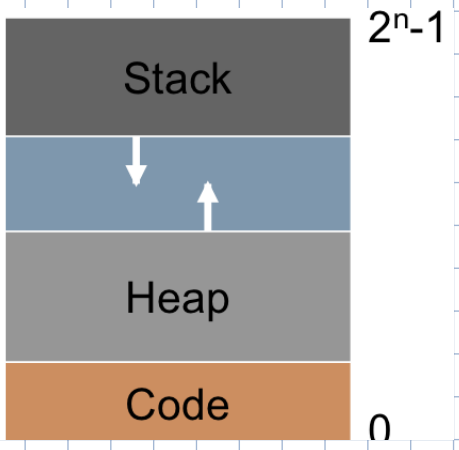
another abstraction

why study memory management?
- code & data location
- where's stack?
- wtf is malloc?
- where's heap?

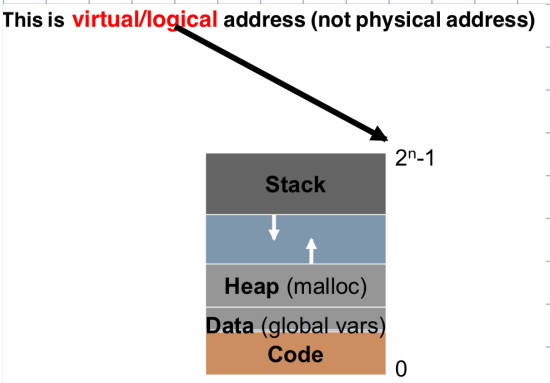


infinite recursive program
- memory address decreases in value

infinite malloc
- mem address grows in value

these are virtual addresses

work w/ virtual addresses, must be translated to physical address
- storage appears contiguous
- translation done w/ MMU



virtual can be predetermined

physically doesn't have to be contiguous
- OS deals with mapping

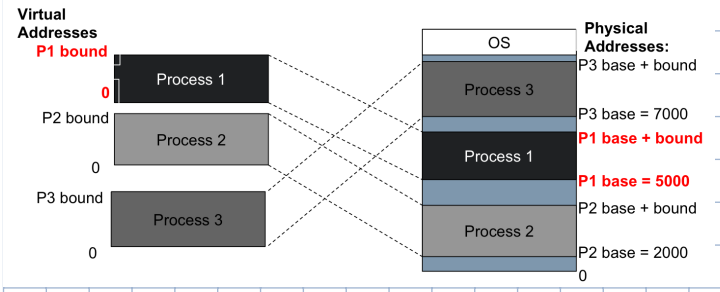
chip programming: 1 at a time, OS can get corrupted

Dynamic Relocation:

Base & Bound — multiple processes using different parts of memory
- each process gets its own contiguous part in memory - not overlapping
- each process has its own virtual memory space

managed by OS → hides physical addresses

base- where virtual memory starts for process
bound- gets this amount of virtual memory



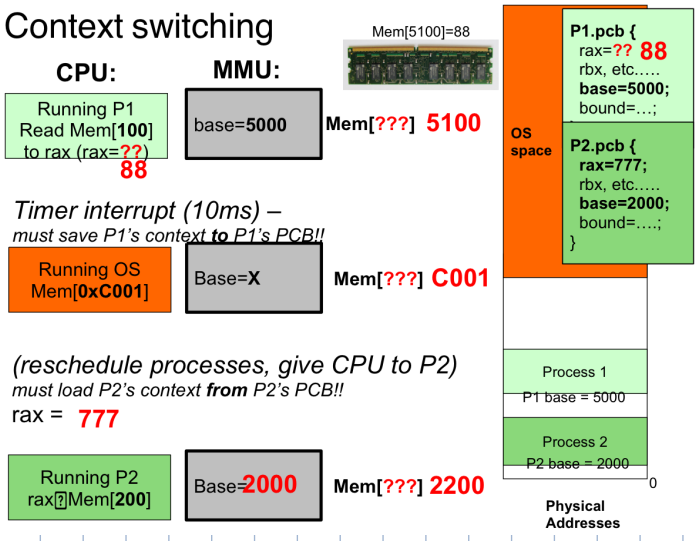
address translation: hardware :)
Memory Management Unit (MMU) set for each CPU

MMU has 2 registers: ① base ② bound

out of bounds? → segfault

Context switching
switch between CPU executing processes

OS has base = 0



MMU prepares values

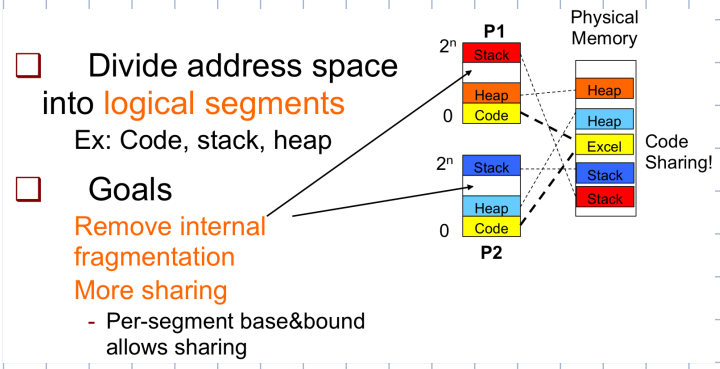
provides private process memory
memory protection

MMU w/ 2 registers

biggest disadvantage of BB → assume fixed-size address space
may need to grow
not dynamic
small process gets big memory → wasted dead (internal fragmentation)
holes between processes → wasted relocation (external fragmentation) costly!
share code? no :(

Segmentation

only allocate space for heap, stack, & code
can have different physical spaces
allows sharing (like excel)



instead of 1 base & 1 bound per process → 1 base & 1 bound per segment (code, stack, heap)

grow stack → change base of stack
grow heap → change bounds

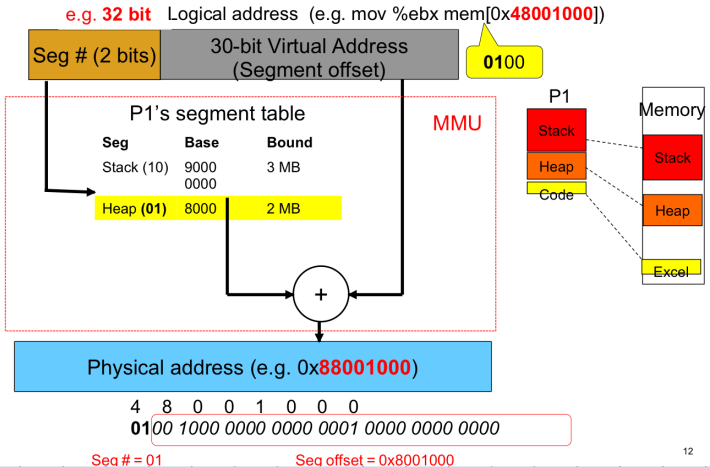
some conflict w/ used space? relocate that space

Segment table

each process has one
more complex MMU

has base & bounds for each segment

translates virtual to physical address
which base to use?
instead of offset, use top bits to indicate specific segment. the rest is the offset



for example:

0x 48100100
↳ 01,00
↳ segment 1 (only have 3 segments here)

access something beyond segment range? → seg fault
↳ change to hit another segment (like stack hitting heap)

00 – code
01 – heap
02 – stack
03 – unused

who decides? between OS, architecture, & compilers

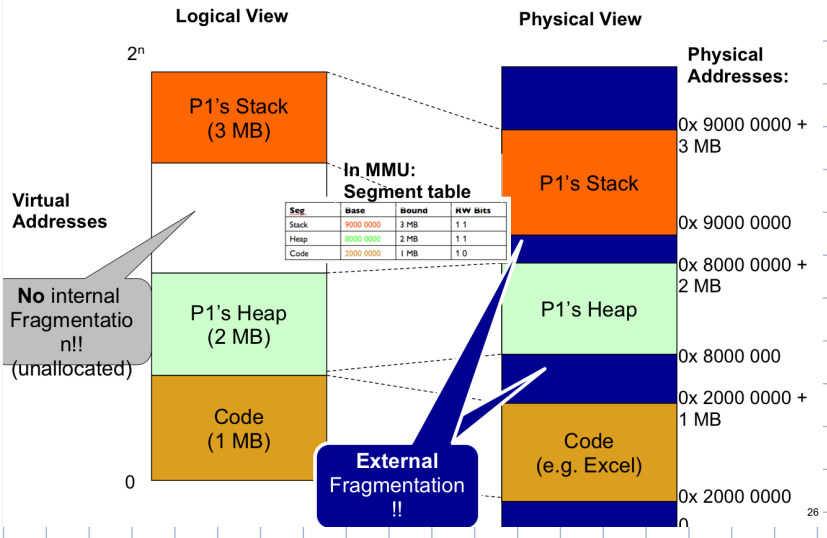
advantages
multiple users running same program
no more internal fragmentation

shared memory

processes can talk to each other

disadvantages
variable sized segments → external fragmentation

each segment must be allocated contiguously
no memory slot for big segment → reshuffle/migration



Paging

instead of contiguous memory for segments, just put them anywhere

break down segments into pages (fixed sized)

hella pages

break down MMU page table to inner & outer tables

have multi-level page tables
pagetables are in memory

store some of this in cache

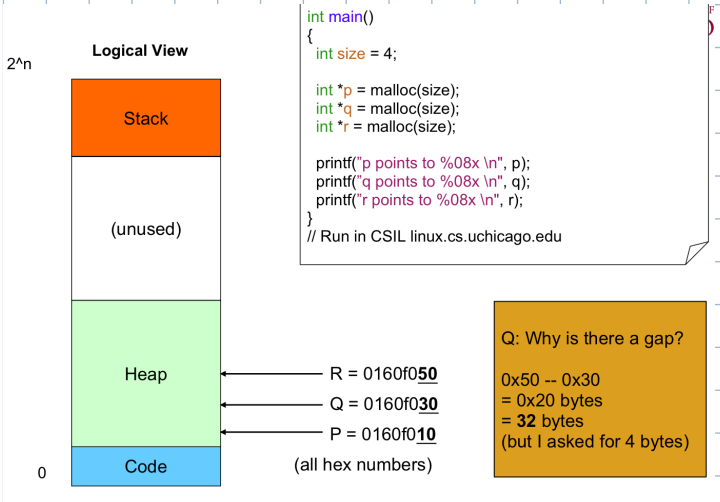
swap space
swapin / swapout

Goals of Memory Management
efficiency
sharing
transparency
protection

Dynamic Memory Allocation

how to manage free space in memory

malloc(size) get size memory from heap → good for dynamic allocation, better than stack



asking for 4 bytes

why is there a space of 32 bytes

Memory is word addressed

32bit: 4bytes 64bit-8bytes

malloc(X) X words

[IMPORTANT!!]

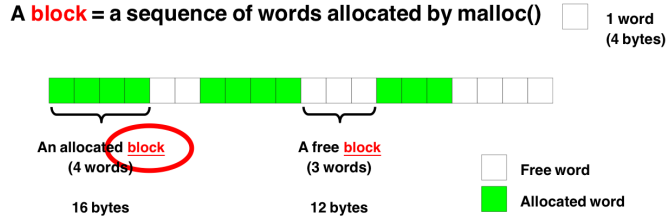
Memory is word addressed

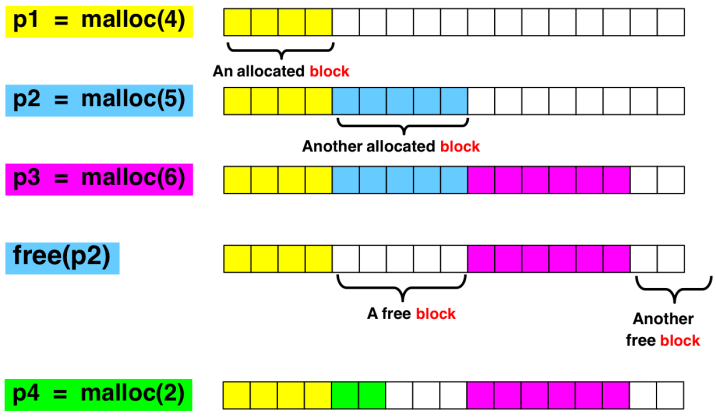
A box = a word = integer size = 4 bytes

In lecture: malloc(X) means malloc (X words), i.e., malloc (X * 4 bytes)

In actual C code: malloc(Y) means Y bytes

A block = a sequence of words allocated by malloc()





malloc (4 words)

reallocate recently freed boxes
reducing external fragmentation

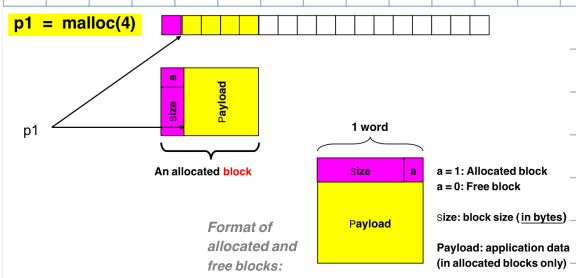
how to manage free blocks?

implicit list - using length
explicit list - among free blocks

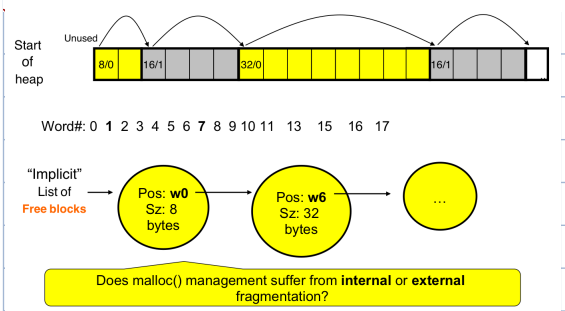
I mplict list
have header w/each block indicating length of data & ready

requires extra word for every block

pointer still goes to payload, not header



Using 20bytes for 16 bytes of info.



yes

Which block use to fulfill malloc?

Best fit: search list, choose "fully fit" fewest bytes left over
- slow
- lots of small leftover blocks

Worst fit: search list, choose most loose fit most bytes left over
- slow
+ leftover blocks are large

First fit: first one fitting request
+ faster
- still slow, early parts of malloc already filled, can traverse most of heap

Next fit: like first fit, but start at last freed block
+ faster than first fit
- still skipping earlier freed blocks

best depends

How does it work?

split block into two blocks: length + payload

free: flip header flag bit to 0

false fragmentation unless we merge 2 recently freed blocks.
instead of 2 contiguous blocks w/ 3 words, combine for 1 6 block
→ now 5 blocks can be allocated here

How to coalesce w/ previous block?

solution: Bidirectional coalescing

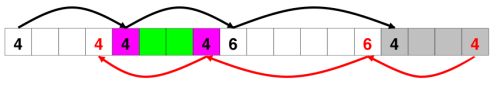
have a header & footer blocks

How to always is header of previous block?

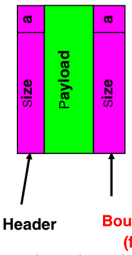
Boundary tags [Knuth73]

Add block **footer**: replicate block header at "bottom" (end) of free block

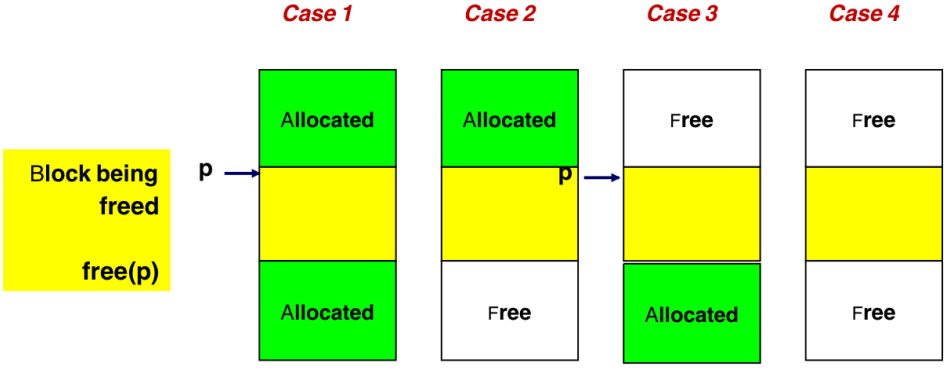
Allows us to traverse the "list" backwards



Format of allocated and free blocks



REAL EXAMPLES / NUMBERS (in "words" Not bytes)

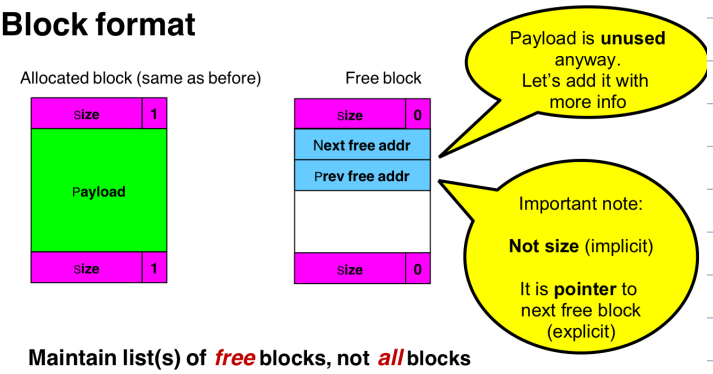


Implicit allows block traversal

Explicit list - list of only free blocks

same as implicit list, first 2 blocks of free point to prev & next free block pointers

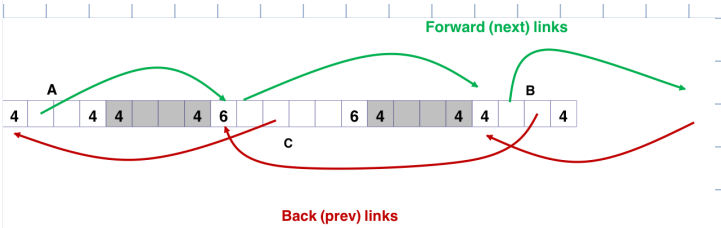
Block format



Maintain list(s) of *free* blocks, not *all* blocks

Goal of malloc() is to manage free space, so it's about the free list

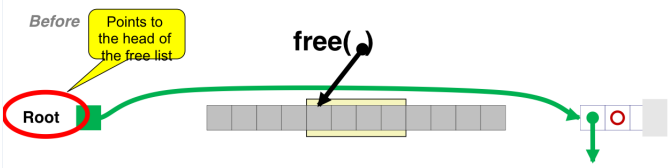
Luckily we track only free blocks, so we can use payload area



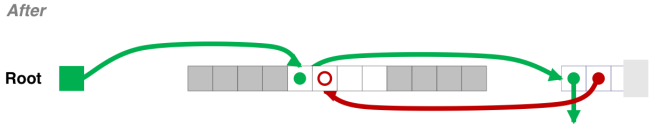
what about freeing?

can use first in last out policy

Freeing With a LIFO Policy (Case 1)



Insert the freed block at the root of the list



Coalescence hard