ADVANCED OPERATING SYSTEMS (FA15-BL-CSCI-P536-36187)

Assignment 6 - Memory Mgmt

Pralhad Sapre & Srivatsan Iyer

Here are the answers to the questions in Exercise of Chapter 9 - Memory management

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.

Files Created

../shell/xsh_printfreelist.c

It is a shell command which traverses memlist to give the desired output.

Demo of functionality

```
xsh $ freemem
Free List:
Block address  Length (dec)  Length (hex)
  0x81041700           20240  0x00004f10
  0x81048bf8           17728  0x00004540
  0x8104dbf8           11856  0x00002e50
  0x81051b10  519652592      0x1ef944f0
```

9.4 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?

Files Modified

To achieve permanent memory allocation we have modified the files

../include/memory.h - to include new pointer variables called *stacktop* and *heaptop*

../system/getstk.c - to allocate space from the highest address in memory

../system/getmem.c - to allocate space from the lowest address in memory

../system/freemem.c - to return just OK, since we are not going to return any allocated memory

../system/meminit.c - to initialize the pointers *stacktop* and *heaptop*

Size comparison

The idea we are using to solve the question is to have one big contiguous block of memory and two pointers *heaptop* and *stacktop*. ***heaptop*** grows from lower addresses towards higher addresses to allocate heap space through the function ***getmem()***. In contrast the ***stacktop*** grows from higher addresses towards lower addresses to allocate stack space through ***getstk()***. Due to the simplicity of the scheme and the fact that no deallocation is possible in the system, the size of these routines is significantly smaller than their original Xinu counterparts. Since no ***memlist*** needs to be maintained to chain together the free memory address in ascending order, the new scheme is highly frugal as far as code length is concerned. An actual metric of code length can be easily obtained from the github link we have submitted. To have an easy understanding of this difference we have kept the code in a separate branch called *xinu_9.4_9.7*.

Demo of functionality

Here is some sample output from the Xinu console

```
xsh $ ps
Stacktop is 2684248044
Pid Name                State Prio Ppid Stack Base Stack Ptr  Stack Size
--- ------------------- ----- ---- ---- ---------- ---------- ----------
  0 prnull              ready   0       0 0x9FFFFFFC 0x9FFFFF44      8192
  1 rdsproc             wait  200       0 0x9FFFDFF8 0x9FFFDAEC     16384
  2 Main process        recv   20       0 0x9FFF9FF4 0x9FFF9F88     65536
  3 shell               recv   50       2 0x9FFE9FF0 0x9FFE9C8C      8192
  4 ps                  curr   20       3 0x9FFE7FEC 0x9FFE7E80      8192
xsh $ echo hellonew
Stacktop is 2684239848
hellonew
xsh $ prodcons 5
Stacktop is 2684231652
Stacktop is 2684230624
Stacktop is 2684229596
```

Produced: 1
Consumed: 1
xsh $ Produced: 2
Consumed: 2
Produced: 3
Consumed: 3
Produced: 4
Consumed: 4
Produced: 5
Consumed: 5

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.

Files Modified and Explanation

We have modified the files ***create.c*** and ***kill.c*** to include the code to count the maximum stack space used by the process. The idea being used is to write the value 0x**FF** in each byte of allocated memory and then to read how many bytes are still 0x**FF** from the top of the stack to calculate the how much a process's stack actually grew in the allocated space. We also noticed that Xinu seems to have some kind of bordering mechanism on the stack. Because of this you can't just start reading for the value 0x**FF** from the top of the stack. Such a naive for loop would terminate giving the impression that the entire stack space was being utilized.

After printing the stack we realized this internal behavior of Xinu that it does have some values on the boundaries of each process's stack. To counter this we have used a fixed offset from the top of the stack and also tried a smarter mechanism to find the longest contiguous sequence of 0x**FF** values. It helps us to gauge how big the process's stack actually grew.

Demo of functionality

xsh $ echo hi
hi
xsh $ Process exited. Total stack memory consumed: 340

xsh $ prodcons 10
Produced: 1
Consumed: 1
xsh $ Produced: 2
Consumed: 2
Process exited. Total stack memory consumed: 148
Produced: 3
Consumed: 3
Produced: 4
Consumed: 4
Produced: 5
Consumed: 5
Produced: 6
Consumed: 6
Produced: 7
Consumed: 7
Produced: 8
Consumed: 8
Produced: 9
Consumed: 9
Produced: 10
Consumed: 10
Process exited. Total stack memory consumed: 208
Process exited. Total stack memory consumed: 220