Virtual Memory

Adapted by Joseph Lunderville
from slides by Dr. Brian Fraser
and course material by Dr. Steve Ko

Topics

1) How can **each process have its own address space?**
2) How can the **OS allocate memory** to processes?
3) What if we run **out of memory?**

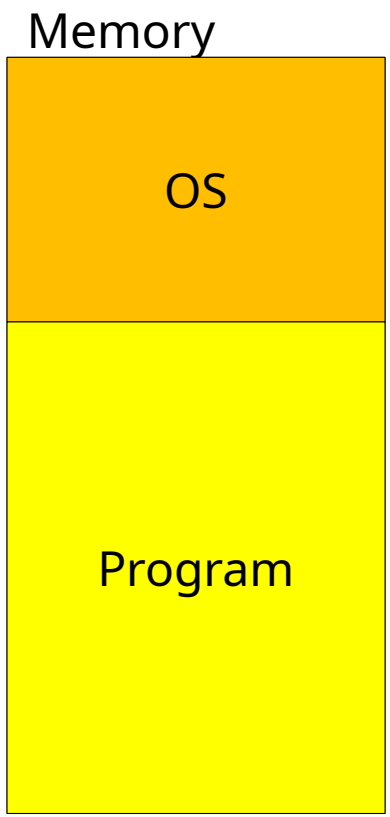
Context:
What problem are we solving?

Details

- Virtual memory is one of the most important OS concepts.
  - It is also a good example that shows the power of abstraction.
- Can find more info in OSTEP book (more depth than we require)
  - Chapter 13 The Abstraction: Address Spaces
    https://pages.cs.wisc.edu/~remzi/OSTEP/vm-intro.pdf
  - Chapter 15 Mechanism: Address Translation
    https://pages.cs.wisc.edu/~remzi/OSTEP/vm-mechanism.pdf
  - Chapter 18 Paging: Introduction
    https://pages.cs.wisc.edu/~remzi/OSTEP/vm-paging.pdf
  - Chapter 16 Segmentation
    https://pages.cs.wisc.edu/~remzi/OSTEP/vm-segmentation.pdf

Memory Layout in the Early Days

- *One address space*
  - Could run only a single program
  - One user at a time: either run batch jobs, or get poor utilization
- *Memory divided into two parts*
  - **OS** and **program**
- Computers are very, very expensive… can we *share* interactive access?



Early Memory Sharing

- *Memory divided into regions*
  - Could run multiple processes
  - Fixed allocations (wasteful!)
- *Segments, overlays*
  - Allow relocation, more efficient sharing
- *Problems*
  - Not granular, regions/segments/overlays are big
  - Extra complexity for programmers to manage overlays manually (overhead if automated)
  - No implicit protection, a “bad” pointer in one process could access another process's memory

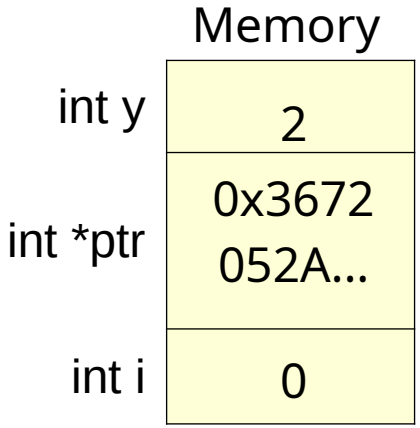
Understanding Memory

Address-Based Memory Operations

- *Variables are a convenience for programmers*
  - The computer really operates on memory
  - Instructions are defined by changes to memory
- *Random Access Memory (RAM)*
  - All addresses are equally fast to access
    - Probabilistically true! Used to be literal
    - Handwave effects of cache, NUMA
  - Contrast e.g. tape, with sequential access

Code:

```
int i = 0;
int *ptr = &i;
int y = i + 2;
```



Locality

- **Programs tend to access a small portion of their memory much more frequently**
- *Code*
  - Executes sequentially, but…
  - Loops jump back and cause a sequence to repeat
  - Branches skip a portion of the sequence (and often the same one)
  - Same function may be called many times
- *Data*
  - Small parts of a larger data structure accessed more often, e.g.
    - Index in a database accessed before every data access
    - In a game, the part of the map the player is in right now
  - At a small scale, e.g. in a loop iterating over an array,
    - Local variables in the loop are accessed repeatedly
    - The array is accessed sequentially

Locality

- **Temporal Locality:**
  - *Recently accessed memory* is likely to be accessed again (code and variables used in a loop)
    - Caches directly help
- **Spatial Locality:**
  - Data *near to recently accessed memory* is likely to be accessed next
  - More sophisticated CPUs with caches usually extend this concept
    - Cache lines often include adjacent data
    - Branch prediction, strided prefetch

Understanding Memory Solutions

- *Fundamental properties of memory use*
  - Programs really work on memory.
  - Programs access the same data over and over again (temporal locality)
  - Programs access data near to previously accessed data (spatial locality)
- ***Can these properties help us design a memory sharing abstraction?***

Audience Participation - Locality

- *Assume a program has just accessed memory locations **6 and 12***
  - *Spatial locality* suggests we might soon access…

    a) 0, 3, 9

    b) 6, 12

    c) 5, 7, 11, 13

    d) 4, 8, 10, 14

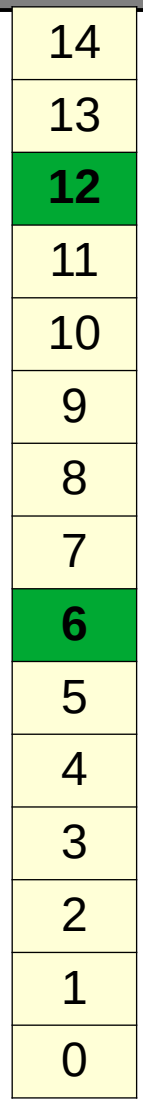
  - *Temporal locality* suggests we might soon access…

    a) 0, 3, 9

    b) 6, 12

    c) 5, 7, 11, 13

    d) 4, 8, 10, 14



Solution:
Virtual Memory

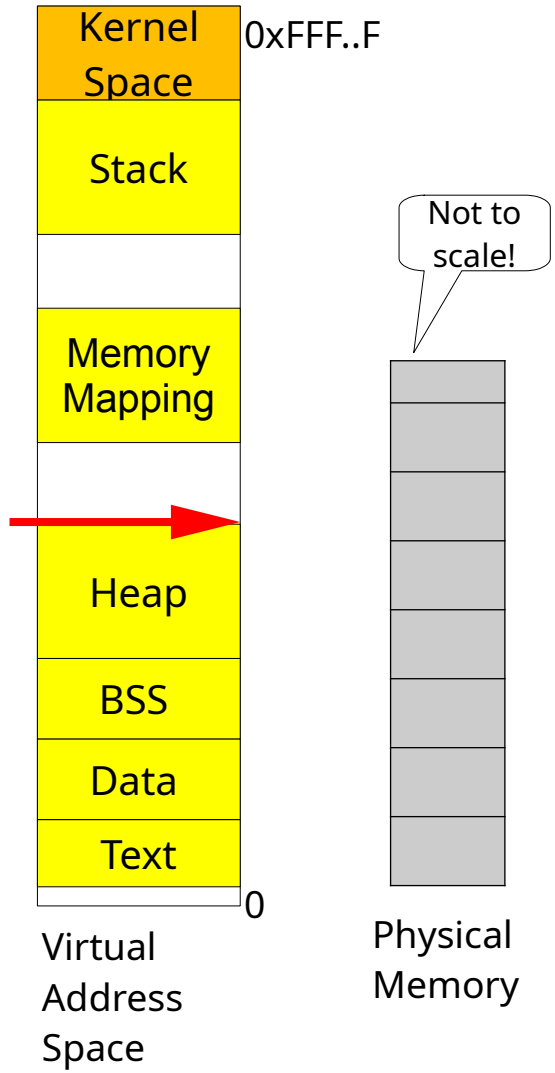
Memory Abstraction

"All problems in computer science can be solved by another level of indirection, except for the problem of too many levels of indirection."
-- David Wheeler

- *Virtual memory is a mechanism to enable*
  - **physical memory sharing** for multiple processes
  - **isolation** of each process's memory access
- ***A process uses virtual memory instead of physical memory***
- *Virtual memory consists of*
  - **Virtual address space** and **address translation**
  - Virtual memory is a good example that demonstrates the power of *abstractions*

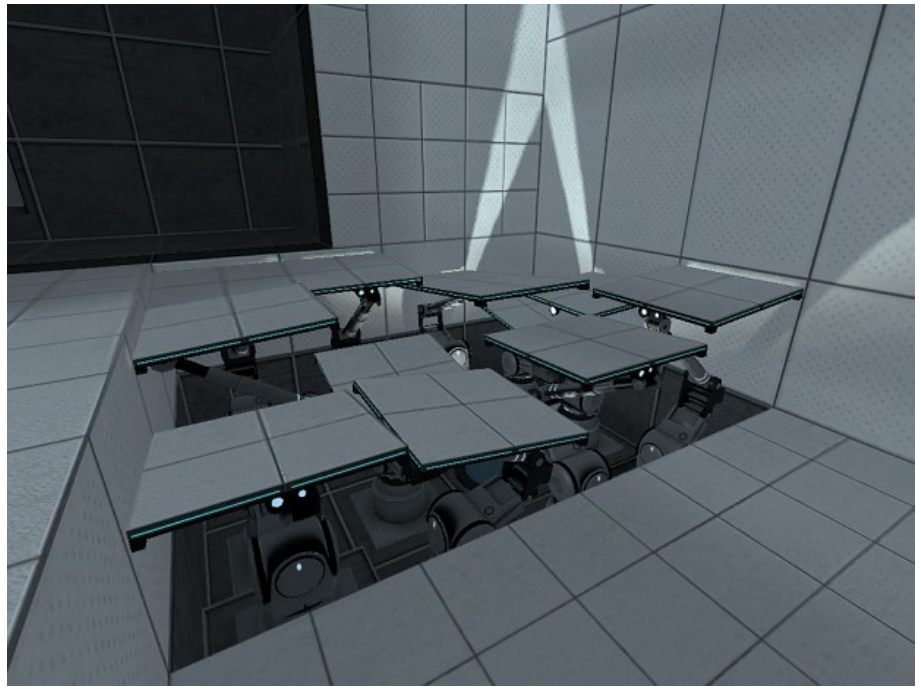
Virtual Address Space

- *All memory discussed so far has been virtual memory!*
  - Virtual address space size is determined by the pointer size: 0 to (e.g.) $2^{64} - 1$
  - Virtual memory is a memory abstraction (imaginary space) that the program & programmer operates in
  - The OS and hardware build us this imaginary space
- *Virtual vs physical*
  - User-level processes works with virtual addresses
  - Kernel-level components can deal with both virtual and physical addresses



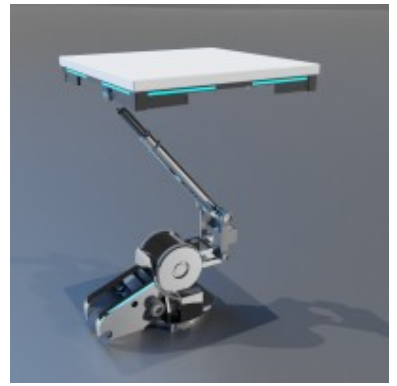
Room Analogy

- *Imagine a process as a room*
  - Its virtual memory space is the surface of the walls
  - There are no real walls, they are an illusion
  - Wall panels are moved into place as needed to make the room



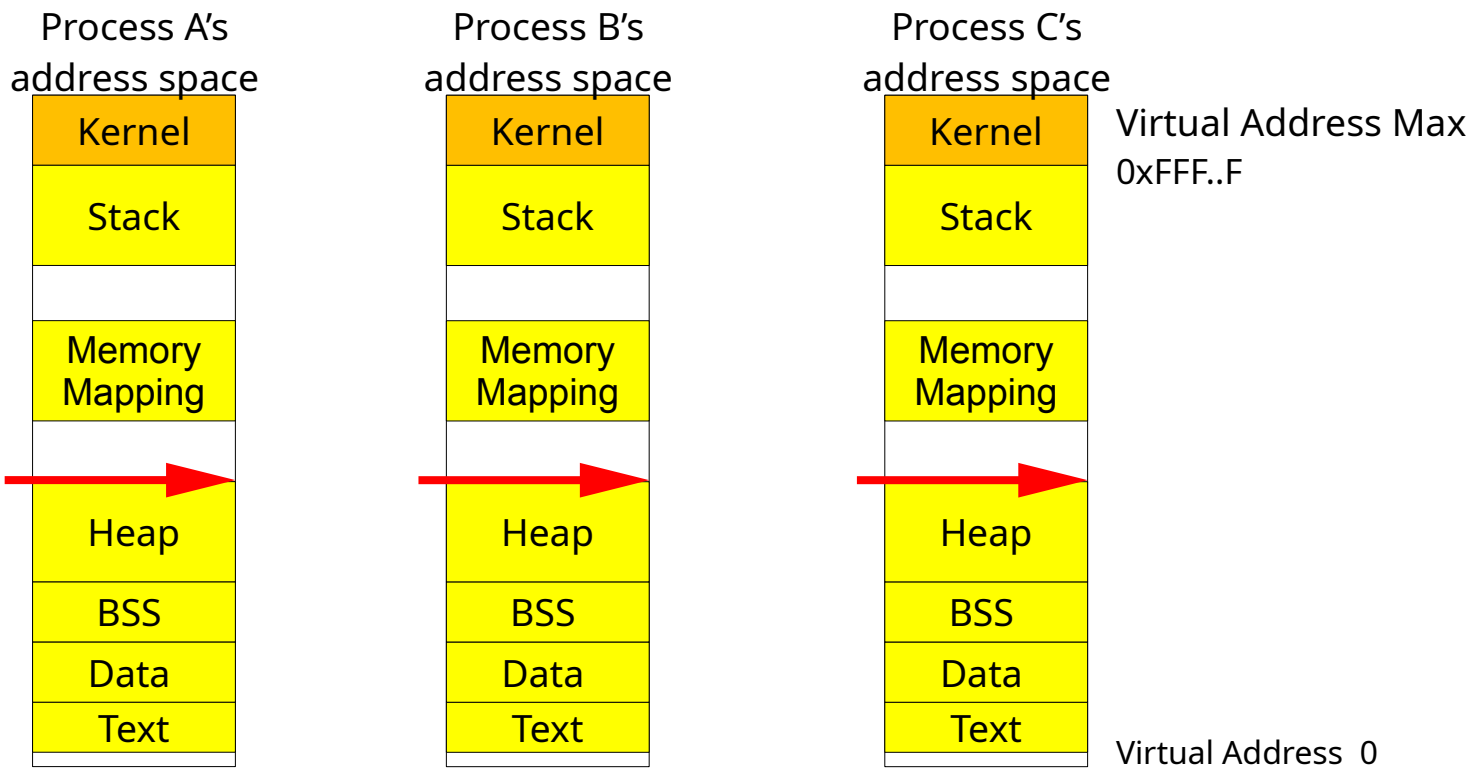
Room Analogy (cont'd)

- Imagine a process as a room
  - Virtual memory space is the walls: pointers can point to the wall, can read/write on wall
  - Walls have (e.g.) 2^{64} locations; much bigger than physical memory
- *OS and hardware only put "physical" memory panels behind **a few areas** of the wall*
  - Operations on areas with physical panels work
  - Operations *outside* of those areas fail (page faults!)
  - E.g., program reads from 0x100 – that's a *virtual memory address*
    - Doesn't (can't!) know what *physical* "panel" is read
- *A physical "panel" is called either a **page frame** or **segment***



Process Virtual Memory

- *Each process gets its own (virtual) address space*
  - 0 to (e.g.) 2^{64} - 1 (or 2^{32} – 1 for a 32-bit architecture)
- *Each address in a virtual address space is a virtual address*
  - (*physical* address points to a *physical* memory location)



Benefits of Virtual Memory

- *A process only sees its own address space*
  - We get memory isolation between processes implicitly!
- *Temporal & spatial locality mean a process likely does not need all its data at once*
  - Don't have 16EB per process of physical memory!
  - OS can “oversubscribe” RAM
  - Copy RAM areas that have not recently been used out to disk (“swap out” or “swap to disk”)
  - This file is called **swap space**
  - Can be loaded back into RAM as needed

Room Analogy

- *Out of memory*
  - We can run out of physical memory panels for our room: go to touch a wall, but there are no panels available to put there
  - So take a panel we haven't used in a while, save what's on it to disk, and then reuse it *where we are now*
- *If Needed Again*
  - When we go to the old panel we saved out we need to restore it: take another physical panel and reload the swapped out data from disk
  - Map virtual memory to the correct physical memory location
  - *Program is paused while all this happens – it never knows the difference!*
- Works across multiple processes
  - OS manages mapping virtual address to physical memory panels
  - Panels are shared across all processes

Address Translation

Address Translation

- *Process knows **virtual addresses;** hardware needs **physical address***
  - Must translate between them!
- *Virtual Memory is divided into regions called **pages***
  - Each page is mapped to a physical memory **“page frame”** or just **“frame”**
  - Kernel controls the mapping
  - Kernel configures hardware to translate virtual addresses into physical addresses

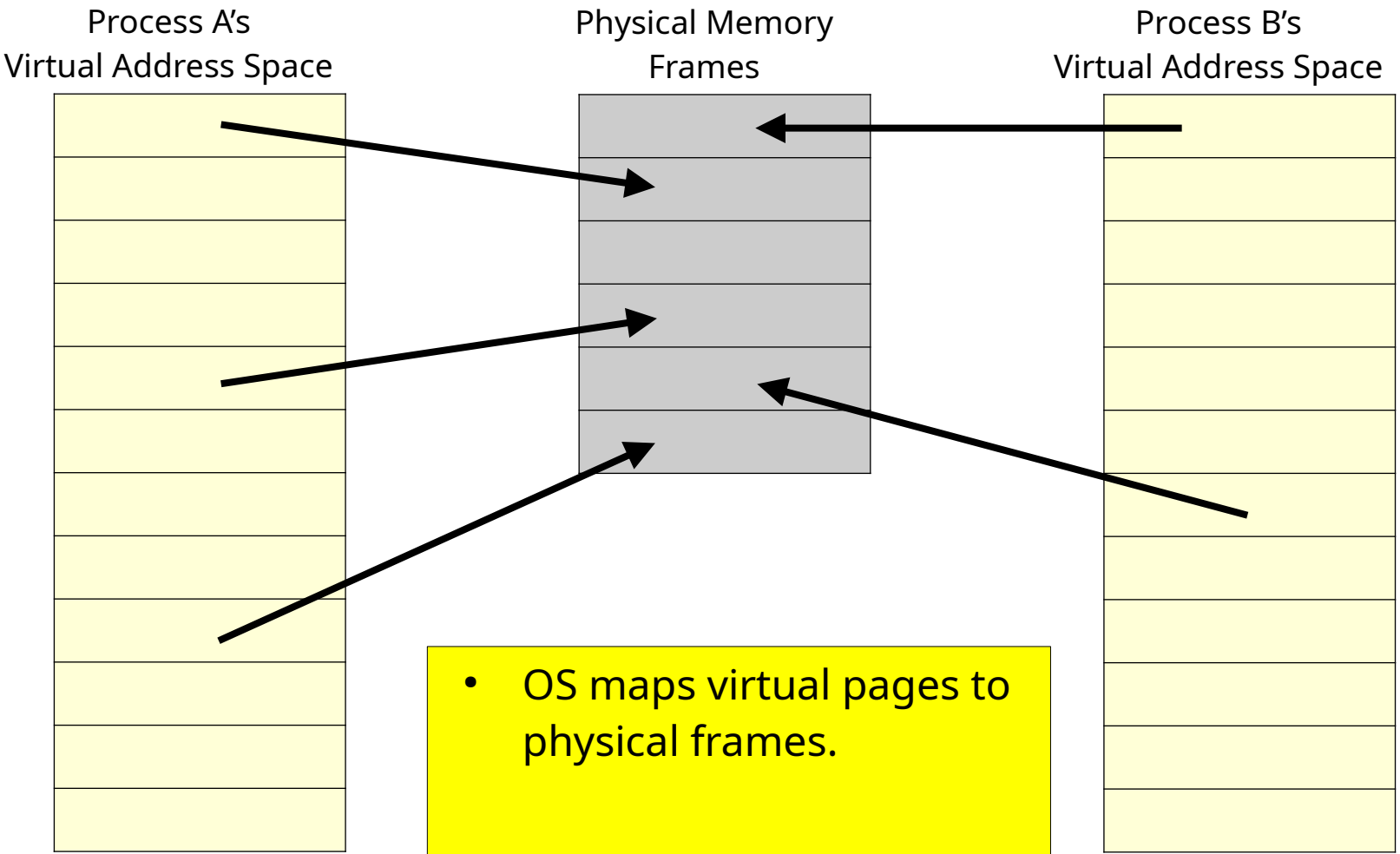
Address Translation

- *Consider a memory operation like:*

```
int *ptr;
*ptr = 10;
```

- *Steps in translation*
  1) Figure out which virtual memory page *ptr is on
  2) Figure out which physical frame it maps to
  3) Redirect the access to the correct physical memory frame and address within it

Address Translation



- OS maps virtual pages to physical frames.

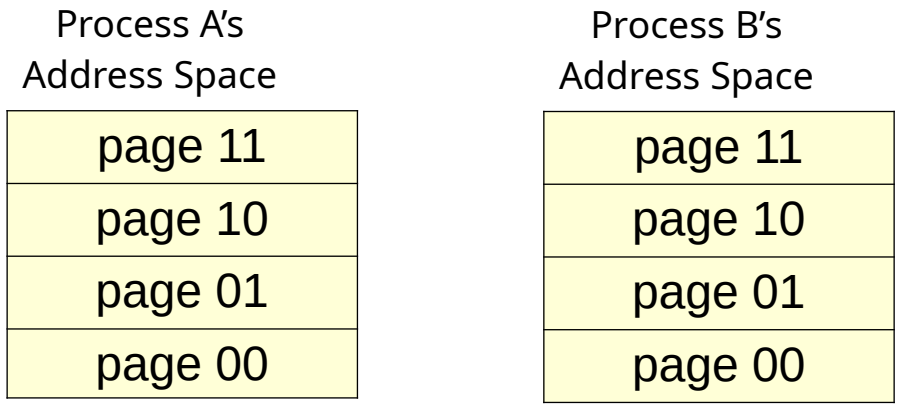
Address Translation

- *Approaches to Mapping “Panels” to Memory*
  - How do we divide our virtual address space into smaller regions (“panels” in our analogy)?

Paging

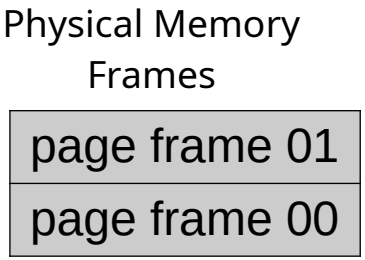
Pages

- *Virtual address space is divided into fixed-size pages*
  - 4 KB is a popular size but modern OSs have bigger pages (e.g., 4 MB) as well
- *Example*
  - If we have 16KB virtual address space and page size 4K – how many pages?
  - We need 4 pages
  - Here are 2 process, each with its own virtual address space; page numbers are in binary:



Page Frames

- *Physical memory divided into page frames (or frames)*
  - Page frames and pages are the same size
- *Example*
  - If we have 8KB of memory with 4KB page size = 2 frames (#'s in binary)



Address Translation

- *A virtual address is divided into two parts*
  - <page number, offset>
- *Example*
  - 4 pages, each of 16 bytes
  - 4 pages need…
    - 2 bits to pick between pages
  - 16 bytes need…
    - 4 bits of offset into the page
  - 6-bit virtual address space divided into 2-bit page numbers and 4-bit offsets
    - Address 100101 is divided into page number 10 and offset 0101
    - Address 000010 is divided into page number 00 and offset 0010

Audience Participation - Address Translation

- *Consider a computer where*
  - each page is 32 bytes
  - have 8 pages
  - What does the memory address 10011010b mean?

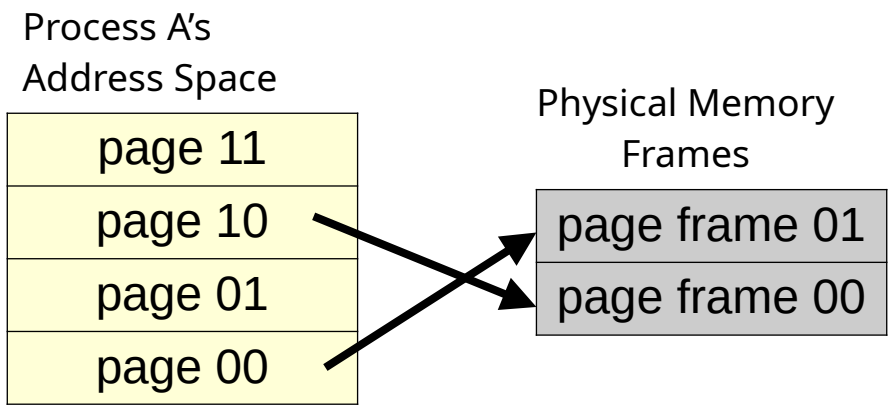
a) Page 10011b, Offset 010b
b) Page 100b, Offset 11010b
c) Page 010b, Offset 10011b
d) Page 11010b, Offset 100b

Page Table

- *When a process accesses a (virtual) address*
  - Paging translates the address's page number to a page frame number
  - The offset is not translated, and *does not change*
- *Kernel maintains a **page table** per process*
  - Maps *page number* (virtual) to a *page **frame** number* (physical)

Page Table

Page Number	Page Frame Number
00	01
10	00

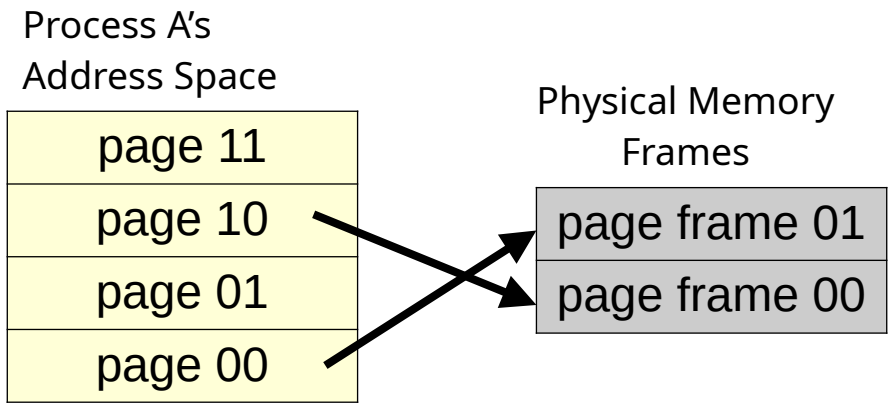


Address Translation Example

- *Example*
  - Convert virtual address 101011b to physical address:
  - Assume 16 byte pages, so offset is 4 bits
  - Address is 6 bits, so page is 2 bits (4 pages)
  - Page: 10b, maps to page frame 00b
  - Offset: 1011b (maps 1:1)
  - So physical memory address 001011b

Page Table

Page Number	Page Frame Number
00	01
10	00



Page Table Size

- There are vastly more (virtual) pages than (physical) page frames
  - Cannot in general map all pages to frames at once
    - This isn't just about memory *used* – there may be plenty of physical memory to satisfy all memory that's *allocated*
    - The *address space* of a process is much larger than what's allocated via `sbrk()`
  - OS only maps a page to a frame when needed (more later)
- Hardware supports converting pointers from virtual to physical addresses
  - OS configures the page table
  - Hardware looks up mappings at runtime

Page Table Size

- *Page Table Size*
  - If *page numbers* use ***n*** bits, the maximum possible *number of pages* is $\mathbf{2^n}$
  - If *offsets* use ***m*** bits, the maximum possible *page size* is $\mathbf{2^m}$
  - The address bit width is $n + m$
- *Example*
  - Page size of 4kB on a 32-bit architecture
  - $m = 12$ because $2^{12} = 4096 = 4k$
  - $n = 20$ because $32 - 12 = 20$
  - Therefore we have 2^{20} pages: this is 1,048,576 = 1M pages!

Audience Participation – Address Translation

- Given the page table below, what is the physical address for (virtual) address 0010 1011 1101 1100b?

a) <000001, 1111011100>

b) <001010, 1111011100>

c) <111010, 1111011100>

d) <000101, 1111011100>

Page Table

Page Number	Page Frame Number
000001	001010
111010	000011
101001	000111
001010	000101

Segmentation

Segmentation

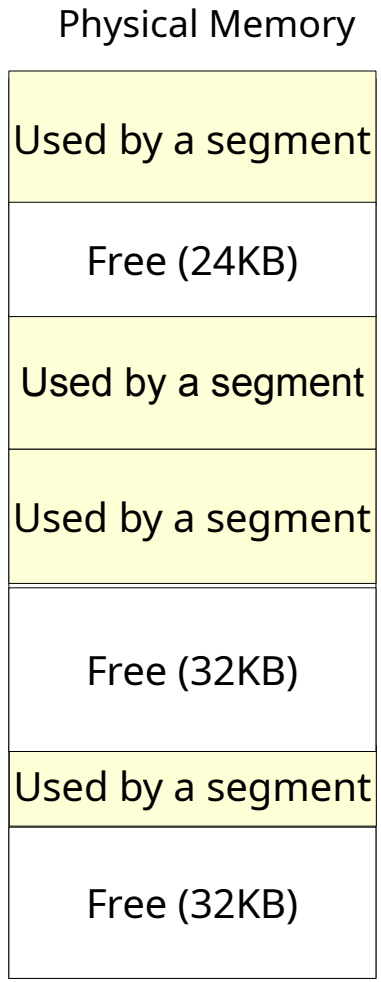
- *Segmented memory uses **variable-size** segments (or sections)*
  - Pointers are made of two *explicit* parts, **segment** and **offset**
  - Instead of taking a uniform virtual address and breaking it *transparently* into page and offset parts, these parts are exposed to the programmer
  - The segment has a “base” physical address
  - Segments are often mapped to a region with some kind of meaning, e.g., text segment, data segment, stack segment, heap segment
- Fun fact: x86 has supported segmentation since the ‘80s
  - …and x86-64 still does
  - …for some reason

Segmentation Address translation

- *A segmented system can map virtual memory to physical memory*
  - In a segmented *virtual* memory system, the segment is a handle
  - To access memory, the processor has to look up the associated physical base address (like it would with page mapping)
  - Hardware usually caches the physical base address when the segment is loaded into a special segment register
  - Now the software has to optimize segment loads explicitly – a headache we didn't want
- *Segmentation is a niche technique and details are beyond course scope*
  - About as common in the Real World as banked memory
  - You don't actually want me to explain banked memory

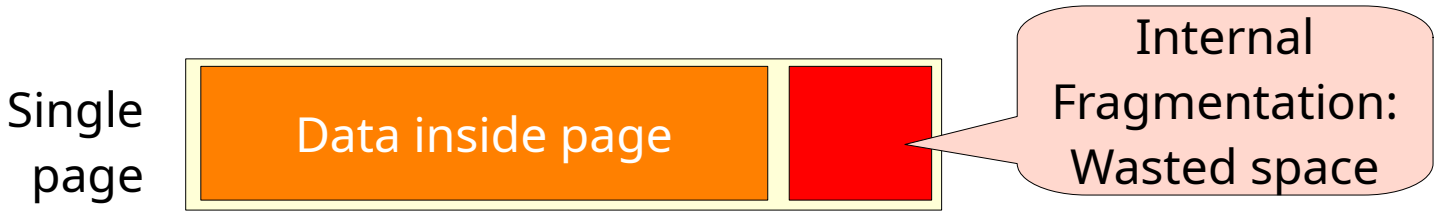
Segmentation and External Fragmentation

- *External fragmentation occurs when free space between allocations is split up*
  - Segments use contiguous blocks of non-uniform size
  - Free space gets fragmented just as in our heap examples
- *Example*
  - Try allocating 40kB… where does it go?
  - A virtual memory system using segments can move segments to defrag, but entire segments must move
- *Resizing segments is also a challenge*
  - What happens when we increase the program break?
  - Systems designed from the ground up to be segmented usually access the OS allocator directly (bad design smell)
  - Annoying and inefficient: pointers become 2x size



Paging and External Fragmentation

- *Paging does not suffer external fragmentation*
  - Every page is the same size
  - Allocations may be discontiguous
  - When you need a page, any page will do
- *Internal Fragmentation*
  - TANSTAAFL: there is very likely to be wasted space at the end of a fixed-size page
  - Called **internal fragmentation,** because the fragment is *inside* the allocation
  - Combat it by keeping *page size* small
  - Small pages mean more page table overhead – there is a tradeoff!



Running out of Memory

Out of Memory

- *Out of memory*
  - Limited physical memory but virtual memory space is vast!
  - Can't bring all virtual pages into physical memory – what do we do?
- *Demand paging and swapping*
  - **Demand paging:**
    a page is brought into memory only when needed (on demand)
  - **Swapping:**
    save an in-use page from memory to disk, and load in the required page
  - **Swap space:**
    disk space dedicated to store swapped-out pages
- *How do we decide which memory page to swap out?*
  - We need a page replacement algorithm
  - Kind of a special case of cache replacement

Demand Paging

- *Why does demand paging work?*
  - Insight: a typical program only needs to access a small portion of its memory space
  - This is about locality of access
- *Recall definitions of locality*
  - **Temporal locality:** if a program accesses a memory location, it is likely that it's going to access it again in the near future
    - Recently used pages are already in memory
  - **Spatial locality:** if a program accesses a memory location, it is likely that it's going to access other memory locations nearby
    - When a memory location is accessed, demand paging brings in the rest of the nearby region as well
    - Farther areas from the same segment are *not* brought in!

Page Replacement Algorithms

- *Page fault*
  - when a memory location is accessed but the corresponding page is not found in physical memory
  - we need to bring the page into a frame
- *The question*
  - When memory is full* (i.e., all page frames are used) and we need to load a new page, which page do we swap out to disk to make space?

    * On a *real* modern computer, often lots of memory is “free”, but filled with disk cache… we do not consider this situation

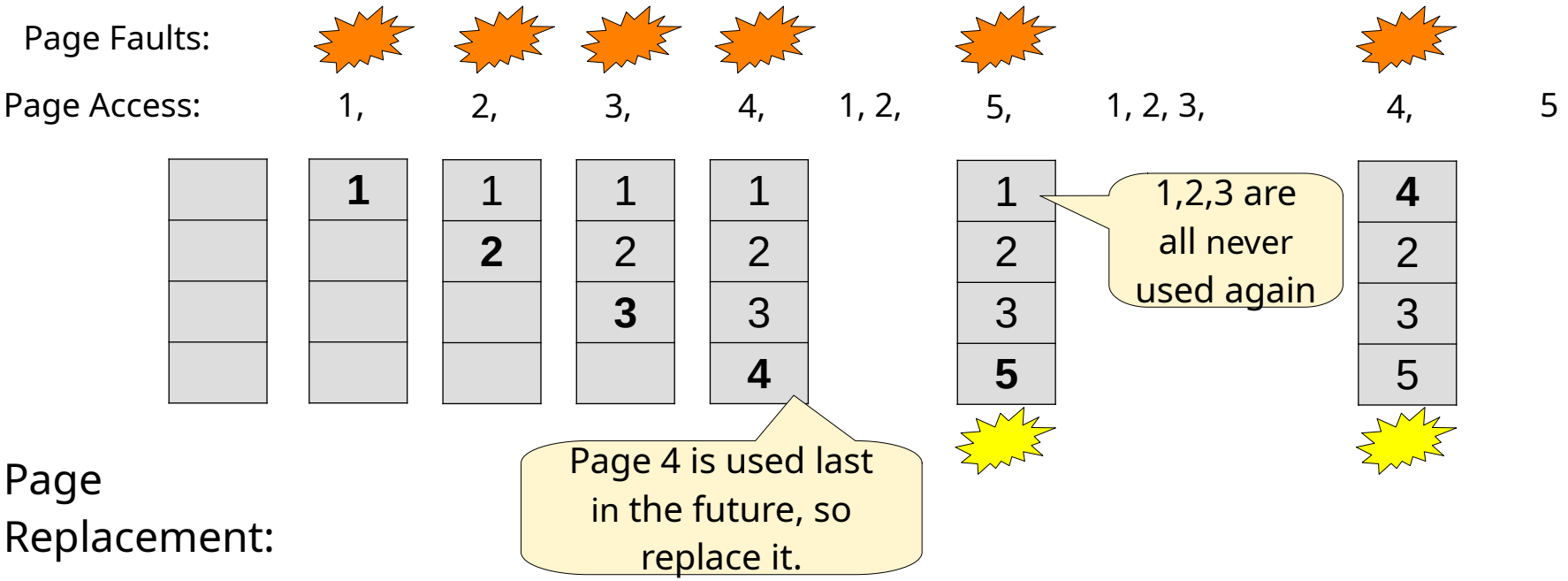
Optimal Page Replacement Algorithm

- *Optimal page replacement algorithm picks the page that will not be used for the longest time*
  - This assumes that we know the future (which is of course impossible)
  - It’s useful as a comparison
  - Page replacement algorithms try to approximate optimal behaviour

Optimal Page Replacement Example

- *Example*
  - Memory has 4 page frames
  - Memory page access order (by page number):

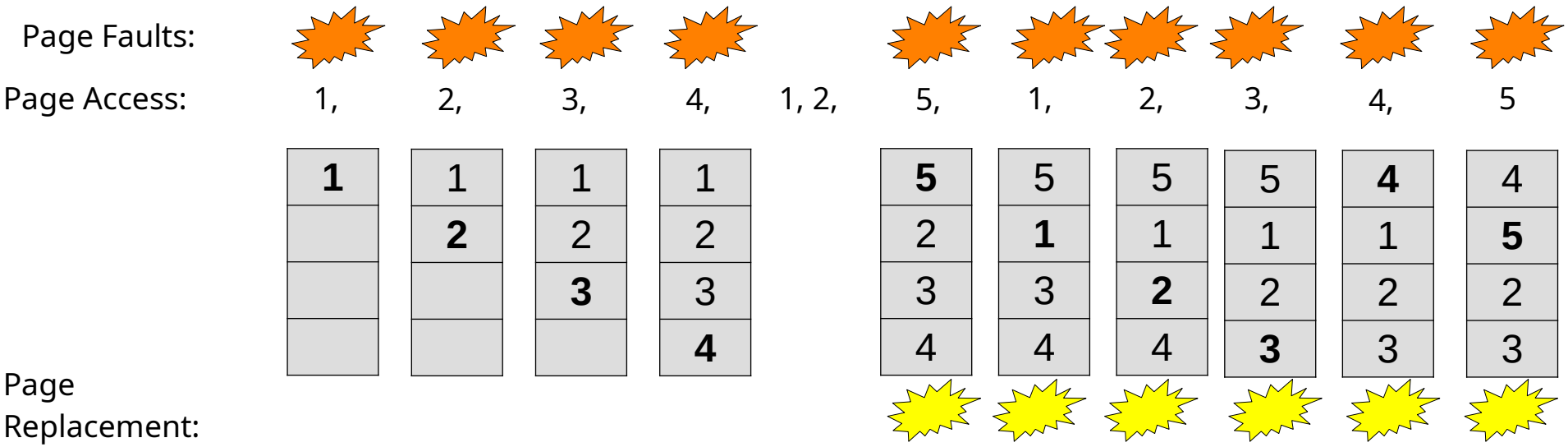
    1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



Page Replacement:

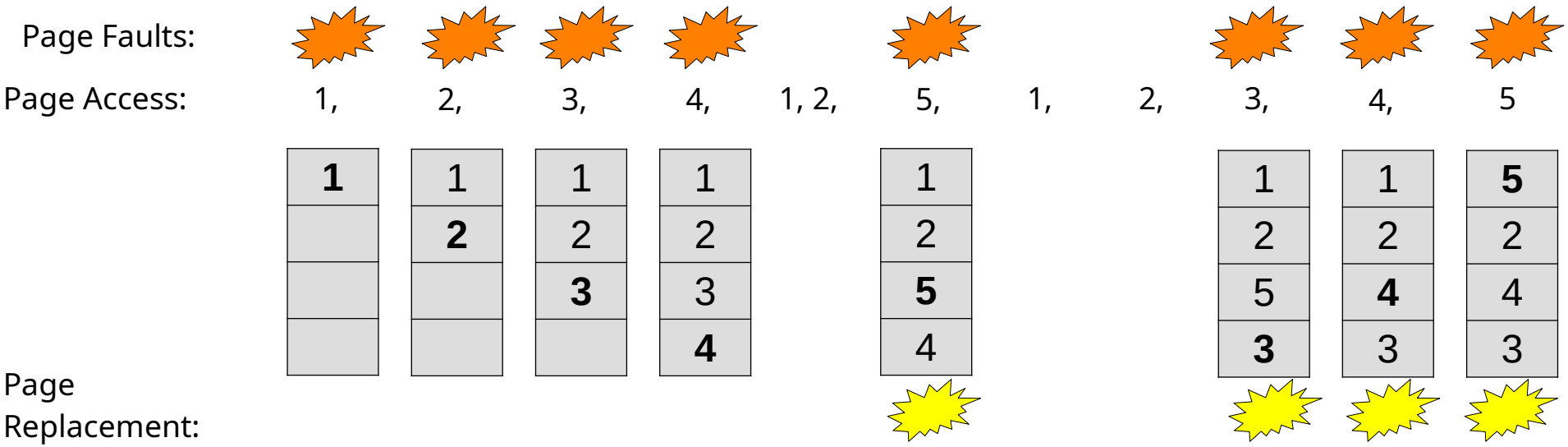
FIFO (First In, First Out)

- *Oldest page gets swapped out first*
  - Keeps track of when a page was *brought in* to memory
  - 10 page faults!
  - Simple, but does not exploit locality well
  - …random selection is often better



LRU (Least Recently Used)

- *Replace page that has gone **unused** for longest period*
  - Closer to approximating the optimal algorithm
  - Tries to infer the future based on past
  - 8 page faults
  - **Tracking oldest access time across many pages is not simple**



Audience Participation - LRU Paging

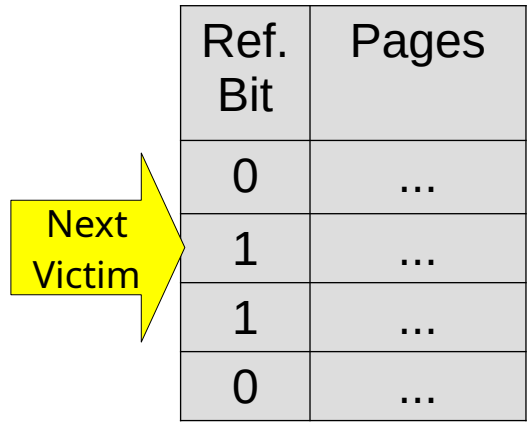
- Consider the following computer:
  - 4 page frames
  - Uses LRU page replacement algorithm
- How many page faults are there for the following sequence of page accesses?

1,	2,	3,	4,	5,	2,	4,	5,	1,	5
*	*	*	*	*(1)				*(3)	

a) 2 page faults
b) 5 page faults
c) 6 page faults
d) 10 page faults

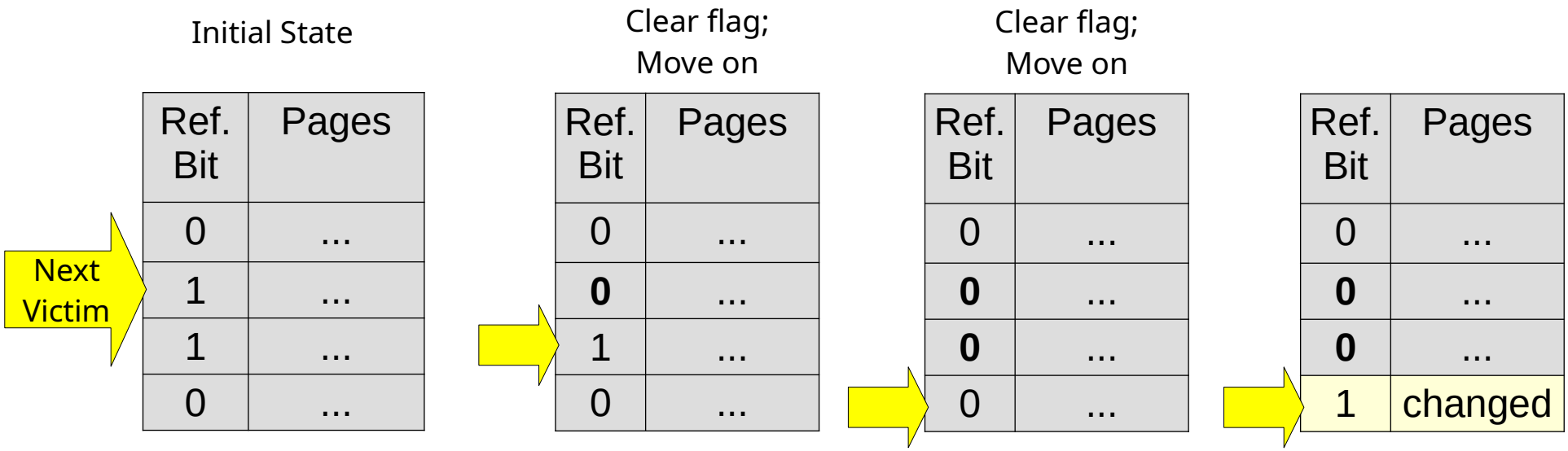
Second Chance

- Second Change is an approximation of LRU
  - Each page has a reference bit (ref_bit), initially = 0
  - When a page is accessed, hardware sets ref_bit to 1
  - Maintain a moving pointer to the next (candidate) "victim"
- When choosing a page to replace, check `ref_bit`
  - If `ref_bit == 0`, replace it
  - Else,
    - Clear ref_bit to 0
    - Leave page in memory (second chance)
    - Move pointer to next page (wrap around)
    - Repeat until a victim is found



Second Chance Example

- *Example*
  - Assume we have triggered a page fault
  - No empty pages, so must replace
  - Let's find the victim page to replace



Audience Participation - Second Chance

- Using second chance page replacement algorithm, which page will be the next victim?

Ref. Bit	Pages
1	Page 110
1	Page 111
0	Page 101
1	Page 001

Next Victim

a) Page 110b
b) Page 111b
c) Page 101b
d) Page 001b

Thrashing

Thrashing

- If a process access a large amount of memory, OS could keep needing to bring new pages into memory
- *Example*
  - A process that jumps through a huge amount of memory, reading one value every 4K (once per page)
- ***Thrashing:***
  - A process is disproportionately spending time swapping pages in and out, and not executing instructions on the CPU

Summary

- Virtual Memory
  - Process works only in the *virtual memory space*
  - OS can flexibly share memory between processes
  - Gives process *memory isolation*
- Address Translation
  - Converting (virtual) addresses to physical addresses
- Paging
  - Virtual memory broken up into *identical size* **pages**
  - Physical memory broken up into **page frames** (“frames”)
- Segmentation
  - Like paging, but different/variable size regions **(segments)**
- Page replacement algorithms
  - **Optimal, FIFO, LRU, Second Chance**