CSE 30: Computer Organization and Systems Programming

Lecture 5: Memory organization (contd)
C run time environment

Diba Mirza
University of California, San Diego
Announcements

• HW1/PA1 is now available on TED, due next Monday (10/12) at 11:30pm
Units of Memory

1KB = $2^{10}$ bytes

1MB = $2^{10}$ KB = $2^{20}$ bytes

1GB = $2^{10}$ MB = $2^{30}$ bytes

1TB = $2^{10}$ GB = $2^{40}$ bytes

1PB = $2^{10}$ TB = $2^{50}$ bytes

1EX (exa) = $2^{10}$ PB = $2^{60}$ bytes
Memory Organization

- If N bits are used to represent memory addresses in a computer, we say it has an N-bit address space.

- ARM is a 32-bit architecture.
  1. 32 bit address space
  2. 32 bit registers
  3. 32 bit instructions
  4. 32 bit (4 byte) integers

\[
\text{sizeof(int)} \quad \text{returns size of integer in bytes}
\]

\[
\text{each register is 4 bytes (32 bits)}
\]
Memory

PI Q: How much memory can be supported on a 32-bit machine?

A. 2GB
B. 4GB
C. 8GB
D. 16GB

How about a 64-bit machine?

32-bit machine \( \Rightarrow \) 32-bit address space

\[ \Rightarrow 2^{32} \text{ addresses} \Rightarrow \boxed{2^{32} \text{ bytes in memory}} \]

\[ \Rightarrow 4 \times 2^{30} \text{ bytes} = 4 \text{ GB} \]

1 GB = \( 2^{30} \) bytes

\[ 2^{64} \text{ bytes} = 2^4 \times 2^{60} \text{ bytes} \]

\[ \Rightarrow 16 \text{ Exa bytes} \approx 16 \text{ EB} \]
The C runtime environment
Steps in program translation

Program in C
Helloworld.c

Program:
Text file stored on computers hard disk or some secondary storage

Compile Time
Compiler

Executable:
Program in machine code + Data in binary

Run Time
Hardware

1000110001100010000000000000000000
1000110011110010000000000000100
1010110011110010000000000000000
1010110001100010000000000000100
1010110001100010000000000000100
What does gcc do?

$ gcc hello.c

“Source”
Program in C

hello.c

#include <stdio.h>
void func1(int a, char *b)
{
    if(a > 0)
    {
        *b = ‘a’ ;
    }
}
int main()
{
    ....
    func1();
    printf(“\abc”);
}

“Executable”:
Equivalent program in machine language

0000 1001 1100 0110
1010 1111 0101 1000
1010 1111 0101 1000
0000 1001 1100 0110
1100 0110 1010 1111
0101 1000 0000 1001
0101 1000 0000 1001
1100 0110 1010 1111
What does gcc do?

$ gcc hello.c \leftarrow \text{compile}$

$ ./a.out \leftarrow \text{run} \quad \text{(executable loaded in memory and processed)}$

Also referred to as “running” the C program

"Source": Program in C

"Executable": Equivalent program in machine language
Steps in gcc

• The translation is actually done in a number of steps
Include code written by others

- Code written by others (libraries) can be included
- `ld` (linkage editor) merges one or more object files with the relevant libraries to produce a single executable

```
hello.c  \rightarrow\  cpp  \rightarrow\  cc1  \rightarrow\  gcc
        \     \   \     \   \     \   \     \   \     \   \     \   \     \   \     \   \     \\
          \   \   \   \   \   \   \   \   \   \   \   \   \   \   \   \   \   \   \   \   \\
             \   \   \   \   \   \   \   \   \   \   \   \   \   \   \   \   \   \   \   \\
                hello.s  \rightarrow\  as  \rightarrow\  ld  \rightarrow\  a.out
```

Library files e.g. \texttt{math.o}: the math library
Steps in gcc

• Ask compiler to show temporary files:
  
  $ gcc –S hello.c → hello.s (Assembly code)
  $ gcc –c hello.c → hello.o (Object file, program in machine code)
  $ gcc –o prog_hello hello.c → hello.o (named executable, called prog_hello)
  $ gcc file1.o file2.o –o prog_hello → creates an executable from object files

```
$ gcc –S hello.c
$ gcc –c hello.c
$ gcc –o prog_hello hello.c
```

```
hello.c
 gcc
 cc1
 hello.s
 as
 hello.o
 ld
 a.out
```
C functions

void foo (int, int); /* This is the function declaration*/

void foo(int a, int b) {
/* This is the function definition */
}

foofile.c
Example C program

hello.c

```c
#include <stdio.h>

void main( ) {
    int i = 15;
    printf(" Hello World %d \n", i);
}
```

```
gcc –o hello hello.c
```
How is ‘other’ code included?

• Include Header files (.h) that contain function declarations - the function interface
• The corresponding .c files contain the actual code

file1.h

```c
#include <stdio.h>
void func1(int a, char *b)
{
    if(a > 0)
    {
        *b = ‘a’ ;
    }
}
int main()
{
    //......
    func1(i, pj);
    printf("\abc");
}
```

file1.c

```c
#include <stdio.h>
void func1(int a, char *b)
{
    if(a > 0)
    {
        *b = ‘a’ ;
    }
}
int main()
{
    //......
    func1(i, pj);
    printf("\abc");
}
```
What happens when we compile hello.c as follows?

```c
#include <stdio.h>

int main()
{
    printf("Hello World \n");
    return 0;
}

void print_hello()
{
    printf("Hello World \n");
}
```

- A. An executable called “a.out” is generated
- B. An executable called “hello” is generated
- C. Compiler error
- D. “Hello world” is printed to standard output
Header Guards

file2.h

```c
#ifndef FILE2_H
#define FILE2_H

void f1(int, char *);
int f2(char *, char *);
#endif
```

file1.h

```c
#include "file2.h"

void func1(int, char *);
int func2(char *, char *);
```

file1.c

```c
#include <stdio.h>

void func1(int a, char *b)
{
    if(a > 0)
    {
        *b = 'a';
    }
}

int main()
{
    ...... 
    func1();
    printf("\abc");
}
```