School of Computing, University of Leeds, 21/22

XJCO2211

Operating Systems

Coursework C1: Memory Management

Deadline: 11th Oct 2021 at 12pm (Beijing time)

This assignment is worth 25% of the final module grade, with a total of 30 marks available. Your submission should include the final version of your code.

Introduction

This coursework consists of 1) a programming task, 2) a short report that explains your implementation, and 3) a set of questions that needs to be answered. The key learning objectives are to:

- Understand the importance of memory management
- Design and implement important components in an operating system
- Understand basic principles of memory allocation in both user space and kernel space
- Understand basic principles of the buddy system

Assessment & Submission

There will be **THREE** assignments on Minerva for the **THREE** parts of your coursework. You will be required to submit both your code and the associated answers to the corresponding assignments. You will also be required to demonstrate the code you have generated.

Code Submission: Upload your code to the assignment (***Memory Management (Code)***) as a .ZIP file named "StudentID_USERNAME.zip" that contains three required files: *memory_consumption.c*, *memory_management.c* and a header file *memory_management.h*.

Report Submission: Upload your report as a PDF file named "StudentID_USERNAME_C1_report.pdf" to the Turnitin assignment (***Memory Management (Report)***) on the module VLE.

Q&A Submission: Upload the report as a PDF named "StudentID_USERNAME_C1_QA.pdf" to the Turnitin assignment (***Memory Management (QA)***) on the module VLE.

Demonstration: You will be assigned a 5 minutes time slot during one of the lab sessions for you to demonstrate your code.

Disclaimer

This is intended as an individual piece of work and, while discussion of the work is encouraged, plagiarism in any form is strictly prohibited. The answers you hand in should be your own work. Standard late penalties apply for work submitted after the deadline.

The Task

Task 1: Programming functions for allocating and freeing memory

Your task is to investigate memory usage monitoring and implement functions that satisfy the following specifications, similar to the native *malloc* and *free* functions.

1. Using the ***free*** or ***vmstat*** tool to estimate memory usage **[3 marks]**
    a. Login to your Linux system, open a terminal and type *man free* or *man vmstat*. Investigate carefully and understand the meanings of different arguments.
    b. Run commands, e.g., *free -m* argument. Find out how much memory is in your system, and how much memory is free in your system.
    c. Implement a simple program, *memory_consumption.c*, that uses (or consumes) a certain amount of memory (e.g. the number of Megabytes to be used may be indicated as an input of a command line argument, and the program can reserve a block of memory of the specified number of bytes). The program could run either indefinitely or for a given time period.
    d. Understand how to monitor runtime memory usage over time when the program is executing. Tune the memory usage in the program and examine what happens when a large amount of memory is used.

2. Implement a *malloc* function **[5 marks]**
    a. The name "*malloc*" stands for memory allocation. The *malloc* function reserves a block of memory of the specified number of bytes and returns a pointer to the allocated memory. If the requested size is zero, the *malloc* function will return *NULL*.
    b. It should have the function prototype *void*_malloc (size_t size)*.
    c. It should return a *void* pointer to the start of the allocated memory.
    d. It should allocate memory on the heap and split up regions of memory.
    e. The pointer returned should be well aligned with any kind of variable.

3. Implement a *free* function **[5 marks]**
    a. The *free* function frees the memory space pointed to by the parameter which must have been returned by a previous call to *_malloc*. If the returned pointer is *NULL*, no operation will be performed.
    b. It should have the function prototype *void*_free(void*ptr).*
    c. It should free the memory and make it available for subsequent allocations.
    d. It should merge necessary regions of memory to mitigate memory fragmentation.
    e. It should return the memory back to the operating system when it is appropriate.

Your implementation should be included in *memory_comsumpton.c*, *memory_management.c* and a header file *memory_management.h*. Your files should be located in the user directory.

Submit your code for the completed solution on Minerva. **[Total of Task 1: 13 marks]**

Task 2: Summary Report

Write a report on your implementation for Task 1 with a **maximum length of 3 sides of A4.** You should reference any additional materials that you have used, including any tutorials or documentation. Your report should be formatted with the main text body in **Arial point size 11**, **1.5 line spacing**, **your StudentID and name in the header** and **page number in the footer**.

In your report you should discuss:

- How you have achieved the functionalities outlined in Task 1 and made your design decisions.
- The main design principle together with the explanation of your code.
- What you have learned from Task 1.

Submit your report on Minerva as a PDF.

The report will be marked as follows:

Understanding	2
Depth (completeness, detail, comparison)	4
Report structure and writing quality	2

[Total of Task 2: 8 marks]

Task 3: Questions and Answers

Question C1.1: Memory Fragmentation

Memory management is a crucial aspect of operating system functionality. A common problem that needs to be addressed is that of memory fragmentation. Explain – with examples – how both internal and external memory fragmentation occur. **[3 marks]**

Question C1.2: Memory Allocation

Discuss the difference between static and dynamic memory allocation in operating systems (e.g. you may examine the Xv6 operating system as an example). Discuss the advantages and disadvantages of static and dynamic approaches. **[3 marks]**

Question C1.3: The Buddy System

The buddy system is a popular memory allocation mechanism that can combine a series of power-of-two allocator with free buddies. The rationale behind it is simple: if a block of the desired size is not available, a large block is broken up in half and the two blocks are buddies to each other. One block is used for the allocation and the other is for free. The blocks are continuously halved as necessary until a block of the desired size is available. Each block has an attribute – named as *order*, and the block size is exactly equal to two to the power of its order – 2^{order}. This means that the size of the block with order *n* is two times of the block with order *n-1*.

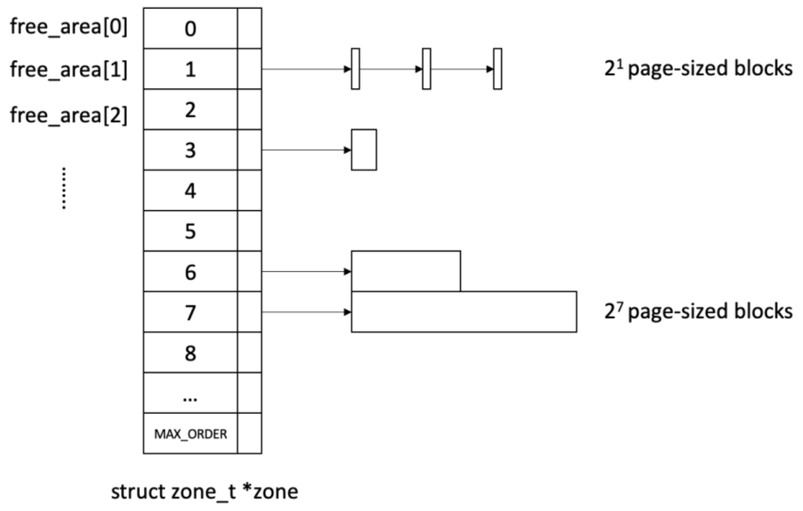
Let us use a struct *zone_t* to store all the available blocks with different sizes.

- In the struct *zone_t*, an array of struct *free_area_t* is used to record the free blocks with the same order, and the index of the array is equal to the order of the blocks. For example, *free_area[3]* records the link pointer of all the free blocks with *order 3*. Thus, the length of the array is equal to the MAX_ORDER + 1.
- In the struct *free_area_t*, apart from the important attribute - *order*, another crucial element is the *free_list* of the struct *free_list_t*, which indicates all the free blocks using a linked list.

```
struct zone_t {
   struct free_area_t free_area [MAX_ORDER+1];
   .....
}
struct free_area_t {
   unsigned int order;
   struct free_list_t free_list;
   ......
}
```

Given the current status of zone which is an instantiated variable storing the information of all the free blocks, as shown in the following figure, explain and illustrate with a figure what the free memory status in zone is after allocating a 2^4 page sized block (order = 4).

[3 marks]



Your answers should be formatted with the main text body in **Arial point size 11**, **1.5 line spacing**, **your StudentID and name in the header** and **page number in the footer in a A4 document.** No max page length required.

Submit your report on Minerva as a PDF. **[Total of Task 3: 9 marks]**

[Total of Coursework C1: 30 marks]