CMPT 295

Unit - Machine-Level Programming

Lecture 14 – Assembly language – Function Call and Stack

Last Lecture

Compiler can produce different instruction combinations when assembling the same C code.

- In x86-64 assembly, there are no conditional statements, however, we can alter the execution flow of a program by using …
  - **cmp*** instruction (compare)
  - **jX** instructions (jump)
  - **call** and **ret** instructions
  - **cmovX** instructions -> conditional move
- In x86-64 assembly, there are no iterative statements, however, we can alter the execution flow of a program by using …
  - **cmp*** instruction
  - **jX** instructions (jump)
- CPU uses these *condition codes* to decide whether a …
  - **jX** instruction (conditional jump) is to be exectued or a
  - **cmovX** instruction (conditional move) is to be exectued
- 2 loop patterns:
  - "*coding the false condition first*" -> `while` loops (hence `for` loops)
  - "*jump-in-middle*" -> `while`, `do-while` (hence `for` loops)

cmp* and **test*** instructions set *condition codes*

Homework

Question about while loop:

in C:

```
while (x < y) {   ①
   // stmts
}
return;
```

③ y-x >0 → y>x loops → g

<0 → y<x exits loop → l

=0 → y=x exits loop → e

in assembly: `# x in %edi, y in %esi`

```
loop:
②   cmpl %edi, %esi
         x     y
    jl endloop
    # stmts
    jmp loop
endloop:
    ret
```

Loop Pattern 1

```
loop:
  if cond false
    goto done:
  stmts
  goto loop:
done:
```

Would this assembly code be the equivalent of our C code? Not quite! We need jle

Demo: alternative way of implementing `if/else` in assembly language

- **`ifelse.c`** and **`ifelse.s`** posted on our course web site

We shall have a look at this code during lecture 15
our review lecture!

Today's Menu

- Introduction
  - C program -> assembly code -> machine level code
- Assembly language basics: data, `move` operation
  - Memory addressing modes
- Operation `leaq` and Arithmetic & logical operations
- Conditional Statement – Condition Code + `cmovX`
- Loops
- Function call – Stack
  - Overview of Function Call
  - Memory Layout and Stack - x86-64 instructions and registers
  - Passing control
  - Passing data – Calling Conventions
  - Managing local data
  - Recursion
- Array
- Buffer Overflow
- Floating-point operations

What happens when a function (*caller*) calls another function (*callee*)?

1. **Control** is passed (i.e., program counter PC is set)…
   - To the beginning of the code in ***callee*** function
   - Back to where ***callee*** function was called in *caller* function
2. **Data** is passed …
   - To ***callee*** function via *function parameter(s)*
   - Back to *caller* function via *return value*
3. **Memory** is …
   - Allocated when ***callee*** function starts executing
   - Deallocated when ***callee*** function stops executing

- Above mechanisms implemented with machine code instructions and described as a set of conventions (which is part of ISA)

caller

```
void who(…) {
   int sum = 0;
   ...
   y = amI(x);
   sum = x + y;
   return;
}
```

value of v[t]

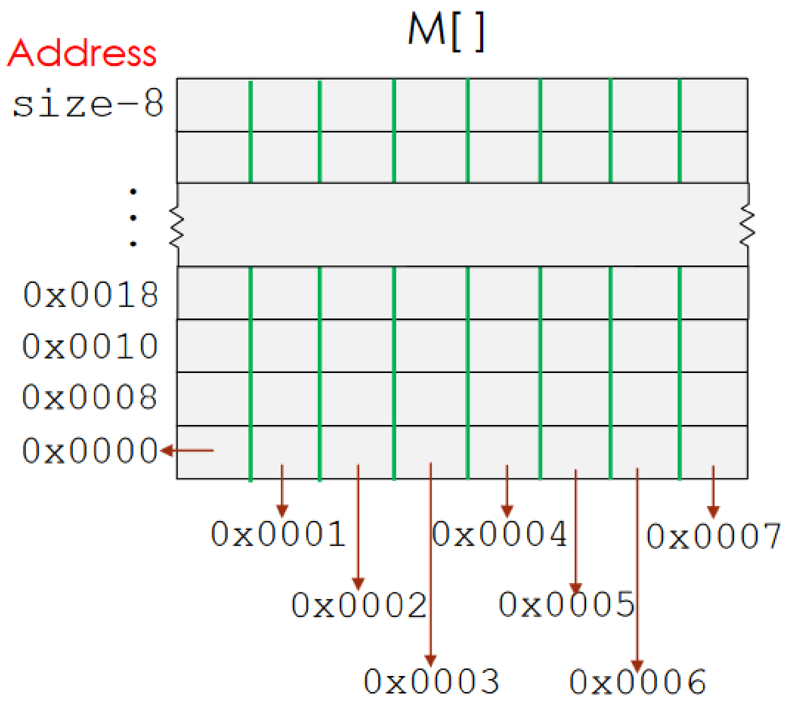
callee

```
int amI(int i)
{
   int t = 3*i;
   int v[10];
   ...
   return v[t];
}
```

Remember from Lecture 2: Closer look at memory

- Seen as a linear (contiguous) array of bytes
- 1 byte (8 bits) smallest addressable unit of memory
  - Each byte has a unique address
  - *Byte-addressable* memory
- Computer reads a ***word*** worth of bits at a time (=> word size)

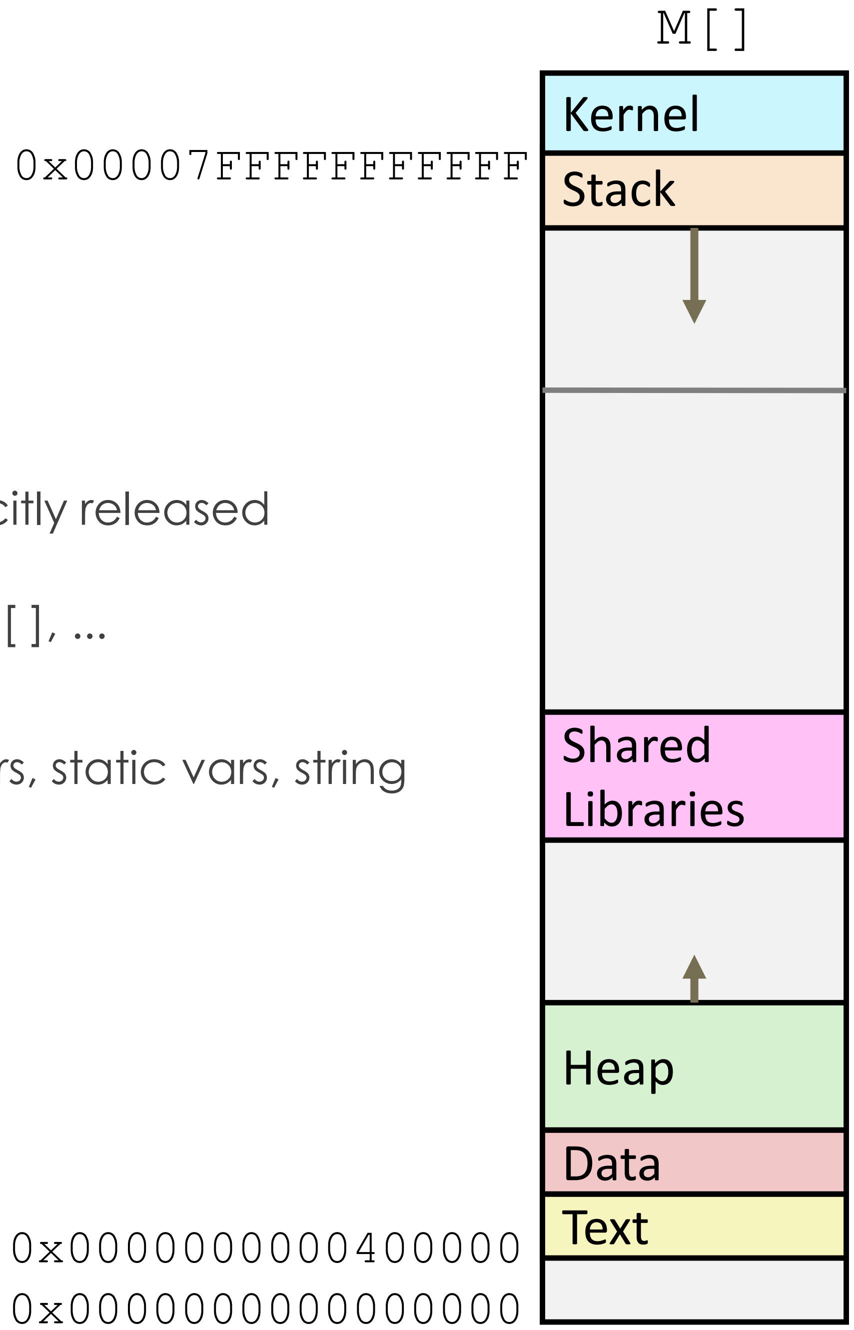
Compressed view of memory



Memory Layout

segments

- **Stack**
  - Runtime stack, e. g., local variables
- **Heap**
  - Dynamically allocated as needed, explicitly released (freed)
  - When call malloc(), free(), new(), delete[ ], ...
- **Data**
  - Statically allocated data, e.g., global vars, static vars, string constants
- **Text**
  - Executable machine instructions
  - Read-only
- **Shared Libraries**
  - Executable machine instructions
  - Read-only



Memory Allocation Example

Where does everything go?

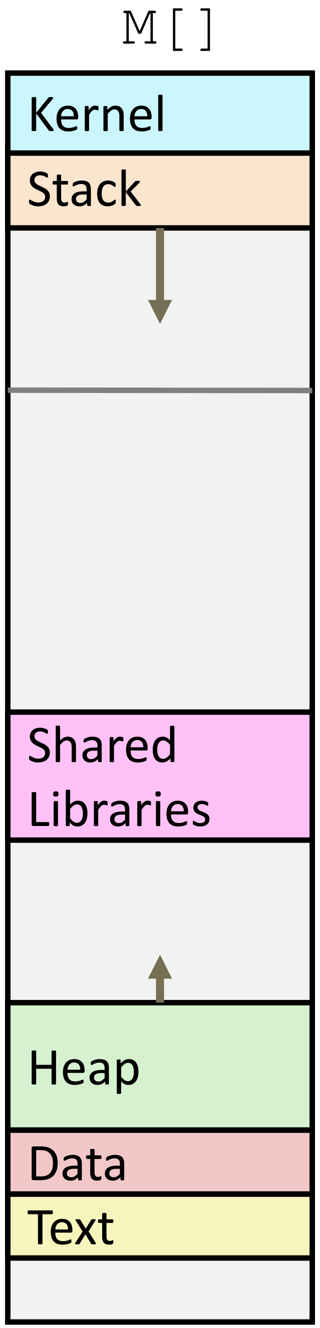
```
#include ...

char hugeArray[1 << 31]; /* 2^31 = 2GB */
int global = 0;

int useless(){ return 0; }

int main ()
{
    void *ptr1, *ptr2;
    int local = 0;
    ptr1 = malloc(1 << 28); /* 2^28 = 256 MB*/
    ptr2 = malloc(1 << 8);  /* 2^8  = 256 B*/

/* Some print statements ... */
}
```



Closer look at function call pattern

Some segment in M[]

- A function may call a function, which may call a function, which may call a function, …

```
who(…) {
   . . .
   . . .
   are();
   . . .
   . . .
}
```

```
are(…) {
   . . .
   you();
   . . .
   you();
   . . .
}
```

```
you(…) {
   . . .
   . . .
   . . .
   . . .
   . . .
}
```

- When a function (***callee***) terminates and returns, its most recent ***caller*** resumes which eventually terminates and returns and its most recent ***caller*** resumes …
- Does this pattern remind you of anything?

Stack - Review

Definition:

A stack is a last-in-first-out (LIFO) data structure with two characteristic operations:

- `push(data)`
- `data = pop( ) or pop(&data)`

Do not have access to anything except what is on (at) top

Source: https://www.thebroad.org/art/robert-therrien/no-title-8

Summary

- Function call mechanisms: 1) passing control, 2) passing data, 3) managing local data on the stack
- Memory layout
  - Stack (local variables …)
  - Heap (dynamically allocated data)
  - Data (statically allocated data)
  - Text / Shared Libraries (program code)
- A “stack” is the right data structure for function call / return

Next Lecture

- Introduction
  - C program -> assembly code -> machine level code
- Assembly language basics: data, `move` operation
  - Memory addressing modes
- Operation `leaq` and Arithmetic & logical operations
- Conditional Statement – Condition Code + `cmovX`
- Loops
- Function call – Stack
  - Overview of Function Call
  - Memory Layout and Stack - x86-64 instructions and registers
  - Passing control
  - Passing data – Calling Conventions
  - Managing local data
  - Recursion
- Array
- Buffer Overflow
- Floating-point operations