Segmentation (contd)

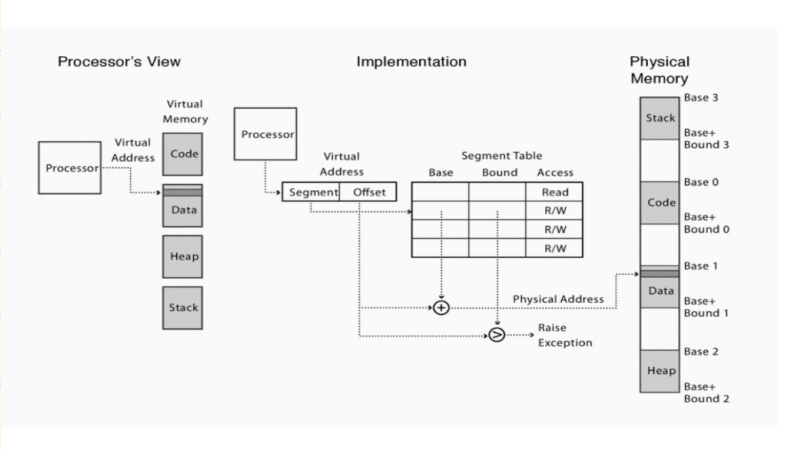
Question: How to support sharing?

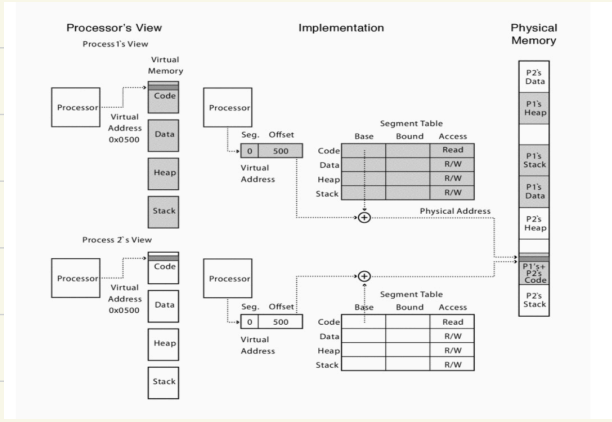
Sometimes, it is useful to share certain memory segments b/w address spaces. eg:- like code sharing.

↳ achieve this by an extra hardware support.

↳ in form of protection bits.

i.e., a few more bits per segment to indicate permissions to read, write, and execute.





Remark: In early systems that use segmentation, the shared code had to be mapped at the same virtual addresses in every process. Otherwise, relocation is needed everytime, which would break sharing. But, the modern systems use position-independent code (PIC) for shared libraries that does not hardcode absolute virtual addresses but uses relative addressing, and therefore the above requirement is unnecessary.

Example of sharing (UNIX fork() + copy-on-write (COW))

Traditionally, when a process calls fork(), the OS creates a child process that's almost an exact copy of the parent.

* same code * same data (heap, stack) * same open files

Naively, this would mean duplicating all of the parent's memory
→ very expensive!

But often, the child doesn't even need all of it (as most children immediately call exec() to load a new program).

So how can you avoid copying memory unnecessarily?

↳ copy-on-write (COW)

i.e., instead of duplicating, here OS can do as follows:

* UNIX fork() → makes a copy of a process

* segments allow a more efficient implementation
  - copy segment table into child

- mark parent and child segments read-only
- if child or parent writes to a segment (eg:- stack, heap)
  trap into kernel (protection fault happens as its RO)
- make a copy of the segment and updates the faulting process's segment descriptor to point to the new memory.

(Note that in the above, if a child calls exec() then the old segments are discarded and it loads new segments for the new program.

Question: In the segmentation approach, what does OS do during context switch?

segment registers must be saved and restored.

What happens when program uses memory beyond end of stack/heap?

- segmentation fault into OS kernel
- kernel allocates more space (sometimes, OS can reject the request too!)
- update the segment size registers

Pros of segmentation

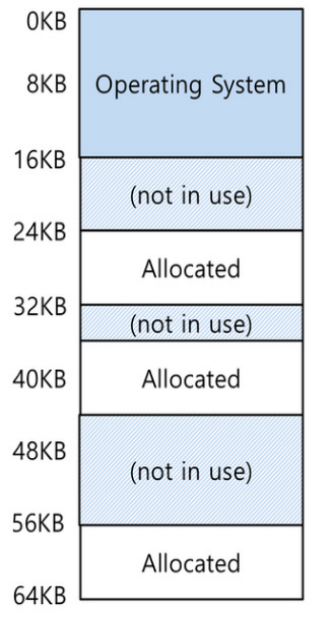
* can share code/data segments b/w processes
* can protect code segment from being overwritten.
* can transparently grow stack/heap as needed.
* can detect if need to copy-on-write.

Cons

* complex memory management
  - need to find chunk of a particular size.
* difficult to allocate new segments, or to grow existing ones.

(because, physical memory may become full of little holes of free spaces)

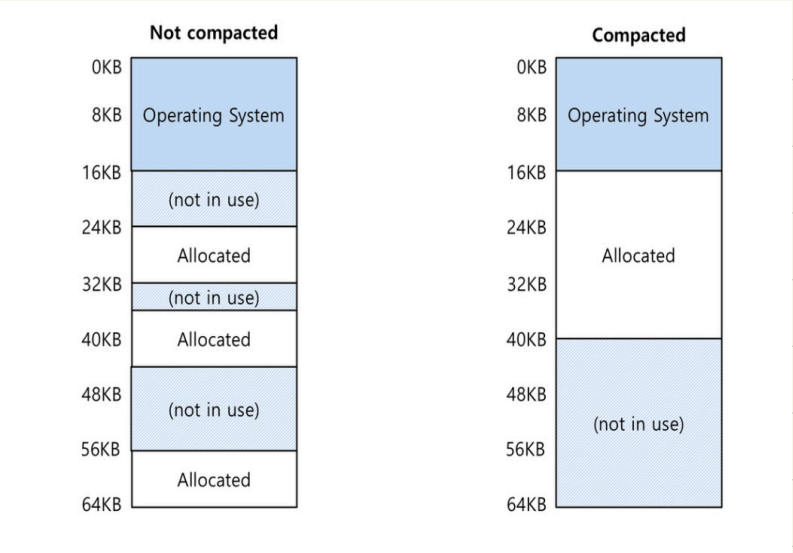
↳ external fragmentation.



* suppose a process needs a 20KB segment.
  - Here there is 24KB free, but no contiguous segment of 20KB!

What to do?

↳ compaction: re-arranging the existing segments in physical memory.



Note that compaction is costly. As it requires to

- stop the running process
- copy data to somewhere
- change segment register value.

Reference: Chapter 16, OSTEP book.

Free space management

* Easy → if the space you are managing is divided into fixed-sized units.

  eg:- Paging (will see later!)

* Difficult and interesting → if the free space you are managing consists of variable-sized units.

  - user-level memory allocation library (as in malloc() and free())
  - OS managing physical memory using segmentation.

Questions

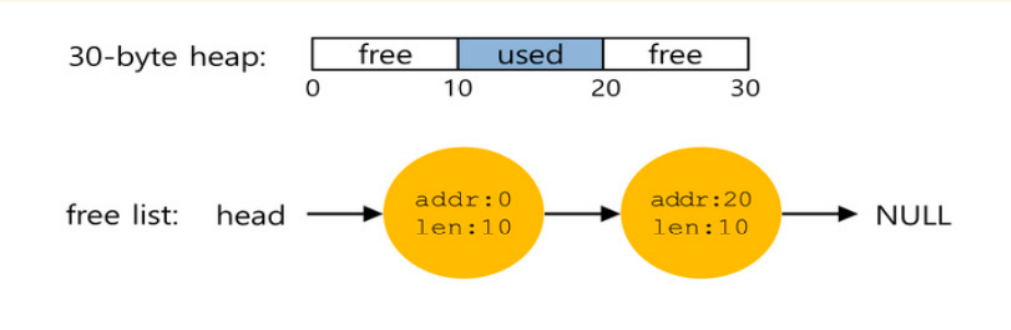
* How to manage free space to satisfy variable-sized requests?
* What strategies can be used to minimize fragmentation?
* What are the time and space overheads?

Common techniques in allocators → splitting
                                → coalescing.

Splitting: finding a free chunk of memory that can satisfy the request and splitting it into two.

eg:- assume the following 30-byte heap and the corresponding free list:

↳ (single fixed size, for the sake of example)

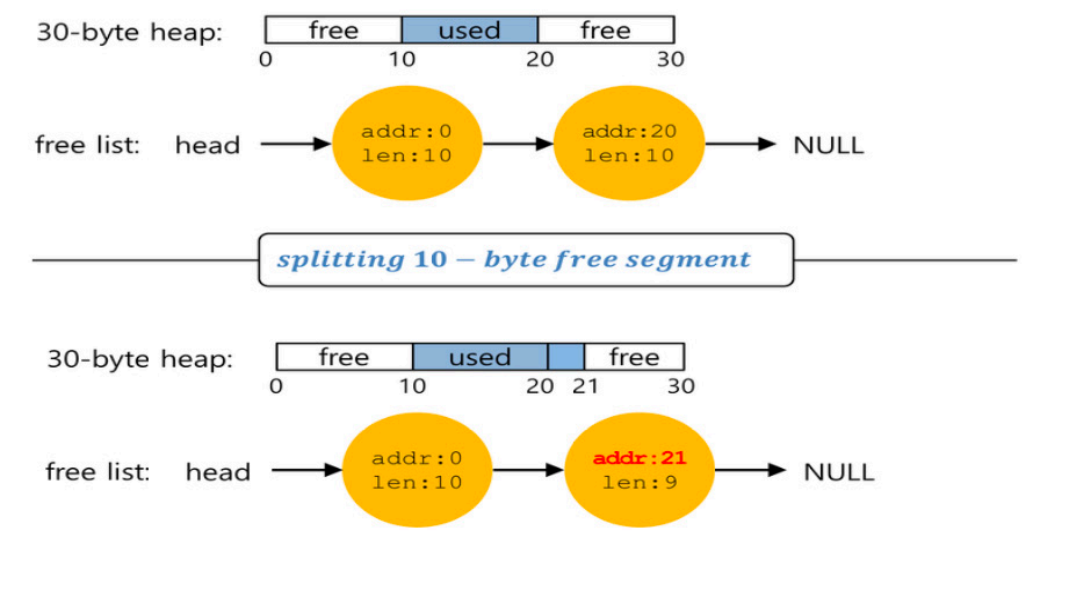


Case-1: request for > 10 bytes → fail!

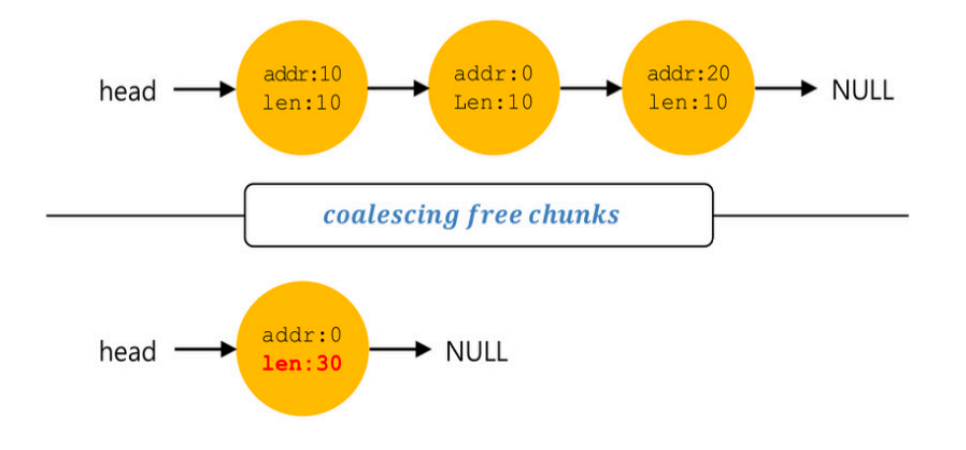
Case-2: request for = 10 bytes → one of the free chunks will be allocated.

Case-3: request for something smaller than 10 bytes?

eg- request for 1 byte were made



Coalescing: Merge a returned free chunk with existing chunks into a single free chunk if addresses of them are nearby.



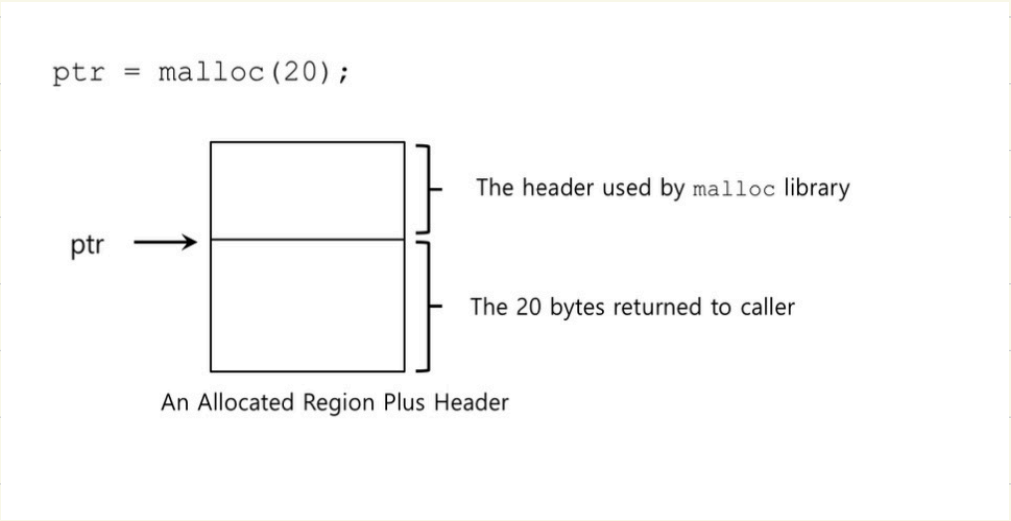
Recall that the interface to free(void *ptr) does not take a size parameter.

Then, how does the library know the size of the memory region that will be back into free list?

↳ most allocators store extra information in a header block.

* The size for free region is the size of the header + size of the space allocated to the user.

i.e., if a user request N bytes, the library searches for a free chunk of size N + the size of the header.



What are the contents of the header?

* size of the allocated region
* "magic number", for additional integrity checking, etc..