CS118 Lab:1

Saw Thinkar Nay Htoo

March 18, 2016

Model the store (memory)

To model the way memory changes as a computation progresses. We will use some global variables, some local variables, and some registers. Use a diagrams to illustrate the code as it progresses, showing how each instruction changes the state.

Analysis

Create some variables of each type. Use the debugger to find the actual memory addresses of each. Use IPE to label a diagram with those addresses and values. Make new diagram for each statement/instruction that changes the store. e.g. A program that does a summation of several numbers (without control structures such as loops and if statements. Let x = 2, y = 3, and z = 4 in r = x + y + z is an example.

Tools

Photoshop is used instead of IPE, with similiar concept of usage which is using the separate layers so that images of the changes can be made easily.

My very own concept of this lab.

All these weeks, I have been trying to figure out almost everything concerning with assembly language programming. Now I feel like I reached up to a point where I can explain some of the contents back with my own understanding of what I have observed. So this lab will not be perfect, and might a bit different from what the professor expects to see although I included all the lab requirement. However, I will still keep improving with this throughout the whole semester. I will try my best to reference and cite all the contents I have read, found out, analyzed and refered.

This is the SML code of the example algorithm which will be later be converted to C then to ASM.

Prototype SML code

SML

```
val x = 2;
val y = 3;
val z = 4;
val w = 6;
val s = 5;
val r = x + y + z;
val r = r + s;
val r = r + w;
```

These are not assigning to the store (SML does not support assignment like this) It is more like symbolic constants putting the value in a symbol table.

Output from SML code

```
Poly/ML 5.5.2 Release
val x = 2: int
val y = 3: int
val z = 4: int
val w = 6: int
val s = 5: int
val r = 9: int
val r = 14: int
val r = 20: int
```

C code implementation of the SML code

Text written to file lab2.c	"C code explanation"
#include <stdio.h>	:C library
#define x 2	:symbolic constant. int x = 2;
const int y = 3;	:global named (typed) constant
int t;	:(int t;) will be stored in bss segment (Uninitialized data segment)
int w =6;	:data segment
int main()	:
{	:int main() is stored in .text segment
register int z = 4;	:
int r = x + y + z;	:register
static int s =5;	:Being operated on the stack. Now r result is x = 2+3+4 =9
r +=s;	:static local in data segment
r += w;	:In this step, the value of s is added to r. r + 5 = 9+5 = 14
printf("r=%i\n",r);	:In this step, the value of w is added to r. r + 6 = 14+6 = 20
return 0;	:This is to print out the last r value which is 20. Thus the output is "r=20".
}	:return statement
	:

The output when running this C program in the console is:

```
r=20
```

In the debugger, the memory addresses of the 4 variables are:

x: is stored in the instruction (add 83 c0 02) at location 0x8048417

y: 0x80484f0 (rodata)

t: 0x8049710 (bss)

w: 0x8049704 (data)

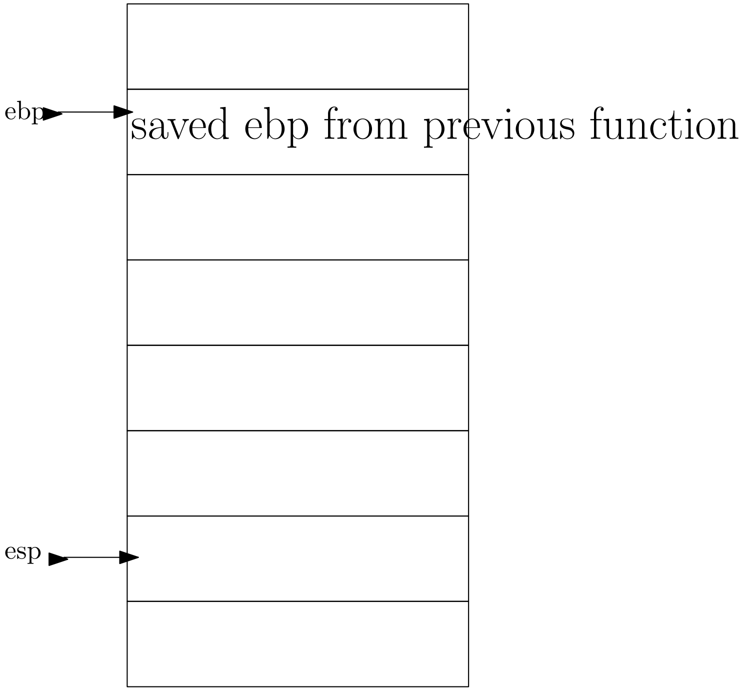
z: ebx register

r: 0xbffff61c r is on the stack 0xc bytes below ebp (changed from 0xbffff61e to 0xbffff61c)

s: 0x8049708 (uninitialized small data)

The stack frame uses register ebp (base pointer) and esp (stack pointer) to find the values related to one function call. The stack pointer points to the most recent value pushed onto the stack. The base pointer points to a mid-point in the frame with the functon parameters above it, and the local variables below it. ebp is at the location oxbffff628.

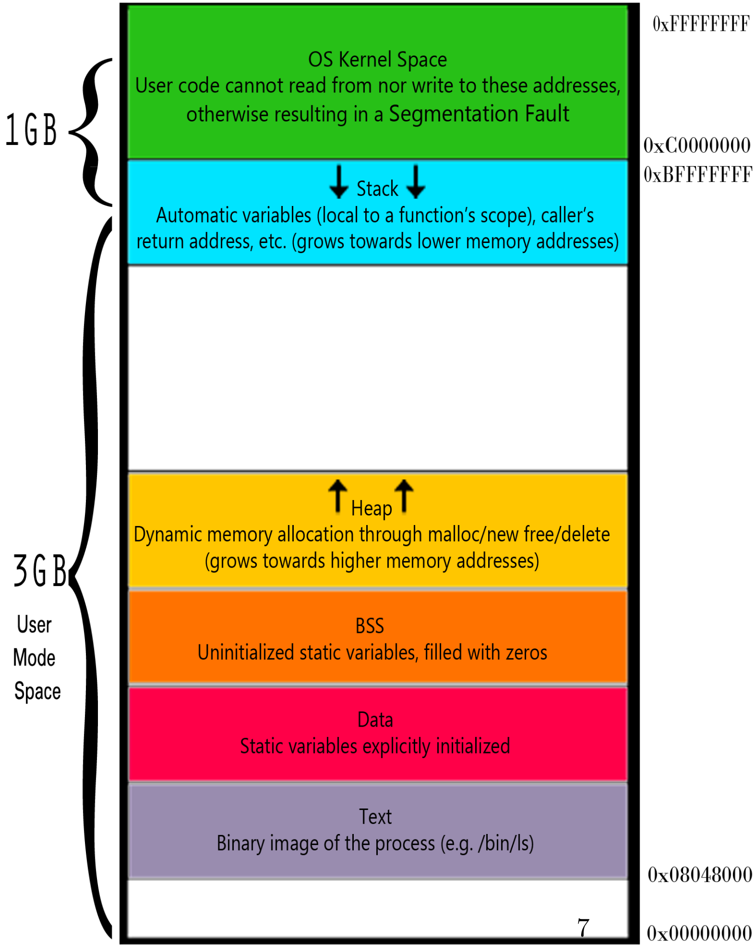
System Stack32 bits = 4 bytes



Top of stack is in lower memory

Memory Segments

Recreated Concept of Memory Segments from this blog.



Below addresses information are obtained from the Symbol Table of the objdump result of my ASM program.

- Stack

- Heap

- BSS
  From 080496c0
  to 080496c8
- Data
  From 080496b0
  to 080496c0
- Text
  From 08048300
  to 08048300
  Though I couldn't find the end address of Text in the symbol table in terminal, it can be obtained from the given memory segment diagram.

Overall explanation of the segments

- Text
  This is the code segment with executable instructions of the ASM program. Usually placed below Stack or Heap in order to prevent overflows from overwriting this segment. Most importantly, the text segment is often Read-only/Execute to prevent from being modifited its instructions accidentally.

- Data
  This segment contains global and static variables which are initialized by the programmer. Furthermore, it can be classified into initialized Read-only(RoData) and initialized Read-Write sections.

- BSS
  (Block Started by Symbol) This segment is also known as Uninitialized data segment. This is a Read-Write segment and usually starts after data segment and have all global and static variables that are initialized to zero or are not initialized. For example, int i.

- Stack
  Stack usuallys located in the higher memory addresses right below the OS kernel space. On the standard x86, it grows downwards to lower addresses, while Heap grows up. (In some architectures, they may grow in different directions. The set of values pushed for one function call is named a stack frame.

- Heap
  Heal is the segment where dynamic memory allocation usually takes place. It is managed by malloc/new and free/delete to adjust its size.

Symbol Table

Symbol Table
In my opinion, it is definitely important to be able to handle well with symbol table while using GDB.
Click here to get the simple idea of what a symbol table is. In the symbol table, size, address, segments and other important information are shown clearly in each catagory.
terminal command to see the Symbol Table of the labasm file:
objdump -x labasm

Symbol Table result in terminal

```
SYMBOL TABLE:
```

Start and end address of BSS segment is shown below.

```
080496c8 g       .bss   00000000              _end
08048300 g     F .text  00000000              _start
080484d8 g     O .rodata        00000004              _fp_hw
080496c0 g       .bss   00000000              __bss_start
```

Start and end address of DATA segment is shown below.

```
080496b0  w      .data  00000000              data_start
00000000       F *UND*  00000000              printf@@GLIBC_2.0
080496c4 g     O .bss   00000004              t
080496c0 g       .data  00000000              _edata
080484c4 g     F .fini  00000000              _fini
080496b0 g       .data  00000000              __data_start
```

Translation of C code to ASM code

- `#define x 2` → .equ x,2
- const int y = 3 → .section .rodata y: .long a
- int t; → .bss .comm t,4
- int w =6; → .data w: .long 6

Text written to file lab2.s

```
.equ x,2
.section .rodata
y: .long 3
.bss
.comm t,4
.data
w: .long 6
```

- int main() → .text .globl main main:
- `printf("r=%in",r");` →
  - .section .rodata msg: .string `"r=%i\n"`
  - .text call printf

Text appended to file lab2.s

```
.text
.globl main
main:
 .section .rodata
 msg: .string "r=%i\n"
.data
s:.long 5
 .text
```

- `int r = x+y+z`

Below is the ASM code for above calculation. 4 bytes is substracted from the stack where esp is pointing at. Then the empty space is filled with x. Then y is moved to eax. Then the value of eax is add to the value of the address where esp is pointing, giving out the result of 5. In next step, 4 is moved to ebx and it is add to the value of the address where esp is pointing which is 5, giving out the result of 9.

Text appended to file lab2.s

```
sub $4,%esp # make room on stack for local variable
movl $x,(%esp) #derefrence the point just like asteroid in CPP.
mov y,%eax
add %eax,(%esp)
mov (%esp),%ebx #ebx now has 5
mov $4,%ebx
add %ebx,(%esp)
mov (%esp),%edx #edx now has 9
```

```
Breakpoint 1, main () at lab2.s:16
16        sub $4,%esp # make room on stack for local variable
(gdb) info r
eax            0x1      1
ecx            0xfa7de268       -92413336
edx            0xbffff654       -1073744300
ebx            0xb7fcc000       -1208172544
esp            0xbffff62c       0xbffff62c
ebp            0x0      0x0
```

This is the initial values and addresses of the registers before making changes.

`sub $4,%esp`

After above line of code, 4 bytes substracted from the stack. Initial esp address 0xbffff62c, then becomes 0xbffff628.

```
$1 = (void *) 0xbffff628
```

`mov y,%eax`

After above line of code, eax now holds the value ”3”.

```
(gdb) p $eax
$1 = 3
```

`add %eax,(%esp)`

`mov(%esp),%ebx`

After the addition is operated, the value, pointed by esp, is moved to ebx in order to print out the result.

```
(gdb) p $ebx
$2 = 5
```

`mov $4,%ebx`

Now a new value is moved into ebx.

```
(gdb) p $ebx
$3 = 4
```

```
add %ebx,(%esp)
mov (%esp),%edx
```

After the addition is operated, the value, pointed by esp, is moved to edx in order to print out the result.

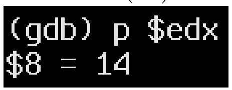
- `int r += s;`

Text appended to file lab2.s

mov s,%ebx
add %ebx,(%esp)
mov (%esp),%edx #edx now has 14

```
mov s,%ebx
add %ebx,(%esp)
mov (%esp),%edx
```

After the addition is operated (s + 9), the value, pointed by esp, is moved to edx in order to print out the result (14).



- `int r += w;`

Text appended to file lab2.s

|*mov w,%ebx*
|*add %ebx,(%esp)*
|*mov (%esp),%edx #edx now has 20*

```
mov w,%ebx
add %ebx,(%esp)
mov (%esp),%edx
```

After the addition is operated (w + 14), the value, pointed by esp, is moved to edx in order to print out the result (20).

```
(gdb) p $edx
$9 = 20
```

- Now print the value of r

Text appended to file lab2.s

> *push (%esp)*
> *push $msg*
> *call printf*
> *add $8, %esp #clean up stack parameters*
> *add $4, %esp #clean up local parameters*
> *ret*

```
   add $8, %esp
add $4, %esp
ret
```

In this case, the print value of esp should be ...28. I don't what happened during the process.

```
(gdb) p $esp
$10 = (void *) 0xbffff638
```

After

```
push (%esp)
```

```
(gdb) p $esp
$11 = (void *) 0xbffff634
```

After

```
push $msg
```

```
(gdb) p $esp
$12 = (void *) 0xbffff630
```

After `call printf`
The address goes back to the initial value after cleaning up the stack and local parameters, by adding 8 and 4 to the addresses.

```
(gdb) p $esp
$13 = (void *) 0xbffff62c
```

LABASM OUTPUT

```
debian@debian:~/labs/lab2$ ./labasm
r=20
```

Shell scripts to make processing easier

This shell script is used to make extracting and processing the source code easier. The -g option to the gcc compiler adds debugging symbols so we can refer to variables even though they are not normally stored in the object code.

Text written to file labcode.sh

```
docsml lab2.doc
gcc -Wall -o labc lab2.c
gcc -Wall -o labasm lab2.s

```

Text written to file labdbg.sh

```
docsml lab2.doc
gcc -Wall -g -o labc lab2.c
gcc -Wall -g -o labasm lab2.s

```

Text written to file labdoc.sh

```
doctex lab2.doc
pptexenv /home/debian/texfot.pl pdflatex lab2.tex
```

Bourne Shell

```
chmod 755 labcode.sh
chmod 755 labdoc.sh
chmod 755 labdbg.sh
poly < lab2.sml > result
./labc > cresult

```

Text written to file dbg_cmds

```
b main
r
info r
```

warning messgae: "Source file is more recent than executable."

Bourne Shell

```
touch labc #to get rid of the warning message mentioned above.
gdb --quiet -x dbg_cmds labc | grep -v 'Ignore\|Quit'| tee labc.log

```

...below is the output of automated script for GDB:

```
Reading symbols from labc...(no debugging symbols found)...done.
Breakpoint 1 at 0x804840a

Breakpoint 1, 0x0804840a in main ()
eax            0x1 1
ecx            0xbffff560 -1073744544
edx            0xbffff584 -1073744508
ebx            0xb7fcc000 -1208172544
esp            0xbffff540 0xbffff540
ebp            0xbffff548 0xbffff548
esi            0x0 0
edi            0x0 0
eip            0x804840a 0x804840a <main+15>
eflags         0x286 [ PF SF IF ]
cs             0x73 115
ss             0x7b 123
ds             0x7b 123
es             0x7b 123
fs             0x0 0
gs             0x33 51
(gdb) quit
A debugging session is active.

Inferior 1 [process 1059] will be killed.
```