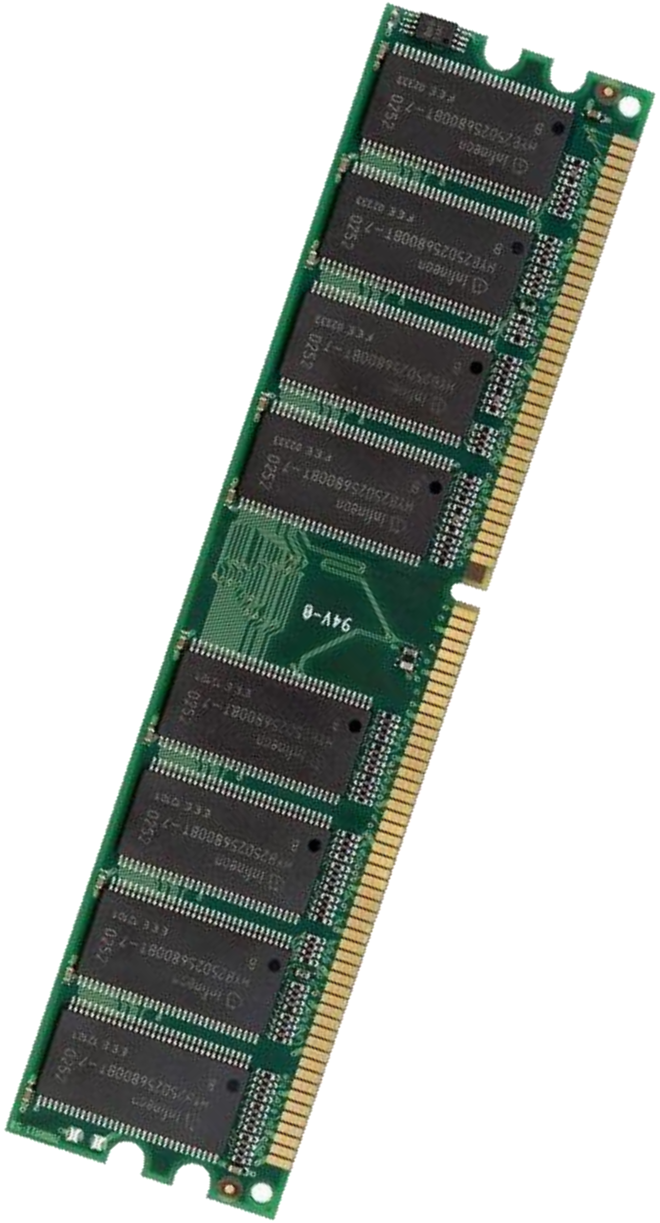
Virtual Memory

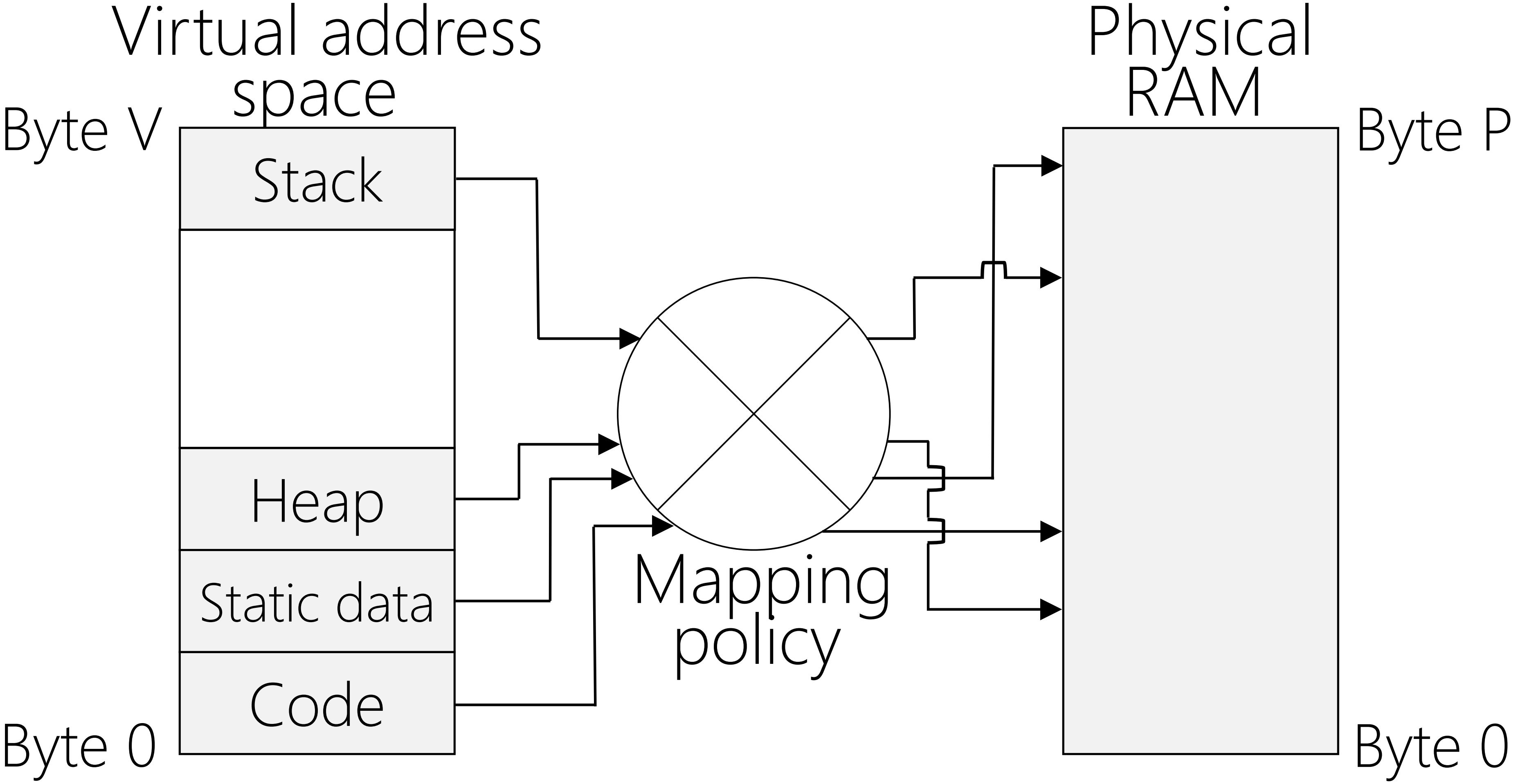
CS 161: Lecture 6

2/16/17

Assigning RAM to Processes

- Each process has an address space
  - The address space contains the process’s code, data, and stack
  - Somehow, the hardware and the OS must map chunks of the virtual address space to physical RAM





Virtual address space

Byte V

Stack

Heap

Static data

Code

Byte 0

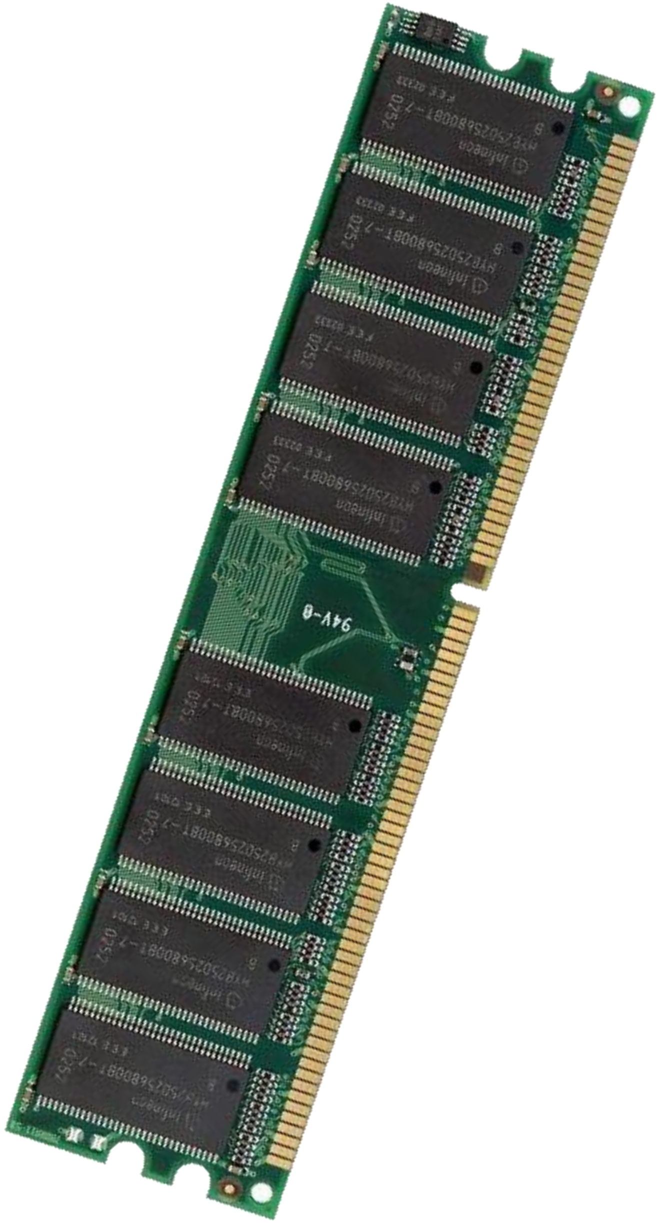
Mapping policy

Physical RAM

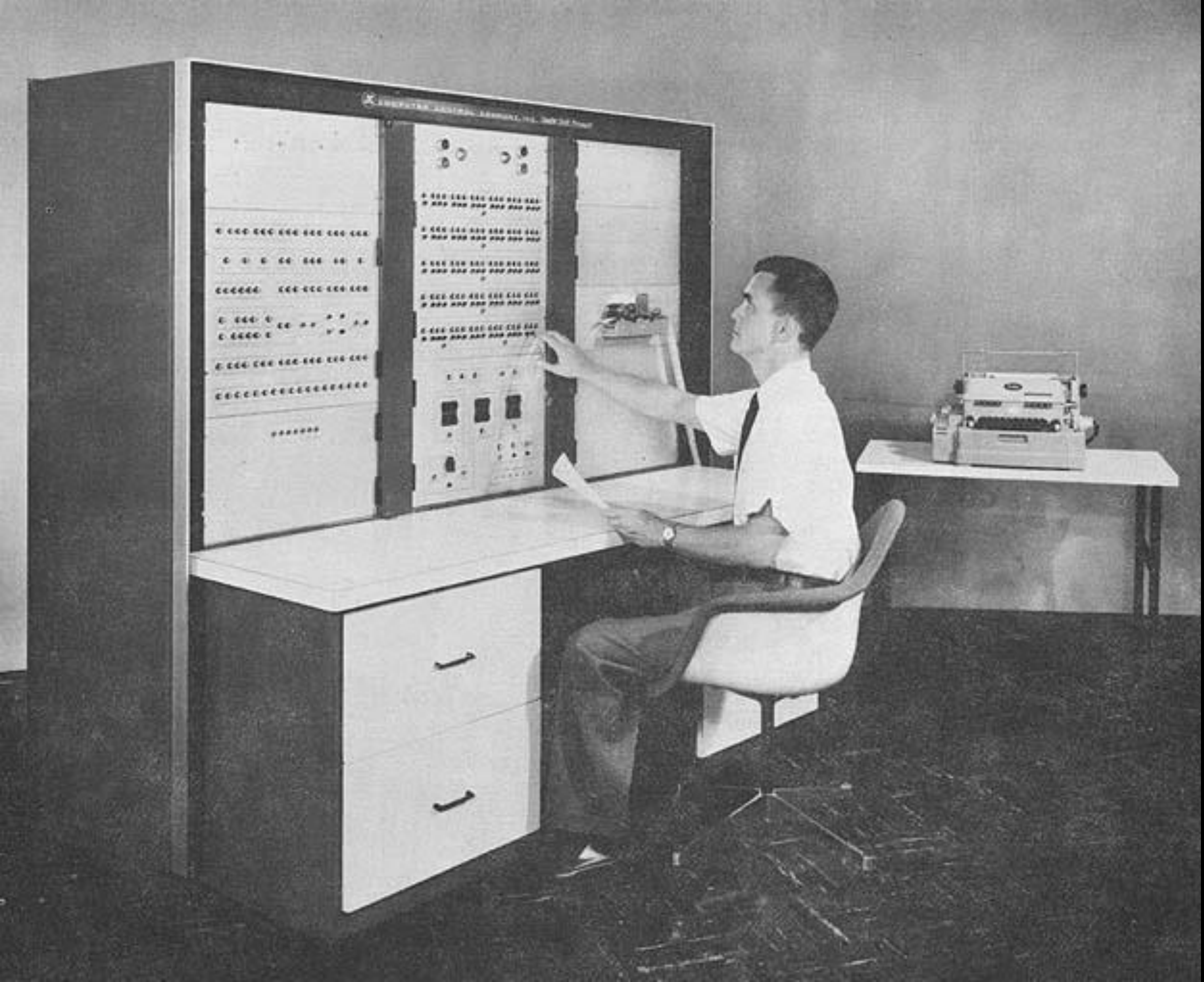
Byte P

Byte 0

Challenges of Managing RAM



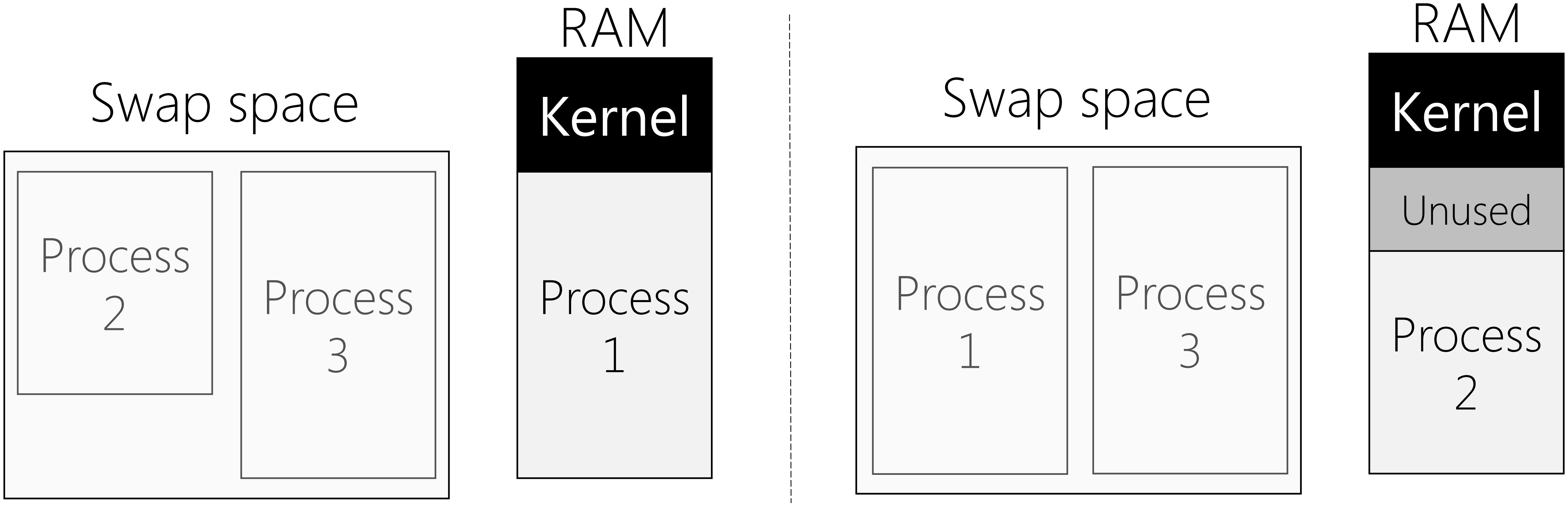
- Oversubscription: A machine has a finite amount of physical RAM, but multiple processes must use that RAM—aggregate size of all address spaces may be larger than the amount of physical RAM!
- Isolation: The OS must prevent different, untrusting processes from tampering with each other's address spaces
- Constrained sharing: In certain cases, processes may want to share RAM pages, e.g.,
  - Shared libraries like libc
  - Shared memory pages to facilitate IPC



THE OLDEN DAYS

Batch Processing

- In olden times, only one process could run at any given moment
  - The entire address space was moved into and out of memory at once
  - Swap space: the persistent storage that held address spaces not in RAM
- Hardware prevented user code from accessing OS memory (which was assumed to live in a certain region of physical RAM)



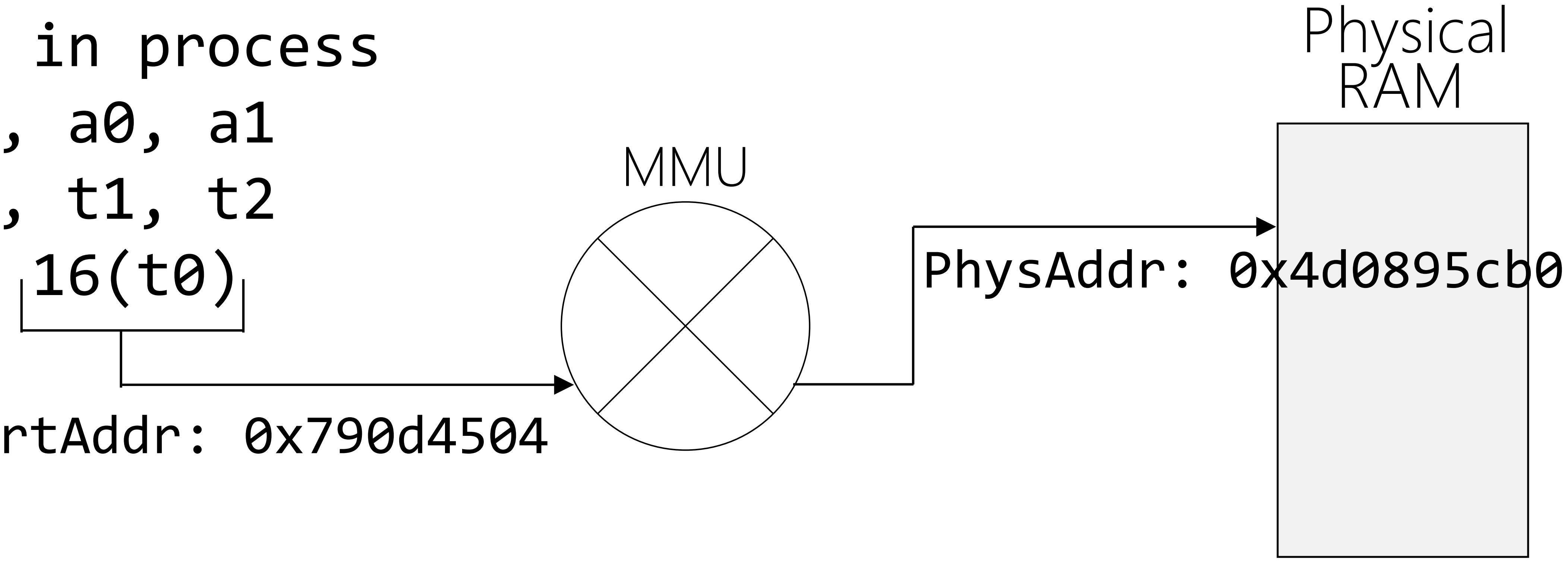
Batch Processing

- Advantages
  - Simple
  - Supports process isolation
  - Cheaper than two computers LOLOL
- Disadvantages
  - An address space could be no larger than the physical RAM . . .
  - . . . but several small address spaces could not be co-located in RAM
  - Context switching is slow: an entire address space must be swapped out, and an entire address space must be swapped in
  - No way for two processes to share RAM

Memory-mapping Units (MMUs)

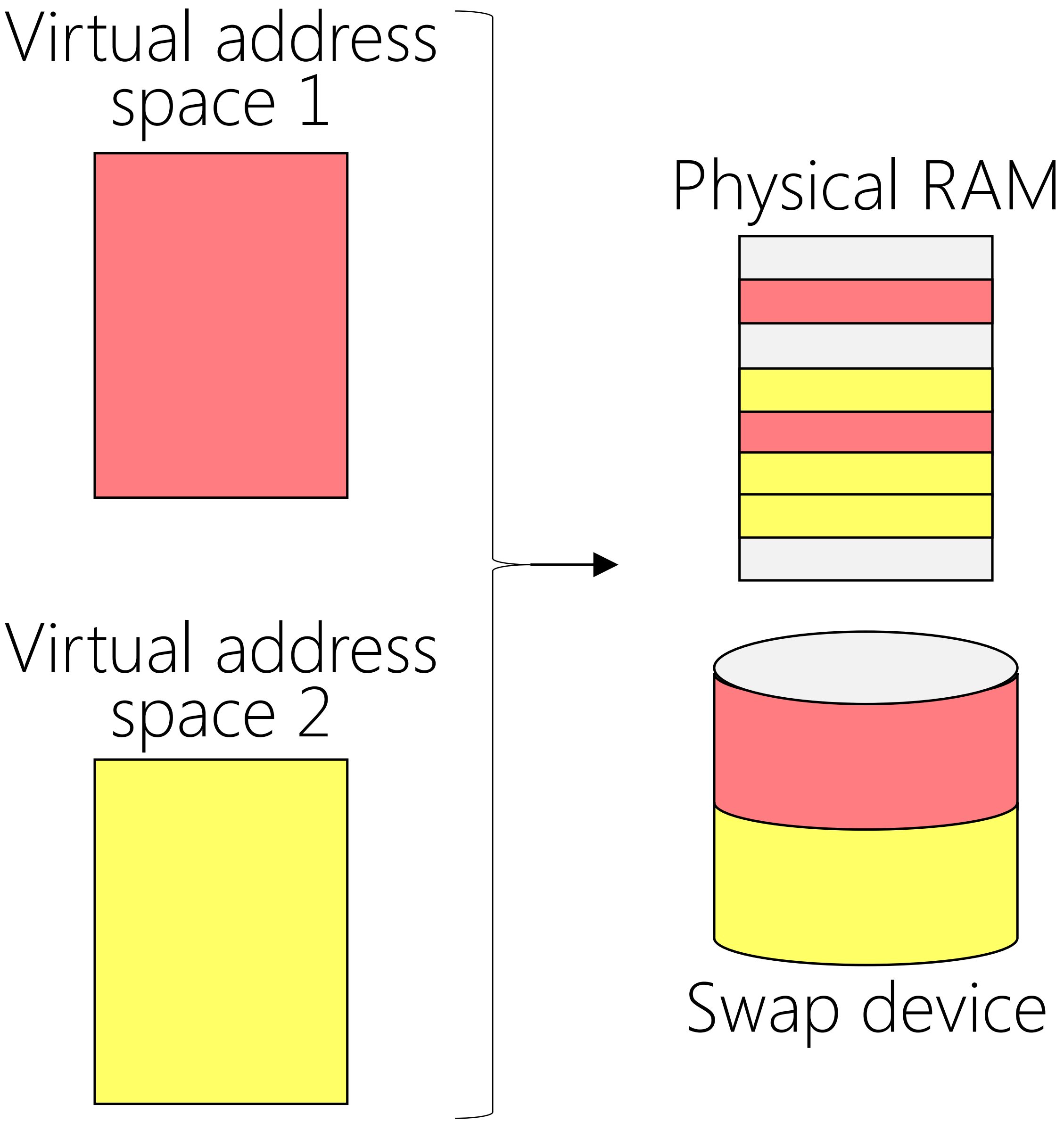
- MMU: A piece of hardware (only configurable by privileged code) that translates virtual addresses to physical addresses
  - Virtual addresses are the addresses that a process generates
  - Physical addresses are what a processor presents to the actual RAM

```
//Code in process
sub t1, a0, a1
add t0, t1, t2
lw t4, 16(t0)
```



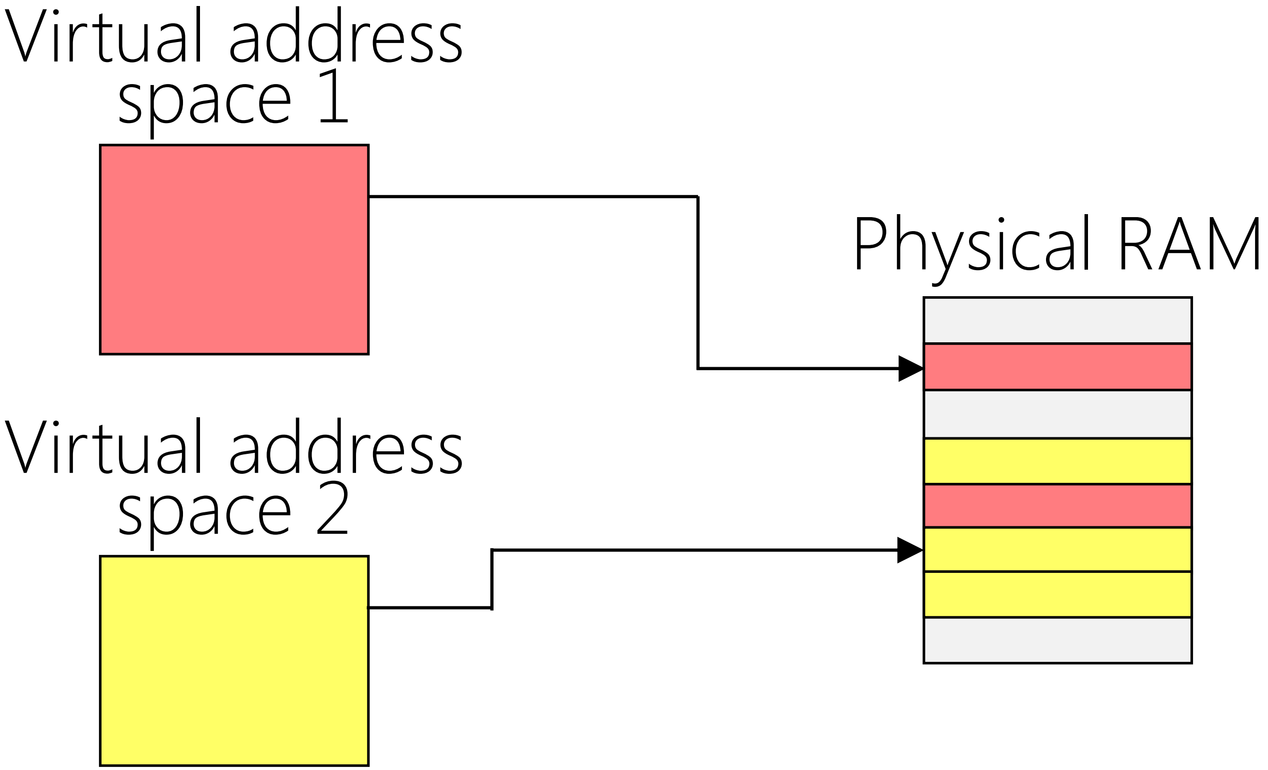
Memory-mapping Units (MMUs)

- Using indirection via the MMU, we want to allow:
  - Over-subscription of physical RAM: at any given time, some/none/all of a virtual address space can be mapped into physical RAM
  - Virtual address spaces to be bigger than physical RAM (and vice versa)
  - Faster context switches: after a context switch to process P, we can lazily bring in P's non-resident memory regions, as P tries to access them

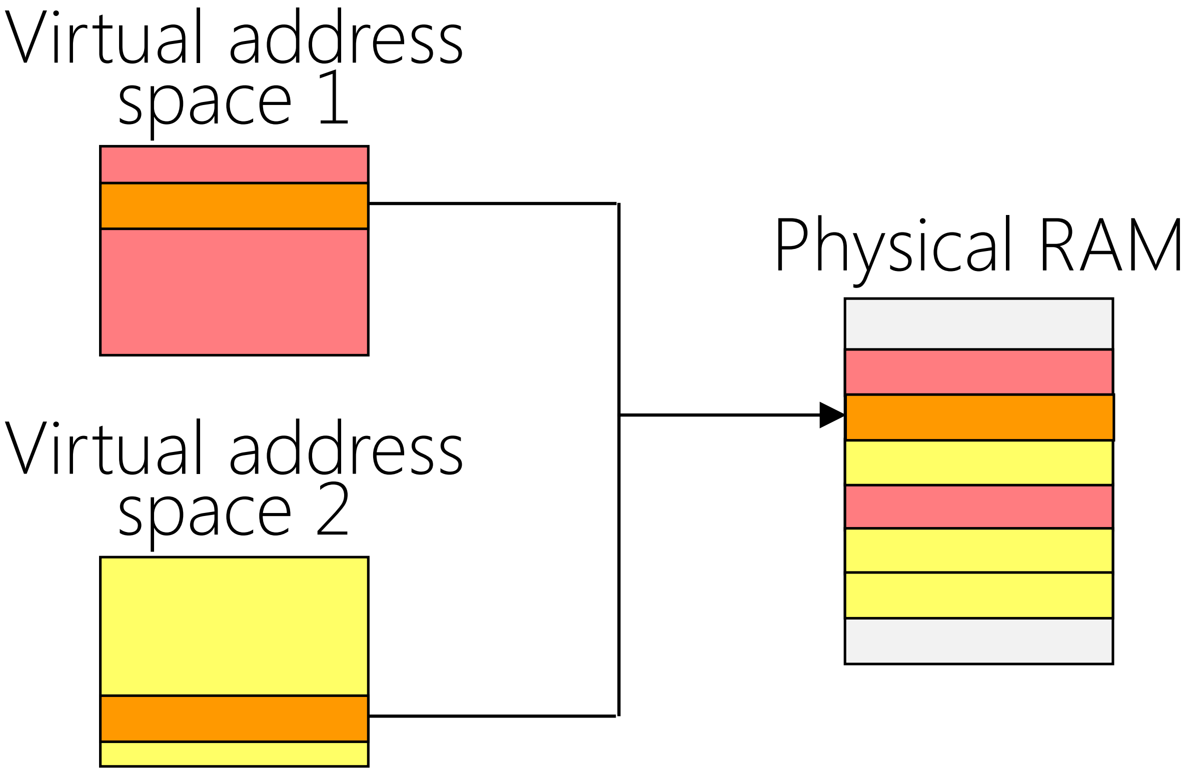


Memory-mapping Units (MMUs)

- Using indirection via the MMU, we want to allow:
  - Protection: the hardware maps the same virtual address in two different processes to different physical addresses



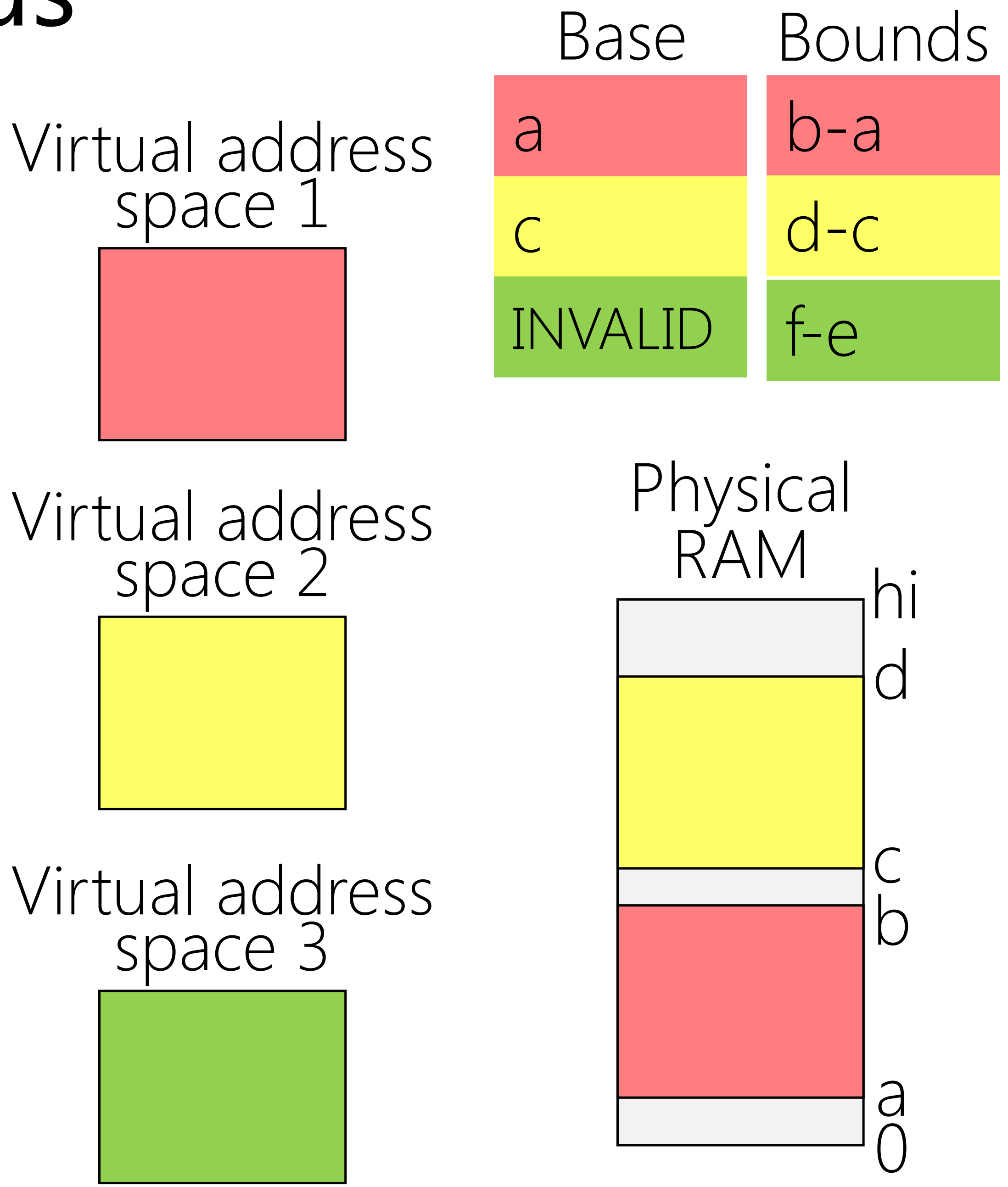
  - Sharing: hardware maps a single region of physical RAM into multiple virtual address spaces



Initial Attempt: Base+Bounds

- Associate each address space with base+bound registers
  - Base register: Contains the physical address where the address space starts (or “invalid” if the address space is not mapped into physical memory)
  - Bound register: Contains the length of the address space in both virtual and physical memory
- Memory translation uses this formula:

```
if(virt_addr > bounds){
    error();
}else{
    phys_addr = base + virt_addr;
}
```



Base	Bounds
a	b-a
c	d-c
INVALID	f-e

Base+Bounds: Pros and Cons

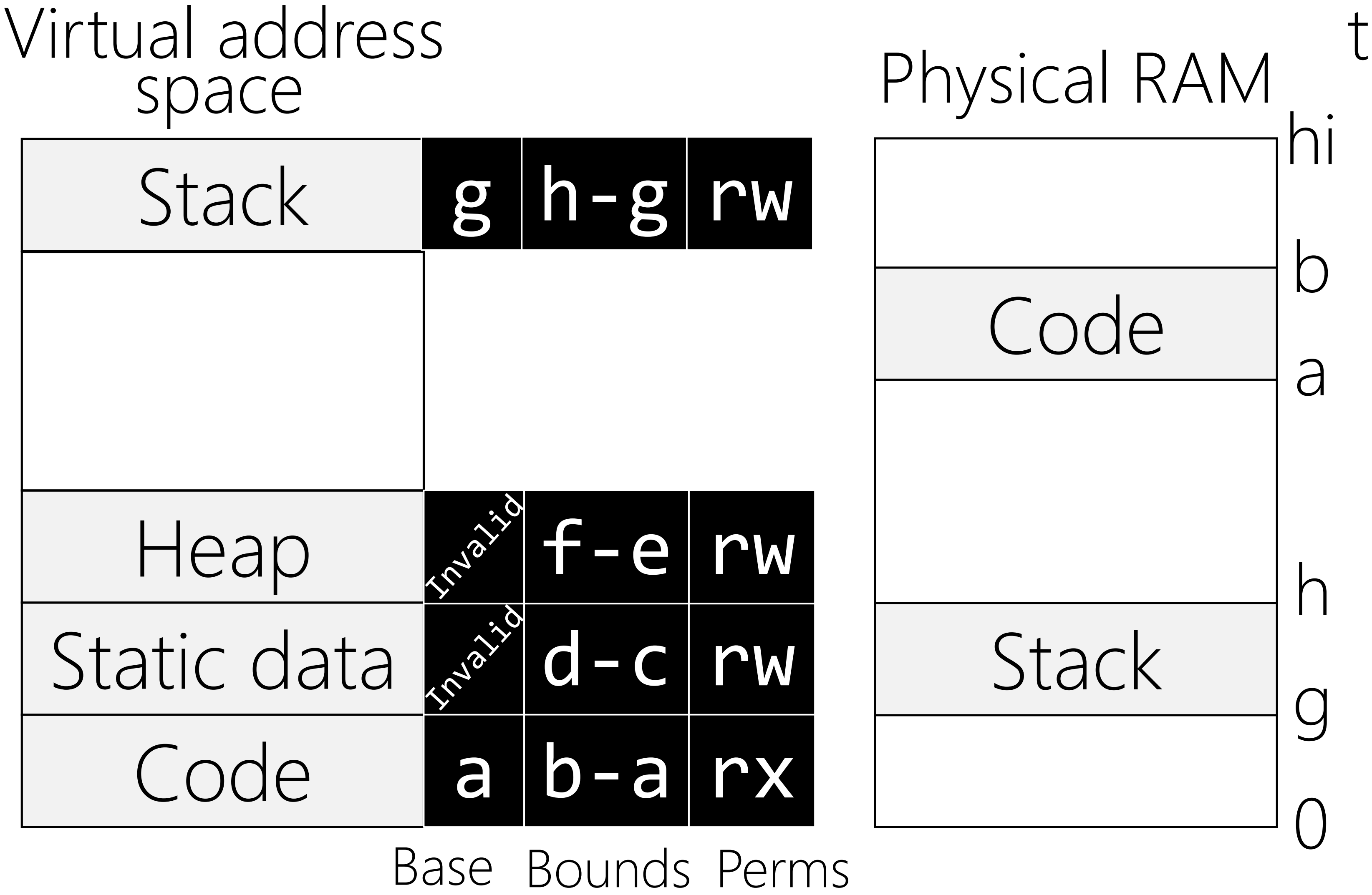
- Advantages
  - Allows each virtual address space to be a different size
  - Allows a virtual address space to be mapped into any physical RAM space of sufficient size
  - Isolation is straightforward: Just ensure that different address spaces don’t have overlapping base+bounds!
- Disadvantages
  - Wastes physical memory if the virtual address space is not completely full (which is often the case due to a hole between the stack and the heap)
  - Tricky to share physical RAM between two virtual address spaces: can only do so by having the bottom of one space overlap with the top of another
  - Have to mark the entire address space as readable+writable+executable: makes it hard to catch bugs and stop attacks

Segmentation

- A single virtual address space has multiple logical segments
  - Code: read but non-write, executable, constant size
  - Static data: read/write, non-executable, constant size
  - Heap: read/write, non-executable[†], dynamic size
  - Stack: read/write, non-executable, dynamic size
- Associate each *segment* with base+bound+protection flags (read/write/execute)
  - At any given moment, some/all/none of the segments can be mapped into physical RAM

[†] Unless a process is performing just-in-time (JIT) compilation!

Segmentation



	Base	Bounds	Perms
Stack	g	h-g	rw
Heap	Invalid	f-e	rw
Static data	Invalid	d-c	rw
Code	a	b-a	rx

Advantages with respect to vanilla base+bounds:

- Segmentation allows the OS to explicitly model sparse address spaces which contain unused regions
- Segmentation also allows the OS to associate different protections (read/write/execute) with different regions

Segmentation

- Address translation uses this formula:

```
seg = find_seg(virt_addr);
if(offset(virt_addr) > seg.bounds){
   error();
}else{
   phys_addr = seg.base + offset(virt_addr);
}
```

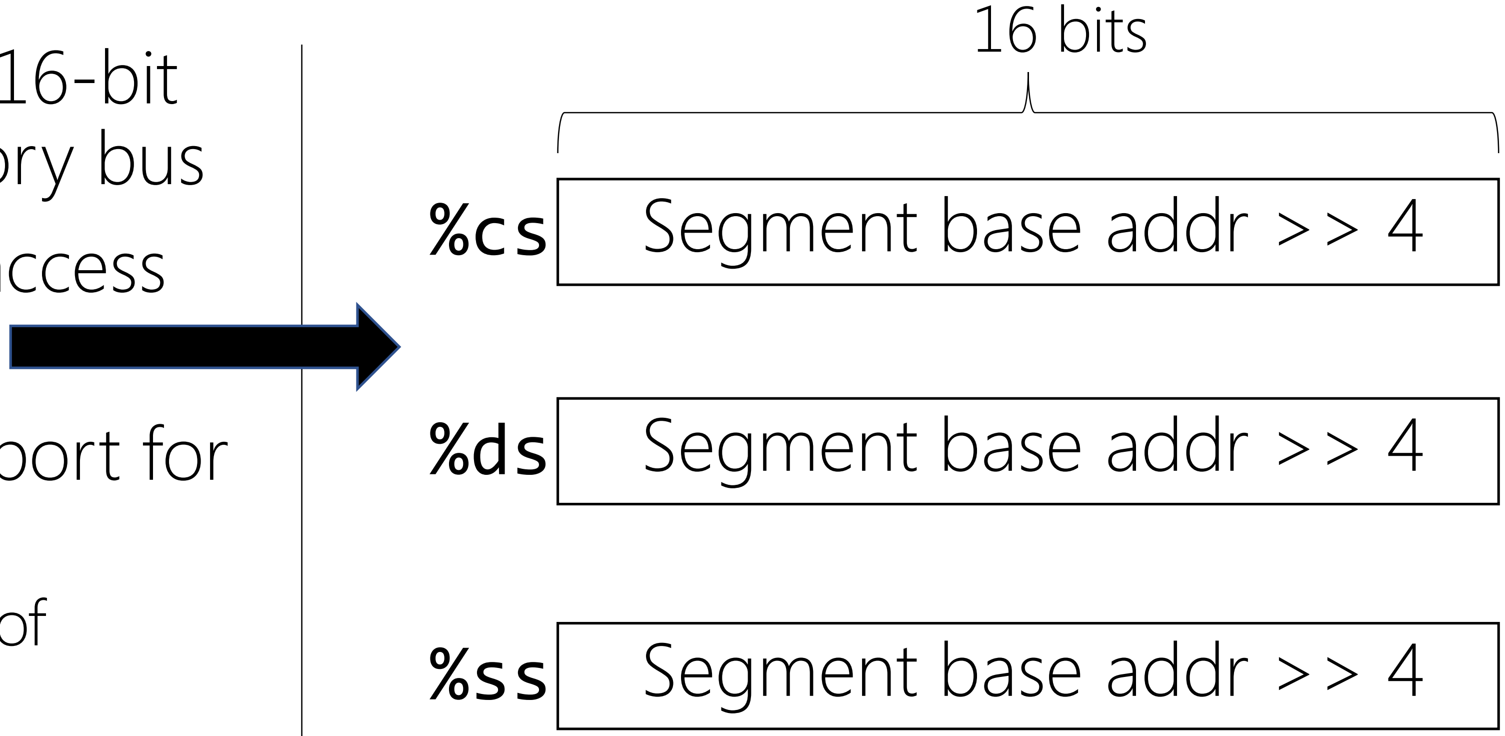
- How do we define **`find_seg(virt_addr)`** and **`offset(virt_addr)`**?
  - Partition approach: Use the high-order bits of **`virt_addr`** to select the segment, and the low-order bits to define the offset
  - Explicit approach: Use **`virt_addr`** as the offset, but force instructions to explicitly define which segments should be used

```
mov 0x42, %ds:16(%eax) //Move the constant 0x42 to
                       //offset %eax+16 in segment %ds
```

```
//Suppose find_seg(virt_addr) and offset(virt_addr) are implicitly
//determined by the instruction type. This scheme is used by x86:
//  cs: code segment (used by control flow instructions, e.g., branches)
//  ss: stack segment (used by push, pop)
//  ds: data segment (used by mov)
//Code directly assigns to ss and ds segment registers using
//instructions like mov; cs changed via branch instructions like jmp

mov %eax, 4(%ebx) //*(%ebx+4) = %eax
                  //   offset(virt_addr) = virt_addr = %ebx+4
                  //   segment = ds

push %eax //*(--%esp) = %eax
          //   offset(virt_addr) = virt_addr = --%esp
          //   segment = ss

jmp 0x64  //%eip = %eip + bytes_in_instr(“jmp”) + 0x64  <---|
          //   offset(virt_addr) = virt_addr = -------------|
          //   segment = cs
```

x86: Real Mode Addressing in the 8086

- Intel’s 8086 chip (1978) had 16-bit registers but a 20-pin memory bus
- Segments allowed code to access 2^20 bytes of physical RAM
- Real mode provided no support for privilege levels
    - All code can access any part of memory
    - All code can execute any instruction
- Even modern x86 chips start execution in real mode: backwards compatibility!



16 bits

%cs Segment base addr >> 4

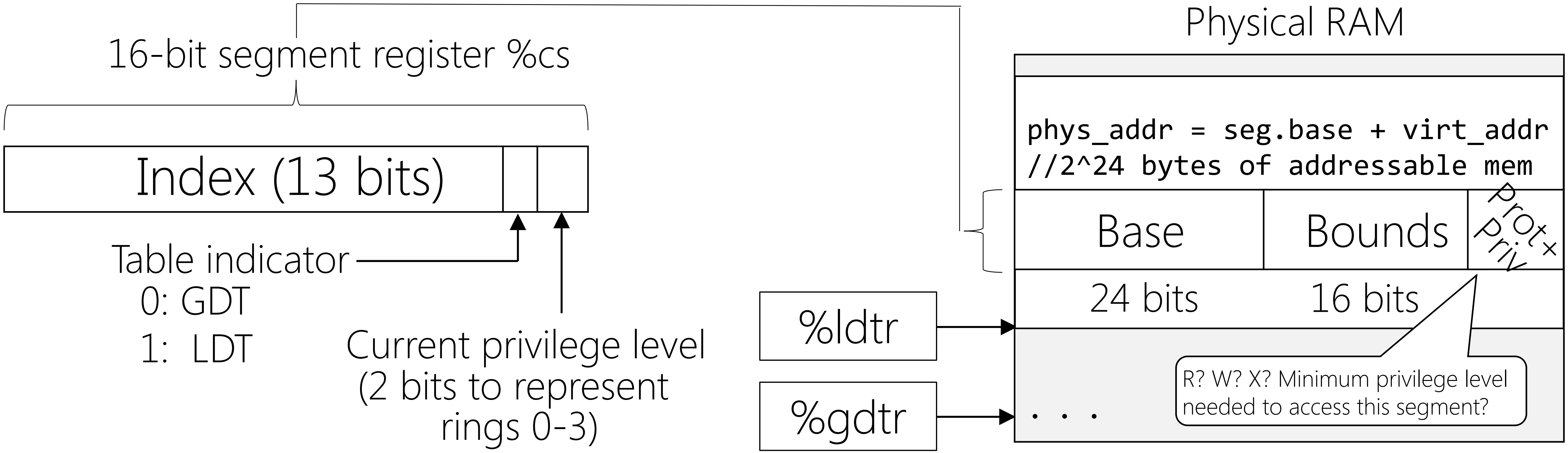
%ds Segment base addr >> 4

%ss Segment base addr >> 4

```
//Hardware forces all segments to
//be 64 KB long. Given a particular
//segment, the hardware presents
//the following address to the
//memory hardware:
//    (seg.base << 4) + virt_addr
```

x86: Protected Mode in the 80286

- The 80286 (which had 16-bit registers) used segment registers like %cs to index into segment descriptor tables
    - Local Descriptor Table (LDT): Describes private, per-process segments; LDT address is pointed to by %ldtr; OS changes LDT during a context switch to a new process
    - Global Descriptor Table (GDT): Describes segments available to all processes (e.g., kernel segments); GDT address is pointed to by %gdtr; not changed on a context switch
- 80286 also added support for privilege levels and memory protections



```
phys_addr = seg.base + virt_addr
//2^24 bytes of addressable mem
```

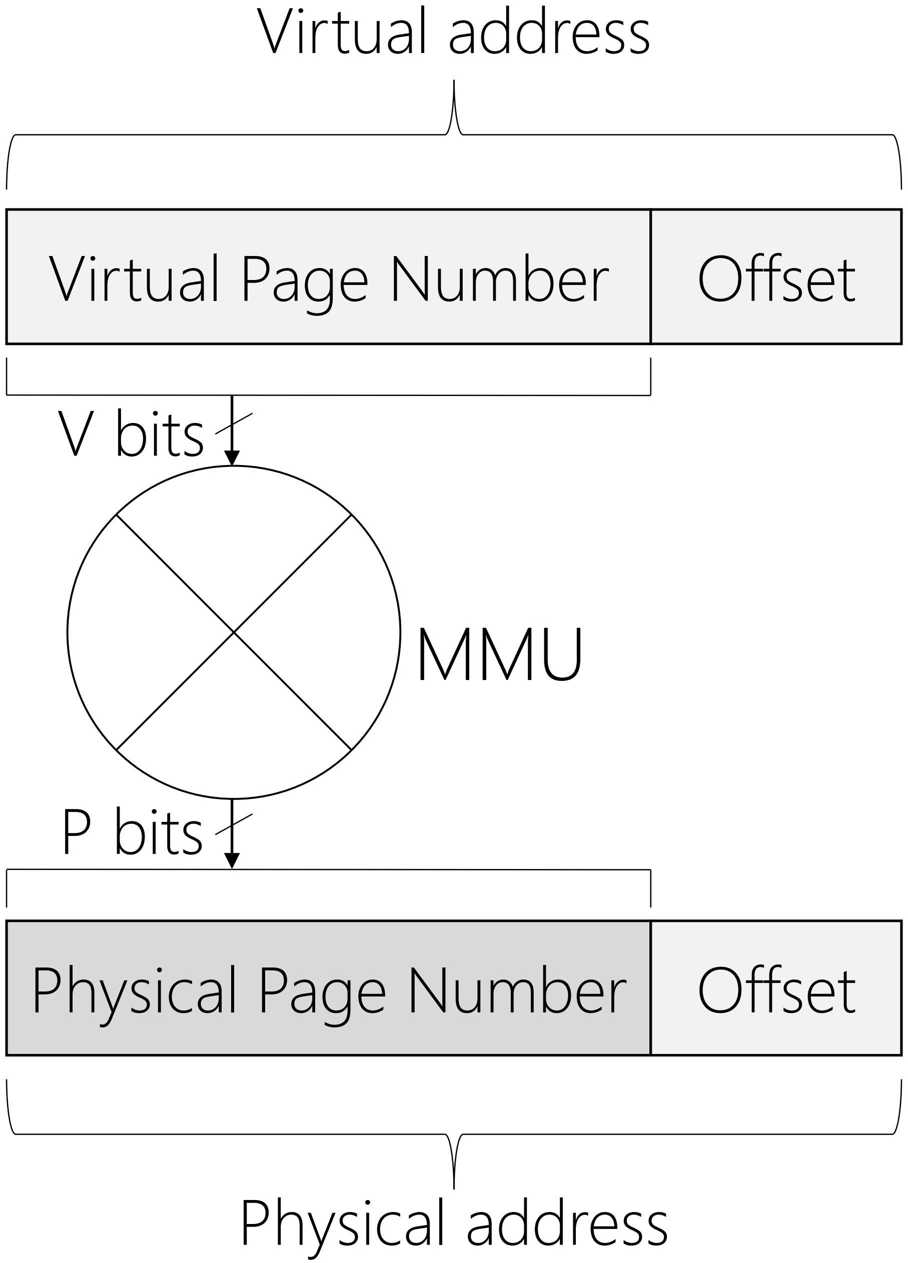
16-bit segment register %cs
Index (13 bits)
Table indicator
0: GDT
1: LDT
Current privilege level (2 bits to represent rings 0-3)
%ldtr
%gdtr
Physical RAM
Base
Bounds
Prot+Priv
24 bits
16 bits
. . .
R? W? X? Minimum privilege level needed to access this segment?

Segmentation: Advantages

- Shared advantages with vanilla base+bounds:
    - Address space metadata is small: an address spaces has few segments, and segment descriptors are just a few bytes each
    - Address space isolation is easy: don't allow the segments of the two address spaces to overlap!
    - A segment can be mapped into any sufficiently-large region of physical RAM
- Advantages over vanilla base+bounds
    - Can share physical memory between two address spaces at the segment granularity, instead of via horrible overlapping tricks
    - Wastes less memory: don't have to map the hole between the stack and the heap into physical RAM
    - Enables segment-granularity memory protections (read, write, execute, privilege mode)

Segmentation: Disadvantages

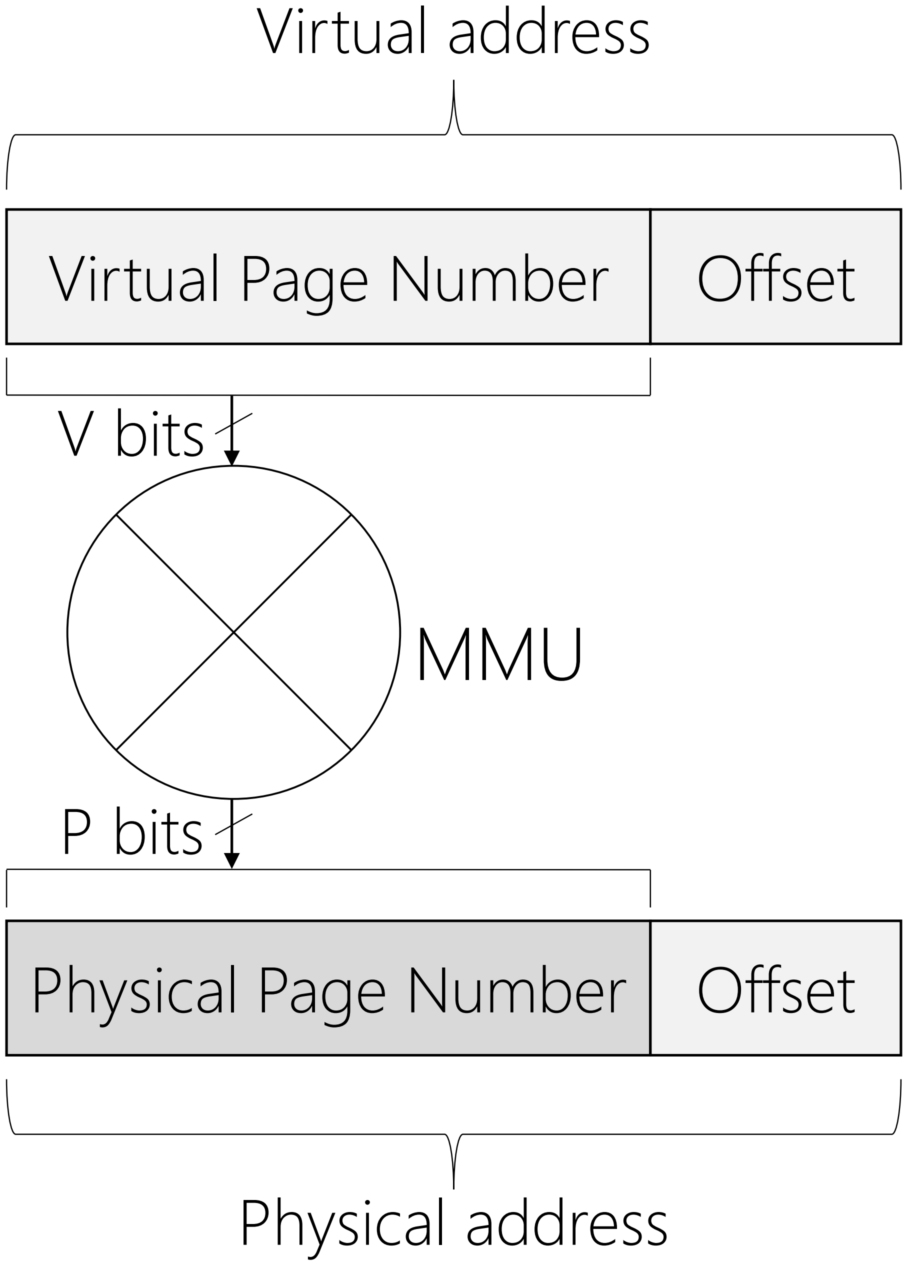
- Segments may be large!
  - If a process wants to access just one byte in a segment, the entire segment must be mapped into physical RAM
  - If a segment is not fully utilized, there is no way to deallocate the unused space—the entire region must be treated as "live"
- When mapping a segment into physical RAM, finding an appropriately-sized free region in physical RAM is irritating, since segments are variable-sized
  - First-fit, worst-fit, best-fit all have icky trade-offs between the time needed to find a free space, and the amount of wasted RAM
- Explicit segment management, e.g., `mov 0x42, %ds:16(%eax)`, is tedious



Paging

- Divide the address space into fixed-sized chunks called pages
  - No need for bounds entries, since the page size is constant
  - Each page aligned to a page-size boundary
- A “segment” is now a collection of pages
- Make each page small (e.g., 4 KB)
  - Good: Can allocate virtual address space with fine granularity
  - Good: Only need to bring the specific pages that process needs into physical RAM
  - Bad: Bookkeeping overhead increases, since there are many pages!

Paging



Virtual address

Virtual Page Number | Offset

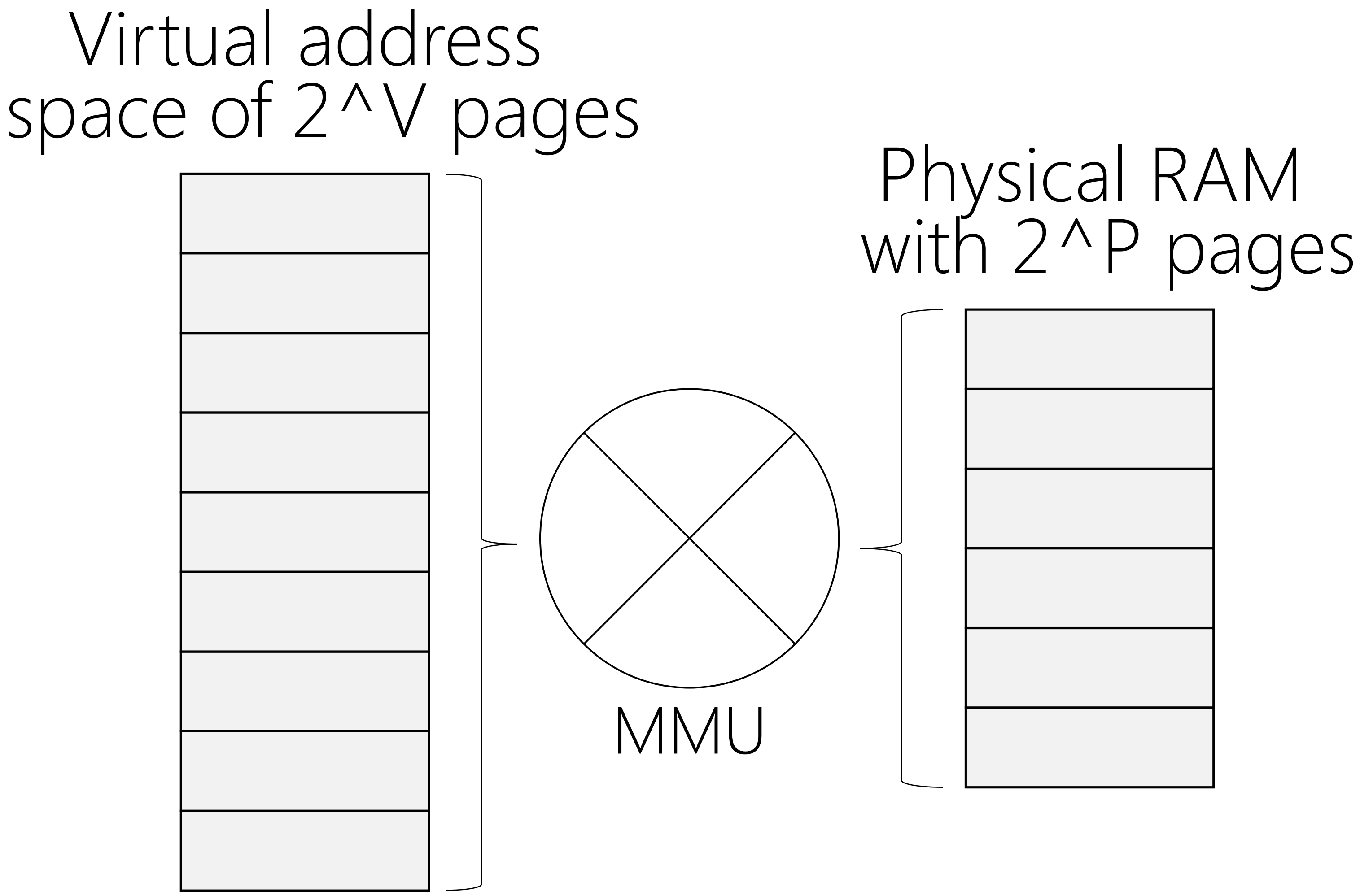
V bits

MMU

P bits

Physical Page Number | Offset

Physical address



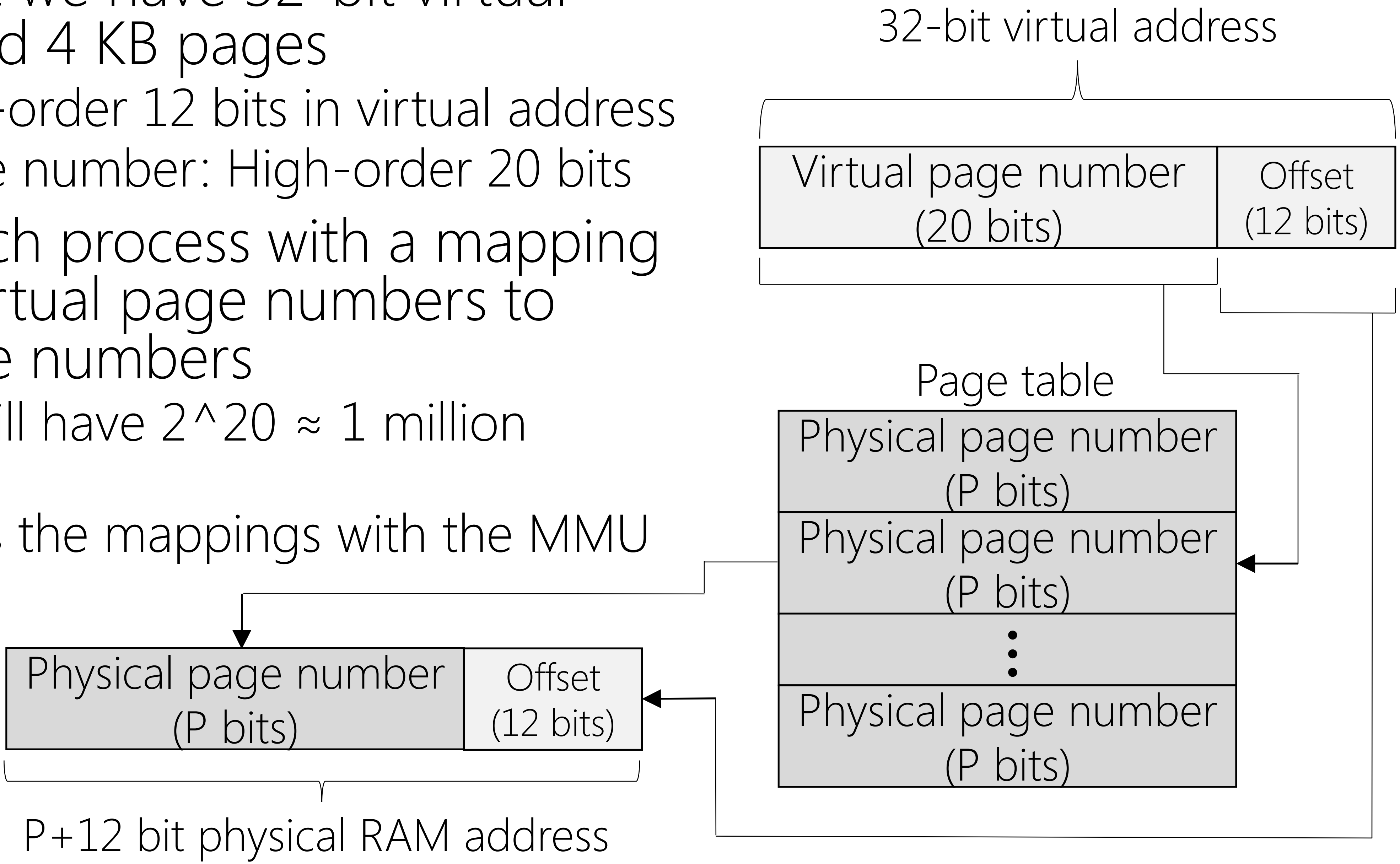
Virtual address space of 2^V pages

Physical RAM with 2^P pages

MMU

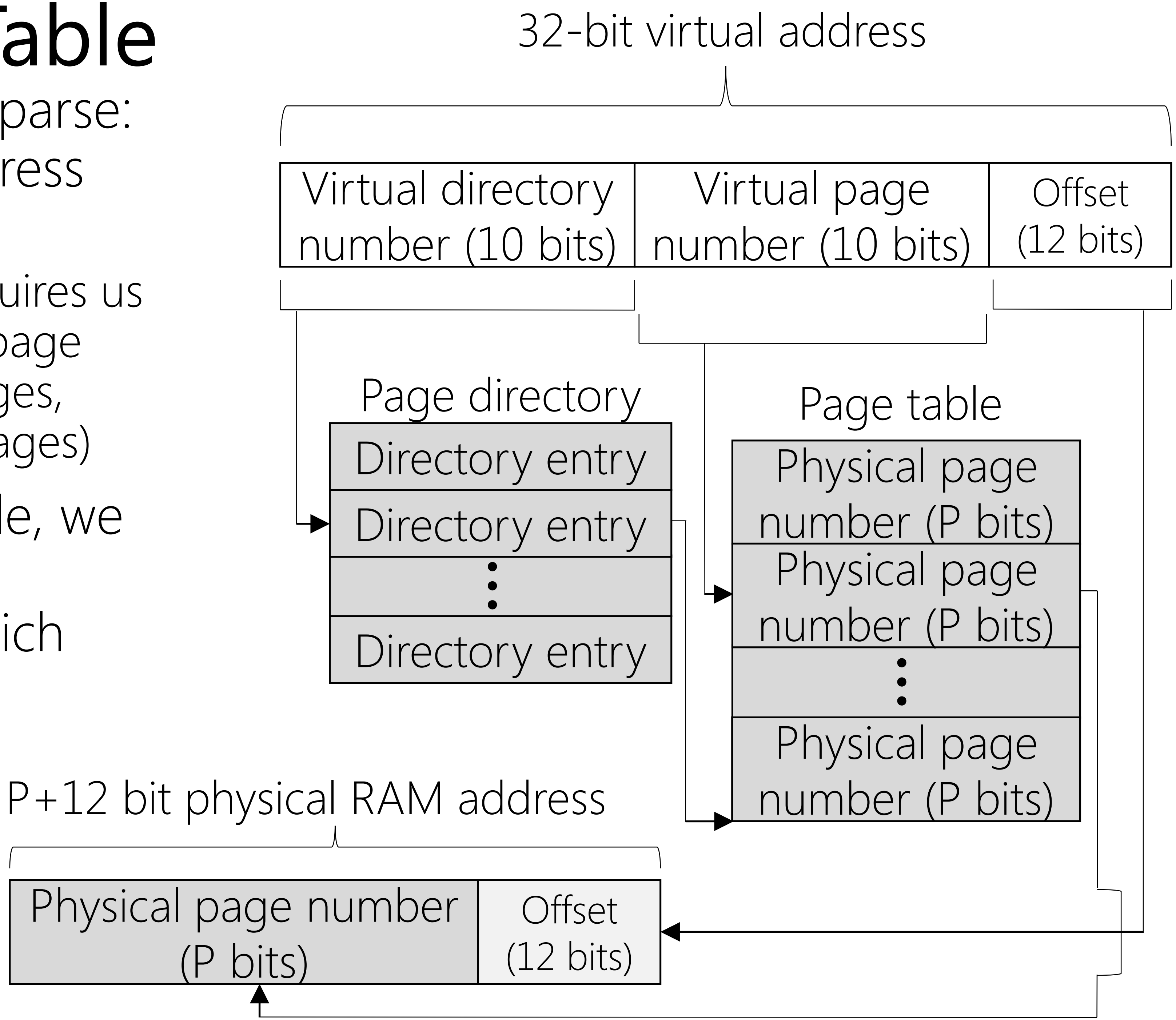
Single-level Page Table

- Suppose that we have 32-bit virtual addresses and 4 KB pages
    - Offset: Low-order 12 bits in virtual address
    - Virtual page number: High-order 20 bits
- Associate each process with a mapping table from virtual page numbers to physical page numbers
    - The table will have 2^20 ≈ 1 million entries!
    - OS registers the mappings with the MMU


32-bit virtual address
Virtual page number (20 bits)
Offset (12 bits)
Page table
Physical page number (P bits)
Physical page number (P bits)
⋮
Physical page number (P bits)
Physical page number (P bits)
Offset (12 bits)
P+12 bit physical RAM address

Two-level Page Table

- Most address spaces are sparse: not every page in the address space is actually used
    - Single-level page table requires us to have an entry for each page (null entries for unused pages, and real entries for used pages)
- With a two-level page table, we don't have to materialize second-level tables for which there are no used pages
    - There may be null entries in both the first and second levels


32-bit virtual address
Virtual directory number (10 bits)
Virtual page number (10 bits)
Offset (12 bits)
Page directory
Directory entry
Directory entry
Directory entry
Page table
Physical page number (P bits)
Physical page number (P bits)
Physical page number (P bits)
P+12 bit physical RAM address
Physical page number (P bits)
Offset (12 bits)

Two-level Page Table: Simple Example

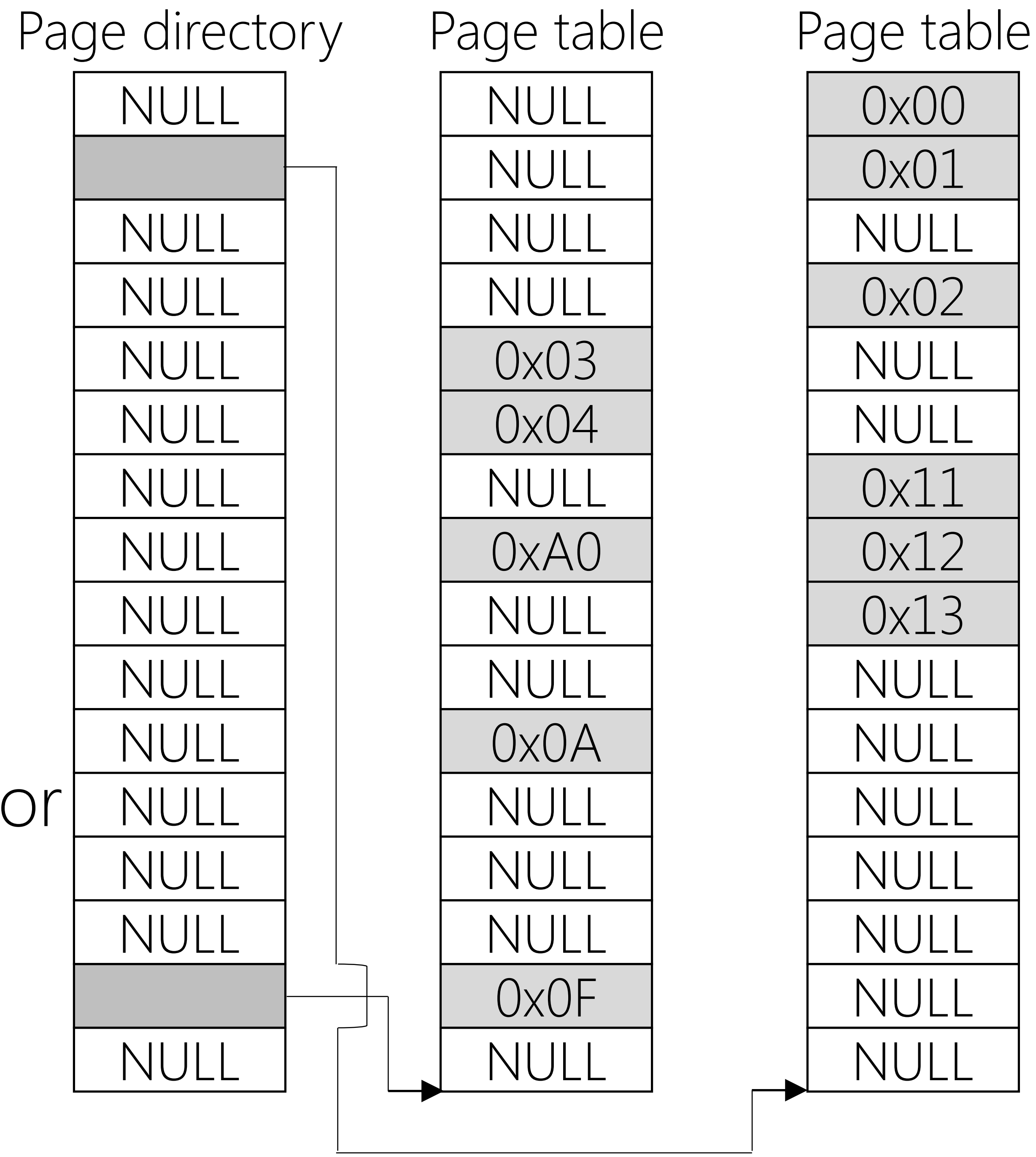
12-bit virtual address

Virtual directory number (4 bits)	Virtual page number (4 bits)	Offset (4 bits)

Questions:

- What is the page size?
- What are the physical addresses for these virtual addresses?

VA 0x133 VA 0x234
VA 0xE23 VA 0xE45
VA 0xEEE VA 0xFEE



Page directory	Page table	Page table
NULL	NULL	0x00
	NULL	0x01
NULL	NULL	NULL
NULL	NULL	0x02
NULL	0x03	NULL
NULL	0x04	NULL
NULL	NULL	0x11
NULL	0xA0	0x12
NULL	NULL	0x13
NULL	NULL	NULL
NULL	0x0A	NULL
NULL	NULL	NULL
NULL	NULL	NULL
NULL	NULL	NULL
	0x0F	NULL
NULL	NULL	NULL

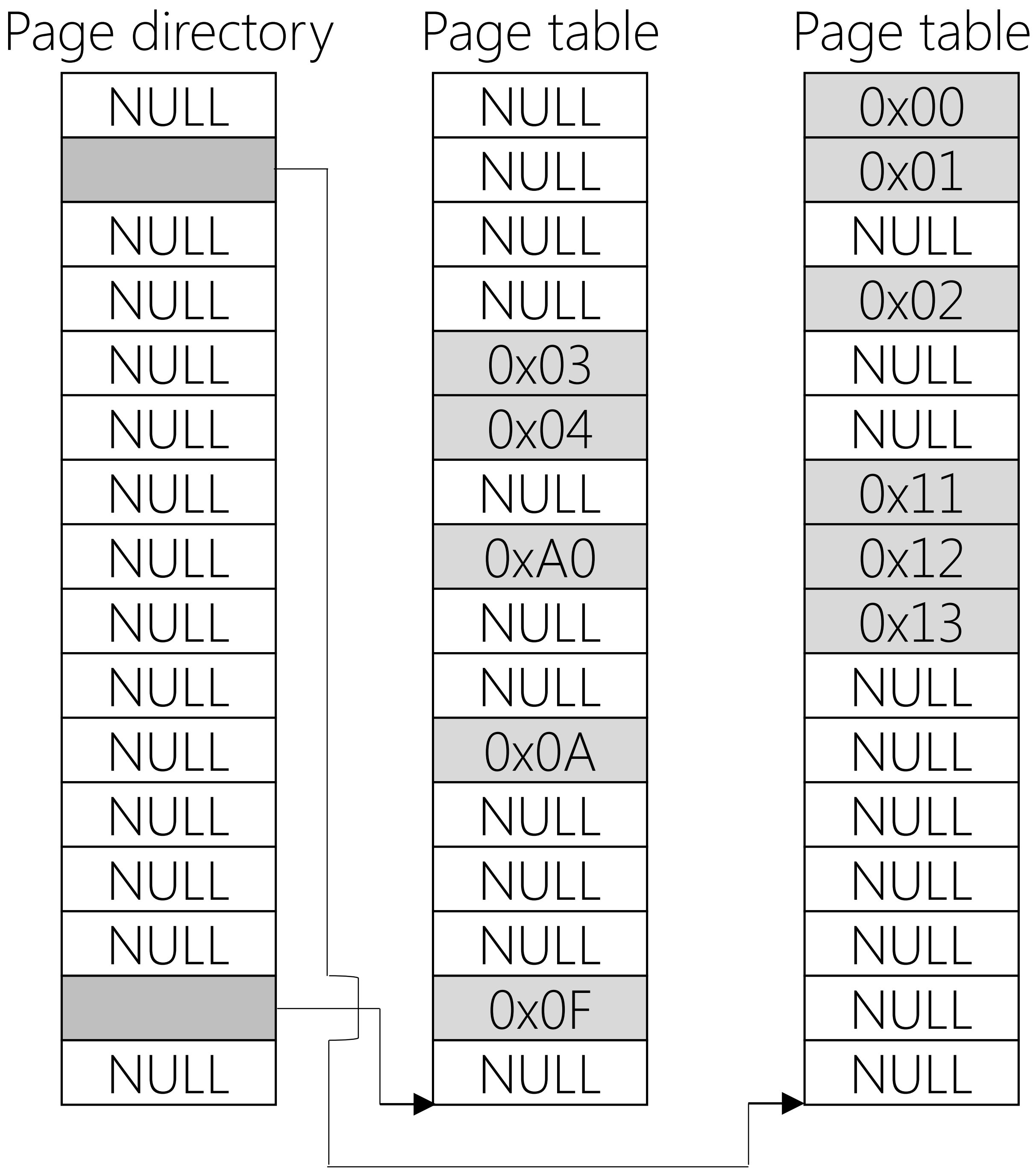
Two-level Page Table: Simple Example

12-bit virtual address

Virtual directory number (4 bits)	Virtual page number (4 bits)	Offset (4 bits)

Questions:

- What is the size of the virtual address space?
- What is the maximum amount of physical memory that an address space can use?
- How many pages are in use?



Page directory	Page table	Page table
NULL	NULL	0x00
	NULL	0x01
NULL	NULL	NULL
NULL	NULL	0x02
NULL	0x03	NULL
NULL	0x04	NULL
NULL	NULL	0x11
NULL	0xA0	0x12
NULL	NULL	0x13
NULL	NULL	NULL
NULL	0x0A	NULL
NULL	NULL	NULL
NULL	NULL	NULL
NULL	NULL	NULL
	0x0F	NULL
NULL	NULL	NULL

Generating Code On The Fly

- A process's code segment is read-only and static size . . .
- . . . but sometimes a process needs to generate code dynamically
    - Ex: The just-in-time (JIT) compiler for a dynamic language like JavaScript will dynamically translate JavaScript statements into machine code; executing the new machine code will be faster than interpretation
    - Ex: Dynamic binary translation tools perform machine-code-to-machine-code translation to inject diagnostics, security checks, etc.
- Dynamic code generation typically places the new code in heap pages which are marked as executable

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>

int main(int argc, char *argv[]){
   //x86 code for:
   //   mov eax, 0
   //   ret
   unsigned char code[] = {0xb8, 0x00, 0x00,
                           0x00, 0x00, 0xc3};

   if(argc != 2){
      fprintf(stderr, "Usage: jit <integer>\n");
      return 1;
   }

   //Overwrite immediate value "0" in mov instruction
   //with the user's value.  Now our code will be:
   //   mov eax, <user's value>
   //   ret
   int num = atoi(argv[1]);
   memcpy(&code[1], &num, 4);

   //Allocate writable+executable memory.
   void *mem = mmap(NULL, sizeof(code),
                    PROT_WRITE | PROT_EXEC,
                    MAP_ANON | MAP_PRIVATE, -1, 0);
   memcpy(mem, code, sizeof(code));

   //The function will return the user's value.
   int (*func)() = mem;
   return func();
}
```