

The slide features a light gray vertical bar on the left. The main content area has a blue header section and a dark gray footer section.

Stack Frames

CS2263 – Systems Software Development

1

The slide features a blue vertical bar on the left and a light gray vertical bar on the right.

References

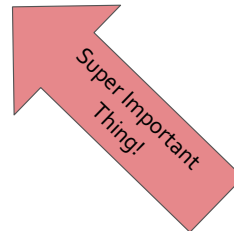
Lu, Yung-Hsiang. 2015. CRC Press. New York. Pp 9-27 (Chapter 2)

2

Learning Outcomes

At the conclusion of this lecture students should be able to:

- List the various forms of memory available in a process
- Explain the process and details of creating and destroying stack frames



3

Addresses and Values

```
int a = 5;
double b = 6.7;
char z = 'c';
```

Symbol	Address (Base ₁₆)	Value
a	0x997	5
b	0x998	6.7
z	0x999	'c'

4

Don't Forget!

Memory Regions

Memory within/to a process

1. Text memory (executable code)
2. Data memory (globals, literal values in the executable code)
3. Heap memory (run-time values)
4. Stack memory (local variable memory)

5

Functions and Function Calls

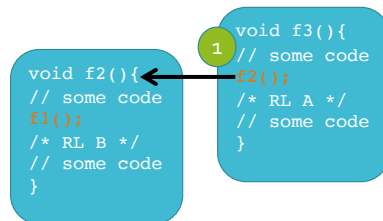
```
1 void f3(){  
  // some code  
  f2();  
  /* RL A */  
  // some code  
}
```

F₃()

Stack

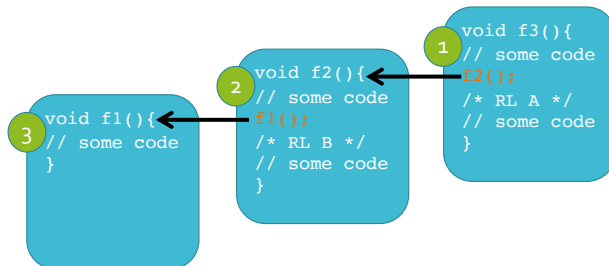
6

Functions and Function Calls



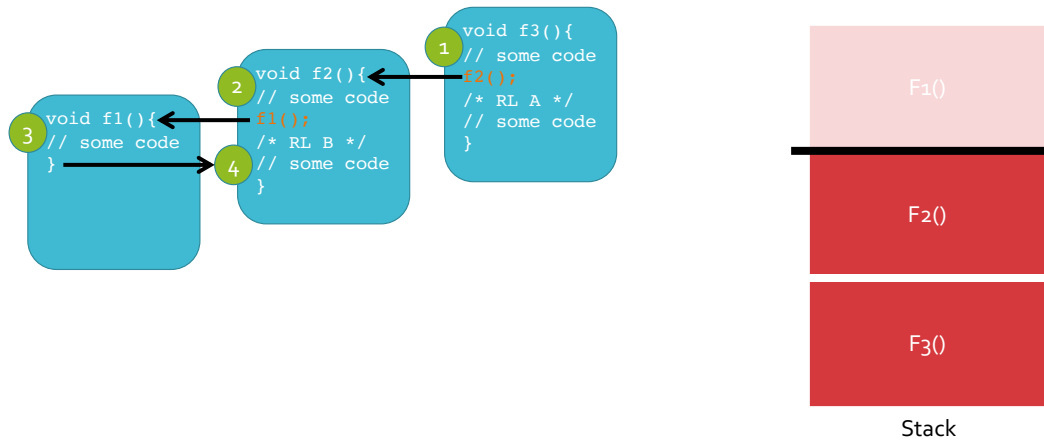
7

Functions and Function Calls



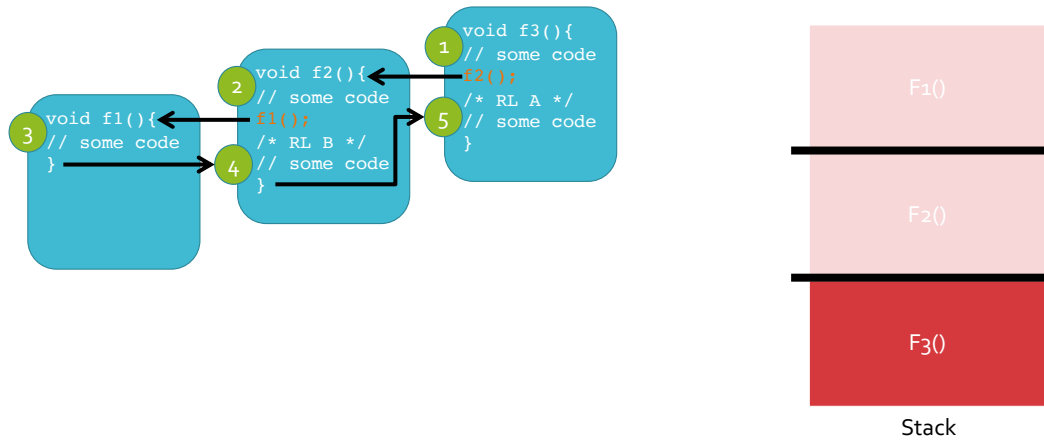
8

Functions and Function Calls



9

Functions and Function Calls



10

Stack memory

- Aka *Call Stack* stores function calls in reverse order
 - Function call pushes the return location (instruction) on the call stack
 - If same function called from multiple lines, each call has a corresponding return location
 - When a function finishes (return or end of function), program continues from line number stored at top of call stack. Top of call stack is then popped.

11

Stack Frame

```
void f3(int a, char b, double c){
    ...
}
void f2(){
    f3(5, 'm', 3.7);
    // RL A
}
```

Frame	Symbol	Address (Base16)	Value
f3	c	0xff6	3.7
	b	0xff7	'm'
	a	0xff8	5
	Return Location	0xff9	RL A

12

Stack Frames

```

void f1(int t, int u){
    ...
}
void f2(int a, int b){
    f1(a - b, a + b);
    // RL B
    ...
}
void f3(){
    f2(5, -17);
    // RL A
    ...
}

```

Frame	Symbol	Address (Base ₁₆)	Value
f2	b	0xffd	-17
	a	0xffe	5
	Return Location	0xfff	RL A

13

Stack Frames

```

void f1(int t, int u){
    ...
}
void f2(int a, int b){
    f1(a - b, a + b);
    // RL B
    ...
}
void f3(){
    f2(5, -17);
    //RL A
    ...
}

```

Frame	Symbol	Address (Base ₁₆)	Value
f1	u	0xffa	-12
	t	0xffb	22
	Return Location	0xffc	RL B
f2	b	0xffd	-17
	a	0xffe	5
	Return Location	0xfff	RL A

14

Local Variables

```
void f1(int k, int m, int p){
    int t = k + m;
    int u = m * p;
}
void f2(){
    f1(5, 11, -8);
    // RL A
}
```

Frame	Symbol	Address (Base ₁₆)	Value
f1	u	0xff4	-88
	t	0xff5	16
	p	0xff6	-8
	m	0xff7	11
	k	0xff8	5
	Return Location	0xff9	RL A

15

Value Address

```
int f1(int k, int m){
    return (k + m);
}
void f2(){
    int u;
    u = f1(11, -8);
    // RL A
}
```

Frame	Symbol	Address (Base ₁₆)	Value
f1	m	0xfdf	-8
	k	0xfe0	11
	Value Address	0xfe1	0xfe3
	Return Location	0xfe2	RL A
f2	u	0xfe3	garbage

16

Arrays I

- Create an array of five ints
`int arr[5];`
- Where is it stored?
 - On the stack
- Prior to C99, array dimensions must be known at compile-time.
- After C99, array dimensions can be variable.
- Array size is not stored – anywhere!

17

Arrays II

```
int arr[5] = {-31, 2, 65, 4, -18};
```

Symbol	Address (Base ₁₆)	Value
<code>arr[0]</code>	0xffb	-31
<code>arr[1]</code>	0xffc	2
<code>arr[2]</code>	0xffd	65
<code>arr[3]</code>	0xffe	4
<code>arr[4]</code>	0xffff	-18

18

Visibility I

```
int f1(int a, int b){
    return (a + b);
}
void f2(){
    int a = 5;
    int b = 6;
    int u;
    u = f1(a + 3, b - 4);
    // RL A
}
```

Frame	Symbol	Address (Base ₁₆)	Value
f1	b	0xff9	2
	a	0xffa	8
	Value Address	0xffb	0xffd
	Return Location	0xffc	RL A
f2	u	0xffd	garbage
	b	0xffe	6
	a	0xfff	5

19

Visibility

```
int f1(int a, int b){
    a = a + 9;
    b = b * 2;
    return (a+b);
}
void f2(){
    int a = 5;
    int b = 6;
    int u;
    u = f1(a + 3, b - 4);
    //RL A
    printf("a = %d, b = %d\n", a, b);
}
```

- What is printed?

20

Viewing the Call Stack I

- The hard way: `memory.c`

```
// Hellooo function prototype
int add(int a, int b);

int main(int argc, char* argv[])
{
    int i;
    int j;
    int sum;
    int iErr;

    /* Report location of variables on the stack */
    printf("Stack memory for main() -----\\n");
    printf("i: %p \\n", &i);
    printf("j: %p \\n", &j);
    printf("sum: %p \\n", &sum);
    printf("iErr: %p \\n", &iErr);
    printf("-----\\n");
```

21

Viewing the Call Stack II

The easy way:

- compile program with `-g` flag:
 - `gcc -o prog prog.c -g`
- Command line debugger (gdb prog) or
- graphical debugger (DDD)

Simple commands:

- *set breakpoint:*
 - `(gdb) b g1` (breaks at beginning of function `g1`)
- *run program:* `(gdb) run`
- *continue after stopping:* `(gdb) continue`
- *print backtrace (call stack):* `(gdb) bt`
- *switch to frame 1:* `(gdb) f 1`
- View data in GUI or with `print` command

22