




**School of Computing Technologies**  
**COSC1114 Operating Systems Principles**

**Assignment 3**

	<b>Assessment Type:</b> Individual assignment; no group work. Submit online via <a href="#">Canvas</a> → <a href="#">Assignments</a> → <a href="#">Assignment 3</a> . Clarifications/updates may be made via announcements and relevant discussion forums.
	<b>Due Date:</b> Week 14, Sunday 29 <sup>th</sup> October 2023, 11:59pm.
	<b>Weighting:</b> 100 marks that contributes 40% of the total assessment.

### 1. Overview

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 that 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 that you have certain needs.

You are to implement your solution to this task in C++ but note that the required function calls are C system calls that are specific to Linux, so you won't be able to compile/run code outside of Linux. This does however include WSL but does not include MacOS, which while having a terminal has a different approach to memory allocation.

### 2. Learning outcomes

This assessment task supports CLOs 1, 2, 3, 4, 5 & 6.

### 3. Assessment details

This assessment will determine your ability to

1. Understand the concepts taught over the weeks 9, 10, and 11 materials.
2. Work independently in self-directed study to research the identified issues.

### 4. Submission

Your assignment should follow the requirement below and submit via [Canvas](#) → [Assignments](#) → [Assignment 3](#). You may resubmit the assignment if you need to, only the most recent version will be marked. Failure to follow the requirements incurs up to 10 marks penalty for this assessment.

You will submit two things **separately**:

1. Your C++ source code in [a zip file](#).

If your student ID is s1234567, then please create a zip file named s1234567\_OS\_A3.zip including the code you have developed and a README file, in which please specify in detail how to run your code on the provided servers.

2. Your report in [pdf format](#).

Use at least 12-point font size. If your student ID is s1234567, then please create a pdf file named s1234567\_OS\_A3.pdf.

It is your responsibility to correctly submit your files. Please verify that your submission is correctly submitted by downloading what you have submitted to see if your submitted file includes the correct content.

Never leave submission to the last minute – you may have difficulty uploading files.

You can submit multiple times – a new submission will override any earlier submissions. **However, if your final submission is after the due time, late penalties will apply.**

## 5. Academic integrity and plagiarism

**The penalties for submitting code that is not your own can be severe. Do not simply copy other people's work, it is not difficult for us to detect copied work and we will pursue such cases.**

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.

All submissions will be checked for academic integrity.

## 6. Late submission policy

A penalty of 10% per day of the total available marks will apply for each day being late. **After 10 days, you will receive zero marks for the assignment.**

If you want to seek an extension of time for assignment submission, you must have a substantial reason for that, such as unexpected circumstances. Reasons such as, unable to cope with study load, is not substantial. Also, you must apply for an extension as soon as possible. Last minute extensions cannot be granted unless it attracts special consideration.

Please find out how to apply for special consideration online at

<https://www.rmit.edu.au/students/student-essentials/assessment-and-results/special-consideration/eligibility-and-how-to-apply>.

Any student wishing an extension must go through the official procedure for applying for extensions and must apply at least a week before the due date. Do not wait till the submission due date to apply for an extension.

## 7. Rubric and marking guidelines

The rubric can also be found in **Canvas → Assignments → Assignment 3.**

Please note that an assignment that cannot be compiled will attain a failure mark so you want to ensure that you compile on one of the university servers and provide instructions on how to do so to avoid misunderstandings.

Requirement	NoAttempt (0%)	Poor (25%)	OK (50%)	Good (75%)	Excellent (100%)
Solution design (10 marks)	No attempt	A reasonable attempt but it does not compile.	Code compiles but is an incomplete/incorrect implementation.	Mostly complete implementation but there are minor problems.	Excellent job. Well done.
General allocation algorithm: (10 marks)	No attempt.	Some attempt but did not get the code working or does not compile.	Code compiles but is an incomplete/incorrect implementation.	Mostly complete implementation but there are minor problems.	Excellent job. Well done.
General deallocation algorithm (20 marks)	No attempt.	Some attempt but did not get the code working or does not compile.	Code compiles but is an incomplete/incorrect implementation.	Mostly complete implementation but there are minor problems.	Excellent job. Well done.
The first-fit algorithm for finding blocks of memory: (10 marks)	No attempt.	Some attempt but did not get the code working or does not compile.	Code compiles but is an incomplete/incorrect implementation.	Mostly complete implementation but there are minor problems.	Excellent job. Well done.
The best-fit algorithm for finding blocks of memory: (15 marks)	No attempt.	Some attempt but did not get the code working or does not compile.	Code compiles but is an incomplete/incorrect implementation.	Mostly complete implementation but there are minor problems.	Excellent job. Well done.
Makefile and Arguments Output (5 marks)	No attempt.	Some attempt but did not get the code working or does not compile.	Code compiles but is an incomplete/incorrect implementation.	Mostly complete implementation but there are minor problems.	Excellent job. Well done.
Code for your experiment (design and implementation): 15 marks	No attempt.	Some attempt but did not get the code working or does not compile.	Code compiles but is an incomplete/incorrect implementation.	Mostly complete implementation but there are minor problems.	Excellent job. Well done.
Report (25 marks)	No attempt.	The report is incomplete.	The report is difficult to follow in places. The results are partially correct.	The report is reasonable easy to read. The results are correct and well-described. The findings can be better interpreted.	Excellent job. Well done.

## 8. Assignment tasks

### 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 value is successful, `brk()` will return 0. You will use this function at the end of your program to reset the heap frontier back to where it was at the start of your program, and it will free all memory you have allocated.

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

This function 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. The program break is the boundary of the heap (data segment) of the program. This is essentially to grow or shrink the heap (data segment). You will use this function to allocate memory if the heap needs to grow to accommodate a new request.

Please find out more about `brk` and `sbrk` on Titan: `$man brk`

### Code Quality

You are expected to compile and run your code on `titan`, `jupiter` or `saturn` servers provided by the school. Please note that while code quality is not specifically marked for, it is part of what is assessed as this course teaches operating systems principles partly through the software you develop. As such lack of comments, lack of error checking, good variable naming, avoidance of magic numbers, etc., may be taken into consideration by your marker in assigning marks for the components of this assignment, although operating systems principles do take priority.

### Makefile

You may use any `c++` features up to and including `c++20`.

A reminder that you can enable `c++20` features on the servers using the command:

```
$scl enable devtoolset-11 bash
```

You can then let the compiler know you wish to use `c++20` features with the flags:

```
-Wall -Werror -std=c++20
```

And these are the flags we want you to use in your assignment.

“`make all`” will build 2 programs (`firstfit`, `bestfit`) at once.

## Task 1 – Solution Design (10 marks)

You will use two linked lists to manage memory – a linked list of occupied chunks (memory allocations) and a linked list of free chunks (to be used for new memory allocations). 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 determine and keep track of some accounting information as well as the memory allocations themselves (**at a minimum**, the location and the size of each allocation). This means the data structures for keeping track will need to be global variables. These global variables should not be referenced outside of the two function calls below.

The core requirements are the two functions:

```
void * alloc(std::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).

These two functions are essentially a simpler version of `malloc` and `free` (you are encouraged to get familiar with these two functions using man pages on Titan). 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.

**Note:** `dealloc` does not return the chunk to OS. It simply makes the chunk free/available to new requests.

It is suggested that you represent an allocation using a class or struct like the following:

```
struct allocation {  
    std::size_t size;  
    void *space;  
};
```

where `space` will have been allocated with `sbrk()` initially. **It is also suggested that you store each allocation in its own pointer rather than having a list of allocation structs.** This is because the `pop()` operations in the standard library call the destructor which may free the memory, depending upon your design.

## Task 2 – Memory Allocation (10 marks)

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 cannot be found, a new request will be sent to the operating system using `sbrk()` system call to grow the heap.

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

**We want you to allocate memory in fixed partition sizes of 32, 64, 128, 256, and 512 bytes only.**

### Task 3 – Allocation Strategies (25 marks)

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 that you find big enough is returned to the user.
- 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 do not have internal fragmentation. (Here we assume the perfect scenario to simplify the task.)

### Task 4 – Memory Deallocation (10 marks)

This will be performed using the `dealloc()` function.

Search for the pointer passed in the allocated list (the linked list of occupied chunks). 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 linked list of free chunks).

### Task 5 – The Experiment (15 marks)

In a separate source file from your allocator, you should make a sequence of interleaved `alloc` and `dealloc` calls to test out your allocation strategies. We have provided you a generator `a3_gen.sh` to create these sequences. **Please find the bash script in the assignment page.**

To generate a file called `datafile` with a sequence of 20 interleaved `alloc` and `dealloc` calls as shown by the screenshot below:

```
[e52483@csitprdap01 ~]$ chmod +x a3_gen.sh
[e52483@csitprdap01 ~]$ ./a3_gen.sh 20 > datafile
[e52483@csitprdap01 ~]$ cat datafile
alloc: 169
alloc: 357
alloc: 197
alloc: 241
alloc: 351
alloc: 400
alloc: 84
alloc: 184
dealloc
dealloc
dealloc
dealloc
alloc: 254
alloc: 296
dealloc
dealloc
dealloc
alloc: 164
alloc: 471
alloc: 32
```

An `alloc` call allocates a suitable free chunk.

A `dealloc` call simply deallocates the last memory chunk allocated, **specifically, in a LIFO (last in, first out) order.**

Your programs are expected to parse command line arguments according to the following format:

- First fit memory allocation: `./firstfit datafile`
- Best fit memory allocation: `./bestfit datafile`

where the `datafile` is the one we generated using the given `alloc_generator`.

Your programs are expected to generate the following output on the screen (only print once in the end):

- The allocated list (the linked list of occupied chunks)

For each chunk, print out the address of the chunk, the total size of the chunk, and the used size of the chunk.

- The free list (the linked list of free chunks)

For each chunk, print out the address of the chunk and the total size of the chunk.

### **Task 6 – Report (25 marks)**

Please ensure your report (no more than 3 pages) is neat and easy to read. You may provide well formatted tables and graphs if you feel you need them.

Based on your experiments, provide data on the performance of the various memory allocation strategies. Discuss the performance and how to improve the performance including new allocation strategies.

**You need think about what performance metrics you want to use, as it may affect your design.**

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.