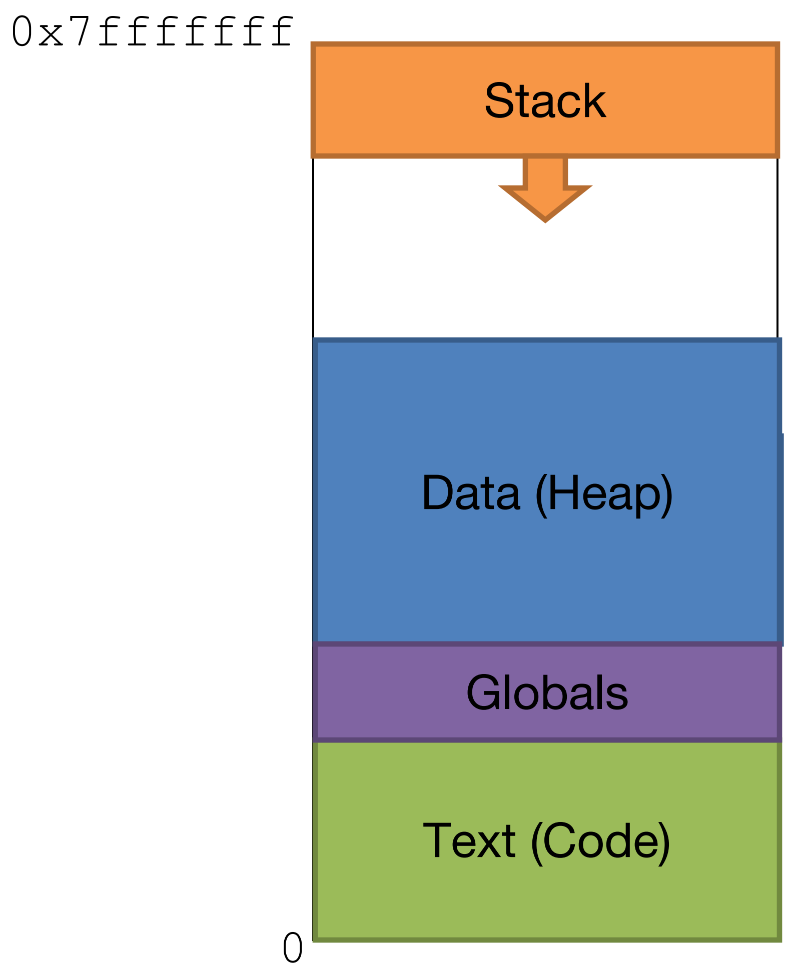
Processes, Address Spaces, and Memory Management

CS449 Spring 2016

Process

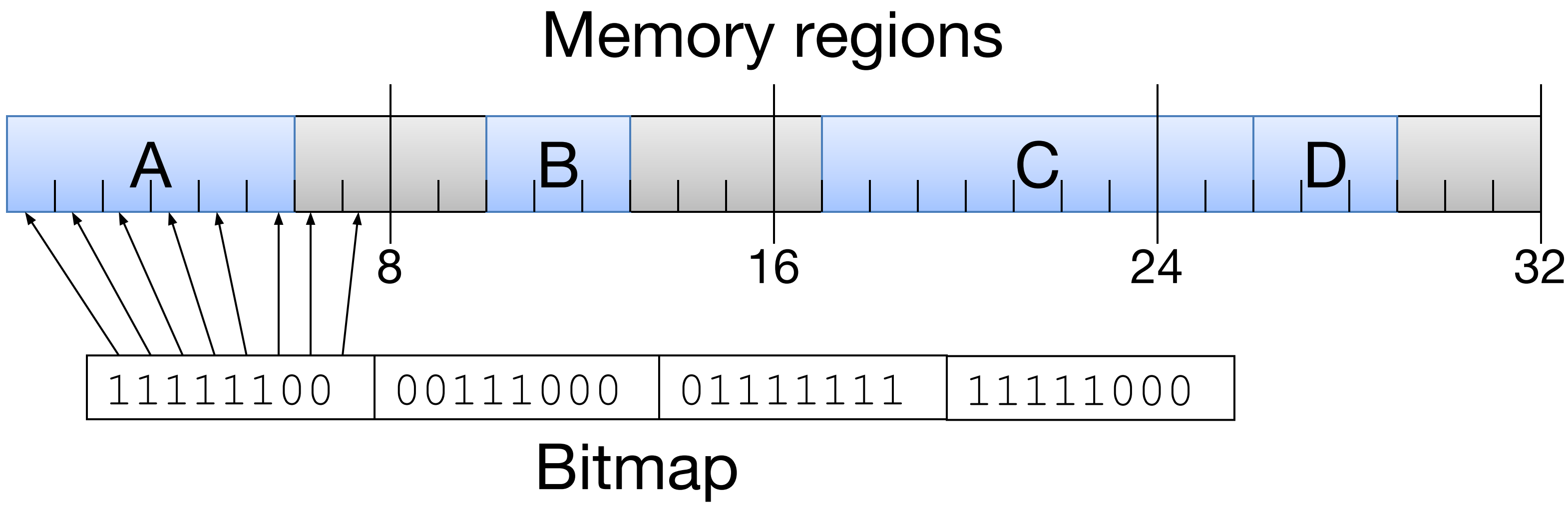
A running program and its associated data

Process's Address Space



Heap Memory Management

Bitmaps



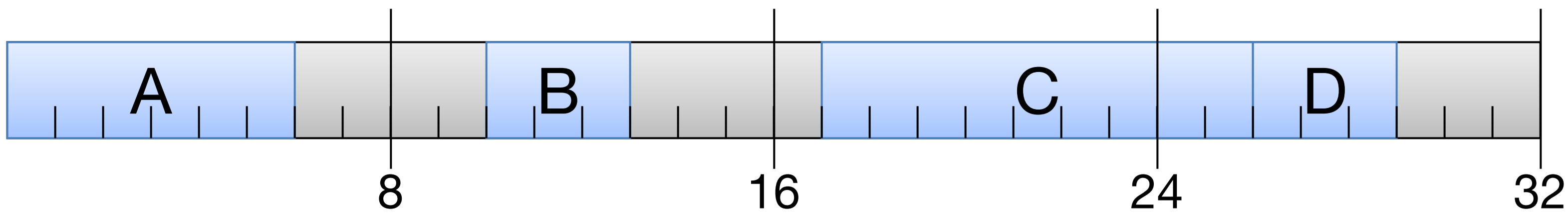
Memory regions
A
B
C
D
8
16
24
32
11111100
00111000
01111111
11111000
Bitmap

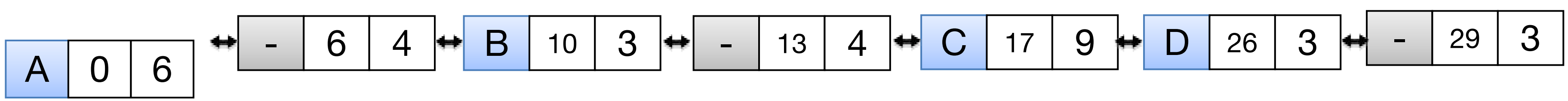
Minimal Units of Allocation

- Break memory up into fixed sized chunks
- Pros:
  - Easier to manage
    (When unsetting bit, surrounding 0s are implicitly coalesced to form a larger contiguous free block)
  - Need just one bit to represent a chunk
- Cons:
  - **Internal fragmentation**: a chunk being only partly full
  - Difficulty in finding large enough contiguous free block

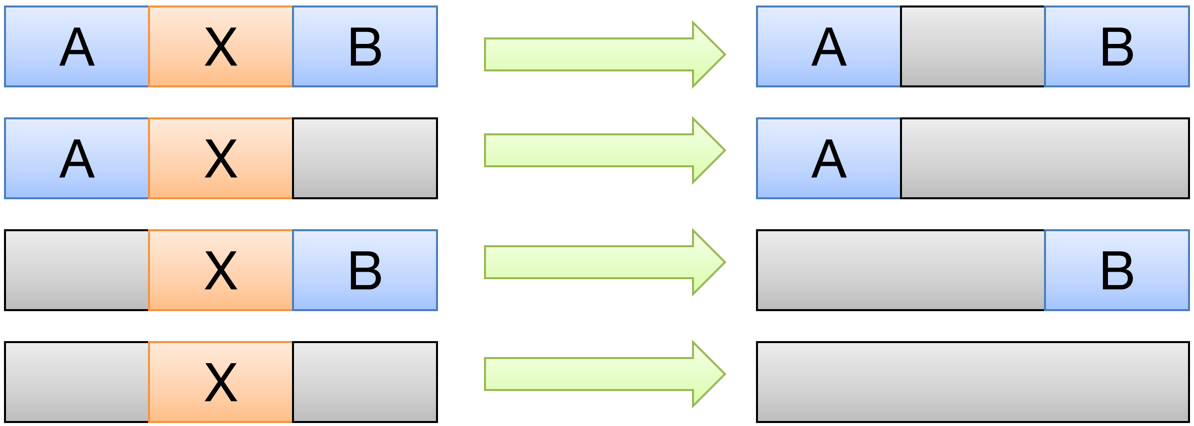
Linked Lists

Memory regions





Reclaiming Freed Memory



Allocation Strategies

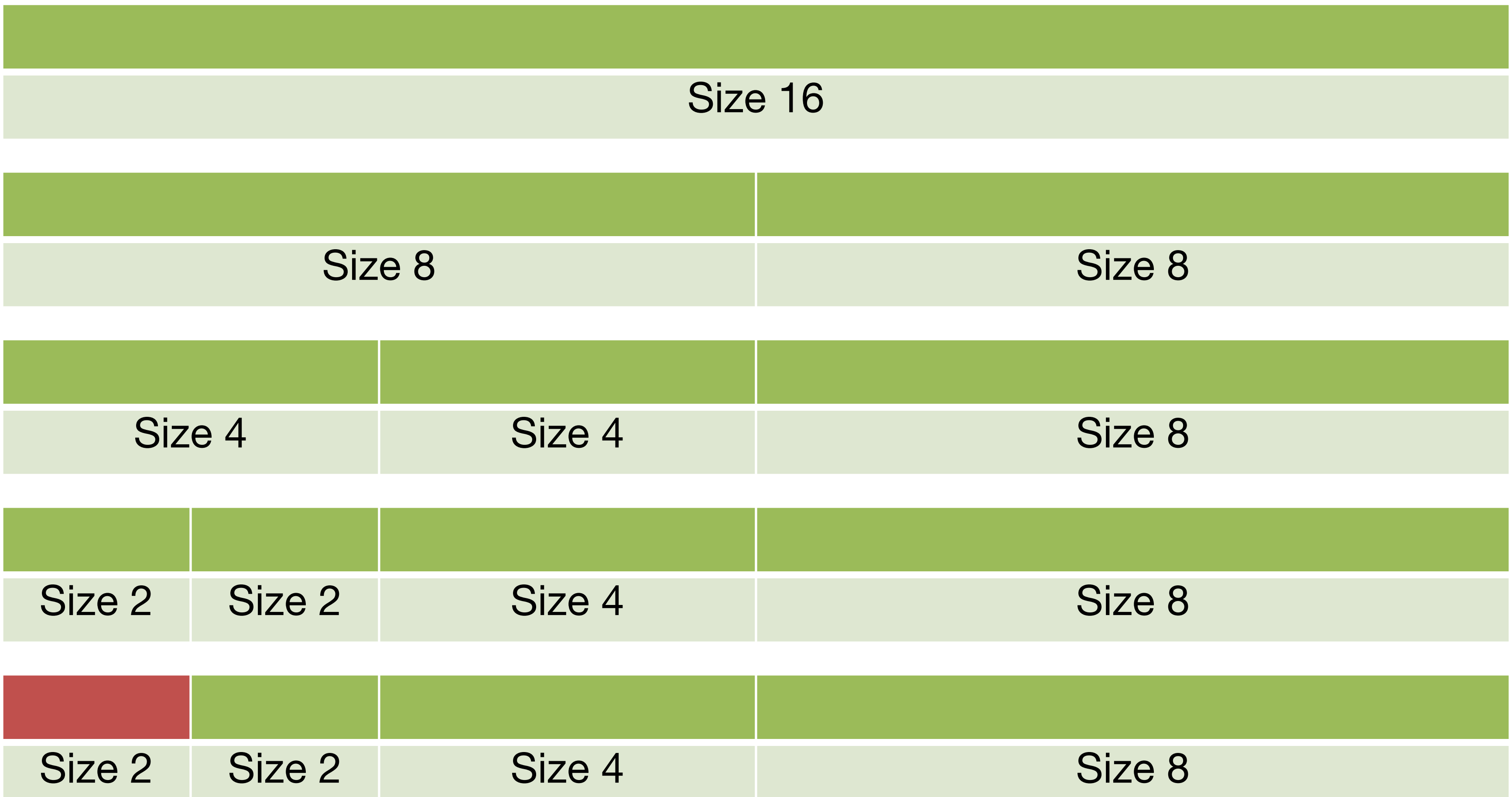
- First fit
  - Find the first free block, starting from the beginning, that can accommodate the request
  - Rationale: simple and reasonably fast
- Next fit
  - Find the first free block, starting where the last search left off, that can accommodate the request
  - Rationale: why search all the way from the beginning when it’s unlikely to turn up useful blocks
- Best fit
  - Find the free block that is closest in size to the request
  - Rationale: do not take up a larger block needlessly

Allocation Strategies Continued

- Worst fit
  - Find the free block with the most left over after fulfilling the allocation request
  - Rationale: best fit typically leaves very small blocks that are useless
- Quick fit
  - Keep several lists of free blocks of common sizes, allocate from the list that nearest matches the request
  - Rationale: allocation very fast and wastes little space
  - Challenge: how to coalesce smaller blocks
- All strategies suffer from **external fragmentation**
  - Having many free blocks that are too small to be useful
  - Need a better coalescing strategy

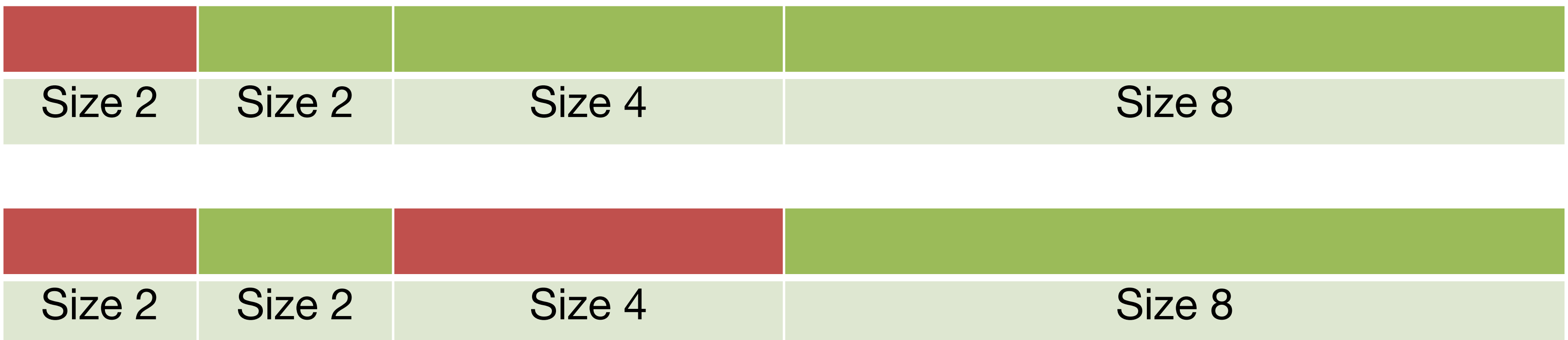
Buddy Allocation

Allocation of size 2 in a region of size 16



Buddy Allocation

Allocation of size 4 in a region of size 16



Buddy De-Allocation

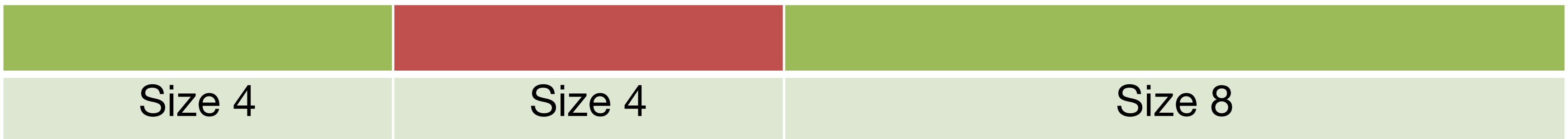
Free region of size 2 in a region of size 16

Mark region as free

Combine with
“buddy”

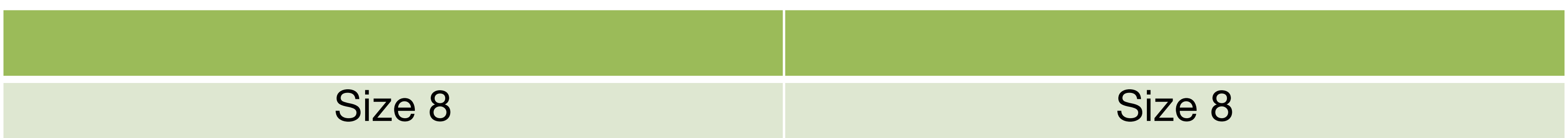
Buddy De-Allocation

Free region of size 4 in a region of size 16



Mark region as free

Combine with “buddies”



Buddy Location

- Given an allocation at address `addr`, where is its buddy?
- In the previous example, we had two buddies of size 4 at addresses 0 and 4
- Since we always have our space, we can force all of our sizes to be powers of 2.
  - Then our two buddies only differ by 1 bit in their number

```
buddy = addr ^ size
```

Buddy Allocation

- Used in the Linux kernel
- Nodes maintained as a binary tree
- Efficient
  - Little external fragmentation (first tries to find empty region before splitting existing region)
  - Coalescing block of size N takes at most $\log_2 N$ steps

Memory Management Pitfall

```
#include <stdlib.h>

int main()
{
  char s1[10], *s2;
  s2 = (char *)realloc(s1, 20);
  free(s2);
  return 0;
}
```

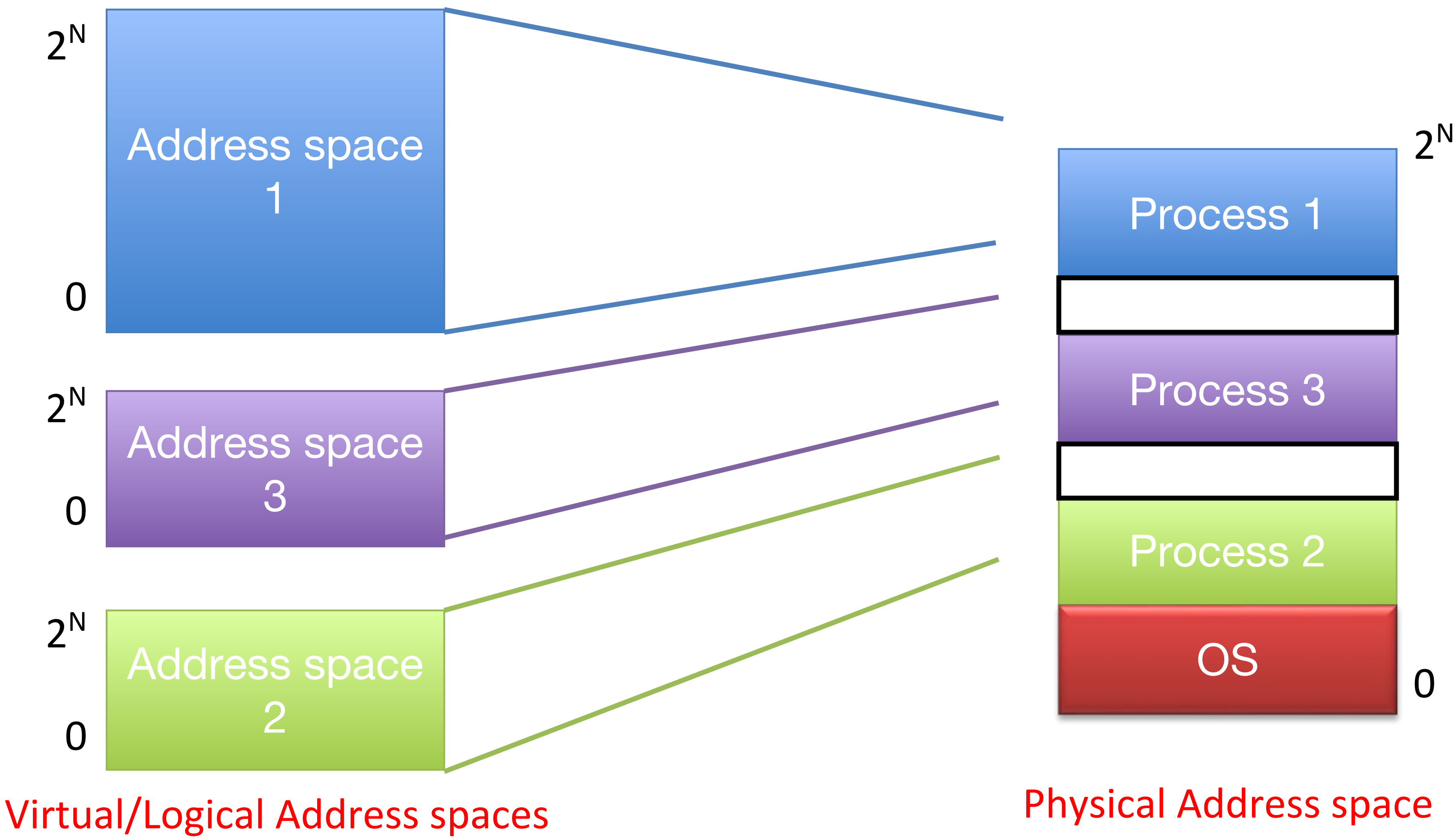
```
>> gcc –g main.c
>> ./a.out
Segmentation fault (core dumped)
>> valgrind ./a.out
[sic]
==26279== Invalid free() / delete / delete[] / realloc()
==26279==    at 0x4A06BE0: realloc (vg_replace_malloc.c:662)
==26279==    by 0x40051C: main (main34.c:6)
==26279==  Address 0x7feffff40 is on thread 1's stack
```

- Never mix manual memory management (heap) with automatic management (stack)!

OS-Level Memory Management

Issue: Sharing of Physical Memory Among Multiple Processes

- Translation from logical to physical addresses

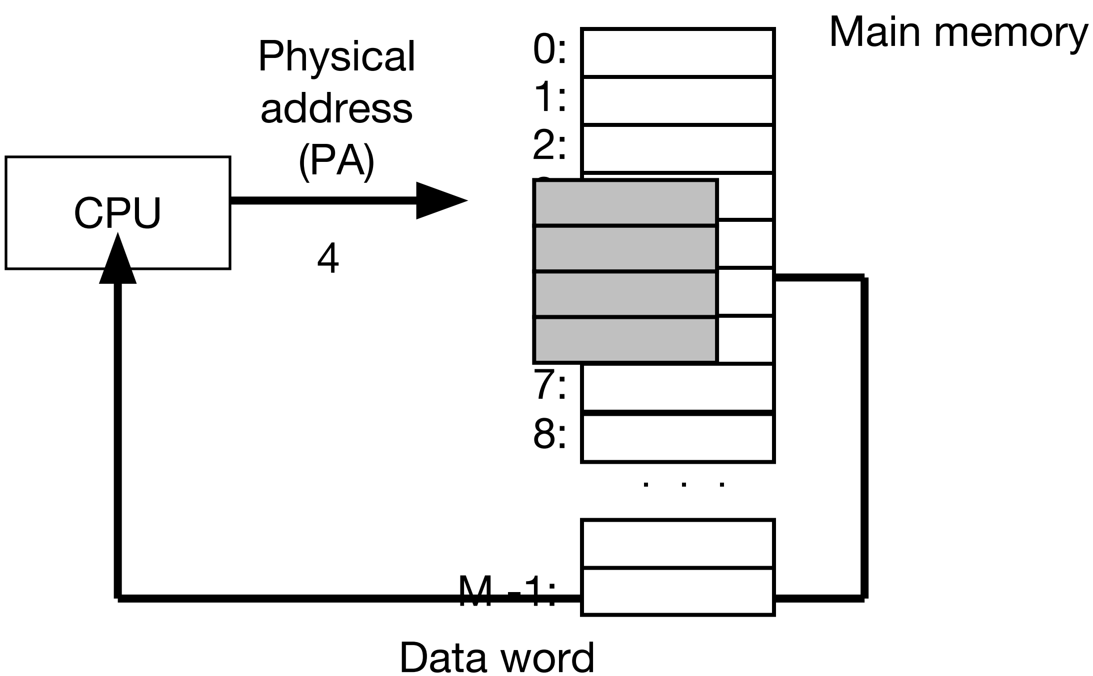


Goals for OS Memory Management

- Transparency
  - Processes not aware memory is shared
  - Run regardless of number and/or locations of processes
- Protection
  - Cannot corrupt OS or other processes
  - Privacy: Cannot read data of other processes
- Efficiency

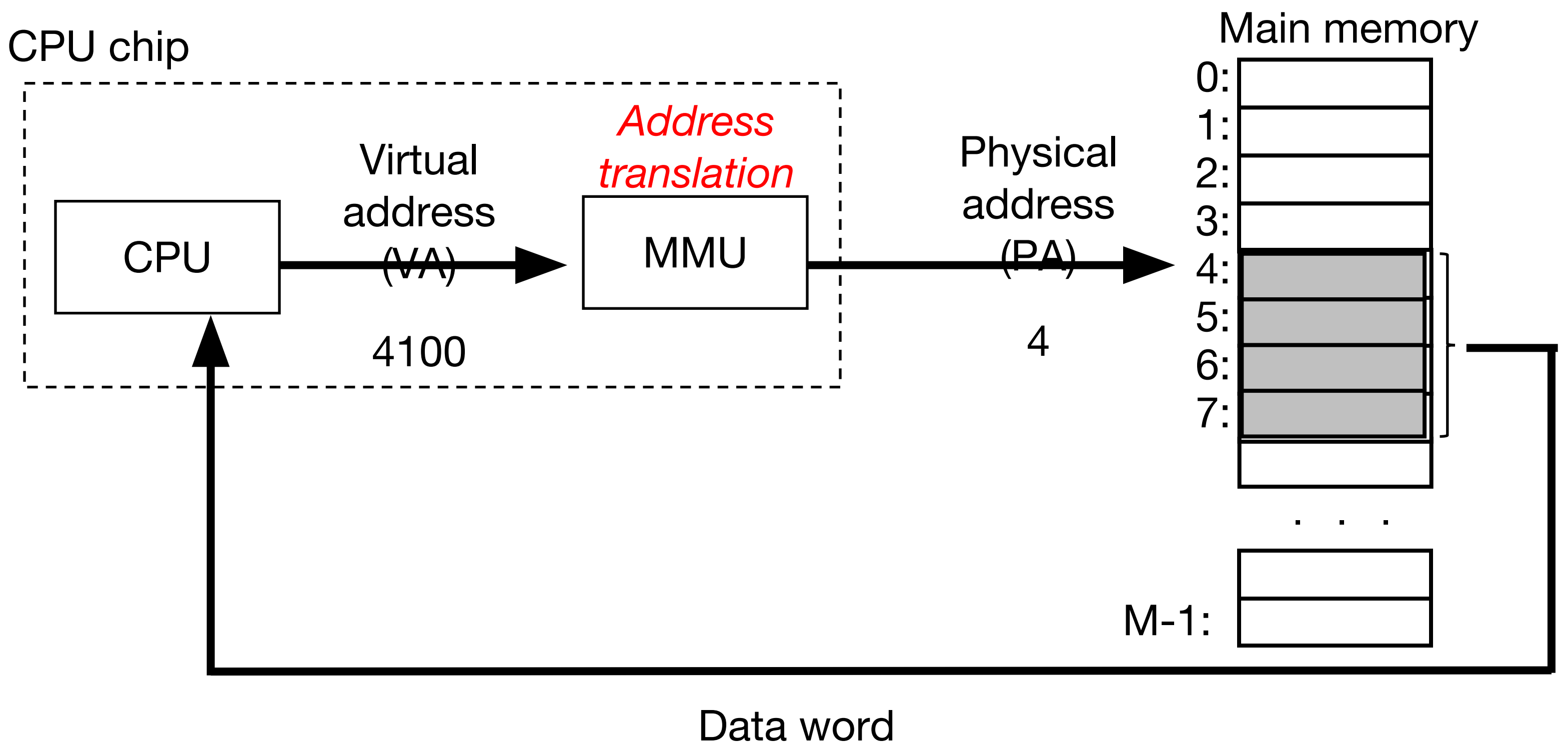
A system with physical addressing

- Main memory - An array of M contiguous byte-sized cells, each with a unique physical address
- Physical addressing
  - Most natural way to access it – Addresses generated by the CPU correspond to bytes in it
  - Used in simple systems like early PCs and embedded microcontrollers (e.g. cars and elevators)



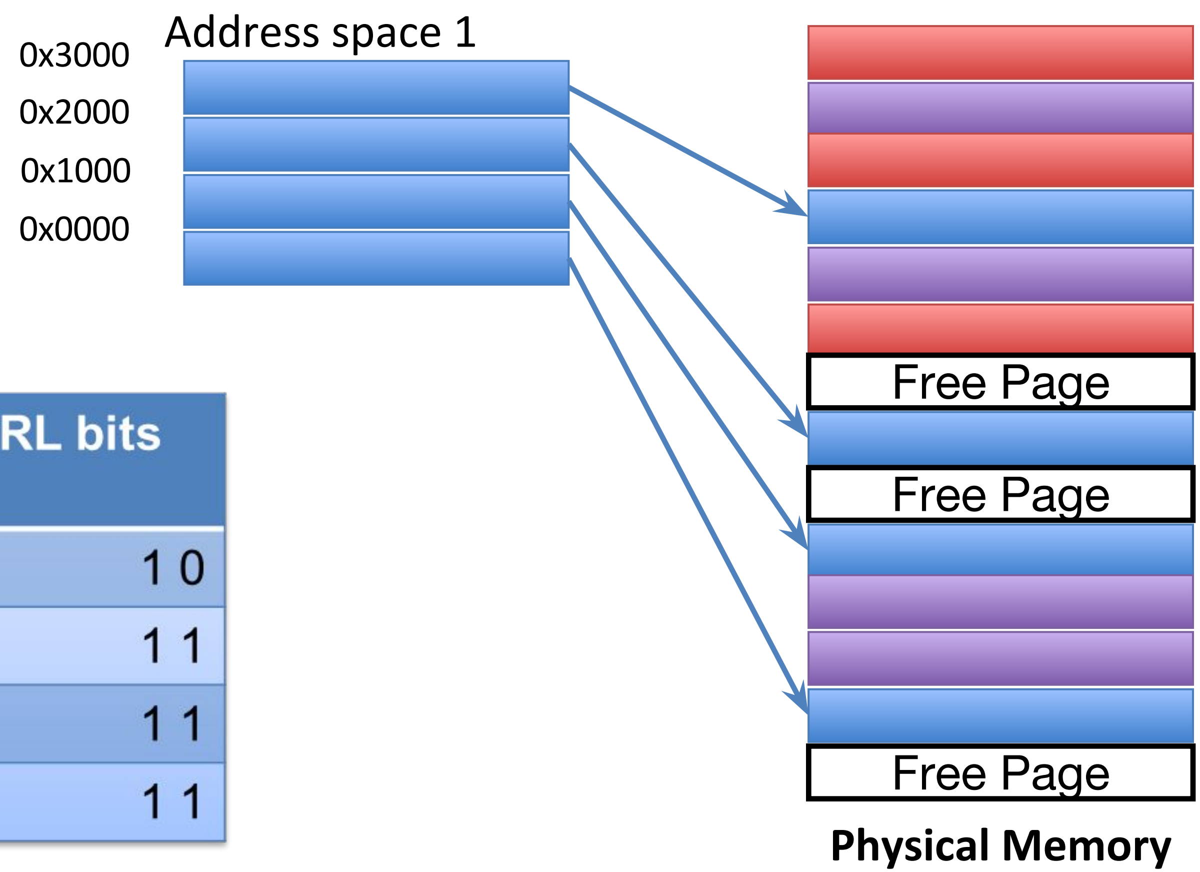
A system with virtual addressing

- Modern processors use virtual addresses
- CPU generates virtual address and address translation is done by dedicated hardware (*memory management unit*) via OS-managed lookup table



Virtual Memory Example

- Mapping of virtual addresses to physical memory



Page Table for process 1

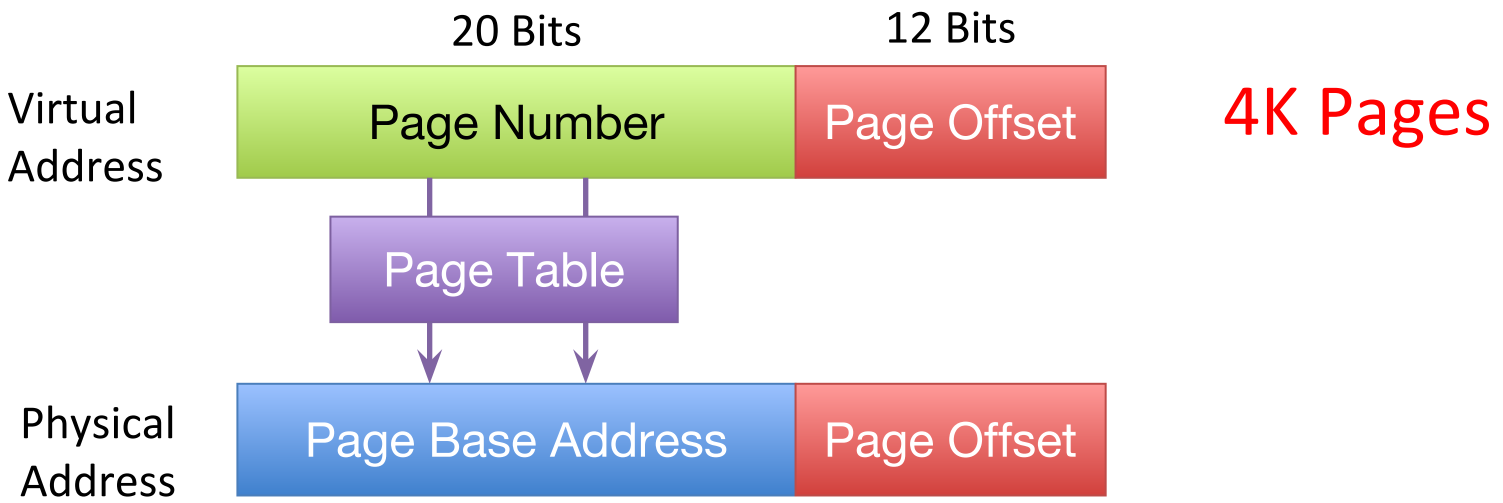
Base Address	CTRL bits
1	1 0
4	1 1
6	1 1
10	1 1

Pages

- Unit of memory allocation from the OS
- Efficient (compared to variable sized blocks)
  - Fast to allocate and free (no need to find a ‘fit’)
  - Easier to translate
- Mostly 4KB, but not always
  - 32 bit x86 supports 4KB and 4MB
  - 64 bit x86 supports 4KB, 2MB and 1GB

Page Translation

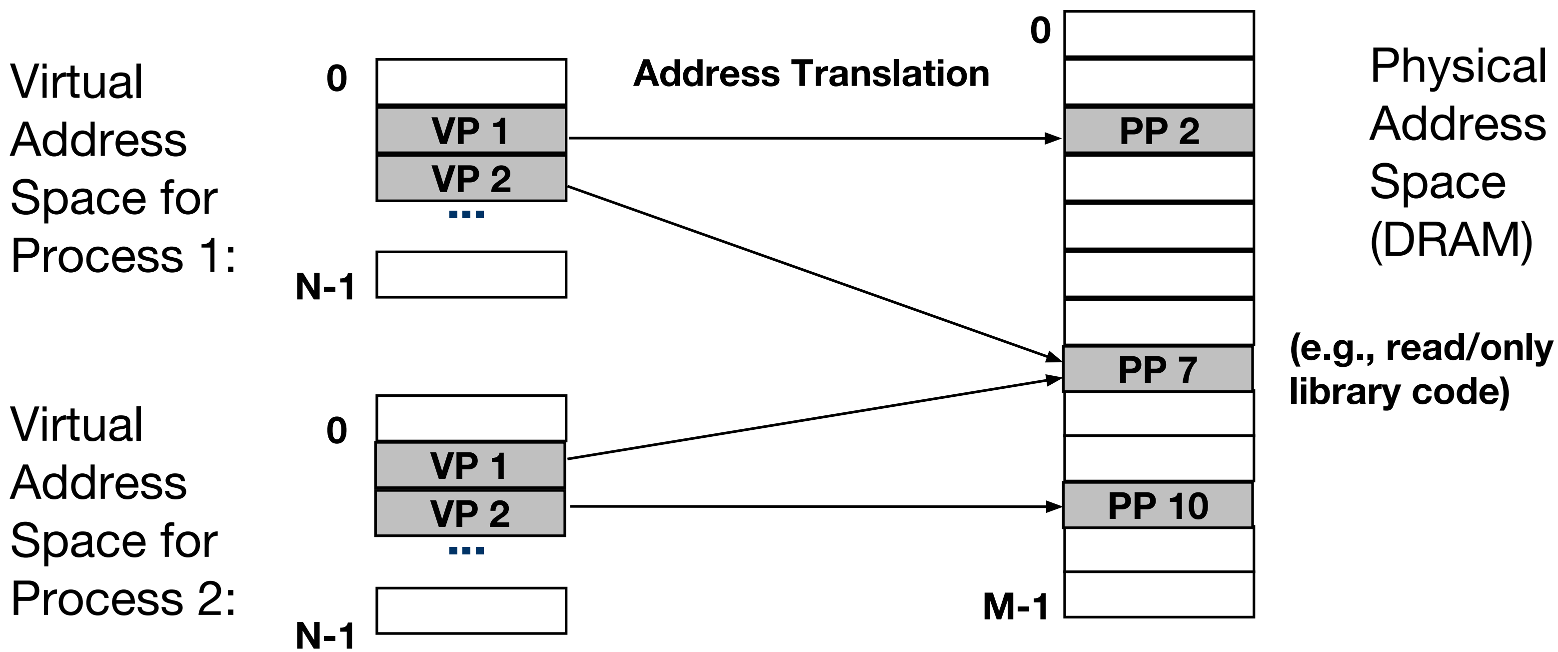
- How are virtual addresses translated to physical addresses
  – Upper bits of address designate page number



- Happens in MMU address translation hardware in CPU
- Page offsets concatenated instead of added due to fixed size
  → More efficient address translation

Transparency: Separate virtual addr. spaces

- Each process has its own virtual address space
  - OS controls how virtual pages are assigned to physical mem.
  - If OS runs out of physical mem., disk ‘swap’ space is assigned to lesser used pages



Protection: Separate Address Spaces + Permission bits

- Page table entry contains access rights information
  - HW enforces this protection (trap into OS if violation occurs)

Must be running in kernel (sys) mode

Page tables with permission bits

Process i:

	SYS	READ	WRITE	Address
VP 0:	No	Yes	No	PP 6
VP 1:	No	Yes	Yes	PP 4
VP 2:	Yes	Yes	Yes	PP 2
⋮				

Process j:

	SYS	READ	WRITE	Address
VP 0:	No	Yes	No	PP 9
VP 1:	Yes	Yes	Yes	PP 6
VP 2:	No	Yes	Yes	PP 11
⋮				

Physical memory