Memory: Part II

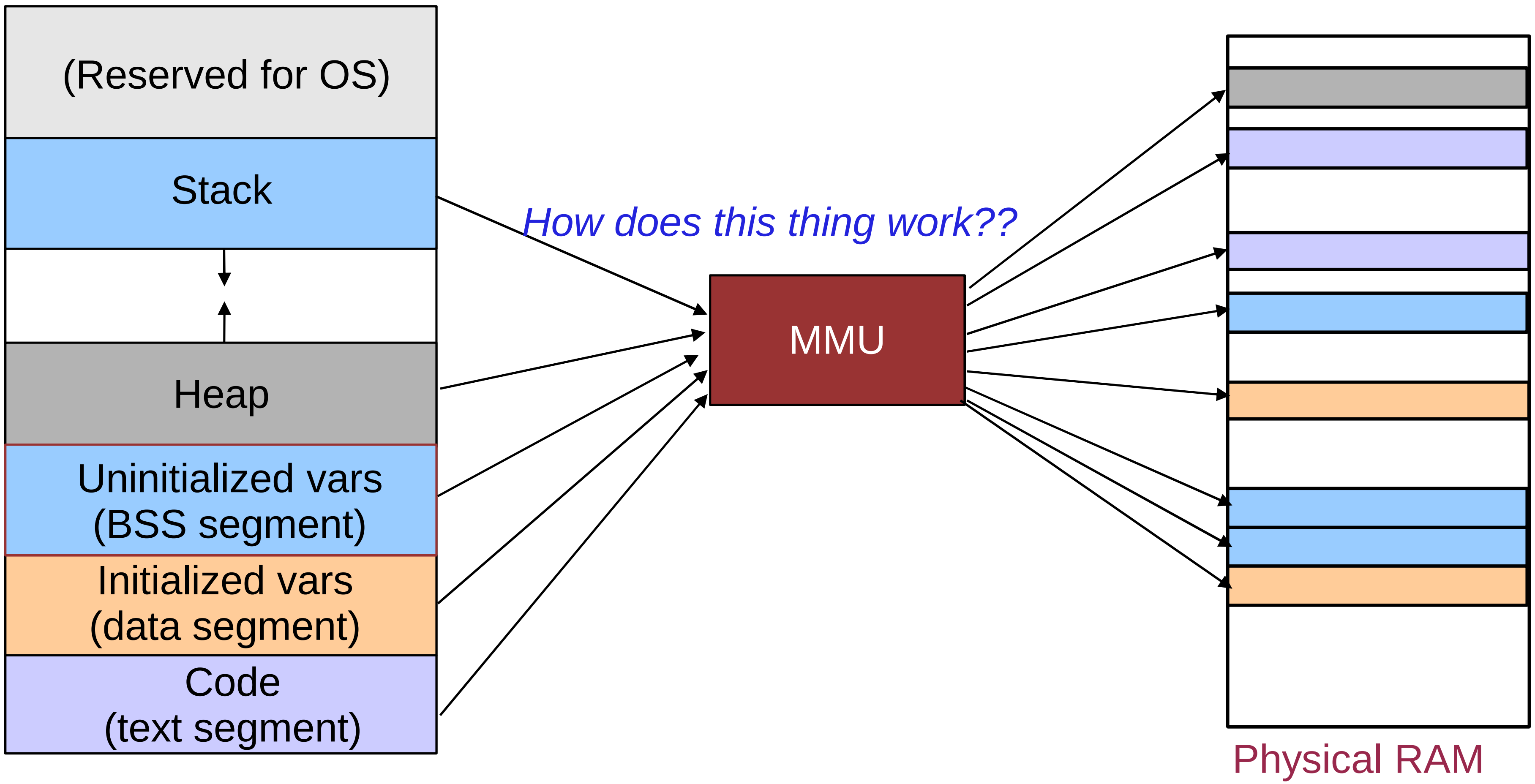
Administrivia

- Matt Caesar's office hours
  - W 5-7pm, 3118
  - And by request

Recap: Virtual Addresses

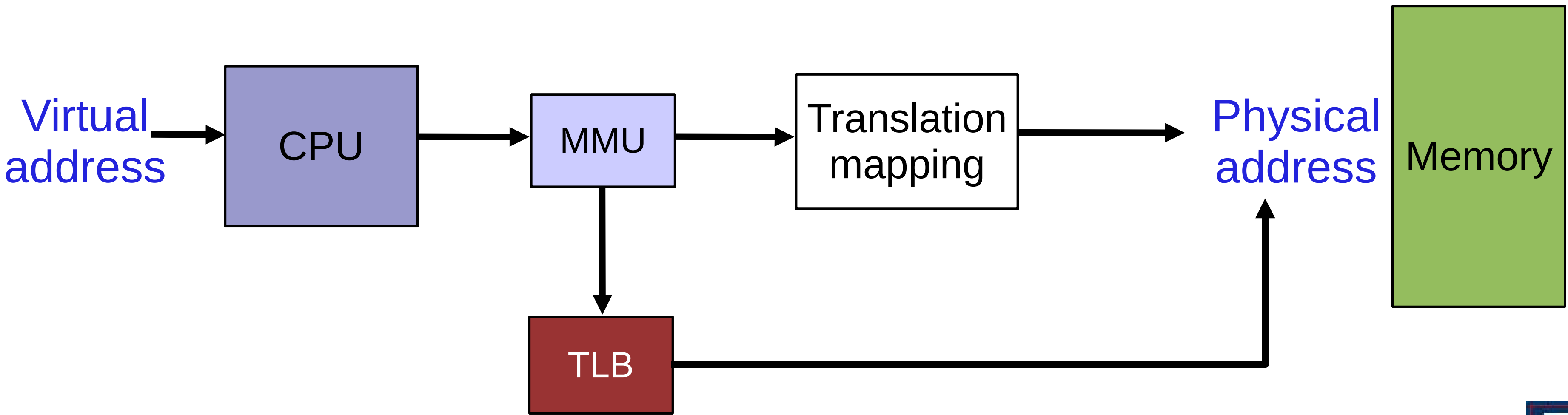
- A *virtual address* is a memory address that a process uses to access its own memory
  - The virtual address is *not the same* as the actual physical RAM address in which it is stored
  - When a process accesses a virtual address, the MMU hardware *translates* the virtual address into a physical address
  - The OS determines the mapping from virtual address to physical address
- Benefit: Isolation
  - Virtual addresses in one process refer to ***different*** physical memory than virtual addresses in another
  - Exception: shared memory regions between processes (discussed later)
- Benefit: Illusion of larger memory space
  - Can store unused parts of virtual memory on disk temporarily
- Benefit: Relocation
  - A program does not need to know which physical addresses it will use when it's run

Mapping virtual to physical addresses



MMU and TLB

- Memory Management Unit (MMU)
  - Hardware that translates a virtual address to a physical address
  - Each memory reference is passed through the MMU
  - Translate a virtual address to a physical address
    - Lots of ways of doing this!
- Translation Lookaside Buffer (TLB)
  - Cache for MMU virtual-to-physical address translations
  - Just an optimization – but an important one!



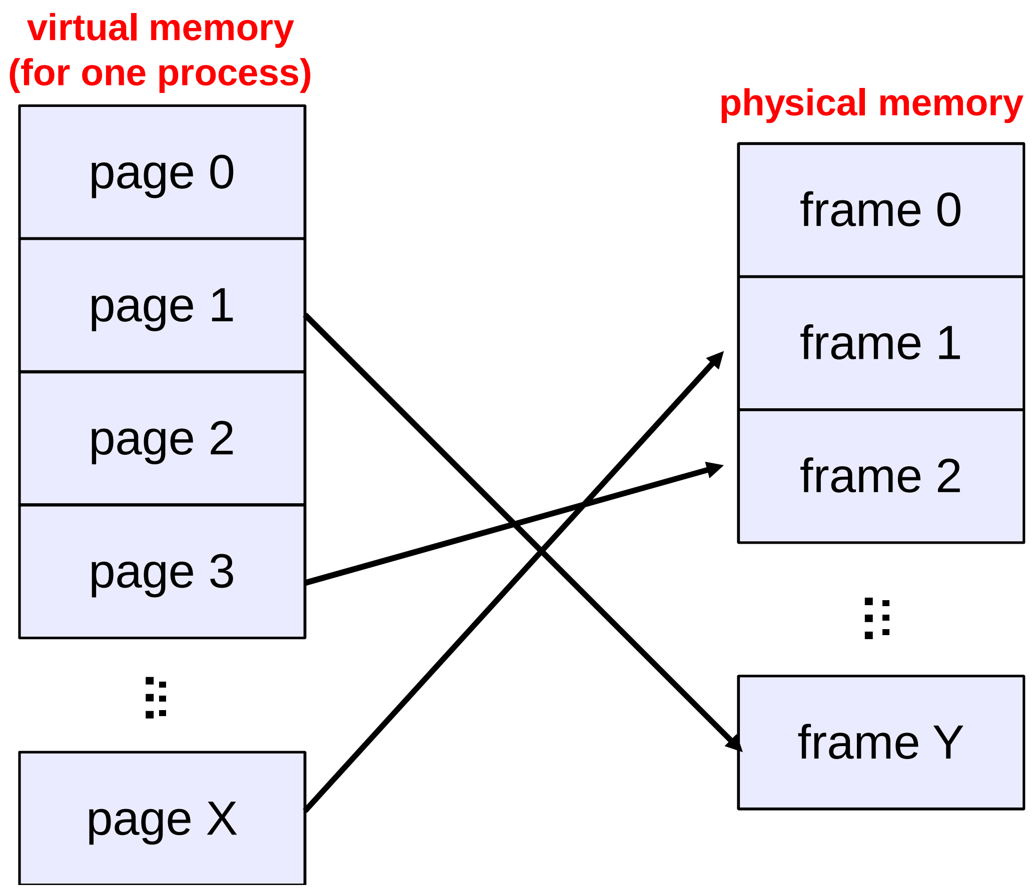
Cache of translations

Recap: dividing up memory

- Fixed partitions
  - Break memory into equally-sized pieces
  - Problem: no single size appropriate for all programs
- Variable partitions (segments)
  - Resize pieces based on process needs
  - Problem: external fragmentation
    - As jobs run and complete, holes left in physical memory
- Modern approach: Paging
  - We'll discuss this today

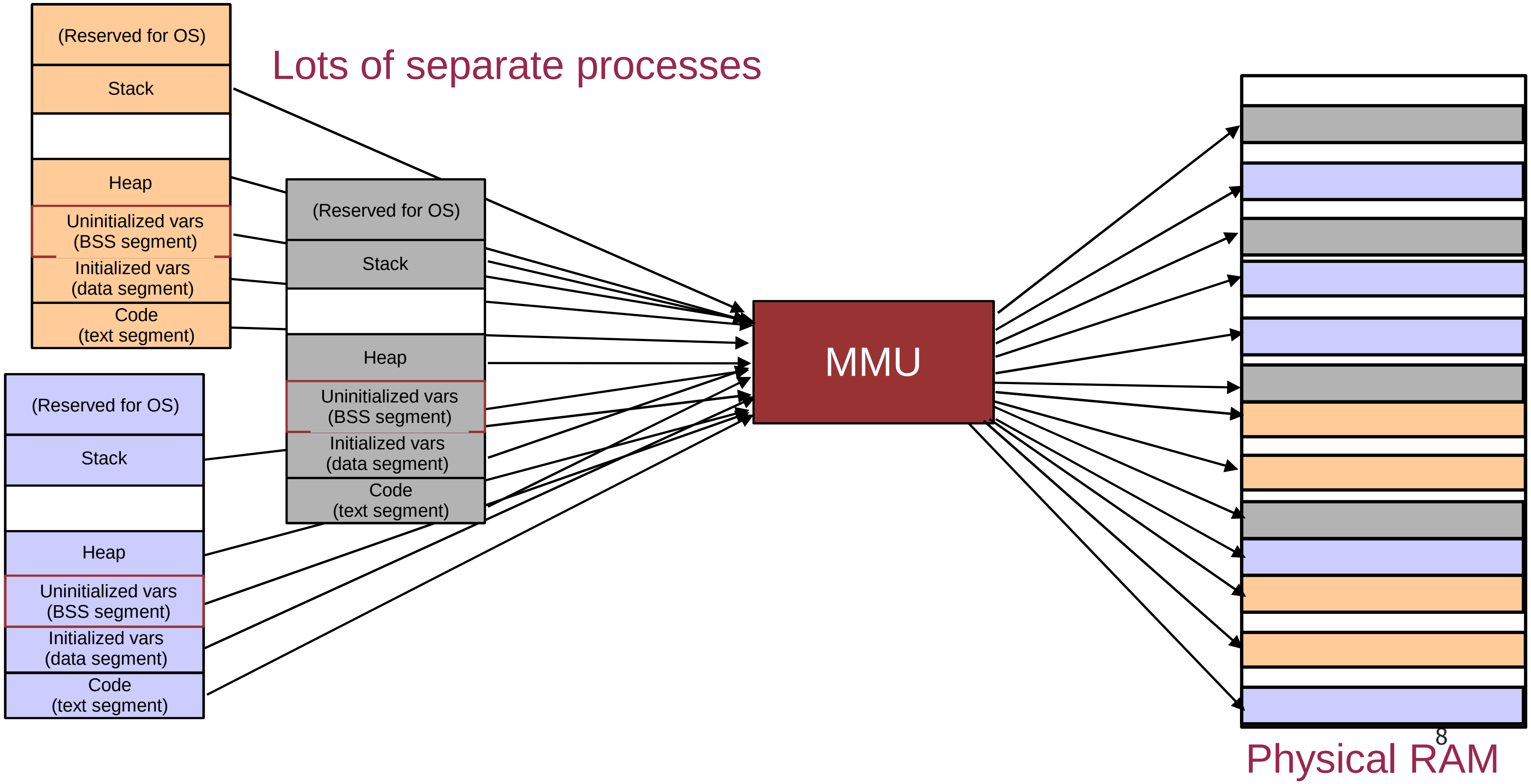
Paging

- Solve the external fragmentation problem by using fixed-size chunks of virtual and physical memory
  - Virtual memory unit called a *page*
  - Physical memory unit called a *frame* (or sometimes *page frame*)



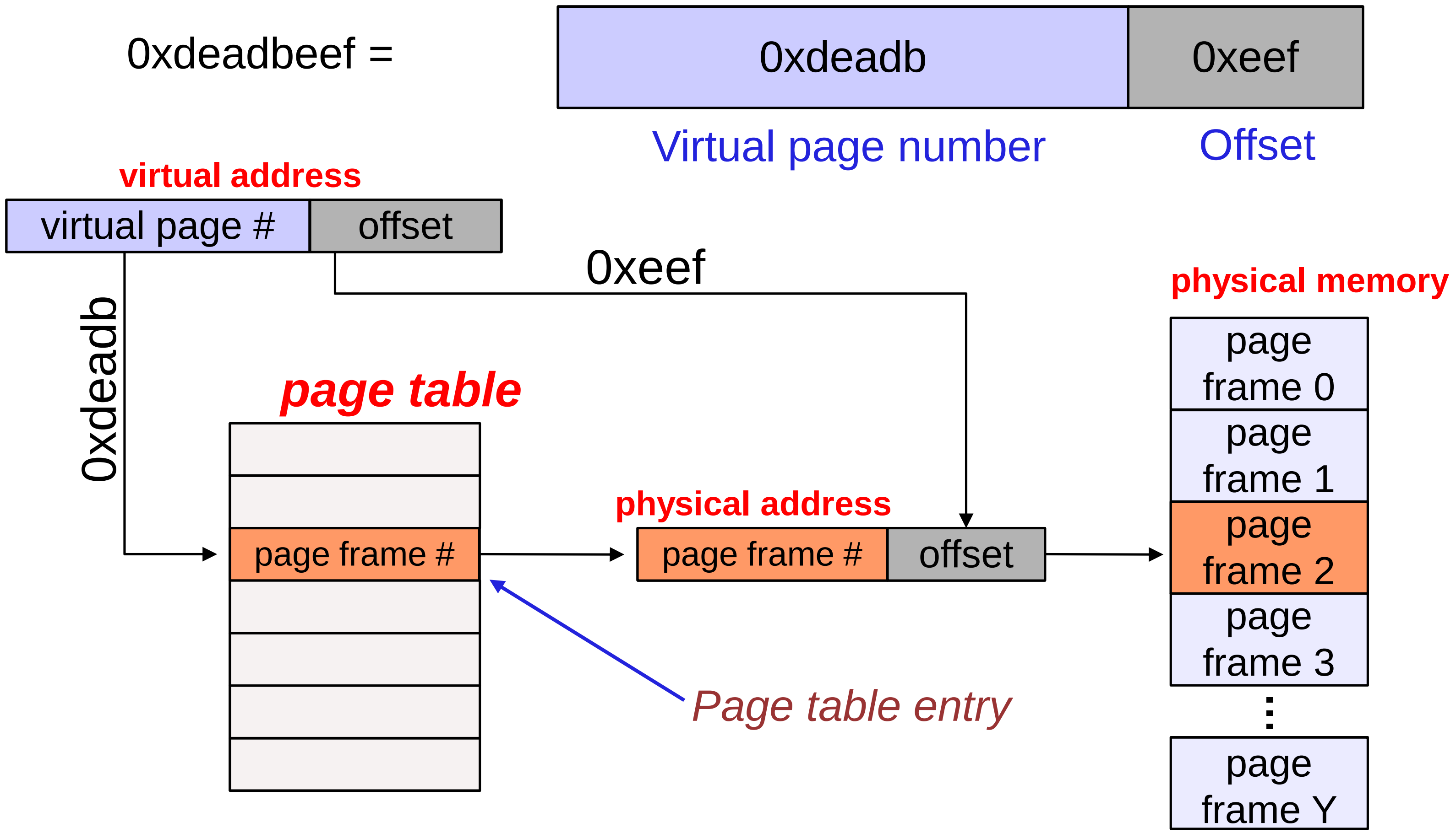
Application Perspective

- Application believes it has a single, contiguous address space ranging from 0 to 2P – 1 bytes
  - Where P is the number of bits in a pointer (e.g., 32 bits)
- In reality, virtual pages are scattered across physical memory
  - This mapping is invisible to the program, and not even under it's control!



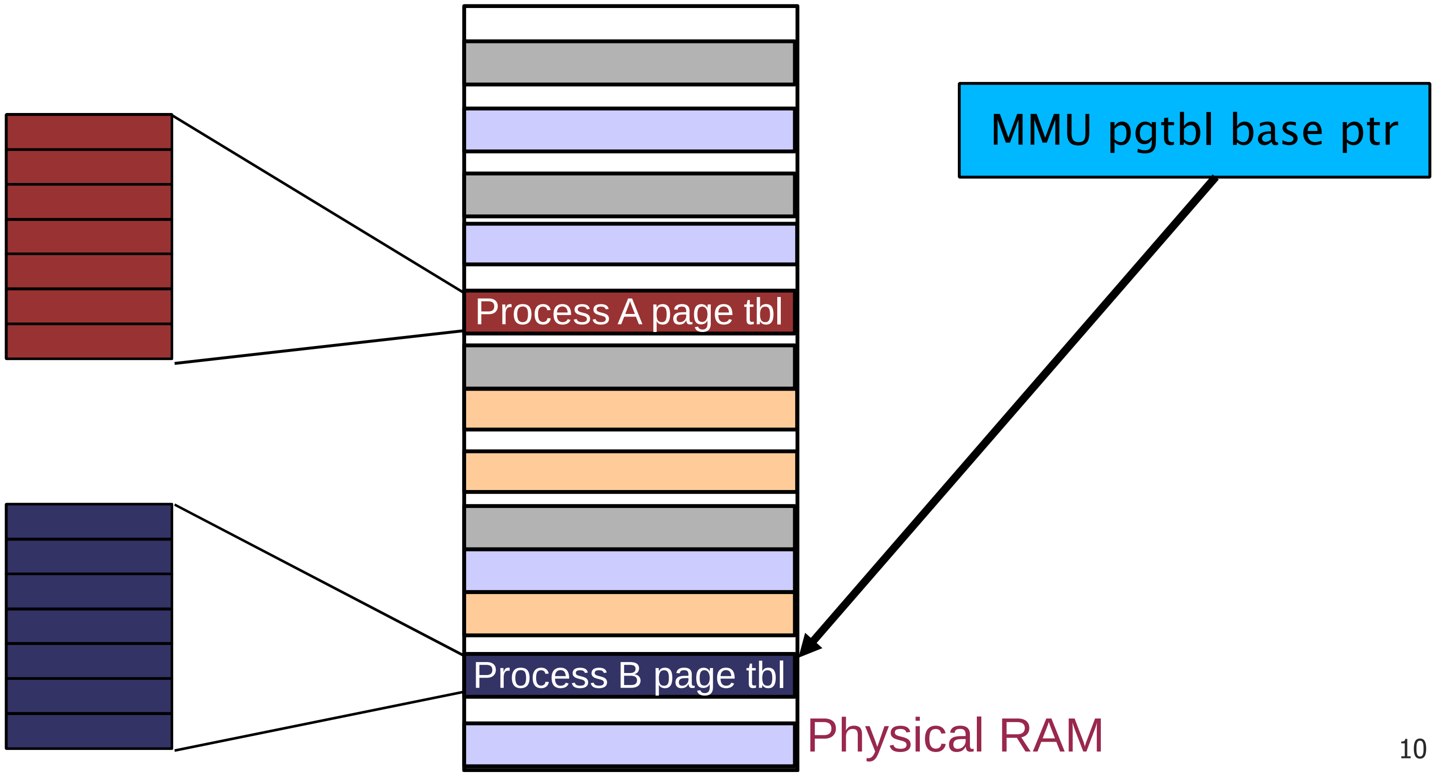
Looking up a Page

- Virtual-to-physical address translation performed by MMU
  - Virtual address is broken into a *virtual page number* and an *offset*
  - Mapping from virtual page to physical frame provided by a *page table* (which is stored in memory)



Page Tables

- Page Tables store the virtual-to-physical address mappings.
- Where are they located? *In memory!*
- OK, then. How does the MMU access them?
  - The MMU has a special register called the *page table base pointer*.
  - This points to the *physical memory address* of the top of the page table for the currently-running process.



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

Advantages of Paging

- Simplifies physical memory management
  - OS maintains a free list of physical page frames
  - To allocate a physical page, just remove an entry from this list
- No external fragmentation!
  - Virtual pages from different processes can be interspersed in physical memory
  - No need to allocate pages in a contiguous fashion
- Allocation of memory can be performed at a fine granularity
  - Only allocate physical memory to those parts of the address space that require it
  - Can swap unused pages out to disk when physical memory is running low
  - Idle programs won't use up a lot of memory (even if their address space is huge!)

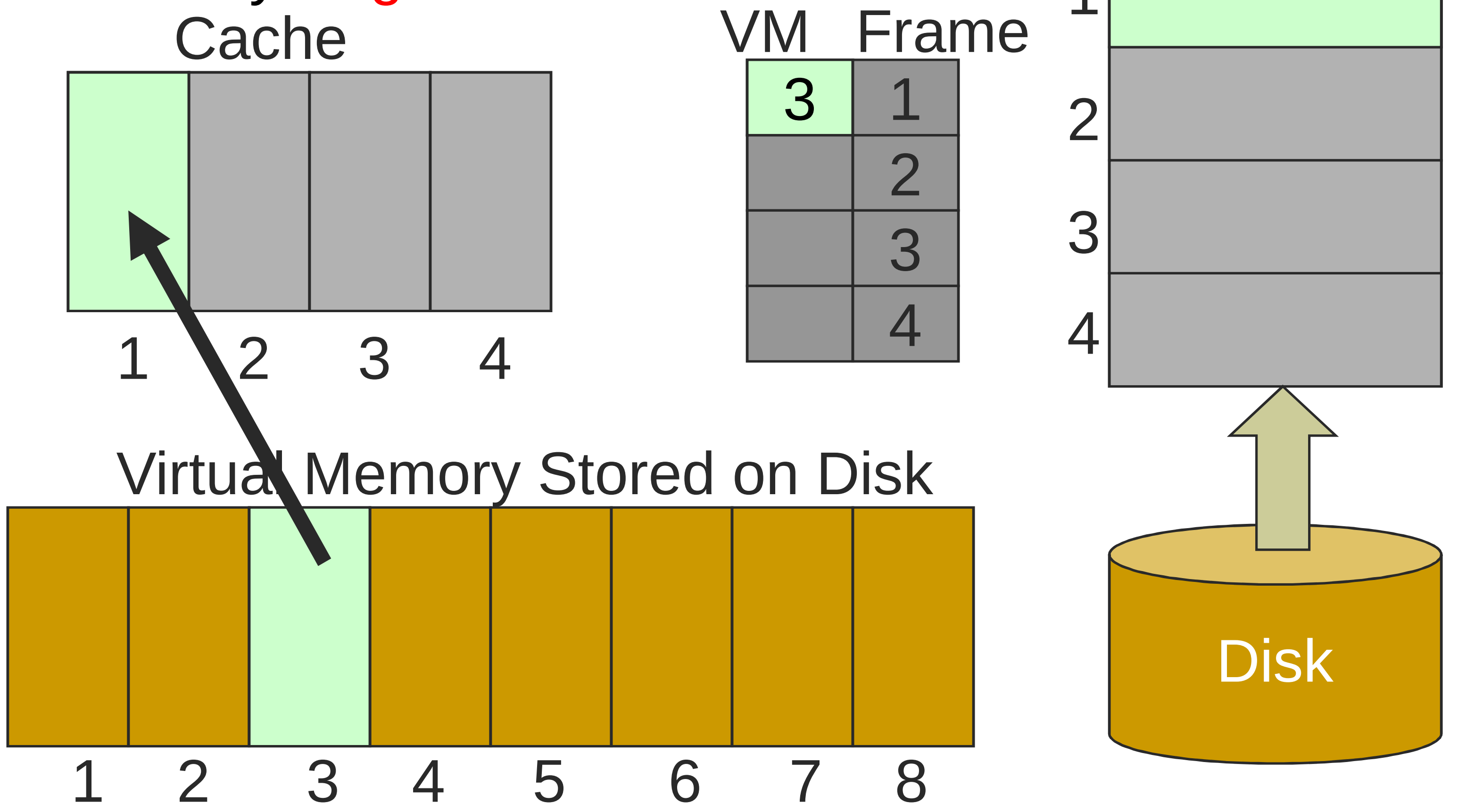
Translation Process

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

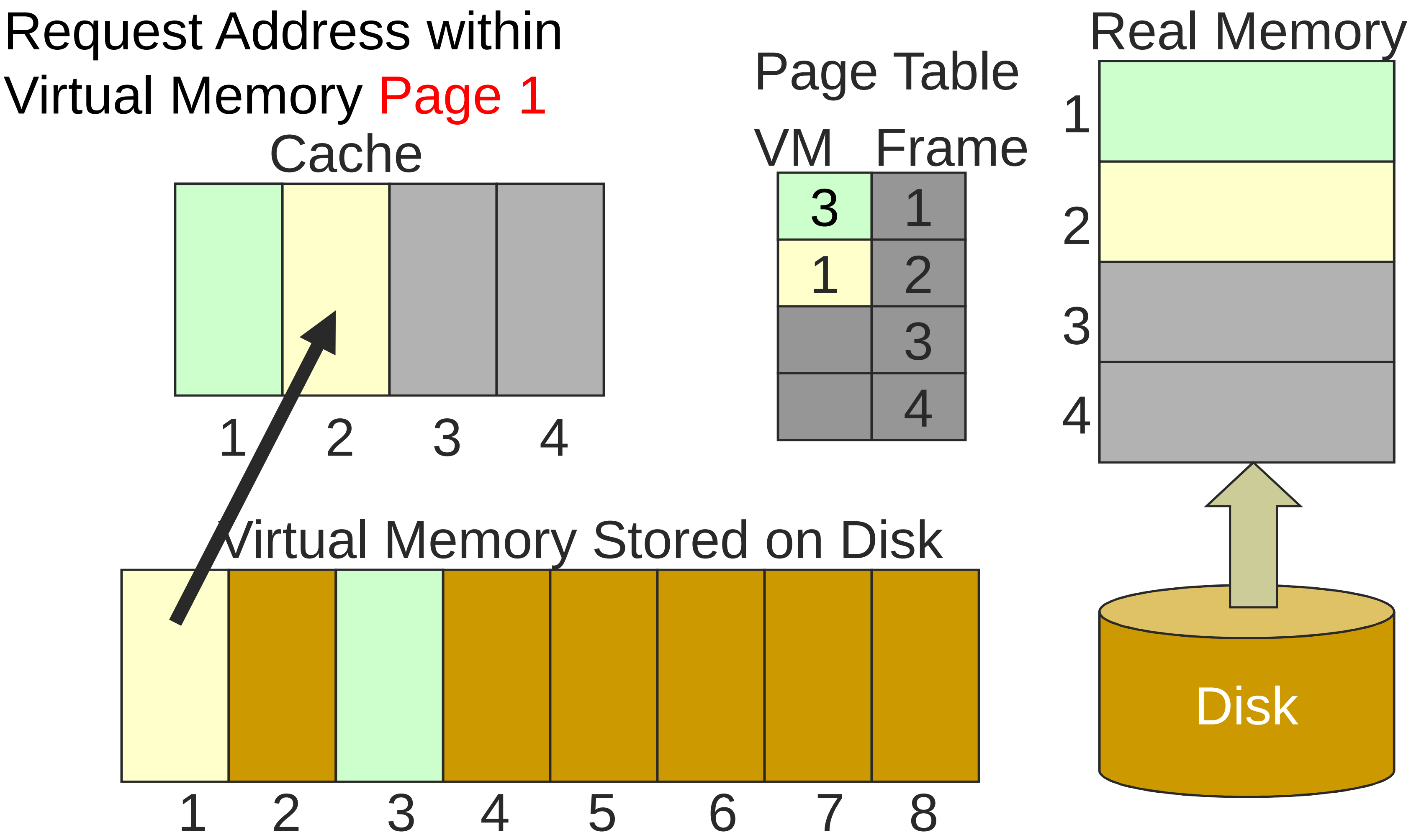
- Each virtual page can be in physical memory or swapped out to disk (called paged)
- What must change on a context switch?
  - Could copy entire contents of table, but this will be slow
  - Instead use an extra layer of indirection and change the pointer to the page table

Paging Example

Request Address within Virtual Memory Page 3

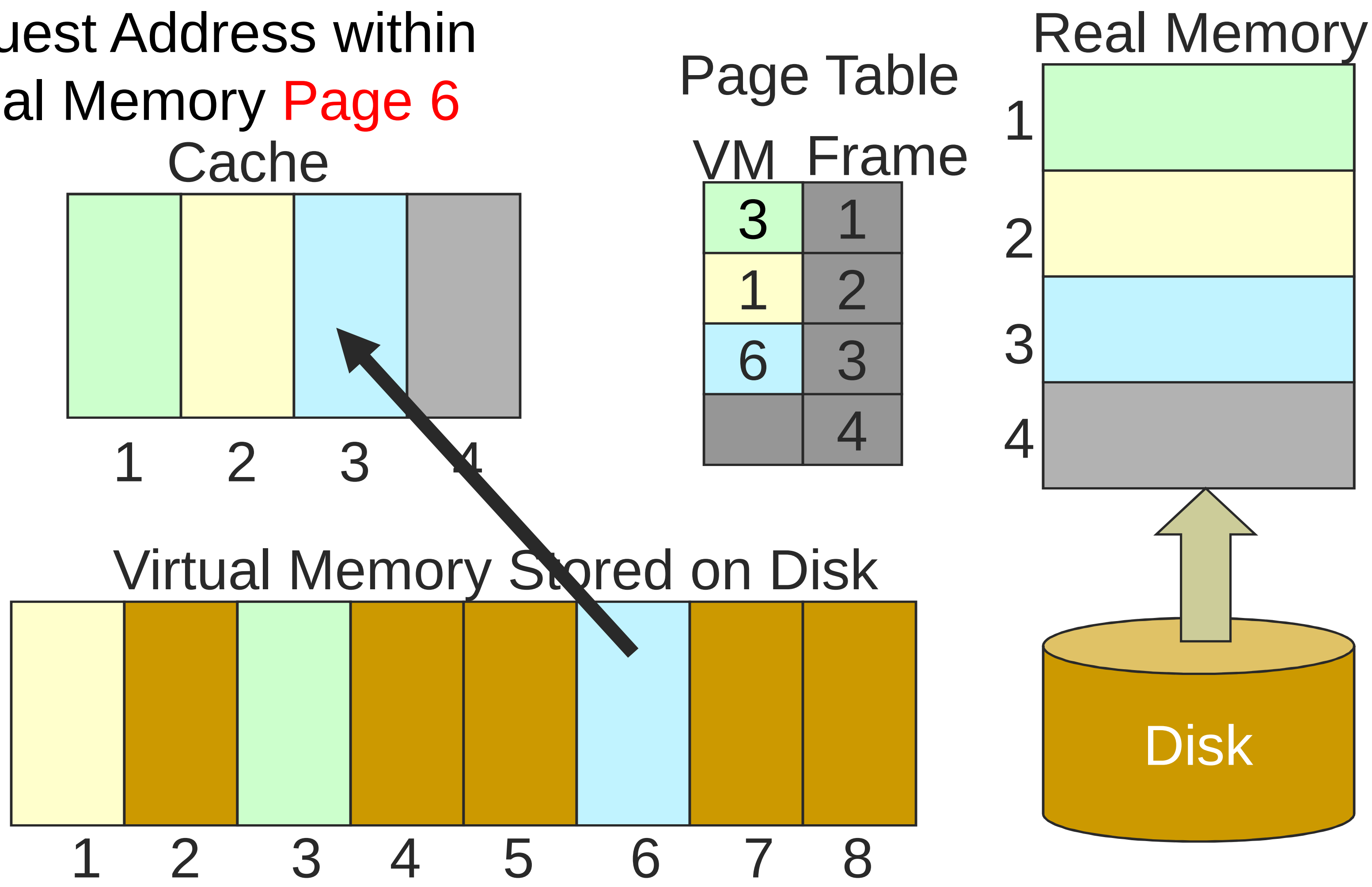


Paging Example



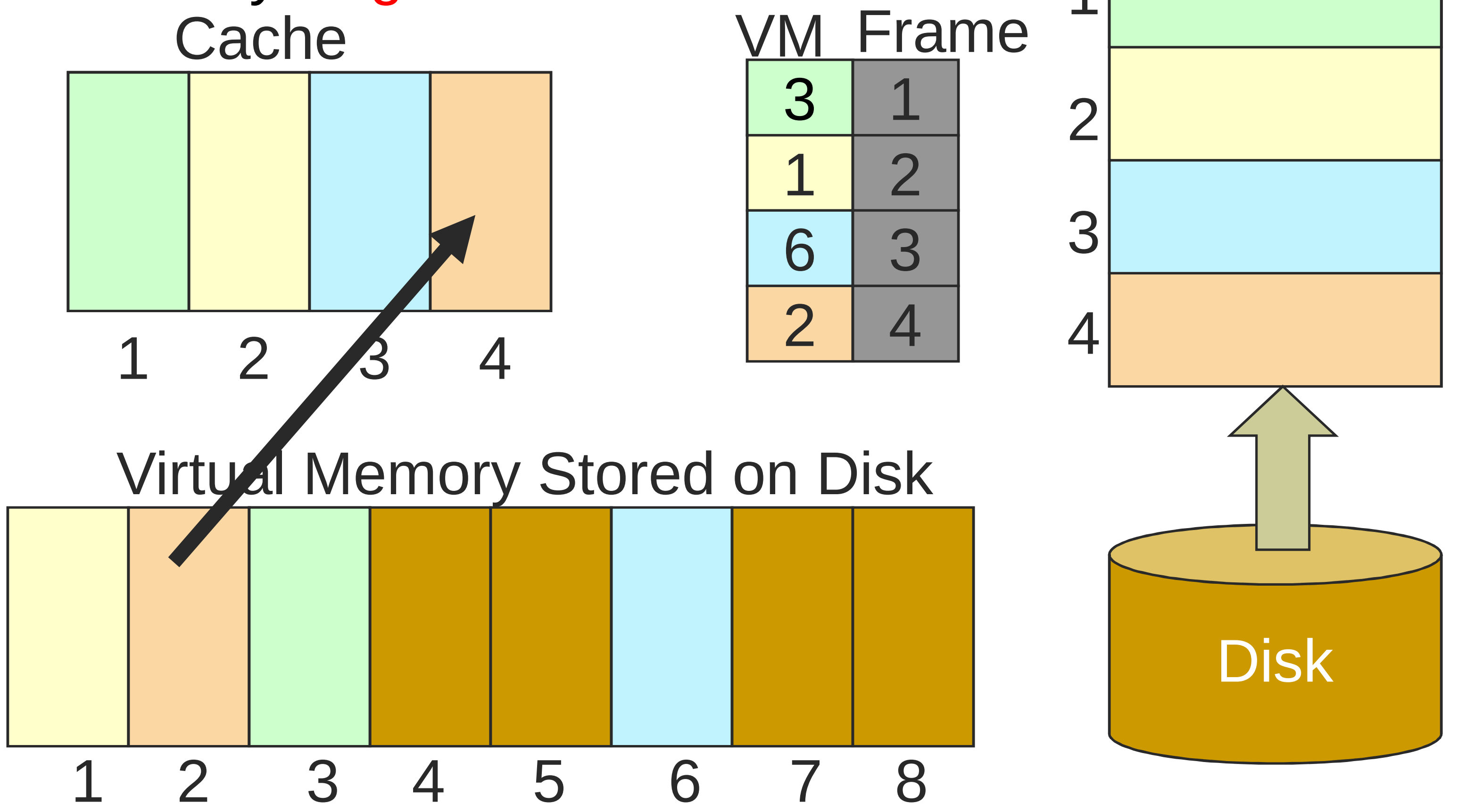
Paging Example

Request Address within Virtual Memory Page 6



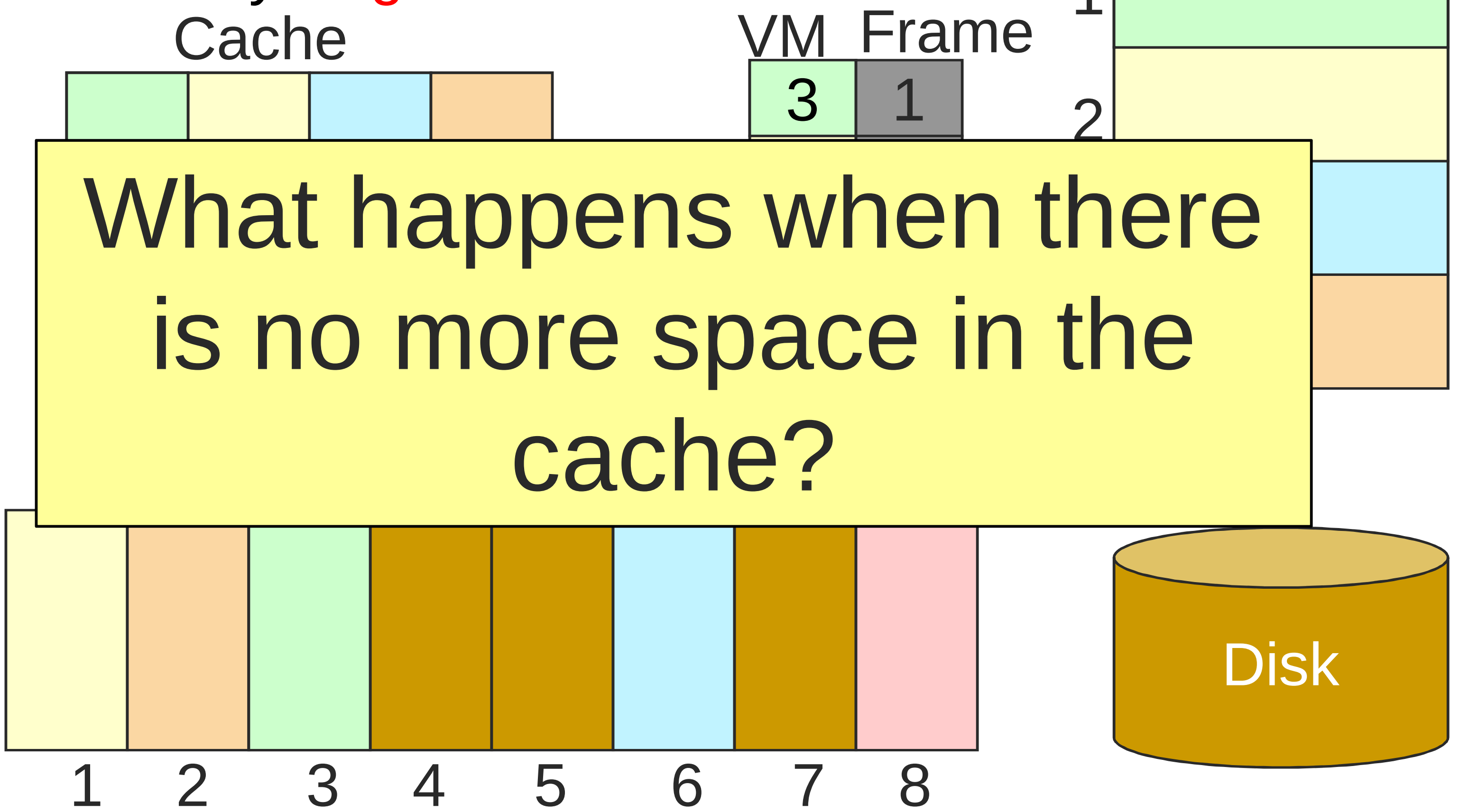
Paging Example

Request Address within Virtual Memory Page 2



Paging Example

Request Address within Virtual Memory Page 8



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

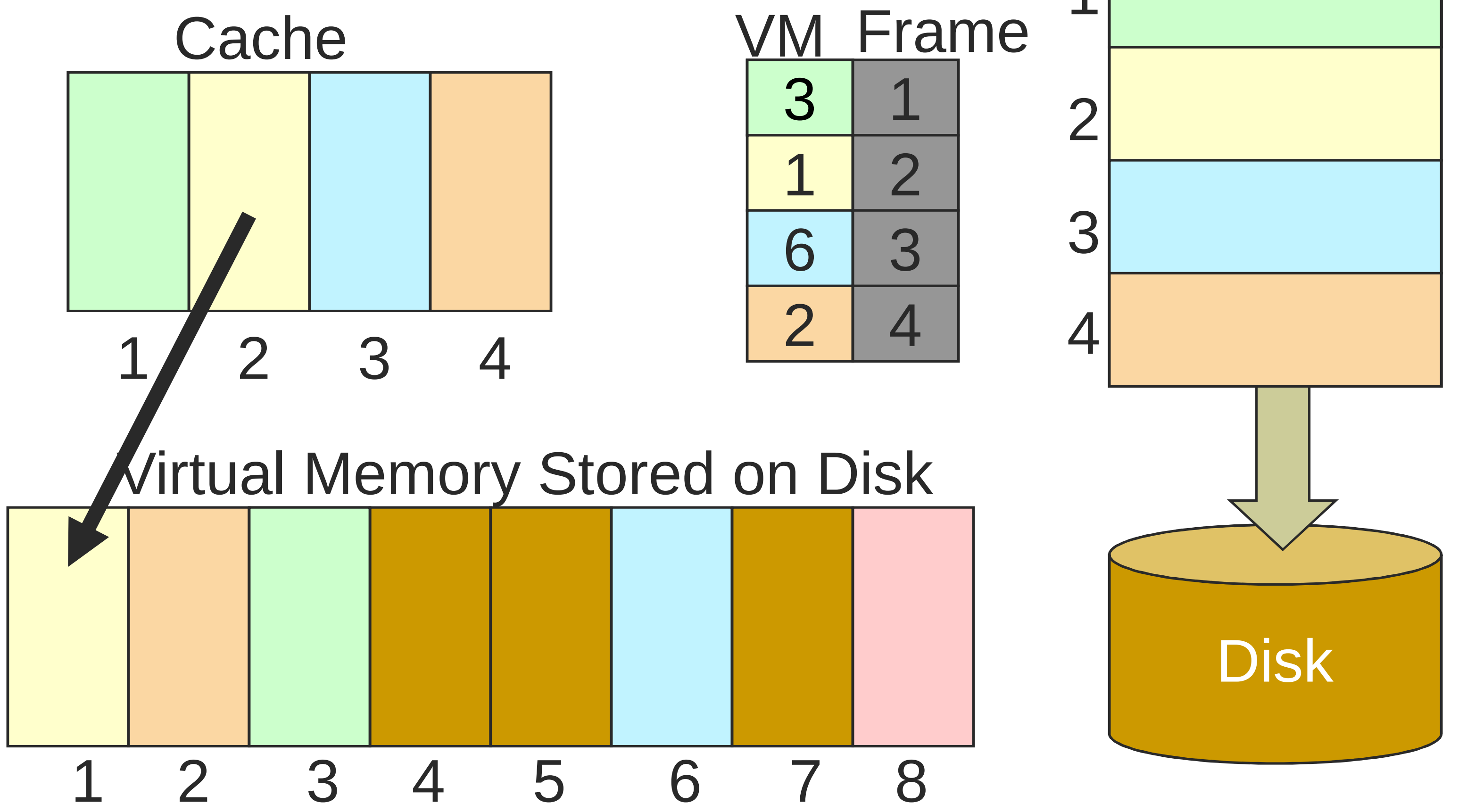
Paging Example

Store Virtual Memory

Page 1 to disk

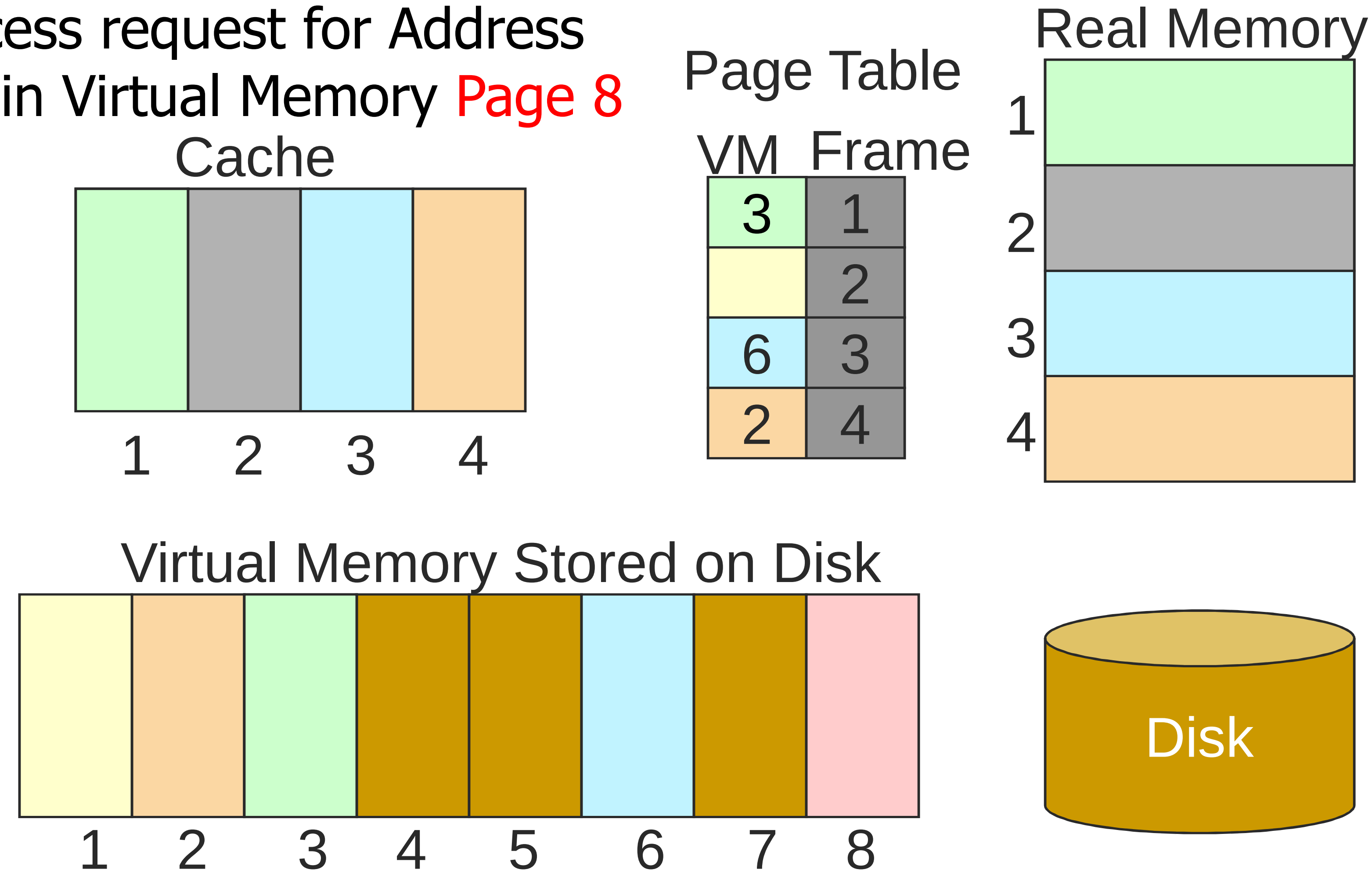
Page Table

VM	Frame
3	1
1	2
6	3
2	4



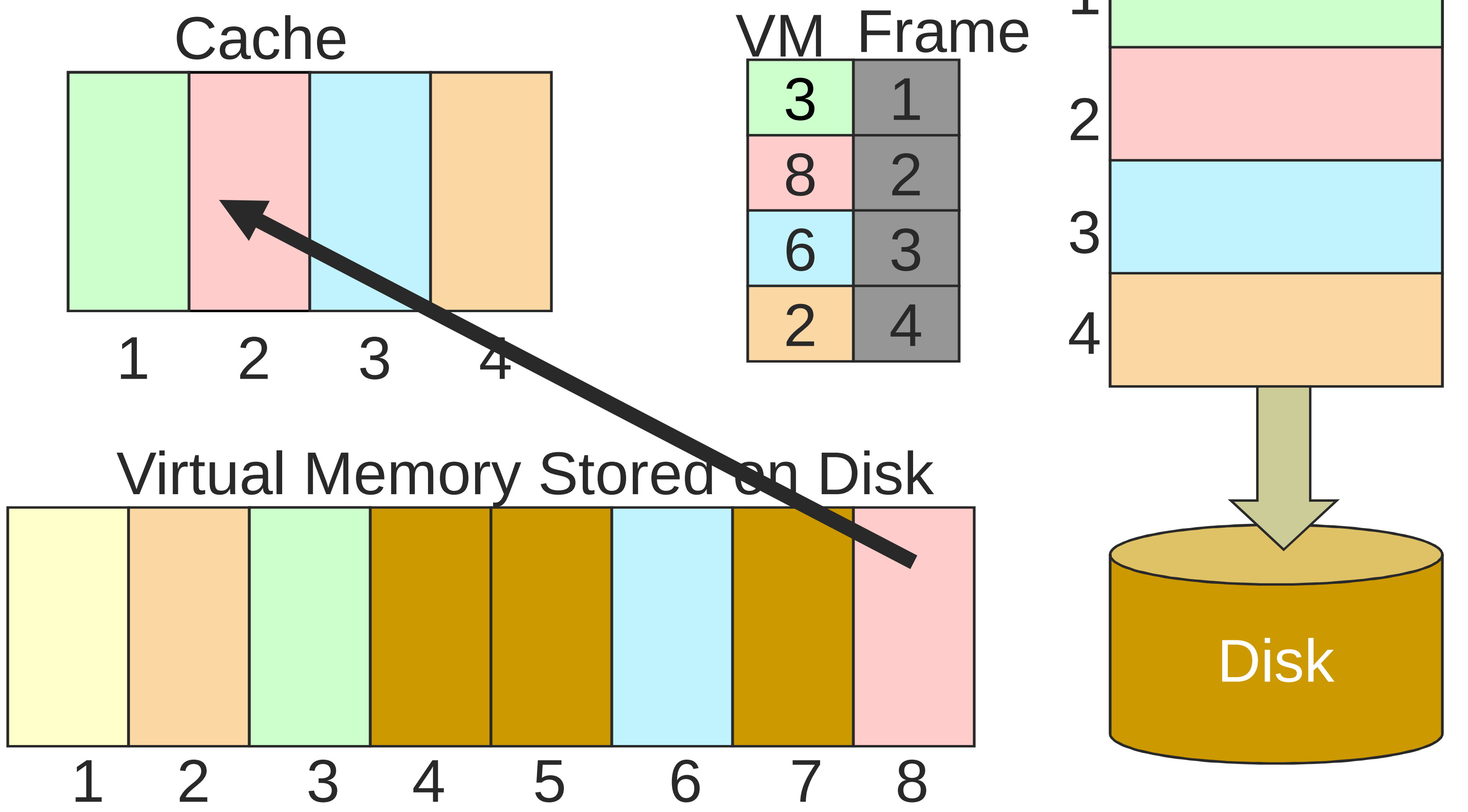
Paging Example

Process request for Address within Virtual Memory Page 8



Paging Example

Load Virtual Memory
Page 8 to cache



Paging

- Like segments, pages can have different protections
  - Read, write, execute
- How does the processor know that a virtual page is not in memory?
  - Resident bit tells the hardware that the virtual address is non-resident

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

Page Table Entry

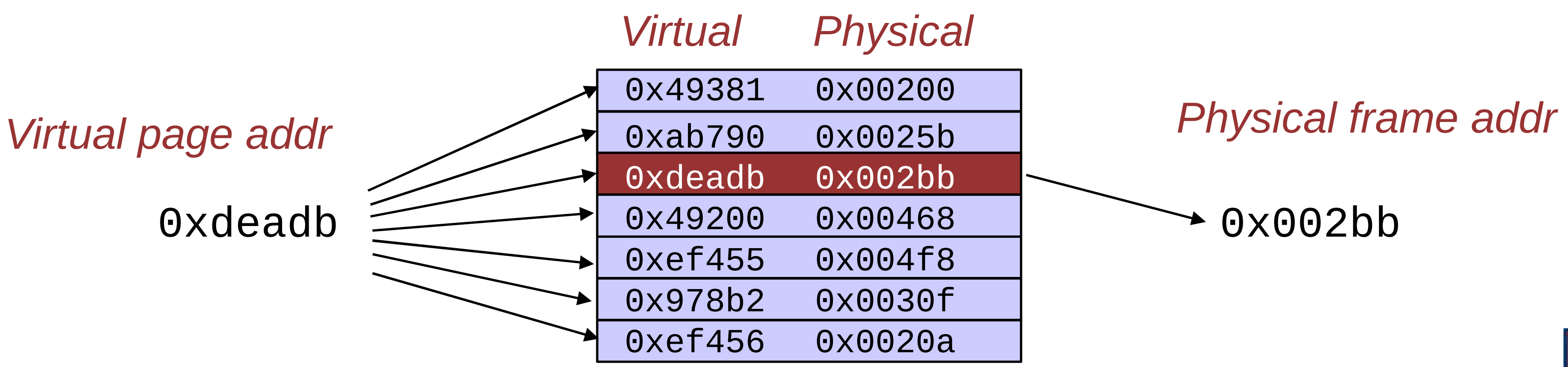
- Typical PTE format (depends on CPU architecture!)

1	1	1	2	20
M	R	V	prot	page frame number

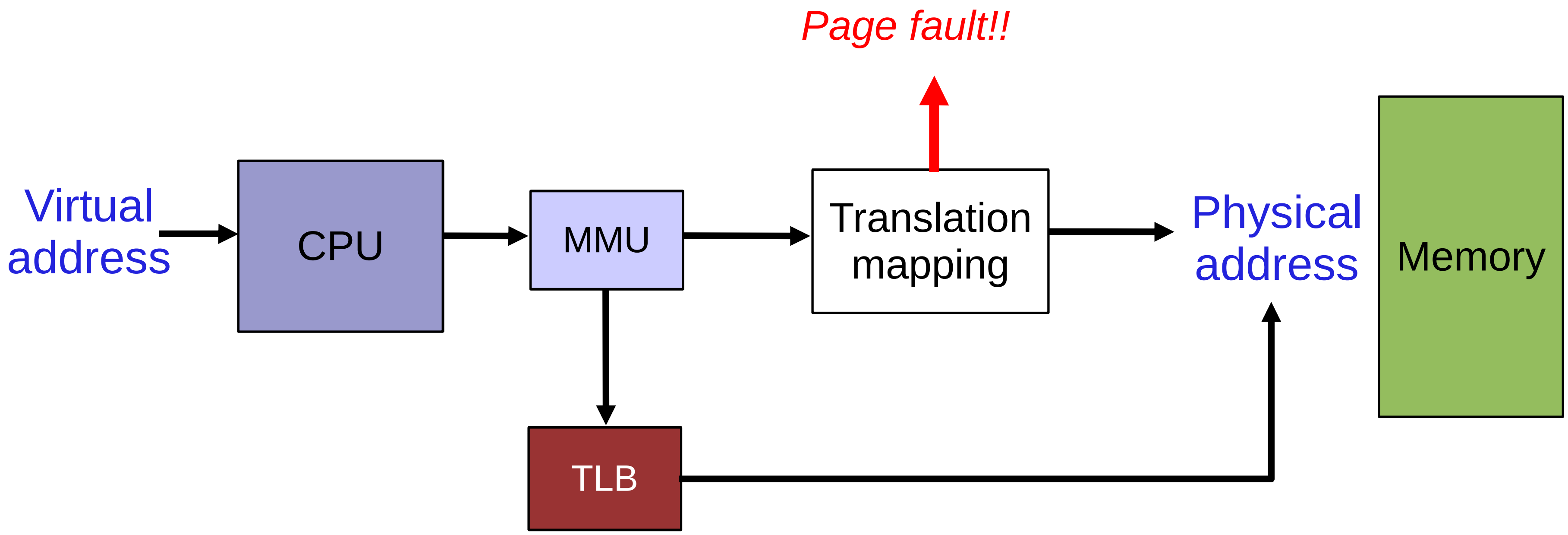
- Various bits accessed by MMU on each page access:
  - Modify bit: Indicates whether a page is "dirty" (modified)
  - Reference bit: Indicates whether a page has been accessed (read or written)
  - Valid bit: Whether the PTE represents a real memory mapping
  - Protection bits: Specify if page is readable, writable, or executable
  - Page frame number: Physical location of page in RAM
    - Why is this 20 bits wide in the above example?

Speeding up lookups with a TLB

- Now we've introduced a high overhead for address translation
  - On every memory access, must have a *separate* access to consult the page tables!
- Solution: *Translation Lookaside Buffer (TLB)*
  - Very fast (but small, eg 128 entries on P6) cache directly on the CPU
  - Caches most recent virtual to physical address translations
  - Implemented as fully associative cache
  - Any address can be stored in any entry in the cache
  - All entries searched "in parallel" on every address translation
  - A TLB miss requires that the MMU actually try to do the address translati



Page Faults



- When a virtual address translation cannot be performed, it's called a *page fault*
  - Triggers trap to kernel to handle fault
  - Page faults are *not* errors
- What could cause a page fault?

Reasons for Page Faults

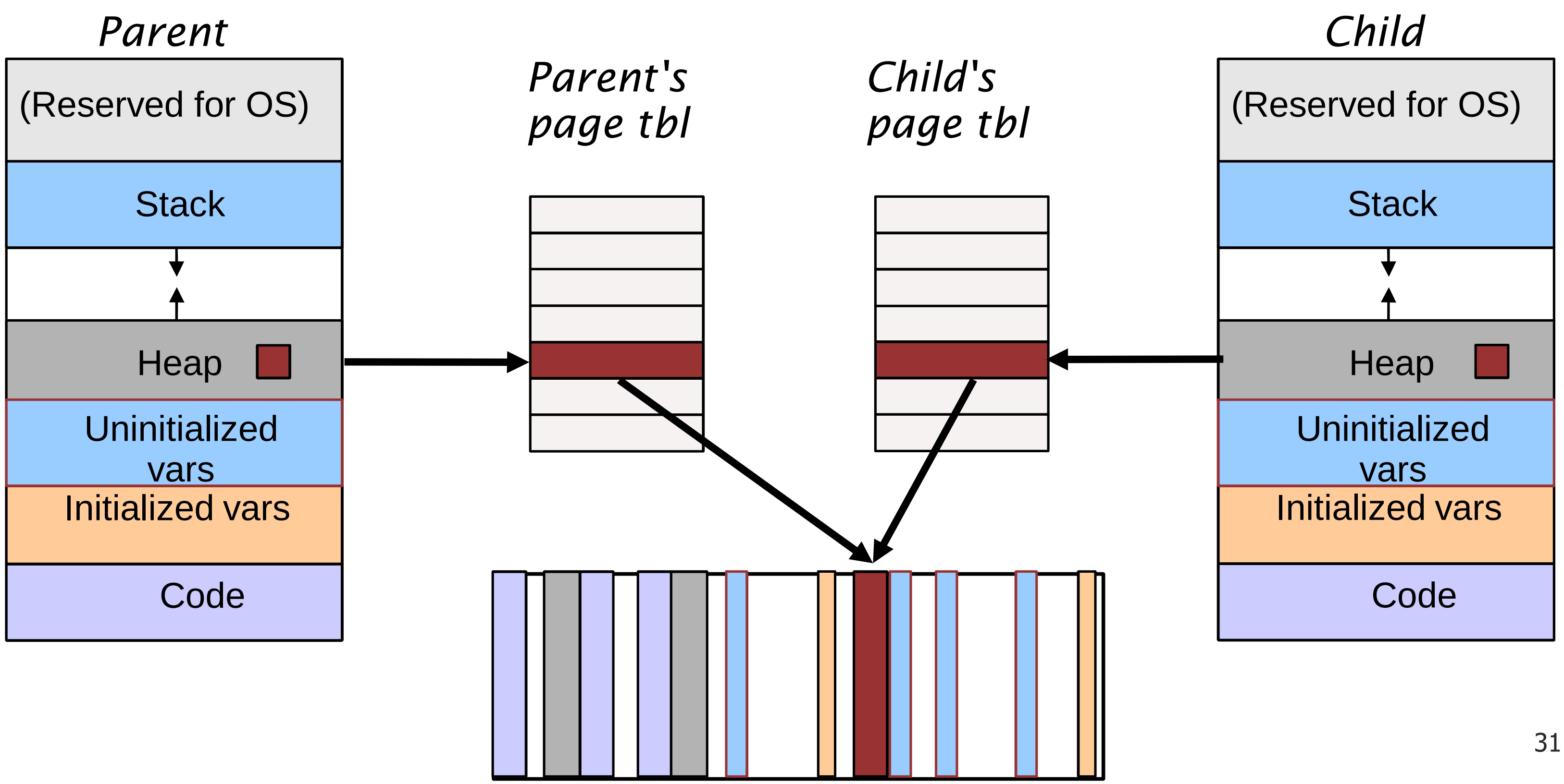
- Write to read only page (protection fault)
  - OS kills the program that made the illegal access
  - Some OSes make *zero page* inaccessible to trap use of NULL pointers
- Read/write to/from page not in memory
  - OS tries to make page available by paging in from the disk

Remember fork()?

- fork() creates an exact copy of a process
- When we fork a new process, does it make sense to make a copy of all of its memory?
  - Why or why not?
- What if the child process doesn't end up touching most of the memory the parent was using?
  - Extreme example: What happens if a process does an exec() immediately after fork()?

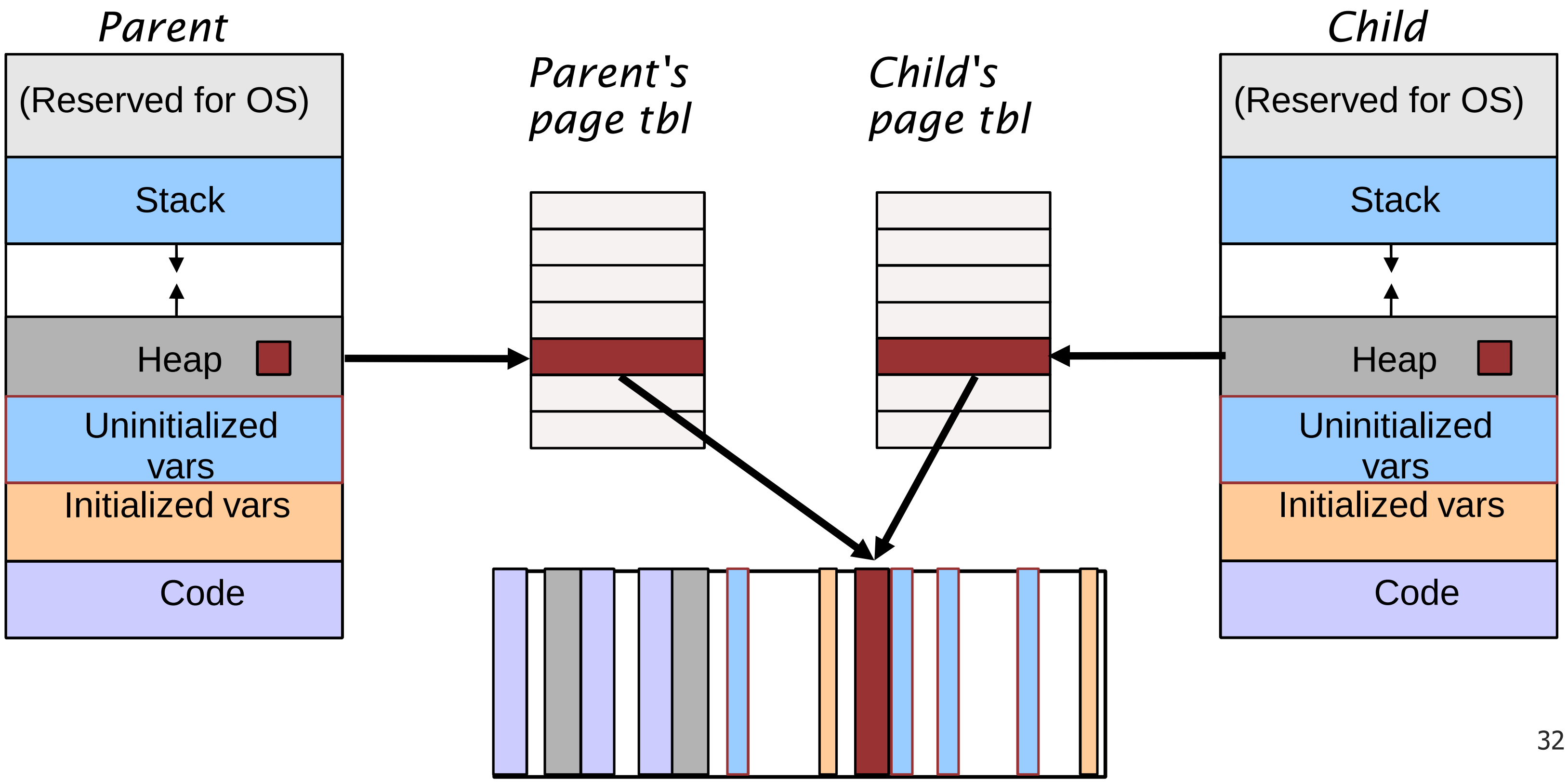
Copy-on-write

- Idea: Give the child process access to the same memory, but don't let it write to any of the pages directly!
  - 1) Parent forks a child process
  - 2) Child gets a copy of the parent's page tables
    - They point to the same physical frames!!!



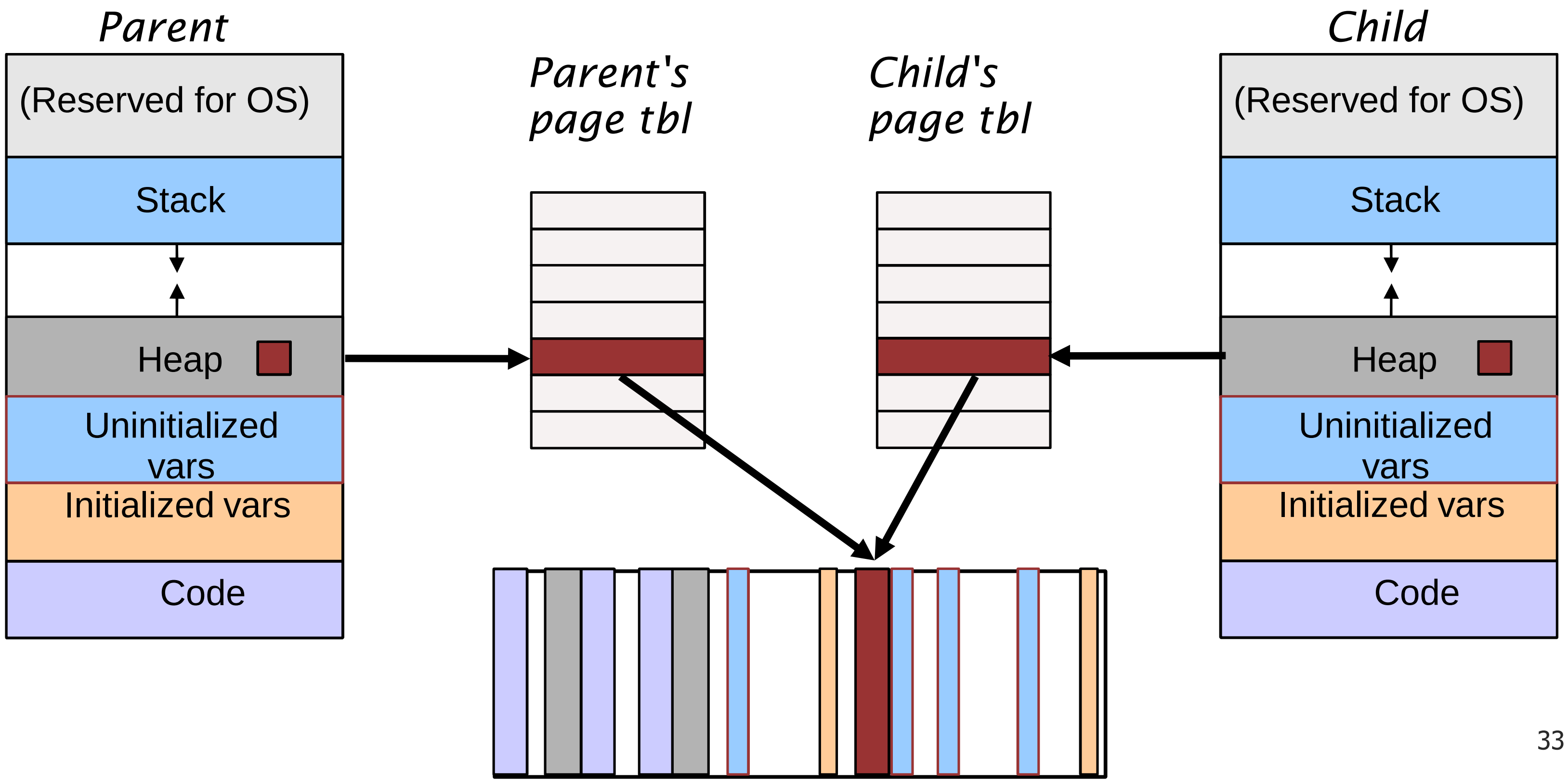
Copy-on-write

- All pages (both parent and child) marked read-only
  - **Why?**



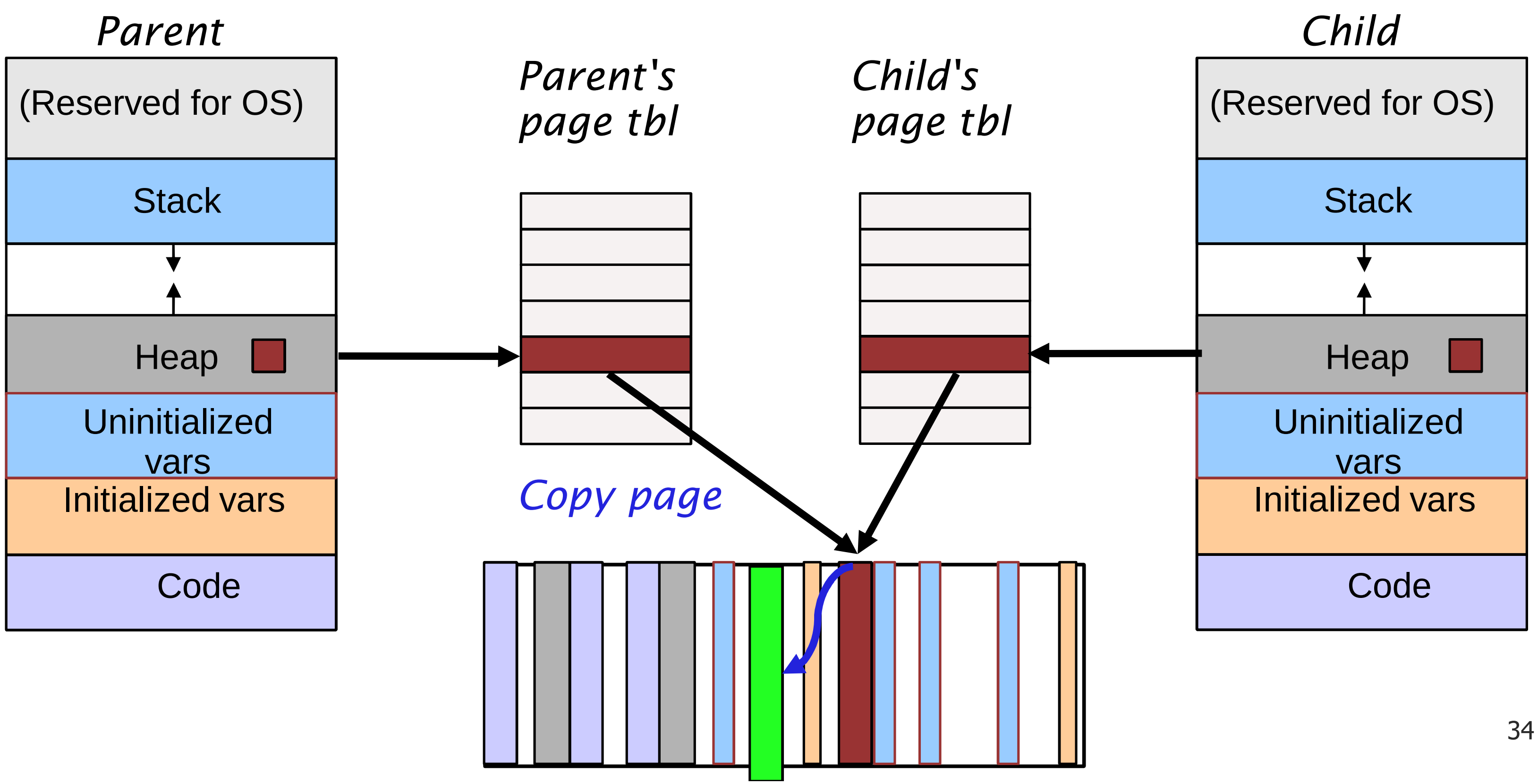
Copy-on-write

- What happens when the child *reads* the page?
  - Just accesses same memory as parent .... niiiiiice
- What happens when the child *writes* the page?
  - Protection fault occurs (page is read-only!)
  - OS copies the page and maps it R/W into the child's addr space



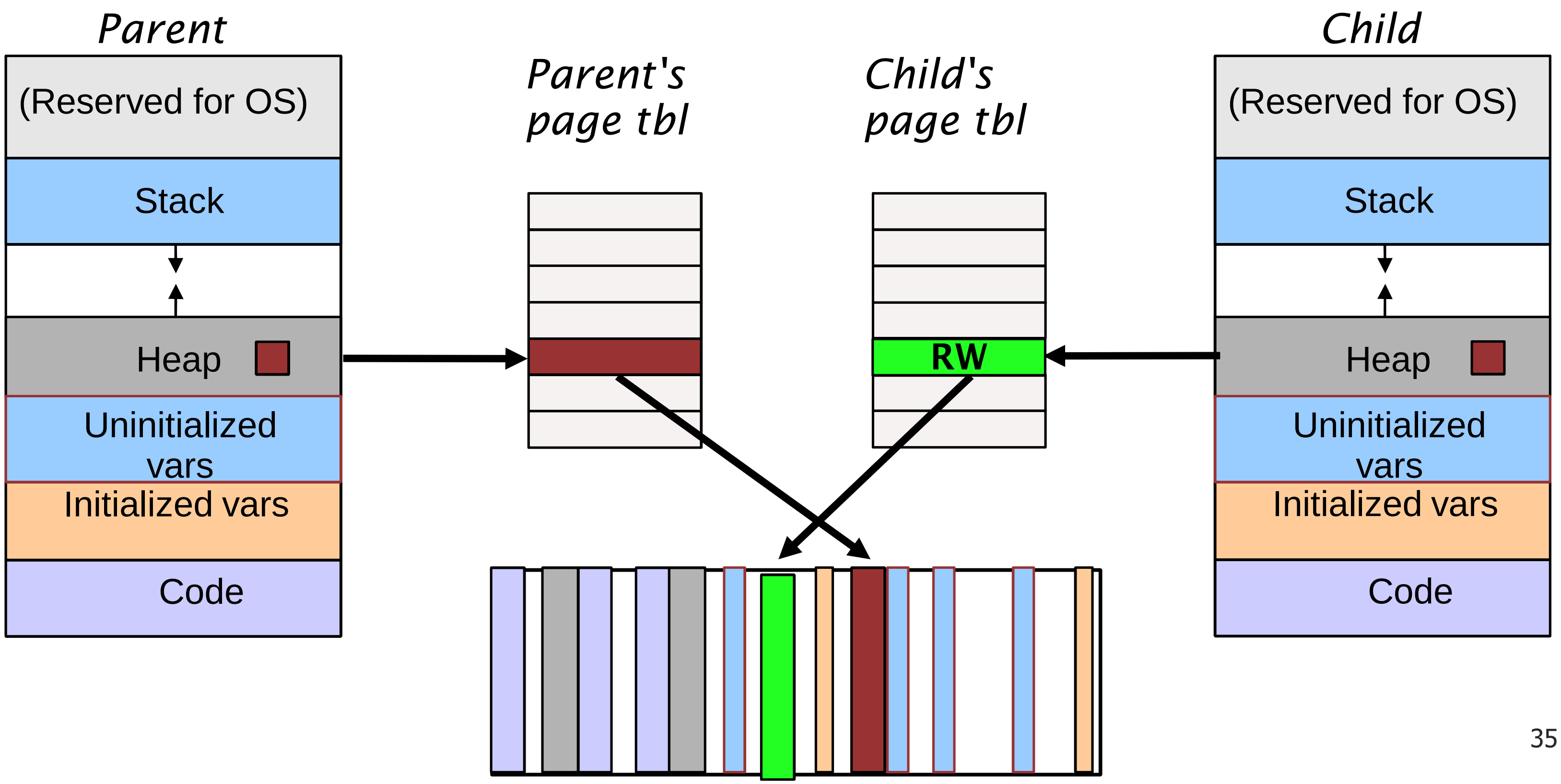
Copy-on-write

- What happens when the child *reads* the page?
  - Just accesses same memory as parent .... niiiiiice
- What happens when the child *writes* the page?
  - Protection fault occurs (page is read-only!)
  - OS copies the page and maps it R/W into the child's addr space



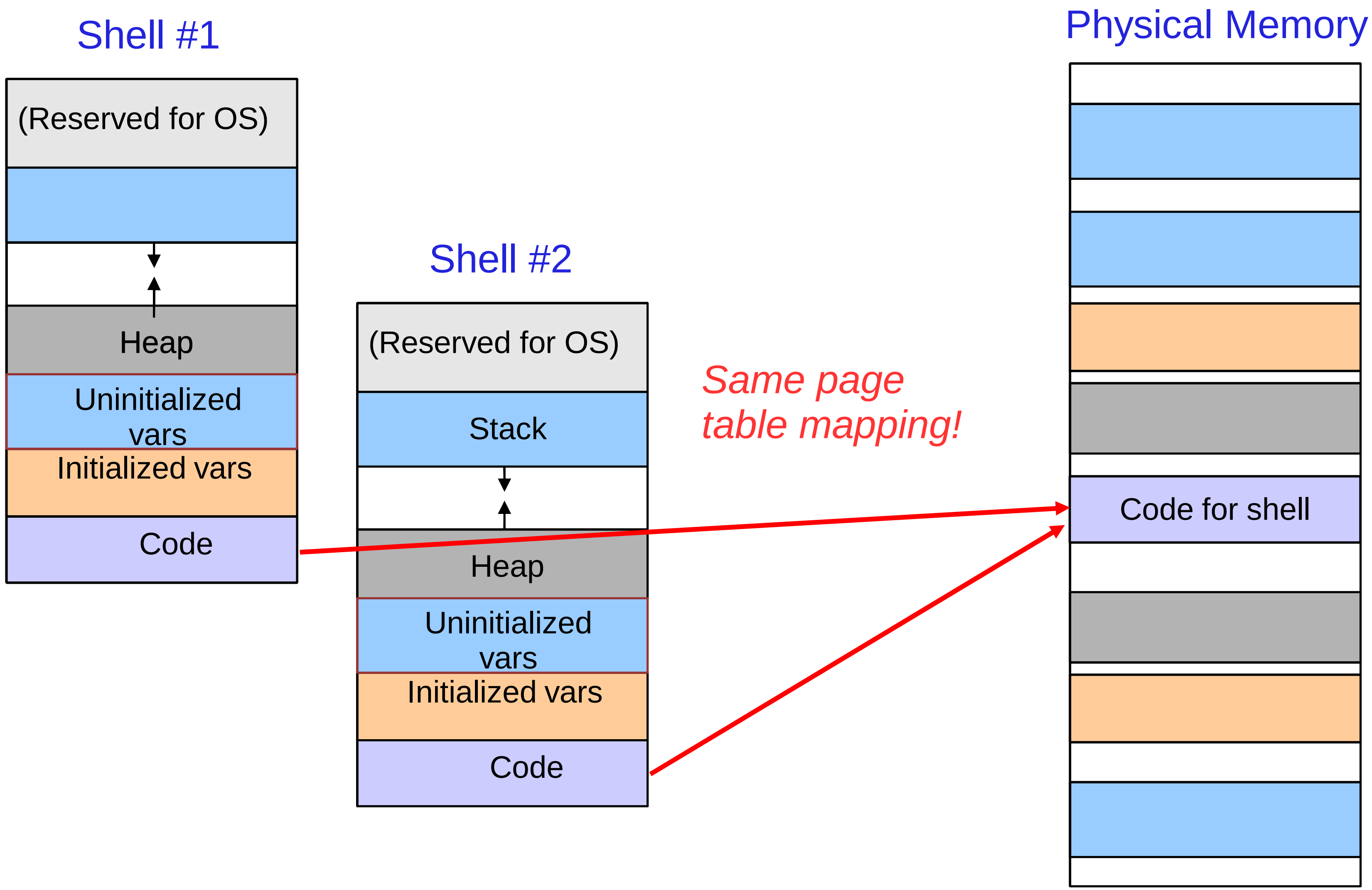
Copy-on-write

- What happens when the child *reads* the page?
  - Just accesses same memory as parent .... niiiiiice
- What happens when the child *writes* the page?
  - Protection fault occurs (page is read-only!)
  - OS copies the page and maps it R/W into the child's addr space



More Page Sharing Tricks

- Can also share code segment



More Page Sharing Tricks

- Can let different processes share read/write memory
  - UNIX supports shared memory through the shmget/shmat/shmdt system calls
  - Allocates a region of memory that is shared across multiple processes
  - Some of the benefits of multiple threads per process, but the rest of the process's address space is protected
- Memory-mapped files
  - Idea: Make a file on disk look like a block of memory
  - Works just like faulting in pages from executable files
  - In fact, many OS's use the same code for both
  - One wrinkle: Writes to the memory region must be reflected in the file
  - How does this work?
  - When writing to the page, mark the "modified" bit in the PTE
  - When page is removed from memory, write back to original file

Benefits of sharing pages

- How much memory savings do we get from sharing pages across identical processes?
  - A lot! Use the “top” command...

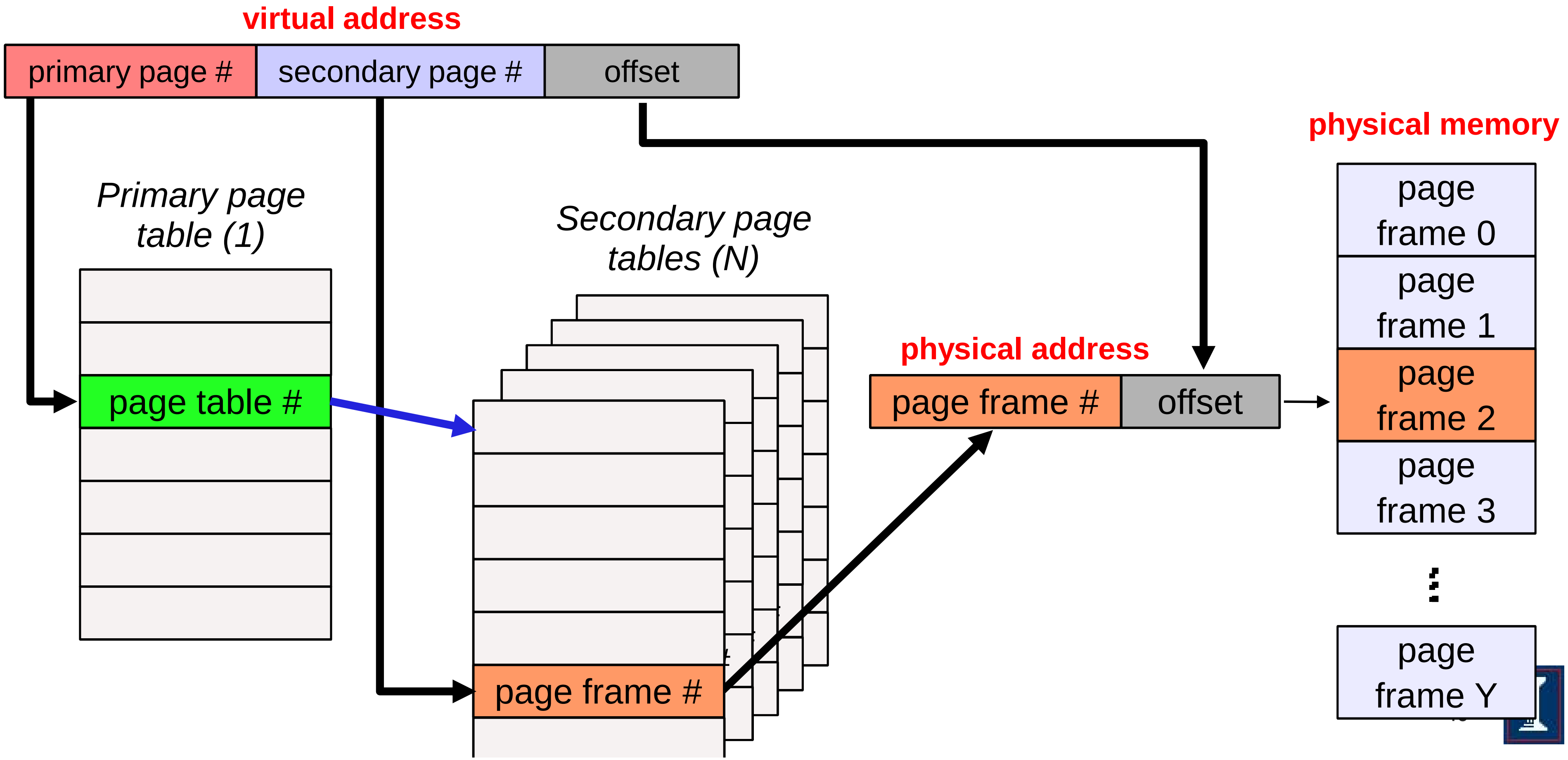
```
Terminal — top — 88x26

Processes:  68 total, 2 running, 1 stuck, 65 sleeping... 246 threads   13:17:30
Load Avg:  0.75, 0.58, 0.52     CPU usage:  7.7% user, 17.9% sys, 74.4% idle
SharedLibs: num =  223, resident = 33.3M code, 4.61M data, 4.80M LinkEdit
MemRegions: num = 17413, resident =  208M + 11.0M private,  546M shared
PhysMem:   618M wired,  261M active,  130M inactive, 1010M used, 13.9M free
VM: 9.79G +  150M   635052(61) pageins, 455424(0) pageouts

  PID COMMAND       %CPU   TIME    #TH #PRTS #MREGS RPRVT  RSHRD  RSIZE  VSIZE
 3784 Grab          5.0% 0:00.51   3   126   159  2.23M+ 7.25M+ 16.8M+  216M+
 3781 less          0.0% 0:00.02   1    13    17   148K   304K   484K  26.7M
 3778 sh            0.0% 0:00.00   1     8    16  88.0K   608K   364K  27.1M
 3777 sh            0.0% 0:00.00   1    13    16  68.0K   608K   544K  27.1M
 3776 man           0.0% 0:00.01   1    13    16   184K   264K   460K  26.7M
 3752 bash          0.0% 0:00.01   1    14    16   228K   696K   816K  27.1M
 3751 login         0.0% 0:00.01   1    16    40   172K   380K   636K  26.9M
 3748 top          12.8% 0:23.16   1    25    20   704K   300K  1.14M  27.0M
 3725 bash          0.0% 0:00.02   1    14    16   228K   696K   812K  27.1M
 3724 login         0.0% 0:00.01   1    16    40   172K   380K   636K  26.9M
 3722 Terminal      0.2% 0:02.31   6    92   140  2.25M  11.1M  10.3M   218M
 3719 WinAppHelp    0.0% 0:00.05   1    57    95   716K  4.10M  3.00M   198M
 3713 mdimport      0.0% 0:00.90   4    68   119  1.59M  3.16M  4.64M  57.8M
 3675 iTunes        3.5% 6:51.76   9   193   370  7.12M  12.1M+ 10.2M   263M
 3670 Address Bo    0.0% 0:02.58   1    92   179  2.21M  5.56M  15.2M   216M
 3659 Mail          0.0% 0:59.65   8   172   415  25.3M  10.9M+ 27.2M   258M
 3084 Skype         0.7% 17:20.32 18   240   452  23.9M  8.65M+ 20.0M   304M
  655 vfstool       0.0% 0:00.07   2    14    29   120K   308K   256K  32.1M
```

Page Table Sizes

- How big are the page tables for a process?
- Well ... we need one PTE per page.
- Say we have a 32-bit address space, and the page size is 4KB
- How many pages?
  - 2^32 == 4GB / 4KB per page == 1,048,576 (1 M pages)
- How big is each PTE?
  - Depends on the CPU architecture ... on the x86, it's 4 bytes.
- So, the total page table size is: 1 M pages * 4 bytes/PTE == 4 Mbytes
  - And that is *per process*
  - If we have 100 running processes, that's over 400 Mbytes of memory just for the page tables.
- Solution: Swap the page tables out to disk!

Multilevel Page Tables

- Main idea: Page the Page Tables
  - Allow portions of the page tables to be kept in memory at a time
  - Secondary page tables can be paged out to disk
  - Only (much smaller) primary page table needs to stay resident



Multilevel Page Tables

- With two levels of page tables, how big is each table?
  - Say we allocate 10 bits to the primary page, 10 bits to the secondary page, 12 bits to the page offset
  - Primary page table is then 2^10 * 4 bytes per PTE == 4 KB
  - Secondary page table is also 4 KB
  - Hey ... that's exactly the size of a page on most systems ... cool
- What happens on a page fault?
  - MMU looks up index in primary page table to get secondary page table
  - MMU tries to access secondary page table
    - May result in another page fault to load the secondary table!
  - MMU looks up index in secondary page table to get PFN
  - CPU can then access physical memory address
- Issues
  - Page translation has very high overhead
    - Up to three memory accesses plus two disk I/Os!!
  - TLB usage is clearly very important.

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?

Paging

- On heavily-loaded systems, memory can fill up
- Need to make room for newly-accessed pages
  - Heuristic: try to move "inactive" pages out to disk
    - What constitutes an "inactive" page?
- Paging
  - Refers to moving individual pages out to disk (and back)
  - We often use the terms "paging" and "swapping" interchangeably
  - Different from context switching
    - Background processes often have their pages remain resident in memory

Page Eviction

- When do we decide to evict a page from memory?
  - Usually, at the same time that we are trying to allocate a new physical page
  - However, the OS keeps a pool of “free pages” around, even when memory is tight, so that allocating a new page can be done quickly
  - The process of evicting pages to disk is then performed in the background

Basic Page Replacement

- How do we replace pages?
  - Find the location of the desired page on disk
  - 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
  - Read the desired page into the (newly) free frame. Update the page and frame tables.
  - Restart the process

Exploiting Locality

- Exploiting locality
  - Temporal locality: Memory accessed recently tends to be accessed again soon
  - Spatial locality: Memory locations near recently-accessed memory is likely to be referenced soon
- Locality helps to reduce the frequency of paging
  - Once something is in memory, it should be used many times
- This depends on many things:
  - The amount of locality and reference patterns in a program
  - The *page replacement policy*
  - The amount of physical memory and the *application footprint*

Evicting the Best Page

- Goal of the page replacement algorithm:
  - Reduce **page fault rate** by selecting the best page to evict
- The "best" pages are those that will never be used again
  - However, it's impossible to know in general whether a page will be touched
  - If you have information on future access patterns, it is possible to *prove* that evicting those pages that will be used the *furthest in the future* will *minimize* the page fault rate
- What is the best algorithm for deciding the order to evict pages?
  - Much attention has been paid to this problem.
  - Used to be a very hot research topic.
  - These days, widely considered solved (at least, solved well enough)

Algorithm: OPT (a.k.a. MIN)

- Evict page that won't be used for the longest time in the future
  - Of course, this requires that we can foresee the future...
  - So OPT cannot be implemented!
- This algorithm has the provably optimal performance
  - Hence the name “OPT”
- OPT is useful as a “yardstick” to compare the performance of other (implementable) algorithms against

The Optimal Page Replacement Algorithm

- Idea:
  - Select the page that will not be needed for the longest time in the future

Time		0	1	2	3	4	5	6	7	8	9	10
Requests			c	a	d	b	e	b	a	b	c	d
Page Frames	0	a	a	a	a	a						
	1	b	b	b	b	b						
	2	c	c	c	c	c						
	3	d	d	d	d	d						
Page faults							X					

The Optimal Page Replacement Algorithm

- Idea:
  - Select the page that will not be needed for the longest time in the future

Time		0	1	2	3	4	5	6	7	8	9	10
Requests			c	a	d	b	e	b	a	b	c	d
Page Frames	0	a	a	a	a	a	a	a	a	a	a	
	1	b	b	b	b	b	b	b	b	b	b	
	2	c	c	c	c	c	c	c	c	c	c	
	3	d	d	d	d	d	e	e	e	e	e	
Page faults							x					x

The Optimal Page Replacement Algorithm

- Idea:
  - Select the page that will not be needed for the longest time in the future

Time		0	1	2	3	4	5	6	7	8	9	10
Requests			c	a	d	b	e	b	a	b	c	d
Page Frames	0	a	a	a	a	a	a	a	a	a	a	a
	1	b	b	b	b	b	b	b	b	b	b	b
	2	c	c	c	c	c	c	c	c	c	c	c
	3	d	d	d	d	d	e	e	e	e	e	d
Page faults							x					x

Algorithms: Random and FIFO

- Random: Throw out a random page
  - Obviously not the best scheme
  - Although very easy to implement!
- FIFO: Throw out pages in the order that they were allocated
  - Maintain a list of allocated pages
  - When the length of the list grows to cover all of physical memory, pop first page off list and allocate it
- Why might FIFO be good?
- Why might FIFO not be so good?

Algorithms: Random and FIFO

- FIFO: Throw out pages in the order that they were allocated
  - Maintain a list of allocated pages
  - When the length of the list grows to cover all of physical memory, pop first page off list and allocate it
- Why might FIFO be good?
  - Maybe the page allocated very long ago isn't used anymore
- Why might FIFO not be so good?
  - Doesn't consider locality of reference!
  - Suffers from Belady's anomaly: Performance of an application might get *worse* as the size of physical memory *increases!!!*

Belady's Anomaly

time →

Access pattern: 0 1 2 3 0 1 4 0 1 2 3 4

Physical memory (3 page frames)

0	0	0	1	2	3	0	0	0	1	4	4
	1	1	2	3	0	1	1	1	4	2	2
		2	**3**	**0**	**1**	**4**	4	4	**2**	**3**	3

9 page faults!

time →

Access pattern: 0 1 2 3 0 1 4 0 1 2 3 4

Physical memory (4 page frames)

0	0	0	0	0	0	1	2	3	4	0	1
	1	1	1	1	1	2	3	4	0	1	2
		2	2	2	2	3	4	0	1	2	3
			3	3	3	**4**	**0**	**1**	**2**	**3**	**4**

10 page faults!

Algorithm: Least Recently Used (LRU)

- Evict the page that was used the longest time ago
  - Keep track of when pages are referenced to make a better decision
  - Use past behavior to predict future behavior
    - LRU uses past information, while OPT uses future information
  - When does LRU work well, and when does it not?
- Implementation
  - Every time a page is accessed, record a timestamp of the access time
  - When choosing a page to evict, scan over all pages and throw out page with oldest timestamp
- Problems with this implementation?

Algorithm: Least Recently Used (LRU)

- Evict the page that was used the longest time ago
  - Keep track of when pages are referenced to make a better decision
  - Use past behavior to predict future behavior
    - *LRU uses past information, while OPT uses future information*
  - When does LRU work well, and when does it not?
- Implementation
  - Every time a page is accessed, record a timestamp of the access time
  - When choosing a page to evict, scan over all pages and throw out page with oldest timestamp
- Problems with this implementation?
  - 32-bit timestamp would double size of PTE
  - Scanning all of the PTEs for lowest timestamp: slow

Least Recently Used (LRU)

- Keep track of when a page is used
- Replace the page that has been used least recently

Time		0	1	2	3	4	5	6	7	8	9	10
Requests			c	a	d	b	e	b	a	b	c	d
Page Frames	0	a										
	1	b										
	2	c										
	3	d										
Page faults												

Least Recently Used (LRU)

- Keep track of when a page is used
- Replace the page that has been used least recently (<u>farthest in the past</u>)

Time		0	1	2	3	4	5	6	7	8	9	10
Requests			c	a	d	b	e	b	a	b	c	d
Page Frames	0	a	a	a	a	a						
	1	b	b	b	b	b						
	2	c	c	c	c	c						
	3	d	d	d	d	d						
Page faults							X					

Least Recently Used (LRU)

- Keep track of when a page is used
- Replace the page that has been used least recently (farthest in the past)

Time		0	1	2	3	4	5	6	7	8	9	10
Requests			c	a	d	b	e	b	a	b	c	d
Page Frames	0	a	a	a	a	a	a	a	a	a		
	1	b	b	b	b	b	b	b	b	b		
	2	c	c	c	c	c	e	e	e	e		
	3	d	d	d	d	d	d	d	d	d		
Page faults							X				X	

Least Recently Used (LRU)

- Keep track of when a page is used
- Replace the page that has been used least recently (farthest in the past)

Time		0	1	2	3	4	5	6	7	8	9	10
Requests			c	a	d	b	e	b	a	b	c	d
Page Frames	0	a	a	a	a	a	a	a	a	a	a	
	1	b	b	b	b	b	b	b	b	b	b	
	2	c	c	c	c	c	e	e	e	e	e	
	3	d	d	d	d	d	d	d	d	d	c	
Page faults							X				X	X

Least Recently Used (LRU)

- Keep track of when a page is used
- Replace the page that has been used least recently (farthest in the past)

Time		0	1	2	3	4	5	6	7	8	9	10
Requests			c	a	d	b	e	b	a	b	c	d
Page Frames	0	a	a	a	a	a	a	a	a	a	a	a
	1	b	b	b	b	b	b	b	b	b	b	b
	2	c	c	c	c	c	e	e	e	e	e	d
	3	d	d	d	d	d	d	d	d	d	c	c
Page faults							X				X	X

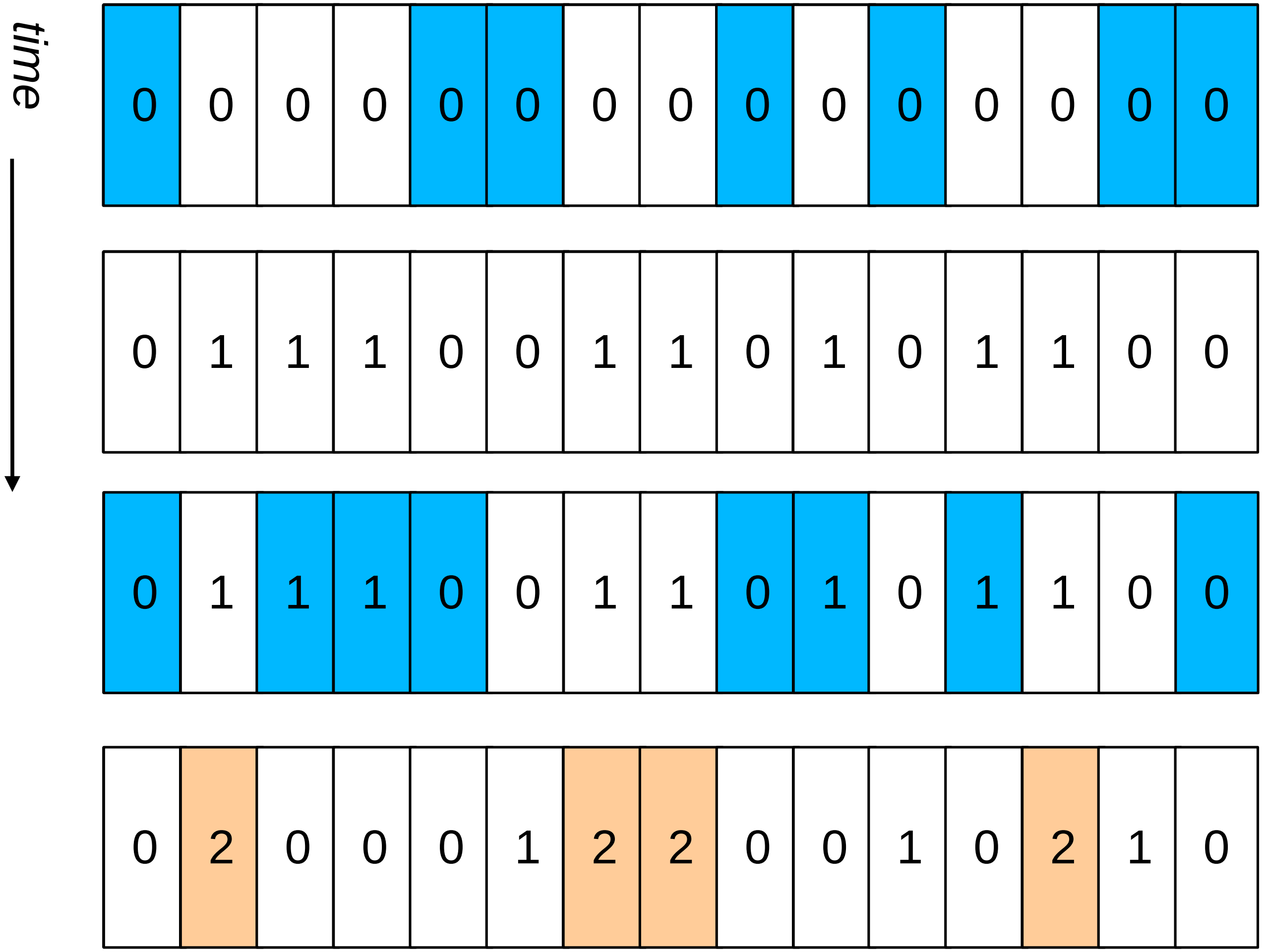
Least Recently Used Issues

- Not optimal
- Does not suffer from Belady's anomaly
- Implementation
  - Use time of last reference
    - Update every time page accessed (use system clock)
    - Page replacement - search for smallest time
  - Use a stack
    - On page access : remove from stack, push on top
    - Victim selection: select page at bottom of stack
- Both approaches require large processing overhead, more space, and hardware support.

Approximating LRU

- Use the PTE reference bit and a small counter per page
  - (Use a counter of, say, 2 or 3 bits in size, and store it in the PTE)
- Periodically (say every 100 msec), scan all physical pages in the system
  - If the page has not been accessed (PTE reference bit == 0), increment (or shift right) the counter
  - If the page has been accessed (reference bit == 1), set counter to zero (or shift right)
  - Clear the PTE reference bit in either case!
- Counter will contain the number of scans since the last reference
  to this page.
  - PTE that contains the highest counter value is the least recently used
  - So, evict the page with the highest counter

Approximate LRU Example



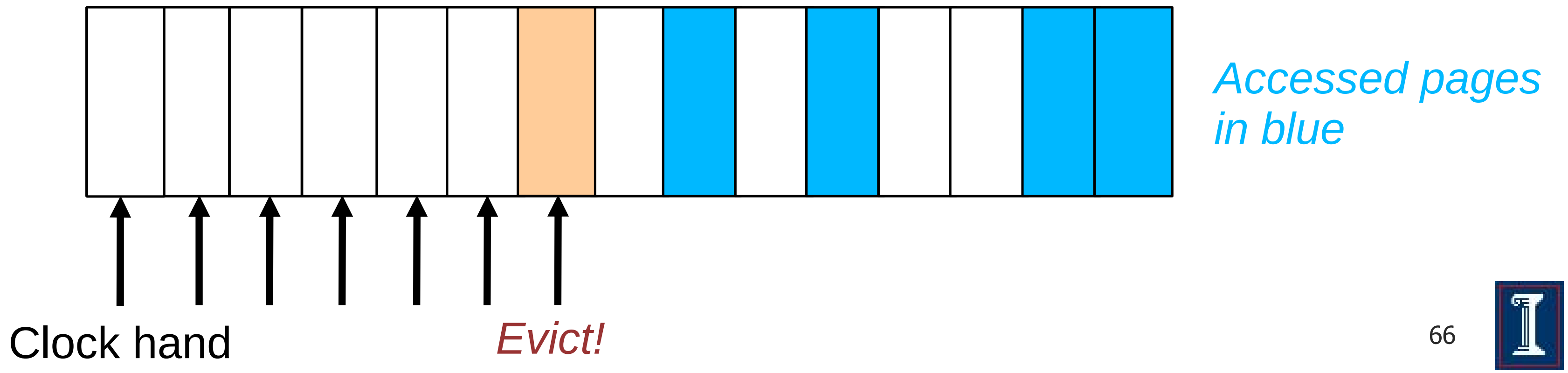
Accessed pages in blue

Increment counter for untouched pages

These pages have the highest counter value and can be evicted.

Algorithm: LRU Second-Chance (Clock)

- LRU requires searching for the page with the highest last-ref count
  - Can do this with a sorted list or a second pass to look for the highest value
- Simpler technique: Second-chance algorithm
  - "Clock hand" scans over all physical pages in the system
    - Clock hand loops around to beginning of memory when it gets to end
  - If PTE reference bit == 1, clear bit and advance hand to give it a second-chance
  - If PTE reference bit == 0, evict this page
    - No need for a counter in the PTE!



Algorithm: LRU Second-Chance (Clock)

- This is a lot like LRU, but operates in an iterative fashion
  - To find a page to evict, just start scanning from current clock hand position
  - What happens if all pages have ref bits set to 1?
  - What is the minimum “age” of a page that has the ref bit set to 0?
- Slight variant -- “nth chance clock”
  - Only evict page if hand has swept by N times
  - Increment per-page counter each time hand passes and ref bit is 0
  - Evict a page if counter >= N
  - Counter cleared to 0 each time page is used

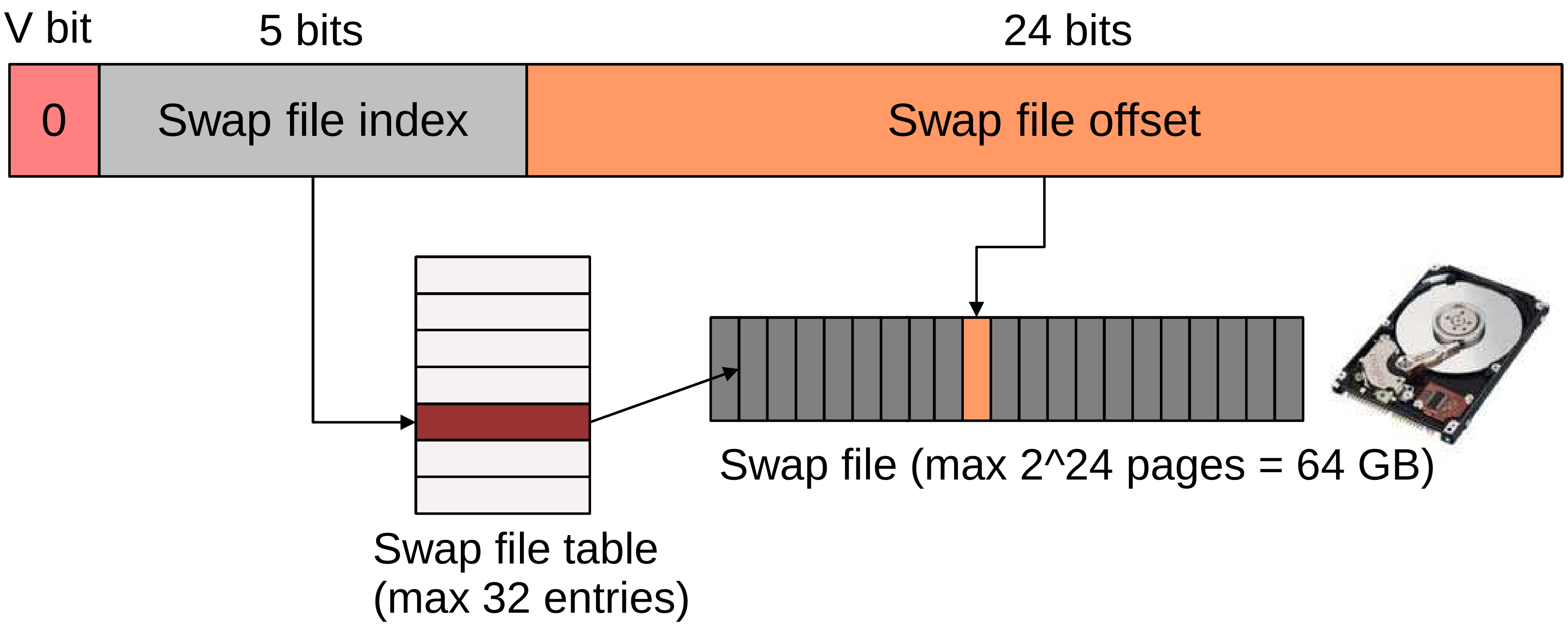
Swap Files

- What happens to the page that we choose to evict?
  - Depends on what kind of page it is and what state it's in!

- OS maintains one or more **swap files** or partitions on disk
  - Special data format for storing pages that have been swapped out

Swap Files

- How do we keep track of where things are on disk?
  - Recall PTE format
  - When V bit is 0, can recycle the PFN field to remember something about the page.



- But ... not all pages are swapped in from swap files!
  - E.g., what about executables?

Page Eviction

- How we evict a page depends on its type.
- Code page:
  - Just remove it from memory – can recover it from the executable file on disk!
- Unmodified (*clean*) data page:
  - If the page has previously been swapped to disk, just remove it from memory
    - Assuming that page's backing store on disk has not been overwritten
  - If the page has never been swapped to disk, allocate new swap space and write the page to it
  - Exception: unmodified zero page – no need to write out to swap at all!
- Modified (*dirty*) data page:
  - If the page has previously been swapped to disk, write page out to the swap space
  - If the page has never been swapped to disk, allocate new swap space and write the page to it

Physical Frame Allocation

- How do we allocate physical memory across multiple processes?
  - What if Process A needs to evict a page from Process B?
  - How do we ensure fairness?
  - How do we avoid having one process hogging the entire memory of the system?
- Local replacement algorithms
  - Per-process limit on the physical memory usage of each process
  - When a process reaches its limit, it evicts pages *from itself*
- Global-replacement algorithms
  - Physical size of processes can grow and shrink over time
  - Allow processes to evict pages from other processes
- Note that one process' paging can impact performance of entire system!
  - One process that does a lot of paging will induce more disk I/O

Working Set

- A process's *working set* is the set of pages that it currently “needs”
- Definition:
  - WS(P, t, w) = the set of pages that process P accessed in the time interval [t-w, t]
  - “w” is usually counted in terms of number of page references
    - A page is in WS if it was referenced in the last w page references
- Working set changes over the lifetime of the process
  - Periods of high locality exhibit **smaller** working set
  - Periods of low locality exhibit **larger** working set
- Basic idea: Give process enough memory for its working set
  - If WS is larger than physical memory allocated to process, it will tend to swap
  - If WS is smaller than memory allocated to process, it's wasteful
  - This amount of memory grows and shrinks over time

Estimating the Working Set

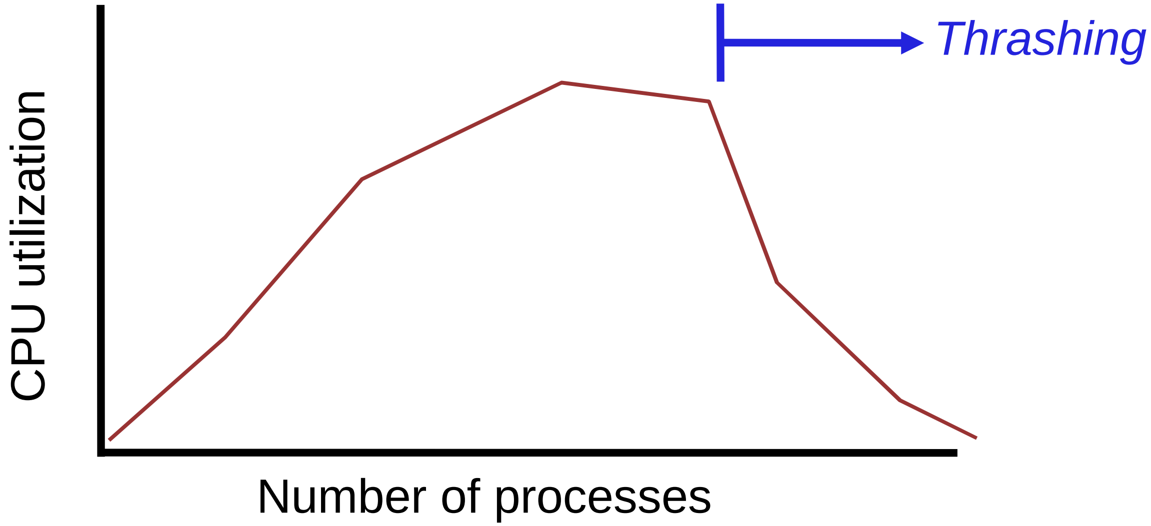
- How do we determine the working set?
- Simple approach: modified clock algorithm
  - Sweep the clock hand at fixed time intervals
  - Record how many seconds since last page reference
  - All pages referenced in last T seconds are in the working set
- Now that we know the working set, how do we allocate memory?
  - If working sets for all processes fit in physical memory, done!
  - Otherwise, reduce memory allocation of larger processes
    - Idea: Big processes will swap anyway, so let the small jobs run unencumbered
  - Very similar to shortest-job-first scheduling: give smaller processes better chance of fitting in memory
- How do we decide the working set time limit T?
  - If T is too large, very few processes will fit in memory
  - If T is too small, system will spend more time swapping
    - Which is better?

Page Fault Frequency

- Dynamically tune memory size of process based on # page faults
- Monitor page fault rate for each process (faults per sec)
- If page fault rate above threshold, give process more memory
  - Should cause process to fault less
  - Doesn't always work!
    - *Recall Belady's Anomaly*
- If page fault rate below threshold, reduce memory allocaton

Thrashing

- As system becomes more loaded, spends more of its time paging
  - Eventually, no useful work gets done!



- System is overcommitted!
  - If the system has too little memory, the page replacement algorithm doesn't matter
- Solutions?
  - Change scheduling priorities to “slow down” processes that are thrashing
  - Identify process that are hogging the system and kill them?
    - Is thrashing a problem on systems with only one user?

Allocation of Page Frames

- Scenario
  - Several physical pages allocated to processes A, B, and C. Process B page faults.
  - Which page should be replaced?
- Allocating memory across processes?
  - Does every process get the same fraction of memory?
  - Different fractions?
  - Should we completely swap some processes out of memory?

Allocation of Page Frames

- Each process needs minimum number of pages
  - Want to make sure that all processes that are loaded into memory can make forward progress
  - Example: IBM 370 – 6 pages to handle SS MOVE instruction:
    - Instruction is 6 bytes, might span 2 pages
    - 2 pages to handle from
    - 2 pages to handle to

Fixed Allocation

- Allocate a minimum number of frames per process
- Consider minimum requirements
  - One page from the current executed instruction
  - Most instructions require two operands
  - Include an extra page for paging out and one for paging in

Equal Allocation

- Allocate an equal number of frames per job
  - Example
    - 100 frames
    - 5 processes
    - Each process gets 20 frames
- Issues
  - But jobs use memory unequally
  - High priority jobs have same number of page frames and low priority jobs
  - Degree of multiprogramming might vary

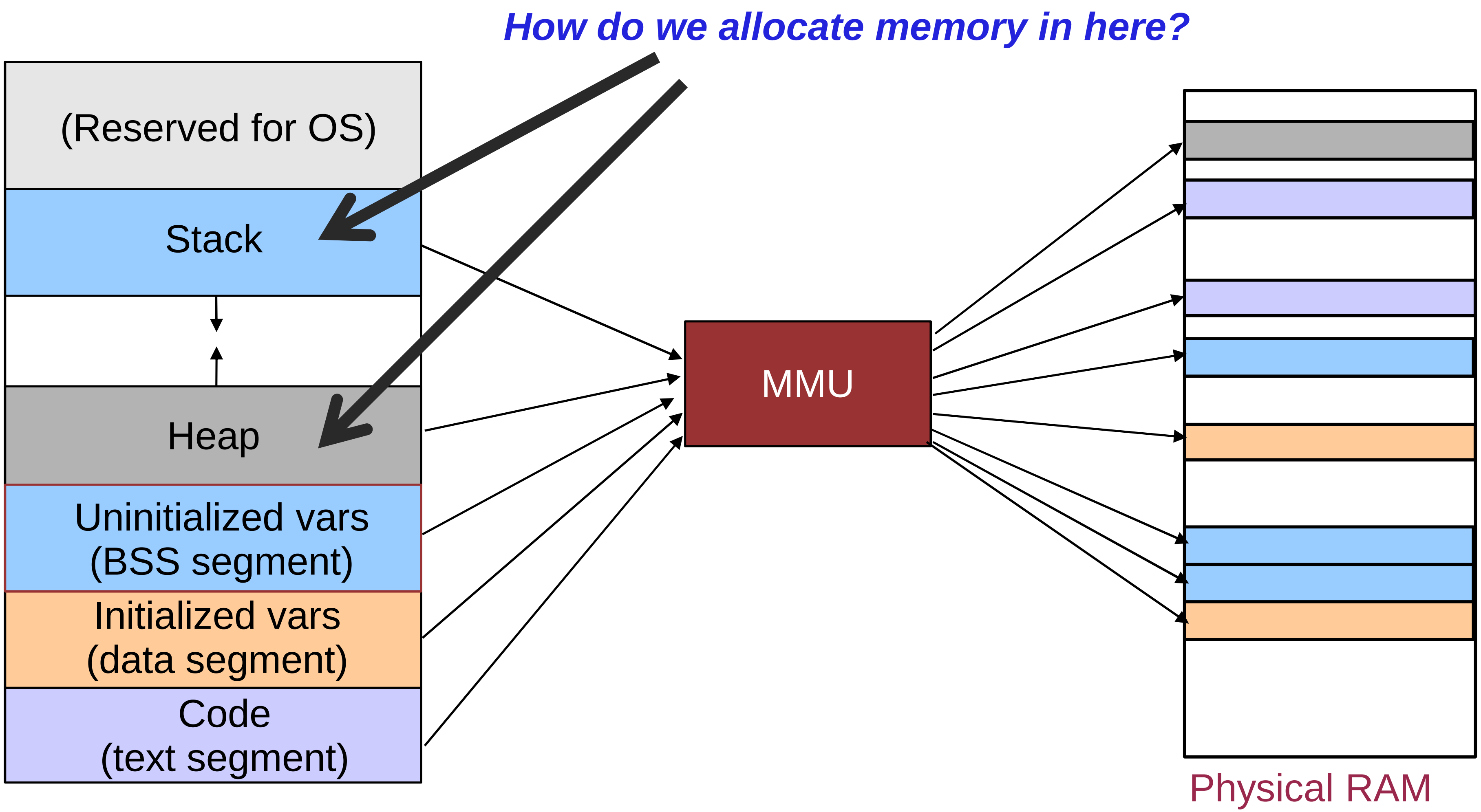
Proportional Allocation

- Allocate a number of frames per job proportional to job size
  - How do you determine job size
    - Run command parameters ?
    - Dynamically?
- Priority Allocation
  - May want to give high priority process more memory than low priority process
  - Use a proportional allocation scheme using priorities instead of size

Allocation of Page Frames

- Possible Replacement Scopes
  - Local replacement
    - Each process selects from only its own set of allocated frames
    - Process slowed down even if other less used pages of memory are available
  - Global replacement
    - Process selects replacement frame from set of all frames
    - One process can take a frame from another
    - Process may not be able to control its page fault rate.

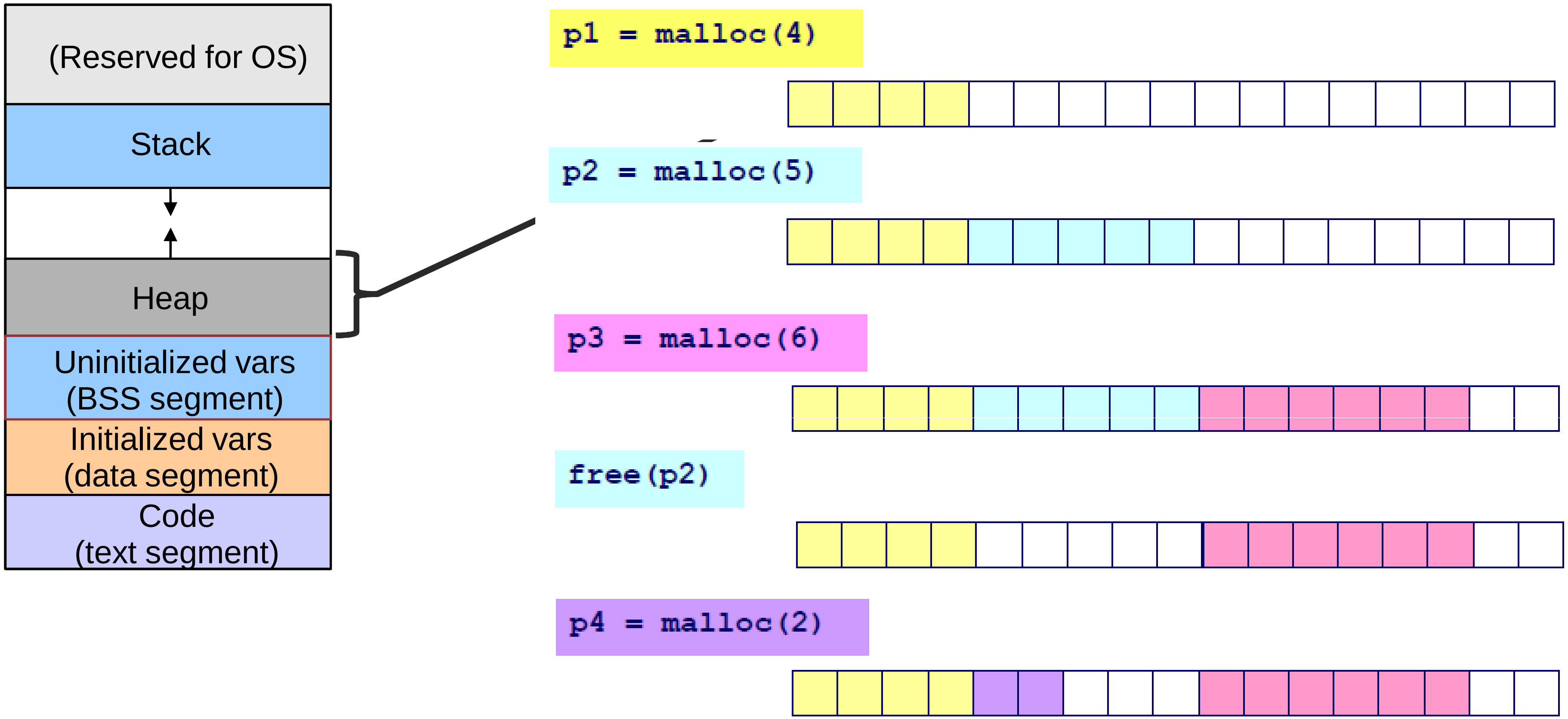
Is paging enough?



Memory allocation w/in a process

- Is paging enough?
- What happens when you declare a variable?
  - Allocating a page for every variable wouldn't be efficient
  - Allocations within a process are much smaller
  - Need to allocate on a finer granularity
- Solution (stack):
  - Function calls follow LIFO semantics
  - So we can use a stack data structure to represent the process's stack
- Solution (heap):
  - This is a much harder problem
  - Need to deal with fragmentation

Challenge of heap allocation



- Problem: program can issue arbitrary sequence of allocation and free requests
  - Can lead to external fragmentation

Challenges of heap allocation

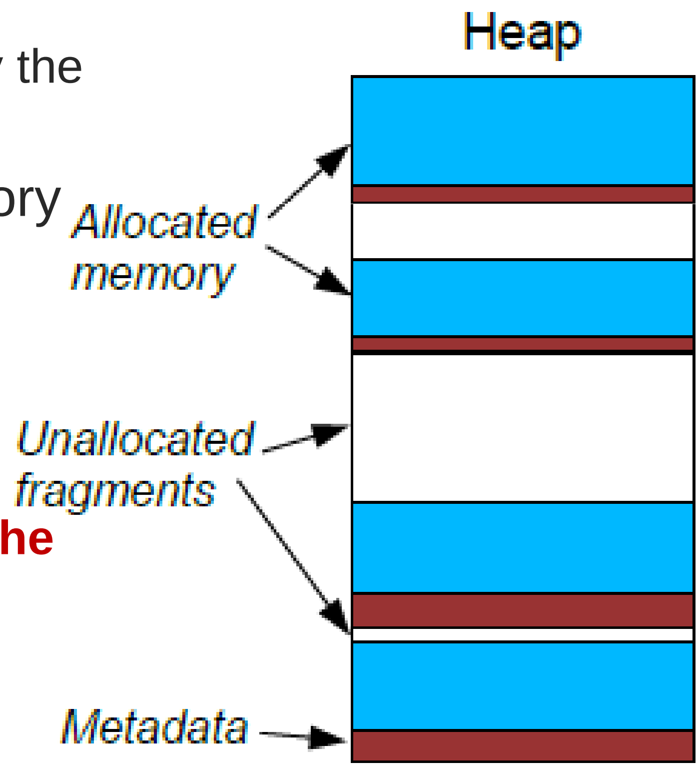
- Can’t control number or size of requested blocks
- Must respond immediately to all allocation requests
  - i.e., can’t reorder or buffer requests
- Must allocate blocks from free memory
  - i.e., can only place allocated blocks in free memory
- Must align blocks so they satisfy all alignment requirements
  - 8 byte alignment for GNU malloc (libc malloc) on Linux boxes
- Can only manipulate and modify free memory
- Can’t move the allocated blocks once they are allocated
  - i.e., compaction is not allowed

Performance Goals: Allocation overhead

- Want our memory allocator to be fast!
  - Minimize the overhead of both allocation and deallocation operations.
- One useful metric is throughput:
  - Given a series of allocate or free requests
  - Maximize the number of completed requests per unit time
- Example:
  - 5,000 malloc calls and 5,000 free calls in 10 seconds
  - Throughput is 1,000 operations/second.
- Note that a fast allocator may not be efficient in terms of memory
- utilization.
  - Faster allocators tend to be "sloppier"
  - To do the best job of space utilization, operations must take more time.
  - Trick is to balance these two conflicting goals.

Performance Goals: Memory Utilization

- Allocators rarely do a perfect job of managing memory.
  - Usually there is some “waste” involved in the process.
- Examples of waste...
  - Extra metadata or internal structures used by the allocator itself
- (example: Keeping track of where free memory is located)
  - Chunks of heap memory that are unallocated (**fragments**)
- We define **memory utilization** as...
  - The **total amount of memory allocated to the application** divided by the total **heap size**
- Ideally, we'd like utilization to be to 100%
  - In practice this is not possible, but would be good to get close.



Conflicting performance goals

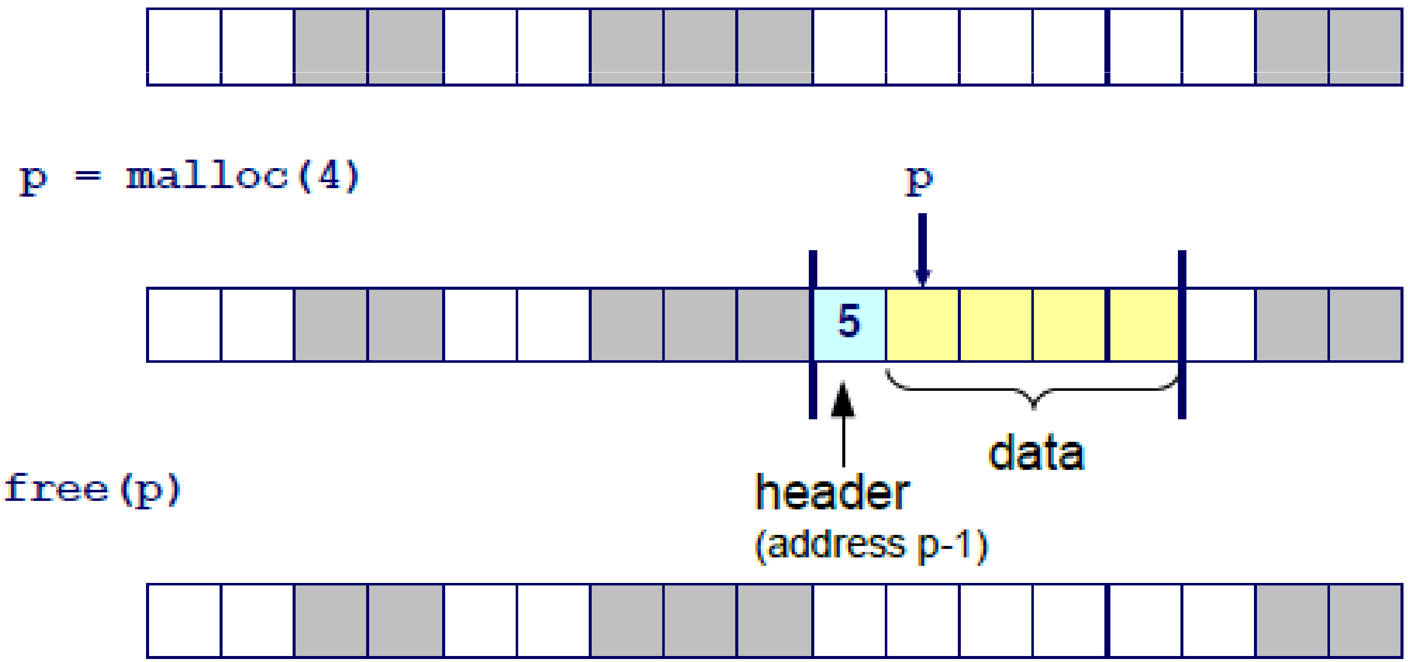
- Note that good throughput and good utilization are difficult to
- achieve simultaneously.
- A fast allocator may not be efficient in terms of memory utilization.
  - Faster allocators tend to be “sloppier” with their memory usage.
- Likewise, a space-efficient allocator may not be very fast
  - To keep track of memory waste (i.e., tracking fragments), the allocation operations generally take longer to run.
- Trick is to balance these two conflicting goals.

Implementation Issues

- How do we know how much memory to free just given a pointer?
- How do we keep track of the free blocks?
- What do we do with the extra space when allocating a memory block that is smaller than the free block it is placed in?
- How do we pick which free block to use for allocation?

Knowing how much to free

- Standard method
  - Keep the length of the block in the header preceding the block
  - Requires an extra word for every allocated block

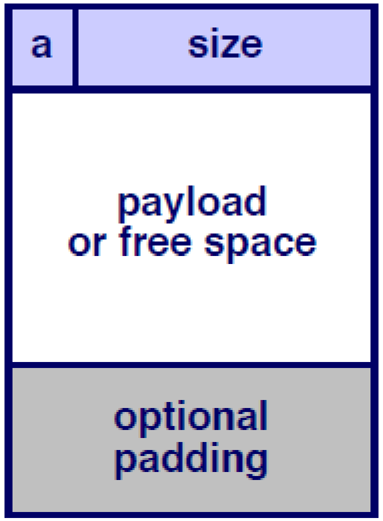


Keeping Track of Free Blocks

- One of the biggest jobs of an allocator is knowing where the free memory is.
- The allocator's approach to this problem affects...
  - Throughput – time to complete a malloc() or free()
  - Space utilization – amount of extra metadata used to track location of free memory.
- There are many approaches to free space management.
  - Next, we will talk about one: **Implicit free lists.**

Implicit Free List

- Idea: Each block contains a header with some extra information.
- Allocated bit indicates whether block is allocated or free.
- Size field indicates entire size of block (including the header)
- Trick: Allocation bit is just the high-order bit of the size word
- For this lecture, let's assume the header size is 1 byte.
- Makes the pictures that I'll show later on easier to understand.
- This means the block size is only 7 bits, so max. block size is 127 bytes (2^7-1).
- Clearly a real implementation would want to use a larger header (e.g., 4 bytes).

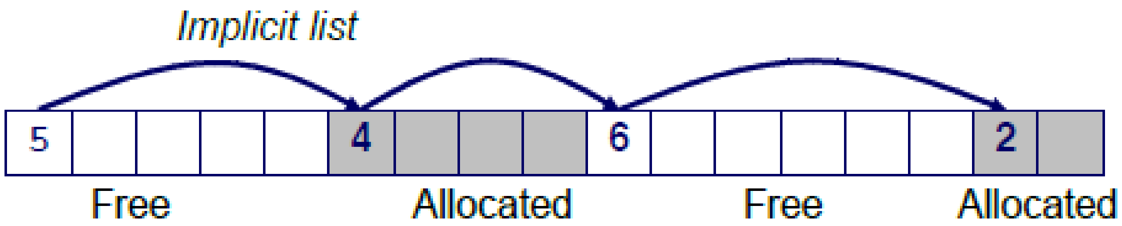


a = 1: block is allocated
a = 0: block is free

size: block size

payload: application data

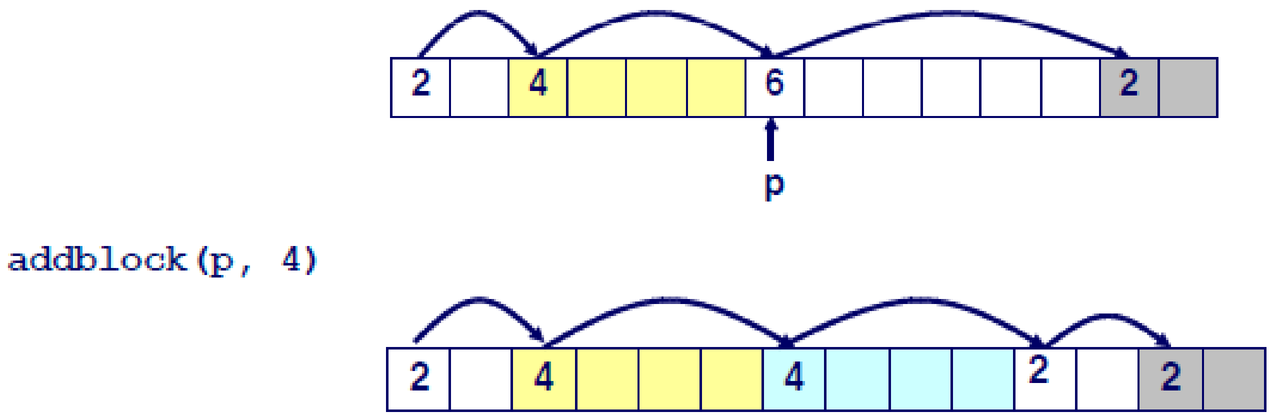
Implicit Free List



- No explicit structure tracking location of free/allocated blocks.
  - Rather, the size word (and allocated bit) in each block form an implicit "block list"
- How do we find a free block in the heap?
- Start scanning from the beginning of the heap.
- Traverse each block until (a) we find a free block and (b) the block is large enough to handle the request.
- This is called the first fit strategy.
  - Could also use next fit, best fit, etc

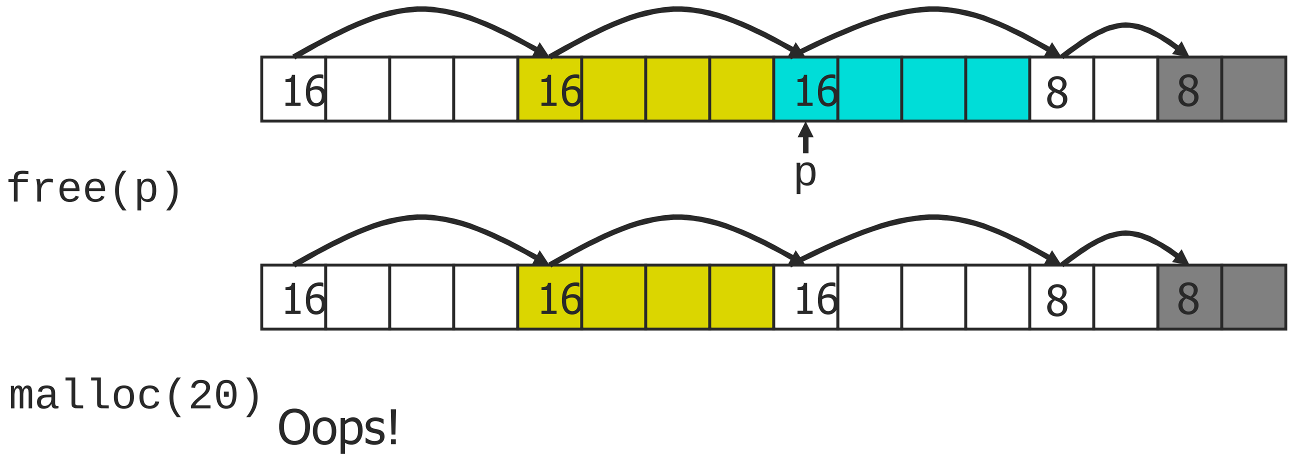
Implicit list: Allocating a Block

- Splitting free blocks
  - Since allocated space might be smaller than free space, we may need to split the free block that we're allocating within



Implicit List: Freeing a Block

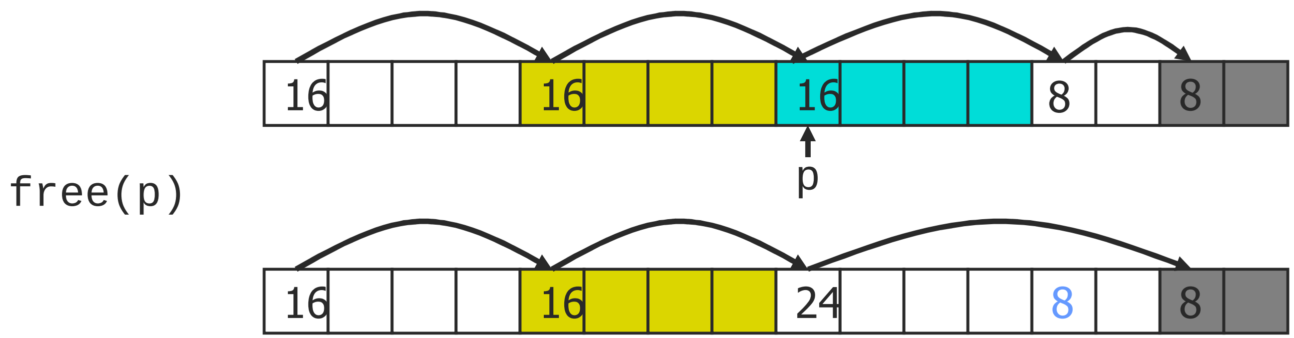
- Simplest implementation:
  - Only need to clear allocated flag
  - void free_block(ptr p) { *p = *p & ~1}
- But can lead to "false fragmentation"



- There's enough free space, but allocator won't find it!

Implicit List: Coalescing

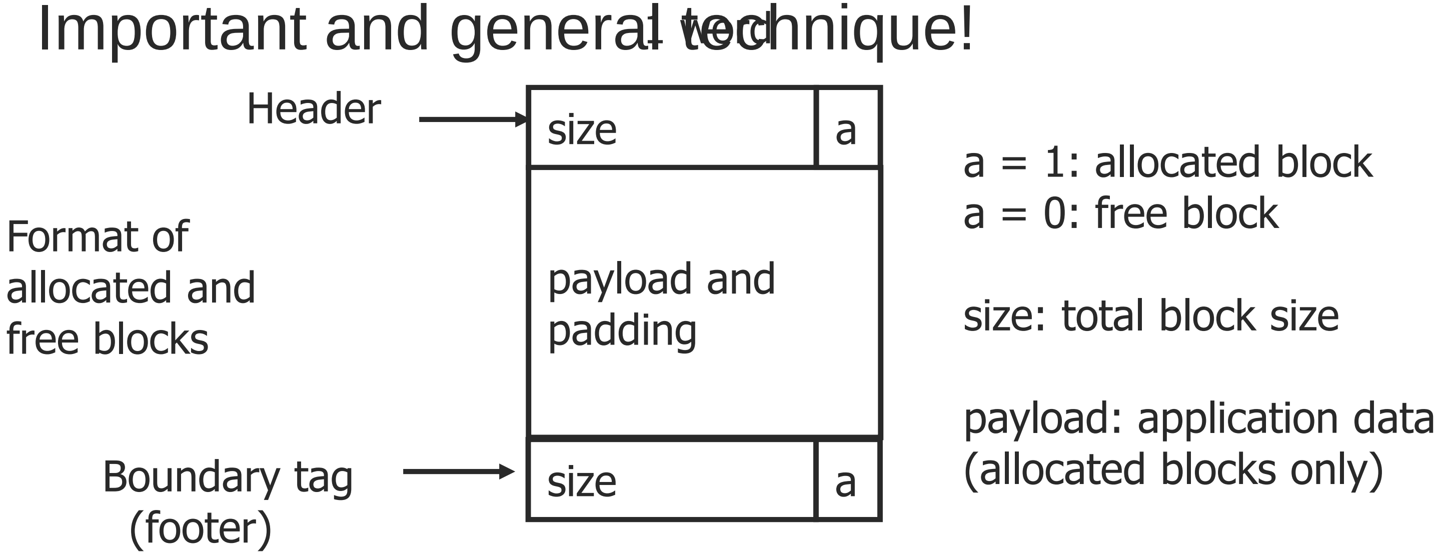
- Join (coalesce) with next and previous block if they are free
  - Coalescing with next block

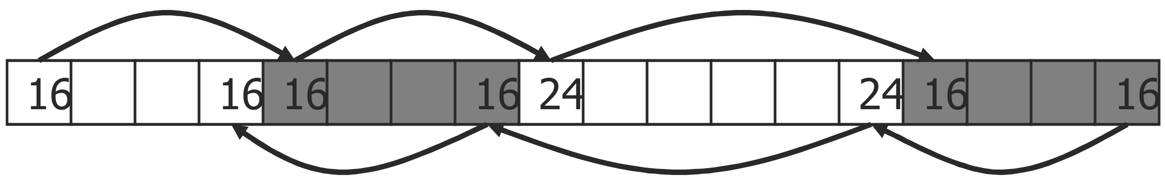


- But how do we coalesce with previous block?

Implicit List: Bidirectional Coalescing

- Boundary tags [Knuth73]
  - Replicate size/allocated word at tail end of all blocks
  - Allows us to traverse "list" backwards, but requires extra space
  - Important and general technique!





Implicit Lists: Summary

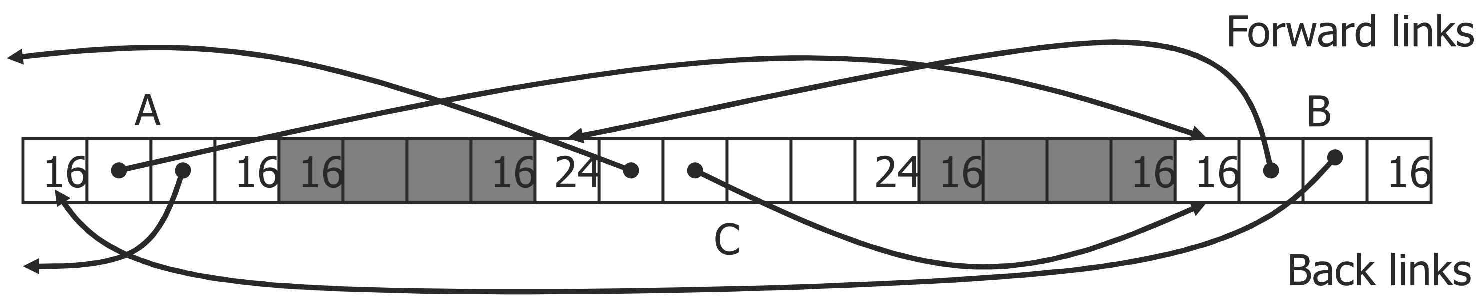
- Implementation: very simple
- Allocate: linear-time worst case
- Free: constant-time worst case—even with coalescing
- Memory usage: will depend on placement policy
  - First, next, or best fit

- Not used in practice for malloc/free because of linear-time allocate, but used in some special-purpose applications

- However, concepts of splitting and boundary tag coalescing are general to *all* allocators

Alternative: Explicit Free Lists

- Use data space for link pointers
  - Typically doubly linked
  - Still need boundary tags for coalescing



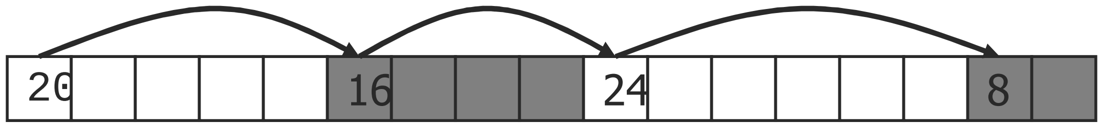
- Links aren't necessarily in same order as blocks!

Freeing with Explicit Free Lists

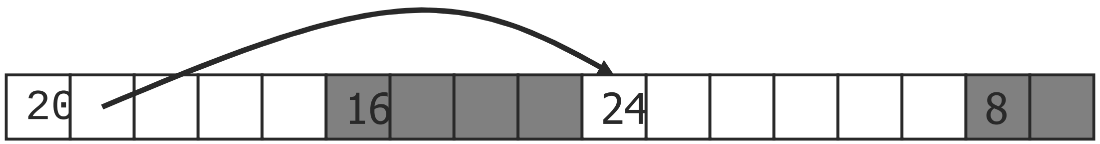
- *Insertion policy*: Where in free list to put newly freed block?
  - LIFO (last-in-first-out) policy
    - Insert freed block at beginning of free list
    - Pro: simple, and constant-time
    - Con: studies suggest fragmentation is worse than address-ordered
  - Address-ordered policy
    - Insert freed blocks so list is always in address order
      - i.e. addr(pred) < addr(curr) < addr(succ)
    - Con: requires search (using boundary tags)
    - Pro: studies suggest fragmentation is better than LIFO

Keeping Track of Free Blocks

- *Method 1*: Implicit list using lengths -- links all blocks



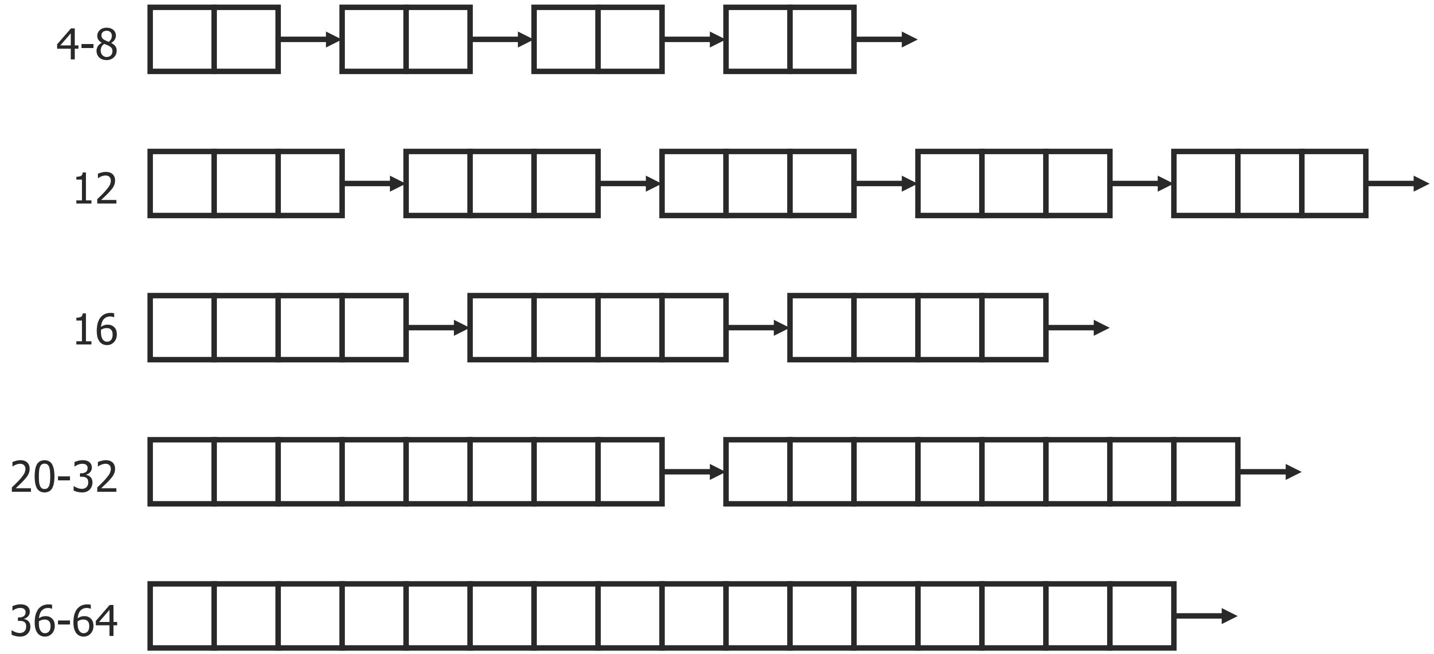
- *Method 2*: Explicit list among the free blocks using pointers within the free blocks



- *Method 3*: Segregated free list
  - Different free lists for different size classes
  - We'll talk about this one next

Segregated Storage

- Each *size class* has its own collection of blocks



  - Often separate size class for every small size (8, 12, 16, …)
  - For larger, typically have size class for each power of 2

Buddy Allocators

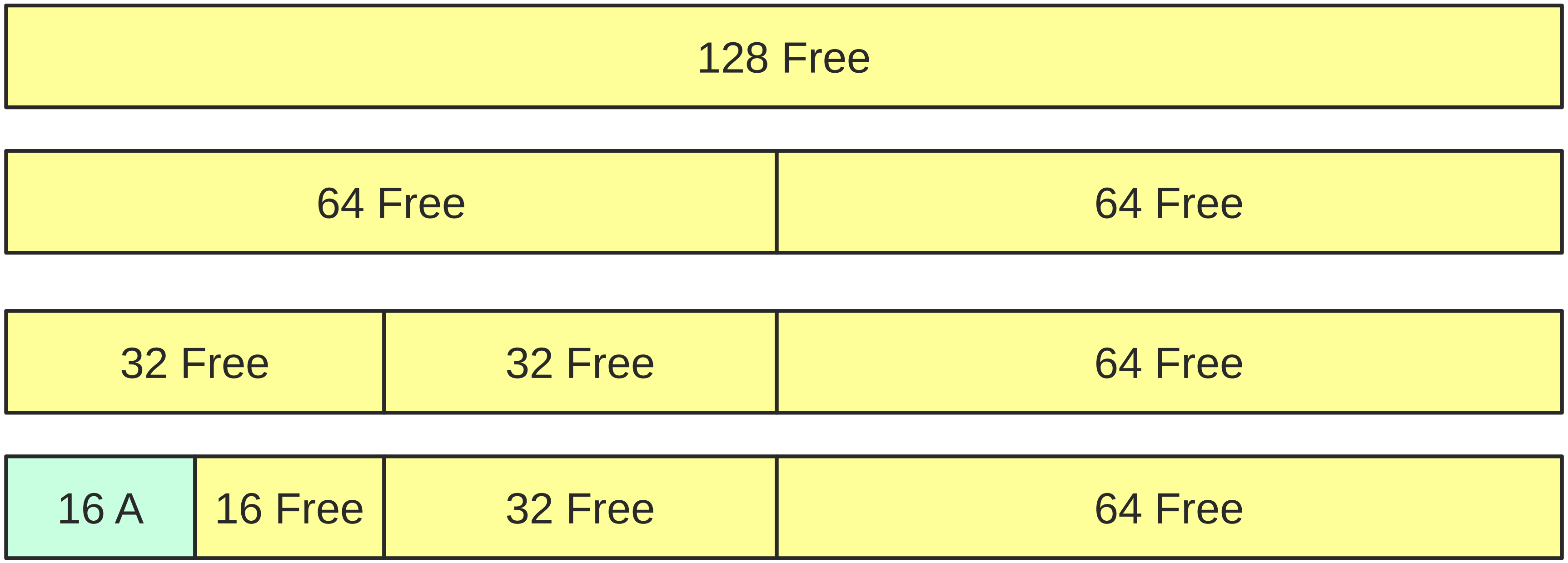
- Special case of segregated fits
- Basic idea:
  - Limited to power-of-two sizes
  - Can only coalesce with "buddy", who is other half of next-higher power of two
- Clever use of low address bits to find buddies
- Problem: large powers of two result in large internal fragmentation (e.g., what if you want to allocate 65537 bytes?)

Buddy System Example

128 Free

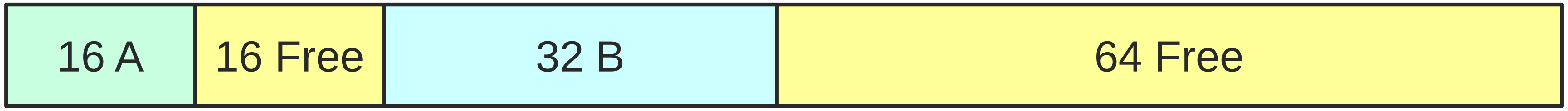
Buddy System Example

Process A requests 16



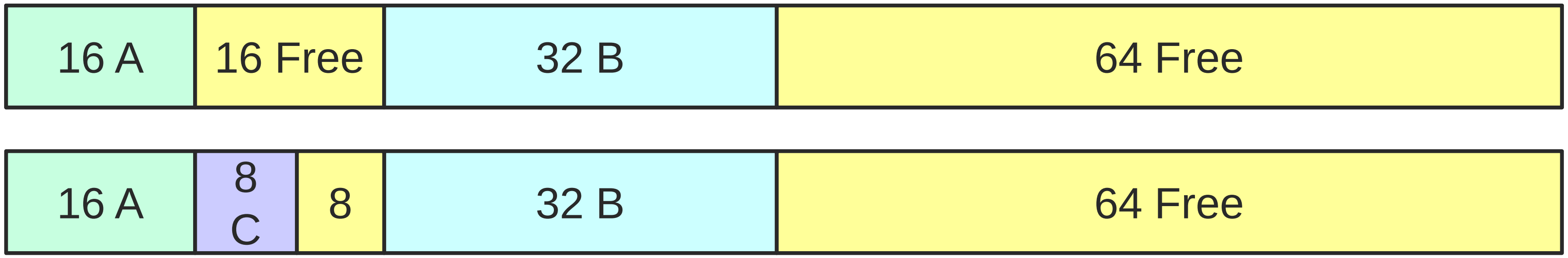
Buddy System Example

Process B requests 32



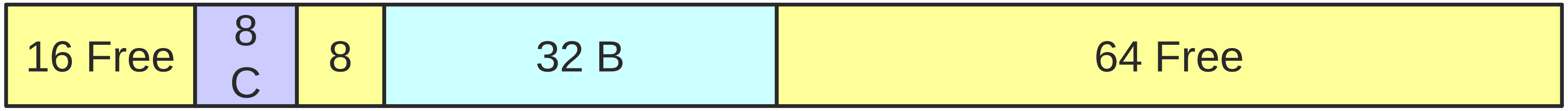
Buddy System Example

Process C requests 8



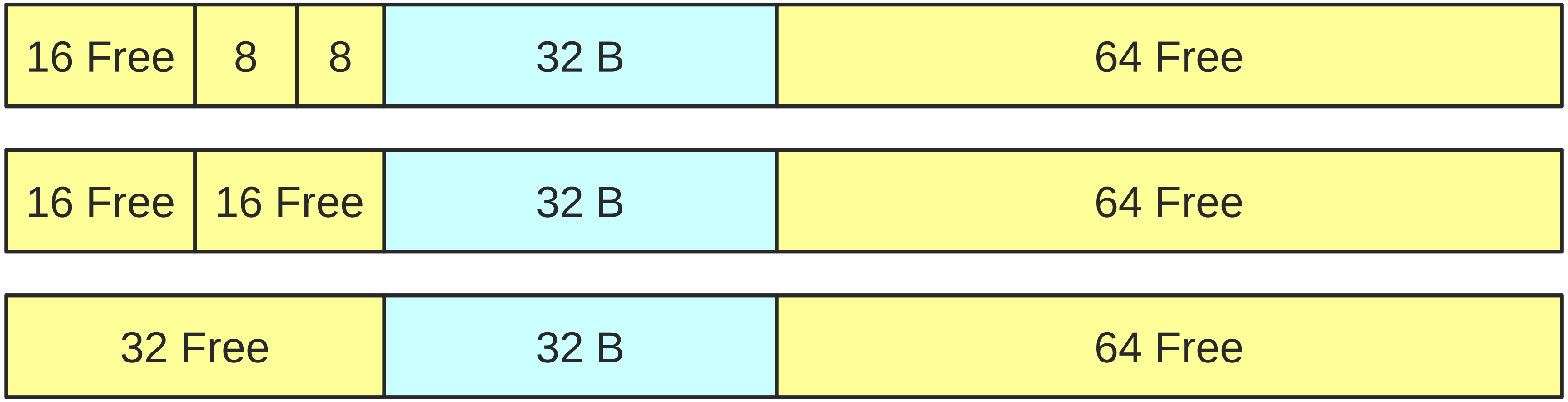
Buddy System Example

Process A exits



Buddy System Example

Process C exits



- Advantage
  - Minimizes external fragmentation
- Disadvantage
  - Internal fragmentation when not 2^n request