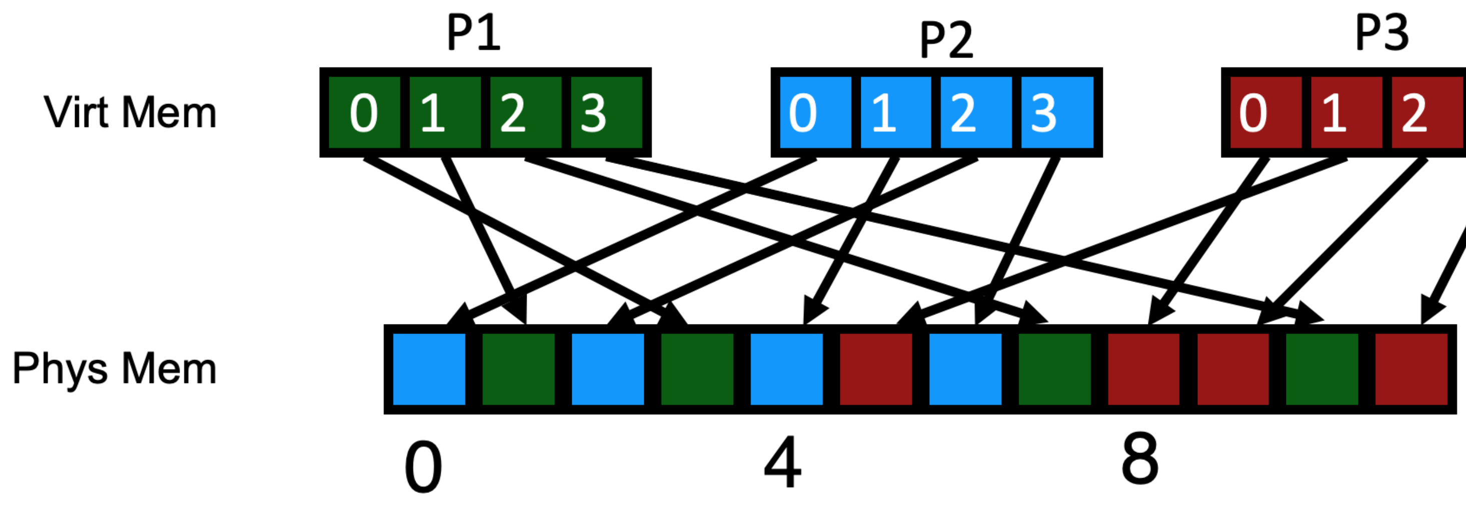
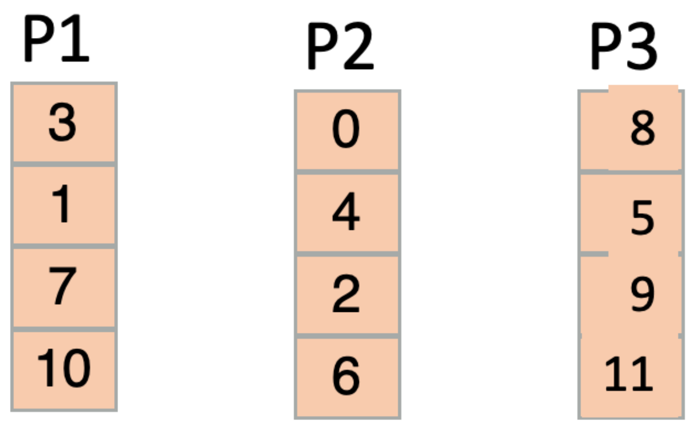
Memory Virtualization

RUTGERS
UNIVERSITY | NEW BRUNSWICK

Use of a page table doubles memory references

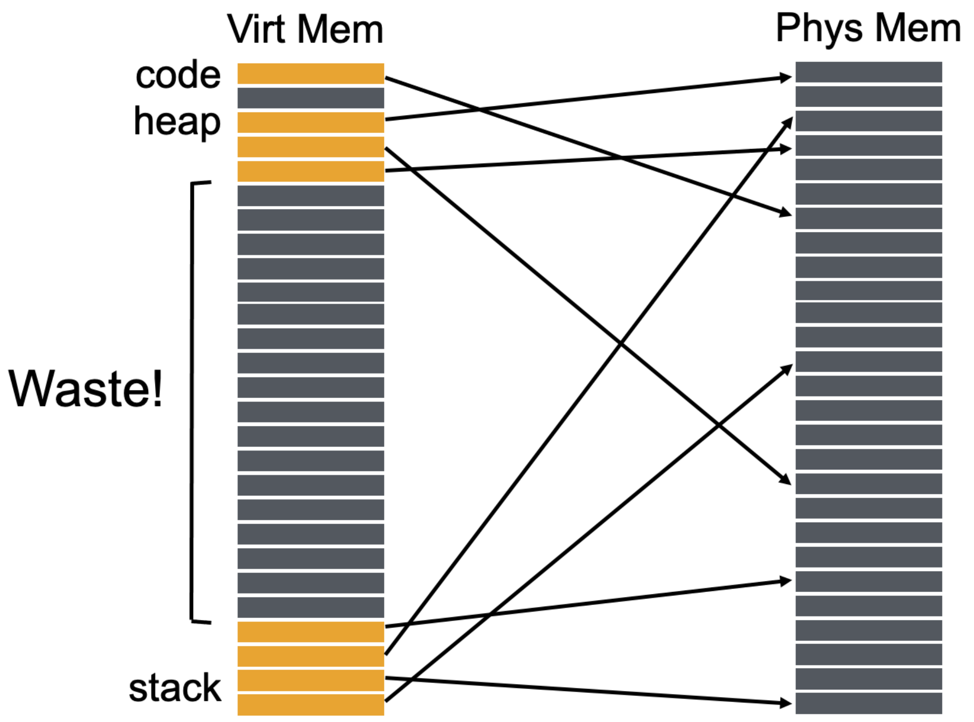


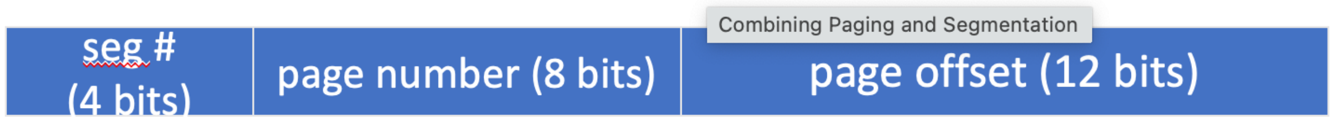


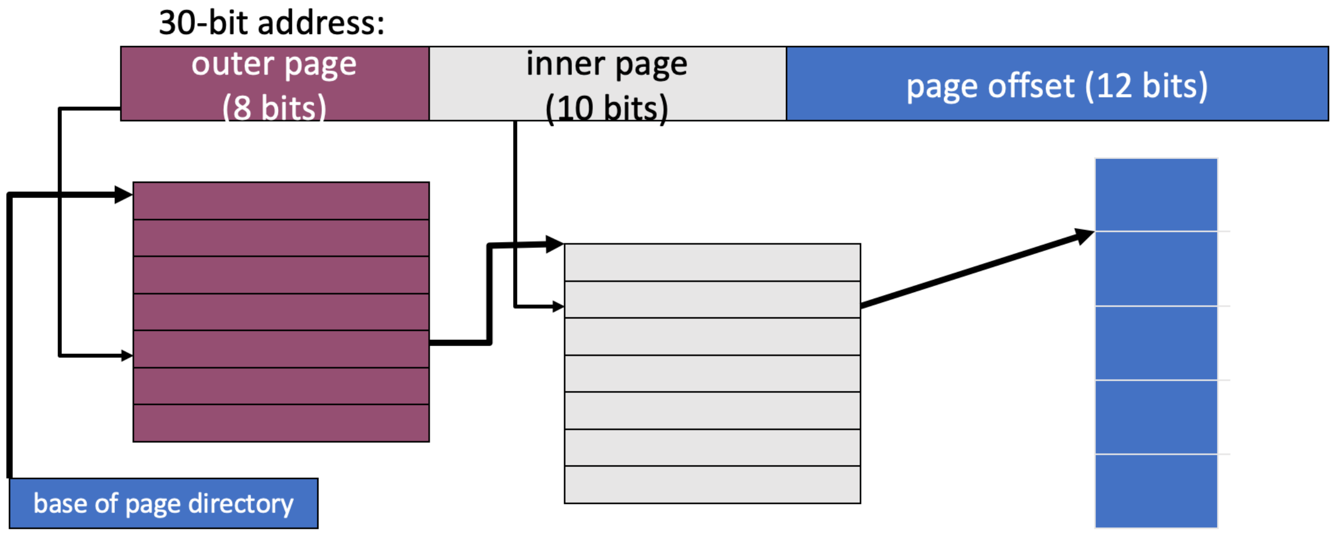
P1	P2	P3
3	0	8
1	4	5
7	2	9
10	6	11

Sparse PTEs

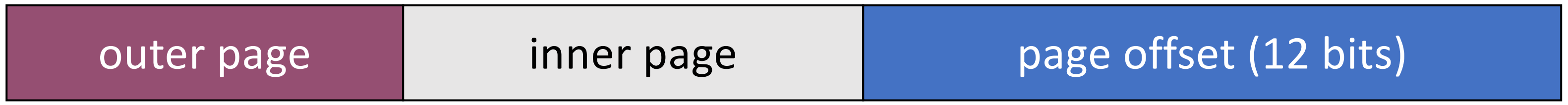
Valid PTEs are continuous







Address format for Multilevel Paging



How should logical address be structured?

- How many bits for each paging level?

Goal?

- Each page table fits within a page
- PTE size * number PTE = page size
  - Assume PTE size = 4 bytes
  - Page size = 2^12 bytes = 4KB
  - number PTE per page = (2^12 bytes per page) / (4 bytes per PTE)
  - → number PTE = 2^10
- → # bits for selecting inner page = 10

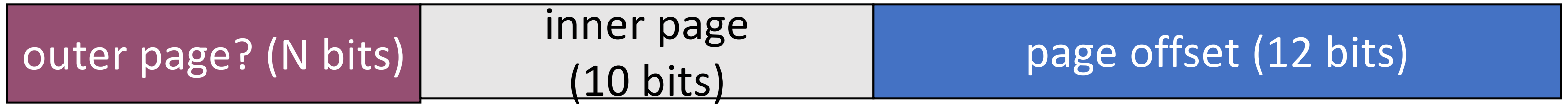
Remaining bits for outer page:

- 30 – 10 – 12 = 8 bits

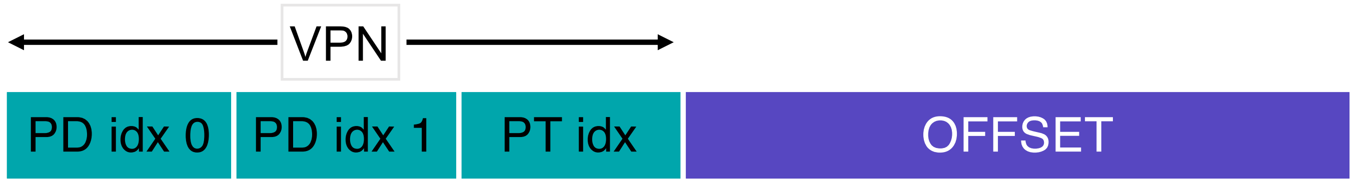
Problem with 2 levels?

Problem: page directory (outer level) may not fit in a page!

Solution:



- Split page directories into pieces
- Use another page dir to refer to the pieces of the page directory



How large is virtual address space with 4 KB pages, 4 byte PTEs, each page table fits in page, given 1, 2, 3 levels?

4KB / 4 bytes → 1K entries per level

1 level: 1K * 4K = **2^22** = 4 MB

2 levels: 1K * 1K * 4K = **2^32** ≈ 4 GB

3 levels: 1K * 1K * 1K * 4K = **2^42** ≈ 4 TB

Review: Paging pros and cons

Advantages

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

Disadvantages

- Page tables can get big
  - Must have one entry for every page of address space
- Accessing page tables is too slow [address this shortly]
  - Doubles the number of memory references per instruction

Translation Steps

H/W: for each mem reference:

(cheap)	1. extract **VPN** (virt page num) from **VA** (virt addr)
(cheap)	2. calculate addr of **PTE** (page table entry)
(expensive)	3. read **PTE** from memory
(cheap)	4. extract **PFN** (page frame num)
(cheap)	5. build **PA** (phys addr)
(expensive)	6. read contents of **PA** from memory into register

Which steps are expensive?

Which expensive step(s) can we (not) avoid?

3) Let's try to avoid having to read PTE from memory!

Translation Lookaside Buffers

How can page translations be made faster?
What is the basic idea of a TLB (Translation Lookaside Buffer)?
What types of workloads perform well with TLBs?
How do TLBs interact with context-switches?

Example: Array Iterator

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

Assume ‘a’ starts at 0x3000

Ignore instruction fetches

What virtual addresses?

load 0x3000

load 0x3004

load 0x3008

load 0x300C

...

What physical addresses?

$\boxed{\text{load 0x100C}}$

load 0x7000

$\boxed{\text{load 0x100C}}$

load 0x7004

$\boxed{\text{load 0x100C}}$

load 0x7008

$\boxed{\text{load 0x100C}}$

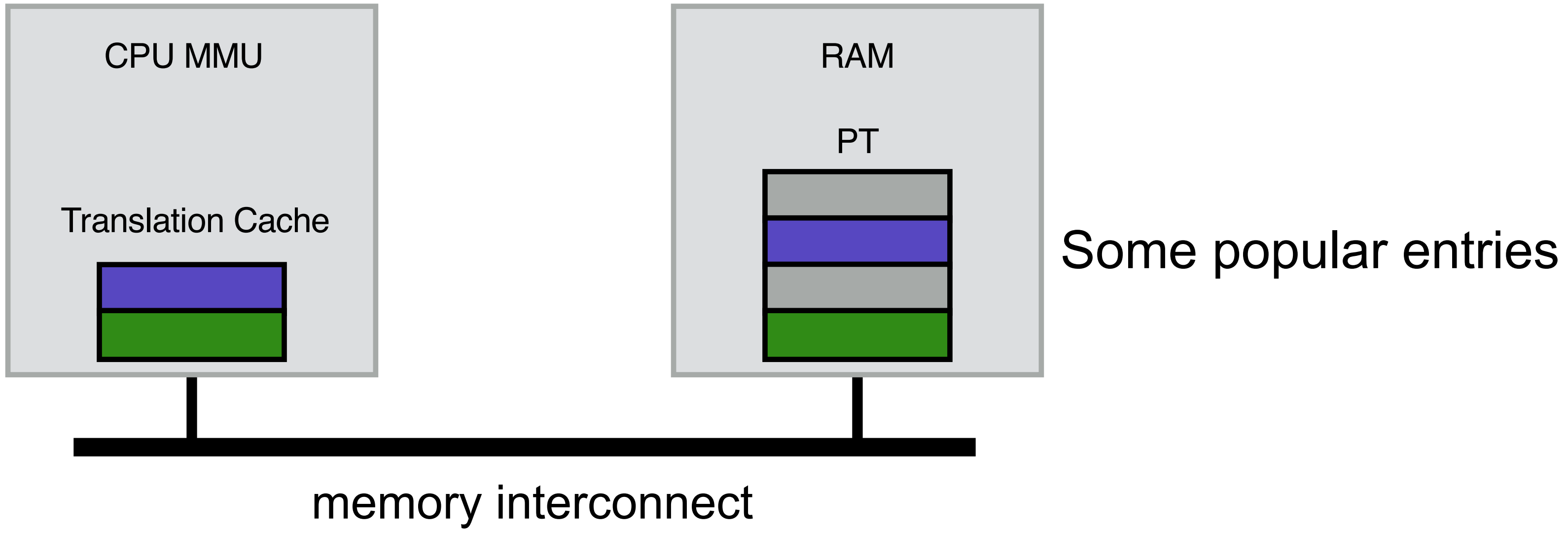
load 0x700C

Observation:

Repeatedly access same PTE because program repeatedly accesses same virtual page

Strategy: Cache Page Translations

We couldn't store entire page table in MMU, but we can store a fast cache



TLB: **T**ranslation **L**ookaside **B**uffer

TLB Control Flow

```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
```

TLB Control Flow

```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
if (Success == True)   // TLB Hit
    if (CanAccess(TlbEntry.ProtectBits) == True)
        Offset   = VirtualAddress & OFFSET_MASK
        PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
        Register = AccessMemory(PhysAddr)
```

TLB Control Flow

```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
if (Success == True)   // TLB Hit
    if (CanAccess(TlbEntry.ProtectBits) == True)
        Offset   = VirtualAddress & OFFSET_MASK
        PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
        Register = AccessMemory(PhysAddr)
    else
        RaiseException(PROTECTION_FAULT)
else                  // TLB Miss
```

TLB Control Flow

```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
if (Success == True)    // TLB Hit
    if (CanAccess(TlbEntry.ProtectBits) == True)
        Offset   = VirtualAddress & OFFSET_MASK
        PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
        Register = AccessMemory(PhysAddr)
    else
        RaiseException(PROTECTION_FAULT)
else                   // TLB Miss
    PTEAddr = PTBR + (VPN * sizeof(PTE))  // assume simple linear page table
    PTE = AccessMemory(PTEAddr)
```

TLB Control Flow

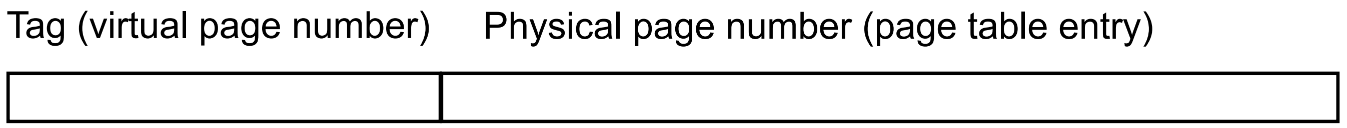
```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
if (Success == True)   // TLB Hit
    if (CanAccess(TlbEntry.ProtectBits) == True)
        Offset   = VirtualAddress & OFFSET_MASK
        PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
        Register = AccessMemory(PhysAddr)
    else
        RaiseException(PROTECTION_FAULT)
else                  // TLB Miss
    PTEAddr = PTBR + (VPN * sizeof(PTE))  // assume simple linear page table
    PTE = AccessMemory(PTEAddr)
    if (PTE.Valid == False)
        RaiseException(SEGMENTATION_FAULT)
    else if (CanAccess(PTE.ProtectBits) == False)
        RaiseException(PROTECTION_FAULT)
```

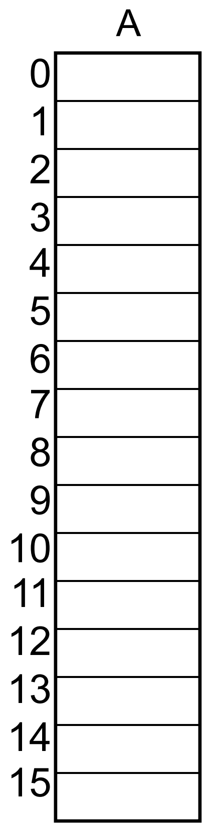
TLB Control Flow

```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
if (Success == True)   // TLB Hit
    if (CanAccess(TlbEntry.ProtectBits) == True)
        Offset   = VirtualAddress & OFFSET_MASK
        PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
        Register = AccessMemory(PhysAddr)
    else
        RaiseException(PROTECTION_FAULT)
else                  // TLB Miss
    PTEAddr = PTBR + (VPN * sizeof(PTE))  // assume simple linear page table
    PTE = AccessMemory(PTEAddr)
    if (PTE.Valid == False)
        RaiseException(SEGMENTATION_FAULT)
    else if (CanAccess(PTE.ProtectBits) == False)
        RaiseException(PROTECTION_FAULT)
    else
        TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
        RetryInstruction()
```

TLB Organization

TLB Entry


Tag (virtual page number)
Physical page number (page table entry)



Direct mapped (num sets = 16)

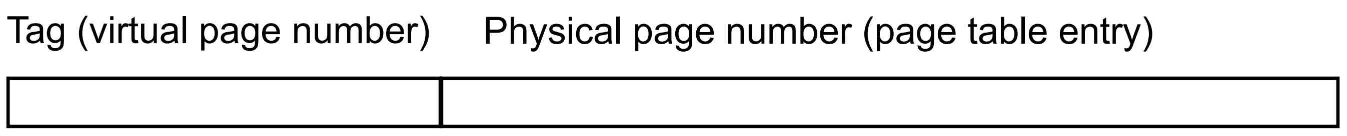
Lookup
- Calculate set (tag % num_sets)
- Search for tag within resulting set

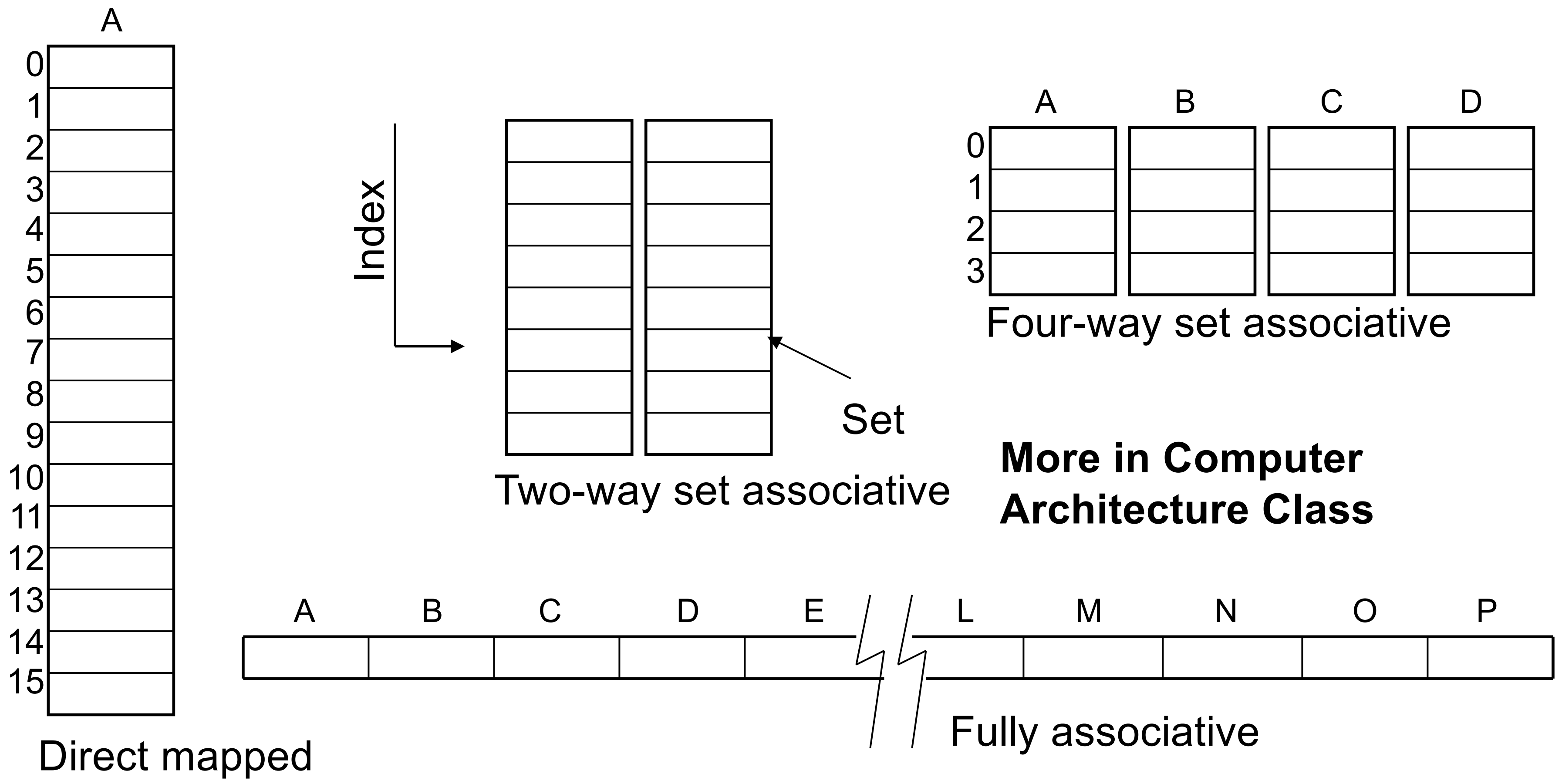
Where is VPN (tag) 18 located?

2

TLB Organization

TLB Entry


Tag (virtual page number)
Physical page number (page table entry)



More in Computer Architecture Class

TLB Associativity Trade-offs

Higher associativity

+ Better utilization, fewer collisions

– Slower

– More hardware

Lower associativity

+ Fast

+ Simple, less hardware

– Greater chance of collisions

TLBs are usually fully associative

Array Iterator (with TLB)

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

Assume following virtual address stream:
load 0x1000

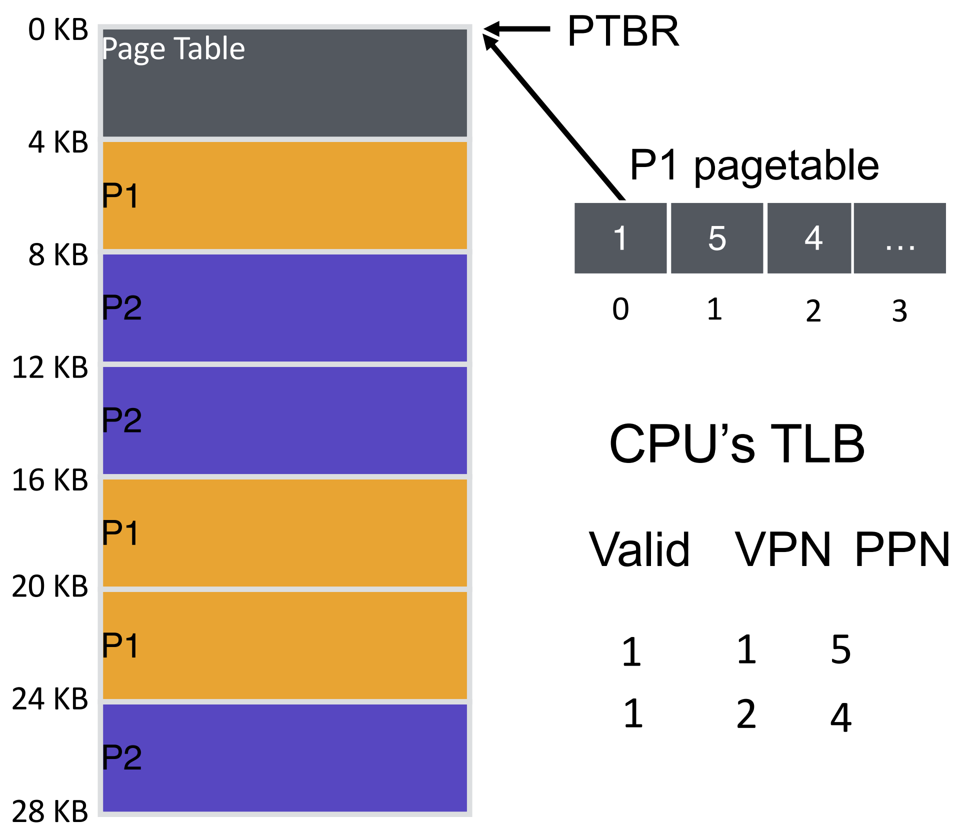
load 0x1004

load 0x1008

load 0x100C
…

What will TLB behavior look like?

TLB Accesses: Sequential Example



CPU's TLB

Valid	VPN	PPN
1	1	5
1	2	4

Virtual	Physical
load 0x1000	load 0x0004 load 0x5000
load 0x1004	(TLB hit) load 0x5004
load 0x1008	(TLB hit) load 0x5008
load 0x100c	(TLB hit) load 0x500C
…	…
load 0x2000	load 0x0008 load 0x4000
load 0x2004	(TLB hit) load 0x4004

Performance Of TLB?

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

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

Miss rate?
  2/2048 = 0.1%

Hit rate? (1 – miss rate)
  99.9%

Would hit rate get better or worse with smaller pages?
  Worse

TLB

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

Increase page size

- Fewer unique page translations needed to access same amount of memory

TLB “reach” in terms of physical memory size:

- Number of TLB entries * Page Size

“Huge pages” used in many real systems.

TLB Performance with Workloads

Sequential array accesses almost always hit in TLB

- Very fast!

What access pattern will be slow?

- Highly random, with no repeat accesses

Workload Access Patterns

Workload A

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

Workload B

```
int sum = 0;
srand(1234);
for (i=0; i<1000; i++) {
   sum += a[rand() % N];
}
srand(1234);
for (i=0; i<1000; i++) {
   sum += a[rand() % N];
}
```

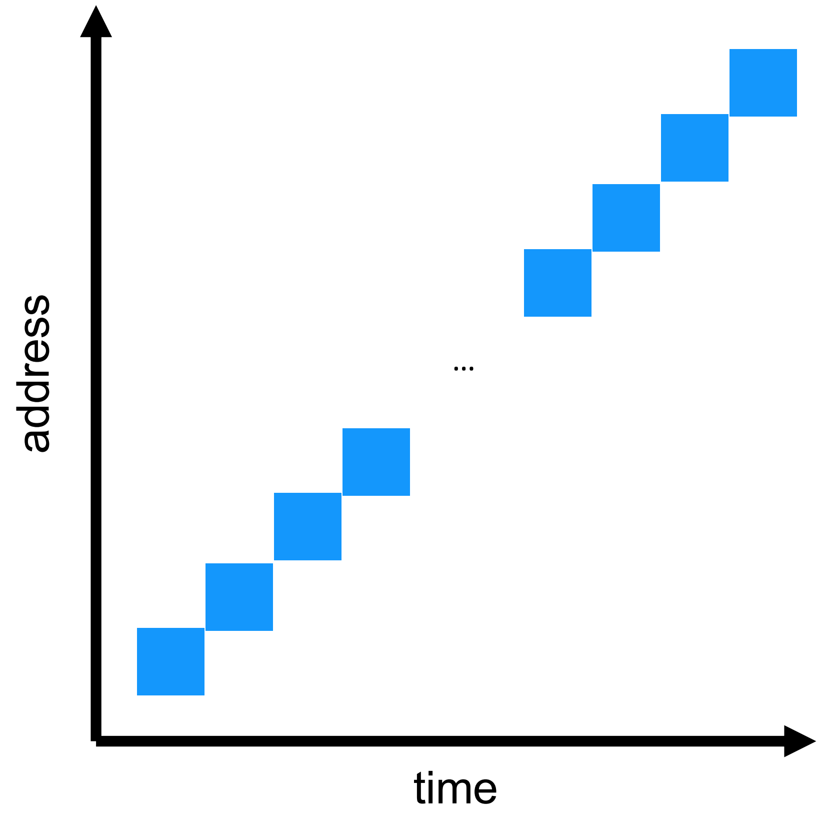
Workload
Access Patterns

Workload A

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

Spatial Locality

Sequential Accesses



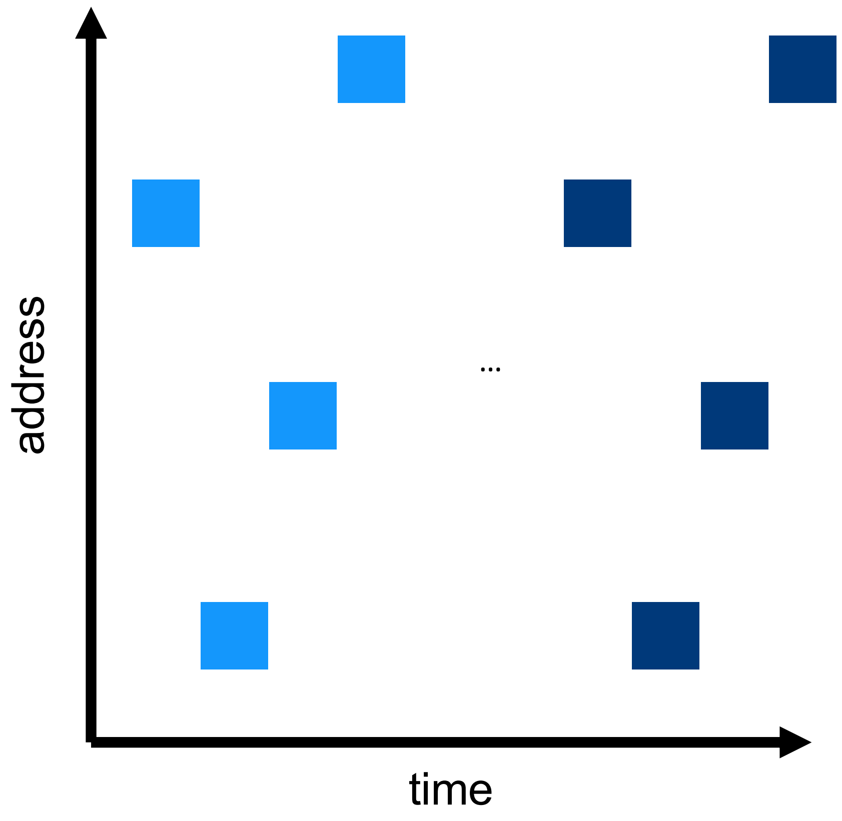
Workload
Access Patterns

Workload B

```
int sum = 0;
srand(1234);
for (i=0; i<1000; i++) {
   sum += a[rand() % N];
}
srand(1234);
for (i=0; i<1000; i++) {
   sum += a[rand() % N];
}
```

Temporal Locality

Repeated Accesses (even if random locations)



Workload Locality

Spatial Locality: future access will be to nearby addresses

Temporal Locality: future access will be repeats to the same data

What TLB characteristics are best for each type?

Spatial:

- Access same page repeatedly; need same VPN → PFN translation
- Same TLB entry re-used

Temporal:

- Access same address near in future
- Same TLB entry re-used in near future
- How near in future? How many TLB entries are there?

Differentiating processes

- So far, we assumed VPNs are unique. They are not (across multiple processes)!
- Option 1: Flush TLBs upon every context switch (valid = 0)
  - Problem: poor performance after each context switch
- Option 2: Attach "address space identifier" to TLB entry

VPN	PFN	valid	prot
10	100	1	rwx
—	—	0	—
10	170	1	rwx
—	—	0	—

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

A full system with TLBs

On TLB miss: lookups with more paging levels more expensive

How much does a miss cost?

Assume 3-level page table, 256-byte pages, 16-bit addresses
Assume ASID of current process is 211
How many physical accesses for each instruction?

(a) 0xAA10: movl 0x1111, %edi

(b) 0xBB13: addl $0x3, %edi

(c) 0x0519: movl %edi, 0xFF10

ASID	VPN	PFN	Valid
211	0xbb	0x91	1
211	0xff	0x23	1
122	0x05	0x91	1
211	0x05	0x12	0

0xaa: (TLB miss -> 3 for addr trans) + 1 instr fetch
0x11: (TLB miss -> 3 for addr trans) + 1 movl

Total: 8

0xbb: (TLB hit -> 0 for addr trans) + 1 instr fetch from 0x9113

Total: 1

0x05: (TLB miss -> 3 for addr trans) + 1 instr fetch
0xff: (TLB hit -> 0 for addr trans) + 1 movl into 0x2310

Total: 5

Summary: Better page tables

Problem:
Simple linear page tables require too much contiguous memory

Many options for efficiently organizing page tables

If OS traps on TLB miss, OS can use any data structure

- e.g., inverted page tables

If Hardware handles TLB miss, page tables must follow specific data structure that hardware knows how to “walk”

- Multi-level page tables used in x86 architecture
- Each page table must fit within a page

Next Topic: What if desired address spaces do not fit in physical memory?

Virtual Memory

Questions answered:

- How to run process when not enough physical memory?
- When should a page be moved from disk to memory?
- What page in memory should be replaced?
- How can the LRU page be approximated efficiently?

Motivation

OS goal: Support processes when not enough physical memory

- Single process with very large address space
- Multiple processes with combined address spaces

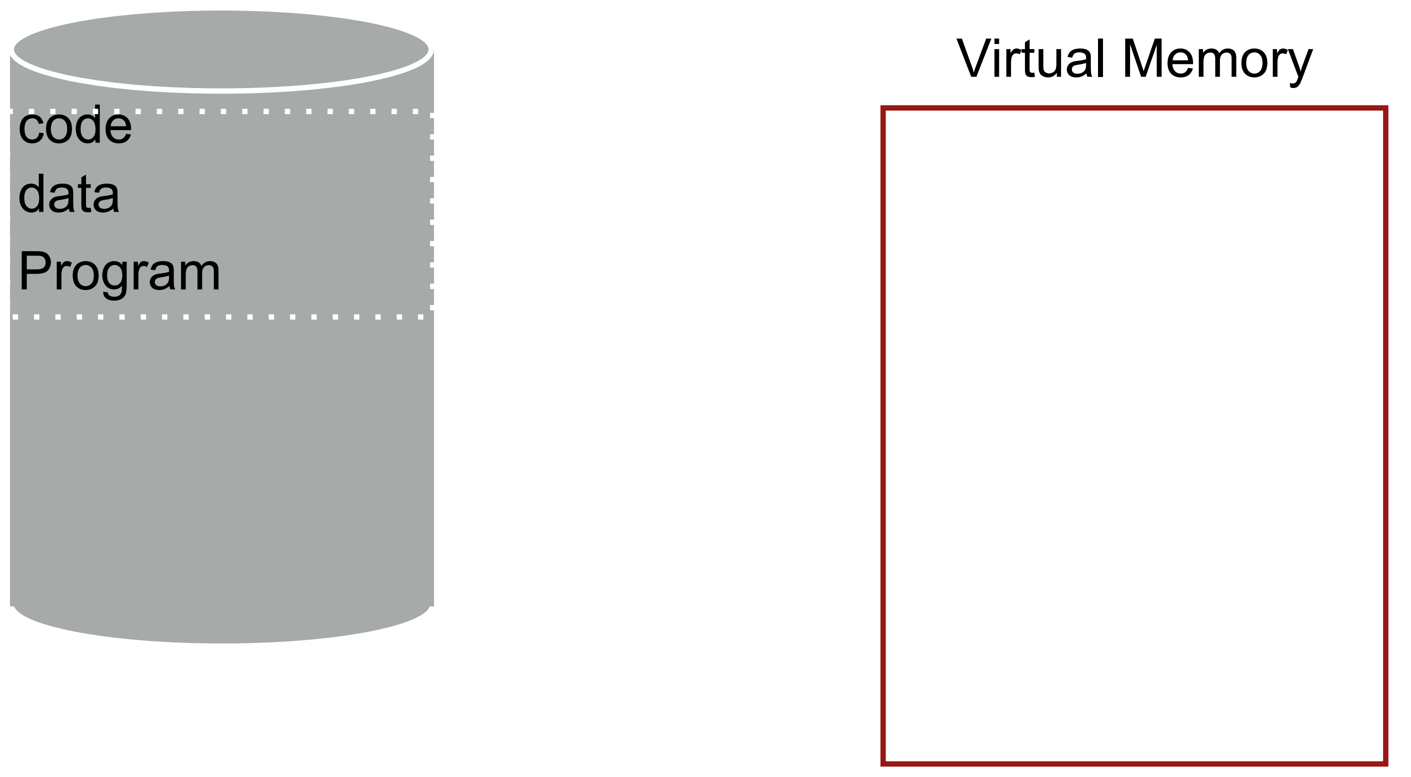
User code should be independent of amount of physical memory

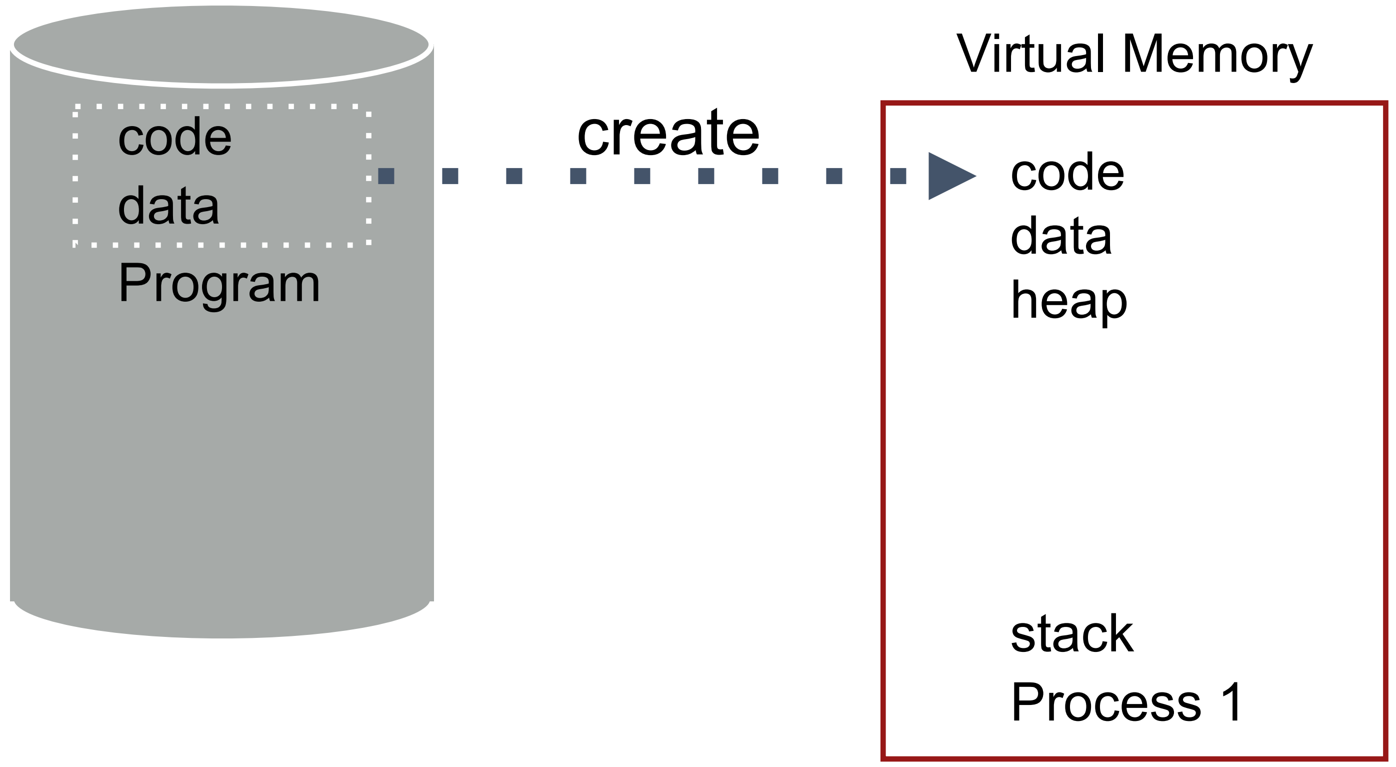
- Correctness, if not performance

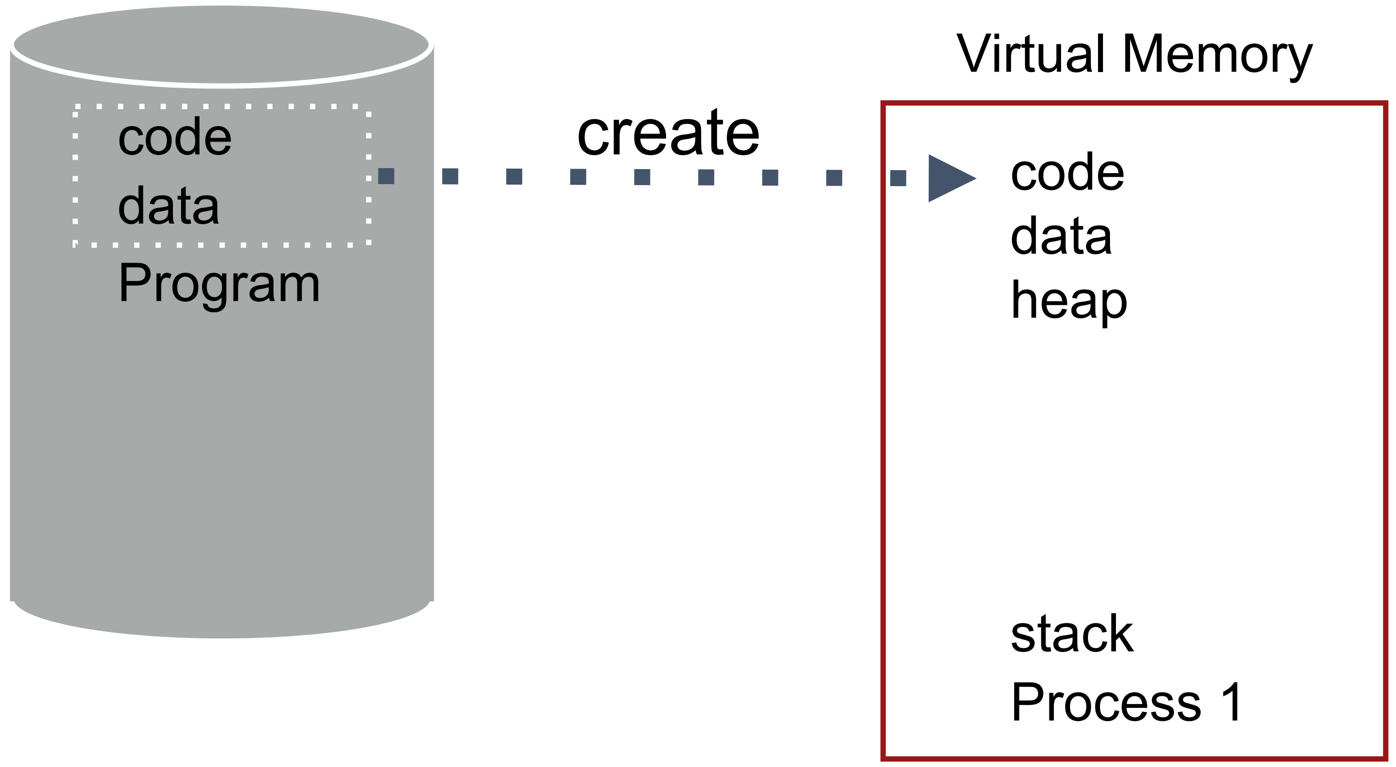
Virtual memory: OS provides illusion of more physical memory

How could we make such an illusion work?

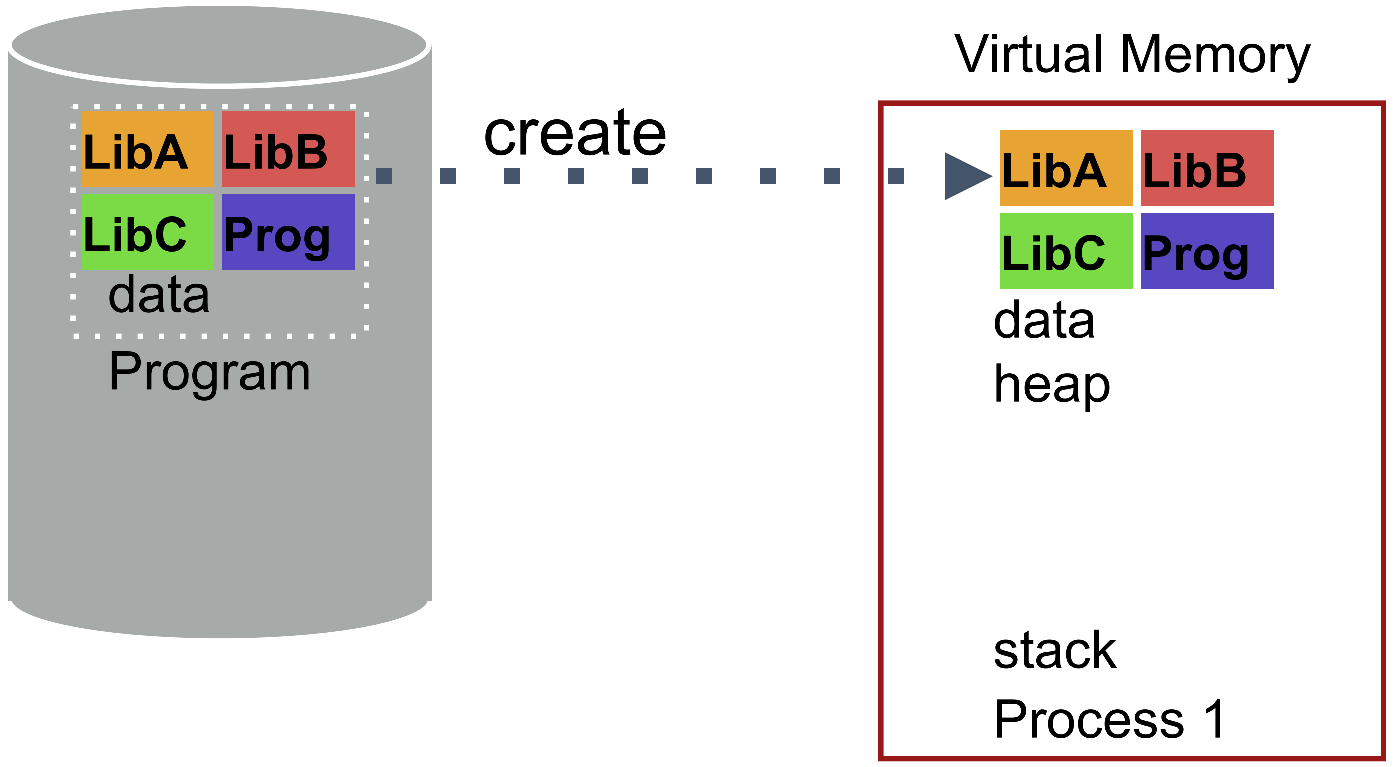
- We rely on key properties of user processes (workload) and machine architecture (hardware)


code
data
Program
Virtual Memory


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

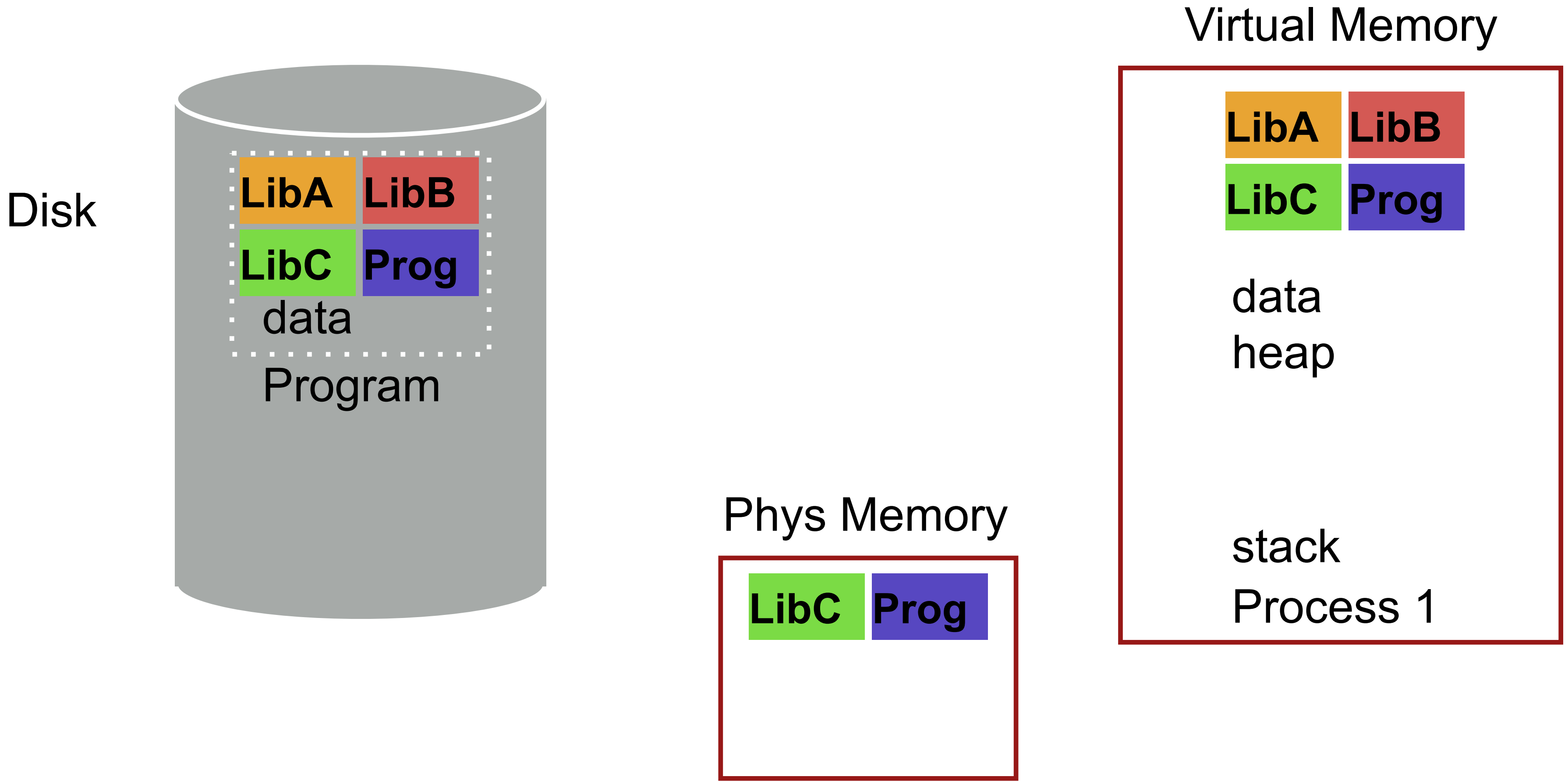


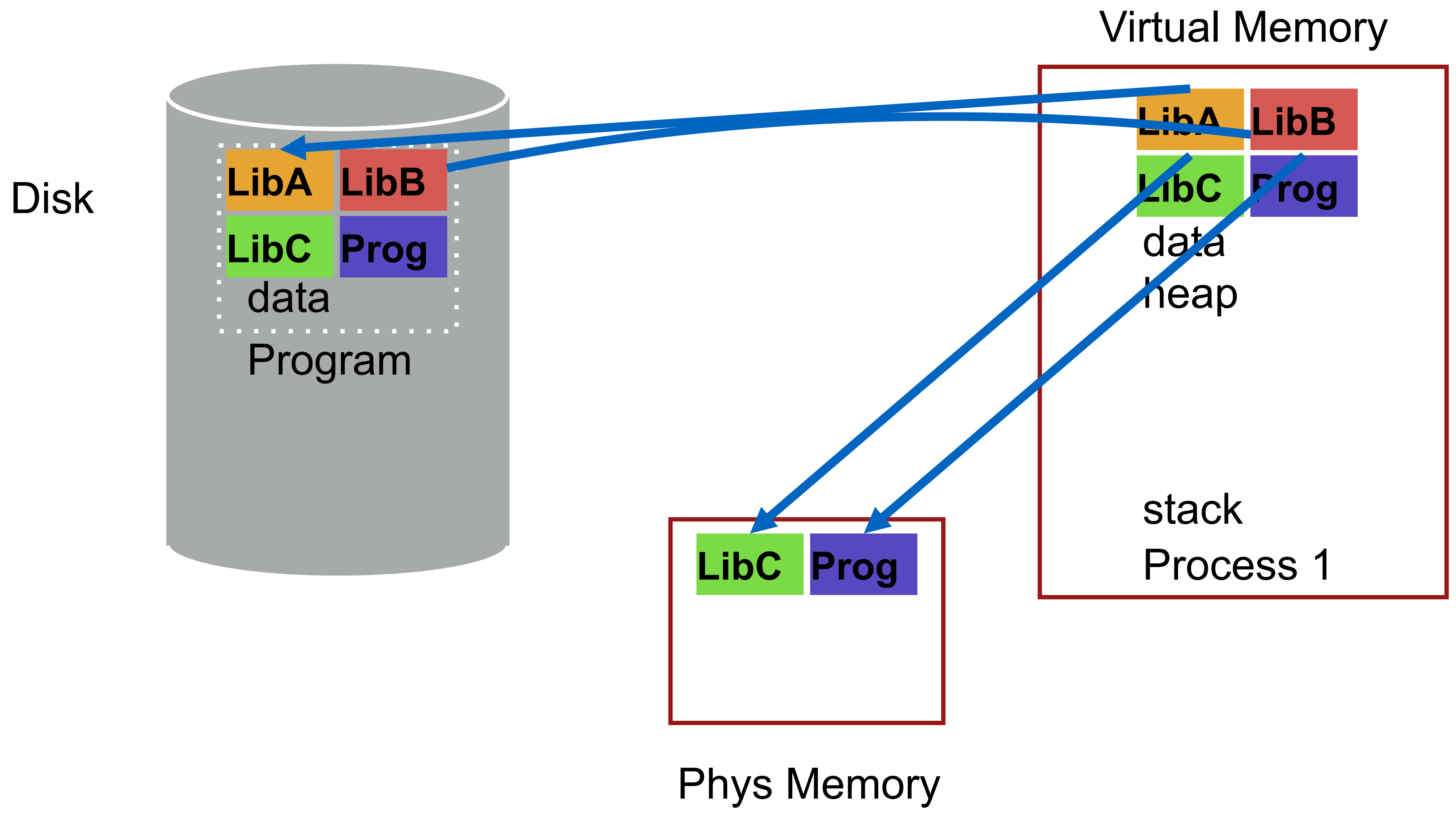
what’s in code?

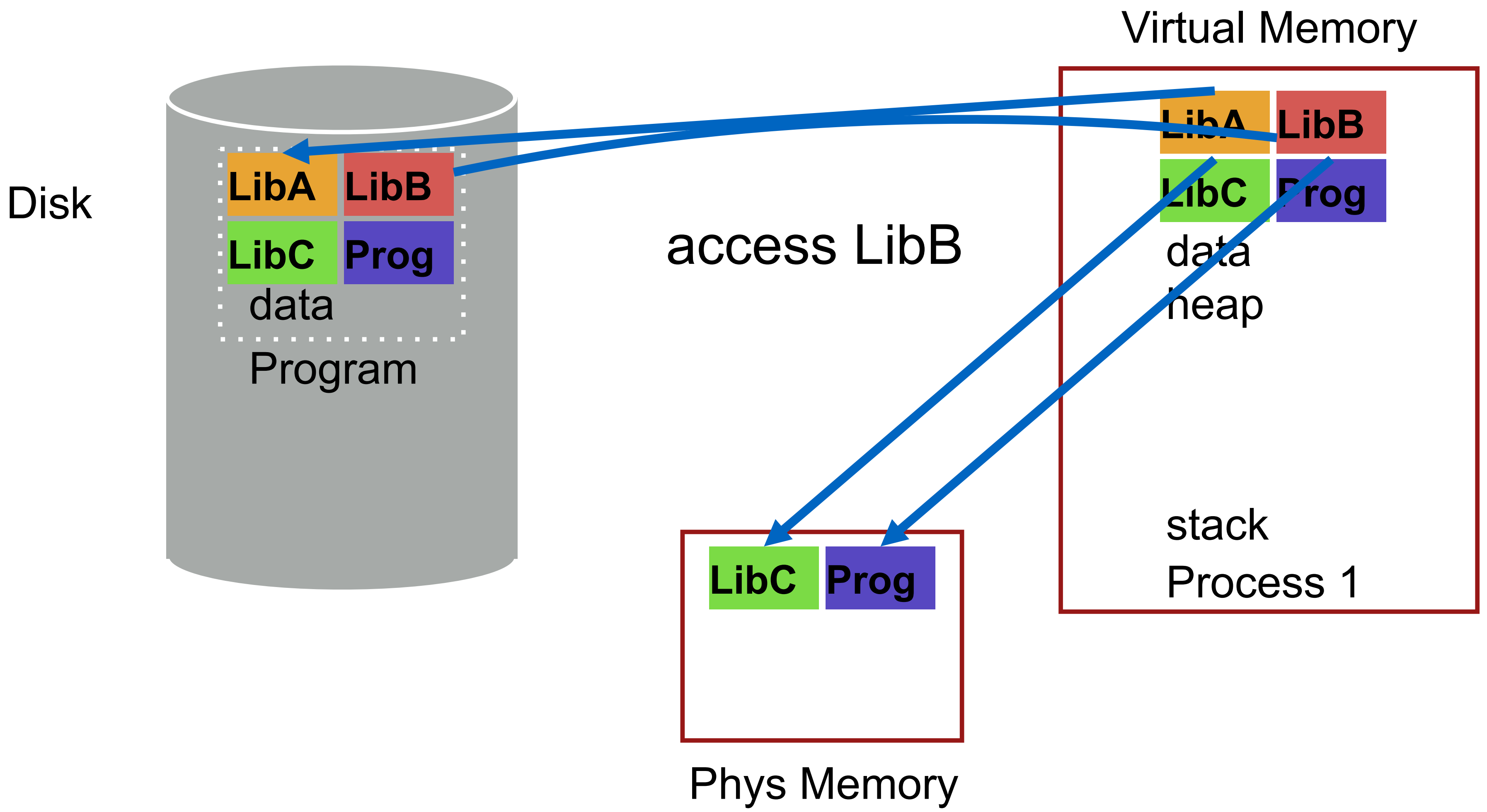


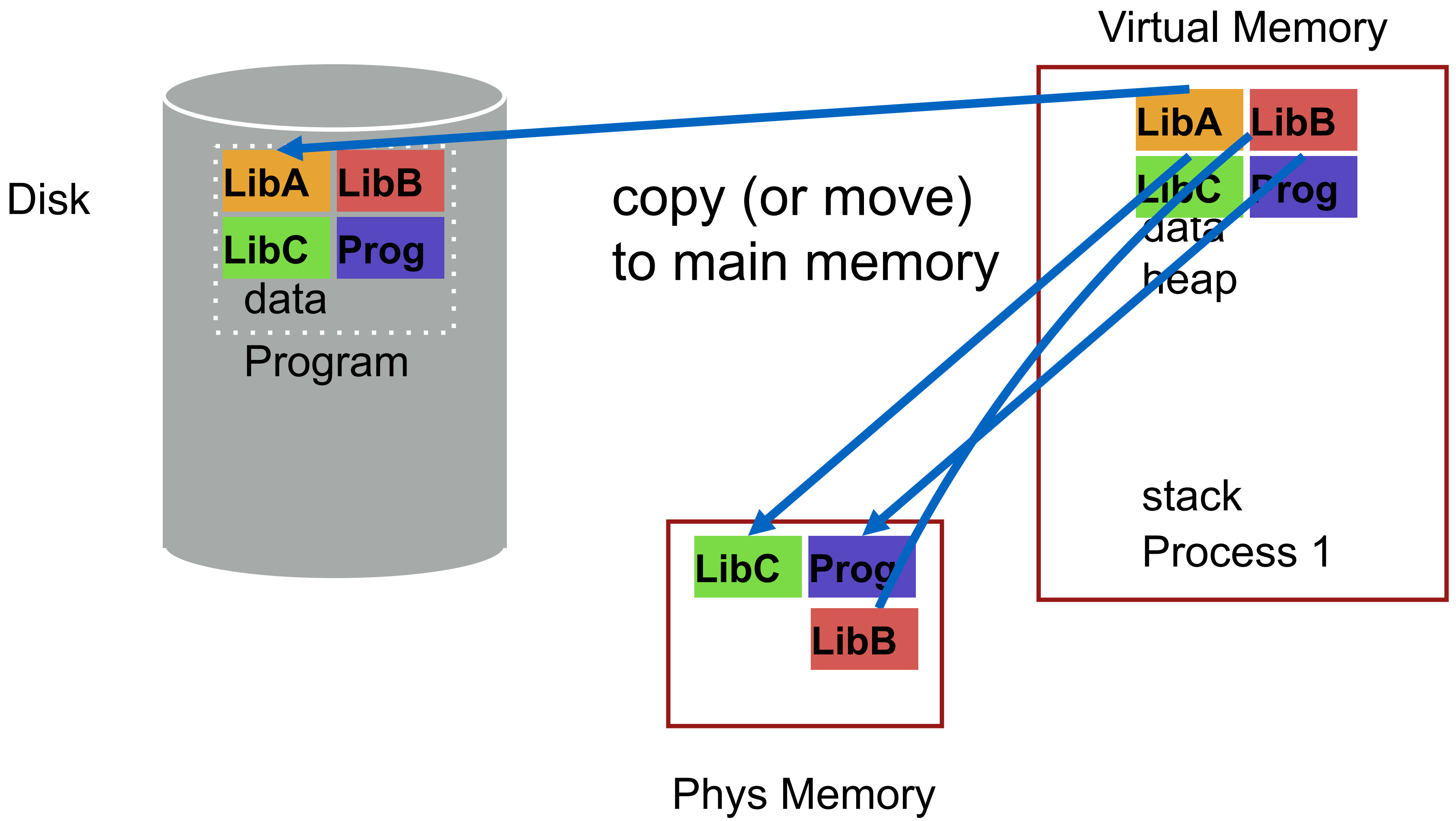
many large libraries, some of which are rarely/never used

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

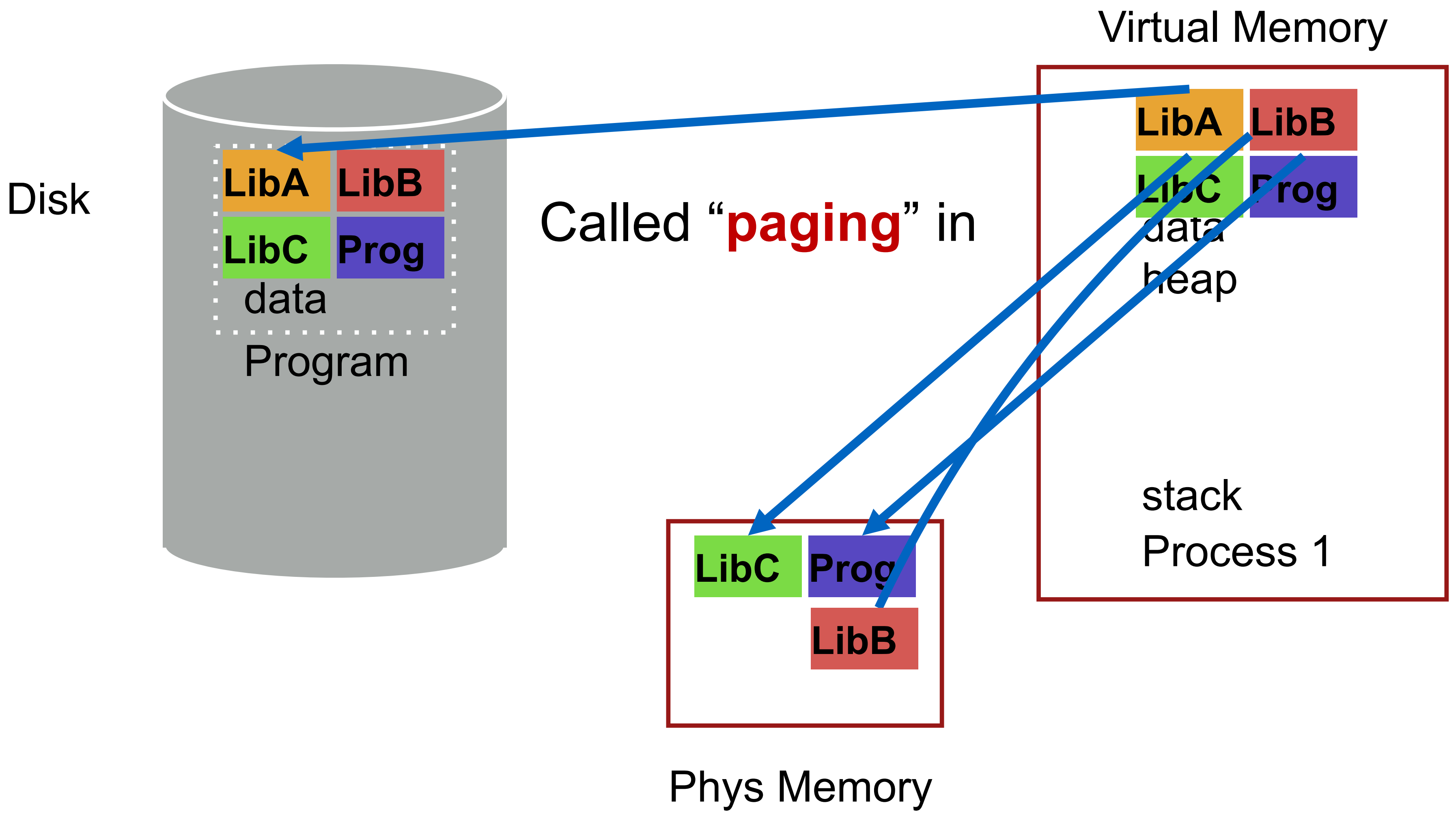

Disk
LibA
LibB
LibC
Prog
data
Program
Phys Memory
LibC
Prog
Virtual Memory
LibA
LibB
LibC
Prog
data
heap
stack
Process 1


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


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


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

Called “**paging**” in



Locality of Reference

Effectively: Using main memory as a cache of process virtual memory contents located on disk

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
  - For example: 90% of time in 10% of code

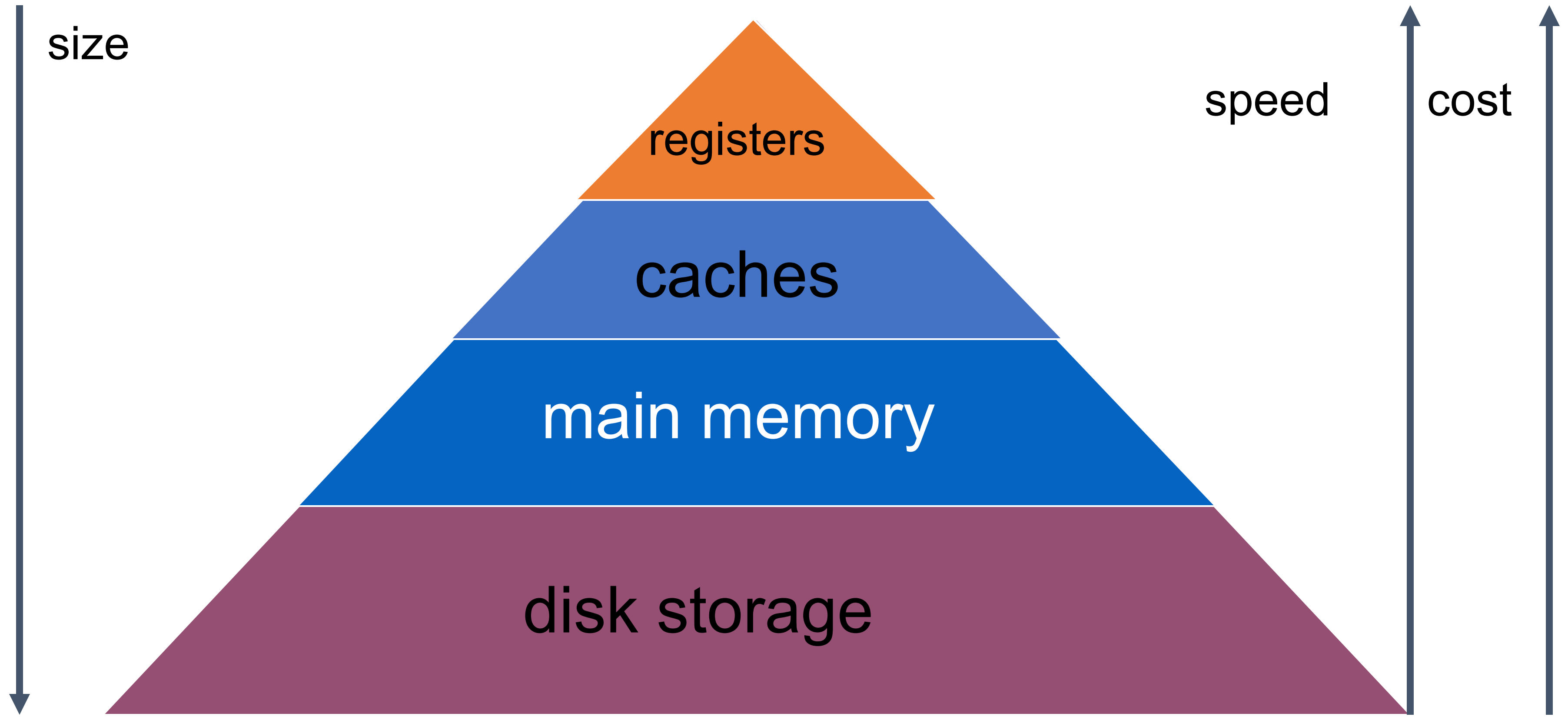
Implication:

- 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



Virtual Memory Intuition

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

Requirements:

- OS must have **mechanism** to identify location of each page in address space either in memory or on disk
- OS must have **policy** for determining which pages live in memory and which on disk

Virtual Address Space Mechanisms

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

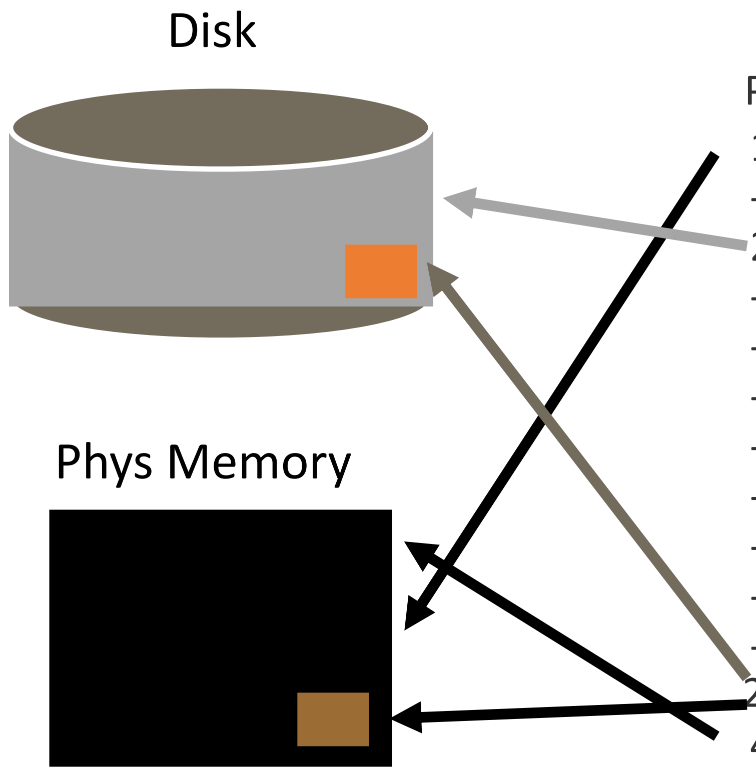
- Physical main memory: Small, fast, expensive
- Disk (backing store): Large, slow, cheap
- Nothing (error): Free

Extend page tables with an extra bit: `present`

- `permissions (r/w), valid, present`
- Page in memory: `present` bit set in PTE
- Page on disk: `present` bit cleared

- PTE with cleared present bit 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	-	-	
-	0	-	-	
28 **16**	1 **1**	rw- **rw-**	0 **1**	// VPN 0xb
4	1	rw-	1	

What if access vpn 0xb?

Virtual Memory Mechanisms

Hardware and OS cooperate to translate addresses

First, hardware checks TLB for virtual address

- if TLB hit, address translation is done; page in physical memory

If TLB miss...

- Hardware or OS walk page tables
- If PTE designates page is present, then page in physical memory

Hardware memory access: Control flow

```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
if (Success == True)   // TLB Hit
    if (CanAccess(TlbEntry.ProtectBits) == True)
        Offset   = VirtualAddress & OFFSET_MASK
        PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
        Register = AccessMemory(PhysAddr)
    else
        RaiseException(PROTECTION_FAULT)
else                  // TLB Miss
    PTEAddr = PTBR + (VPN * sizeof(PTE))
    PTE = AccessMemory(PTEAddr)
    if (PTE.Valid == False)
        RaiseException(SEGMENTATION_FAULT)
    else
        if (CanAccess(PTE.ProtectBits) == False)
            RaiseException(PROTECTION_FAULT)
```

Hardware memory access: Control flow

```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
if (Success == True)   // TLB Hit
    if (CanAccess(TlbEntry.ProtectBits) == True)
        Offset   = VirtualAddress & OFFSET_MASK
        PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
        Register = AccessMemory(PhysAddr)
    else
        RaiseException(PROTECTION_FAULT)
else                  // TLB Miss
    PTEAddr = PTBR + (VPN * sizeof(PTE))
    PTE = AccessMemory(PTEAddr)
    if (PTE.Valid == False)
        RaiseException(SEGMENTATION_FAULT)
    else
        if (CanAccess(PTE.ProtectBits) == False)
            RaiseException(PROTECTION_FAULT)
        else if (PTE.Present == True)
            // assuming hardware-managed TLB
            TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
            RetryInstruction()
```

Hardware memory access: Control flow

```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
if (Success == True)   // TLB Hit
    if (CanAccess(TlbEntry.ProtectBits) == True)
        Offset   = VirtualAddress & OFFSET_MASK
        PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
        Register = AccessMemory(PhysAddr)
    else
        RaiseException(PROTECTION_FAULT)
else                  // TLB Miss
    PTEAddr = PTBR + (VPN * sizeof(PTE))
    PTE = AccessMemory(PTEAddr)
    if (PTE.Valid == False)
        RaiseException(SEGMENTATION_FAULT)
    else
        if (CanAccess(PTE.ProtectBits) == False)
            RaiseException(PROTECTION_FAULT)
        else if (PTE.Present == True)
            // assuming hardware-managed TLB
            TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
            RetryInstruction()
        else if (PTE.Present == False)
            RaiseException(PAGE_FAULT)
```

Virtual Memory Mechanisms

If page fault (i.e., `present` bit is cleared)

- Trap into OS (not handled by hardware. Why?)
- OS selects victim page in memory to replace
- Write victim page out to disk if modified. Add `modified` `("dirty")` bit to PTE
- OS reads referenced page from disk into memory
- Page table is updated, `present` bit is set
- Process continues execution

What should scheduler do?

Mechanism for Continuing a Process

Continuing a process after a page fault is tricky

- Want page fault to be transparent to user
- Page fault may have occurred in middle of instruction
  - When instruction is being fetched
  - When data is being loaded or stored
- Requires hardware support
  - precise interrupts: stop CPU pipeline such that instructions before faulting instruction have completed, and those after can be restarted

Complexity depends upon instruction set

- Can faulting instruction be restarted from beginning?
  - Example: `move +(SP), R2`
  - Must track side effects so hardware can roll them back if needed

Virtual Memory Policies

Goal: Minimize number of page faults

- Page faults require milliseconds to handle (reading from disk)
- Implication: Plenty of time for OS to make good decision

OS has two decisions

- Page selection
  - **When** should a page (or pages) on disk be **brought into** memory?
- Page replacement
  - **Which** resident page (or pages) in memory should be **thrown out** to disk?

Average Memory Access Time (AMAT)

Hit% = portion of accesses that go straight to RAM

Miss% = portion of accesses that go to disk first

Tm = time for memory access

Td = time for disk access

AMAT = (Tm) + (Miss% * Td)

Page Selection

When should a page be brought from disk into memory?

Demand paging: Load page only when page fault occurs

- Intuition: Wait until page must absolutely be in memory
- When process starts: No pages are loaded in memory
- Problems: Pay the cost of a page fault for every newly accessed page

Page Selection

When should a page be brought from disk into memory?

Pre-paging (anticipatory, prefetching): Load page before referenced

- OS predicts future accesses (oracle) and brings pages into memory early
- Works well for some access patterns (e.g., sequential)
- Problems?

Page Selection

When should a page be brought from disk into memory?

Hints: Combine above with user-supplied hints about page references

- User specifies: may need page in future, don’t need this page anymore, or sequential access pattern, ...
- Example: `madvise()` in Unix

Page Replacement

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

OPT: Replace page not used for longest time in future

- Advantages: Guaranteed to minimize number of page faults
- Disadvantages: Requires that OS predict the future; Not practical, but good for comparison

OPT Replacement Example

Page reference string: 1,2,3,1,2,4,1,4,2,3, 2

Three pages
of physical memory

OPT

Miss: 1,2,3

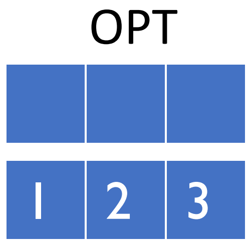
Metric:
Miss
count

OPT Replacement Example

Page reference string: **1,2,3**,1,2,4,1,4,2,3, 2

Three pages of physical memory

Miss: 1,2,3

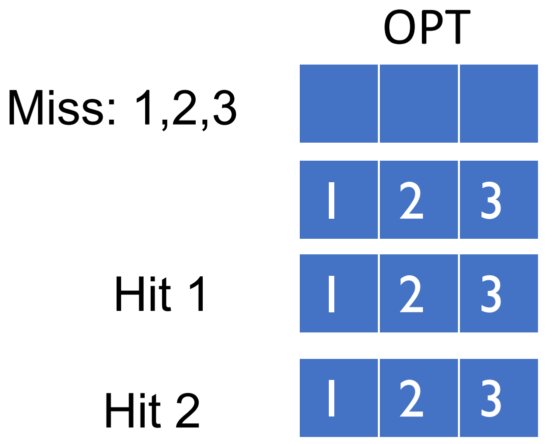


Metric:
Miss count : 3

OPT Replacement Example

Page reference string: 1,2,3,**1,2**,4,1,4,2,3, 2

Three pages
of physical memory



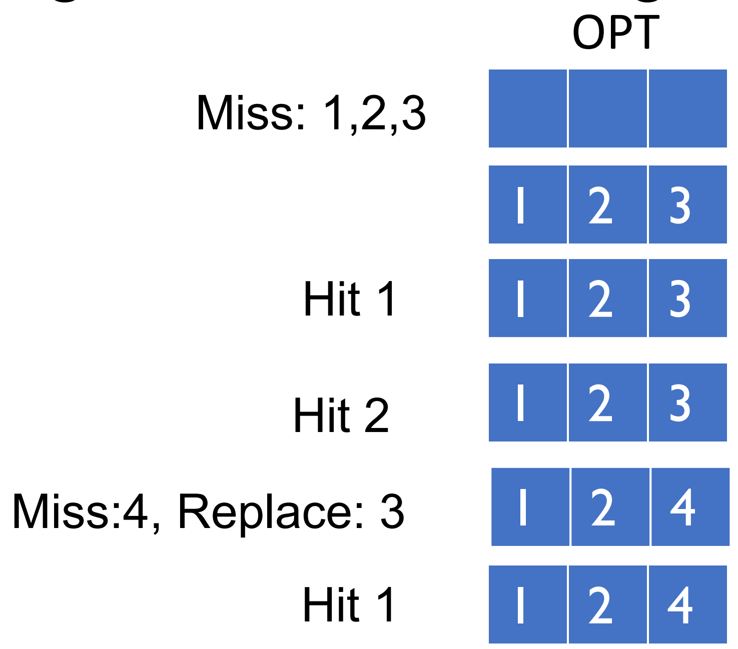
Metric:
Miss count : 3

Compulsory
misses

OPT Replacement Example

Page reference string: 1,2,3,1,2,**4,1**,4,2,3, 2

Three pages
of physical memory



Metric:
Miss count: 4

capacity
miss

OPT Replacement Example

Page reference string: 1,2,3,1,2,4,**1**,**4,2,**3, 2

Three pages
of physical memory

OPT

Miss: 1,2,3

Hit 1

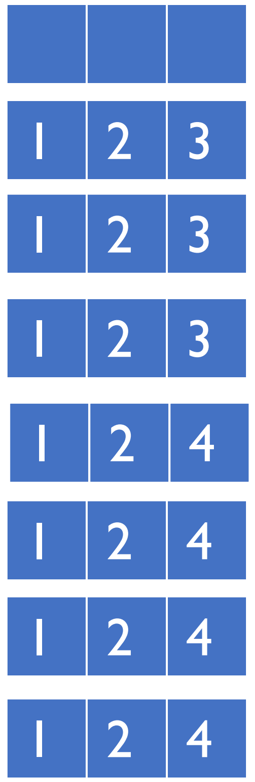
Hit 2

Miss:4, Replace: 3

Hit 1

Hit: 4

Hit: 2



Metric:
Miss count: 4

OPT Replacement Example

Page reference string: 1,2,3,1,2,4,**1**,4,2,**3, 2**

Three pages
of physical memory

	OPT		
Miss: 1,2,3			
	1	2	3
Hit 1	1	2	3
Hit 2	1	2	3
Miss:4, Replace: 3	1	2	4
Hit 1	1	2	4
Hit: 4	1	2	4
Hit: 2	1	2	4
Miss:3, Replace: 1	2	3	4
Hit: 2	2	3	4

Metric:
AMAT?
Miss count : 5

5 misses, 4 compulsory misses

AMAT = (Tm) + (Miss% * Td)

Assume Tm = 100ns
Assume Td = 1000000 ns (1millisec)

AMAT = ?

FIFO

FIFO: Replace page that has been in memory the longest

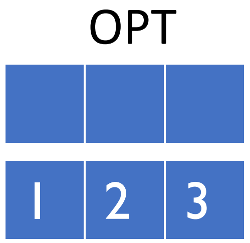
- Intuition: First referenced long time ago, done with it now
- Advantages: Fair: All pages receive equal residency; Easy to implement (circular buffer)
- Disadvantage: Some pages may always be needed

FIFO Example

Page reference string: **1,2,3**,1,2,4,1,4,2,3,2

Three pages
of physical memory

Miss: 1,2,3



Metric:
Miss count: 3

FIFO Example

Page reference string: 1,2,3,**1,2,4**,1,4,2,3,2

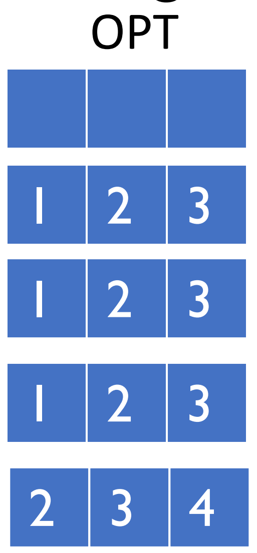
Three pages
of physical memory

Miss: 1,2,3

Hit: 1

Hit: 2

Miss:4, Replace:1



Metric:
Miss count: 4

FIFO Example

Page reference string: 1,2,3,1,2,4,**1,4**,2,3,2

Three pages
of physical memory

		OPT	
Miss: 1,2,3			
	1	2	3
Hit: 1	1	2	3
Hit: 2	1	2	3
Miss:4, Replace:1	2	3	4
Miss:1, Replace:2	3	4	1
Hit: 4	3	4	1

Metric:
Miss count: 5

FIFO Example

Page reference string: 1,2,3,1,2,4,1,4,**2,3**,2

Three pages
of physical memory

	OPT		
Miss: 1,2,3			
	1	2	3
Hit: 1	1	2	3
Hit: 2	1	2	3
Miss:4, Replace:1	2	3	4
Miss:1, Replace:2	3	4	1
Hit: 4	3	4	1
Miss:2, Replace:3	4	1	2
Miss:3, Replace:4	1	2	3

Metric:
Miss count : 7

FIFO Example

Page reference string: 1,2,3,1,2,4,1,4,2,3,**2**

Three pages
of physical memory

OPT

Miss: 1,2,3			
	1	2	3
Hit: 1	1	2	3
Hit: 2	1	2	3
Miss:4, Replace:1	2	3	4
Miss:1, Replace:2	3	4	1
Hit: 4	3	4	1
Miss:2, Replace:3	4	1	2
Miss:3, Replace:4	1	2	3
Hit: 2	1	2	3

Metric:
Miss count : 7

FIFO Example

Page reference string: 1,2,3,1,2,4,**1**,4,2,3,2

Three pages
of physical memory

OPT

Miss: 1,2,3			
	1	2	3
Hit: 1	1	2	3
Hit: 2	1	2	3
Miss:4, Replace:1	2	3	4
Miss:1, Replace:2	3	4	1
Hit: 4	3	4	1
Miss:2, Replace:3	4	1	2
Miss:3, Replace:4	1	2	3
Hit: 2	1	2	3

7 total misses, 4 compulsory misses

AMAT = (Tm) + (Miss% * Td)

Assume Tm = 100ns
Assume Td = 1000000 ns (1millisec)

AMAT = ?

LRU Example – Replace Least Recently Used

Page reference string: 1,2,3,1,2,4,1,4,2,3,2

Three pages
of physical memory

OPT

Miss: 1,2,3			
	1	2	3
Hit: 1	1	2	3
Hit: 2	1	2	3
Miss:4, Replace:3	1	2	4
Hit: 1	1	2	4
Hit: 4	1	2	4
Hit: 2	1	2	4
Miss:3, Replace:1	2	4	3
Hit: 2	2	4	3

Metric: Miss count	5 total misses 4 compulsory misses
	In this example, same as OPT!

Page Replacement Comparison

Add more physical memory, what happens to performance?

- LRU, OPT: Add more memory, guaranteed to have fewer (or same number of) page faults
  - Smaller memory sizes are guaranteed to contain a subset of larger memory sizes
  - Stack property: smaller cache a subset of bigger cache
- FIFO: Add more memory, usually have fewer page faults
  - Belady's anomaly: but there are cases where we have more page faults!

Consider access stream: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Consider physical memory size: 3 pages vs. 4 pages

How many misses with FIFO?

3 pages: 9 misses
4 pages: 10 misses

Problems with LRU-based Replacement

LRU does not consider frequency of accesses

- Is a page accessed **once** in the past equal to one accessed **N** times?
- Common workload problem:
  - Scan (sequential read, never used again) one large data region flushes memory

Solution: Track frequency of accesses to page

Pure LFU (Least-frequently-used) replacement

- Problem: LFU can never forget pages from the far past

Implementing LRU

Perfect LRU on Software

- OS maintains ordered list of physical pages by reference time
- When page is referenced: Move page to front of list
- When need victim: Pick page at back of list
- Trade-off: Slow on memory reference, fast on replacement

Perfect LRU on Hardware

- Associate timestamp with each page (e.g., PTE)
- When page is referenced: Associate current system timestamp with page
- When need victim: Scan through registers to find oldest timestamp
- Trade-off: Fast on memory reference, slow on replacement (especially as size of memory grows)

In practice, do not implement Perfect LRU

- LRU is an approximation anyway, so approximate more
- Goal: Find an old page, but not necessarily the oldest

Clock Algorithm

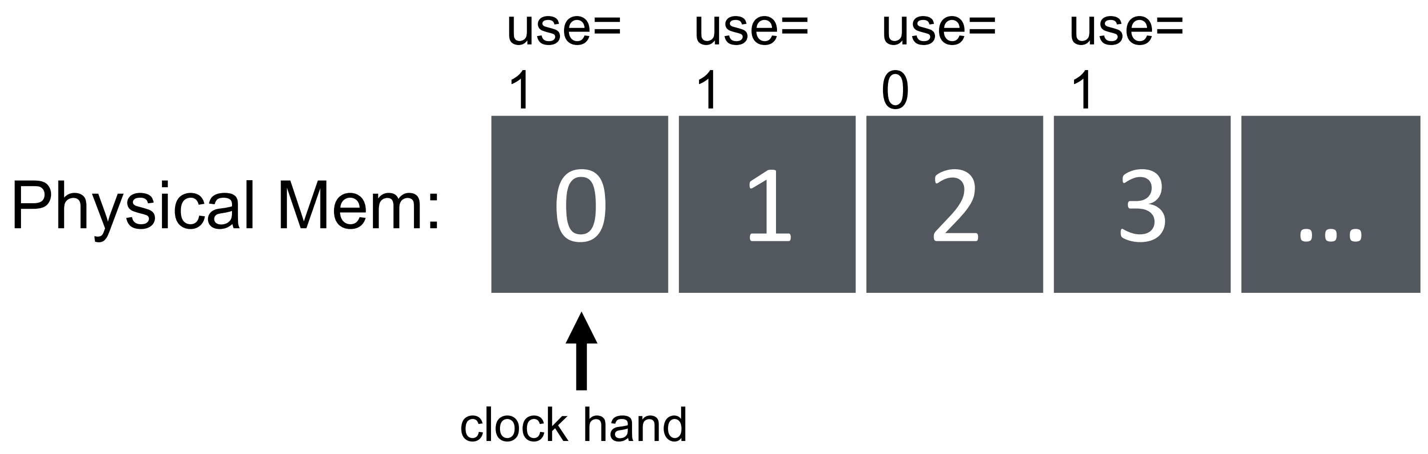
Hardware

- Keep `use` (or `reference`) bit for each page frame
- When page is referenced: set `use` bit

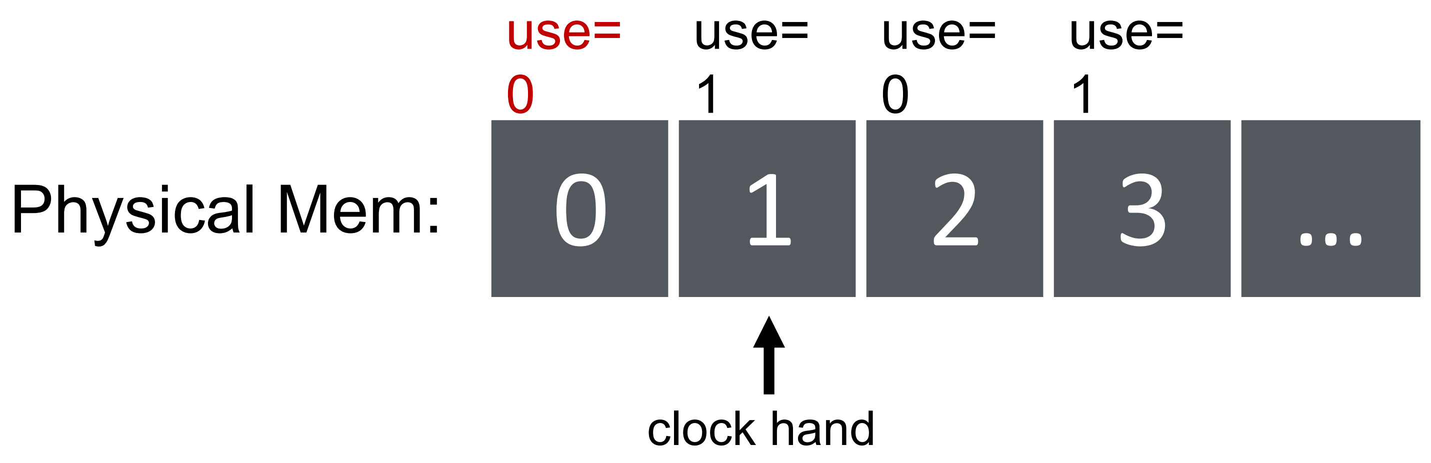
Operating System

- Page replacement: Look for page with `use` bit cleared (has not been referenced for a while)
- Implementation:
  - Keep pointer to last examined page frame (“clock hand”)
  - Traverse pages in circular fashion (like a clock)
  - Clear `use` bits as you search
  - Stop when find page with already cleared `use` bit, replace this page

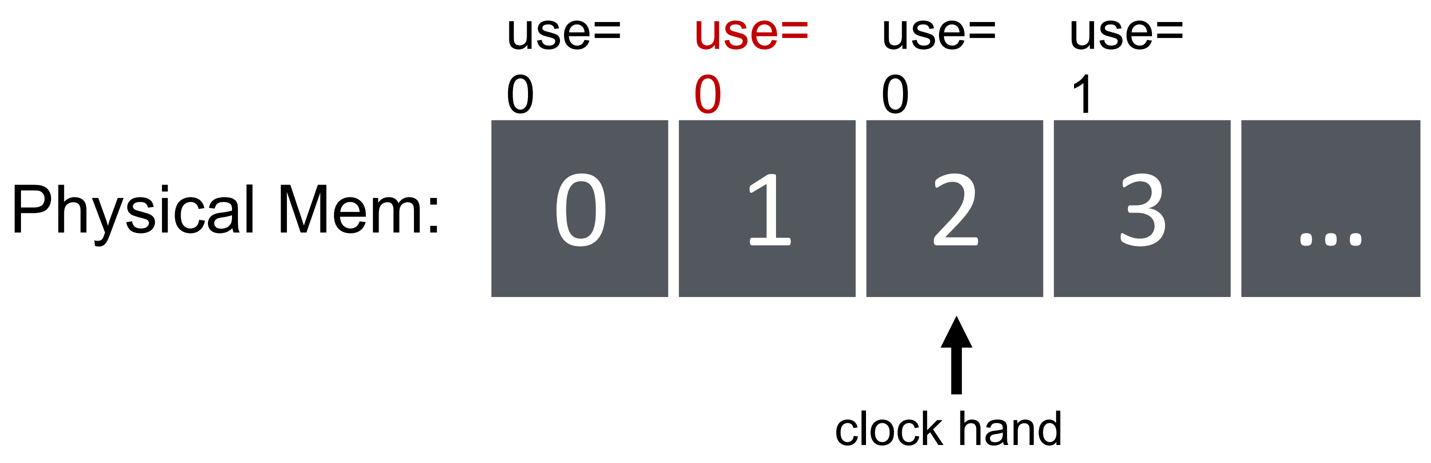
Clock:
Look For a Page



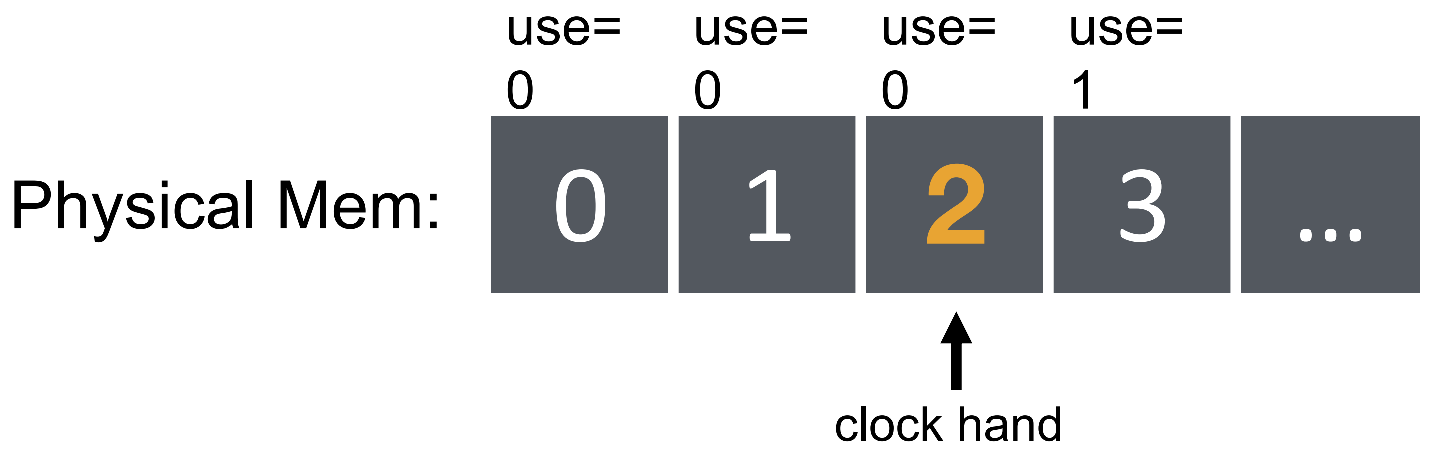
Clock: Look For a Page



Clock:
Look For a Page

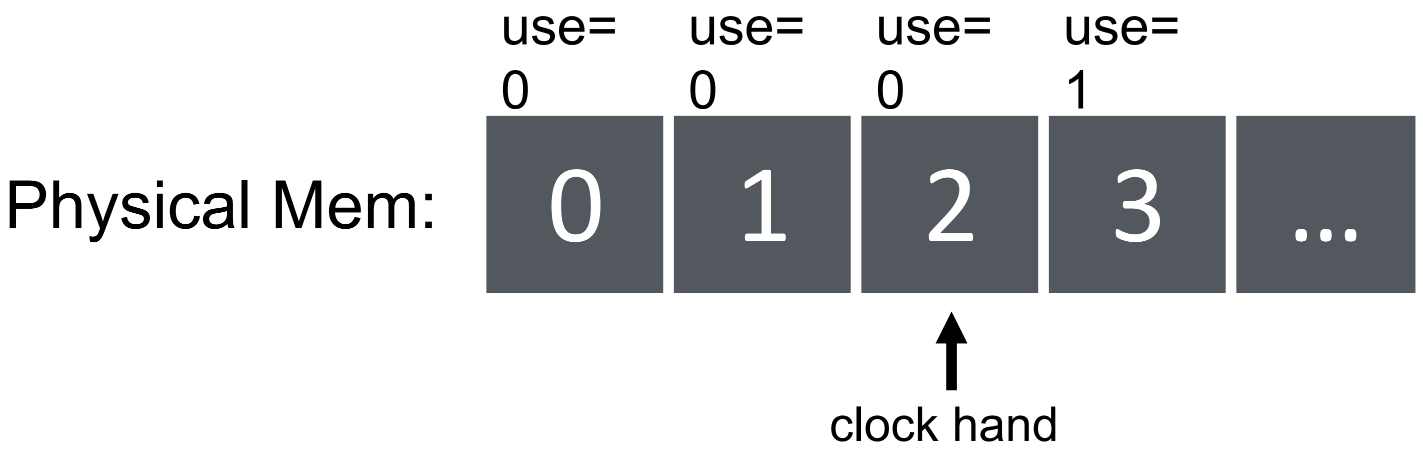


Clock: Look For a Page



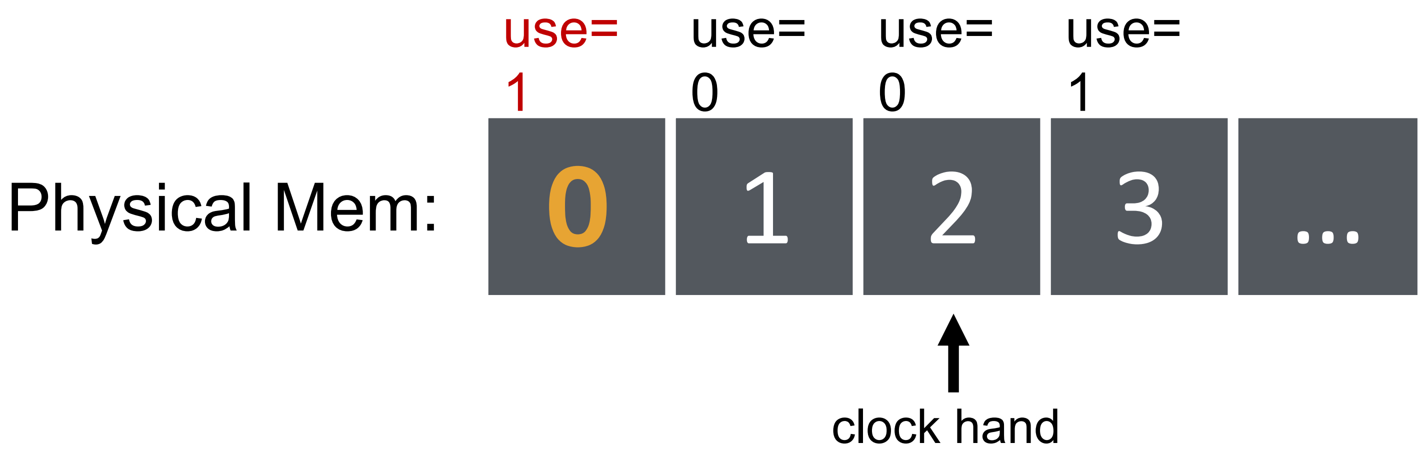
evict **page 2** because it has not been recently used

Clock:
Look For a Page

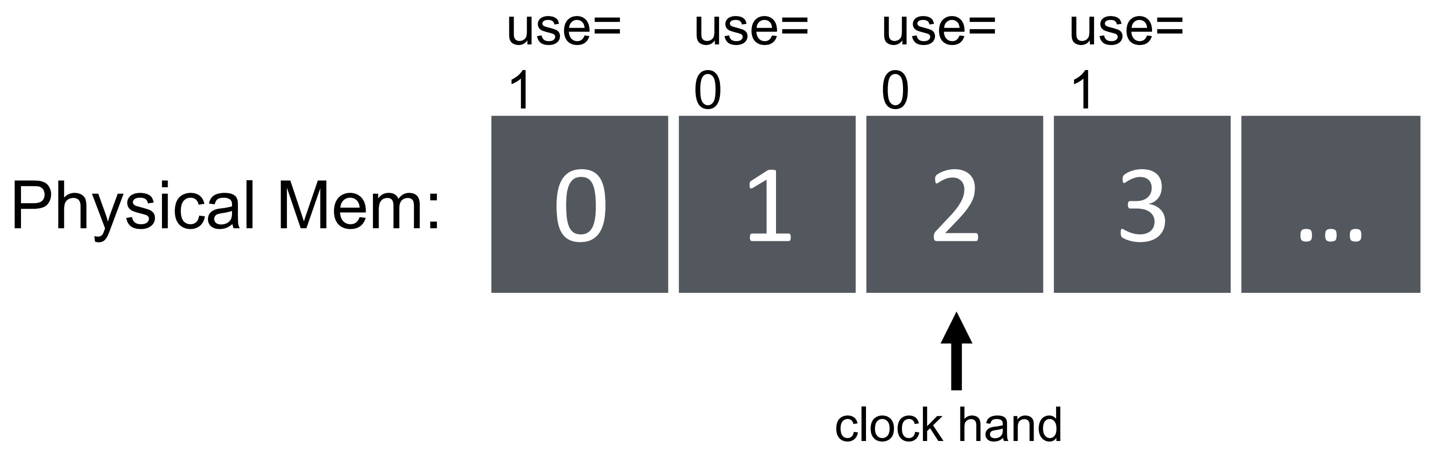


page 0 is accessed…

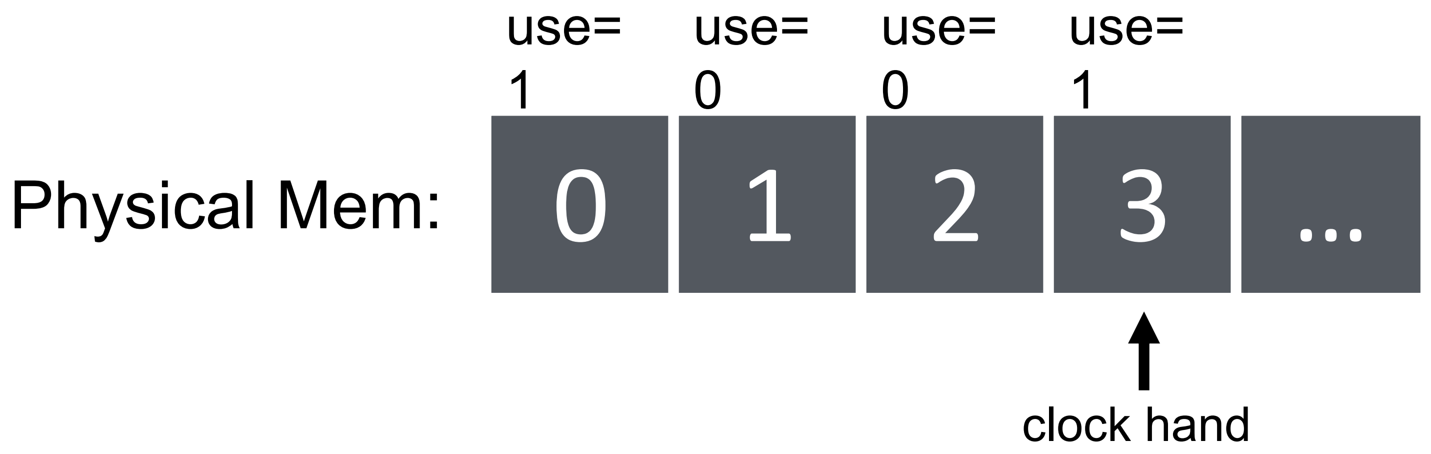
Clock:
Look For a Page



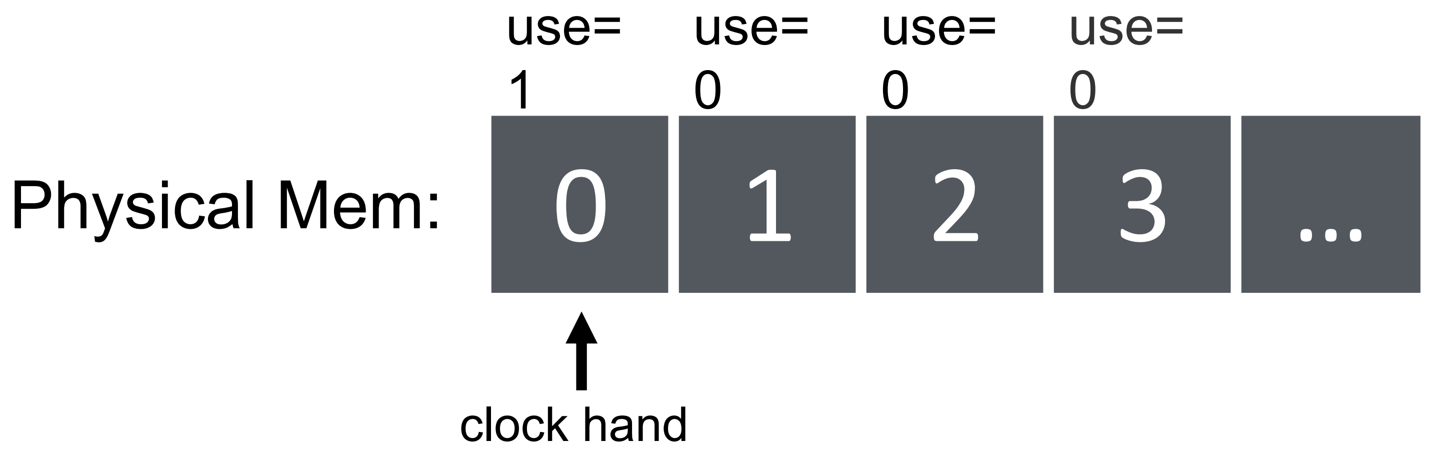
Clock:
Look For a Page



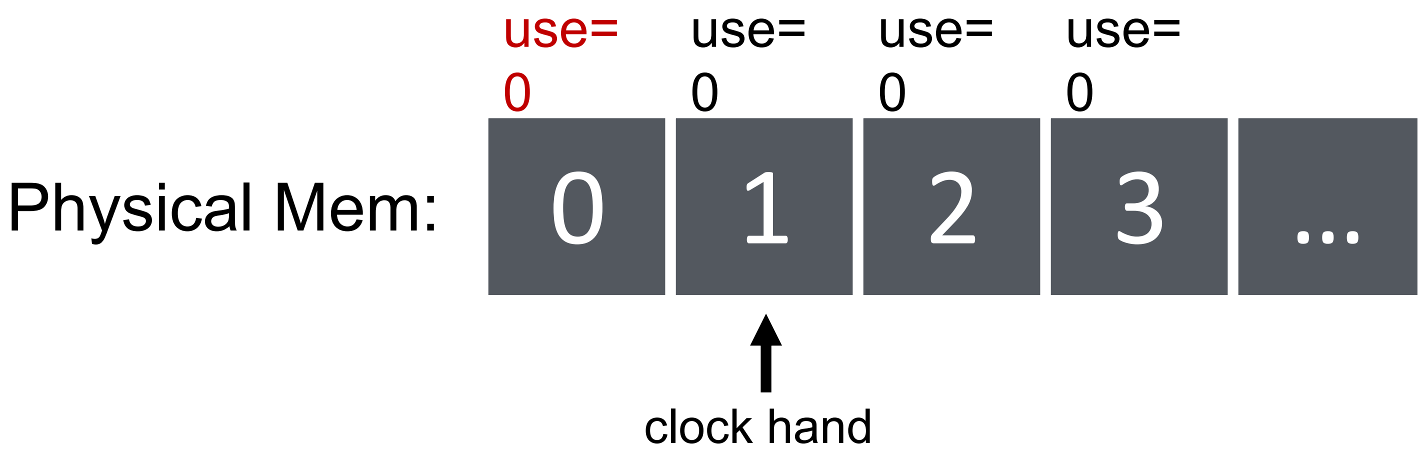
Clock: Look For a Page



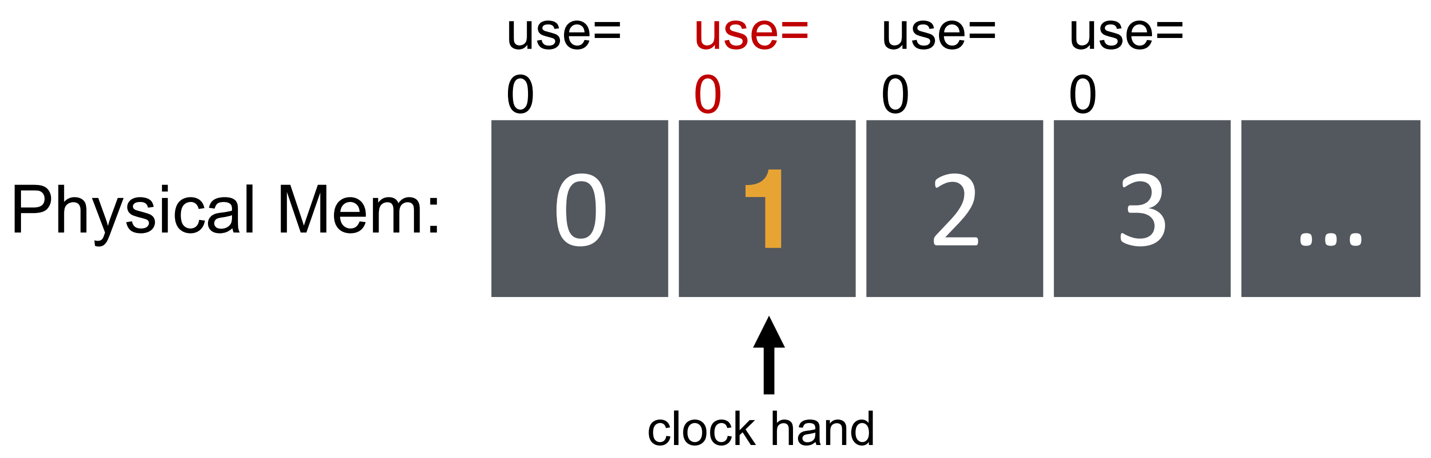
Clock:
Look For a Page



Clock: Look For a Page



Clock:
Look For a Page



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

Clock Extensions

Use modified (“dirty”) bit to prefer to retain modified pages in memory

- Intuition: More expensive to replace dirty pages
  - Modified pages must be written to disk, clean pages do not have to be
- First replace pages that have `use` bit and `modified` bit cleared

Replace multiple pages at once

- Intuition: Expensive to run replacement algorithm and to write single block to disk
- Find multiple victims each time and track free list

Add software counter (“chance”) to track use frequency

- Intuition: Want to differentiate pages by how much they are accessed
- Increment software counter if `use` bit is 0
- Replace when chance exceeds some specified limit

What if no hardware support?

What can the OS do if hardware does not have `use` bit (or `dirty` bit)?

- Can the OS “emulate” these bits?

Think about this question:

- Can the OS get control (i.e., generate a trap) every time `use` bit should be set? (i.e., when a page is accessed?)

Conclusion

Illusion of virtual memory: Processes can run when the sum of virtual address spaces is larger than physical memory

Mechanism:

- Extend page table entry with “present” bit
- OS handles page faults (or page misses) by reading in the desired page from disk

Policy:

- Page selection – demand paging, prefetching, hints
- Page replacement – OPT, FIFO, LRU, others

Implementations (clock) approximate LRU