Memory Management

Objectives

1. Write programs that dynamically allocate/deallocate memory
2. Write programs that overflow the heap and stack memories
3. Write programs that use Linux timers to measure the execution time of code blocks
4. Use size command in Linux
5. Study a program that experimentally calculates the L1 data cache size

Aim

The aim of this session is familiarize yourselves with the memory management aspects.

Section 1 - Dynamic Memory Allocation

There are four library functions provided by C defined under **<stdlib.h>** header file to facilitate dynamic memory allocation in C programming. They are:

- *malloc()*
- *calloc()*
- *free()*
- *realloc()*

Malloc dynamically allocates a single large block of memory with the specified size. It initializes each block with default garbage value. If space is insufficient, allocation fails and returns a NULL pointer.

Calloc dynamically allocates the specified number of blocks of memory of the specified type. It initializes each block with a default value '0'. If space is insufficient, allocation fails and returns a NULL pointer.

Realloc dynamically allocates extra memory. If the memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to dynamically allocate extra memory. Reallocation of memory maintains the already present value and new blocks will be initialized with default garbage value.

"free" dynamically de-allocates the memory. The memory allocated using functions malloc() and calloc() is not de-allocated on their own. Hence the free() method is used, whenever the dynamic memory allocation takes place. It helps to reduce wastage of memory by freeing it.

Heap is a region of a process's memory used to allocate memory dynamically. Heap memory can be allocated using malloc() and calloc() functions and resize using realloc() function. The data in heap can be accessed globally and once we allocate memory on heap it is our responsibility to free that memory space after use.

Task 1: Study the 'malloc.c', 'calloc.c' and 'realloc.c' programs.

The syntax of the aforementioned functions is explained in detail, in the lecture slides. Make sure you understand how these programs work. Make sure you deallocate the memory allocated using free function.

Exit function: void exit(int status) function terminates the running process immediately. For successful termination use exit(0) or equivalently exit(EXIT_SUCCESS). Use exit(EXIT_FAILURE) for not successful termination. It is important to note that 'return 0' exits from the function while exit(0) exits from the program.

Heap Overflow

Heap overflow: Heap can be overflown and thus crashed in two different ways. There is an example for every case.

Task2. **Modify Heap_crash1.c file, to make the heap crash.**

In this program, we are allocating an int (4bytes) many times, without deallocating it. If this procedure is applied many times, then the heap will fill up. The question you need to answer is how many times should we allocate an integer (4bytes) for the heap to be filled up? It depends on the size of your physical memory (DDR); normally, if the DDR size is of 4Gbytes, you will need to allocate about 2Gbytes of memory. If you are using a virtual machine, the heap memory size will be smaller.

Task3. **Modify the Heap_crash2.c file to make the heap crash.**

In this program we are allocating a large amount of memory. How much memory are we allowed to allocate?

Stack Overflow

Stack is a special region of a process's memory, which is used to store local variables used inside the function, parameters passed through a function and their return addresses. When a function exits, all the variables associated with a function are deleted. The user does not have any need to free up stack space manually.

The stack size is always fixed and limited. If a program uses more memory space than the stack size then a stack overflow might occur and can result in a program crash. See below a way to do that.

Task4. Modify the Stackcrash.c program to make the stack to crash.

To this end, you must increase the size of the array inside the function; if the array size becomes bigger than the stack size then the program will crash. Normally, the default stack size is about 1Mbyte. Can we allocate such a big array inside main()?

Section 2 – Experimental procedure that calculates the L1 data cache size

In cache_benchmark.c, I have written a program that writes an array into memory many times. In cache_benchmark() routine, the X[N] array is written 'TIMES' times. I have included a 'weird' code in the beginning, which maps this thread to core number zero. You do not have to understand how this code works. However, you should know that if we run a single-thread program on Linux or Windows, the Operating System will toggle the process amongst different CPU cores. This is done in order to reduce the total heat dissipation of the CPU. So, this code maps the process to CPU #0.

Task5: Compile and run the program cache_benchmark.c for N=1000,2000,4000,8000,16000 and 32000. Measure the execution time for each case. Draw a graph, where execution time is the y-axis and N is the x-axis. To compile this program you will need extra gcc options, which they are given in the first line of cache_benchmark.c. Make sure the execution time lasts a few seconds, otherwise the execution time measured is not accurate. This is because other processes run too; if the execution time of this process is much larger than the others', then the value measured is more accurate. To this end, you must appropriately specify the 'TIMES' value.

You will realize that the execution time is not linear to the input size. Why?

Answer: As long as X[ ] is smaller than L1 data cache size, then it is stored 'TIMES' times to L1 data cache memory, which is very fast. If X[ ] no longer fits in L1 data cache, then it is written 'TIMES' times to L2 cache, which is slower.

Section 3 – OPTIONAL (NOT ASSESSED)

Size command in Linux

The size command lists the section sizes of executables. For more information type on a shell terminal

man size

To use this command you must compile your program and then type 'size name_of_executable'.

Task6. Use size command for stack_heap.c file. Compile using

gcc stack_heap.c -o p

then run using

size p

You will see that

text	*data*	*bss*	*dec*	*hex*	*filename*
2342	*632*	*8*	*2982*	*ba6*	*p*

Text is the executable size, data is the size of the initialized data, bss is the size of the unitialized data, dec is the sum of text, data and bss and hex is dec in hex.

It is important to note that the aforementioned values do not refer solely to the source code shown in stack_heap.c, but to the libraries used too.

Task7. Write the following outside main() ‘double a;’ and then recompile and run ‘size’ command again. What changed?