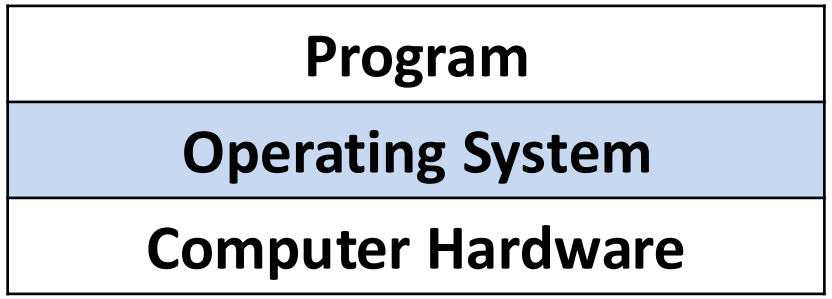
CS 31: Introduction to Computer Systems

24 Virtual Memory

04-22-2025

The Operating System

(1) How a Computer Runs a Program:



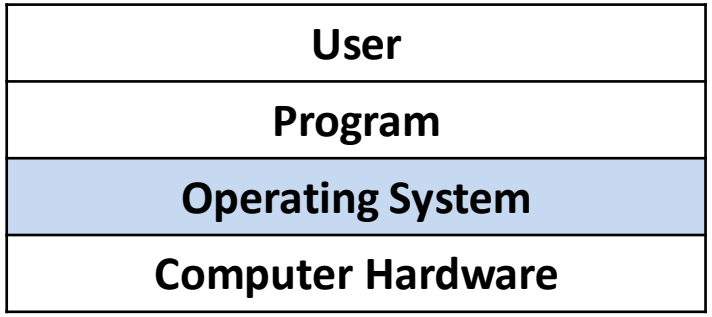
- OS role in running programs on system

(2) How to Efficiently Run Programs

- OS abstractions and HW management for running programs efficiently

Operating System

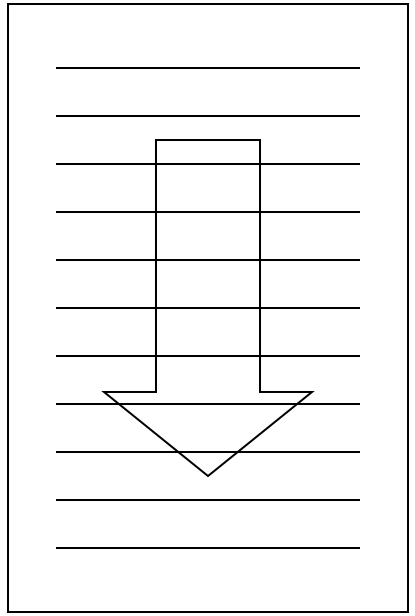
Special SW sits between the HW & User/Program:



1. Manages the underlying HW
   - Coordinates shared access to HW
   - Efficiently schedules/manages HW resources
2. Provides easy-to-use interface to the HW
   - just type: `./a.out` to run a program
   - `fopen`, `fscanf` to interact with stored data (files)

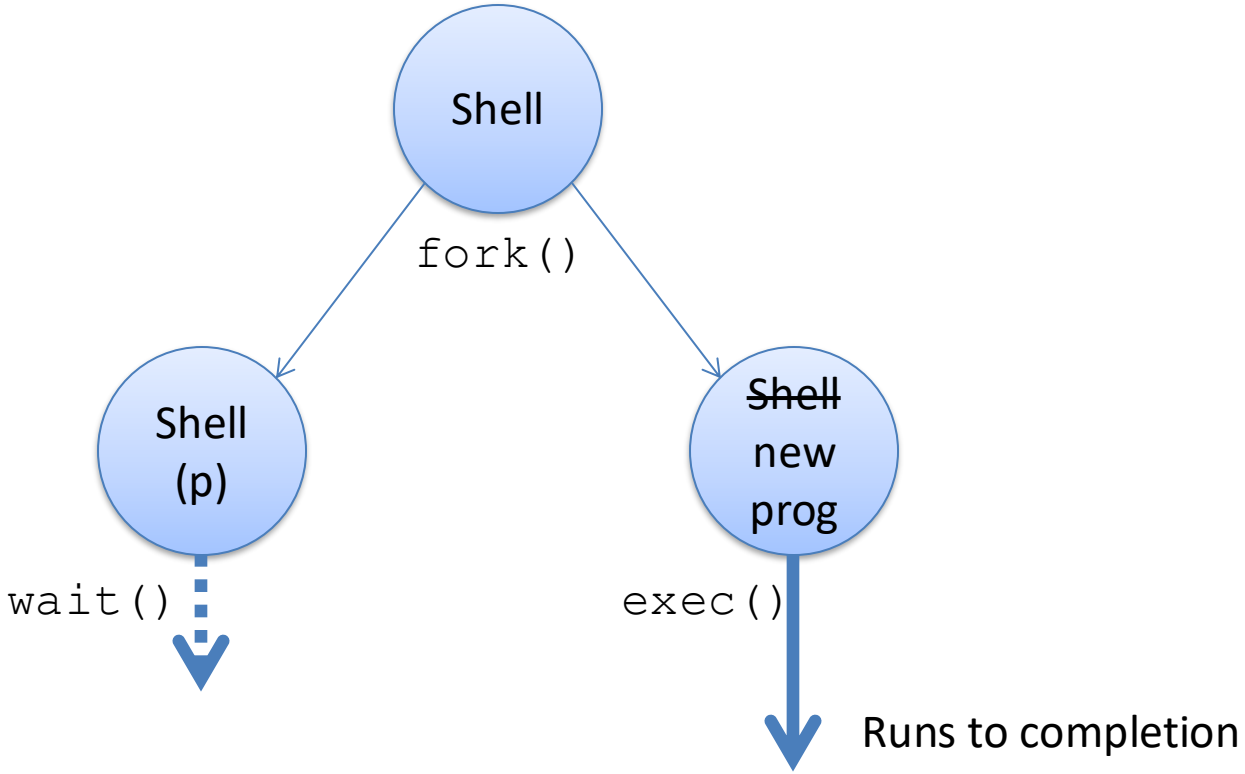
Anatomy of a Process

- Abstraction of a running program
  - a dynamic “program in execution”
- OS keeps track of process state
  - What each process is doing
  - Which one gets to run next
- Basic operations
  - Suspend/resume (context switch)
  - Start (spawn), terminate (kill)



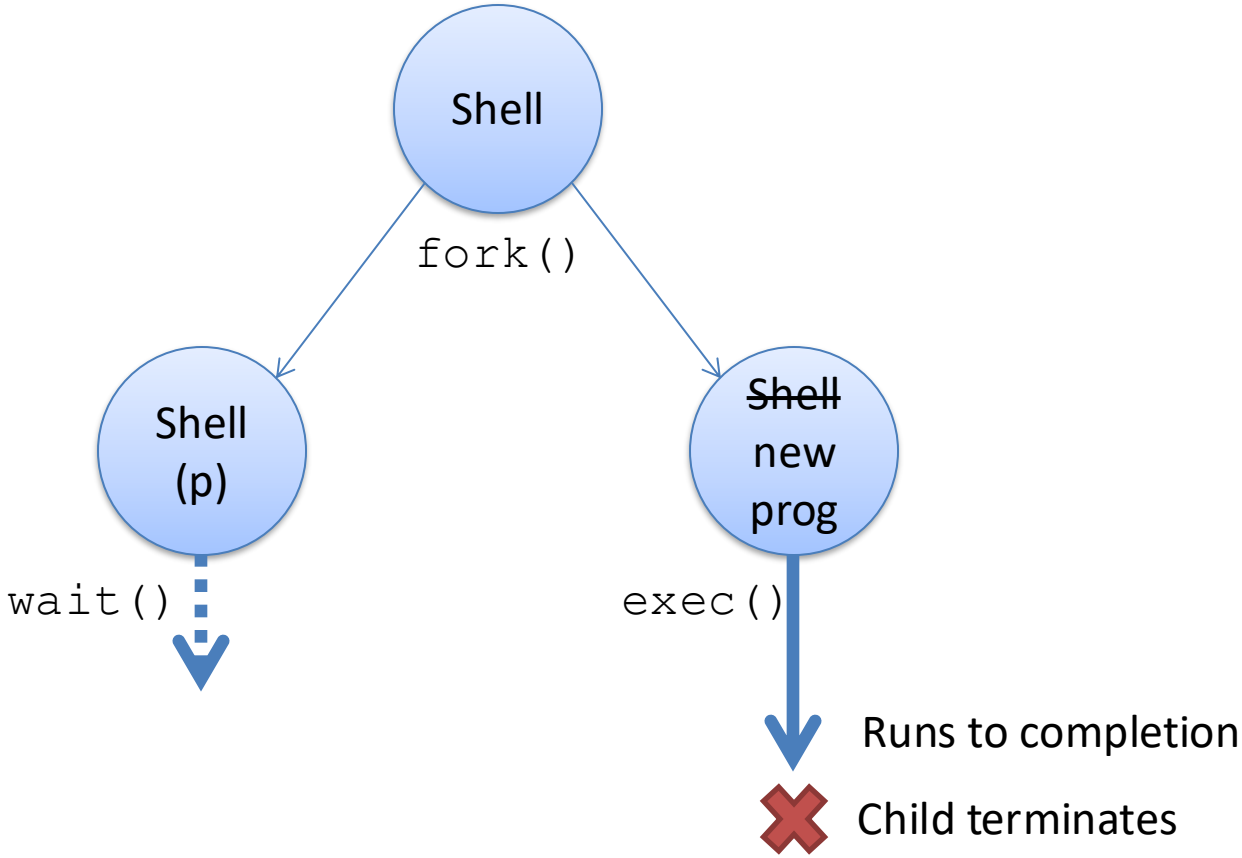
Common `fork()` usage: Shell

2. child: `exec()` user-requested program



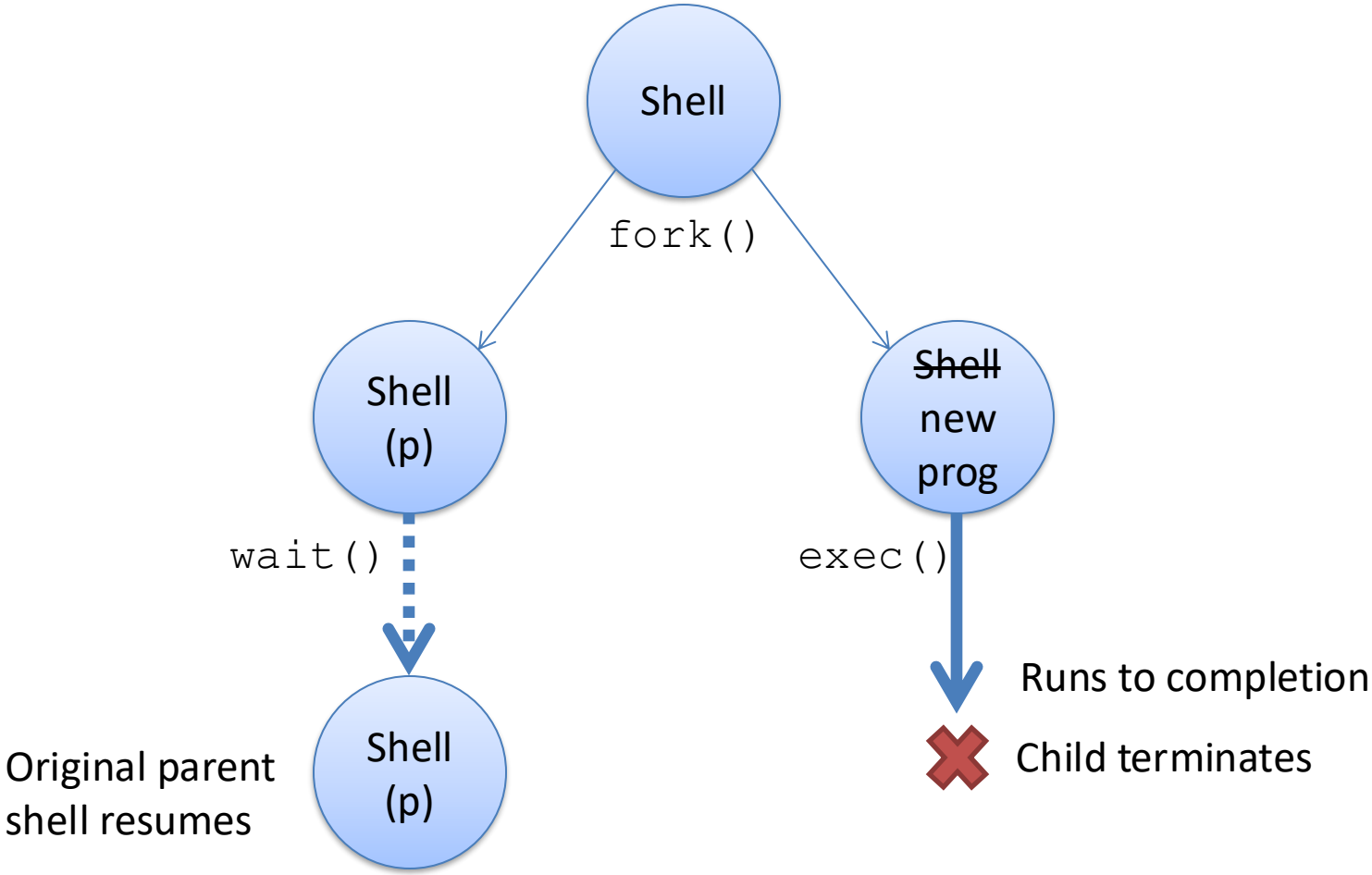
Common `fork()` usage: Shell

3. child program terminates, cycle repeats



Common `fork()` usage: Shell

3. child program terminates, cycle repeats



Process Management: Summary

- A process is the unit of execution.
- Processes are represented as Process Control Blocks in the OS
  - PCBs contain process state, scheduling and memory management information, etc
- A process is either New, Ready, Waiting, Running, or Terminated.
- On a uniprocessor, there is at most one running process at a time.
- The program currently executing on the CPU is changed by performing a context switch
- Processes communicate either with message passing or shared memory

Process: an OS Abstraction

Process: an instance of a running program

Process implements 2 main abstractions:

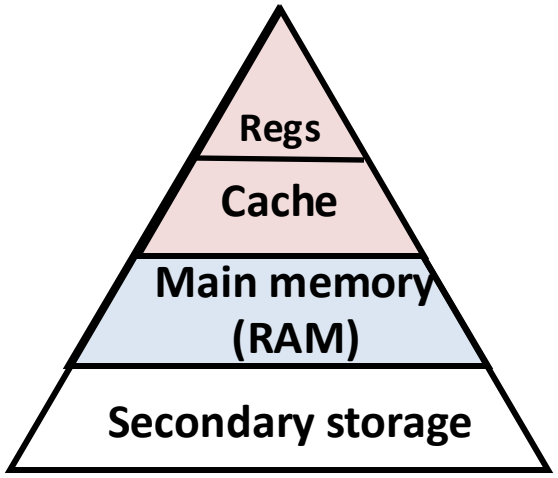
1. Lone View of use of the HW:

   maintains logical control flow of own instructions on CPU with protection from seeing effects of others sharing CPU

2. Private Virtual Address Space

   - Each Process gets its own text, stack, heap, …, Memory space, can't touch each other's space
   - (e.g.) Multiple processes running same a.out:
     - each gets own private variables (say x)
     - P1 executing x=3 doesn't modify P2's x value!

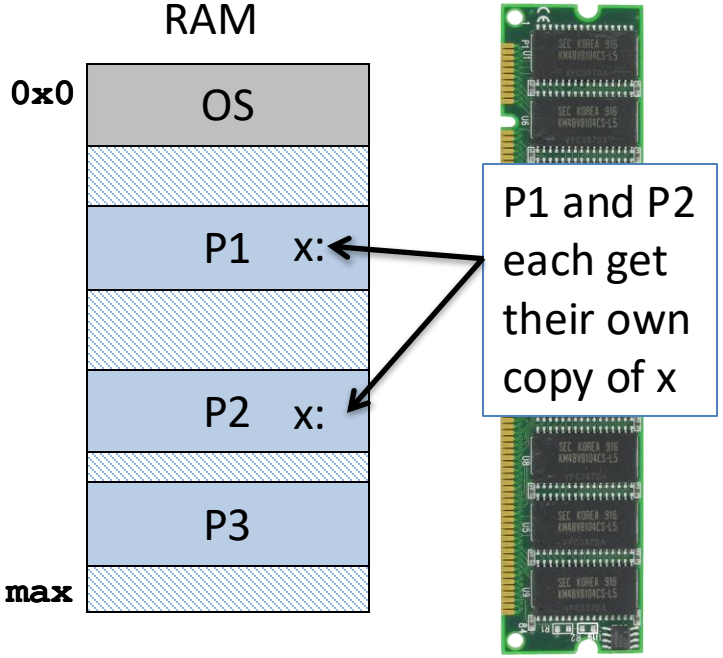
What we Know about Physical Memory (Main Memory (RAM))



RAM is array of addressable bytes, from 0x0 to max

(e.g.) address 0 to 2^{30}-1 for 1 GB of RAM space

- The OS needs to be in RAM
  - Usually loaded at address 0x0
- Physical Storage for running processes: Process Address Spaces are stored in RAM

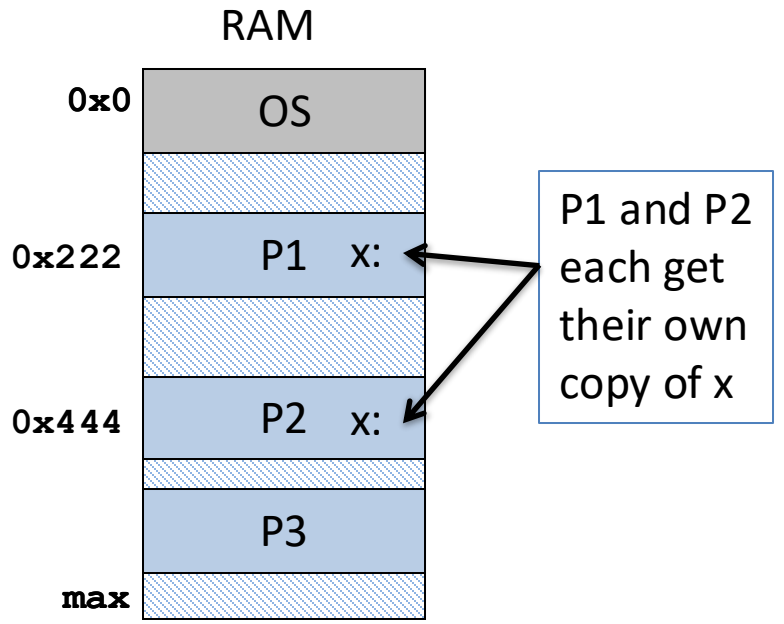


Memory and Addresses

Consider processes P1 and P2 both running the same `a.out`.
They both get their own private address space stored in RAM.

Q: What are P1 and P2's addresses for x?

A: Same. B: Different



A1: Cleary x has different RAM memory addresses in P1 and P2

Q: What about addresses from x86 instructions?

```
movq 0x1234, %rax
```

`0x1234` is the same address for every process executing this code

A2: Cleary x has same memory addresses in P1 & P2 instructions

Processes and Memory Addresses

- When running two programs, get two separate processes (P1 & P2) each with separate address space.
- The code they execute could generate the same memory addresses (and very likely does, particularly when running the same a.out):

Example of P1 and P2's execution of the same instruction		
P1:	R[%rax]: 100 R[%rdx]: 0x1238 **0xf234 movq %rax, (%rdx)**	CPU → store 100 to Memory Address 0x1238
P2:	R[%rax]: 30 R[%rdx]: 0x1238 **0xf234 movq %rax, (%rdx)**	CPU → store 30 to Memory Address 0x1238

Q: how to prevent pid P1 and pid P2 from mucking up each other's execution state?

Solution: Virtual Memory Abstraction

- Each Process has its own virtual address space.
- Virtual Address Space: Process's view of its own memory
  - 2^N contiguous addressable bytes (byte 0, 1, 2, …, 2^{32} -1)
  - Addresses in two process' virtual address space may be same
    - P1's virtual address for x is 0x1238
    - P2's virtual address for x is 0x1238

Abstraction: the addresses from x86 instructions executed by CPU)

- Virtual Address mapped to different Physical addresses (in RAM)
    - P1's virtual address 0x1238 maps to different physical address in RAM than P2's virtual address 0x1238

Reality: addresses to access RAM

- Virtual Memory: adds level of indirection to memory references
  - system translates (maps) virtual address view to physical address view

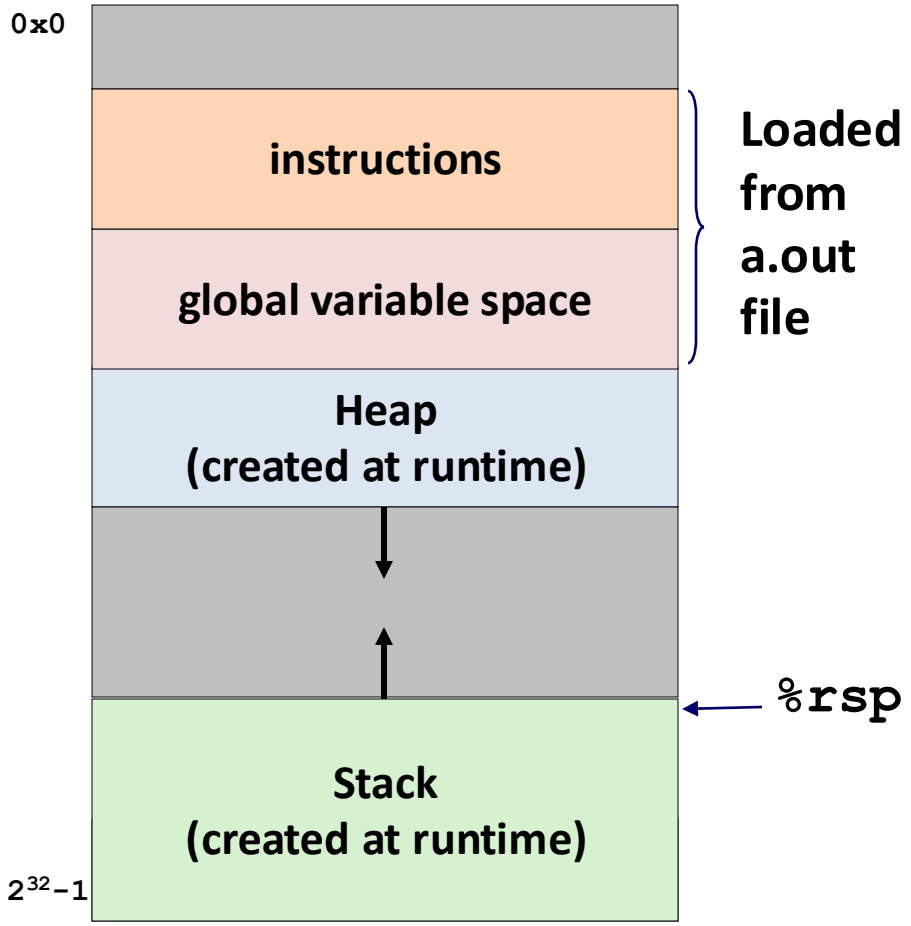
Every Process gets its own Virtual Address Space (VAS)

On fork, the OS:

- Creates child's VAS, a copy of its parent's
- OS keep track of child's VAS

On exec, OS:

- Initializes VAS:
  - Loads in code and data
  - Creates empty stack and heap space

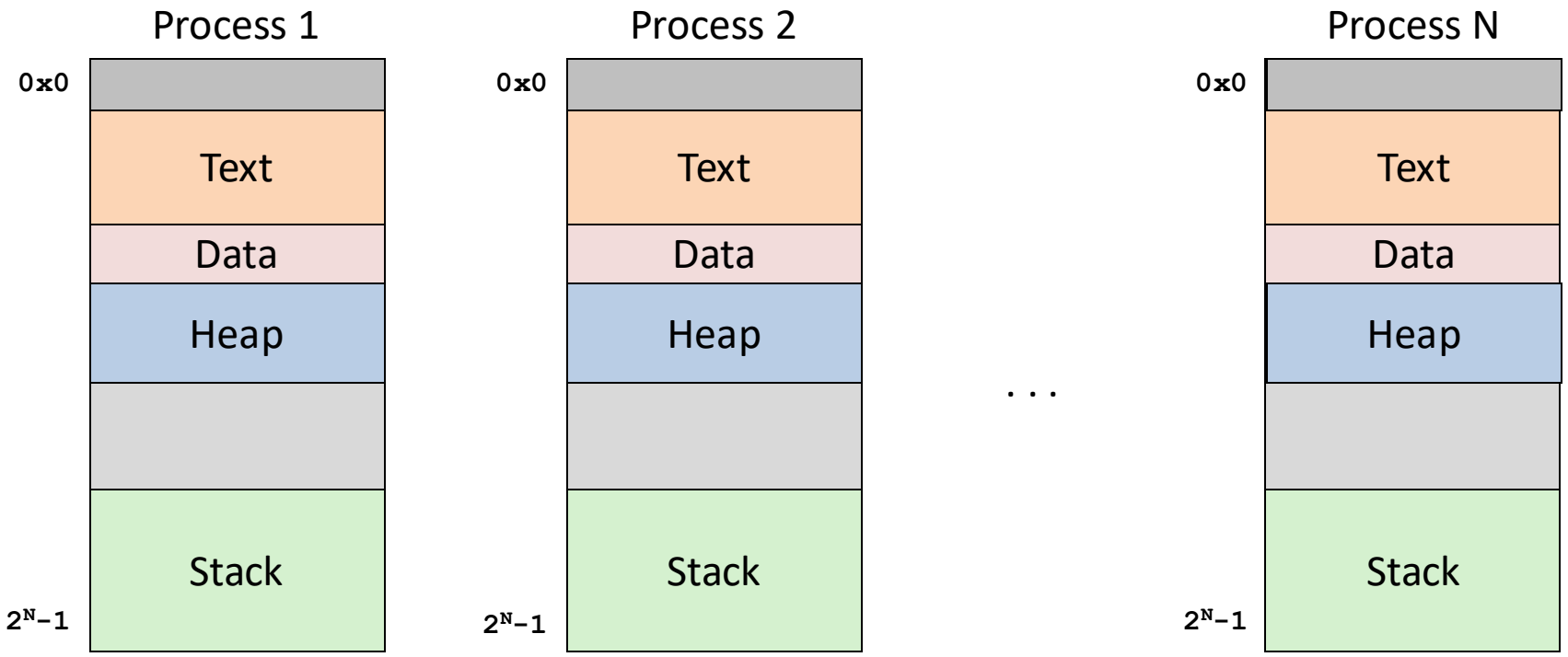


Process's virtual (logical) Address Space

Virtual Memory Abstraction Goal

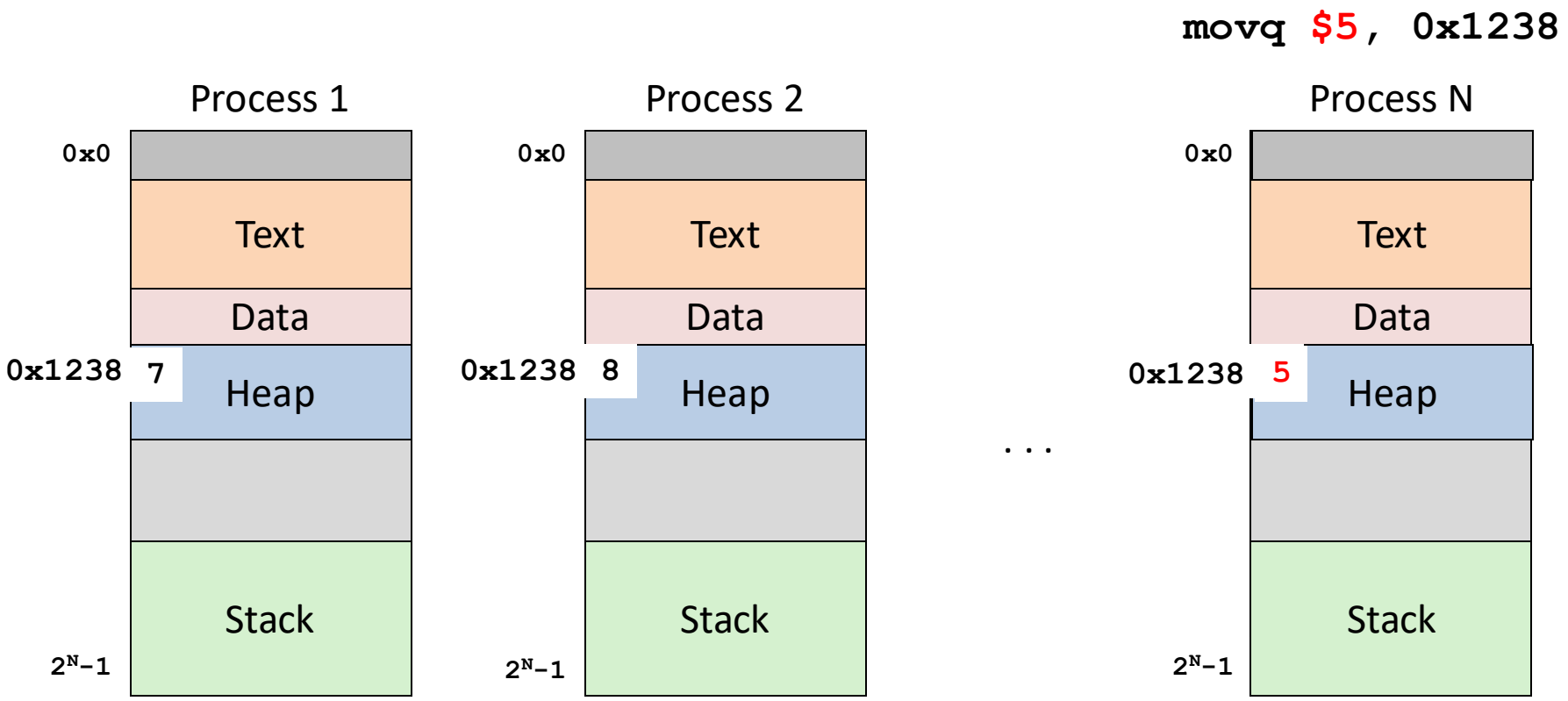
Every process has the same virtual memory layout

- Exact contents differ, but layout is the same
- Address space of 2^N bytes addressed from 0x0 to (2^N-1)
  - N could be 32, 48, 64 (2^{32} is 4GB of memory, 2^{64} is huge!)



Private Virtual Memory Abstraction

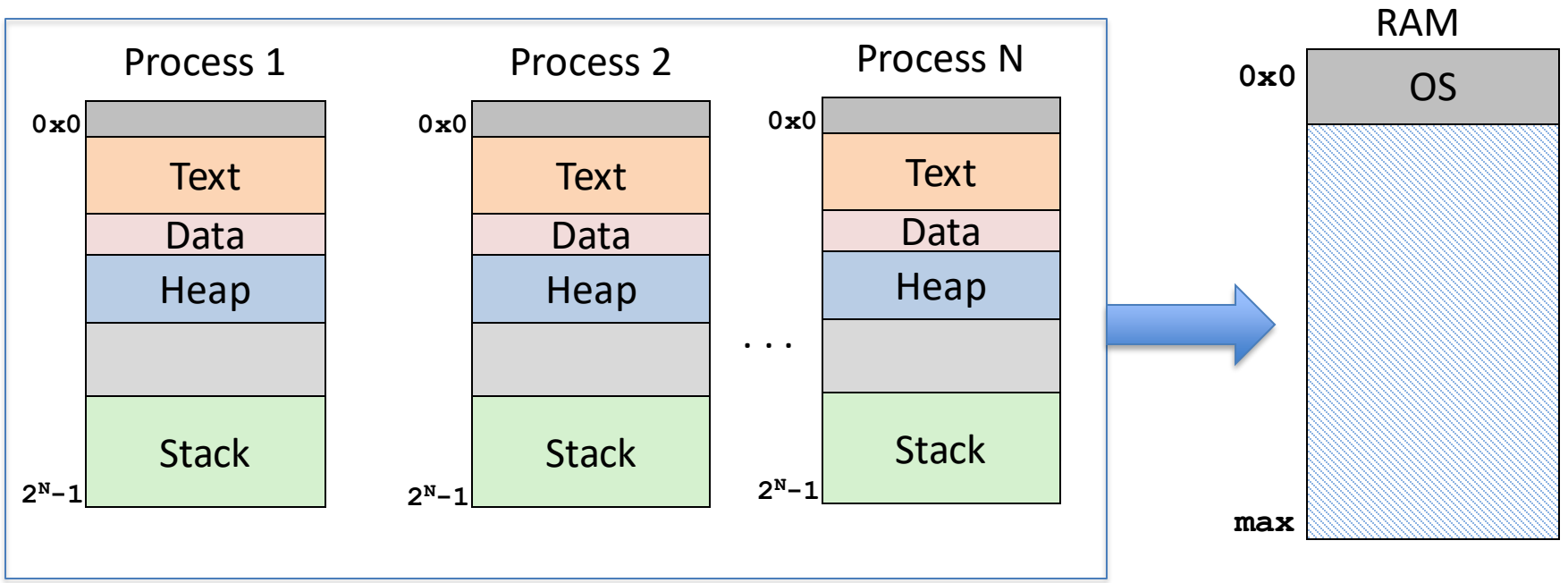
- One process could modify a value at a particular virtual address (ex 0x1238)
  - this will not interfere with any other process' values stored at the same address in their virtual memories.



Physical Memory (RAM): Reality

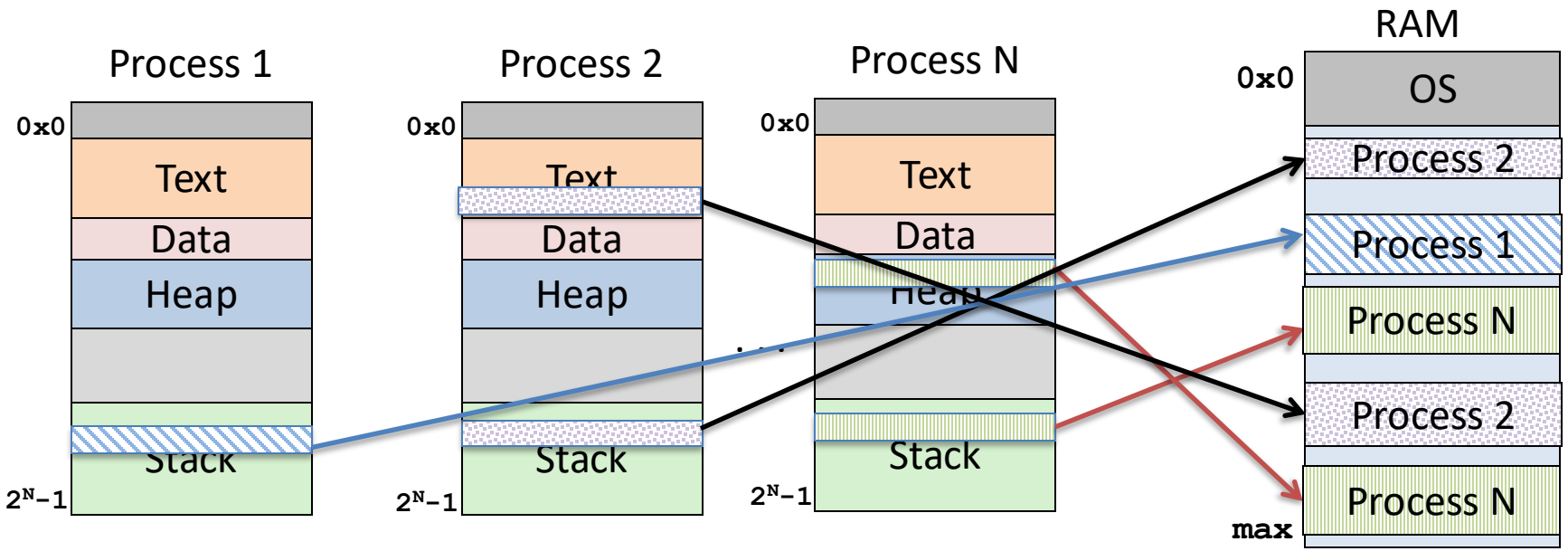
RAM is array of addressable bytes, from 0x0 to max

- max is way less than 2^{64} bytes
- Process' VAS need to be loaded into RAM
  - Each VAS could be 2^{32} or 2^{64} bytes
  - Individual VAS may be larger than RAM, together much larger



Virtual Addresses Map to Physical Addresses

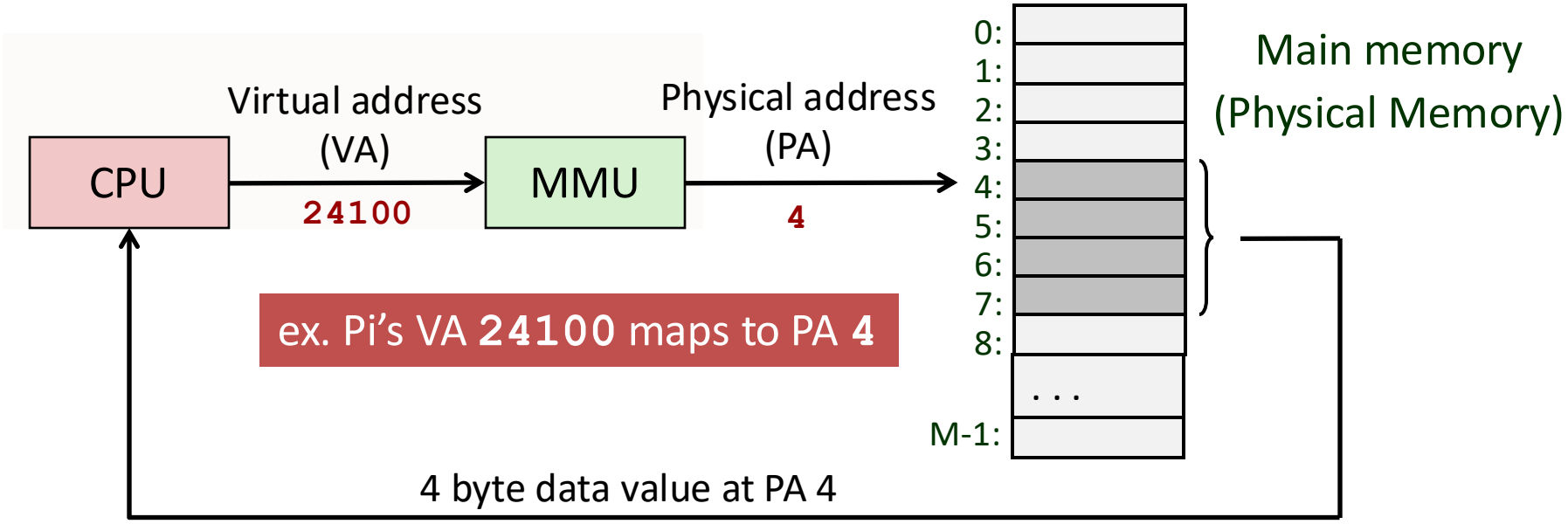
- Process' Virtual Addresses map to different Physical Addresses
- RAM cannot necessarily store even one full process' VAS
- Parts of Process' VAS must be stored at different RAM addresses



Each Processes's view of its memory (virtual address) gets mapped to reality (physical addresses)

Mapping Virtual to Physical Addresses

- CPU generates Virtual Addresses (VA):
  - From instruction execution (e.g. movl)
  - VA: Process's view of its address space
- Memory Mapping Unit (MMU) translates VA→PA
  - Physical Address (PA): Physical Memory Addresses



Where and what is the MMU?

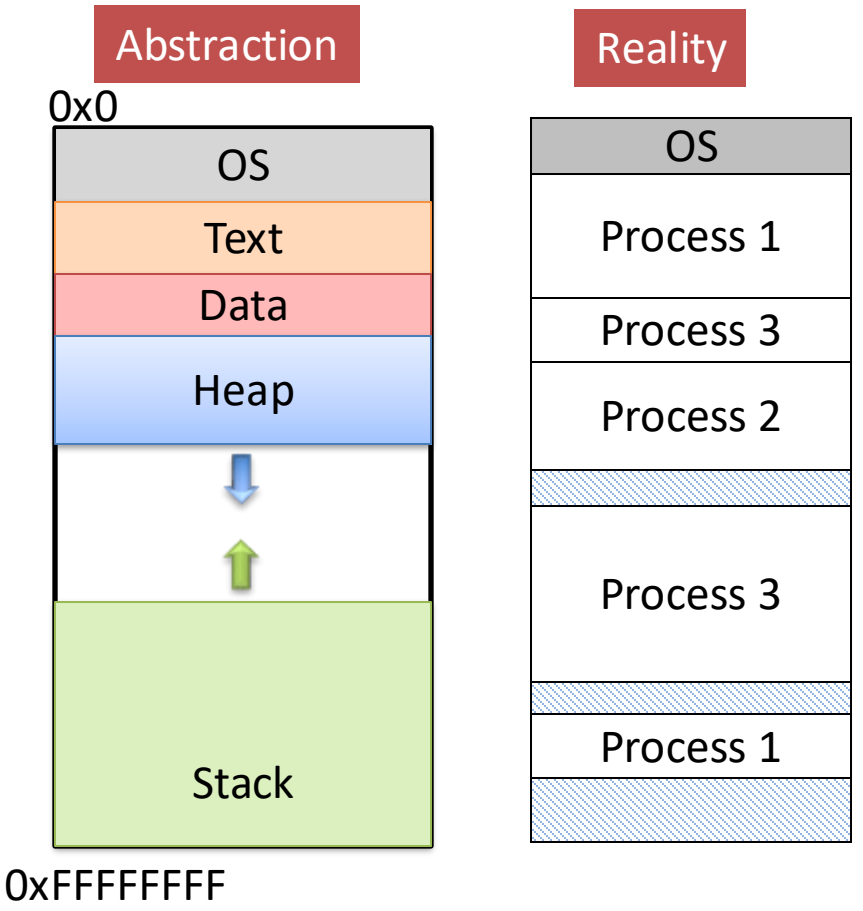
- Parts of memory mapping are implemented in HW and others in SW (OS)
  - OS and HW work together to implement Virtual to Physical Address Translation
    - Some address translation can be done using h/w circuitry
    - Some address translation by the OS
  - OS's implements mapping data structures that are stored in RAM
  - On context switch, OS sets up state needed by HW circuitry to map process' VA (Virtual Address) to PA (Physical Address)

Memory

- Abstraction goal: make every process think it has the same memory layout.
  - MUCH simpler for compiler if the stack always starts at 0xFFFFFFFF, etc.

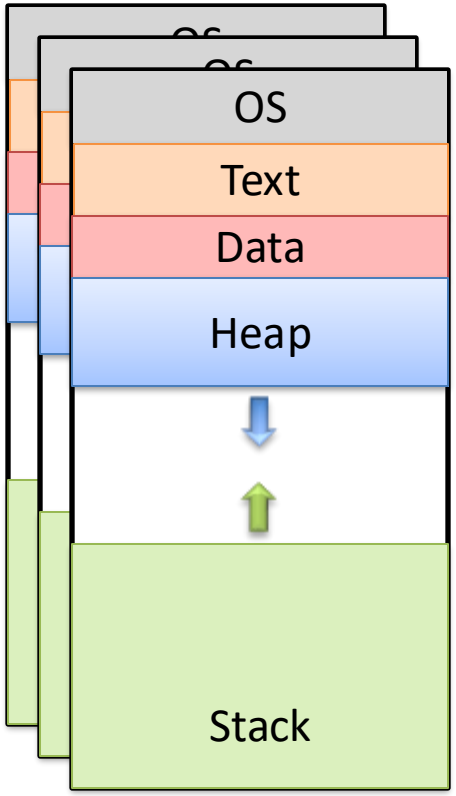
- Reality: there's only so much memory to go around, and no two processes should use the same (physical) memory addresses.

OS (with help from hardware) will keep track of who's using each memory region.



Memory Terminology

Virtual (logical) Memory: The abstract view of memory given to processes. Each process gets an independent view of the memory.



Virtual address space (VAS): fixed size.

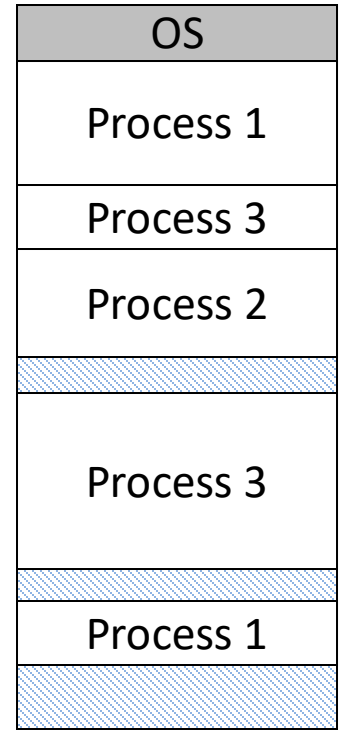
0x0

0xFFFFFFFF

Address Space: Range of addresses for a region of memory.

The set of available storage locations.

Physical Memory: The contents of the hardware (RAM) memory. Managed by OS. Only ONE of these for the entire machine!



0x…

(Determined by amount of installed RAM.)

Paging Vocabulary

- For each process, the virtual address space is divided into fixed-size pages.
- For the system, the physical memory is divided into fixed-size frames.
- The size of a page is equal to that of a frame.
  - Often 4 KB in practice.

Main Idea

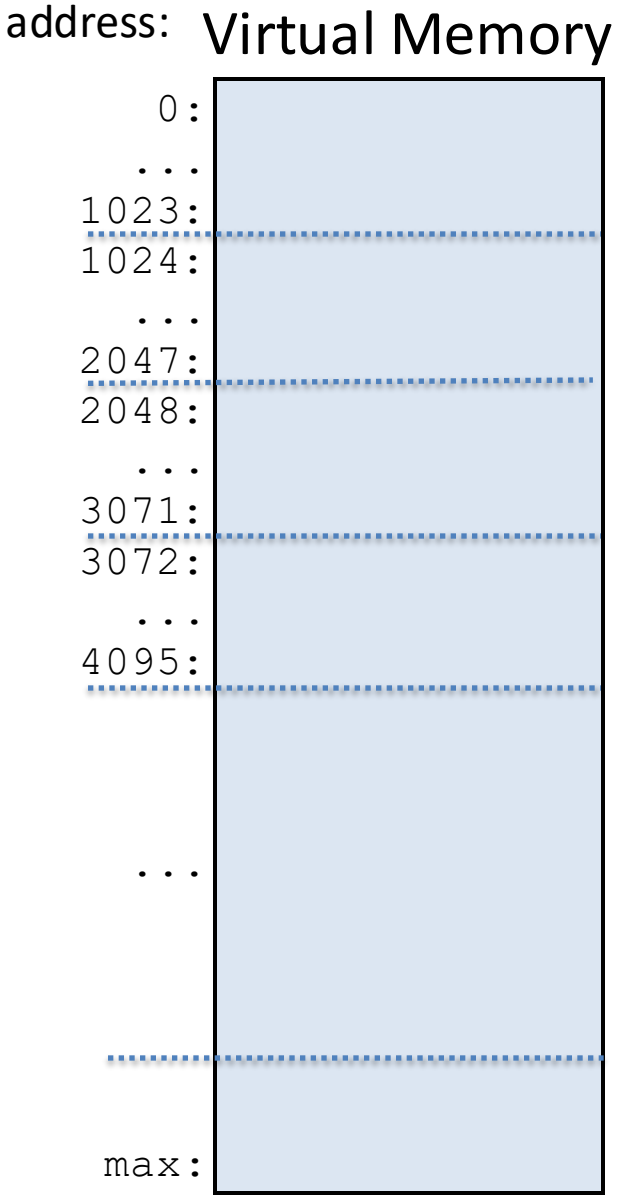
- ANY virtual page can be stored in any available frame.
  - find an appropriately-sized memory gap?
  - very easy!– they're all the same size.

- For each process, OS keeps a table mapping:
  - each virtual page maps to a physical frame.

Paging

- Common implementation of Virtual Memory
  - OS divides each process's VA space into some number of contiguous pages
    - Typically 4KB, but it varies and is often configurable
  - OS divides RAM into some number of physical frames of the same size (i.e. typically 4KB)
  - Any page of VM can be stored in any frame of PM

Virtual Memory Pages



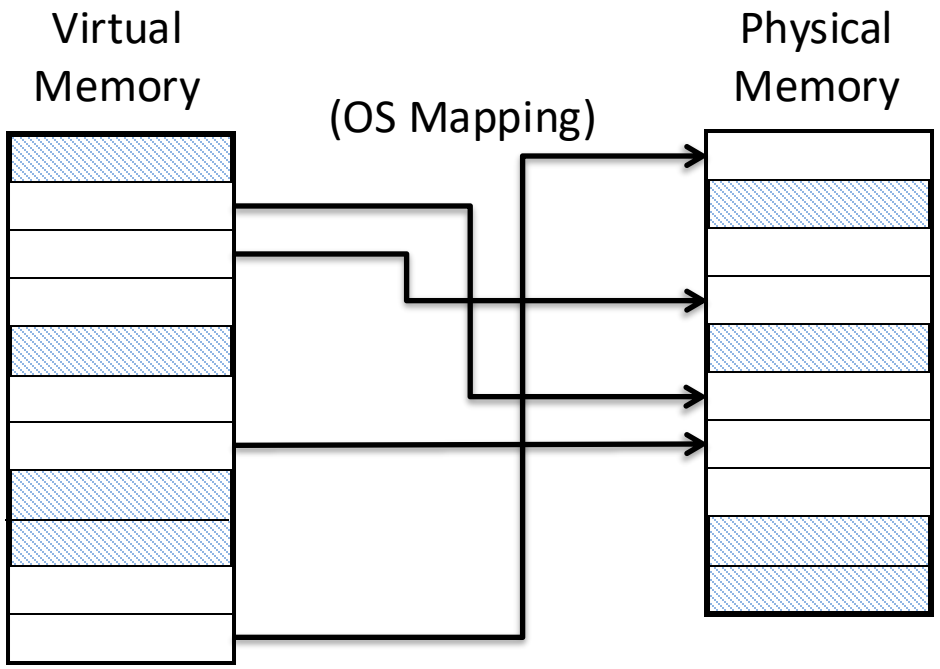
Page: grouping of a contiguous chunk of memory space of some size

e.g. Page is 1Kb (2^{10} bytes) (1024 bytes)

Memory as an array of pages

Main Idea

- ANY virtual page can be stored in any available frame.
  - find an appropriately-sized memory gap?
  - very easy!– they're all the same size.



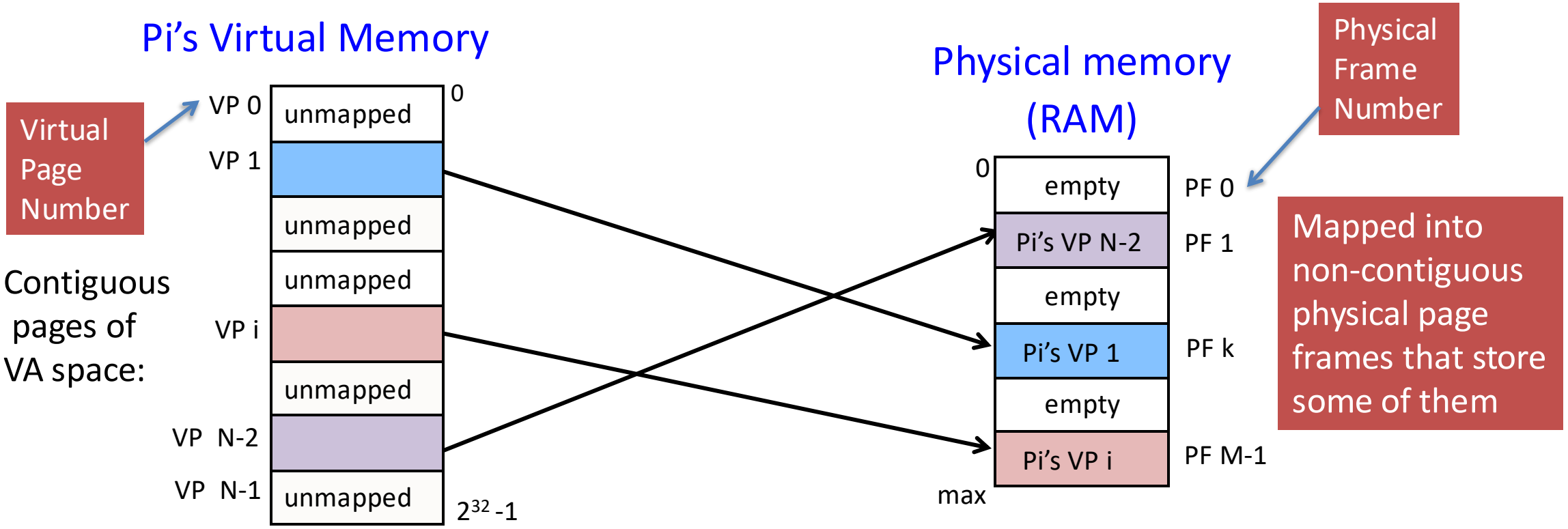
Implications for fragmentation?

Internal Fragmentation: About the same. Process can always request memory and not use it.

Paging

VM and PM divided into Page-sized chunks

Any page of VM can be stored in any frame of PM

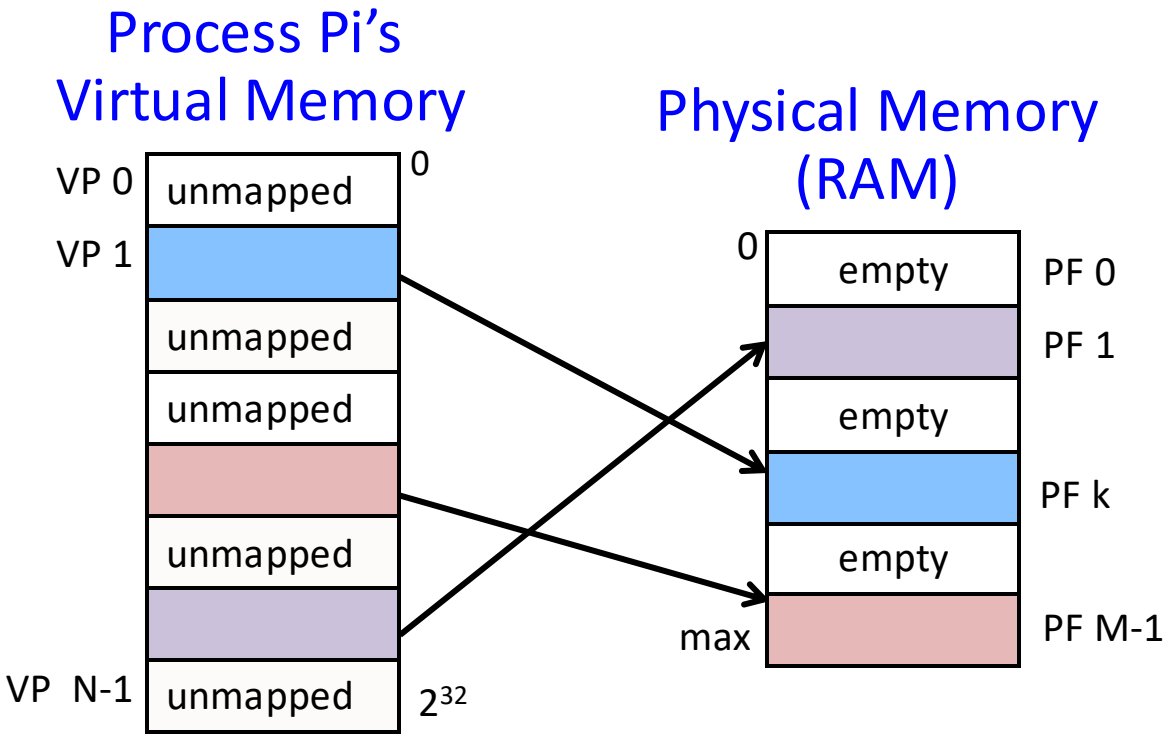


Virtual Pages of Pi's logical address space

Physical Frames of RAM (physical Mem.)

Paging

Any page of VM can be stored in any frame of PM



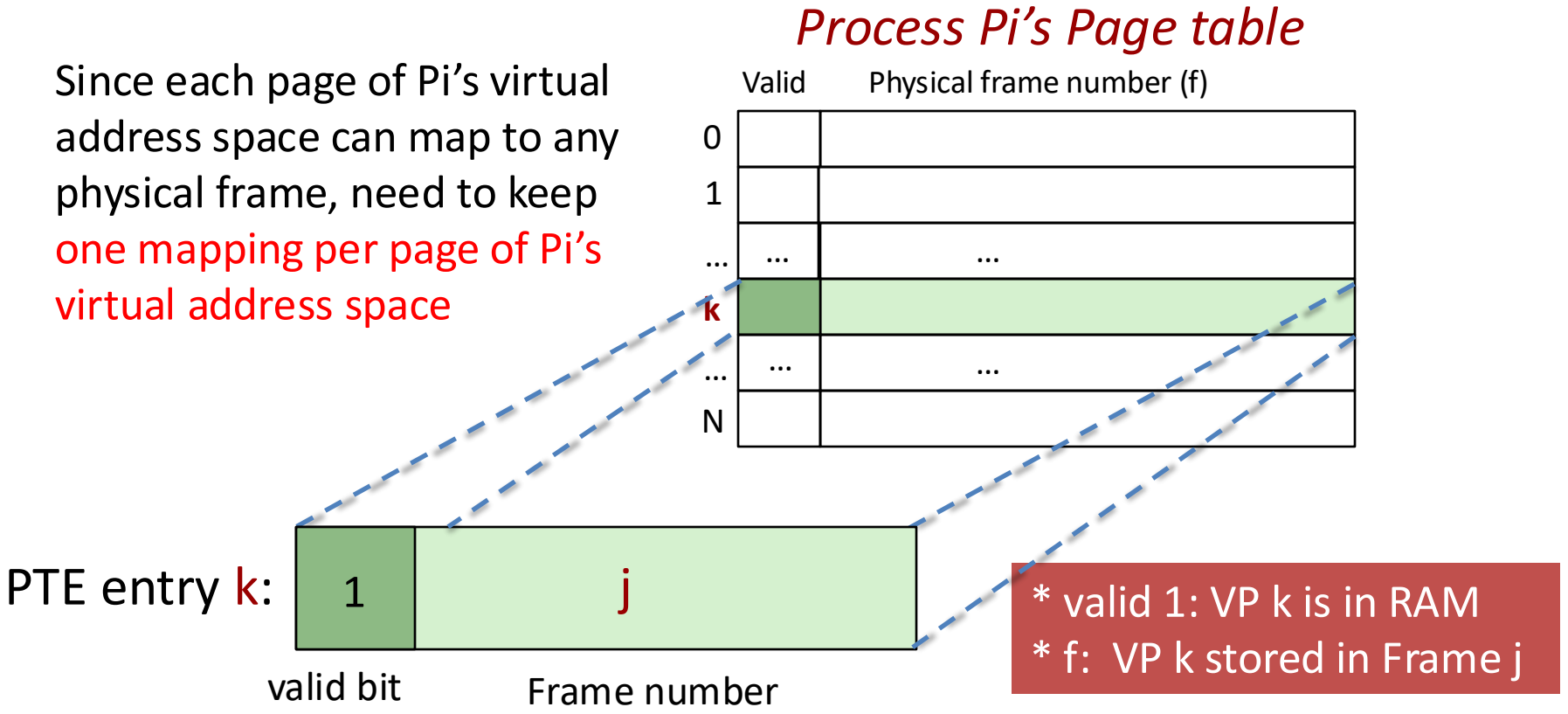
Q: How to know which Physical Frame of RAM stores Pi's Virtual Page p?

A: OS has to keep mapping of VP to PF for each process

Page Tables

OS keeps a page table data structure for each process

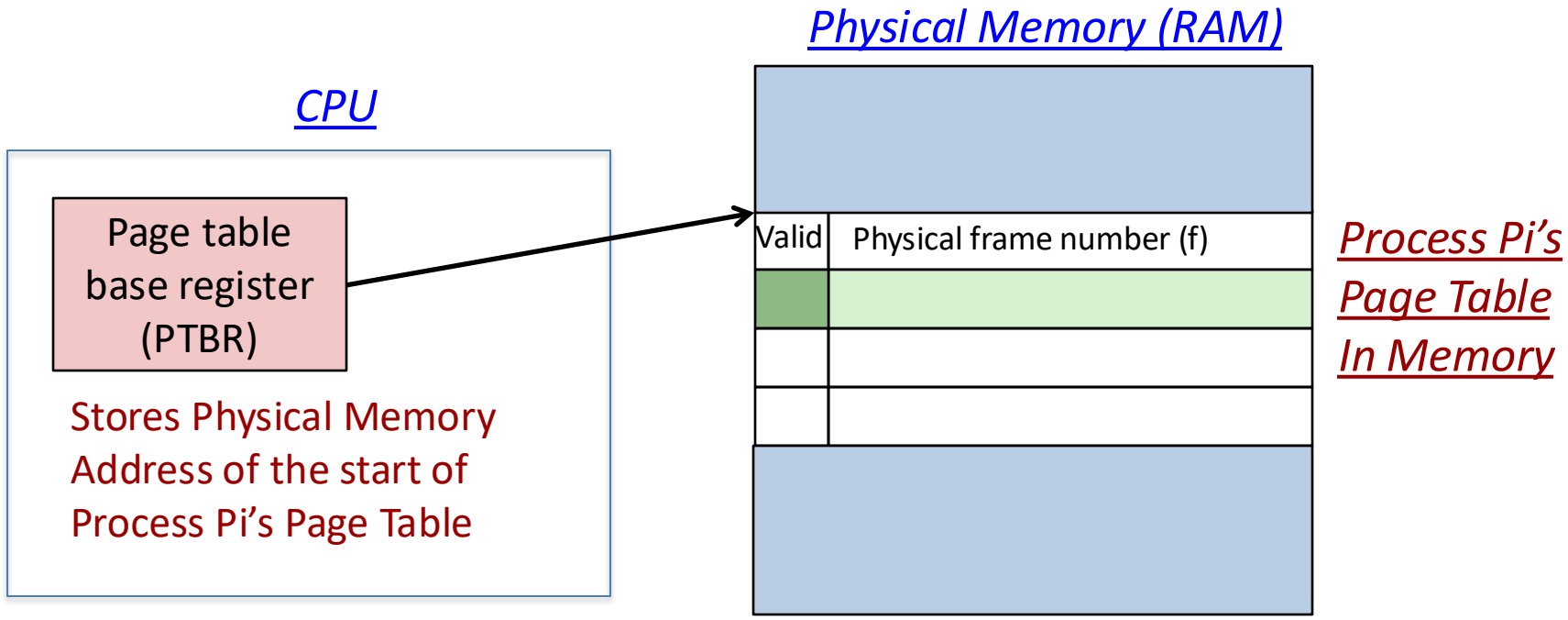
Each page table entry (PTE) maps one of the process's virtual pages to a physical page frame of RAM



Page Tables are Stored in Memory

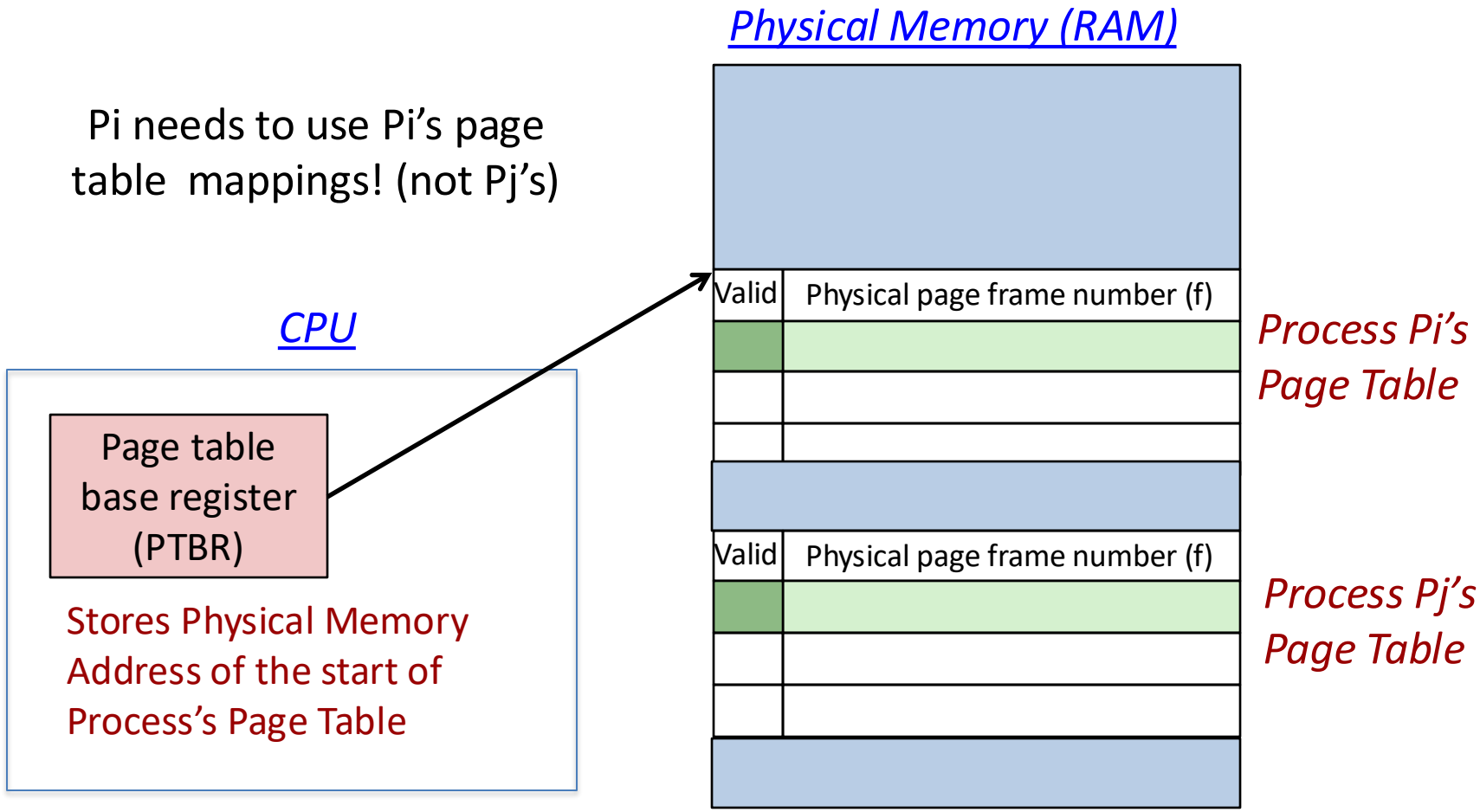
With Each Process, the OS keeps the base address of its Page Table in memory

- Page Table: array of PTEs indexed by virtual page number
- On a context switch: OS loads Pi's page table base address into a special register: Page Table Base Register



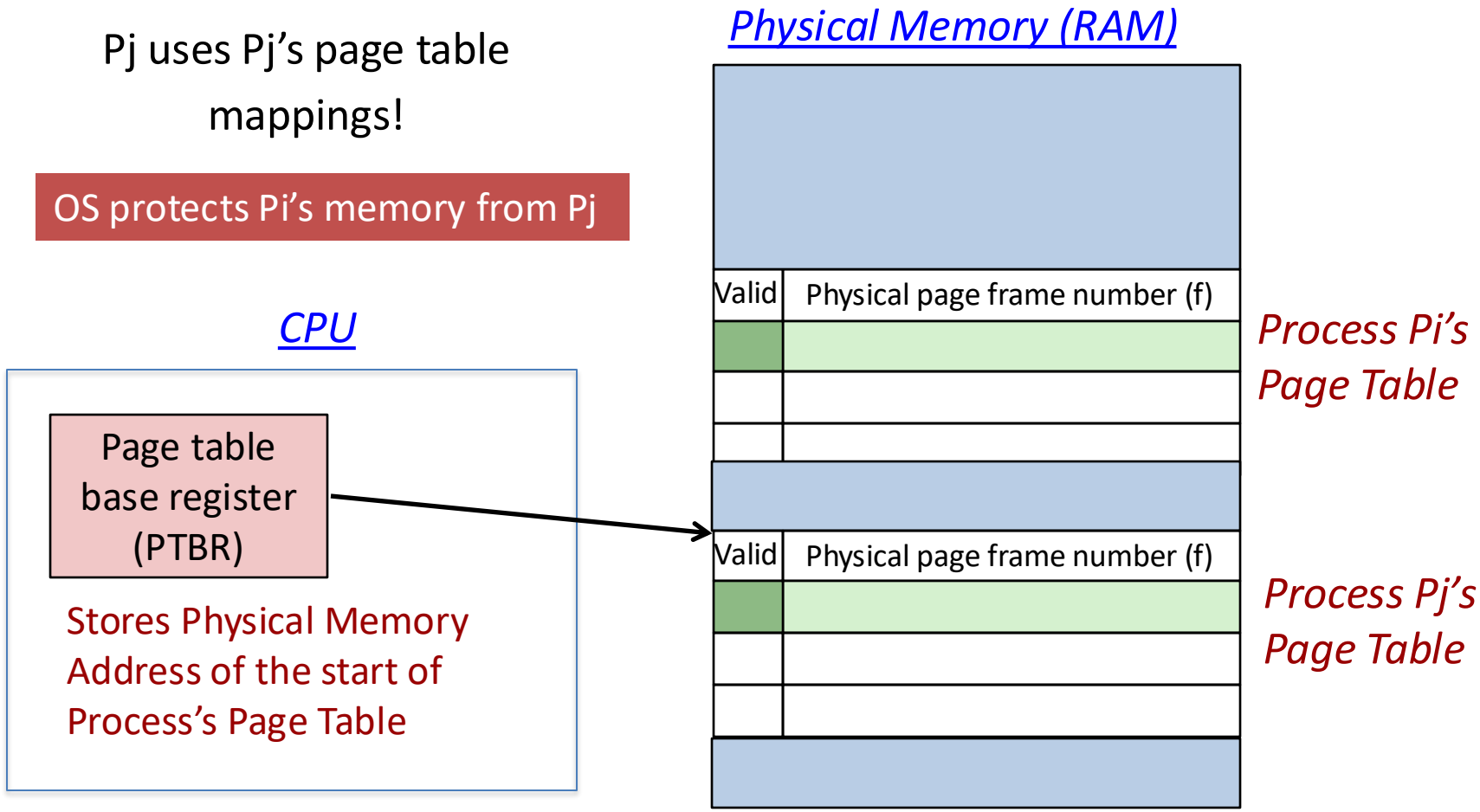
OS and PTBR on Context Switch

On a context switch Pi to Pj: OS saves Pi's PTBR value and loads Pj's PTBR value into PTBR



OS and PTBR on Context Switch

On a context switch Pi to Pj: OS saves Pi's PTBR value and loads Pj's PTBR value into PTBR

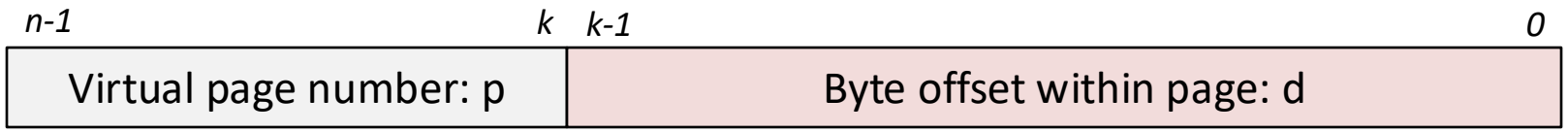


Virtual Addresses

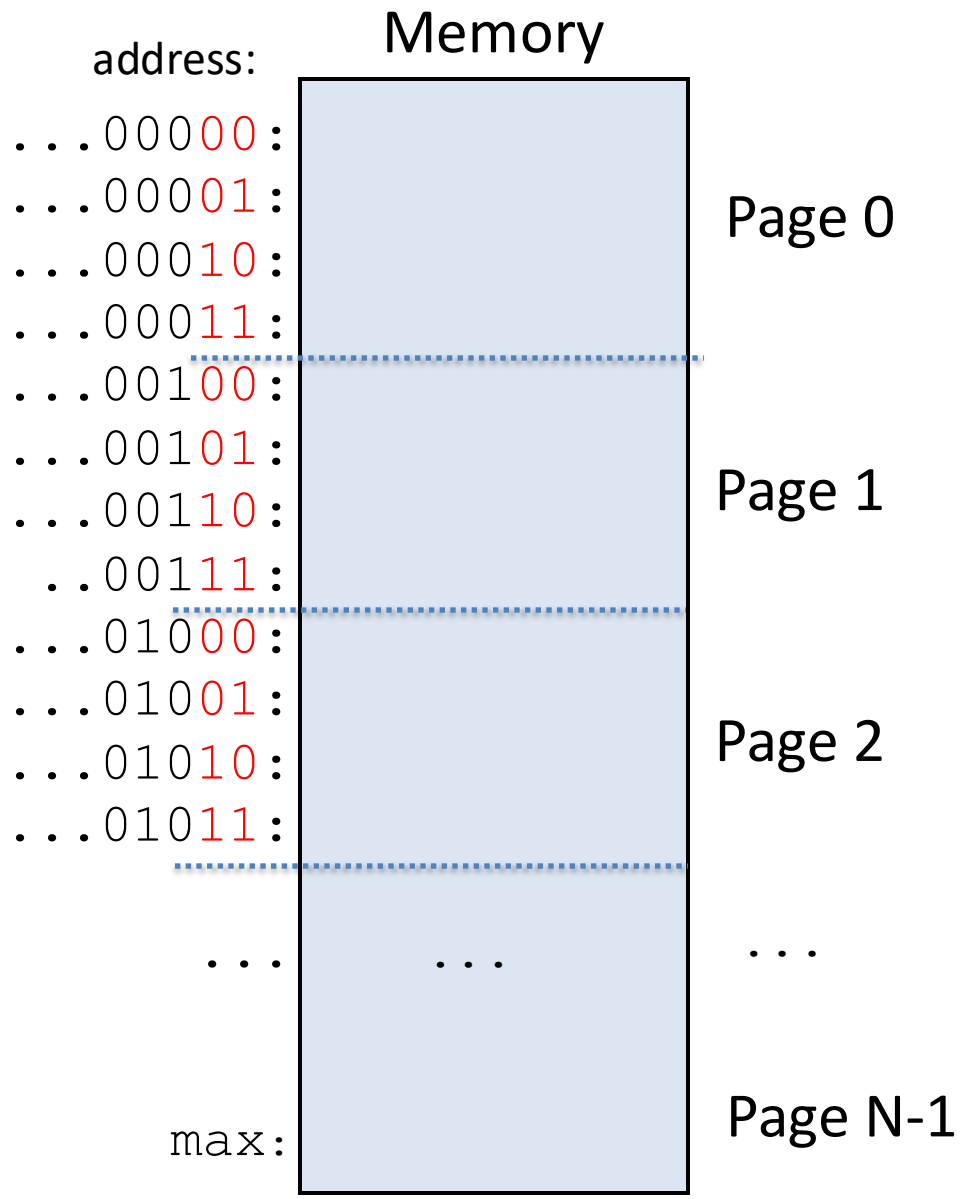
Every byte's virtual address is divided into 2 parts:

1. Byte offset within a page (d): low-order bits
   - Number of bits depends on the page size
   - 4KB pages: 4KB is 2^{12}, so 12 bits for page offset
2. Page Number (p): high-order bits
   - Whichever high-order bits are left after byte offset bits

For a Virtual Address Space of 2^n bytes, with page size of 2^k bytes, the VA bits are interpreted as:



Page Offset and Page Number



e.g.
If page size if 4 bytes

page offset is:
low-order 2 bytes
(which byte of 4 in the page)

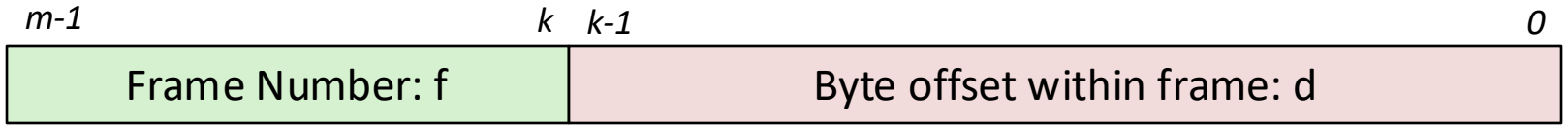
page number is:
remaining high-order bits

Physical Addresses

Every byte's physical address is divided into 2 parts:

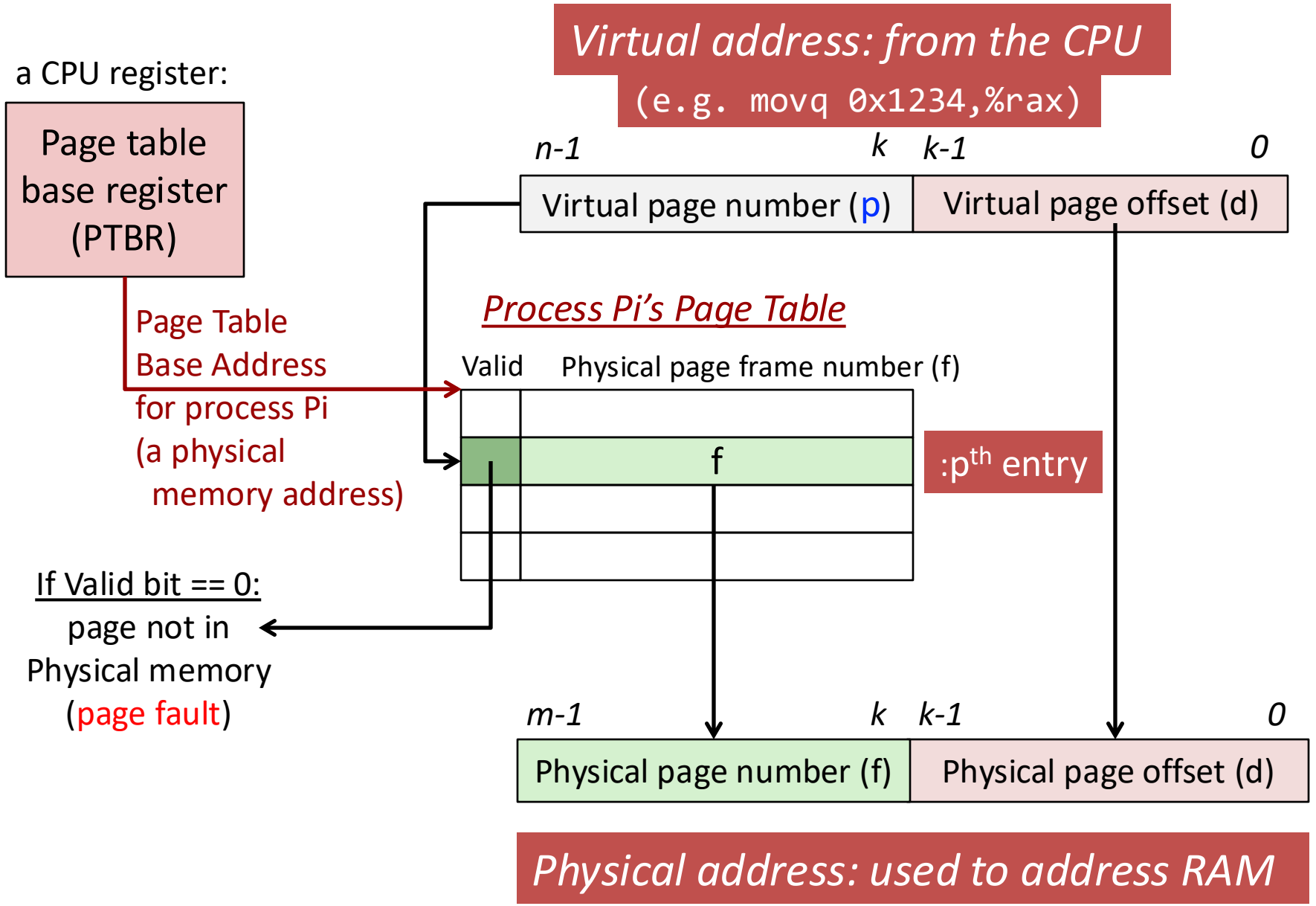
1. Byte offset within a page (d): low-order bits
   - Number of bits depends on the page size
   - Virtual Page and physical frame are the same size, so the byte offset bits from the VA are identical to the byte offset bits in the PA
2. Frame Number (f): high-order bits
   - The high-order bits that are left over after the byte offset bits

For a Physical Addresses space of 2^m bytes, with a page (and frame) size size of 2^k bytes, the PA bits are interpreted as:



Number of Frames of PAS and Pages of VAS can differ: VAS and PAS can be different sizes

Virtual to Physical Address Translation



Example

Virtual Address: 8 bits, Page Size: 16 bytes, 8 page frames of RAM

- For each VA below, break into its Page Number and Offset and translate to is Physical Address using part of the Page Table below

Virtual Address	Page Number	Offset	Physical Address
00110101			
00011100			
00000010			
00111101			
00100101			

Page Table (1st part of it)		
Entry Num	Valid	Frame Num
0000	1	101
0001	1	110
0010	1	111
0011	1	010
0100	1	000

1. How many bits of VA for page offset?
2. How many bits of VA for page number?
3. How many bits are physical addresses?

Example: Answer these questions 1st

Virtual Address: 8 bits

Page Size: 16 bytes

8 page frames of RAM

1. How many bits of VA for page offset?

2. How many bits of VA for page number?

3. For VA 10011010:
    1. what is the page number?
    2. what is the page offset?

4. How many bits are in a physical address?

Example

Virtual Address: 8 bits, Page Size: 16 bytes, 8 page frames of RAM

- For each VA below, break into its Page Number and Offset and translate to is Physical Address using part of the Page Table below

Virtual Address	Page Number	Offset	Physical Address
00110101			
00011100			
00000010			
00111101			
00100101			

Page Table (1st part of it)		
Entry Num	Valid	Frame Num
0000	1	101
0001	1	110
0010	1	111
0011	1	010
0100	1	000

1. How many bits of VA for page offset?
2. How many bits of VA for page number?
3. How many bits are physical addresses?

Example Solution

Virtual Addresses are 8 bits and Page Size is 16 bytes:

low order 4 bits for page offset ($2^4 = 16$)

remaining high-order 4 bits for virtual page number

Virtual Address	Page Number	Offset	Physical Address
00110101	**0011**	**0101**	**0100101**
00011100			
00000010			
00111101			
00100101			

Page Table		
Entry Num	Valid	Frame Num
0000	1	101
0001	1	110
0010	1	111
0011	1	**010**
0100	1	000

8 Frames of Physical Memory, each 16 bytes:

- 3 bits for frame num
- 4 bits for page offset

PAs are 7 bits

Example Solution

Virtual Addresses are 8 bits and Page Size is 16 bytes:

low order 4 bits for page offset ($2^4 = 16$)

remaining high-order 4 bits for virtual page number

Virtual Address	Page Number	Offset	Physical Address
00110101	0011	0101	0100101
00011100	0001	1100	1101100
00000010	0000	0010	1010010
00111101	0011	1101	0101101
00100101	0010	0101	1110101

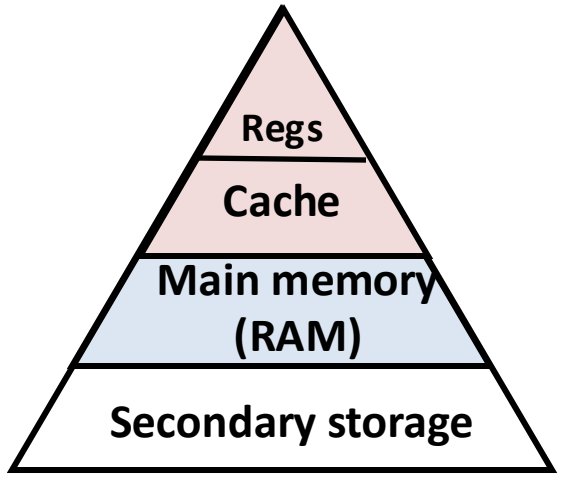
Page Table		
Entry Num	Valid	Frame Num
0000	1	101
0001	1	110
0010	1	111
0011	1	010
0100	1	000

8 Frames of Physical Memory, each 16 bytes:

- 3 bits for frame num
- 4 bits for page offset

PAs are 7 bits

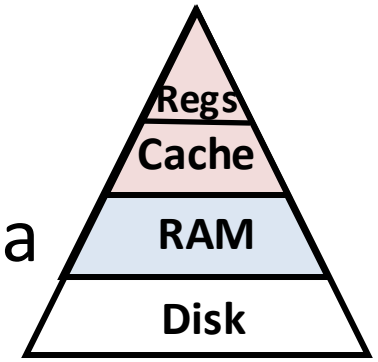
Not All pages of Pi's VAS need to be stored in RAM (and they might not all fit)



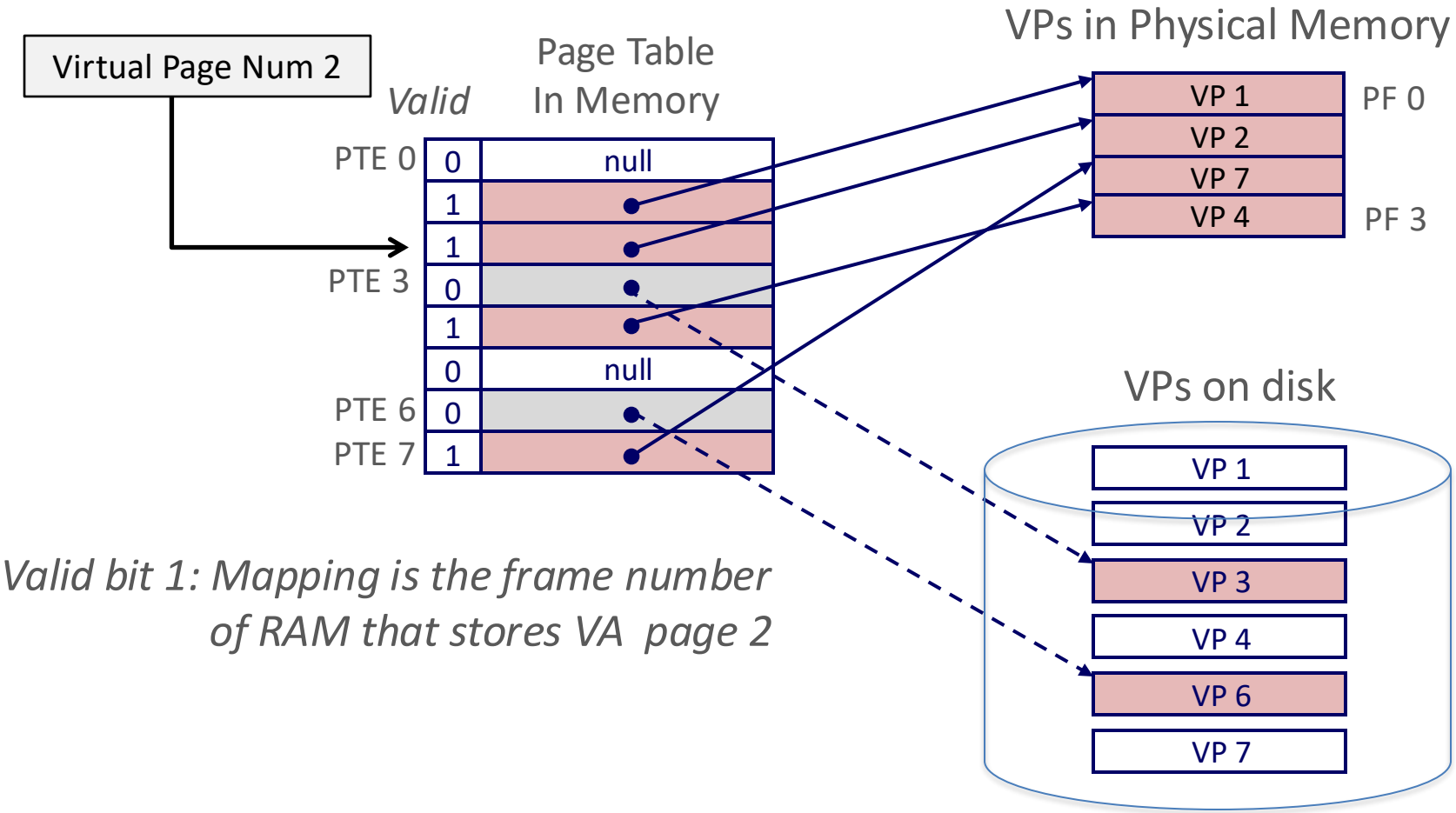
+ More Efficient Use of RAM
  - Unused or rarely used pages of a Pi's virtual address space can be on disk
  - RAM space for storing pages that are actively being used
  - Main Memory is a cache for Virtual Address space on disk

Page Table Valid bit indicates if a Virtual Page is currently stored in RAM
  - 1: yes, PTE entry stores valid Frame num mapping
  - 0: no, PTE entry stores disk location of page

Page Hit

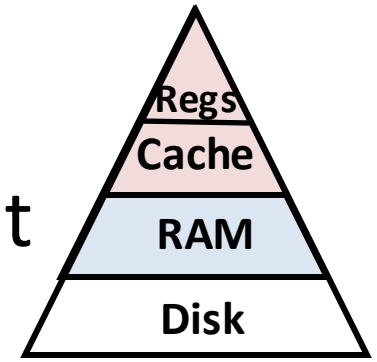


Page hit: reference to VM address that is on a page stored in Physical Memory

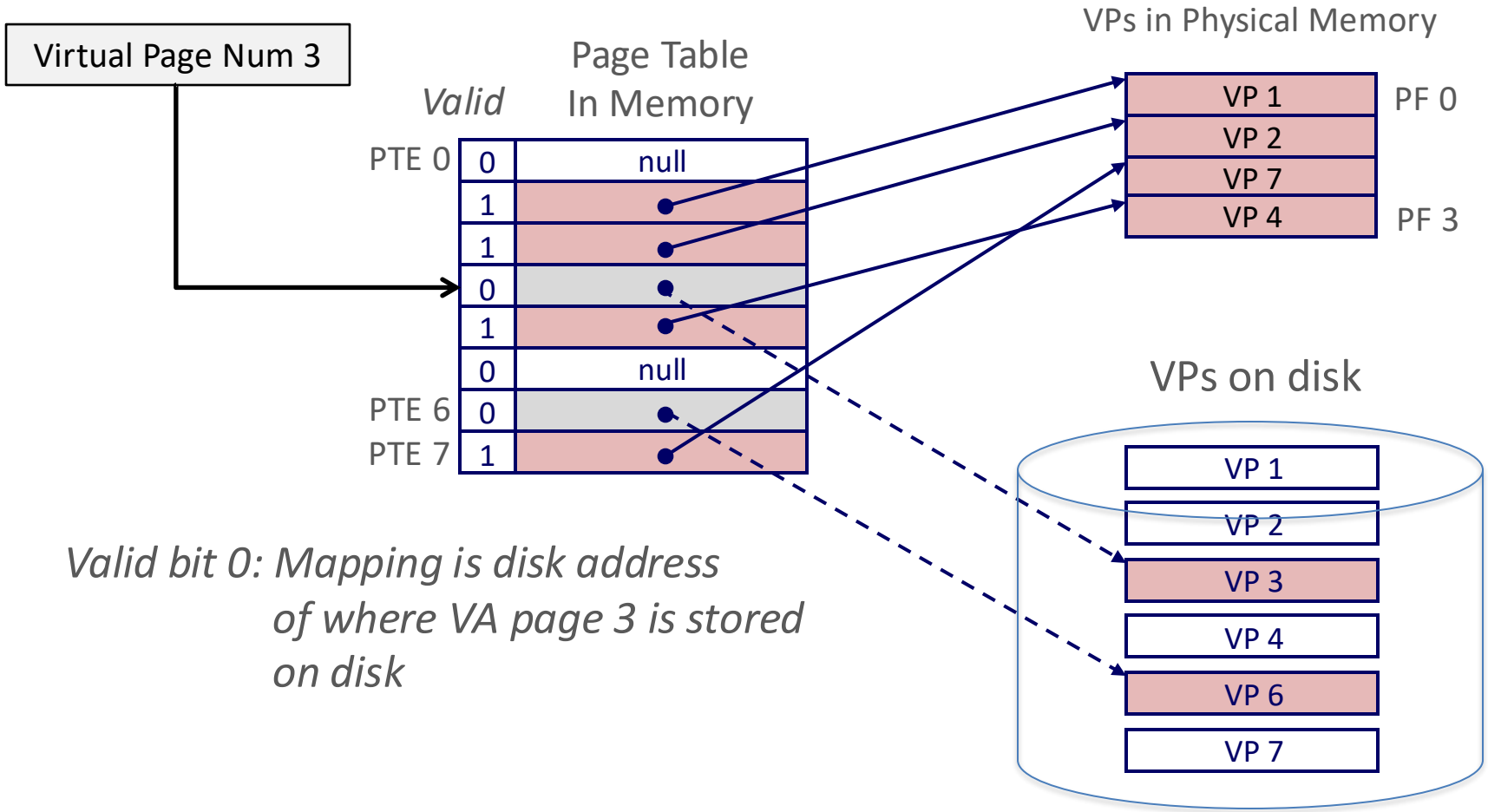


Valid bit 1: Mapping is the frame number of RAM that stores VA page 2

Page Fault



- *Page fault:* reference to VM address that is not in physical memory



Valid bit 0: Mapping is disk address of where VA page 3 is stored on disk

Page Fault Handling by the OS

Page fault causes an interrupt to get OS to handle it

mechanism of handling a page fault:

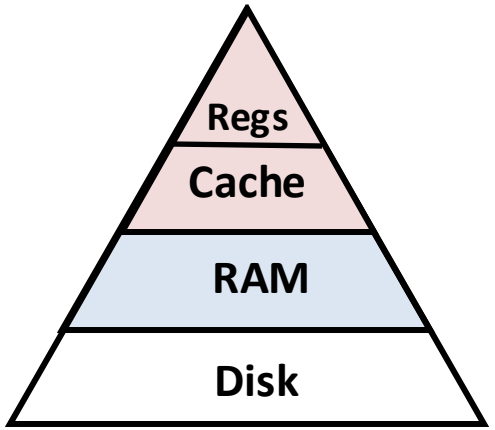
1. reads in the virtual page from disk
2. stores it in a physical memory frame
3. Updates PTE with frame num & valid bit = 1
4. Restarts instruction that cause the page fault

If RAM is full, the OS needs to pick a page to kick out of RAM: OS needs a page replacement **policy**

– FIFO, Random, LRU, …

Page Faults are Expensive

- Disk: 5-6 orders magnitude slower than RAM
  - Very expensive; but if very rare, tolerable
- Example
  - RAM access time: 100 nsec
  - Disk access time: 10 msec
  - p = page fault probability
  - Effective access time: 100 + p × 10,000,000 nsec
  - If p = 0.1% (99.9% hit rate), effective access time = 10,100 nsec!
- Good Replacement Policy can have huge effect on performance (keep page fault rate low)



CS45 (and CS44) investigate page replacement policies

Another Paging Example

Step through stream of Virtual Addresses from two processes context switched on and off CPU:

1. Show how bits of each address is used for each Virtual Address & its Physical Address mapping
2. Translate each VA to its PA using the appropriate PTE
3. Update Pi and Pj's PTEs appropriately as you go
4. Show the history of the contents of RAM as these addresses are accessed

   (which virtual page of which process does it store)
5. Implement FIFO (First-In-First-Out) page replacement

order of memory accesses of 2 processes, Pi and Pj, running on CPU (note context switches)

Pi running:	VA (page #, offset)	PA (frame #, offset)	Page Fault?
00011101			
01100001			
11100001			
00010010			
Pj running (context switch)			
00011101			
00010010			
01100001			
11100001			
Pi running: (context switch)			
00011010			
01100100			
Pj running: (context switch)			
11101111			
01100110			

Pi's Page Table

Pi's	Valid	Frame
0	0	
1	0	
2	0	
3	0	
4	0	
5	0	
6	0	
…		
14	0	
15	0	

Pj's Page Table

Pj's	Valid	Frame
0	0	
1	0	
2	0	
3	0	
4	0	
5	0	
6	0	
…		
14	0	
15	0	

RAM

frame	Whose Page
0	
1	
2	
3	

Page Size: 16 bytes
4 frames of RAM
8 bit virtual addresses

Let's figure out VA and PA bits and sizes first….

Virtual Address (VA): 8 bits

Page Size: 16 bytes

4 frames of RAM

Q1: How many bits of VA for page offset?

Q2: How many bits of VA for page number?

Q3: How many bits are physical addresses? How many for frame number, and for frame offset?

Virtual Address: 8 bits

Page Size: 16 bytes

4 frames of RAM

Q1: How many bits of VA for page offset? 4

- page size is 16 bytes (2^4) ➔ need 4 bits to address all 16 bytes in the page

Q2: How many bits of VA for page number? 4

- Num bits in VA – Num bits for page offset = 8 – 4 = 4

Q3: How many bits are physical addresses? 6

- 4 bits for the frame offset (to address bytes in each frame frame of 16 bytes)
- Plus 2 bits to specify which of 4 frame numbers (2^2 == 4)

Pi's	Valid	Frame
0	0	
1	0	
2	0	
3	0	
4	0	
5	0	
6	0	
…		
14	0	
15	0	

Pj's	Valid	Frame
0	0	
1	0	
2	0	
3	0	
4	0	
5	0	
6	0	
…		
14	0	
15	0	

RAM	Whose Page
0	
1	
2	
3	

Page Size: 16 bytes
4 frames of RAM

8 bit VA: 4 bits page num
4 bits page offset

6 bit PA: 2 bits frame num
4 bits frame offset

Pi running:	VA (page #, offset)	PA (frame #, offset)	Page Fault?
00011101	0001 1101		Yes: Valid bit ==0
01100001			
11100001			
00010010			

Pi's	Valid	Frame
0	0	
1	0	
2	0	
3	0	
4	0	
5	0	
6	0	
…		
14	0	
15	0	

Pj's	Valid	Frame
0	0	
1	0	
2	0	
3	0	
4	0	
5	0	
6	0	
…		
14	0	
15	0	

RAM	Whose Page
0	
1	
2	
3	

Page Size: 16 bytes
4 frames of RAM

8 bit VA: 4 bits page num
4 bits page offset

6 bit PA: 2 bits frame num
4 bits frame offset

Pi running:	VA (page #, offset)	PA (frame #, offset)	Page Fault?
00011101	0001 1101		Yes: Valid bit ==0
01100001			
11100001			
00010010			

Pi's	Valid	Frame
0	0	
1	0	
2	0	
3	0	
4	0	
5	0	
6	0	
…		
14	0	
15	0	

Pj's	Valid	Frame
0	0	
1	0	
2	0	
3	0	
4	0	
5	0	
6	0	
…		
14	0	
15	0	

RAM	Whose Page
0	Pi:0001
1	
2	
3	

Page Size: 16 bytes
4 frames of RAM

8 bit VA: 4 bits page num
4 bits page offset

6 bit PA: 2 bits frame num
4 bits frame offset

Pi running:	VA (page #, offset)	PA (frame #, offset)	Page Fault?
00011101	0001 1101	00 1101	Yes: Valid bit ==0
01100001			
11100001			
00010010			

Pi's	Valid	Frame
0	0	
1	~~0~~ 1	0
2	0	
3	0	
4	0	
5	0	
6	~~0~~ 1	1
…		
14	~~0~~ 1	2
15	0	

Pj's	Valid	Frame
0	0	
1	0	
2	0	
3	0	
4	0	
5	0	
6	0	
…		
14	0	
15	0	

RAM	Whose Page
0	Pi:1
1	Pi:6
2	Pi:14
3	

4 bits for page offset
8 bit virtual address
6 bit physical addresses

Pi running:	VA (page #, offset)	PA (frame #, offset)	Page Fault?
00011101	(0001, 1101)	001101 (00, 1101)	Yes: Valid bit ==0
01100001	(0110, 0001)	010001 (01, 0001)	Yes: Valid bit == 0
11100001	(1110, 0001)	100001 (10, 0001)	Yes: Valid bit == 0
00010010	(0001, 0010)	000010 (00, 0010)	No: V==1, Frame#: 0

Pi's	Valid	Frame
0	0	
1	~~1~~ 0	0
2	0	
3	0	
4	0	
5	0	
6	~~1~~ 0	1
…		
14	1	2
15	0	

Pj's	Valid	Frame
0	0	
1	~~0~~ 1	3
2	0	
3	0	
4	0	
5	0	
6	~~0~~ 1	0
…		
14	~~0~~ 1	1
15	0	

RAM	Whose Page
0	~~Pi:1~~, Pj:6
1	~~Pi:6~~, Pj:14
2	Pi:14
3	Pj:1

4 bits for page offset
8 bit virtual address
6 bit physical addresses

Pj running	VA (page #, offset)	PA (frame #, offset)	Page Fault?
00011101	(0001, 1101)	111101 (11, 1101)	Yes (V==0)
00010010	(0001, 0010)	110010 (11, 0010)	No (V==1, Frame#: 3)
01100001	(0110, 0001)	000001 (00, 0001)	Yes, replace Frame 0
11100001	(1110, 0001)	010010 (01, 0010)	Yes, replace 1

Pi's	Valid	Frame
0	0	
1	~~0~~ 1	~~0~~ 2
2	0	
3	0	
4	0	
5	0	
6	~~0~~ 1	~~1~~ 3
…		
14	~~1~~ 0	2
15	0	

Pj's	Valid	Frame
0	0	
1	~~1~~ 0	3
2	0	
3	0	
4	0	
5	0	
6	1	0
…		
14	1	1
15	0	

RAM	Whose Page
0	~~Pi:1~~, Pj:6
1	~~Pi:6~~, Pj:14
2	~~Pi:14~~, Pi:1
3	~~Pj:1~~, Pi:6

4 bits for page offset
8 bit virtual address
6 bit physical addresses

Pi running:	VA (page #, offset)	PA (frame #, offset)	Page Fault?
00011010	(0001, 1010)	101010 (10, 1010)	Yes (V==0) replace 2
01100100	(0110, 0100)	110100 (11, 0100)	Yes (V==0) replace 3
Pj running:			
11101111	(1110, 1111)	011111 (01, 1111)	No (V==1) F#1
01100110	(0110, 0110)	000110 (00, 0110)	No (V==1), F#0

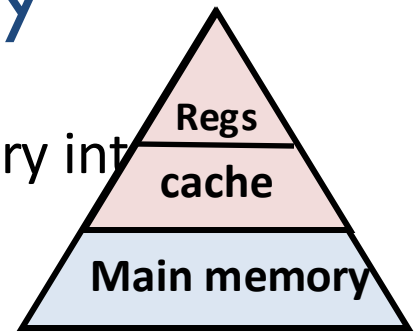
Announcements

- Thanksgiving and Assignments
  - Ninja sessions next week: Monday and Tuesday
  - Lab 9: encourage you to complete by Monday evening
  - Lab 10: assigned Tuesday next week (attend Tues Ninja session)
  - HW9: assigned tomorrow, due Tues after Thanksgiving
- Optional: you may pick partner for lab 10
  - You and your partner submit form By Friday 5pm (link in *Announcements* at top of home page)
- More Practice with Processes and Virtual Memory
  - Exercises at end of Chapt. 13
  - Early Access Interactive "Ask me another" Questions

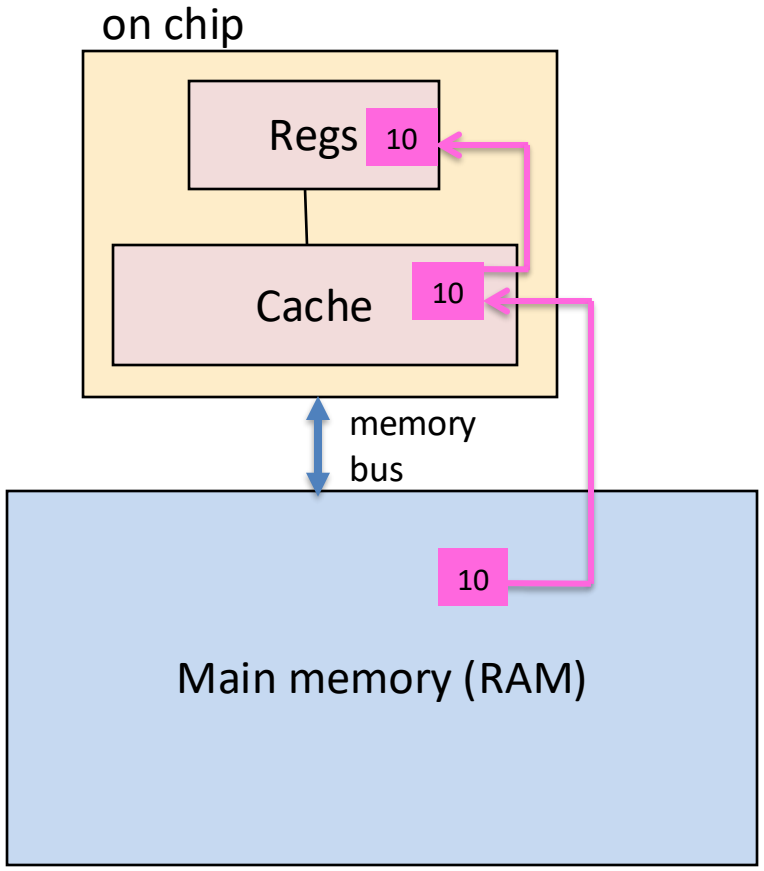
    (can generate an infinite number of questions of the same type)

Addresses & the Memory Hierarchy

```
movl -8(%rbp), %rax    # load from Memory int
```



Virtual address: value of `-8(%rbp)`



How value is read into Register:

1. Check if the value at the load address is already in the cache. If so, copy it from cache into register & done.

   Use Virtual Address for cache lookup*

2. Else, cache miss, need to read in from RAM, copy into Cache, copy into register.

   Need Physical Address to read from RAM

*common, but some HWs could use physical addresses for lookup

Example: Addresses and Memory

32 bit Virtual Address: 00000001111011001011111111100010

1. First try to find byte(s) in the cache
   (Direct Mapped, 2^{10} lines, 8 byte blocks)

Divide VA into: tag, index, byte offset

00000001111011001011 1111111100 010

(3941) (1020) (2)

Tag (19 bits)	Index (10 bits)	Byte offset (3 bits)
3941	1020	2

Cache Memory (not RAM)

Line	V	D	Tag	Data
0				
1				
2				
3				
4				
…	…			
1020	1	0	1323	
1021				
1022				
1023				

a. Use index bits to find line (1024)
b. Valid but Tag (3941) doesn't match one in cache line (1323)

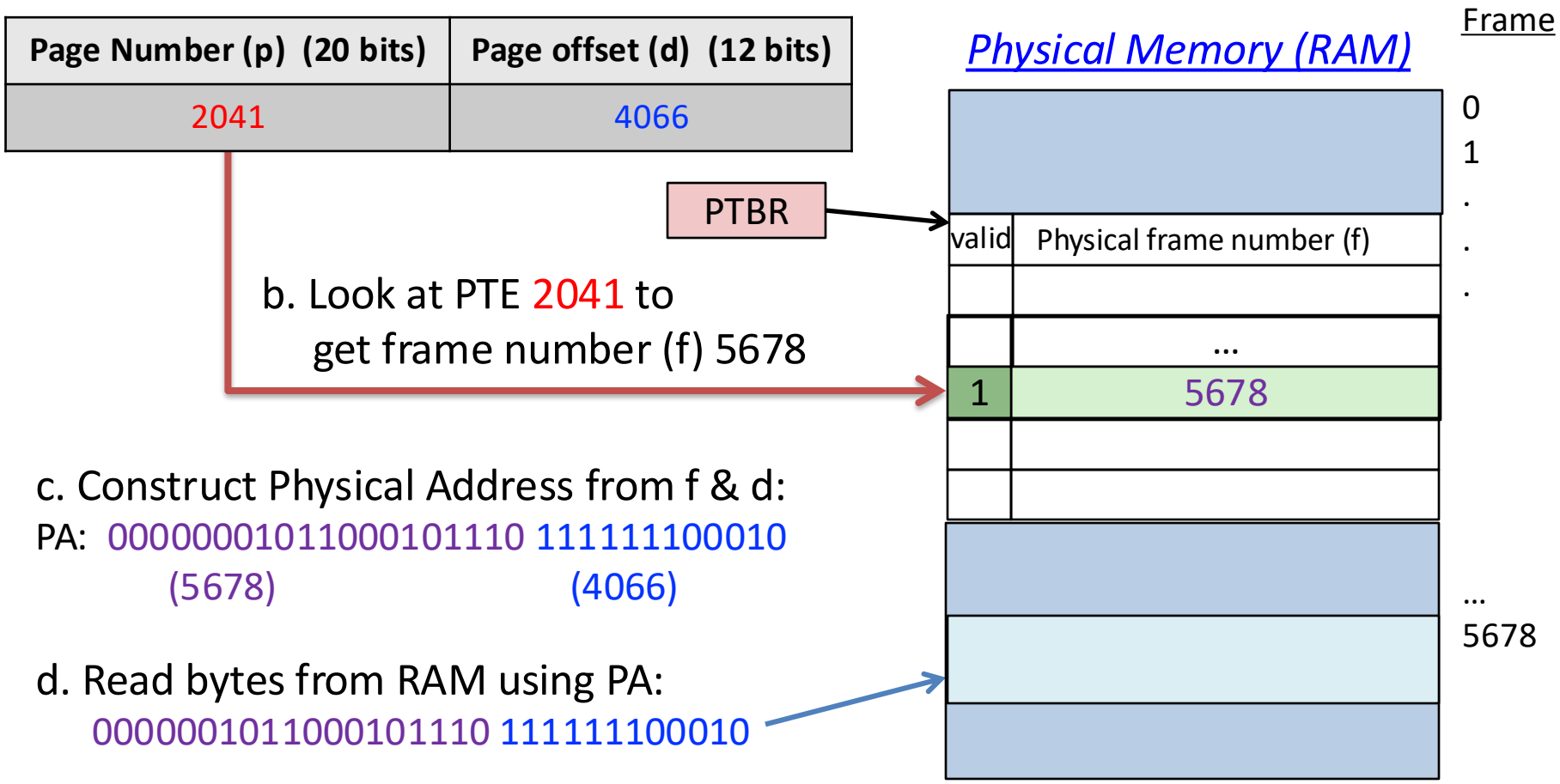
cache miss → need to read bytes from RAM (using PA)

2. Read from RAM: 1st get PA associated with this VA

a. Divide up VA bits into page number and offset (assume 4KB page size)

32 bit VA: 00000001111101100101 111111100010

(2041) (4066)



c. Construct Physical Address from f & d:

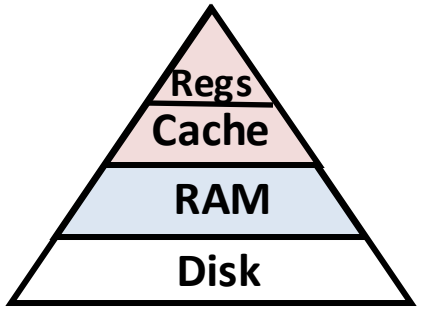
PA: 00000001011000101110 111111100010

(5678) (4066)

d. Read bytes from RAM using PA:

00000001011000101110 111111100010

Why Virtual Memory?



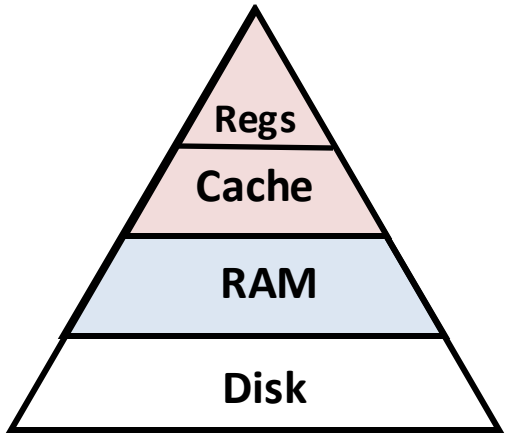
- Uses main memory Efficiently
  - + Main Memory ~a cache for the parts of virtual address spaces that are being accessed (rest can stay on disk)
  - + Pages from processes address spaces can be mapped into any Physical Memory frames
- Simplifies memory management
  - + Each process gets same uniform linear address space
- Isolates address spaces
  - + One process can't interfere with another's memory
    - MMU won't map into another P_i's address space in RAM
    - OS swaps PTBR on process context switch (P_i can only use P_i's PTE mappings)

Why Not Virtual Memory?

Memory access is now more expensive:

- Extra Memory space to store Page Tables

- Extra Address Translation costs per access
  - Two memory accesses per byte fetched from RAM:
    1 to access the PTE in RAM
    1 to access the byte of data in RAM



Can we do better than this?

Every Memory access is now more expensive:

Two memory accesses per byte fetched from RAM:

1 to access the PTE + 1 to access the byte of data

every RAM access is twice as slow!!!

Q: Can we make this faster?

Can we do something to avoid having to access the PTE in RAM on every load or store (ex. `movq $10,-8(%rbp)`) to program virtual address space stored in RAM ?

Can we do better than this?

Every RAM access is 2 times as slow with Paging!

1 RAM to access the PTE + 1 to access data byte

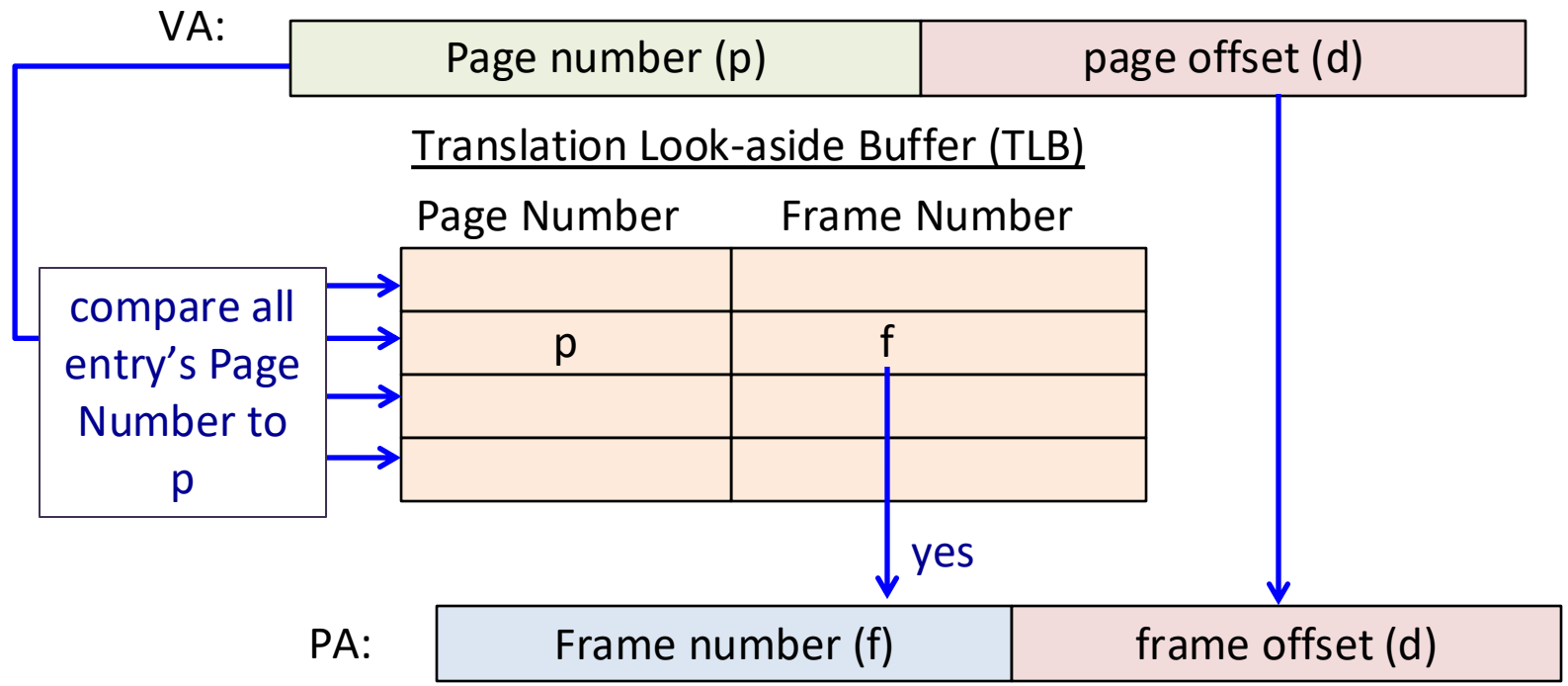
Q: What technique have we seen to make Memory accesses faster?

Q: Do we expect locality in VA to PA?

A: Cache VA to PA Translations (Mappings)

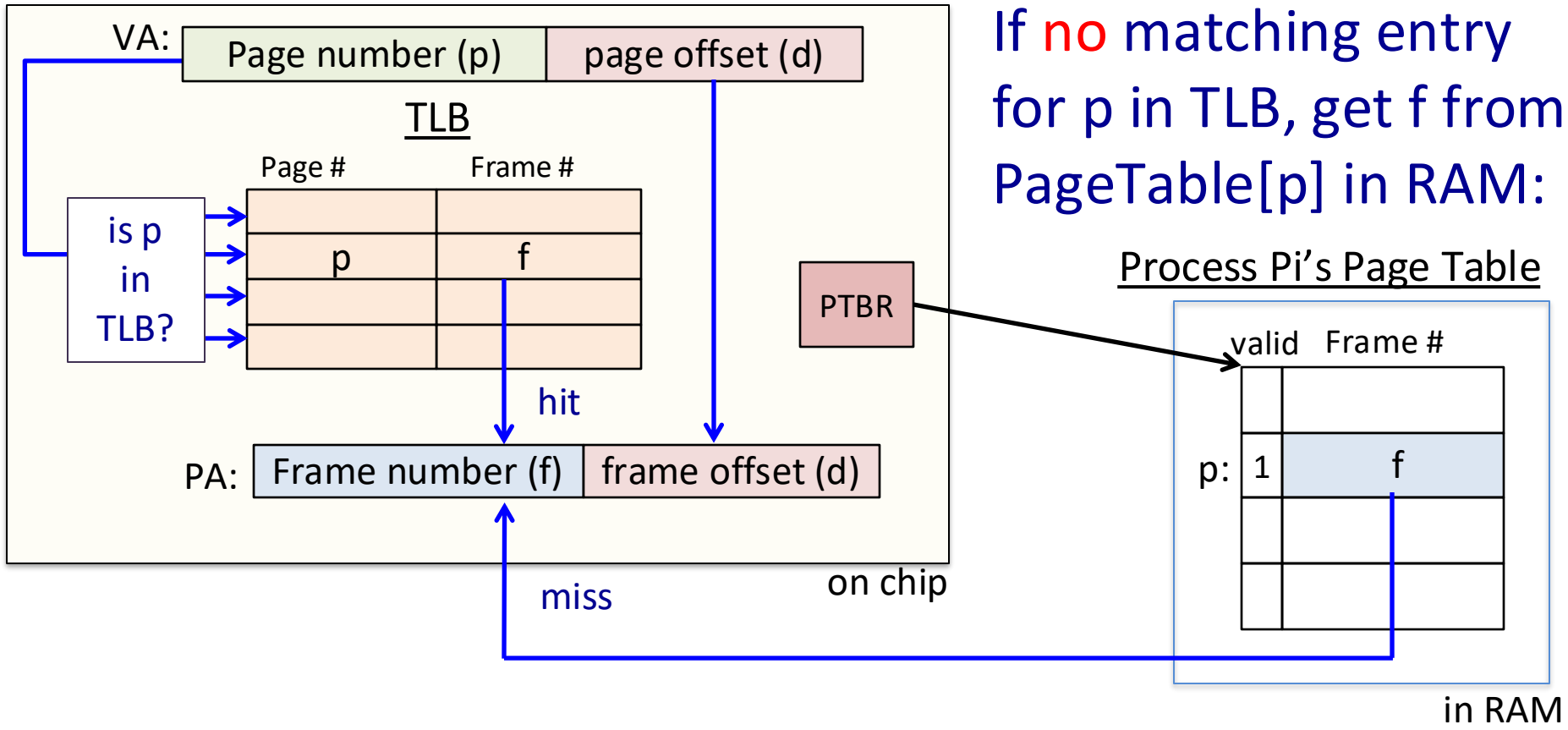
- Cache recent translations of Virtual Page Number to Physical Frame Number
- If Cache contains a translation, don't need to access PTE in RAM to translate VA to PA!

TLB: Translation Look-aside Buffer



- Fast, small HW cache that keeps most recent page# to frame# translations
  - Fully associative hardware lookup
  - On hit: get Physical Frame number from cache vs. PTE
  - On miss: get PTE from RAM, cache translation in TLB

TLB: Translation Look-aside Buffer



If no matching entry for p in TLB, get f from PageTable[p] in RAM:

- Only on a TLB miss do need to go to RAM to access the PTE to get the page# to frame# mapping

VM Summary

- Operating System implements Abstraction of Virtual Memory
  - Part of implementing the “Lone View”
  - Makes more Efficient use of Memory
- Paging is typical implementation
  - Mapping Virtual to Physical Addresses
  - Page Tables with each Process
  - Need some HW support
    - PTBR stores base address of a Pi’s Page Table in memory
    - OS saves/restores value on CXS
- Trade-offs in choosing VM or not

  +’s of VM usually way outweigh the –’s