COSC1112/1114 Operating Systems Principles

Semester 2, 2019,
100 narks = 25% of semester mark

Assignment 1 – Segmented Memory Allocator

To a large extent, programming is about management of memory as it's one of the key resources that we have at our disposal as programmers.

What we want you to consider is the costs involved in the memory allocation strategies you're going to implement. We want you to gain an appreciation of the complexity of memory allocation and the fact it is not a cost free process.

We would certainly not advise you to reimplement your own memory allocator unless you find in a particular project you have certain needs.

Please note that you may choose to implement your solution to this task in C or in C++ as that shouldn't matter.

- **Late submission penalties apply**
  Unless an extension has been granted (for procedures and grounds see http://www.rmit.edu.au/students/assessment/extension), a penalty of 10% of the total project score will be deducted per day, and no submissions will be accepted 5 days beyond the due date.

- **Special Consideration**
  With the exception of dire circumstances, no extension requests will be considered within 5 working days of the submission date. ("Dire Circumstances" means things like hospitalization of you or a close relative, etc.) Persons requesting a late extension may be required to prove that a

Academic Integrity

All work you submit for this course must be your own work unless otherwise credited. Even so, we cannot give you marks for work that is not your even if credited. It is a condition of submission of assignments that you agree with RMIT's Assessment Declaration available here: https://www.rmit.edu.au/students/student-essentials/assessment-and-exams/assessment/assessment-declaration

All submissions will be checked for academic integrity.

Programming Task

The System Calls

The basic system calls used to allocate memory and free memory in UNIX-like operating systems are brk() and sbrk(). These functions allocate and deallocate memory by shifting the “program break” or heap frontier up and down. The heap starts at low addresses and so shifting the program break upwards allocates memory and shifting it downwards deallocates memory.

The function prototypes for these are:

```
int brk(void * addr);
```

This function sets the program break to be the value passed in by addr. If the setting of this values is successful, brk() will return 0.

```
void * sbrk(intptr_t increment);
```

adds or subtracts from the current program break to allocate memory (if a positive number is provided) and to free memory if a negative number is provided.

In this assignment you will use these functions to allocate memory. We won’t be using these functions to free memory as that’s a bit more complicated a proposition.

Solution Design

You will use two linked lists to manage memory – a linked list of memory allocations and a linked list of chunks that have been freed. You are not expected to implement the linked lists yourself. At this point you should know how linked lists work and should be able to use an already existing library. As such there are no marks awarded for the linked list itself.

You will however need to keep track of some accounting information as well as the memory allocations themselves (at a mimimum, the size of each allocation).

The three core requirements are the two functions:

```
void * alloc(size_t chunk_size);
```

and

```
void dealloc(void * chunk);
```

and a way to be able to set the allocation strategy from other code (more on this later).

You may not have come across the size_t type before but it’s just an unsigned long integer that is guaranteed to be able to hold any valid size of memory. So, it’s the size of the memory chunk that has been requested.

The pointer passed into dealloc is the chunk to be “freed” and so it must be a chunk that was previously allocated with the alloc function.

Memory Allocation

Memory allocation will be implemented in the function alloc() as specified above. This function will first look in the free list for a chunk big enough to meet the allocation request (based on the previously set strategy). If a chunk large enough is found and it is larger than what you need, you should split the chunk into what is returned and what is left over. You can cast to a char pointer to do this.

If a chunk large enough cannot be found, a new request will be sent to the operating system using either the brk() or sbrk() system calls.

In either case, the chunk and its metadata will be added to the allocated list before returning the pointer to the memory itself back to the caller that requested the memory allocation.

Allocation Strategies

As part of memory allocation you will need to implement a search for a chunk using an allocation strategy. The allocation strategies you need to implement are:

- **first fit:** the first chunk you find that is big enough, you split and return to the user a chunk that is big enough to meet their needs. You add the rest of the chunk back onto the free list.
- **Best fit:** you find the chunk whose size is the closest match to the allocation need. Often that will be exactly the size required by the client so you may not have to split it.
- **Worst fit:** find the largest chunk and split off the part you need and store that in the allocated list. Store the rest of the chunk back on the free list.

Memory Deallocation

This will be performed using the dealloc() function. Search for the pointer passed in in the allocation list. If it is not found, you should abort the program as this is a fatal error – you can’t free memory that was never allocated.

However, if you find the chunk, simply remove it from the allocated list and move it to the free list.

The Experiment

In a separate source file from your allocator, you should set up some dynamic arrays to be allocated with your allocator. You want to vary the size of what is allocated – not just big or small but odd and even numbers, etc. You may consider creating some arrays of various types and sizes and fill them with random numbers. You want to repeat your tests on data structures using loops and you want to run sufficient iterations with varied sizes and collect some meaningful statistics. I have found the function getrusage() to be a useful function to collect such statistics in the past. Please be aware though that not all fields in the rusage struct are filled in by the kernel.

A dataset that I found useful in testing my allocator with strings (I just used char* rather than std::string for this) was: https://github.com/dominictarr/random-name

Makefile

You will need to provide a Makefile that separately builds your source files into object files and then links them all together. This is something you should have learnt how to do in your previous studies.

The Report

This section is expected to be in a well formatted report. You should provide a cover page and table of contents and clear heading, and well formatted tables and graphs if you feel you need them.

Provide a description of the method and reasoning behind your choices of experimental data.

Based on your experiments, provide data on the performance of the various memory allocation strategies.

We are particularly interested in the how the size of your lists grow with various types of data and the various memory selection strategies we have asked you to implement. Is the average memory chunk in the free list larger or smaller with a given strategy, Also, is there evidence of fragmentation? What kind of fragmentation? Describe a strategy to reduce fragmentation so we can have much less wasted memory.

Are these allocation algorithms wasteful in any way? What would be a way to make them less wasteful?

Which memory allocation strategy was best? Why do you think it was the best? Could a different algorithm be preferable with different data?

Detailed Marking Guide

Please note that an assignment that cannot be compiled will attain a FAIL mark so please ensure that you compile on one of the university servers (titan, jupiter, saturn) and provide instructions on how to do so to avoid misunderstandings.

- Demonstrations 2.5 marks each in weeks 4 and 6. In week 4 you need to show the ability to allocate and use a chunk of memory from either `brk()` or `sbrk()`. In week 6 you will need to show at least one strategy implemented and the ability to use that strategy allocate and free and array in a loop.
- Solution design: 5 marks
- general allocation algorithm (not counting the strategies) 8 marks
- general deallocation algorithm 8 marks
- the firstfit algorithm for finding blocks of memory 8 marks
- the bestfit algorithm for finding blocks of memory 8 marks
- the worstfit algorithm for finding blockss of memory 8 marks
- Make file (5 marks)
- Code for your experiment 10 marks
- Using good coding standards (5 marks)

Submit your code in a zip file including your Makefile. Submitting in another archive format will be penalised.

- Report: 30 marks

  Please note that a report that lacks experimental evidence but is based on solid research cannot get above 50% of the mark for the report. All materials outside of provided course materials that are accessed must be referenced and a bibliography provided. You must submit your report via the provided turnitin submission link

  - formatting 5 marks
  - description of your method and reasoning behind experimental design 5 marks
  - presentation of your experimental findings (charts, tables, etc) 8 marks
  - Discussion of memory fragmentation and the meaning of your findings 8 marks
  - Conclusion and recommendation of the best allocator 4 marks