

Homework 7

Solution Key

Problem 7.1

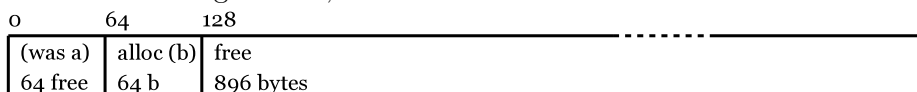
In this problem, you'll simulate a memory allocation algorithm. . . As requests come in, you'll subdivide it, labelling each piece according to size and allocation status. . . .

Hand-simulate the following sequence of allocations (and deallocations), first according to the first-fit algorithm, then according to best-fit, and finally according to worst-fit.

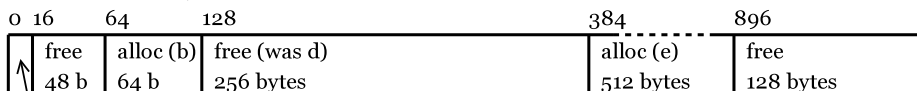
```

a = malloc (64);           f = malloc (64);
b = malloc (64);           g = malloc (128);
free (a);                  h = malloc (32);
c = malloc (16);           free (f);
d = malloc (256);          i = malloc (112);
e = malloc (512);          free (g);
free (d);                  j = malloc (240);
    
```

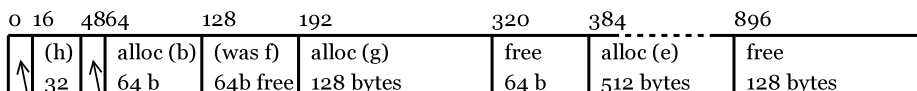
For each of the algorithms, the situation after the first `free` is as follows:



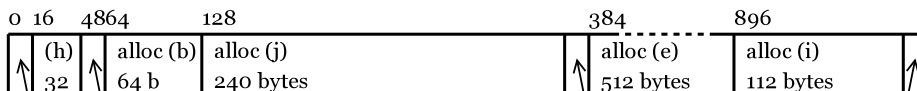
For the first-fit algorithm, the situation develops as follows (snapshots shown after each `free`):



16 (c)

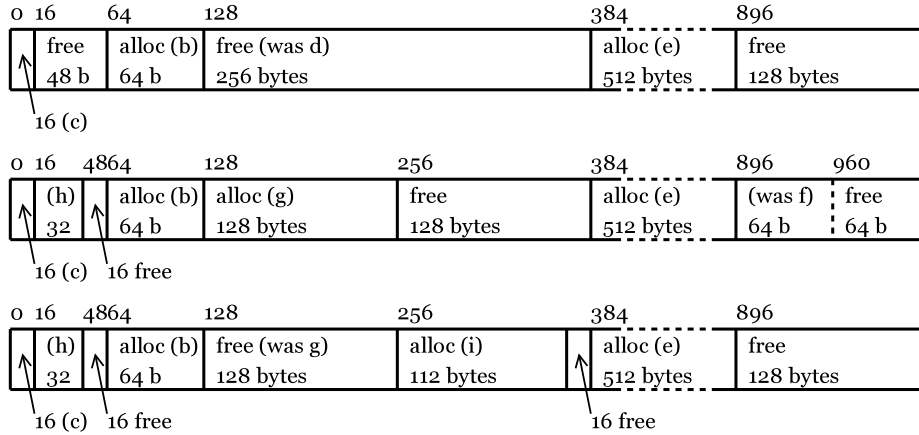


16 (c) 16 free



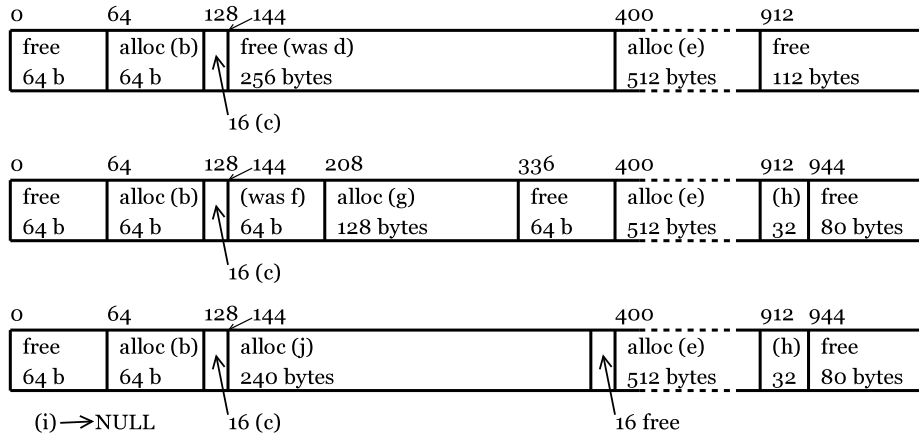
16 (c) 16 free 16 free 16 free

For the best-fit algorithm, the smallest slot that fits is chosen, yielding the following sequence (assuming that the first best-fit is chosen in case of tie):



The final allocation fails due to lack of contiguous available memory, and *j* is set to NULL. If the alternate assumption is made, that the last best-fit is chosen in a tie, *i* gets allocated in the last block and freeing up *g* makes a contiguous block for the *j* allocation—this makes the final snapshot the same as for first-fit, in this case.

For the worst-fit algorithm, the *largest* slot is always chosen. This yields the following snapshots:



In this case, it is the *i* allocation that fails due to lack of *contiguous* available space; however, after *g* is freed, it combines with the adjacent free blocks to make a space big enough that the *j* allocation can be successful, as shown in the final snapshot.

Problem 7.2

Consider the code we wrote in class on Monday (essentially same as on p127 of the MOS book), and the modifications we made to it Wednesday (to be more like p130).

- a. *Write out a series of instructions that would cause the non-semaphore version to break. . . . Once you've written out the table, indicate how the semaphore solution from Wednesday prevents that particular situation.*

This is one typical solution:

Producer	Consumer
<code>item = produce_item()</code>	
<code>check if count == N (it isn't)</code>	
<code>insert_item(item)</code>	
	<code>check if count == 0 (it is)</code>
<code>count = count + 1</code>	
<code>check if count == 1 (it is)</code>	
<code>wakeup(consumer)</code>	
	<code>sleep()</code>

Crucially, the check in the consumer column comes before the increment in the producer column; and the `sleep` in the consumer comes after the `wakeup` in the producer. Once the consumer is asleep, there is nothing to decrement the count, and the count will never again be 1 on the last line of the producer, so the consumer can never wake up.

The semaphore solution prevents this by making the two lines listed in the consumer column along with the decrement into a single atomic event (`down(&full)`).

- b. *Then, separately, draw another diagram of instructions that, first, includes one producer and two consumers, and second, highlights a different problem in the Monday/p127 code that is also solved by the semaphore solution. Indicate how semaphores solve this one, too.*

There are at least two related problems that could arise. This is one:

Producer	Consumer1	Consumer2
item = produce_item()		
count == N? (no)		
insert_item(item)		
count = count + 1		
	count == 0? (no)	
	item = remove_item()	
		count == 0? (no)
		item = remove_item()

The first consumer is interrupted after removing an item but before decrementing the counter, so the second consumer think's it's ok to remove one. This problem is solved by the decrement being bundled into the semaphore test that is performed *before* the removal.

Another problem is as follows:

Producer	Consumer1	Consumer2
item = produce_item()		
count == N? (no, 0)		
insert_item(item)		
count = count + 1 (1)		
count == 1? (yes)		
wakeup(consumer)		
item = produce_item()		
count == N? (no, 1)		
insert_item(item)		
count = count + 1 (2)		
	count == 0? (no)	
	item = remove_item()	
	retrieve count (2)	
		count == 0? (no)
		item = remove_item()
		retrieve count (2)
		subtract one (1)
		store count (1)
	subtract one (1)	
	store count (1)	

Here, *count should* be 0 but will be 1 because the first consumer is interrupted between retrieving the old value and storing the new one. It's solved by making the decrement atomic.