Stack Trace

Class Notes

Primary memory (RAM) is logically divided into two regions

- **Kernel space**: The users cannot read or write to these addresses, doing so will result in segmentation fault.

- **User space** : It consists of **Stack segment**(which grows down) , **memory mapping segment** (file and anonymous mappings), **BSS segment** (uninitialized variables filled with zeros) , **Data segment** (static variables initialized) ,**Text segment** (stores the binary images of processes)

Stack:

- Stack is a segment mapped to address space which is used to allocate stack frames to store local data of the processes.

- A stack frame is initialized during function initialization and it is released when a function returns or terminates.

- Stack frames are of two types: Local data frames, Argument frames.

Translating source functions into assembly equivalent :

- To understand the functionality of stack let us consider one C program.

```
main()
{
 int a=100;
 int b=200;
 int c;
 c = a+b;
}
```

Steps involved in translating C code into assembly equivalent:

Assumption:

- Code translation x86 32-bit architecture with GNU compiler.

Translation steps:

- Identify non executable instructions within source file
- Create a function symbol table and resolve all non executable declarations
- Translate executable instruction into assembly source as per the following template.

Function name:

Pre-amble

Function body;

Post amble

Symbol table:

- A table which holds all the information about the variables declared.

```
main()
{
        int a=100;
        int b=200;        } Non executable statements
        int c;
        c = a+b;
}
```

Symbol Name	Type	Composition	Offset Address
a	int	4	-12(%ebp)
b	int	4	-8(%ebp)
c	int	4	-4(%ebp)
	Total	12	

Translating Source Code to Assembly:

```
main()
{
        int a=100;
        int b=100;
        int c;
        c = a+b;
}
```

```
main:
        pushl   %ebp
        movl    %esp,   %ebp
        subl    $12,    %esp
        movl    $100,   -12(%ebp)
        movl    $100,   -8(%ebp)
        movl    -8(%ebp),   %eax
        addl    -12(%ebp),  %eax
        movl    %eax,   -4(%ebp)
        leave
        ret
```

pre - amble

post - amble

- Above shows the assembly code of a main function, which does not calls any other function.

Steps for translating function-calls:

- Starting from the right most argument push each parameter on to top of stack
- Step into called functions using call instruction
- Read the return value of the function from %eax accumulator
- Release the space allocated for arguments.

Now let's see the assembly code for a C program which calls a function.

```
main()
{
        int a=100;
        int b=100;
        int c;
        c = func(a,b);
}
```

```
main:
        pushl   %ebp
        movl    %esp,   %ebp
        subl    $12,    %esp
        movl    $100,   -12(%ebp)
        movl    $100,   -8(%ebp)
        pushl   -8(%ebp)
        pushl   -12(%ebp)
        call    func
        movl    %eax, -4(%ebp)
        addl    $8,   %esp
```

- The pre-amble (**{**) are a pair of instructions responsible for initializing stack frame.
- **Call** instruction in the above code executes the following steps
    1) Stores the return address of the caller function on top of the stack.
    2) Assigns the instruction pointer with the base address of called function.
    3) With respect to position of **ebp**, arguments reside in higher addresses. Local data is allocated in local addresses.

The stack for the above program is shown below