

**PROJECT REPORT ON**  
**INTERACTIVE MEMORY MANIPULATION**

By

SIDDHANT JAJOO  
SATYA MEHTA

**UNDER THE GUIDANCE OF:**

PROFESSOR KEVIN GROSS,  
UNIVERSITY OF COLORADO,BOULDER

DATED:10/08/2018.

# **TABLE OF CONTENTS**

<b>PROJECT PLAN</b>	<b>3</b>
<b>SYNTAX DOCUMENTATION</b>	<b>4</b>
<b>REPORT QUESTIONS</b>	<b>6</b>
<b>APPENDIX</b>	<b>10</b>

# PROJECT PLAN

Sr. no	Milestone	Task	Likely hours	Date
		Develop and document project plan.	2	
1	Submit project plan			09/20/18
		Build environment setup Git Repository. Initial Makefile, main, and framework. Basic command line prompt and parse. Help function.	0.5 0.25 0.5 0.5 4	
2	Command prompt with help			09/25/18
		Allocate Free Display Write	1 1 2 4	
3	Display and Modify			10/01/18
		Execution time measurement Invert	3 2	
4	Timed Function			
		Experiments and write-ups Write Pattern Verify Pattern	4 5 3	
5	Feature complete			
		Final Report	10	
6	Submit final report			10/06/18

# SYNTAX DOCUMENTATION

The Following are the syntax to invoke different functions :

1. **help**- Help function displays all the available commands fro the user to input in order to invoke different functions described in the help menu.
2. **allocate**- The user specifies number of 32 bit word to allocate a memory block.
3. **write** -The user specifies offset or memory location where 32 bit hexadecimal data can be written into it.
4. **display**- The user specifies an offset or memory location of the contents that needs to be displayed.
5. **writetime**- The user specifies offset or memory location,number of words to be written starting from the base address and seed value to generate a pseudo random number and writes the value of the pseudo random number generated to the specified memory location and returns the execution time of the command.
6. **verifytime**- The user can verify the pseudo random number generated using the writetime command and returns the execution time of the command. The inputs required from the user for this command are offset or memory location,number of words to be written starting from the base address and seed value.
7. **free**- this function is used to free up the memory block.
8. **ex**- Exits the application.
9. **main**- Gives control on all the functions listed above.

# TEST CASE INPUT FILE

help  
allocate  
3  
write  
N  
0  
1  
123f  
123d  
dispmem  
N  
0  
inverttime  
N  
0  
1  
writetime  
N  
0  
1  
2  
verifytime  
N  
0  
1  
2  
free

# REPORT QUESTIONS

1) How well did your effort estimates match with actual effort? Provide examples of both inaccurate and accurate estimates and discuss any contributing factors to the success or failings of your effort estimates.

Ans. Most of the milestones were achieved before the estimated time. The first milestone which was the project plan was completed within an hour. Initial tasks such as Makefile took little more time as estimated as it required some reading. The help function required around 4-5 hours as we implemented multiple algorithms before finalising one. The milestone “Command Prompt with help” was achieved well before the estimated time. Allocate and free didn’t required much time was completed well before estimated time whereas display and write required pointer manipulation hence, it consumed more than 6 hours. There were times we got stuck for not using global pointer’s efficiently which was resolved at a later stage. In addition to this, we did face segmentation fault (core dumped) at various stages which were solved using local char pointers. The milestone display and modify was completed on 1st October 2018.

We estimated time for ‘time measurement’ task of 3 hours but it turned out to be the simplest as it only required a header file to be included and just few lines of code. Invert function was also implemented before estimated time as just the memory block pointer was passed and inverted using ‘^’ operand.

Pattern generation consumed about 5 hours as it required reading and finalisation of perfect and easy algorithm to generate a random pattern. Write pattern took less time than verify pattern so our estimation was not quite accurate. Total time for write pattern and verify pattern was around 8 hours.

This project requires user inputs at various stages during the application. These input are in the form of offset memory address or base memory address which were required in many functions. It was observed that these inputs of memory location or the offset was required in many functions. This led to complexity and irregular pointer manipulation. Therefore, we devised a common function named `specifymem()` which was called when user was required to enter the input.

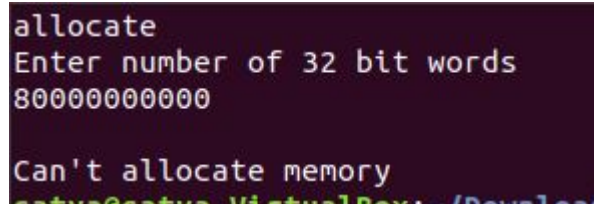
2) Were there any tasks that you forgot to include in your project plan?

Ans: We implemented the offset input from the user at a later stage hence, it is not included in the project plan. We had decided to include a list of error control codes but only settled for the ones that we deemed were necessary. We would look forward to including them in future.

3) What is the largest block of memory you can allocate? Discuss.

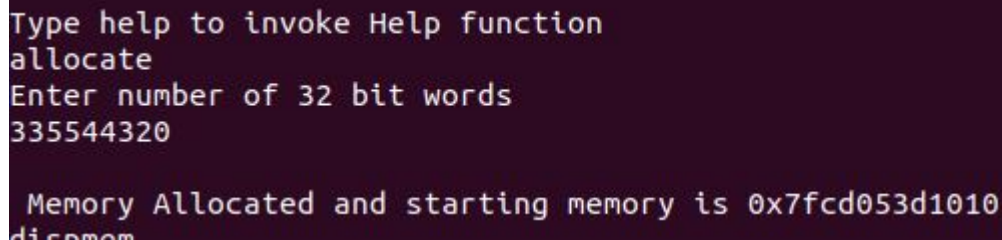
Ans: We did try to allocate different number of memory blocks. The maximum value that was tried was 80000000000 number of 32 bit words to which the response was “can’t allocate memory” as shown in the screenshot below. Further, we also tried 335544320 32 bit words which was successfully allocated. Doing some research on the internet, we found multiple answers on malloc function. As we are working on virtual box the malloc allocates memory depending on the free memory available.

The images below shows some examples which we tried during application run.



```
allocate
Enter number of 32 bit words
80000000000

Can't allocate memory
catya@catya-VirtualBox: /Downloads
```



```
Type help to invoke Help function
allocate
Enter number of 32 bit words
335544320

Memory Allocated and starting memory is 0x7fcd053d1010
discom
```

4) What happens if you try to read outside of the allocated block?

Ans: If you try to access the memory outside the allocated block we get an error which says “Invalid!, your offset cannot be greater than number of words.”

This is one of the error control code that we have incorporated in our code.

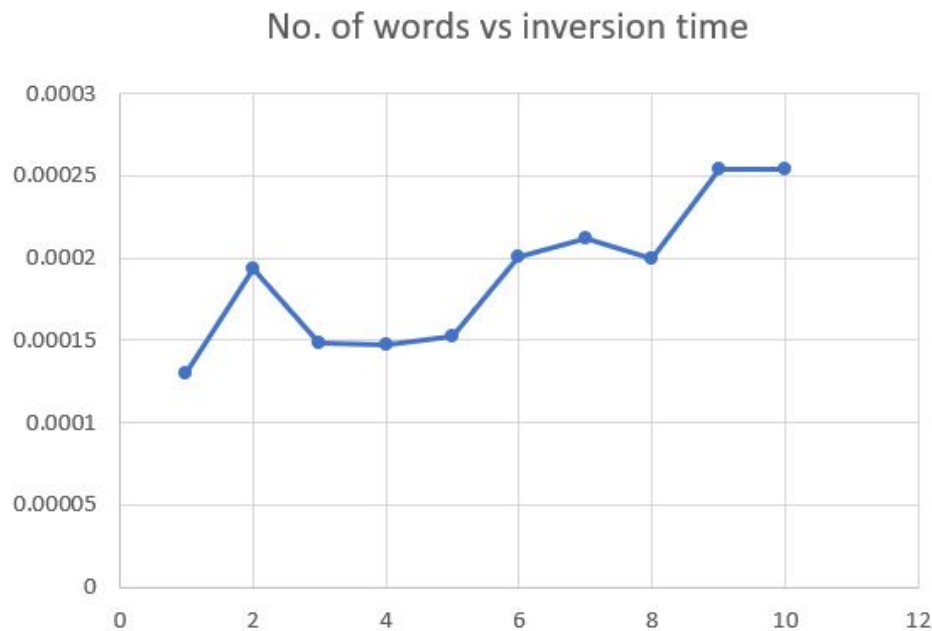
5) What happens if you try to write outside of the allocated block?

Ans: If you try to write the memory outside the allocated block we get an error which says “Invalid!, your offset cannot be greater than number of words.”

This is one of the error control code that we have incorporated in our code.

6)Analyze the time it takes to invert memory for different memory block sizes. Is there a linear relationship between time and size?

Ans: There is no linear relationship between time and size. The time taken for the inversion depends on other unknown factors. We have analyzed the time for words ranging from 1 to 10.



7)What are the limitations of your time measurements?

Ans: Time taken for inversion is in fractions of seconds. Therefore, we are not able to measure accurate time. Also, if we measure inversion time repeatedly for the same number of words there is a slight difference in the results obtained.

8)What improvements you can make to the invert function, your program or your build, to make it run faster? How much improvement did you achieve?

Ans: We can increase the speed for inverting any value by using all of computer's resources. This would engage the computer to use all the 64 bits on the bus thus making it faster.

9)What algorithm did you choose for your pattern generator? How does it generate its pattern? What are its strengths and weaknesses?

Ans: The most vulnerable weakness in any random number generator code is what if the random number generated is a zero. If any random number generated is a zero there will be a series of number of value zero generation for the rest of the memory locations. This will itself fail the purpose of the code. In order to avoid this , we use a prime number which never gives a remainder that is zero and thus never producing a zero value as a random number. In addition to this , the prime number has been selected to be a very large number which thus reduces the probability of the random numbers getting repeated.

We have used three constants in this pseudo random number generator. One being a prime number so that the remainder of a number when divided by this prime number never returns



a zero. Further a seed value is taken as an input and an initial random number is defined to generate a random number.

Two new random numbers are generated using these constants - initial random number and a prime number. These random numbers are used in an expression to obtain a number.

If this number is negative, the number is added with the prime number to obtain a positive random number.

The following is the pseudo code:

Initial random number = constant 1

Prime number = constant 2

->Two new constants are :

$C1 = \text{prime} / \text{initial random number}$

$C2 = \text{prime} \% \text{initial random number}$

$\text{Number} = (\text{initial random number} * (\text{number} \% C1)) - (C2 * \text{floor}(\text{number} / C1))$

If (number is negative )

$\text{Number} = \text{number} + \text{prime}$

$\text{Number} = (\text{number} * \text{initial random number}) \% \text{prime}$

This value of the number is then stored in the specified memory location.

# APPENDIX

This report has comprehensively covered all the command descriptions, project plan and the time taken to complete the project as compared to the estimated time, a test case and answers to all the report questions.

In order to replace the if else structure, extensible use of function pointers have been incorporated in the project. Lookup Tables have been created in order to solve this issue. Different header and source files have been created to support modularity ,modification, adding additional commands and features.

This project has covered all the error controls and bound checks as deemed necessary. For example if the user inputs a wrong hexadecimal value of a memory location the program displays “invalid memory location” and asks again to input for the memory location.

In addition to this if the offset input is beyond the allocated memory or if the 32 bit word is more than the memory is allocated , it displays a message “invalid offset”/“invalid words” and asks the user to input the parameters again.

The pseudo code for the random number generator was referenced from [https://www.cs.utah.edu/~germain/PPS/Topics/random\\_numbers.html](https://www.cs.utah.edu/~germain/PPS/Topics/random_numbers.html) .