62588
Operating Systems

Mandatory Assignment 2

Group 08

Mark Bidstrup s215782

Xiao Chen s215821

Thor Jensen s215794

Mark Nielsen s204434

Lucas Schoubye s215801

Deadline: 26. October 2022

1) Why is it so important that adjacent free blocks not be left as such? What would happen if they were permitted?

Memory segmentation over time will lead to external fragmentation, where free chunks of memory are scattered across the memory pool. We want to make sure that any adjacent free blocks are consolidated, as otherwise we would get many small chunks too small to assign a program to, making a part of the memory effectively useless while preventing larger programs from being allocated. Specifically, if adjacent blocks are not consolidated, there may be a scenario where there is in fact enough contiguous space in memory to fulfill an allocation request, but where a search algorithm fails to recognize this because the individual blocks are too small.

2) Which function(s) need to be concerned about adjacent free blocks?

If myfree() consolidates adjacent free blocks whenever it is used, it is the only function that will need to be concerned about it.

3) Name one advantage of each strategy.

First-fit: It is fast and easy to implement

Best-fit: Each particular allocation creates the least free/wasted space compared to the other strategies, and should result in a bigger "largest free block", especially when overall memory utilization is high (see a more detailed discussion under question 4)

Worst-fit: Can reduce external fragmentation - since we allocate a chunk out of the largest block, the remaining free block has the highest chance of still being useful (as there should be a 'large' block left over, rather than a tiny one)

Next-fit: It is fast and doesn't overly favor the first half of the memory space (as first-fit does)

4) Run the stress test on all strategies, and look at the results (tests.out). What is the significance of "Average largest free block"? Which strategy generally has the best performance in this metric? Why do you think this is?

The larger the largest free block is, the larger an allocation request can be accepted. We would expect that the largest average free block size would occur under best fit, as best fit always allocates the smallest possible block and leaves the larger blocks free.

Best fit and first fit both do well in the tests and generally result in a higher average largest free block size than the other strategies. In the test results, we see that with a low fill ratio, first fit seems to perform equally as well as best fit (as only a small part of the memory is allocated, and thus there will be large free chunks in the lower half of the memory space), but when the fill ratio

is higher (in almost all the test cases when fill ratio > 50%), best fit has the better results - as we would expect.

5) In the stress test results (see Question 4), what is the significance of "Average number of small blocks"? Which strategy generally has the best performance in this metric? Why do you think this is?

The higher the number of small free blocks that are in the memory, the more memory is effectively wasted as it is unlikely that these blocks can be assigned/allocated to a program. The best performance in this metric thus means a low number of small free blocks. We would expect that worst fit has the best performance (i.e., the lowest number of small free blocks), as worst fit always allocates memory from the largest free block, leaving a large remaining block with a high chance of being useful for further allocations (as opposed to a tiny, useless chunk being left over).

Conversely, we would expect best fit to have the worst performance (i.e., highest number of small free blocks), as each allocation will almost always leave behind a small chunk of memory which will likely be wasted.

The test results support both of the above hypotheses.

6) Eventually, the many mallocs and frees produce many small blocks scattered across the memory pool. There may be enough space to allocate a new block, but not in one place. It is possible to compact the memory, so all the free blocks are moved to one large free block. How would you implement this in the system you have built?

The easiest way to implement this would be to go through each allocated block in order and move them up to the end of the previous block, effectively gathering the used blocks at the start of the memory and the free blocks at the end of the memory. Note that this would necessitate interrupting each process while it was being moved, and that the compaction process would result in significant wasted time for the CPU.

7) If you did implement memory compaction, what changes would you need to make in how such a system is invoked (i.e. from a user's perspective)?

The user would not need to make any active changes in how to use/ invoke the system, but memory compaction might have a negative impact on the perceived speed of processing for the user as running memory compaction can be time consuming. How much the processing speed is negatively impacted with-compaction vs. without-compaction would depend heavily on which strategy is being used, as some strategies (such as best fit) would likely require relatively frequent compaction (with a larger negative impact on perceived processing speed), whereas other strategies e.g., worst fit would not require compaction to be run nearly that frequently (thus having less impact on the processing speed).

8) How would you use the system you have built to implement realloc? (Brief explanation; no code)

Reductions in size can be implemented by changing the size variable of the block in question. If the next block is free, one can then simply update the size and memory ptr of the free block to make the free block larger. If the next block (after then realloc'ed block) is not free, one would need to insert a new free block right after the realloc'ed block, representing the space the original block was reduced by.

If realloc increases the size of a block, we consider two cases:

1) If the next block is free and is large enough, we can simply increase the size of the realloc'ed block and reduce the size of the next free block accordingly (and of course update the ptr of the free block). If the increase in size exactly equals the size of the free block, we will remove that block altogether, i.e. remove that node from the linked list.
2) If the next block is not free, or if it is free but is not large enough for the realloc'ed block to grow into, then we would need to find another free block in memory which is big enough for the realloc request. If such a block exists, we would allocate the required amount of memory from that block, copy over the data from the existing block, and finally free the original block, effectively 'moving' to another location in memory as a result of the realloc request. If there is no free block big enough for the realloc request (but there is enough space across the memory pool as a whole), one could use compaction to consolidate all the free blocks to ensure that there is a free block that is large enough for the realloc request.

9) Which function(s) need to know which strategy is being used? Briefly explain why this/these and not others.

myalloc() needs to know which strategy is being used, since the decision of which block to allocate obviously depends on the strategy. It is the only function which needs this knowledge, as it is the only function that deals with memory allocation (and the strategies are all about allocation).

myfree(), by contrast, does not need to have any knowledge of the strategy, since its only job is to free a given block and potentially consolidate/ coalesce with any free neighbor blocks. The memory status/property functions such as mem_holes(), mem_allocated(), mem_free() etc. also do not need any knowledge of the allocation strategy.

10) Give one advantage of implementing memory management using a linked list over a bit array, where every bit tells whether its corresponding byte is allocated.

The dynamic nature of linked-lists are better to work with than arrays. For example, in case that the memory pool needs to grow, one can simply insert a new node at the end of the list, as opposed to needing to create an entirely new array and then moving/ copying over all of the data. Insertion, deletion and rearrangement of data as a whole is easy since linked lists can be stored non sequentially but appear as if the data is stored right next to each other in a compact way.

It may also be easier to code a memory program using memory segmentation with linked lists than using a bit array, since each block/ node in the linked list is a simple way to represent a chunk of memory, and to keep all of the associated book-keeping data such as a block's alloc status etc. With a bit array, additional data structures would be needed to keep track of book-keeping data.