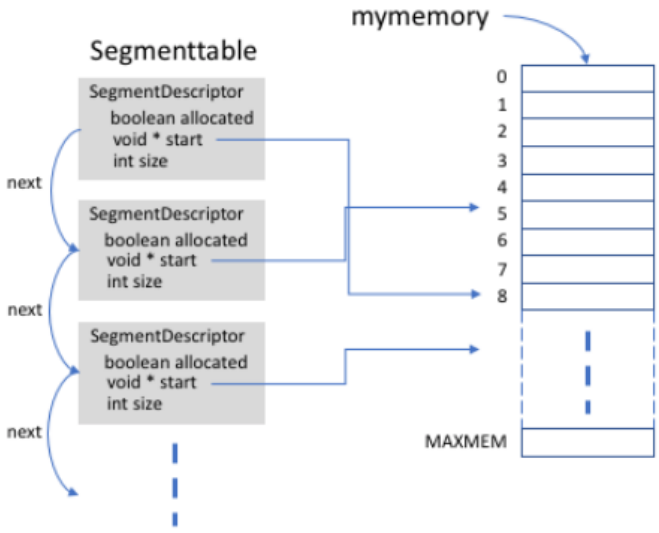
Assessment 1 Memory Management

In this assessment, you will implement a simple memory management and allocation system.

You will implement your own simplified version of the function malloc() that allows you to allocate memory segments, your version of function free() to deallocate these segments, and a function defrag() that defragments your “memory”.

For the purpose of this assessment, our “memory” is a large array of bytes of length MAXMEM. In order to represent in your program how memory is allocated, an important part of this assessment is to manage a memory segmentation table that records which segments of memory are allocated and which parts are free. This segmentation table is a list of segment descriptors:



Each segment descriptor contains a pointer “start” that points to the first byte of the allocated part in mymemory, and a variable “size” that records the number of bytes in the allocated segment.

At the start of your program, the segmentation will be one entry in this segmentation table that indicates that your whole “memory” is one single free segment. An allocated segment is characterised by the pointer variable, for which the memory is allocated. The segmentation table can be implemented as a linked list, so that entries in this list can be dynamically added and removed.

The complete public interface for this memory management system consists of the following functions:

`void initialize()`

- this function initialises the segmentation table and the memory array, it has to be called first before the other functions can be used.

`void * mymalloc ( size_t size)`:

- this function “allocates” memory of a particular size and returns a pointer to the first byte of the allocated segment

`void myfree ( void * ptr)`:
- frees a previously allocated memory

`void mydefrag( void ** ptrlist )`:
- defragments the memory and compacts smaller segments into larger ones

How to start your assessment

Two files are provided for this assessment: **`mymemory.c`**and **`mymemory.h`**, containing basic data structures for the implementation of the segmentation table and the memory management. In **`mymemory.h`**, the structure for a segment descriptor is provided:

```
typedef struct segmentdescriptor {
   Byte     allocated ;
   void   * start ;
   size_t   size  ;
   struct segmentdescriptor * next ;
} Segment_t ;
```

Make yourself familiar with this structure. The segment table can be implemented as a dynamic list (using the “next” pointer to link descriptor nodes together). The following attributes encode essential information about a segment:
- The attribute “**`allocated`**” represents a Boolean (in **`mymemory.h`**, the two macros **`TRUE`** and **`FALSE`**are defined and should be used for “**`allocated`**”.
- The attribute “**`start`**” points to the byte in mymemory where the allocated segment starts
- The attribute “**`size`**” defines the number of bytes of a segment

We use the pointer variable “**`segmenttable`**” to point to this list of segment descriptors. Please note that this pointer variable is declared as “**`extern`**” in mymemory.h :

```
extern Segment_t * segmenttable ;
```

Recall, that “extern” makes this variable global in your program. Recall, that there can be many “extern” declarations, but only one definition of this variable may and must exist in your code base of your program, you can find this definition in file mymemory.c :

```
Segment_t * segmenttable = NULL ;
```

Your “memory” is an array of bytes of size MAXMEM. It is also declared “extern” in mymemory.h:

```
extern Byte mymemory [ MAXMEM ] ;
```

The corresponding definition is contained in memory.c:

```
Byte mymemory [MAXMEM] ;
```

Assessment Requirements

CGS D3-D1

a) Implement the function “initialize()” in mymemory.c. This function has to initialize the memory and the segmentation table:

- As the first step, initialize the complete memory array **`mymemory`** with ‘\0’.
- As the second step, initialise the segment table. We assume that initially, there exists only one segment descriptor that describes the whole memory as one free segment:
  - allocate the segment descriptor
  - set the attributes “allocated”, “start” and “size” accordingly

b) Implement two functions that print the content of the memory and the segmentation table
- Function “void printsegmenttable()”
  - This function traverses the segmentation table and prints out the information of each segment descriptor. Try to create a printout that looks, for example, similar to the following:

    Segment 0
    allocated = FALSE
    start = 0x561188230060
    size = 1024

Please note: In order to print a pointer value, you can use the print format **`"%p"`**.

- Function “void printmemory()”
  - This function traverses the memory array “mymemory” and prints the content of each byte. Please check whether characters are printable (a function isprintable() is provided in mymemory.c) and for any non-printable characters, print a ‘.’

  - You can choose your own format for printing the memory content.
    As a suggestion: A convenient form for a printout can be the following, which prints the content of 10 bytes per line in both hexadecimal (“%02x”) and as a character (“%c”), and the number of the first byte in the line:

```
[   0] 00 00 00 00 00 00 00 00 00 00 | ..........
[  10] 00 00 00 00 00 00 00 00 00 00 | ..........
[  20] 00 00 00 00 00 00 00 00 00 00 | ..........
[  30] 00 00 00 00 00 00 00 00 00 00 | ..........
[  40] 00 00 00 00 00 00 00 00 00 00 | ..........
[  50] 00 00 00 00 00 00 00 00 00 00 | ..........
[  60] 00 00 00 00 00 00 00 00 00 00 | ..........
[  70] 00 00 00 00 00 00 00 00 00 00 | ..........
[  80] 00 00 00 00 00 00 00 00 00 00 | ..........
[  90] 00 00 00 00 00 00 00 00 00 00 | ..........
[ 100] 00 00 00 00 00 00 00 00 00 00 | ..........
[ 110] 00 00 00 00 00 00 00 00 00 00 | ..........
[ 120] 00 00 00 00 00 00 00 00 00 00 | ..........
   Etc …
```

```
[1010] 00 00 00 00 00 00 00 00 00 00 | ..........
[1020] 00 00 00 00
```

In order to print a hexadecimal number, you can use the print format **"%02x"**.
In order to test and demonstrate your implementation, write a test program containing the main() function, and call it "shell.c".

Your test program shell.c should perform the following three actions:

- Call initialize() to initialize the segmentation table and your memory
- Print the segmentation table
- Print the memory

Include any header files required, such as mymemory.h, in your test program shell.c (do NOT include any .c files!!).
In mymemory.h, add all required function prototypes as forward references (see examples at the end of the file).

Your shell.c may consist of the following statements:

```
#include <stdio.h>
#include <string.h>
#include "mymemory.h"


int main()
{
   printf ( "shell> start\n");

   initialize() ;


   printmemory() ;
   printsegmenttable() ;

   printf ( "shell> end\n");

   return 0;
}
```

Submission for CGS D3-D1

In order to achieve at least a D3, your submission must include a make file for building your solution, the required header and program files. Your solution should compile and run. Write a brief report that contains a step-by-step explanation how to run the submission. Provide an explanation what the result of such an execution is.
Submit your test program, shell.c, as well as mymemory.c, mymemory.h, and a Makefile that allows your program to be compiled. Put files into a directory CGS_D3_D1.

CGS C3-C1

a) Implement the function mymalloc().

`void * mymalloc ( size_t size)`:

- this function "allocates" memory of a particular size and returns a pointer to the first byte of the allocated segment in mymemory.

This function returns a pointer to a location in your memory array. This function is used in the same way as the C function malloc().

b) Implement functions to manage your segmentation table as a linked list. Use these functions in the implementation of mymalloc():

- Function
  `Segment_t * findFree ( Segment_t * list, size_t size )`
  - This function searches for a segment in the list that is (a) free (allocated == FALSE), and (b) is at least as large as the required size expressed by parameter "size"
  - This function either returns a pointer to a found segment descriptor, or NULL if no such segment exists any more

- Function
  `void insert ( Segment_t * oldSegment, Segment_t * newSegment )`
  - This function inserts new segment descriptors after an existing segment descriptor

Explanation:
Function mymalloc() allocates a segment of your memory array mymemory. It first has to find a free segment in the segment table. If such a segment is found, it has to test whether this segment is large enough. If yes, then the found free segment has to be split into two smaller segments:

(a) the first sub-segment is the allocated segment for the memory request, and
(b) the second sub-segment describes the remaining free memory from the original segment.

You have to create a new segment descriptor and insert it into that segmentation table, as there is now an additional segment after the split.

Extend your test program shell.c with, at least, the following steps:

- Allocate memory for character array 1
- Allocate memory for character array 2

```
#include <stdio.h>
#include <string.h>
#include "mymemory.h"


int main()
{
   printf ( "shell> start\n");

   initialize() ;

   char * ptr1 = (char *) mymalloc ( 10 ) ;
   strcpy (ptr1, "this test");
   printf( "shell> content of allocated memory: %s\n", ptr1 ) ;


   char * ptr2 = (char *) mymalloc ( 10 ) ;
   strcpy (ptr2, "this test");
   printf( "shell> content of allocated memory: %s\n", ptr2 ) ;


   printmemory() ;
   printsegmenttable() ;

   printf ( "shell> end\n");

   return 0;
}
```

Submission

For a CGS C3, submit shell.c, mymemory.c, mymemory.h, and a Makefile that allows your program to be compiled. Put files into a separate directory CGS_C3_C1.

Parts missing in your solution may reduce the CGS mark. Try to get with your implementation as far as possible.

CGS B3-B1

Implement the function myfree().

`void myfree ( void * ptr):`

- frees a previously allocated memory

The function myfree (void * ptr) takes a pointer as parameter, “ptr”. This is the pointer that points to a location in your memory array “mymemory”, where the allocated segment starts. In the segmentation table, each segment descriptor has this pointer recorded as well (as the “start” pointer). In your implementation, you have to search your segmentation table for this entry, using the parameter provided in the myfree() function.

Implement a function to manage your segmentation table as a linked list. Use these functions in the implementation of myfree():

- Function
  `Segment_t * findSegment ( Segment_t * list, void * ptr )`

  - This function searches for a segment in the list where list->start == ptr
  - This function either returns a pointer to a found segment descriptor, or NULL if no such segment exists

Extend your test program shell.c with the following steps:

```
#include <stdio.h>
#include <string.h>
#include "mymemory.h"


int main()
{
   printf ( "shell> start\n");

   initialize() ;

   char * ptr1 = (char *) mymalloc ( 10 ) ;
   strcpy (ptr1, "this test");
   printf( "shell> content of allocated memory: %s\n", ptr1 ) ;


   char * ptr2 = (char *) mymalloc ( 10 ) ;
   strcpy (ptr2, "this test");
   printf( "shell> content of allocated memory: %s\n", ptr2 ) ;


   printmemory() ;
   printsegmenttable() ;

   free ( ptr1 ) ;

   printmemory() ;
   printsegmentationtable() ;

   free ( ptr2 ) ;

   printmemory() ;
   printsegmentationtable() ;

   printf ( "shell> end\n");

   return 0;
}
```

Submission

For a CGS B3, submit shell.c, mymemory.c, mymemory.h, and a Makefile that allows your program to be compiled. Put files into a separate directory CGS_B3_B1. Parts missing in your solution may reduce the CGS mark. Try to get with your implementation as far as possible.

CGS A5-A1

Implement the function mydefrag(), that defragments the memory

`void mydefrag( void ** ptrlist )`:

- defragments the memory and compacts smaller segments into larger ones

Please note that you have to manipulate the segmentation list (moving free memory entries, combining them into larger segments). If you move allocated memory segments, you have to redirect (a) the start pointers in the segment descriptors of the segmentation table and (b) the pointer variables in your program to the changed start positions of the memory segments as well – please note the pointer-pointer list parameter of mydefrag() that should be used for this purpose.

Function mydefrag() expects a list of pointers as parameter – this is used to redirect any pointer in your program that points to locations in your memory array.

Please note: instead of using a pointer list, you can also investigate how to use a variable number of parameters in C functions and provide your solution in such a form.

Optional: you may want to implement further helper functions to manage your segmentation table as a linked list. Use these functions in the implementation of mydefrag():

- Function (two possibilities)

```
int delSegment ( Segment_t * list, void * ptr )
int delSegment ( Segment_t * list, Segment_t * segment )
```

  - This function searches for a segment in the segmentation table and deletes this segment descriptor
  - This function returns 0 if no such segment exists, 1 otherwise

- Function (two possibilities)

```
void * moveSegment ( Segment_t * list, void * ptr )
void * moveSegment ( Segment_t * list, Segment * segment )
```

  - This function moves a segment within your memory array (the content in the memory has to be moved as well
  - This function returns the new “start” pointer of the segment, or NULL in case of a failure situation

Extend your test program shell.c to demonstrate how defragmentation of the memory works. Perform a series of allocation operation, freeing memory, in order to fragment the memory.

Submission

For a CGS A5-A1, submit shell.c, mymemory.c, mymemory.h, and a Makefile that allows your program to be compiled.

Put files into a separate directory CGS_A5_A1. Provide detailed comments about the implementation of the required functions (explain it by walking the reader through the most important implemented statements and explain their purpose). Parts missing in your solution may reduce the CGS mark. Try to get with your implementation as far as possible.

Submission Procedure

You are required to submit in electronic as form:

- one zip file called **`cs3026_assessment1_<your_username>.zip`** or **`cs4096_assessment1_<your_username>.zip`**

Submit your work, using the following method:

- In MyAberdeen, go to "Assessment", and do the following:
  - Go to the area after the text explaining the submission procedure
  - click on "Assessment 01 (submit here)"
  - click on "Browse My Computer" (this is underneath "2. Assignment Submission")
  - find your zip file and complete this upload
- As a backup, send an email to raja.akram@abdn.ac.uk with the following **exact** subject line:
  - "CS3026/CS4096 Submission Assessment 1"
  - Attach to this email your zip file containing your submission.

Please use your University email address to submit your assessment.

All submissions should be documented, and this documentation must include your name and userid at the top of each file. Please submit (a) the complete source code (plus any necessary make files, text files, configuration files etc to compile and run your submission) and (b) a report describing your submission and how to operate your application. Make a zip file containing all this information and send it to the email address above.