William Emmanuel
wre9fz
inlab8.pdf
March 26

12x86 Post-Lab Report

Part 1: Paremeter passing in assembly

The following C++ code was compiled into assembly:

```
void paremPass(int x, int y) { cout << x+y << endl; }
int main() { paremPass(1, 2);  return 0; }
```

```
main:
.LFB967:
        .cfi_startproc
        push    ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        mov     ebp, esp
        .cfi_def_cfa_register 5
        and     esp, -16
        sub     esp, 16
        mov     DWORD PTR [esp+4], 2
        mov     DWORD PTR [esp], 1
        call    _Z9paremPassii
        mov     eax, 0
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
```

```
_Z9paremPassii:
.LFB966:
        .cfi_startproc
        push    ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        mov     ebp, esp
        .cfi_def_cfa_register 5
        sub     esp, 24
        mov     eax, DWORD PTR [ebp+12]
        mov     edx, DWORD PTR [ebp+8]
        add     eax, edx
        mov     DWORD PTR [esp+4], eax
        mov     DWORD PTR [esp], OFFSET FLAT:_ZSt4cout
        call    _ZNSolsEi
        mov     DWORD PTR [esp+4], OFFSET
FLAT:_ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_
        mov     DWORD PTR [esp], eax
        call    _ZNSolsEPFRSoS_E
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
```

This demonstrates a simple int function call by value. Two ints are passed into the paremPass function from main, are added together, and the sum printed. The relevant assembly code is bolded for the main and paremPass functions above.

In both functions, the first steps are the pushing of an old base pointer and resetting of the base pointer to the current stack pointer. This is undone in the final lines of each function, when the old rbp is restored from the stack.

In main, the first step of calling paremPass is the input of parameters. The inputs must be placed in locations that are accessible for the paremPass function—on to the stack. First, the stack pointer is masked with -16. This gives some space in between the stack pointer and the variables to be stored, which could increase performance. Then, space for these variables is made on the stack. Since the stack starts at the highest memory address and grows downwards, allocation is done by subtracting from the stack address. From this point, x and y are placed on the stack. Int y is first copied into the stack ,as it is the second parameter. The parameters are pushed in reverse order specifically for cases in which an unknown number of parameters are being passed into a function. This ensures that [ebp+8] will always reference the first parameters passed in in the callee subroutine. After y is passed in, x is then put on the

stack. These variable stores are done via DWORD pointers—an int in c++ is 4 bytes, the size of a double in assembly. So, the stack pointer + 4 and the stack pointer base address correspond to the storage locations for int y and int x respectively.

Once both parameters are in a location where paremPass can access them, the function is called. Again, the base pointer is pushed and reset to the stack pointer location. Then, the 24 bytes are allocated on the stack (for reasons not explained fully here—for a combination of the final sum computed and I/O). Then, the ebp+12 and ebp+8 variables (y and x) are loaded into the registers eax and edx. Edx is then added to eax, and this value is stored in the allocated space esp+4. Then, I/O takes place (too complicated to explain!). Finally, the subroutine returns and main resumes. So, in short: the caller function allocated space on the stack, placed the parameters in reverse order, called the subroutine which accesses these stack variables, added them, and stored them for other subroutines.

The following C++ code was compiled into assembly to illustrate passing of double data type:

```
void paremPass(float x, float y) {int a = 0;   a = x+y;}
int main() { paremPass(1.0, 2.0); return 0; }
```

```
void paremPass(float x, float y) {
_Z9paremPassff:
.LFB966:
        .cfi_startproc
        push    ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        mov     ebp, esp
        .cfi_def_cfa_register 5
        sub     esp, 20
        mov     DWORD PTR [ebp-4], 0
        fld     DWORD PTR [ebp+8]
        fadd    DWORD PTR [ebp+12]
        fnstcw  WORD PTR [ebp-18]
        movzx   eax, WORD PTR [ebp-18]
        mov     ah, 12
        mov     WORD PTR [ebp-20], ax
        fldcw   WORD PTR [ebp-20]
        fistp   DWORD PTR [ebp-4]
        fldcw   WORD PTR [ebp-18]
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
```

```
main:
.LFB967:
        .cfi_startproc
        push    ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        mov     ebp, esp
        .cfi_def_cfa_register 5
        sub     esp, 8
        mov     eax, 0x40000000
        mov     DWORD PTR [esp+4], eax
        mov     eax, 0x3f800000
        mov     DWORD PTR [esp], eax
        call    _Z9paremPassff
        mov     eax, 0
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
```

There are just a few differences between the int passing and float passing. Again, room is made on the stack for parameter variables (DWORD PTR, 4 byte float pointers). The paremPass function is then called. A different set of register operators are used for the floats—for example, fadd is used instead of add. Again, this subroutines operates much the same way as the int program above. Room is made on the stack for variables (including a local variable, which is stored as 0). Any locations above esp (esp + N) are parameters passed in; any locations below esp (esp - N) are local variables declared in the subroutine. So, one can see variable a being stored at ebp – 4, and a parameter being accessed at ebp + 12. Again, the program functions much the same as a the int program. If doubles were input instead of

floats, a quad pointer would have to be used—QWORD PTR. If something like a char were input, then only a BYTE PTR would have to be used. Again, the programs function the same, only possibly changing in the amount of space allocated for the pointer (whether it be 1, 2, 4, or 8 bytes).

A simple C++ program was written to illustrate pass by reference in assembly.

```
void paremPass(int &x, int &y) { int temp = x;  x = y;  y = temp; }
int main() { int a = 1; int b = 2; paremPass(a, b); return 0; }
```

```
main:
.LFB967:
        .cfi_startproc
        push    ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        mov     ebp, esp
        .cfi_def_cfa_register 5
        sub     esp, 24
        mov     DWORD PTR [ebp-8], 1
        mov     DWORD PTR [ebp-4], 2
        lea     eax, [ebp-4]
        mov     DWORD PTR [esp+4], eax
        lea     eax, [ebp-8]
        mov     DWORD PTR [esp], eax
        call    _Z9paremPassRiS_
        mov     eax, 0
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
```

```
_Z9paremPassRiS_:
.LFB966:
        .cfi_startproc
        push    ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        mov     ebp, esp
        .cfi_def_cfa_register 5
        sub     esp, 16
        mov     eax, DWORD PTR [ebp+8]
        mov     eax, DWORD PTR [eax]
        mov     DWORD PTR [ebp-4], eax
        mov     eax, DWORD PTR [ebp+12]
        mov     edx, DWORD PTR [eax]
        mov     eax, DWORD PTR [ebp+8]
        mov     DWORD PTR [eax], edx
        mov     eax, DWORD PTR [ebp+12]
        mov     edx, DWORD PTR [ebp-4]
        mov     DWORD PTR [eax], edx
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
```

This simple swap program works mostly by mechanisms discussed above. Again, first the int parameters are pushed onto the stack. Allocation is made on the stack for two pointers to 4 byte ints at spaces esp+4 and esp. Before this, the variables a and b are loaded locally at locations ebp-4 and ebp-8. These are the actual locations of variables a and b; the *address* of this pointer is then loaded into eax via the lea command. The address of the esp+4 DWORD PTR is then set to this address (that is, the address ebp-4). The same is done for esp and ebp + 8. So, we can see here how things are passed by reference: the variables are first stored in a location in relation to the base pointer, not the stack pointer. Then, the effective address of this variable is loaded into the register, which is copied into the stack for passing into a subroutine. The subroutine can then manipulate those parameters using the address to the original base-pointer defined variables. This seems to function identically to passing by pointer.

The following C++ code was used to illustrate the passing of arrays as a parameter in assembly:

```
void paremPass(int a[], int length) { int b = a[1]; int c = a[4]; }
int main() {int a[5] = { 1, 2, 3, 4, 5};paremPass(a, 5); return 0; }
```

William Emmanuel
wre9fz
inlab8.pdf
March 26

```
_Z9paremPassPii:
.LFB966:
        .cfi_startproc
        push    ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        mov     ebp, esp
        .cfi_def_cfa_register 5
        sub     esp, 16
        mov     eax, DWORD PTR [ebp+8]
        mov     eax, DWORD PTR [eax+4]
        mov     DWORD PTR [ebp-8], eax
        mov     eax, DWORD PTR [ebp+8]
        mov     eax, DWORD PTR [eax+16]
        mov     DWORD PTR [ebp-4], eax
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
.LFE966:
        .size   _Z9paremPassPii, .-_Z9paremPassPii
        .globl  main
        .type   main, @function
```

```
main:
.LFB967:
        .cfi_startproc
        push    ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        mov     ebp, esp
        .cfi_def_cfa_register 5
        sub     esp, 40
        mov     DWORD PTR [ebp-20], 1
        mov     DWORD PTR [ebp-16], 2
        mov     DWORD PTR [ebp-12], 3
        mov     DWORD PTR [ebp-8], 4
        mov     DWORD PTR [ebp-4], 5
        mov     DWORD PTR [esp+4], 5
        lea     eax, [ebp-20]
        mov     DWORD PTR [esp], eax
        call    _Z9paremPassPii
        mov     eax, 0
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
```

The array passing works very similarly to the pass by reference. First, the esp in this program is incremented by 40 bytes to make room for the array and pointers. Then, the array is declared in terms of ebp—each int is 4 bytes, so the addresses start at ebp-20 and extend to ebp-4. Then, since the size is a parameter, it is loaded at esp+4. Eax is then set to the first element of the array, ebp-20. This is done via the load effective address operator. Then, this address is stored in esp so that the function can be called.

In the subroutine, two local variables are allocated. Then, to set those variables equal to the values in the passed in array, the effective address is loaded from ebp + 8. Then, 4 bytes are added to this effective address to move from element 0 to element 1 (size of one int). Then, a local variable at ebp-8 is set equal to eax. So, the access of elements goes as follows: load from ebp+8 (address to address of array start). Then, 4 is added to the address of the array start, and it is dereferenced to get the actual value.

Finally, the passing of a user defined object was explored. The linked list object from a previous lab was used. In short, a list object was created and passed into a subroutine. This subroutine then inserted an element using the insertAtTail() function. The code for this got too complex and long to list here; but, it functioned the same as previously discussed operations. First, space was allocated for the pointer to the full List object. The instantiation of this List object was interesting—it was passed to a subroutine, and this subroutine returned the address of the object in eax. This address was then passed into the paremPass function. Again, like the array the base of the object served as the starting point for data. To access other data, a certain number was added to this base memory address.

William Emmanuel
wre9fz
inlab8.pdf
March 26

Part 2: Objects in Assembly

A trivial function was created called “object” (actual code can be found at end of lab report). It included an add function (stored an input int), a public field and a private field. A test .cpp file then called on these functions and fields. The following relevant assembly code was produced, with important lines bolded:

From testObject.s:

```
main:
.LFB966:
        .cfi_startproc
        push    ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        mov     ebp, esp
        .cfi_def_cfa_register 5
        and     esp, -16
        sub     esp, 32
        lea     eax, [esp+20]
        mov     DWORD PTR [esp], eax
        call    _ZN6objectC1Ev
        mov     DWORD PTR [esp+4], 3
        lea     eax, [esp+20]
        mov     DWORD PTR [esp], eax
        call    _ZN6object3addEi
        mov     eax, DWORD PTR [esp+20]
        mov     DWORD PTR [esp+28], eax
        mov     eax, 0
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
```

From object.s

```
_ZN6objectC2Ev:
.LFB967:
        .cfi_startproc
        push    ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        mov     ebp, esp
        .cfi_def_cfa_register 5
        sub     esp, 16
        mov     DWORD PTR [ebp-8], 0
        mov     DWORD PTR [ebp-4], 1
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
_ZN6object3addEi:
.LFB969:
        .cfi_startproc
        push    ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        mov     ebp, esp
        .cfi_def_cfa_register 5
        sub     esp, 16
        mov     eax, DWORD PTR [ebp+12]
        mov     DWORD PTR [ebp-4], eax
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
```

First, the instantiation of the object begins by allocating the needed memory for all fields held in the object (in this case, 32 bytes). The allocation takes place by moving back the stack pointer 32 bytes. To ensure efficiency, the stack pointer is masked with -16—an operation that would increase the space in between the instantiated variables and the stack pointer.

From there, the memory address of this data block is loaded (esp+20 in this case) into the stack pointer. This sets up everything for the callee constructor function—when it is called, it can immediately begin to store variables, since the stack pointer is set to esp+20 location (around which are bytes allocated for the object).

Then, the constructor is called. The compiler makes a separate .s file for the object cpp file; it is here that the constructor is found. In this constructor, the data fields are given values. Remember, these fields already were allocated in the main assembly subroutine. They are in a sense allocated again—the

stack pointer is moved down, but really this space was allocated previously. The fields “pub” and “priv” (pub is public int, priv is private int) are given initial values of 1 and 0 respectively; the values can be found in the first 8 bytes allocated (rbp-4 and rbp-8, each int 4 bytes long). This initialization is all the constructor does; it then returns back to the main subroutine for continued execution. Again, the fields of the object instantiated are almost like an array—they occupy a block of memory together, and are accessed via adding to the offset “base” pointer of the object. The base is accessed by changing the stack pointer to this address.

A public method of the object is then called. This is done by first loading the memory address of the object instantiated (the object “x” constructed previously) into a register—in this example, the effective address of [esp+20] is loaded. This valued is copied into the stack pointer, and the method is called. The method of an object is virtually indistinguishable from a public function. The compiler treats them the same; then, when a method is called on an object, the object address is passed into the method along with any other parameters. Again, the method is object-independent—it only knows where object fields are because the address block is passed in.

When creating an object with many different types (ints, doubles, floats, chars, etc.) the same ideas apply. Basically, an array is allocated—all of the values for each field are found next to each other, and accessing them is just a matter of knowing the base address.

For accessing of variables outside of methods or functions, the compiler will just directly reference it. In the above code for example, object.pub is accessed via mov eax, [esp+20]. There is no checking of whether this variable is public or private—this is done all by the compiler at compilation time.

(I found this post particularly helpful in writing this :
http://answers.google.com/answers/threadview/id/488746.html
)

William Emmanuel
wre9fz
inlab8.pdf
March 26

Code used for objectTest.cpp, object.cpp and object.h

From objectTest.cpp:

```
#include "object.h"
int main() {
  object x = object();
  x.add(3);
  int a = x.pub;
  return 0;
}
```

From object.h:

```
class object {
 public:
  object();
  int pub;
  void add(int prive);
 private:
  int priv;
};
```

From object.cpp:

```
object::object() {
  int priv = 0;
  int pub = 1;
}
void object::add(int prive) {
  int priv = prive;
}
```