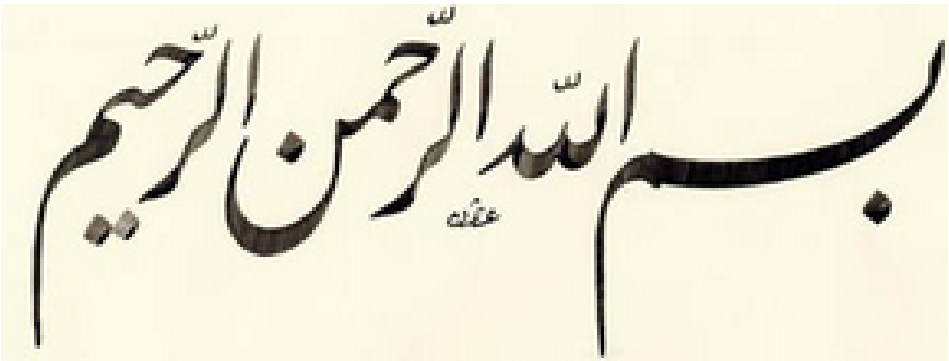

بسم الله الرحمن الرحيم

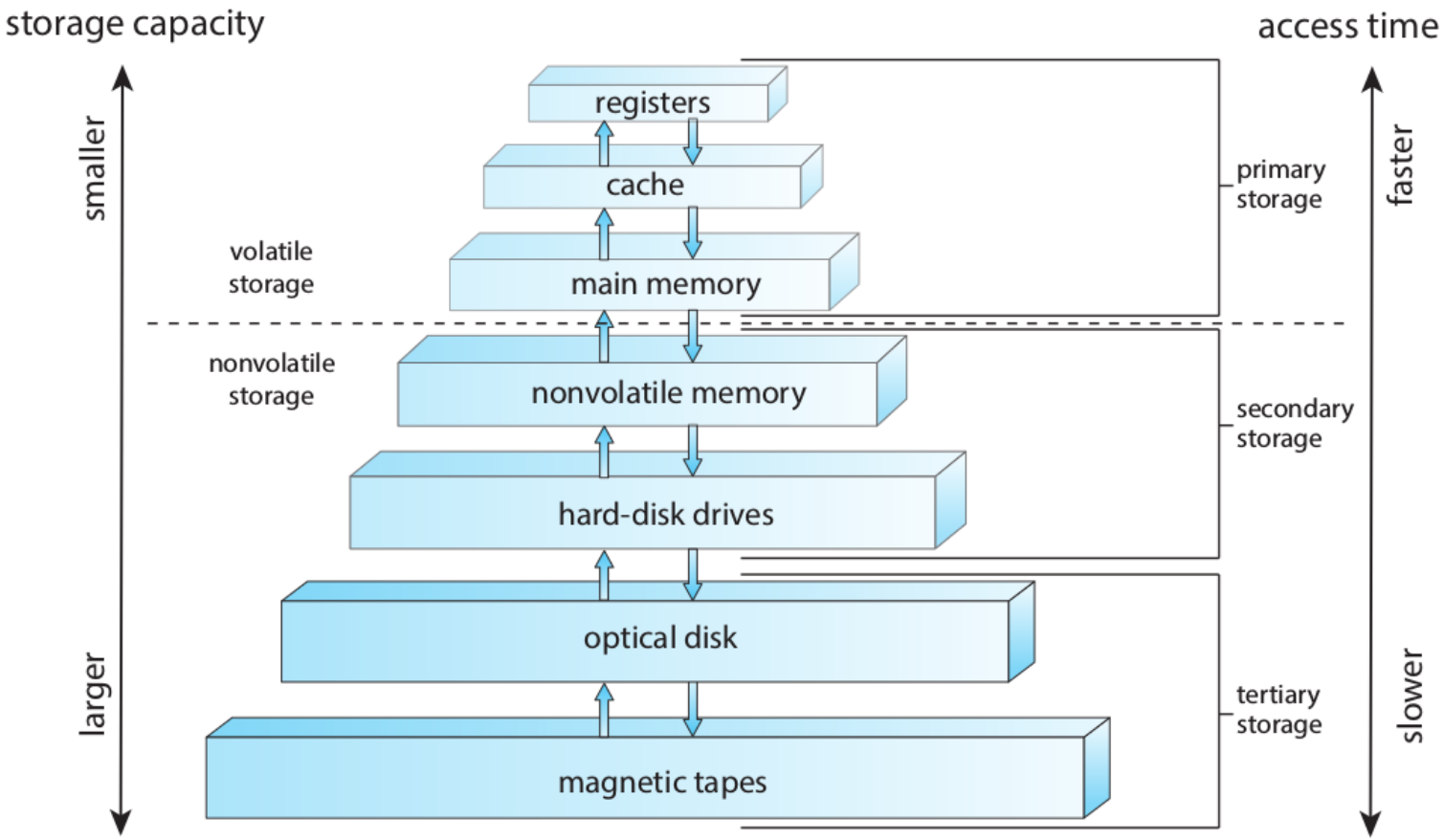
Piece 1

» Virtualization

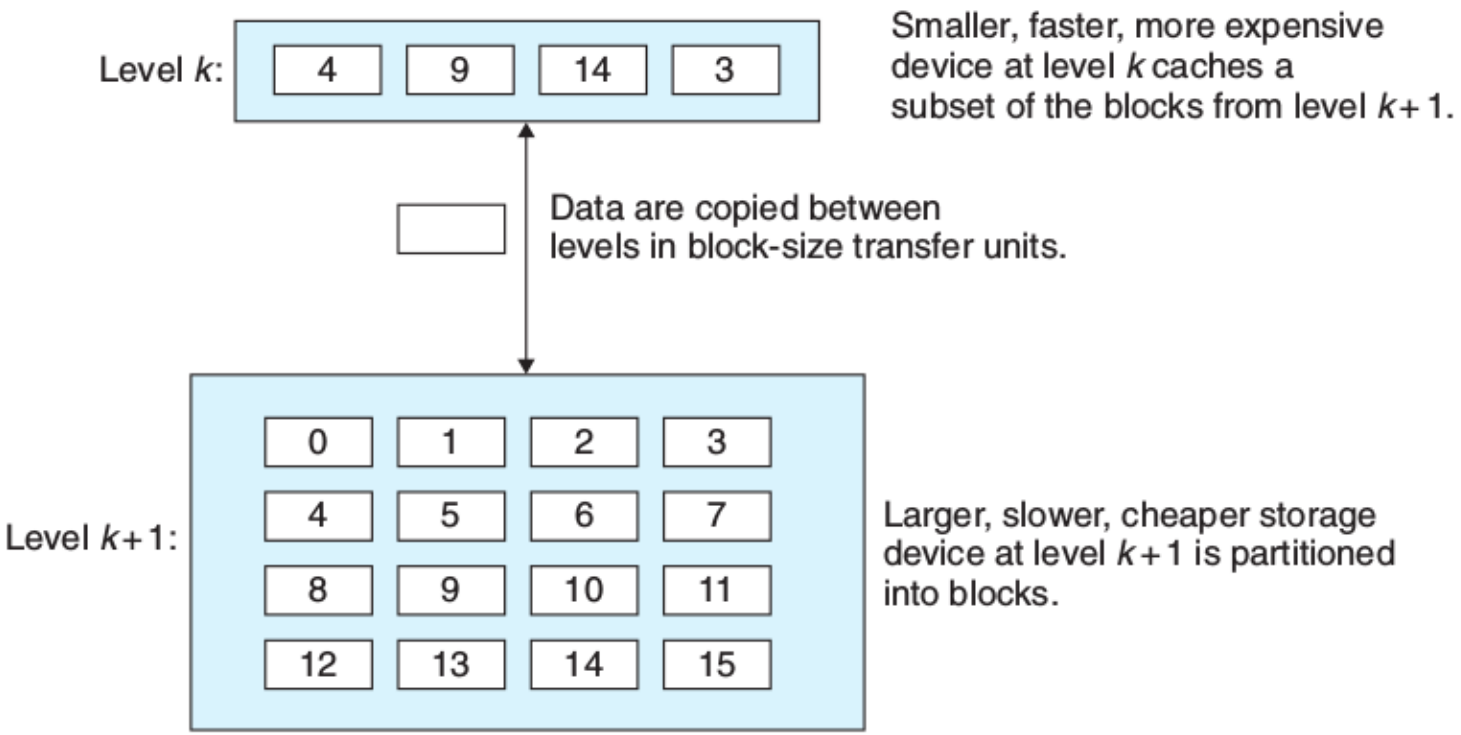
» Memory

Preliminaries

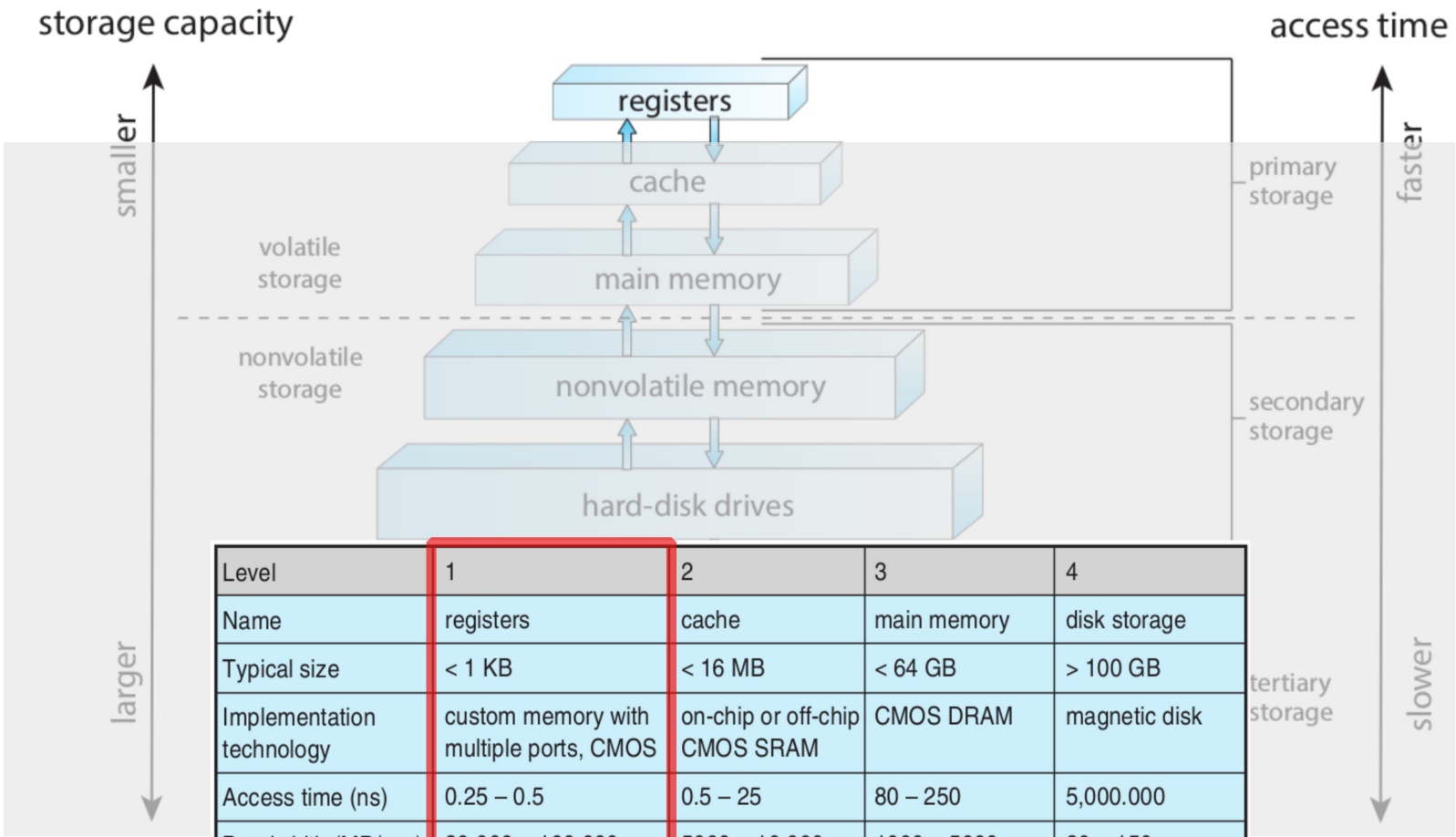
Storage Devices Form a Hierarchy



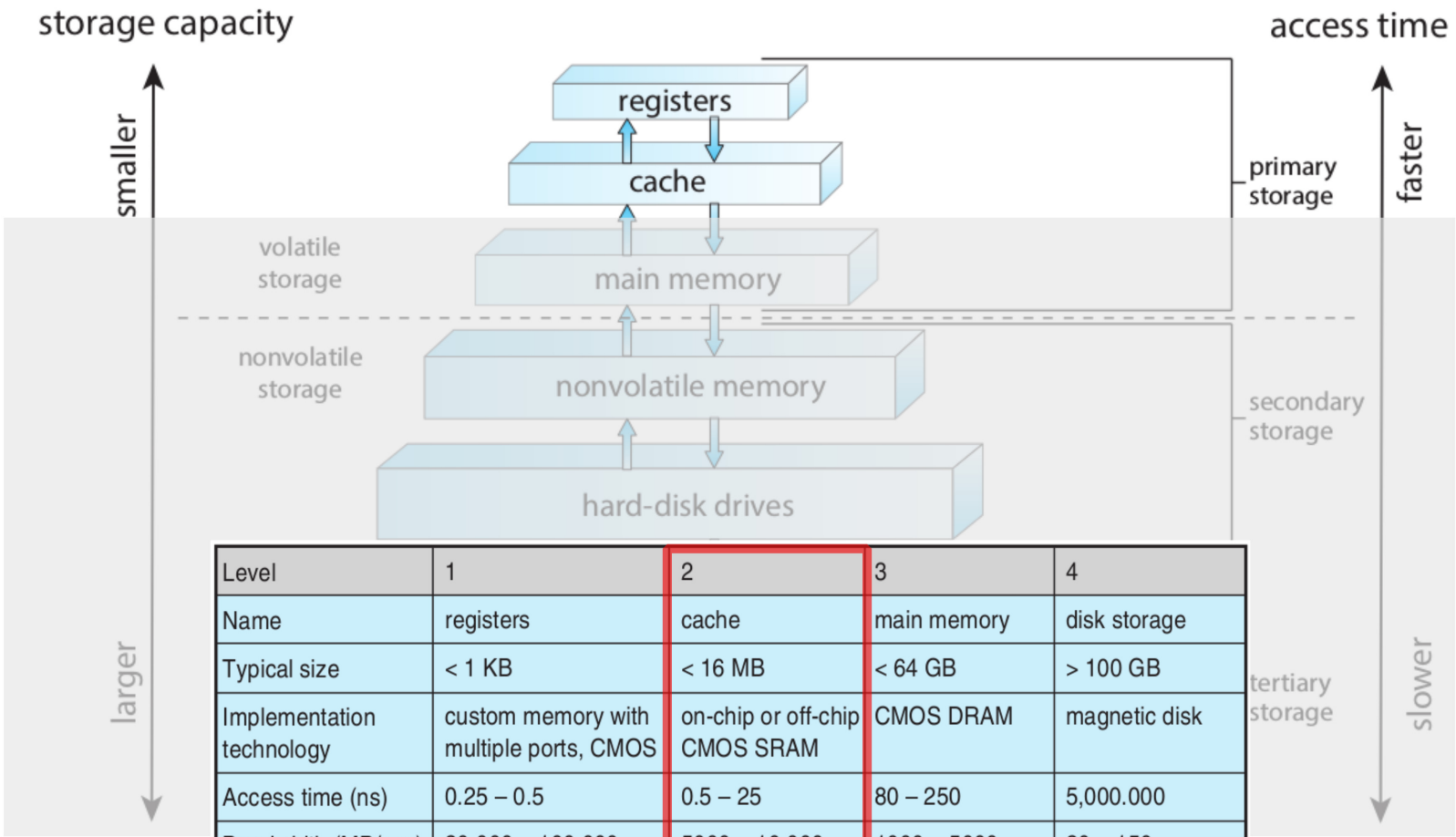
Cache



The basic principle of caching in a memory hierarchy



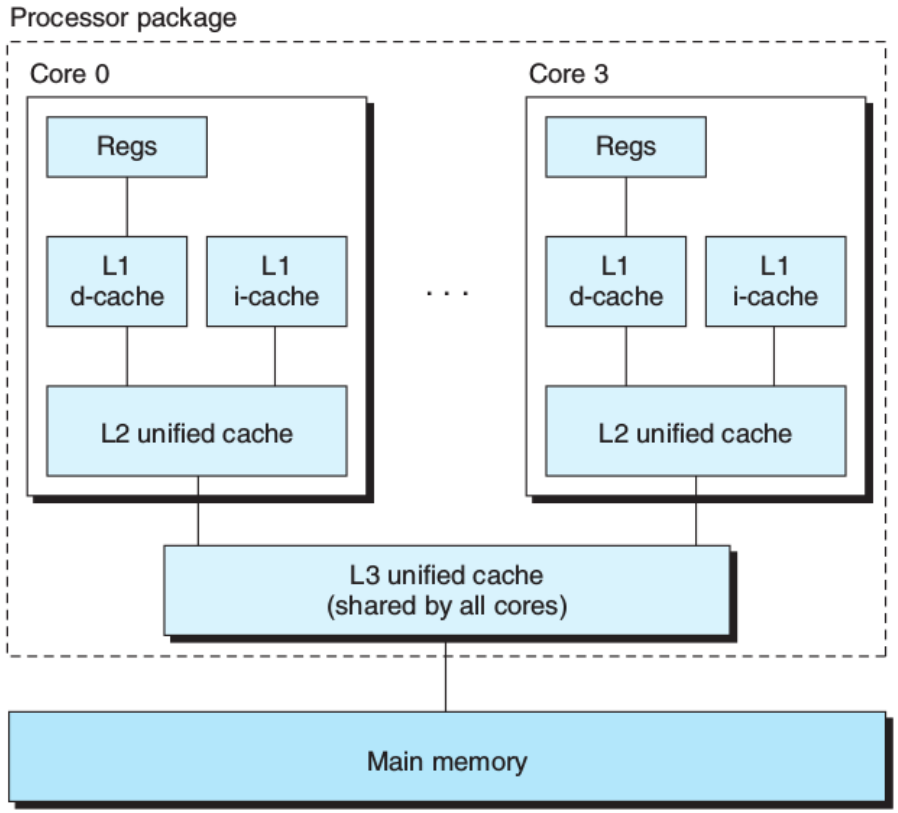
Level	1	2	3	4
Name	registers	cache	main memory	disk storage
Typical size	< 1 KB	< 16 MB	< 64 GB	> 100 GB
Implementation technology	custom memory with multiple ports, CMOS	on-chip or off-chip CMOS SRAM	CMOS DRAM	magnetic disk
Access time (ns)	0.25 – 0.5	0.5 – 25	80 – 250	5,000.000
Bandwidth (MB/sec)	20,000 – 100,000	5000 – 10,000	1000 – 5000	20 – 150
Managed by	compiler	hardware	operating system	operating system
Backed by	cache	main memory	disk	CD or tape



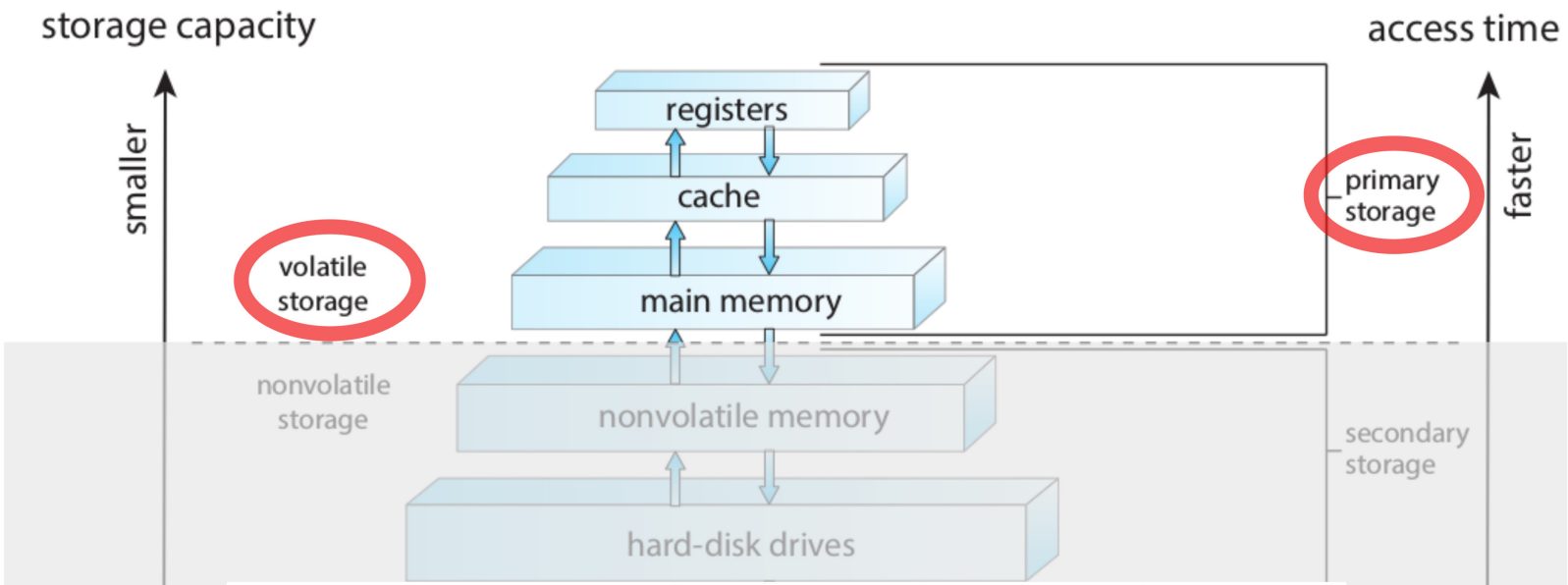
Level	1	2	3	4
Name	registers	cache	main memory	disk storage
Typical size	< 1 KB	< 16 MB	< 64 GB	> 100 GB
Implementation technology	custom memory with multiple ports, CMOS	on-chip or off-chip CMOS SRAM	CMOS DRAM	magnetic disk
Access time (ns)	0.25 – 0.5	0.5 – 25	80 – 250	5,000.000
Bandwidth (MB/sec)	20,000 – 100,000	5000 – 10,000	1000 – 5000	20 – 150
Managed by	compiler	hardware	operating system	operating system
Backed by	cache	main memory	disk	CD or tape

on my laptop:

```
$ lscpu
Architecture:            x86_64
CPU op-mode(s):          32-bit, 64-bit
Byte Order:              Little Endian
CPU(s):                  8
On-line CPU(s) list:     0-7
Thread(s) per core:      2
Core(s) per socket:      4
Socket(s):               1
NUMA node(s):            1
Vendor ID:               GenuineIntel
CPU family:              6
Model:                   60
Model name:              Intel(R) Core(TM) i7
Stepping:                3
CPU MHz:                 1169.609
CPU max MHz:             3200.0000
CPU min MHz:             800.0000
BogoMIPS:                4389.90
Virtualization:          VT-x
L1d cache:               32K
L1i cache:               32K
L2 cache:                256K
L3 cache:                6144K
```

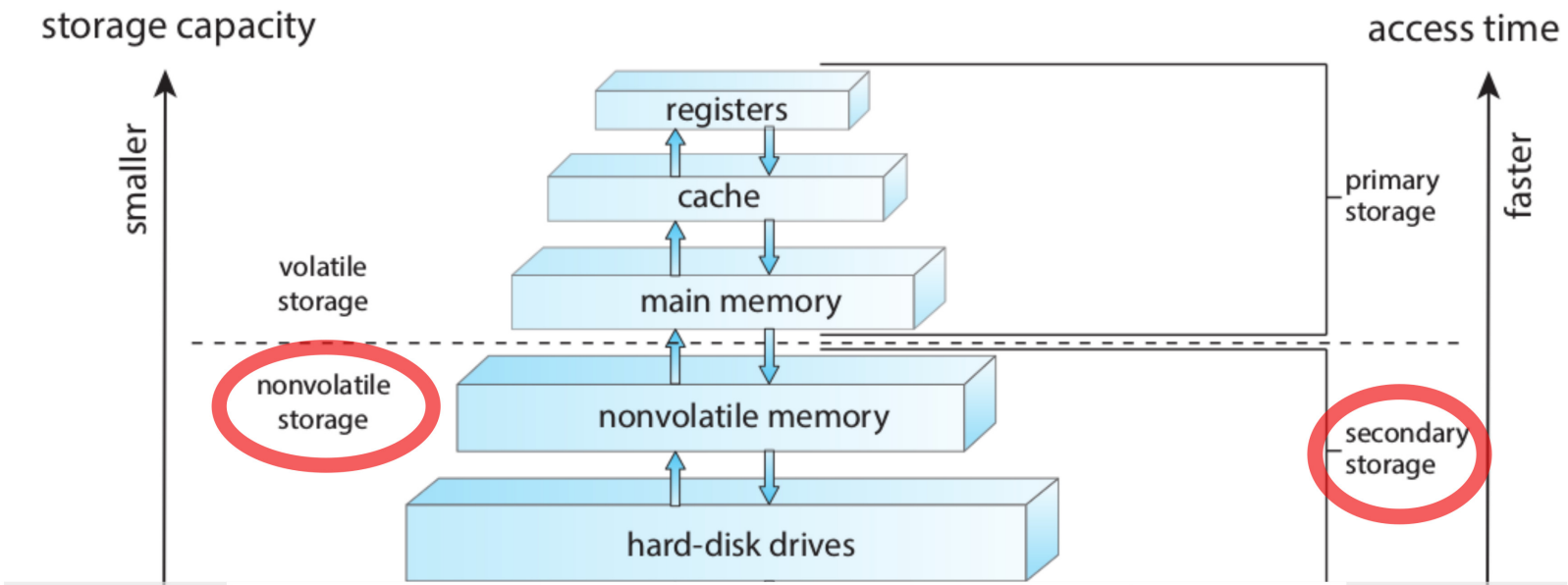


Intel Core i7 cache hierarchy



Level	1	2	3	4
Name	registers	cache	main memory	disk storage
Typical size	< 1 KB	< 16 MB	< 64 GB	> 100 GB
Implementation technology	custom memory with multiple ports, CMOS	on-chip or off-chip CMOS SRAM	CMOS DRAM	magnetic disk
Access time (ns)	0.25 – 0.5	0.5 – 25	80 – 250	5,000.000
Bandwidth (MB/sec)	20,000 – 100,000	5000 – 10,000	1000 – 5000	20 – 150
Managed by	compiler	hardware	operating system	operating system
Backed by	cache	main memory	disk	CD or tape

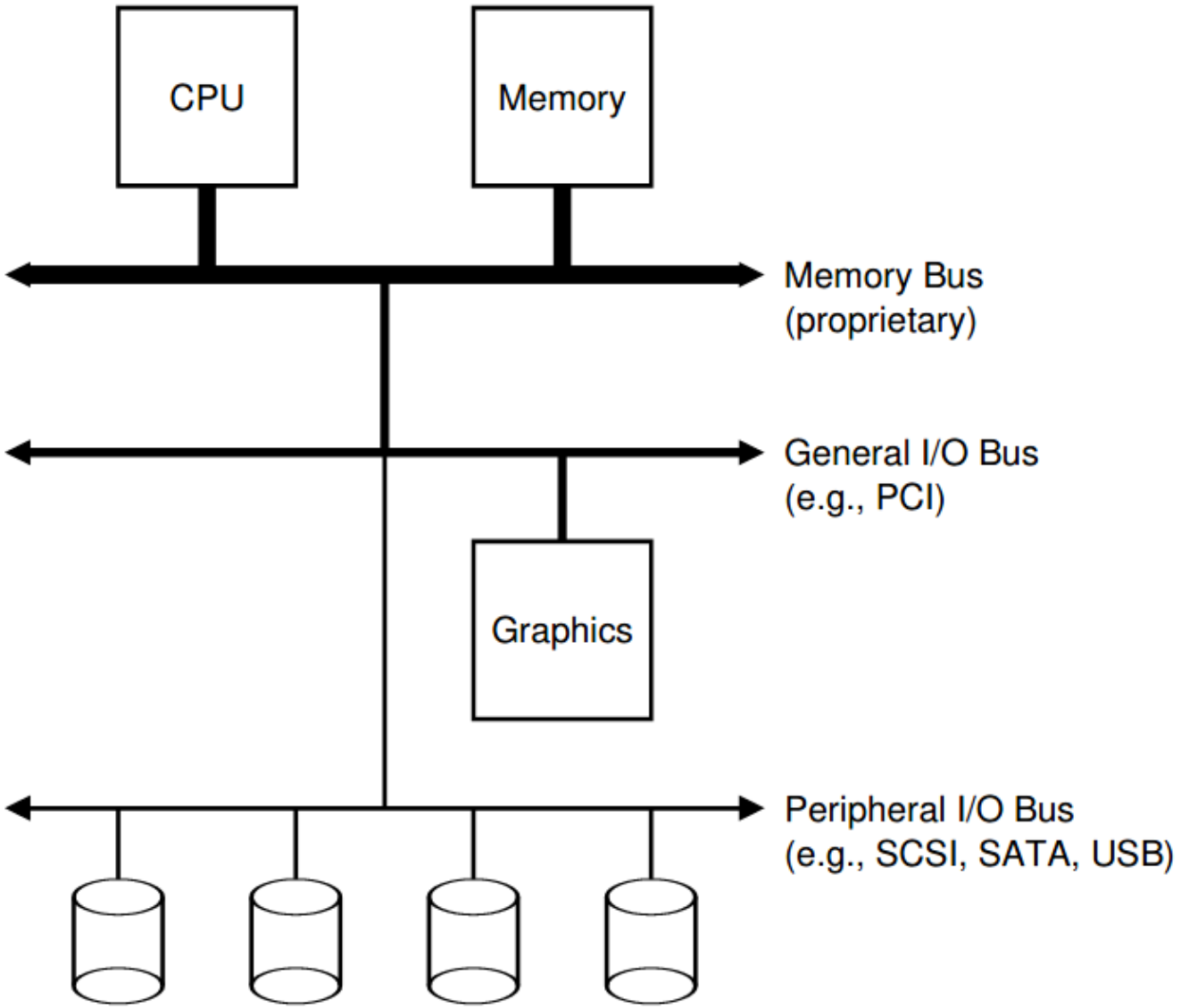
So far, everything is volatile!



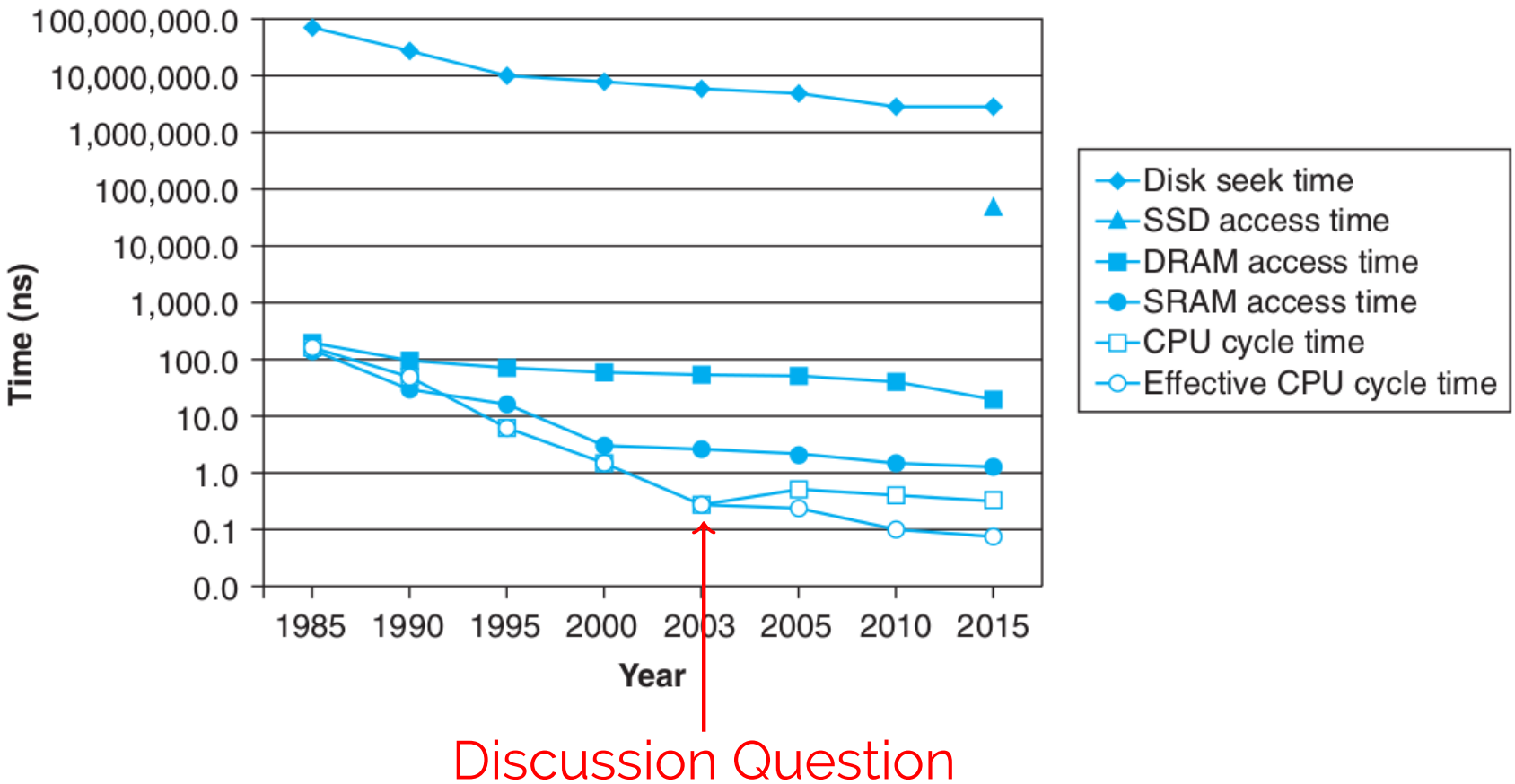
Level	1	2	3	4
Name	registers	cache	main memory	disk storage
Typical size	< 1 KB	< 16 MB	< 64 GB	> 100 GB
Implementation technology	custom memory with multiple ports, CMOS	on-chip or off-chip CMOS SRAM	CMOS DRAM	magnetic disk
Access time (ns)	0.25 – 0.5	0.5 – 25	80 – 250	5,000.000
Bandwidth (MB/sec)	20,000 – 100,000	5000 – 10,000	1000 – 5000	20 – 150
Managed by	compiler	hardware	operating system	operating system
Backed by	cache	main memory	disk	CD or tape

From here, everything is non-volatile!

Speed: technology and distance



The gap between disk, DRAM, and CPU speeds



Locality is the rescue

Note

Modern computers make heavy use of SRAM-based caches to try to bridge the processor-memory gap. This approach works because of a fundamental property of application programs known as **locality.**

Level	1	2	3	4
Name	registers	cache	main memory	disk storage
Typical size	< 1 KB	< 16 MB	< 64 GB	> 100 GB
Implementation technology	custom memory with multiple ports, CMOS	on-chip or off-chip CMOS SRAM	CMOS DRAM	magnetic disk
Access time (ns)	0.25 – 0.5	0.5 – 25	80 – 250	5,000.000
Bandwidth (MB/sec)	20,000 – 100,000	5000 – 10,000	1000 – 5000	20 – 150
Managed by	compiler	hardware	operating system	operating system
Backed by	cache	main memory	disk	CD or tape

This section

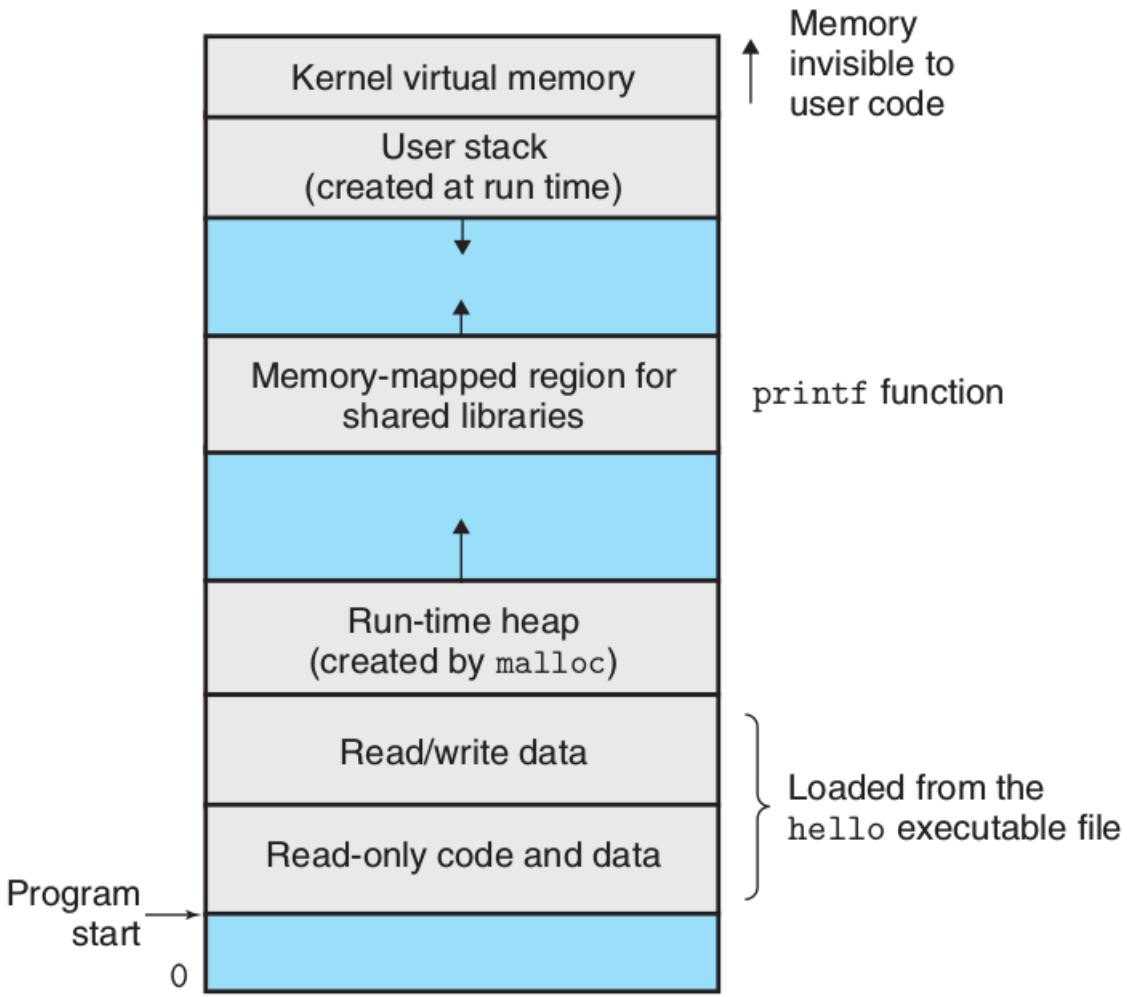
Address Space

Virtual CPU: *illusion* of private CPU registers

Virtual RAM: *illusion* of private memory

- every address from a user program is virtual,
- every address from a user program is virtual,
- every address from a user program is virtual,
- every ....

Process virtual address space. (The regions are not drawn to scale.)



```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    printf("location of code : %p\n", main);
    printf("location of heap : %p\n", malloc(sizeof(int)));
    int x = 3;
    printf("location of stack: %p\n", &x);
    return 0;
}
```

```
$ ./va
location of code :  0x4005f4
location of heap :  0x602420
location of stack:  0x7fffffffd934
```

- code comes first in the address space, then the heap, and the stack is all the way at the other end of this large virtual space.

All of these addresses are virtual, and will be translated by the OS and hardware in order to their true physical locations.

Memory Accesses

```
int main()
{
  int x=0;
  x = x + 3;
  return x;
}
```

```
<main>:
   0:   movl   $0x0,-0x4(%rsp)
   8:   addl   $0x3,-0x4(%rsp)
   d:   mov    -0x4(%rsp),%eax
  11:   retq
```

```
$ gcc -c -fomit-frame-pointer vm.c
$ objdump -S vm.o
```

Memory Accesses

```
int main()                <main>:
{                           0:   movl   $0x0,-0x4(%rsp)
  int x=0;                  8:   addl   $0x3,-0x4(%rsp)
  x = x + 3;                d:   mov    -0x4(%rsp),%eax
  return x;                11:   retq
}
```

Fetch instruction at addr `0x0`

Exec, store `0` to `x` (a stack location at addr `M[rsp-4]`)

Fetch instruction at addr `0x8`

Exec, load, add, and store

Fetch instruction at addr `0xd`

Exec, load from `x`

Fetch instruction at addr `0x11`

Exec, load from `M[rsp]` to `eip`

Memory Accesses

Exercise: count the number of memory access for the following code:

```
int main()
{
  register int x;
  x = x + 3;
  return x;
}
```

hint: Name the source file as exvm.c and use the following shell instructions:

```
$ gcc -c -fomit-frame-pointer exvm.c
$ objdump -S exvm.o
```

Time Sharing

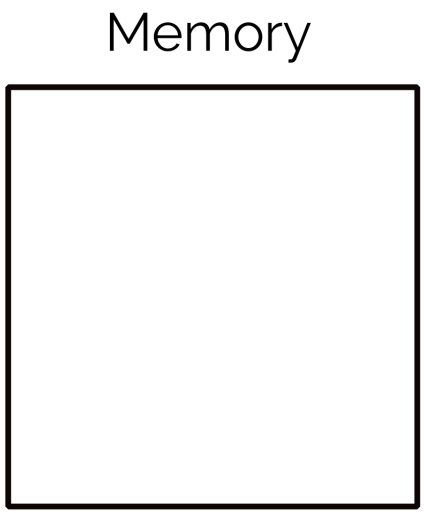
Problem: How to Run Multiple Processes?

Addresses are "hardcoded" into process binaries. How to avoid collisions?

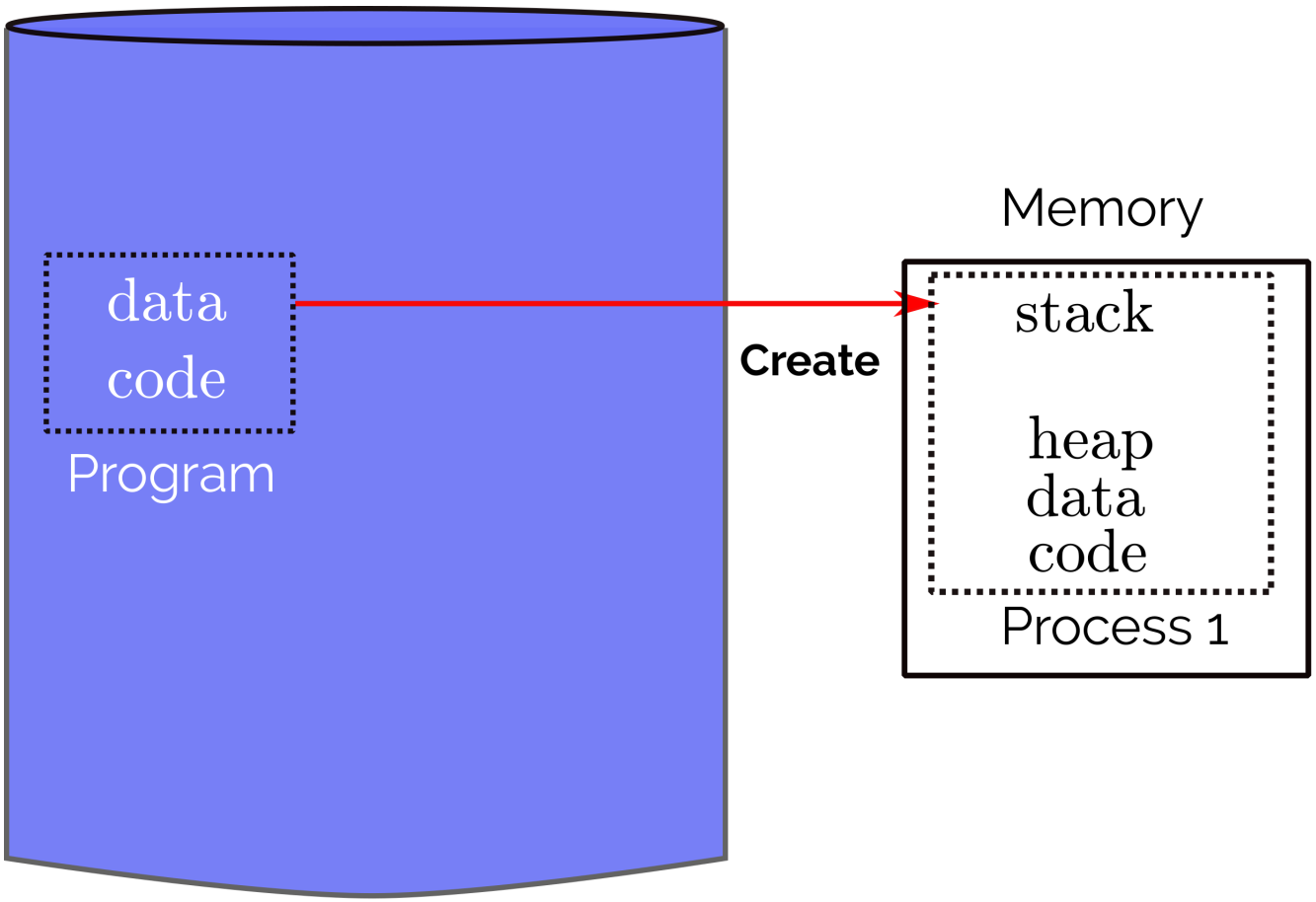
Approaches:

- **Time Sharing**
- Static Relocation
- Base
- Base + Bounds
- Segmentation
- Paging

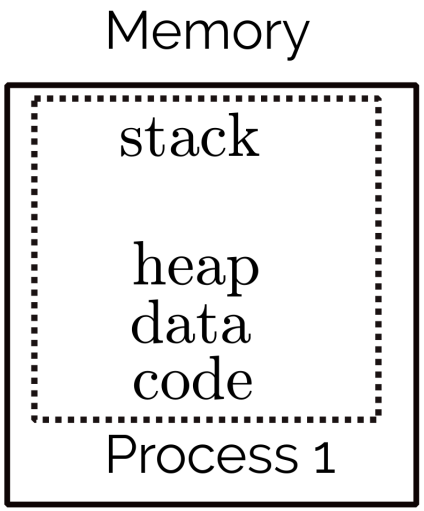
Time sharing



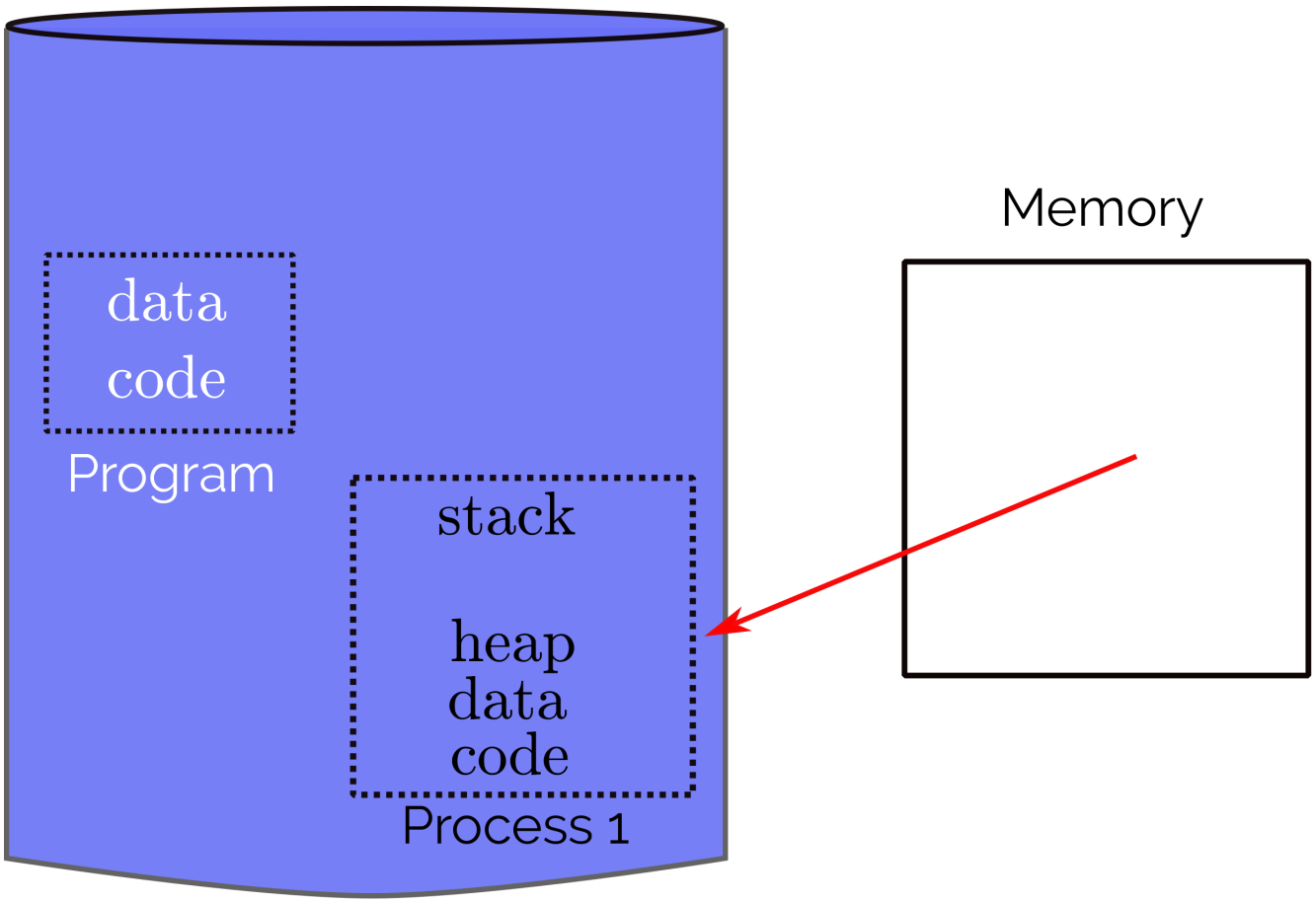
Time sharing



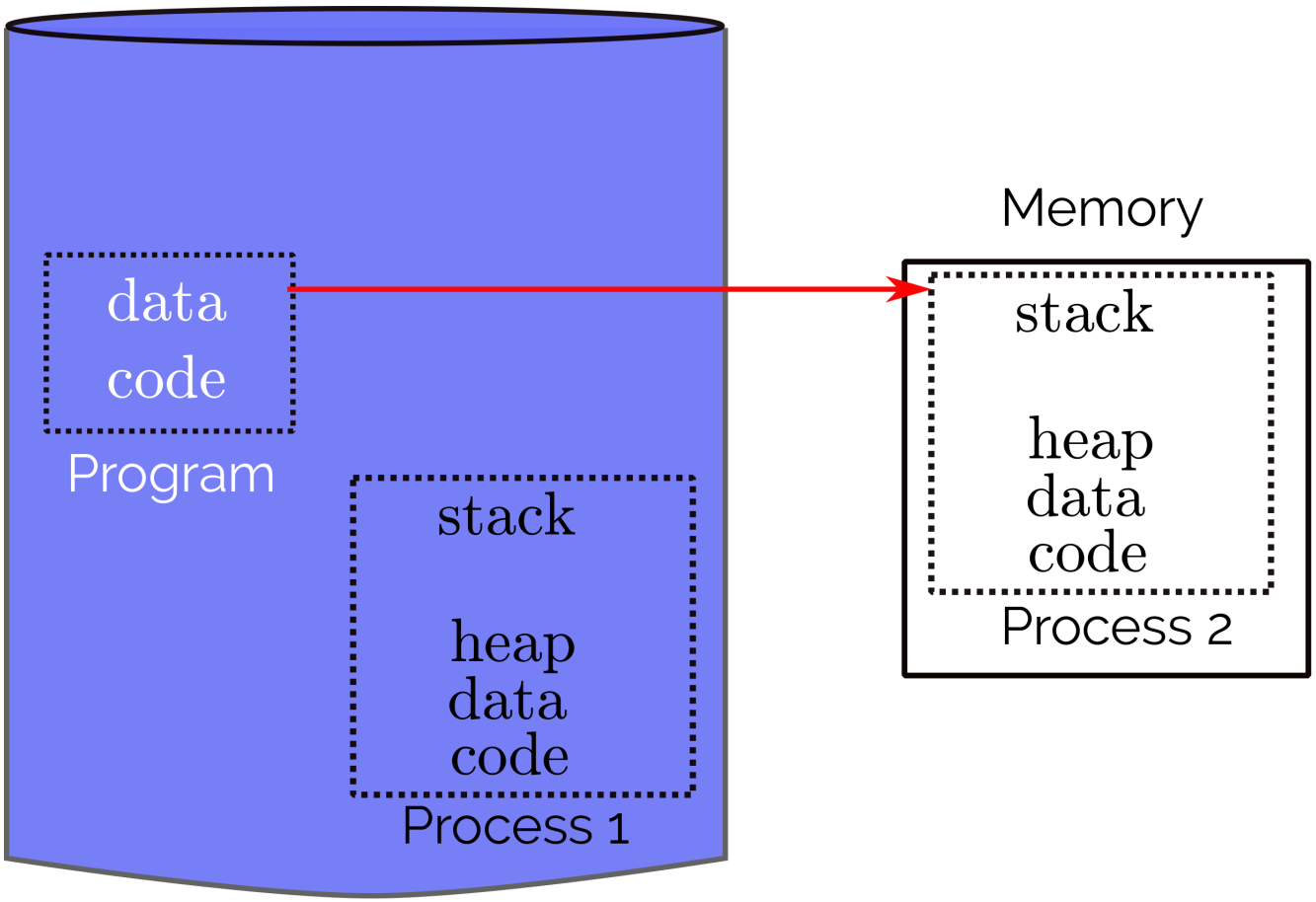
Time sharing



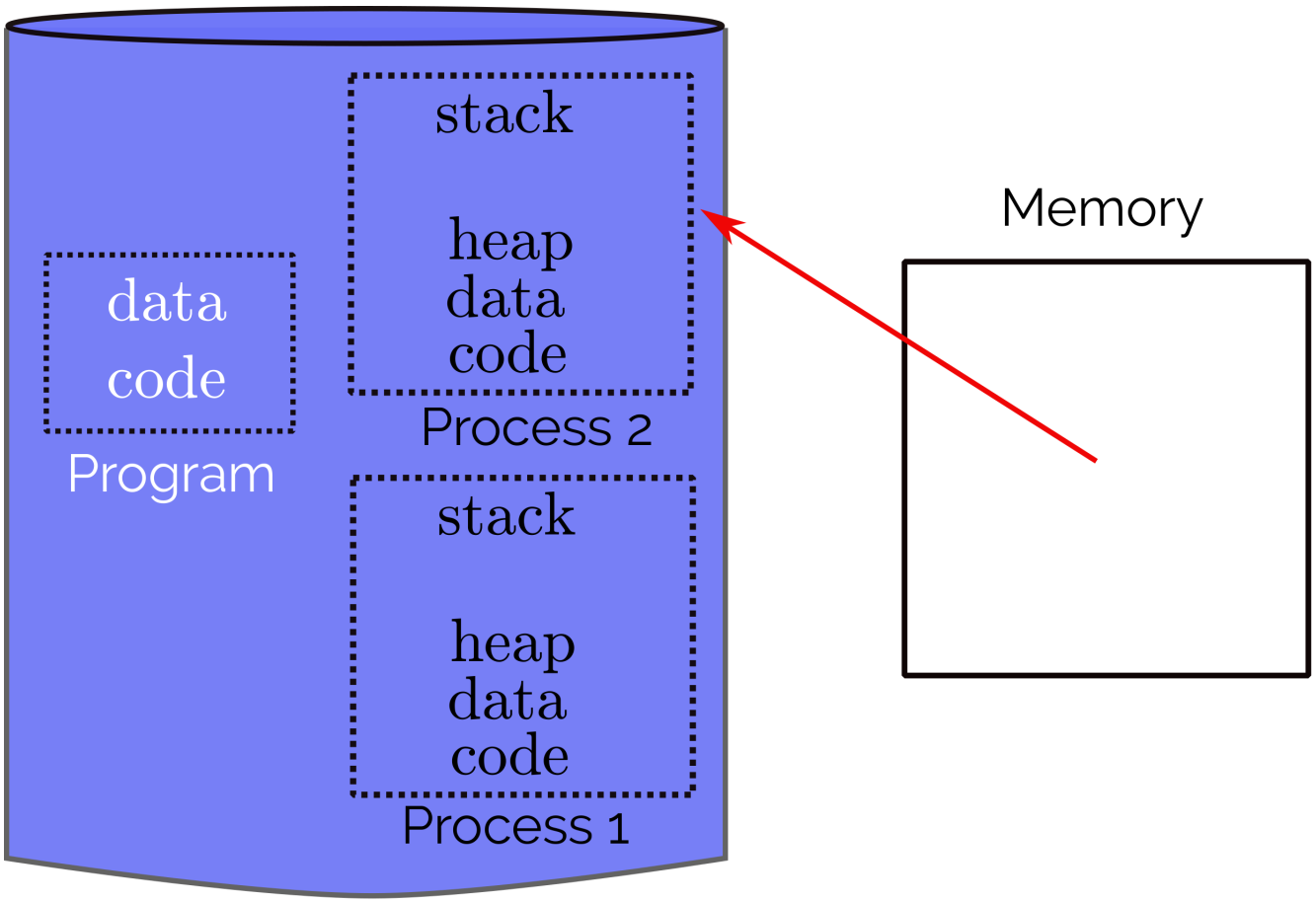
Time sharing



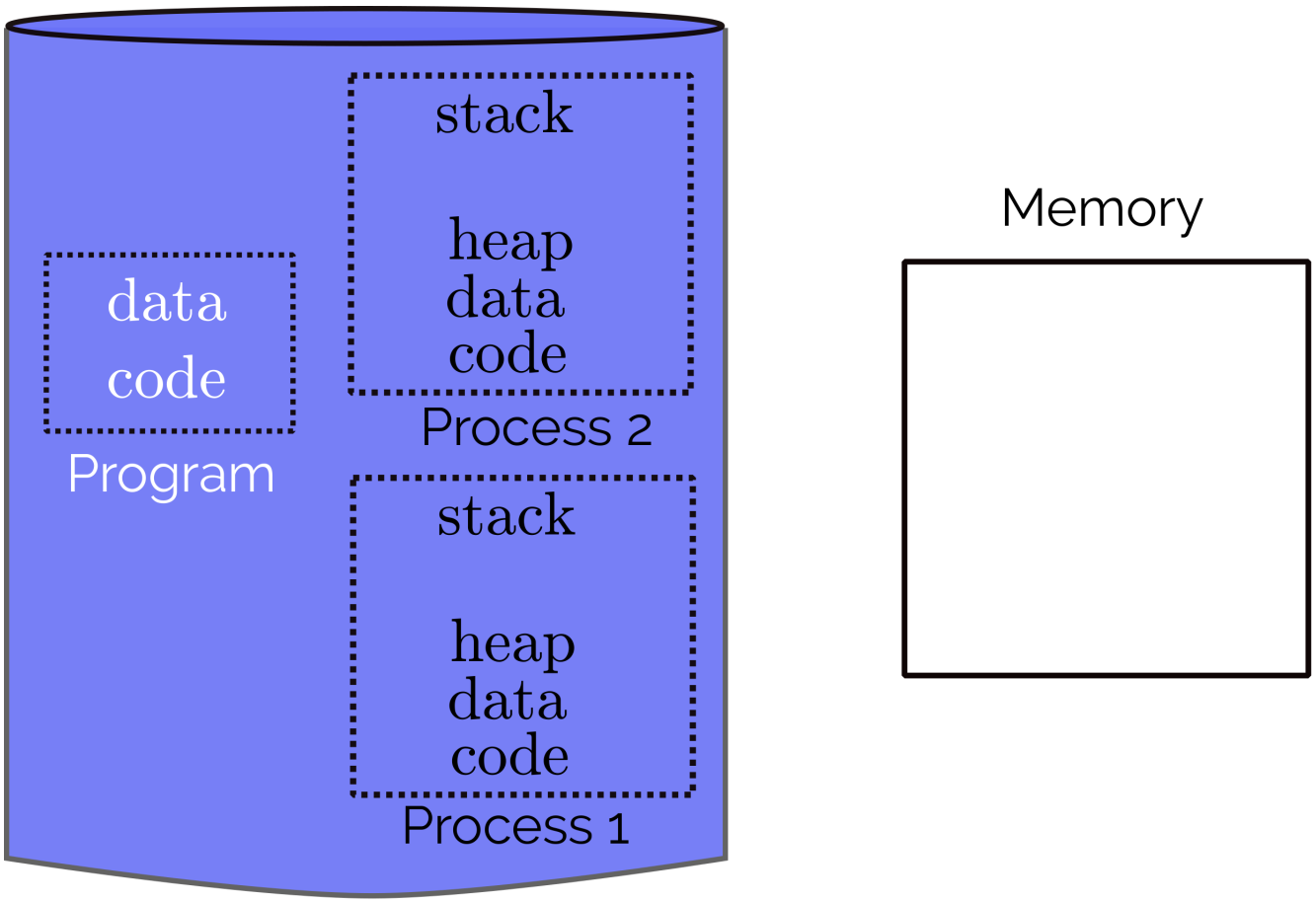
Time sharing



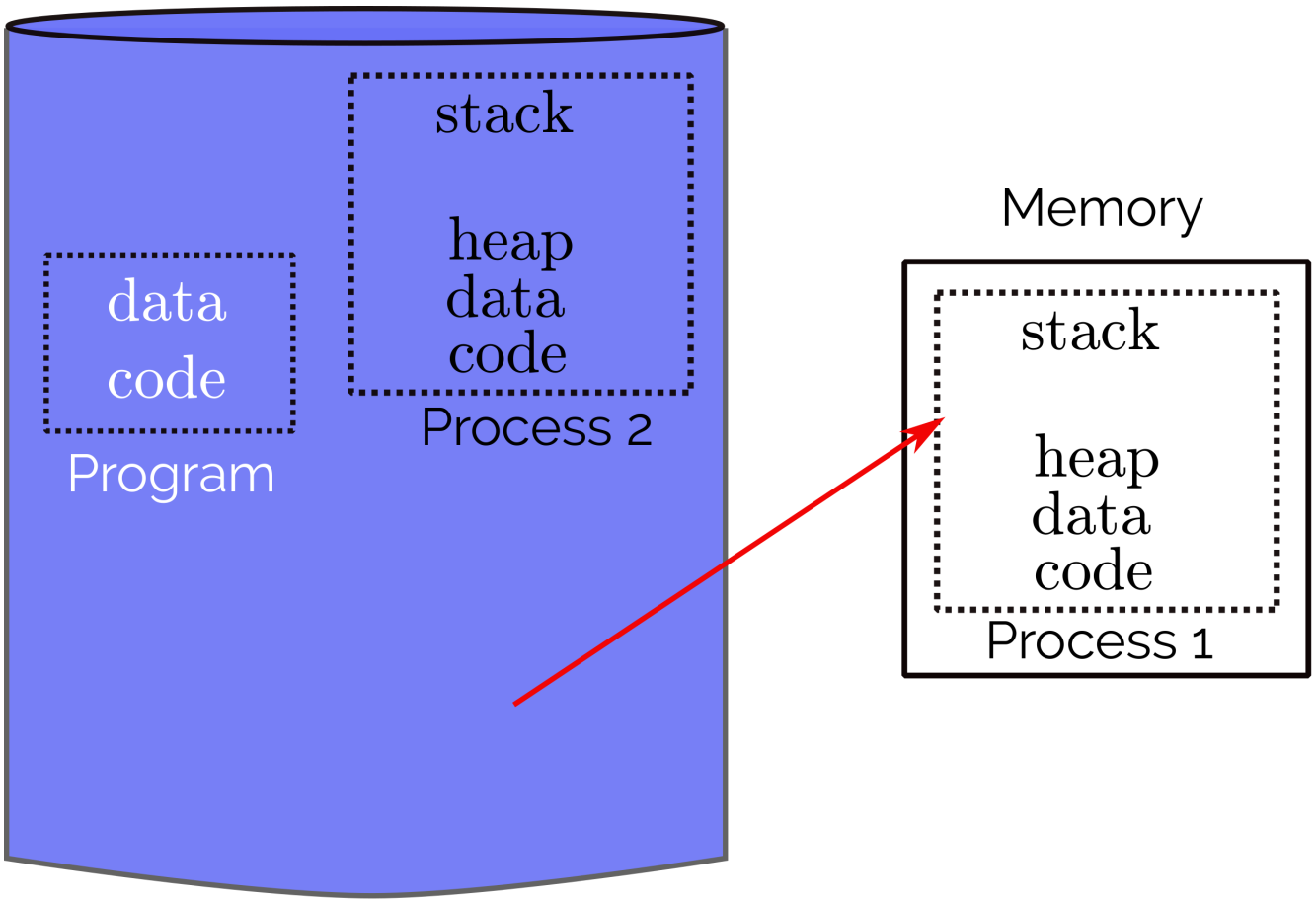
Time sharing



Time sharing



Time sharing



Time sharing

Problems?

What schedulers would time sharing work well with?

Alternative: space sharing

Static Relocation

Problem: How to Run Multiple Processes?

Approaches:

- Time Sharing
- **Static Relocation**
- Base
- Base + Bounds
- Segmentation
- Paging

Static Relocation

Idea: Load processes into disjoint chunks of memory purely via software (the *loader*)

$\Rightarrow$ rewrite each program before loading it as a process

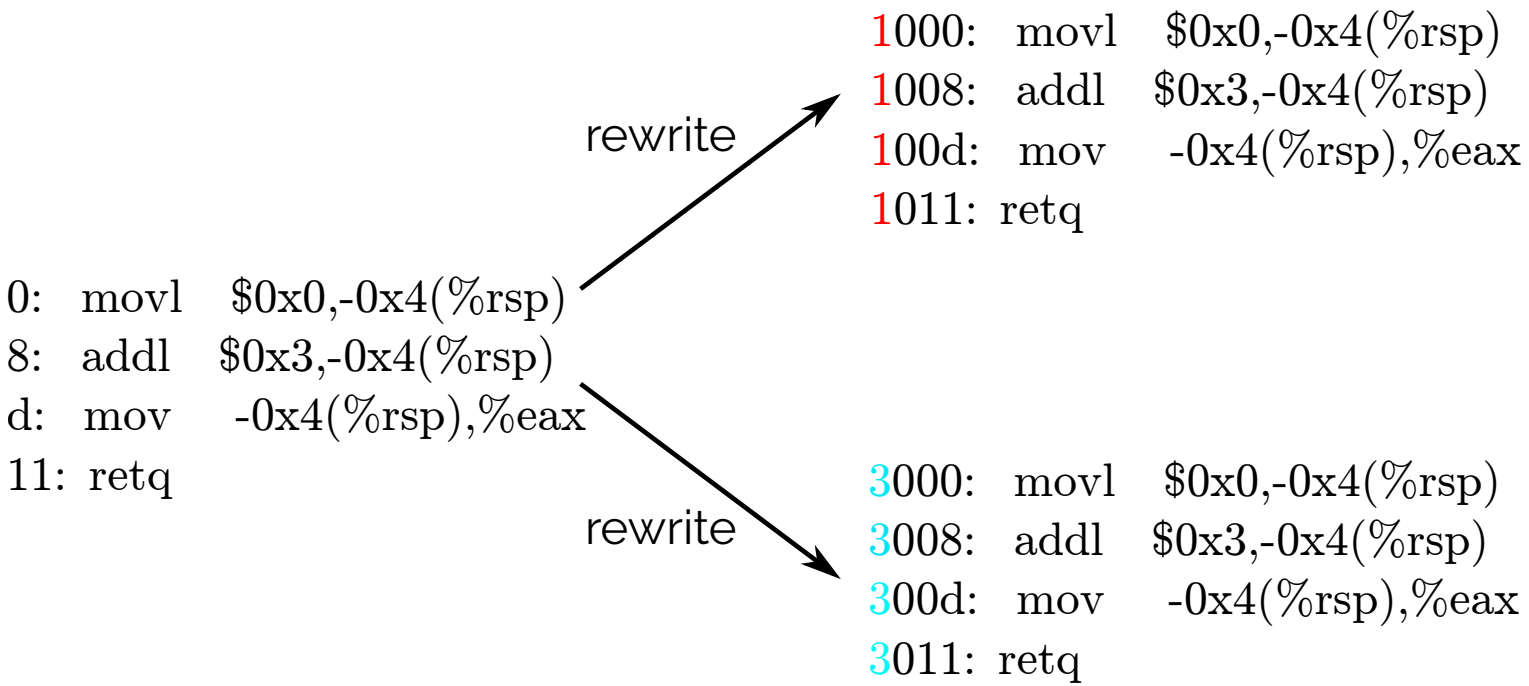
- Change jumps, loads, etc.

Example

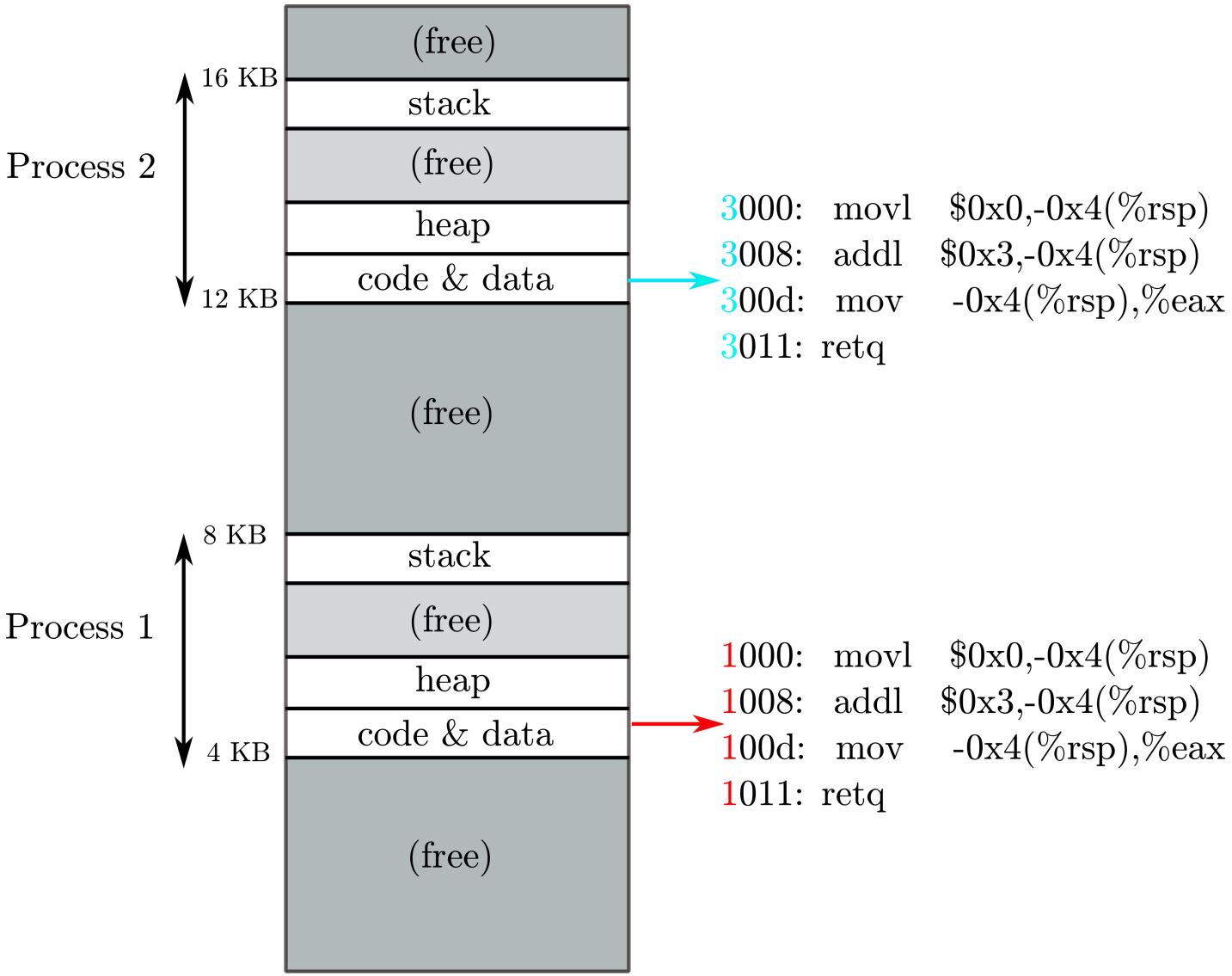
If there is an instruction in the form of `movl 1000, %eax` and the address space of the program was loaded starting at address 3000 (and not 0, as the program thinks), the loader would rewrite the instruction to offset each address by 3000 (e.g., `movl 4000, %eax`)

- Can any addresses be unchanged?

Static Relocation



Static Relocation



Static Relocation

Problems?

numerous! (protection, difficult to relocate an address space to another location)

Base

Problem: How to Run Multiple Processes?

Approaches:

- Time Sharing
- Static Relocation
- **Base**
- Base + Bounds
- Segmentation
- Paging

Base

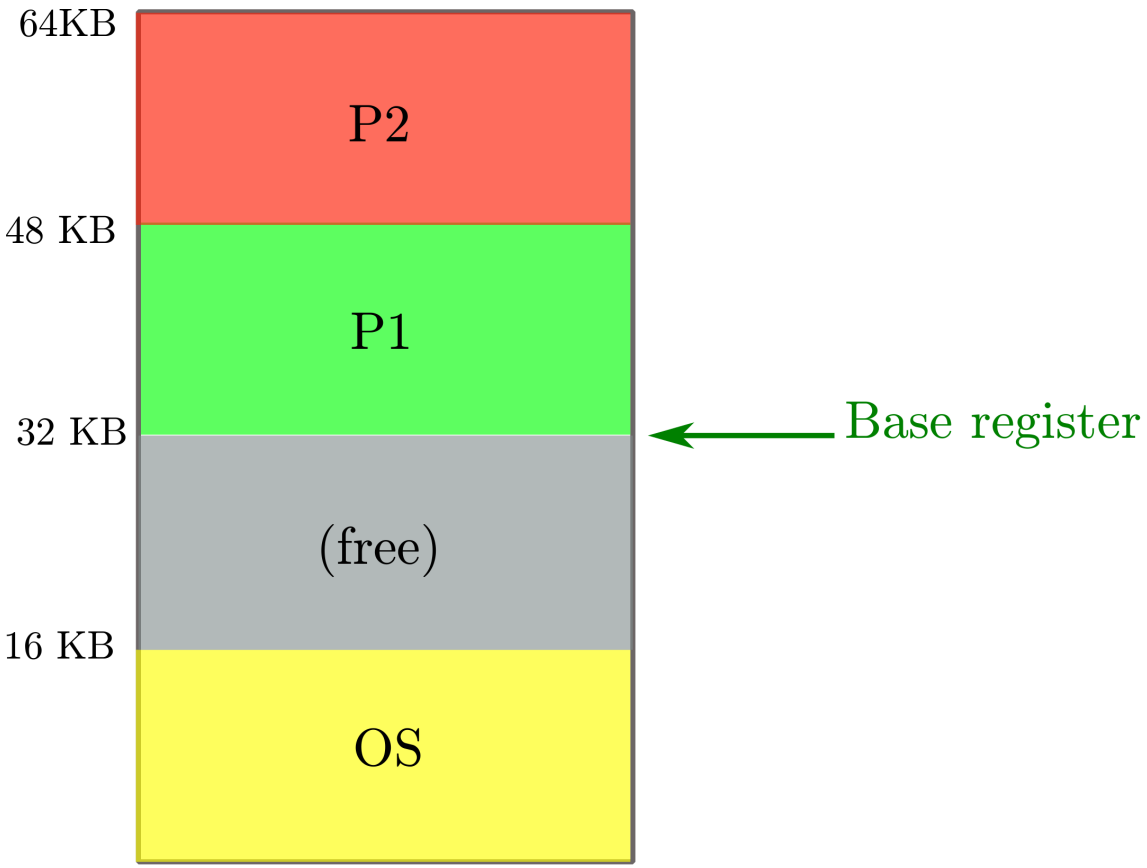
Idea: translate virtual addresses to physical by adding a fixed offset each time.

Store offset in a base register.

Each process has a *different value* in the base register when running.

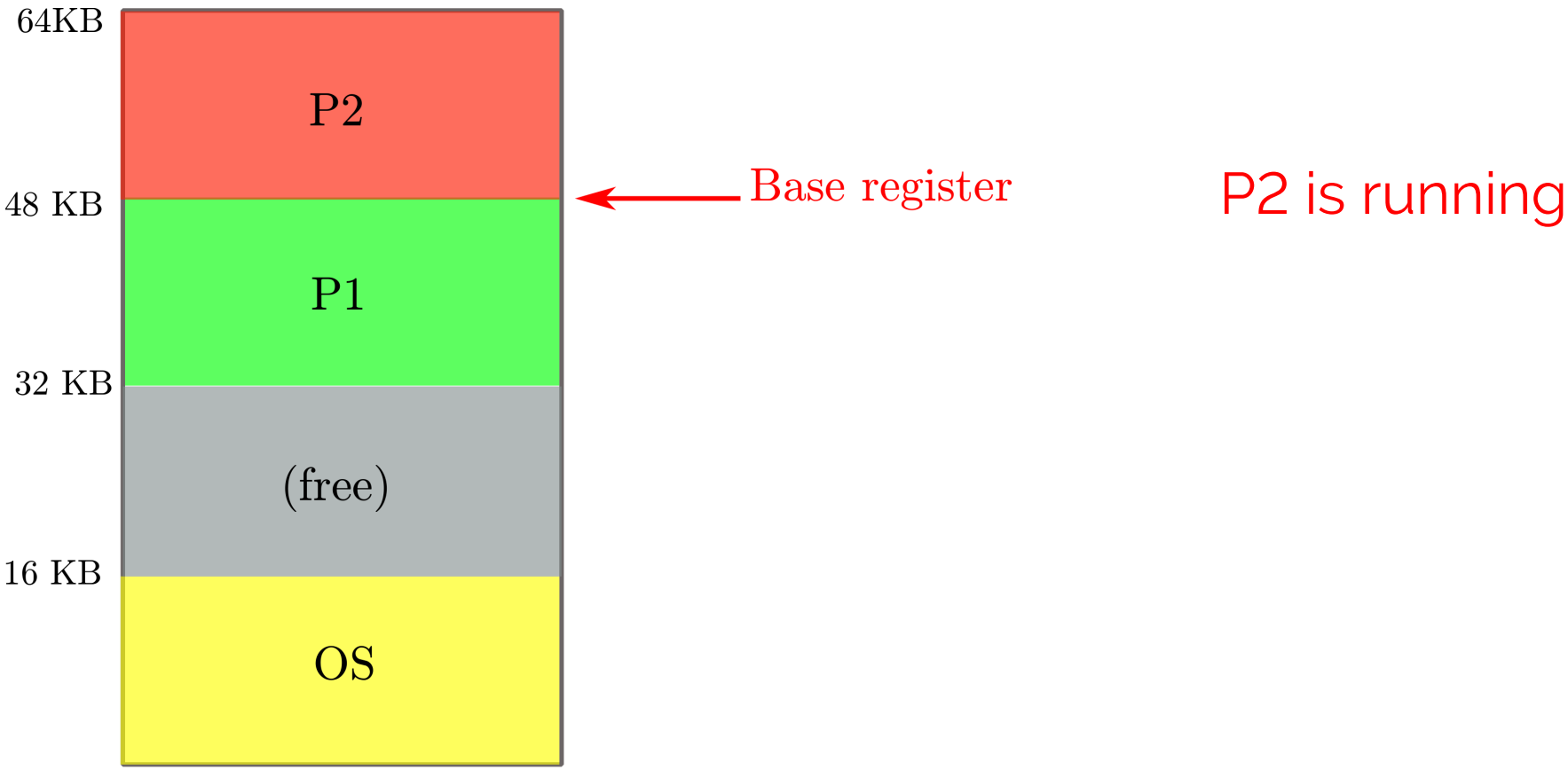
This is a "*dynamic relocation*" technique

Base



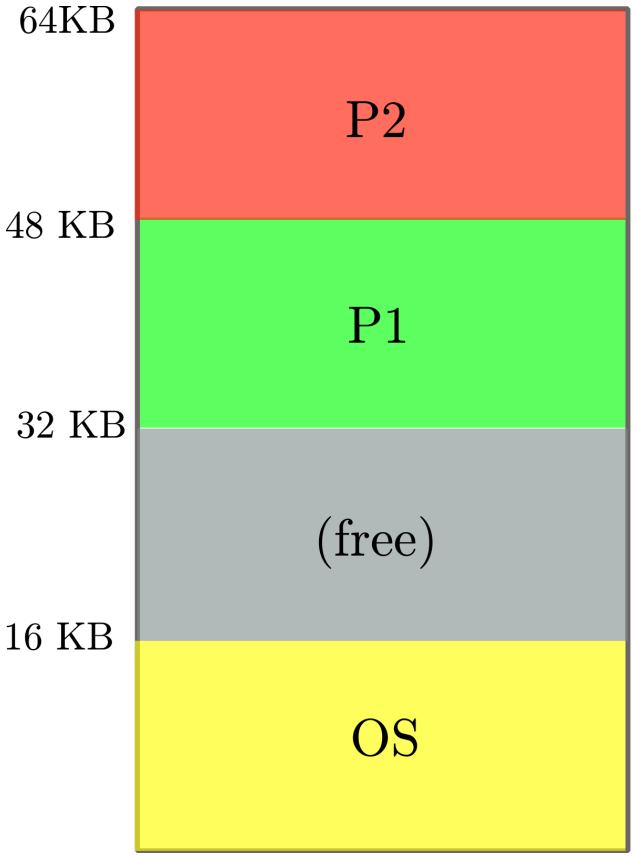
P1 is running

Base



P2 is running

Base



	Virtual	Physical
P1:	load 100, R1	load 32868, R1
P2:	load 100, R1	load 49252, R1
P2:	load 1000, R1	load 50152, R1
P1:	load 1000, R1	load 33768, R1

Base

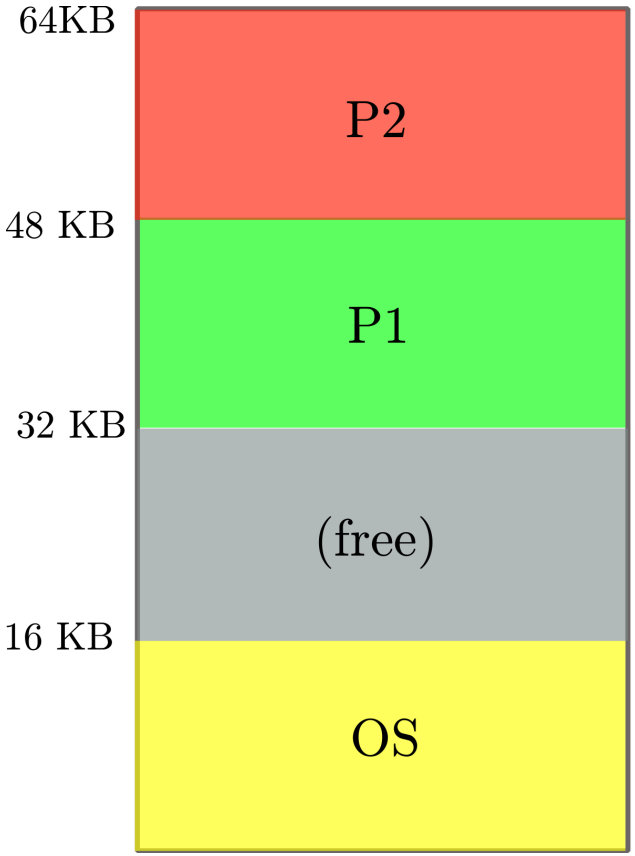
Who should do translation with base register?

(1) process, (2) OS, or (3) HW

Who should modify the base register?

(1) process, (2) OS, or (3) HW

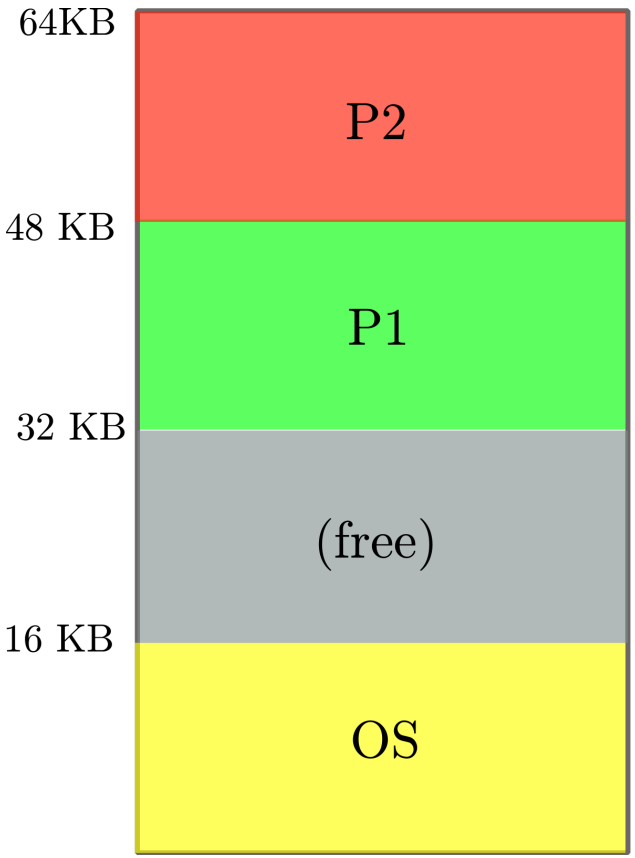
Base



	Virtual	Physical
P1:	load 100, R1	load 32868, R1
P2:	load 100, R1	load 49252, R1
P2:	load 1000, R1	load 50152, R1
P1:	load 1000, R1	load 33768, R1

Can P2 hurt P1?
Can P1 hurt P2?

Base



	Virtual	Physical
P1:	load 100, R1	load 32868, R1
P2:	load 100, R1	load 49252, R1
P2:	load 1000, R1	load 50152, R1
P1:	load 1000, R1	load 33768, R1
P1:	load 17000, R1	load 49768, R1

Can P2 hurt P1?

Can P1 hurt P2?

Problem: How to Run Multiple Processes?

Approaches:

- Time Sharing
- Static Relocation
- Base
- Base + Bounds
- Segmentation
- Paging

Base + Bounds

Base + Bounds

Idea: limit the address space with a bounds register

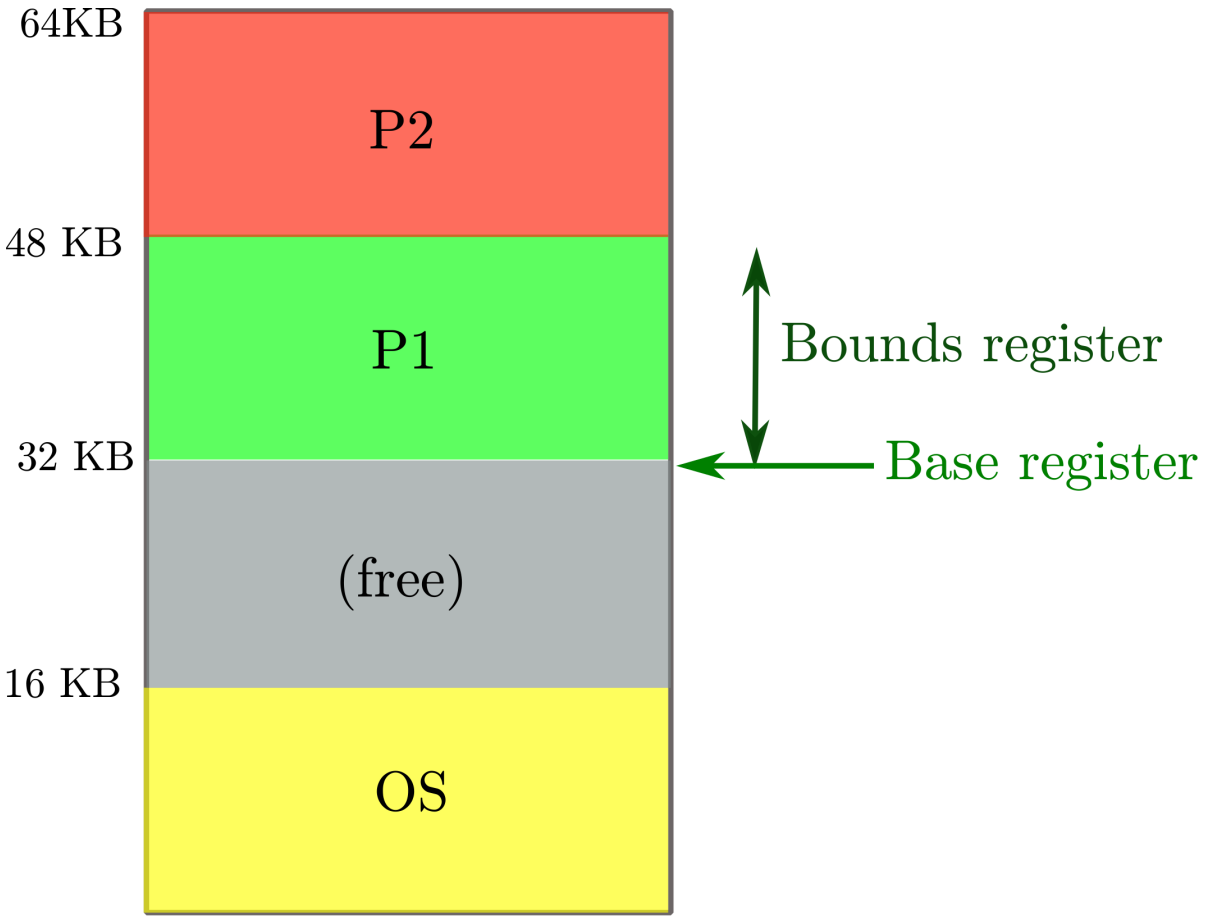
Base register: smallest physical addr (or starting location)

Bounds register: size of the virtual address space

Sometimes defined as largest physical address (base + size)

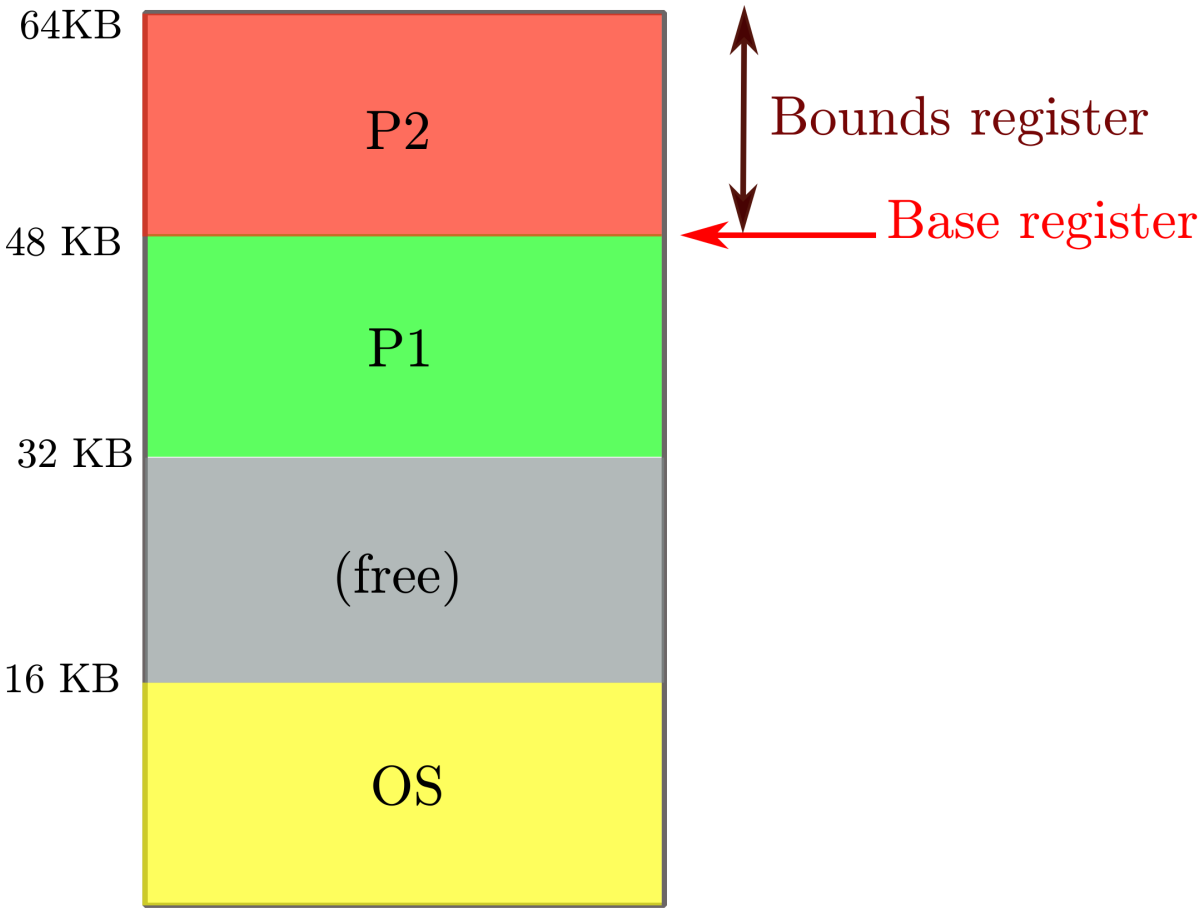
What happens if you load/store after bounds?

Base + Bounds



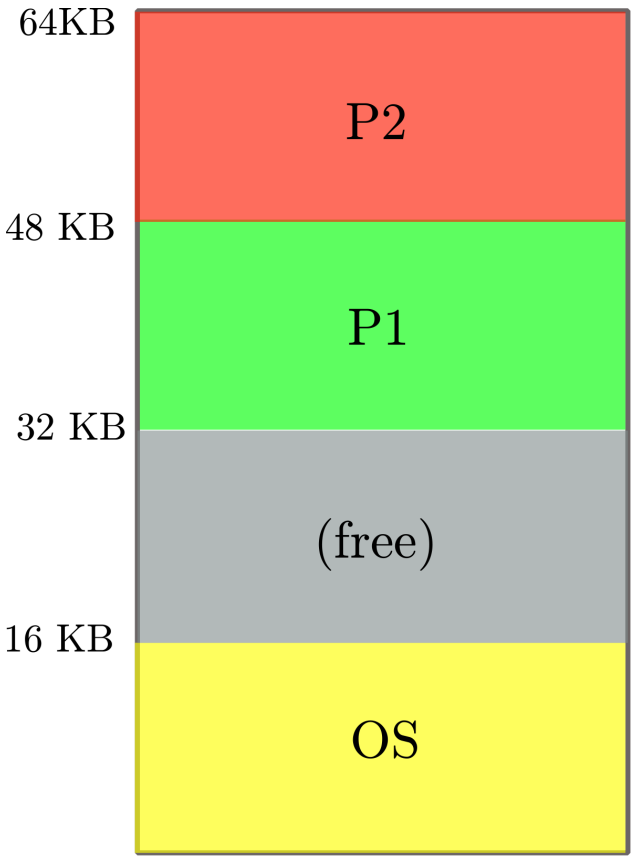
P1 is running

Base + Bounds



P2 is running

Base + Bounds

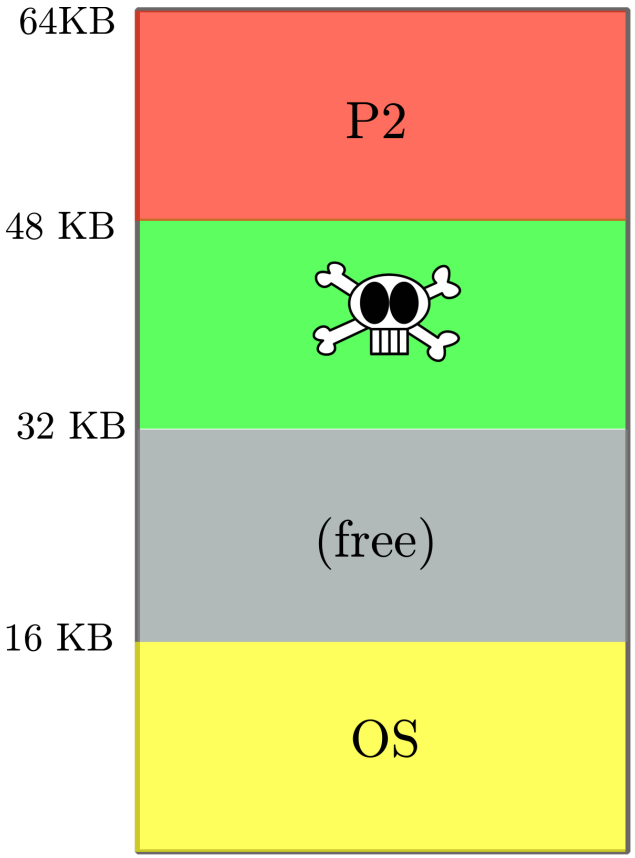


	Virtual	Physical
P1:	load 100, R1	load 32868, R1
P2:	load 100, R1	load 49252, R1
P2:	load 1000, R1	load 50152, R1
P1:	load 1000, R1	load 33768, R1
P1:	load 17000, R1	

Can P2 hurt P1?

Can P1 hurt P2?

Base + Bounds



	Virtual	Physical
P1:	load 100, R1	load 32868, R1
P2:	load 100, R1	load 49252, R1
P2:	load 1000, R1	load 50152, R1
P1:	load 1000, R1	load 33768, R1
P1:	load 1700, R1	Fault (>16KB)

Can P2 hurt P1?

Can P1 hurt P2?

Base + Bounds

Hardware support

A pair of registers (base and bounds) per CPU

Ability to translate virtual addresses and check if within bounds

Extended LDE **A minimal management unit(MMU)**

Privileged instructions to modify the base and bounds registers

Generate exceptions & arrange for the OS “out-of-bounds" exception handler to run.

Base + Bounds

Operating System Issues

Memory management (free list)

Base/bounds management at each context switch

PCB $\leftrightarrow$ base and bound registers

Exception handling

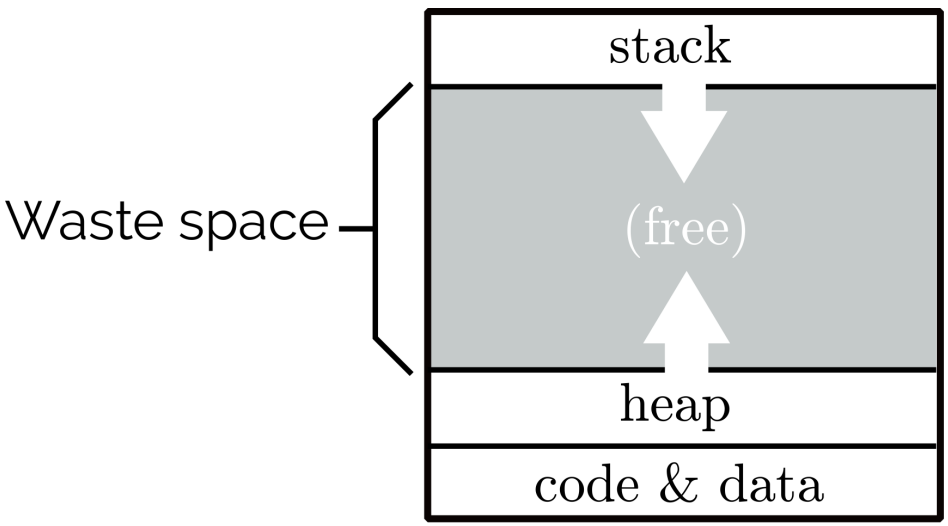
Base + Bounds

Pros

- fast & simple
- little bookkeeping overhead (2 registers / proc)

Cons

- sparse address spaces wastes memory (**internal fragmentation**)
- not as flexible as we would like
- **No partial sharing**: Cannot share limited parts of address space



Segmentation

Problem: How to Run Multiple Processes?

Approaches:

- Time Sharing
- Static Relocation
- Base
- Base + Bounds
- **Segmentation**
- Paging

Segmentation

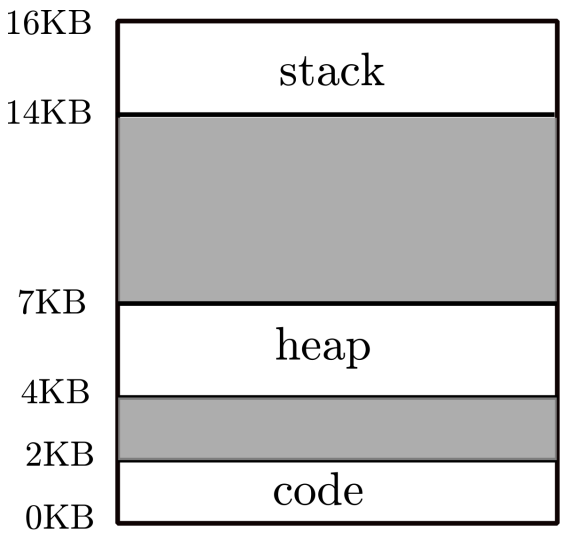
Idea: instead of having just one base and bounds pair in our MMU, why not have a base and bounds pair per logical segment of the address space?

Segment: a contiguous portion of the address space of a particular length: code, stack, and heap

OS can place segments in the memory independently $\Rightarrow$ not filling physical memory with unused address space

Resize segments (independently) as needed!

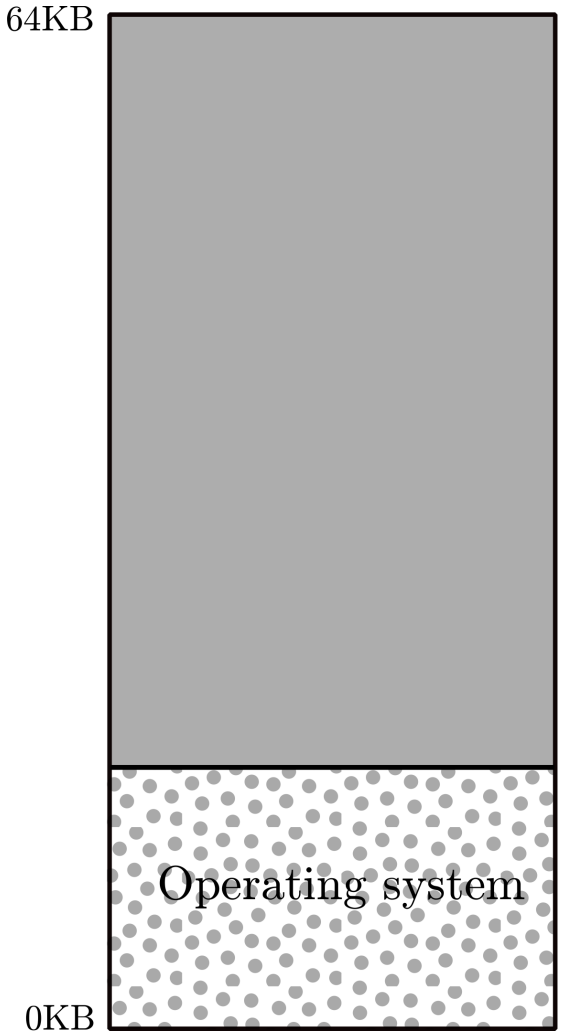
Segmentation



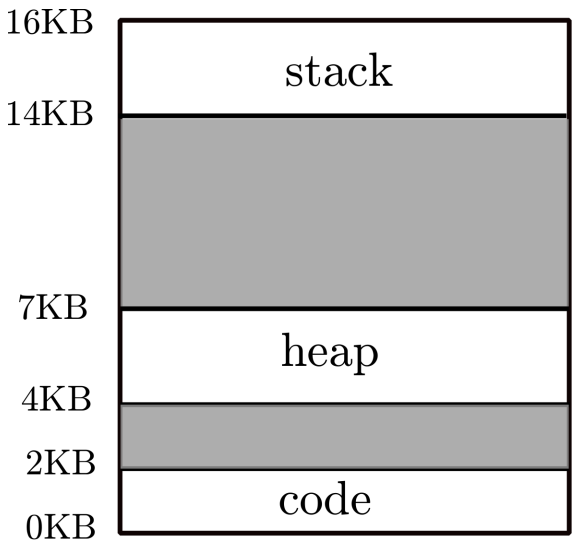
The address space

Segmentation

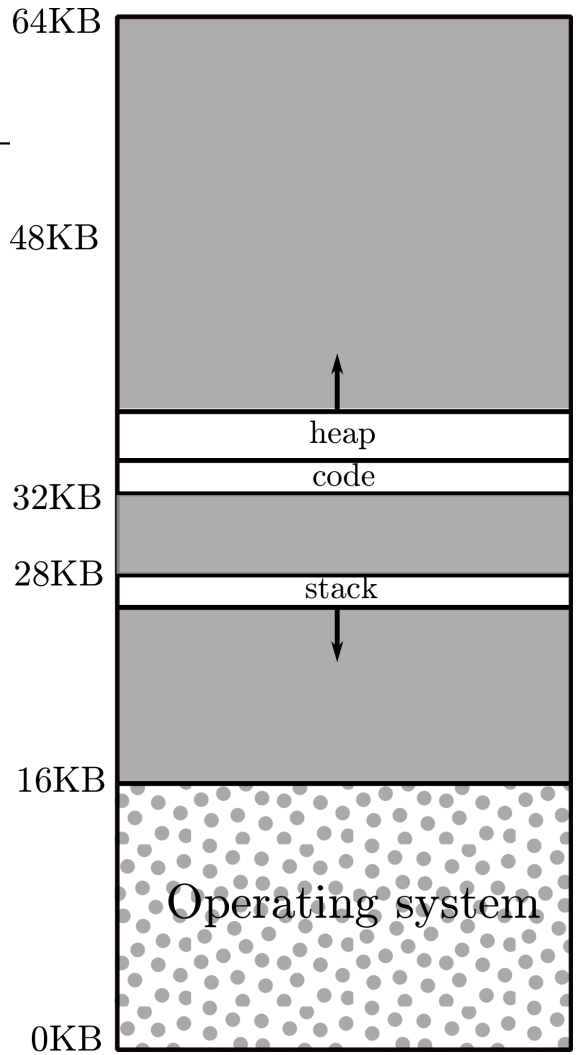
The physical memory



Segmentation



Segment	Base	Size
Code	32K	2K
Heap	34K	3K
Stack	28K	2K



Example

Virtual	Physical
100	100 + 32KB (3286)
4200	104 + 34KB (34920)
8KB	Fault

Segmentation

How hardware knows

which segment we are referring to? what the offset is?

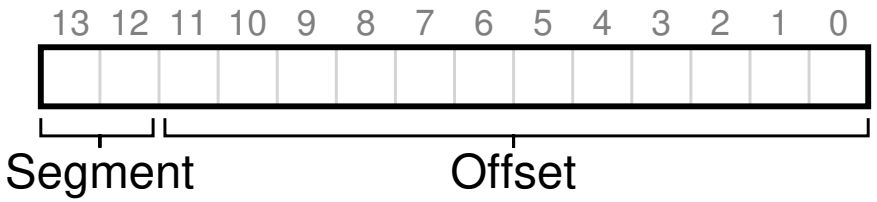
explicit approach:

Top bits of logical address select segment

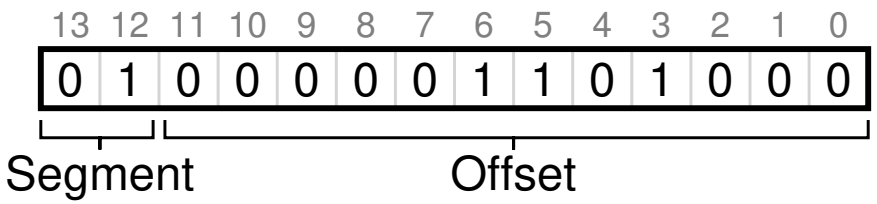
Low bits of logical address select offset within segment

Segmentation

For example, say addresses are 14 bits.
Use 2 bits for segment, 12 bits for offset



An address might look like 4200: 1 068



Choose some segment numbering, such as
0: code + data, 1: heap, 3: stack

Segmentation

```
// get top 2 bits of 14-bit VA
Segment = (VirtualAddress & SEG_MASK) >> SEG_SHIFT
// now get offset
Offset  = VirtualAddress & OFFSET_MASK
if (Offset >= Bounds[Segment])
  RaiseException(PROTECTION_FAULT)
else
  PhysAddr = Base[Segment] + Offset
  Register = AccessMemory(PhysAddr)
```

base and bounds are arrays (with one entry per segment)

In our running example

- SEGMASK = 0x3000, SEGSHIFT = 12, and OFFSETMASK = 0xFFF

Segmentation

What about the stack?

stack starts at “base" but grows backwards

Translation of stack must be treated differently

1- a little extra hardware support

Segment	Base	Size
Code	32K	2K
Heap	34K	3K
Stack	28K	2K

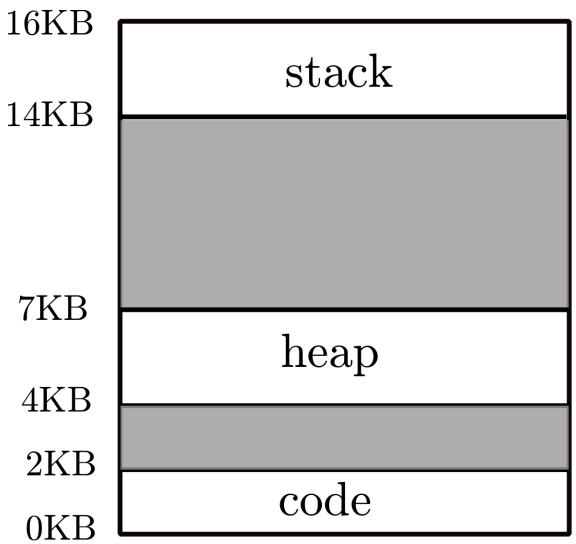
⟶

Segment	Base	Size	Grows Positive?
Code	32K	2K	1
Heap	34K	3K	1
Stack	28K	2K	0

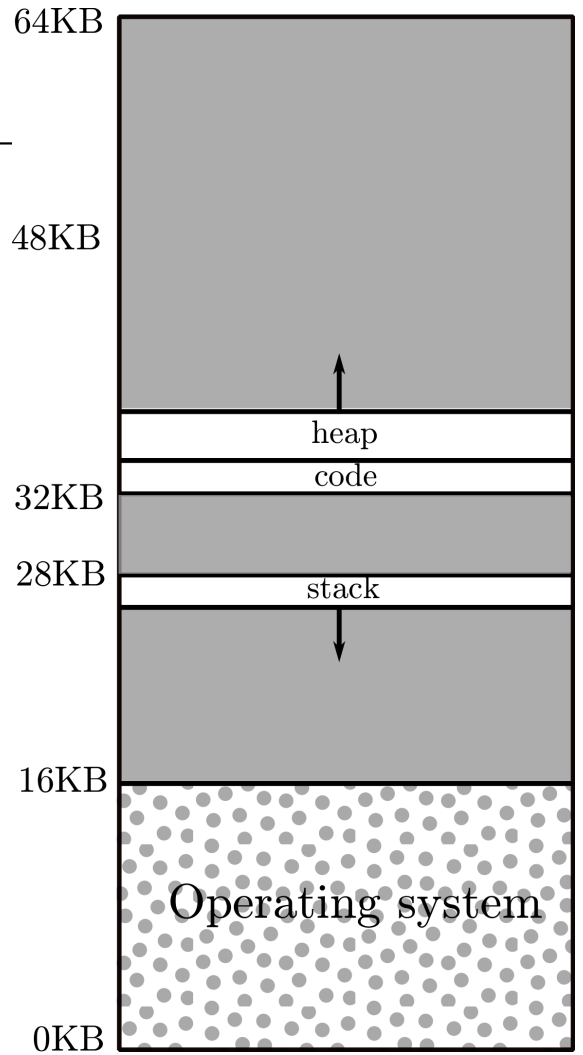
2- using different formula to obtain physical address

```
StackOffset = MAXSEGSIZE - Offset
// fault condition: StackOffset >= Bounds(Segment)
PhysAddr = Base[Segment] - StackOffset
```

Segmentation



Seg	Base	Size	Gr.
00	32K	2K	1
01	34K	3K	1
11	28K	2K	0



Example

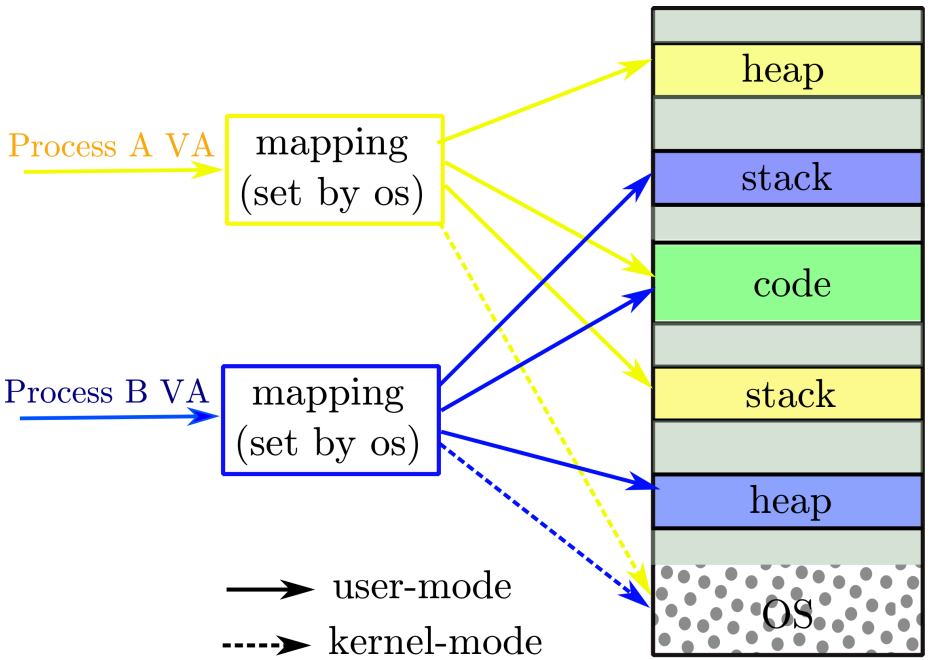
Virtual	Physical
15KB	28KB – (4KB – 3KB) (27KB)

Segmentation

Code Sharing

Now it is possible to share the (user-mode) code segments in the physical memory.

- For the processes with the same code segment



Segmentation

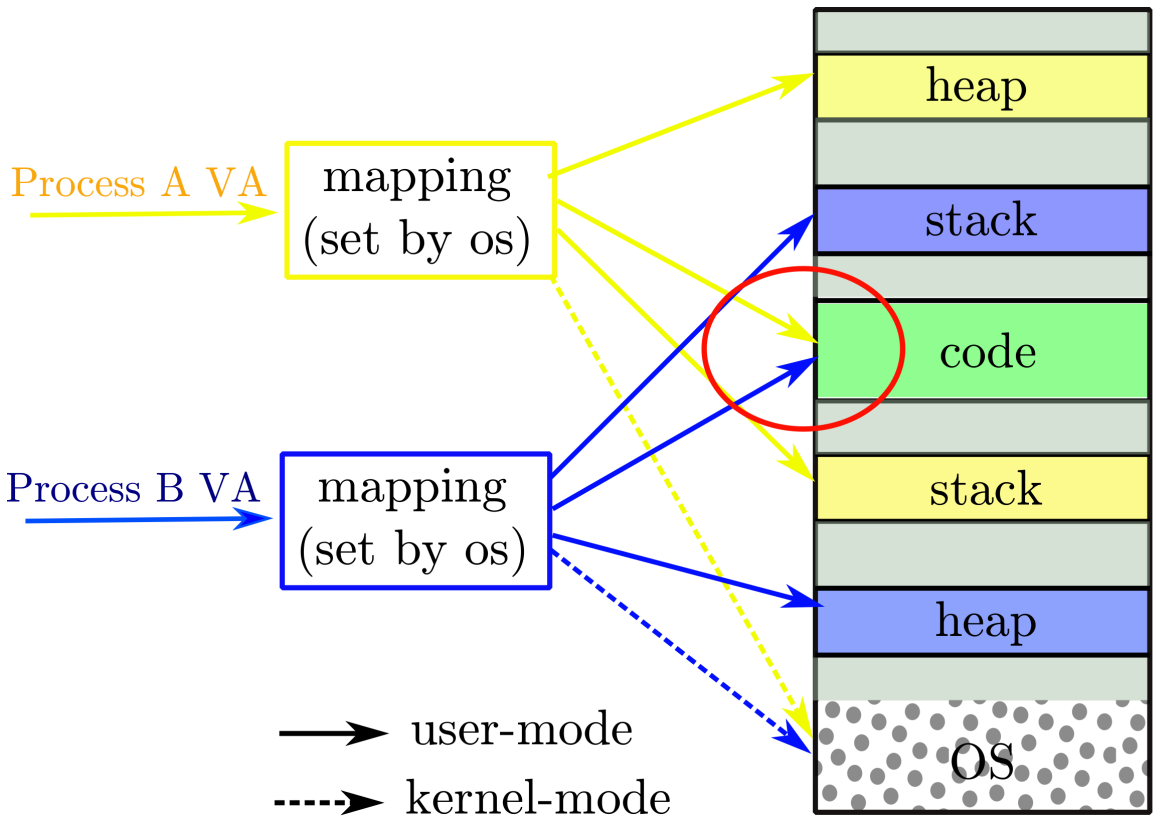
Code Sharing

needs a little more extra hw support

Segment	Base	Size	Grows Positive?	Protection
Code	32K	2K	1	Read-Execute
Heap	34K	3K	1	Read-Write
Stack	28K	2K	0	Read-Write

Segmentation

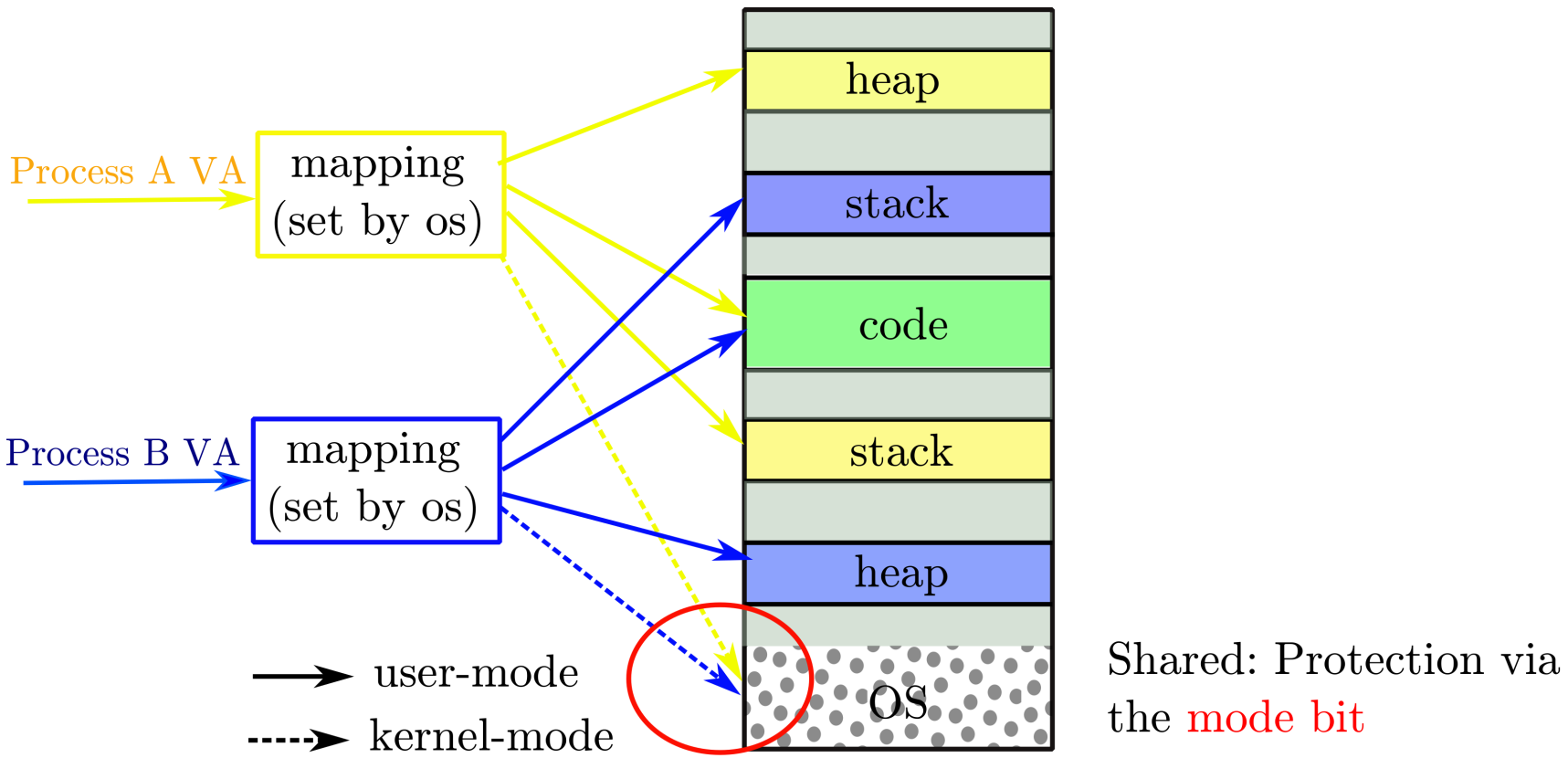
Code Sharing



Shared: Protection via the access control bits

Segmentation

Code Sharing



Segmentation

Pros

- Enables sparse allocation of address space

  Heap: If no data on free list, dynamic memory allocator requests more from OS (e.g., UNIX: `malloc` calls `sbrk()`)

  Stack: OS recognizes reference outside legal segment, extends stack implicitly
- Different protection for different segments
- Enables sharing of selected segments

Cons

- Each segment must be allocated contiguously

  May not have sufficient physical memory for large segments

  External Fragmentation

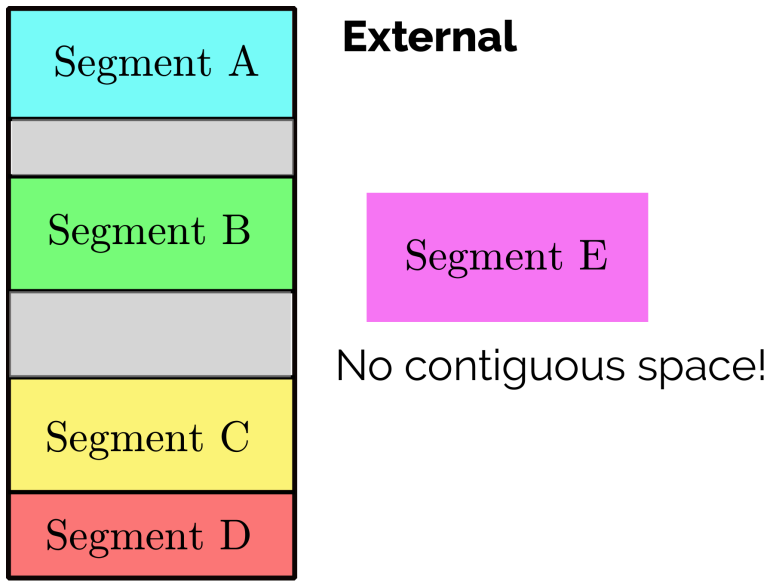
Fragmentation

Definition: Free memory that can't be used

Why?

Free memory (hole) is too small and scattered

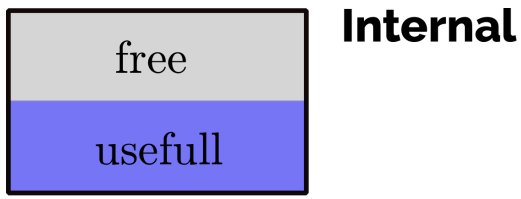
Rules for allocating memory prohibit using this free space



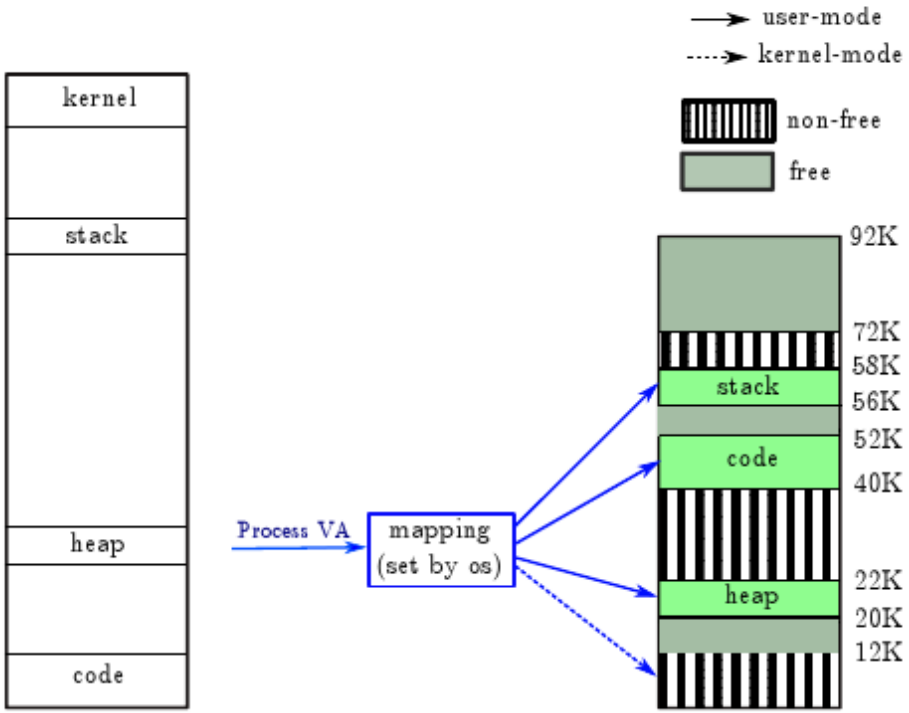
Types of fragmentation

External: Visible to allocator (e.g., OS)

Internal: Visible to requester (e.g., if must allocate at some granularity)



سوال ۱: در یک سیستم عامل فضای آدرس‌دهی هر پردازه ۱۷ بیتی (۱۲۸ کیلو بایتی) است و از روش قطعه‌سازی (segmentation) برای مجازی سازی حافظه استفاده می‌کند (تعداد قطعات برابر با ۴ است). شکل زیر وضعیت حافظه اصلی و همچنین فضای آدرس دهی یک پردازه مشخص را در یک زمان به خصوص نشان می‌دهد (مقیاس‌بندی دقیق بلوک‌های نشان داده شده بر حسب اندازه آنها مد نظر نبوده است).



الف) جدول مربوط به مدیریت حافظه شامل آدرس‌های پایه و اندازه، عملیات مجاز هنگام دسترسی و نحوه رشد هر یک از قطعات code ، stack و heap را تعیین نمایید.

ب) آدرس‌های زیر که توسط پردازه تولید می‌شود به چه آدرس‌های فیزیکی ترجمه می‌شود؟

0x100

0x17810

ج) فرض کنید هر یک از قسمت‌های stack و heap می‌توانند تا 16 KB رشد کنند (البته با این فرض که حافظه ثانویه نداریم می‌بایست در حافظه اصلی امکان اختصاص فضای بیشتر وجود داشته باشد و گرنه پردازه خاتمه می‌یابد). در این صورت اگر برنامه به ترتیب قصد استفاده از آدرس‌های زیر را داشته باشد چه اتفاقی می‌افتد؟ ضمن توضیح اگر در هر مورد مقادیری از جدول مدیریت حافظه تغییر می‌کند آنها را مشخص کنید (به ازای هر مورد حداکثر دو تغییر مقدار مجاز است).

0x14800
0xb000

Review

Match Description

- Time Sharing
- Static Relocation
- Base
- Base + Bounds
- Segmentation

- add per-process starting location to virt addr to obtain phys addr
- one process uses RAM at a time
- many base+bound pairs
- rewrite code before run
- dynamic approach that verifies address is in valid range

Paging

Problem: How to Run Multiple Processes?

Approaches:

- Time Sharing
- Static Relocation
- Base
- Base + Bounds
- Segmentation
- **Paging**

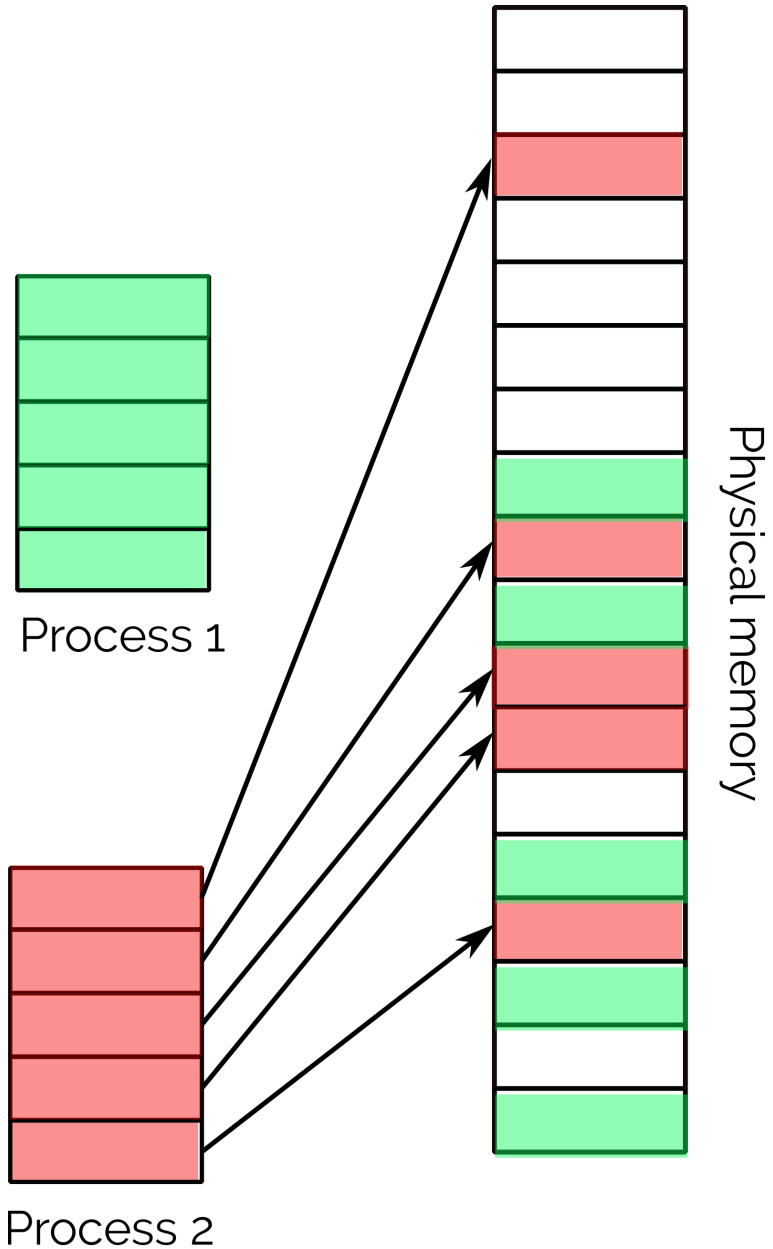
Paging

Idea: Divide address spaces and physical memory into fixed-sized pages

- Size: 2^n, Example: 4KB
- Physical page: page frame

Eliminate the requirement that address space is contiguous

- Eliminate external fragmentation
- Grow segments as needed

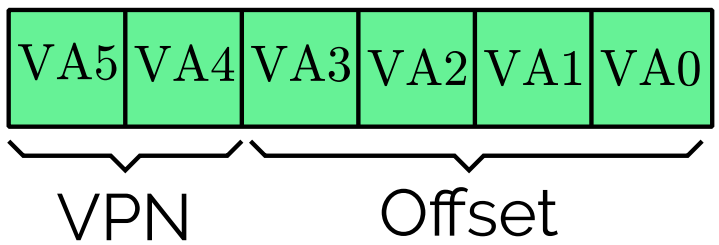


Paging

Translation

For segmentation, high bits ⇒ segment, low bits ⇒ offset

For paging, high bits ⇒ page, low bits ⇒ offset



How many low bits do we need?

Example

Given known page size, how many bits are needed in address to specify offset in page?

Page Size	Low Bits (offset)
16 bytes	4
1 KB	10
1 MB	20
512 bytes	9
4 KB	12

Example

Given number of bits in virtual address and bits for offset, how many bits for virtual page number?

Page Size	Low Bits (offset)	Virt Addr Bits	High Bits (VPN)
16 bytes	4	10	6
1 KB	10	20	10
1 MB	20	32	12
512 bytes	9	16	7
4 KB	12	32	20

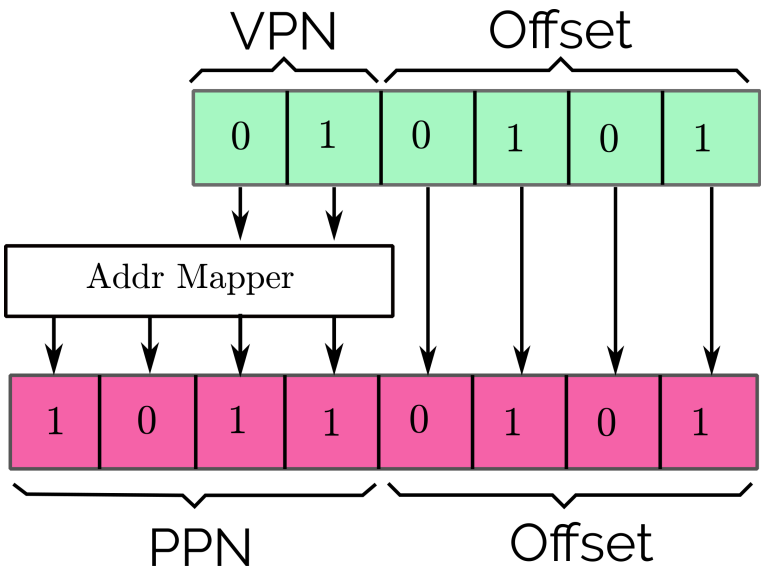
Example

Given number of bits for vpn, how many virtual pages can there be in an address space?

Page Size	Low Bits (offset)	Virt Addr Bits	High Bits (VPN)	Virt Pages
16 bytes	4	10	6	64
1 KB	10	20	10	1 KB
1 MB	20	32	12	4 KB
512 bytes	9	16	7	128
4 KB	12	32	20	1 MB

Paging

Number of bits in virtual address format does not need to equal number of bits in physical address format



How should OS translate VPN to PPN?

For segmentation, OS used a formula (e.g., phys addr = virt_offset + base_reg)

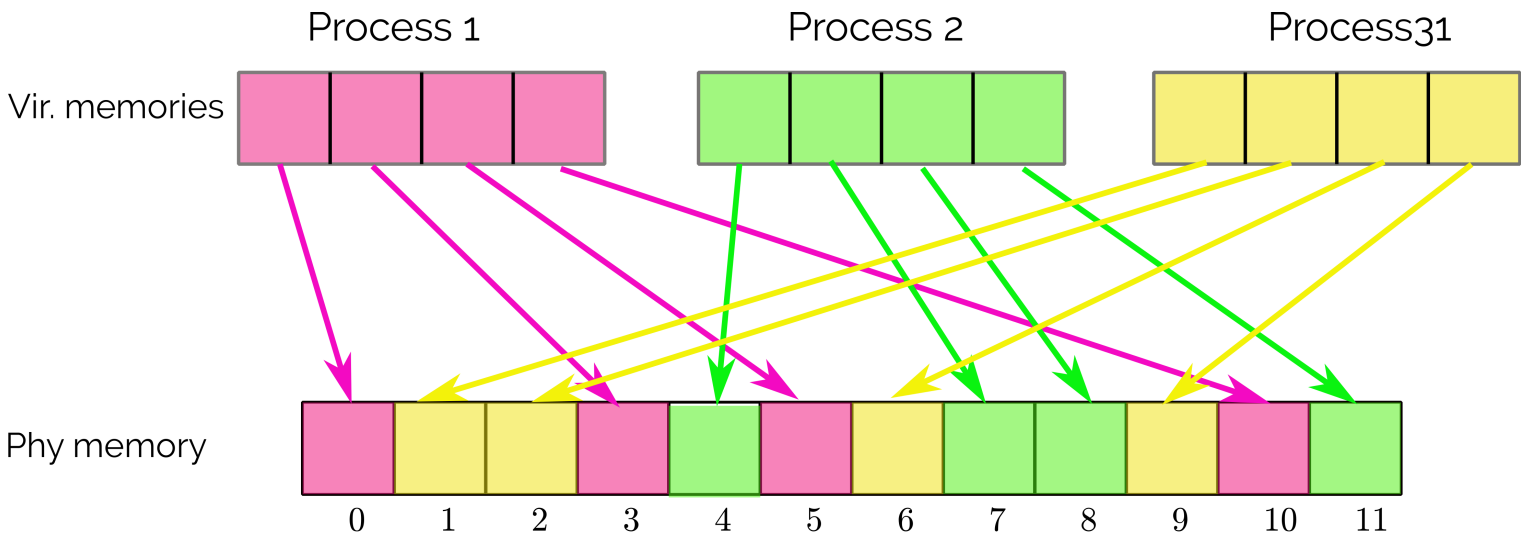
For paging, OS needs more general mapping mechanism

What data structure is good?

Big array: linear page table

More advanced, later!

Paging



Page tables

Process 1	Process 2	Process 3
0	4	1
3	7	2
5	8	6
10	11	9

Paging: too big

Where Are Pagetables Stored?

How big is a typical page table?

assume 32-bit address space

assume 4 KB pages

assume 4 byte entries

umm..., $2 \times 2 = 4, \ldots$ $\Rightarrow$ Final answer: 4 MB (per process) ✔

(400 MB with 100 processes! ...)

Paging: too big

Where Are Pagetables Stored?

They are **BIG**

⇒ Store each page table in memory

Hardware finds page table base with register (e.g., CR3 on x86)

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

Paging: too big

What other info is in page table entries (**PTE**) besides translation?

- ☑☒ **valid bit:** marking all the unused pages in the address space invalid → saving memory
-  **protection bits:** read, write, or execution
-  **present bit:** this page is in physical memory or on disk?
-  **reference bit:** to track whether a page has been accessed (used during page replacement)
-  **dirty bit:** Does this page have been modified since it was brought into memory?

Discussion Question

If we do not have a valid bit, ... <hint: OSTEP> ...?

Paging: too slow

Address Translation

```
VPN     = (VirtualAddress & VPN_MASK) >> SHIFT
PTEAddr = PageTableBaseRegister + (VPN*sizeof(PTE))
PTE = AccessMemory(PTEAddr)
PFN = PTE.PFN
-----------------
offset   = VirtualAddress & OFFSET_MASKPhys
-----------------
Addr = (PFN << SHIFT) | offset
```

Linear page table

Paging: too slow

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

PT
2
0
80
99

- Assume PT is at 0x5000
- Assume PTE's are 4 bytes
- Assume 4KB pages

Paging: too slow

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

PT
2
0
80
99

- Assume PT is at 0x5000
- Assume PTE's are 4 bytes
- Assume 4KB pages

Phy Memory Accesses with segmentation:

5 (3 instrs, 2 movl)

Paging: too slow

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

PT
2
0
80
99

- Assume PT is at 0x5000
- Assume PTE's are 4 bytes
- Assume 4KB pages

Phy Memory Accesses with paging:

Fetch instruction at logical addr 0x0010; vpn?

- Access page table to get ppn for vpn 0
- Mem ref 1: 0x5000
- Learn vpn 0 is at ppn 2
- Fetch instruction at 0x2010 (Mem ref 2)

Exec, load from logical addr 0x1100; vpn?

- Access page table to get ppn for vpn 1
- Mem ref 3: 0x5004
- Learn vpn 1 is at ppn 0
- Movl from 0x0100 into reg (Mem ref 4)

Paging: too slow

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

PT
2
0
80
99

- Assume PT is at 0x5000
- Assume PTE's are 4 bytes
- Assume 4KB pages

Phy Memory Accesses with paging:

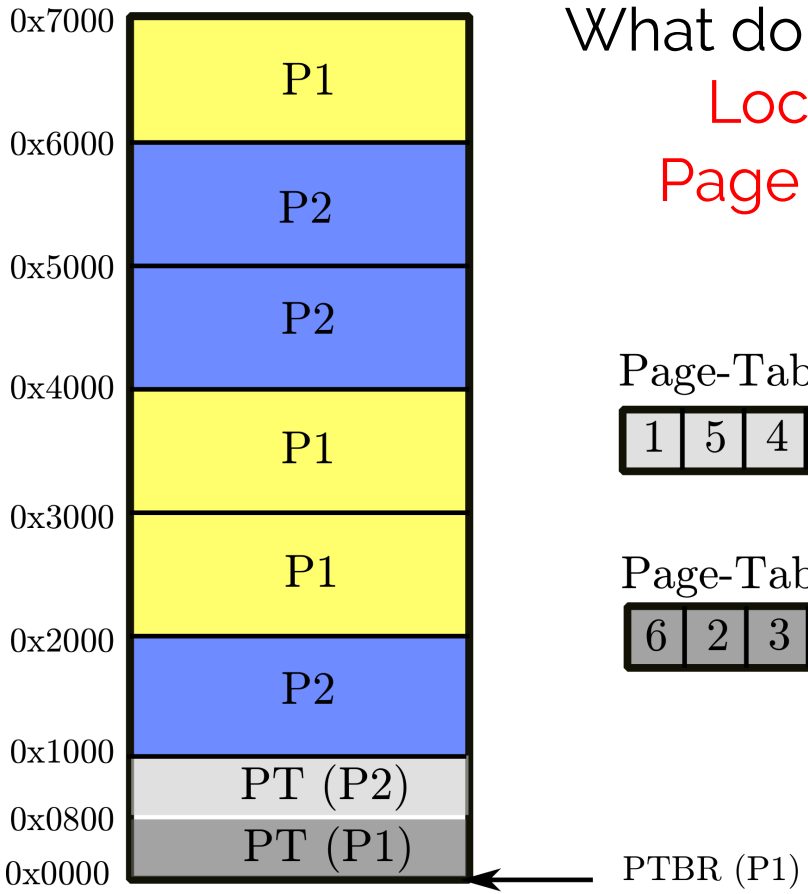
Fetch instruction at logical addr 0x0010; vpn?

Exec, load from logical addr 0x1100; vpn?

Pagetable is slow!!! Doubles memory references

- Mem ref 1: 0x5000
- Learn vpn 0 is at ppn 2
- Fetch instruction at 0x2010 (Mem ref 2)
- Mem ref 3: 0x5004
- Learn vpn 1 is at ppn 0
- Movl from 0x0100 into reg (Mem ref 4)

Paging



What do we need to know?

Location of page table in memory (ptbr)

Page table entity (PTE) size (4 B)

Page-Table (P2)

1	5	4	.....

Page-Table (P1)

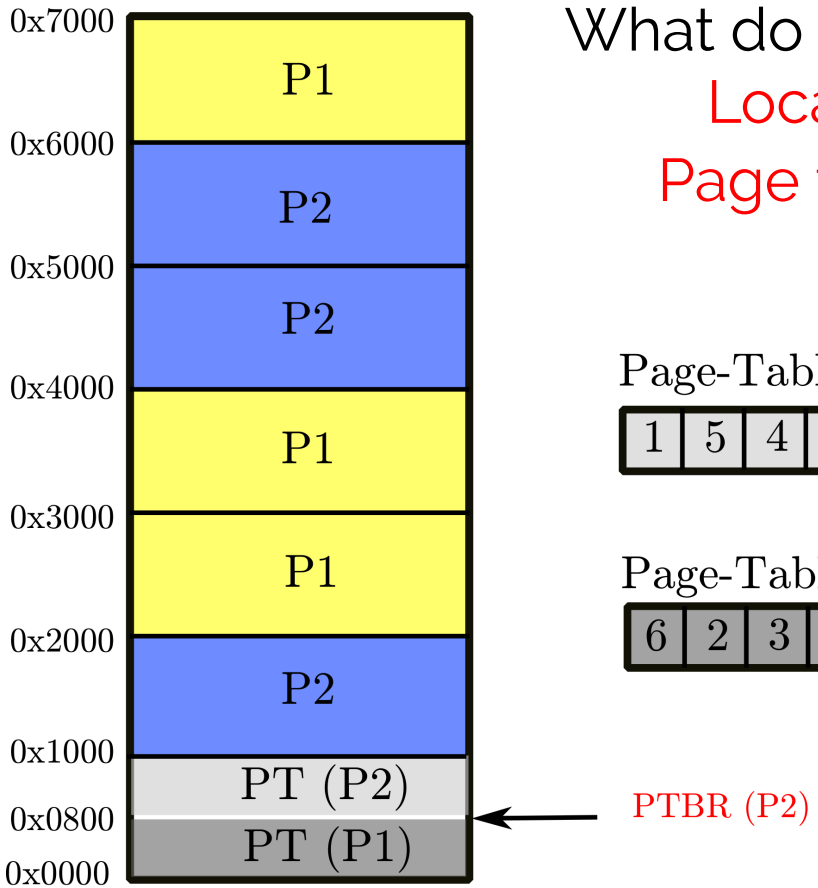
6	2	3	.....

load 0x0000

load 0x1444

load 0x0000
load 0x6000
load 0x0004
load 0x2444

Paging



What do we need to know?

Location of page table in memory (ptbr)

Page table entity (PTE) size (4 B)

Page-Table (P2)

1	5	4	.....

Page-Table (P1)

6	2	3	.....

load 0x0000

load 0x1444

load 0x1444

load 0x0000
load 0x6000
load 0x0004
load 0x2444
load 0x0804
load 0x5444

Paging

Pros

- No external fragmentation
  - don't need to find contiguous RAM
- All free pages are equivalent
  - Easy to manage, allocate, and free pages

Cons

- Page tables are too big
  - Must have one entry for every page of address space
- Accessing page tables is too slow [**today's focus**]
  - Doubles number of memory references per instruction

Translation Look-aside Buffer (TLB)

Paging Translation Steps

H/W: for each mem reference:

- ¢ extract VPN (virt page num) from VA (virt addr)
- ¢ calculate addr of PTE (page table entry)
- $ read PTE from memory — we will avoid in today's lecture
- ¢ extract PFN (page frame num)
- ¢ build PA (phys addr)
- $ read contents of PA from memory into register

Example: Array Iterator

```
int sum = 0;
for (i=0; i<N; i++){
    sum += a[i];
}
```

Assume 'a' starts at 0x3000

Ignore instruction fetches

ptbr: 0x1000; PTE 4 bytes each

VPN 3 $\rightarrow$ PPN 7

load 0x3000
load 0x100C
load 0x7000
load 0x3004
load 0x100C
load 0x7004
load 0x3008
load 0x100C
load 0x7008
load 0x300C
load 0x100C
load 0x700C
...

Example: Array Iterator

```
int sum = 0;
for (i=0; i<N; i++){
sum += a[i];
}
```

Assume 'a' starts at 0x3000

Ignore instruction fetches

ptbr: 0x1000; PTE 4 bytes each

VPN 3 $\rightarrow$ PPN 7

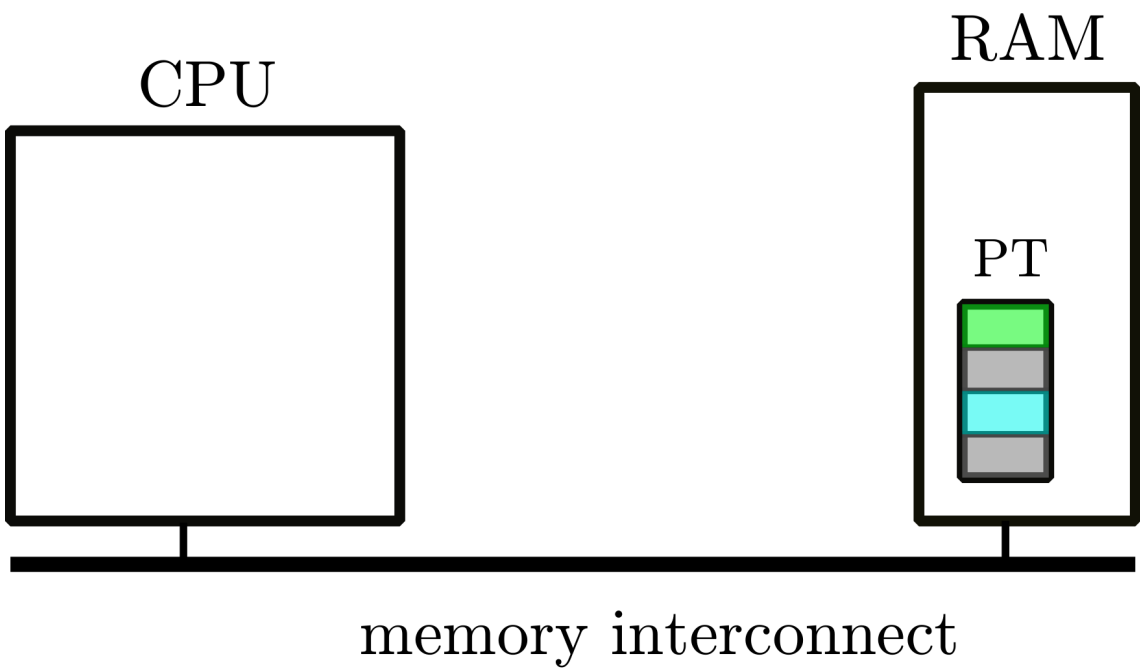
load 0x3000
load 0x100C
load 0x7000
load 0x3004
load 0x100C
load 0x7004
load 0x3008
load 0x100C
load 0x7008
load 0x300C
load 0x100C
load 0x700C
...

Observation: Repeatedly access same PTE because program repeatedly accesses same virtual page

Strategy: Cache Page Translations

Take advantage of repetition.

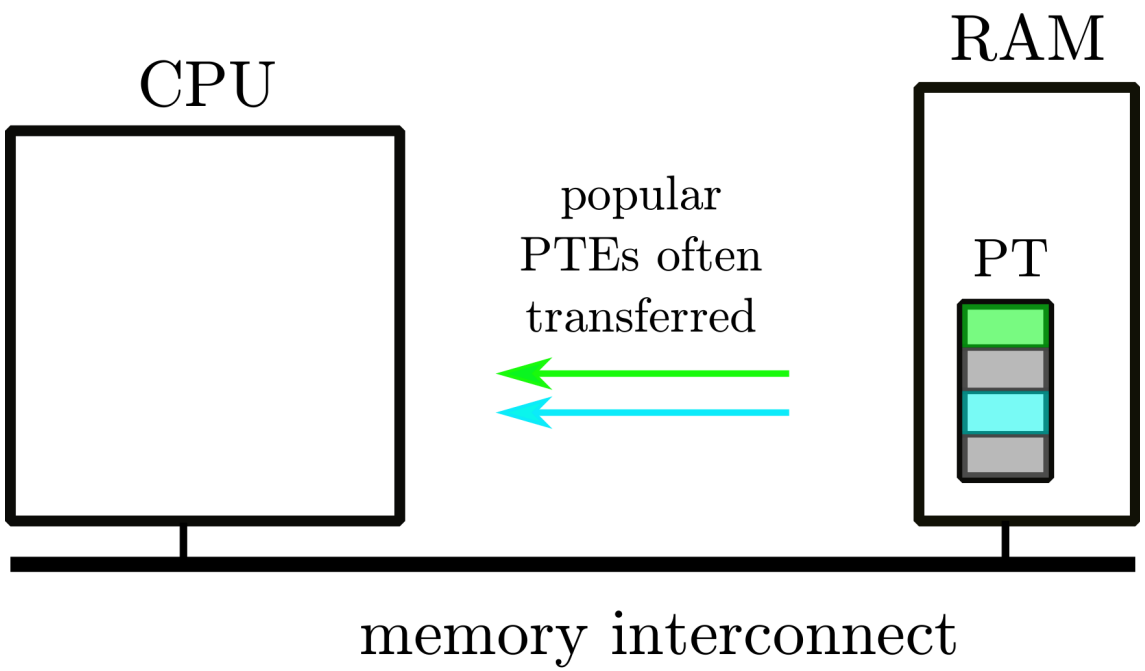
Use a CPU cache.



Strategy: Cache Page Translations

Take advantage of repetition.

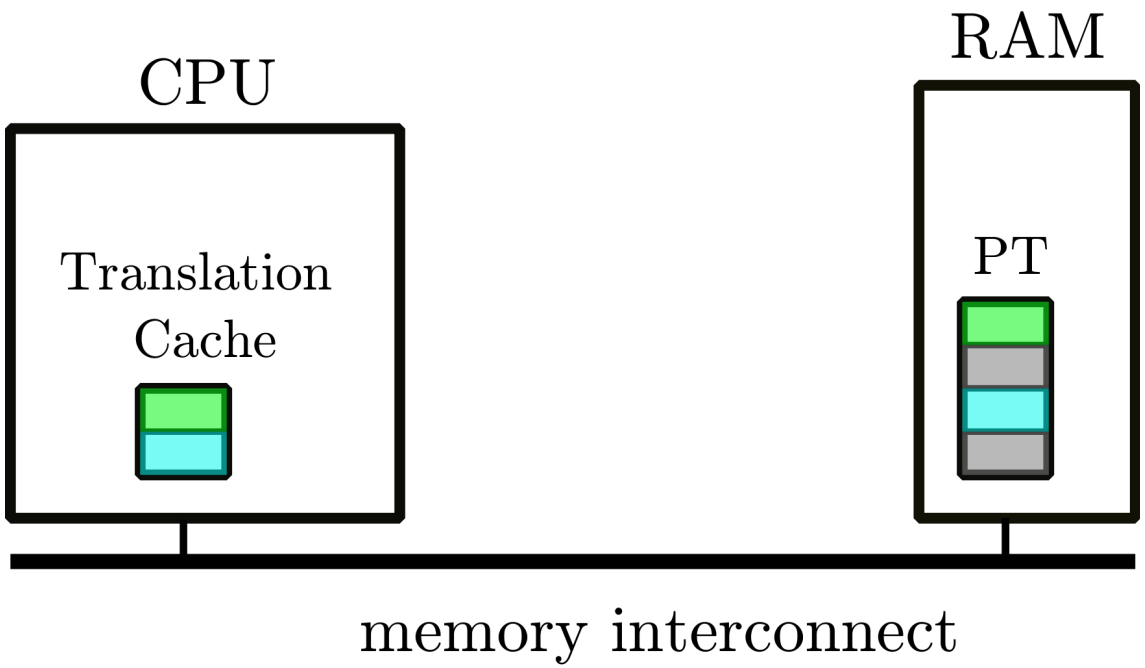
Use a CPU cache.



Strategy: Cache Page Translations

Take advantage of repetition.

Use a CPU cache.



TLB: Translation Lookaside Buffer

Example: Array Iterator with TLB

```
int sum = 0;
for (i=0; i<2048; i++){
sum += a[i];
}
```

Array Iterator

Virt

load 0x1000

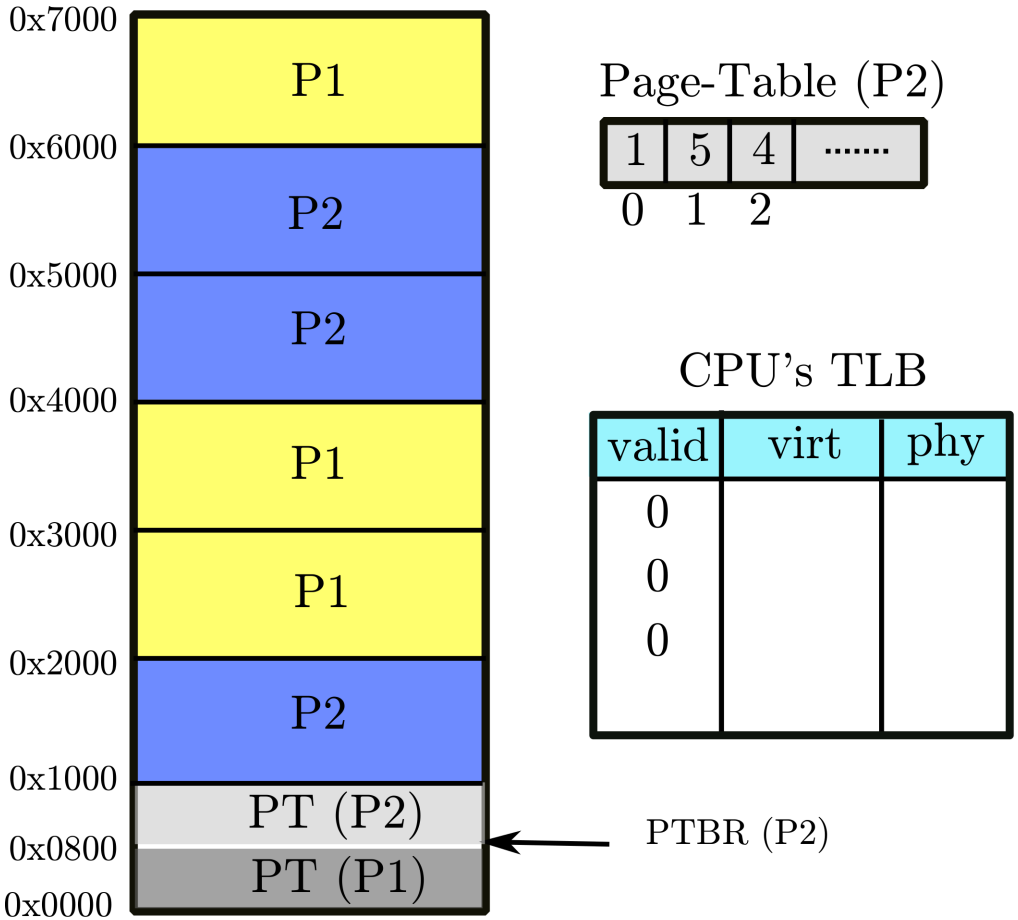
load 0x1004

load 0x1008

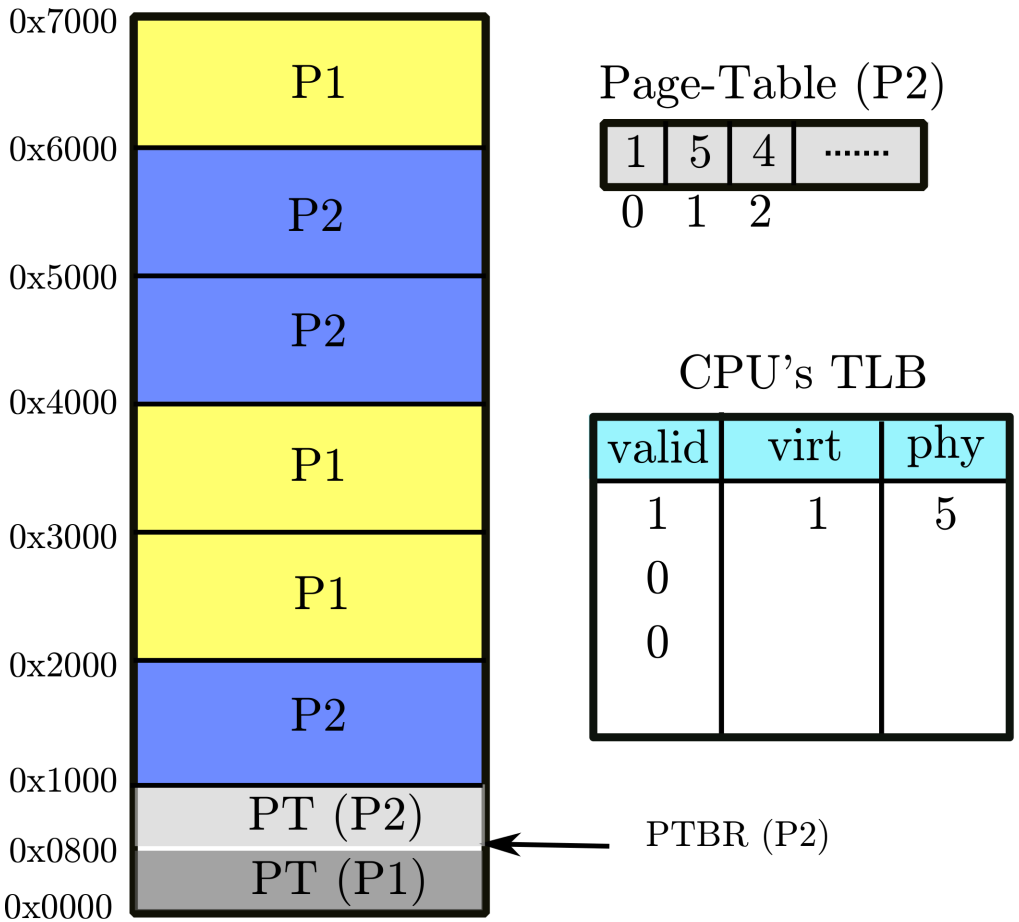
load 0x100C

...

Example: Array Iterator with TLB



Example: Array Iterator with TLB



Page-Table (P2)

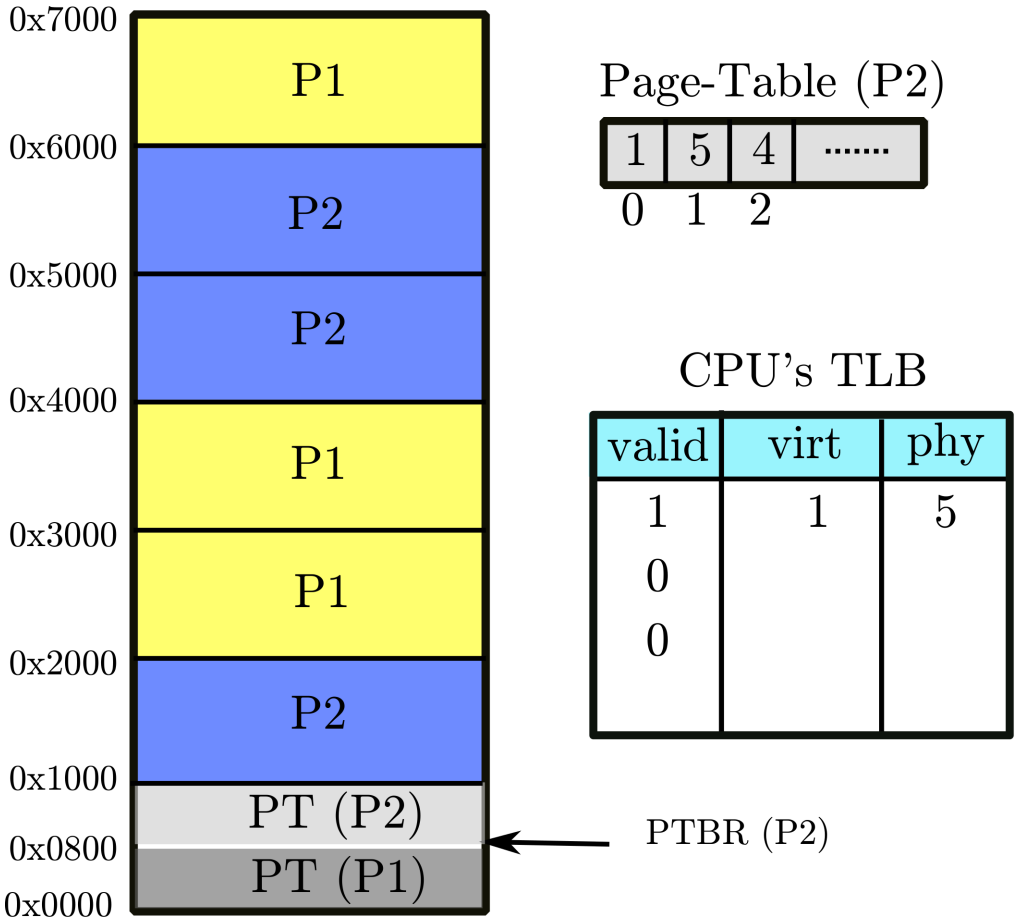
1	5	4	.....
0	1	2	

CPU's TLB

valid	virt	phy
1	1	5
0		
0		

Virt	Phy
load 0x1000	load 0x0804
load 0x1004	
load 0x1008	
load 0x100C	
...	

Example: Array Iterator with TLB



Page-Table (P2)

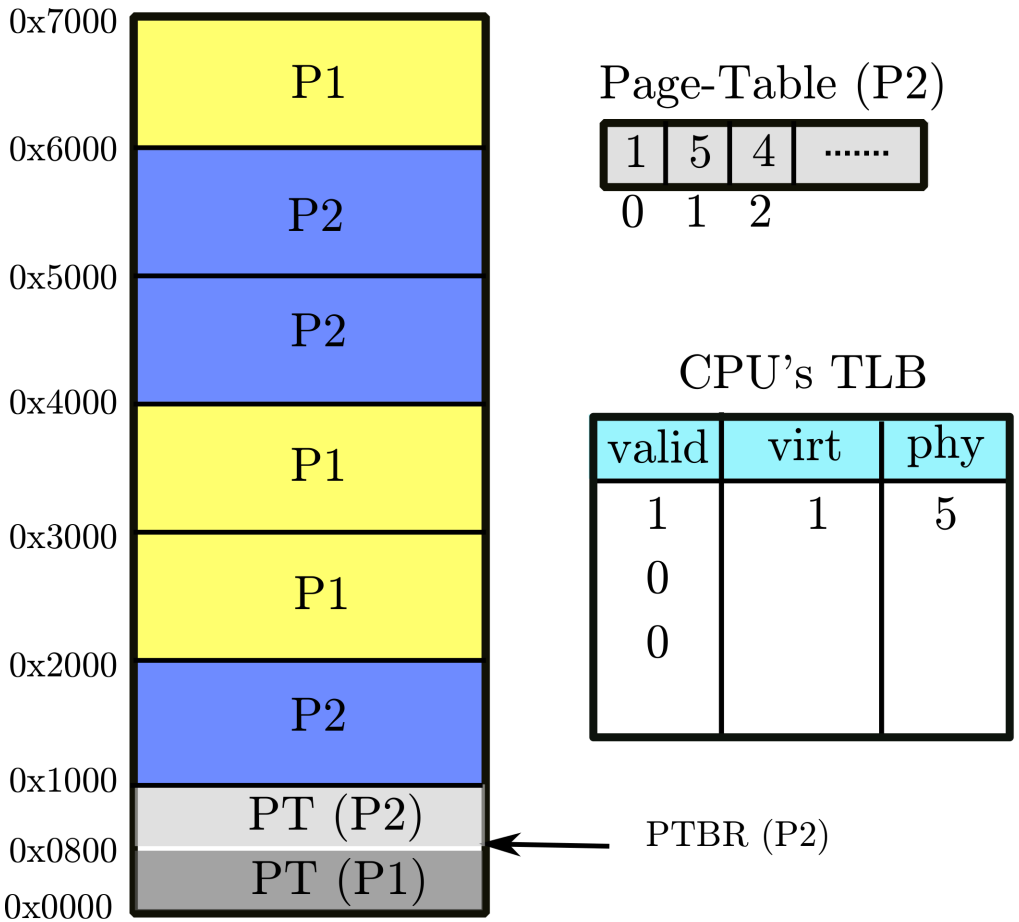
1	5	4	.....
0	1	2	

CPU's TLB

valid	virt	phy
1	1	5
0		
0		

Virt	Phy
load 0x1000	load 0x0804
	load 0x5000
load 0x1004	
load 0x1008	
load 0x100C	
...	

Example: Array Iterator with TLB



Page-Table (P2)

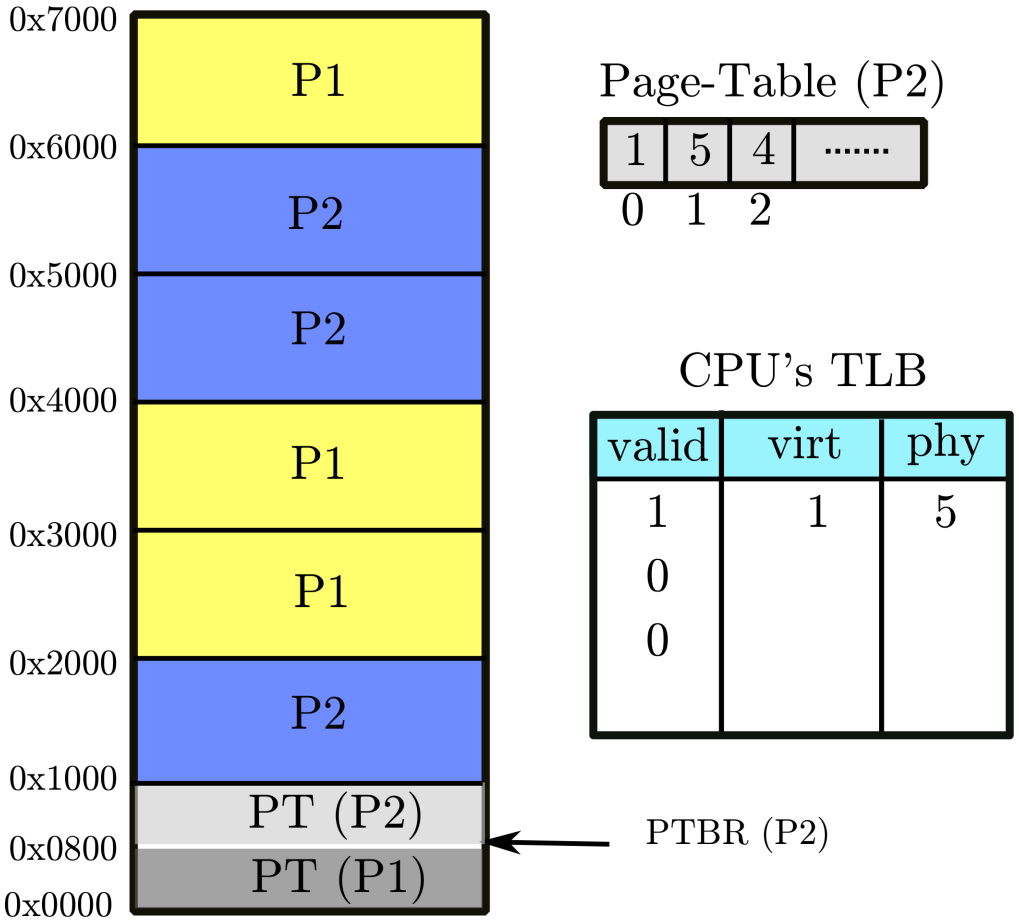
1	5	4	.....
0	1	2	

CPU's TLB

valid	virt	phy
1	1	5
0		
0		

Virt	Phy
load 0x1000	load 0x0804
	load 0x5000
load 0x1004	(TLB)
load 0x1008	
load 0x100C	
...	

Example: Array Iterator with TLB



Page-Table (P2)

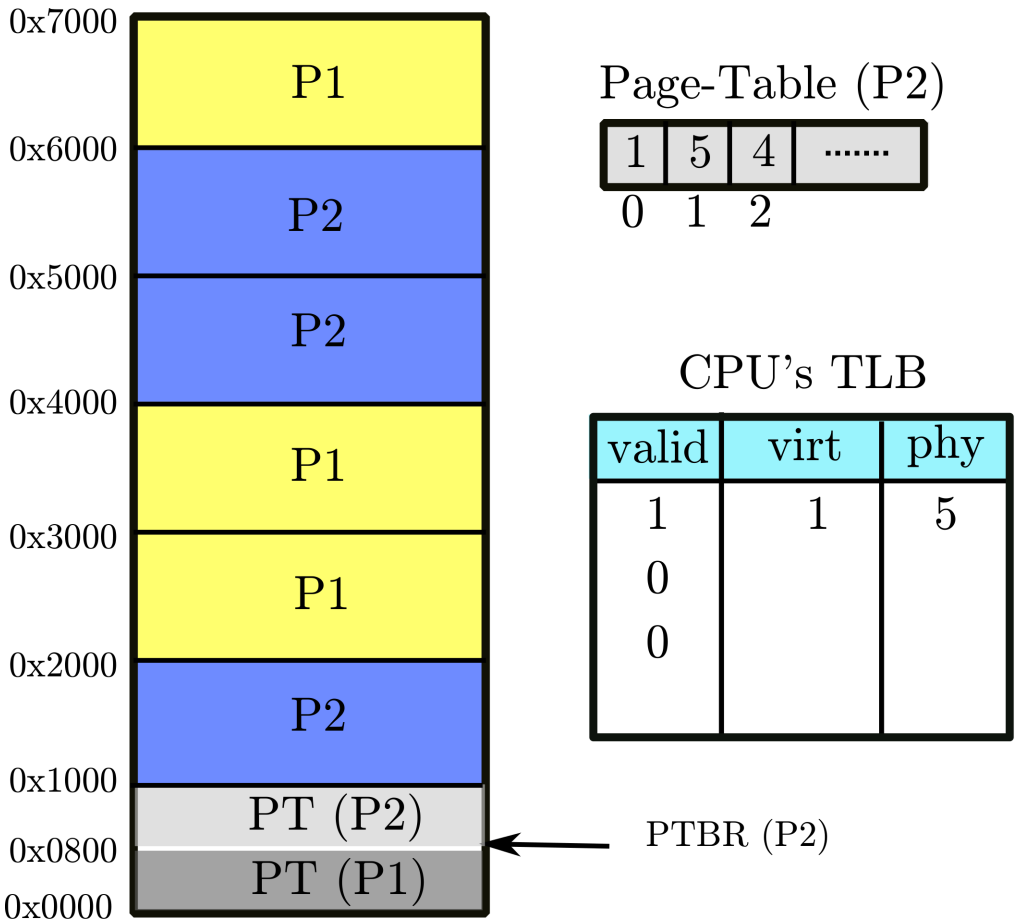
1	5	4	.....
0	1	2	

CPU's TLB

valid	virt	phy
1	1	5
0		
0		

Virt	Phy
load 0x1000	load 0x0804
	load 0x5000
load 0x1004	(TLB)
	load 0x5004
load 0x1008	
load 0x100C	
...	

Example: Array Iterator with TLB



Page-Table (P2)

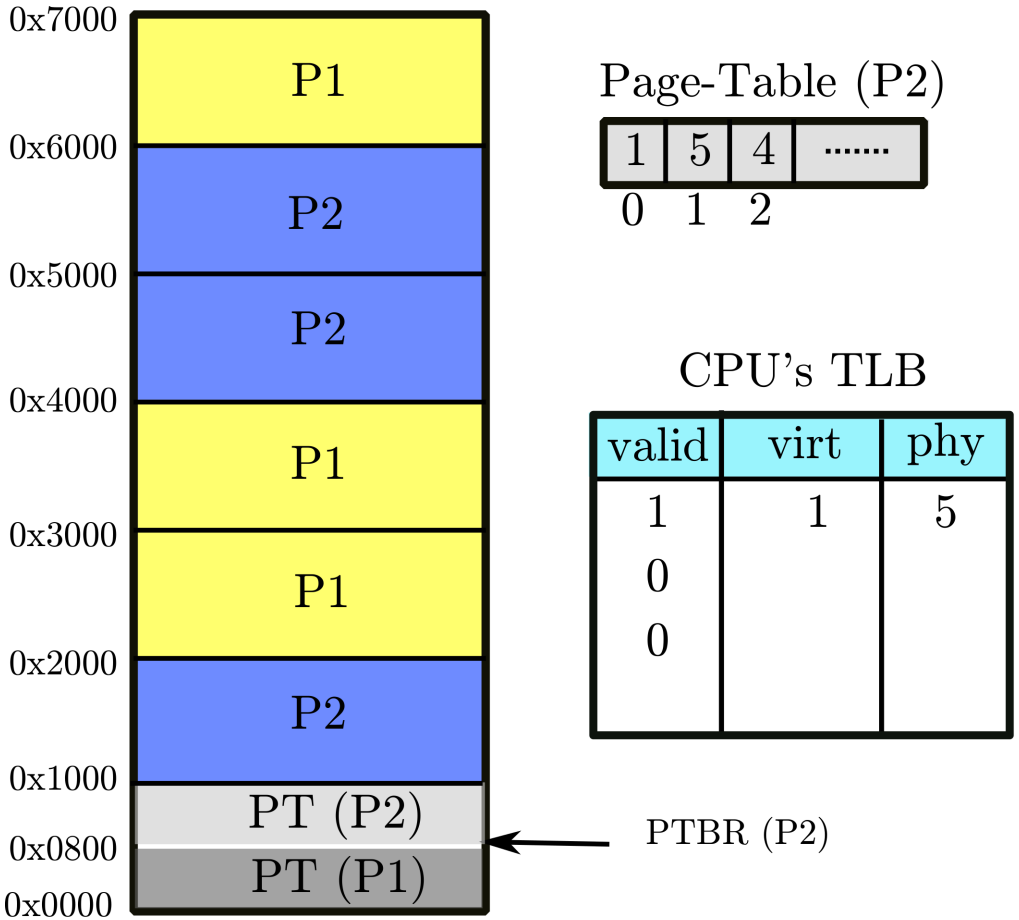
1	5	4	.....
0	1	2	

CPU's TLB

valid	virt	phy
1	1	5
0		
0		

Virt	Phy
load 0x1000	load 0x0804
	load 0x5000
load 0x1004	(TLB)
	load 0x5004
load 0x1008	(TLB)
	load 0x5008
load 0x100C	(TLB)
	load 0x500C
...	

Example: Array Iterator with TLB



Page-Table (P2)

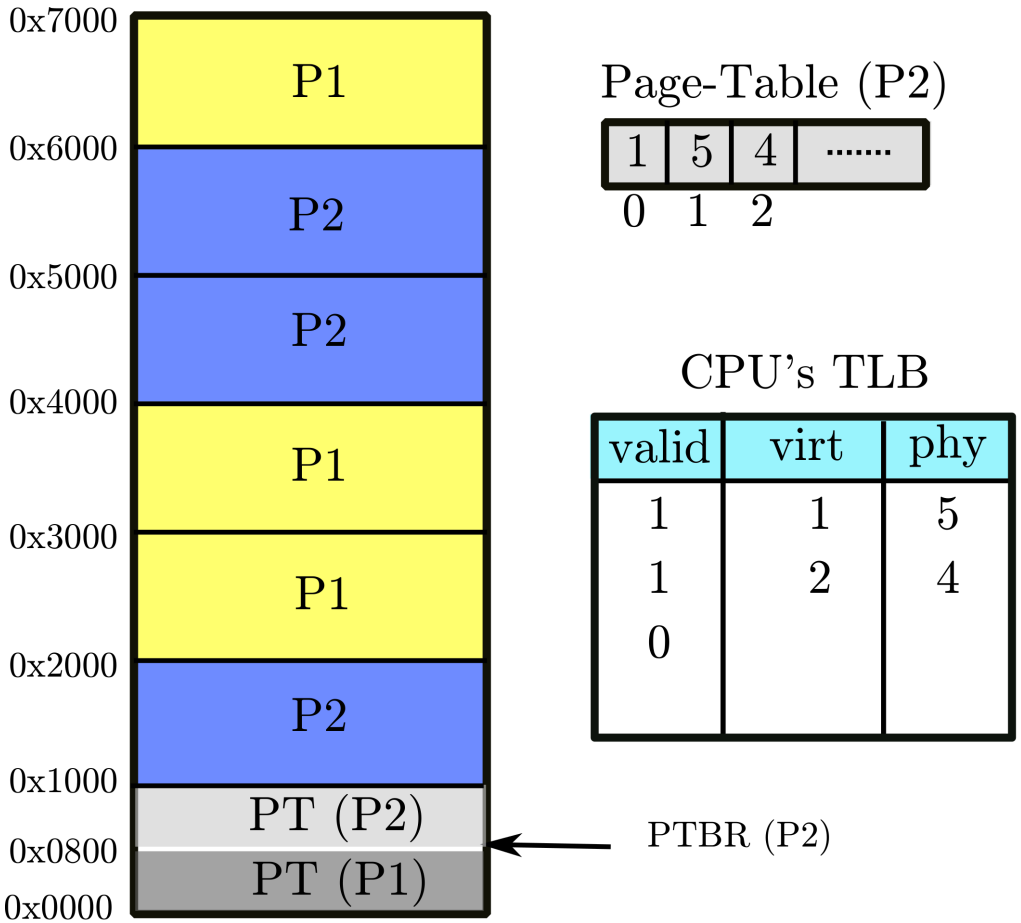
1	5	4	.....
0	1	2	

CPU's TLB

valid	virt	phy
1	1	5
0		
0		

Virt	Phy
load 0x1000	load 0x0804
	load 0x5000
load 0x1004	(TLB)
	load 0x5004
load 0x1008	(TLB)
	load 0x5008
load 0x100C	(TLB)
	load 0x500C
...	
load 0x2000	

Example: Array Iterator with TLB



Page-Table (P2)

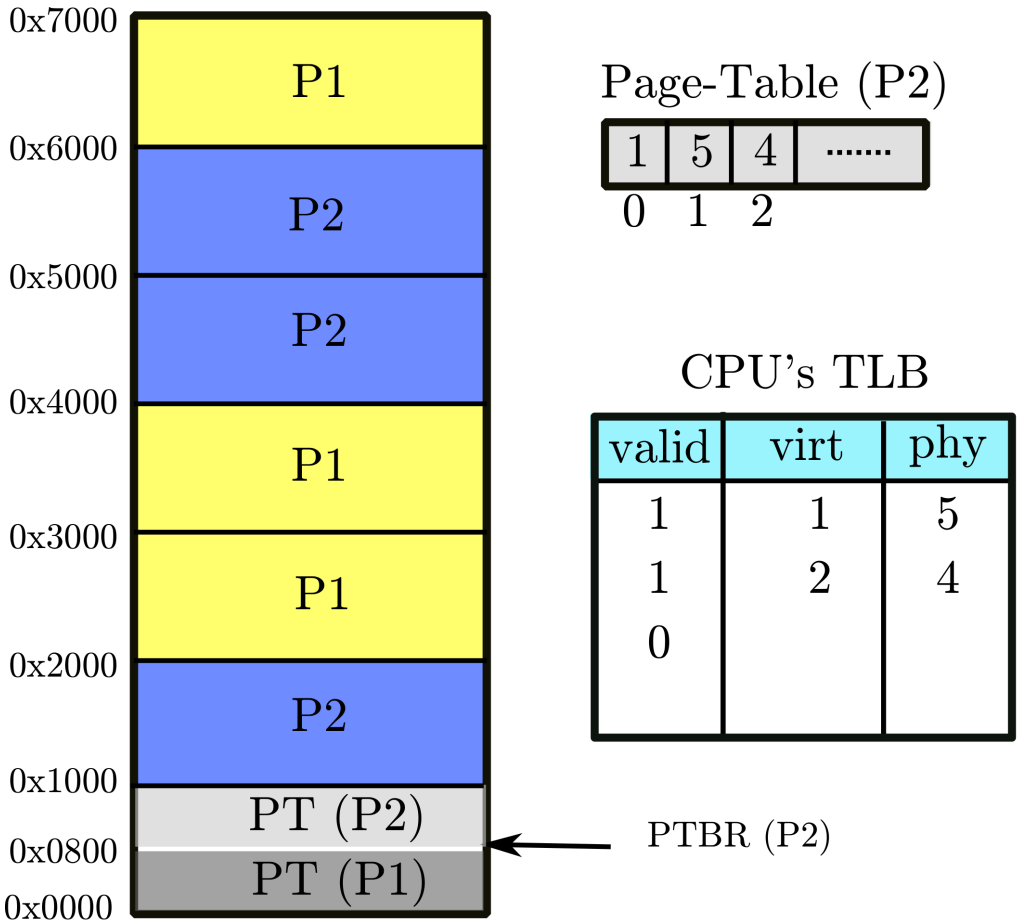
1	5	4	.....
0	1	2	

CPU's TLB

valid	virt	phy
1	1	5
1	2	4
0		

Virt	Phy
load 0x1000	load 0x0804
	load 0x5000
load 0x1004	(TLB)
	load 0x5004
load 0x1008	(TLB)
	load 0x5008
load 0x100C	(TLB)
	load 0x500C
...	...
load 0x2000	load 0x0808

Example: Array Iterator with TLB



Page-Table (P2)

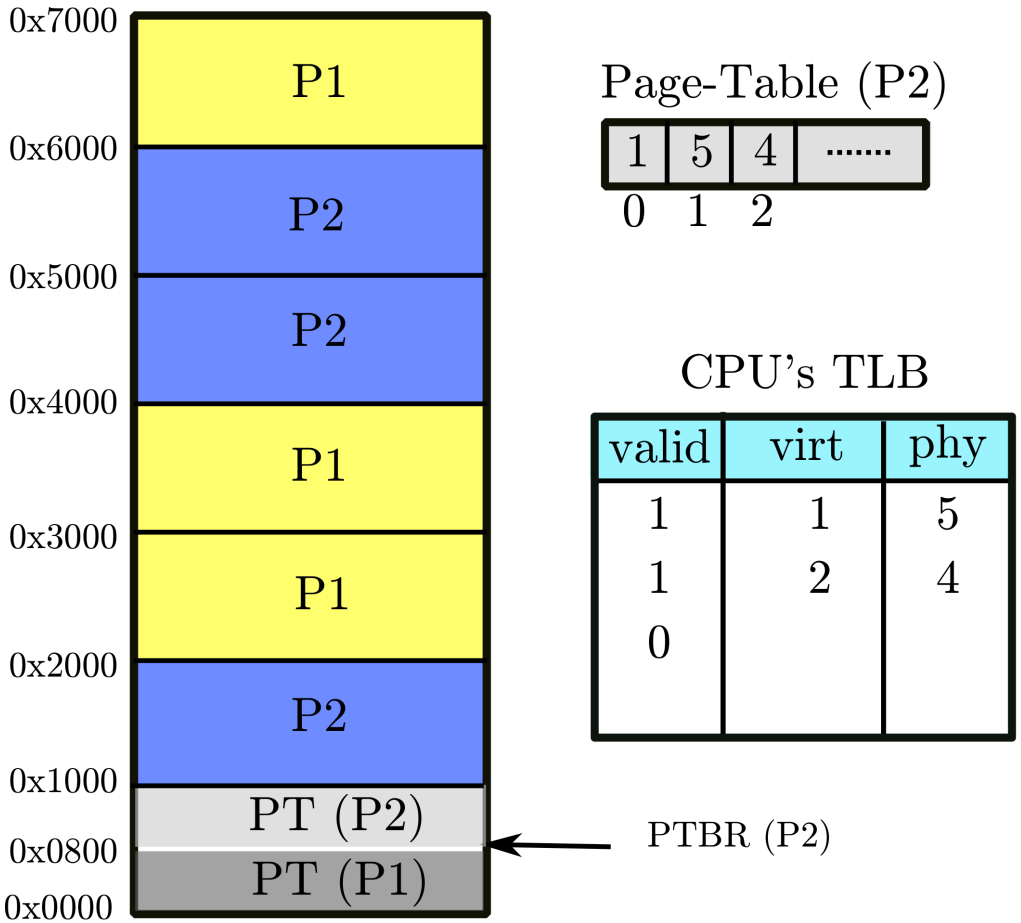
1	5	4	.....
0	1	2	

CPU's TLB

valid	virt	phy
1	1	5
1	2	4
0		

Virt	Phy
load 0x1000	load 0x0804
	load 0x5000
load 0x1004	(TLB)
	load 0x5004
load 0x1008	(TLB)
	load 0x5008
load 0x100C	(TLB)
	load 0x500C
...	...
load 0x2000	load 0x0808
	load 0x4000

Example: Array Iterator with TLB



Page-Table (P2)

1	5	4	.....
0	1	2	

CPU's TLB

valid	virt	phy
1	1	5
1	2	4
0		

Virt	Phy
load 0x1000	load 0x0804
	load 0x5000
load 0x1004	(TLB)
	load 0x5004
load 0x1008	(TLB)
	load 0x5008
load 0x100C	(TLB)
	load 0x500C
...	...
load 0x2000	load 0x0808
	load 0x4000
load 0x2004	(TLB)
	load 0x4004

TLB Performance?

```
int sum = 0;
for (i=0; i<2048; i++){
sum += a[i];
}
```

(assume 4KB pages)
Calculate miss rate of TLB for data: # TLB misses / # TLB lookups

- # TLB lookups?
  = number of accesses to a = 2048
- # TLB misses?
  = number of unique pages accessed
  = 2048/(elements of 'a' per 4K page)
  = 2K/(4K/sizeof(int)) = 2K/1K =2 (if a%4096 is 0, else ?)
- Miss rate: 2/2048 = 0.1%
- Hit rate (1 - miss rate): 99.9%

Remark 1

Q: How can system improve TLB performance (hit rate) given fixed number of TLB entries?

A: Increase page size

Fewer unique page translations needed to access same amount of memory

Remark 2

Locality does matter!

Both Spatial Locality & Temporal Locality

Near future access will be to nearby addresses

Who Handles The TLB Miss? H/W or OS?

H/W: CPU must know where page-tables are

- CR3 register on x86
- Page-table structure fixed and agreed upon between HW and OS
- HW "walks" the page-table and fills TLB

OS: CPU traps into OS upon TLB miss

- "Software-managed TLB"
- OS interprets page-tables as it chooses
- Modifying TLB entries is privileged
  - otherwise what could process do?

Need same protection bits in TLB as pagetable

- rwx

Replacement Policy

LRU: evict Least-Recently Used TLB slot when needed

tries to take advantage of locality in the memory-reference stream

Random: Evict randomly chosen entry

useful due to its simplicity and ability to avoid corner-case behaviors

Exercise: Specify a scenario in which using LRU, we get a TLB miss on each memory access (i.e., hit rate $= 0$)

Context Switch

What happens if a process uses cached TLB entries from another process?

Solutions?

1- Flush TLB on each switch (sets all valid bits to 0)

Costly; lose all recently cached translations

2- Track which entries are for which process

Address Space Identifier (ASID)

Tag each TLB entry with an ASID

Like PID but usually with fewer bits (e.g., 8 instead of 32)

Example

VPN	PFN	valid	prot	ASID
10	100	1	rwx	1
–	–	0	–	–
10	170	1	rwx	2

Example

VPN	PFN	valid	prot	ASID
10	101	1	rx	1
–	–	0	–	–
50	101	1	rx	2

Code sharing!

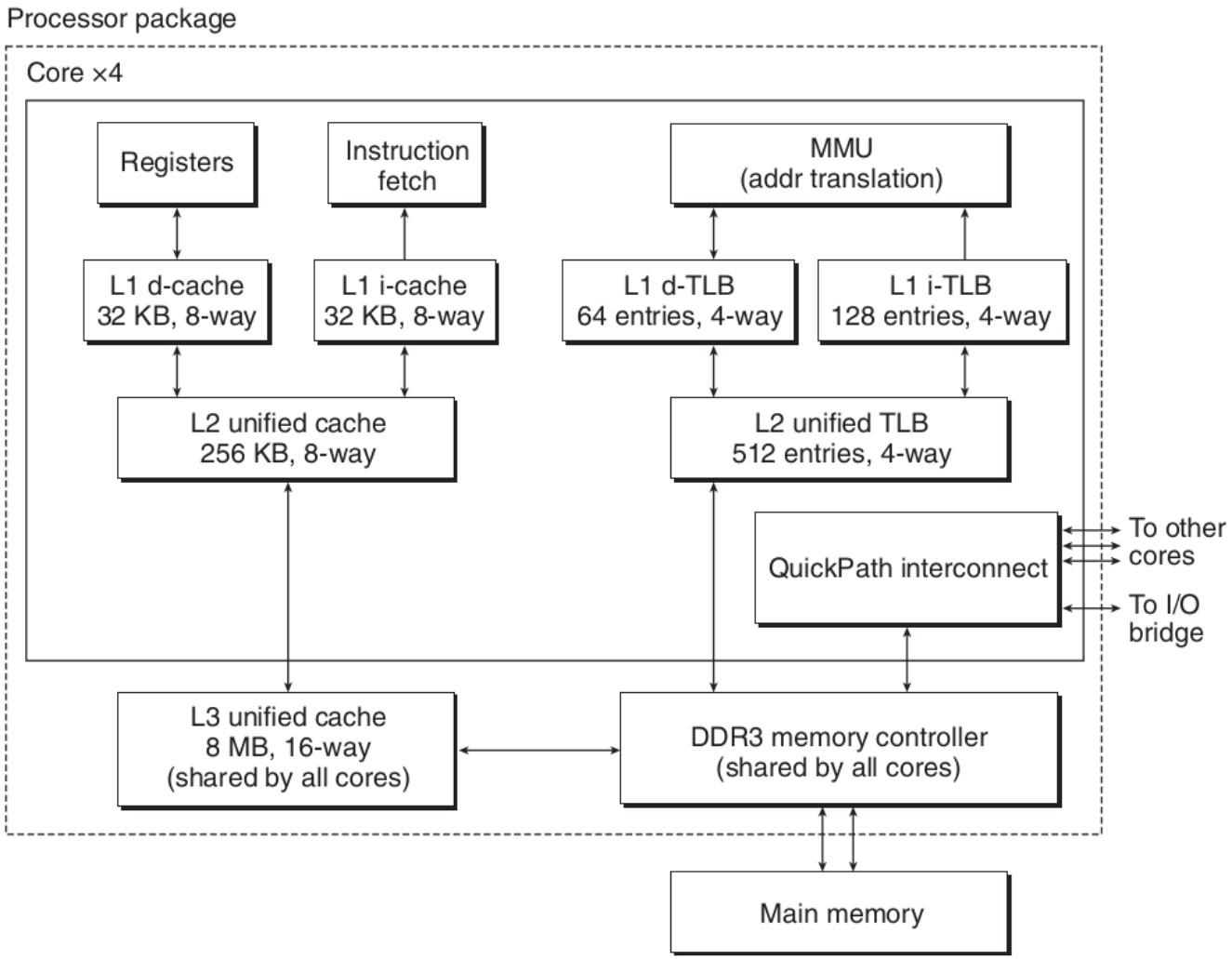
▶ OS must, on a context switch, set some privileged register to the ASID of the current process.

Note

Context switches are expensive
Even with ASID, other processes "pollute" TLB

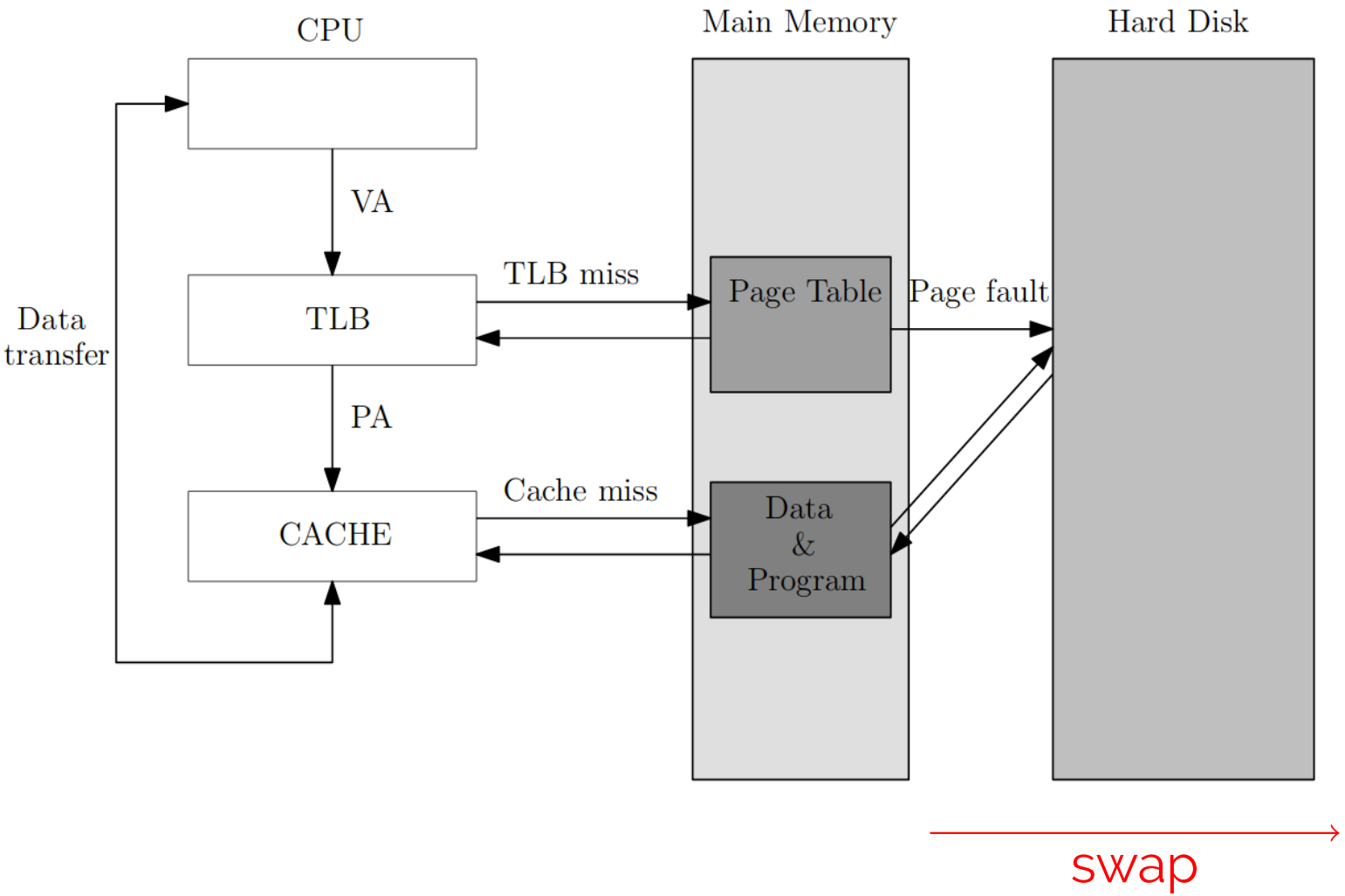
- TLB is small; number of TLB entries ≈ 32, 64, and some may be even reserved for OS!
- Discard process A's TLB entries for process B's entries

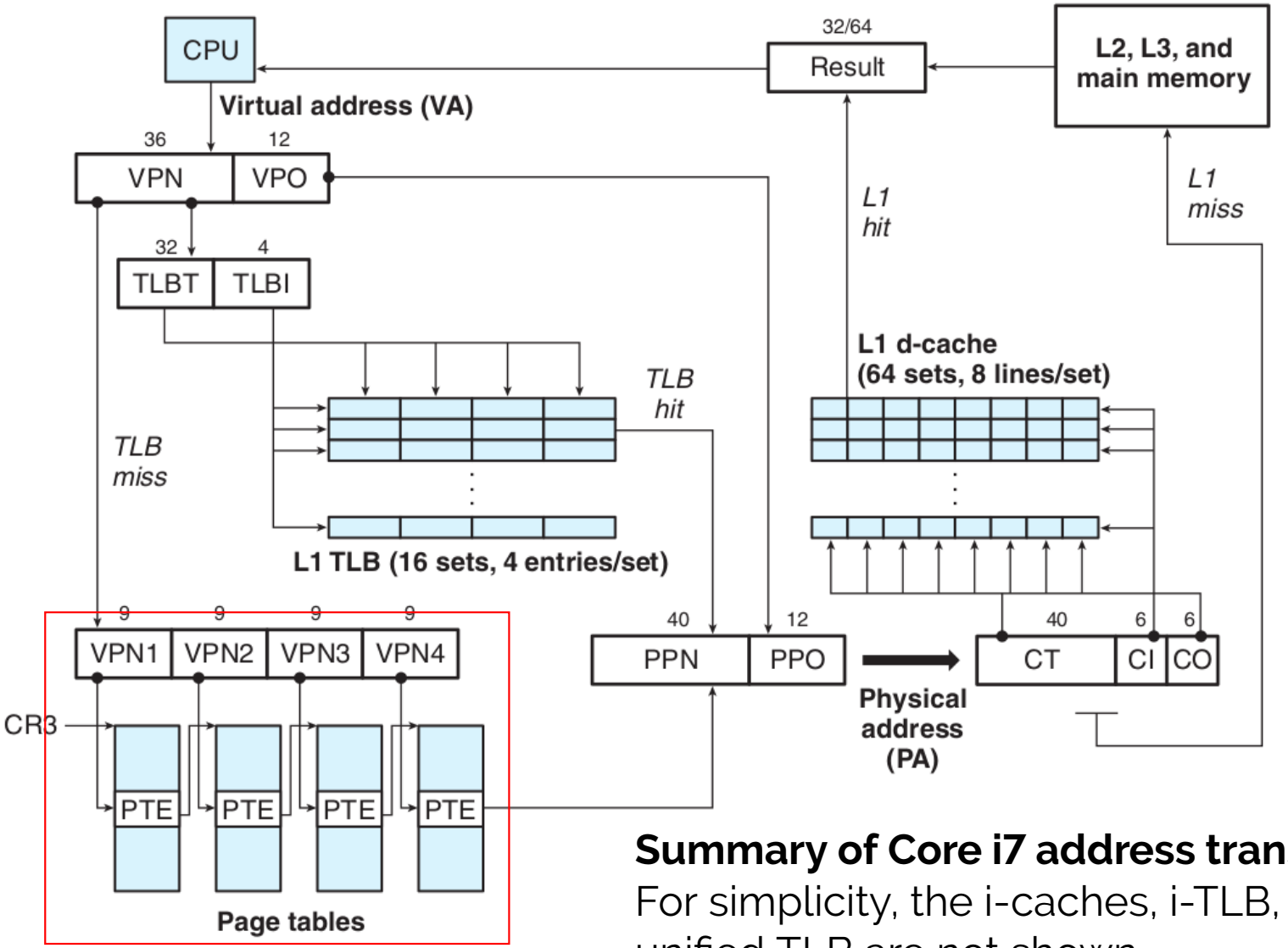
Memory Hierarchy
Revisited (TLB included)



Memory Hierarchy

Revisited (TLB included)





Summary of Core i7 address translation. For simplicity, the i-caches, i-TLB, and L2 unified TLB are not shown.

Multilevel paging

قطعه کد زیر را در نظر بگیرید:

```
1 int a[1024][1024], b[1024][1024], c[1024][1024];
2 multiply() {
3 int i, j, k;
4 for (i = 0; i < 1024; i++)
5   for (j = 0; j < 1024; j++)
6     for (k = 0; k < 1024; k++)
7       c[i][j] += a[i,k] * b[k,j];
8 }
```

فرض کنید که فایل باینری اجرایی این برنامه در یک صفحه و قسمت پشته (stack) نیز در یک صفحه جایگذاری شده است. همچنین اندازه هر عدد صحیح را ۴ بایت و تعداد رکوردهای TLB را ۸ عدد در نظر بگیرید که همواره ترجمه شماره صفحات مجازی که نسبت به سایر صفحات جدیدتر استفاده شده‌اند را در برمی‌گیرند. در این صورت تعداد فقدان‌های TLB هنگام اجرای این برنامه چه تعداد است؟ (فرض کنید در ابتدای کار TLB خالی است)

Smaller Pages

Problem

- 32 bit virtual address space, 4KB page sizes $\rightarrow$ a page table of 2^{20} entities
- if each entity is 4 bytes $\rightarrow$ 4MB memory to store the page table of **one** process
- With hundreds of active processes, hundreds of MB just for page tables!

Simple Solution: bigger page sizes

Does not really work due to internal fragmentation

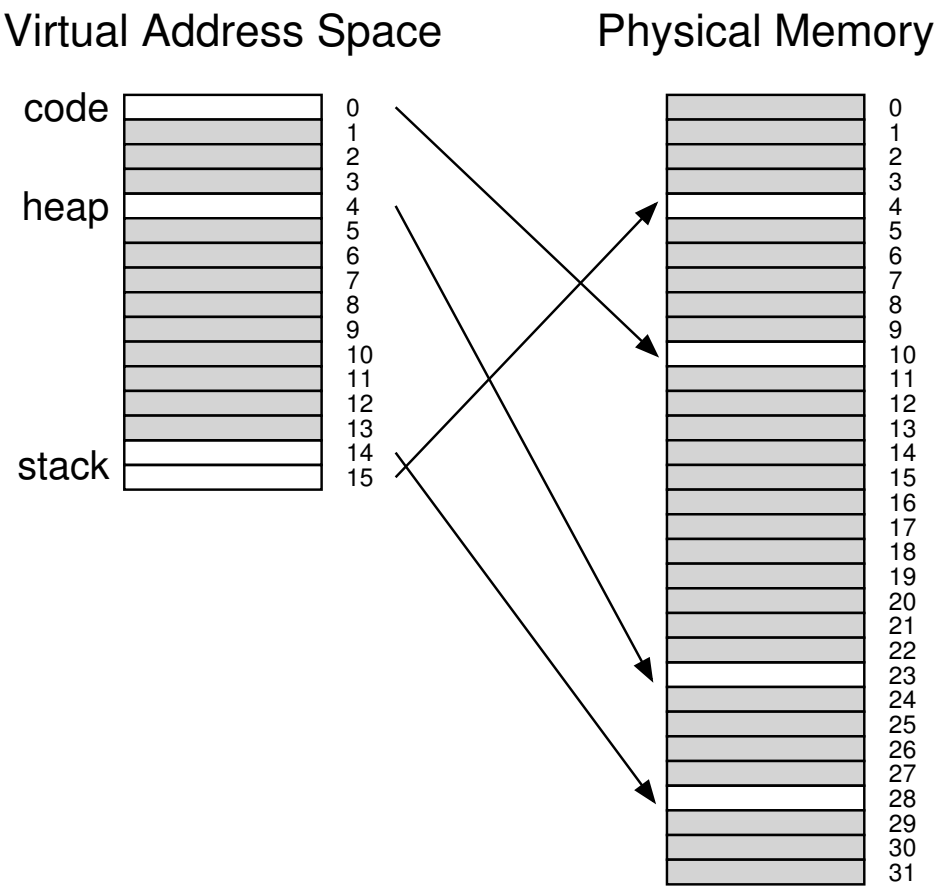
Therefore, small page sizes are used in practice

- 4KB in x86, 8 KB in SPARCv9

Hybrid Approach: Paging and Segments

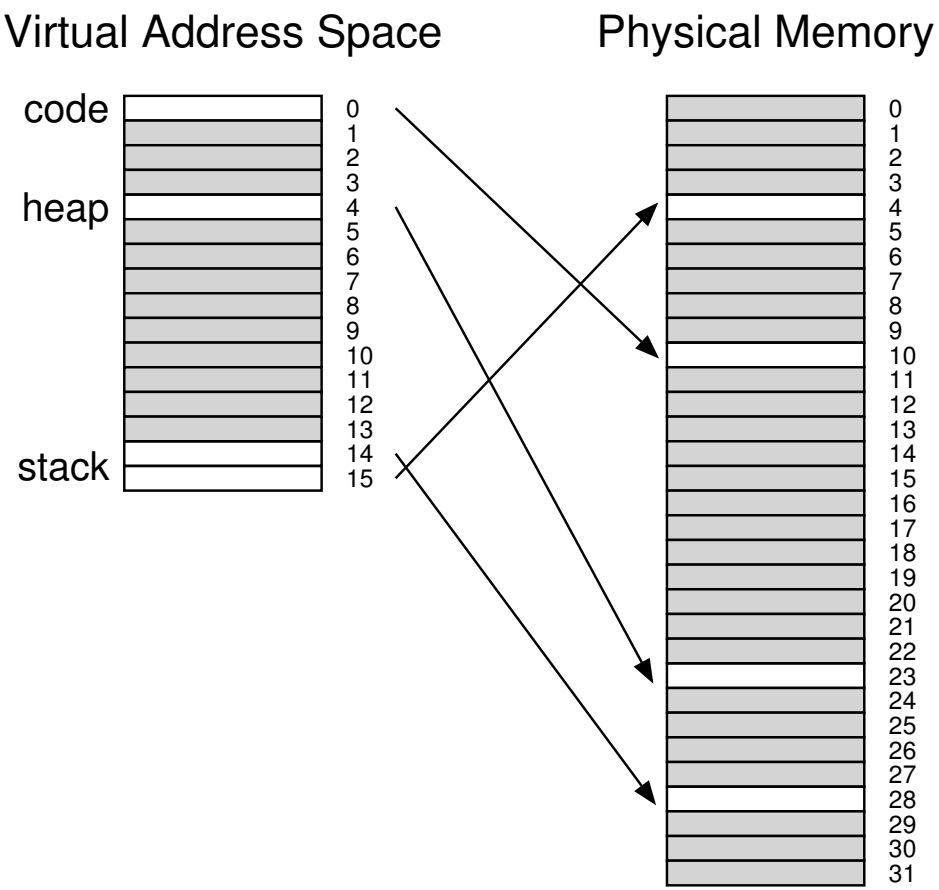
Seen a Zeedonk?

- In most cases the entire address space is not used. Instead few parts of code, heap and stacks are in use.
- Having a page table that includes PTEs for the entire address space is thus not meaningful and a waste of memory



PFN	valid	prot	present	dirty
10	1	r-x	1	0
-	0	–	-	-
-	0	–	-	-
-	0	–	-	-
23	1	rw-	1	1
-	0	–	-	-
-	0	–	-	-
-	0	–	-	-
-	0	–	-	-
-	0	–	-	-
-	0	–	-	-
-	0	–	-	-
-	0	–	-	-
-	0	–	-	-
-	0	–	-	-
28	1	rw-	1	1
4	1	rw-	1	1

A 16KB Address Space With 1KB Pages



PFN	valid	prot	present	dirty
10	1	r-x	1	0
-	0	–	-	-
-	0	–	-	-
-	0	–	-	-
23	1	rw-	1	1
-	0	–	-	-
-	0	–	-	-
-	0	–	-	-
-	0	–	-	-
-	0	–	-	-
-	0	–	-	-
-	0	–	-	-
-	0	–	-	-
-	0	–	-	-
28	1	rw-	1	1
4	1	rw-	1	1

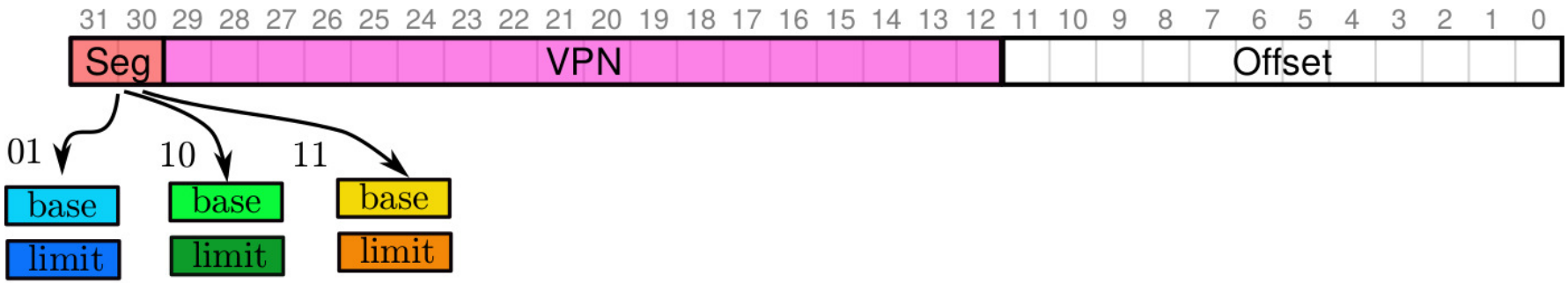
Lots of unused PTEs!
how to avoid storing these?

A 16KB Address Space With 1KB Pages

Imagine the page table of a 32-bit address space and all the potential wasted space in there!

The Hybrid approach

idea: Instead of having a single page table for the entire address space of the process, why not have one per logical segment? (e.g., code, heap, and stack)



Base and bound register pairs

Base register: physical address of the page table of that segment

Bound (limit) register: the end of the page table(i.e., how many valid pages it has)

On a TLB miss

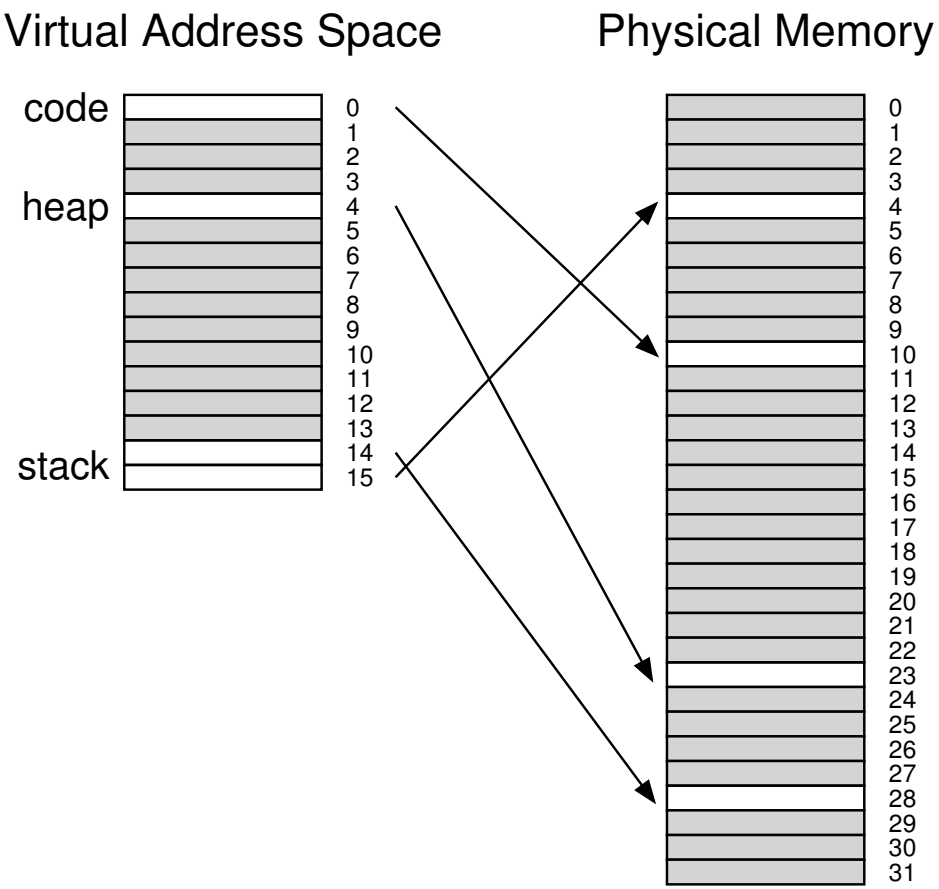
using the base register to find the address of the page table (PT) corresponding to the SN

using VPN to find the particular PTE in the PT

```
1SN             = (VirtualAddress & SEG_MASK) >> SN_SHIFT
2VPN            = (VirtualAddress & VPN_MASK) >> VPN_SHIFT
3ddressOfPTE = Base[SN] + (VPN*sizeof(PTE))
4
```

A difference with linear pagetable

A more critical difference: the presence of a bounds register per segment ⇒ **significant memory savings**



PFN	valid	prot	present	dirty
10	1	r-x	1	0

PFN	valid	prot	present	dirty
23	1	rw-	1	1

PFN	valid	prot	present	dirty
28	1	rw-	1	1
4	1	rw-	1	1

A 16KB Address Space With 1KB Pages

Challenges with the hybrid method:

- Not that much effective for the processes with large but sparsely used heap
- External fragmentation (in the part of the memory used to store PTs of processes)

Class Question

In an architecture with paged segmentation, the 32-bit virtual address is divided into fields as follows:

```
| 4 bit segment number | 12 bit page number | 16 bit offset |
```

The segment and page tables are as follows (all values in hexadecimal):

Segment Table

0	Page Table A
1	Page Table B
x	(rest invalid)

Page Table A

0	CAFE
1	DEAD
2	BEEF
3	BA11
x	(rest invalid)

Page Table B

0	F000
1	D8BF
3	BA11
x	(rest invalid)

Translate each of the following virtual addresses (answer "invalid virtual address" if the virtual address is invalid):

a. 00000000

b. 20022002

c. 10015555

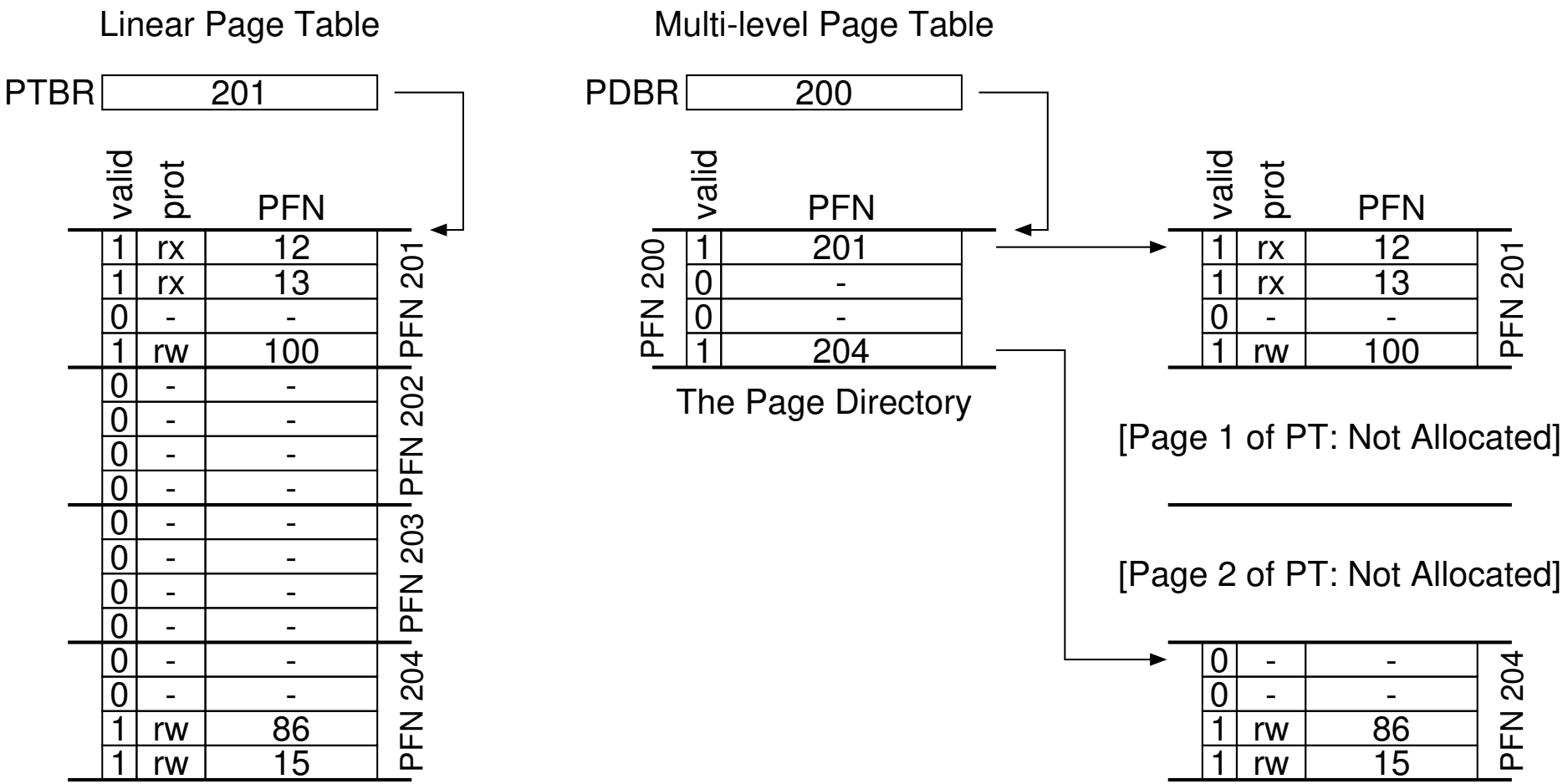
یک سیستم مدیریت حافظه از نوع سگمنتی صفحه بندی شده (paged segmentation) را در نظر بگیرید. همچنین فرض کنید در این سیستم مدیریت حافظه اندازه هر سگمنت بتواند حداکثر تا ۴ گیگابایت رشد کند و اندازه هر رکورد جدول صفحه ۴ بایت باشد. برای آنکه متوسط سربار (مجموع فضای لازم برای ذخیره جداول و قطعه شدگی داخلی ۱) این سیستم مدیریت حافظه حداقل شود اندازه صفحات می‌بایست چند بایت باشد؟

راهنمایی: برای محاسبه مقدار متوسط سربار فرض کنید اندازه هر سگمنت عددی تصادفی بین ۰ و ۴ گیگابایت است. همچنین به دلیل مدیریت خوب حافظه توسط پردازه‌ها، در سگمنت‌های از نوع heap نیز فضای خالی قابل ملاحظه‌ای ایجاد نمی‌شود.

Multi-level Page Tables

idea: First, chop up the page table into page-sized units; then, if an entire page is invalid, don't allocate that page of the page table at all.

To track whether a page of the page table is valid (and if valid, where it is in memory), use a new structure, called the page directory.



Linear (Left) And Multi-Level (Right) Page Tables

► if the PDE is valid, it means that at least one of the pages of the page table that the entry points to (via the PFN) is valid

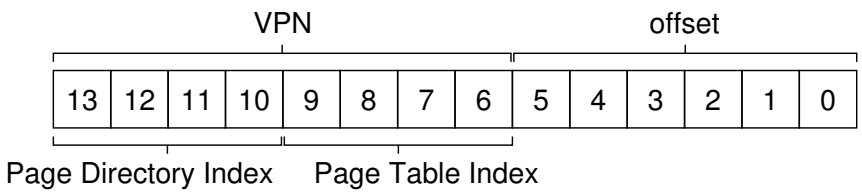
Example

Imagine a small address space of size 16KB, with 64-byte pages. Also assume the length of PTE (and PDE) is 4 bytes.

offset = 6 bits
VA = 14 bits
VPN = 8 bits
# PTEs in each page = 16
# pages of page tables
$= 2^8/2^4 = 16$

PT Index = 4 bits

PD Index = 4bits



VA to PA translation:

```
1 PDEAddr = PageDirBase + (PDIndex*sizeof(PDE))
2 PDE     =AccessMemory(PDEAddr)
3 PTEAddr = (PDE.PFN << SHIFT) + (PTIndex*sizeof(PTE))
4 PTE     =AccessMemory(PTEAddr)
```

Example (continue)

Assume the following page directory and page table for a process:

Page Directory		Page of PT (@PFN:100)			Page of PT (@PFN:101)		
PFN	valid?	PFN	valid	prot	PFN	valid	prot
100	1	10	1	r-x	—	0	—
—	0	23	1	r-x	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	80	1	rw-	—	0	—
—	0	59	1	rw-	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	55	1	rw-
101	1	—	0	—	45	1	rw-

How do you translate `11 1111 1000 0000`?

Example (continue)

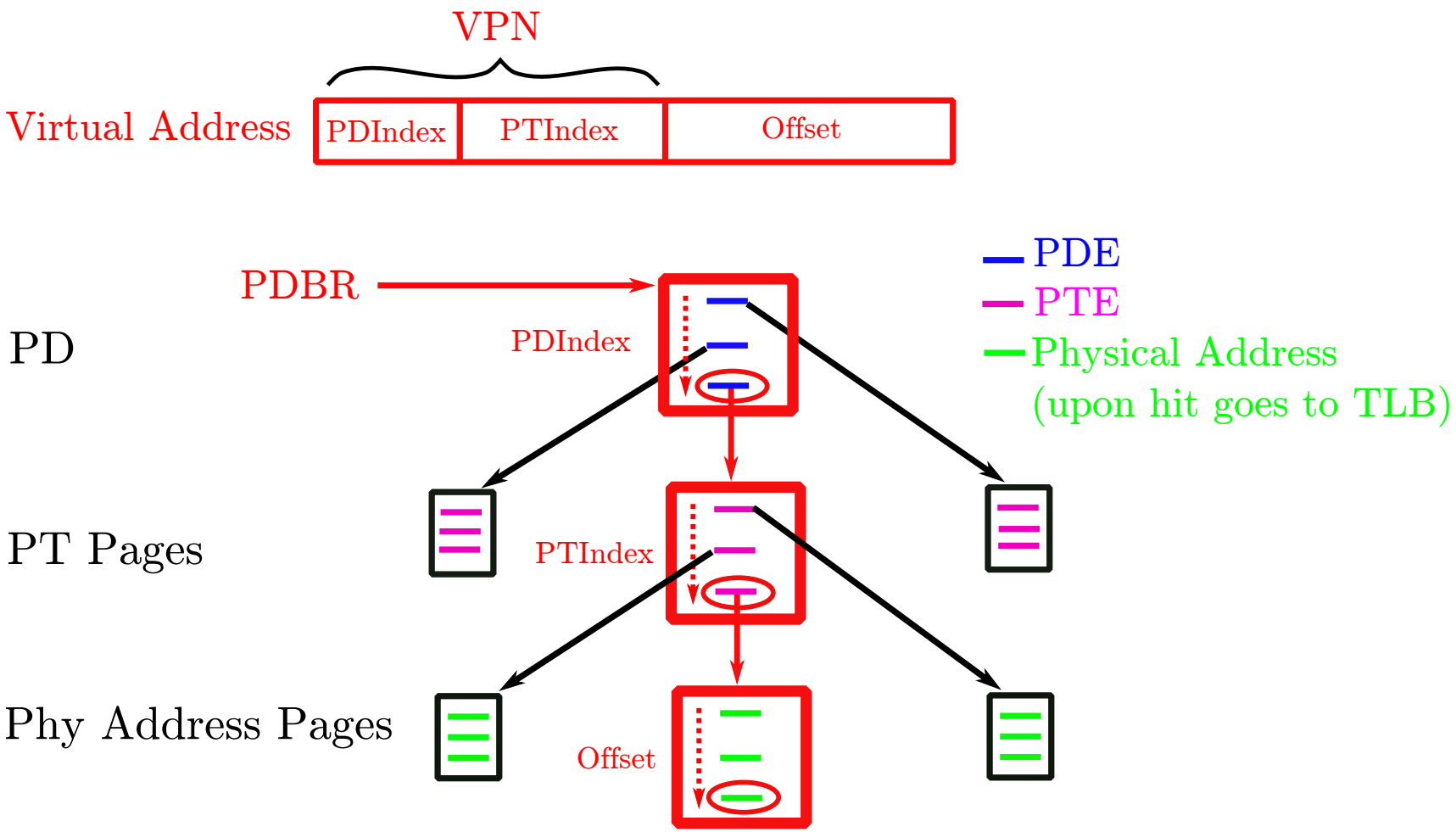
Assume the following page directory and page table for a process:

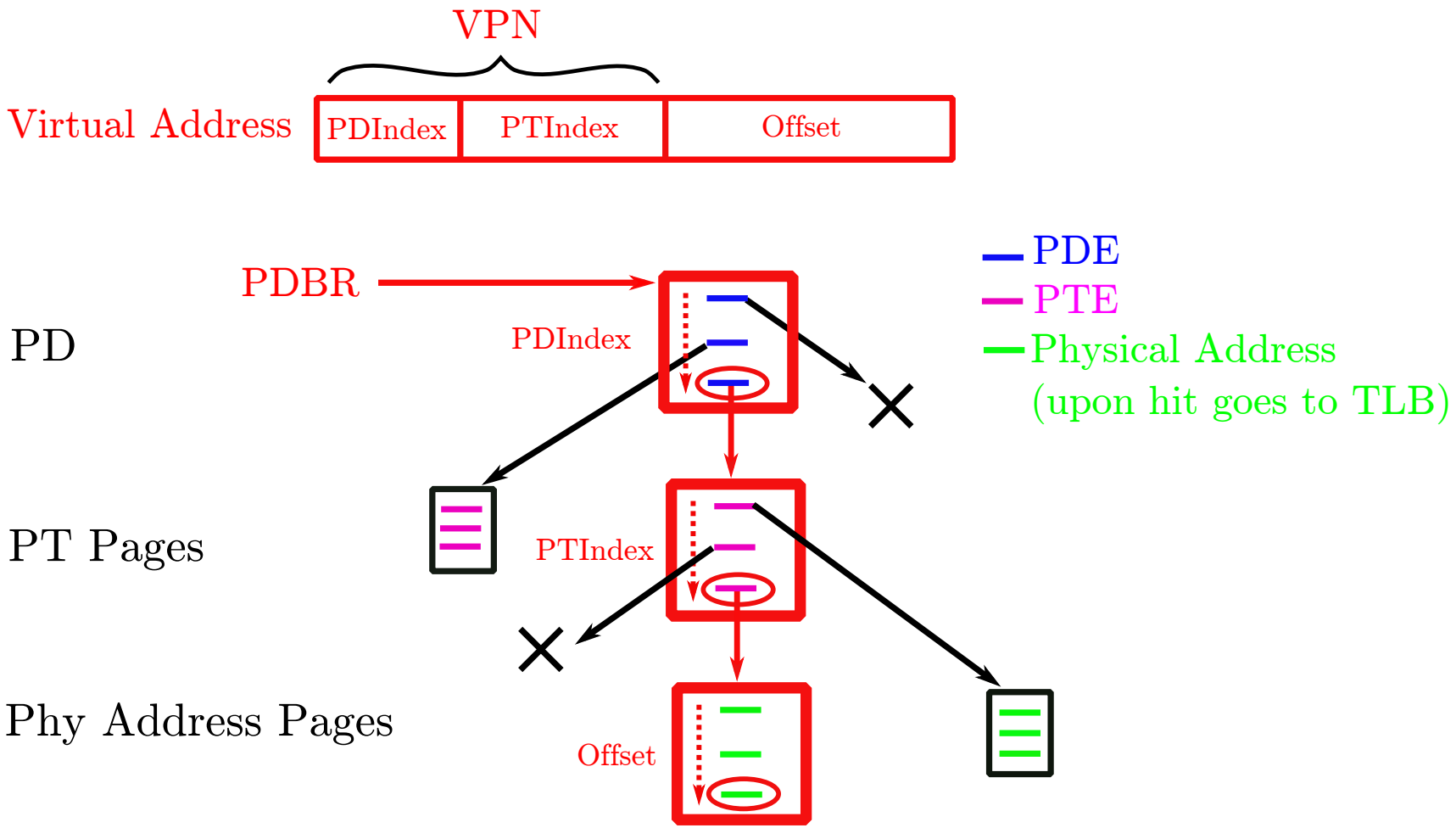
Page Directory		Page of PT (@PFN:100)			Page of PT (@PFN:101)		
PFN	valid?	PFN	valid	prot	PFN	valid	prot
100	1	10	1	r-x	—	0	—
—	0	23	1	r-x	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	80	1	rw-	—	0	—
—	0	59	1	rw-	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	55	1	rw-
101	1	—	0	—	45	1	rw-

How do you translate `11 1111 1000 0000`?

Answer: `00 1101 1100 0000`

2 level paging:





- What makes savings is that many pages in each level may not exist (the valid bit is 0)

Multi-level Page Table Control Flow

```
1 VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2 (Success, TlbEntry) = TLB_Lookup(VPN)
3 if (Success == True)    // TLB Hit
4   if (CanAccess(TlbEntry.ProtectBits) == True)
5     Offset   = VirtualAddress & OFFSET_MASK
6     PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7     Register = AccessMemory(PhysAddr)
8   else
9     RaiseException(PROTECTION_FAULT)
10 else                    // TLB Miss
11 // first, get page directory entry
12   PDIndex = (VPN & PD_MASK) >> PD_SHIFT
13   PDEAddr = PDBR + (PDIndex*sizeof(PDE))
14   PDE     =AccessMemory(PDEAddr)
15   if (PDE.Valid == False)
16     RaiseException(SEGMENTATION_FAULT)
17   else
18 // PDE is valid: now fetch PTE from page table
19     PTIndex = (VPN & PT_MASK) >> PT_SHIFT
20     PTEAddr = (PDE.PFN << SHIFT) + (PTIndex*sizeof(PTE))
21     PTE     =AccessMemory(PTEAddr)
22     if (PTE.Valid == False)
23       RaiseException(SEGMENTATION_FAULT)
24     else if (CanAccess(PTE.ProtectBits) == False)
25       RaiseException(PROTECTION_FAULT)
26     else
27       TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
28       RetryInstruction()
```

Pros

Saving memory (with respect to linear array PT)

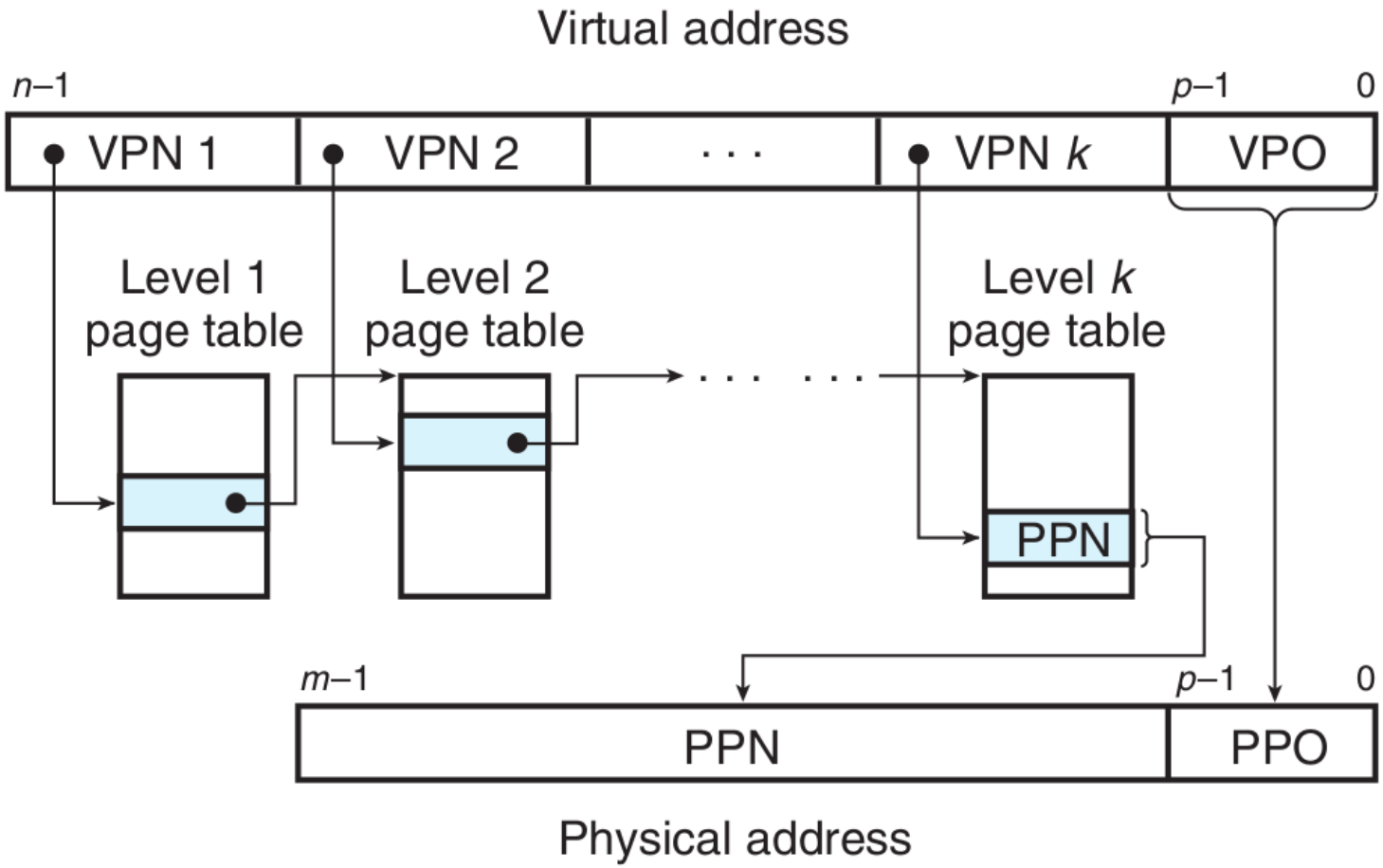
Ease of memory management (compared to the hybrid method)

Cons

Two memory access is required (in a two level strategy)

A time-space trade-off

More than 2 level paging is also possible.



فرض کنید که فضای آدرس دهی مجازی ۱۵ بیتی است و اندازه هر صفحه ۳۲ بایت است. همچنین فرض کنید که از روش paging با ۲ سطح استفاده میکنیم، یعنی یک page directory داریم که به صفحات مختلف page table اشاره می کند. هر مولفه page directory (یا همان PDE) یک بایت است. هر PDE را که در نظر بگیریم، پر ارزشترین بیت (سمت چپ ترین بیت)، به عنوان بیت اعتبار (valid bit) در نظر گرفته می شود و سایر بیتها (۷ بیت)، PFN صفحه ای از page table را مشخص می کنند. مولفه های page table (یا همان PTE ها) ساختار مشابهی دارند. یعنی هر PTE نیز یک بایت است که پر ارزشترین بیت، بیت اعتبار بوده و ۷ بیت مابقی PFN صفحه مورد نظر (صفحه ای که می خواهیم نهایتا به آن دسترسی پیدا کنیم) را مشخص می کنند.

فرض کنید که page directory در صفحه ای با شماره فیزیکی ۲۰ قرار دارد (PFN=20).

محتوای صفحات فیزیکی زیر در اختیار شما قرار داده شده است (محتوای هر بایت در فرمت هگز نشان داده شده است).

Page with PFN=5	8b 66 10 45 af e7 3c cb b1 a7 31 d2 2a e8 b2 65 f8 a0 cb 18 58 66 24 0e ae 2e c9 74 c3 1c ab a6
Page with PFN=9	6d 5d cd 8a f7 0d a5 f3 a9 62 06 2b 5d b9 2f 4e ce ba b5 a6 a0 ec bd 69 98 43 73 cf 08 1c f1 35
Page with PFN=20	f0 ac 8e ea 23 20 19 f5 6f fa fc 5a 34 a8 89 c9 bf 5f 5e 76 32 4b cb 0d ec 06 bb 34 a7 93 81 89
Page with PFN=58	2d ce 12 17 3e 10 8c 42 53 8e a1 60 86 fc 7e 26 ee b9 5e 1b ad 6b d5 8f 85 95 c6 7b f5 db f0 5a
Page with PFN=105	b9 9a 05 73 bd 4b 2d 29 5c 87 e5 7f 0f 3a 9a 43 c0 1f a3 61 44 52 51 e1 e6 05 bc 8a e0 40 3b 59
Page with PFN=127	be 91 ec 48 fb d6 78 4b 8a 42 75 cc a2 08 4f f4 ce 9e f8 b9 07 a8 5e dd 59 de a3 a1 6a ec 7c 6f

به سوالات زیر پاسخ دهید:

الف) فرض کنید فرآیندی میخواهد بایتی که در آدرس مجازی 0x3a3b قرار دارد را بخواند.

الف_۱) برای ترجمه آدرس مجازی به آدرس فیزیکی به چه صفحاتی می بایست دسترسی پیدا شود؟جواب خود را توضیح دهید.

الف_۲) آیا این آدرس مجازی به این فرآیند تخصیص داده شده است؟ اگر «خیر» چرا؟ و اگر «بله»، بایت مورد نظر که قصد خواندن آن را داریم در چه صفحه ای قرار دارد و مقدار آن چیست؟ (توضیح دهید.)

ب) قسمت قبل را برای آدرس مجازی 0x303b تکرار کنید.

Swapping

Thus far, we've assumed that

an address space is unrealistically small and fits into physical memory.

In fact, we've been assuming that every address space of every running process fits into memory.

We will now relax these big assumptions, and assume that we wish to **support many concurrently-running large address spaces.**

Remember: Big address space $\rightarrow$ Ease of life for programmers

Locality of Reference

Leverage locality of reference within processes

Spatial: reference memory addresses near previously referenced addresses

Temporal: reference memory addresses that have referenced in the past Processes spend majority of time in small portion of code

Estimate: 90% of time in 10% of code

Implication:

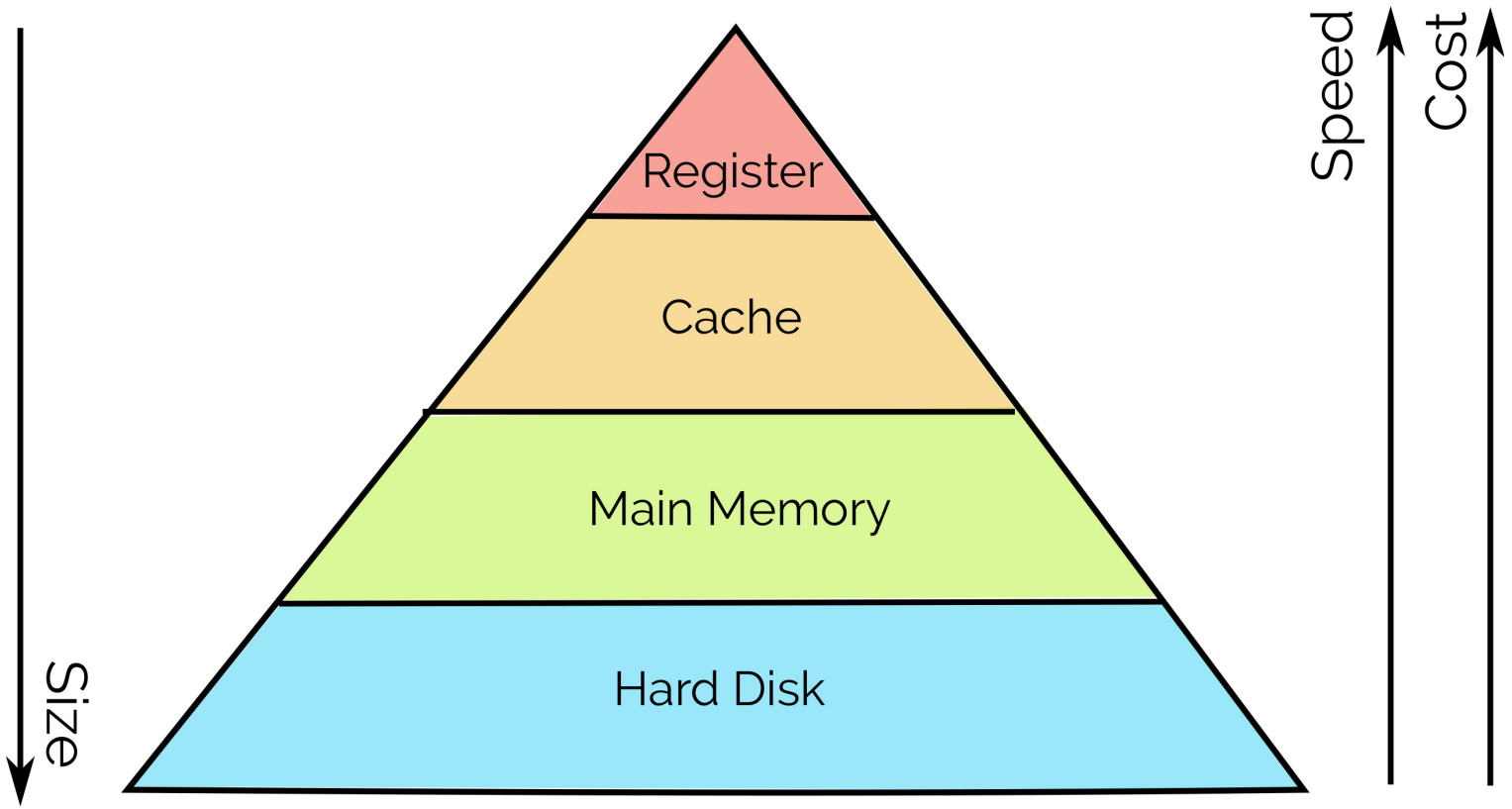
Process only uses small amount of address space at any moment

Only small amount of address space must be resident in physical memory

Memory Hierarchy

Leverage memory hierarchy of machine architecture

Each layer acts as “backing store" for layer above



idea: OS keeps unreferenced pages on **disk**

Slower, cheaper backing store than memory

Process can run when not all pages are loaded into main memory

OS and hardware cooperate to provide illusion of large disk as fast as main memory

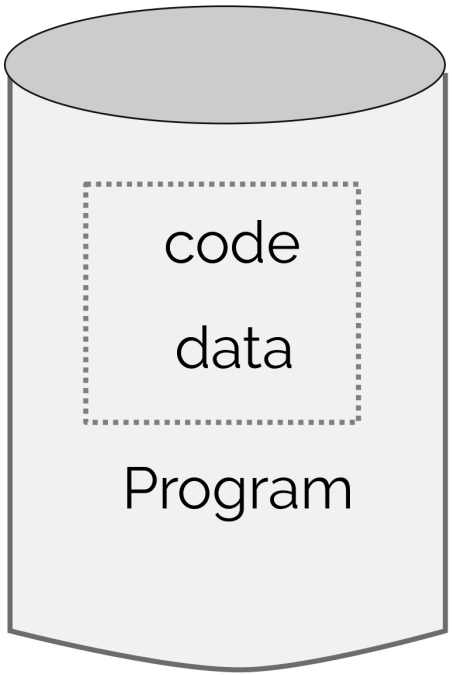
Same behavior as if all of address space in main memory

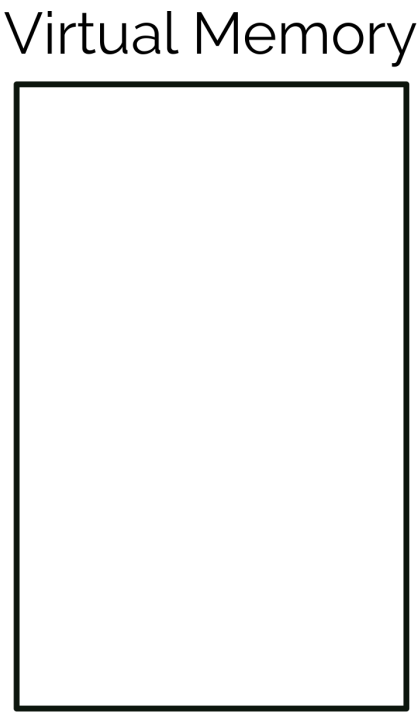
Hopefully have similar performance

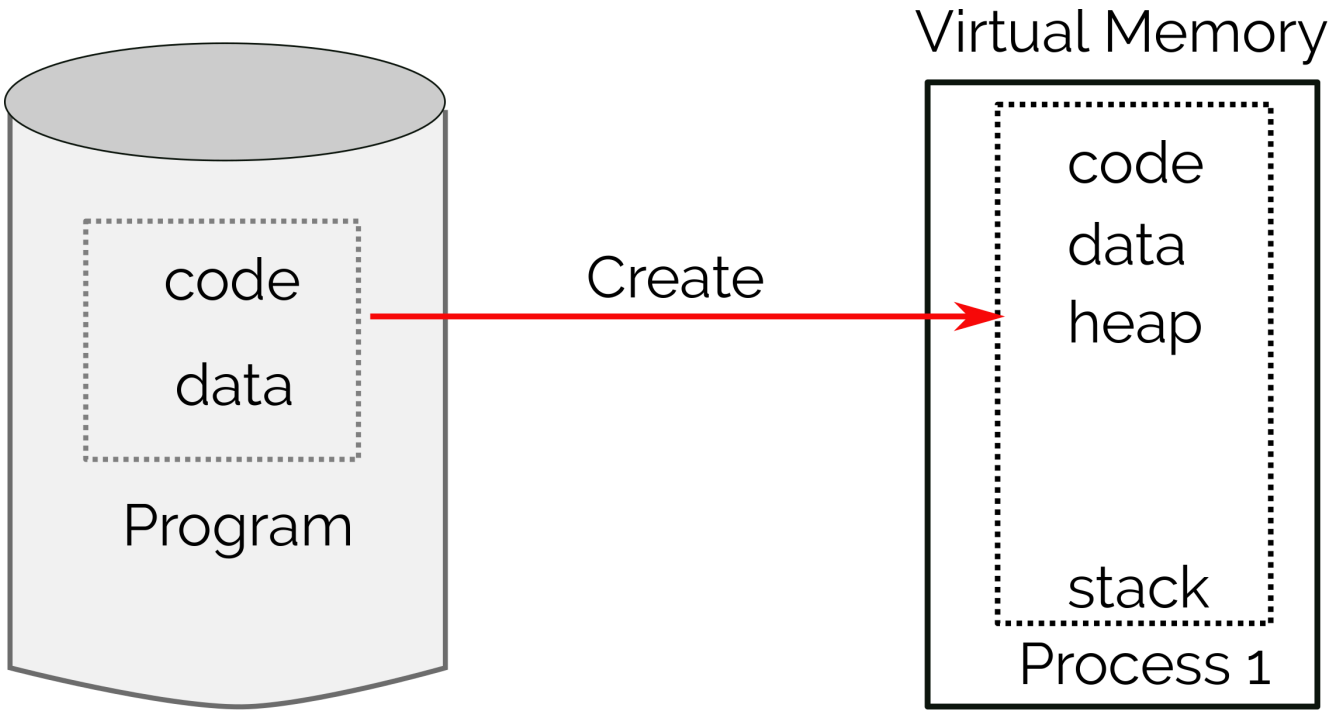
Swap space: Reserved space on the disk for *swapping* pages out of memory to it and *swapping* pages into memory from it.

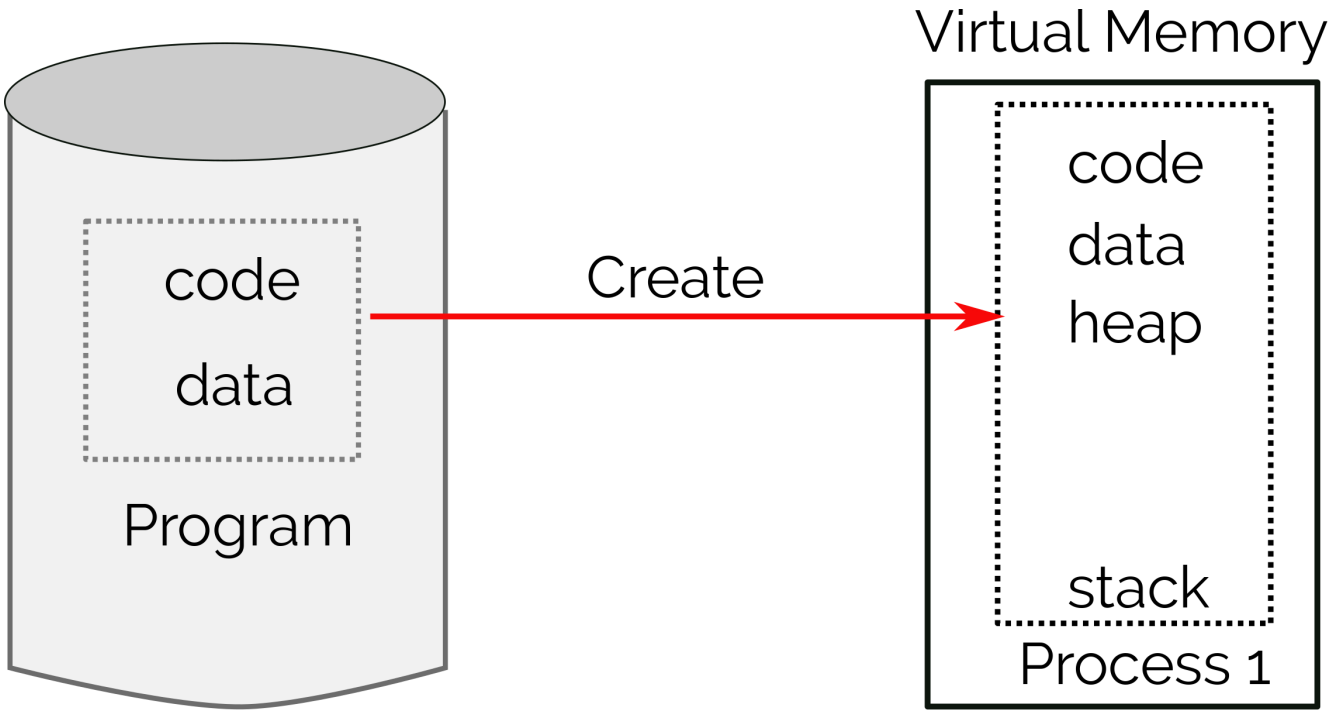
swap space is not the only on-disk location for swapping traffic.

the binary file itself!

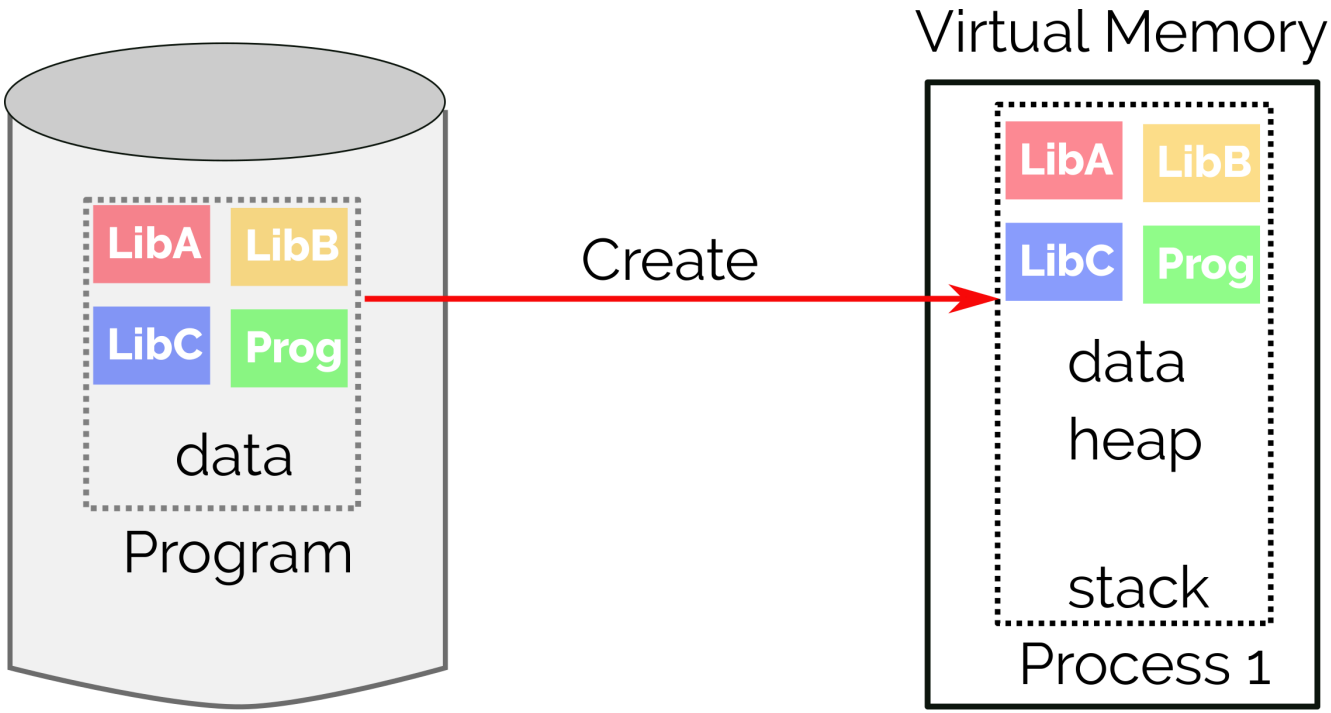

code
data
Program


Virtual Memory


Virtual Memory
code
data
heap
stack
Process 1
Create
code
data
Program

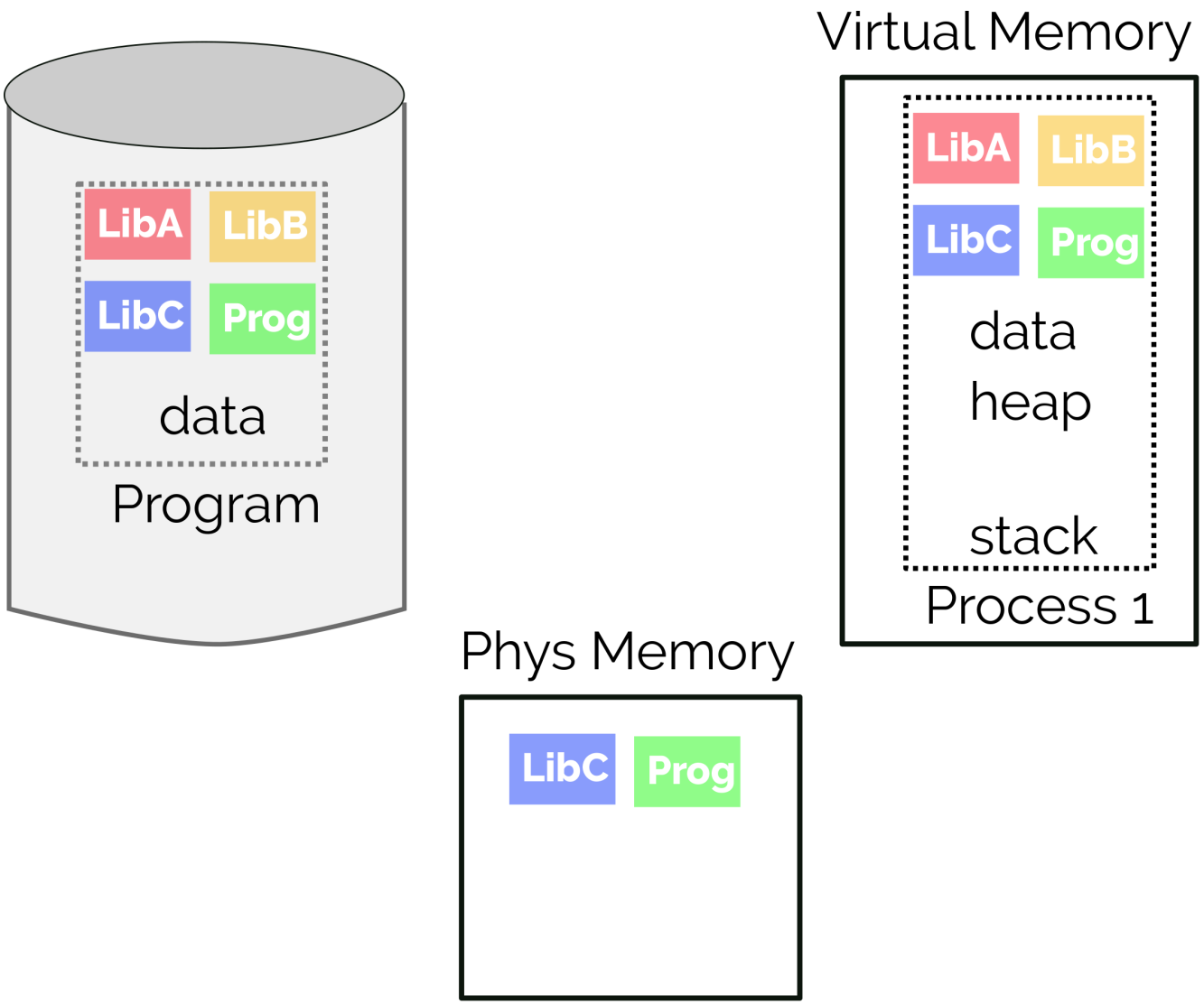


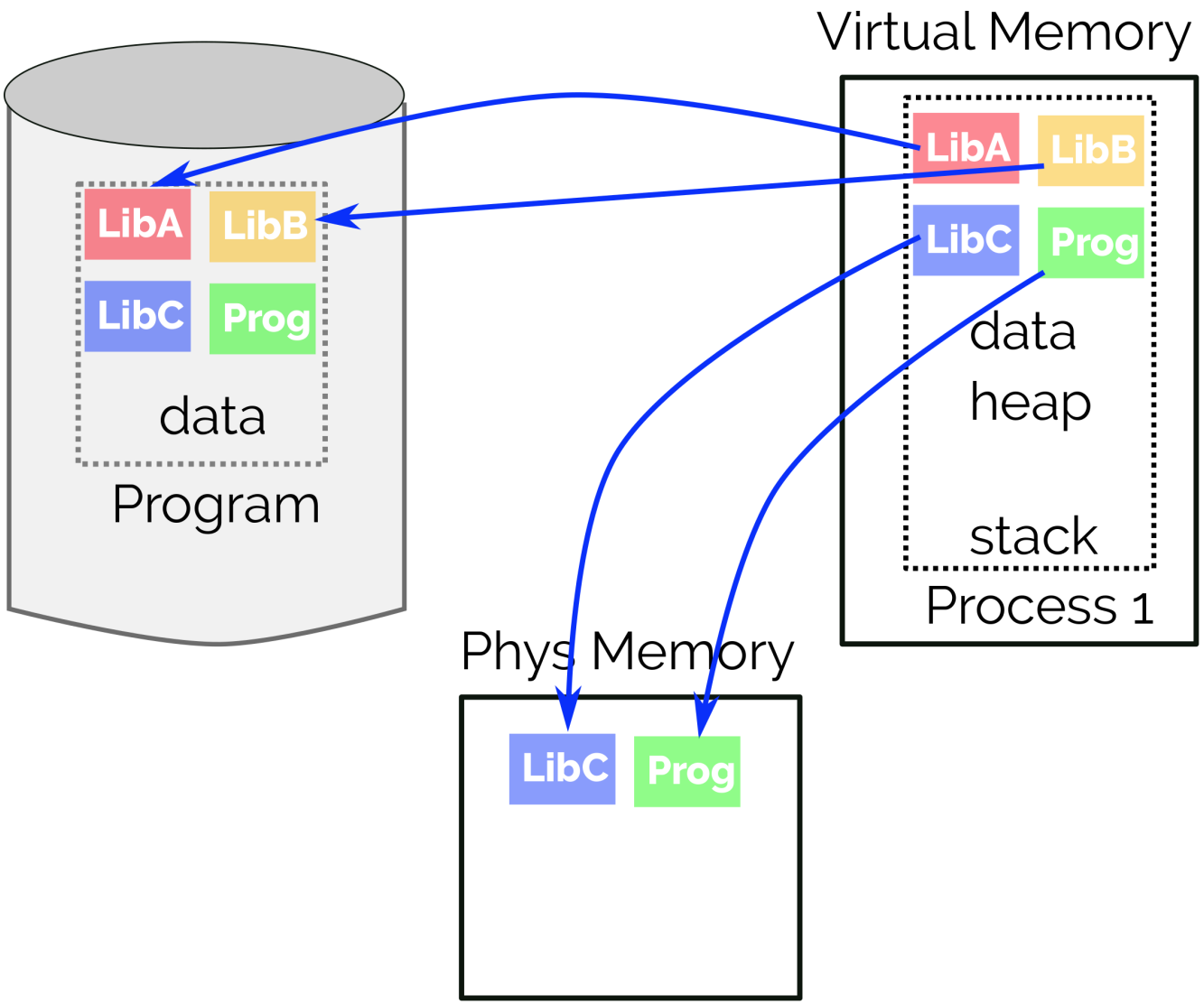
what's in the code?

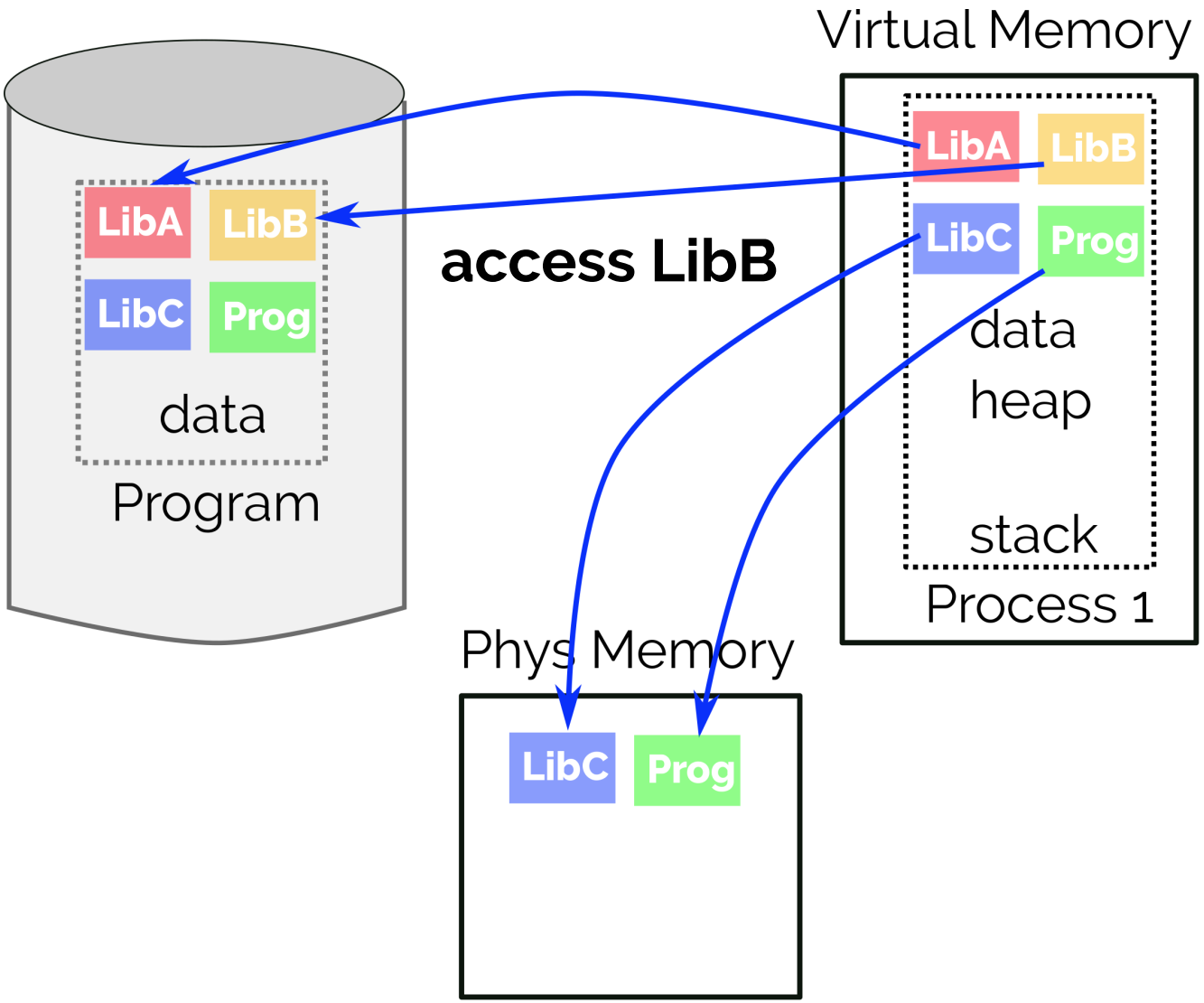


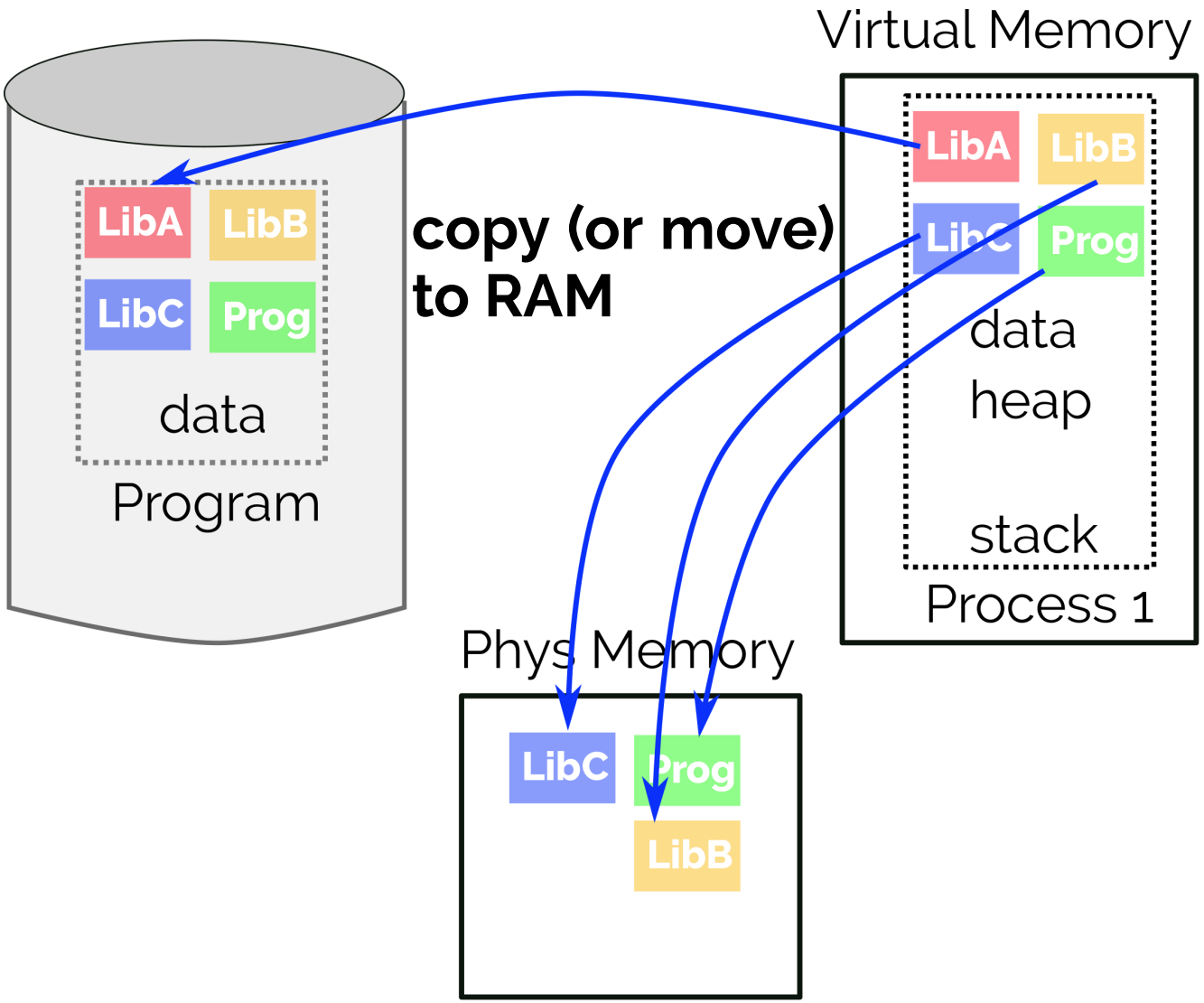
many large libraries, some of which are rarely/never used

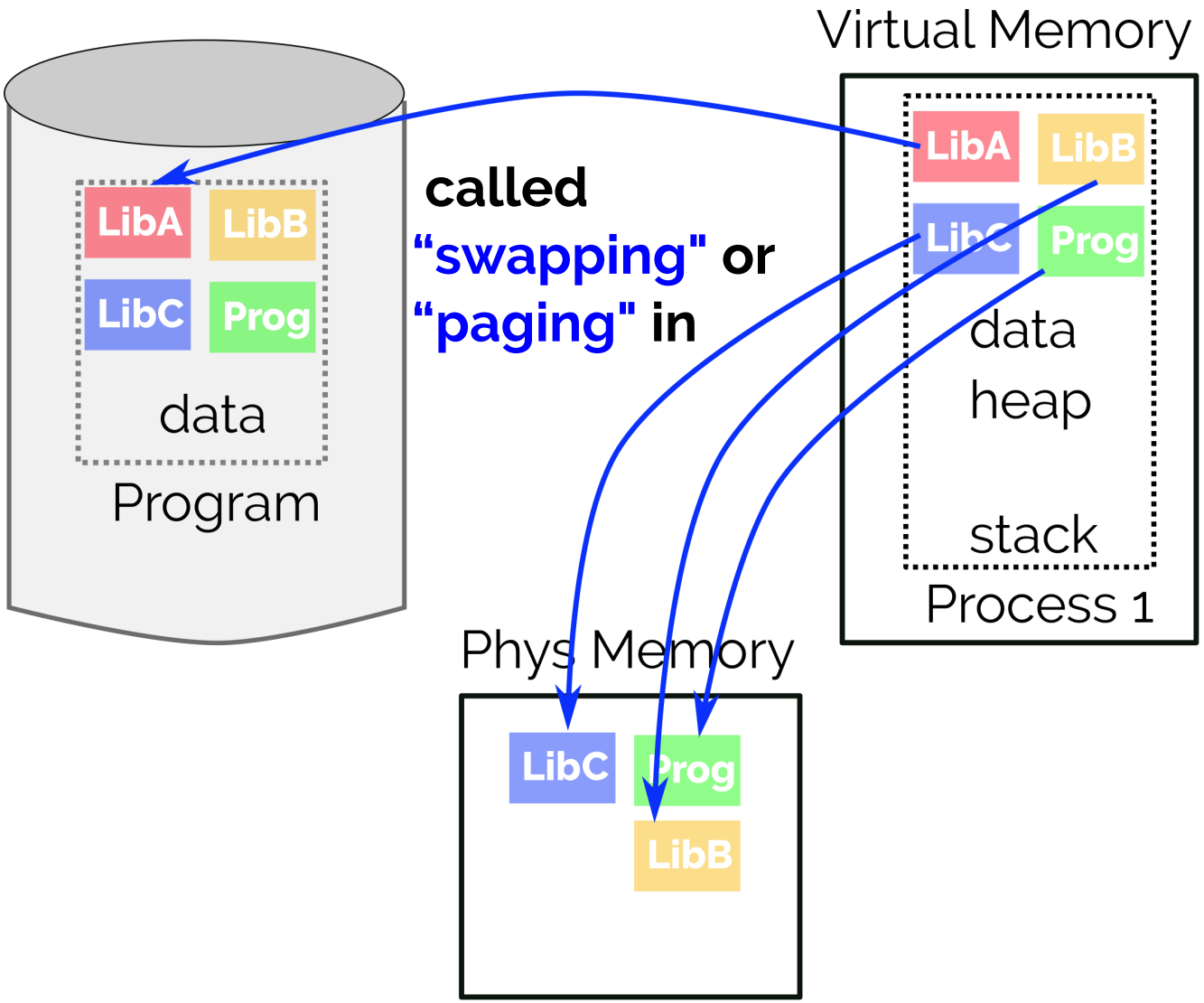
How to avoid wasting physical pages to back rarely used virtual pages?


LibA
LibB
LibC
Prog
data
Program
Virtual Memory
LibA
LibB
LibC
Prog
data
heap
stack
Process 1
Phys Memory
LibC
Prog


Virtual Memory
LibA
LibB
LibC
Prog
data
heap
stack
Process 1
LibA
LibB
LibC
Prog
data
Program
Phys Memory
LibC
Prog


Virtual Memory
LibA
LibB
LibC
Prog
data
heap
stack
Process 1
access LibB
LibA
LibB
LibC
Prog
data
Program
Phys Memory
LibC
Prog


Virtual Memory
LibA
LibB
LibC
Prog
data
heap
stack
Process 1
copy (or move)
to RAM
LibA
LibB
LibC
Prog
data
Program
Phys Memory
LibC
Prog
LibB


Virtual Memory
LibA
LibB
LibC
Prog
data
heap
stack
Process 1
called
"swapping" or
"paging" in
LibA
LibB
LibC
Prog
data
Program
Phys Memory
LibC
Prog
LibB

Mechanism

Virtual Address Space Mechanisms

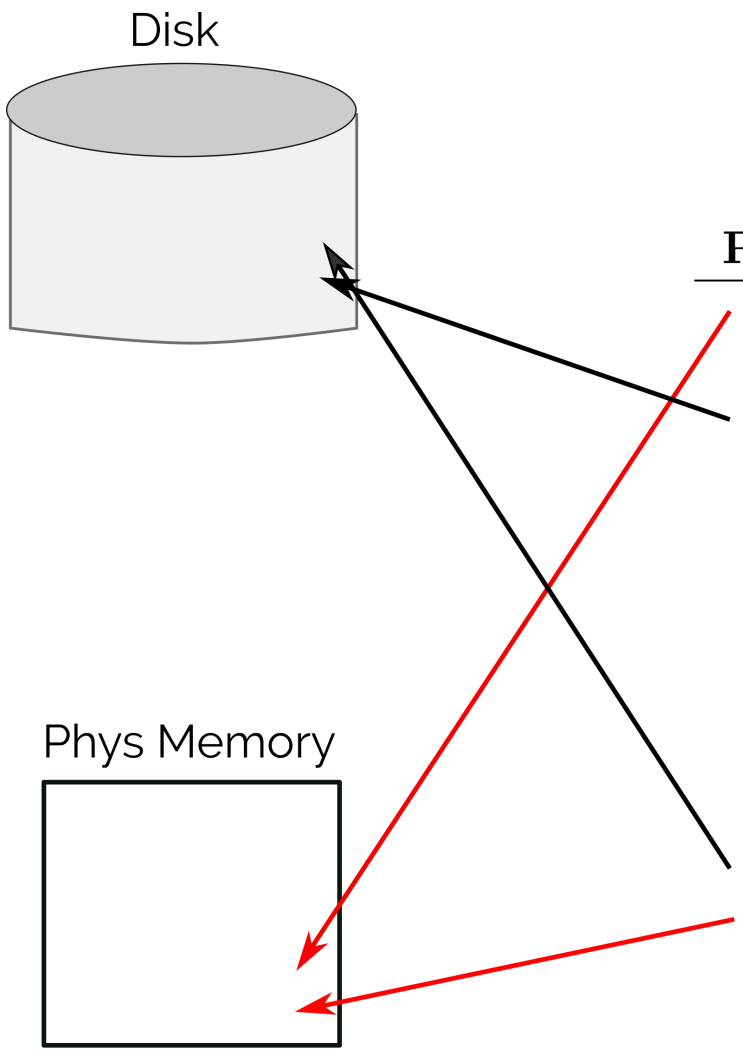
Each page in virtual address space maps to one of four locations:

- Cache: very small, very fast, very expensive
- Physical main memory: Small, fast, expensive
- Disk: Large, slow, cheap
- Nothing: Free ☺, (error)

Extend page tables with an extra bit: present

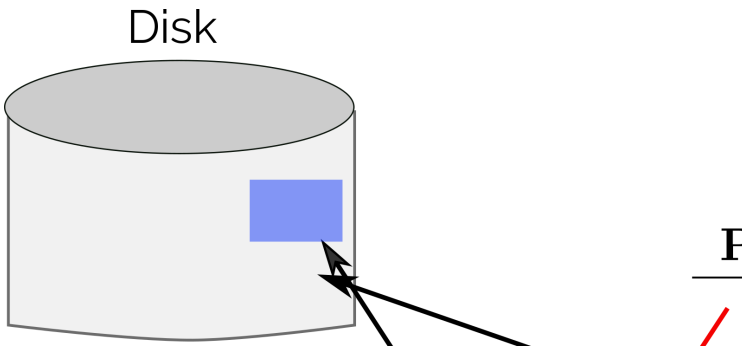
- Page in memory: present bit set in PTE
- Page on disk: present bit cleared
  - PTE points to block on disk
  - Causes trap into OS when page is referenced
  - Trap: page fault

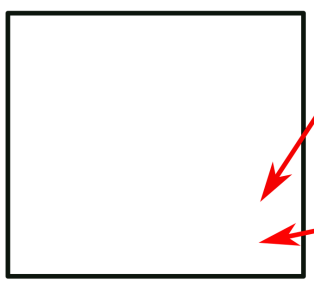
Present bit



PFN	Valid	Prot	Present
10	1	r-x	1
-	0	-	-
23	1	rw-	0
-	0	-	-
-	0	-	-
-	0	-	-
-	0	-	-
-	0	-	-
-	0	-	-
-	0	-	-
28	1	rw-	0
4	1	rw-	1

Present bit

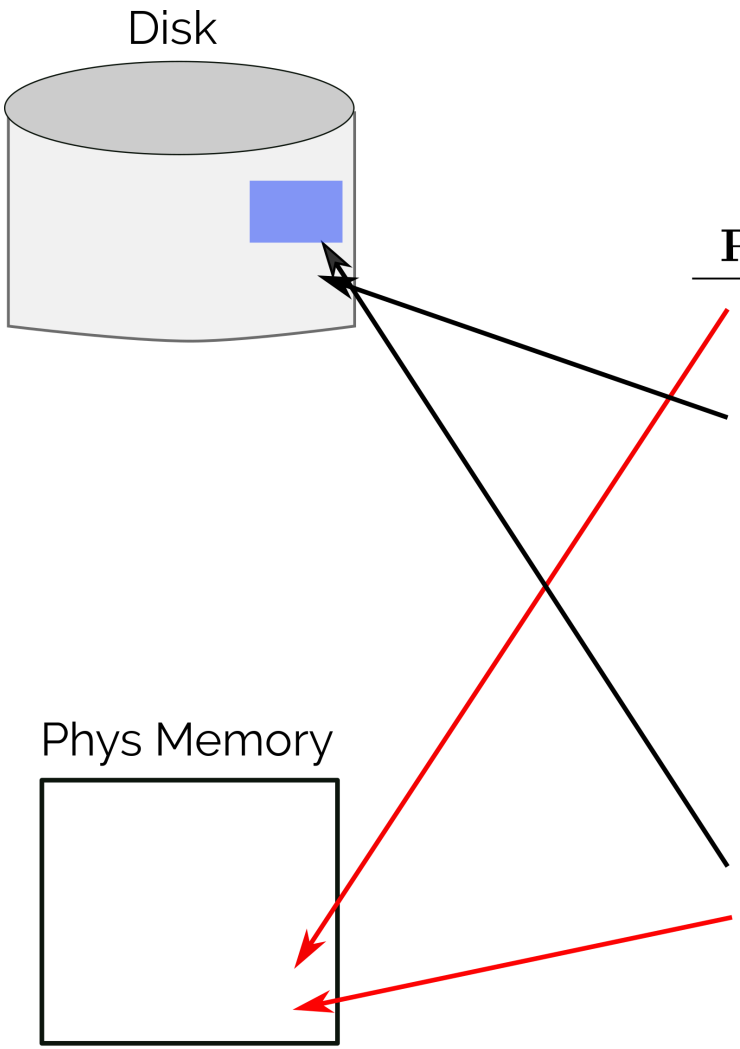




PFN	Valid	Prot	Present	
10	1	r-x	1	
-	0	-	-	
23	1	rw-	0	
-	0	-	-	
-	0	-	-	
-	0	-	-	
-	0	-	-	
-	0	-	-	
-	0	-	-	
-	0	-	-	
28	1	rw-	0	← access
4	1	rw-	1	

Present bit

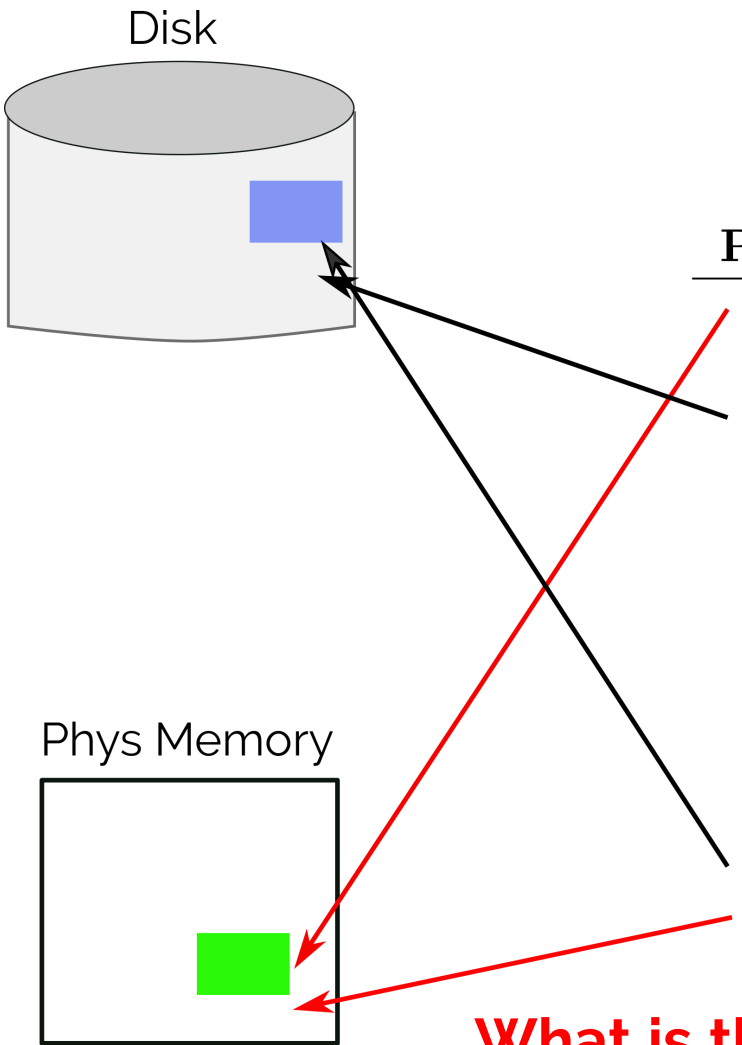
Assume no free page frame in main memory



PFN	Valid	Prot	Present	
10	1	r-x	1	
-	0	-	-	
23	1	rw-	0	
-	0	-	-	
-	0	-	-	
-	0	-	-	
-	0	-	-	
-	0	-	-	
-	0	-	-	
-	0	-	-	
28	1	rw-	0	← access
4	1	rw-	1	

Present bit

Assume no free page frame in main memory

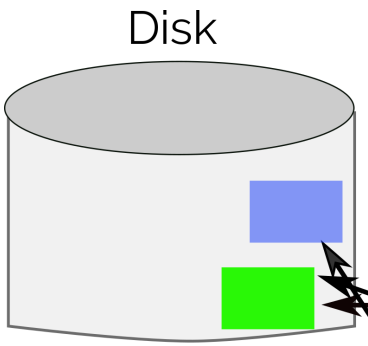


PFN	Valid	Prot	Present	
10	1	r-x	1	← evict
-	0	-	-	
23	1	rw-	0	
-	0	-	-	
-	0	-	-	
-	0	-	-	
-	0	-	-	
-	0	-	-	
-	0	-	-	
-	0	-	-	
28	1	rw-	0	← access
4	1	rw-	1	

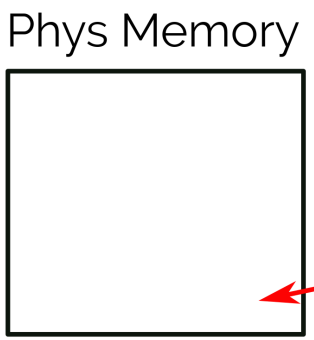
What is the replacement policy?
(to be discussed)

Present bit

Assume no free page frame in main memory

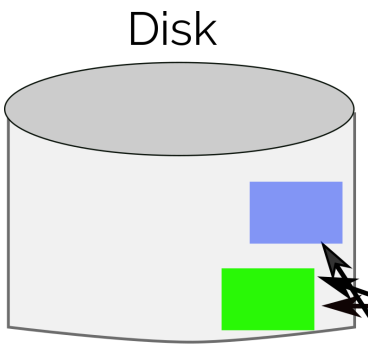


PFN	Valid	Prot	Present	
5	1	r-x	0	← evict
-	0	-	-	
23	1	rw-	0	
-	0	-	-	
-	0	-	-	
-	0	-	-	
-	0	-	-	
-	0	-	-	
-	0	-	-	
-	0	-	-	
28	1	rw-	0	← access
4	1	rw-	1	



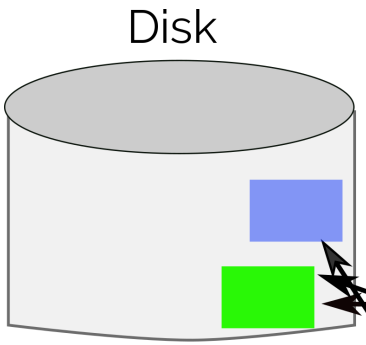
Present bit

called **swapping** or **paging** out



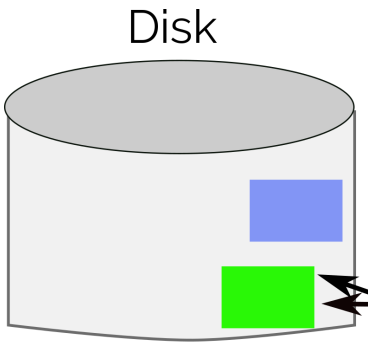
PFN	Valid	Prot	Present	
5	1	r-x	0	← evict
-	0	-	-	
23	1	rw-	0	
-	0	-	-	
-	0	-	-	
-	0	-	-	
-	0	-	-	
-	0	-	-	
-	0	-	-	
-	0	-	-	
28	1	rw-	0	← access
4	1	rw-	1	

Present bit



PFN	Valid	Prot	Pres	t
5	1	r-x	0	
-	0	-	-	
23	1	rw-	0	
-	0	-	-	
-	0	-	-	
-	0	-	-	
-	0	-	-	
-	0	-	-	
-	0	-	-	
-	0	-	-	
28	1	rw-	0	← access
4	1	rw-	1	

Present bit



PFN	Valid	Prot	Pres	t
5	1	r-x	0	
-	0	-	-	
23	1	rw-	0	
-	0	-	-	
-	0	-	-	
-	0	-	-	
-	0	-	-	
-	0	-	-	
-	0	-	-	
-	0	-	-	
10	1	rw-	1	← access
4	1	rw-	1	

Page-Fault Control Flow Algorithm (Hardware)

```
1 VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2 (Success, TlbEntry) = TLB_Lookup(VPN)
3 if (Success == True)    // TLB Hit
4   if (CanAccess(TlbEntry.ProtectBits) == True)
5     Offset   = VirtualAddress & OFFSET_MASK
6     PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7     Register = AccessMemory(PhysAddr)
8   else
9     RaiseException(PROTECTION_FAULT)
10 else                    // TLB Miss
11   PTEAddr = PTBR + (VPN*sizeof(PTE))
12   PTE =AccessMemory(PTEAddr)
13   if (PTE.Valid == False)
14     RaiseException(SEGMENTATION_FAULT)
15   else
16     if (CanAccess(PTE.ProtectBits) == False)
17       RaiseException(PROTECTION_FAULT)
18     else if (PTE.Present == True)
19       // assuming hardware-managed TLB
20       TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
21       RetryInstruction()
22     else if (PTE.Present == False)
23       RaiseException(PAGE_FAULT)
```

Page-Fault Control Flow Algorithm (Software)

```
1  PFN = FindFreePhysicalPage()
2  if (PFN == -1)                  // no free page found
3    PFN = EvictPage()           // run replacement algorithm
4  DiskRead(PTE.DiskAddr, PFN)// sleep (waiting for I/O)
5  PTE.present = True              // update page table with
     present
6  PTE.PFN     = PFN               // bit and translation (PFN)
7  RetryInstruction()              // retry instruction
8
```

Note

Why we do not have HW manged page faults? (like TLBs)
I) page faults to disk are slow → overheads of running software are minimal.
II) to be able to handle a page fault, the hardware would have to understand swap space, how to issue I/Os to the disk, and a lot of other details which it currently doesn't know much about.

Page-Fault Control Flow Algorithm (Software)

```
1   PFN = FindFreePhysicalPage()
2   if (PFN == -1)                    // no free page found
3   PFN = EvictPage()              // run replacement algorithm
4   DiskRead(PTE.DiskAddr, PFN)// sleep (waiting for I/O)
5   PTE.present = True                // update page table with
      present
6   PTE.PFN     = PFN                 // bit and translation (PFN)
7   RetryInstruction()                // retry instruction
8
```

what to evict?
what to read?
(Policy)

Upon access, we must load the desired page.

Do we prefetch other adjacent pages?
(disks have high *fixed* costs)

Prefetching more means we will have to evict more.

What to evict? [today's focus]

Policy

Replacement Policy

Which page in main memory should selected as victim?

Write out victim page to disk if modified (dirty bit set)

If victim page is not modified (clean), just discard

Performance Metric: maximizing hit rate

Optimal

FIFO

Random

LRU

Approximating LRU (Clock)

Optimal strategy

MIN by Belady, 1966

evict pages to be accessed furthest in future

simple (but, unfortunately, difficult to implement!)

Example: optimal algorithm

Access	Hit	State (after)
1		
2		
3		
4		
1		
2		
3		
4		
3		
2		
1		

assume
cache size 3

Example: optimal algorithm

Access	Hit	State (after)
1	no	1
2	no	1,2
3	no	1,2,3
4	no	???
1		
2		
3		
4		
3		
2		
1		

assume
cache size 3

Example: optimal algorithm

Access	Hit	State (after)
1	no	1
2	no	1,2
3	no	1,2,3
4	no	1,2,4
1		
2		
3		
4		
3		
2		
1		

assume
cache size 3

Example: optimal algorithm

Access	Hit	State (after)
1	no	1
2	no	1,2
3	no	1,2,3
4	no	1,2,4
1	yes	1,2,4
2	yes	1,2,4
3	no	2,3,4
4	yes	2,3,4
3	yes	2,3,4
2	yes	2,3,4
1	no	...

assume
cache size 3

Example: optimal algorithm

hit rate? ($\frac{Hit}{Hit+Miss}$)

Access	Hit	State (after)
1	no	1
2	no	1,2
3	no	1,2,3
4	no	1,2,4
1	yes	1,2,4
2	yes	1,2,4
3	no	2,3,4
4	yes	2,3,4
3	yes	2,3,4
2	yes	2,3,4
1	no	...

assume
cache size 3

Example: optimal algorithm

no fair
compulsory miss

Access	Hit	State (after)
1	no	1
2	no	1,2
3	no	1,2,3
4	no	1,2,4
1	yes	1,2,4
2	yes	1,2,4
3	no	2,3,4
4	yes	2,3,4
3	yes	2,3,4
2	yes	2,3,4
1	no	...

assume
cache size 3

Example: optimal algorithm

no fair compulsory miss

hit rate modulo compulsory? (ignore the first miss to a given page)

Access	Hit	State (after)
1	no	1
2	no	1,2
3	no	1,2,3
4	no	1,2,4
1	yes	1,2,4
2	yes	1,2,4
3	no	2,3,4
4	yes	2,3,4
3	yes	2,3,4
2	yes	2,3,4
1	no	...

assume cache size 3

Note

Unfortunately, as we saw before in the development of scheduling policies, the future is not generally known; you can't build the optimal policy for a general-purpose operating system. The optimal policy will thus serve only as a comparison point, to know how close we are to "perfect".

FIFO

Items are evicted in the order they are inserted

quite simple to implement

Example: FIFO

Access	Hit	State (after)
1		
2		
3		
4		
1		
2		
3		
4		
3		
2		
1		

assume
cache size 3

Example: FIFO

Access	Hit	State (after)
1	no	1
2	no	1,2
3	no	1,2,3
4	no	???
1		
2		
3		
4		
3		
2		
1		

assume
cache size 3

Example: FIFO

Access	Hit	State (after)
1	no	1
2	no	1,2
3	no	1,2,3
4	no	2,3,4
1		
2		
3		
4		
3		
2		
1		

assume
cache size 3

Example: FIFO

Access	Hit	State (after)
1	no	1
2	no	1,2
3	no	1,2,3
4	no	2,3,4
1	no	3,4,1
2	no	4,1,2
3	no	1,2,3
4	no	2,3,4
3	yes	2,3,4
2	yes	2,3,4
1	no	...

assume
cache size 3

Random

picks a random page to replace under memory pressure.

quite simple to implement

LRU, LFU

LRU: evict least-recently used

LFU: evict least-frequently used

Consider history

Principle of locality

Example: LRU

Access	Hit	State (after)
1		
2		
3		
4		
1		
2		
3		
4		
3		
2		
1		

assume
cache size 3

Example: LRU

Access	Hit	State (after)
1	no	1
2	no	1,2
3	no	1,2,3
4	no	???
1		
2		
3		
4		
3		
2		
1		

assume
cache size 3

Example: LRU

Access	Hit	State (after)
1	no	1
2	no	1,2
3	no	1,2,3
4	no	2,3,4
1		
2		
3		
4		
3		
2		
1		

assume
cache size 3

Example: LRU

Access	Hit	State (after)
1	no	1
2	no	1,2
3	no	1,2,3
4	no	2,3,4
1	no	3,4,1
2	no	4,1,2
3	no	1,2,3
4	no	2,3,4
3	yes	2,3,4
2	yes	2,3,4
1	no	...

assume
cache size 3

Example: LRU

Access	Hit	State (after)
1	no	1
2	no	1,2
3	no	1,2,3
4	no	2,3,4
1	no	3,4,1
2	no	4,1,2
3	no	1,2,3
4	no	2,3,4
3	yes	2,3,4
2	yes	2,3,4
1	no	...

assume
cache size 3

Same as FIFO?

Example: LRU & FIFO

LRU

Access	Hit	State (after)
1	no	1
2	no	1,2
3	no	1,2,3
1	yes	1,2,3
2	yes	1,2,3
4	no	1,2,4
1	yes	1,2,4
4	yes	1,2,4
2	yes	1,2,4
3	no	3,2,4
2	yes	3,2,4

FIFO

Access	Hit	State (after)
1	no	1
2	no	1,2
3	no	1,2,3
1	yes	1,2,3
2	yes	1,2,3
4	no	2,3,4
1	no	3,4,1
4	yes	3,4,1
2	no	4,1,2
3	no	1,2,3
2	yes	1,2,3

Example: LRU & FIFO

LRU

Access	Hit	State (after)
1	no	1
2	no	1,2
3	no	1,2,3
1	yes	1,2,3
2	yes	1,2,3
4	no	1,2,4
1	yes	1,2,4
4	yes	1,2,4
2	yes	1,2,4
3	no	3,2,4
2	yes	3,2,4

FIFO

Access	Hit	State (after)
1	no	1
2	no	1,2
3	no	1,2,3
1	yes	1,2,3
2	yes	1,2,3
4	no	2,3,4
1	no	3,4,1
4	yes	3,4,1
2	no	4,1,2
3	no	1,2,3
2	yes	1,2,3

Workload (locality) matters

Workload Examples

100 unique pages

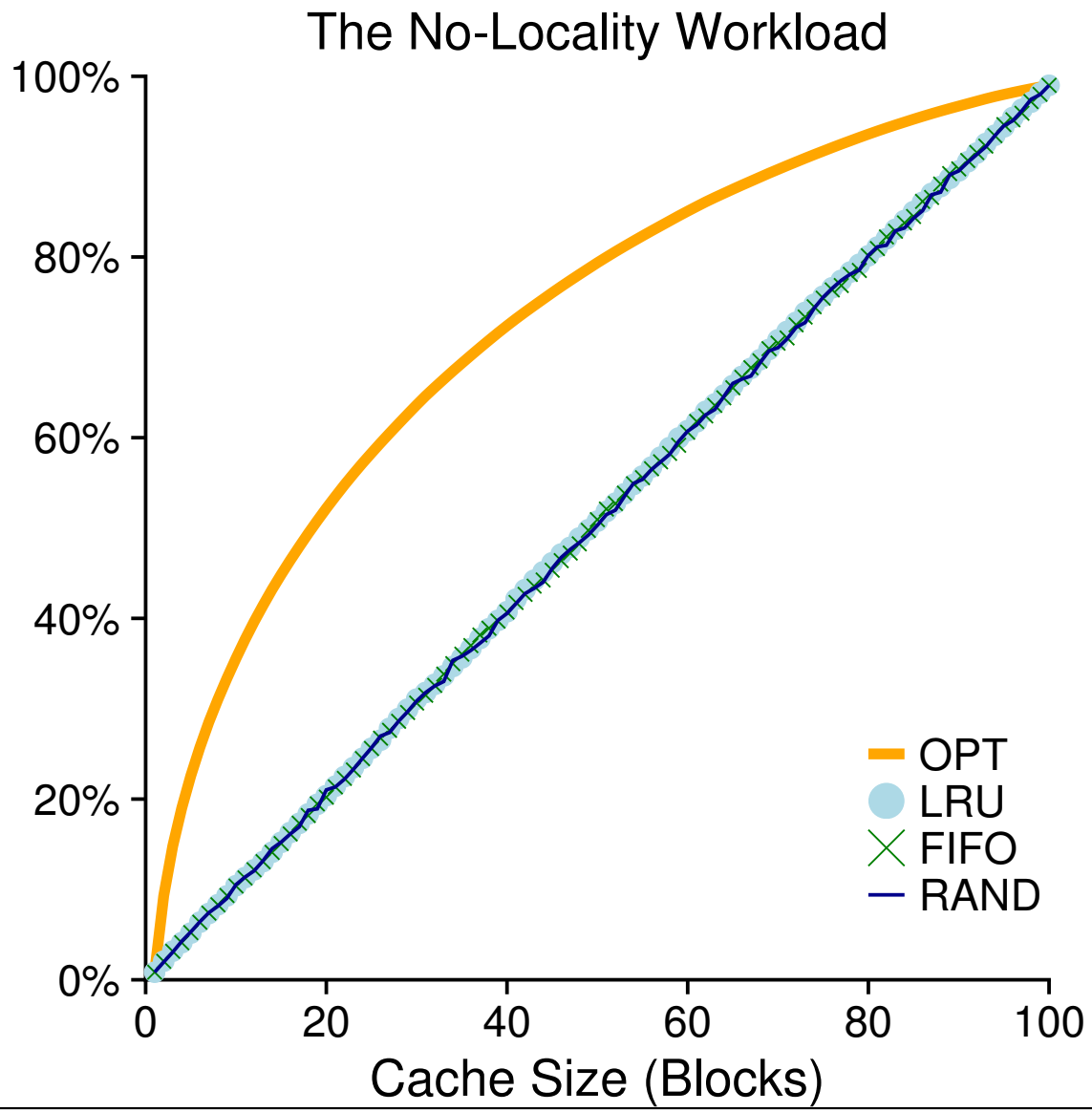
10,000 pages are accessed

cache size is changed from 1 page to 100 page

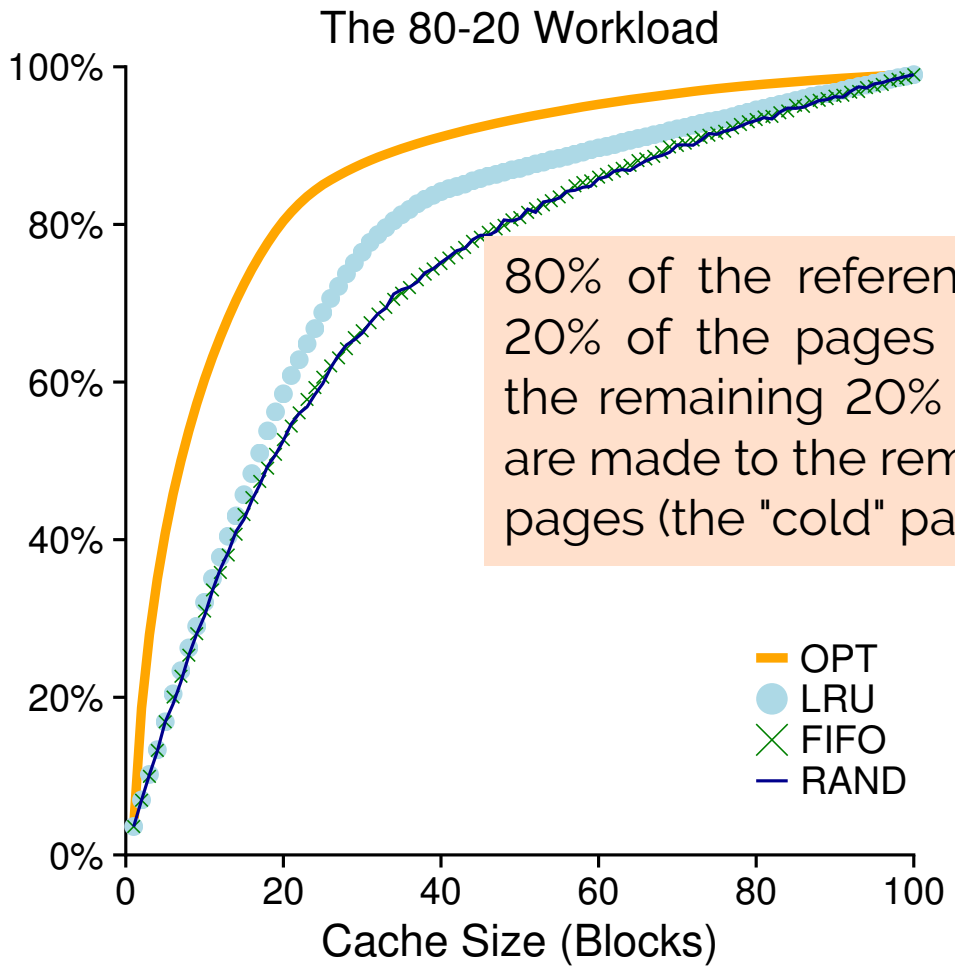
Scenarios:

No locality
80-20 Locality
Looping sequential

Workload Examples

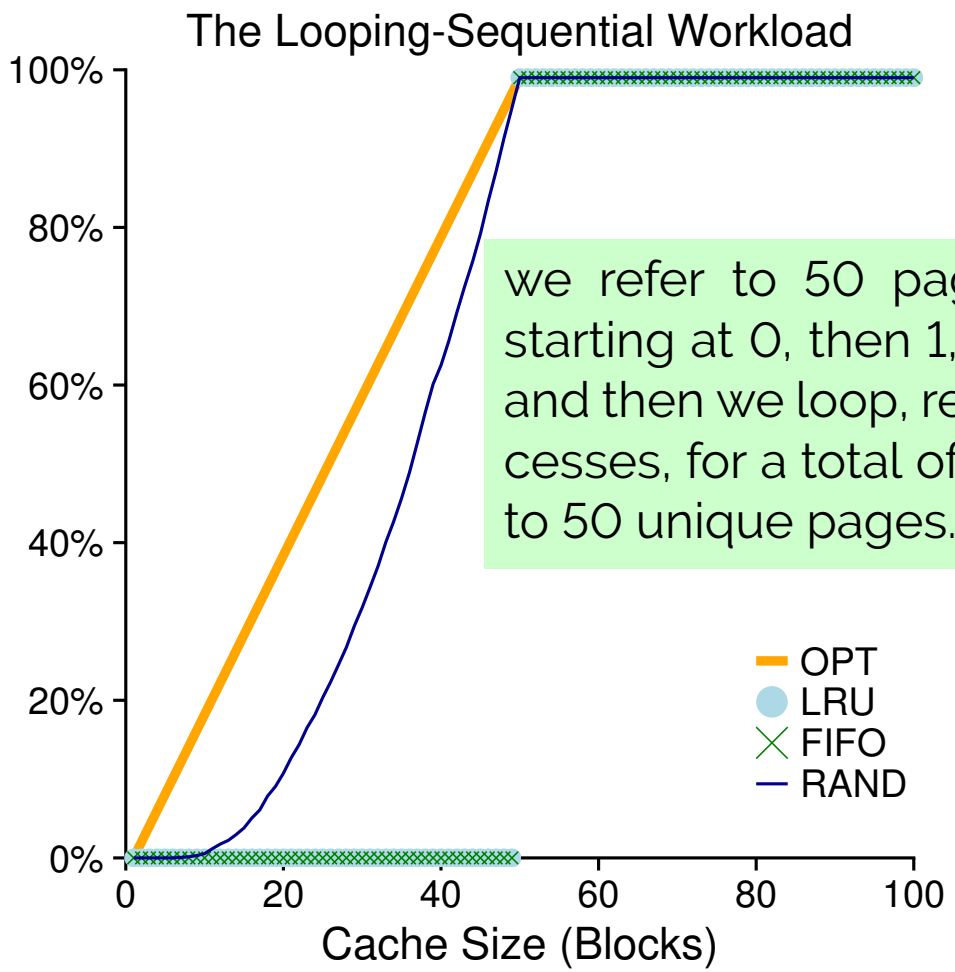


Workload Examples



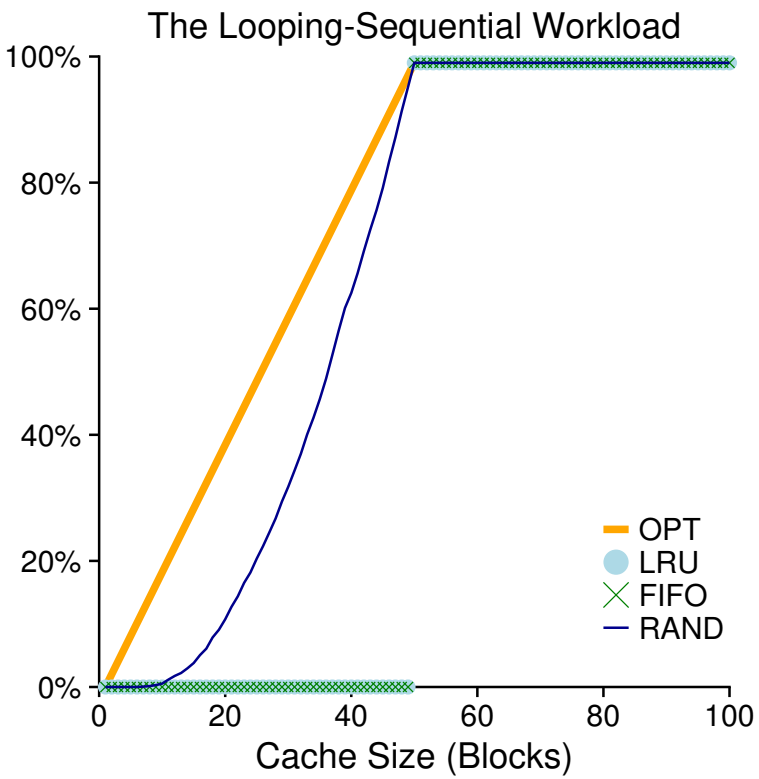
80% of the references are made to 20% of the pages (the "hot" pages); the remaining 20% of the references are made to the remaining 80% of the pages (the "cold" pages).

Workload Examples



we refer to 50 pages in sequence, starting at 0, then 1, ..., up to page 49, and then we loop, repeating those accesses, for a total of 10,000 accesses to 50 unique pages.

Workload Examples



Random has some nice properties;

one such property is not having weird corner-case behaviors.

LRU Hardware Support

What is needed?

Timestamps

An array of timestamps (one for each Phy. frame), updated by HW on each mem access

OS could simply scan all the time fields in the system to find the least-recently-used page.

(Why can't OS alone track this?)

LRU Hardware Support

What is needed?

Timestamps

A time-stamp for each page, updated by HW on each memory access

(Why can't OS alone track this?)

OS could simply scan all the time stamps to find the least-recently-used page. ⊠ Slow

Approximating LRU

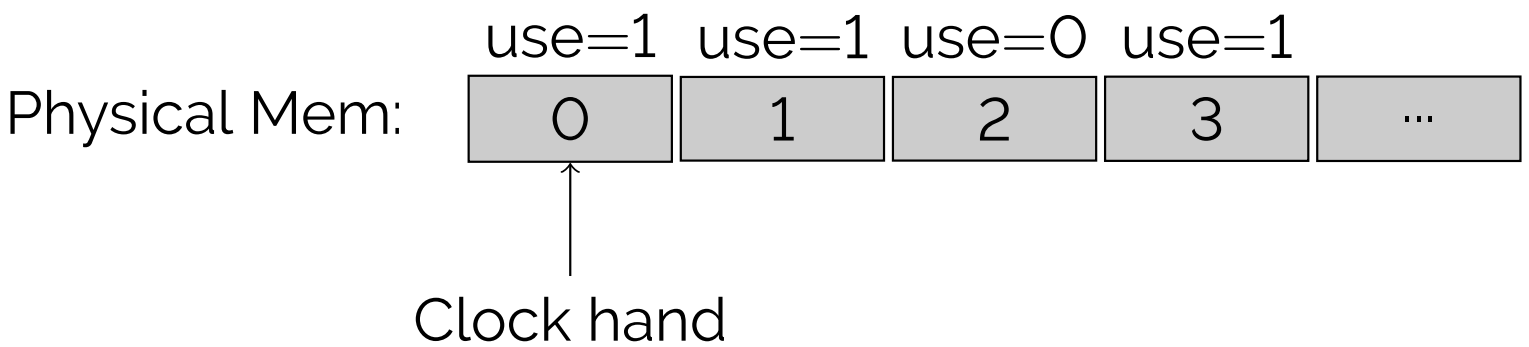
A cheap LRU approximation

Reference (or use) bits

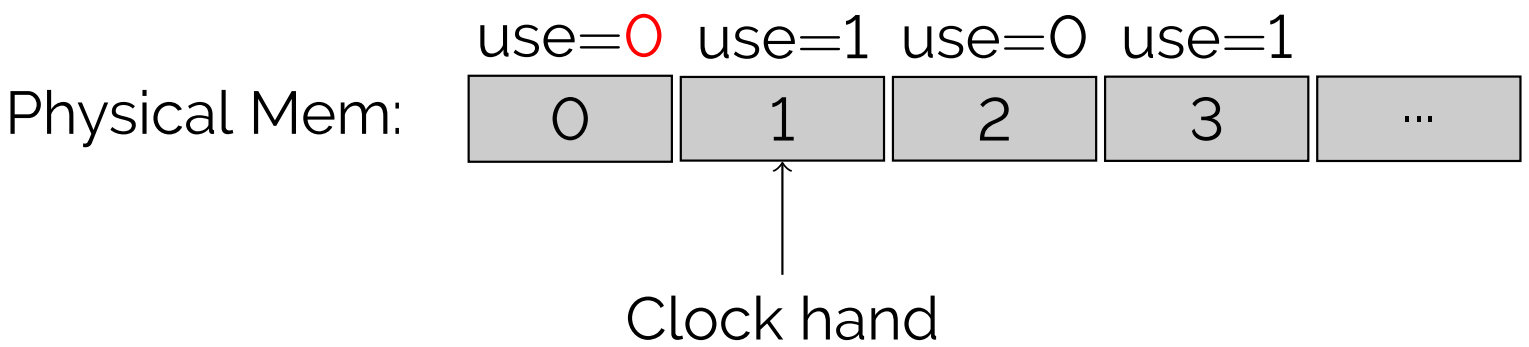
set upon access by HW, cleared by OS

clock algorithm (by Corbato, 1969)

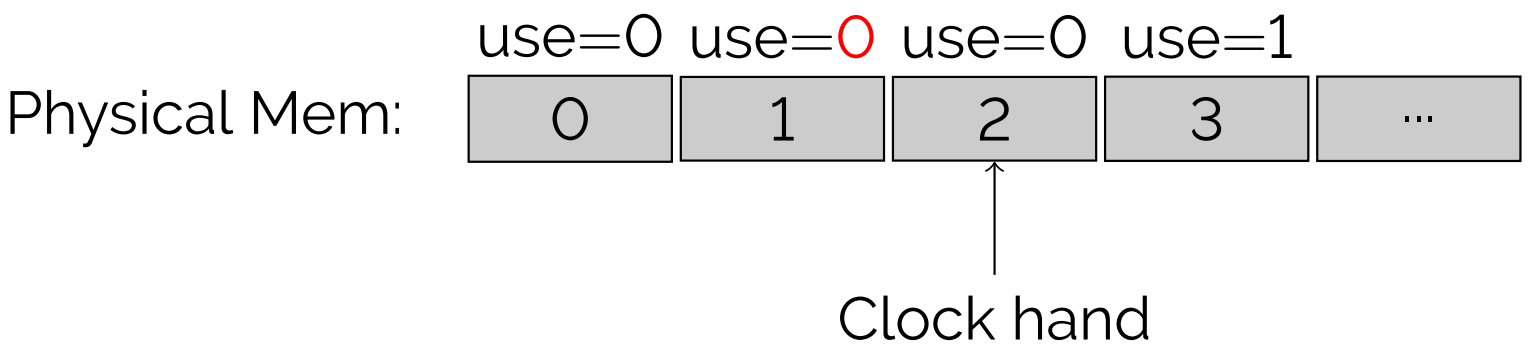
Clock: Look For a Page



Clock: Look For a Page

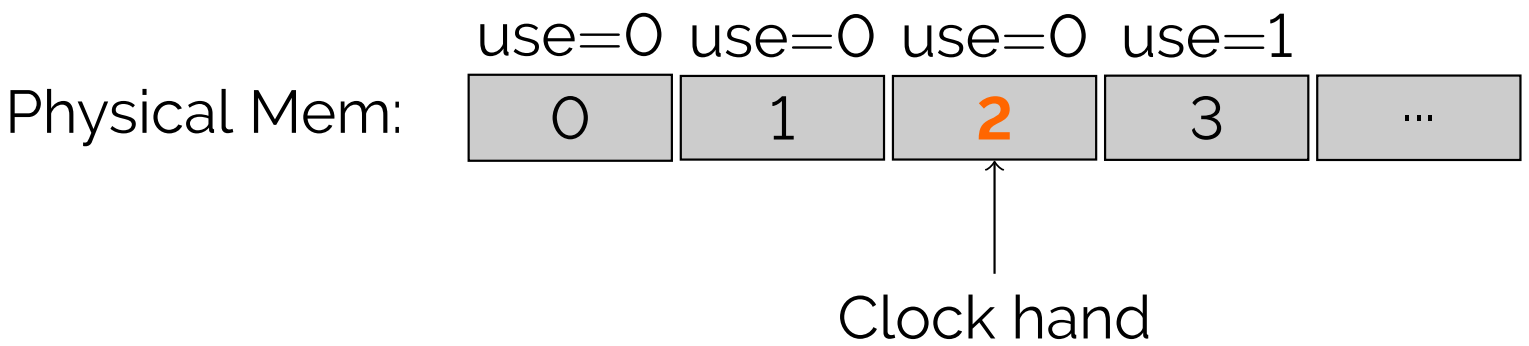


Clock: Look For a Page

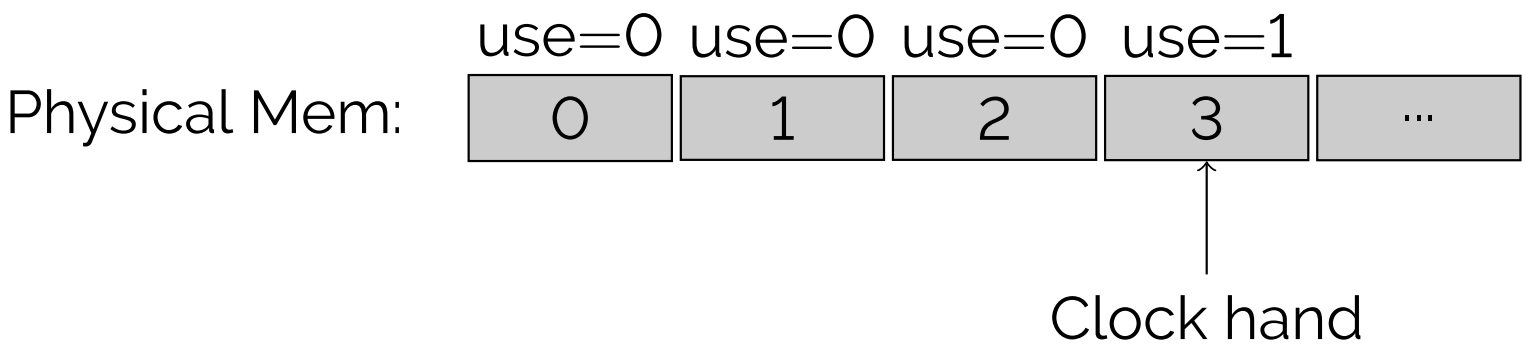


Clock: Look For a Page

evict **page 2** because it has not been recently used

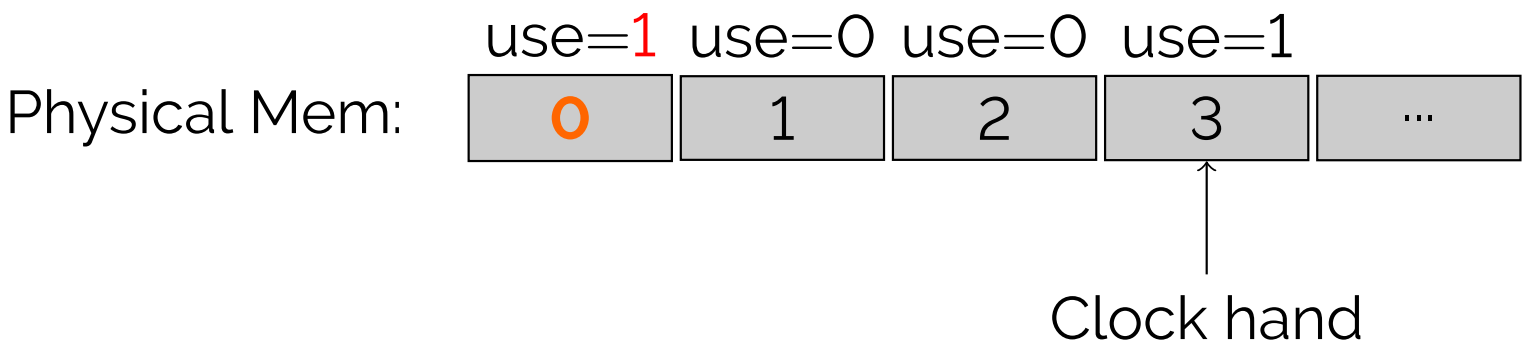


Clock

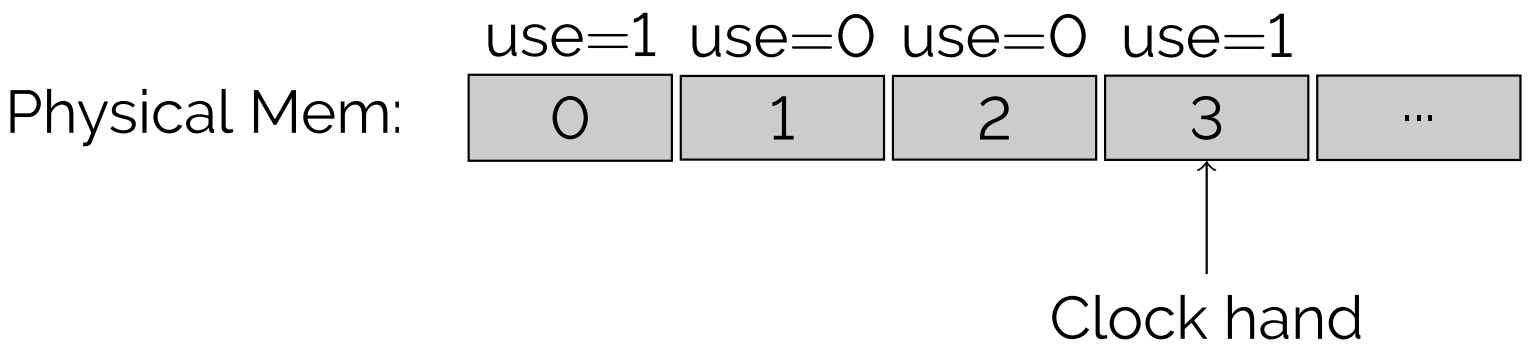


Clock

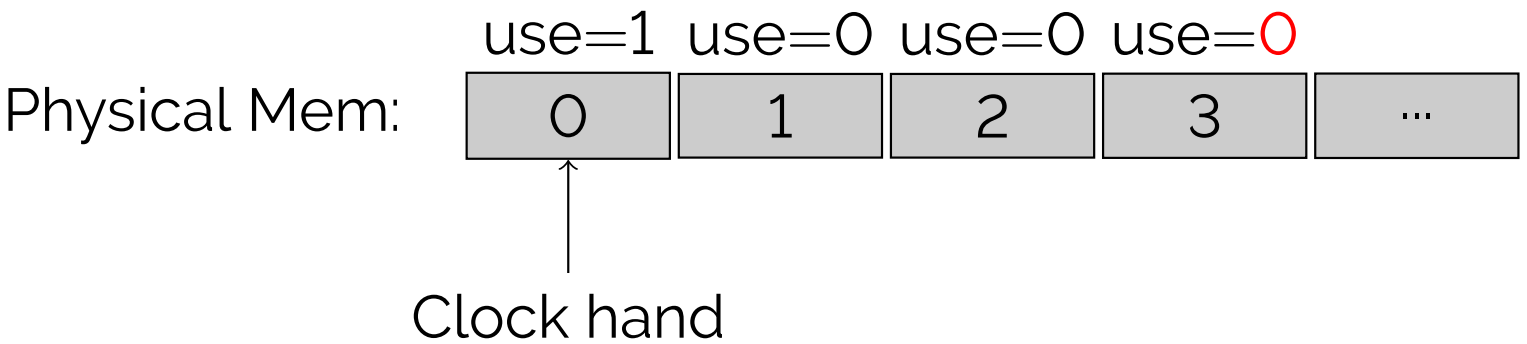
page 0 is accessed



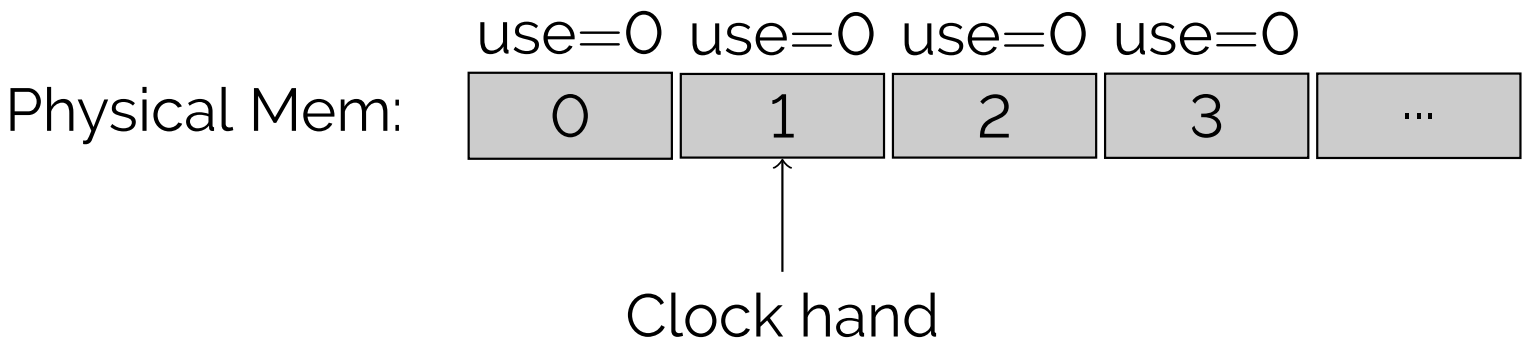
Clock: Look For a Page



Clock: Look For a Page

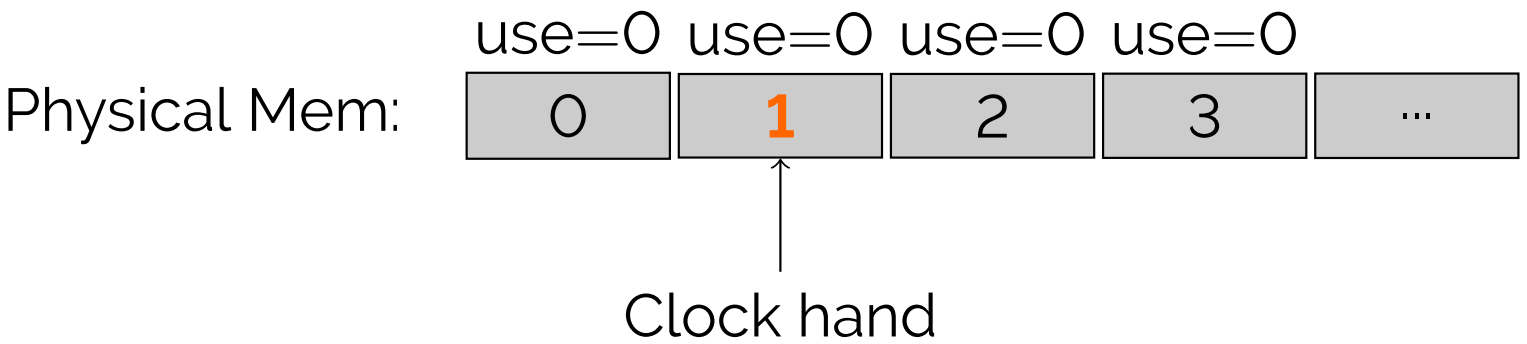


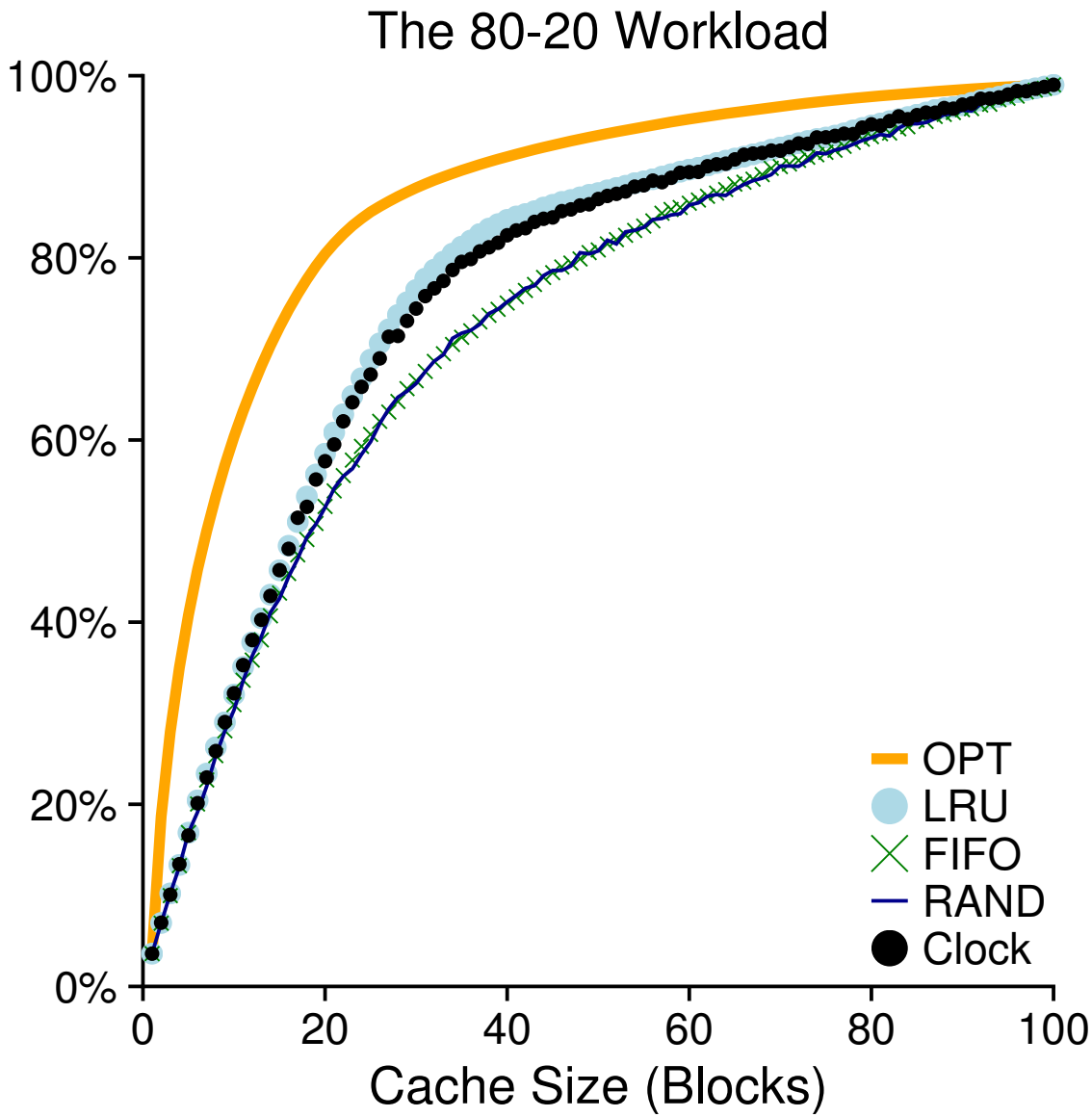
Clock: Look For a Page



Clock: Look For a Page

evict **page 1** because it has not been recently used




The 80-20 Workload
100%
80%
60%
40%
20%
0%
0
20
40
60
80
100
Cache Size (Blocks)
OPT
LRU
FIFO
RAND
Clock

Clock
Considering Dirty Pages

Eviction of clean pages is free
(*just reuse*)

Eviction of dirty pages is costly
(*needs writing back to the disk*)

- Some VM systems prefer to evict clean pages over dirty pages.
  - HW support: modified (dirty) bit per page
    - Set any time a page is written
  - The clock algorithm,for example, could be changed to to evict unused clean pages first, then unused dirty, and so forth.

Other Replacement Policies

Memory hogs: programs that use a lot of memory and make it hard for other programs to run.

Policies thus far (e.g., LRU) are susceptible to hogging, and don't share memory fairly among processes.

Segmented FIFO

one queue per process with a *resident set size* (RSS)

Segmented FIFO & Second-chance lists

After RSS, a page is removed to a clean or dirty page list.

A free page is taken first from the clean list.

Other Replacement Policies

2Q (Linux version)

- Active & inactive lists
- OSTEP ...

Miscellaneous

Swap Daemon

When Replacements Really Occur?

Not a good practice to evict after the memory is full

When the OS notices that there are fewer than L pages available, a background thread runs and evicts pages until there are H pages available.

The background thread: **swap daemon** or **page daemon**

Clustering or grouping of writes $\Rightarrow$ higher disk efficiency

Aside: Do Work in the Background

When you have some work to do, it is often a good idea to do it in the background to increase efficiency and to allow for grouping of operations.
Operating systems often do work in the background; for example,many systems buffer file writes in memory before actually writing the data to disk.

- increased disk efficiency
- improved latency of writes
- the possibility of work reduction
- better use of idle time

» Plan for your idle time

Lazy Optimizations

Demand Paging

When to bring a page into memory?

OS brings the page (e.g. a code page in the file system) into memory when it is accessed.

Mechanism: Present bit

Lazy Optimizations

Demand Zeroing

When to really serve a request for a new page?

Without demand zeroing: finding a free page, zeroing it (why?) and mapping it into the address space

With demand zeroing:

when the page is requested, OS does very little work: **puts an entry in the page-table that marks the page inaccessible**.

the OS does the needed work, if the process reads or writes the page (within the exception handler).

Lazy Optimizations

Copy-On-Write (COW)

When to copy a page from one address space to another?

Instead of copying it right away, OS can map and mark it read-only in both address spaces.

If one of the address spaces tries to write to the page $\rightarrow$ trap into the OS.

This is at this time when OS (lazily) allocates a new page, fill it with the data, and map this new page into the address space of the faulting process.

Useful for shared libraries, and fork

Aside: Be Lazy

Laziness can put off work until later, which is beneficial within an OS for a number of reasons.

First, putting off work might reduce the latency of the current operation, thus improving responsiveness; for example, operating systems often report that writes to a file succeeded immediately, and only write them to disk later in the background.

Second, and more importantly, laziness sometimes obviates the need to do the work at all;for example, delaying a write until the file is deleted removes the need to do the write at all.

» Laziness is sometimes good in life!

Security And Buffer Overflows

Probably the biggest difference between modern and ancient VM systems is the emphasis on security in the modern era.

Machines are more **interconnected.**

Developers have implemented a variety of defensive countermeasures.

Security And Buffer Overflows

Buffer overflow attack

Bug: the developer assumes an input will not be overly long, and thus (trustingly) copies the input into a buffer.

```
1int some_function(char*input) {
2  char dest_buffer[100];
3  strcpy(dest_buffer, input); // oops, unbounded
     copy!
4}
```

If the input is long, it overflows the buffer, thus overwriting memory of the target.

In many cases: crash, but can be used for **injecting codes** ...

Security And Buffer Overflows

Buffer overflow attack

Defense: prevent execution of any code found within certain regions (e.g., the stack) of an address space

Security And Buffer Overflows

Buffer overflow attack

Defense: prevent execution of any code found within certain regions (e.g., the stack) of an address space

Clever attackers are ... clever

Return-oriented programming (ROP)

variable length code
"head" in "the addres"
ISA: extremely dense

```
f7 c7 07 00 00 00        test $0x00000007, %edi
0f 95 45 c3              setnzb -61(%ebp)
```

Starting one byte later, the attacker instead obtains

```
c7 07 00 00 00 0f        movl $0x0f000000, (%edi)
95                       xchg %ebp, %eax
45                       inc %ebp
c3                       ret
```

The Geometry of Innocent Flesh on the Bone: Return-into-libc ...

Security And Buffer Overflows

Buffer overflow attack

Defense: prevent execution of any code found within certain regions (e.g., the stack) of an address space

Clever attackers are ... clever

Return-oriented programming (ROP)

Defense: address space layout randomization (ASLR)

Demo

Security And Buffer Overflows

Buffer overflow attack

Defense: prevent execution of any code found within certain regions (e.g., the stack) of an address space

Clever attackers are ... clever

Return-oriented programming (ROP)

Defense: address space layout randomization (ASLR)

Clever attackers are ... clever

Discussion Question

Find out this new attack!

Memory API

Two types of run-time memory: stack and heap

Stack: locations and deallocations are managed implicitly by the **compiler** (*automatic* memory)

- Declaration: easy, for example `int x;`
- The rest is handled by the compiler: make space on the stack, deallocation after functional return

Heap: all allocations and deallocations are explicitly handled the programmer.

- A heavy responsibility
- the cause of many bugs
- `malloc` , `free` , `sbrk` , `mmap` , ...

Memory API

Two types of run-time memory: stack and heap

Stack: locations and deallocations are managed implicitly by the **compiler** (*automatic* memory)

Declaration: easy, for example `int x;`

The rest is handled by the compiler: make space on the stack, deallocation after functional return

Heap: all allocations and deallocations are explicitly handled the programmer.

A heavy responsibility

the cause of many bugs

`malloc` , `free` , `sbrk` , `mmap` , ...

our focus here

```
1  void func() {
2    int *x = (int*) malloc(10*sizeof(int));
3    ...
4    free (x);
5  }
```

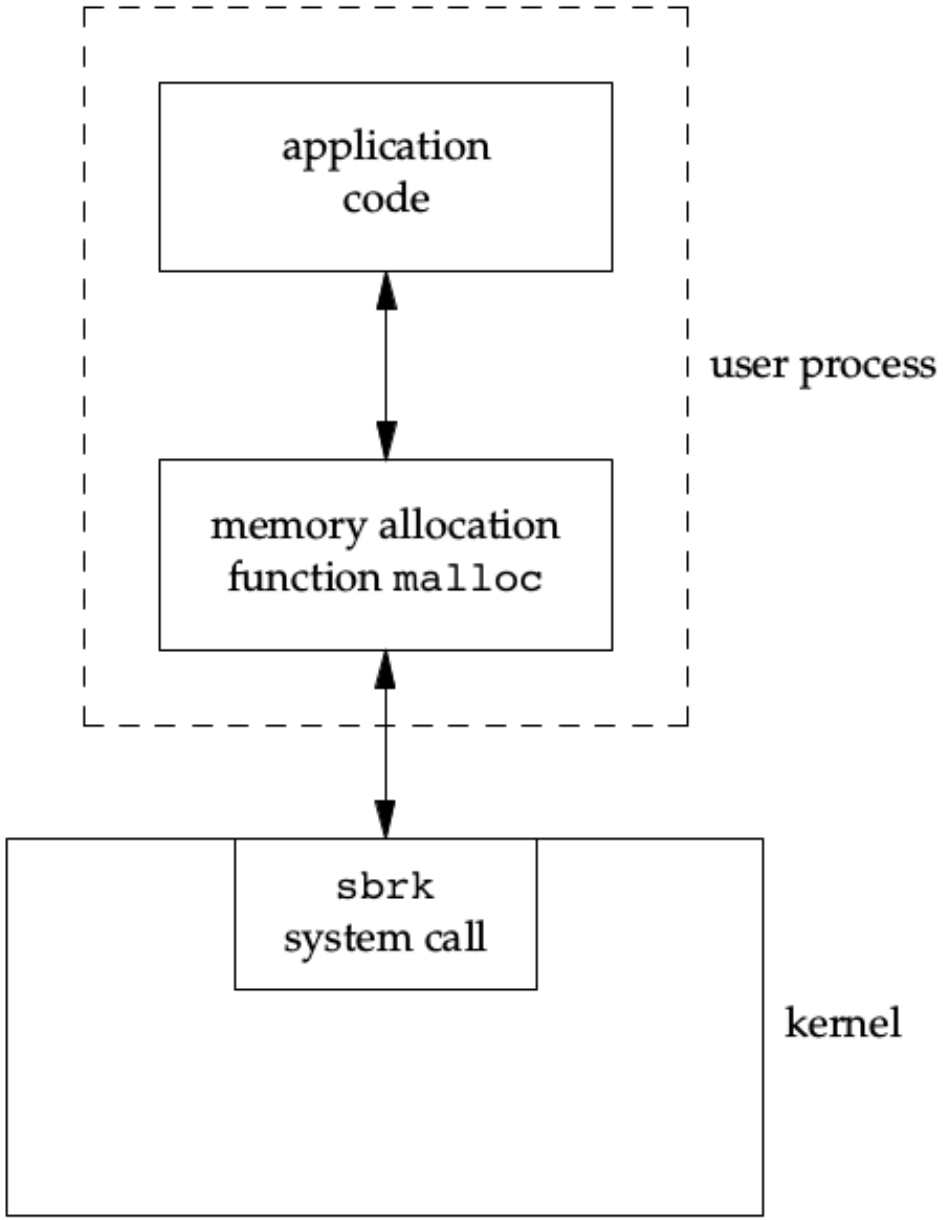
both stack and heap allocation occur on this line

`sizeof` : operator

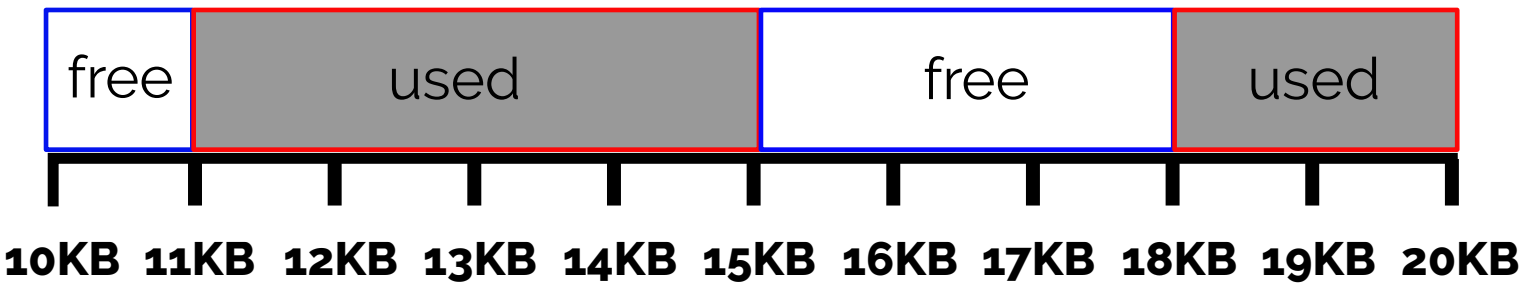
`free` does not take the size, just the address → tracked by the memory-allocation library

`malloc` and `free` **are not system calls**

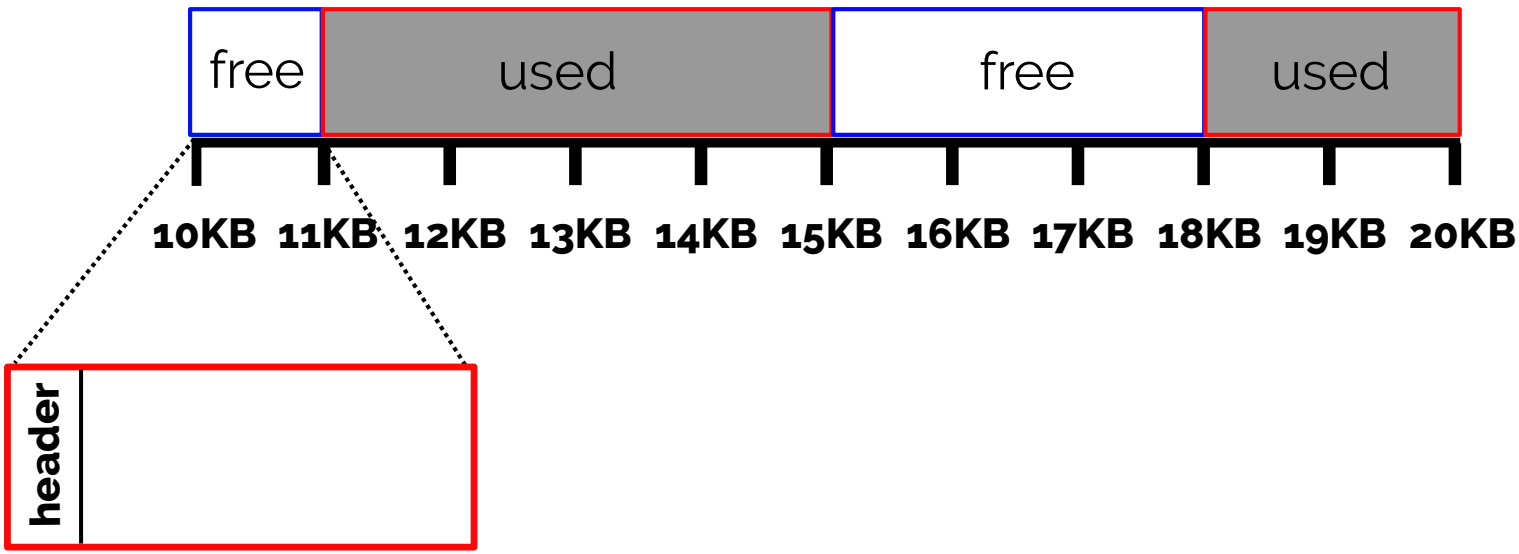
allocating through system calls (such as `brk`)
bookkeeping of free and used spaces
manages the free space


application
code
user process
memory allocation
function malloc
sbrk
system call
kernel

malloc


free
used
free
used
10KB
11KB
12KB
13KB
14KB
15KB
16KB
17KB
18KB
19KB
20KB

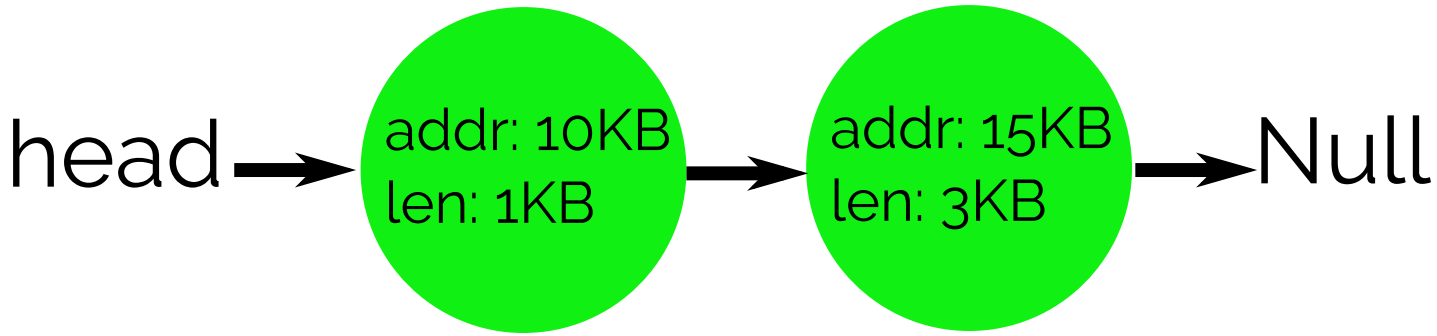
malloc

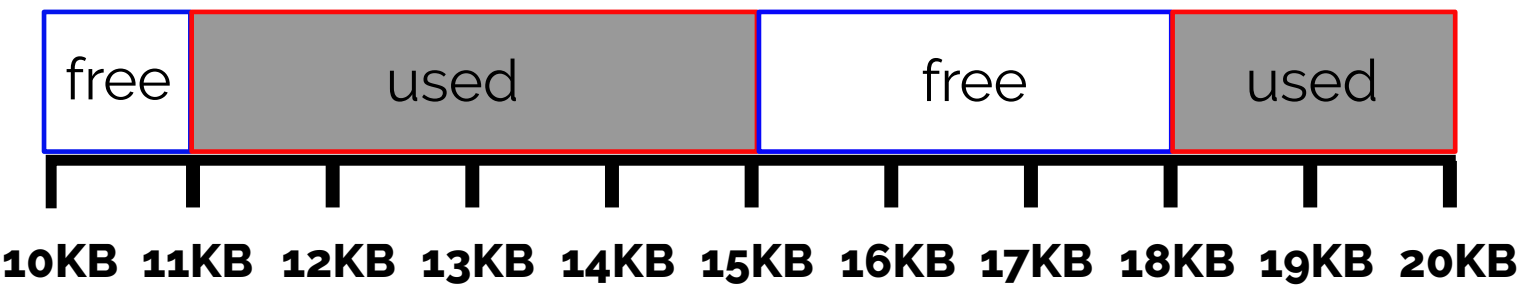


header: size, next*

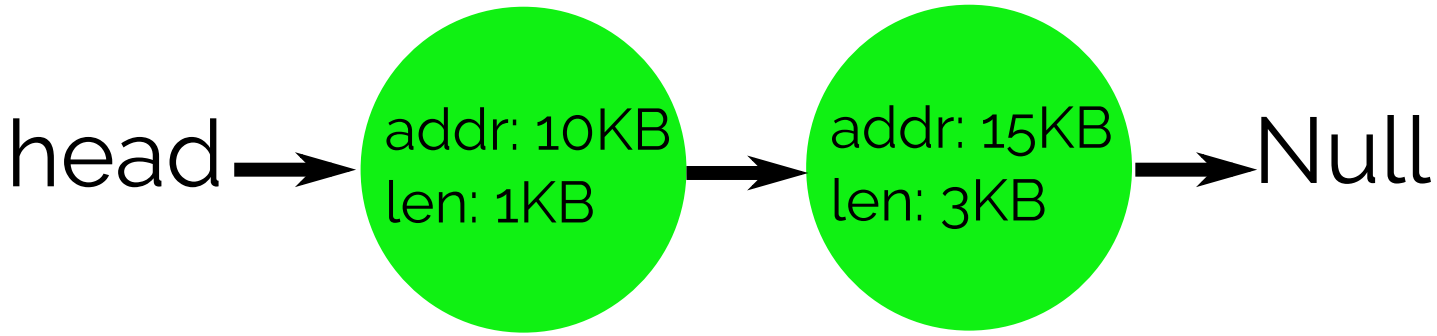
malloc

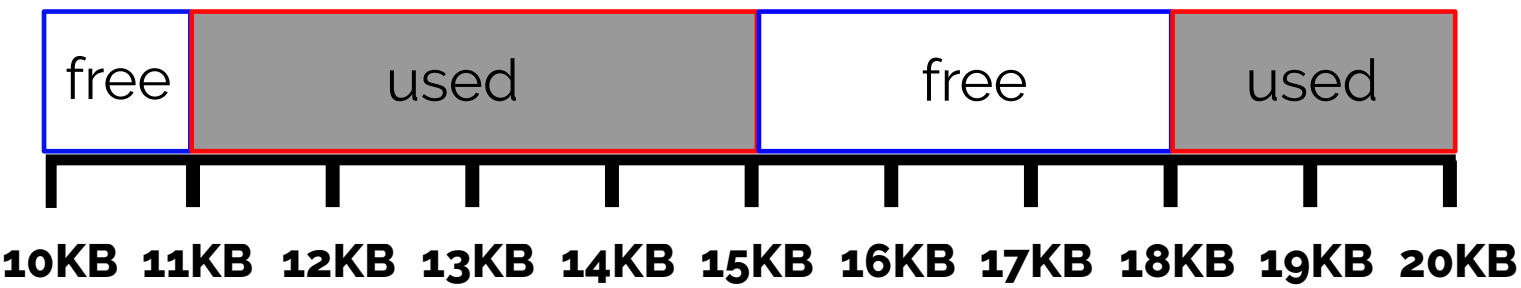
Free list





malloc





malloc(1kB)

Memory management policy is required

Free space management:

easy when free spaces have the same size, for example in paging

challenging when free spaces have variable sizes, for example in segmentation (by OS) and heap (by memory allocation library)

the problem of internal fragmentation

The **ideal allocator** is both fast and minimizes fragmentation

Note

Unfortunately, because the stream of allocation and free requests can be arbitrary (after all, they are determined by the programmer), any particular strategy can do quite badly given the wrong set of inputs

The allocated chunk to a request must have space to accommodate the request. In addition to this preliminary condition, we can have different strategies:

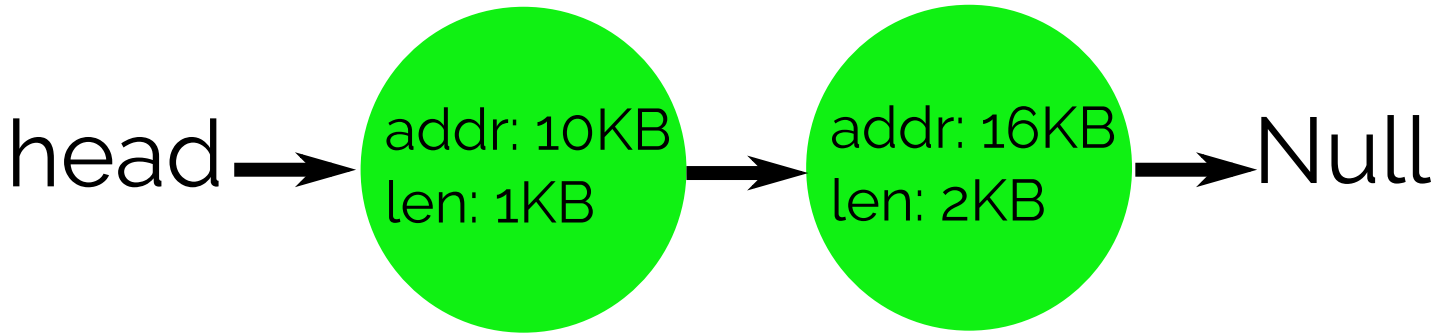
Best Fit: The smallest

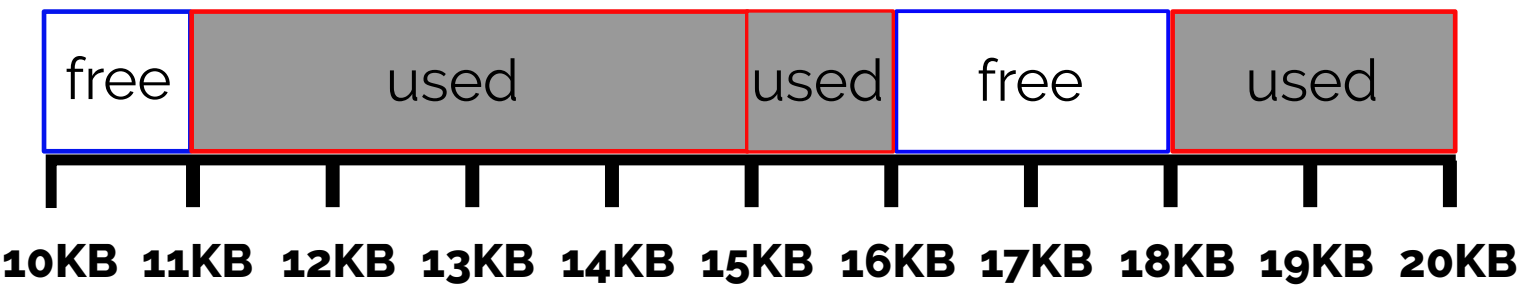
Worst Fit: The largest

First Fit: The first

Next Fit: Start searching the free-list for the first-fit from the last point one was looking

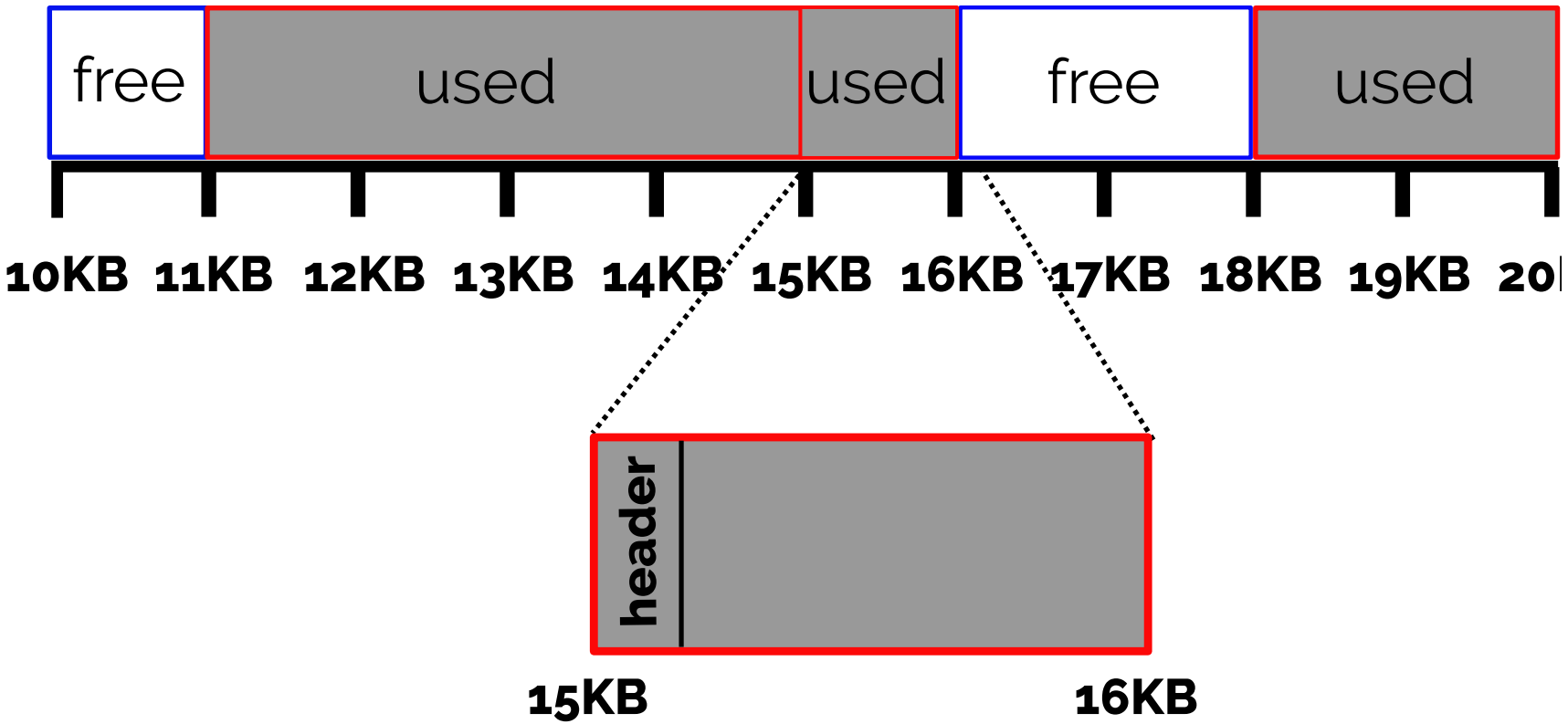
malloc





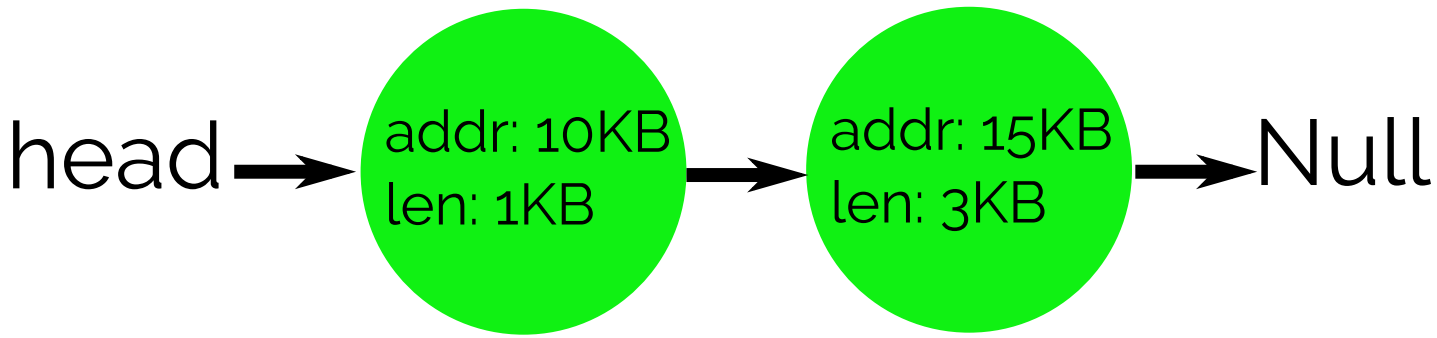
malloc(1kB) = 15KB

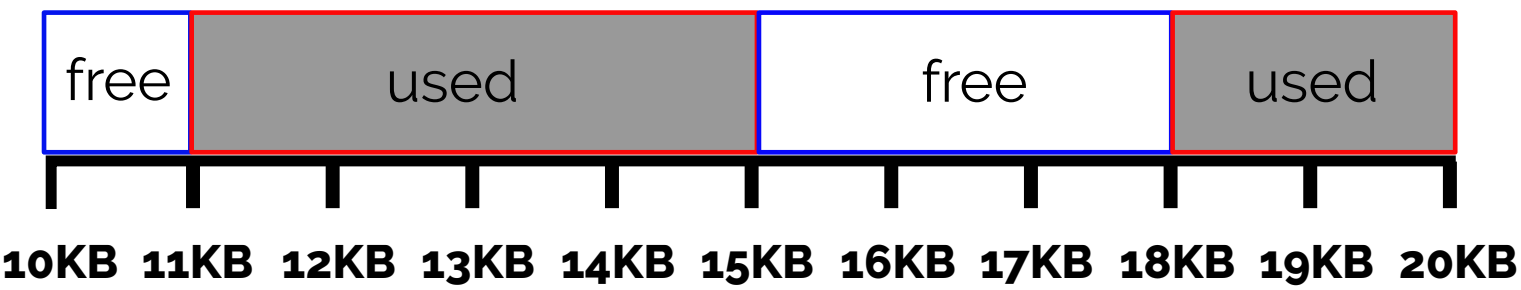
free



header: magic number, size, ...

free





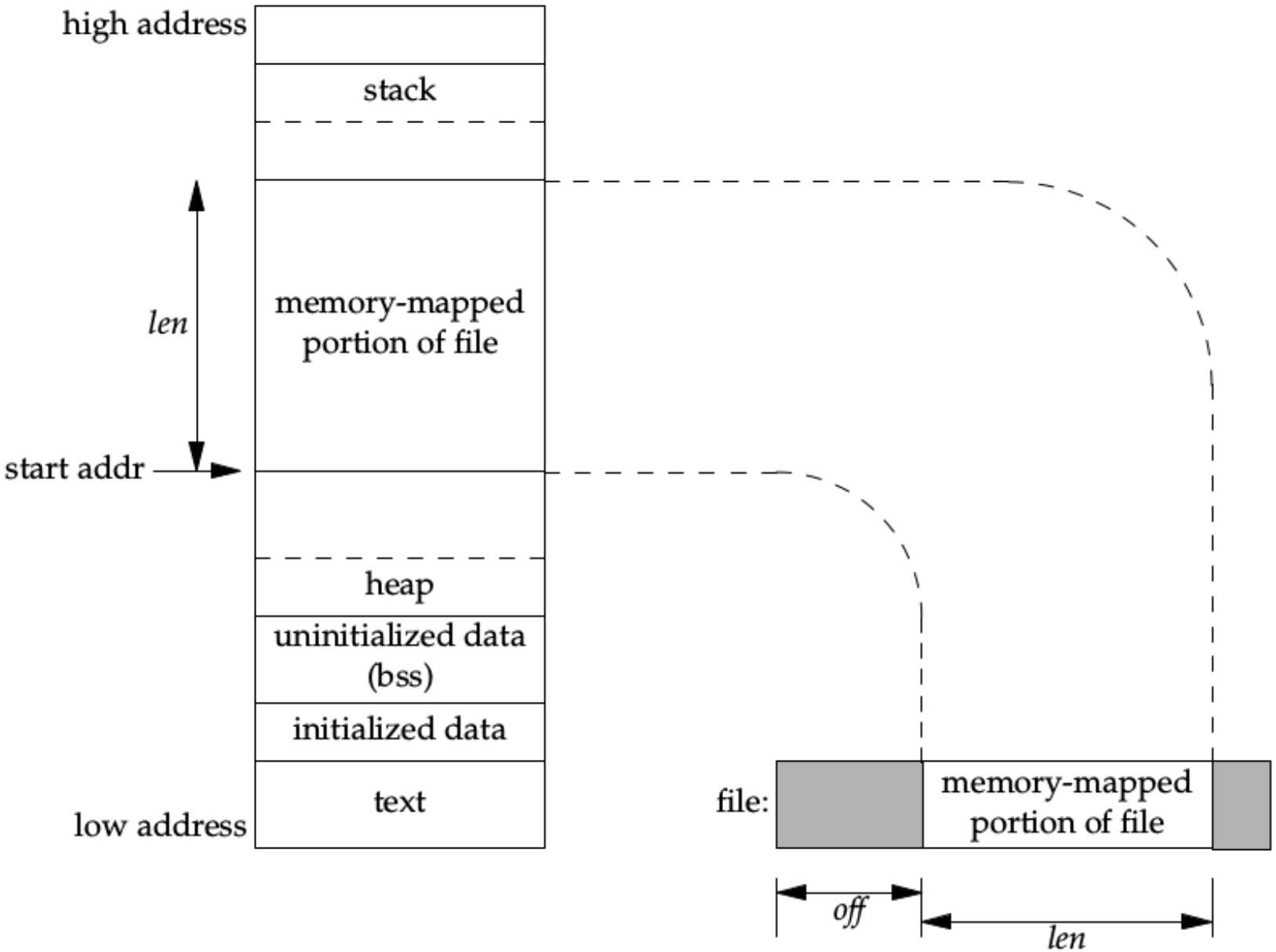
free(15KB)

Underlying System Calls

`sbrk` , `brk` : change the location of the program's **break**: the location of the end of the heap

`mmap` : map or unmap files or devices into memory

Map a file on disk into a buffer in memory


high address
stack
memory-mapped
portion of file
len
start addr
heap
uninitialized data
(bss)
initialized data
text
low address
file:
memory-mapped
portion of file
off
len

Underlying System Calls

`sbrk` , `brk` : change the location of the program's **break**: the location of the end of the heap

`mmap` : map or unmap files or devices into memory

Map a file on disk into a buffer in memory

Create a heap-like region within the address space (**anonymous**: not associated with any particular file but rather with swap space)

Ken Thompson (1943-present)

A computer scientist at Bell labs

Winner of the 1983 Turing Award, (together with Dennis Ritchie) for his work on the Unix operating system.