CS61: Systems Programming and Machine Organization
Harvard University, Fall 2009

# Lecture 15:
# Virtual Memory

Prof. Matt Welsh

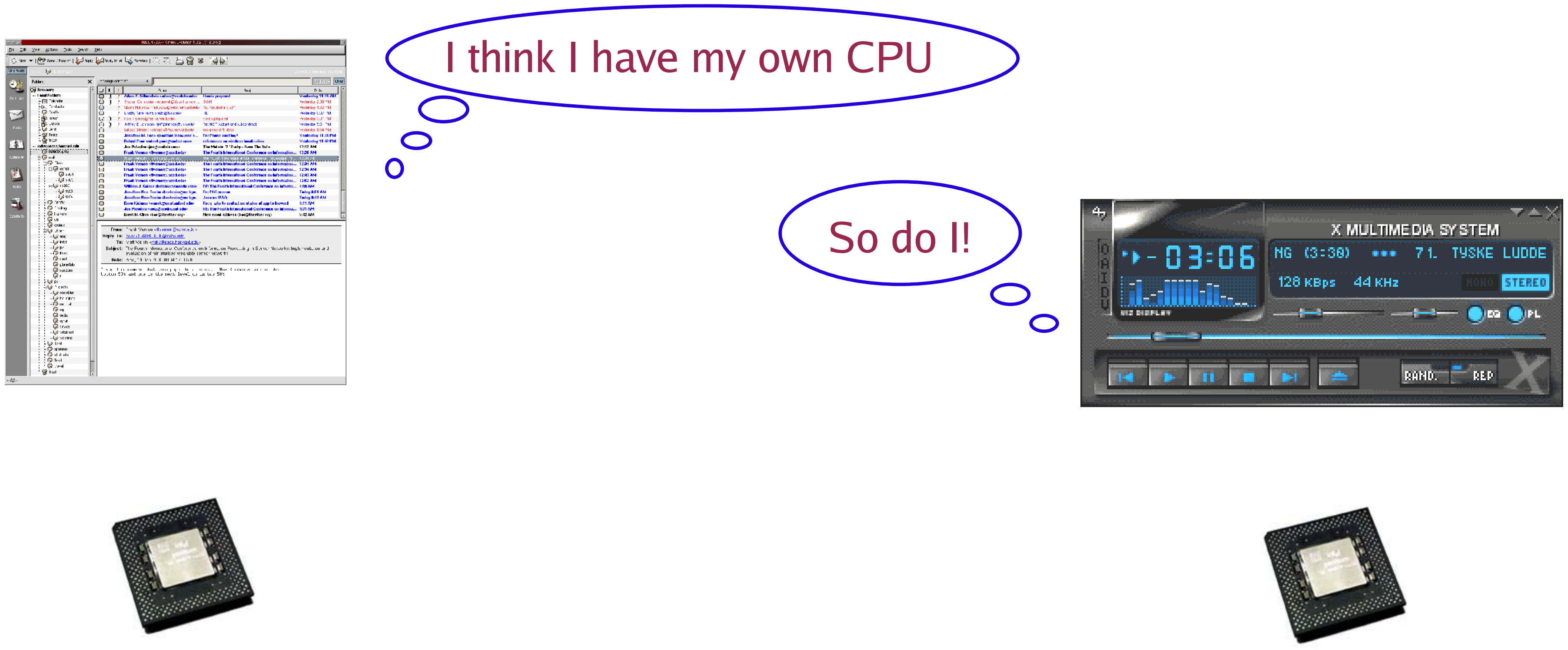
October 22, 2009

## Topics for today

- How can multiple programs run simultaneously on one machine?
- Virtual memory: Allowing multiple programs to share physical RAM.
- Some benefits of virtual memory.
- Some approaches to implementing VM: Partitions and Paging.
- The Memory Management Unit (MMU) and Translation Lookaside Buffer (TLB).

# Running multiple programs at once

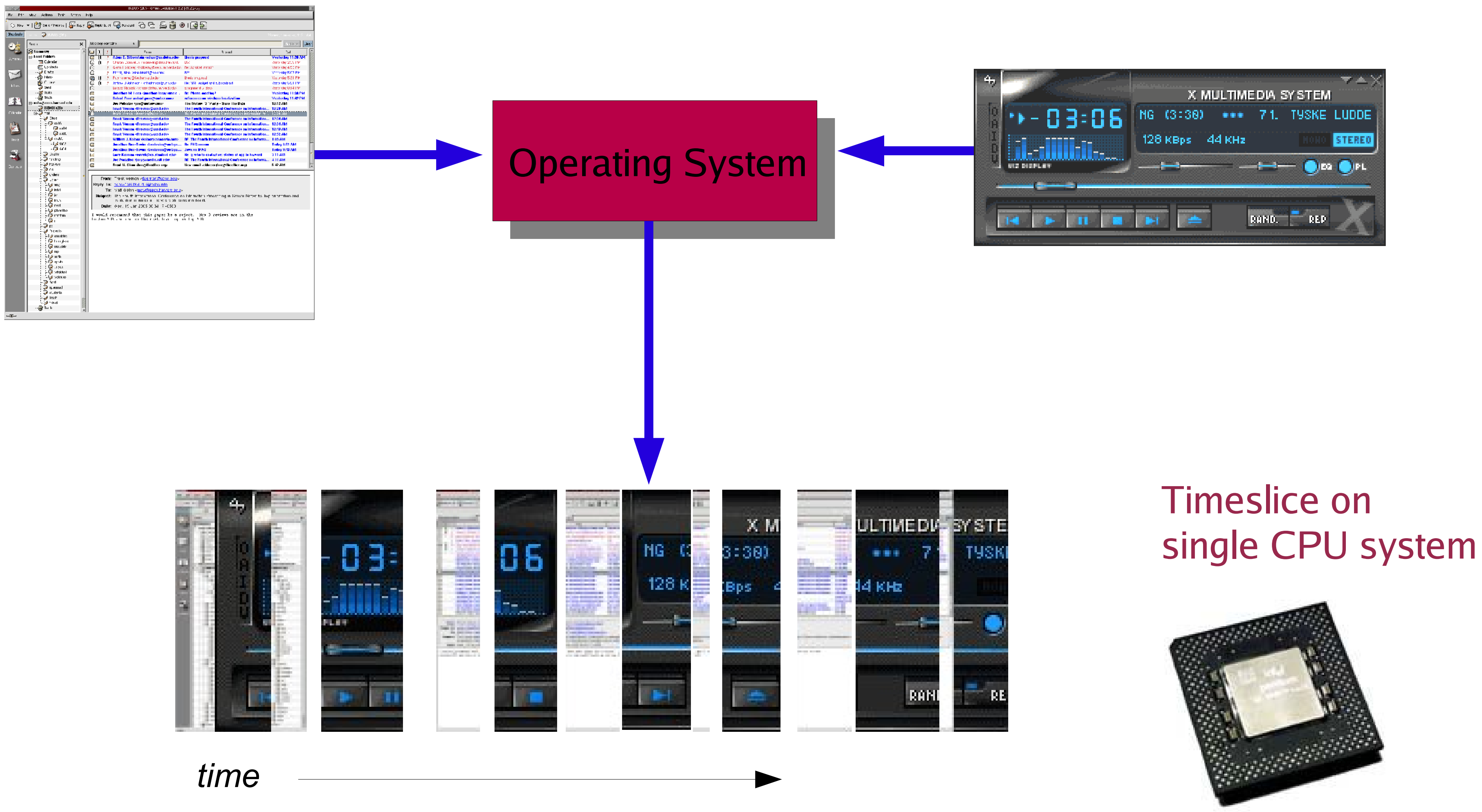
- So far, we have discussed memory management for a *single* program running on a computer.
- Obviously, most modern computers run multiple programs simultaneously.
  - This is called **multiprogramming** or **multitasking.**



# Timeslicing

The OS **timeslices** multiple applications on a single CPU

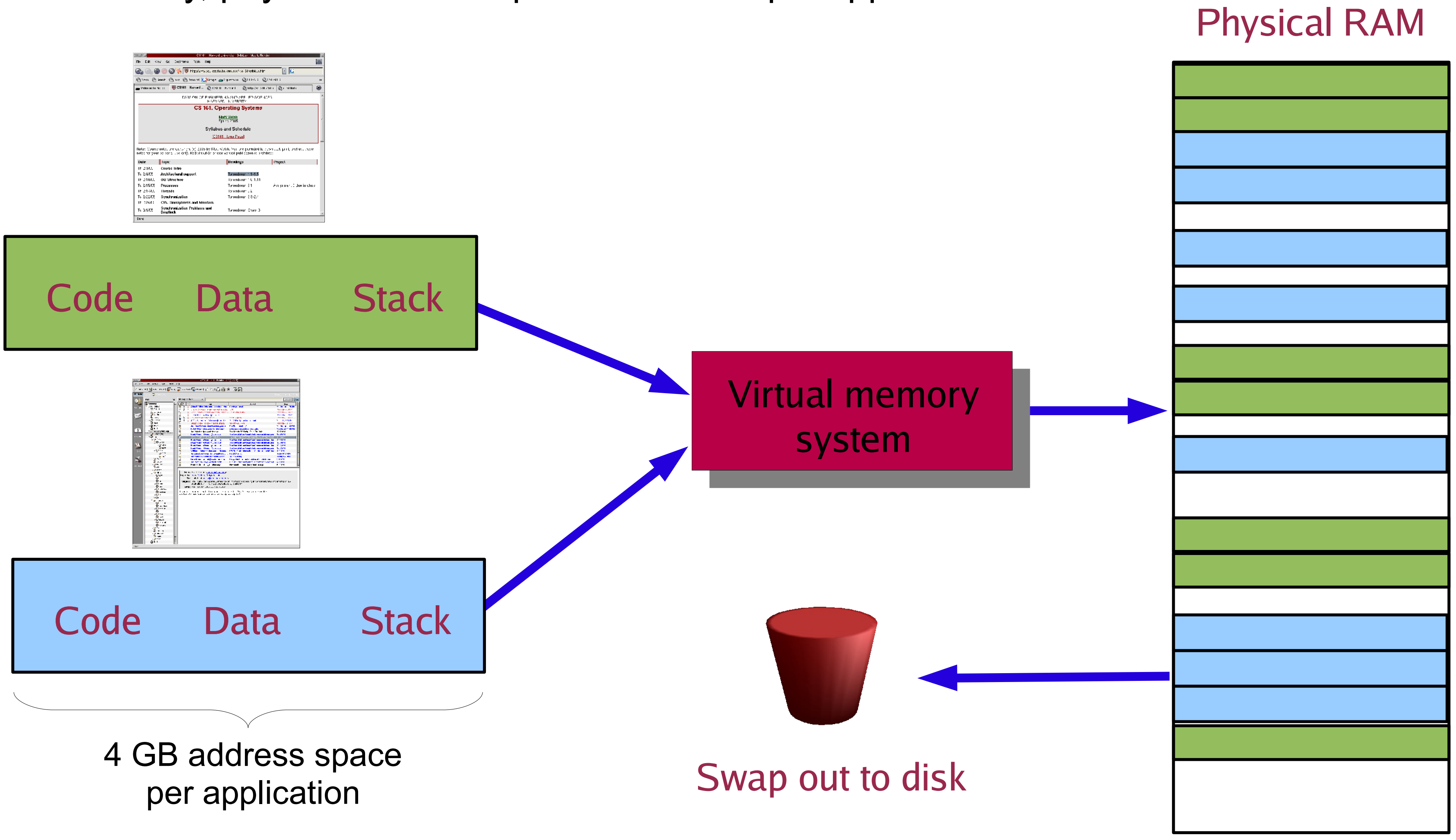
- Switches between applications extremely rapidly, i.e., 100 times/sec
- Each switch between two applications is called a **context switch**.



# Virtual Memory

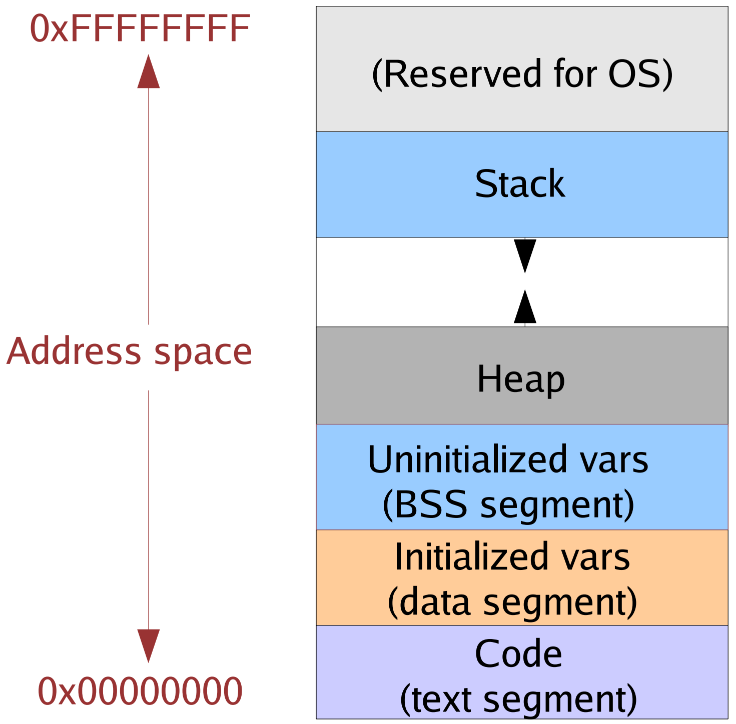
OS also gives every application the illusion of having 4 GB of memory!

- In reality, physical RAM is split across multiple applications



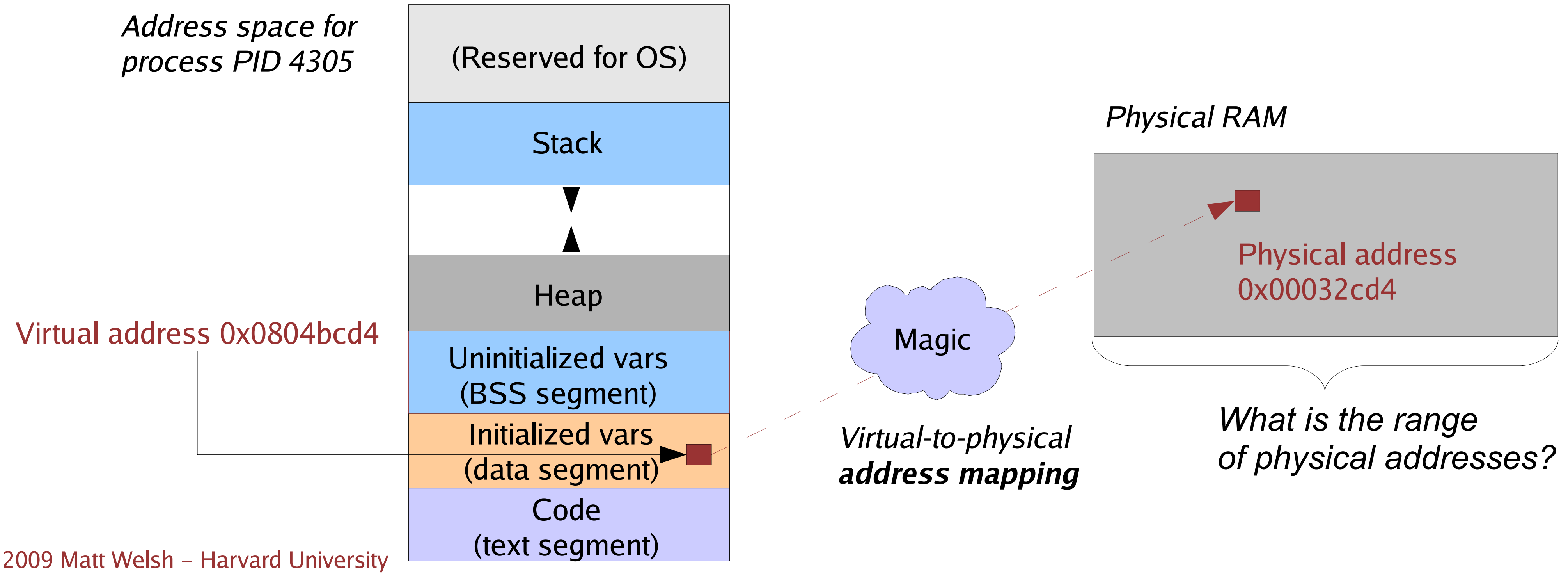
# Virtual memory basics

- Each program running on the machine is called a **process**.
  - Each process has a **process ID** (a number) and an **owner** – the user ID running the process.
  - The UNIX command "`ps`" prints out information about the running processes.
- Each process has its own **address space**
  - The contents of that process' memory, visible only to that process.



# Virtual addresses

- A process uses **virtual addresses** to refer to memory locations in its address space.
  - *Every memory address we have talked about in this class so far has been a virtual address!*
- The OS and hardware map the virtual address to a **physical address**
  - Physical address is the location in the DRAM chips.
  - In general, physical addresses are *hidden* from processes.


Address space for process PID 4305
(Reserved for OS)
Stack
Heap
Uninitialized vars (BSS segment)
Initialized vars (data segment)
Code (text segment)
Virtual address 0x0804bcd4
Magic
Virtual-to-physical address mapping
Physical RAM
Physical address 0x00032cd4
What is the range of physical addresses?

## Virtual memory benefits

VM allows the physical memory to be *shared* by multiple running processes.

- Even better, the processes don't even need to know that this sharing is going on!

VM *isolates* processes from each other.

- OS ensures that virtual addresses in one process are mapped to *different* physical addresses than virtual addresses in another process.
- So, two processes can both refer to virtual address “0x0804bc40” ... but they can be mapped to *two different locations* in physical memory.

There's no way for a process to access memory outside of its own address space.

- By definition, **every** address a processes accesses is within its own address space.
- So the idea of accessing something “outside of your address space” is nonsensical.

# More VM benefits

VM allows the OS to **swap** portions of a process' address space out to disk, when they have not been accessed in a while.

- Allows OS to “recycle” physical memory for different uses (e.g., another process).
- In this way, the system can run many more programs than could otherwise fit into a given amount of physical memory.

Likewise, OS can avoid allocating physical memory to addresses that the process does not access.

- Example: Program that never executes some portion of its code (e.g., debugging or error handling routine).
- Why should the OS place that code in physical RAM, if it hasn't been called yet?

# Implementation questions

How do we map virtual addresses to physical addresses?

How do we keep track of which physical addresses are being used?

How do we swap things to and from disk?

How do we make this transparent to the process?

How do we make it all fast?

Hint: Virtual memory requires **hardware support**.

- Can't do all of this without some specialized hardware:
  the **memory management unit (MMU)**
- These days the MMU is typically built into the CPU, but we often talk about it separately, since its function is very different than the rest of the CPU.
- The OS and the MMU work together to make virtual memory happen.

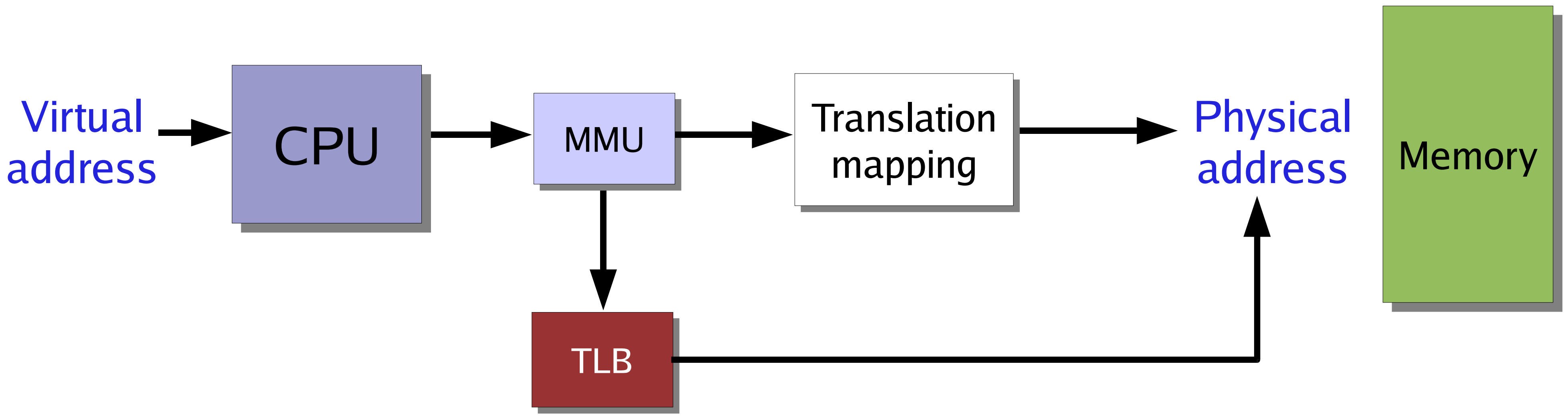
# Hardware support: MMU and TLB

## Memory Management Unit (MMU)

- Hardware that translates a virtual address to a physical address
- Each memory access by the CPU is passed through the MMU
- Translate a virtual address to a physical address
  - *Lots of ways of doing this!*

## Translation Lookaside Buffer (TLB)

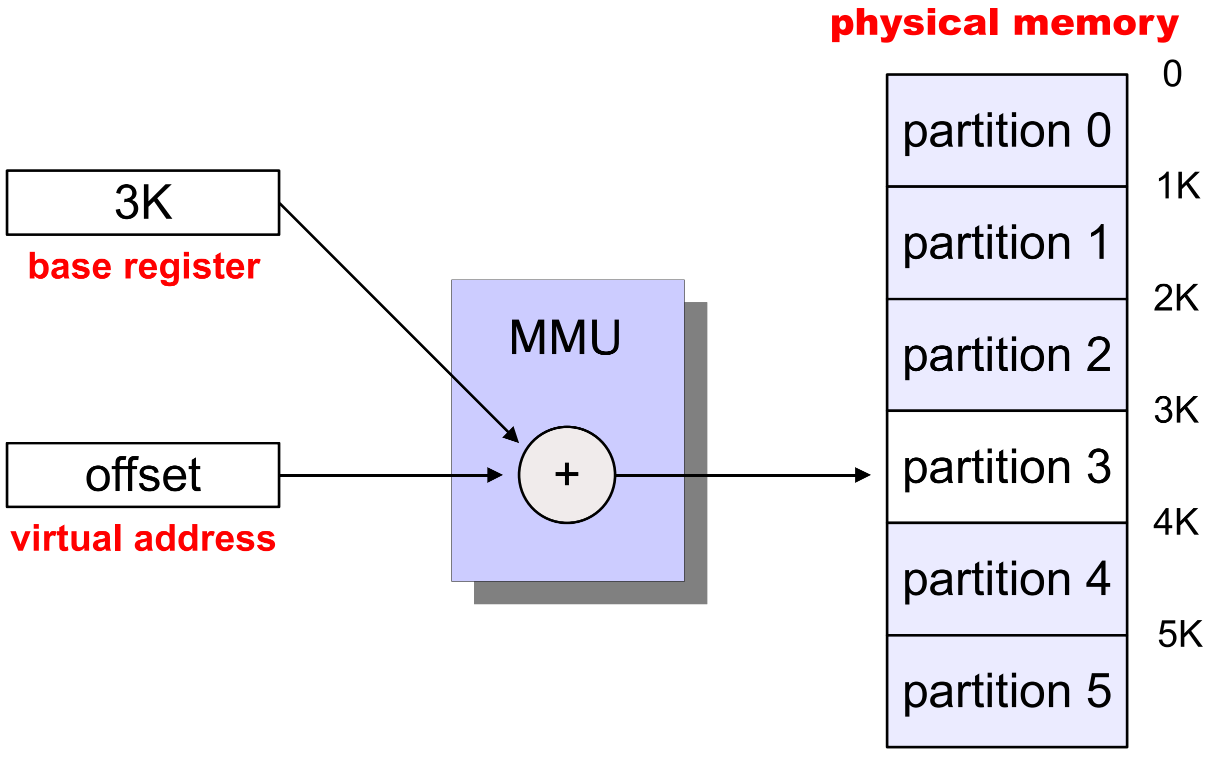
- **Cache** for MMU virtual-to-physical address translations
- Just an optimization – but an important one!



Cache of translations

# Simple approach: Fixed Partitions

- Break memory into fixed-size *partitions*
  - Hardware requirement: *base register*
  - Translation from virtual to physical address: simply add base register to vaddr



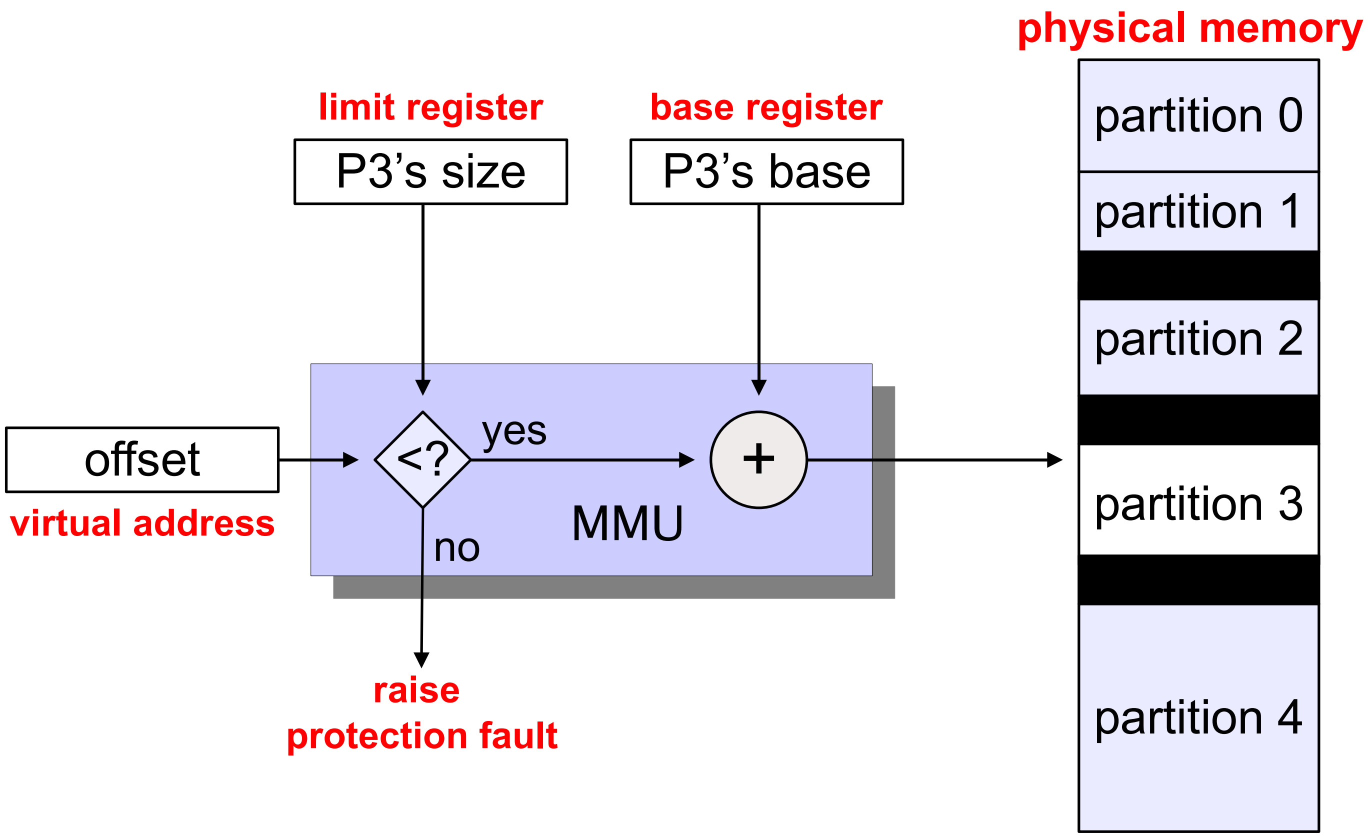
- Advantages and disadvantages of this approach?

# Simple approach: Fixed Partitions

- Advantages:
  - Fast context switch – only need to update base register
  - Simple memory management code: Locate empty partition when running new process
- Disadvantages:
  - Internal fragmentation
    - *Must consume entire partition, rest of partition is "wasted"*
  - Static partition sizes
    - *No single size is appropriate for all programs!*

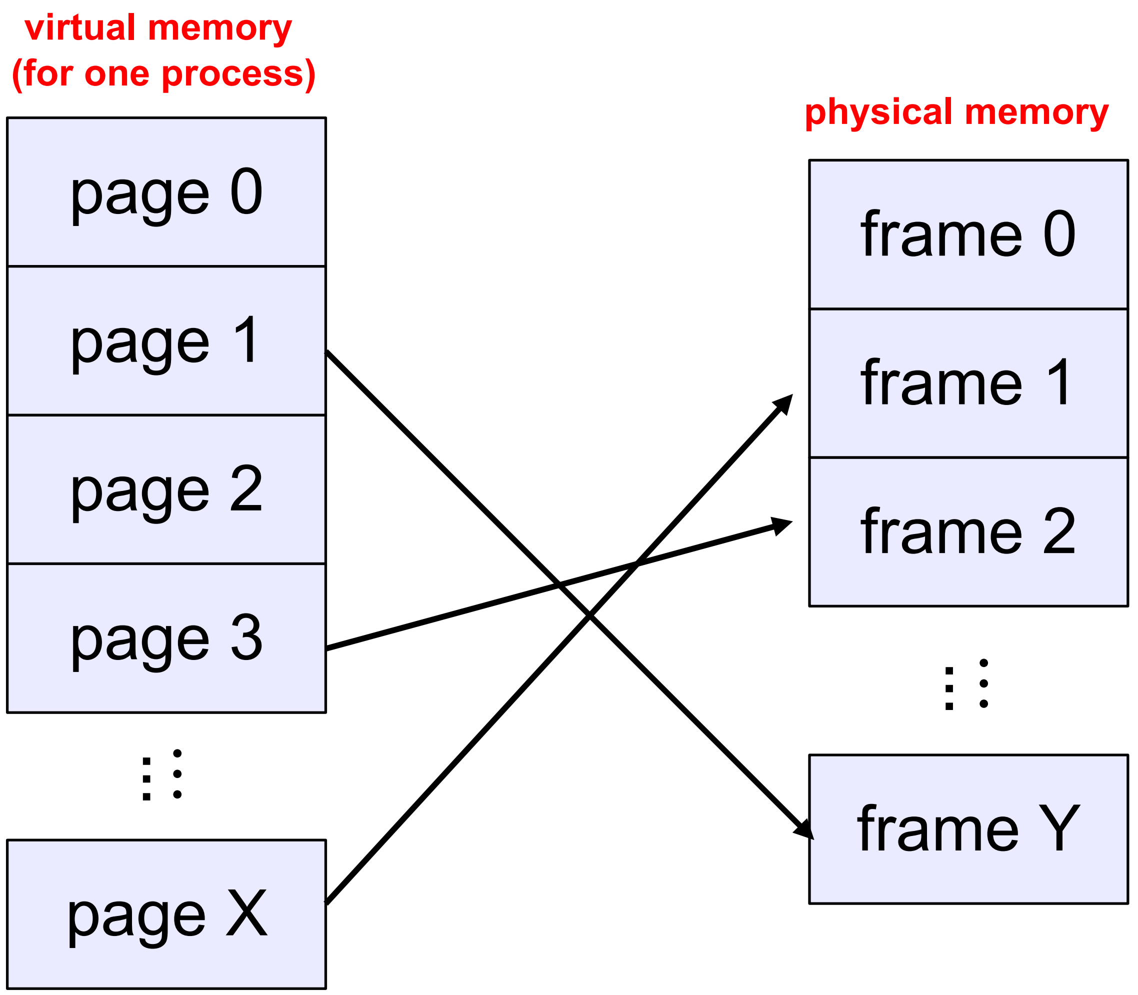
# Variable Partitions

- Obvious next step: Allow variable-sized partitions
  - Now requires both a *base register* and a *limit register* for performing memory access
  - Solves the internal fragmentation problem: size partition based on process needs
- New problem: *external fragmentation*
  - As processes run and complete, holes are left in physical memory



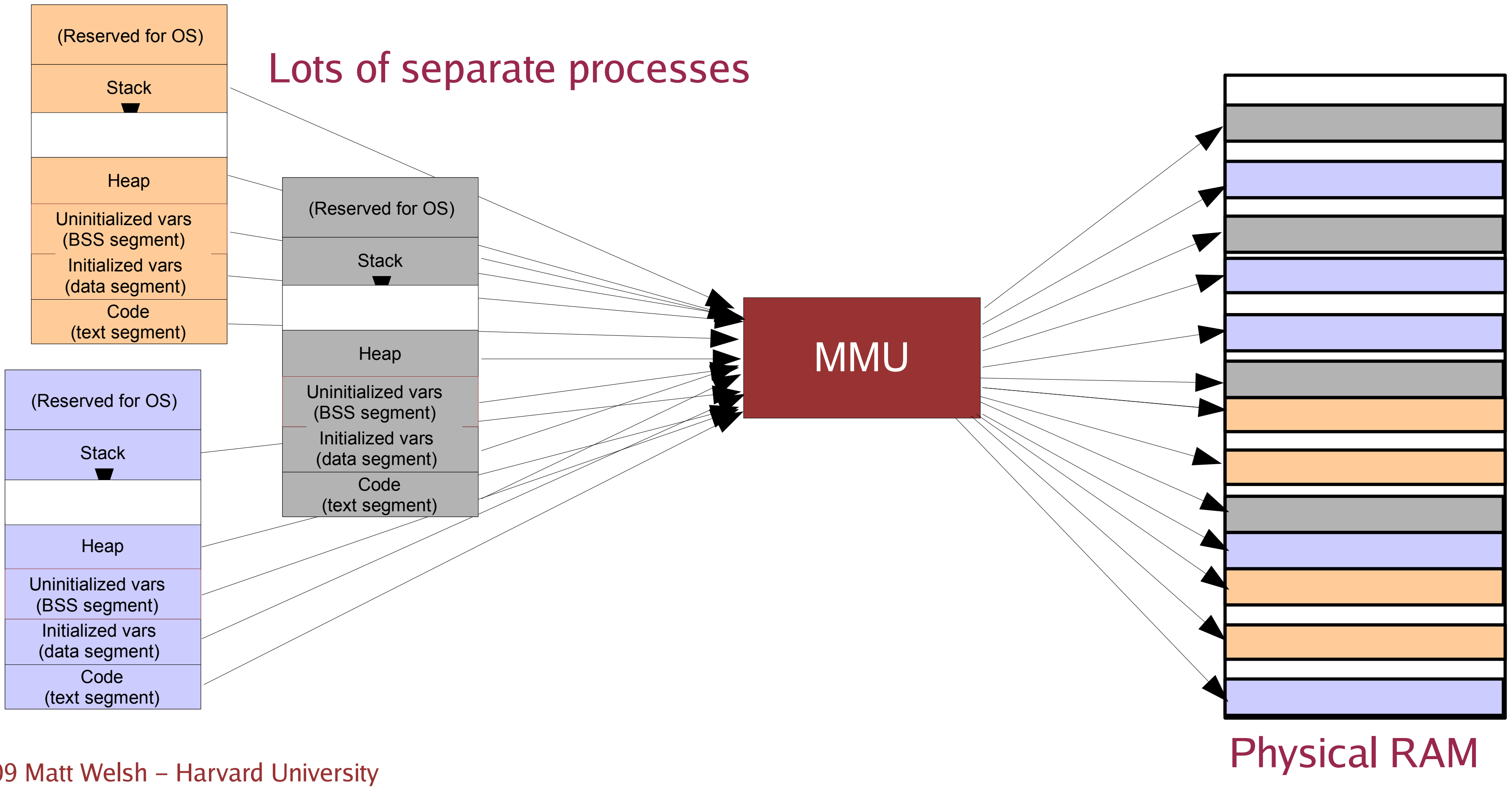
# Modern technique: Paging

- 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

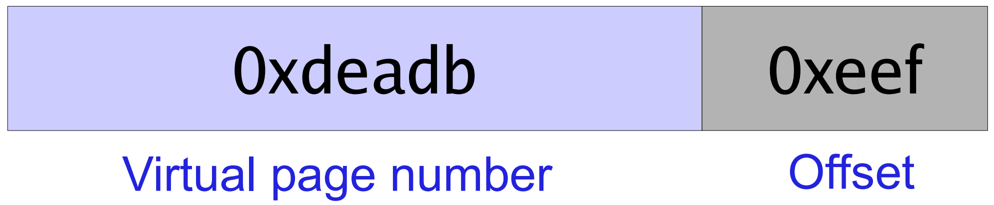
- Application believes it has a single, contiguous address space ranging from 0 to $2^P - 1$ bytes
  - Where P is the number of bits in a pointer (e.g., 32 bits)
- In reality, virtual pages are scattered across physical memory
  - This mapping is invisible to the program, and not even under it's control!


(Reserved for OS)
Stack
Heap
Uninitialized vars (BSS segment)
Initialized vars (data segment)
Code (text segment)
Lots of separate processes
MMU
Physical RAM

# Virtual Address Translation

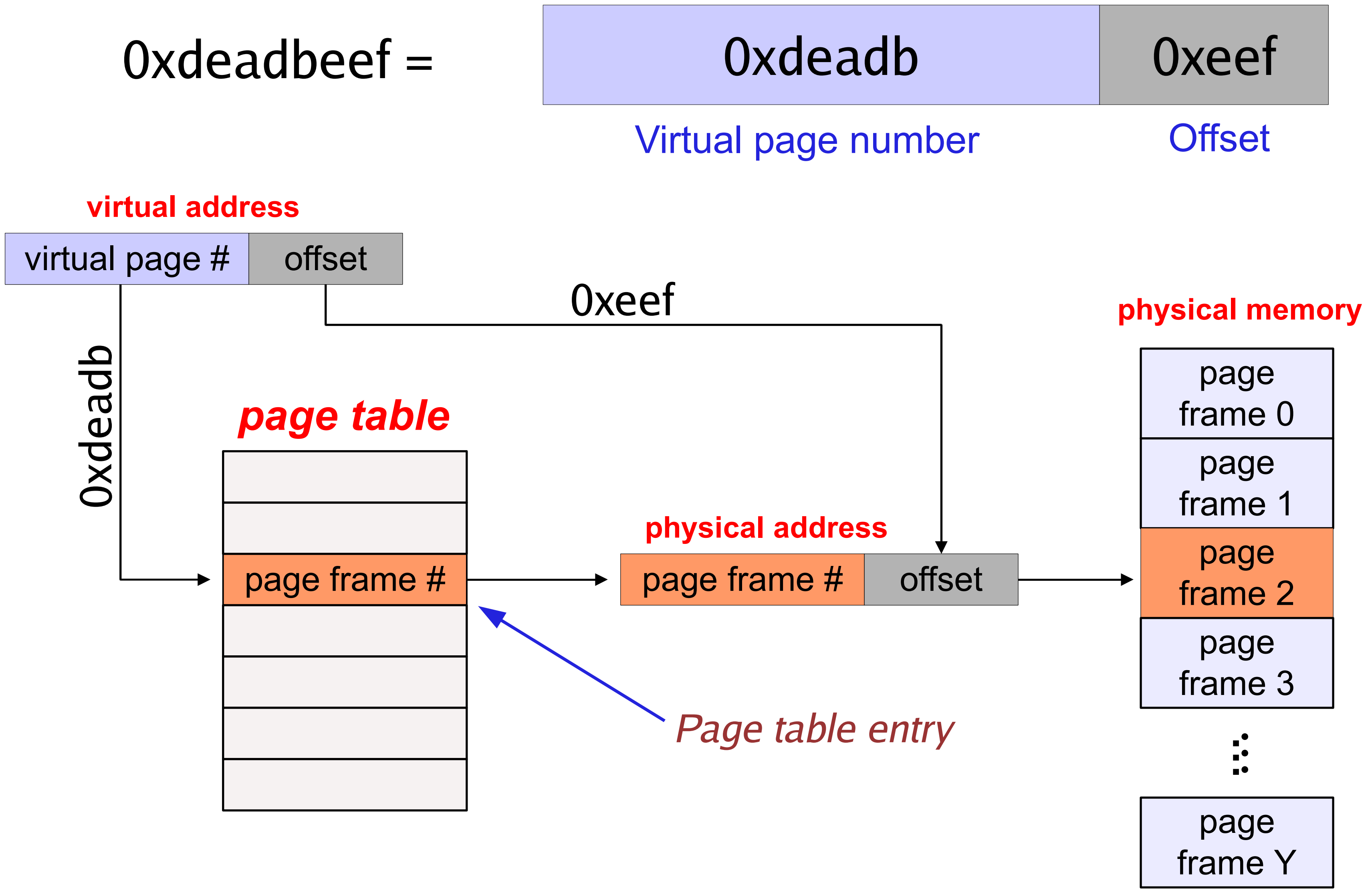
- Virtual-to-physical address translation performed by MMU
  - Virtual address is broken into a *virtual page number* and an *offset*
  - Mapping from virtual page to physical frame provided by a *page table*

0xdeadbeef =



# Virtual Address Translation

- 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*

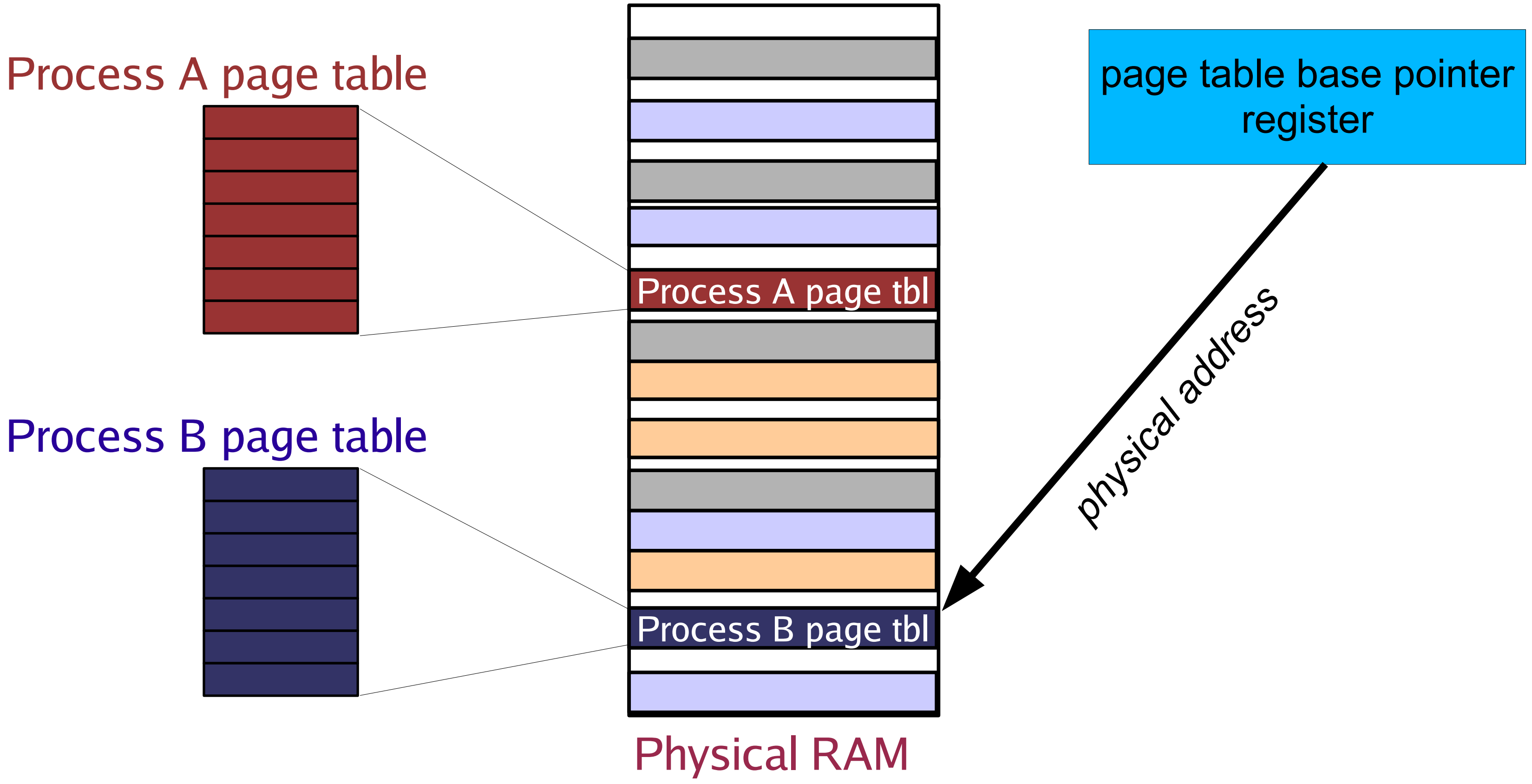


## Advantages of paging

- 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!)

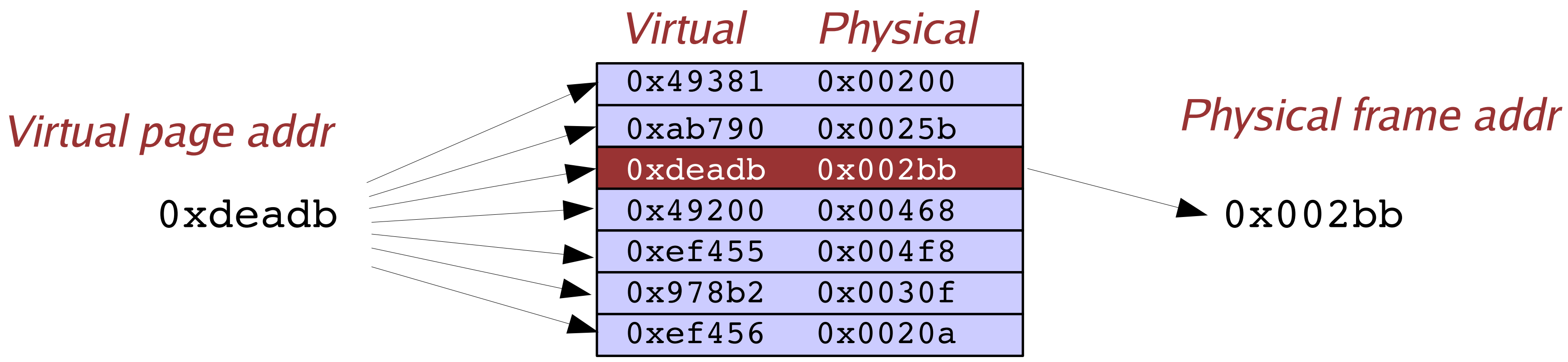
# Page Tables

- 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 page tables for the currently-running process.



# The 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) cache directly on the CPU
    - *Pentium 6 systems have separate data and instruction TLBs, 64 entries each*
  - 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 translation



| Virtual | Physical |
|---|---|
| 0x49381 | 0x00200 |
| 0xab790 | 0x0025b |
| 0xdeadb | 0x002bb |
| 0x49200 | 0x00468 |
| 0xef455 | 0x004f8 |
| 0x978b2 | 0x0030f |
| 0xef456 | 0x0020a |

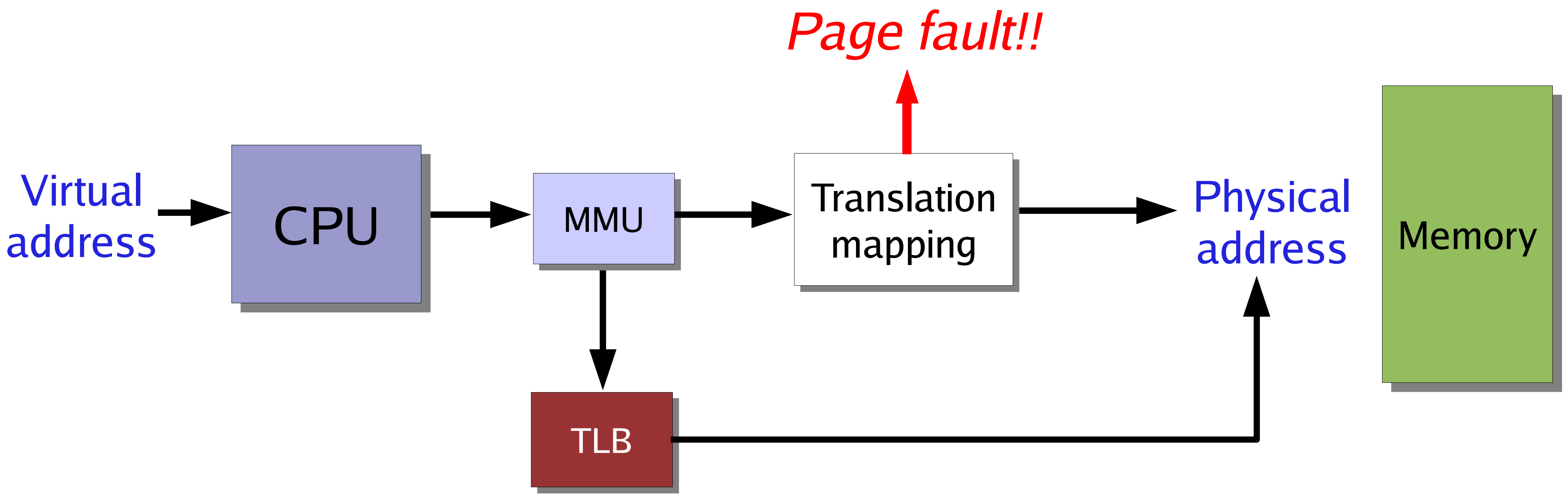
## Page Table Size

- 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?

## Page Table Size

- 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!
  - My brain hurts ... Take CS161 next year :-)

# Page Faults

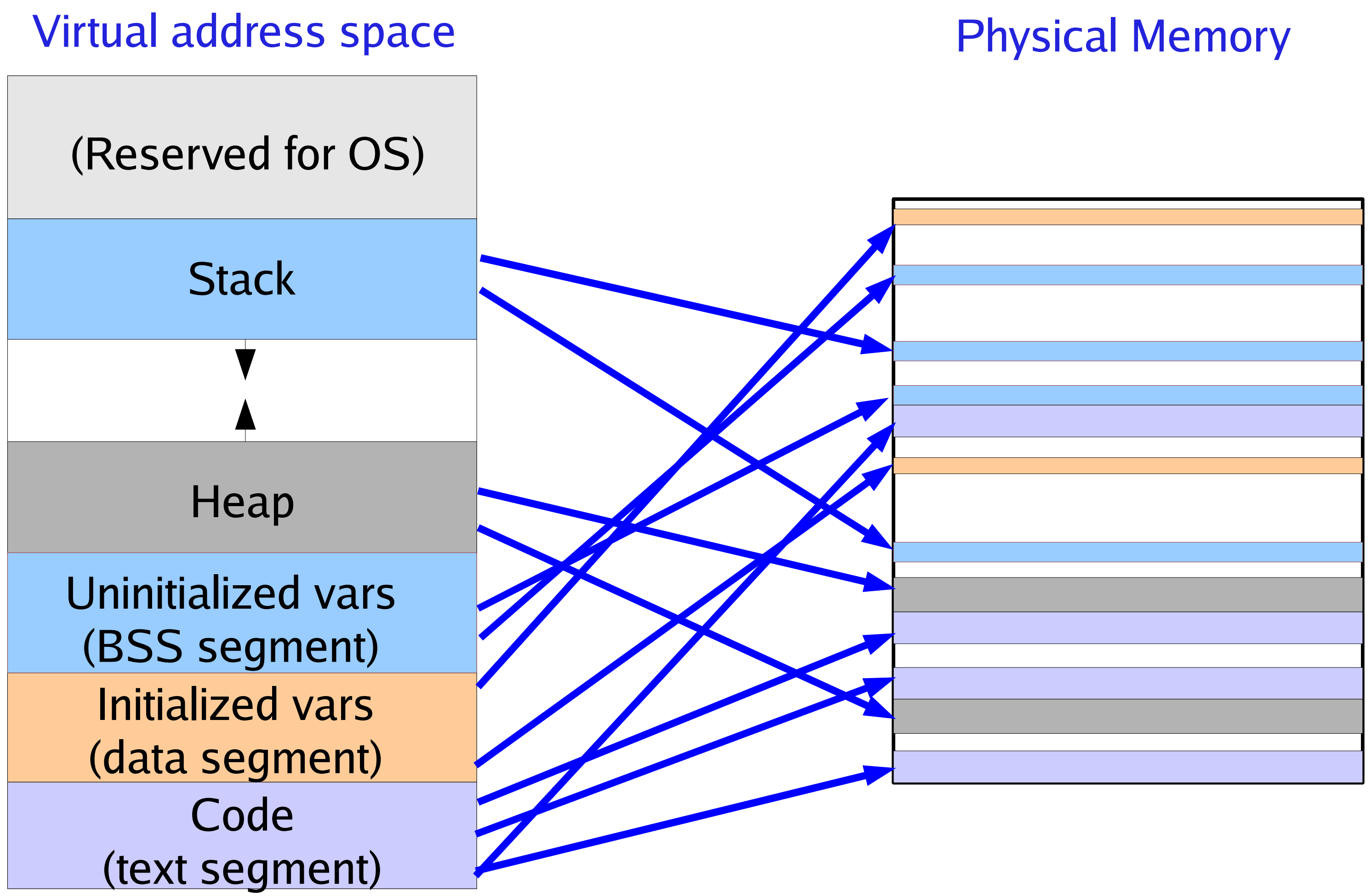


- When a virtual address translation cannot be performed, it's called a *page fault*
- Happens when the page table entry is marked as “invalid”
  - This can happen for a variety of reasons.
  - For example, a bad memory address.
  - Or, something that was previously swapped out to disk!

# Demand Paging

Does it make sense to read an entire program into memory at once?

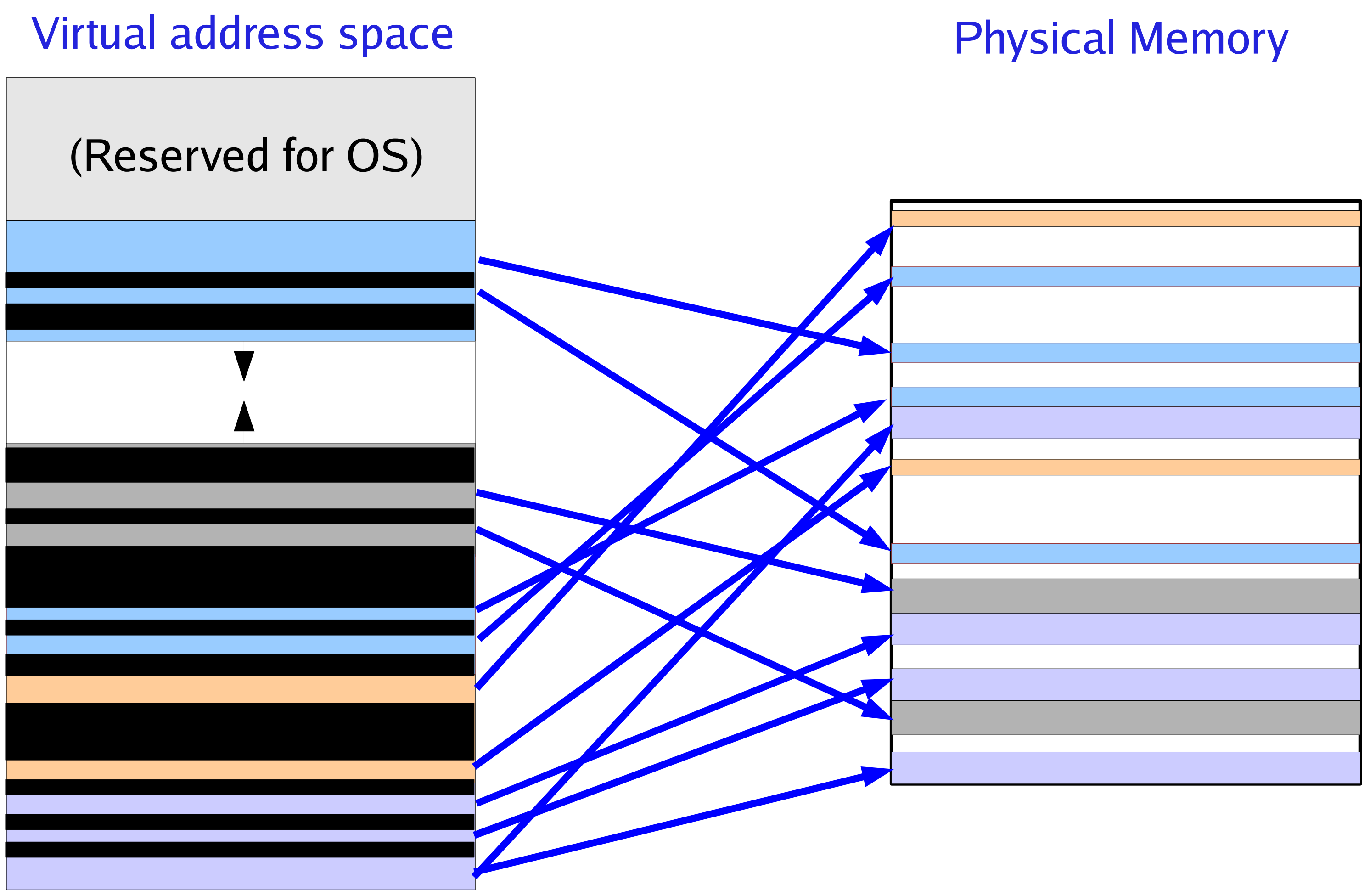
- No! Remember that only a small portion of a program's code may be used!
- For example, if you never use the “save as PDF” feature in OpenOffice...



# Demand Paging

OS may leave "holes" in the process address space

- Ranges of virtual addresses that have no mapping to physical memory.

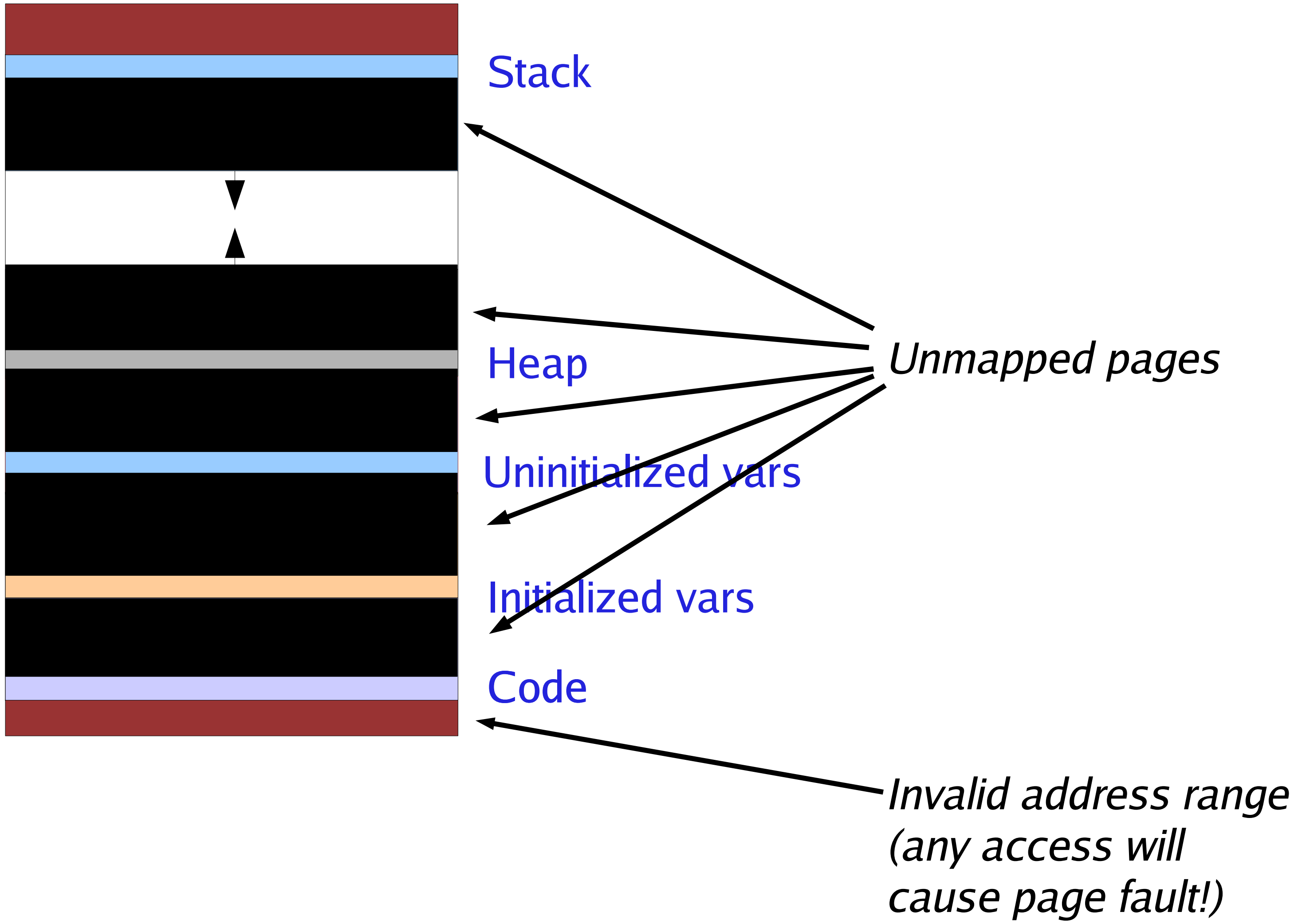


## What are these "holes"?

- Three kinds of "holes" in a process's page tables:
- 1. Pages that are on disk
  - Pages that were swapped out to disk to save memory
  - Also includes code pages in an executable file
    - *When a page fault occurs, load the corresponding page from disk*
- 2. Pages that have not been accessed yet
  - For example, newly-allocated memory
    - *When a page fault occurs, allocate a new physical page*
  - What are the contents of the newly-allocated page???
- 3. Pages that are invalid
  - For example, the "null page" at address 0x0
    - *When a page fault occurs, kill the offending process*

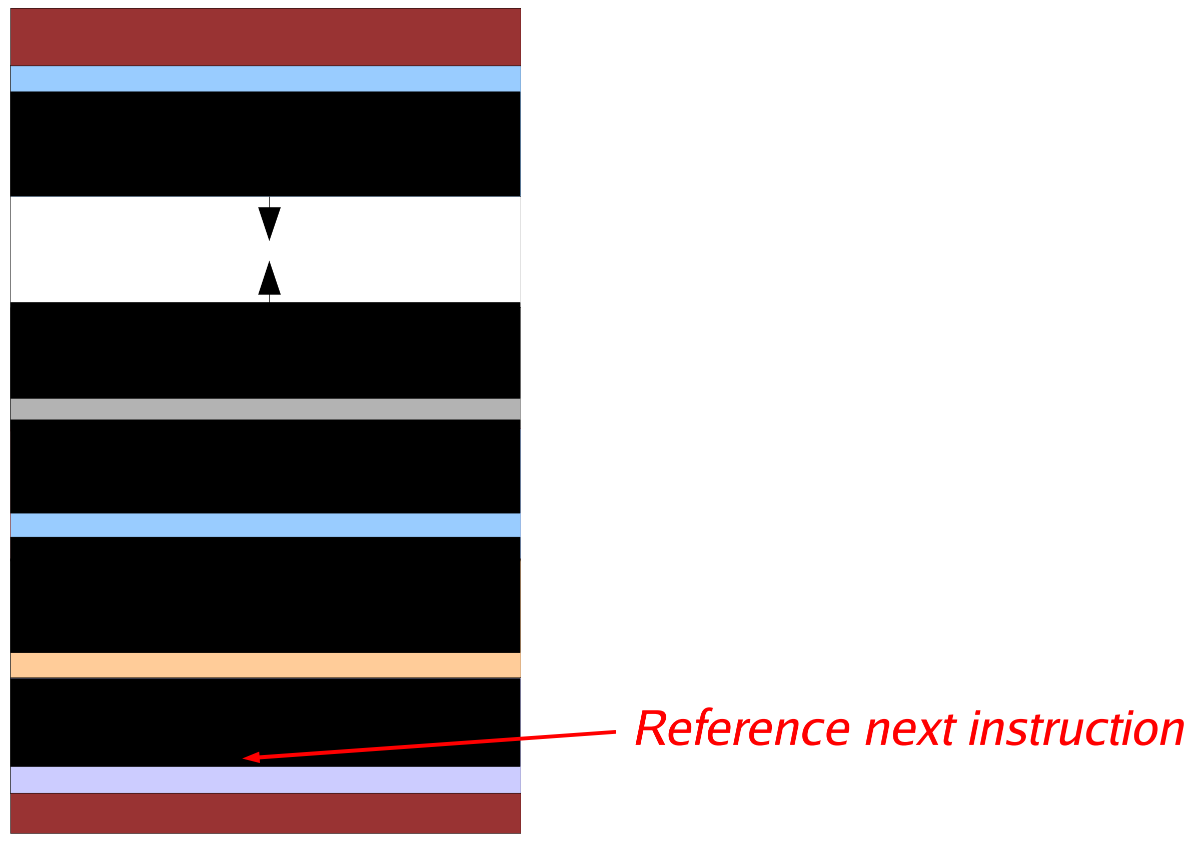
## Starting up a process

- What does a process's address space look like when it first starts up?



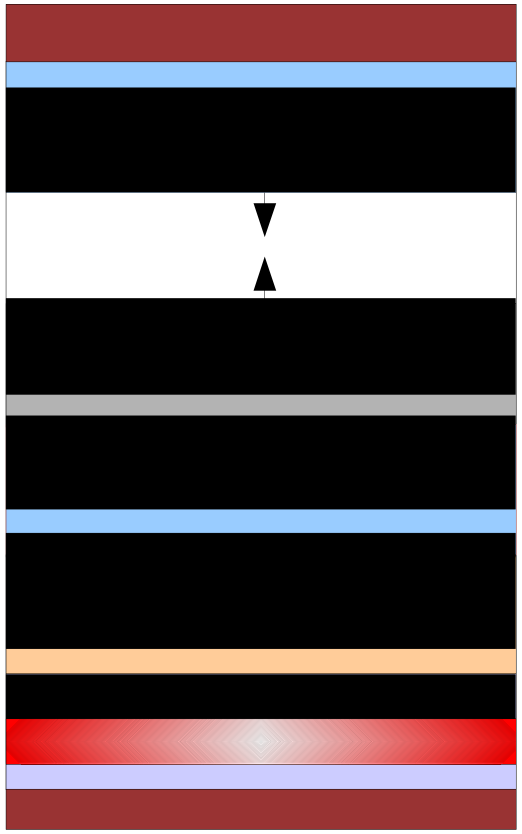
## Starting up a process

- What does a process's address space look like when it first starts up?



# Starting up a process

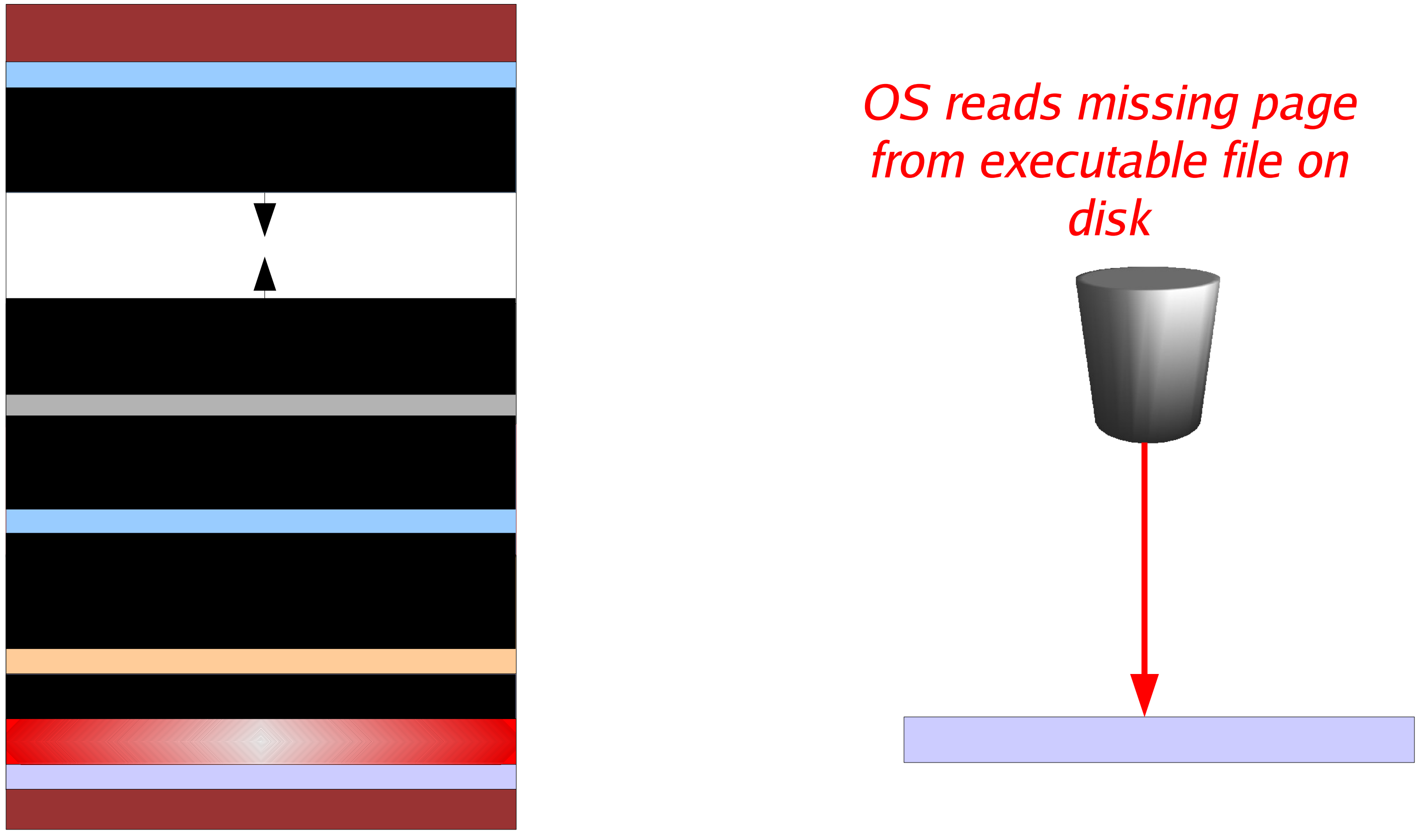
- What does a process's address space look like when it first starts up?



*Page fault!!!*

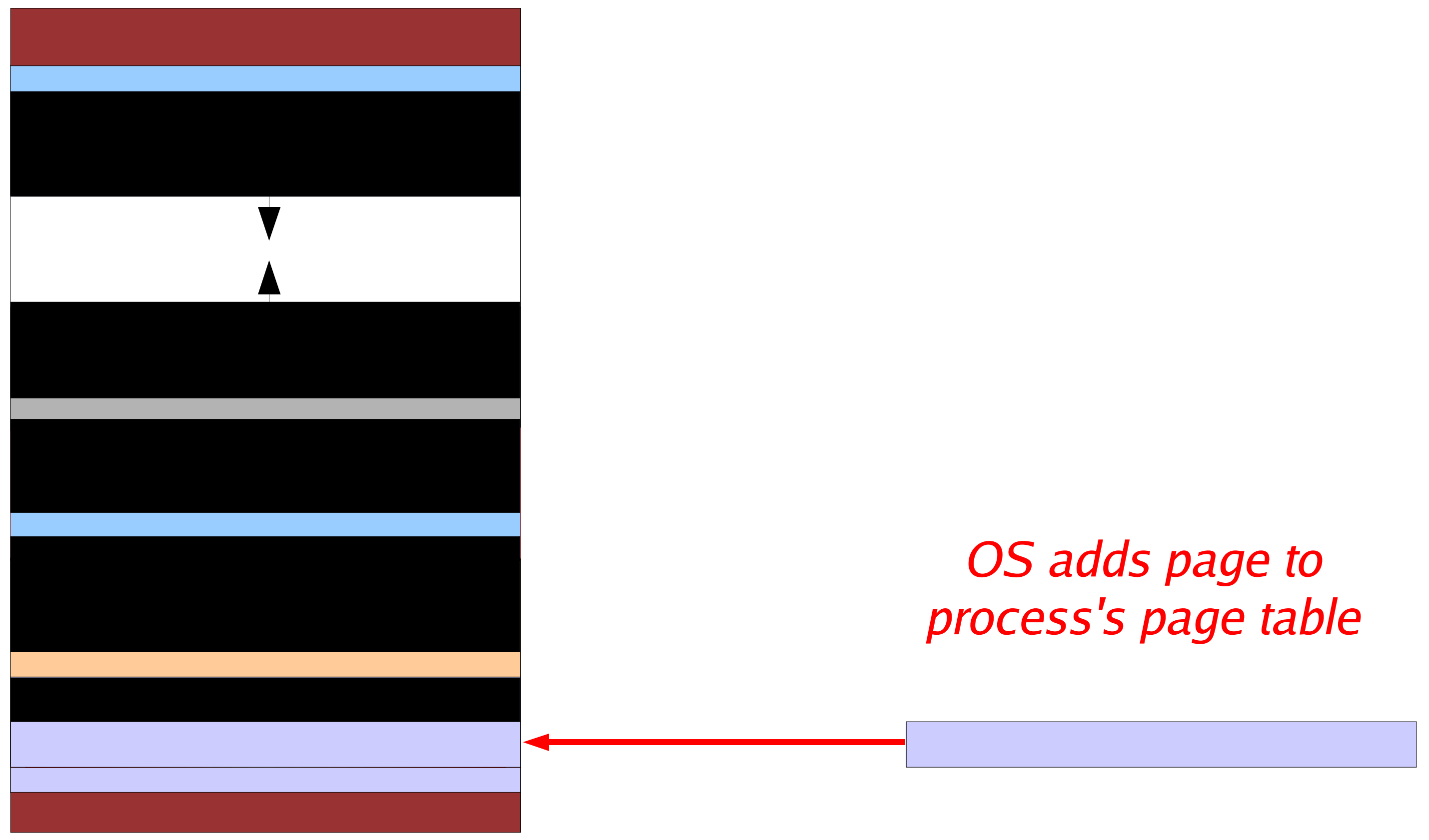
# Starting up a process

- What does a process's address space look like when it first starts up?



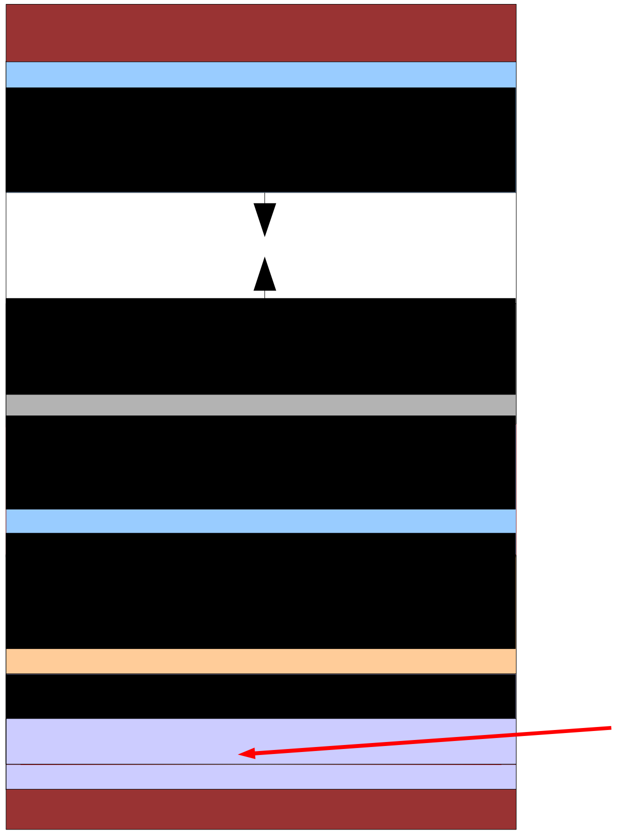
## Starting up a process

- What does a process's address space look like when it first starts up?



## Starting up a process

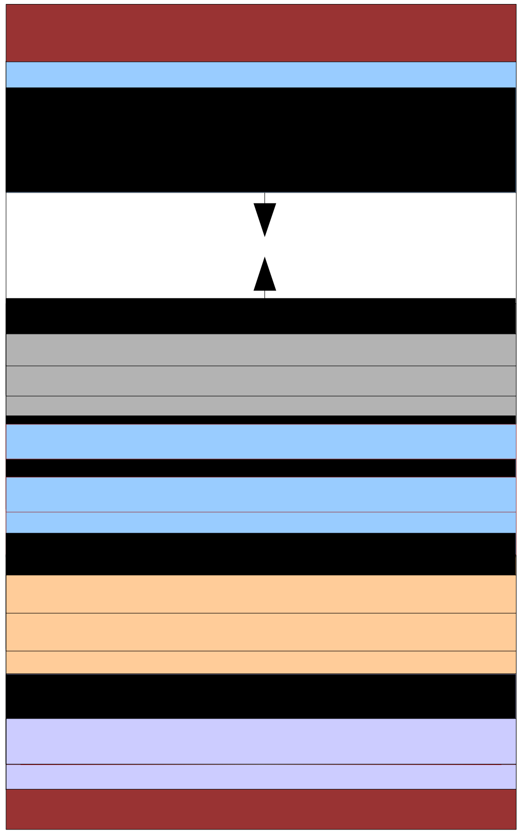
- What does a process's address space look like when it first starts up?



*Process resumes at the next instruction*

## Starting up a process

- What does a process's address space look like when it first starts up?



*Over time, more pages are brought in from the executable as needed*

Obviously, page faults are expensive!

- Cause a trap into the OS, and possibly reading data from disk.
- In terms of overhead,
  Cache hit << TLB hit (memory access) << TLB miss (look in page tables) << Page fault
- Once again, locality is everything!!!

# Application Perspective

- Three requirements for virtual memory:
- Isolation
  - One process cannot access another's pages. Why?
    - *Process can only refer to its own virtual addresses.*
    - *O/S responsible for ensuring that each process uses disjoint* ***physical*** *pages*
- Fast Translation
  - Translation from virtual to physical is fast. Why?
    - *MMU (on the CPU) translates each virtual address to a physical address.*
    - *TLB caches recent virtual->physical translations*
- Fast Context Switching
  - Context Switching is fast. Why?
    - *Only need to swap pointer to current page tables when context switching!*
    - *(Though there is one more step ... what is it?)*

# Take-away points

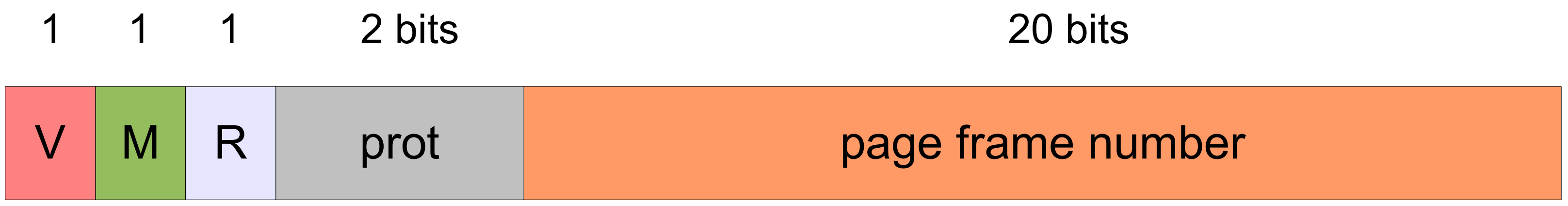
- Virtual memory is an abstraction provided to applications by the OS.
  - Gives each program the illusion of having its own 4 GB address space.
- VM has a bunch of benefits:
  - Allow multiple programs to share (possibly limited) physical RAM.
  - Allows the OS to wrap parts of a program out to disk.
  - Allows the OS to avoid allocating physical RAM to memory locations that aren't accessed.
- VM is implemented by the MMU and OS working together.
  - MMU does virtual-to-physical address translations.
  - OS sets up the page tables that control those translations.
- The TLB is a cache for recent virtual-to-physical mappings.
  - Avoids lookup in page tables for each memory access.
- The OS handles page faults triggered by the MMU.
  - This is the basis for swapping and demand paging.

## Topics for next time

- Systems programming in UNIX
- File abstraction and I/O operations
- Robust and buffered I/O
- Accessing metadata and directories
- Fun with filehandles

# Page Table Entries (PTEs)

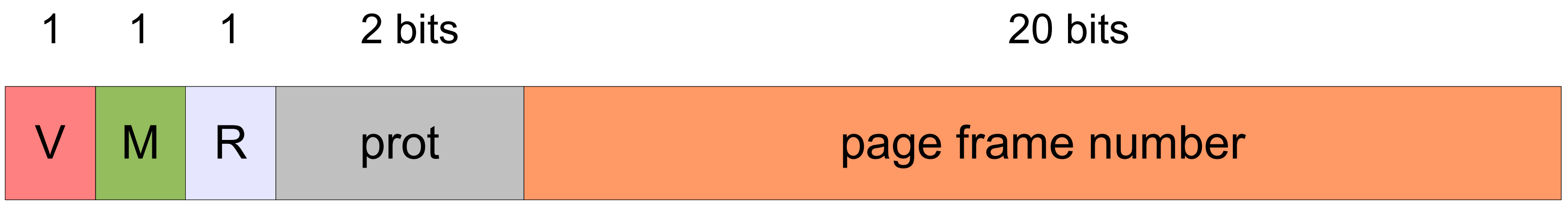
- Typical PTE format (depends on CPU architecture!)



- Various bits accessed by MMU on each page access:
  - Valid bit (V): Whether the corresponding page is in memory
  - Modify bit (M): Indicates whether a page is "*dirty*" (modified)
  - Reference bit (R): Indicates whether a page has been accessed (read or written)
  - 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???**

# Page Table Entries (PTEs)

- What are these bits useful for?



- The R bit is used to decide which pages have been accessed recently.
  - Used to decide which pages can be swapped out.
- The M bit is used to tell whether a page has been modified
  - Why might this be useful?
- Protection bits used to prevent certain pages from being written.
  - Why might this be useful?
- *How are these bits updated?*