Practical Session – Week 6

Objectives

1. To learn how cache memory affects execution time and how to appropriately initialize arrays
2. To develop a software program that measures the L1 data cache size
3. To apply reverse engineering in order to understand how the stack works

Tasks

1. **Efficient Array Initialization**: Open Visual Studio in windows and create a new empty C++ project. Copy paste the code given on DLE (*week6.cpp* file). This program runs multiple times a function that initializes a two dimensional array and measures the execution time ('*row_column_wise()*' function). Please note that to get a good measurement of the execution time, it needs to be at least some seconds (this is because other processes run too); thus, we run the routine multiple times. If the execution time is either too small or too large, modify the '*t*' value accordingly. Get an accurate execution time measurement of '*row_column_wise()*' routine with a) only the first loop kernel uncommented, b) only the second loop kernel uncommented. Have you realised that initializing an array in a row-wise manner is many times faster. Can you tell why? Recall from the lecture, how the array elements are loaded into cache memories.

2. **Experimental procedure computing the L1 data cache size.** Open Visual Studio in windows and create a new empty C++ project. Copy paste the code given on DLE (*week6.cpp file*). Comment the '*row_column_wise()*' and run the '*cache_benchmark()*' routine for *N=1000, 2000, 4000, 8000, 16000, 32000*. Measure the execution time in each case. Make the graph of execution time versus N. Explain the results. Please note that to get a good execution time measurement, execution time needs to be at least some seconds. If the execution time is either too small or too large, modify the '*t*' value accordingly. Try to explain what this program does. Note that when the X[] is larger than the cache size, the execution time is further increased. This is a simple program to measure the L1 data cache size.

3. **Understanding how the Stack Works:** Assume that the following C-code is not given and just its binary is provided. Use gdb debugger to reverse engineer its assembly code. *Make sure you understand where the function operands are stored when a function is called*. *Make sure you understand how the stack works.* Keep in mind that the %rsp stack pointer must always be aligned to 16 and therefore space is wasted. Moreover, keep in mind that you are looking at unoptimized code (remember you compiled without specifying –O1/3); by enabling the compiler to be more aggressive the function is inlined (copied pasted to main()) .

   Every time a function is called, its operands are always stored in the *(%rdi, %rsi, %rdx, % rcx, %r8, %r9*) registers in that order. If there are more than six operands, the rest are stored in the stack. The return value of the function is always stored into %rax. The aforementioned registers are 8bytes each; in the case where the function operands are of 4bytes instead of 8, then the (%edi, %esi, %edx, % ecx, %r8, %r9) and eax registers, are used, respectively (they are the same registers).

```
#include <stdio.h>

int funct1(int a, int b, int c, int d);

int main(){

int a, b, c, d, temp, i;
a=2; b=3;  c=5; d=9;

 temp=funct1(a,b,c,d);

 printf("\nThe result is %d \n", temp);
return 0;
}

int funct1(int a, int b, int c, int d){

int result;
result=(a+b+c+d)<<2;
return result;
}
```


The assembly code of main() will look like this:


```
  0x0000000000400520 <+0>:    push   %rbp            //save old base pointer
  0x0000000000400521 <+1>:    mov    %rsp,%rbp       //make stack pointer the base pointer
0x0000000000400524 <+4>:    sub    $0x20,%rsp      //allocates 32bytes in the stack (rsp must be 16byte aligned)
```



```
  0x0000000000400528 <+8>:    movl   $0x2,-0x14(%rbp)  //store the 1st operand to the stack
  0x000000000040052f <+15>:   movl   $0x3,-0x10(%rbp)  //store the 2nd operand to the stack
  0x0000000000400536 <+22>:   movl   $0x5,-0xc(%rbp)   //store the 3rd operand to the stack
  0x000000000040053d <+29>:   movl   $0x9,-0x8(%rbp)   //store the 4th operand to the stack
  0x0000000000400544 <+36>:   mov    -0x8(%rbp),%ecx   //store the 4th operand to ecx
  0x0000000000400547 <+39>:   mov    -0xc(%rbp),%edx   //store the 3rd operand to edx
  0x000000000040054a <+42>:   mov    -0x10(%rbp),%esi  //store the 2nd operand to esi
  0x000000000040054d <+45>:   mov    -0x14(%rbp),%eax //store the 1st operand to eax
  0x0000000000400550 <+48>:   mov    %eax,%edi         //store the 1st operand to edi
  0x0000000000400552 <+50>:   callq  0x400575 <funct1>  //call funct1. Its operands are in edi,esi,edx,ecx
  0x0000000000400557 <+55>:   mov    %eax,-0x4(%rbp) //store the return value (eax) to the stack
  0x000000000040055a <+58>:   mov    -0x4(%rbp),%eax
  0x000000000040055d <+61>:   mov    %eax,%esi         //store temp to esi
  0x000000000040055f <+63>:   mov    $0x400654,%edi    //store a value related to printf to edi
  0x0000000000400564 <+68>:   mov    $0x0,%eax
  0x0000000000400569 <+73>:   callq  0x4003f0 <printf@plt>  //call printf its operands are in edi,esi
  0x000000000040056e <+78>:   mov    $0x0,%eax         //printf returns nothing
  0x0000000000400573 <+83>:   leaveq                   //copy rbp to rsp and pop rbp
  0x0000000000400574 <+84>:   retq                     //pop %rip
```

The assembly code of funct1() will look like this:

```
0x0000000000400575 <+0>:    push   %rbp                    //save old base pointer
0x0000000000400576 <+1>:    mov    %rsp,%rbp               //make stack pointer the base pointer
0x0000000000400579 <+4>:    mov    %edi,-0x14(%rbp)        //store the 1st func operand to the stack
0x000000000040057c <+7>:    mov    %esi,-0x18(%rbp)        //store the 2nd func operand
0x000000000040057f <+10>:   mov    %edx,-0x1c(%rbp)        //store the 3rd func operand
0x0000000000400582 <+13>:   mov    %ecx,-0x20(%rbp)        //store the 4th func operand
0x0000000000400585 <+16>:   mov    -0x18(%rbp),%eax        //store 'b' to eax
0x0000000000400588 <+19>:   mov    -0x14(%rbp),%edx        //store 'a' to edx
0x000000000040058b <+22>:   add    %eax,%edx               //edx=a+b
0x000000000040058d <+24>:   mov    -0x1c(%rbp),%eax        //put 'c' into eax
0x0000000000400590 <+27>:   add    %eax,%edx               //edx=(a+b)+c
0x0000000000400592 <+29>:   mov    -0x20(%rbp),%eax        //put 'd' to eax
0x0000000000400595 <+32>:   add    %edx,%eax               //eax=(a+b+c)+d
0x0000000000400597 <+34>:   shl    $0x2,%eax               //shift 2 positions to the left
0x000000000040059a <+37>:   mov    %eax,-0x4(%rbp)         //store the result to the stack
0x000000000040059d <+40>:   mov    -0x4(%rbp),%eax       //eax contains the return value (always)
0x00000000004005a0 <+43>:   pop    %rbp                    //restore the base pointer
0x00000000004005a1 <+44>:   retq                           //pop %rip
```

This code is automatically generated without applying any optimizations and this is why it is not efficient. As you can realize there are redundant operations. If you compile the code with '-O2' option and therefore force the compiler to optimize the code, you will realize that the function is inlined into the main() function. This optimization is called 'function inline' and improves the program's performance as all these loads/stores from the stack are not performed.

In Figure 1, how %rsp and %rbp change their values is presented. Please note that each function has its own stack frame (%rsp and %rbp values are appropriately updated). You can use (*x $rbp -0x14*) command to print the contents of a specific stack memory location. Use gdb step by step and print the register/memory contents before and after each instruction.

Some time ago it was noticed that we do not need two registers for manipulating the stack, but just one (%rsp). Gcc compiler keeps the base pointer by default on x86, but allows the optimization with the *'-fomit-frame-pointer'* compilation flag. This compiler option uses only the %rsp and not the %rbp and therefore a) saving two instructions in the prologue and epilogue and b) makes %rbp register available for general usage.

Note that in funct1() temporary data are stored in the stack without adjusting the stack pointer (%rsp). This is applied from gcc only to leaf functions (functions that do not call other functions) and for temporary used data only; these data cannot be used by other functions. This technique saves two instructions (sub and add instructions). This area beyond the location pointed to by *'%rsp'* is also known as red zone and its size is of 128-bytes. Red zone is considered to be reserved and not used in other cases.

You can find more examples is in https://zhu45.org/posts/2017/Jul/30/understanding-how-function-call-works/ .

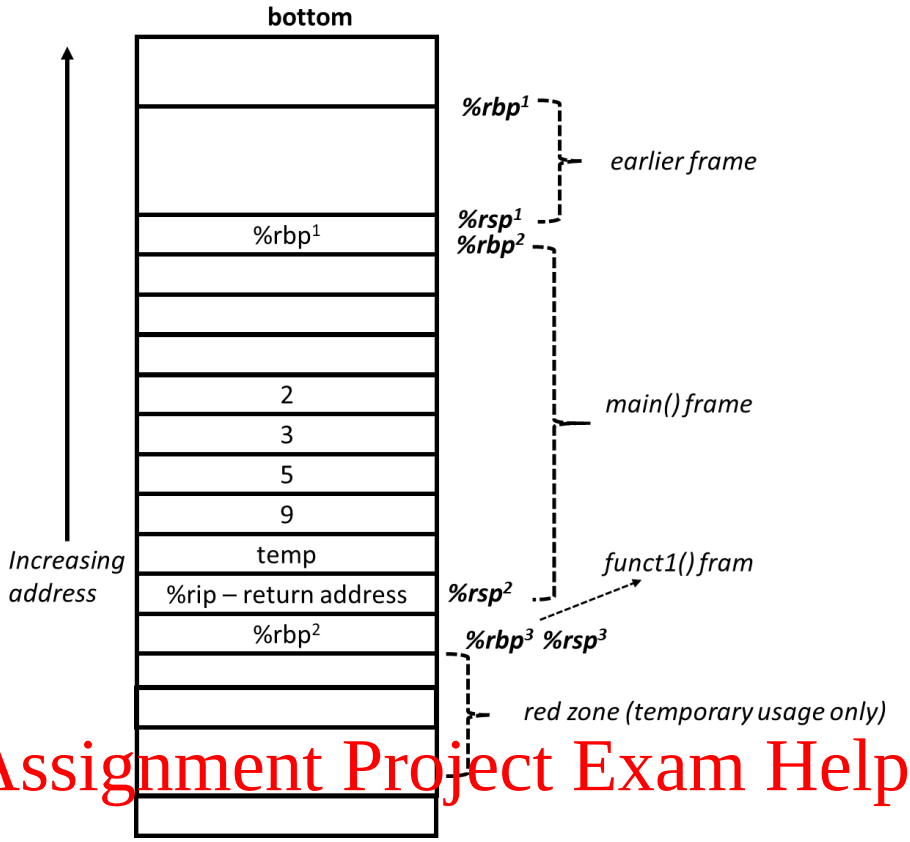

Figure 1. The Stack memory and its contents