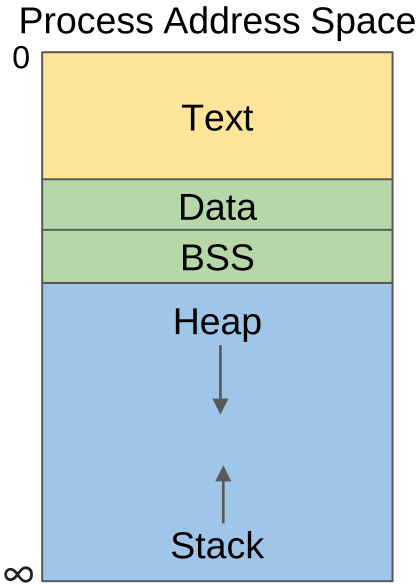
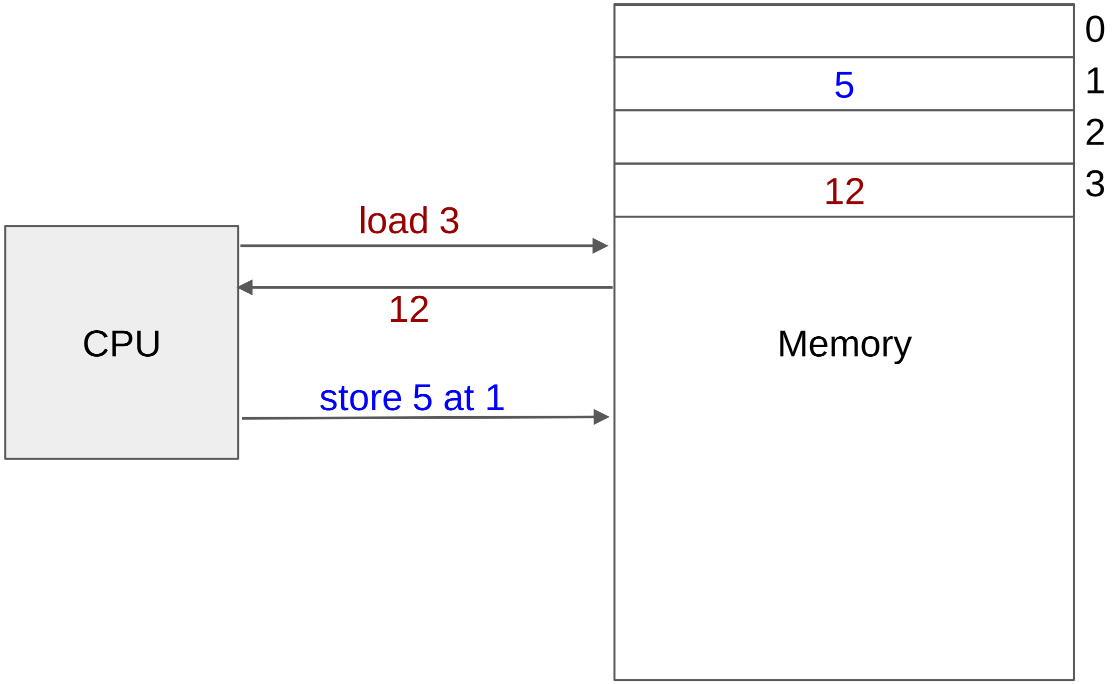
Memory Management

- Relocation
- Protection
- Sharing
- Logical Organization
- Physical Organization
    - Main
    - Secondary

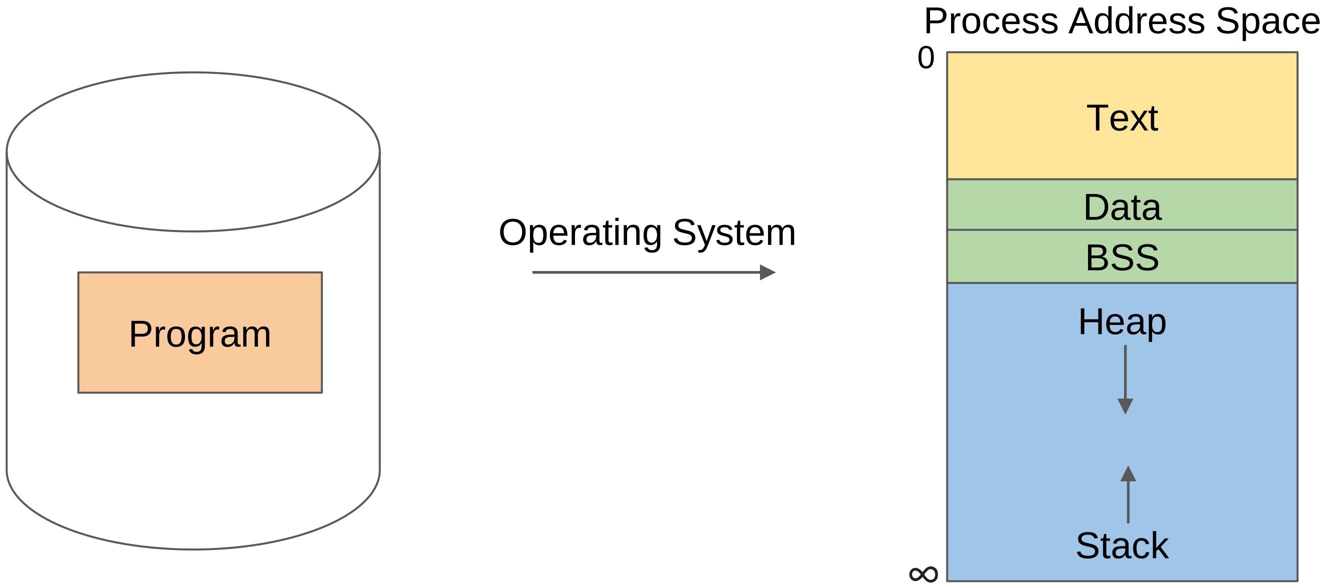


Memory

Logically - An array of bytes accessed by address



Program vs Process



Memory Allocation

- When processes start memory needs to be allocated
  - Process Control Block
  - Text
  - Data

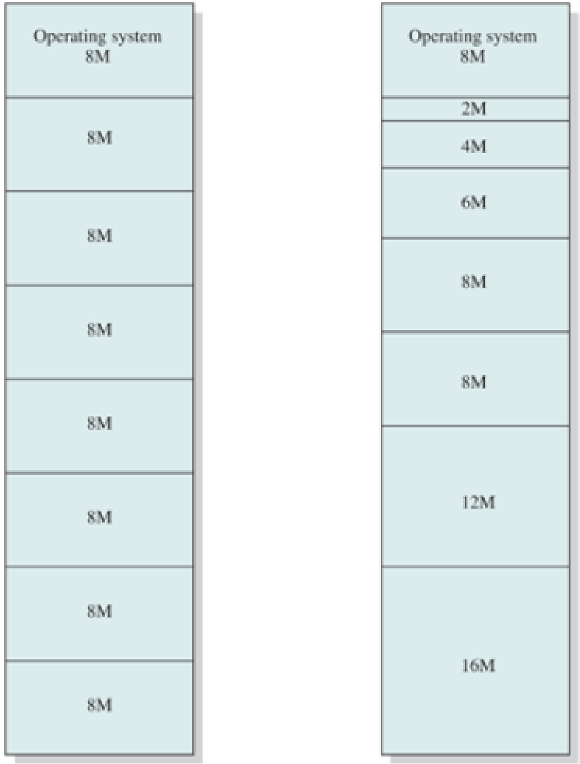
- Processes want to dynamically allocate and free memory
  - Stack
  - Heap

- Memory is finite - not all processes will fit in memory at the same time

Memory Allocation - Fixed Partitions

- Every process gets the same size memory partition

- Problems:
    - Internal fragmentation
    - Some processes need more memory than others



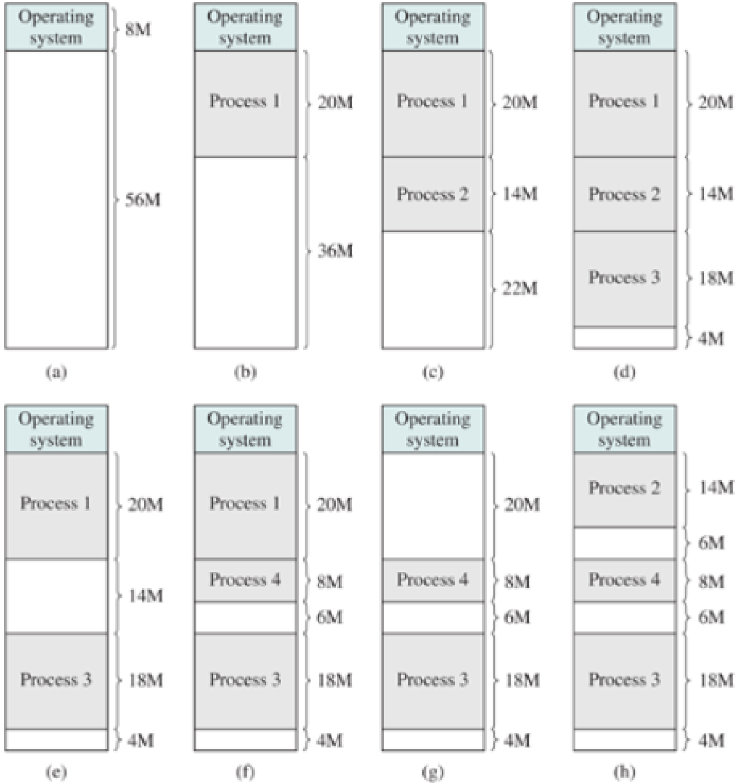
(a) Equal-size partitions (b) Unequal-size partitions

Memory Allocation - Dynamic Partitions

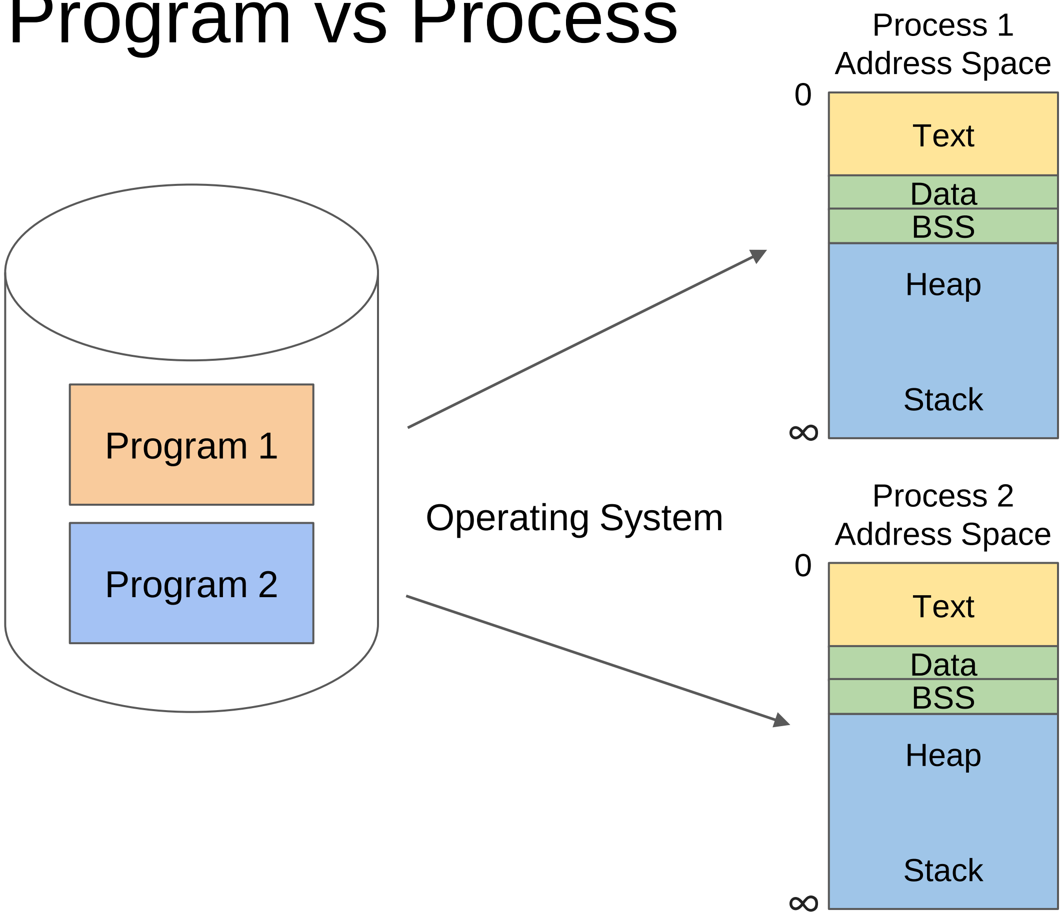
- Processes get memory partition that is the exact size for what they need

- Where do we put the process allocation?
  - First Fit
  - Next Fit
  - Best Fit
  - Worst Fit

- Problems:
  - External fragmentation
  - May not know how much a memory a process will need when it starts



Program vs Process



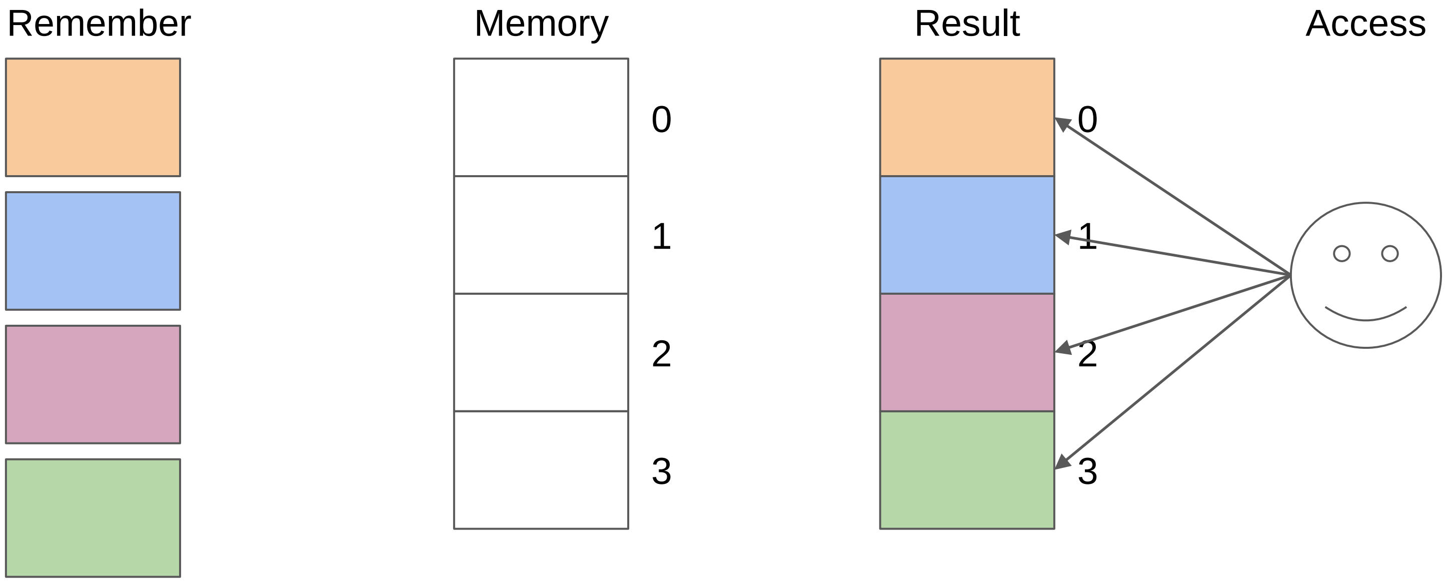
Problem:

- Memory is finite
- Both processes want to see an unlimited address space
- Processes don't want to know about the existence of other processes

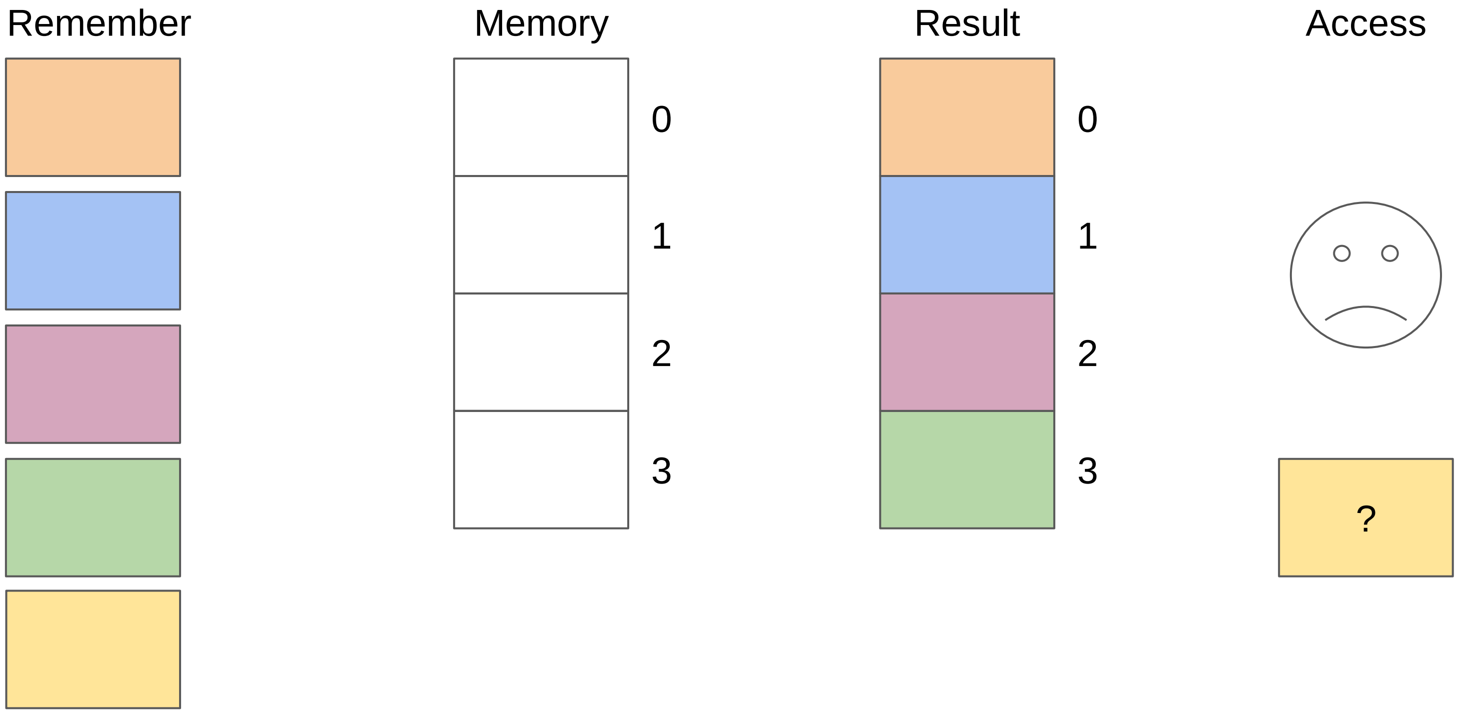
Solution:

Paging

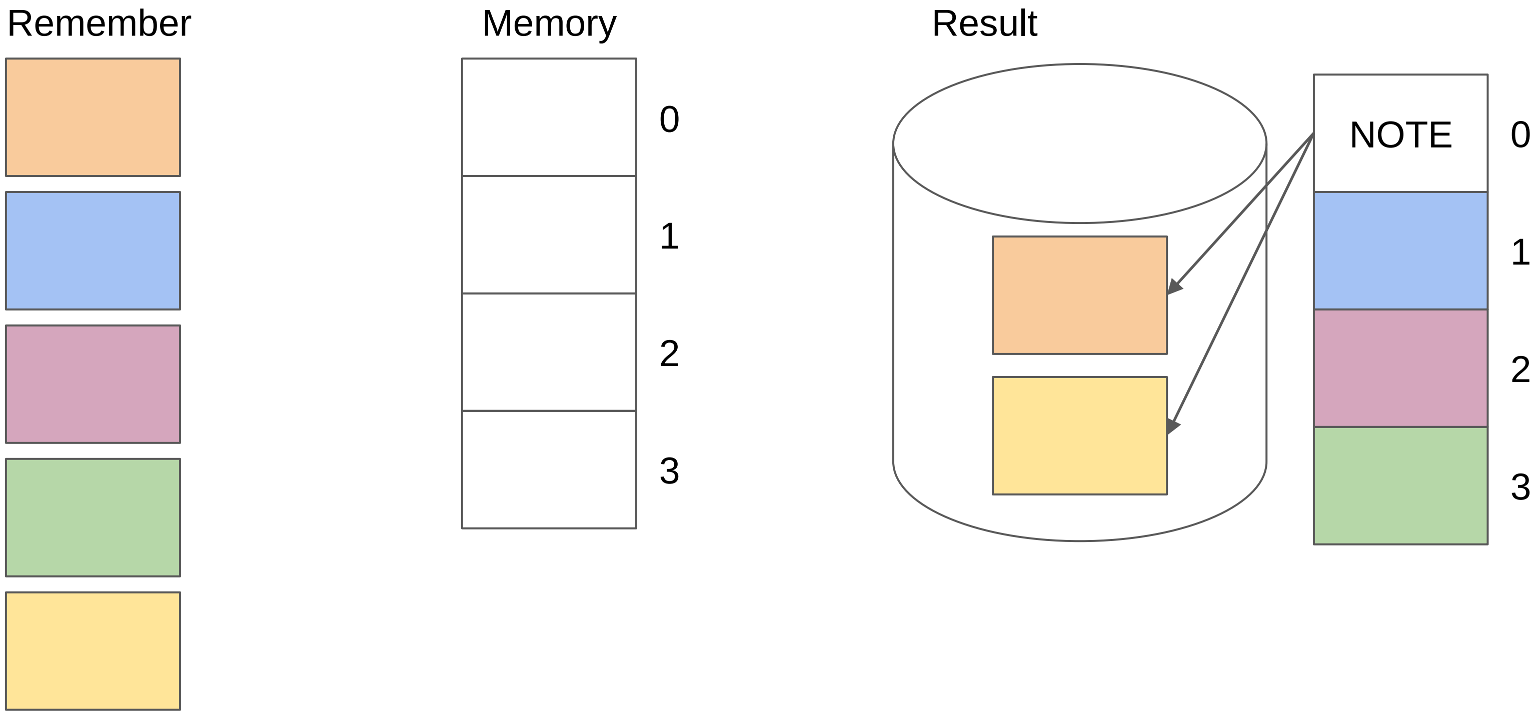
Paging - A Perfect World



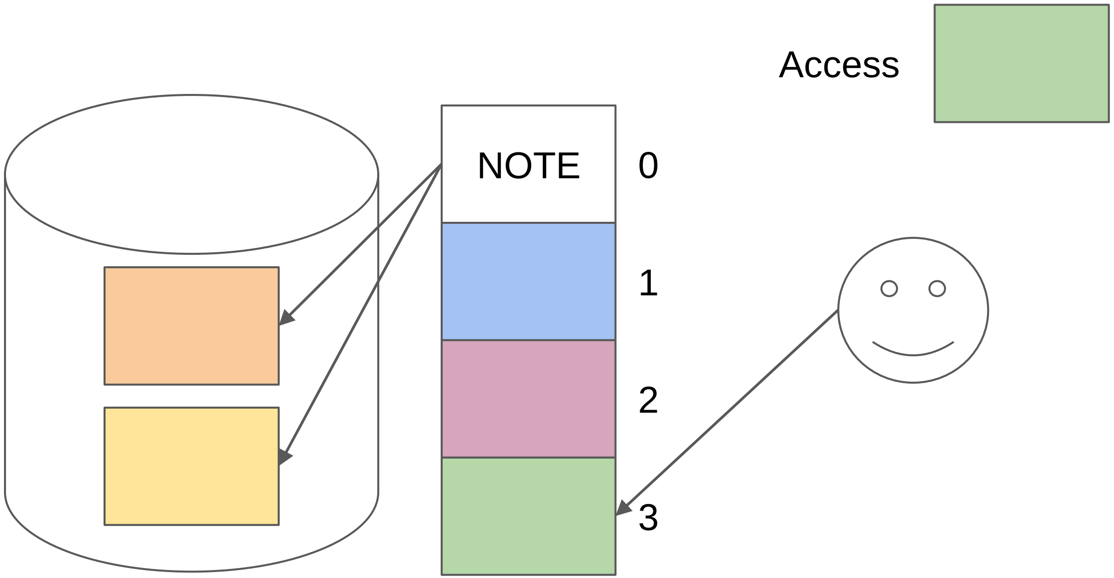
Paging - The World is Not Perfect



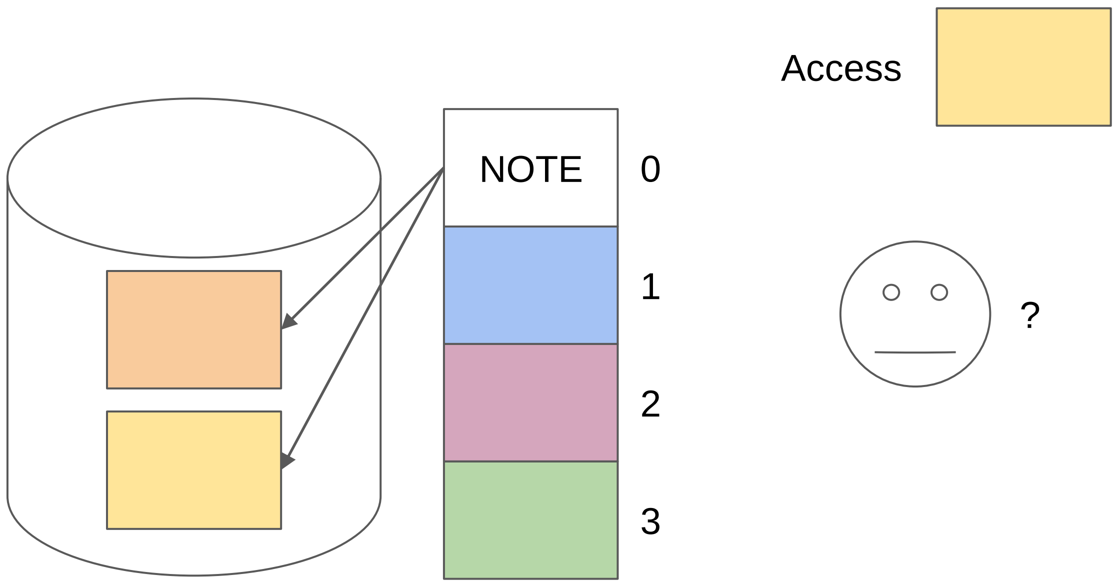
Paging



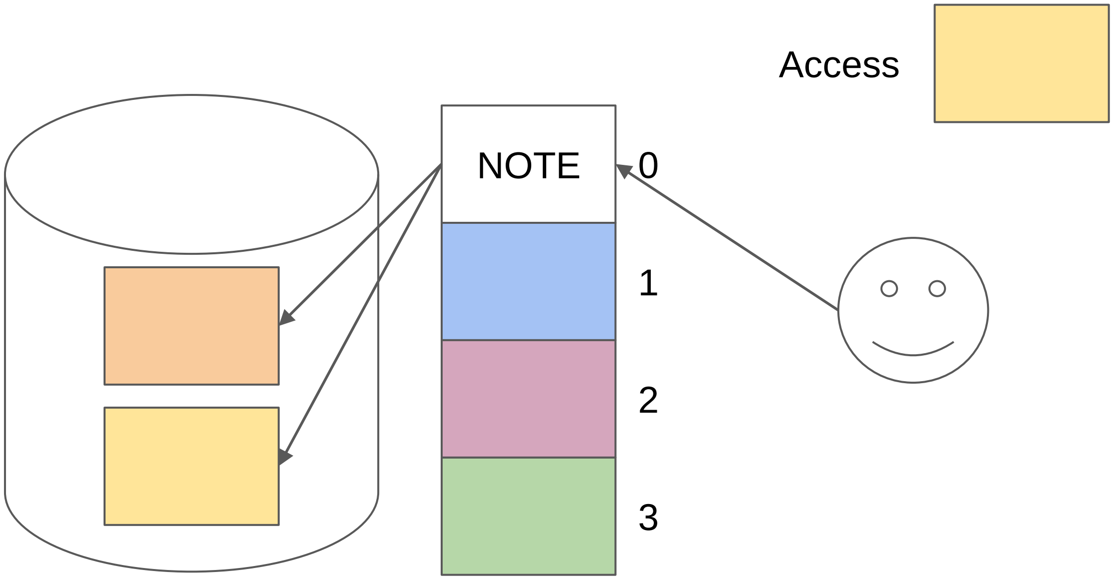
Paging



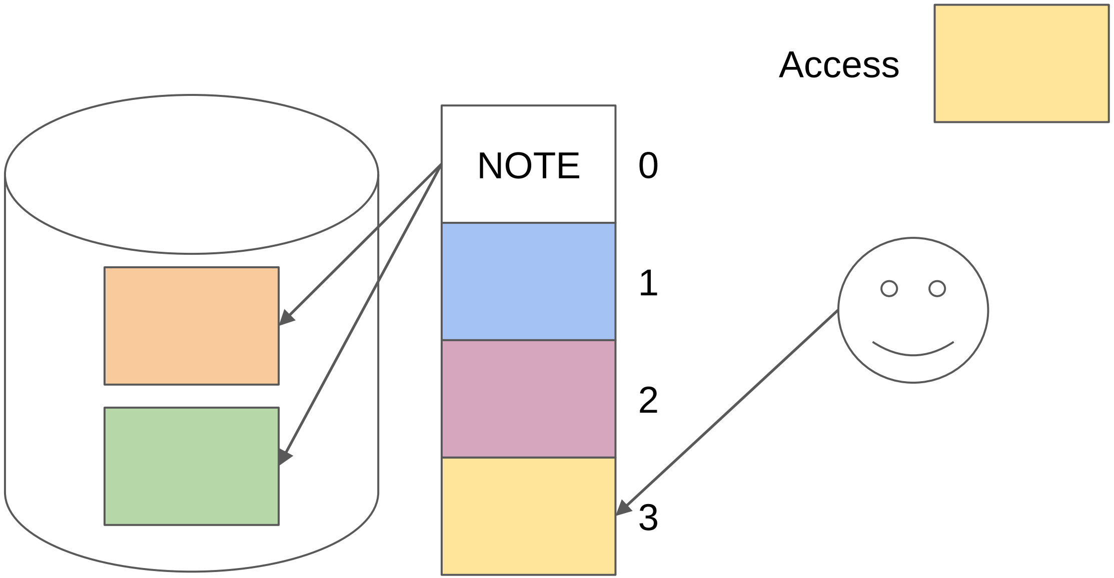
Paging



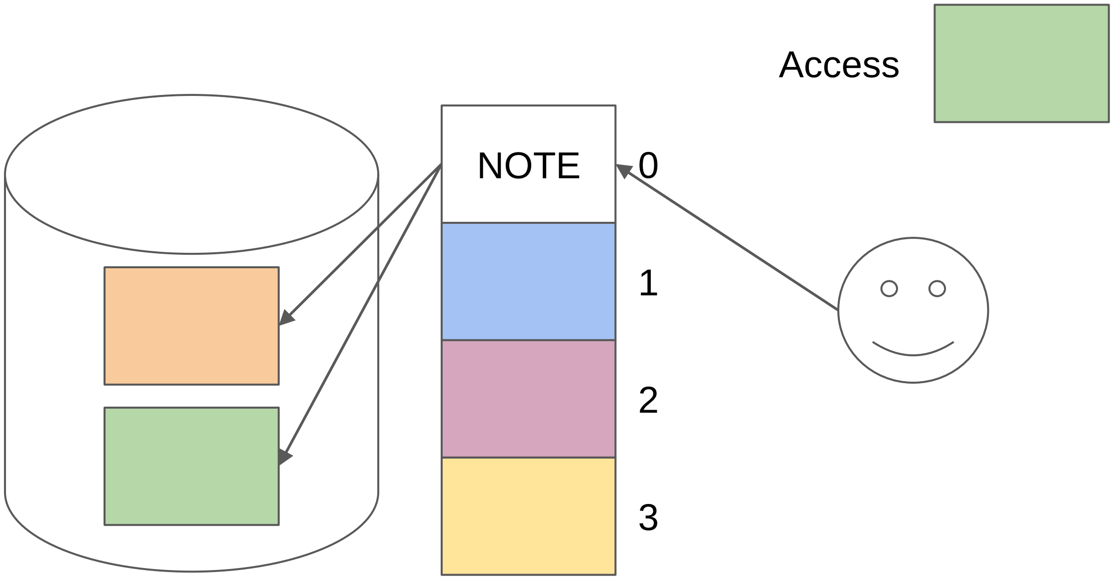
Paging - Swapping



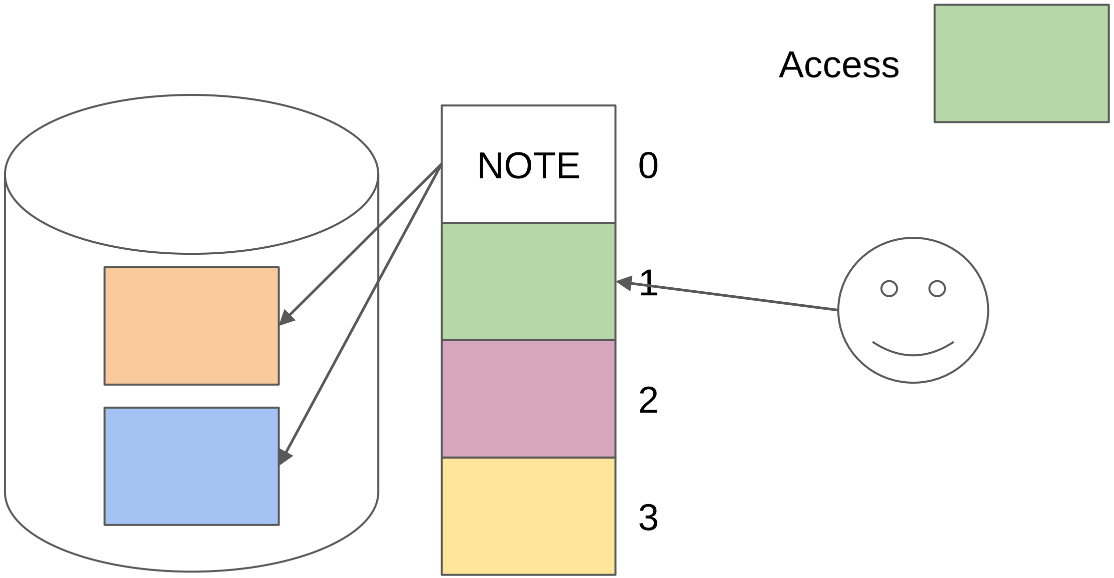
Paging - Swapping



Paging - Swapping



Paging - Swapping



Process Managed Paging - Problem

- The process expects data to always be in the same place

Consider:

```
void foo() {
    int i = 0;
    int green = 0;
    for(i = 0; i < 10; i++) {
        printf("%p\n"), &green);
    }
}
```

Expected Result:

```
0x7ffe4fcbe6d0
0x7ffe4fcbe6d0
0x7ffe4fcbe6d0
0x7ffe4fcbe6d0
0x7ffe4fcbe6d0
0x7ffe4fcbe6d0
0x7ffe4fcbe6d0
0x7ffe4fcbe6d0
0x7ffe4fcbe6d0
0x7ffe4fcbe6d0
```

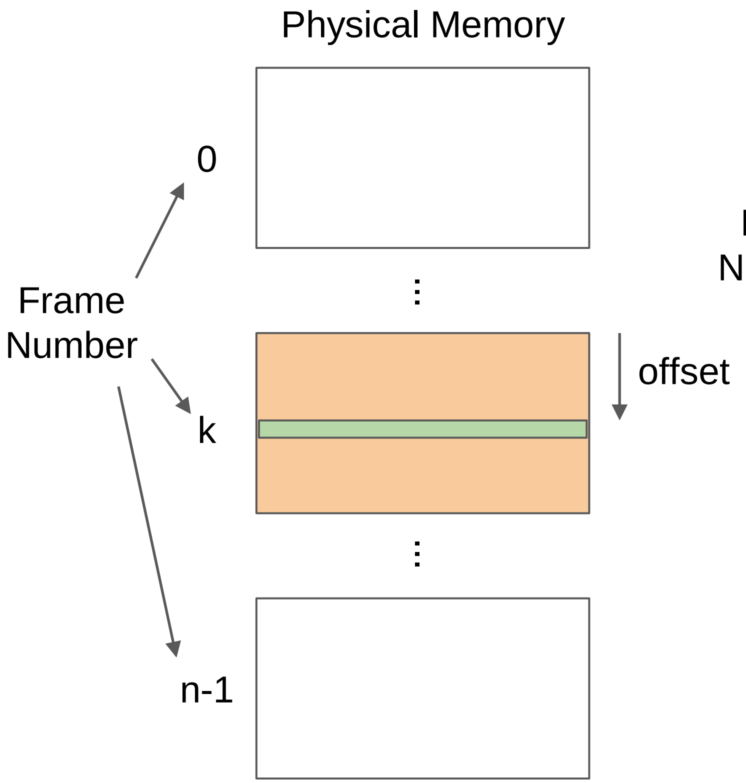
- What happens when there are multiple processes?
- Memory management should not have to be done by the process

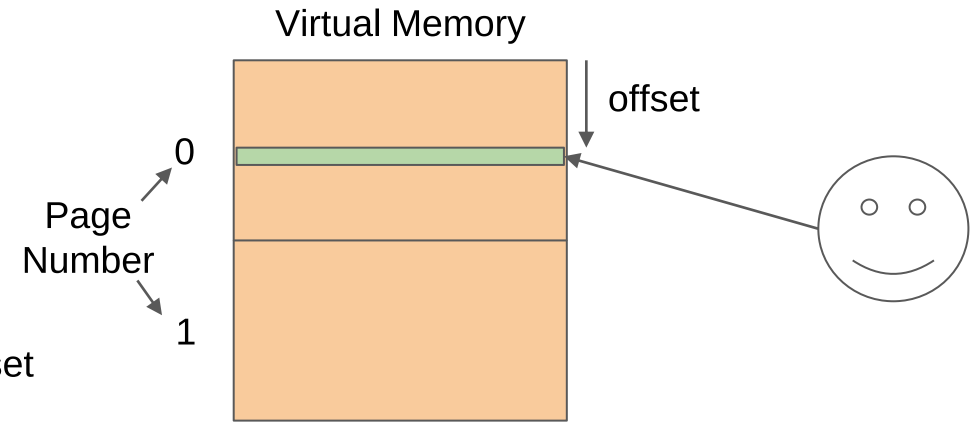
Operating System Managed Paging

Requirement - Allow programs to access the same location (virtual address - VA) for data even when the data is moved around in memory (physical address - PA)

- Divide program memory into a series of equal sized pieces - pages
- Divide physical memory into pieces (same size as pages) - frames
- Copy pages from disk to memory as they are needed
- Copy pages from memory to disk when there are no free frames
- Record which frame the page is located or that it isn’t in memory
- When a program accesses data at a virtual address, translate the access to the correct physical address.

Address Translation





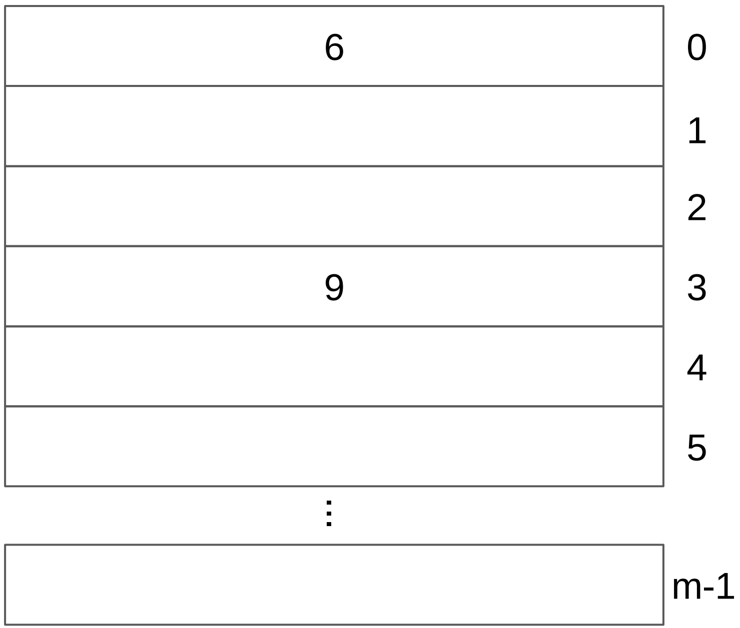
Fact:

- Page Size == Frame Size
- Page Offset == Frame Offset

Computation:

- Offset = Virtual Address % Page Size
- Physical Address = k * Page Size + Offset

Address Translation - Page Table



- The ‘notebook’ for storing where (which frame) pages are located
- Indexed by page number
- Stores frame number

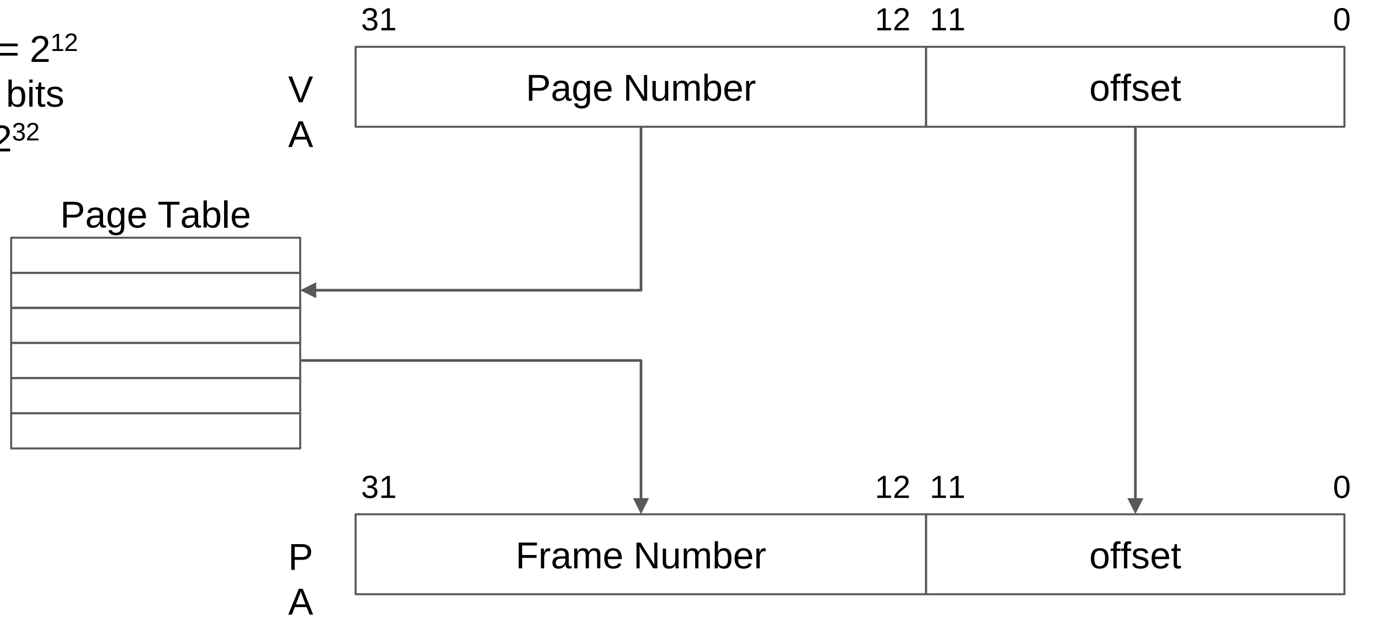
Fact:
- Page Size == Frame Size
- Page Offset == Frame Offset

Computation:
- Page Number = Virtual Address / Page Size
- Offset = Virtual Address % Page Size
- Frame Number = Page Table[Page Number]
- Physical Address =
  Frame Number * Page Size + Offset

Address Translation

- Math is hard - Hardware is easy
- Ensure page size is a power of 2 - address translation becomes routing bits

Page Size = 4096 = 2^{12}
Address Size = 32 bits
Max Addresses = 2^{32}



Paging - Address Translation Hardware

- Memory Management Unit (MMU)
  Translates virtual addresses to physical address

- Problem:
  - MMU needs page table to translate VA to PA
  - Page table is located in memory
  - MMU requires additional memory access to get page table entry

- Solution: Translation Lookaside Buffer (TLB)
  Cache within the MMU that stores page table entries

What’s in a Page Table Entry?

Address Structure:

31 — 12	11 — 0
Page/Frame Number	offset

- Page table entry size = Address size
- Present/Valid - Is the page in memory? Yes/No (1 bit)
- Protection
  - Are the contents of the page readable? Yes/No (1 bit)
  - Are the contents of the page writable? Yes/No (1 bit)
- Accessed - Was the page access recently? Yes/No (1 bit)
- Dirty - Has the page been modified since it’s been in memory? Yes/No (1 bit)

Page Fault / Page Replacement

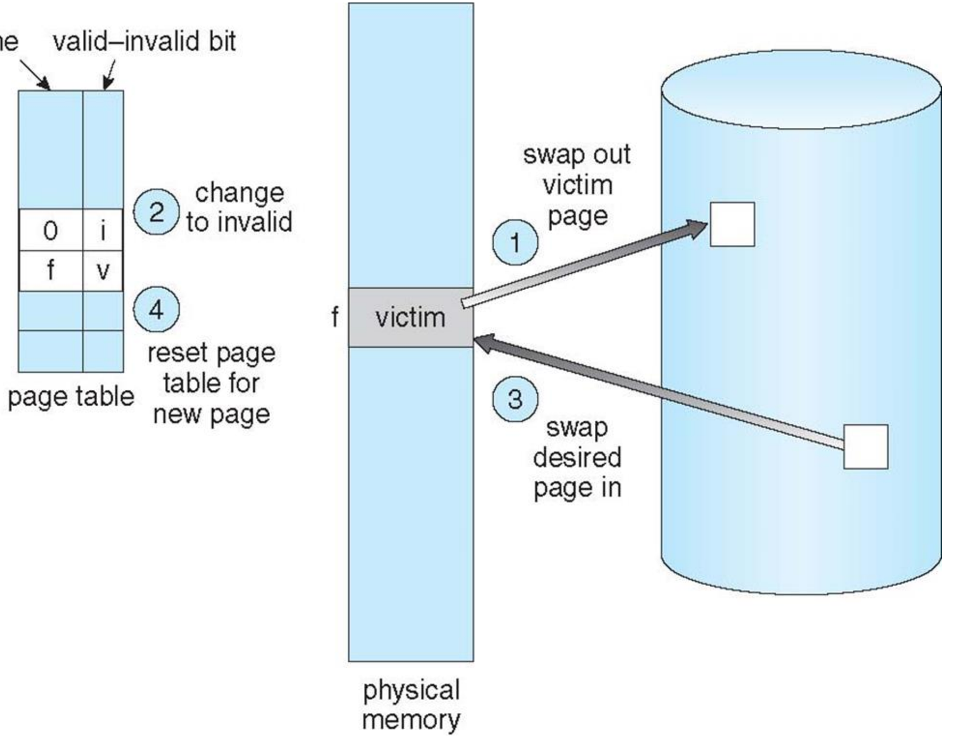
- Triggered by MMU when requested page is not in a frame
- MMU sends page fault interrupt
- Operating system services interrupt
    - Determines frame for page
        - Free frame if available
        - Chose a victim page to send to disk
    - Populates the frame with new page
    - Updates the page table
- What happens if present bit in page table is not set AND page does not exist?

Page Replacement

1. Find the location of the desired page on disk
2. Find a free frame:
    - If there is a free frame, use it
    - If there is no free frame, use a page replacement algorithm to select a victim frame
        - Write victim frame to disk if dirty
3. Bring the desired page into the (newly) free frame; update the page and frame tables
4. Continue the process by restarting the instruction that caused the trap

Note now potentially 2 page transfers for page fault

Page Replacement



Virtual Memory Advantages

- Isolation -
  Allows multiple processes memory without interfering with each other
- Abstraction -
  Allows a process to use all the memory they ‘want’
- Efficiency -
  - Locality - A process typically only uses a subset of pages (working set)
  - Sharing - Read only page (e.g. code) can be shared between multiple processes

Copy on Write and Page Pools

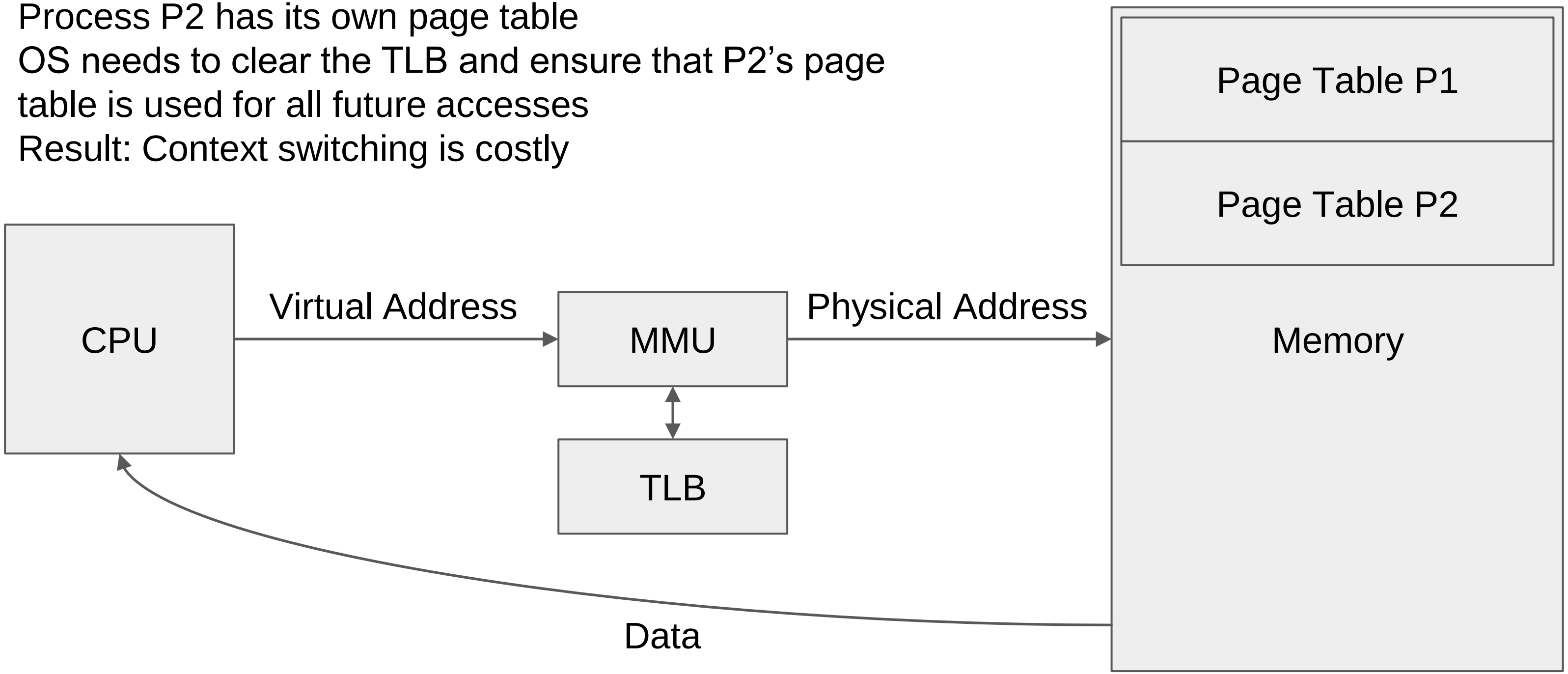
- Copy-on-Write (COW) allows both parent and child processes to initially share the same pages in memory
  - If either process modifies a shared page, only then is the page copied
- COW allows more efficient process creation as only modified pages are copied
- In general, free pages are allocated from a pool of zero-fill-on-demand pages
  - Why zero-out a page before allocating it?

Virtual Memory Dilemmas

- What happens when a process is context switched?
- How is a victim page chosen?
- What happens when a process working set is large? Thrashing
- What happens when the page table gets big
    - 32 bit address space and 4096 byte pages/frames => 2^{20} (1048576) page table entries
      4 bytes per entry => 4 MiB for page table
    - 64 bit address space and 4096 byte pages/frames => 2^{52} page table entries
      4 bytes per entry => 16 TiB for page table

Context Switching

- OS needs to context switch process P1 for process P2
- TLB contains page table entry cache for P1
- Process P2 has its own page table
- OS needs to clear the TLB and ensure that P2's page table is used for all future accesses
- Result: Context switching is costly



Frame Allocation and Page Replacement

- Frame Allocation - How many frames to give each process?

- Page Replacement algorithm
  - First in / First Out (FIFO)
  - Least Recently Used
  - Optimal

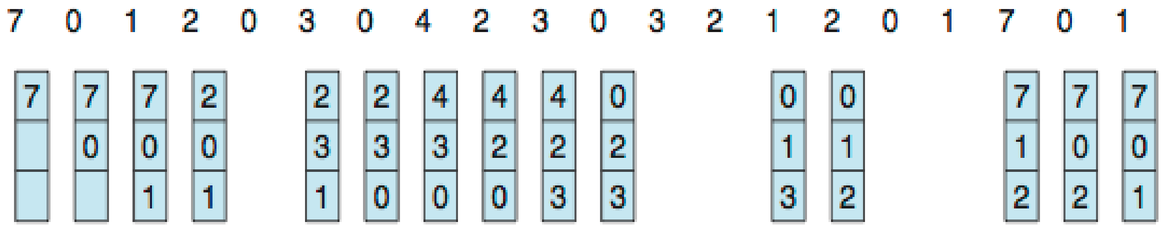
- Want lowest page-fault rate on both first access and re-access

First in / First Out

- The first page brought into memory is the first victim page
- Fast to choose a victim
    - Treat frames like a linked list and keep track of the head pointer

- Example: 7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

  3 frames (3 pages can be in memory at a time per process)



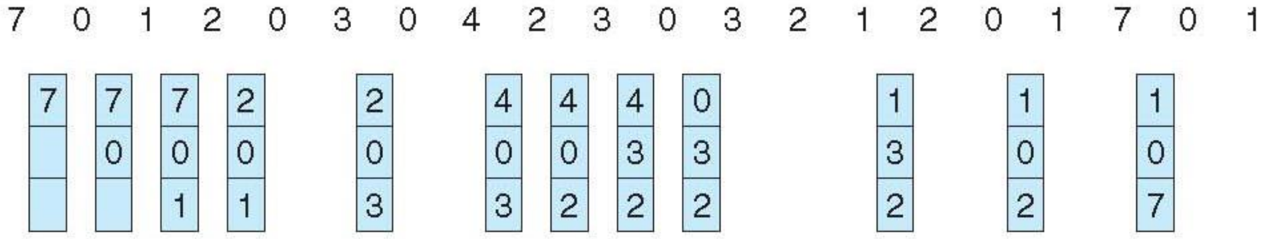
- 15 page faults

Least Recently Used

- The victim page is the ‘oldest’ page
- Idea take advantage of temporal and spatial locality
  - Temporal – If a process accesses a page, it’s going to access it again soon
  - Spatial - If a process accesses a page, it’s going to access a location close to it soon

- Requires lots of bookkeeping to keep track of access time

- Example:



- 12 page faults

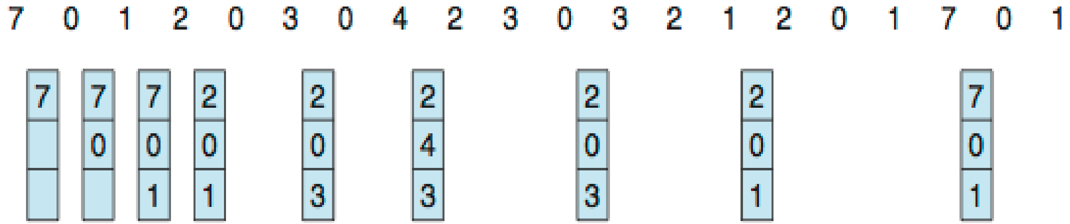
Optimal – Least Needed in the Future

- The victim page is the page that will not be used for longest period

- Idea take advantage of temporal and spatial locality
  - Temporal – If a process accesses a page, it's going to access it again soon
  - Spatial - If a process accesses a page, it's going to access a location close to it soon

- Not possible – Can't predict the future

- Example:



- 9 page faults

Virtual Memory Tradeoffs

- Increasing page size decreases size of the page table, increasing performance
  - BUT smaller pages result in less fragmentation and thus better performance
- Increasing page size results in better hard drive performance, as the majority of hard drive access time is seek and latency time, not transfer time
  - BUT a smaller page size may result in less total IO, therefore giving better performance
- All in all, it depends on both spatial and temporal locality relationships of the executing program
- General trend is toward larger page sizes

Thrashing

If a process does not have “enough” pages, the page-fault rate is very high. This leads to:

- low CPU utilization
- operating system thinks that it needs to increase the degree of multiprogramming
- another process added to the system