**CS 4540 — Operating Systems, Spring 2016**
**Assignment #5**

## Introduction

The *dynamic memory allocation* problem is concerned with satisfying a request for a *memory block* of size *n* from a list of available memory blocks (a.k.a. free memory blocks or *memory holes*). It can be solved using several memory allocation strategies, including *first-fit, best-fit,* and *worst-fit* memory allocation strategies.

## Program Specification

Develop a C program that simulates the *allocator* implementing the first-fit memory allocation strategy.

Both used and free memory blocks are represented by *the memory block list*. It is a linked list, with each node of the list including the starting address and the ending address of a memory block plus the identifier of the process using the block (free blocks include no process id). The nodes in the list are ordered based on the starting addresses of the block.

Example in Figure 1 shows the *memory block list* with 3 used blocks and 1 free block for the memory of size 1024 bytes:

1) the occupied block from Byte 0 to Byte 29, used by Process 0 (*processID* = 0);

2) the occupied block from Byte 30 to Byte 44, used by Process 1 (*processID* = 1);

3) the occupied block from Byte 45 to Byte 66, used by Process 2 (*processID* = 2); and

4) the free block from Byte 67 to Byte 1023.

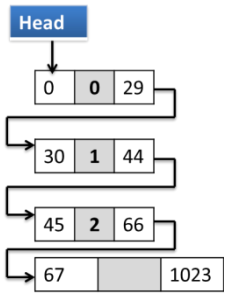



Figure 1. List of memory blocks (a linked list).

The allocator (and your simulation program) works as follows. Once a request for a memory block is received from a process, the allocator gives it the first free block from the memory block list that is large enough to satisfy the request, and updates the memory block list accordingly.

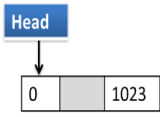
If there is no free block large enough to accommodate the request, the request is rejected by the allocator. (Note that this is a simplifying assumption, making this assignment easier.)

## Example

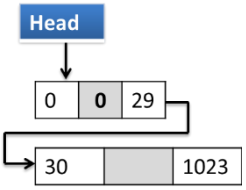
Consider the example shown in Figure 2. The initial state of the memory block list, representing empty memory, is shown in Figure 2a. The first request from Process 0 for a memory block of size 30 bytes results in creating the new list node, and inserting it into the list at the appropriate position, as shown in Figure 2b. Note that the starting address in the free block node (the second node) is updated.

Figure 2c shows the state of the memory block list after two memory blocks (of size 15 and 22 bytes, resp.) are allocated to Processes 1 and 2, respectively. Note that the starting address in the free block node (the fourth node) is updated.

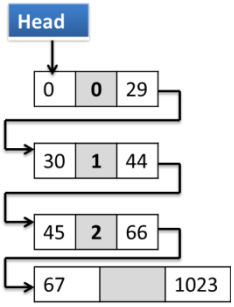
When a process terminates, the memory block it had is deallocated. Deallocation is done by updating the block (node) information, because the block that used to be occupied becomes a memory hole. Figure 2d shows the list just moments after Process 1 terminates and its memory is deallocated (freed).



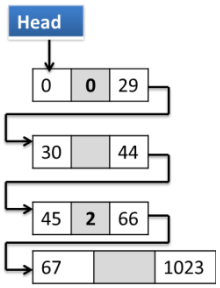

a) The initial memory block list.



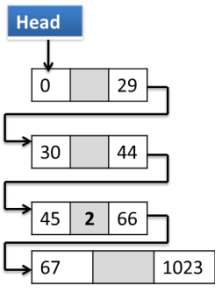

b) After allocating a memory block for P0.



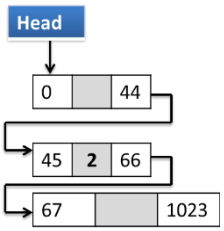

c) After allocating two more memory blocks—for P1 and P2.



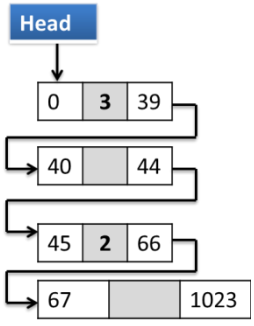

d) After deallocating the memory block from P1.



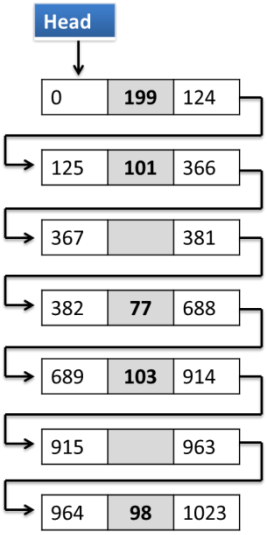

e) After deallocating the memory block from P0



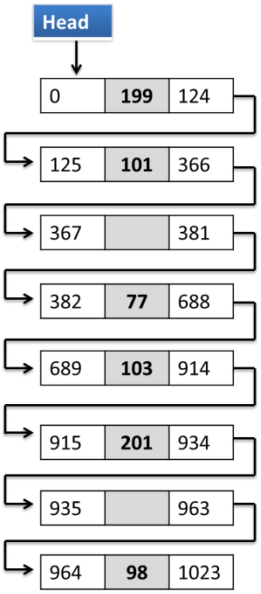

f) Merging consecutive holes.




g) After allocating a memory block for P3.




h) After a sequence of allocation, deallocation, merging operations.




i) After allocating a memory block for P201.

Figure 2. Example illustrating the first-fit memory allocation strategy.

Figure 2e shows the list just moments after Process 0 terminates and its memory is deallocated. However the deallocation process for Process 0 is not completed yet since the newly freed block (formerly used by Process 0) is adjacent to another free block. The neighboring holes (from Byte 0 to Byte 29, and from Byte 30 to Byte 44) *must* be merged right away into a single hole: from Byte 0 to Byte 44. The memory block list after this merge is shown in Figure 2f.

When Process 3 asks for a memory block of size 40 bytes, the allocator gives it space from Byte 0 to Byte 39, resulting in a small hole (from Byte 40 to Byte 44); this is shown in Figure 2g.

Suppose that after a sequence of allocation, deallocation, and merging operations, the state of the memory block list is as shown in Figure 2h. Suppose further that the allocator receives from Process 200 a request for a memory block of size 116 bytes. The only available holes (represented by the third and sixth node) are too small (with 15 bytes and 49 bytes, resp.). Consequently, the request from Process 200 is rejected (ignored) by the allocator.

When Process 201 asks for 20 bytes of memory, the allocator finds enough space in the sixth node (from Byte 915 to Byte 963). The allocator uses 20 bytes of the 49 bytes for Process 201, and leaves the remaining 29 bytes as a memory hole. Figure 2i shows the state of the memory block list after this allocation.

**Implementation Requirements**

1) Run your simulator with the assumption that the memory size is just 1024 bytes.
2) A sequence of process block allocation requests, deallocation requests (upon process termination), and list display (print) requests is given in the file *CS4540-A5-App1.txt* stored in Elearning (also enclosed as Appendix 1 below). This input must *not* be changed (allow read-only access to the input file).
3) To handle a process request for a memory block, the simulator must call the function *alloc(processID, blockSize, …)*. You may use more than two parameters if you wish.
4) To handle deallocation of a memory block for a (terminated) process, the simulator must call the function *dealloc(processId, …)*. You may use more than one parameter if you wish.
5) To handle the list display request, the simulator must call the function *displayList()*, which prints the contents of the memory block list, node by node. The output should be nicely formatted. You might use a tabular format with columns *startAt, endAt, processID*, and the table rows ordered by the starting address (cf. Appendix 2).
6) Your allocator must report how many times it ignored a memory allocation request.

**Hints**

1) You might decide that a doubly-linked list is more appropriate to use than a singly-linked one.
2) You might benefit from the following web pages, discussing linked lists in C:
   a) http://www.thegeekstuff.com/2012/08/c-linked-list-example/
   b) http://www.cprogramming.com/tutorial/c/lesson15.html

**Reporting Requirements**

1) The file created by the *script* command must show the output from *displayList()* calls.

2) In addition to all normal requirements, your Assignment Report must answer (in a separate file) the following questions:

   a) Does the first-fit strategy efficiently manage the memory holes?

   b) Why or why not?

**SLC Report Requirements**

For each program write a full SOFTWARE LIFE CYCLE (SLC) report (analogous to the SLC report presented in class).

Please note that your reporting job is easier due to the following simplifications:

1) The PROBLEM SPECIFICATION section (Step 1) should summarize the specs.
2) The PROGRAM STRUCTURE DESIGN section (Step 2) will have a few modules to name (Substep *2.1. Modules and Their Basic Structure*), and few modules to provide pseudocode for (Substep *2.2.Pseudocode for the Modules*).
3) The sections for RISK ANALYSIS (Step 3), VERIFICATION (Step 4), REFINING THE PROGRAM (Step 7), PRODUCTION (Step 8) and MAINTENANCE (Step 9) can include just 1-2 sentences (analogous in the SLC report presented in class).
4) The CODING section (Step 5) should have as many code refinement levels as needed (the first will just add function headers and trailers, as shown in class).
5) Regarding the TESTING section, you might refer to the file which shows your program output (the one created by *script* command).
6) Using mono-spaced fonts like (Courier) for source code will add a significant value for code readability and quality. Please use this font only for source code in your report.

**Coding, Running and Submission Requirements**

1) Follow *C Code Style Guide* and the proper programming style it requires, including comments, blank lines, indentations, spaces, etc.
2) All programs must be compiled and executed on Ubuntu running within the Oracle VirtualBox.
3) Remember about *Assignment Submission Instructions* (guidelines), to be followed for each program of this assignment.
4) In addition to what *Assignment Submission Instructions* require, provide a *makefile*, and, if needed a README file explaining how to compile and run your program.

**Submission Checklist**

For each program you need to submit:

1) The SLC report (including 9 SLC steps, with as many pseudocode and code refinements as needed);
2) *makefile*, and, if needed a README file explaining how to compile and run the program;
3) The files created by the *script* command (including program output to the terminal, if any);
4) The additional output files (if any in addition to the output to the terminal);

5) Remember about answering (in a separate file) Questions 2a-2b from Reporting Requirements.

Submit your complete assignment package (all files) via Elearning as a zip file. Use `hw<number>_<LastName>.zip` as the format for the name of the zipped file (e.g., `hw1_Smith.zip`)

**-------- Good luck! -----**

## APPENDIX 1. Simulator Input

This sequence of allocate, deallocate, and display requests is the input to your simulation program.

```
allocate (0, 44)
allocate (1, 30)
allocate (2, 17)
allocate (3, 23)
allocate (4, 11)
allocate (5, 14)
allocate (6, 33)
deallocate (0)
deallocate (5)
allocate (7, 21)
allocate (8, 10)
deallocate (3)
allocate (9, 8)
displayList()
deallocate (1)
displayList()
allocate (10, 18)
allocate (11, 9)
allocate (12, 15)
allocate (13, 38)
allocate (14, 18)
allocate (15, 50)
allocate (16, 28)
allocate (17, 33)
allocate (18, 24)
deallocate (16)
allocate (19, 17)
allocate (20, 37)
allocate (21, 44)
deallocate (15)
allocate (22, 16)
allocate (23, 15)
deallocate (13)
allocate (24, 18)
allocate (25, 43)
allocate (26, 33)
allocate (27, 29)
```

```
allocate (28, 45)
allocate (29, 28)
allocate (30, 40)
allocate (31, 38)
allocate (32, 39)
allocate (33, 25)
allocate (34, 48)
allocate (35, 43)
allocate (36, 33)
deallocate (27)
allocate (37, 21)
allocate (38, 45)
deallocate (34)
allocate (39, 24)
allocate (40, 54)
allocate (41, 34)
deallocate (25)
allocate (42, 34)
deallocate (31)
deallocate (20)
allocate (43, 25)
allocate (44, 27)
deallocate (8)
allocate (45, 47)
displayList()
deallocate (32)
allocate (46, 26)
allocate (47, 30)
allocate (48, 25)
deallocate (30)
allocate (49, 29)
deallocate (22)
allocate (50, 26)
displayList()
```

## APPENDIX 2. Example of output (suggestion)

The output of `displayList()` might be as follows:

| startAt | endAt | processID |
|---|---|---|
| 0 | 14 | 0 |
| 15 | 66 | 1 |
| 67 | 169 | |
| . | . | . |
| . | . | . |
| . | . | . |
| 999 | 1010 | |
| 1011 | 1023 | 50 |
| | | |