



Stack Memory Summary (aka “*Don’t forget the stack*”)

CS2263 – Systems Software Development

1

References

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



2

Lecture Learning Outcomes

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

- List and describe the parts that make up a process in memory
- Explain how calling a function affects the stack
- Explain how stack frames and variable scope are related
- List the various forms of memory available in a process
- Explain the process and details of creating and destroying stack frames



3

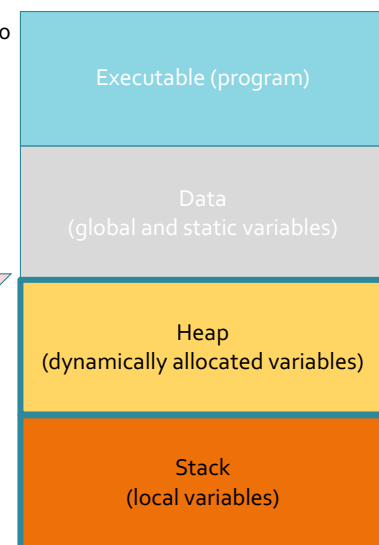
Process Anatomy

- Occupies an assigned region of RAM
- Subdivided into sections:
 - Text (executable)
 - Data
 - Heap
 - Stack

Address: 0x0000

Increasing

Address: 0xffff

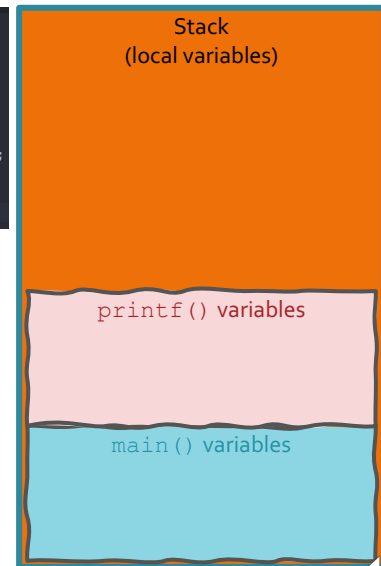


4

Stack Frames

```
// first.c
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char ** argv)
{
    int a = 5;
    int b = 17;
    printf("main: a = %d, b = %d, argc = %d\n", a, b, argc);
    return EXIT_SUCCESS;
}
```

- The OS allocates the process memory, establishes the process memory regions, loads the executable, transfers control
- `main()` begins
 - Create a stack frame!
- `printf()` begins
 - Create a stack frame!
- `printf()` ends
 - Forget the stack frame
- `main()` ends
 - Forget the stack frame
- The OS deallocates the process memory



5

Functions and Function Calls

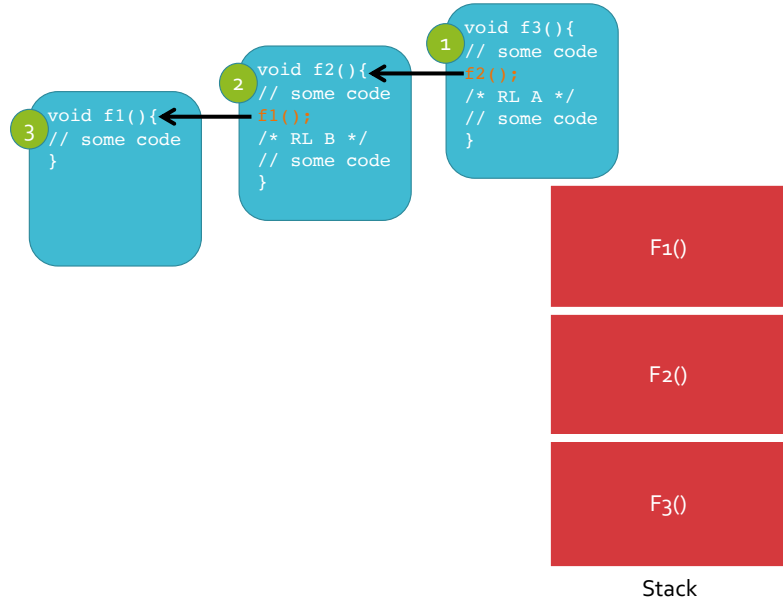
```
void f2(){
    // some code
    f1();
    /* RL B */
    // some code
}
```

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



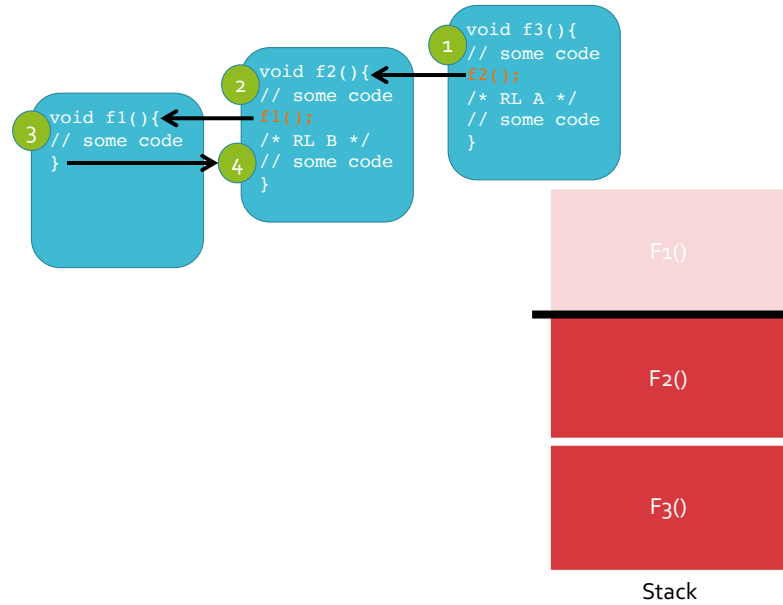
6

Functions and Function Calls



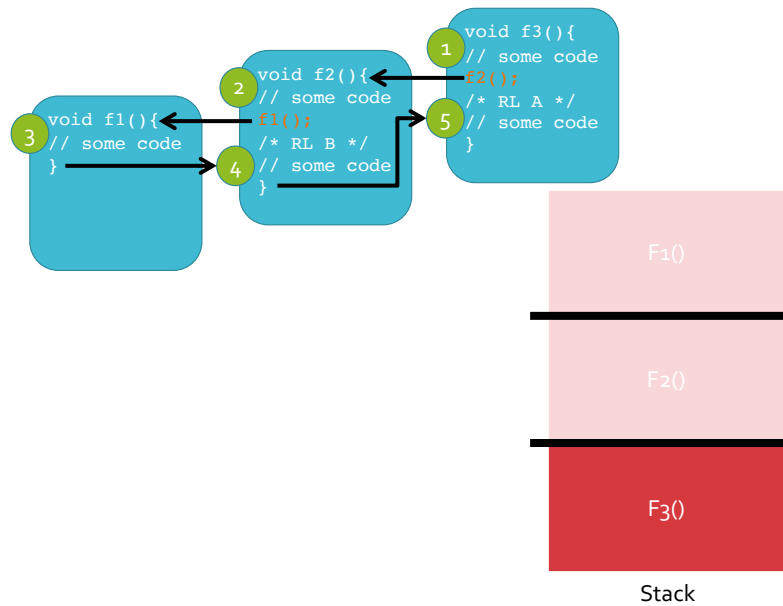
7

Functions and Function Calls



8

Functions and Function Calls



9

Building a Stack Frame I (arrayVals.c)

```

int main(void){
  int iErr;
  int iVal;
  int a[MAXVALS];
  char* prompt = "value: ";
}

```

```

-----
main prompt (char*): 0x7ffc99b27808
main a (int[MAXVALS]): 0x7ffc99b27810
main iVal (int): 0x7ffc99b27824
main iErr (int): 0x7ffc99b27828

```

10

Building a Stack Frame II (arrayVals.c)

What does this tell us?

- Variables are allocated on the stack in the order they are declared
- An int is 4 bytes
- An array starts “up the stack” and increases with memory addressing
 - we knew that, right?
- A char* is 8 bytes (0x810 - 0x808 ► 8_D)
 - an address is 64bits/8bytes

```
-----
main prompt (char*):    0x7ffc99b27808
main a (int[MAXVALS]): 0x7ffc99b27810
main iVal (int):       0x7ffc99b27824
main iErr (int):       0x7ffc99b27828
```



11

Building a Stack Frame III (arrayVals.c)

```
void arrayVals(int iN, int* arr, int val)
{
    for(int i=0; i<iN; i++)
        arr[i]= val;
}
```

```
arrayVals arr (int*):    0x7ffc99b277d0
arrayVals val (int):     0x7ffc99b277d8
arrayVals iN (int):      0x7ffc99b277dc
```

----- 0x808 - (0x7dc +4) ► 40_D bytes: return address (RL), return value, etcetera

```
main prompt (char*):    0x7ffc99b27808
main a (int[MAXVALS]): 0x7ffc99b27810
main iVal (int):       0x7ffc99b27824
main iErr (int):       0x7ffc99b27828
```



12

Building a Stack Frame IV (arrayVals.c)

What does this tell us?

- An array is really only a pointer (address variable)
 - an address is 64bits/8bytes
- Parameters get put on the stack first
 - The compiler does some odd things with the ordering within parameters

```
arrayVals arr (int*): 0x7ffc99b277d0
arrayVals val (int): 0x7ffc99b277d8
arrayVals iN (int): 0x7ffc99b277dc
----
ox808 - (ox7dc +4) ► 40D bytes: return address (RL), return value, etcetera
main prompt (char*): 0x7ffc99b27808
main a (int[MAXVALS]): 0x7ffc99b27810
main iVal (int): 0x7ffc99b27824
main iErr (int): 0x7ffc99b27828
```

