CSS422 Final Project

Thumb-2 Implementation Work of Memory -Related C Standard Library Functions.

Disclaim: This project is modified based on Professor Munehiro Fukuda's project design.
Last Updated Date: 05/05/2022
Updates: Updated explanations and instructions about provided template files.

1. Objective

Through this final project, you will implement memory-related C standard library functions using Thumb-2. You'll understand the following concepts at the ARM assembly language level:

- CPU operating modes: user and supervisor modes
- System-call handling procedures
- C to assembler argument passing (APCS: ARM Procedure Call Standard)
- Stack operations to implement recursions at the assembly language level
- Buddy memory allocation

2. Project Overview

Using the Thumb-2 assembly language, you will implement four functions of the C standard library (See ***Table 1***) that will be invoked from a C program named driver.c. You will use a provided file, driver_keil.c to test your implementation. **These functions must be coded in the Thumb-2 assembly language.** Some of them can be implemented in stdlib.s running in the unprivileged thread mode (=user mode), whereas the others need to be implemented as supervisor calls, i.e., in the handler mode (= supervisor mode).
For more details, log in one of the CSS Linux servers and type from the Linux shell:
`man 3 functionName` where ***functionName*** is bezro, strncpy, malloc, or free.

Table 1: C standard lib functions to be implemented in the final project

C standard lib functions	In stdlib.s [*1]	As SVC [*2]
`bzero( void *s, size_t n )` writes n zeroed bytes to the string s. If n is zero, bzero( ) does nothing. https://man7.org/linux/man-pages/man3/bzero.3.html	Yes	
`strncpy( char *dst, const char *src, size_t len )` copies at most len characters from src into dst. It returns dst. https://man7.org/linux/man-pages/man3/strncpy.3p.html	Yes	
`malloc( size_t size )` allocates size bytes of memory and returns a pointer to the allocated memory. If successful, it returns a pointer to allocated memory. Otherwise, it returns a NULL pointer. https://man7.org/linux/man-pages/man3/malloc.3p.html		Yes
`free( void *ptr )` Deallocates the memory allocation pointed to by ptr. If ptr is a NULL pointer, no operation is performed. If successful, it returns a pointer to allocated memory. Otherwise, it returns a NULL pointer. https://man7.org/linux/man-pages/man3/free.3p.html		Yes

*1: To be implemented in stdlib.s in the unprivileged thread mode
*2: To be passed as an SVC to SVC_Handler in the privileged handler mode

The driver.c we use is shown in ***Listing 1***. It tests all the above four functions. Please note that printf() in the code should be removed when you test your assembly implementation, because we won't implement the printf( ) standard function.

Listing 1: driver.c program to understand how the required functions work

```
#include <strings.h> // bzero, strncpy
#include <stdlib.h>  // malloc, free
#include <stdio.h>   // printf


int main( ) {
  char stringA[40] = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabc\0";
  char stringB[40];

  bzero( stringB, 40 );
  strncpy( stringB, stringA, 40 );
  bzero( stringA, 40 );
  printf( "%s\n", stringA );
  printf( "%s\n", stringB );

  void* mem1 = malloc( 1024 );
  void* mem2 = malloc( 1024 );
  void* mem3 = malloc( 8192 );
  void* mem4 = malloc( 4096 );
  void* mem5 = malloc( 512 );
  void* mem6 = malloc( 1024 );
  void* mem7 = malloc( 512 );

  free( mem6 );
  free( mem5 );
  free( mem1 );
  free( mem7 );
  free( mem2 );

  void* mem8 = malloc( 4096 );

  free( mem4 );
  free( mem3 );
  free( mem8 );


  return 0;
}
```

3. System Overview and Execution Sequence

3.1. Memory overview

The project maps all code to 0x00000000 – 0x1FFFFFFF in the ARM's ROM space (as the Keil C compiler/ARM assembler does). Additionally, you need to define **multiple dedicated memory spaces** over 0x20001000 – 0x20007FFF in the ARM's **SRAM** space. These dedicates spaces include (1) a heap space, (2) user stack (PSP), (3) SVC stack (MSP), (4) memory control block (MCB) to manage the heap space, and (5) all the SVC-related parameters. See ***Table 2***.

Table 2: Memory overview

Address	Size (hex)	Size (B)	Usage
0x20007B00 – 0x20007B7F	0x00000080	128B	(5) System call table used by svc.s
0x20006C00 – 0x20007AFF	0x00000F00	3.8KB	Not used for now
0x20006800 – 0x20006BFF	0x00000400	1KB	(4) Memory Control Block to manage in heap.s
0x20006000 – 0x200067FF	0x00000800	2KB	Not used for now.
0x20005800 – 0x20005FFF	0x00000800	2KB	(3) SVC (handler) stack: used by all other files
0x20005000 – 0x200057FF	0x00000800	2KB	(2) User (thread) stack: used by driver.c stdlib.s
0x20001000 – 0x20004FFF	0x00004000	16KB	(1) Heap space controlled by malloc/free
0x20000000 – 0x20000FFF	0x00001000	4KB	Keil C compiler-reserved global data
0x00000000 – 0x1FFFFFFF	0x20000000	512MB	ROM Space: all code mapped to this space

Since we compile driver.c (driver_keil.c in your final submission) together with assembly programs, the Keil C compiler automatically reserves driver.c-related global data to some space within 0x20000000 – 0x20000FFF, which makes it difficult to start Process Stack Pointer (PSP) exactly at 0x20005800 toward the lower address and to start Master Stack Pointer (MSP) exactly at 0x20006000 toward the lower address. So, it's sufficient to map PSP and MSP around but not exactly at 0x20005800 and 0x20006000, respectively. For the purpose of this memory allocation, you should declare the space as shown in ***Listing 2***:

Listing 2: The memory space definition in Thumb-2

```
Heap_Size           EQU     0x00005000
                    AREA    HEAP, NOINIT, READWRITE, ALIGN=3
__heap_base
Heap_Mem            SPACE   Heap_Size
__heap_limit

Handler_Stack_Size  EQU    0x00000800
Thread_Stack_Size   EQU    0x00000800
                    AREA   STACK, NOINIT, READWRITE, ALIGN=3
Thread_Stack_Mem    SPACE  Thread_Stack_Size
__initial_user_sp
Handler_Stack_Mem   SPACE  Handler_Stack_Size
__initial_sp
```

3.2. Initialization and system call sequences

(1) **Initialization.** The ARM processor reads the first 8 bytes to set MSP and the next 8 bytes to jump to the Reset_Handler routine (as you studied in the class). You don't have to change the original vector table. Reset_Handler initializes all the data structures you've developed and finally calls __main with ***Listing 3***.

Listing 3: The last two instructions in Reset_Handler (startup_TM4C129.s)

```
LDR     R0, =__main
BX      R0
```

These last two statements are from the original startup_TM4C129.s. Then, the main( ) function in driver.c is invoked.

(2) **System calls.** Whenever main( ) calls any of stdlib functions including bzero(), strncpy(), malloc(), and free(), the control needs to move to stdlib.s. In other words, you need to define these function protocols in stdlib.s, as shown in ***Listing 4***:

Listing 4: The framework of stdlib.s

```
                AREA    |.text|, CODE, READONLY, ALIGN=2
                THUMB

                EXPORT _bzero
_bzero
                ; Your code to implement the body of bzero( )
                MOV             pc, lr ; Return to main( )

                EXPORT _strncpy
_strncpy
                ; Your code to implement the body of strncpy( )
                MOV             pc, lr ; Return to main( )

                EXPORT _malloc
_malloc
                ; Your code to invoke the SVC_Handler routine in startup_TM4C129.s
                MOV             pc, lr ; Return to main( )

                EXPORT _free
_free
                ; Your code to invoke the SVC_Handler routine in startup_TM4C129.s
                MOV             pc, lr ; Return to main( )

                END
```

Among these four functions, you'll implement the entire logic of bzero( ) and strncpy( ) as they may be executed in the user mode. However, the other two functions must be handled as a system call. To do so, you need to write code invoke SVC_Handler in startup_TM4C129.s. Based on the Linux system call convention, **use R7 to maintain the system call number**. Arguments to a system call should follow ARM Procedure Call Standard (APCS), as summarized in ***Table 3***.

Table 3: System Call Parameters

System Call Name	R7	R0	R1
malloc	1	arg0: size	
free	2	arg0: ptr	

SVC_Handler must invoke *_systemcall_table_jump* in svc.s. This in turn means you must prepare the svc.s file to implement *_systemcall_table_jump*. This function initializes the system call table in *_systemcall_table_init* as shown in ***Table 4***.

Table 4: System Call Jump Table

Memory address	System Calls	Jump destination
0x20007B08	#2: free( )	_kfree in heap.s
0x20007B04	#1: malloc( )	_kalloc in heap.s
0x20007B00	#0	Reserved

Each entry in ***Table 4*** records the routine to jump. For this purpose, svc.s needs to import the addresses of these routines, using the code snippet shown in ***Listing 5***.

Listing 5: Entry points to kernel functions imported in svc.s

```
IMPORT  _kfree
IMPORT  _kalloc
```

When called from SVC_Handler, *_systemcall_table_jump* checks R7, (i.e., the system call#) and refers to the corresponding jump table entry, and invokes the actual routine. The merit of using svc.s is to minimize your modifications onto startup_TM4C129.s.

3.3. Structure of your implementation

The software components you need for this final project are summarized in ***Table 5***.

Table 5: A summary of software components implemented in this final project

Source files	Functions to implement	Functions/routines to call	Control[1:0]
driver_keil.c	main( )	→ _bzero → _strncpy → _malloc → _free	11 User/PSP*1
stdlib.s	_bzero: entirely implemented here _strncpy: entirely implemented here _malloc: invokes an SVC _free: invokes an SVC	 → SVC_Handler → SVC_Handler	11 User/PSP*1
startup_TM4C129.s	Reset_Handler SVC_Handler	→ _kinit → _systemcall_table_init → __main →_systemcall_table_jump	00 PriThr/MSP*2 00 Handler/MSP*3
svc.s	_systemcall_table_init: see 3.2.(2) _systemcall_table_jump: see 3.2.(2)	 → _kalloc → _kfree	00 Handler/MSP*3
heap.s	_kinit: initializes memory control blocks _kalloc: buddy allocation coded _kfree: buddy de-allocation coded		00 Handler/MSP*3

*1: running under the unprivileged thread mode, using process stack pointer
*2: running under the privileged thread mode, using master stack pointer
*3: running under the privileged handler mode, using master stack pointer

4. Buddy Memory Allocation and Test Scenario

In this project, you also need to implement the buddy memory allocation in Thumb-2.

4.1. Algorithms

If you have already taken CSS430: Operating Systems, have your OS textbook in your hand and read Section 10.8.1 Buddy System. Since the CSS ordinary course sequence assumes CSS422 taken before CSS430, here is a copy of Section 10.8.1:

10.8.1 Buddy System

The buddy system allocates memory from a fixed-size segment consisting of physically contiguous pages. Memory is allocated from this segment using a power-of-2 allocator, which satisfies requests in units sized as a power of 2 (4 KB, 8 KB, 16 KB, and so forth). A request in units not appropriately sized is rounded up to the next highest power of 2. For example, a request for 11 KB is satisfied with a 16-KB segment.

Let's consider a simple example. Assume the size of a memory segment is initially 256 KB and the kernel requests 21 KB of memory. The segment is initially divided into two buddies—which we will call AL and AR—each 128 KB in size. One of these buddies is further divided into two 64-KB buddies—BL and BR. However, the next-highest power of 2 from 21 KB is 32 KB so either BL or BR is again divided into two 32-KB buddies, CL and CR. One of these buddies is used to satisfy the 21-KB request. This scheme is illustrated in Figure 10.26, where CL is the segment allocated to the 21-KB request.
An advantage of the buddy system is how quickly adjacent buddies can be combined to form larger segments using a technique known as coalescing. In Figure 10.26, for example, when the kernel releases the CL unit it was allocated, the system can coalesce CL and CR into a 64-KB segment. This segment, BL, can in turn be coalesced with its buddy BR to form a 128-KB segment. Ultimately, we can end up with the original 256-KB segment.

The obvious drawback to the buddy system is that rounding up to the next highest power of 2 is very likely to cause fragmentation within allocated segments. For example, a 33-KB request can only be satisfied with a 64-KB segment. In fact, we cannot guarantee that less than 50 percent of the allocated unit will be wasted due to internal fragmentation. In the following section, we explore a memory allocation scheme where no space is lost due to fragmentation.

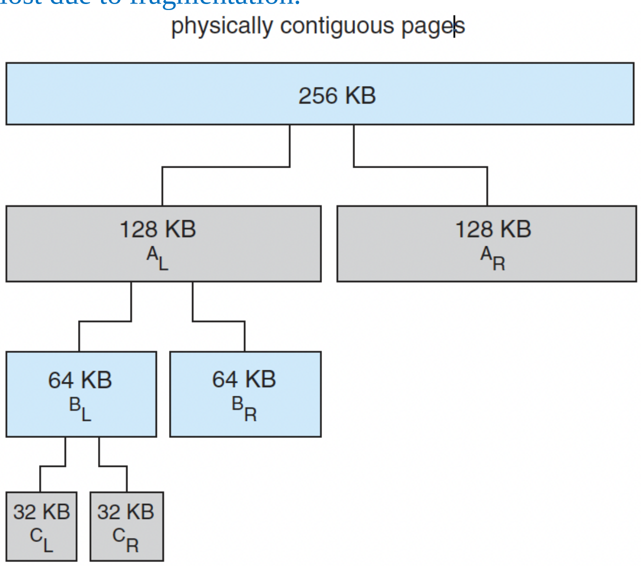


Figure 10.26 Buddy system allocation.

4.2. Implementation over 0x20001000 – 0x20004FFF (Heap space controlled by malloc/free)

As the memory range (Heap space controlled by malloc/free) we use is 0x20001000 – 0x20004FFF (See ***Table 2***), the entire contiguous size is 16KB. This space will be recursively divided into 2 subspaces of 8KB, each further divided into 2 pieces of 4KB, all the way to 32B. Therefore, one extreme allocates 16KB entirely at once, whereas the other extreme allocates 512 different spaces, each with 32 bytes. To address this finest case, (i.e., handling 512 spaces), we allocate a memory control block (MCB) of 512 entries, each with 2 bytes, in the 1KB space over 10x20006800 – 0x20006BFF (Memory control block to manage heap space). Each entry corresponds to a different 32-byte heap space. For instance, MCB entries are defined as:

```
short mcb[512];
```

Then, mcb[0] points to the heaps space at 0x20001000, whereas mcb[511] corresponds to 0x20004FE0. However, each mcb[i] does not have to manage only 32 bytes. It can manage up to a contiguous 16KB space. Therefore, each mcb[i] has the size information of a heap space it is currently managing. The size can be 32 bytes to 16KB and thus be represented with 5 to 16 bits, in other words with mcb[i]'s bits #15 - #4. We also use mcb[i]'s LSB, (i.e., bit #0) to indicate if the given heap space is available (= 0) or in use (= 1). ***Table 6*** shows each mcb[i]'s bit usage.

Table 6: Each mcb entry's bit usage

Bit number	Description
#15 – #4	The heap size this mcb entry is currently managing
#3 – #1	Reserved
#0	0: available, 1: in use

Let's consider a simple memory allocation scenario where main( ) requests 4KB and thereafter 8KB heap spaces with malloc(4096) and malloc(8192). Based on the buddy system algorithm, this scenario allocates 0x2000100 – 0x20001FFF for the first 4KB request and 0x20003000 – 0x20004FFF for the second 8KB request. ***Table 7*** shows this allocation. Only mcb[0], mcb[128], and mcb[256] are used to indicate in-use or available spaces. All the other mcb entries are not used yet.

Table 7: Heap space and mcb contents

Heap Address	Memory Availability		MCB	MCB Address	Contents
0x20001000 – 0x20001FFF	4KB in use		mcb[0]	0x20006800	4097_{10} (0x1001)
0x20002000 – 0x20002FFF	4KB available		mcb[128]	0x20006900	4096_{10} (0x1000)
0x20003000 – 0x20003FFF	8KB in use		mcb[256]	0x20006A00	8193_{10} (0x2001)
0x20004000 – 0x20004FFF					

4.3. Implementation

For each implementation of _kinit, _kalloc, and _kfree, refer to ***Figure 1*** that illustrates how mcb entries are updated.

(1) **_kinit:** The initialization must writes 16384_{10} (0x4000) onto mcb[0] at 0x20006800-0x20006801, indicating that the entire 16KB space is available. All the other mcb entries from 0x20006802 to 0x20006BFE must be zero-initialized (step 1 in ***Figure 1***).

(2) **_kalloc:** Your implementation must use recursions. When _kalloc(size) is called with a size requested, it should call a helper function, say _ralloc, to recursively choose the left half or the right half of the current range until the requested size fits in a halved range. For instance, in ***Figure 1***, the first malloc(4096) call is relayed to _kalloc(4096) that then calls _ralloc(4096, mcb[0], mcb[511]) or _ralloc(4096, 20006800, 20006BFE). See step 2 in ***Figure 1***. The _ralloc call finds mcb[0] at 0x20006800 has 16384B (16KB) available, halves it, and chooses the left half by calling itself with _ralloc(4096, mcb[0], mcb[255]) or _ralloc(4096, 2006800, 200069FE). At this time, make sure that the right half managed by mcb[256] at 0x20006A00 must be updated with 8192 as

its available space (step 3 in ***Figure 1***). Since the range is still 8192 bytes > 4096 bytes, _ralloc chooses the left by calling itself with _ralloc(4096, mcb[0], mcb[127]) or _ralloc(4096, 20006800, 200068FE). Make sure that the right half managed by mcb[128] at 0x2006900 is updated to 4096. The left half in the range between mcb[0]-mcb[17] or 0x20006800-200068FF fits the requested size of 4096. Therefore, ralloc( ) records 4097_{10} (0x1001) into mcb[0] at 0x20006800-0x20006801 (step 4 in ***Figure 1***).

The second malloc(8192) is handled as follows: _kalloc(8192) calls _ralloc(8192, mcb[0], mcb[511]) or _ralloc(8192, 20006800, 20006BFE) (step 5 in ***Figure 1***) to choose the right half with _ralloc(8192, 20006A00, 20006BFE), because mcb[0] at 0x20006800-0x2006801 has a value of 4097 indicating that the left half (0x20006800 – 0x200069FE) is in use. Since mcb[256] at 0x20006A00-0x20006A01 is available, _ralloc saves 8193 (0x2001) there (step 6 in ***Figure 1***).

(3) **_kfree:** Your _kfree implementation must also use recursions. The _kfree(*ptr) function calls a helper function, _rfree(the corresponding mcb[]). If main( ) calls free(0x20001000), it is relayed to _kfree(0x20001000) that calls _rfree(mcb[0]) or _rfree(0x20006800) to reset its bit #0 from in-use to available (step 7 in ***Figure 1***). Then, check its right buddy at mcb[128] (or 0x20006900). If its bit #0 is 0, indicating the availability, zero-reinitialize mcb[128] at 0x20006900 and make sure that mcb[0] at 0x20006800 shows an availability of 8192 bytes (step 8 in ***Figure 1***). Recursively check the buddy at higher layers. So, the next higher layer's buddy is mcb[256]-mcb[511] at 0x2006A00-0x2006BFE. Check mcb[256]'s contents, (at 0x20006A00-0x20006A01). In ***Figure 1***, the content is 0x2001, showing that 8KB is being occupied. Therefore, stop _kfree's recursive calls.

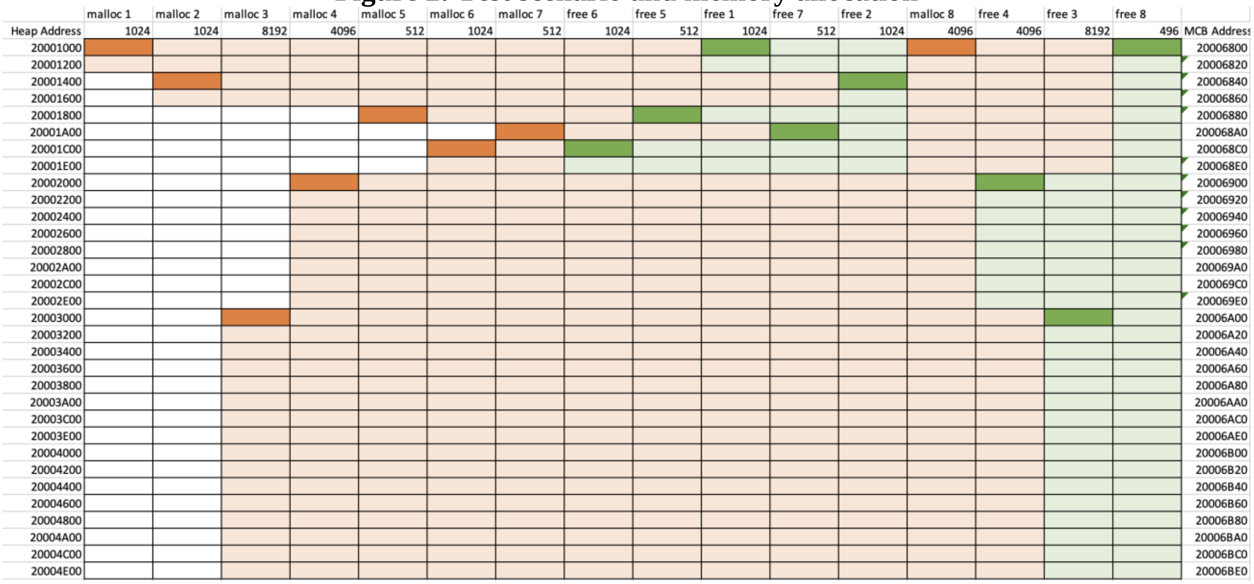
Figure 1: Recursive _ralloc/_rfree calls, each updating mcb entries

		step 1	step 2	step 3	step 4	step 5	step 6	step 7	step 8
		_kinit( )	_kalloc(4096)			_kalloc(8192)		_kfree(20001000)	
			_ralloc(4096, 2006800, 2006BFE)	_ralloc(4096, 2006800, 20069FE)	_ralloc(4096, 2006800, 20068FE)	_ralloc(8192, 20068, 2006BFE)	_ralloc(8192, 2006A00, 2006BFE)	_rfree(20006800)	recursive _rfee(20006800)
mcb[ ]	MCB Address								
mcb[0]	0x20006800	0x4000	0x4000	0x2000	0x1001	0x1001	0x1001	0x1000	0x2000
	:	0x0000	0x0000						
mcb[127]	0x200068FE	0x0000	0x0000						
mcb[128]	0x20006900	0x0000	0x0000		0x1000	0x1000	0x1000	0x1000	0x0000
	:	0x0000	0x0000						
mcb[255]	0x200069FE	0x0000	0x0000						
mcb[256]	0x20006A00	0x0000	0x0000	0x2000	0x2000	0x2000	0x2001	0x2001	0x2001
	:	0x0000	0x0000						
mcb[383]	0x20006AFE	0x0000	0x0000						
mcb[384]	0x20006B00	0x0000	0x0000						
	:	0x0000	0x0000						
mcb[511]	0x200006BFE	0x0000	0x0000						

4.4. Test Scenario

Looking back to ***Listing 1*** (driver.c code example), you are supposed to verify your Thumb-2 implementation of malloc( ) and free( ) with repetitive system call invocations that allocate/deallocate mem1 – mem8 spaces. ***Figure 2*** illustrates how the heap space is allocated and deallocated when you run driver.c. Orange indicates allocated spaces and green means de-allocated spaces.

Figure 2: Test scenario and memory allocation



5. Implementation Steps, Timeline, and Submissions

Since it is hard to implement everything in assembly code at once, the final project will take the following two parts. To work on your project, distinguish the following **three versions** of driver.c program as shown in ***Table 9***. They are all provided and available from Canvas→files→Project→code.

Table 9: Provided driver programs

File name	Tasks
driver.c	This is a C program that can be compiled with gcc and executable on Linux. You used this file to understand how the required functions work. **You DO NOT need to change this file at all.**
driver_cpg.c	This is a C program that should be used for testing your heap.c in Part 1 toward your midpoint report. **The difference from driver.c is:** - **malloc( ) and free( ) are renamed to _malloc( ) and _free( ),** so that the compiler can use your own implementation of _malloc( ) and _free( ). - printf( ) are included to verify your implementation.
driver_keil.c	This is a C program that can be compiled with Keil C compiler and executable with your ARM/THUMB-2 assembly code. You will use this file to test stdlib.s implementation in Part 1 toward your midpoint report. You will also use this file to test the final implementation of stdlibs., heap.s, and svc.s in the Part 2 work. **The difference from driver.c is:** - **all stdlib functions bzero( ), strncpy( ), malloc( ), and free( ) are renamed to _bzero( ), _strncpy( ), _malloc( ), and _free( ),** so that the compiler can use your own implementation.

5.1. Part 1 toward the midpoint report (due on 2nd class date in week 7)
Part 1 intends to help you understand and develop the following two features:

(1) The reset sequence from the assembly language level all the way to main( ) in C that calls back down to stdlib.s in the assembly language level.

startup_tm4c129.s → main( ) in driver_keil.c → stdlib.s

Your actual work on Keil uVision is summarized below in ***Table 10***.

Table 10: Keil uVision work toward the midpoint report

Files you will work on	Tasks
startup_tm4c129.s	Revise the Reset_Handler routine as follows: - Set up and switch PSP (Process Stack Pointer) - Call __main.
stdlib.s	_bzero and _strncpy: Receive arguments from main( ), based on APCS, and **complete the entire implementation within stdlib.s.** _malloc and _free: Receive arguments from main( ), based on APCS, **but does nothing by simply returning to main( ).** **You can use the provided** stdlib_template.s to implement stdlib.s.

For this task, you need to build your project in Keil uVision.
In Keil uVision, start the debugger and take a memory snap of stringA and stringB after an execution.

(2) Implement the buddy memory allocation using C language
Use driver_cpg.c that calls _malloc( ) and _free( ) in heap.c. You can find heap_template.c in Canvas→files→Project→code. This is a template that hopefully makes it easy for you to implement the buddy memory allocation in C. Your C implementation must use a recursion. When you complete your C programs, rename heap_template.c to heap.c. ***Table 11*** summarizes the implementation in Part 1.

Table 11: Linux C programming work toward the midpoint report

Files you will work on	Tasks
driver_cpg.c	No need to change.
heap.c	_malloc( ) and _free( ) in heap.c will internally call _kinit( ), _kalloc( ), and _kfree( ). As mentioned in Section 4.3, _kalloc( ) and _kfree( ) will use recursive _ralloc( ) and _rfree( ) helper functions. In Part 2 (section 5.2), _kinit( ), _kalloc( ), _ralloc( ), _kfree( ), and _rfreee( ) will be implemented using ARM/THUMB-2 in heap.s.

For this task, you SHOULD NOT use Keil uVision. You need to compile and run your code as follow:
gcc driver_cpg.c heap.c -o ./a.out
./a.out

Submission Items:
For Part 1, please submit the following materials listed in ***Table 12***.

Table 12: Part-1 Submission and Grading

Materials	Remarks	Grade points (out of 25pts)
startup_tm4c129.s	From your Keil uVision project	2pts
stdlib.s	From your Keil uVision project	5pts
Two memory snapshots: stringA and stringB	From your Keil uVision project	4pts
heap.c	From your Linux C program	10pts
a.out execution results	From your Linux C execution	4pts

5.2. Part 2 toward the final report (due on 2nd class date in week 11, i.e., final's week)

Part 2 intends to complete all assembly components using ARM/THUMB-2. Your tasks in Part 2 are summarized below in ***Table 13***.

Table 13: Part-2 Work Items

Files you will work on	Tasks
startup_tm4c129.s	Correct the Reset_Handler routine if necessary. Thereafter, **add subroutine calls**: - _kinit: initialization in heap.s - _systemcall_table_init: initialization in svc.s (***Table 4*** in Section 3.2.(2)) **Implement the following routine**: - SVC_Handler: invoke _system_call_table_jump in svc.s
driver_keil.c	No need to change.
stdlib.s	Correct _bzero and _strncpy if necessary. _malloc and _free: Receive arguments from main( ), based on APCS and **relay each call to SVC_Handler.**
svc.s	Refer to Section 3.2.(2). Based on the system call # in R7, jump to the corresponding function through the system call jump table shown in ***Table 4***.
heap.s	Implement the following 5 routines, based on the C implementation in heap.c. _kinit: mcb initialization _kalloc: the entry point to invoke the _ralloc recursive helper function _ralloc: a recursive helper function to allocate a space _kfree: the entry point to invoke the _rfree recursive helper function _rfree: a recursive helper function to free the space and merge the buddy space if possible

Test all your assembly language implementation with driver_keil.c on Keil uVision's debugger session. Take all memory snapshots of mcb addresses corresponding to mem1 – mem8 upon their allocation and deallocation.

Submission Items:

Please submit the following materials listed in ***Table 14***.

Table 14: Part-2 Submission and Grading

Materials	Remarks	Grade points (out of 75pts)
Your zipped Keil uVision project (47pts)	startup_tm4c129.s (9pts) Reset_Handler SVC_Handler	 3pts 6pts
	driver_keil.c (0pt)	0pt (provided code)
	stdlib.s (0pt) _bzero( ) _strncpy( ) _malloc( ) _free( )	 0pt (provided code) 0pt (provided code) 0pt (provided code) 0pt (provided code)
	svc.s (10pts) _systemcall_table_init _systemcall_table_jump	 5pts 5pts
	heap.s (28pts) _kinit _kalloc _kfree	 4pts 12pts 12pts

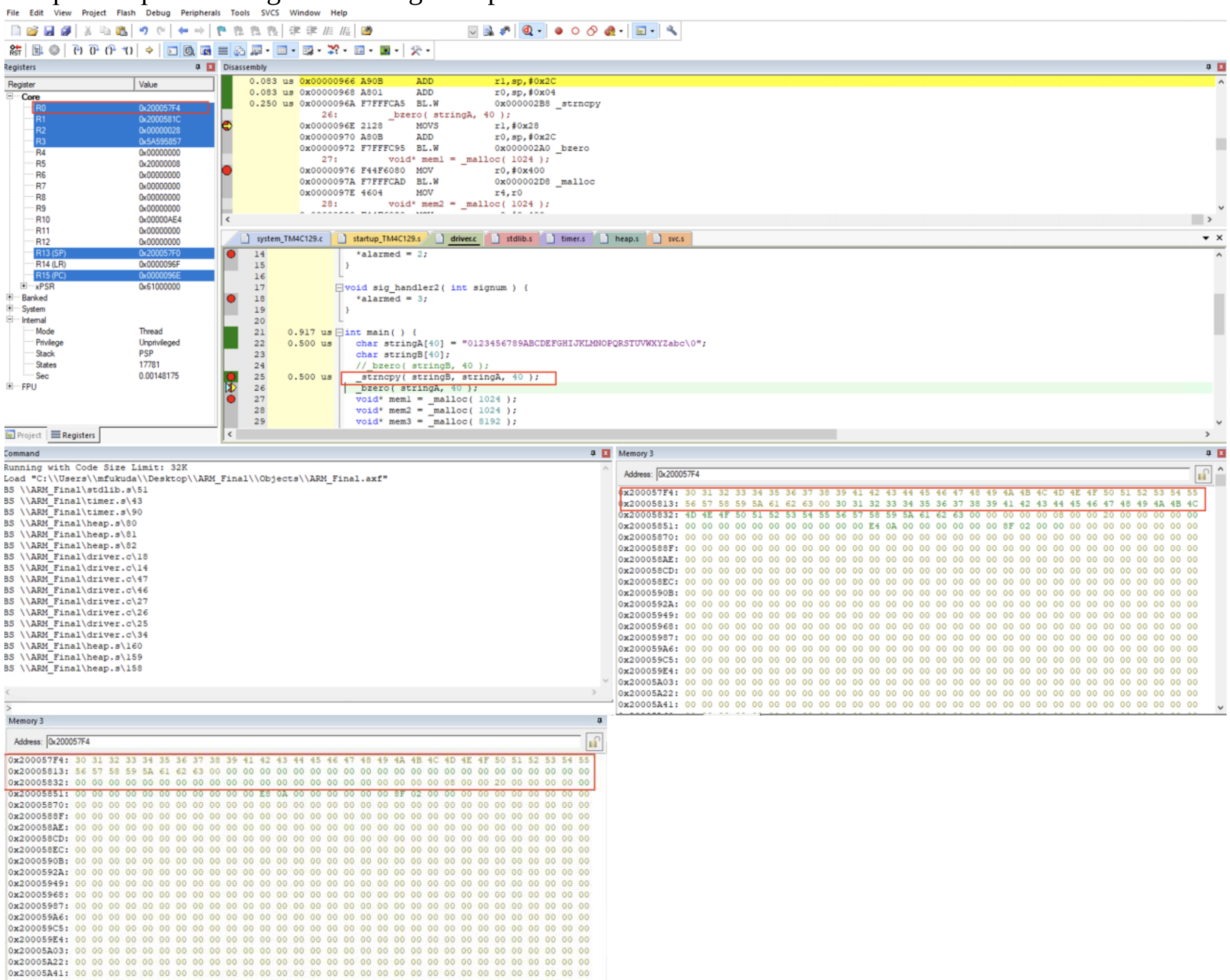
Execution snapshots (16pts)	_strncpy(stringB, stringA, 40);
	_bzero(stringA, 40);
	void* mem1 = _malloc(1024);
	void* mem2 = _malloc(1024);
	void* mem3 = _malloc(8192);
	void* mem4 = _malloc(4096);
	void* mem5 = _malloc(512);
	void* mem6 = _malloc(1024);
	void* mem7 = _malloc(512);
	_free(mem6);
	_free(mem5);
	_free(mem1);
	_free(mem7);
	_free(mem2);
	void* mem8 = _malloc(4096);
	_free(mem4);
	_free(mem3);
	_free(mem8);
		0pt (provided code)
		0pt (provided code)
		1pt
		1pt
		1pt
		1pt
		1pt
		1pt
		1pt
		1pt
		1pt
		1pt
		1pt
		1pt
		1pt
		1pt
		1pt
		1pt

Documentation (12pts)	A two-page summary of your implementation
	- Narratives
	  o What you implemented.
	  o What was missing.
	- Any Diagrams (at least one)
		
		5pts
		5pts
		2pts

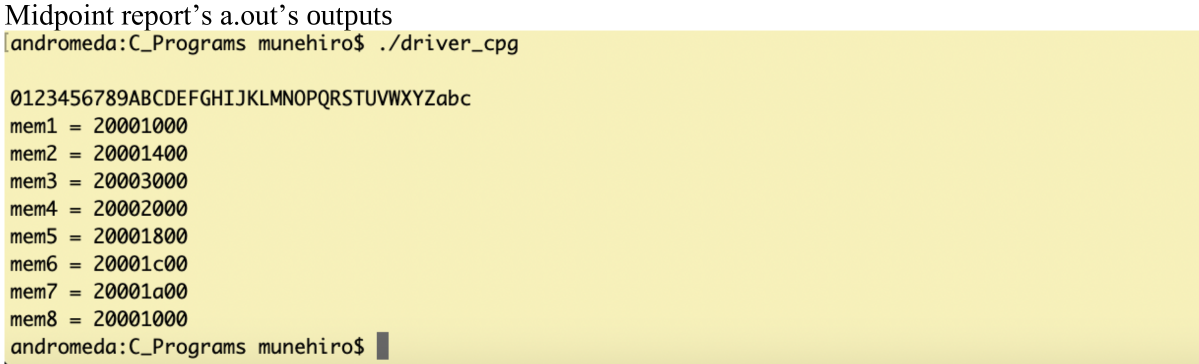
5.3. Execution Snapshots

To clarify what you need to turn in execution results, sample snapshots from the key answer are given below. Don't reuse them. **Any reuse of these snapshots below will result in an academic misconduct.**

(a) Midpoint report's stringA and stringB snapshots



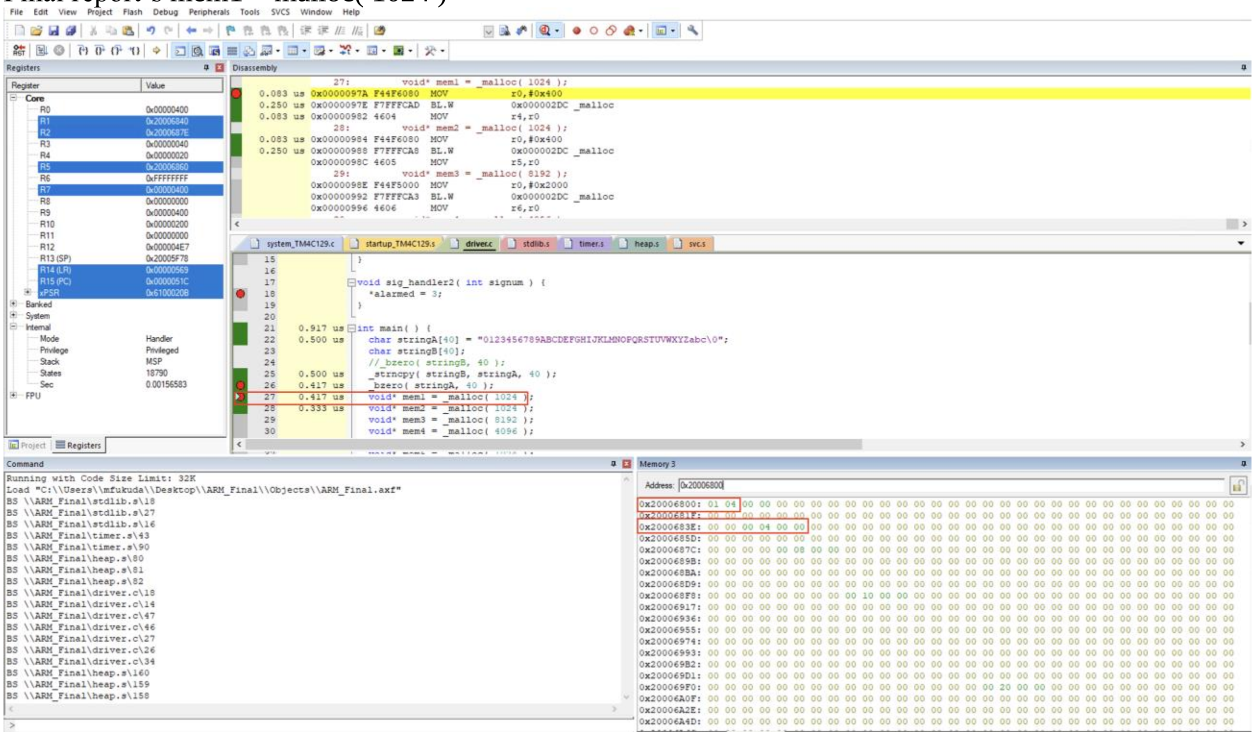
(b) Midpoint report's a.out's outputs



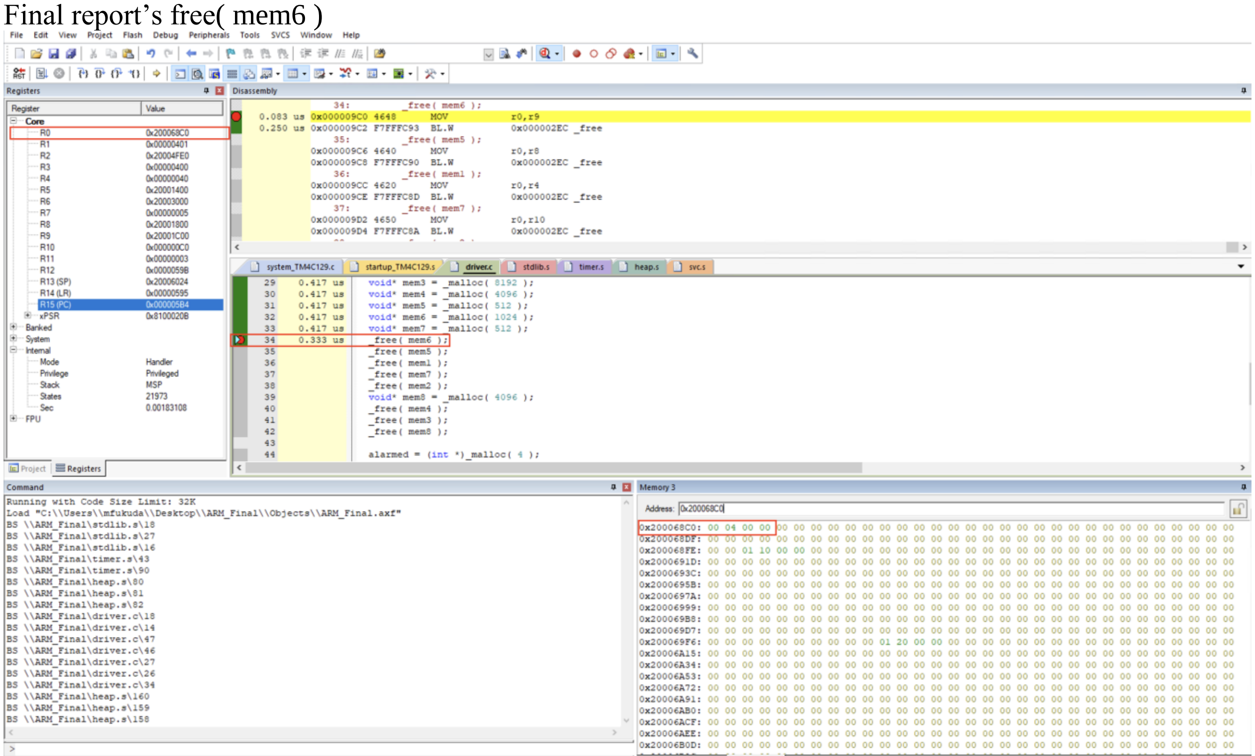
```
andromeda:C_Programs munehiro$ ./driver_cpg

0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabc
mem1 = 20001000
mem2 = 20001400
mem3 = 20003000
mem4 = 20002000
mem5 = 20001800
mem6 = 20001c00
mem7 = 20001a00
mem8 = 20001000
andromeda:C_Programs munehiro$
```

(c) Final report's mem1 = malloc(1024)



(d) Final report's free(mem6)



6. Final notes

(1) Follow the final project specification.
   a. Use the memory spaces exactly specified in this document.
   b. Use the function and routine names specified in this document.
   c. Attach the execution results as specified in this document (see Tables 12 and 14).

(2) Check Canvas→files→Project→code folder for additional materials.

(3) Start your implementation early and keep up your plan.