Virtual Memory Review

C.V. Wright

CS 491/591

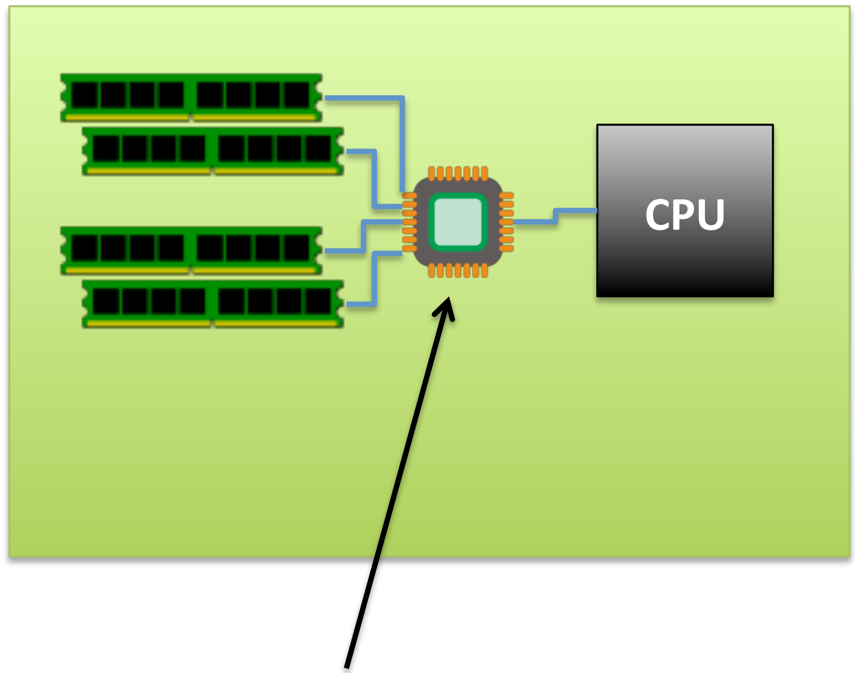
Fall 2015

Memory (Hardware)

Computer memory is typically implemented as a set of chips on pluggable modules

Physical configuration can vary substantially from machine to machine

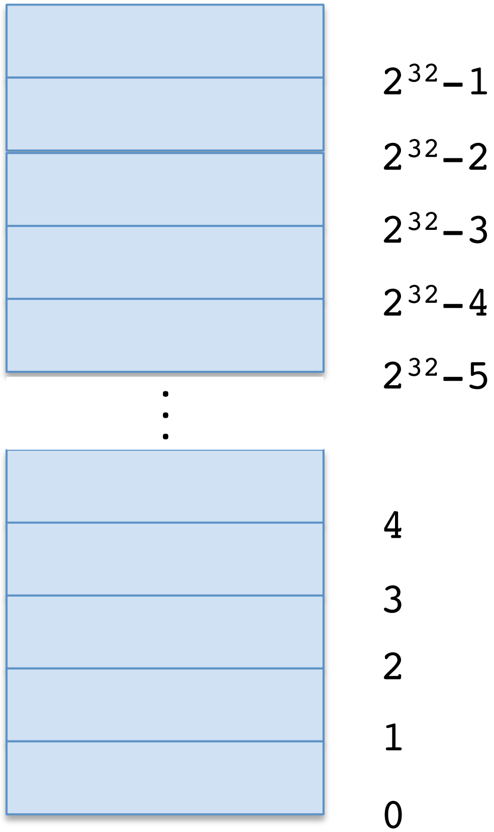
- How many modules?
- How many chips per module?
- How many bytes per chip?



Memory controller presents a nice, much simpler interface to software running on the CPU

Memory (from software)

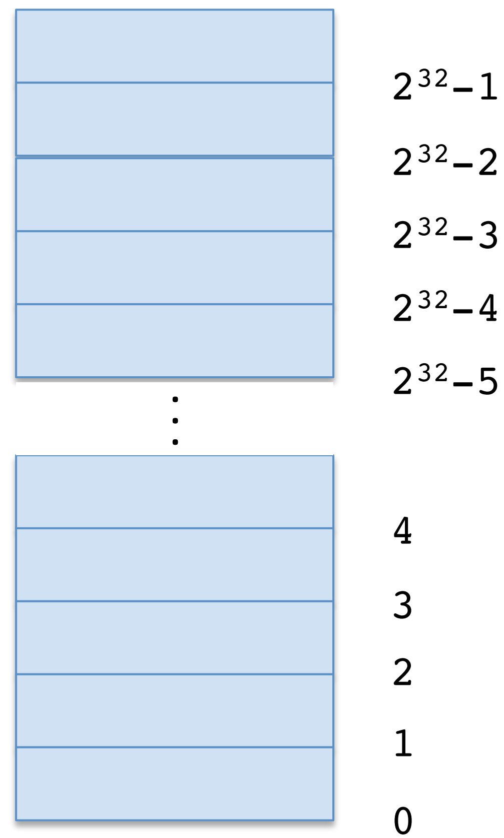
We normally think of memory
as a big array of bytes
(and it is)



Memory (from software)

For example, in C,
pointers are addresses:

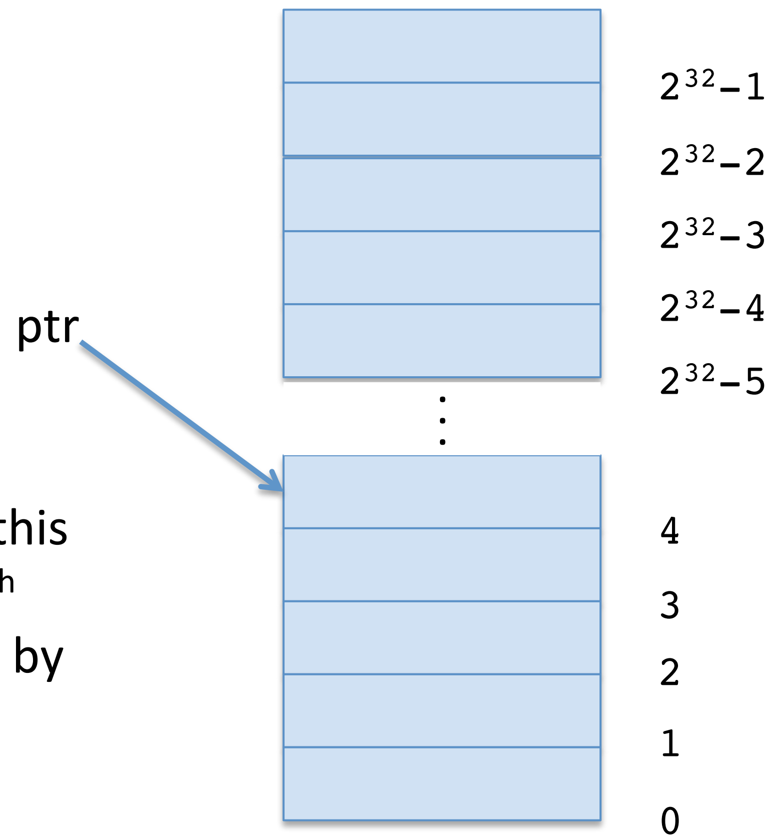
```
char *ptr = 4;
```



Memory

For example, in C,
pointers are addresses:

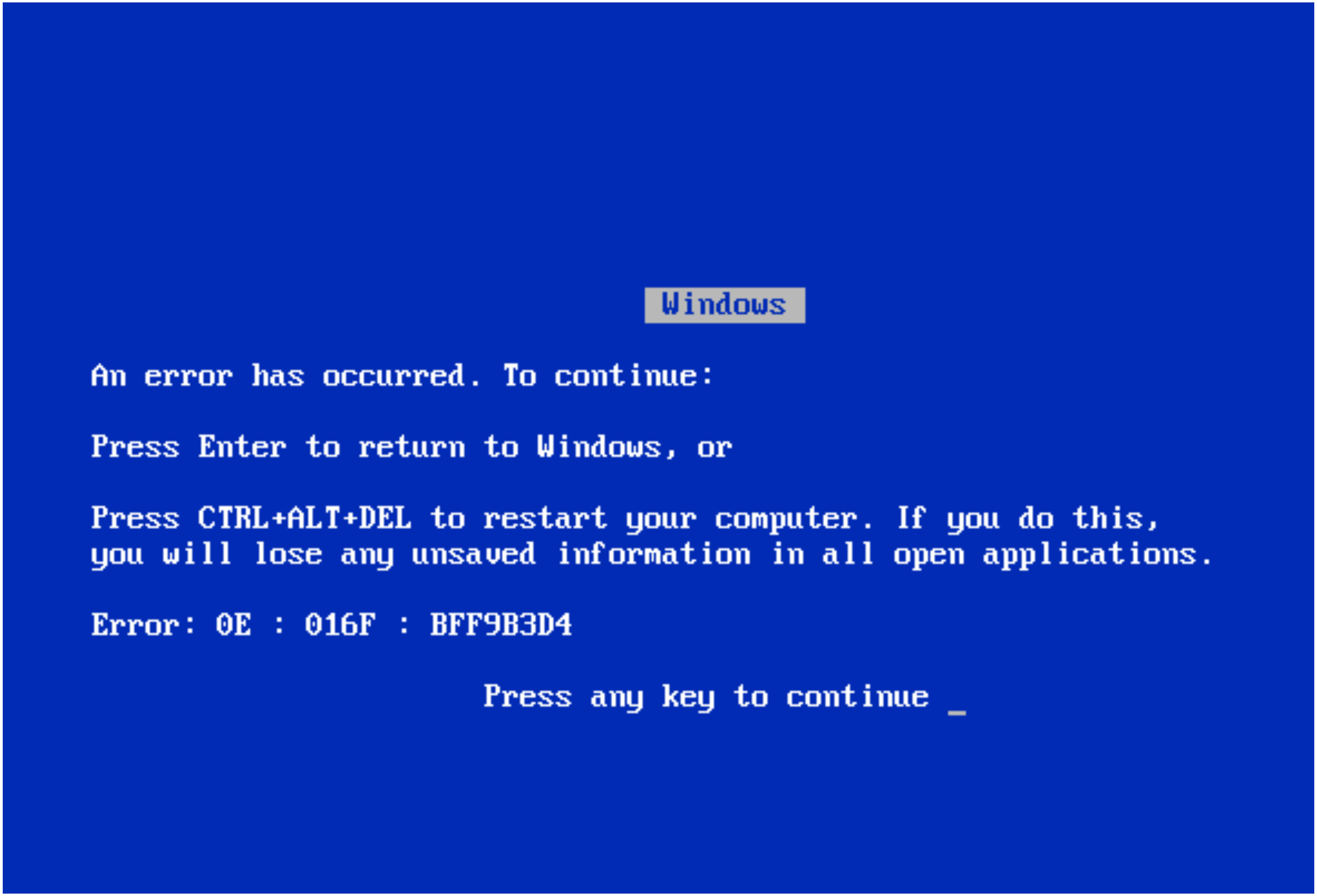
```
char *ptr = 4;
```



Running on "bare metal," this code would point to the 5th byte presented to the CPU by the memory controller.

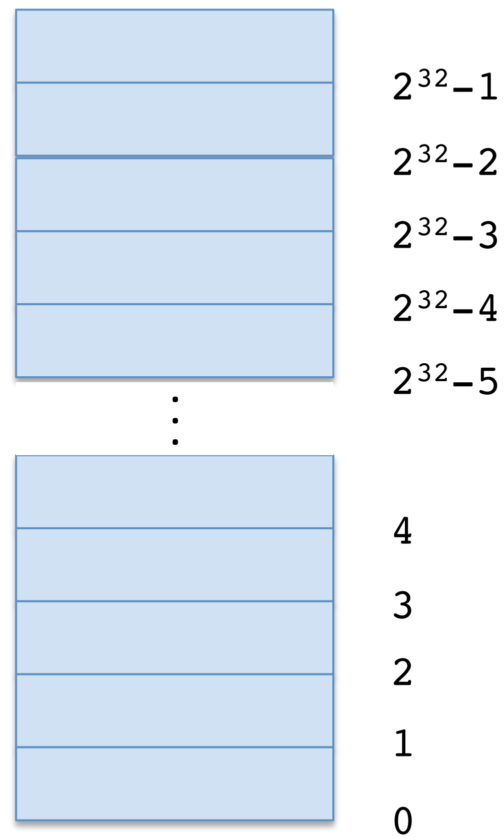
What NOT to do – Memory Protection

- Don't let application programs overwrite each other in memory
- Don't let application programs overwrite the OS in memory
  - http://oreilly.com/centers/windows/brochure/architecture.html
  - http://www.memorymanagement.org/articles/mac.html


Windows
An error has occurred. To continue:
Press Enter to return to Windows, or
Press CTRL+ALT+DEL to restart your computer. If you do this,
you will lose any unsaved information in all open applications.
Error: 0E : 016F : BFF9B3D4
Press any key to continue _

Memory Protection

Problem:
How can we enable multiple programs to use memory at the same time, without interfering with each other?



Memory Protection

Strawman Approach:
Give each program a different region of the memory

Program 1

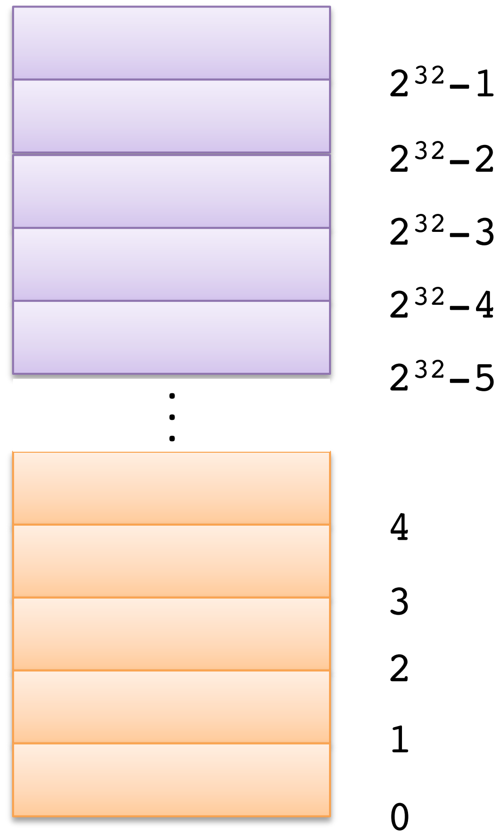
```
…
char *ptr = 4;
…
```

Program 2

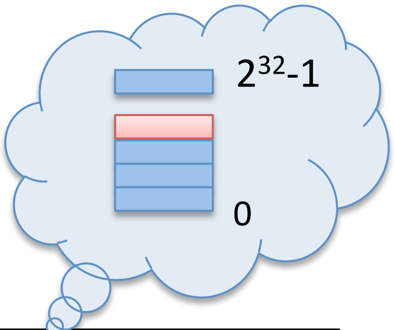
…
`char *ptr = `2^{32}`-4;`
…

Problems:

- Not very flexible
  - Need to know how much memory each program will need in advance
- Requires changes to source code
  - Programs need to know which addresses they can use

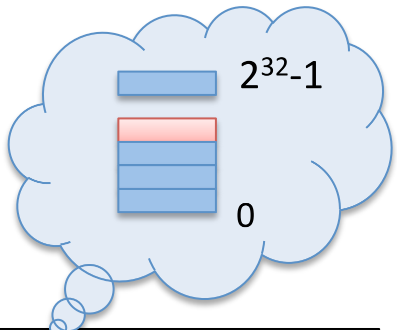


Virtual Memory



Program 1

```
…
char *ptr = 4;
…
```

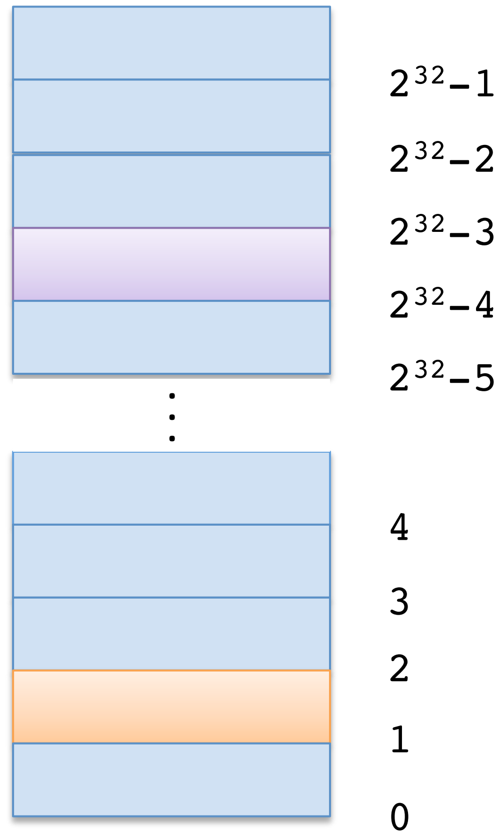


Program 2

```
…
char *ptr = 4;
…
```

Core concepts:

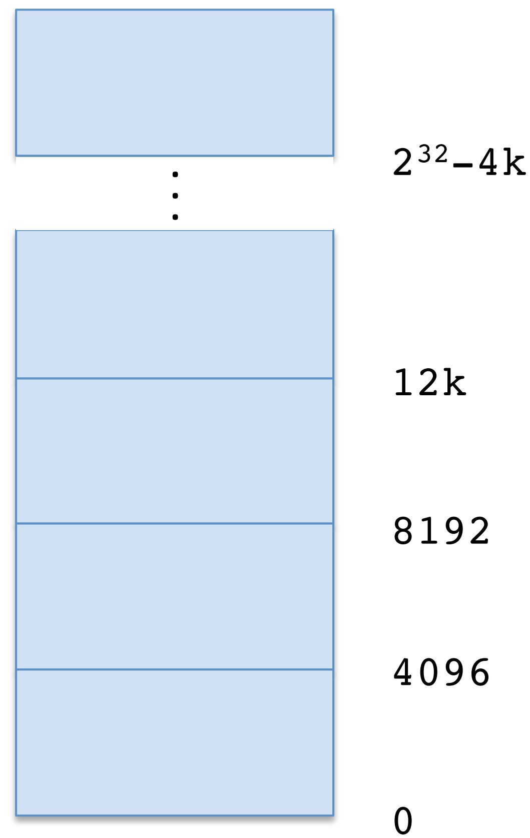
- Let each program run as if it has the whole machine to itself
  - No need to re-write source code
  - Transparently re-map programs' "virtual" addresses to real, physical addresses on the fly
- Dynamically allocate physical memory to programs as they need it



Paged Virtual Memory

Divide the *address space* into fixed-size logical chunks, or *pages*

Typical page size: 4KB

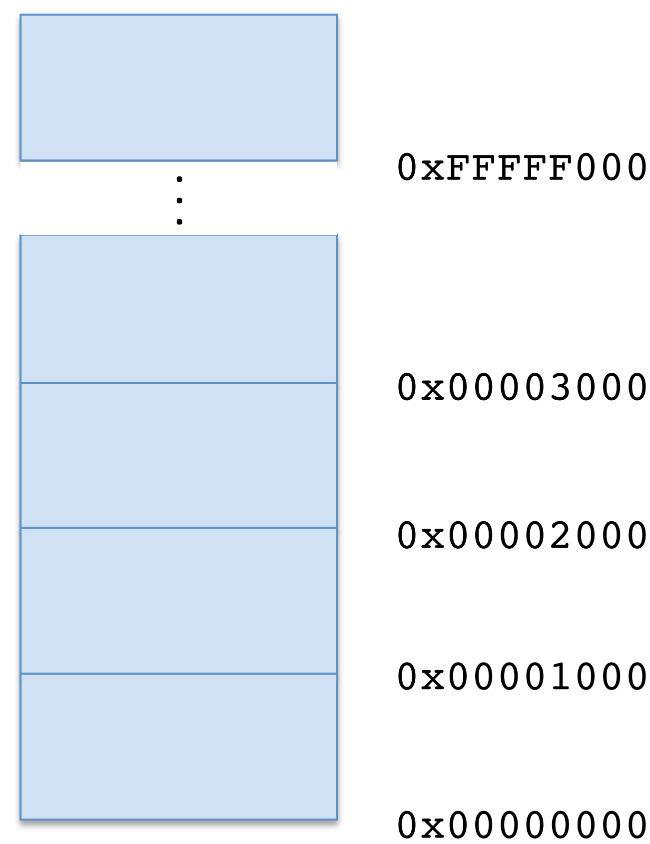


Paged Virtual Memory

Notice: All addresses within a given page share the same high-order bits

So we can think of each address as a *page number*, followed by an *offset* within that page

Example:
Address `0x000010FF`

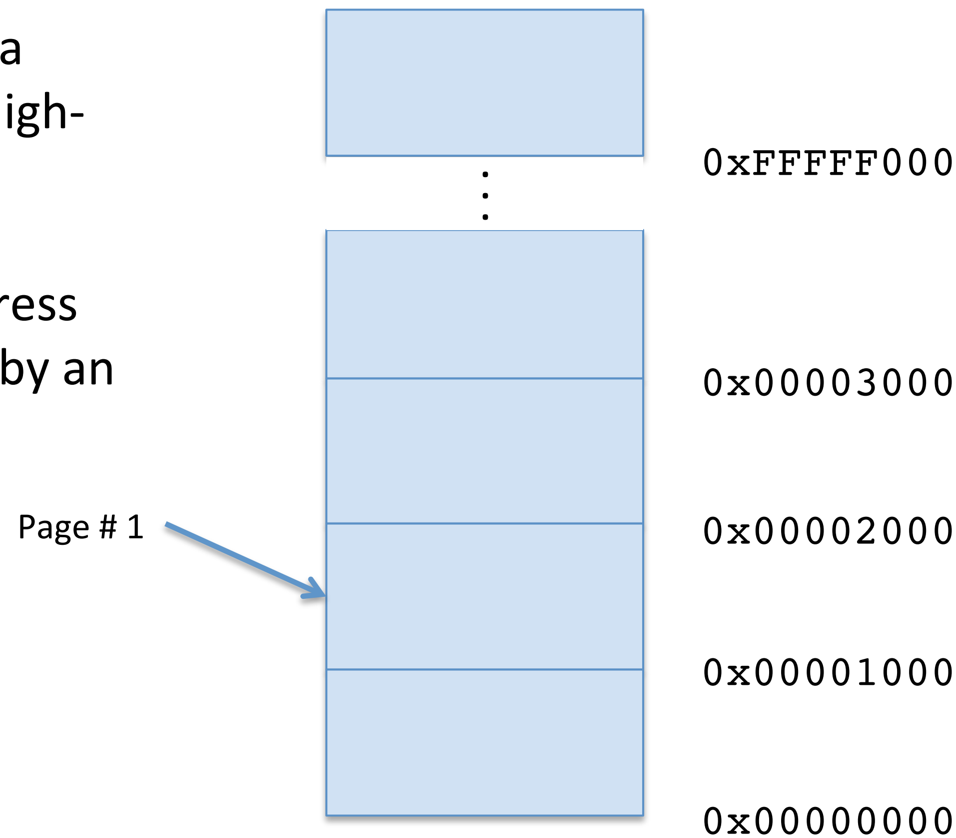


Paged Virtual Memory

Notice: All addresses within a given page share the same high-order bits

So we can think of each address as a *page number*, followed by an *offset* within that page

Example:
Address `0x000010FF`

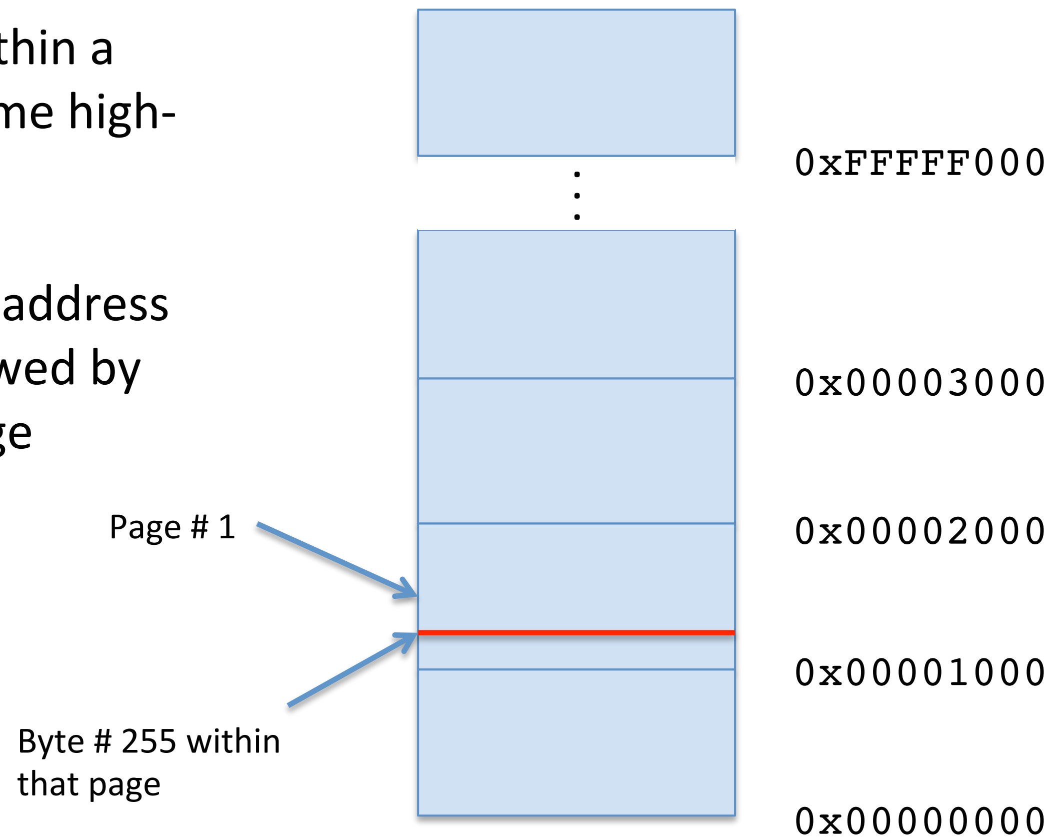


Paged Virtual Memory

Notice: All addresses within a given page share the same high-order bits

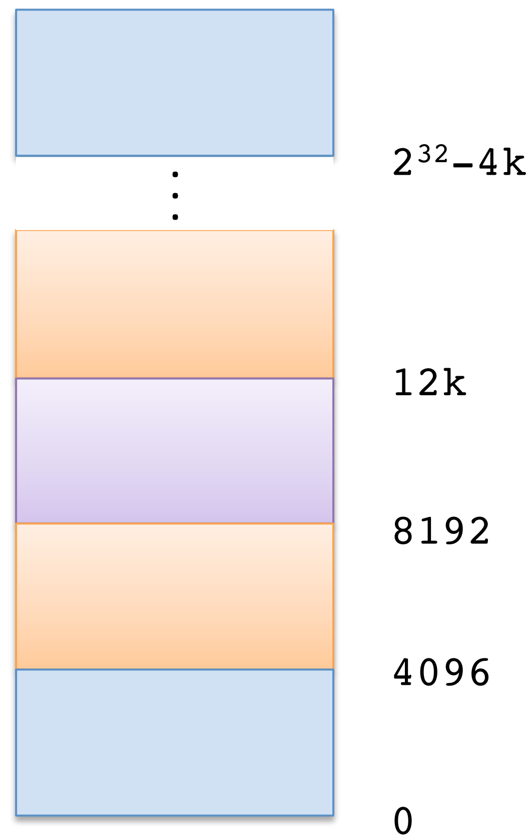
So we can think of each address as a *page number*, followed by an *offset* within that page

Example:
Address `0x000010FF`

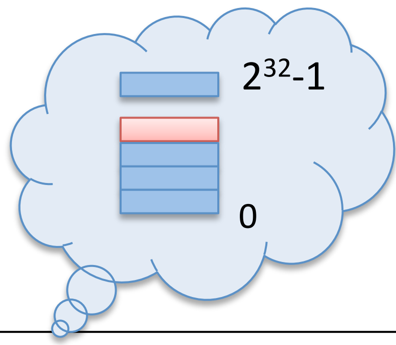


Paged Virtual Memory

Allocate physical memory to processes by the page



Paged Virtual Memory

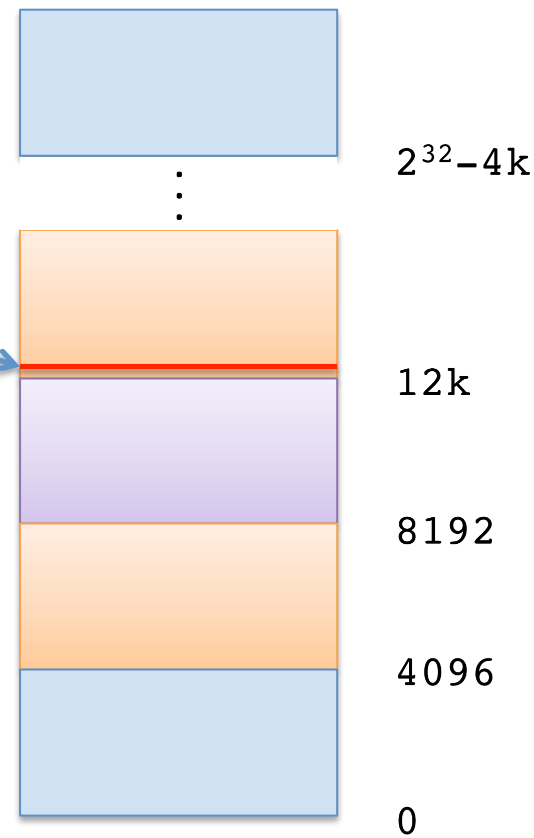


Program 1

```
…
char *ptr = 0x00000004;
…
```

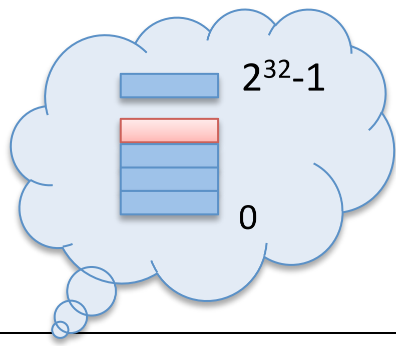
Processes work with virtual addresses

Virtual addresses get re-mapped to physical addresses by the system

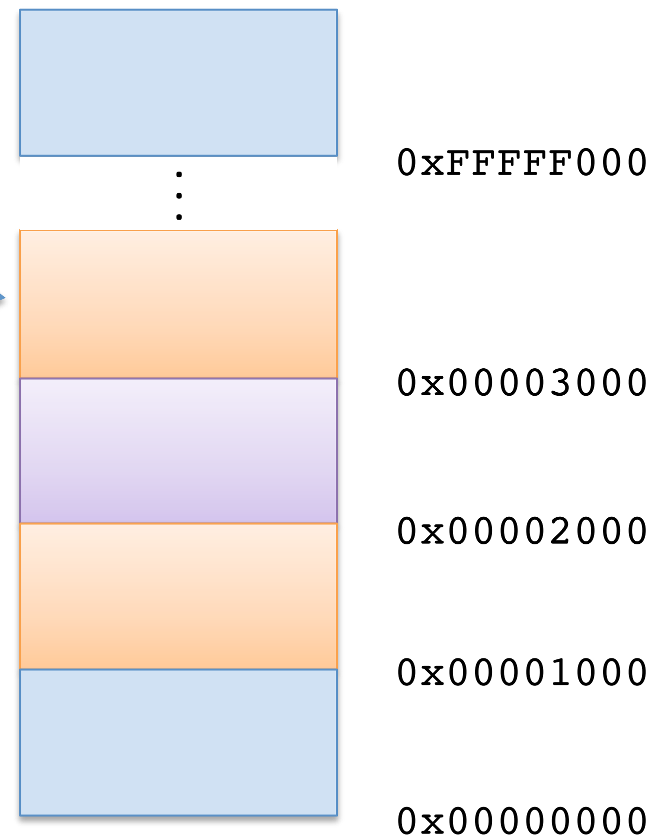


Paged Virtual Memory

Translate virtual addresses to physical addresses by re-mapping page numbers



Physical page # 3



Program 1

```
…
char *ptr = 0x00000004;
…
```

Virtual page # 0

Table-Based Address Translation

How do we store the mapping from virtual page #'s to physical page numbers?

Simple solution: Use an array!

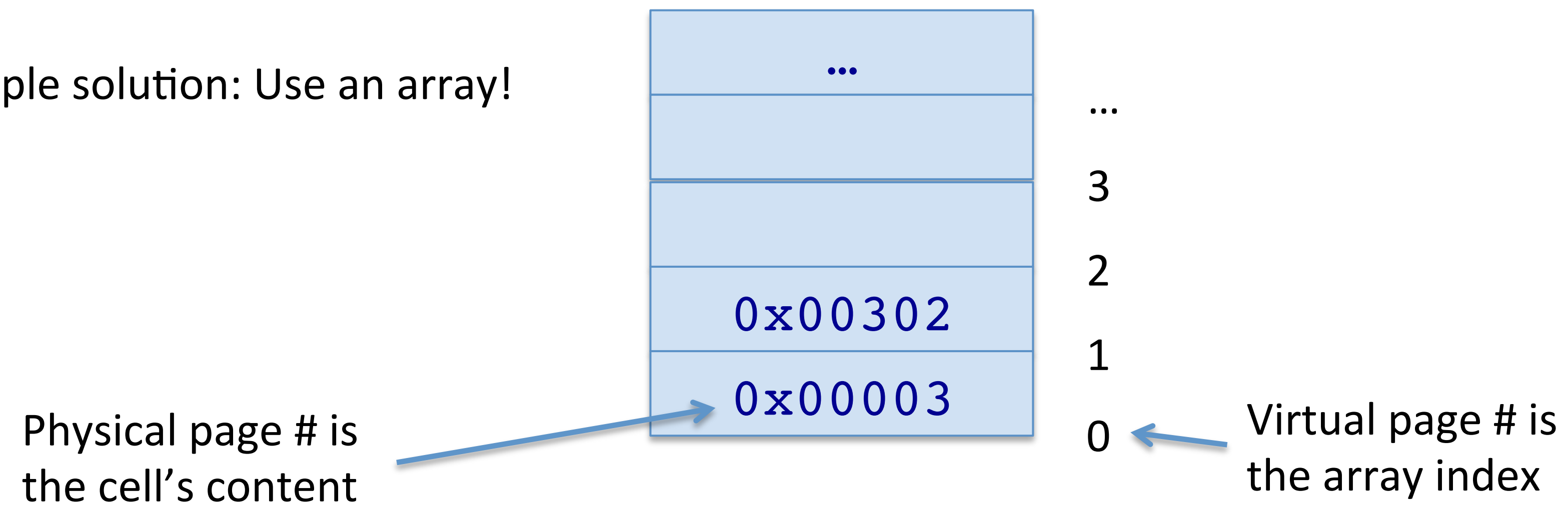
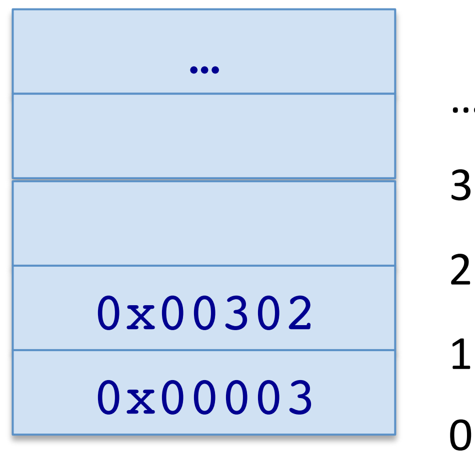


Table-Based Address Translation

How do we store the mapping from virtual page #'s to physical page numbers?

Virtual address

0x00001234

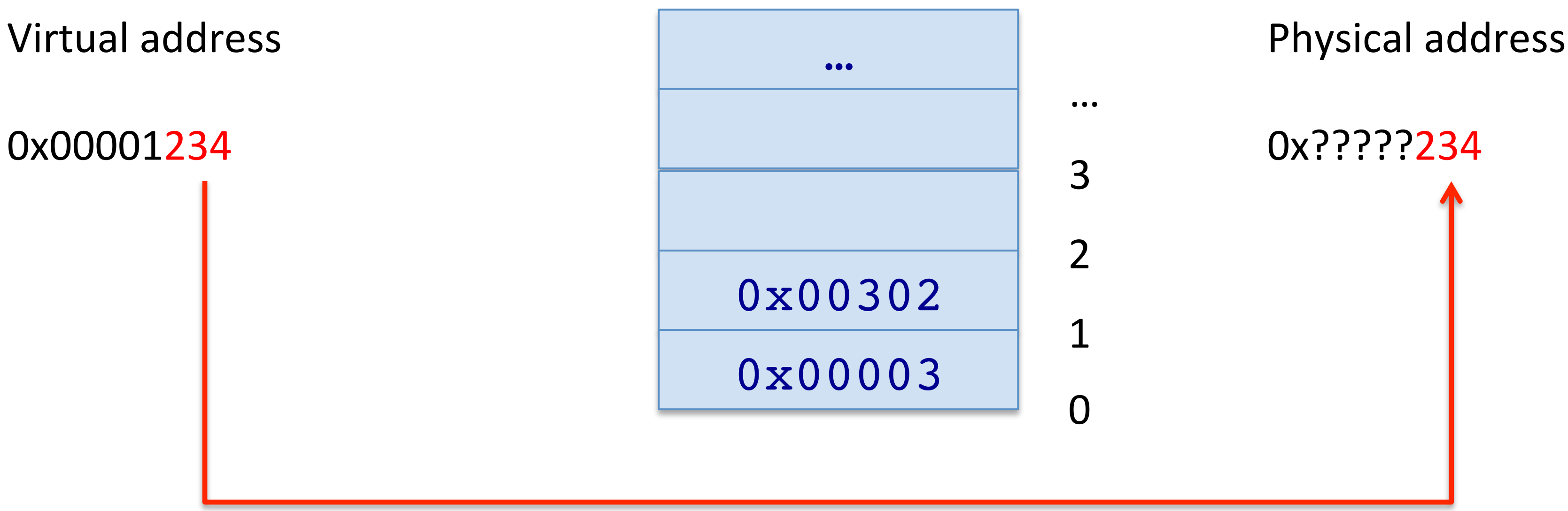


Physical address

0x????????

Address Translation: Page Tables

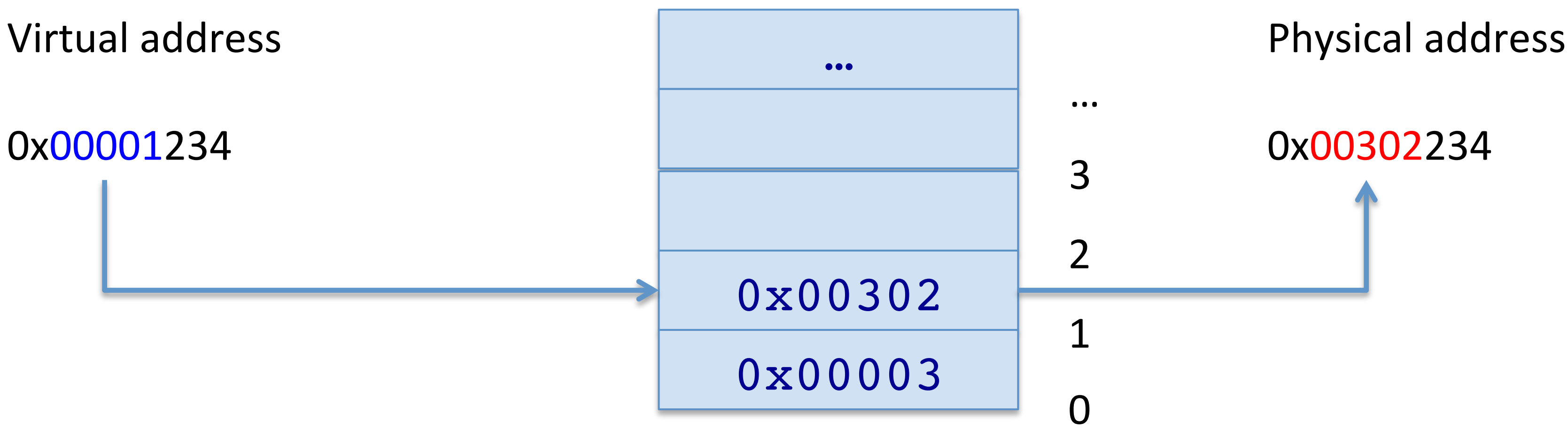
Example: Mapping virtual addresses to physical addresses



Offset is unchanged by translation

Table-Based Address Translation

Example: Mapping virtual addresses to physical addresses



Page number is resolved by table lookup

Table-Based Address Translation

Question: How big does our array need to be?

(for a 32-bit address space
with 4KB pages)

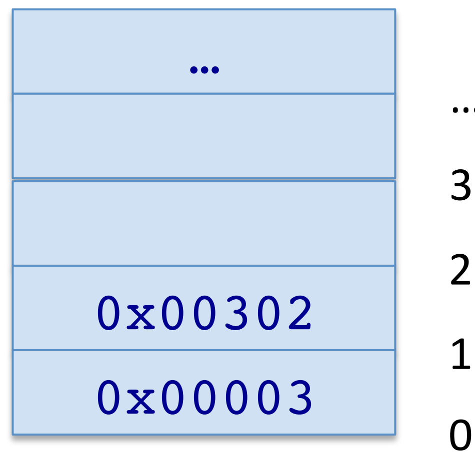


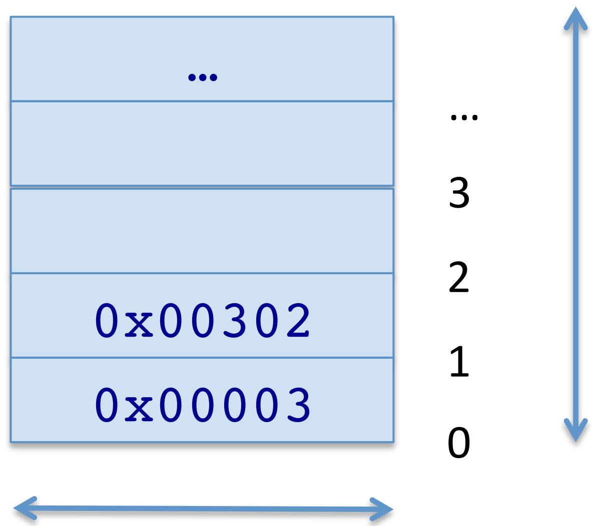
Table-Based Address Translation

Question: How big does our array need to be?

(for a 32-bit address space
with 4KB pages)

Answer: About 2.5 MB!

And we need one for
every process! Ouch!



Need 2^{20} (~1 million) entries to provide mappings for all virtual pages

Each entry is 20 bits wide

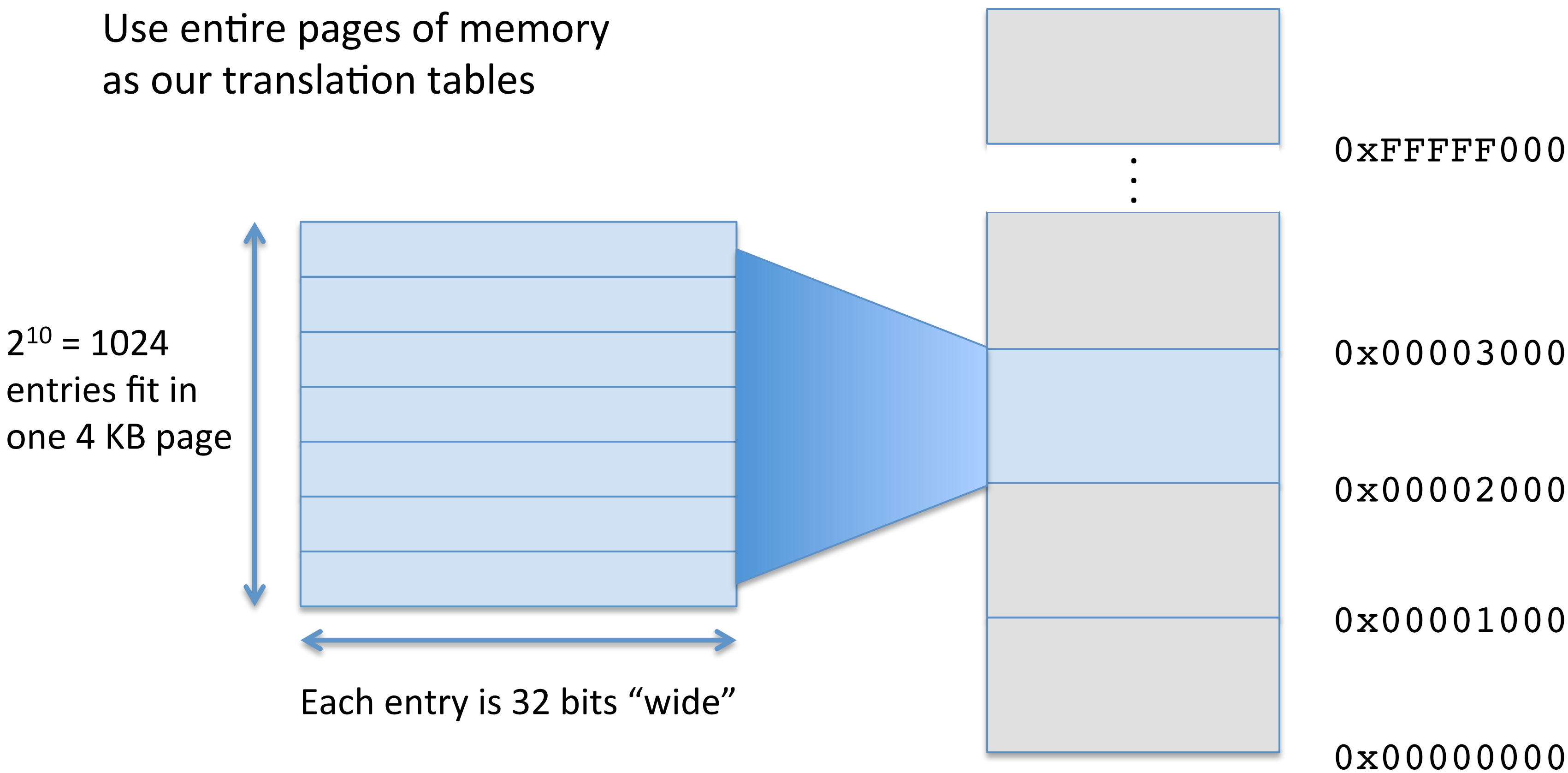
Virtual Memory with Page Tables

- Implement table-based address resolution

- Reduce memory requirements by adding another layer of indirection

Page Tables

Use entire pages of memory as our translation tables

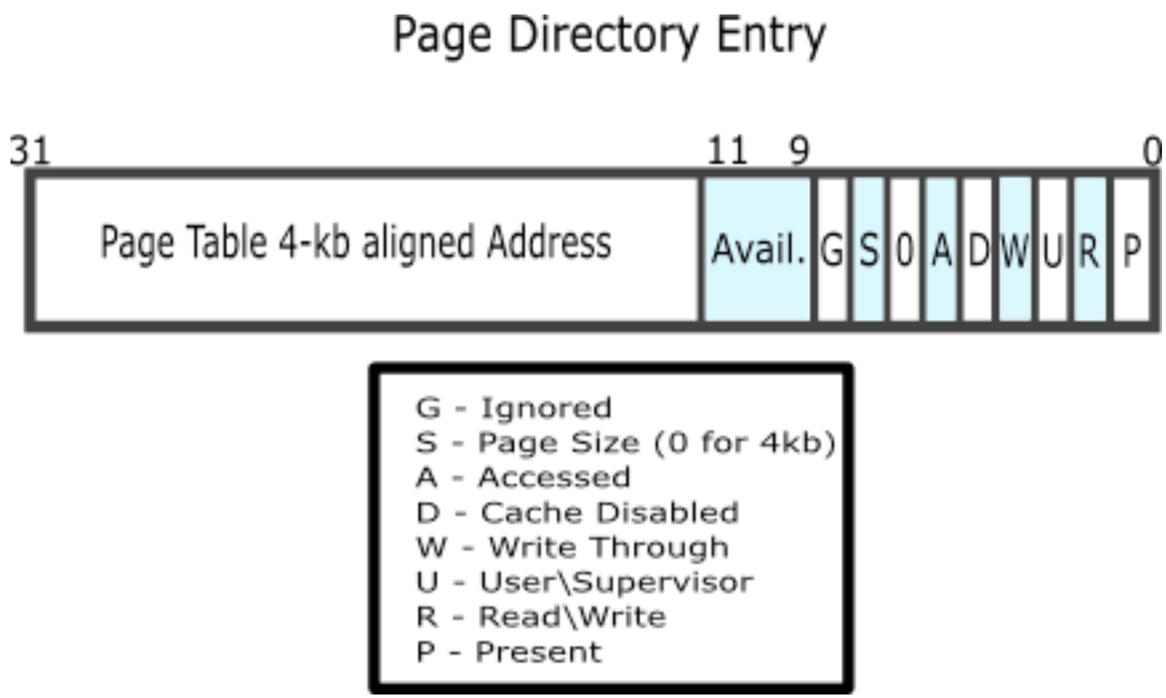


Page Table Entry Structure

20 bits: physical page number | 12 bits: bookkeeping

Bookkeeping:
0/1 – Valid? – Is this a valid entry? Or is this entry empty?
0/1 – “Dirty” bit – Has the page been modified since it was last saved to disk?
0/1 – Privileged? – Allow access to all programs, or only to privileged?
0/1 – Write – Allow writes to this page?
0/1 – Execute – Allow executing this page as code? **New! Data Execution Prevention (DEP)**
(Intel: Execute Disable (ED) AMD: No Execute (NX))

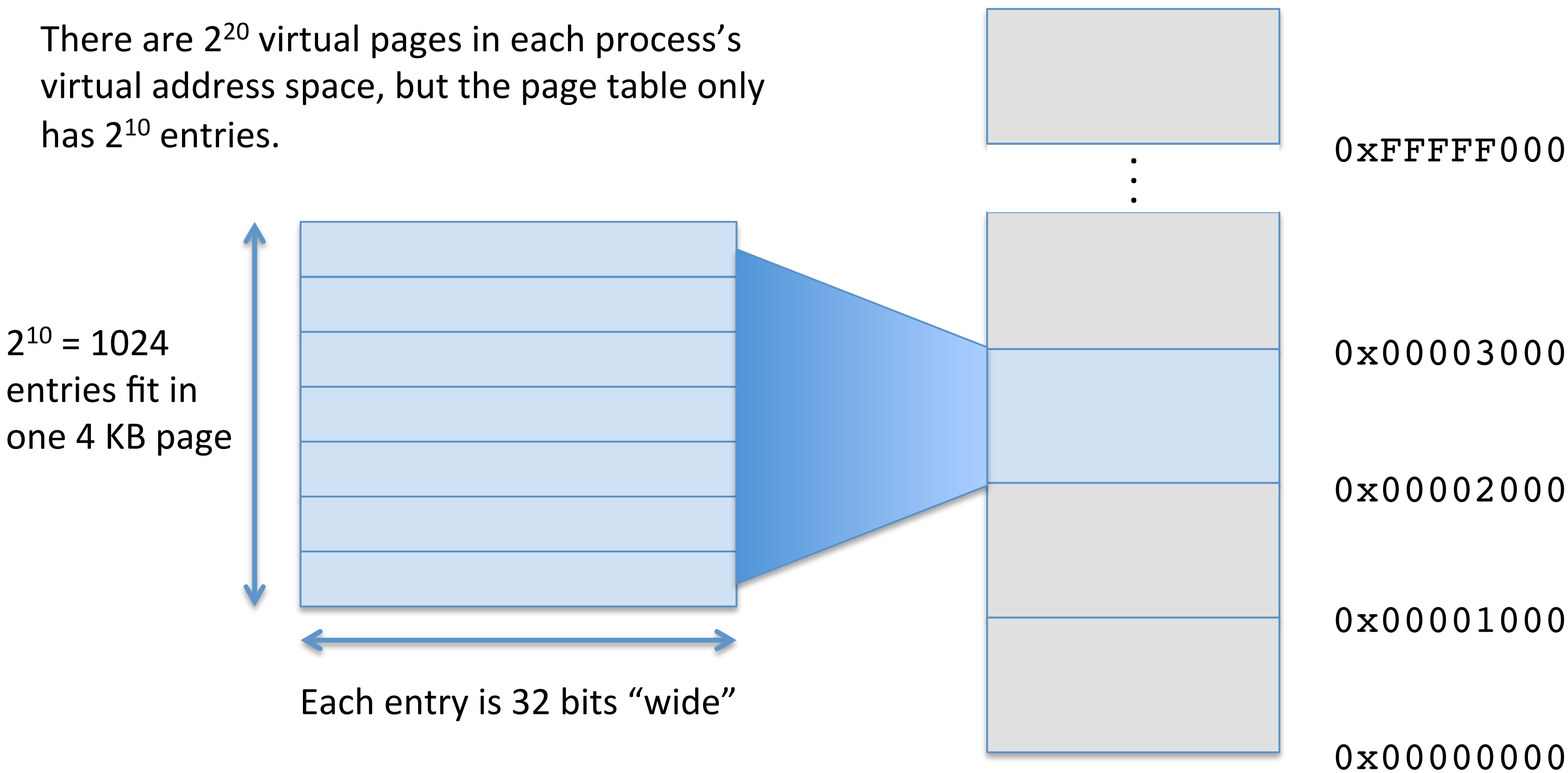
Page Table Entries on x86



Credit: http://wiki.osdev.org/Paging

A problem with our page tables?

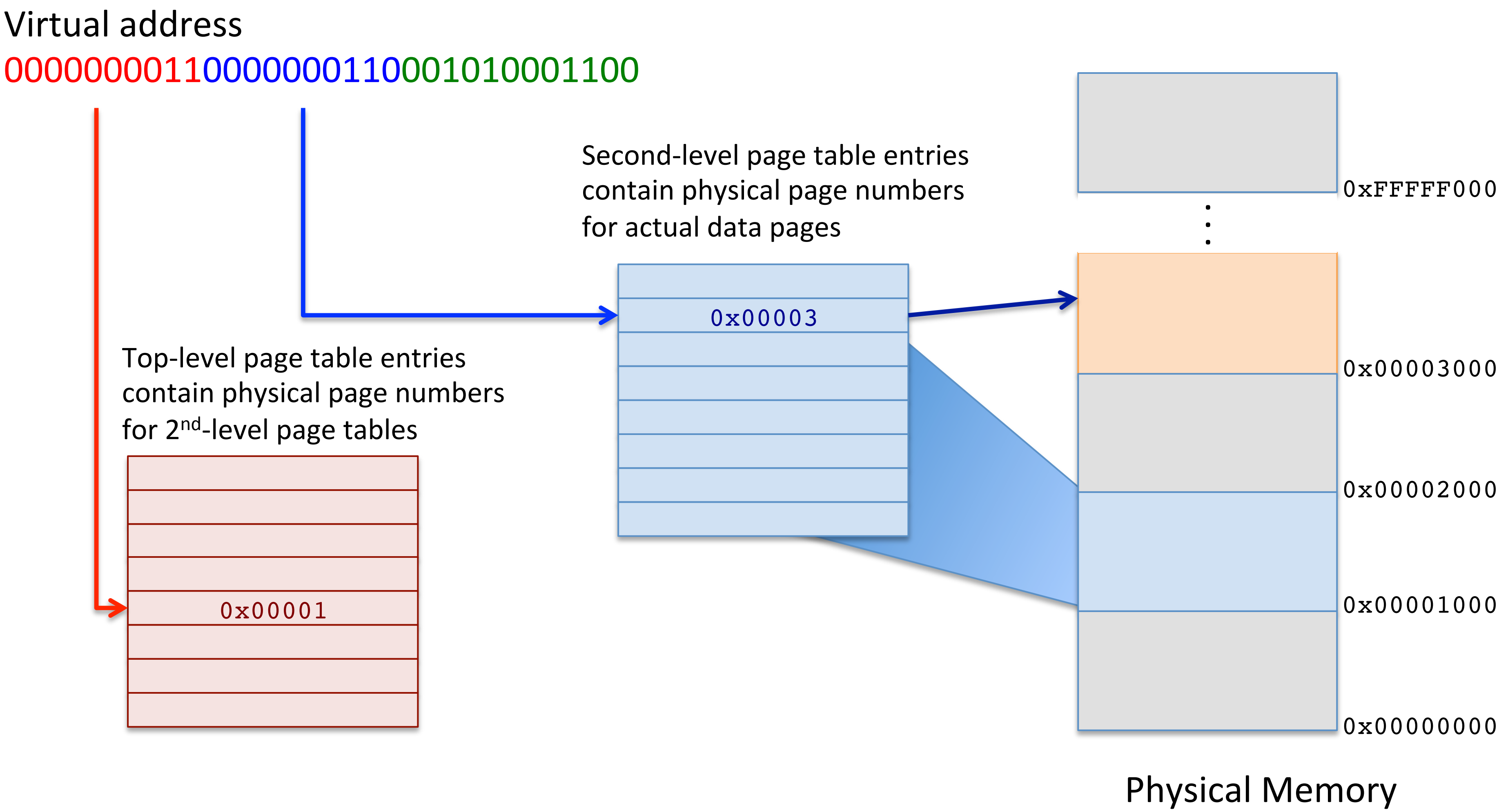
There are 2^{20} virtual pages in each process's virtual address space, but the page table only has 2^{10} entries.



Solution: Multi-level Page Tables

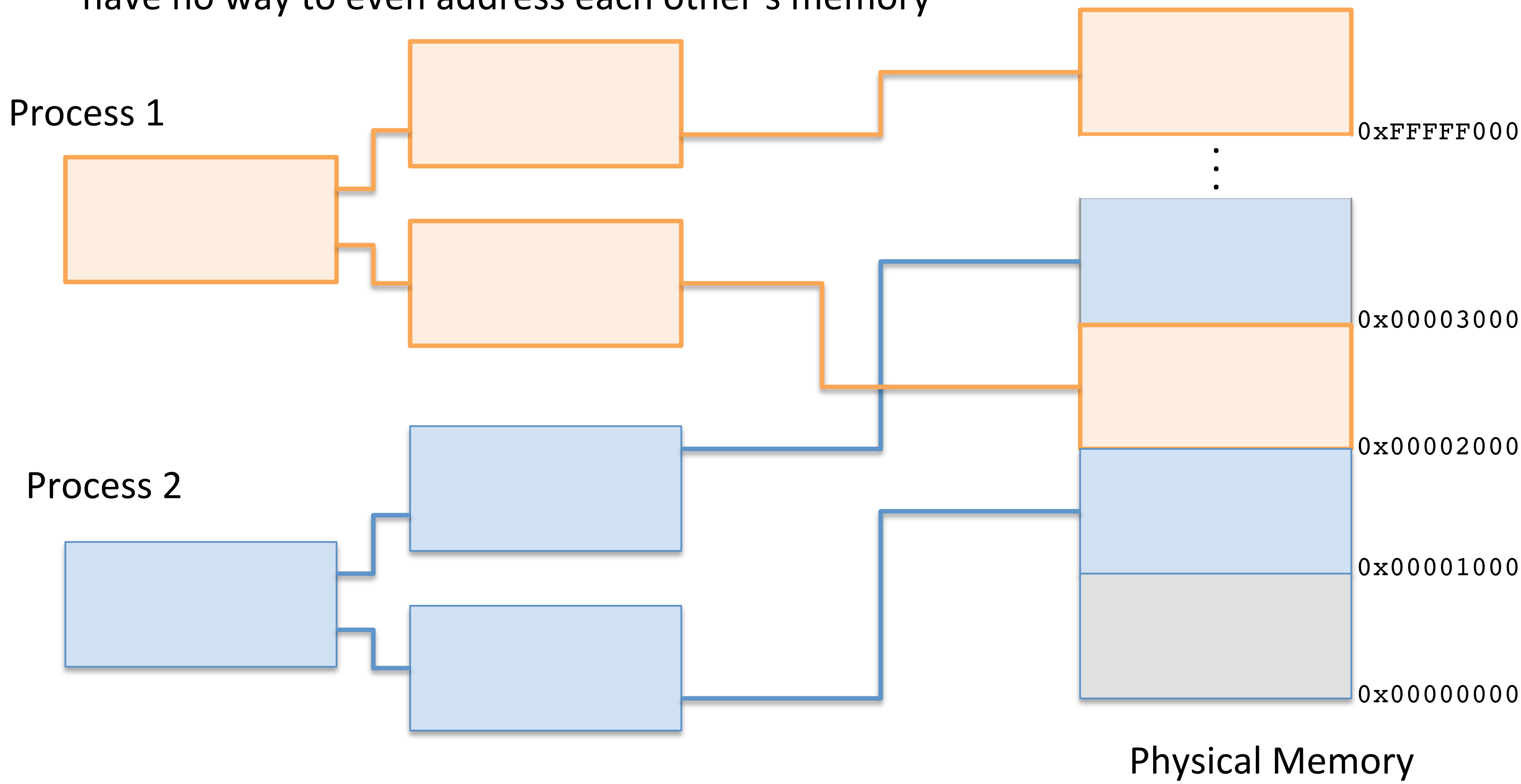
Virtual address
00000000110000000110001010001100

Solution: Multi-level Page Tables



Protection of Programs' Memory

Without the proper page table entries, Process 1 and Process 2 have no way to even address each other's memory



Shared Libraries

To conserve RAM, processes can share physical pages of library code by mapping it read-only in the virtual address spaces of both processes