AOS Assignment 6

List of files created:

1. shell -> xsh_memmgmt.c
2. include -> newmem.h
3. system -> memgetstk.c
4. system -> memcreate.c
5. system -> memkill.c
6. apps -> test.c

Question 9.1:
Write a function that walks the list of free memory blocks and prints a line with the address and length of each block.

Solution:
In this question, we have implemented a new shell command 'memmgmt' where it displays the length and address of each free memory block. As there is a single free block with the entire free memory as it's length, it outputs single block.

Question 9.2:
Replace the low-level memory management functions with a set of functions that allocate heap and stack memory permanently (i.e., without providing a mechanism to return storage to a free list). How do the sizes of the new allocation routines compare to the sizes of getstk and getmem?

Solution:
In this question, we implemented a new shell command which creates a new process 'test' which has implementation of a recursive function, executes the process and then kills it but does not free up the stack and heap memory allocated to the process. The new functions created to allocate stack to the process is memgetstk.c and heap memory is allocated using memget.c.

The new allocation routines differ from getstk and getmem as below:

1. memgetstk() vs getstk():

    - While allocating stack for the process, we have not implemented any memory allocation strategy, as a result of which it follows first fit by default.

- Here, if the size of memory requested is less than or equal to the free memory block size, the memory is allocated using memory partitioning.
- On the other hand, getstk() implements 'last fit'. For our question, the stack is allocated from the lowest free available block.
- This does not follow the XINU convention of process stack growing downward from the highest address of free memory & heap growing upward from the lowest address. Thus stack and heap memory is allocated in a similar fashion here.

2. getmem():

    - When heap memory is allocated for a process we call getmem and freemem when the process exits. For this assignment, we do not call freemem i.e. we do not free up the memory allocated to the process.
    - getmem() allocates heap memory and one needs to free up explicitly while stack memoty gets freed up when process terminates.

As discussed when process ends system implicitly calls kill(), which in turn free up stack space, here we are displaying maximum stack space used by the process.
memkill() kills the prcess without freeing up the stack space. As a result of which free block size is reduced by the process stack space which is 1024 in our case.

When we free up the allocated heap and stack memory in our implementation, it may lead to internal fragmentation within the free memory blocks. There are different techniques which can be employeed to take care of fragmentation problem.

Question 9.7:
Many embedded systems go through a prototype stage, in which the system is built on a general platform, and a final stage, in which minimal hardware is designed for the system. In terms of memory management, one question concerns the size of the stack needed by each process. Modify the code to allow the system to measure the maximum stack space used by a process and report the maximum stack size when the process exits.

Solution:
In this question, we implemented a new shell command which creates a new process ‘test’, executes it and then kills the process but does not free up the stack and heap memory allocated to the process.

In order to measure the stack used by the process, we have initialized the free memory blocks of stack with a value ‘A’ before resuming the process. When the process exits, we check for this value in the stack, and the sum of number of blocks where the value is not ‘A’ gives us the stack utilized by the process.