University of St Andrews
School of Computer Science

CS3104 – Operating Systems

Assignment: P1 – A memory allocation library

Deadline: 16 October 2019 *Credits*: 40% of coursework mark

MMS is the definitive source for deadline and credit details

You are expected to have read and understood all the information in this specification and any accompanying documents at least a week before the deadline. You must contact the lecturer regarding any queries well in advance of the deadline.

Aim / Learning objectives

The purpose of this assignment is three-fold:

- to increase your experience of system level programming in C;
- to give you experience of using memory mapping for VM allocation in C;
- to reinforce your understanding of memory management algorithms.

Requirements

You are **required to use the C programming language** to implement a memory allocation library similar to *malloc*. The library allows user application code to request variable sized regions of memory and release these regions after use. Your implementation does not have to be thread-safe. Starter code can be found here:

http://studres.cs.st-andrews.ac.uk/CS3104/Practicals/P1-Malloc/code.tgz

Copy it to your practical folder and extract with `tar xvzf code.tgz`

The interface for your memory allocation library is defined in 'myalloc.h'. Your allocation library should implement a free list and support **coalescing** of adjacent free regions (see lecture notes).

Submission

You are required to submit an archive file to the P1 slot on MMS. It should contain a directory called "code" containing your solution and a short report in ***PDF*** format. The report must explain the design and implementation of your allocation library and your approach to testing and debugging. It may be helpful to use diagrams to explain data representation in a free list and any structures used.

Approach

To implement your memory allocation library you must reserve one or more regions of the VM from which program allocation requests can be granted. You can choose to allocate a single large region of VM initially or request VM incrementally. The benefit of an incremental approach lies in the ability to release VM when it is no longer needed by your library. Use the 'mmap' library function to allocate the heap space for your allocation library. Specify 'MAP_PRIVATE | MAP_ANONYMOUS' flags to create a mapping that is not backed by a file.

Implementation techniques are discussed in "Dynamic Storage Allocation: A Survey and Critical Review" by Wilson et al., published in the proceedings of the International Workshop on Memory Management, Kinross, Scotland, September 1995. You should read at least section 3.2 of this paper.

Please note that any solution which simply invokes mmap() or malloc() will not be accepted.

https://studres.cs.st-andrews.ac.uk/CS3104/Practicals/P1-Malloc/wilson95dynamic.pdf
https://studres.cs.st-andrews.ac.uk/CS3104/Practicals/P1-Malloc/section3.2_pages27-30_wilson95dynamic.pdf

Program structure

Your memory allocation library should be compiled as a static library that can be linked to application code. Your test application should be statically linked against your allocation library.

Starter code for the implementation is in 'myalloc.c'. ***You do not edit anything other than this file.*** This starter implementation uses calls to 'malloc' and 'free' in its implementation of 'myalloc' and 'myfree'. This passes the tests, but is not itself an acceptable solution.

A Makefile is provided. The Makefile compiles 'myalloc.c' as a static library and links the library into the executables for the test program. The submitted file should compile without additional libraries.

Autochecking

This practical will use the School's autochecker. See these as a starting point and make sure to also test your implementation in other ways. Make sure that you can execute the following command from the directory containing 'myalloc.c' and try to make your library pass these basic tests:

```
stacscheck /cs/studres/CS3104/Practicals/P1-Malloc/Tests
```

You can have a look at the source code for the tests by looking inside the Tests directory shown above. This could help you debug your implementation.

Note that additional tests may be used during marking. You are strongly advised to finish a correct, well-tested basic implementation before attempting advanced features.

Assessment

Only 'myalloc.c' will be considered during grading. 'myalloc.c' will be copied to a testing folder, compiled into a static library and linked to a number of test programs. Your code submission must not rely on additional source files or changes to the Makefile.

Marking will follow the guidelines given in the school student handbook (see link in next section). Some specific descriptors for this assignment are given below:

Mark range	Descriptor
1 - 6	A submission that does not compile or run, or a trivial implementation, e.g. simply calling malloc() and free() instead of implementing the functionality.
7 - 10	A solution which implements some of the requirements and allows the user to allocate memory, but is too unstable for real use.
11 - 13	An implementation which is mostly correct, but has some significant problems. This could include an incorrect implementation of a free list, inefficient operation due to poor choice of data structures, or excessive memory overhead.
14 - 16	A well-written and well-tested implementation which uses a correctly implemented free list, supports coalescing of free regions, has no major bugs, and is accompanied by a clearly written report.
17 - 18	An excellent implementation which goes beyond the original specification by implementing advanced functionality such as word alignment, incremental memory allocation, thread awareness, sensible optimisations, or any other sensible extension, and is accompanied by an excellent report.
19 – 20	An exceptional implementation showing independent reading and research, several extensions and an excellent quality of code, accompanied by an exceptional, insightful report.

Hints

Keep all functionality in the file myalloc.c. Complex build systems with many source files make it much harder to understand and mark the code.

Make small steps and make sure your implementation is correct before moving on. Do not attempt any advanced functionality until your basic implementation is solid, well-tested and well documented in the report.

You should show evidence of testing your implementation and verifying that it works correctly in a range of scenarios.

Understand the difference between a free list and a basic linked list -- this will impact the time it takes to find a free area of memory and therefore the performance of your implementation.

Outputting debugging information to stderr will help you understand what is happening inside your implementation and help you detect bugs -- don't work blindly.

Quality is more important than quantity. There is no reason why an excellent implementation should contain more than a few hundred lines of code.

Occasionally someone attempts a solution based on sbrk() instead of mmap(). There is nothing wrong with this (as long as you implement a good free list), but you may be making your life more difficult.

Policies and Guidelines

Marking
See the standard mark descriptors in the School Student Handbook:
http://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/feedback.html#Mark_Descriptors

Lateness penalty
The standard penalty for late submission applies (Scheme B: 1 mark per 8 hour period, or part thereof):
http://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/assessment.html#lateness-penalties

Good academic practice
The University policy on Good Academic Practice applies:
https://www.st-andrews.ac.uk/students/rules/academicpractice/