Agenda

Virtual memory

- Mapping virtual to physical addresses, MMU
- Paging
- Page Tables
- Page Replacement Algorithms

Virtual Memory

Problem: So far, a process is either entirely in main memory or entirely on backing store (i.e., swapped in or swapped out).

- Huge Limitation: The entire program must fit into a contiguous block of memory

Solution: Give each Process has its own **virtual address space (VAS)**

- VAS Memory: large and contiguous
- Physical Memory: limited and non-contiguous

Virtual Memory

Abstraction
(the addresses from Assembly code executed by CPU)

- Virtual Address Space (VAS): 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 `0x1234`
    - P2's virtual address for x is `0x1234`

Reality
(addresses to access RAM)

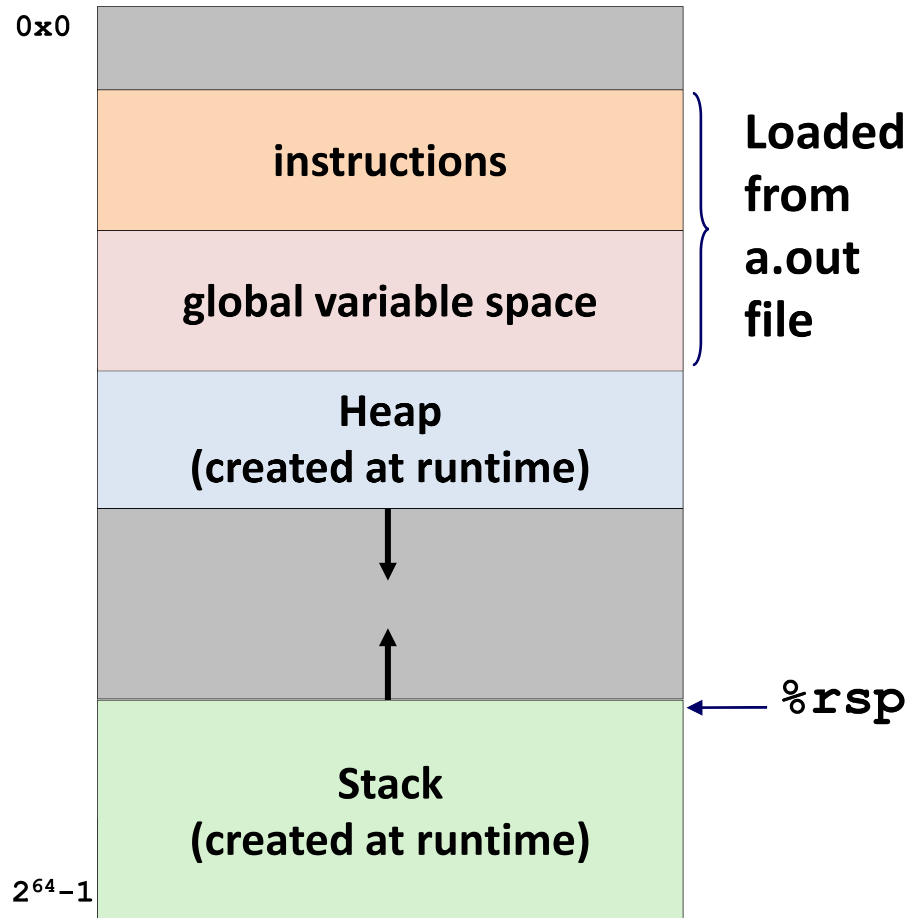
- Virtual Address mapped to different Physical addresses (in RAM)
    - P1's virtual address `0x1234` maps to different physical address in RAM than P2's virtual address `0x1234`

- 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)

Recall: On fork, OS:

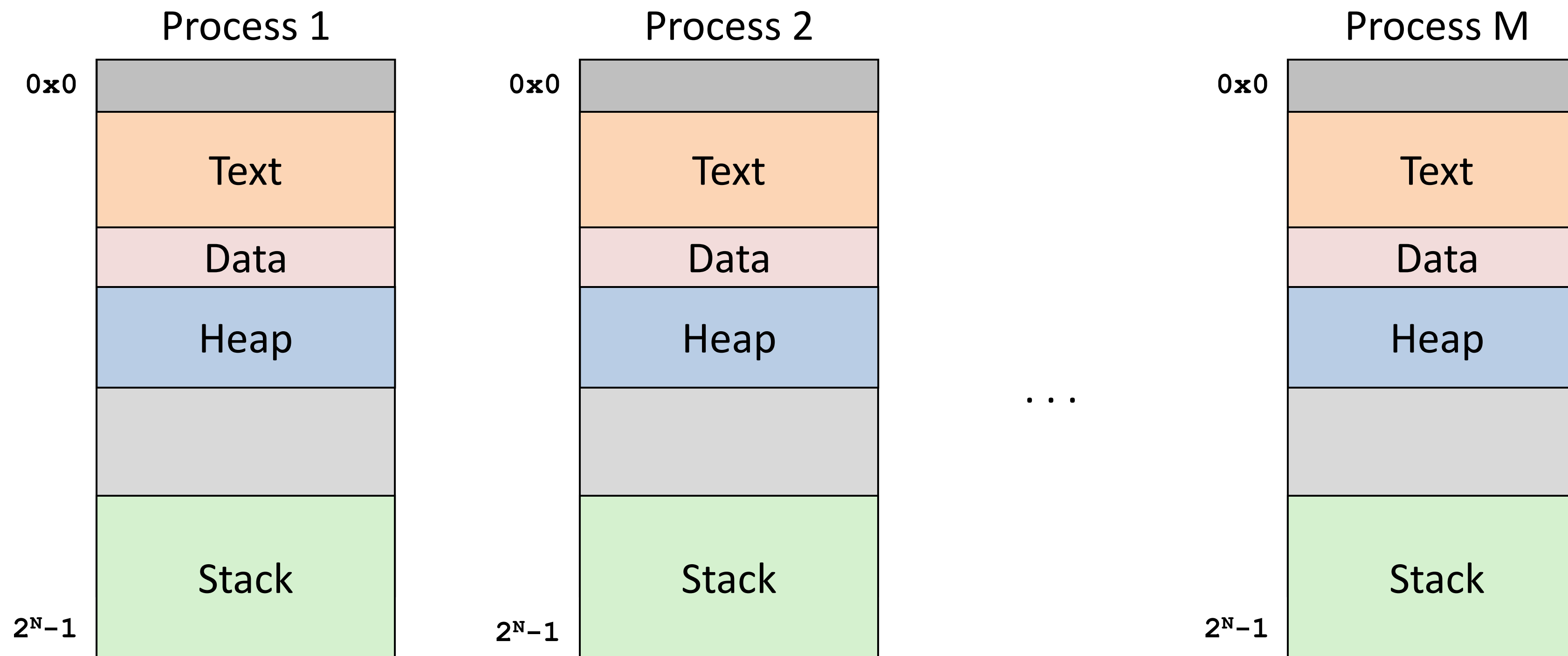
- Creates copy of parent's process and uses it to initialize the child process.
  - This corresponds to copying the parent VAS to the child VAS
- OS keep track of child's VAS



Process's virtual (logical) Address Space

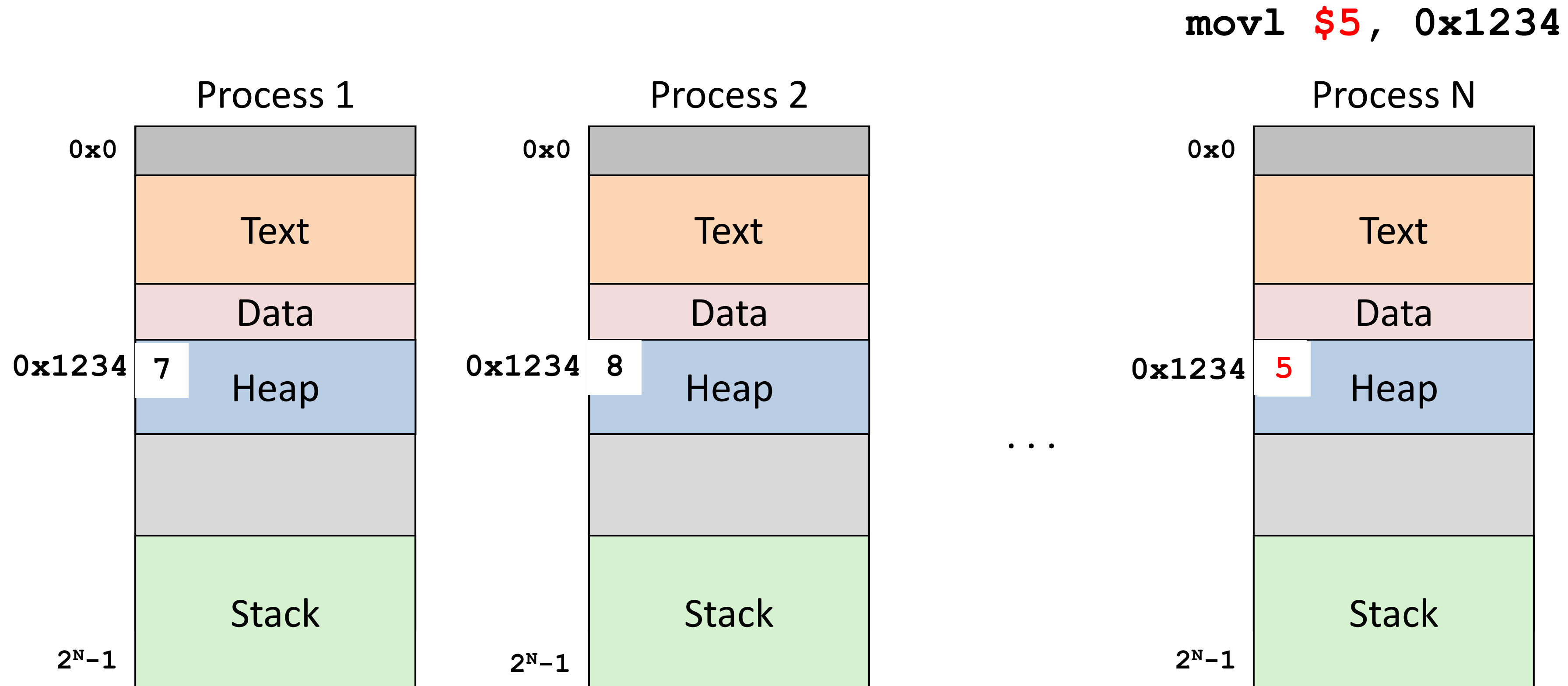
Every process has the same virtual memory layout

– 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!)



Every process has its own physical memory

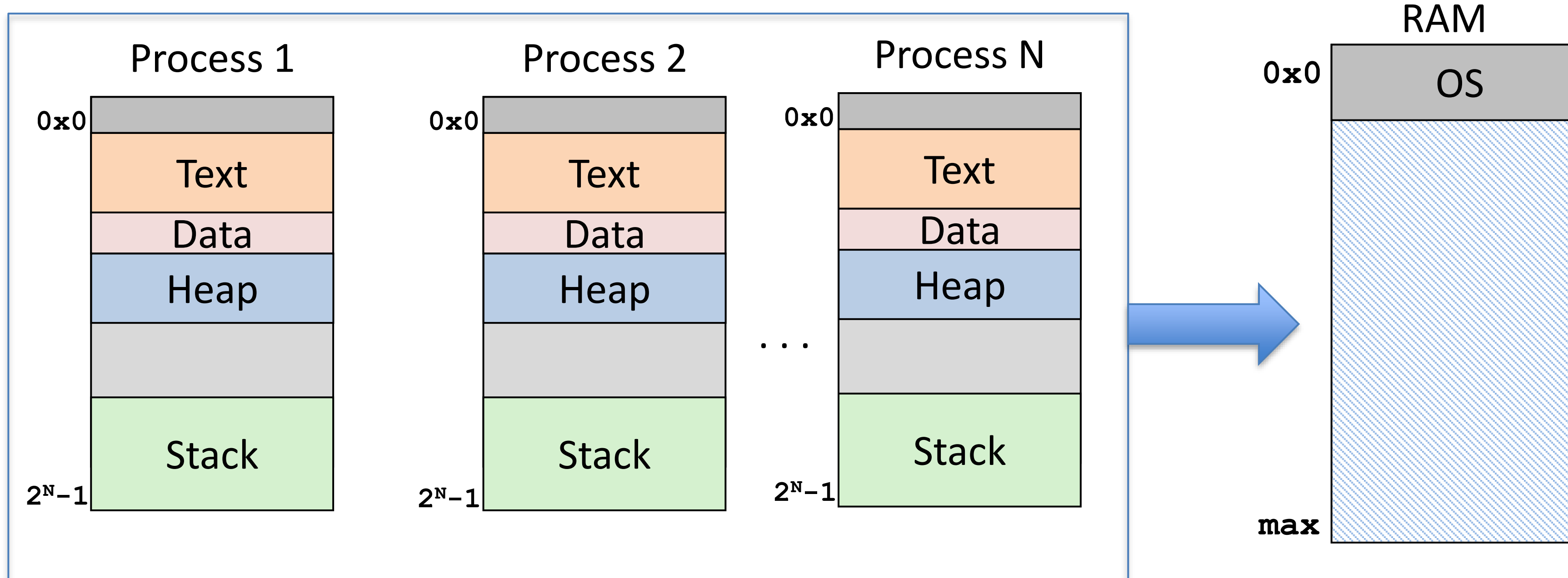
- One process could modify a value at a particular virtual address (ex 0x1234), and it will not interfere with any other process' values stored at the same address in their virtual memories.



Physical Memory (RAM): Reality

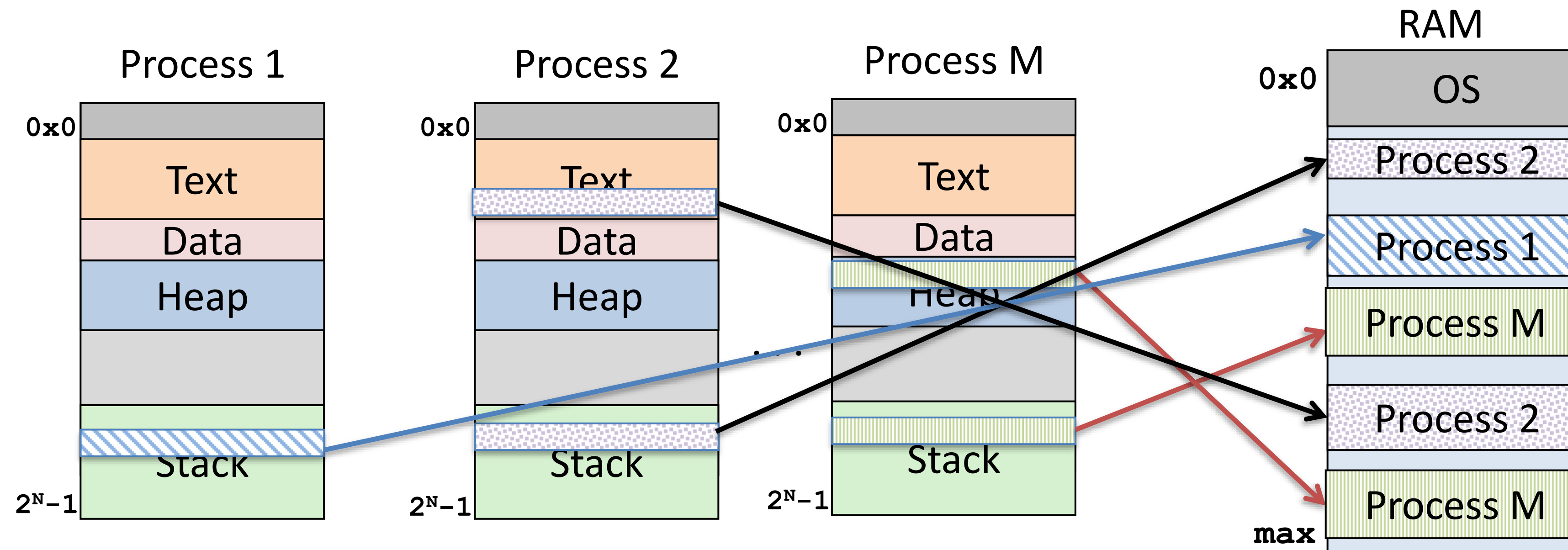
RAM is array of addressable bytes, from 0x0 to max

- 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

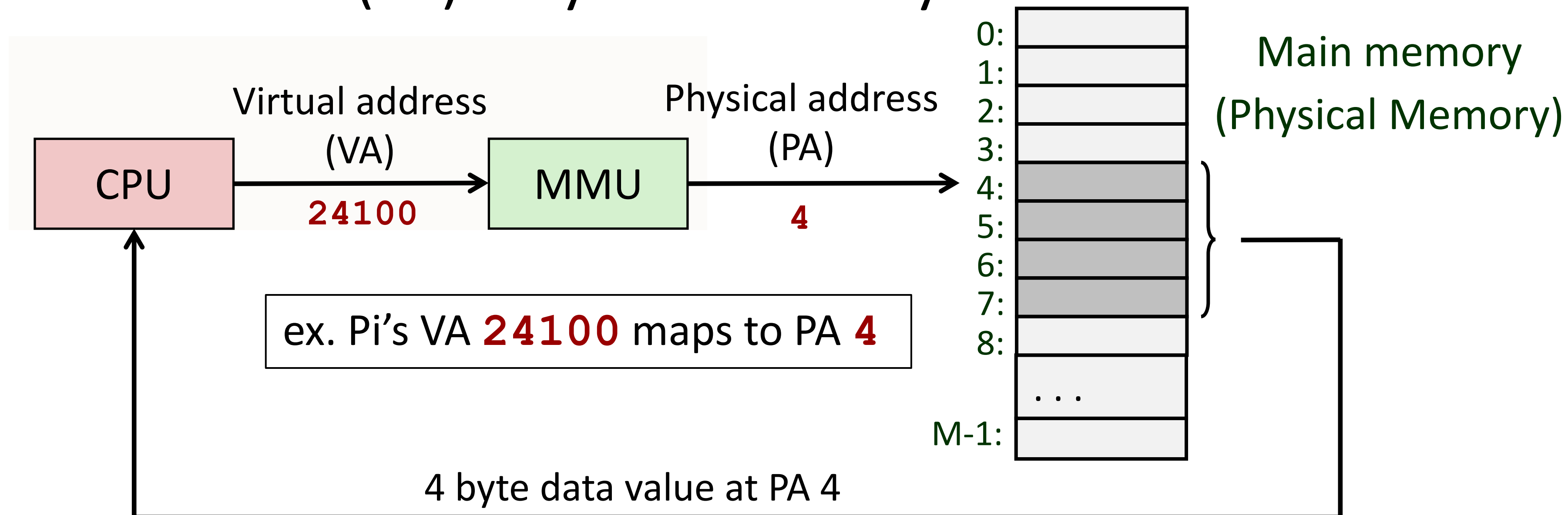
- Process' Virtual Addresses map to different Physical Addresses
- RAM cannot necessarily store even one full process' VAS
- Parts of Process' VAS must be stored at different RAM addrs



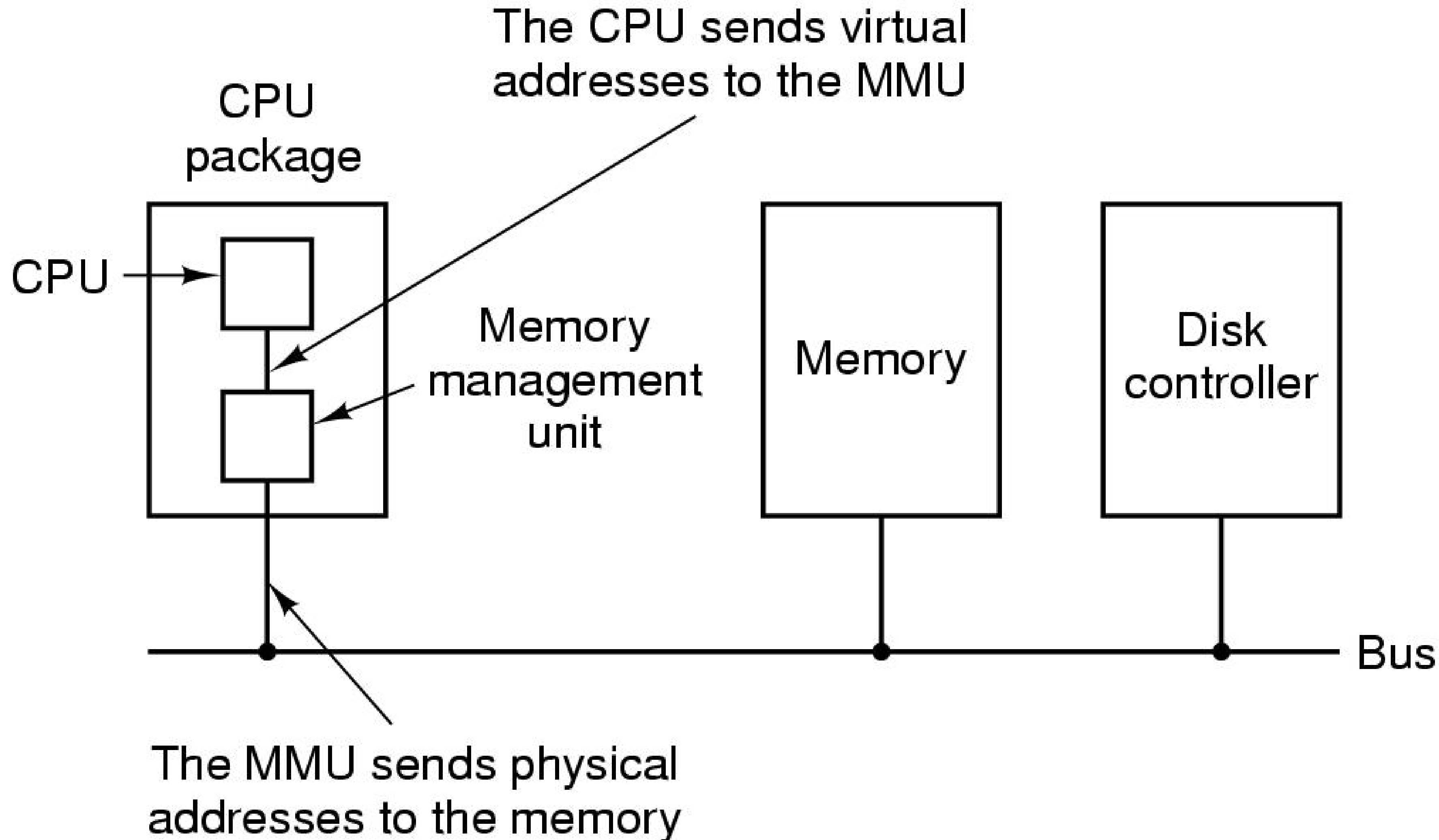
Each Pi's veiw of its memory (virtual/logical address) gets mapped to reality (physical addresses)

Mapping Virtual to Physical Addresses

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



The MMU is part of the CPU



Can you think of some challenges to creating a good MMU?

Virtual Addresses

Virtual memory gives a complete separation of logical and physical addressing spaces

Typically a virtual address is 32-bit or 64-bit

- Processes can have 4GB/256 TiB (current x86_64 manufacturer max) of virtual memory
- The VAS of multiple processes typically will not fit into memory (RAM)

A **page** is a fixed-size, contiguous chunk of *virtual* memory

- Using pages, only part of the process need be in memory. Furthermore, the part of the VAS in memory do not need to be contiguous

A **page frame** is a fixed-size, contiguous chunk of *physical* memory

- The MMU needs to map between pages (virtual memory) and page frames (physical memory)

Paging

Paging is the process of swapping pages into/out of memory

Pages and page frames are the same size

Transfers between memory and disk are in page-sized units

A **page table** stores the relationships between virtual and physical pages

A **page fault** occurs when we try to access a page that is not loaded into memory.

In this case, we must fetch the missing page (and possibly swap an existing page out)

Review: Working with powers of 2

1 KB = 1024 Bytes = 2^{10} = 0x400

Means we can store a total of 2^{10} values, ranging from 0 to $2^{10}-1$

2 KB = 2048 Bytes = 2 * 2^{10} = 2^{11} = 0x800

4 KB = 2096 Bytes = 2^2 * 2^{10} = 2^{12} = 0x1000

1 MB = 1024 * 1024 Bytes = 2^{20} = 0x0010 0000

1 GB = 1024 * 1024 * 1024 = 2^{30} = 0x 4000 0000

1 TiB = 1024 * 1024 * 1024 * 1024 = 2^{40} = 0x 0100 0000 0000

Exercise: How many bytes are 8 KB? Express the number of values that can be expressed with 8KB, as a power of two and as hexadecimal number.

NOTE: In decimal, the values are less than with base 2: Mega is 10^6, Giga is 10^9, Terra is 10^{12}

Example: Paging

Suppose we have

- 16-bit addresses.
- 32 KB of physical memory
- 4KB page sizes.

What is the possible range of virtual addresses?

How many pages do we have?

How many page frames do we have?

Example: Paging

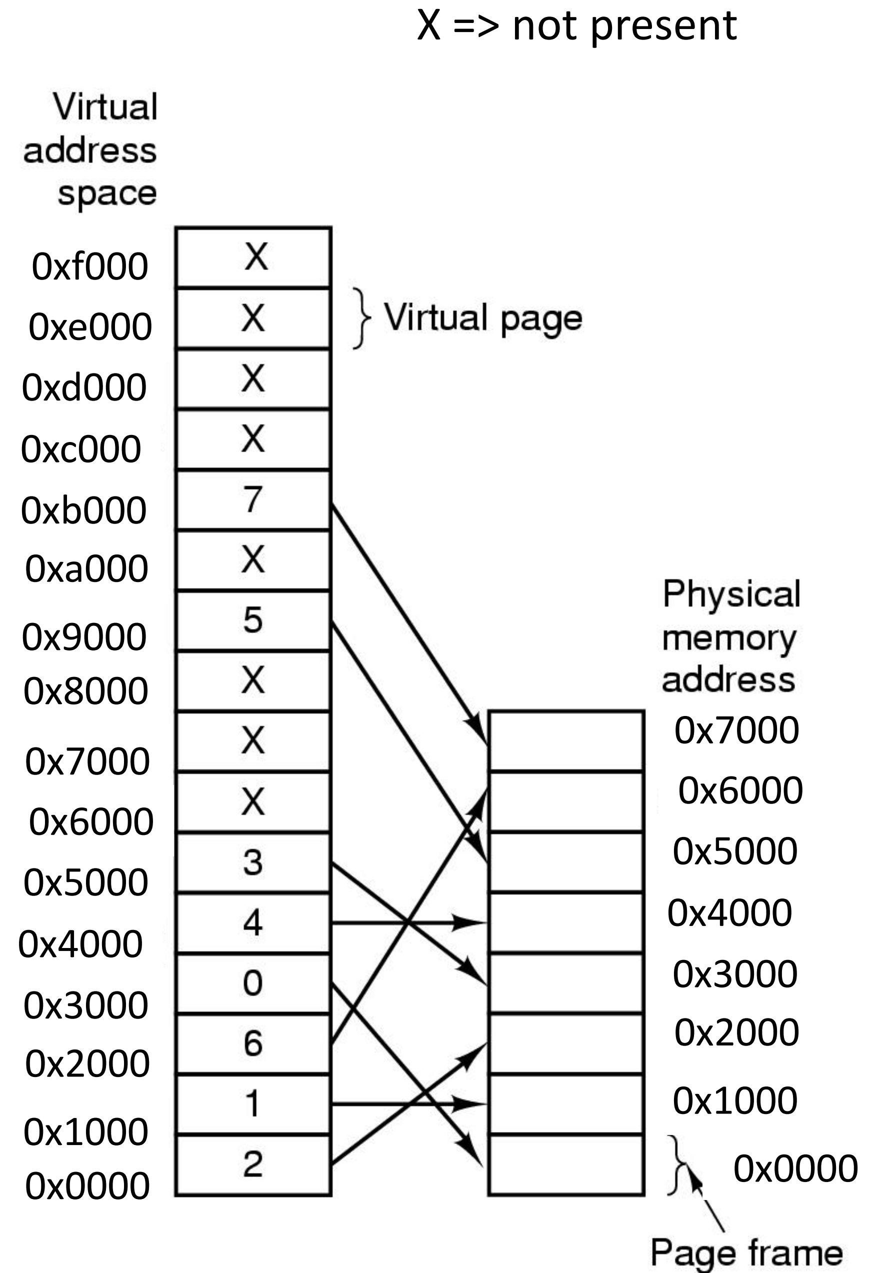
Convert the following addresses:

MOV REG, 0x0000

MOV REG, 0x2000

MOV REG, 0x5014

MOV REG, 0x3128



Converting from virtual to physical addresses

A virtual address is really a pair (*p*, *o*) of addresses

- e.g. (pair, offset)
- Low-order bits give an offset within the page
- High-order bits specifies page number – this is the part that gets translated

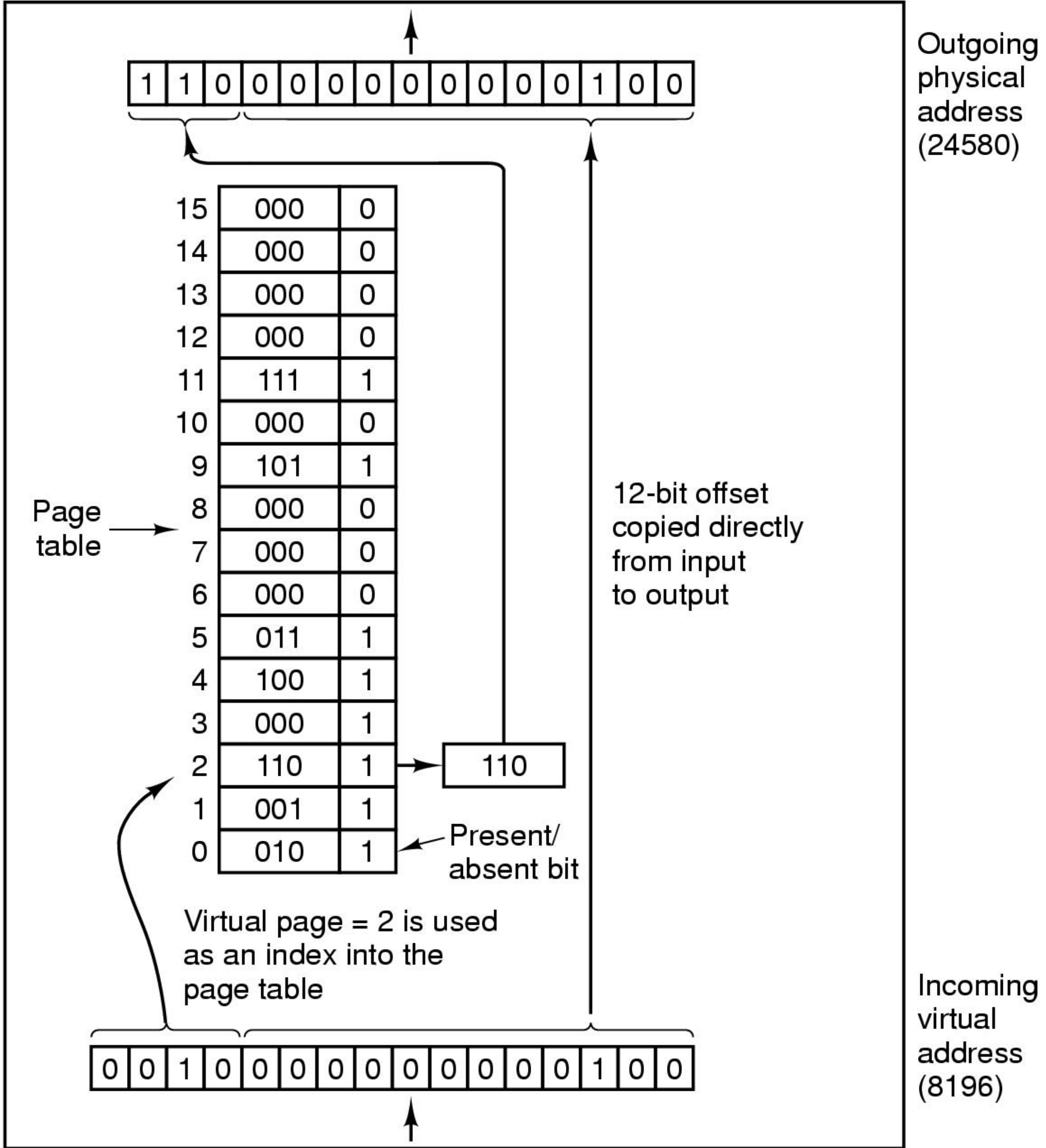
The MMU translates the page number *p* to a frame number

- The physical address is then (*f*, *o*) – this is what goes on the memory bus

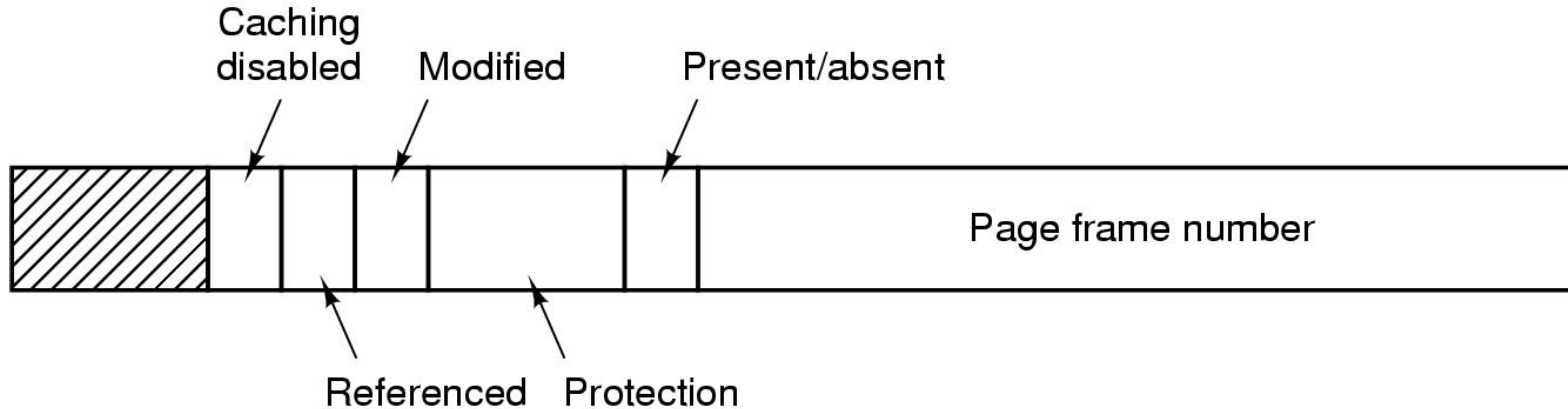
p is an index into the page table for the translation

Example: Page Table

Convert address 8196 (0x2004)



Aside: a typical page table entry



- Page frame number: Page in physical memory
- Present/absent: Whether the page is already loaded in memory or not
- Protection: 3 bits to indicate that type of access permitted, read/write/execute
- Modified: Whether the page needs to be written back to disk, sometimes called the **dirty bit**
- Referenced: Whether the page has been used by a process. E.g. has someone read or written to it?
- caching disabled: option for niche circumstances

Page Table Challenges

- The mapping must be fast
  - virtual-to-physical mapping is computed how often?
- The page table can be extremely large
  - Modern computers use virtual addresses of at least 32-bits – with 4KB page size, this gives $2^{20} \approx 1$ million pages
  - Recall that each process keeps its own page table!

Speed: Translation Lookaside Buffers (TLB)

Insight: Only a small fraction of pages are frequently read

Use hardware to make frequent lookups fast (without page table)

Keep full page table in memory (RAM) for all other lookups

The **translation lookaside buffer (TLB)** is a small hardware device for mapping virtual addresses to physical addresses without going through page table.

The TLB checks all entries in parallel.

When a page is not in the TLB, one of the entries is evicted and replaced with the new page mapping. If the page is modified, the modified bit is also written back to memory.

Example – Translation Lookaside Buffer

Suppose this table was generated by executing a loop

- Loop: 19, 20, 21
- Array Data: 129, 130
- Array calculations: 140
- Stack: 860, 861

Suppose we query the table for page 131 (mapped to page frame 63). It's not there.

Replace entry 129 with 131.

Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

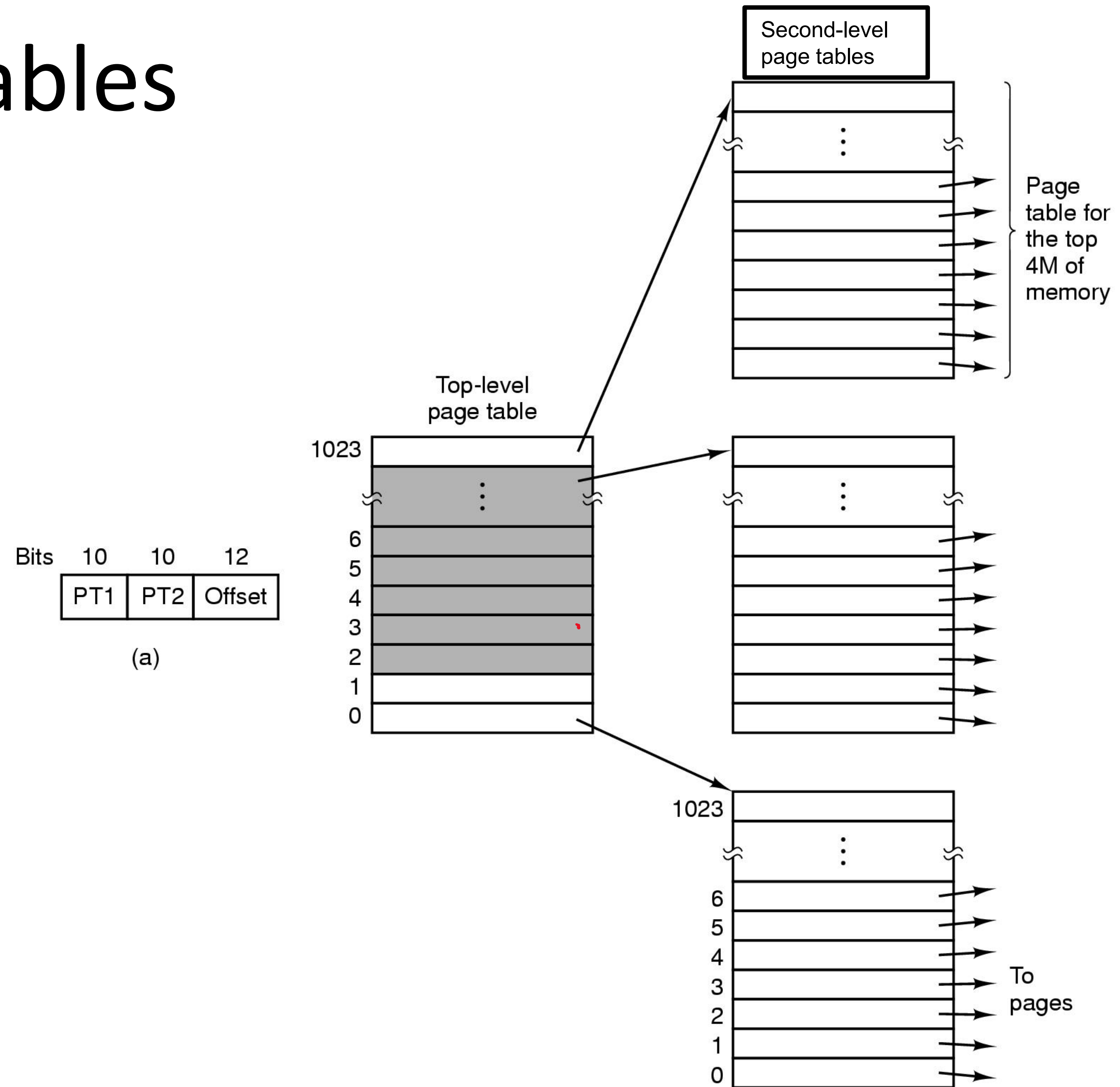
Size: Multilevel Page Tables

Idea: Use multiple levels of page table. Allocate second-level page tables only as needed.

Example: Suppose we have 32 bit addresses. 4KB page sizes.

How many virtual pages?

How many bits are needed for 4KB page offsets?



Example: Multilevel Page Addressing

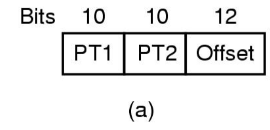
Suppose we have 32-bit address, 4 KB page sizes, and the same two-level page tables as the previous slide. Compute the indices into the page tables and offset for the following addresses:

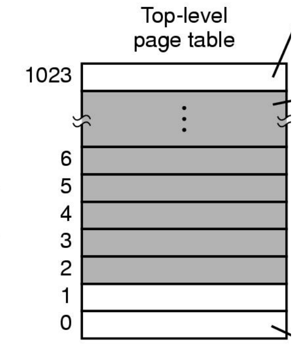
0x00403004

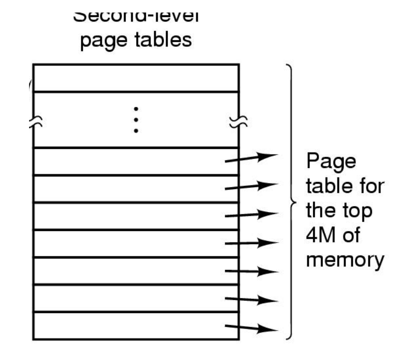
0x00c0500a

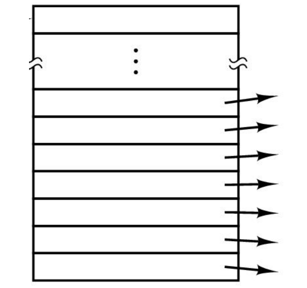
Example: Multi-level Page Addressing

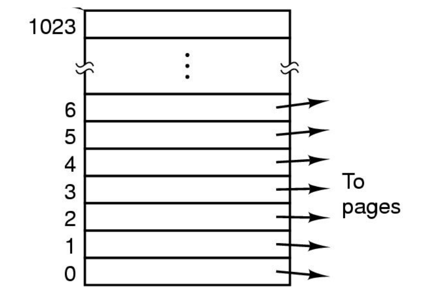
Draw the tables used for address:
0x00403004





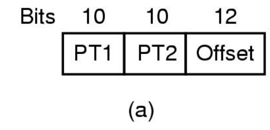


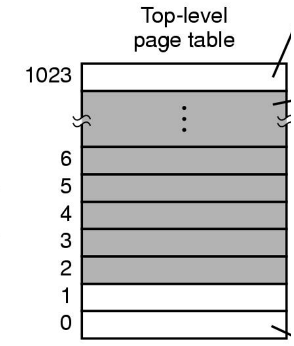


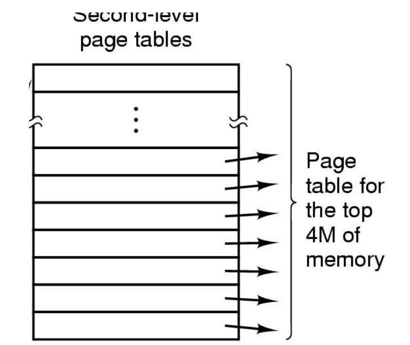


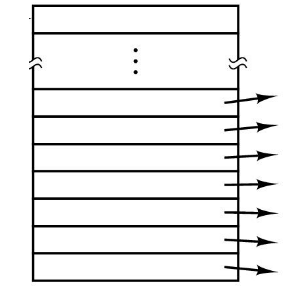
Example: Multi-level Page Addressing

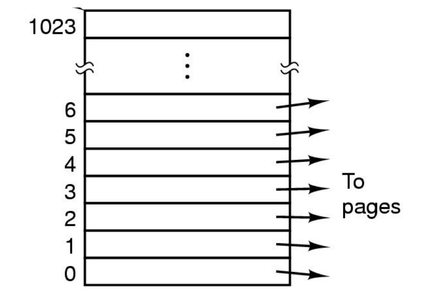
Draw the tables used for address:
0x00c0500a











Size: Inverted Page Tables

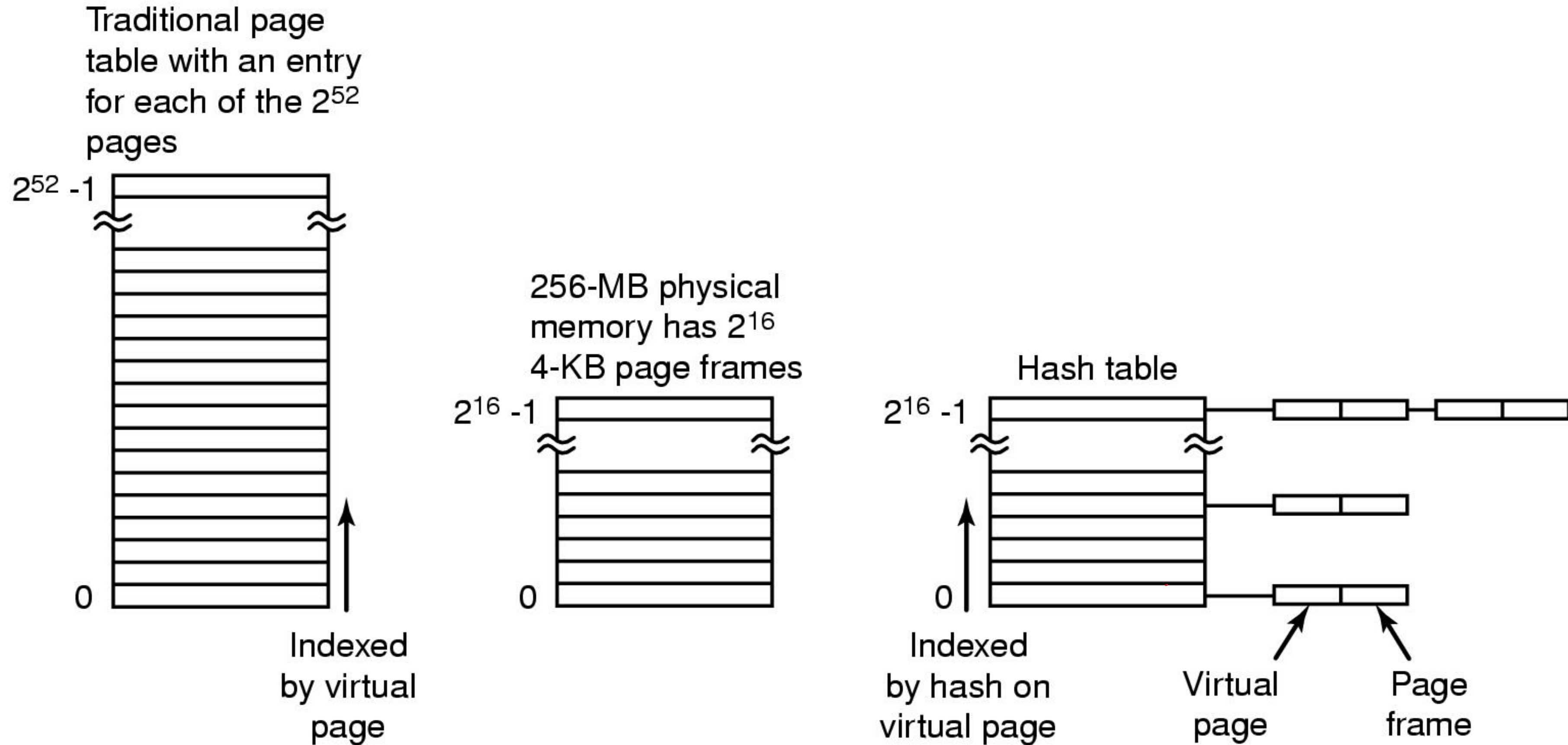
Idea: Index based on page frames, not virtual pages

Since physical memory is much smaller than virtual memory, we use much less space

But translating from virtual to physical pages is slow

But that's ok if most of the time, entries are in the TLB

Example: Inverted Page Tables



Comparison of a traditional page table with an inverted page table

Page Replacement Algorithms

Recall: When a virtual address translates to a page not currently in memory, it generates a **page fault**

Page fault forces choice

- need to make room for incoming page
- which page must be removed?
  - Ideally, not one that will be needed again soon!

Modified page must first be saved

- unmodified just overwritten

Optimal Page Replacement Algorithm

What would the ideal page replacement algorithm be?

Algorithm #1: Not Recently Used (NRU)

Idea: Use the Reference/Modified bits to check if a page has been used recently

Q: When are the R and M bits set?

How it works:

- When the process starts, the OS sets R = 0, M = 0
- Periodically, the R bit is cleared, say on clock interrupt
- When a page fault occurs, the OS categorizes each page
  - Class 1: not referenced, not modified
  - Class 2: not referenced, modified
  - Class 3: referenced, not modified
  - Class 4: referenced, modified
- Remove a random page from the lowest numbered non-empty class

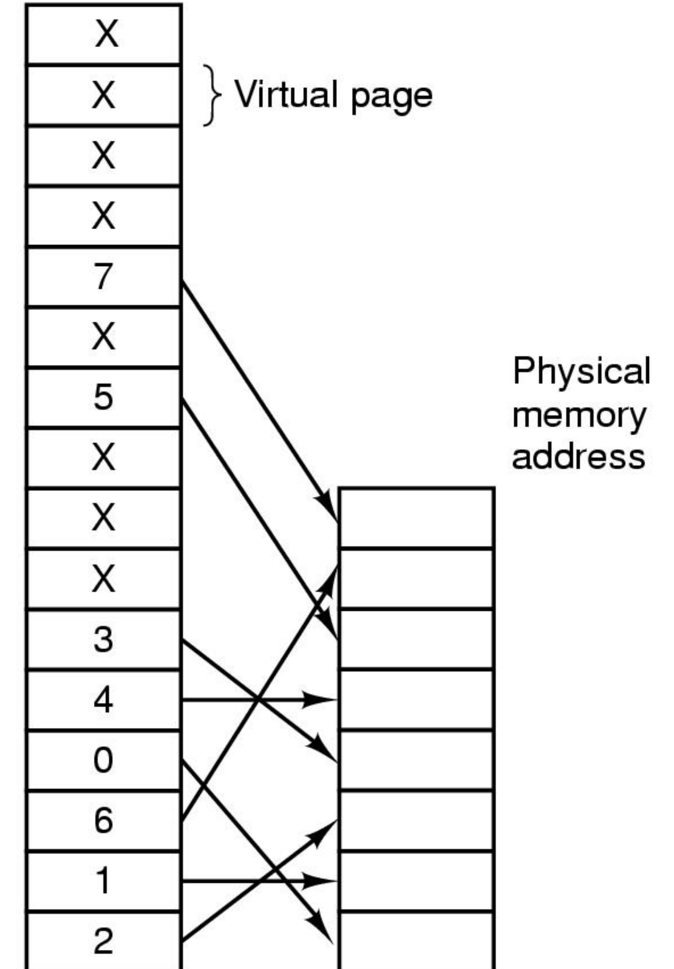
Exercise: Not recently used

- 16-bit addresses
- 64 KB virtual memory
- 32 KB Physical memory
- 4 KB page size

```
mov 0x0, reg
mov $0x19, 0x2000
clock interrupt
mov reg, 0x90a3
mov 0x5014, reg
```

1. Fill in the address boundaries of each page in hexadecimal
2. Assume all R/M flags are zero to start. Show how the flags would update based on the instructions above. Which instructions would lead to a page fault?
3. For each accessed page, what category is it in?
4. Give a page that would be removed by NRU

Virtual address space



Algorithm #2: FIFO

Idea: Avoid need to look through entire page table by keeping pages in a linked list

How it works:

- When a page is loaded for the first time into memory, put it at the end of the list
- When a page fault occurs, replace the page at the beginning of the list

Disadvantage: the first page, e.g. the one that has been in memory the longest, might be a frequently used page

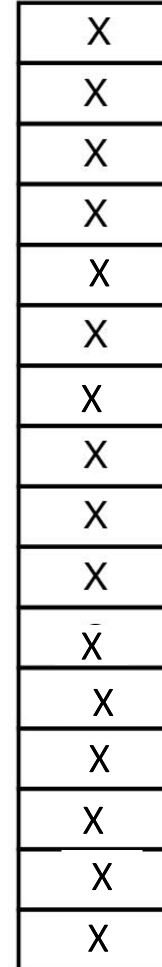
Exercise: FIFO

- 16-bit addresses
- 64 KB virtual memory
- 32 KB Physical memory
- 4 KB page size

```
mov 0x0, reg
mov $0x19, 0x2000
clock interrupt
mov reg, 0x90a3
mov 0x5014, reg
```

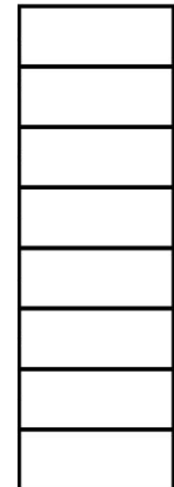
Assume that all page frames are in use. The FIFO linked list of frames is 7, 0, 2, 4, 1, 3, 5, 6. Show what pages would be assigned based on the above commands.

Virtual address space



Virtual page

Physical memory address



F

Algorithm #3: Second Chance

Idea: Improve FIFO to avoid paging out heavily used pages

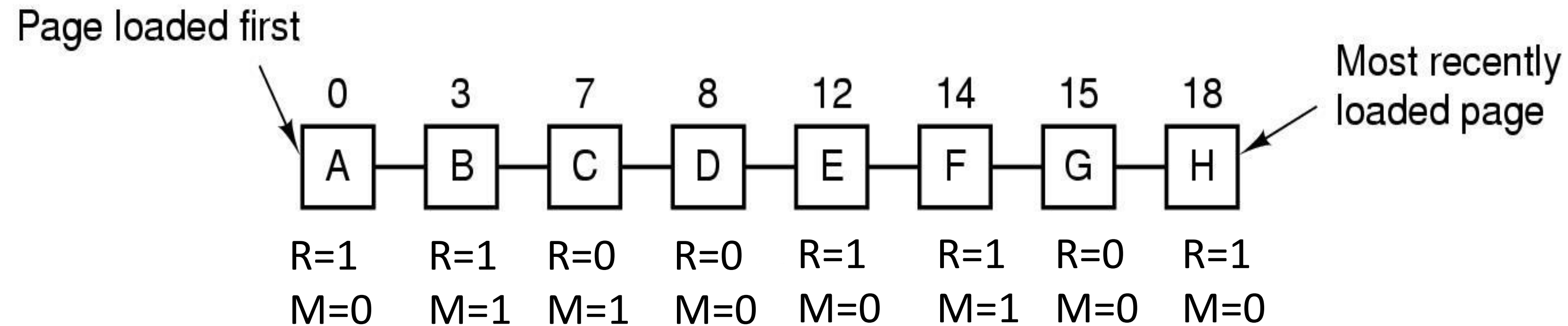
How it works:

- Check the R bit of the oldest page
  - If R = 0, the page is old AND unused so evict it
  - If R = 1, set R = 0, update load time to now, and move the page to the end of the list

Note that if all pages have R = 1, the performance is the same as FIFO

Example: Second Chance

Suppose a page fault occurs at time 20. Show the new list.



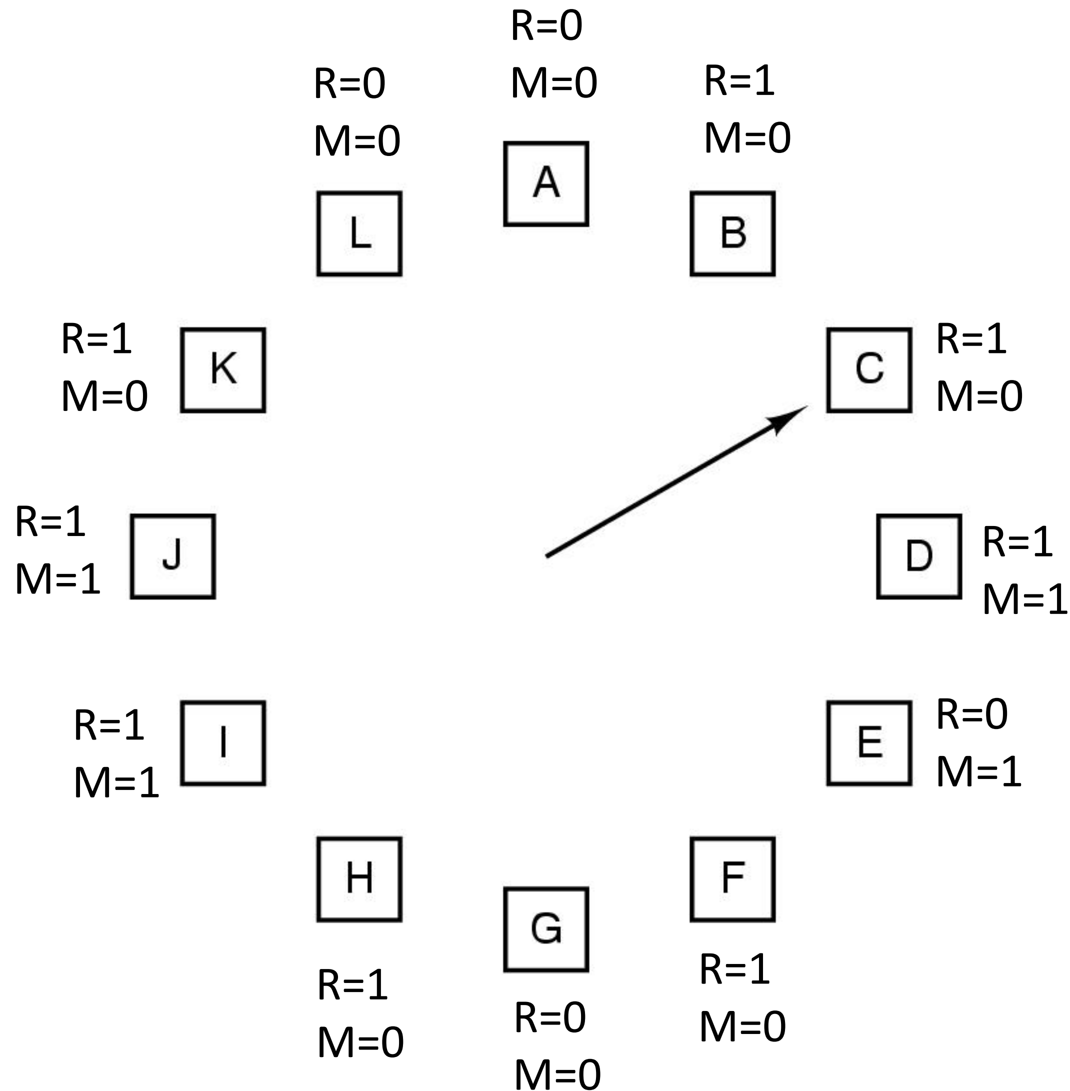
Algorithm #4: The Clock

Idea: Improve FIFO/Second Chance to avoid updating a large list of pages

How it works:

- Use a circular queue, imagine that it arranges the pages like the numbers of a clock
- Keep a pointer to the oldest page. When a page fault occurs
  - If R = 0, replace page and move hand forward
  - If R = 1, set R = 0 and advance the hand forward
- If all pages have R = 1, evict first page

Algorithm #4: The Clock



Suppose a page fault occurs.

Show how the clock would change.

What page would be evicted?

Algorithm #5: Least Recently Used (LRU)

Insight: Pages that were recently used in the past are likely to be used in the future.

How can we keep track of which pages have been used recently?

- Idea: Update linked list so that frequently used pages are at the end? (too slow)
- Idea: Keep counts of references? **Not Frequently Used (NFU)** algorithm
  - At each clock interrupt increment the counter based on the R bit

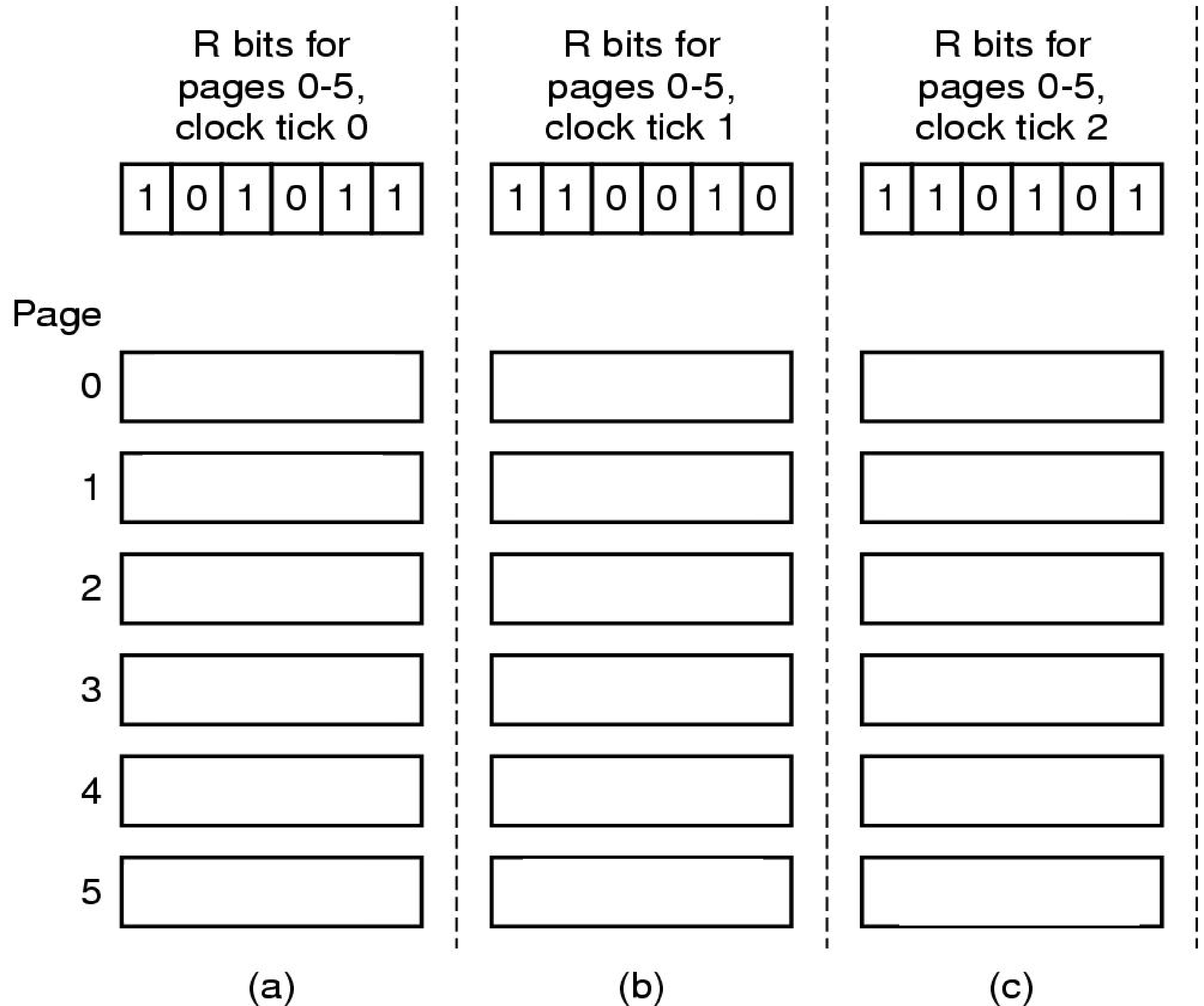
Exercise: NFU

Suppose we have 6 pages.

Each column represents a clock tick.

Update the page counts for clock ticks 0, 1 and 2

What page has been referenced the most?



Not frequently used

What could possibly go wrong?

Example: Suppose we are working with 3 pages

- Page 0 was accessed 1000 times between clock ticks 0 and 1000
- Page 1 was accessed 100 times between clock ticks 500 and 2000
- Page 3 was accessed 50 times between clock ticks 2000 and 2500

Least recently used

Idea: Modify NFU to apply aging to counts

How it works:

- Each clock tick
  - Age counts by shifting them right (this divides counts by 2)
  - Set most-significant bit to R bit
- On page fault, evict page with lowest count

Example: Suppose the current count for page 1 is 0x28 and its referenced bit was set to 1 during the previous clock tick. What will the new count be?

Exercise: LRU

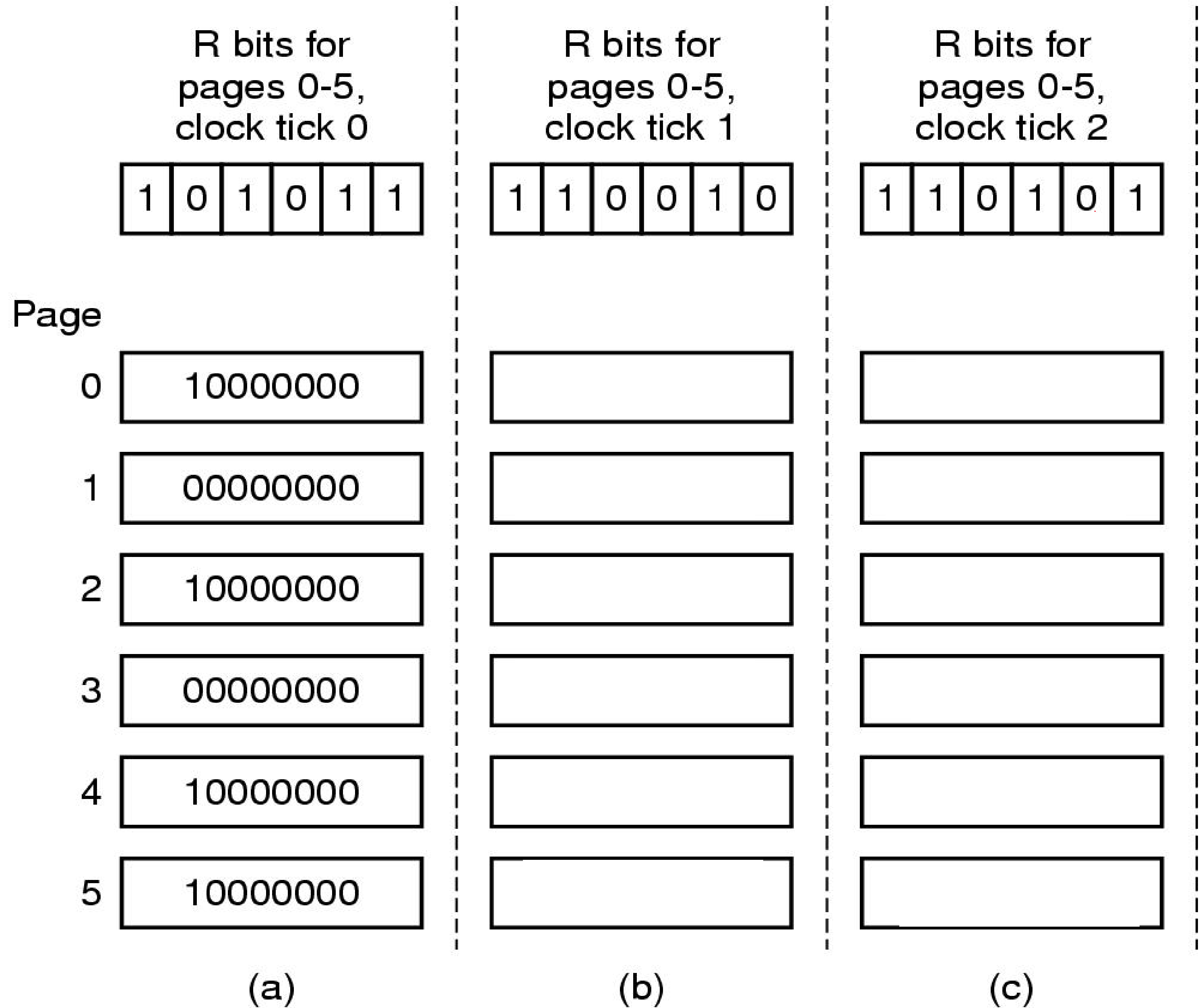
Suppose we have 6 pages.

Each column represents a clock tick.

Update the page counts for clock tick 1 and 2

Which page has the lowest counts?

page



Algorithm #6: The Working Set

Most systems implement **demand paging** , meaning that pages are only loaded as they are needed.

Demand paging is efficient because processes exhibit a **locality of reference**, e.g. only a relatively small fraction of its pages are in use at one time

Aside: If memory is too small to hold the entire working set, many instructions will lead to page faults. This is called **thrashing**.

But what if we knew the set of pages a process is likely to use? Then we could pre-load these pages when the process starts (called **prepaging**)

- We would greatly reduce the number of page faults

Working Set Model

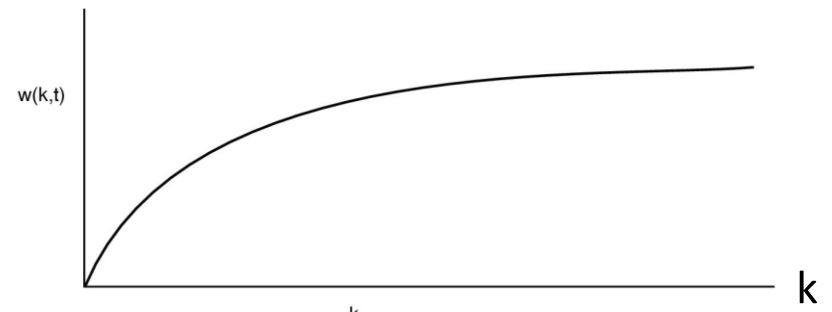
Goal: Leverage fact that processes do not reference their memory space uniformly. Instead references cluster in a small number of pages

- References tend to cluster on a small number of pages: data, instruction and stack

The **working set** is the set of pages currently in use by a process.

For any time *t*, let *w*(*k, t*) be the set of pages used by the *k* most recent memory references.

- monotonically nondecreasing function of k
- varies slowly with time



The Working Set Page Replacement Algorithm

Estimating the working set for a process

- Idea: keep track of all pages used in the last *k* memory references? (too expensive)
- Idea: record pages used in the last time interval (easier, simpler)
  - Use the **current virtual time**, e.g. the amount of time a process has spent running on the CPU

How it works:

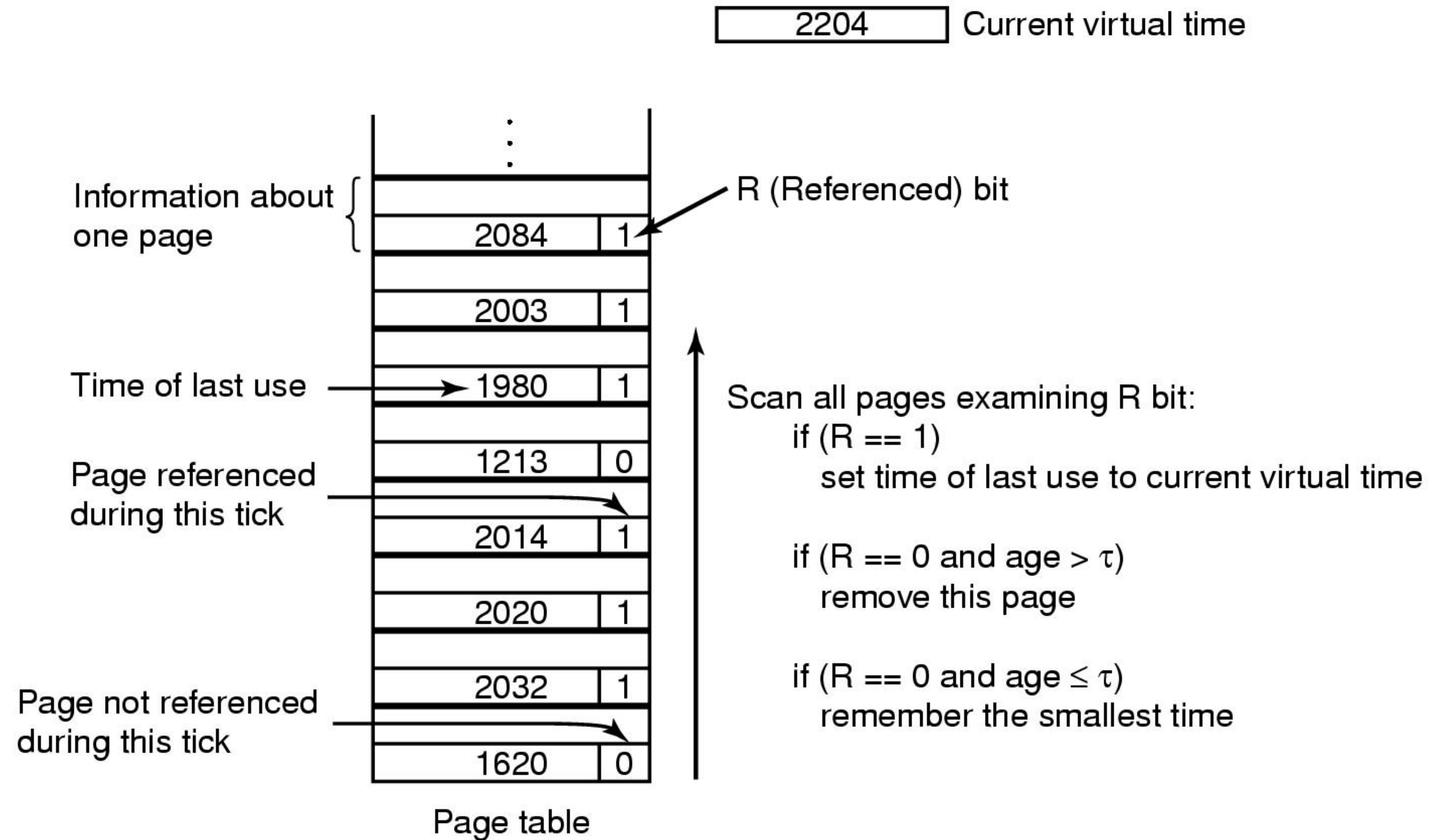
- Let τ be the time interval
- On each page fault, scan the page table
  - If R = 1, update `time_of_last_use` (tolu) based on the current virtual time of the process
  - If R = 0, compute `age = current_time – time_of_last_use` and compare to τ.
    - If age > τ, replace this page with the new page
    - If age <= τ, keep looking.
  - If no page is found, evict the page with the R= 0 and smallest `time_of_last_use`
  - If still no page is found, evict a random page
- Clock interrupt sets R to 0

Example: The Working Set Page Replacement Algorithm

Suppose tau = 800.

Update the diagram according to the algorithm

What page will be evicted next?

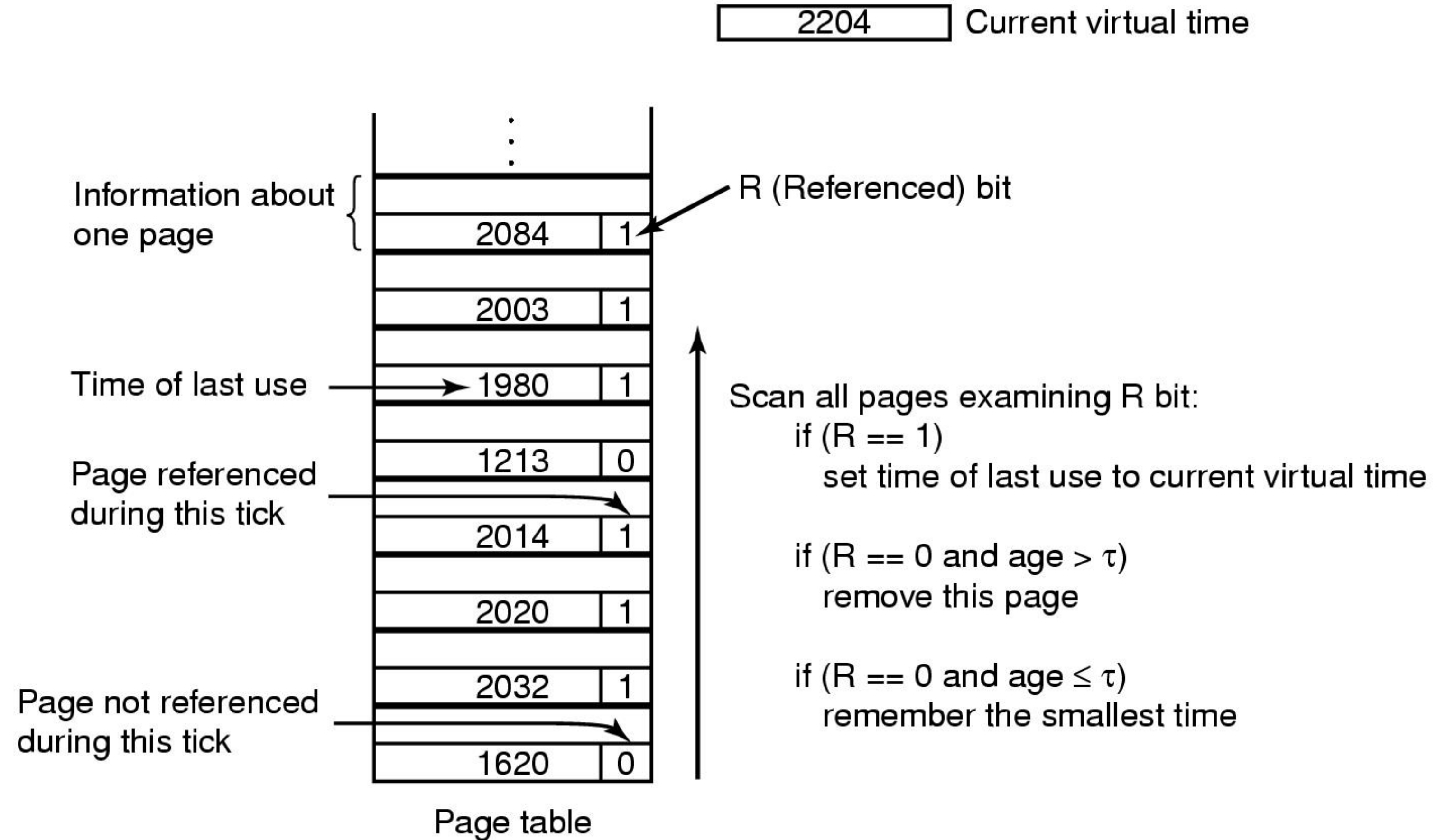


Exercise: The Working Set Page Replacement Algorithm

Suppose tau = 1000.

Update the diagram according to the algorithm.

What page will be evicted next?



Algorithm #7: WSClock Replacement Algorithm

Idea: Combine the working set algorithm with the clock algorithm

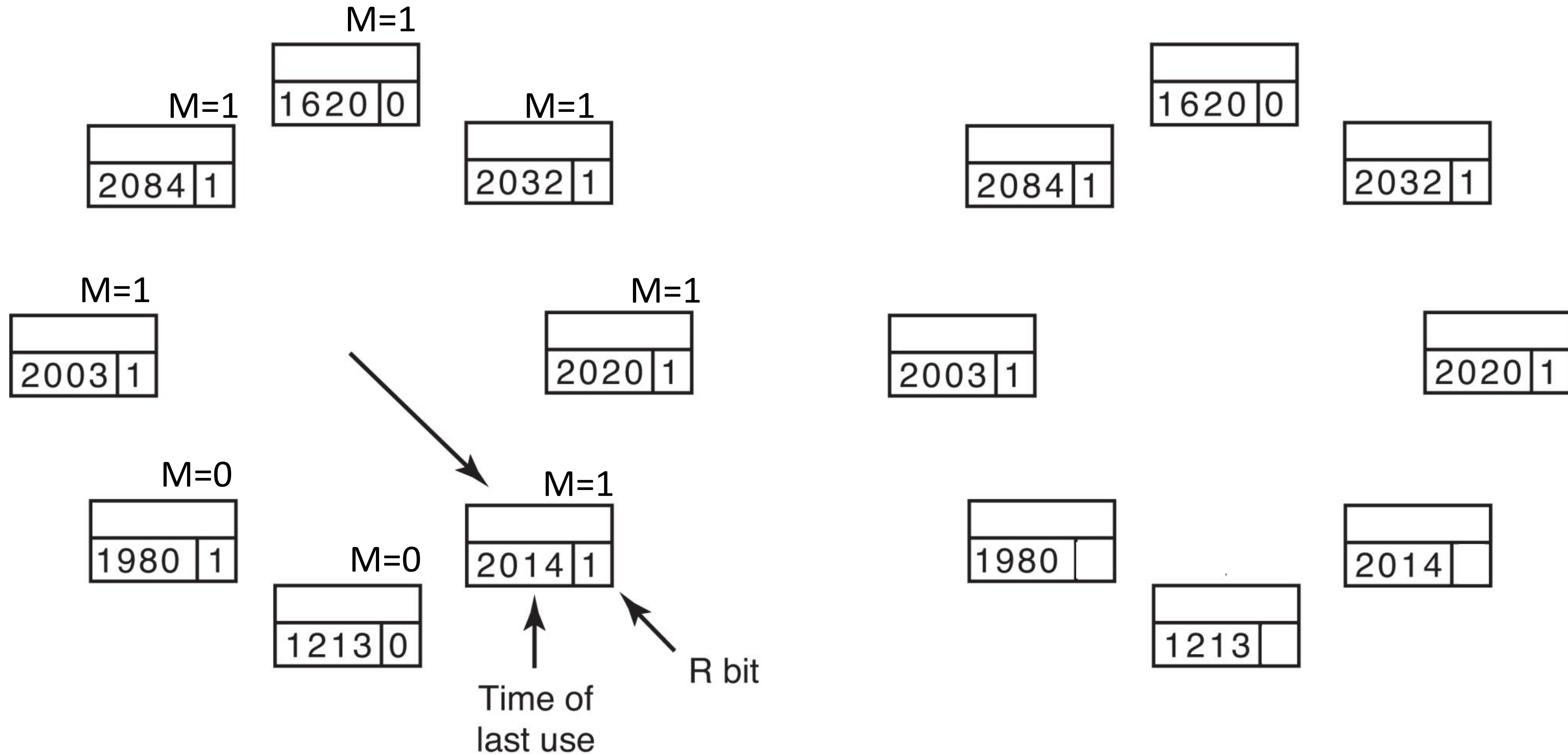
Goal: Avoid scanning the entire page table

How it works:

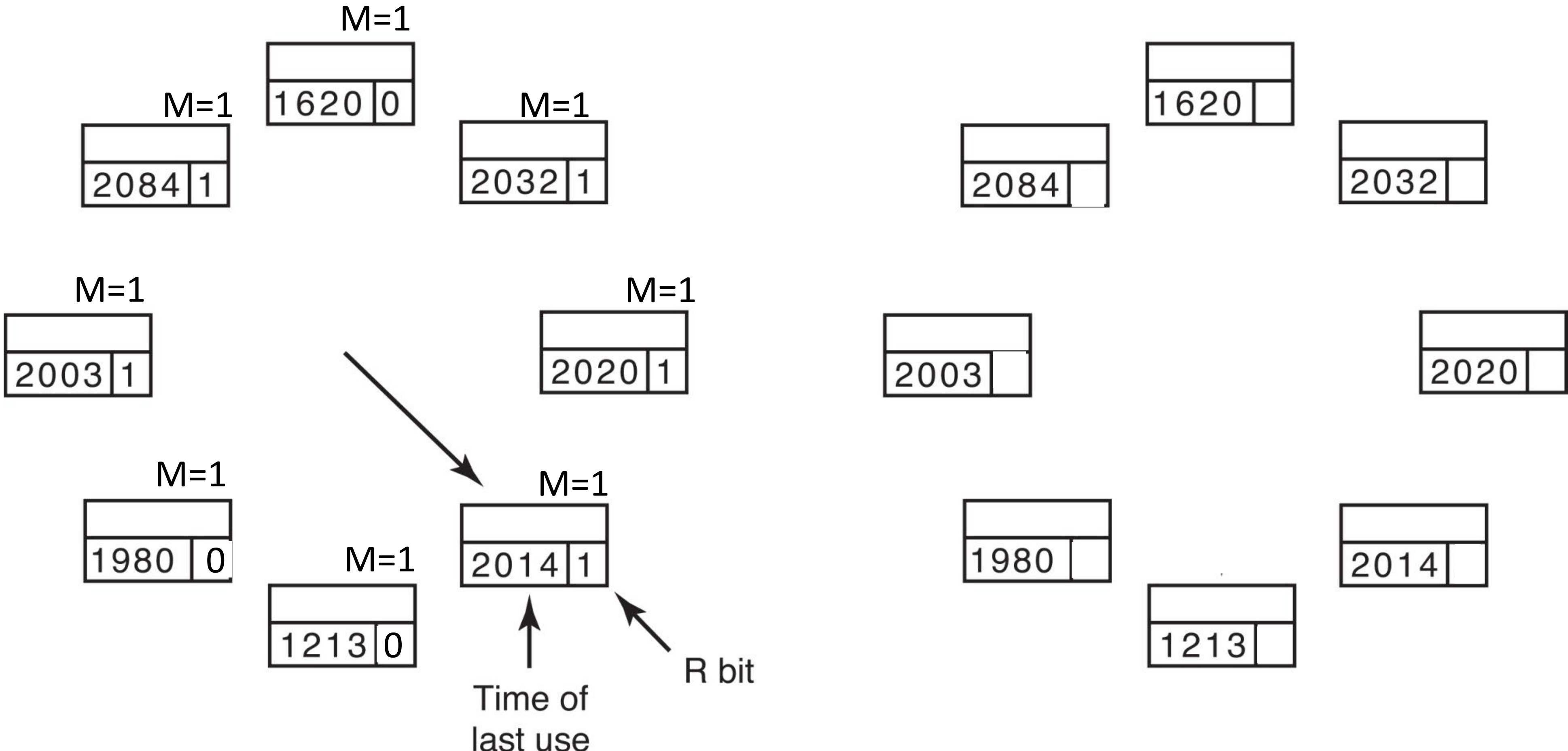
- New pages are added to the end of a circular linked list. Keep a pointer to the first element.
- When a page fault occurs
  - If R = 1, the page has been used recently. Set R = 0. Update tolu. Check the next page.
  - If R = 0 && M = 0 && age > τ, replace the page (easy)
  - If R = 0 && M = 1, schedule the page for writing but don’t wait for it to finish. Check the next page.
- Suppose we go through the list but haven’t found a page
  - If at least one page was schedule for writing, wait for one of them to finish. Then replace that page.
  - If no pages were scheduled for writing, claim the oldest clean page (M = 0)

Example: WSClock

Suppose τ = 200
What page will be replaced?
How will the entries update?



Example: WSClock



Example: WSClock

Suppose $\tau = 200$
What page will be replaced?
How will the entries update?

2204 Current virtual time

M=1
1620 0

M=1
2084 1

M=1
2032 1

M=1
2003 1

M=1
2020 1

M=0
1980 1

M=1
1213 0

M=1
2014 1

Time of last use

R bit

1620 0
2084 1
2032 1
2003 1
2020 1
1980
1213
2014

Review of Page Replacement Algorithms

Algorithm	Comment
Optimal	Not implementable, but useful as a benchmark
NRU (Not Recently Used)	Very crude
FIFO (First-In, First-Out)	Might throw out important pages
Second chance	Big improvement over FIFO
Clock	Realistic
LRU (Least Recently Used)	Excellent, but difficult to implement exactly
NFU (Not Frequently Used)	Fairly crude approximation to LRU
Aging	Efficient algorithm that approximates LRU well
Working set	Somewhat expensive to implement
WSClock	Good efficient algorithm

LRU and WSClock are the best methods

Implementation Details

Local vs. Global page replacement strategies

Should page replacement be managed within processes or across them?

Load Control

How should we swap processes out if thrashing occurs?

Page Size Choice

Balancing **internal fragmentation** (e.g. waste of space within a page) with page table and TLB sizes

Strategies for reducing the pages needed by processes

Shared libraries

Shared application pages (example: **copy on write** for forked processes)

Memory-mapped files (shared memory)

Paging daemon

Running a background process to free unused pages (like garbage collection)

Implementation: Paging and process

Summary: Four times that paging happens

- Process creation
- Process execution
- Page faults
- Process termination

A **swap area** on disk is used to store pages that are swapped out

A **swap partition,** or **backing store,** is reserved on disk for processes to use

When a process begins running, the MMU and TLB need to be flushed for it

When a page fault occurs, the OS has the find the address that triggered the fault

When the process exits, the OS releases its page table, its pages, and its disk space