

# Operating systems

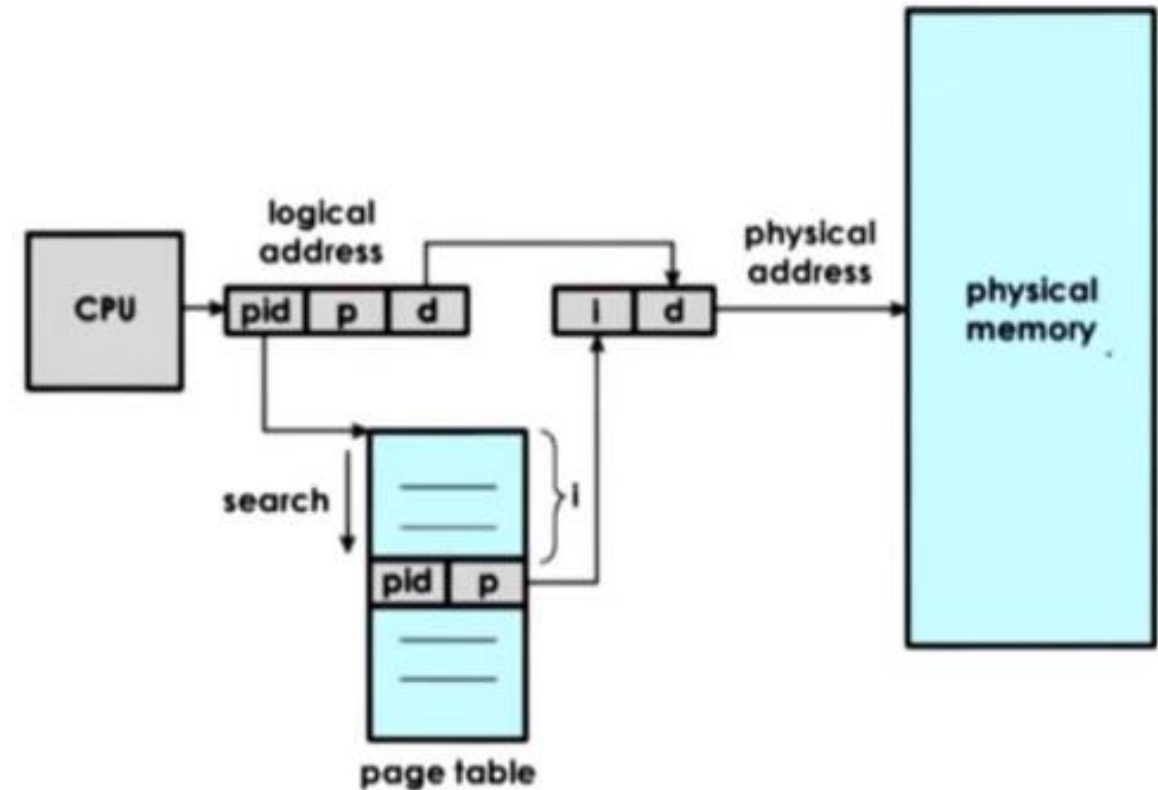
Week 6-2

Memory Management



# Inverted Page Tables

- Page table entries contain information, one for each element of the physical memory
- Each of the page table elements will correspond to one physical frame number
- To find the translation, the page table is searched based on the process ID and the first part of the virtual address, similar to what we saw before



# Inverted Page Tables... in details

- When the appropriate (pid) Process-id and V-address entry found into this page table. The index, the element where this information is stored will denote the physical frame number of the memory location that's indexed by this logical address
  - Combined with the actual offset to produce the physical address that's being coreference from the CPU
- The problem with inverted page tables is that we have to perform a linear search of the page table to see which one of its entry matches the (pidp) information that's part of the logical address was presented by the CPU



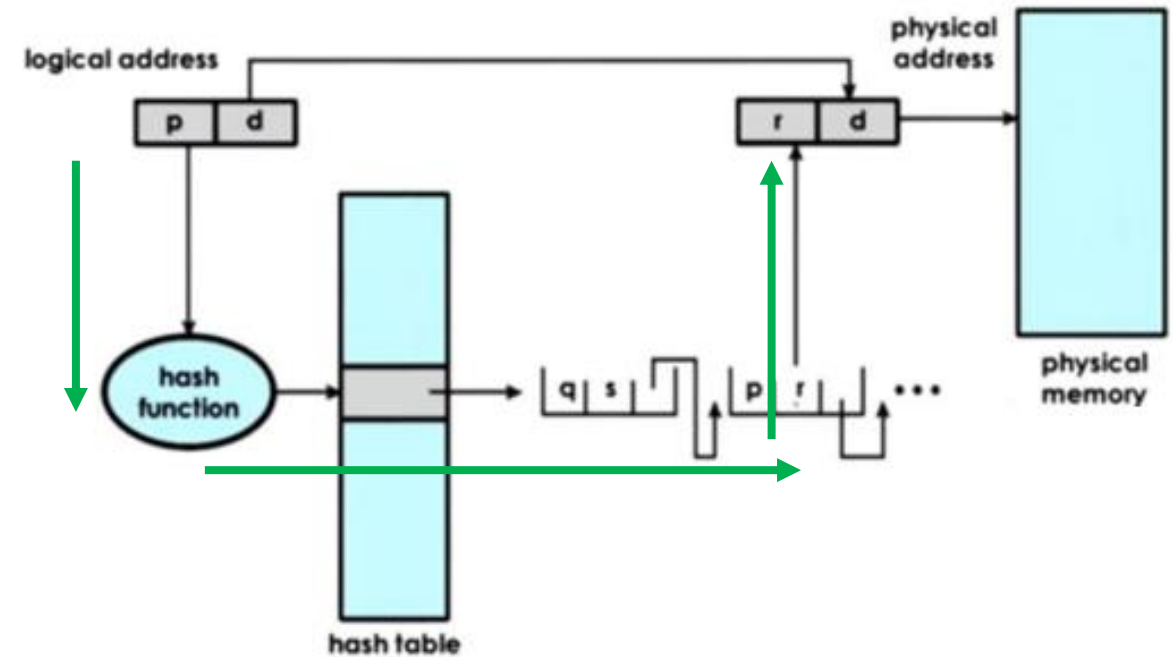
# Inverted Page Tables Memory Search

- Since the physical memory can be arbitrarily assigned to different processes, the table isn't really ordered. There may be two consecutive entries that represent memory allocated to two different processes, and there really isn't some clever search technique to speed up this process.
- In practice, the TLB will catch a lot of these memory references, so this detailed search is not performed very frequently



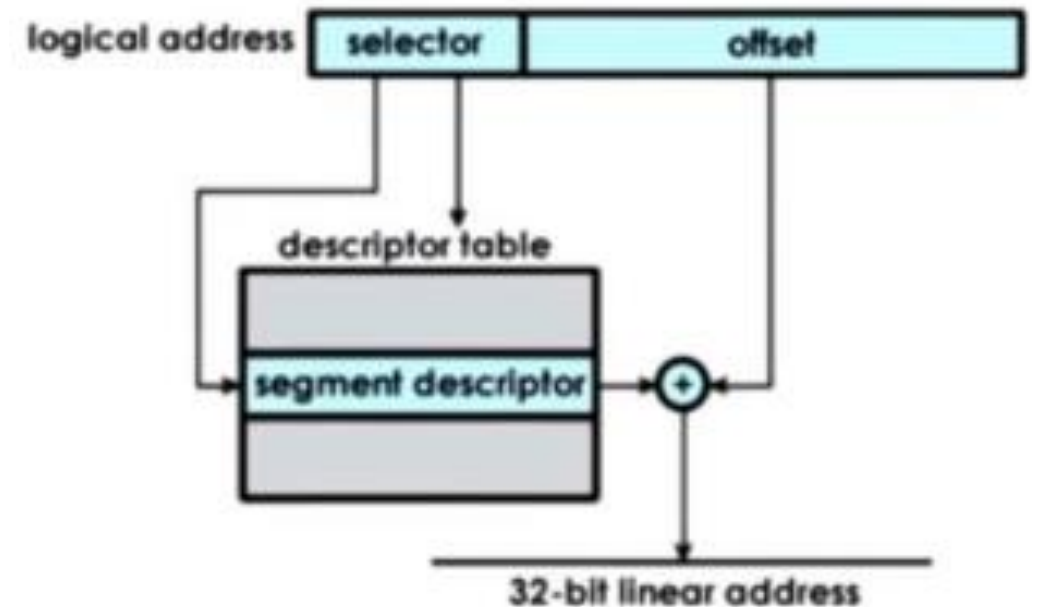
# Memory Search.... in details

- Hashing Page tables
- A hash is computed on a part of the address and that is an entry into the hash table that points to a linked list of possible matches for this part of the address
- That allows to speed up the process of the linear search to narrow it down to a few possible entries into the inverted page table, as a result speeding up the address translation



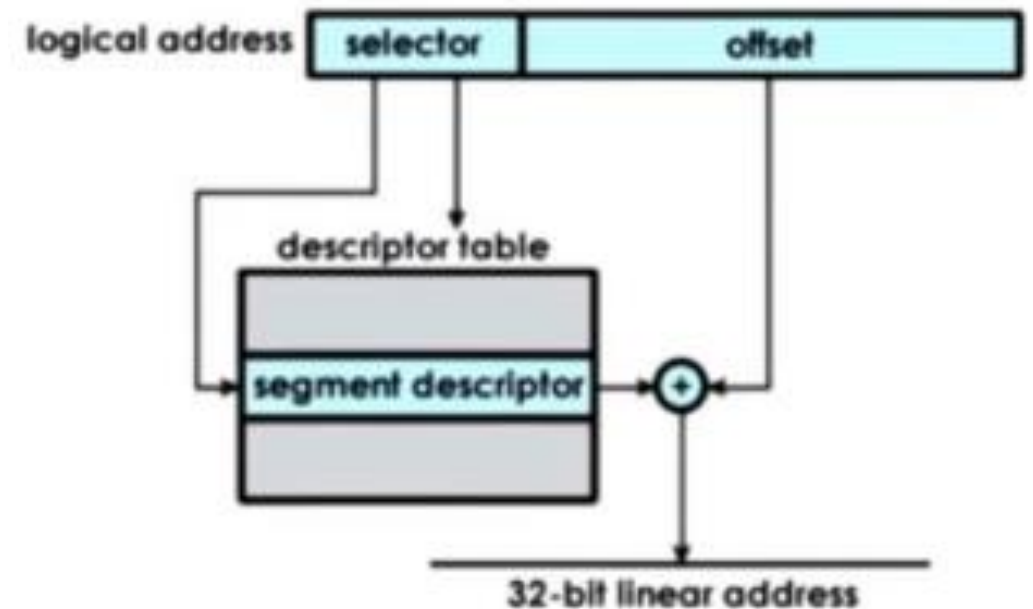
# Segmentation

- The address space is divided into components of arbitrary granularity of arbitrary size, and typically the different segments will correspond to some logically meaningful components of the address space like the code, the heap data, etc.
- A virtual address in the segmented memory mode includes a segment descriptor and an offset



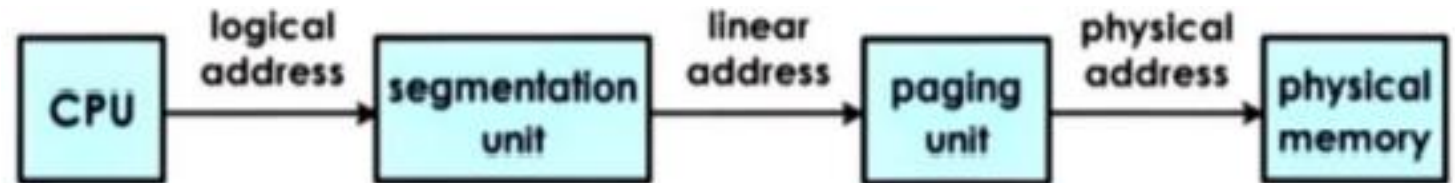
- Address = descriptor + offset
- The segment descriptor is used in combination with a descriptor table to produce information regarding the physical address of the segment
- The segment descriptor and the offset are combined to produce the linear address of the memory reference
- The segment would be defined by its base address and its limit registers, which implies also the segment size

Segment Size = Segment base + limit registers



# Segmentation and Paging

- In practice segmentation and paging are used together
- The address that's produced using this process (**linear address**), is then passed to the paging unit that is a multilevel hierarchical page table, where ultimately to compute the actual physical address that is used to reference the appropriate memory location



- Architecture applied:
  - 1A x86.32 → segmentation + paging e.g., Linux uses up to 8K per process /global segment
  - 1A x86.46 → Paging is the default but also segmentation if enabled





# Page Size

- Offset determines what is the total amount of addresses in the page, and therefore it determined the page size
  - 10-bit offset  $\rightarrow 2^{10}$ , 1KB page size
  - 12-bit offset  $\rightarrow 2^{12}$ , 4KB page size
- What are the page sizes in real systems?
  - Linux x86: 4KB(default), 2MB, 1GB

More bits in the address are used for these offset bits

	large	huge
page size	2 MB	1 GB
offset bits	21 bits	30 bits



# Large Page Sizes

- One benefit of using larger page sizes is that more bits in the address are used for offset bits, as a result fewer (pids) are used to represent the virtual page number, so there will be fewer entries that are needed in the page table
- The use of large page sizes will significantly reduce the size of the page table
- Another benefit is increased number of TLB hits, just because we'll be able to translate more of the physical memory using the TLB cache

	large	huge
page size	2 MB	1 GB
offset bits	21 bits	30 bits
reduction factor (on page table size)	x512	x1024



## POP\_Q

- What would be the downside to larger pages? Explain



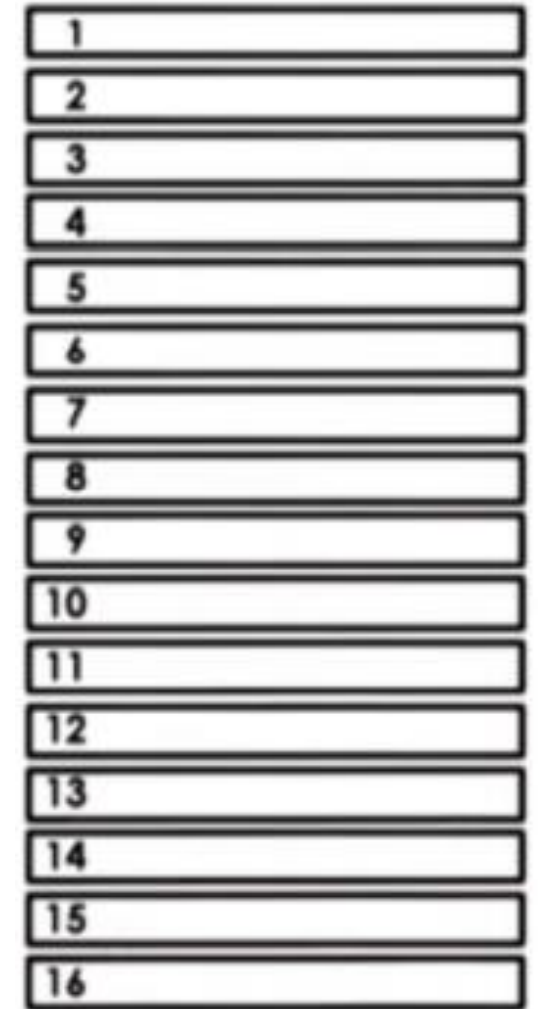
# Memory Allocation


- Explains how the operating system decides how to allocate a particular portion of the memory to a process
- This is the job of the memory allocation mechanisms that are part of the memory management subsystem of an operating system
- Memory allocation incorporates mechanisms that decide what are the physical pages that will be allocated to particular virtual memory regions and validity access checks
- Memory allocators can exist;
  - Kernel level → for allocating memory regions, static process state and kernel state
  - User-level → for dynamic process state allocation



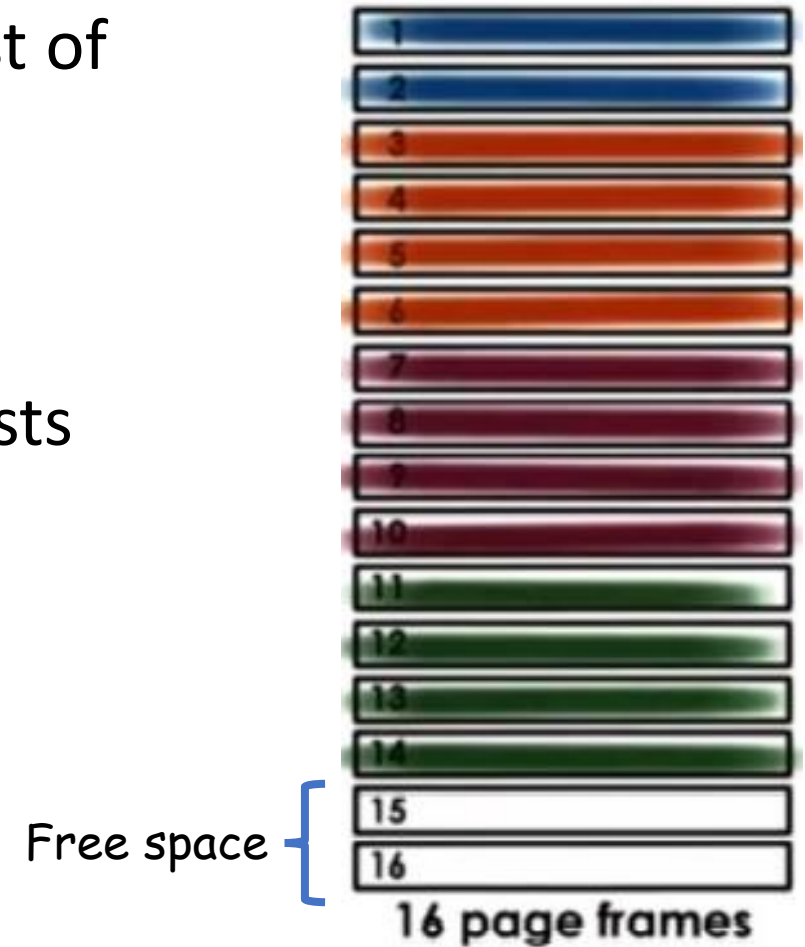
# Memory Allocation Challenge

- Consider a page-based memory manager that needs to manage 16 physical page frames
- Assume this memory manager tasks requests of sizes 2- or 4-page frames, and let's say it's facing the following sequence of memory requests:

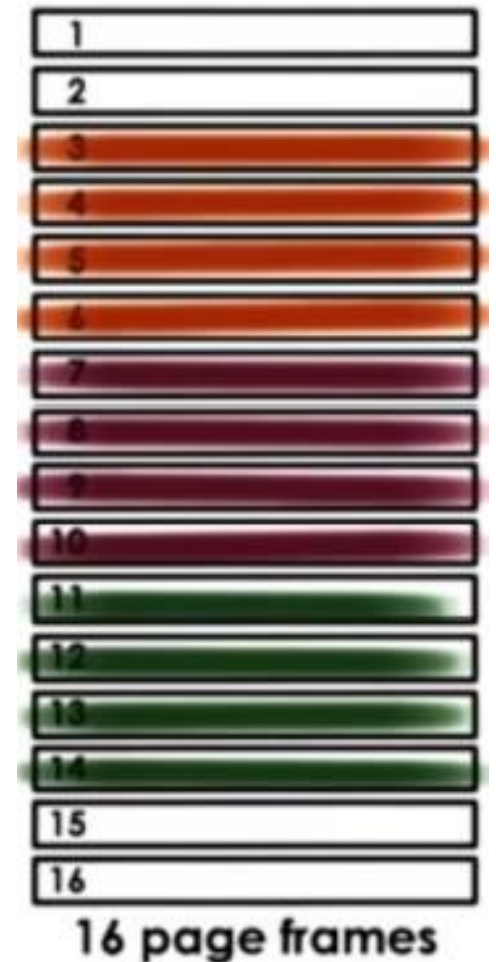


16 page frames 

- The first memory allocation is for the request of 2-page frames and then the rest of the requests are for 4-pages each
- The memory allocator allocates these requests in the order of requests



- Next, the two pages that were initially allocated now **freed**
- **Can you see the problem?**
  - If at this point a next request comes to allocate four pages, there are four free pages in the system
  - However, this allocator cannot satisfy this request since these pages are not continuous (assuming that this specific request for continuous pages)



# External Fragmentation Problem

- Where we have multiple interleaved **allocate** and **free** operations and as a result of them, we have holes of free memory that's not continuous. And therefore, requests for largest consecutive memory allocations cannot be satisfied





# Alternative ...

In the previous example:

- The allocator had a policy in which the free memory was handed out to consecutive requests in a sort of first come first served manner
- Consider an alternative in which the allocator knows about the requests that are coming (consecutive regions of 2- and 4- page frames)
- In this case, when a second request for an allocating 4- pages occurs, the memory allocator isn't allocating it immediately after the first allocation but instead is leaving some gap



- The second allocation for pages comes in at a granularity of 4-pages, and then the allocations are satisfied
- Later, when the free request comes in, these two first pages are freed. The system again has four free pages however, they're consecutive



16 page frames



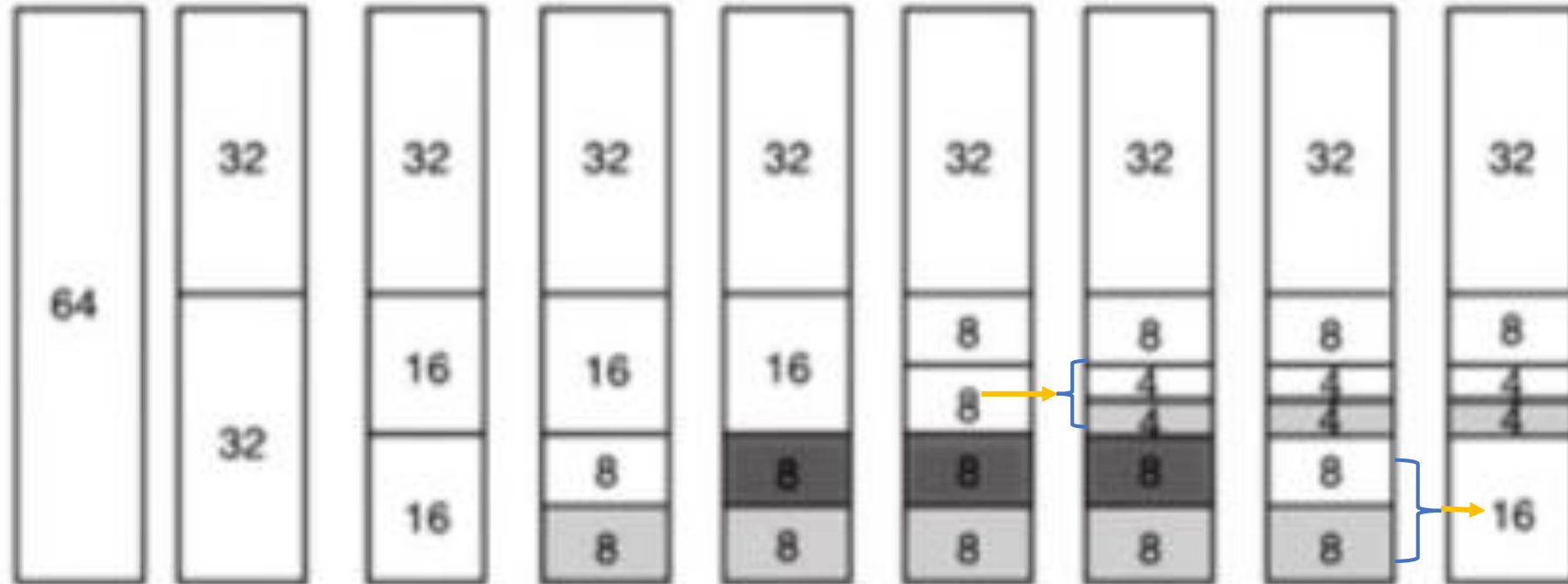
# Linux Kernel Allocators.... Buddy allocator

Linux kernel basic allocation mechanisms:

- Buddy allocator
  - Starts with  $2^x$  area
  - On request,
    - The allocator will subdivide this large area into small  $2^x$  chunks such that every one of them is also a power of two
    - It will continue subdividing until it finds the smallest chunk that's of a size that's a power that can satisfy the request



# Buddy allocator Example



- Fragmentation still exists in the buddy allocator, but its benefits are that when a request is freed, it has a way to quickly find out how to aggregate data
- The aggregation of the free areas can be performed really well and really fast



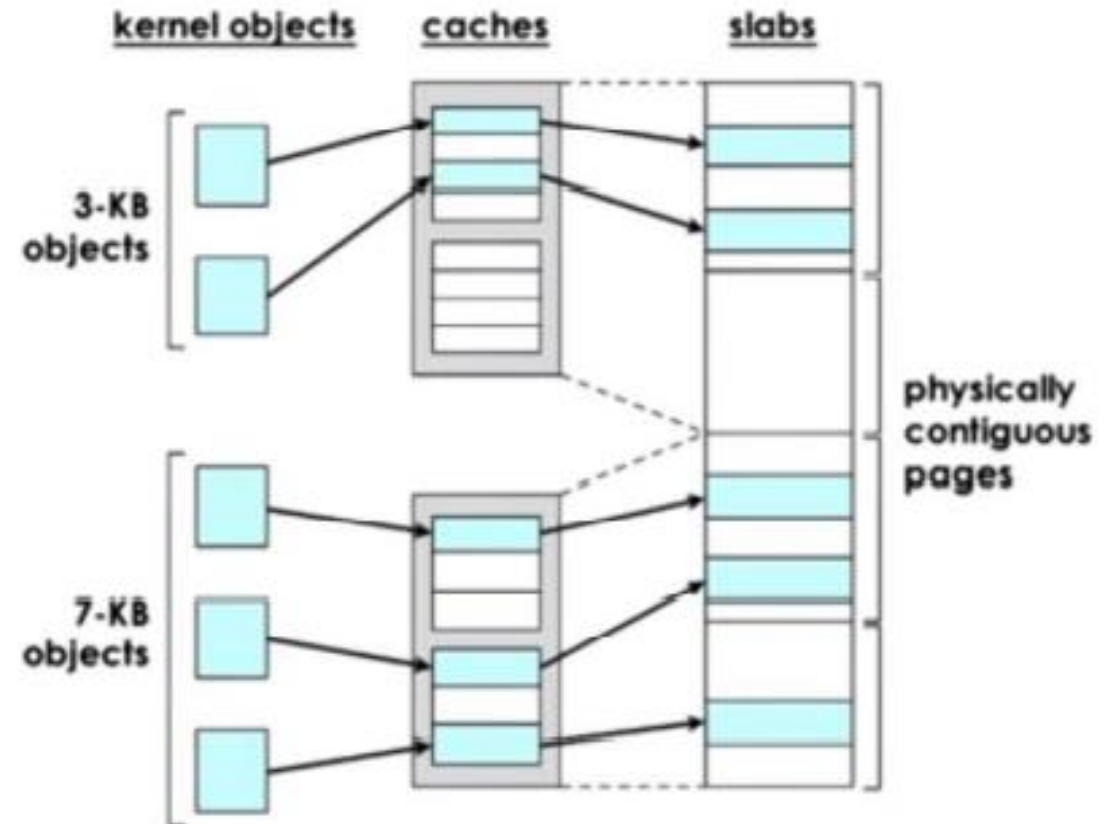
# Buddy is not Buddy all the time

- Define that allocations using the buddy algorithm have to be made in a granularity of a power of two, means that there will be some internal fragmentation using the buddy allocator, this is particularly a problem because there are a lot of data structures that are common in the Linux kernel that are not of a size that's close to a power of two
- task data structure, `task_struct` 1.7k
- To fix this issue, Linux also uses the slab allocator in the kernel



# Linux Kernel Allocators.... Slab allocator

- The allocator builds custom object caches on top of slabs
- The slabs themselves represent continuously allocated physical memory
- When the kernel starts, it will pre-create caches for different object types



# Linux Kernel Allocators.... Slab allocator

- For instance, it will have a cache for a `task_struct` or for the directory entry objects. Therein, when an allocation comes from a particular object type it will use one of the elements in the cache
- If none of the entries is available, then the kernel will create another slab and it will reallocate an additional portion of continuous physical memory to be managed by this slab allocator
- The benefit of this slab allocator is that it avoids internal fragmentation. These entities that are allocated in the slab, they're of the exact same size as the common kernel objects. Also, external fragmentation is not really an issue



## Last word ..

The combination of the slab allocator and the buddy allocator that are used in the Linux kernel are effective methods to deal with both the fragmentation and the free memory management challenges that are present regarding memory management in operating systems





# Demand Paging

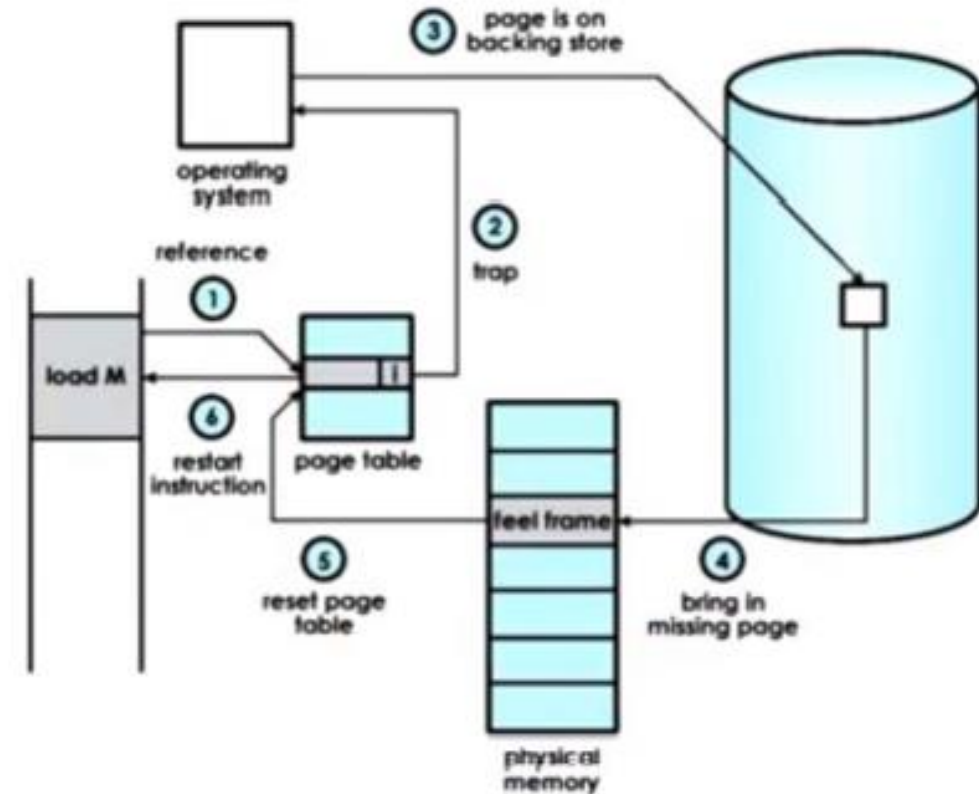
- Since the physical memory is much smaller than the addressable virtual memory, allocated pages don't always have to be present in physical memory
- Instead, the backing physical page frame can be repeatedly saved and restored to and from some secondary storage like disks. This process is referred to as **paging or demand paging**
- Traditionally with demand paging, pages are moved between main memory and a storage device such as disk, where a swap partition resides or even flash device or the memory of another node



# Demand Paging in Details

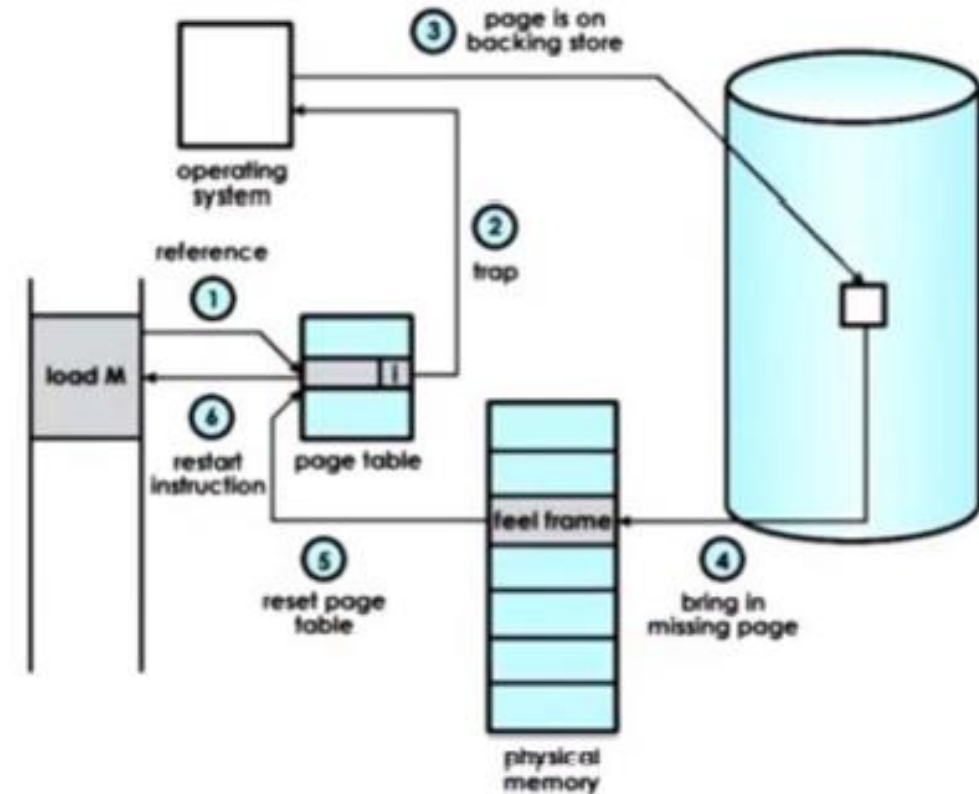
1- When a page is not present in memory, it has its present bit in the page table entry that's set to zero

2- When there is reference to the page, then the memory management unit will raise an exception, and that will cause a trap into the operating system kernel



# Demand Paging in Details

- On an access, the memory management unit will raise an exception, that's called the page fault, and this will be pushed into the operating system so it will trap into the operating system kernel
- At that point, the OS kernel can determine that this exception is a page fault

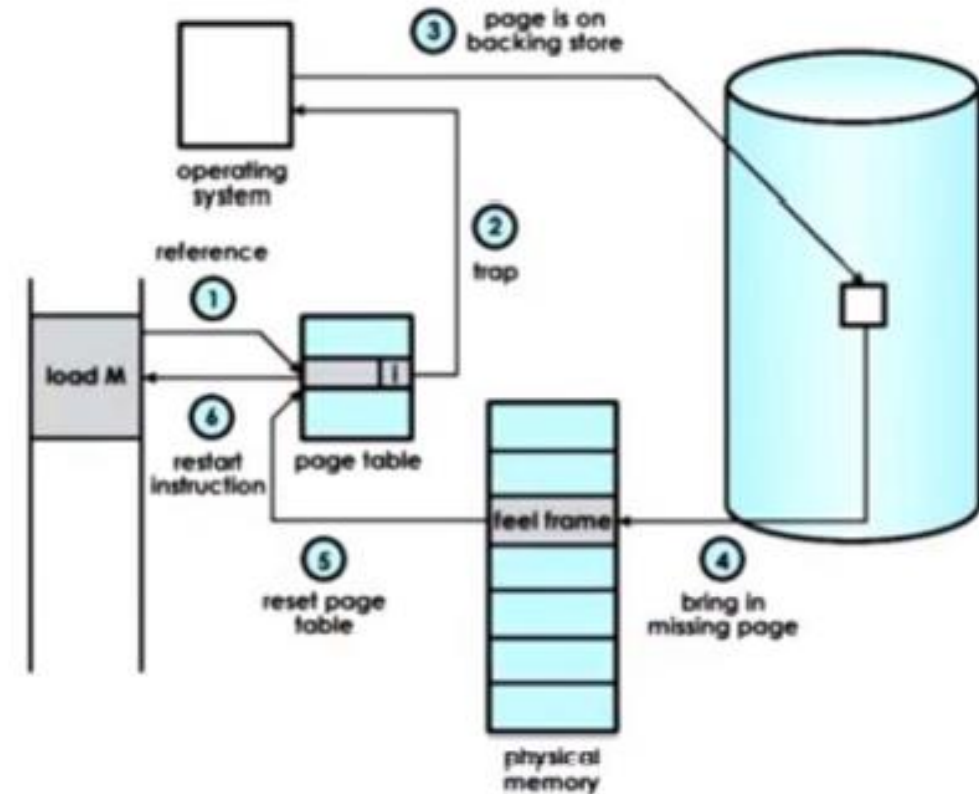


# Demand Paging in Details

3- It can determine that had previously swapped out this memory page onto disk.

4- It can establish what is the correct disk access that needs to be performed and it will issue an I/O operation to retrieve this page

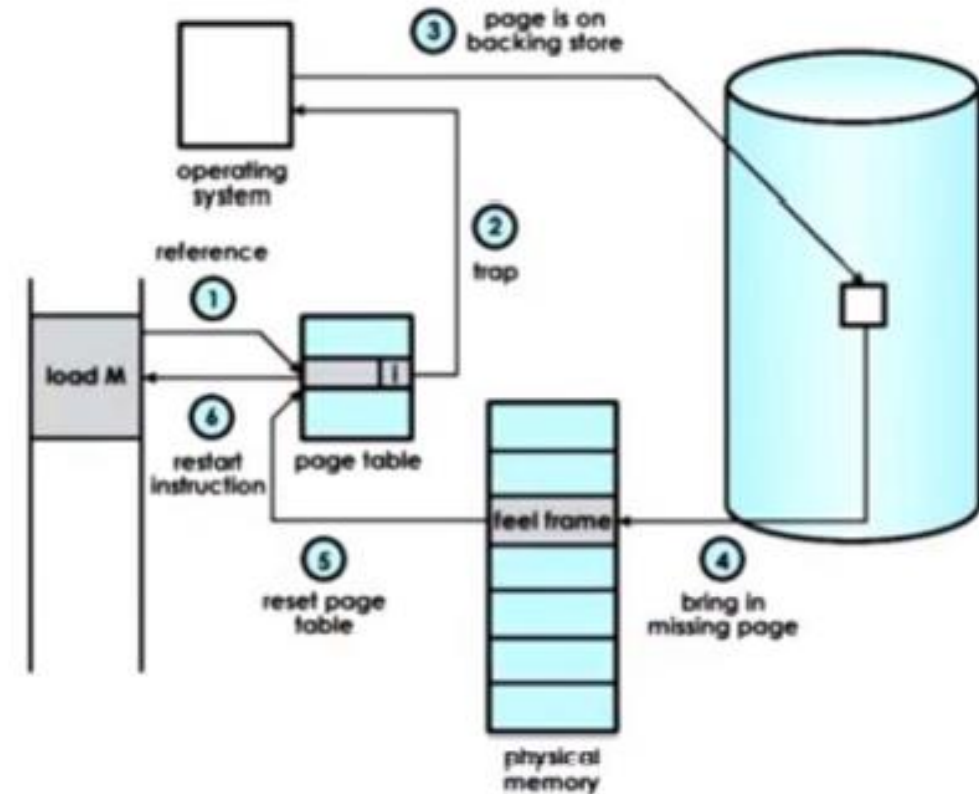
Once this page is brought into memory, the OS will determine a free frame where this page can be placed



# Demand Paging in Details

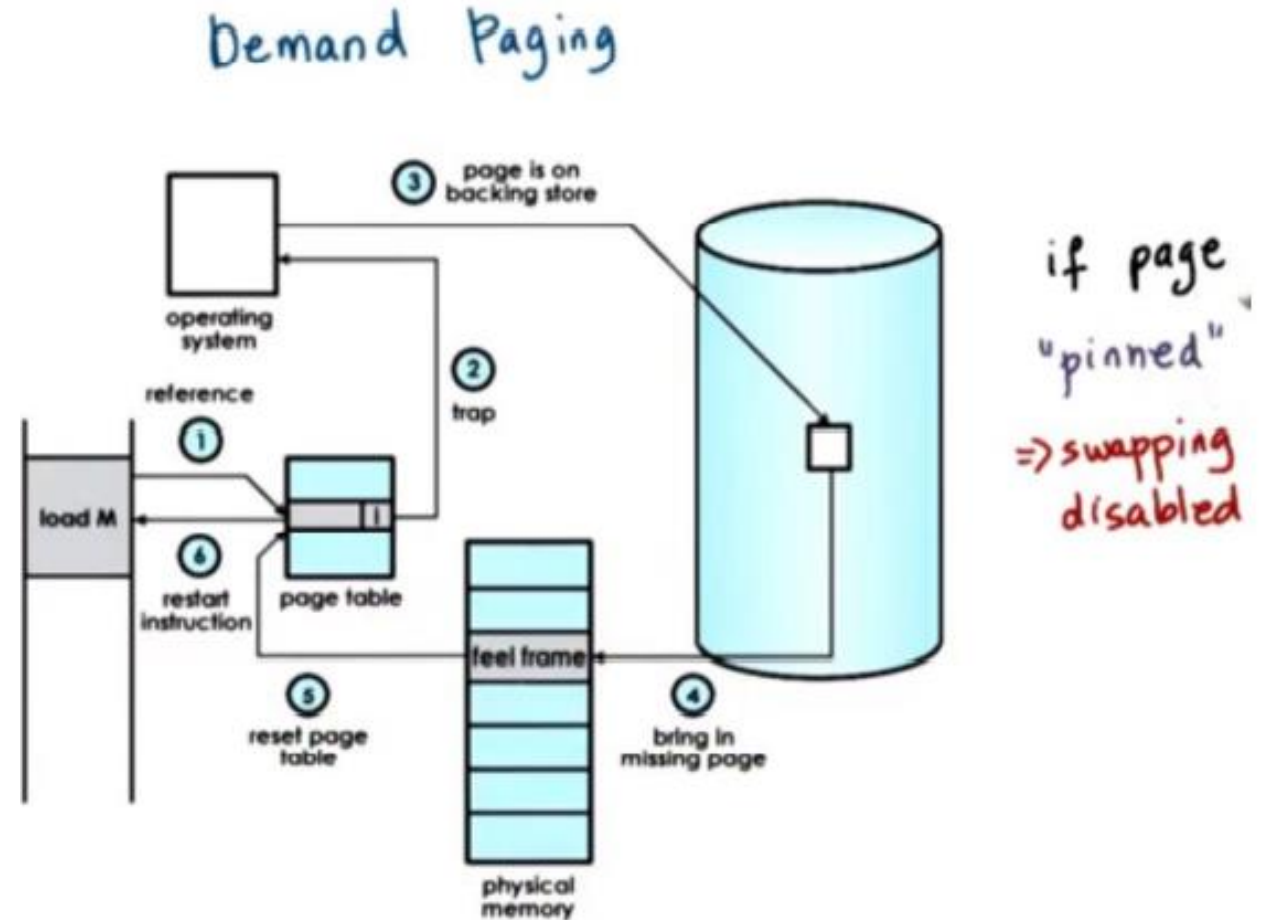
5-And it can use the page frame number for that page to appropriately update the page table entry that corresponds to the virtual address of that page

6- At that point, control is pushed back into the process that caused this reference, and the program counter will be restarted with the same instructions so that this reference will now be made again. Except at this point, the page table will find valid entry with a reference to this particular physical location



# Demand Paging in Details

If, for whatever reason, we require a page to be constantly present in memory, or if we require it to maintain the same physical address during its lifetime, then we will have to **pin the page**, and at that point we basically disable the swapping



# Page Replacement

- When should pages be swapped out of physical memory and onto disk?
  - Periodically when the amount of occupied memory reaches a particular threshold the operating system will run some page out daemon that will look for pages that can be freed
  - Also, when the CPU usage is below a certain threshold so as not to disrupt the execution of some applications too much
- Which pages should be swapped out?
  - The pages that will not be used in the future are the one that should be swapped out.....how we would know such pages!!!!
  - Historic information at how recently or how frequently has a page been used, and use that to inform a prediction regarding the page's future use



# Feeding up Physical Memory .... Access bit

- The intuition here is that a page that has been used most recently is more likely to be needed in the immediate future, whereas a page that hasn't been accessed in a very long time is less likely to be needed
- This policy is referred to as the LRU policy, least recently used, and it uses the **access bit** that's available on modern hardware to keep track of the information to whether or not the page is referenced or not





# Feeding up Physical Memory .... Dirty bit

- Other useful candidates for pages that should be freed up from physical memory are the pages that don't need to be written out to disk, to secondary storage because the process of writing pages out to the secondary storage consumes cycles, so we'd like to avoid the overhead of the memory management
- The operating system can rely on the **dirty bit** that's maintained by the MMU hardware that keeps track of which particular page has been modified. So not just accessed and referenced however, modified during a particular period of time



# Feeding up Physical Memory .... Non-swappable pages

- In addition, there may be certain pages containing important kernel state or used for I/O operations that should never be swapped out. Then making sure that these pages are not considered by whatever replacement algorithms are executed in the operating system is going to be important



# Feeding up Physical Memory .... LINUX

- In Linux and most OS's, number of parameters are available to allow the system administrator to configure the swapping nature of the system
  - Thresholds that control when our page is swapped out
  - Parameters such as how many pages should be replaced during a period of time
  - Also, Linux categorizes the pages into different types and then that helps narrow down the decision process
  - Finally, the default replacement algorithm in Linux is a variation of the LRU policy and it gives a second chance. It basically performs two scans of a set of pages before determining which one/s are really the ones that should be swapped out and reclaimed



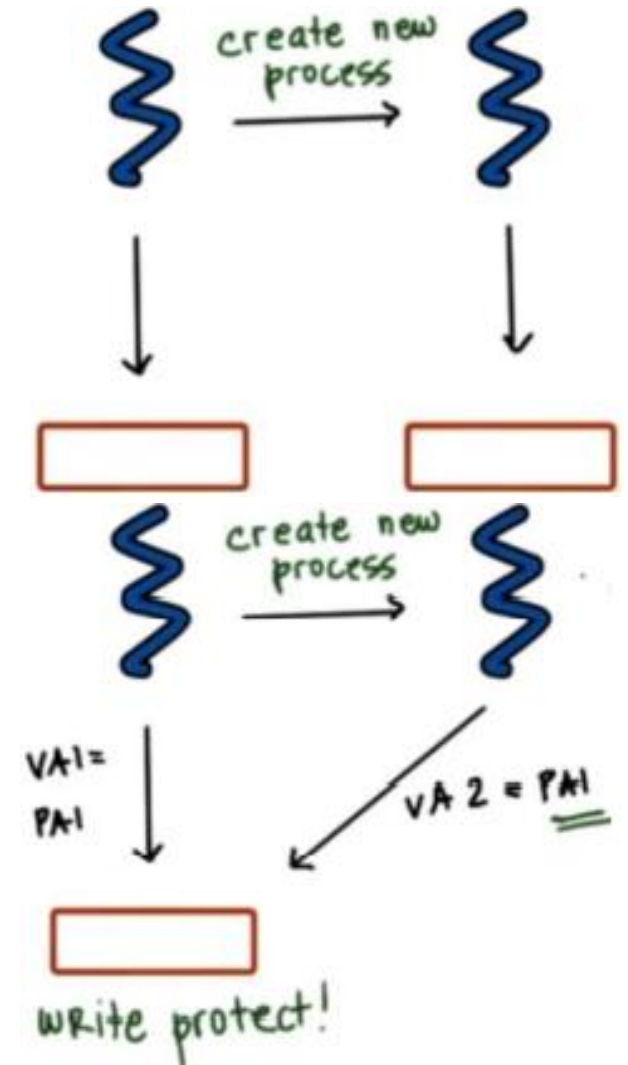
# Copy-On-Write COW

- So far, we saw that OS rely on the memory management unit hardware, to perform address translations and to also validate the accesses to enforce protection in similar mechanisms
- But the same hardware can also be used to build number of other useful services and optimizations, beyond just the address translation. One such mechanism is called Copy-on-Write or COW
- Failure management through checkpointing ....

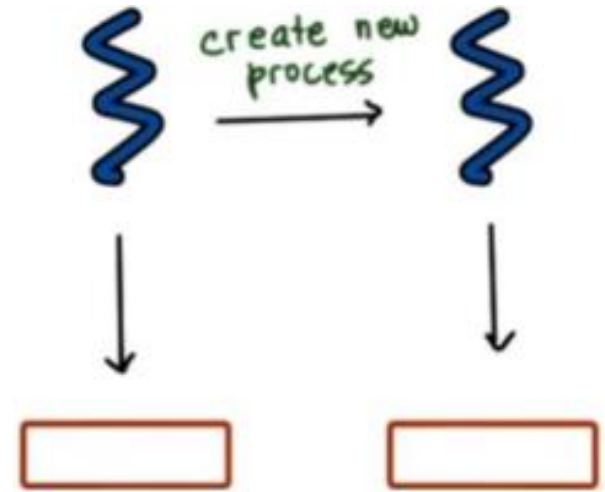


# Copy-On-Write COW

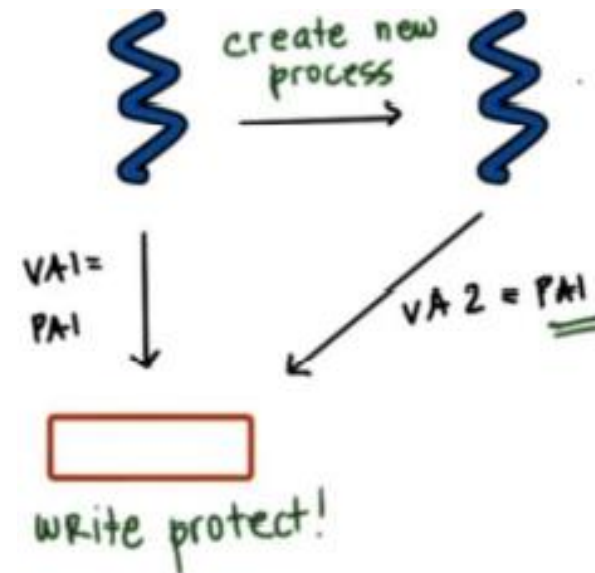
- To create a new process, a parent process needs to be created by copying its entire address space. However, many of the pages are static, they don't change. So, it's not clear why we should keep multiple copies
- In order to avoid unnecessary copying on creation. The virtual address space of the new process or portions of it at least, will be mapped (point) to the original page that had the original address space content



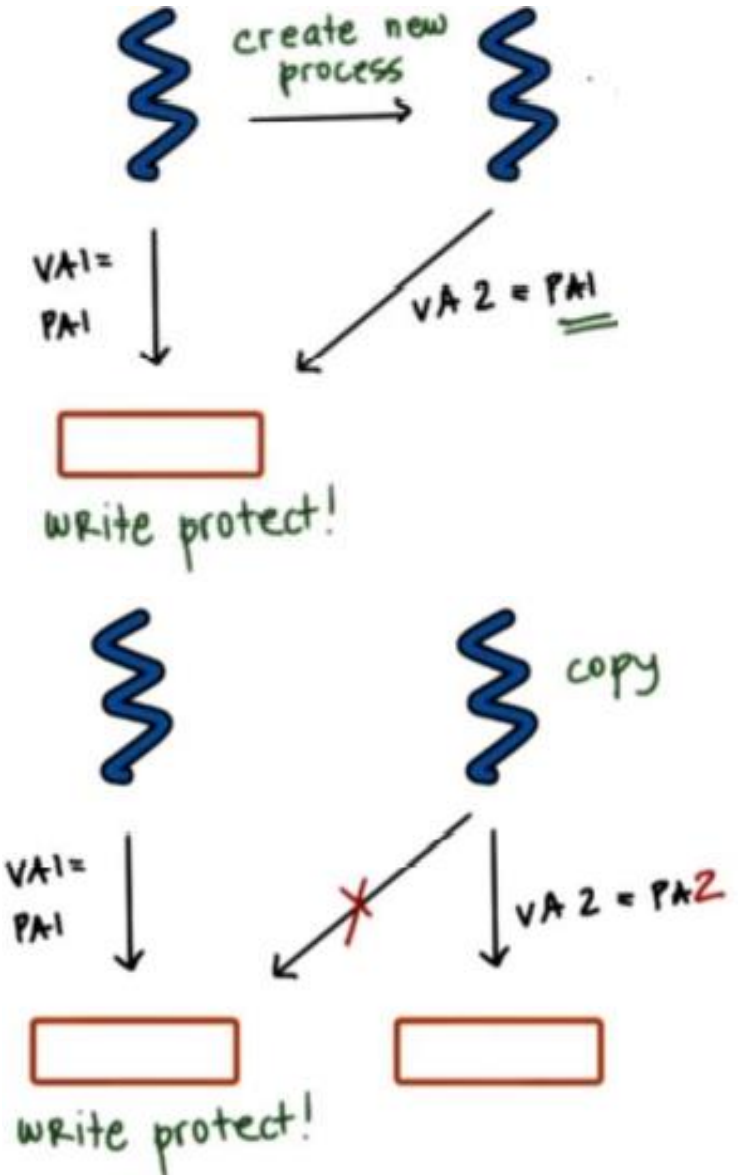
- On process create (problem case)
  - Copy entire parent address space
  - Many pages are static, why to keep multiple copies?



- On process create (solution)
  - Map new virtual address to original page
  - Write protect original page



- On process create (solution)
  - On **READ**
    - Map new virtual address to original page
    - Write protect original page
    - Save memory and time to copy
  - On **WRITE**
    - Page fault and copy
    - Page copy cost only if necessary (COW)



# Failure Management Checkpointing

- Checkpointing is a technique that's used as part of the failure and recovery management that operating system or systems software in general supports
- The idea behind checkpointing is to periodically save the entire process state
- The failure may be unavoidable however with checkpointing, the process doesn't have to be restarted from the beginning. It can be restarted from the nearest checkpoint point. And so, the recovery will be much faster





# Checkpointing in Practice

- A simple approach to checkpoint would be to pause the execution of the process and copy its entire state
- A better approach will take advantage of the hardware support for memory management and will try to optimize the disruption the checkpointing will cause on the execution of the process
- Using the hardware support, we can **write protect** the entire address space of the process and try to copy everything at once



# Checkpointing in Practice ... Diffs

- Since the process will continue executing, we will not attempt to pause it, it will continue dirtying pages
- So, then we can track the dirtied pages, again using the hardware MMU support and we will copy only the **diffs**, only those pages that have been modified. That will allow us to provide for an incremental check point
- This will result making recovery process more complex since we will have to rebuild the image of the process using multiple such diffs
- Or also, in the background, these diffs can be aggregated to produce more complete checkpoints of the process



# From Checkpointing to .... 1- Debugging

- The basic mechanisms used in checkpoint can also be used in other services. For instance, debugging relies often on a technique called **Rewind-Replay**
- Here rewind means that we will restart the execution of the same process from some earlier point (checkpoint), and then we will move forward and see whether we can establish the error to define what is the bug in our program
- We can gradually go back to older and older checkpoints until we find the error



# From Checkpointing to .... 2- Migration

- Checkpoint the process to another machine and then we restart it on that other machine. It will continue its execution on the other location
- implemented as if we are performing repeated checkpoints in a fast loop until ultimately, there is so few dirtied state from the process that something like the pause and copy approach becomes acceptable
- This is useful in scenarios such as disaster recoveries, to continue the process on another machine that will not crash
- Resulting in saving on power and energy or utilize resources better



# The End

- Pop-Q slide 11 due Sat 8:00pm
- Quiz 4 on 7-1
  - Material weeks 5 and 6