Memory

Address Spaces and Memory

- Process
  - One or more thread
  - One address space
- Thread
  - Stream of execution
  - Unit of concurrency
- Address space
  - Memory space that threads use
  - Unit of data

Address Space Abstraction

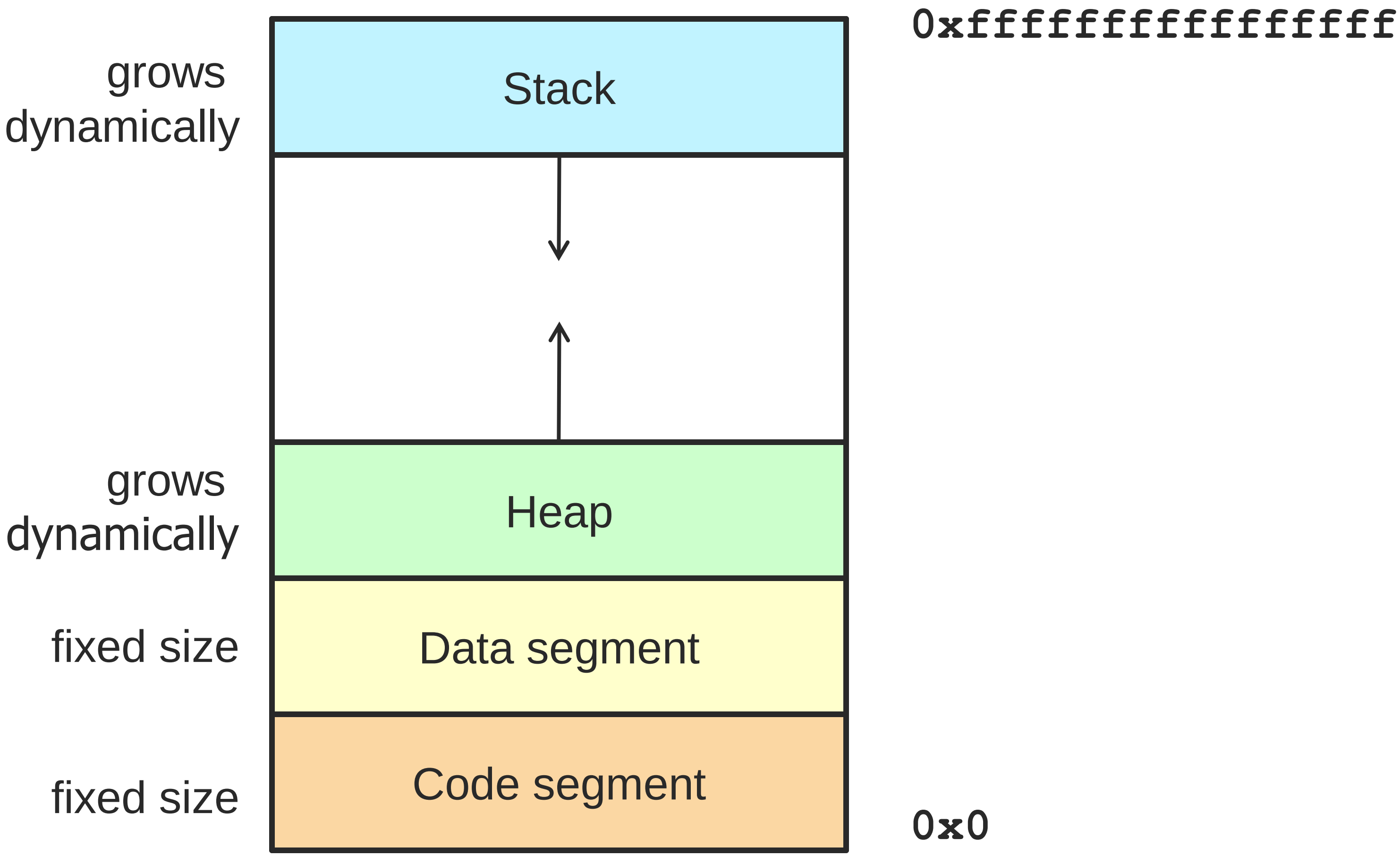
- Address space
  - All memory data
  - i.e., program code, stack, data segment

- Hardware interface (physical reality)
  - Computer has one small, shared memory
- Application interface (illusion)
  - Each process wants private, large memory

Address Space Illusions

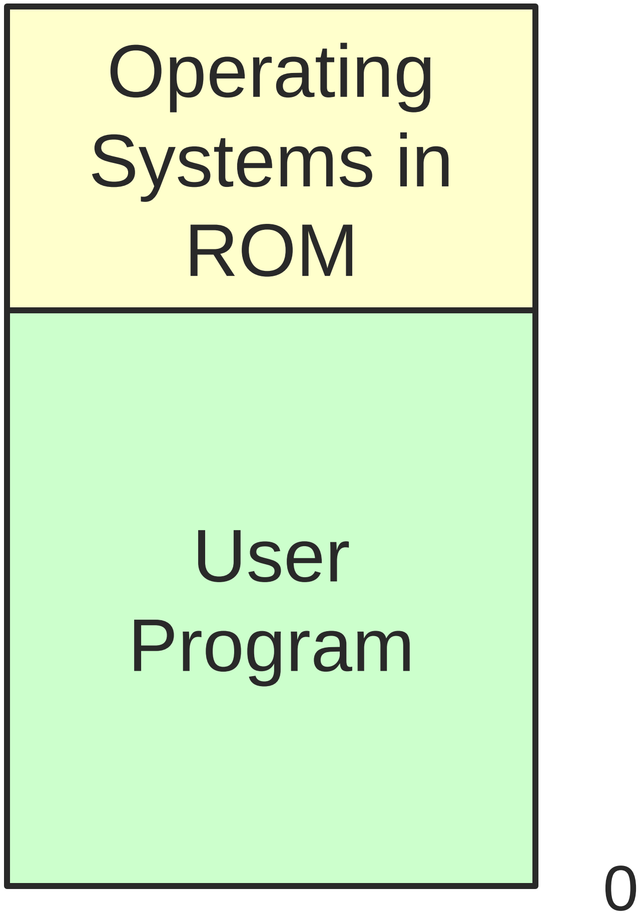
- Address independence
- Protection
- Virtual memory

Address Space



Uni-programming

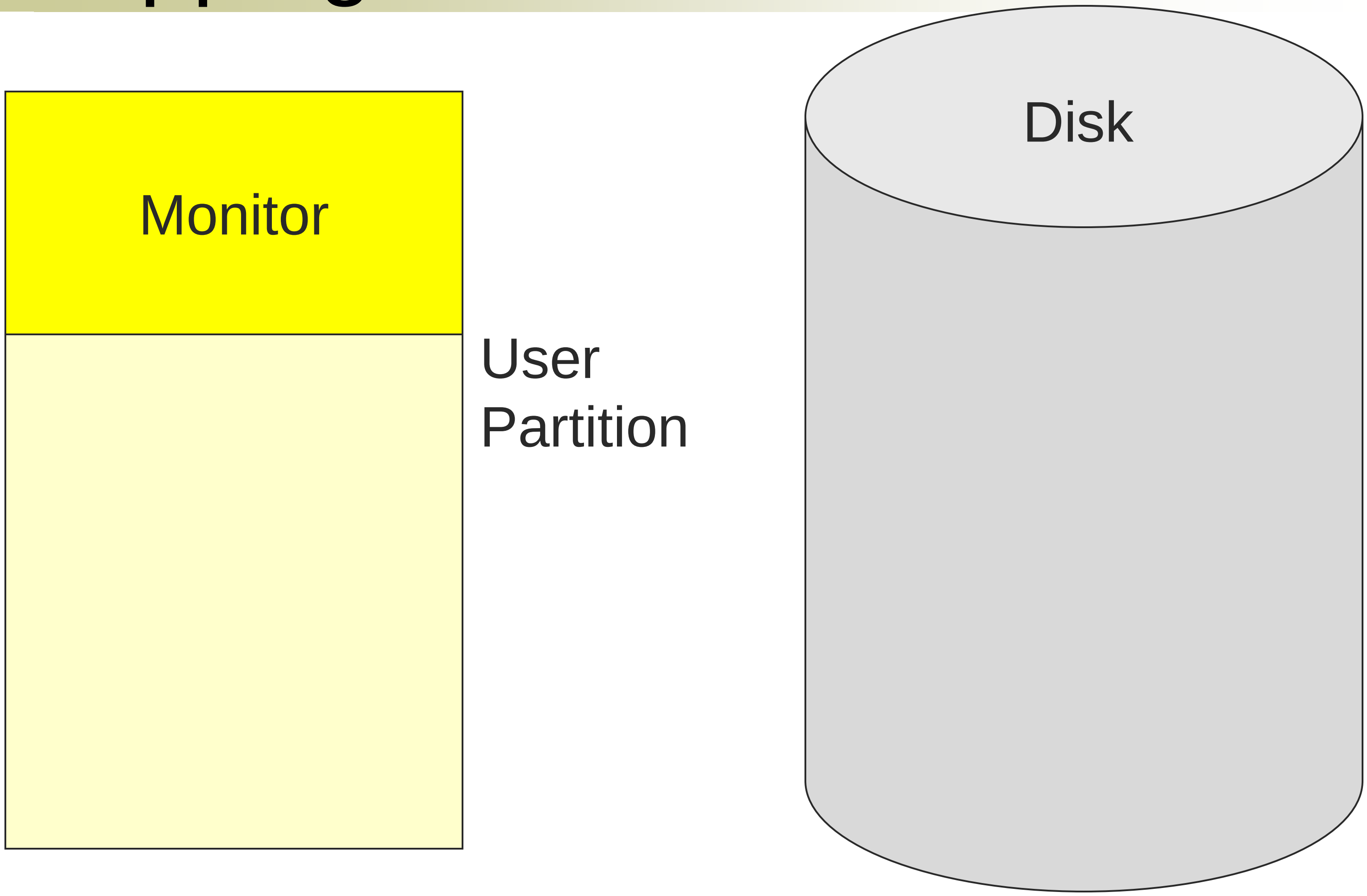
- 1 process runs at a time
- Always load process into the same spot
- How do you switch processes?
- What illusions does this provide?
  - Independence, protection, virtual memory?
- Problems?



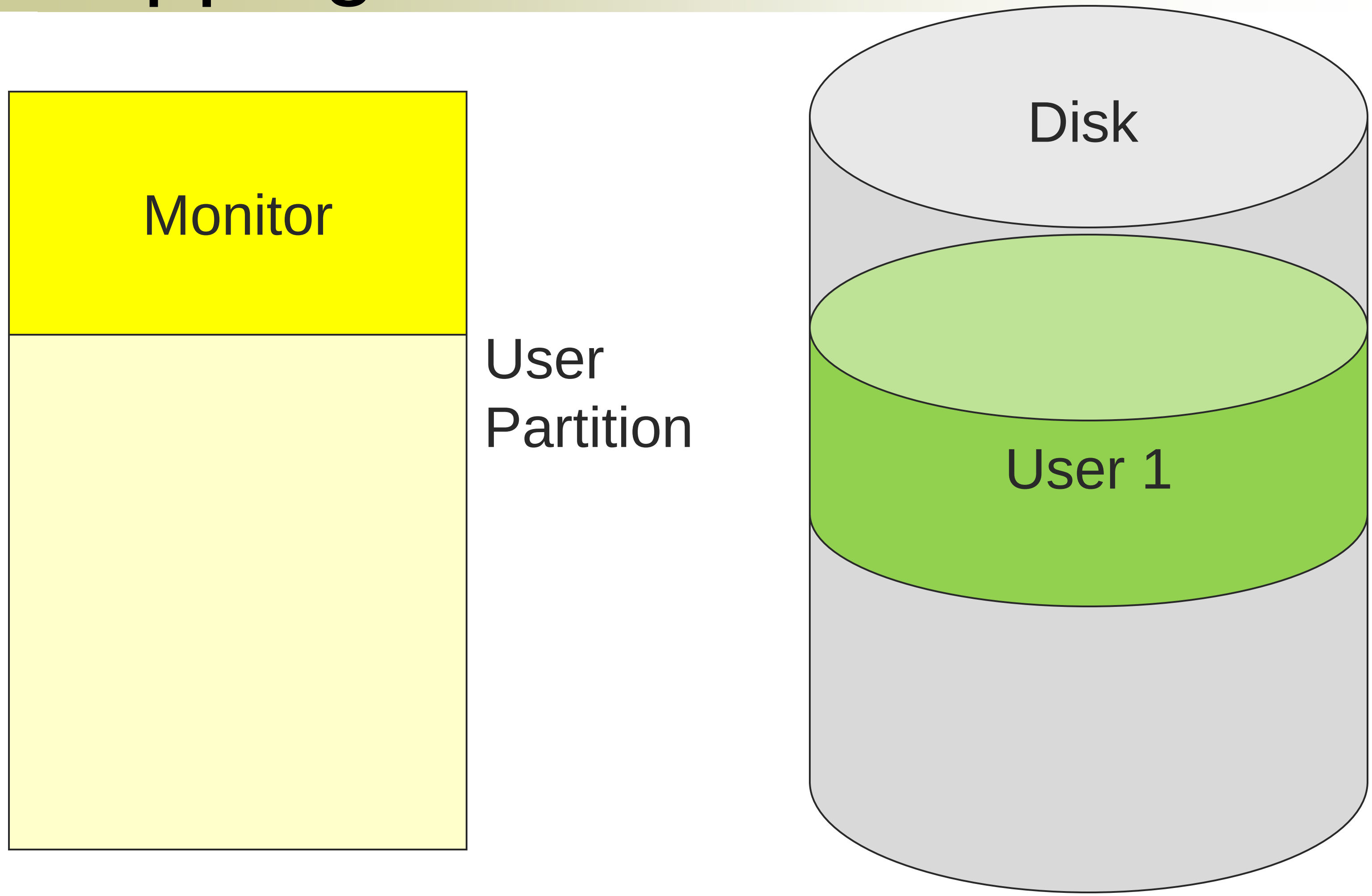
Multi-Programming

- Multiple processes in memory at the same time
- What if there are more processes than what could fit into the memory?
  - Swapping
- Memory allocation changes as
  - Processes come into memory
  - Processes leave memory
    - Swapped to disk
    - Complete execution

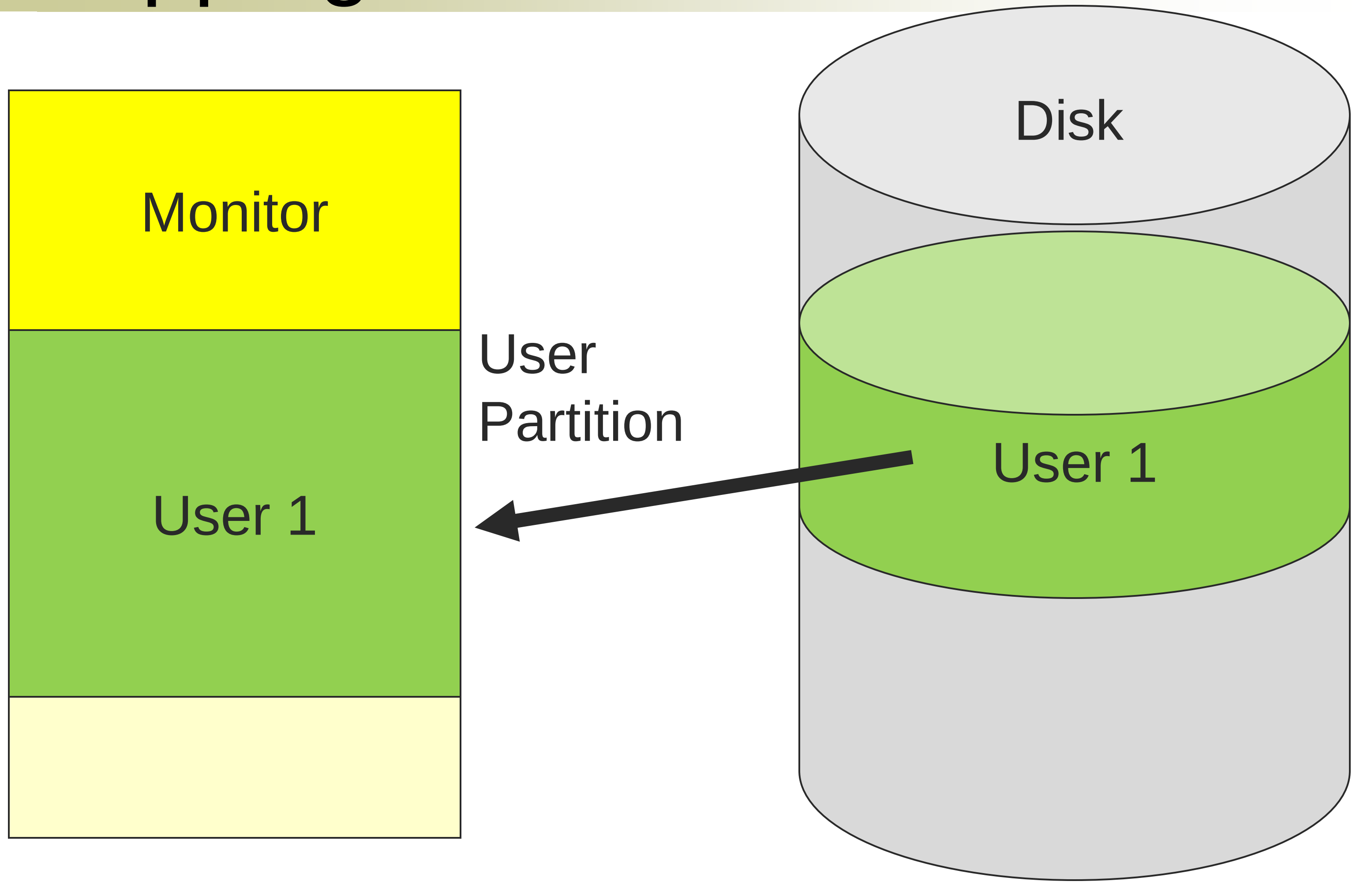
Swapping



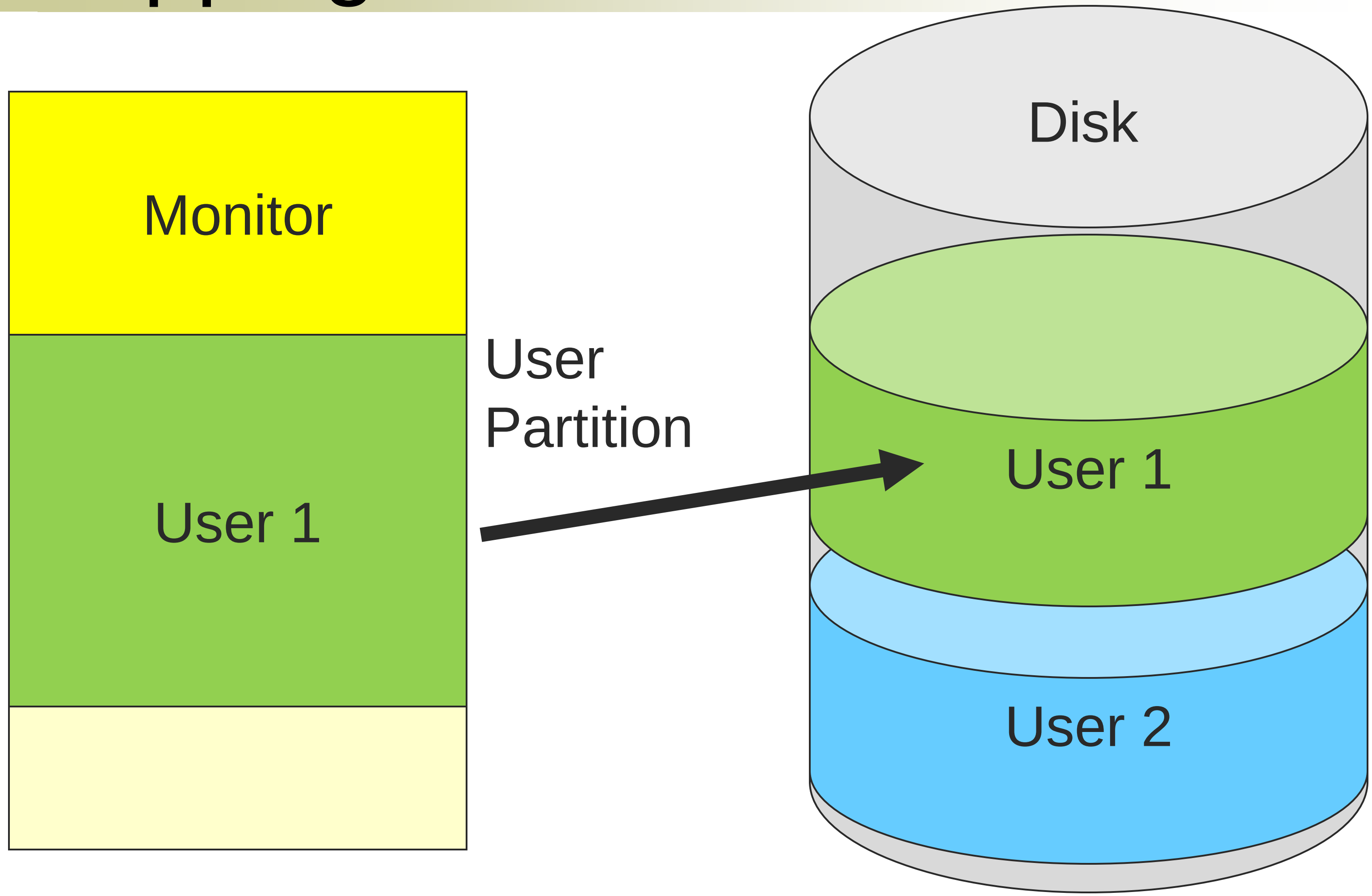
Swapping



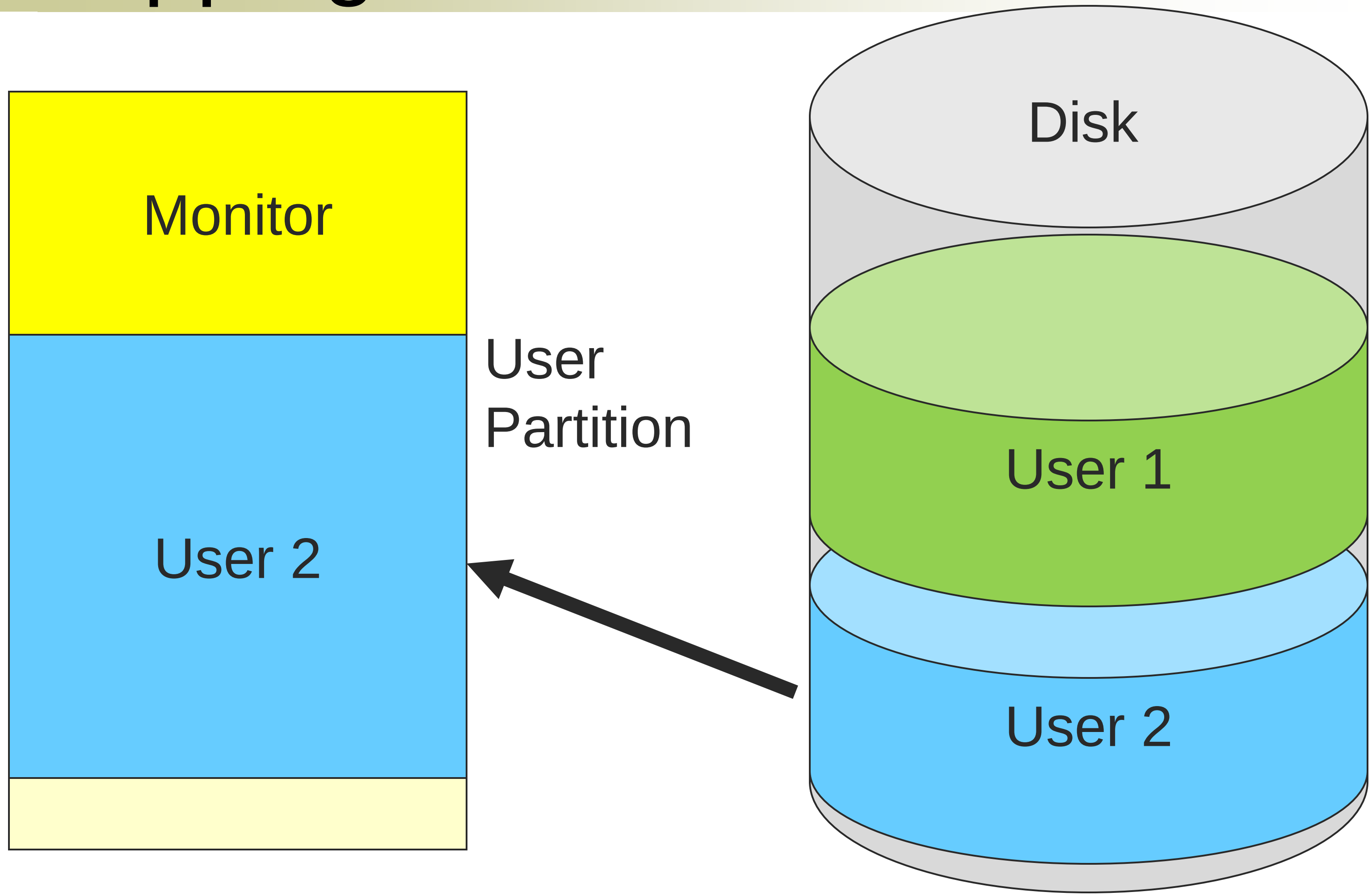
Swapping



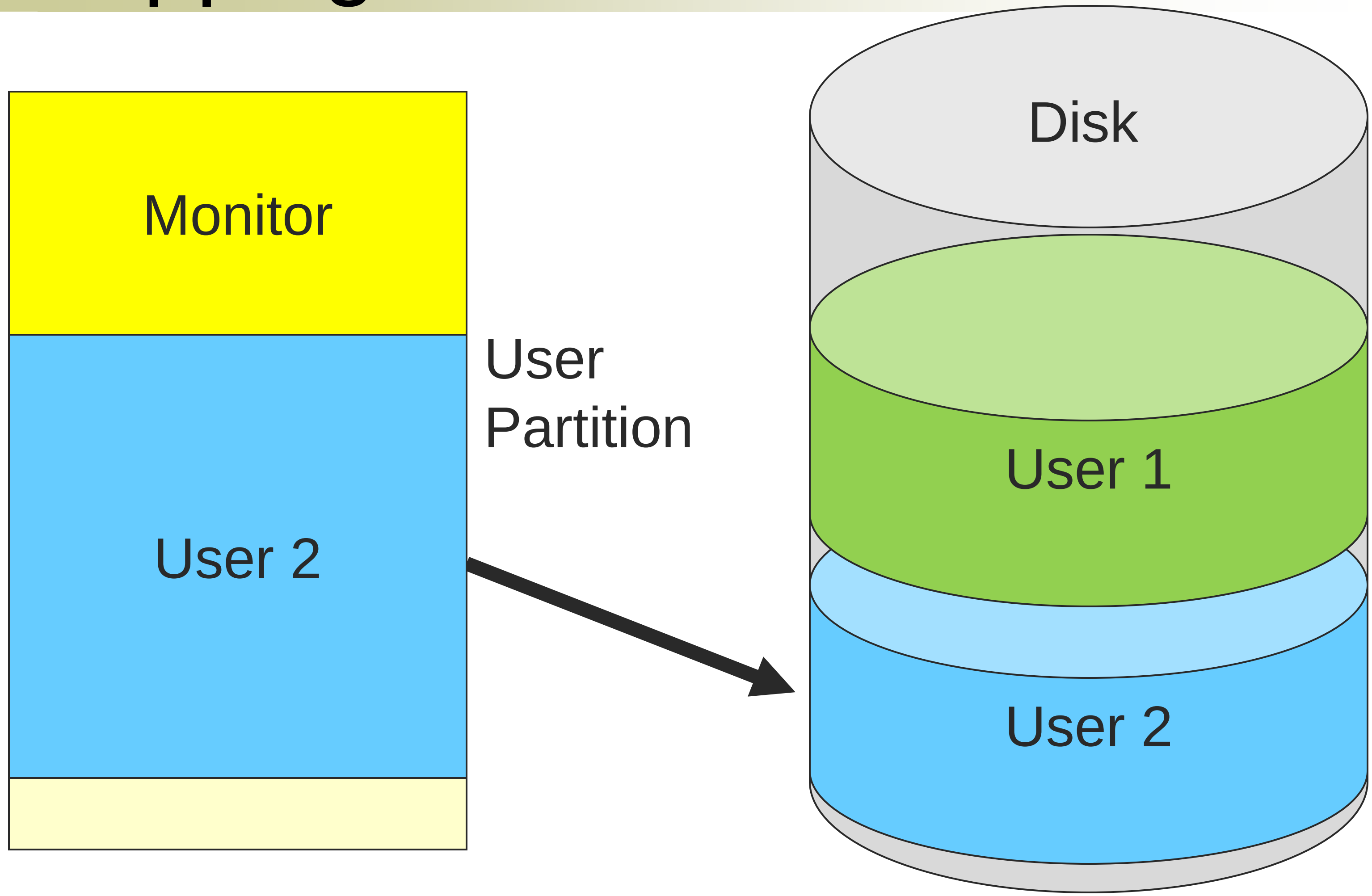
Swapping



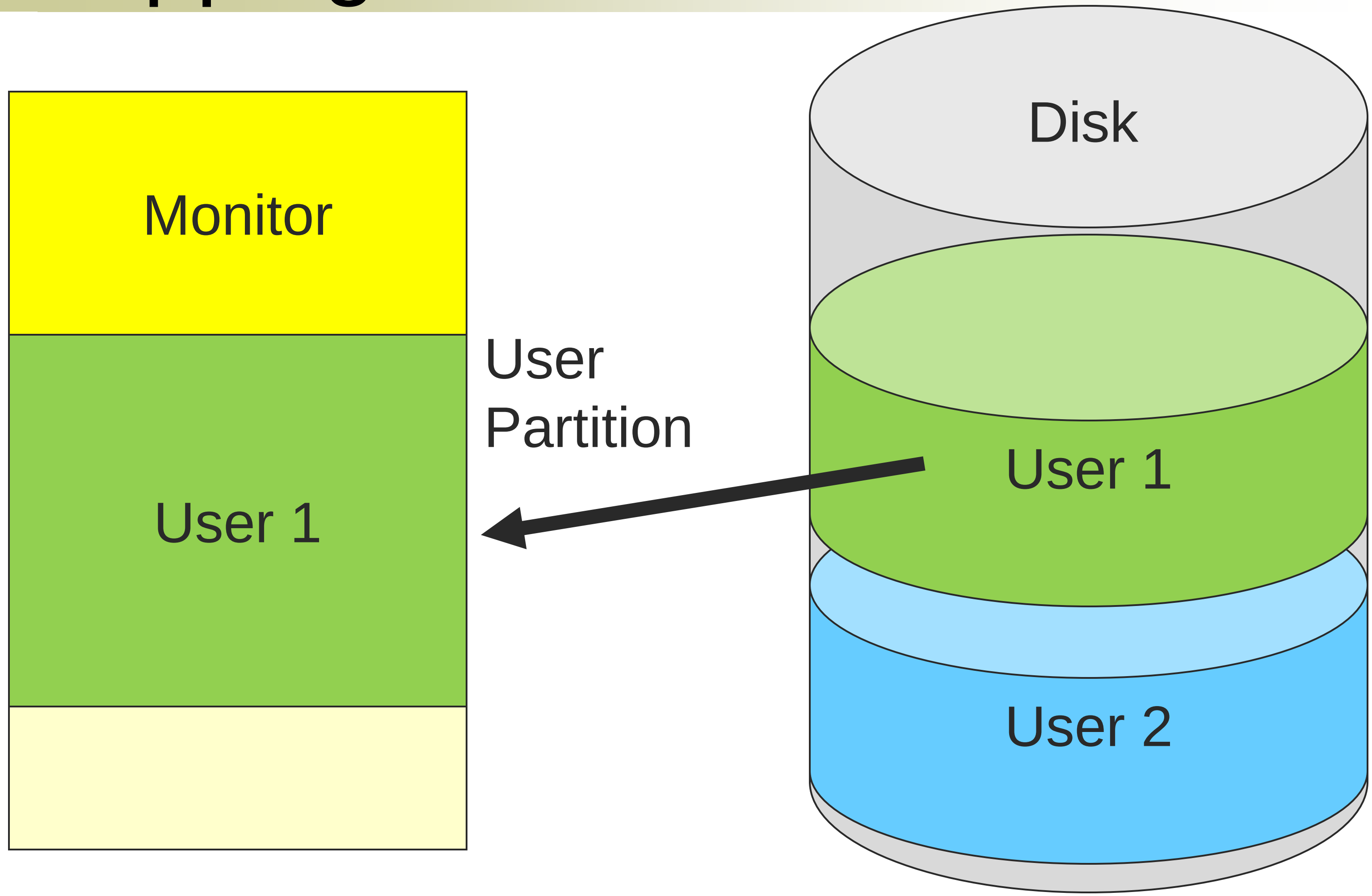
Swapping



Swapping



Swapping



Example

- Consider a system in which memory consists of the following hole sizes in memory order:
  - 10K, 4K, 20K, 18K, 7K, 9K, 12K, and 15K.
  - Which hole is taken for successive requests of:
    - 12K
    - 10K
    - 9K

Example

- Consider a system in which memory consists of the following hole sizes in memory order:
  - 10K, 4K, 20K, 18K, 7K, 9K, 12K, and 15K.
  - Which hole is taken for successive requests of:
    - 12K
    - 10K
    - 9K

First fit:	Best fit:	Worst fit:
20K, 10K, 18K.	12K, 10K, 9K.	20K, 18K, and 15K.

Storage Placement Strategies

- Best fit
  - Produces the smallest leftover hole
  - Creates small holes that cannot be used
- Worst Fit
  - Produces the largest leftover hole
  - Difficult to run large programs
- First Fit
  - Creates average size holes

- First-fit and best-fit better than worst-fit in terms of speed and storage utilization

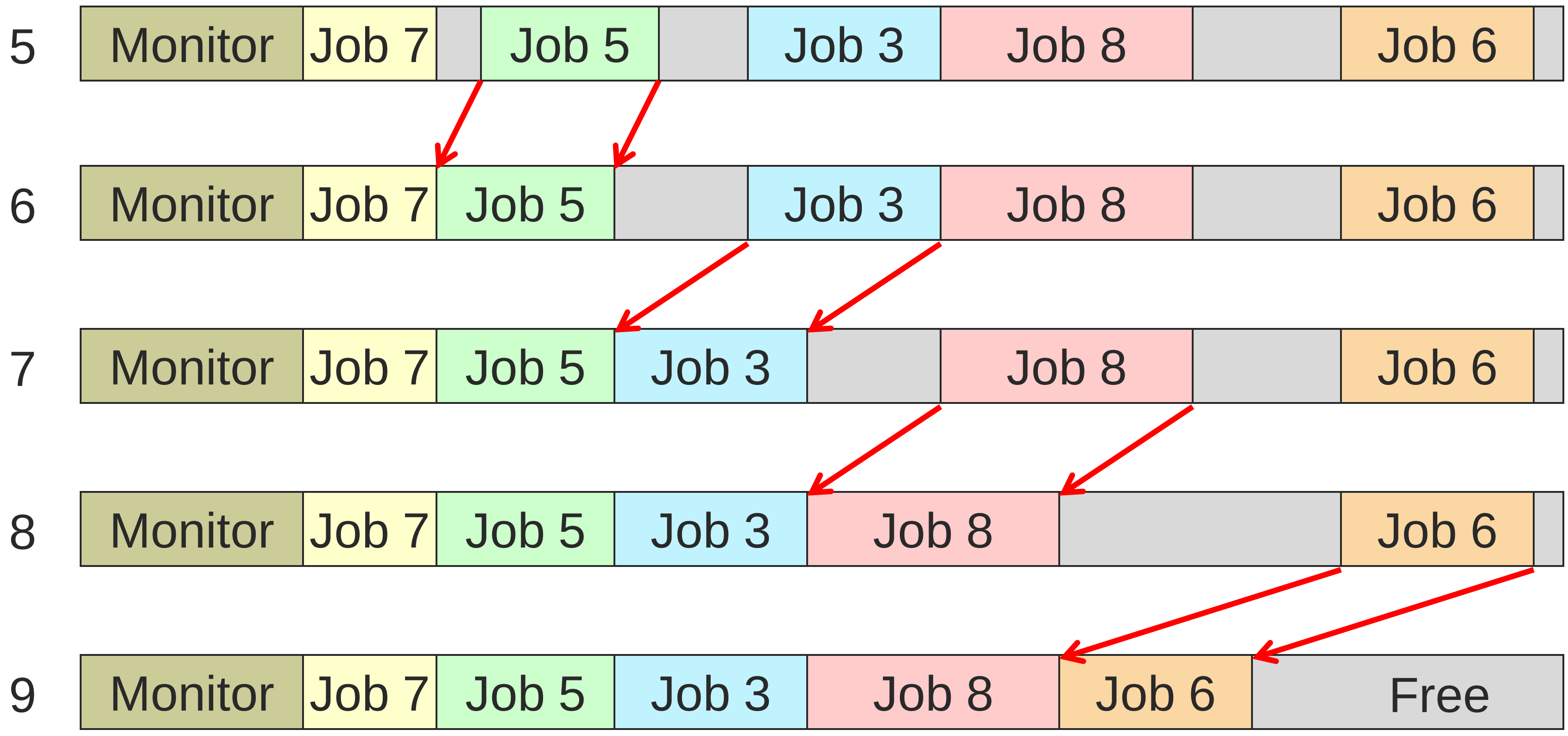
Fragmentation

- External Fragmentation
  - Memory space exists to satisfy a request, but it is not contiguous
- Internal Fragmentation
  - Allocated memory may be slightly larger than requested memory
  - The size difference is memory internal to a partition, but not being used

Compaction

- Reduce external fragmentation by compaction
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible only if relocation is dynamic, and is done at execution time

Solve Fragmentation w. Compaction



Limitations of Swapping

- Problems with swapping
  - Process must fit into physical memory (impossible to run larger processes)
  - Memory becomes fragmented
  - Processes are either in memory or on disk
    - Half and half doesn’t do any good

Virtual memory

- Basic idea
  - Allow the OS to hand out more memory than exists on the system
  - Keep recently used stuff in physical memory
  - Move less recently used stuff to disk
  - Keep all of this hidden from processes
- Process view
  - Processes still see an address space from 0 – max address
  - Movement of information to and from disk handled by the OS without process help

Multi-programming

- Multiple processes in memory at the same time
  - What do we really need?
    - Address translation
    - Protection

Address Translation

- Goals
  - Avoid conflicting addresses
- Approaches
  - Static
    - Translate before you execute
  - Dynamic
    - Translate during execution, could change

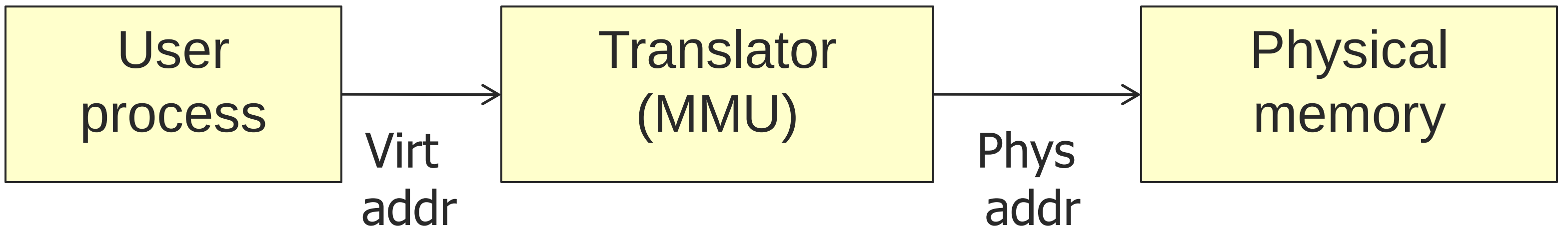
Dynamic Translation

- Translate every memory reference from virtual address to physical address
  - Virtual address
    - An address viewed by the user process
  - Physical address
    - An address viewed by the physical memory

Virtual Addresses

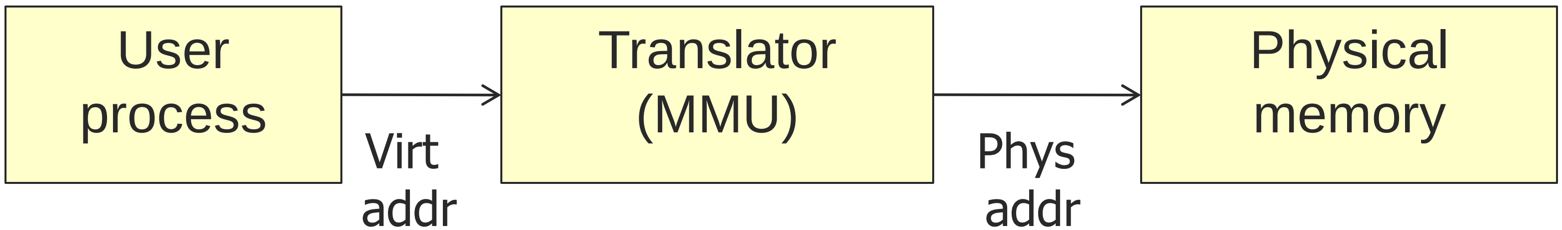
- Different jobs run at different addresses
  - Program never sees physical address
  - At link-time
    - Linker must know program's starting memory address
  - Correct starting address when a program starts in memory

Dynamic Address Translation



- Translation enforces protection
  - One process can't even refer to another process's address space
- Translation enables virtual memory
  - A virtual address only needs to be in physical memory when it is being accessed
  - Change translations on the fly as different virtual addresses occupy physical memory

Dynamic Address Translation



- Implementation tradeoffs
  - Flexibility (e.g., sharing, growth, virtual memory)
  - Size of translation data
  - Speed of translation

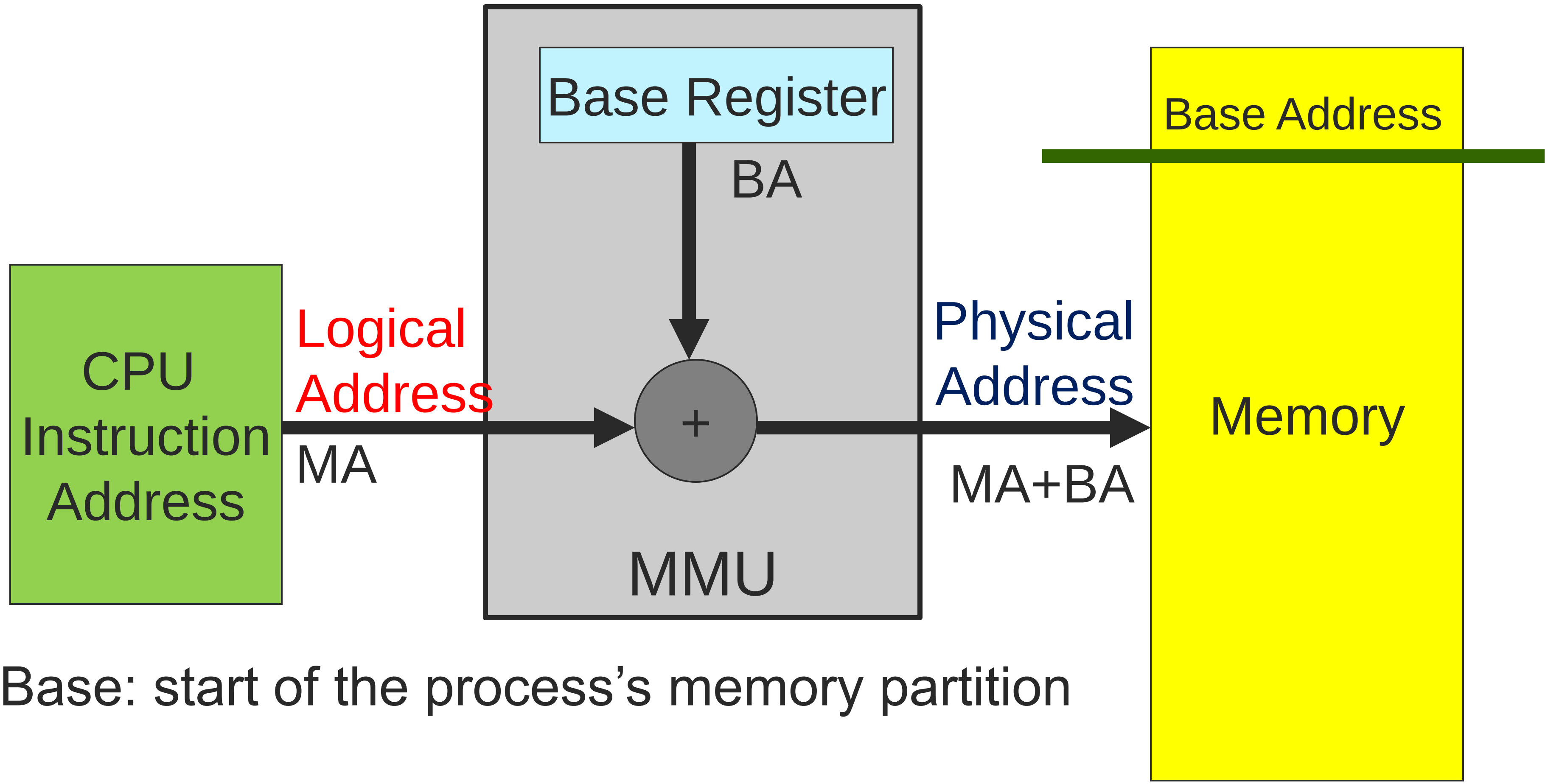
Dynamic Address Translation

- Load each process into contiguous regions of physical memory

- Logical or "Virtual" addresses
  - Logical address space
  - Range: 0 to max

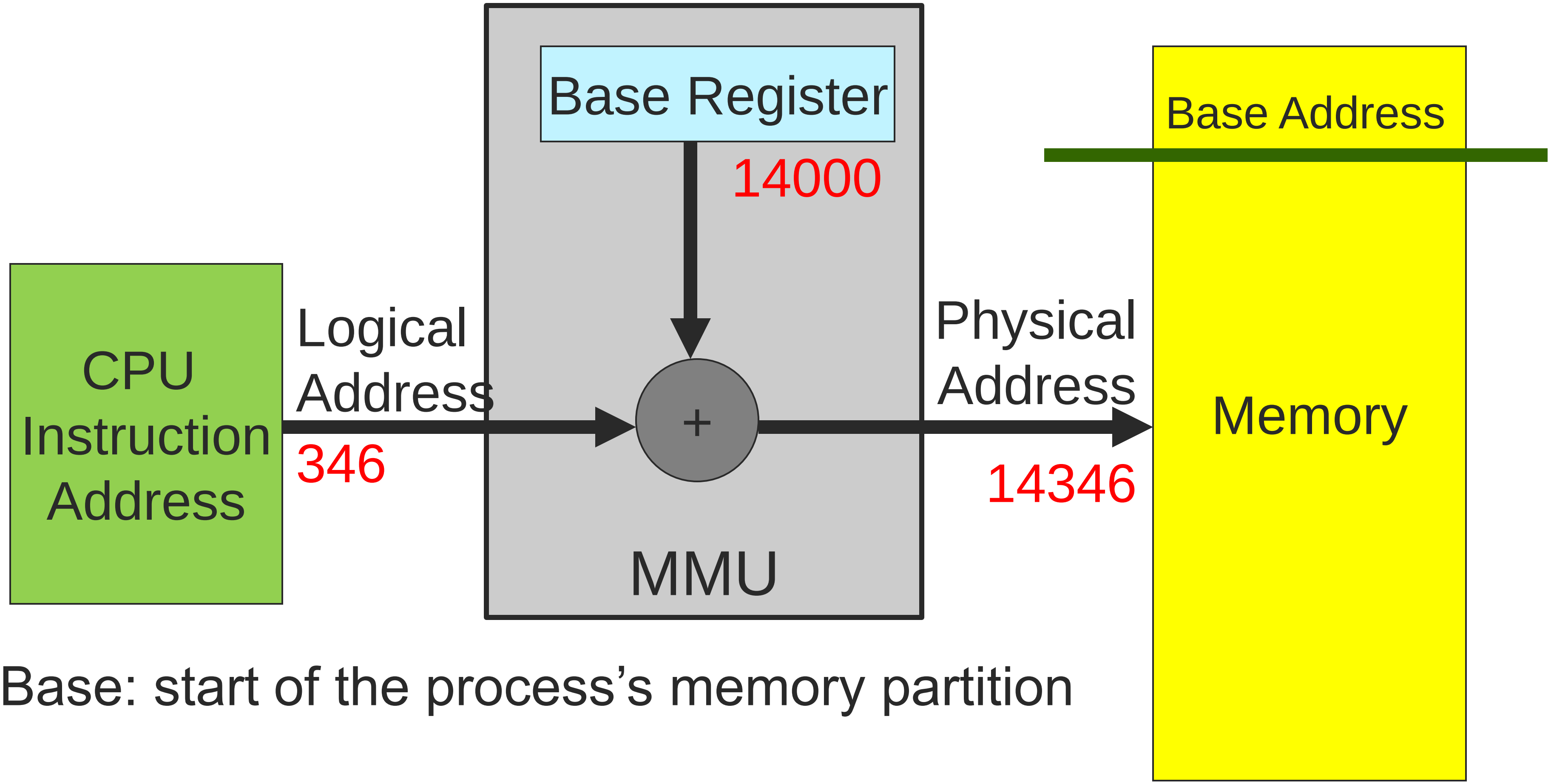
- Physical addresses
  - Physical address space
  - Range: R+0 to R+max for base value R

Base Register



Base: start of the process's memory partition

Base Register



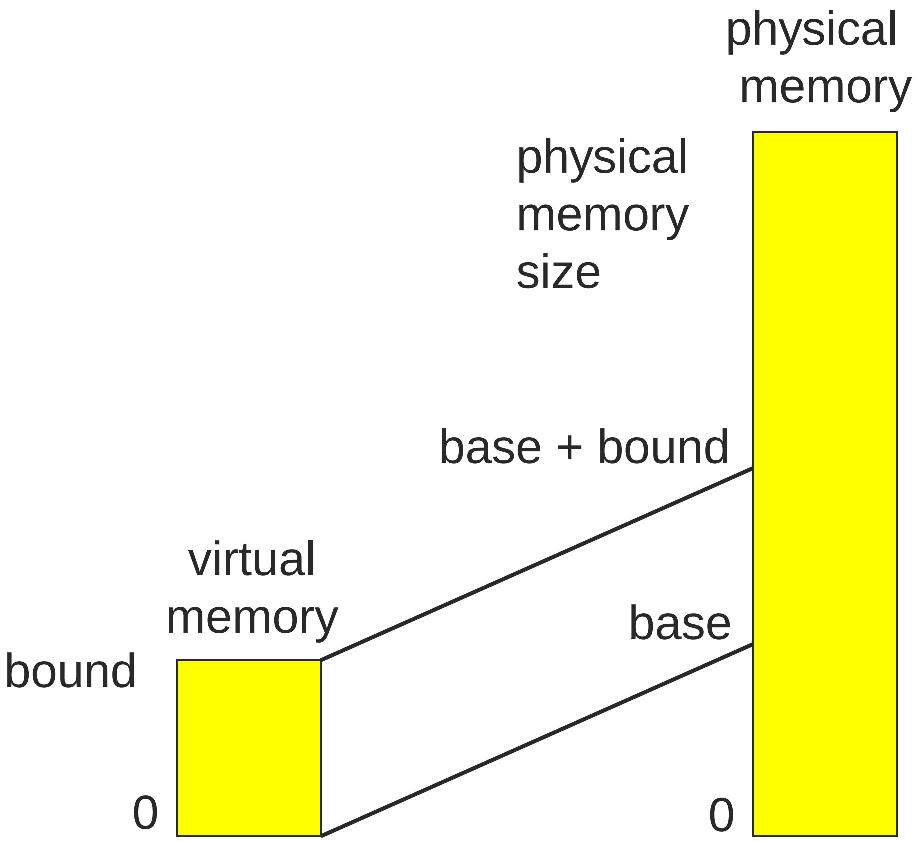
Base: start of the process's memory partition

Protection

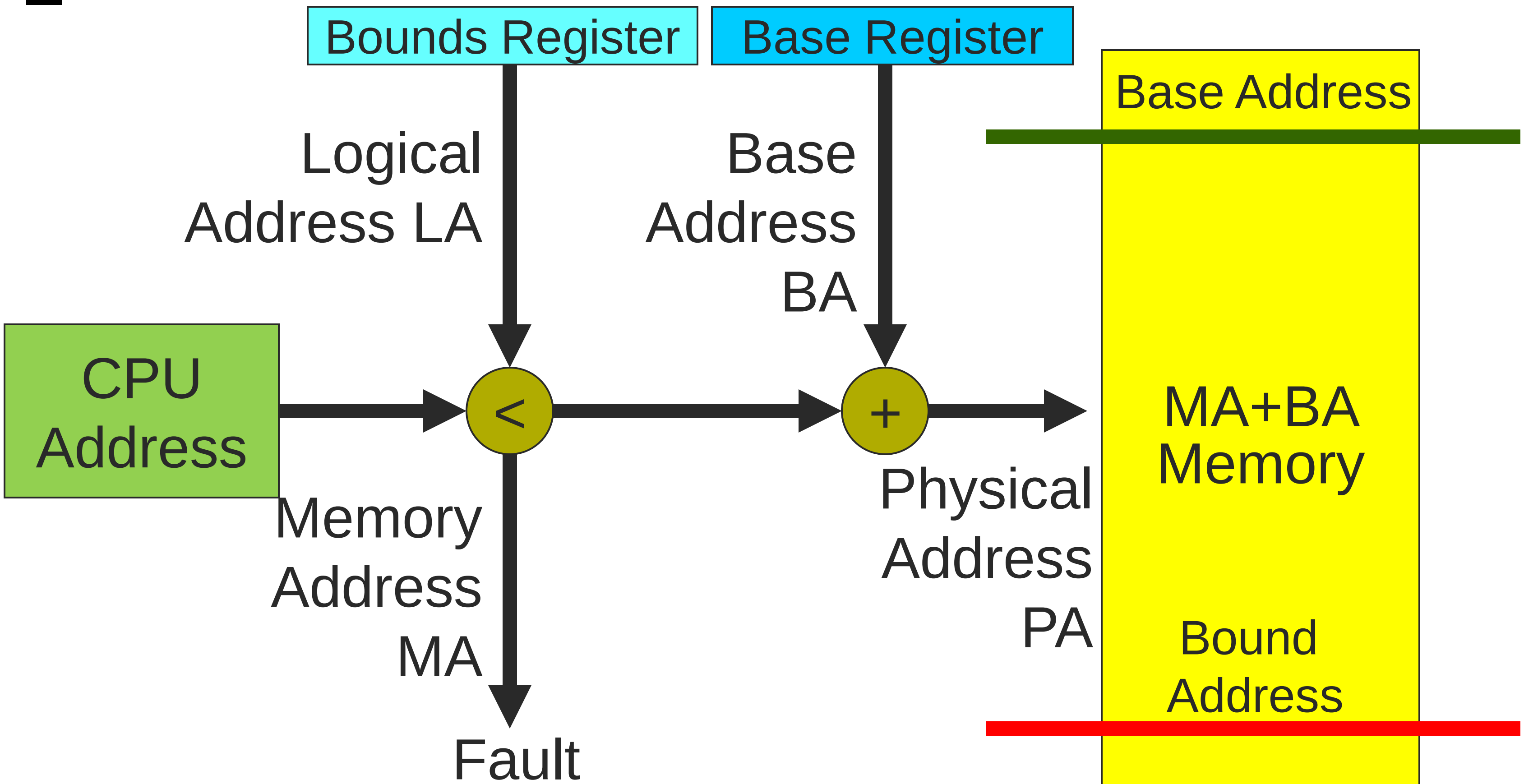
- Problem
  - How to prevent a malicious process from writing or jumping into other user's or OS partitions
- Solution
  - Base bounds registers

Base and bounds

```
if (virt addr > bound)
    trap to kernel
} else {
    phys addr =
      virt addr + base
}
```



Base and bounds



Base: start of the process's memory partition
Bound: length of the process's memory partition

Base and bounds

- What must change during a context switch?
- Can a proc change its own base and bound?
- Can you share memory with another process?

Base and bounds

- How does the kernel handle the address space growing?
  - You are the OS designer, come up with an algorithm for allowing processes to grow

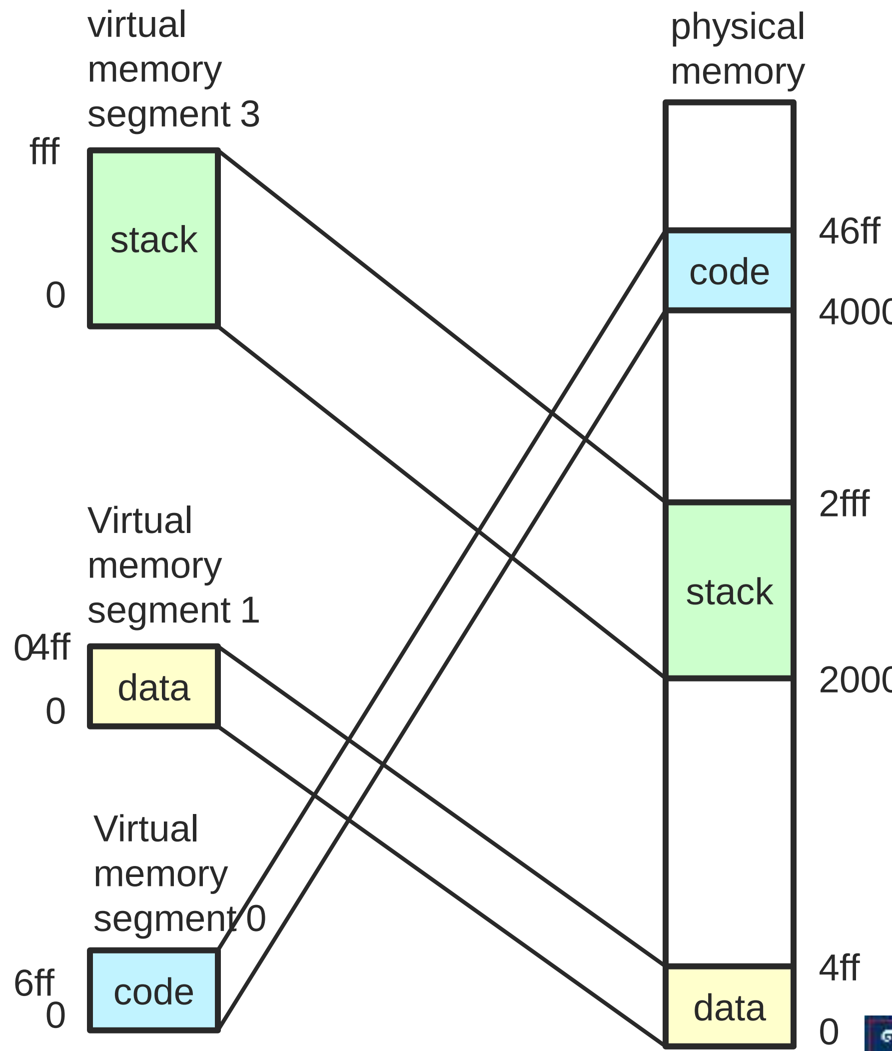
Segmentation

- Segment
  - Region of contiguous memory
- Segmentation
  - Generalized base and bounds with support for multiple segments at once

Segmentation

- Segments are specified many different ways
- What are the advantages over base and bounds?
- What must be changed on context switch?

Seg #	Base	Bound	Description
0	4000	700	Code segment
1	0	500	Data segment
2	Unused		
3	2000	1000	Stack segment



Problem with Segmentation and B&B

- What was the key abstraction not supported well by segmentation and by B&B?
  - How could you support this using B&B and segmentation?

- Note: x86 used to support segmentation, now effectively deprecated with x86-64

Paging

- Allocate physical memory in terms of fixed-size chunks
  - Fixed unit makes it easier to allocate
  - Any free physical page can store any virtual page
- Virtual address
  - Virtual page # (high bits of address)
  - Offset (low bits of address, e.g., bits 11-0 for 4k page)

Translation Table

Virtual page #	Physical page #
0	10
1	15
2	20
3	invalid
…	invalid
1048575	invalid

Translation Process

```
if (virtual page is invalid or non-resident or
   protected) {
    trap to OS fault handler
} else {
    physical page # = pageTable[virtpage#]
                           .physPageNum
}
```

- What must change on a context switch?
- Each virtual page can be in physical memory or swapped out to disk (called paged)

Paging

- How does the processor know that a virtual page is not in memory?
- Like segments, pages can have different protections
  - Read, write, execute

Valid vs. Resident

- Resident
  - Virtual page is in memory
  - NOT an error for a program to access non-resident page
- Valid
  - Virtual page is legal for the program to access
  - e.g., part of the address space

Valid vs. Resident

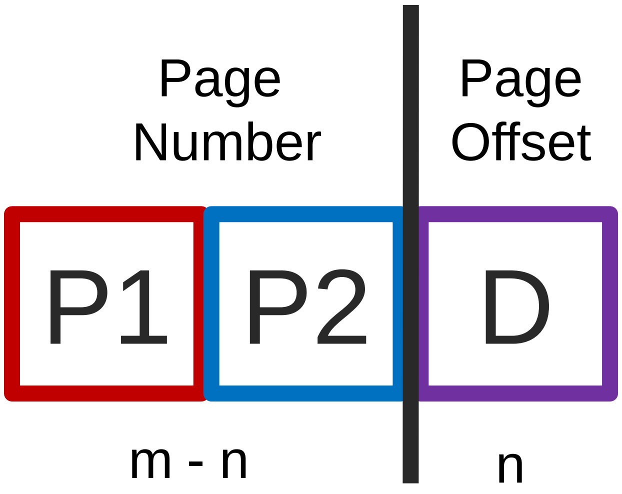
- Who makes a page resident/non-resident?
- Who makes a virtual page valid/invalid?
- Why would a process want one if its virtual pages to be invalid?

Valid vs. Resident

- Who makes a page resident/non-resident?
  - OS memory manager
- Who makes a virtual page valid/invalid?
  - User actions
- Why would a process want one if its virtual pages to be invalid?
  - Avoid accidental memory references to bad locations

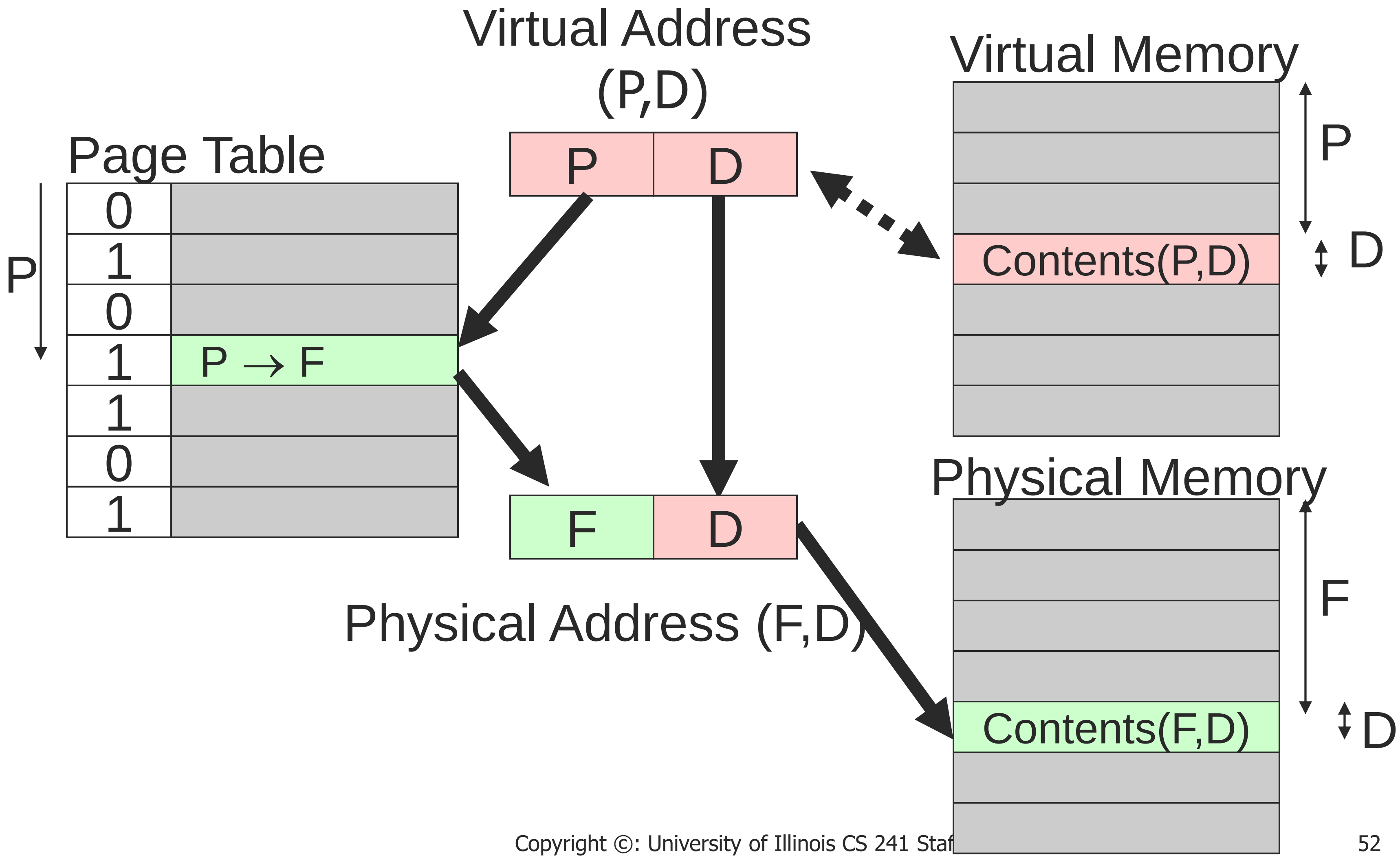
Address Translation Scheme

- Address generated by CPU is divided into
  - Page number (p)
    - An index into a page table
    - Contains base address of each page in physical memory
  - Page offset (d)
    - Combined with base address
    - Defines the physical memory address that is sent to the memory unit

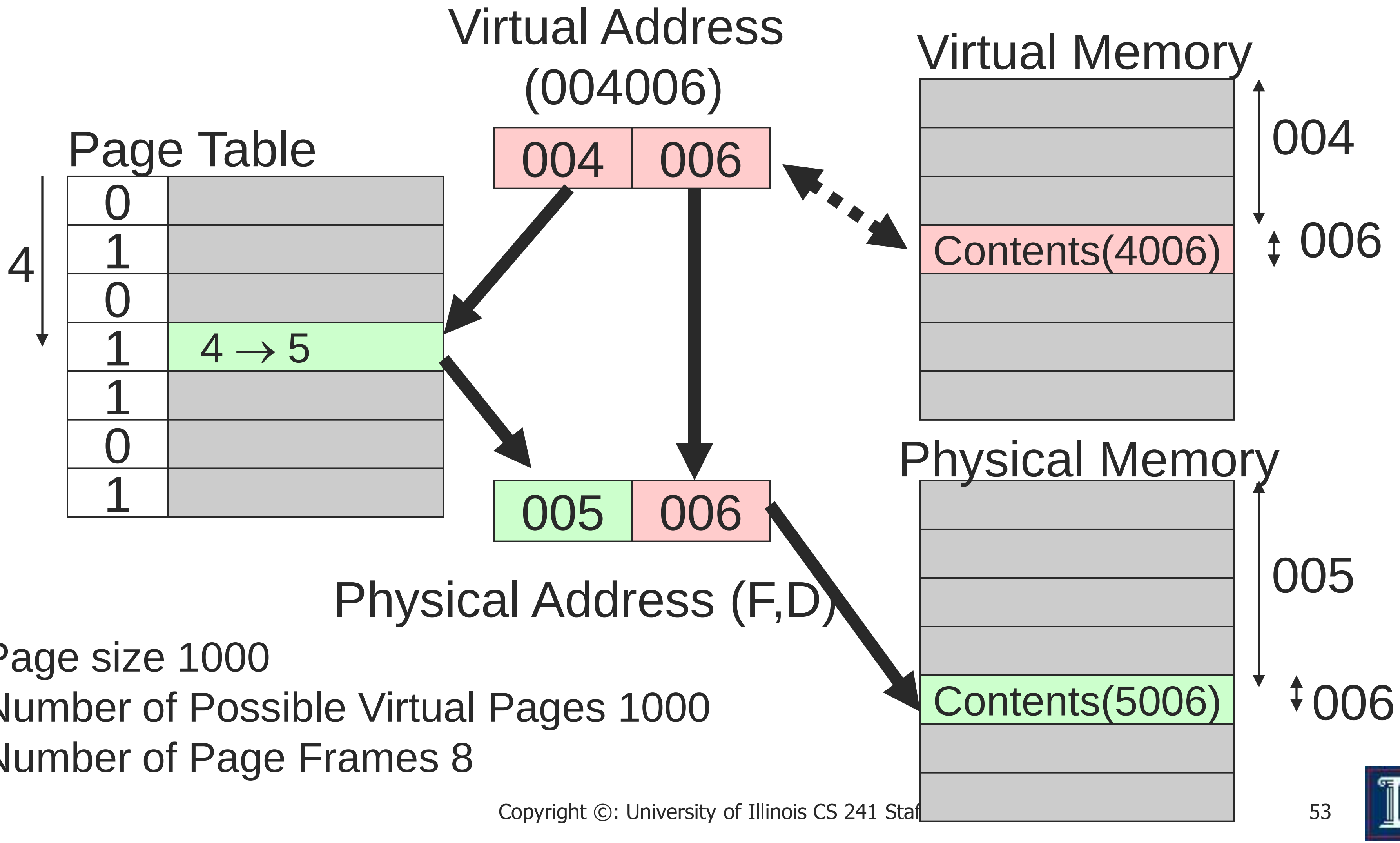


For given logical address space 2m and page size 2n

Page Mapping Hardware



Page Mapping Hardware



Page size 1000
Number of Possible Virtual Pages 1000
Number of Page Frames 8

Page Faults

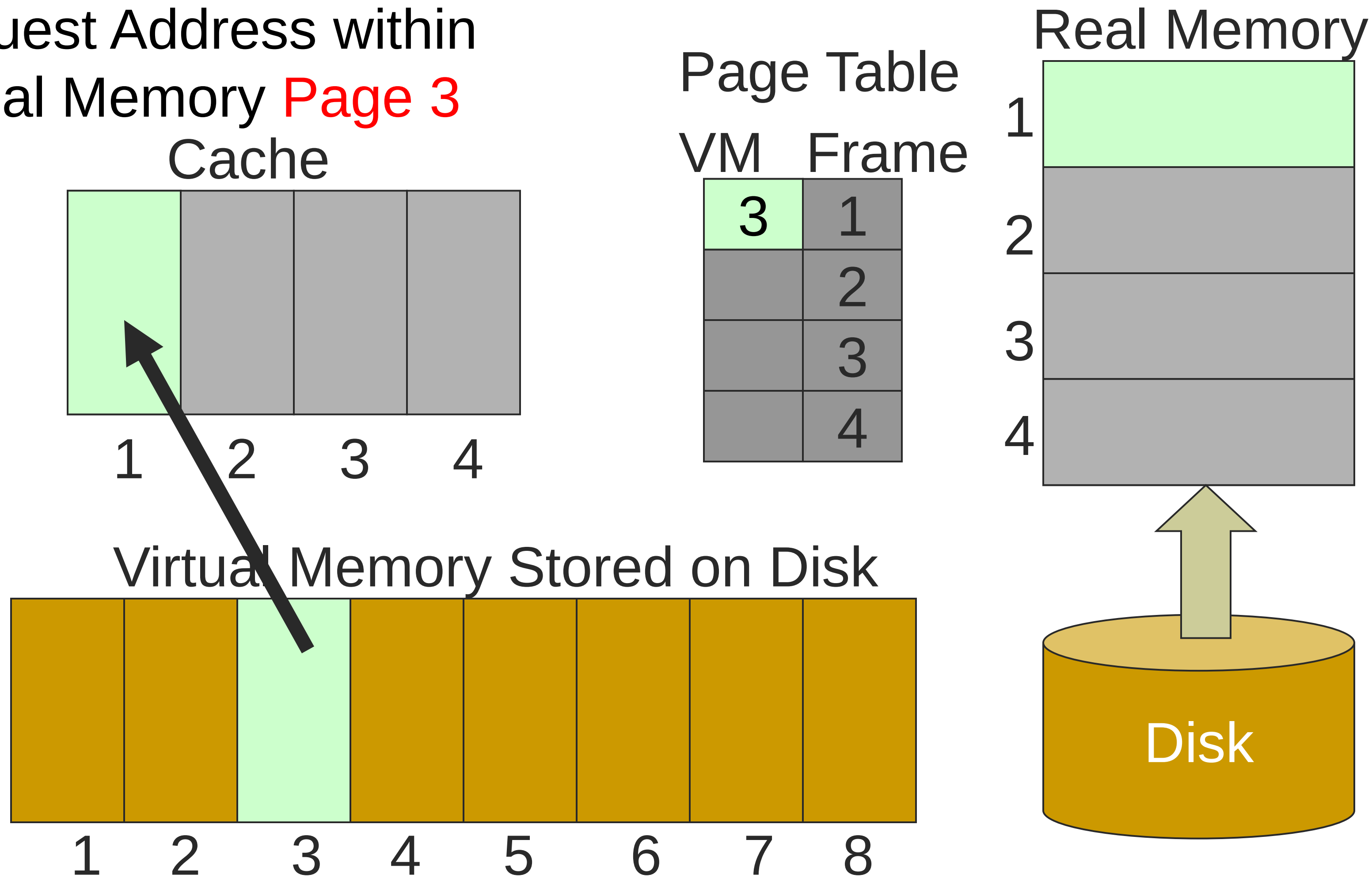
- What happens when a program accesses a virtual page that is not mapped into any physical page?
  - Hardware triggers a page fault
- Page fault handler
  - Find any available free physical page
  - If none, evict some resident page to disk
  - Allocate a free physical page
  - Load the faulted virtual page to the prepared physical page
  - Modify the page table

Paging

- Paging is how an OS achieves VM
- Goal
  - Provide user with virtual memory that is as big as user needs
- Implementation
  - Store virtual memory on disk
  - Cache parts of virtual memory being used in real memory
  - Load and store cached virtual memory without user program intervention

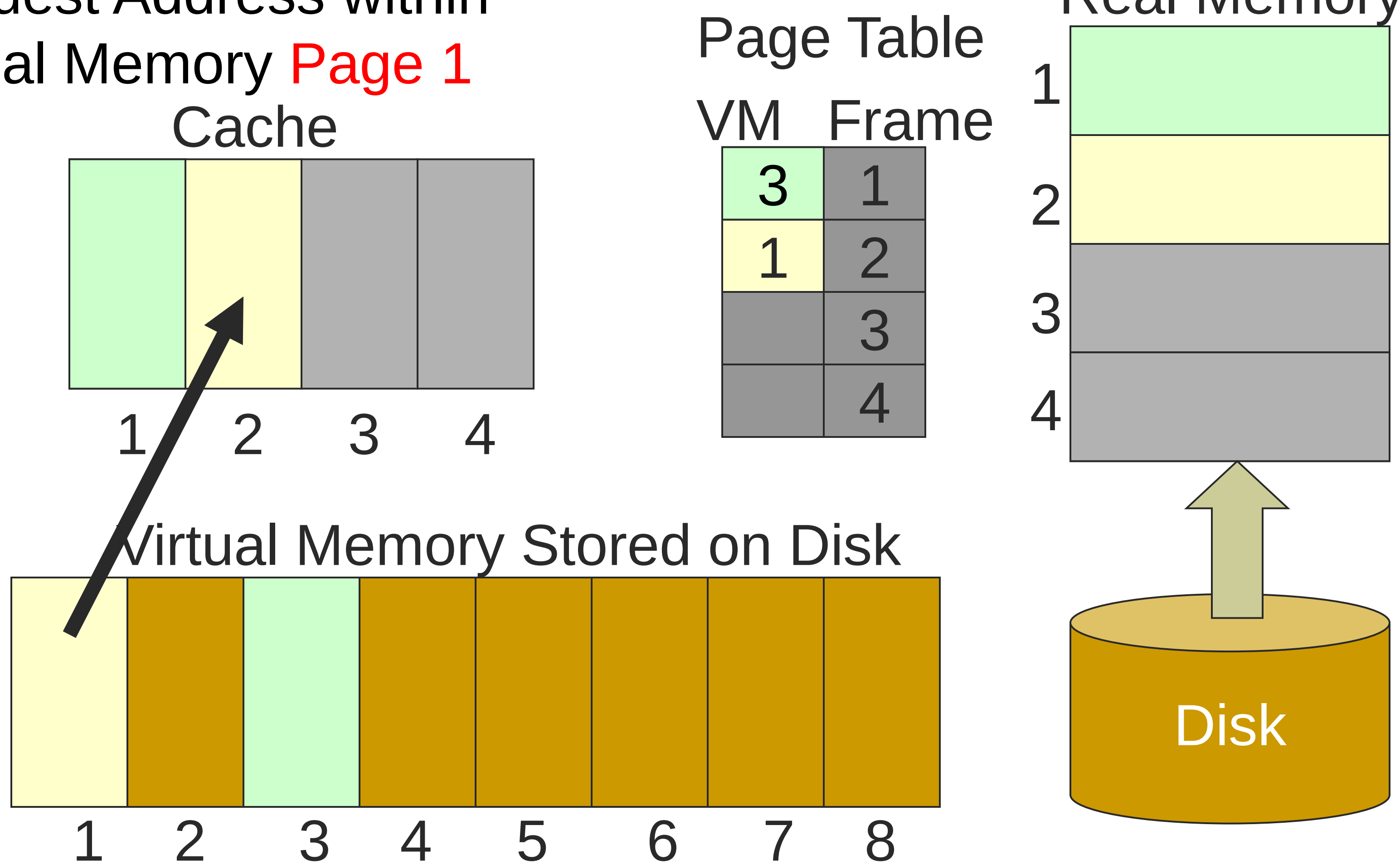
Paging Request

Request Address within Virtual Memory Page 3



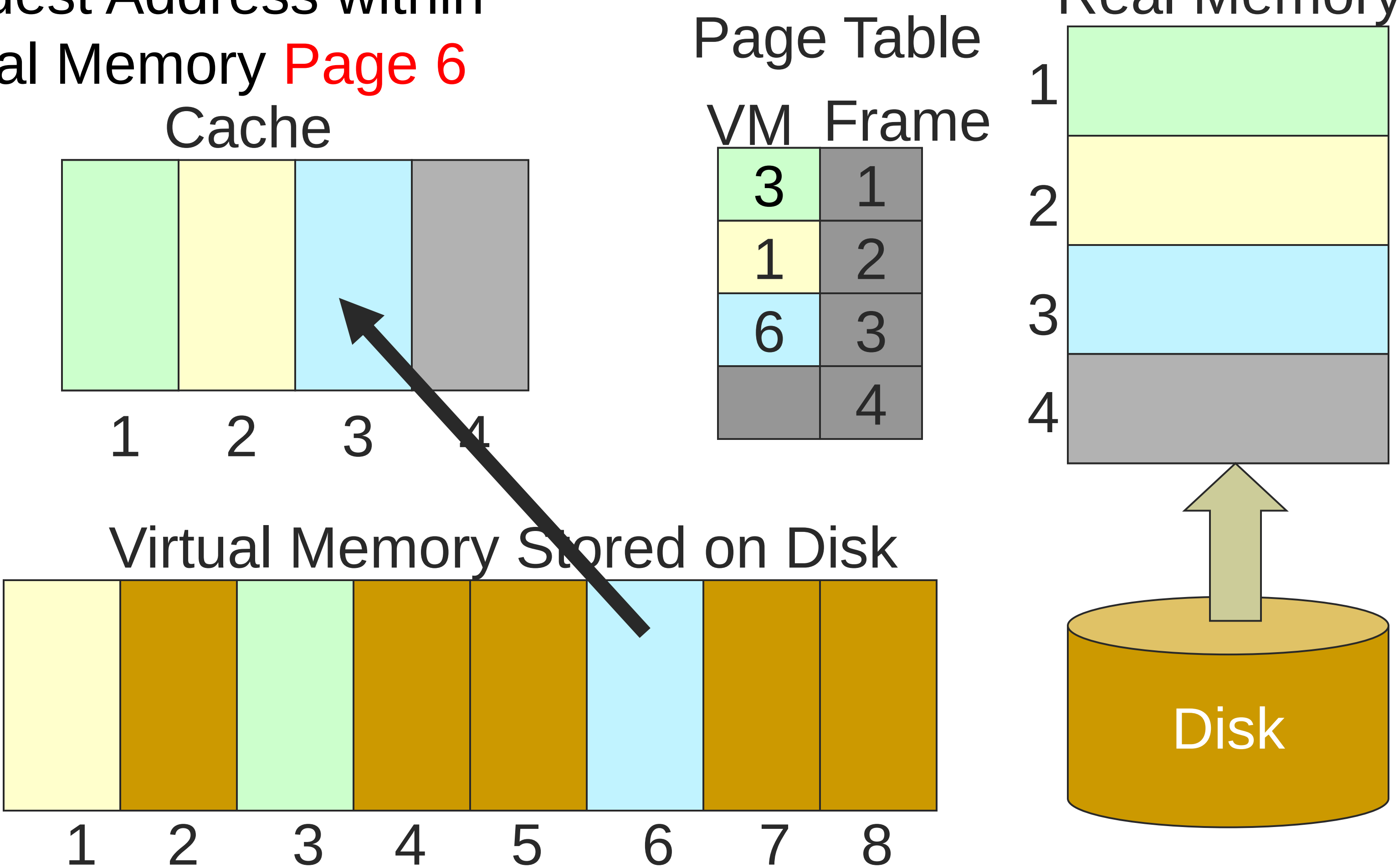
Paging Request

Request Address within Virtual Memory Page 1



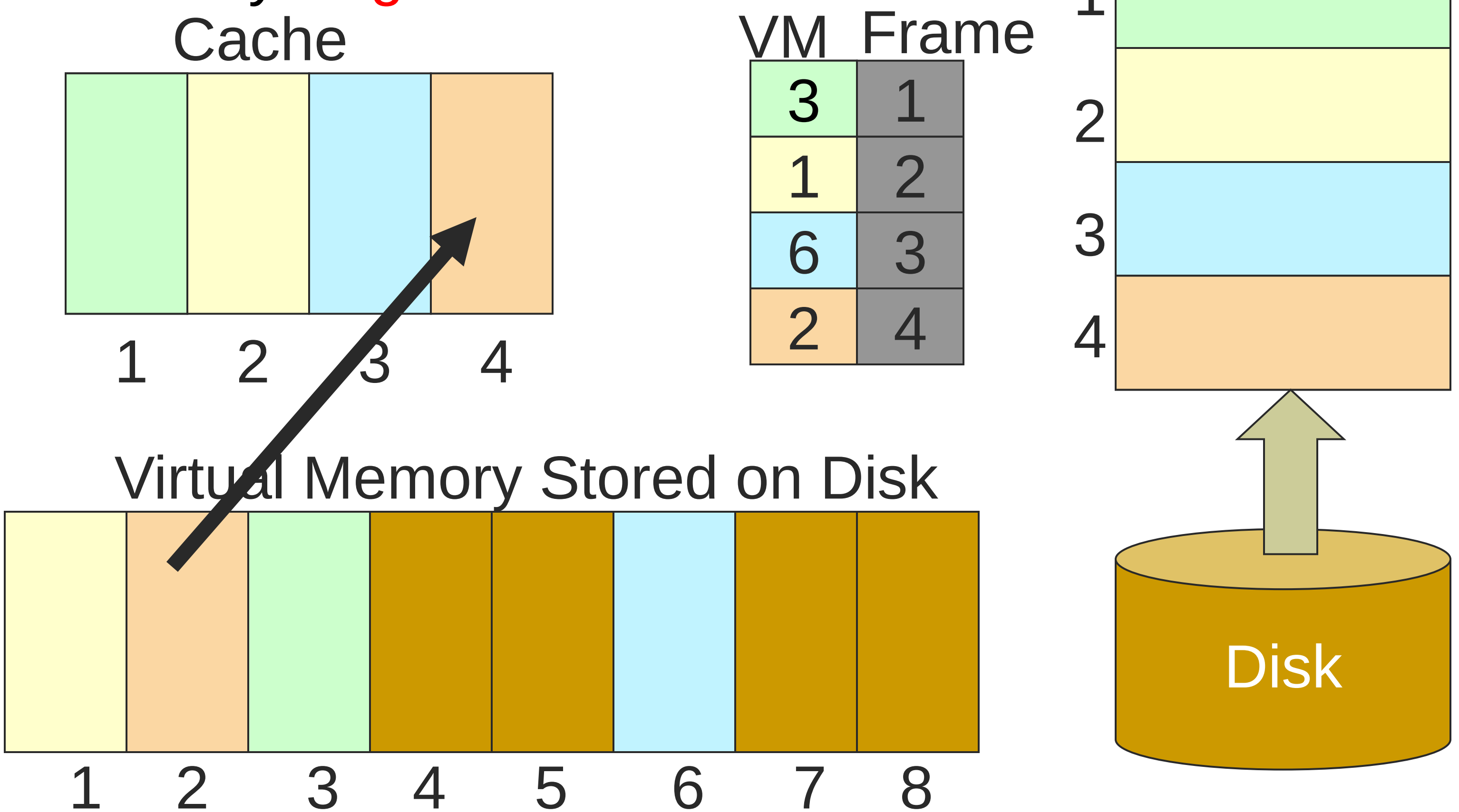
Paging Request

Request Address within Virtual Memory Page 6



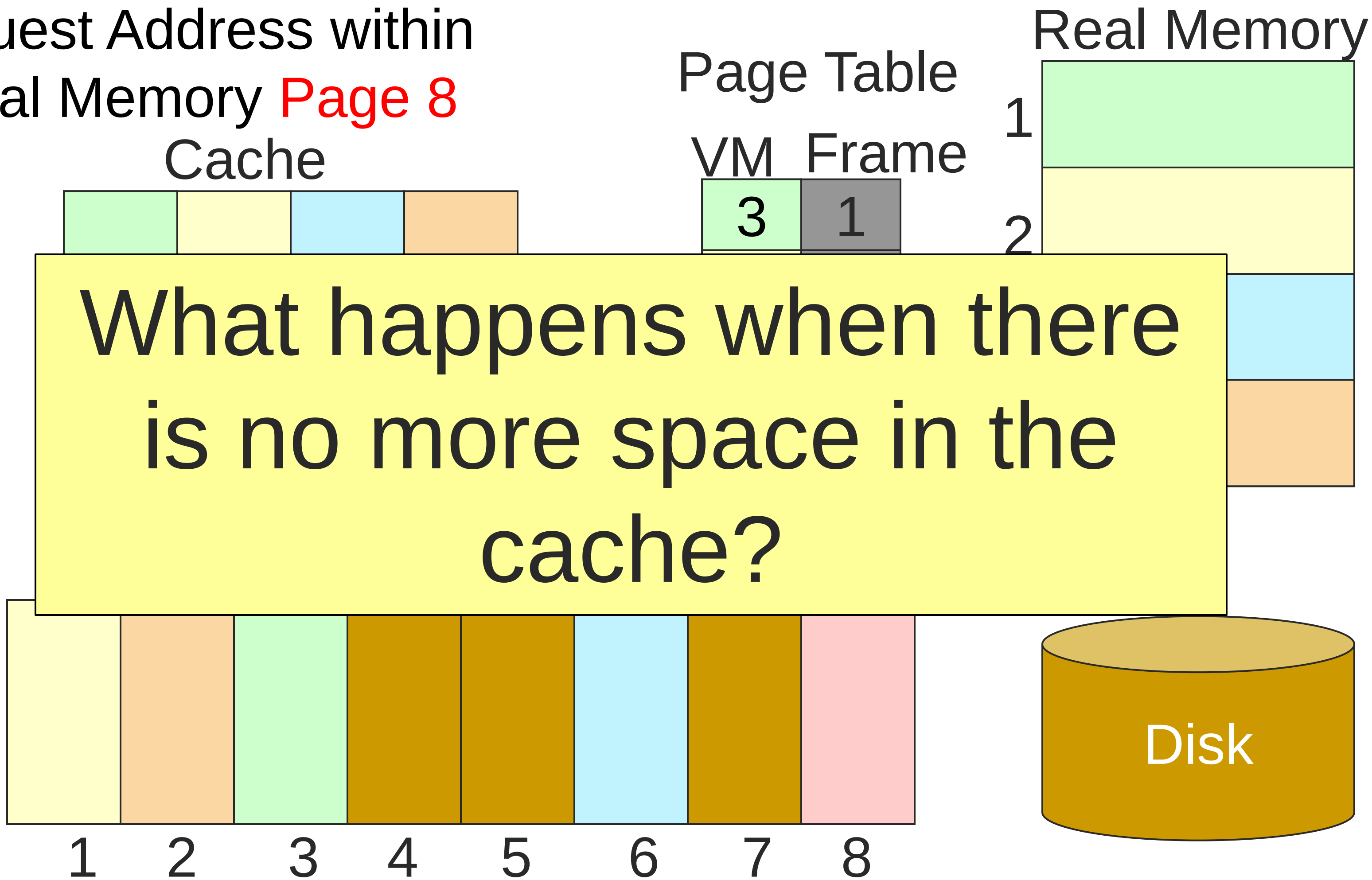
Paging Request

Request Address within
Virtual Memory Page 2



Paging Request

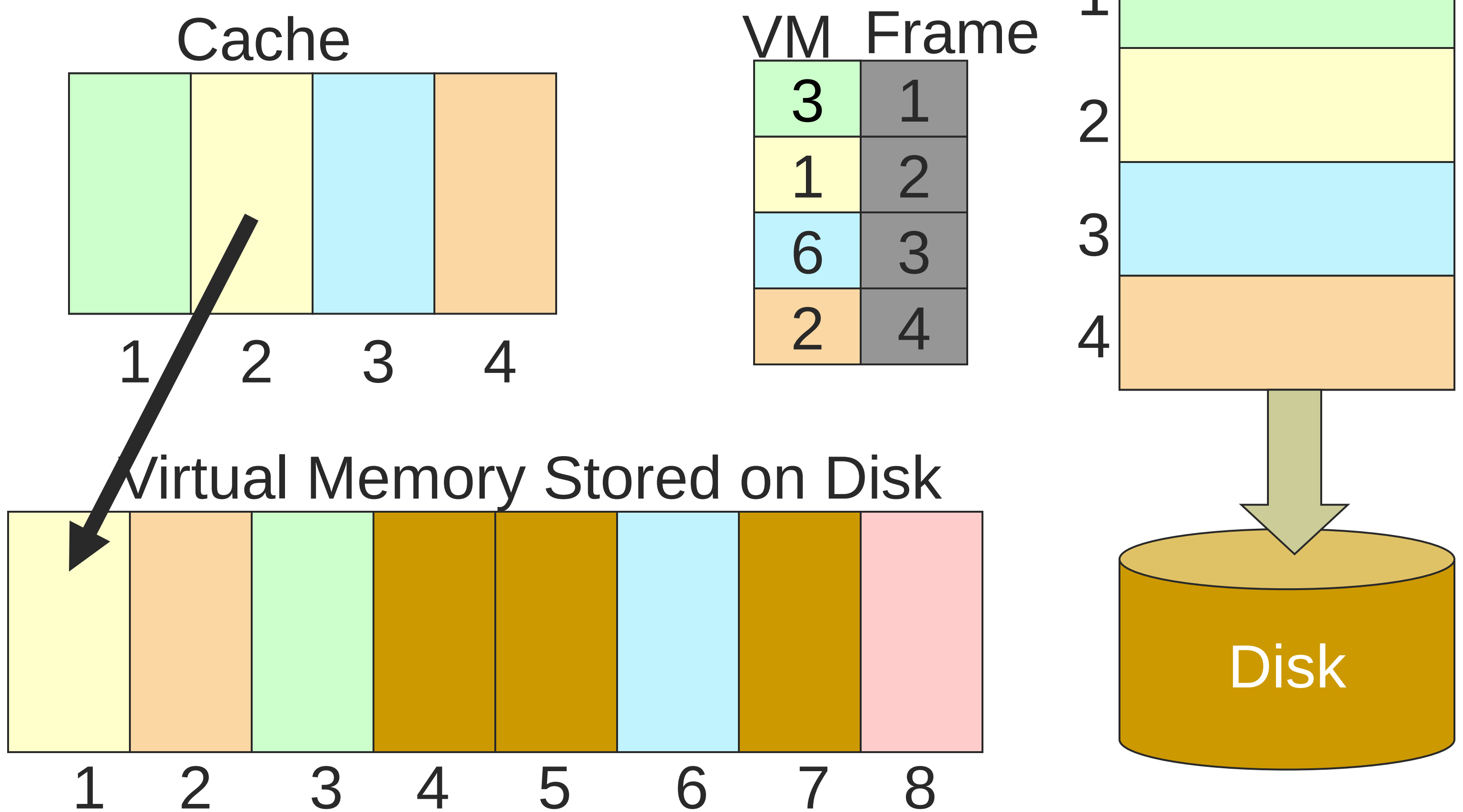
Request Address within Virtual Memory Page 8



What happens when there is no more space in the cache?

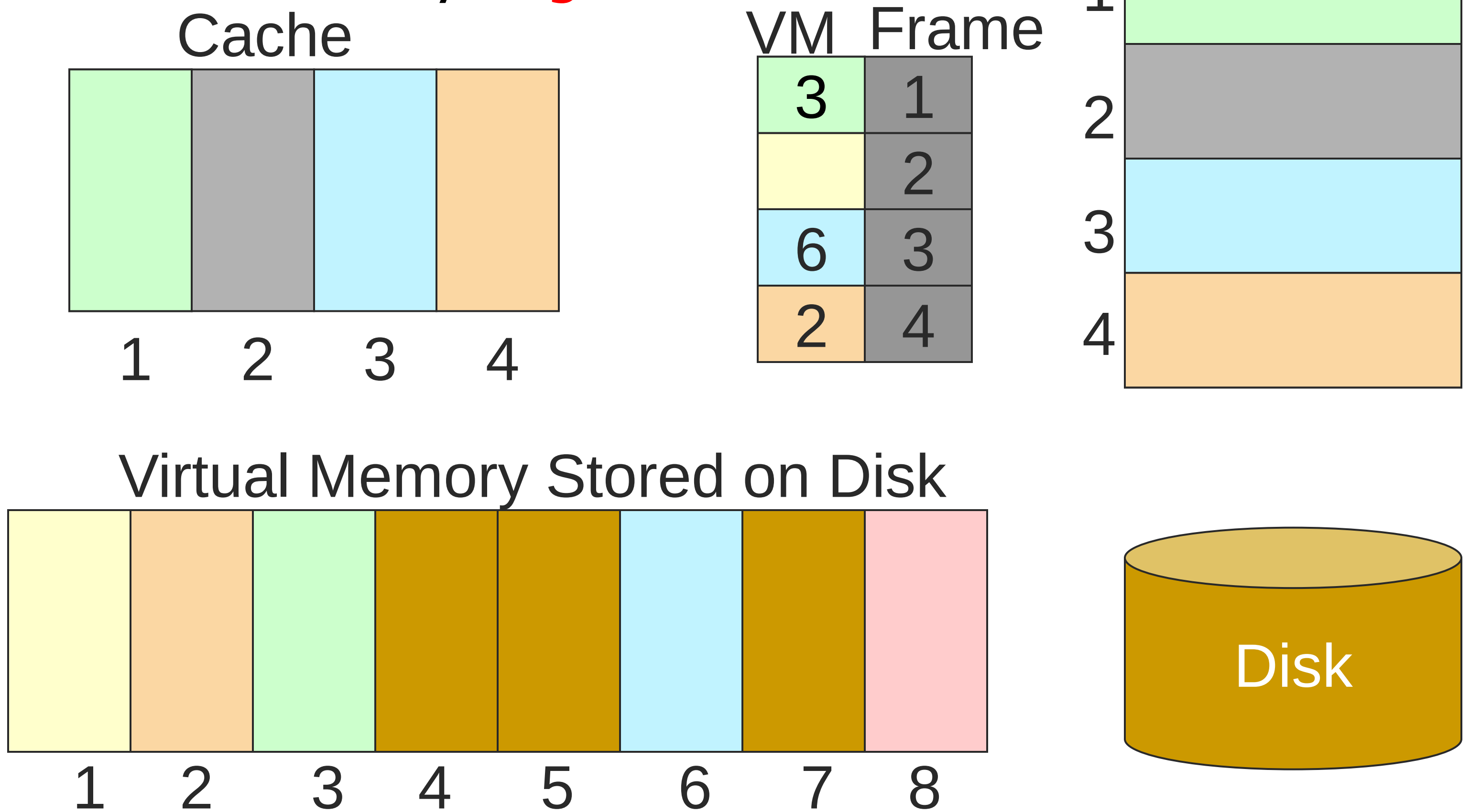
Paging Request

Store Virtual Memory
Page 1 to disk



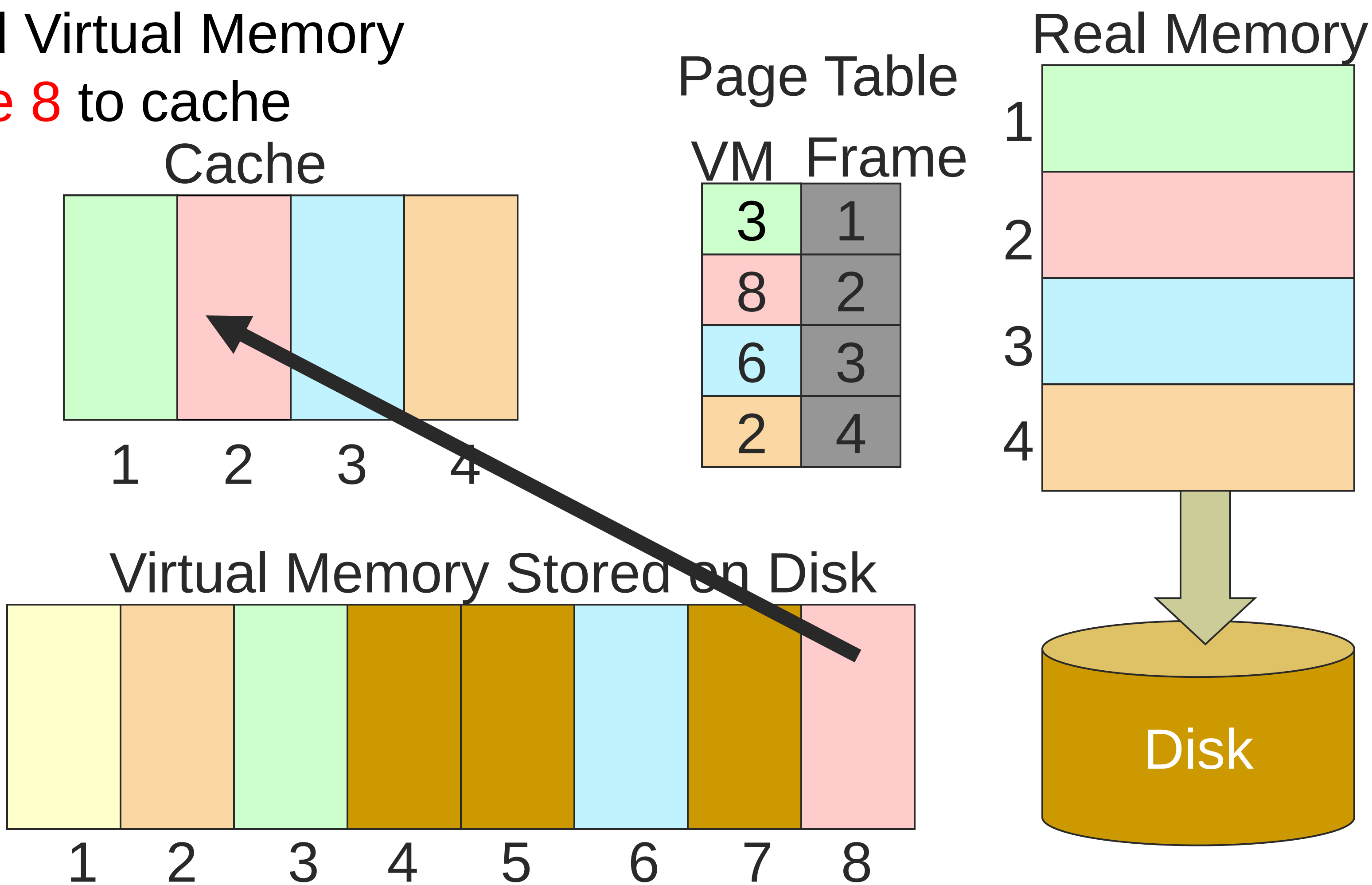
Paging Request

Process request for Address within Virtual Memory Page 8



Paging Request

Load Virtual Memory
Page 8 to cache



Paging Issues

- Page size
  - Typically 2^n
    - usually 512, 1k, 2k, 4k, or 8k
  - Example
    - 32 bit VM address may have 2^{20} (1 meg) pages with 4k (2^{12}) bytes per page
    - 2^{20} (1 meg) 32 bit page entries take 2^{22} bytes (4 meg)
  - Page frames must map into real memory

Paging Issues

- Physical memory size: 32 MB (2^{25})
  - Page size 4K bytes
  - How many pages?
    - 2^{13}
- NO external fragmentation
- Internal fragmentation on last page ONLY

Discussion

- How can paging be made faster?
  - Mapping must be done for every reference
  - More memory = more pages!
  - Hardware registers (one per page)
  - Keep page table in memory
- Is one level of paging sufficient?
- Sharing and protections?

Multi-level Translation

- Standard page table is a simple array
  - Might take huge amounts of memory for sparse address space.
    - 32 bit address space (4KB pages): $2^{20} * 4 = 4$ MB
    - 64 bit address space (4KB pages): $2^{52} * 8 = 32$ PB!
  - Multi-level translation changes this into a tree

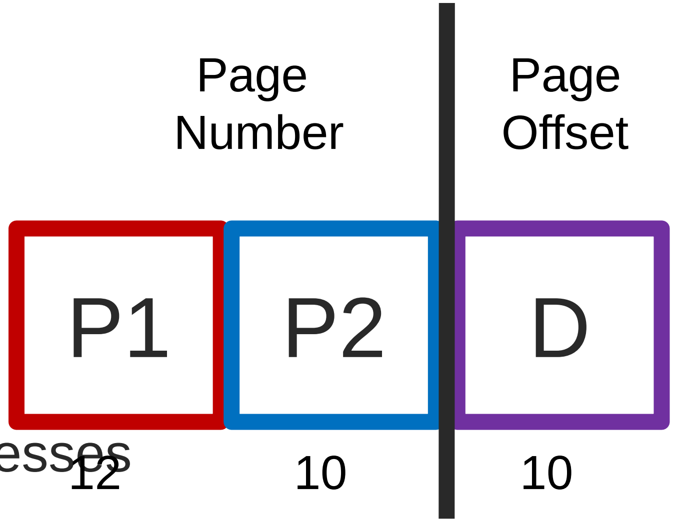
- E.g., two-level page table on 32 bit machine
  - Level 1 – virtual address bits 31-22 index
  - Level 2 – virtual address bits 21-12 index
  - Offset: bits 11-0 (4KB page)

Multilevel Paging and Performance

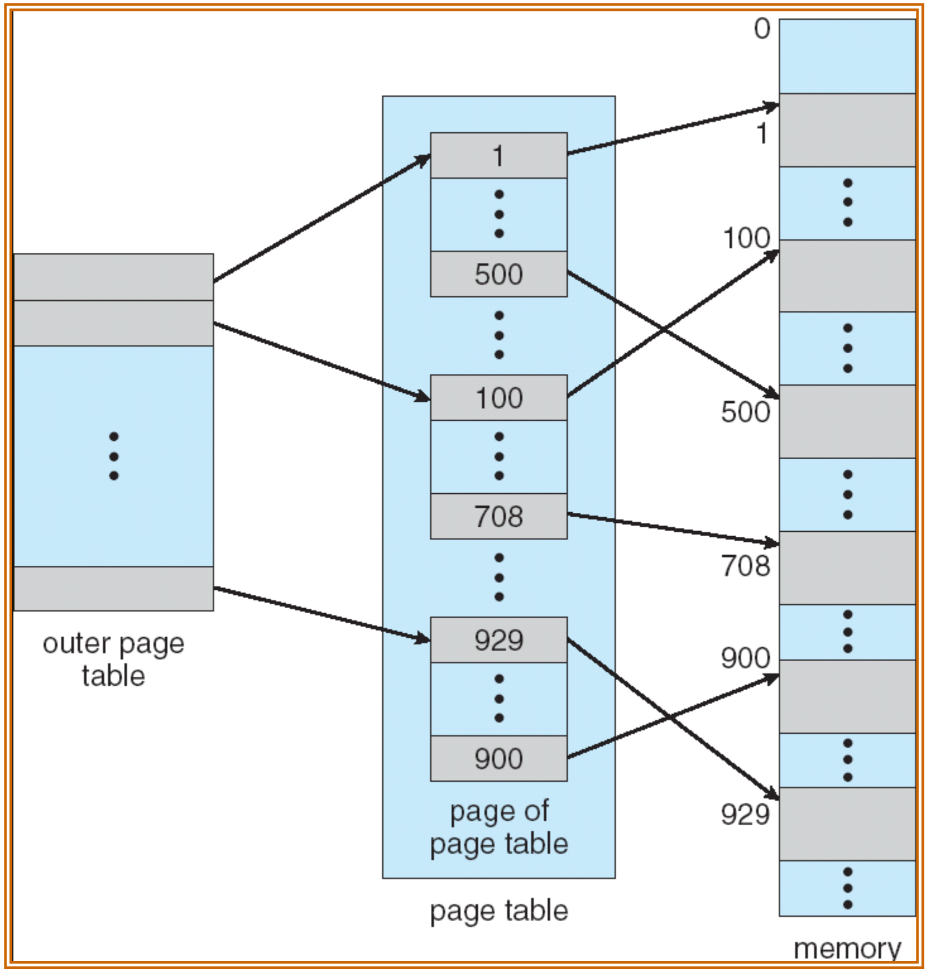
- Each level is stored as a separate table in memory
  - Converting a logical address to a physical one with a three-level page table may take four memory accesses
  - Why?

Addressing on Two-Level Page Table

- 32-bit Architecture
  - 4096= 2^{12} B Page
- 4K Page of Logical Memory
  - 4096 addressable bytes
- Page the Page Table
  - 4K pages as well
  - 1024 addressable 4byte addresses



Two-Level Page-Table



Problem (from Tanenbaum)

- A computer with a 32-bit address uses a two-level page table. Virtual addresses split into a 9-bit top-level page table field, an 11-bit second-level page table field, and an offset. How large are the pages and how many are there in the address space?

Problem

- Assume single-level page table
- Page table entry
  - Top 20 bits for physical address
  - Bottom 12 for permissions, etc.
  - Just like x86 page table entries
- Write a function, translate, that converts a virtual address to a physical address

Return the physical address

```
ulong translate(ulong va, pte_t *pt) {



}
```

Discussion

- How can paging be made faster?
  - Mapping must be done for every reference
    - 2 level page table, 3 memory ops per each load/store

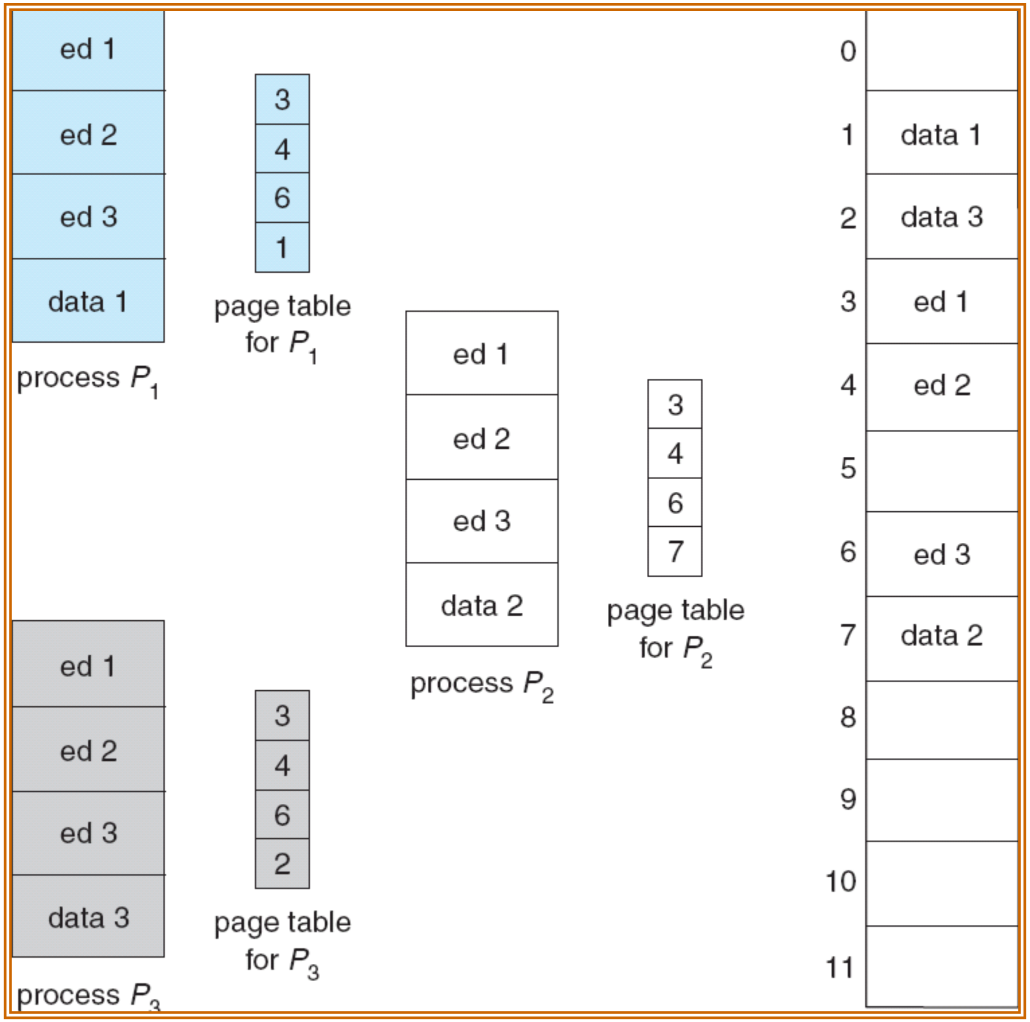
Paging - Caching the Page Table

- Cache page table entries in registers
  - Called a translation lookaside buffer
    - i.e., TLB
- Keep page table in memory
  - Location given by a *page table base register*
- Page table base register changed at context switch time

Sharing Pages

- Shared code
  - One copy of read-only code shared (e.g., libraries) among processes (e.g., text editors, compilers, web browsers).
- Private code and data
  - Each process keeps a separate copy of the code and data

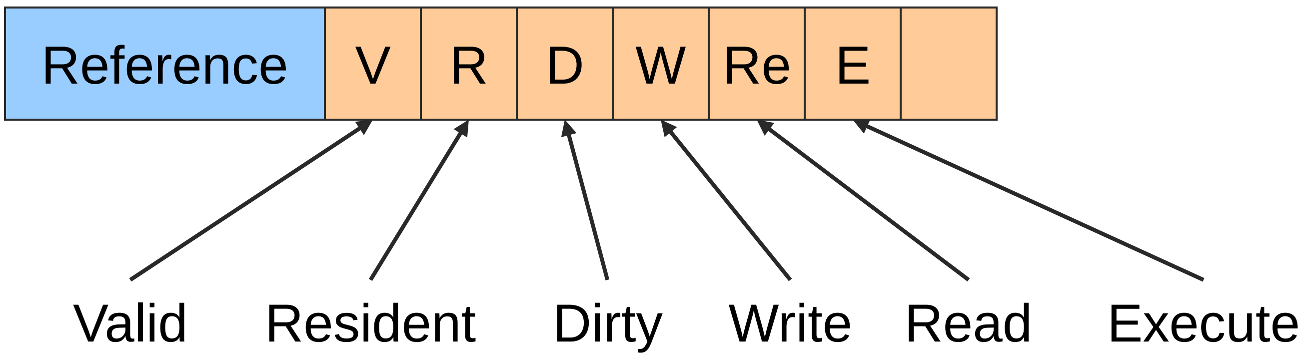
Shared Pages



Page Protection

- Can add read, write, execute protection bits to page table to protect memory
  - Check is done by hardware during access
  - Can give shared memory location different protections from different processes by having different page table protection access bits
- Valid-invalid bit attached to each entry in the page table
  - “valid” indicates that the associated page is in the process’ logical address space
  - “invalid” indicates that the page is not in the process’ logical address space

Page Protection



- Reference — page has been accessed
- Valid — page exists
- Resident — page is cached in primary memory
- Dirty — page has changed since page in

Demand Paging

- Never bring a page into primary memory until its needed
- Fetch Strategies
  - When should a page be brought into primary (main) memory from secondary (disk) storage.
- Placement Strategies
  - When a page is brought into primary storage, where should it be put?
- Replacement Strategies
  - Which page now in primary storage should be removed from primary storage when some other page or segment needs to be brought in and there is not enough room

Issue: Eviction

- Hopefully, kick out a less-useful page
  - Dirty pages require writing, clean pages don't
  - Where do you write? To "swap space"
- Goal: kick out the page that's least useful
- Problem: how do you determine utility?
  - Heuristic: temporal locality exists
  - Kick out pages that aren't likely to be used again

Principal of Optimality

- Definition
  - Each page is labeled with the number of instructions that will be executed before that page is first referenced
  - The optimal page replacement algorithm: choose the page with the highest label to be removed from the memory.
- Impractical: requires knowledge of future references
- If future references are known
  - should use pre paging to allow paging to be overlapped with computation.

Page Replacement Strategies

- Random page replacement
  - Choose a page randomly
- FIFO - First in First Out
  - Replace the page that has been in primary memory the longest
- LRU - Least Recently Used
  - Replace the page that has not been used for the longest time
- LFU - Least Frequently Used
  - Replace the page that is used least often
- NRU - Not Recently Used
  - An approximation to LRU.
- Working Set
  - Keep in memory those pages that the process is actively using.

Benefits of Virtual Memory

- Especially helpful in multiprogrammed system
  - CPU schedules process B while process A waits for its memory to be retrieved from disk
- Use secondary storage($)
  - Extend DRAM($$$) with reasonable performance
- Protection
  - Programs do not step over each other

Benefits of Virtual Memory

- Convenience
  - Flat address space
  - Programs have the same view of the world
  - Load and store cached virtual memory without user program intervention
- Reduce fragmentation
  - Make cacheable units all the same size (page)