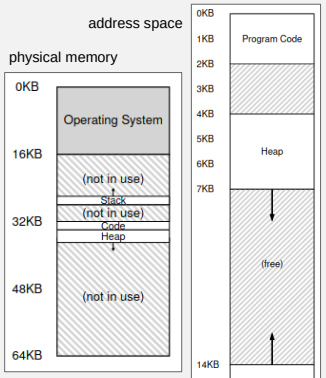
Segmentation Recap

- split process address space into *segments*
  - code, heap, stack or code/heap, stack
    - *fine-grained segmentation* allows for more, smaller segments
  - the process still sees one big contiguous address space but the segments can be placed separately in memory
- hardware support

Segment	Base	Size (max 4K)	Grows Positive?	Protection
$Code_{00}$	32K	2K	1	Read-Execute
$Heap_{01}$	34K	3K	1	Read-Write
$Stack_{11}$	28K	2K	0	Read-Write

  - base, bound registers
  - direction-of-growth bit
  - protection bits
  - circuitry for address translation



The OS Side of Things

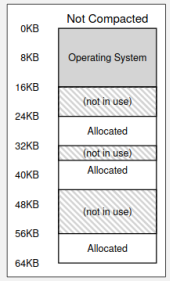
- the hardware provides the mechanisms
  - address translation
  - bounds checking
  - protection bits
- the OS provides the management
  - allocates memory for new processes, deallocates memory from terminated processes
  - grow heap if needed to satisfy a `malloc()`
    - `sbrk()` system call to resize the heap (called by `malloc()`, not programmer)
  - keeps track of free vs used memory via *free list*
  - saves and restores segment registers on context switch
  - provides exception handlers to deal with processes that try to do something illegal

Advantages – Segmentation

- avoids internal fragmentation
- better support for sparse address spaces
  - unused gaps between stack and heap are not tied up
- allows sharing of code segments between processes

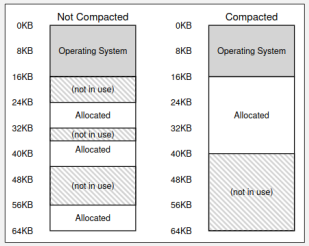
Drawbacks and Limitations – Segmentation

- simplifying assumptions
  - ~~address space~~ segment occupies contiguous chunk of physical memory
  - ~~address space~~ segment fits entirely within physical memory
  - ~~address spaces are all the same size~~
- *external fragmentation*
  - memory gets chopped up into a bunch of little pieces
    - there may be enough total space available for a new process or to grow a segment, but not in one chunk
- a large but sparse heap must still reside entirely in memory
  - unused space tied up with a process



Dealing With External Fragmentation

- periodic *compaction* combines small free chunks but creates new issues
  - expensive (memory- and CPU-intensive)
  - lack of free space between segments makes resizing harder



- algorithmically minimizing fragmentation via clever allocation schemes is a cheaper way to somewhat reduce (but not eliminate) the need for compaction
  - e.g. best-fit, worst-fit, first-fit
  - e.g. buddy algorithm

Memory Allocation Algorithms

- best-fit – find the smallest available chunk that is big enough
  - keeps big chunks for bigger needs
  - tends to end up with lots of little sliver leftovers
- worst-fit – find the largest available chunk (error if it isn't big enough)
  - keeps the remaining unallocated chunks as big as possible
  - biggest chunks get split up first, increasing the chances of not having a big enough chunk later
- first-fit – use the first chunk found that is big enough
  - fast
  - middle ground between best- and worst-fit in terms of fragmentation

Memory Allocation Algorithms

- buddy algorithm
  - only allocate chunks in power-of-two multiples of some smallest possible block size, up to some max order
    - e.g. if smallest possible block is 64K and the max order is 4, then possible chunks are 2' 64K for $0 \leq i \leq 4$ i.e. 64K, 128K, 256K, 512K, 1024K
  - if the desired chunk size isn't available, split a larger chunk in half
  - when freeing memory, if the neighboring block is also free, combine
    - repeat until max size or neighbor isn't free
  - pros and cons
    - keeps external fragmentation low
    - provides cheap compaction
    - re-introduces internal fragmentation due to limited block sizes allocated

Buddy Algorithm Example

order k = block of size 2^k 64K

Step	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K
1. The initial situation.																
1	2^4															
2. Program A requests memory 34 K, order 0. 1. No order 0 blocks are available, so an order 4 block is split, creating two order 3 blocks. 2. Still no order 0 blocks available, so the first order 3 block is split, creating two order 2 blocks. 3. Still no order 0 blocks available, so the first order 2 block is split, creating two order 1 blocks. 4. Still no order 0 blocks available, so the first order 1 block is split, creating two order 0 blocks. 5. Now an order 0 block is available, so it is allocated to A.																
2.1	2^3								2^3							
2.2	2^2								2^2				2^3			
2.3	2^1				2^1				2^2				2^3			
2.4	2^0	2^0			2^1				2^2				2^3			
2.5	A: 2^0	2^0			2^1				2^2				2^3			
3. Program B requests memory 66 K, order 1. An order 1 block is available, so it is allocated to B. 4. Program C requests memory 35 K, order 0. An order 0 block is available, so it is allocated to C.																
3	A: 2^0	2^0			B: 2^1				2^2				2^3			
4	A: 2^0	C: 2^0			B: 2^1				2^2				2^3			

Buddy Algorithm Example

order k =
block of size 2^k 64K

Step	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K

5. Program D requests memory 67 K, order 1.
   1. No order 1 blocks are available, so an order 2 block is split, creating two order 1 blocks.
   2. Now an order 1 block is available, so it is allocated to D.

5.1	A: 2^0	C: 2^0	B: 2^1	2^1	2^1	2^3
5.2	A: 2^0	C: 2^0	B: 2^1	D: 2^1	2^1	2^3

6. Program B releases its memory, freeing one order 1 block.

6	A: 2^0	C: 2^0	2^1	D: 2^1	2^1	2^3

7. Program D releases its memory.
   1. One order 1 block is freed.
   2. Since the buddy block of the newly freed block is also free, the two are merged into one order 2 block.

7.1	A: 2^0	C: 2^0	2^1	2^1	2^1	2^3
7.2	A: 2^0	C: 2^0	2^1	2^2		2^3

8. Program A releases its memory, freeing one order 0 block.

8	2^0	C: 2^0	2^1	2^2	2^3

Buddy Algorithm Example

order k =
block of size 2^k 64K

Step	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K

9. Program C releases its memory.
   1. One order 0 block is freed.
   2. Since the buddy block of the newly freed block is also free, the two are merged into one order 1 block.
   3. Since the buddy block of the newly formed order 1 block is also free, the two are merged into one order 2 block.
   4. Since the buddy block of the newly formed order 2 block is also free, the two are merged into one order 3 block.
   5. Since the buddy block of the newly formed order 3 block is also free, the two are merged into one order 4 block.

8	2^0	C: 2^0	2^1	2^2	2^3
9.1	2^0	2^0	2^1	2^2	2^3
9.2	2^1		2^1	2^2	2^3
9.3	2^2			2^2	2^3
9.4	2^3				2^3
9.5	2^4				