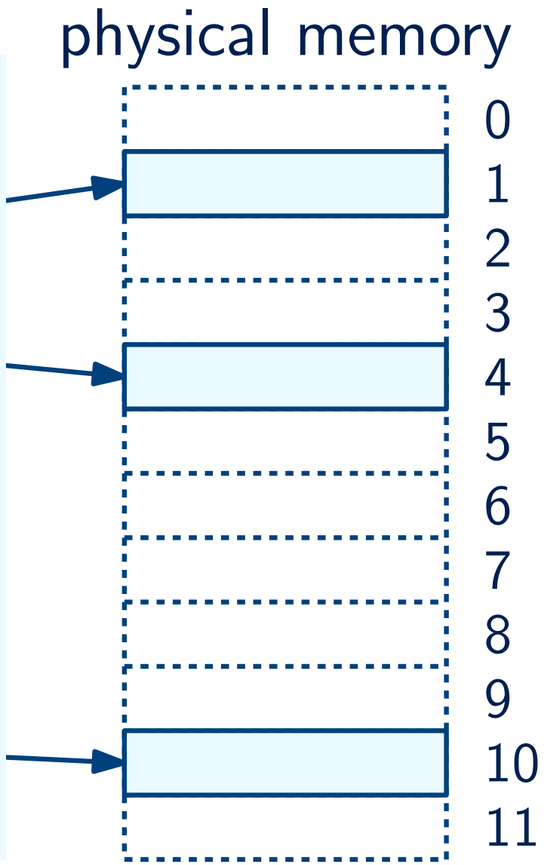
- **Present bit** in the page table marks whether the page is in memory
- Access to an invalid page—crash
- Access to a not present page—call OS to load the page from disk

reading from disk is slow enough to allow context switch!



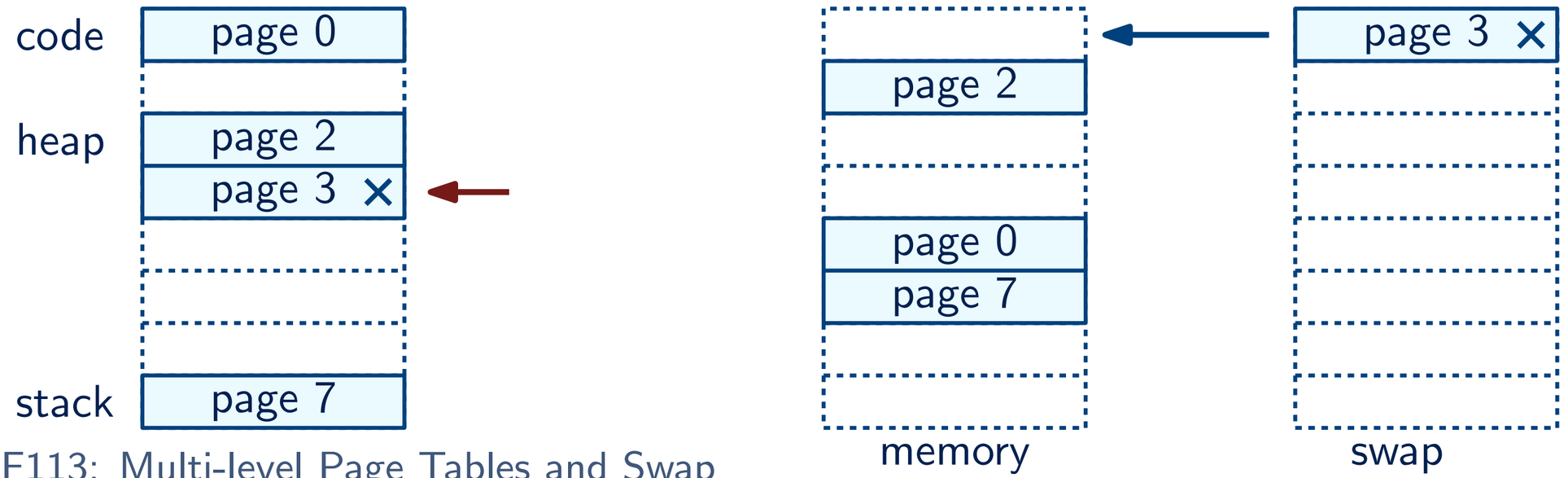
VPN	PFN	valid	pr.	prot.
0	4	1	1	r-x
1	-	0	0	---
2	1	1	1	rw-
3	-	1	0	rw-
4	-	0	0	---
5	-	0	0	---
6	-	0	0	---
7	10	1	1	rw-

page table



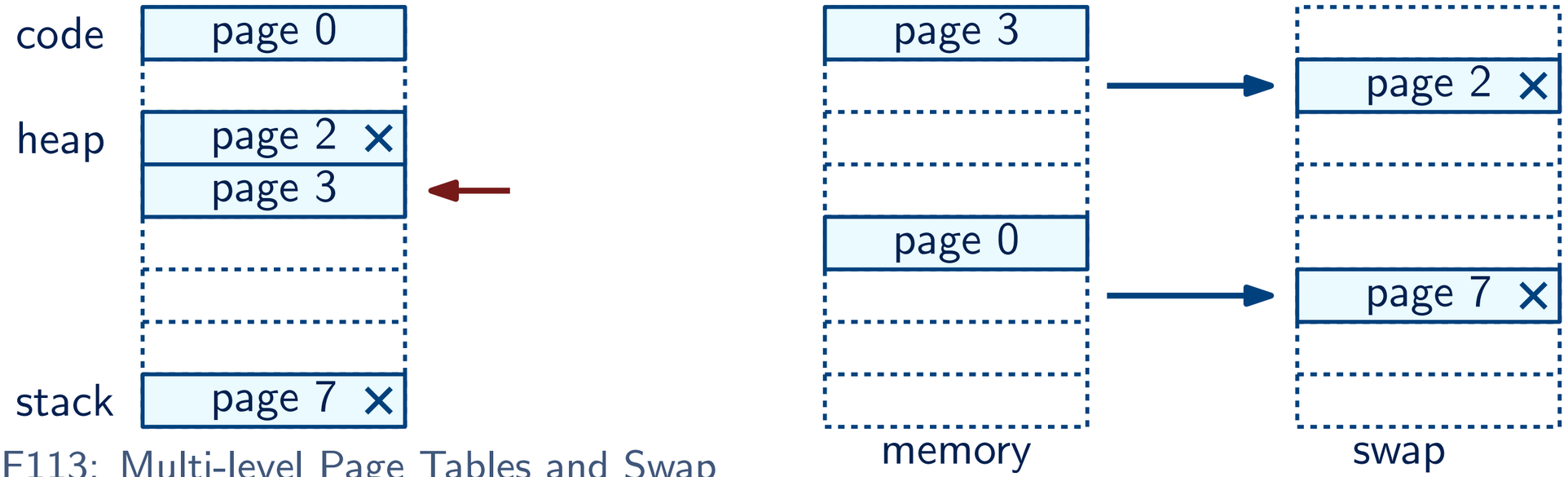
Between memory and swap

- A page is moved from swap upon access
  - Sometimes also in advance, e.g., the next code page



Between memory and swap

- A page is moved from swap upon access
  - Sometimes also in advance, e.g., the next code page
- A frame must be free for the page to be brought back
- OS will try to keep some memory free proactively
  - If less than `low` frames are free, OS removes pages to reach `high`
  - Moving pages to disk in bulk saves time—buffering



Swapping policies

How to decide which page to swap?

- With good policy OS multiplicates memory size for free
- With bad policy every memory access can turn into a disk access...

memory 100 ns SSD 0.1 ms HDD 10 ms

- Can see memory as a **cache** of pages
  - Fast but small memory
  - Large but slow disk
  - Want to maximize the number of hits

Example:
Memory vs HDD, 10% miss rate
Average access= $100\text{ns} + 0.1 \cdot 10\text{ms} = 1.0001ms$
Memory vs HDD, 0.1% miss rate
Average access= $100\text{ns} + 0.001 \cdot 10\text{ms} = 10.1\mu s$

- Same considerations as for other caches: TLB, L1/L2, ...
- Swap specialty: misses are very costly!

Example policies

- **Optimal:** swap page that is furthest in the future
- **Random:** swap a random page
- **FIFO:** swap the earliest page
- **LRU**: swap the least recently used page

query	cache	result
0	empty	miss
1	0	miss
2	0, 1	miss
0	0, 1, 2	hit
1	0, 1, 2	hit
3	0, 1, 2	miss
0	0, 1, 3	hit
3	0, 1, 3	hit
1	0, 1, 3	hit

optimal

$4/9 \sim 44.4\%$ miss rate

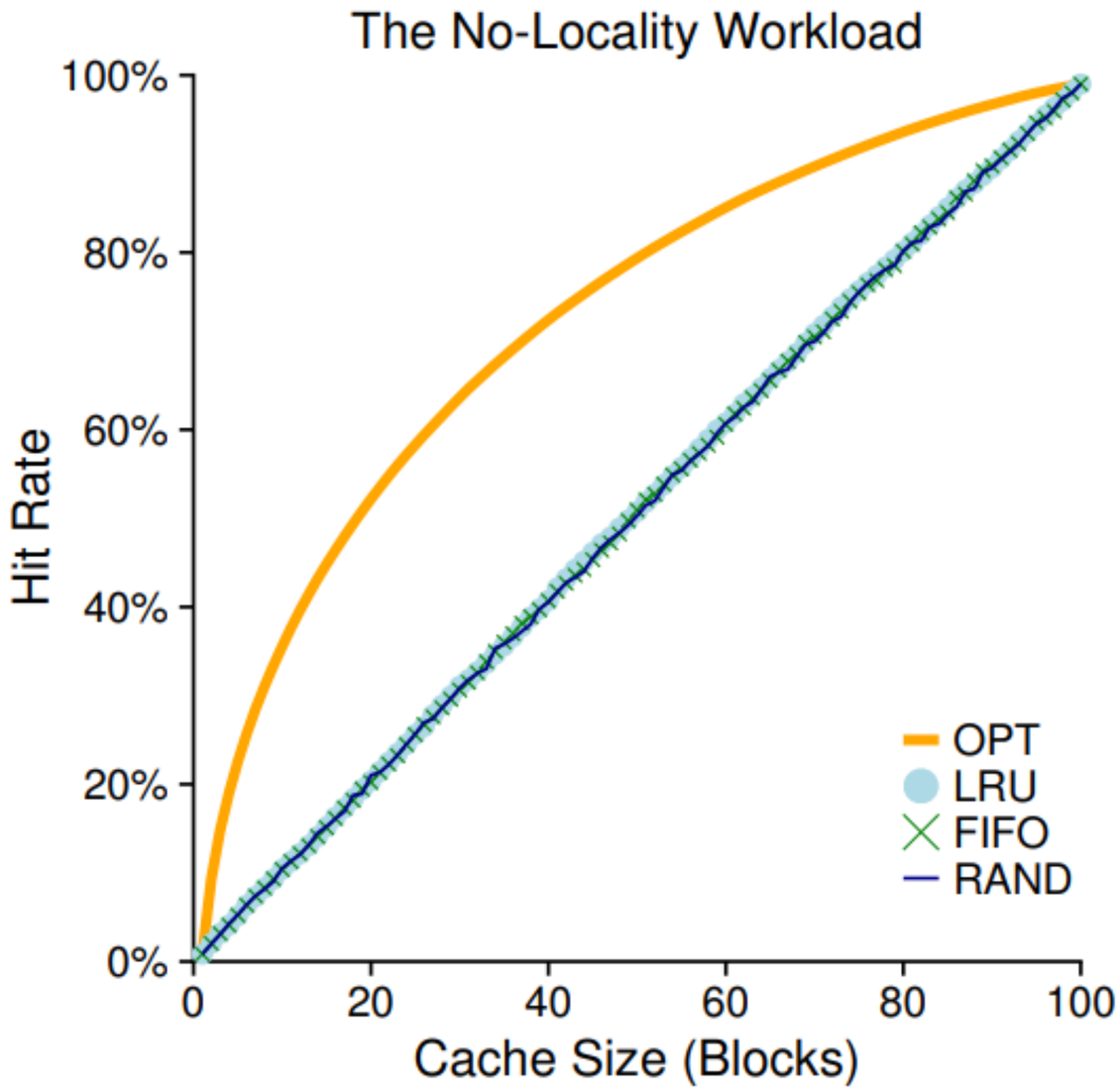
FIFO

$5/9 \sim 55.6\%$ miss rate

query	cache	result
0	empty	miss
1	0	miss
2	0, 1	miss
0	0, 1, 2	hit
1	0, 1, 2	hit
3	0, 1, 2	miss
→ 0	1, 2, 3	miss
3	1, 2, 3	hit
1	1, 2, 3	hit

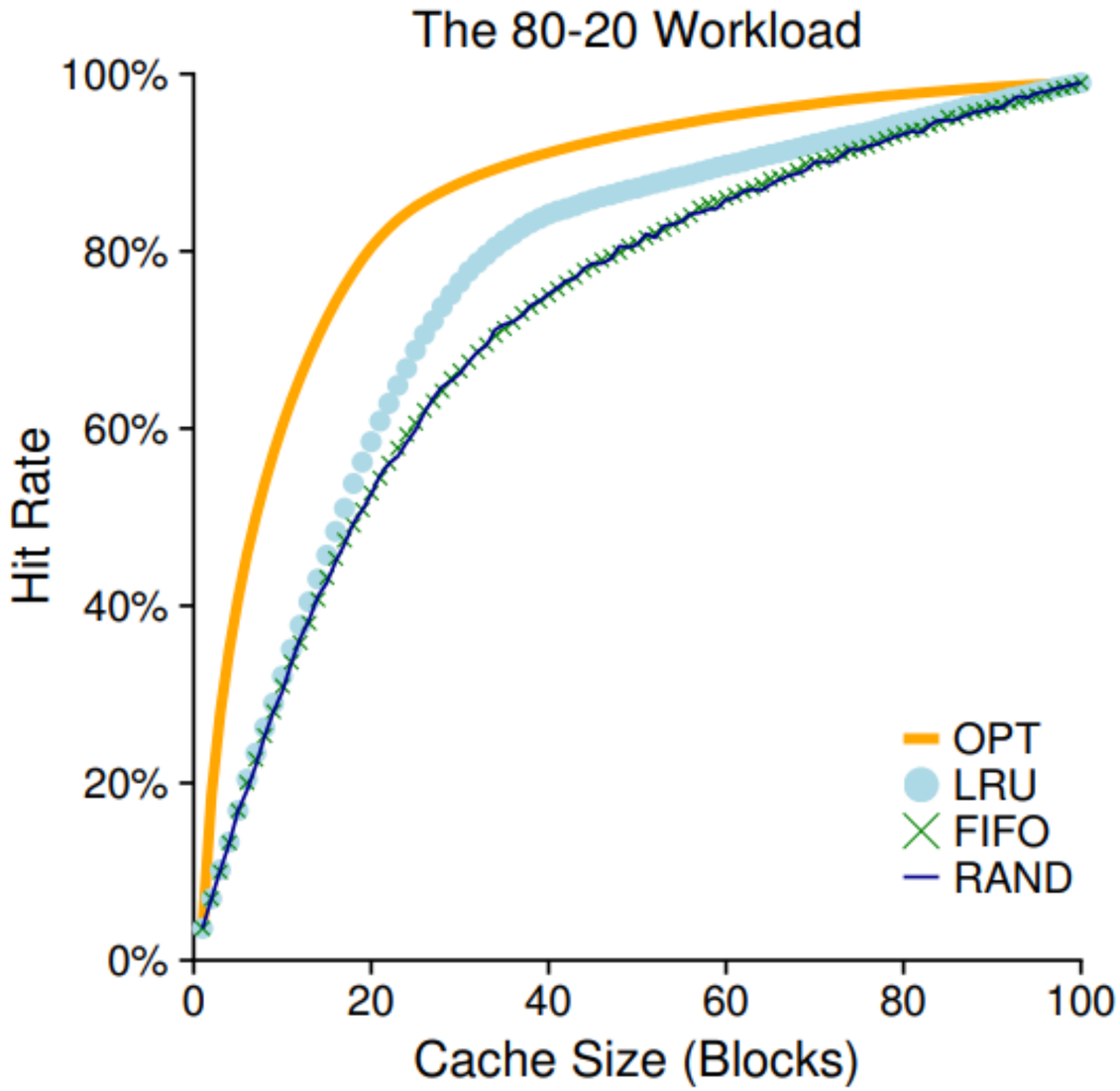
Policies vs scenarios

- **No-locality:** 100 pages, full random
  - Nothing helps, except knowing the future



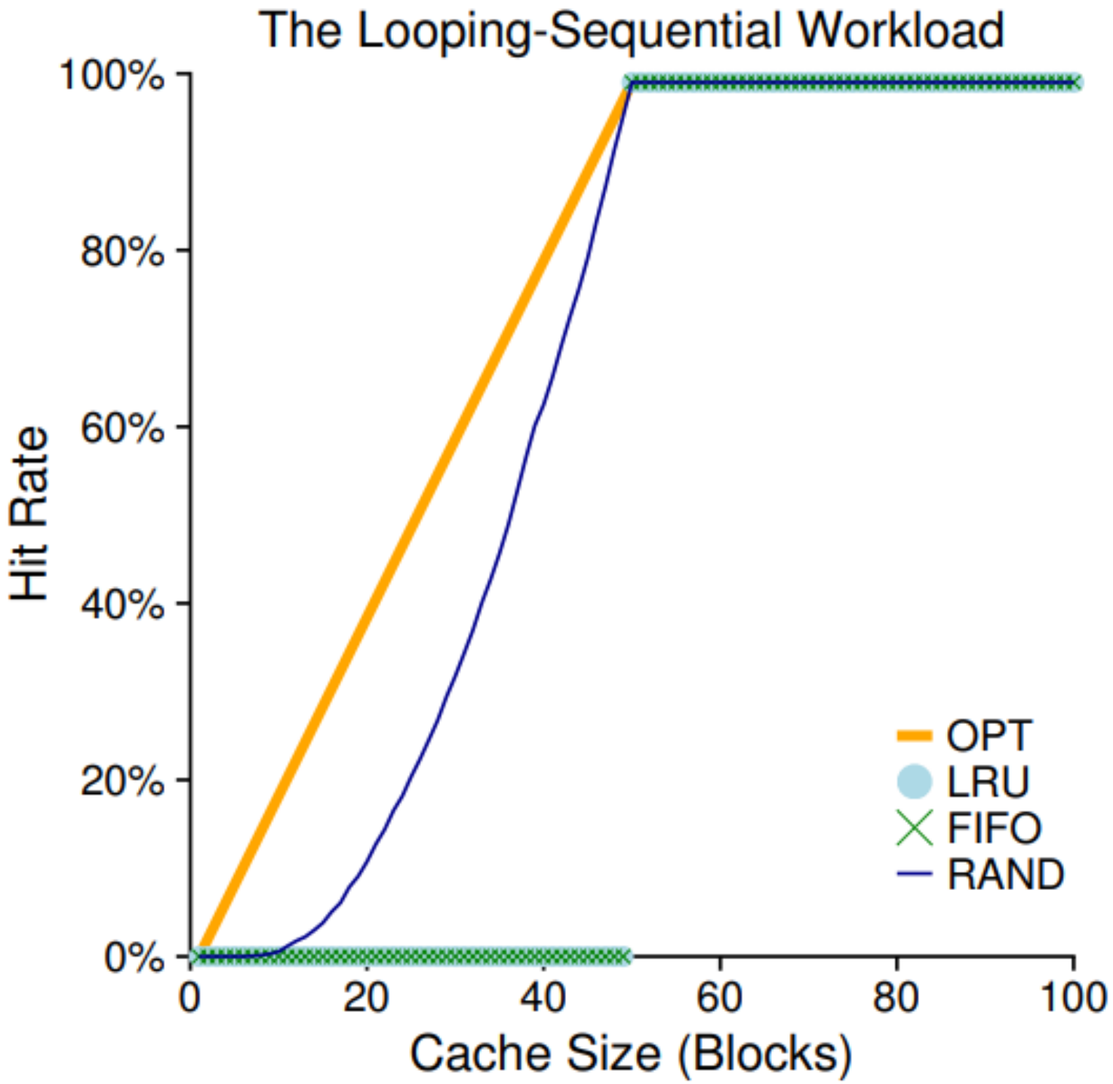
Policies vs scenarios

- **No-locality:** 100 pages, full random
  - Nothing helps, except knowing the future
- **80-20:** 20% of pages get 80% of queries
  - LRU predicts "heavy hitters"



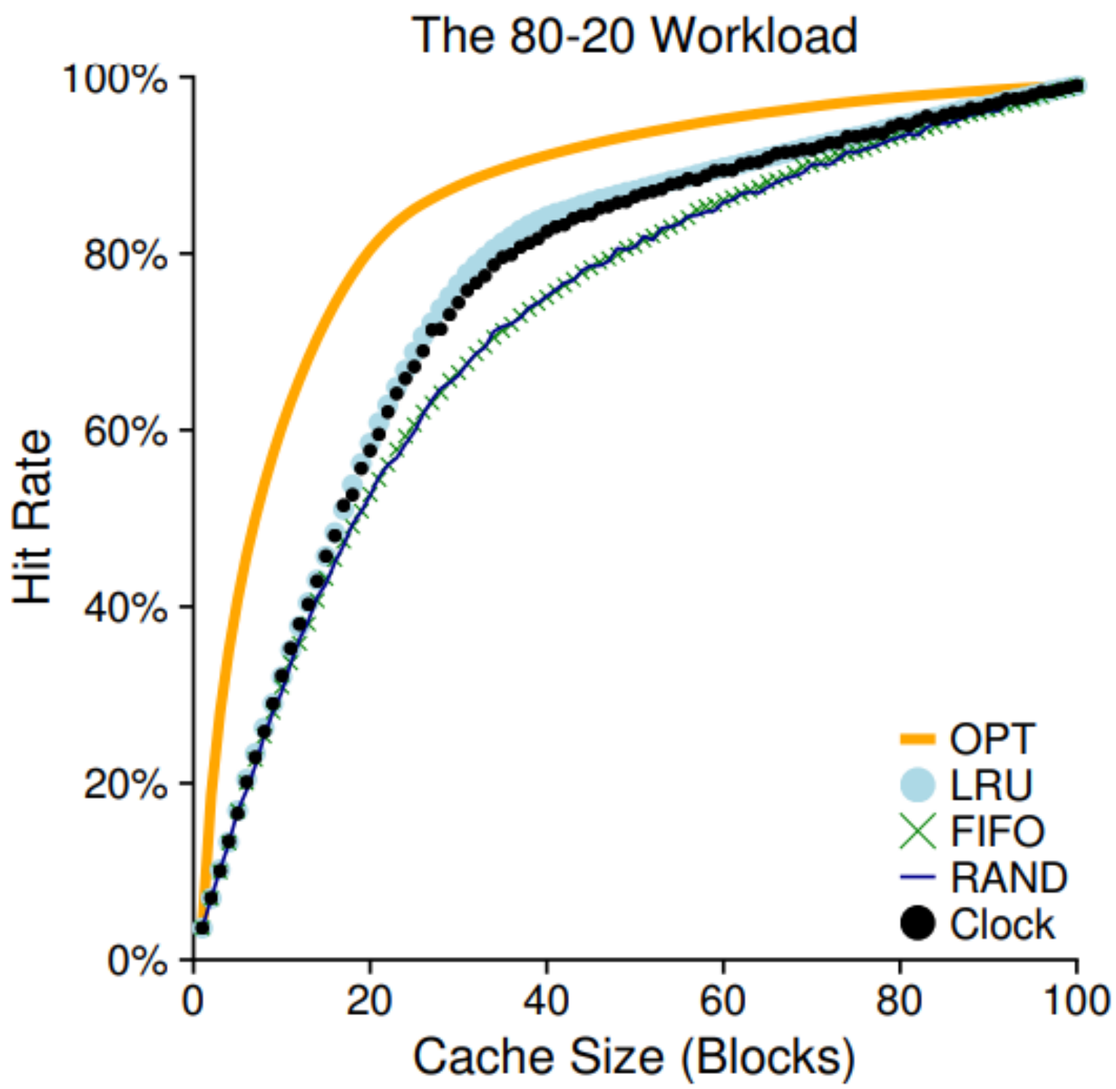
Policies vs scenarios

- **No-locality:** 100 pages, full random
  - Nothing helps, except knowing the future
- **80-20:** 20% of pages get 80% of queries
  - LRU predicts "heavy hitters"
- **Looping-sequential:** 50 pages accessed in the same sequence, over and over
  - Worst-case for LRU



Policies vs scenarios

- **No-locality:** 100 pages, full random
  - Nothing helps, except knowing the future
- **80-20:** 20% of pages get 80% of queries
  - LRU predicts "heavy hitters"
- **Looping-sequential:** 50 pages accessed in the same sequence, over and over
  - Worst-case for LRU
- **Clock policy:** approximates LRU by only keeping a single "accessed" bit
  - More realistic



Memory usage tools

- `pmap` shows the memory map of a process

```
pmap -x <PID>
```

- `free` displays memory usage

```
free -h
```

- `vmstat` allows to keep track of memory, swap, cpu, also in real-time

```
vmstat 1
```

- `swapon -s` to show swap devices

```
swapon -s
```

- Use `mem.c` from Chapter 21 Homework to experiment with these

Summary

- Multi-level page tables solve the space problem of page tables
- Levels provide trade-off between the size of table and extra memory accesses
- Paging makes it easy to swap unpopular pages to disk, saving space
- Good policy is needed to keep miss rates very low
- Modern OS would use some approximations to LRU
- Spread of SSDs shifts balance:
  misses are less costly $\rightarrow$ more headspace for policies
- Look at Chapters 20–22 for more details