CS 61C

Discussion 2: C Memory Management, RISC-V Intro

John Yang
Summer 2019

Announcements

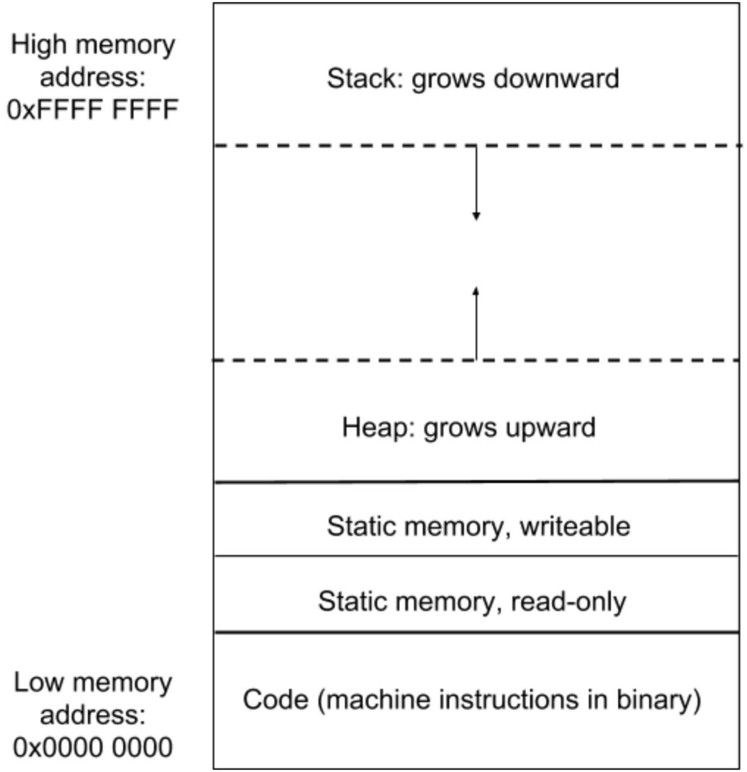
- EPA: https://tinyurl.com/john61c
- Sign ups for small group tutoring now open!
- Project 1 Released, Due July 5th

Today's Goal

- Preview how a program is mapped into memory during execution
- Answer the question of where is a program actually stored + executed in a computer?

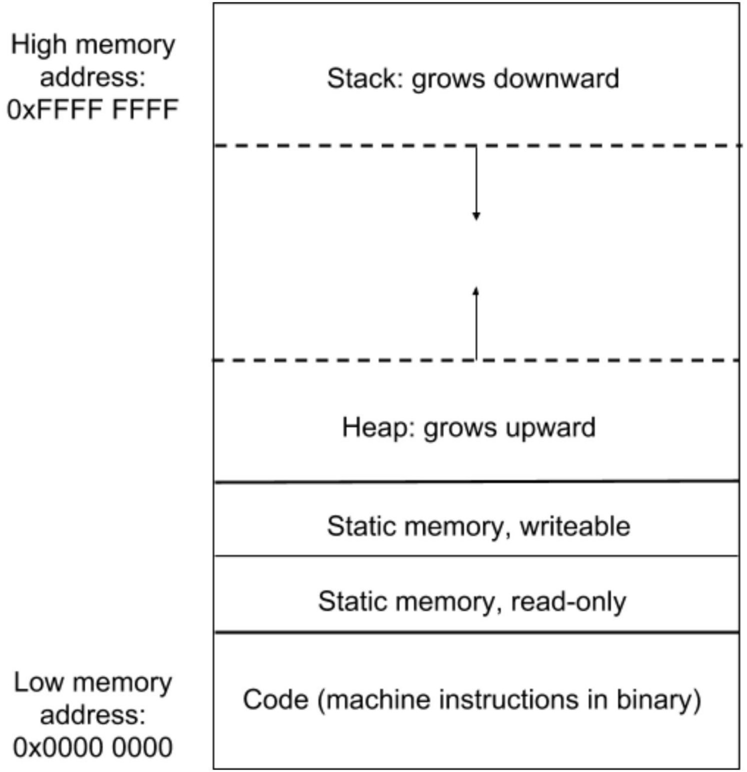
Memory Management

Memory Model



- Stack
  - Local variables
  - Char arrays when instantiated as follows:
    - char str[3] = “hi”; or char str[] = “hi”;
    - char str[2]; str[0] = ‘h’; str[1] = ‘\0’;
  - Garbage after the frame closes (unreliable)
- Heap
  - Dynamically allocated using malloc, calloc, realloc, free
  - Persists (reliable), but could leak memory
- Static
  - Loads when program starts
  - Static & global variables
  - String literals (char *hi = “hello”;)
- Code
  - Loads when program starts
  - Instructions being run

Memory Model



Parts of memory:

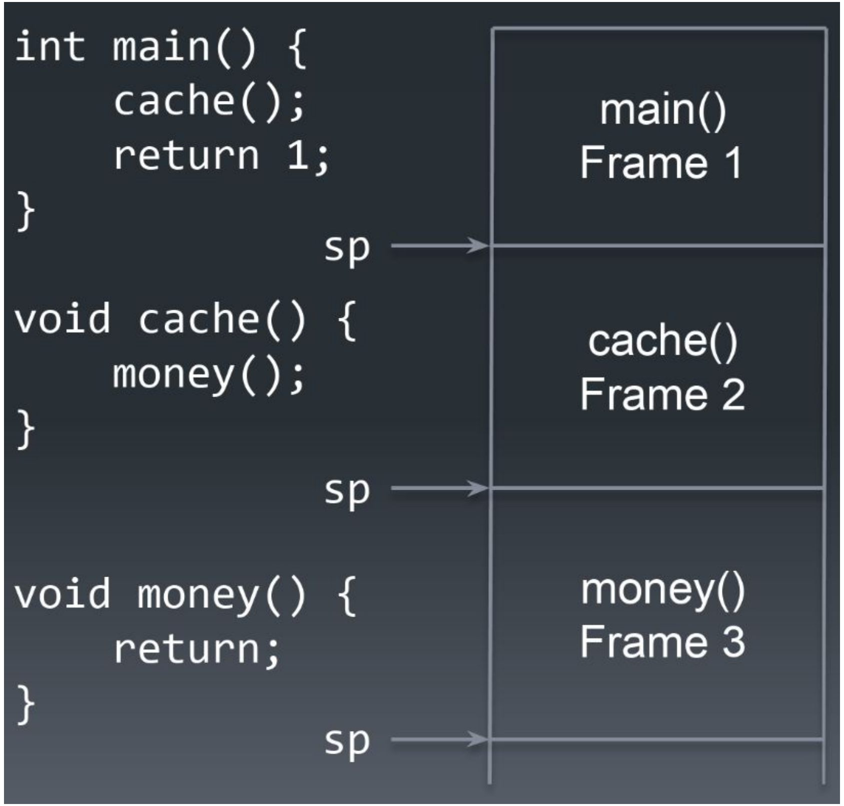
- **Stack:** function local variables, strings allocated as arrays (see next slide)
- **Heap:** dynamically allocated memory (with malloc, calloc, realloc)
- **Static:** global variables, statically allocated strings (see next slide)
- **Code:** machine instructions

C keywords: they might not mean what you think they do! Read the links

- static
- const

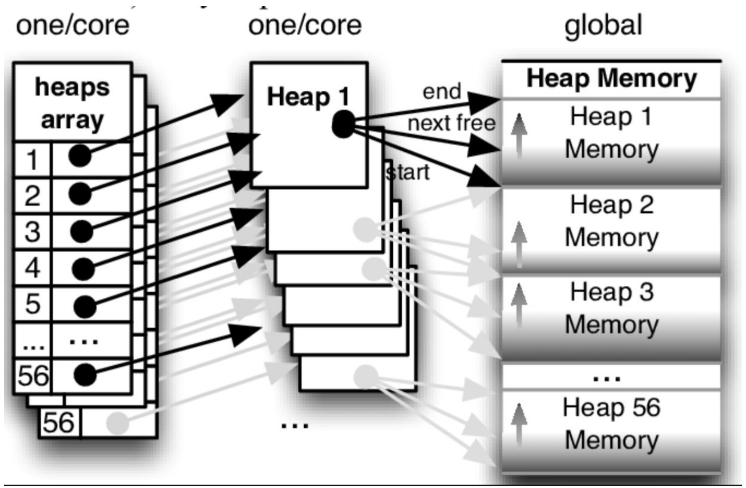
Stack

- What is it?
  - Stores any local variables of function
  - Analogy: everything inside the 61A environment diagrams
- Stack Growth
  - Stack grows *downward*
  - New frame created per function call
  - Variables declared *later* in execution have progressively *smaller* addresses
  - When function finishes executing,
    - Corresponding frame is removed
    - Stack pointer moves back up
  - “Last in First Out”



Heap

- What is it?
  - Stores variables and data we want to persist across functions.
- Heap Growth
  - Heap grows *upward*
  - *No* automatic management, allocate + free variable-sized memory *manually*
  - Variables declared *later* in execution have progressively *larger* addresses



Memory Management Functions

- `void* malloc( size_t size );`
    - Allocates a portion of heap memory corresponding to size!
    - Memory is uninitialized (may contain garbage values)
    - Implementation:
        - Returns a (void *) that can be cast to any other type of pointer
        - Takes in a number "size" and allocates that many bytes
        - Self-Check: What should I pass in for size if I want enough memory to store a single integer?

```c
typedef struct {
    char * name;
    int age;
} person;

person * myperson = malloc(sizeof(person));
myperson->name = "John";
myperson->age = 27;
```

Memory Management Functions

- `void* calloc( size_t num, size_t size );`
  - Same as “malloc”, also initializes all data inside allocated memory to 0
  - num - number of objects
  - size - size of each object
- `void *realloc( void *ptr, size_t new_size );`
  - “Resize” a chunk of memory, saves us trouble of having to manually copy elements
  - Creates a new pointer to a chunk of memory of the specified.
  - Does *NOT* initialize extra space to 0 (unlike “calloc”)
- `void free( void* ptr );`
  - Deallocates heap memory that “ptr” points at.
  - **Note**: Make sure the number of “malloc” + “calloc” calls = number of “free calls”
    - Why? To avoid memory leaks!

RISC-V

RISC-V Context

- *Problem*: A computer cannot directly act on C code. How can we convert the code into something a computer can understand and process?

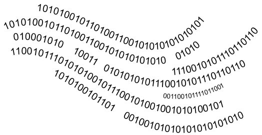
High-level language programs (e.g., C) → Assembly language program (e.g, RISC-V) → Machine Language Program (e.g., RISC-V) → Logic Circuit Description

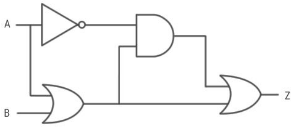
```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int *a;
    a = (int*)malloc(4*sizeof(int));
    printf("Hello, World!\n");
    free(a);
    free(a);

    return 0;
}
```

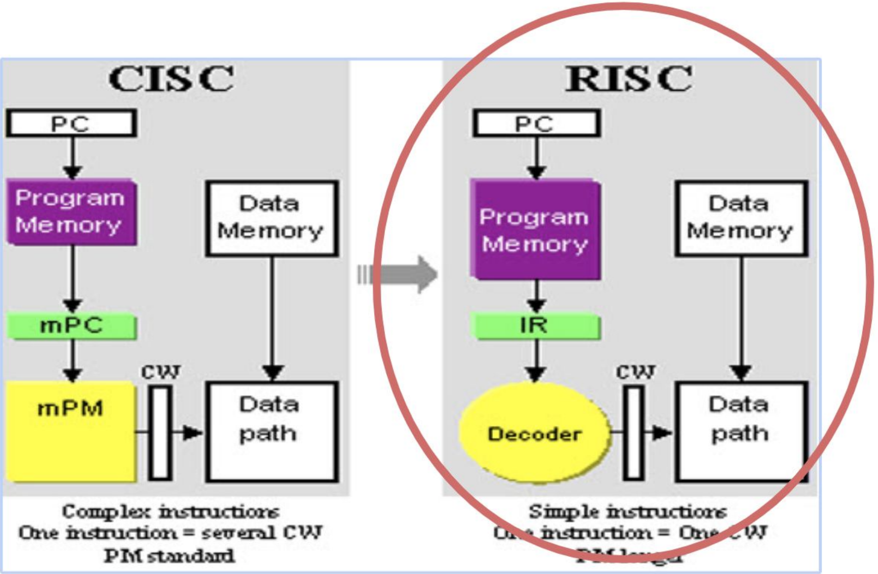
```
ori   x18 x0 16
addi  x20 x18 0
beq   x20, x18, HERE
add   x20 x20, x20
```





RISC-V Introduction

- Fast Facts
  - Reduced Instruction Set Computer
  - Fast, lightweight instruction set architecture
  - Developed in Berkeley in 2010!
- Purpose
  - Breaking down the high-level abstraction of code we’ve written in C
  - Compared to Alternative CISC,
    - Smaller instruction set
    - More efficient run time



Registers

- In machine code, we work with **registers**!
- Registers: physical slots built straight into the computer's CPU
  - 32 Registers in Total
  - Labeled x0 to x31
  - Each register has conventional purpose (refer to diagram on right)
  - Operations with registers are very fast!

REGISTER NAME, USE, CALLING CONVENTION ④

REGISTER	NAME	USE	SAVER
x0	zero	The constant value 0	N.A.
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	--
x4	tp	Thread pointer	--
x5-x7	t0-t2	Temporaries	Caller
x8	s0/fp	Saved register/Frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Function arguments/Return values	Caller
x12-x17	a2-a7	Function arguments	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporaries	Caller
f0-f7	ft0-ft7	FP Temporaries	Caller
f8-f9	fs0-fs1	FP Saved registers	Callee
f10-f11	fa0-fa1	FP Function arguments/Return values	Caller
f12-f17	fa2-fa7	FP Function arguments	Caller
f18-f27	fs2-fs11	FP Saved registers	Callee
f28-f31	ft8-ft11	R[rd] = R[rs1] + R[rs2]	Caller

RISC-V Syntax

- <operation> <destination reg.>, <operand 1 reg.>, <operand 2 reg.>

  ```
  add x1, x2, x3
  ```

  Adds the values in register 2 and 3, store the result in register 1
  In C: a = b + c

- <operation> <destination reg.>, <imm_offset>(<dest/src>)

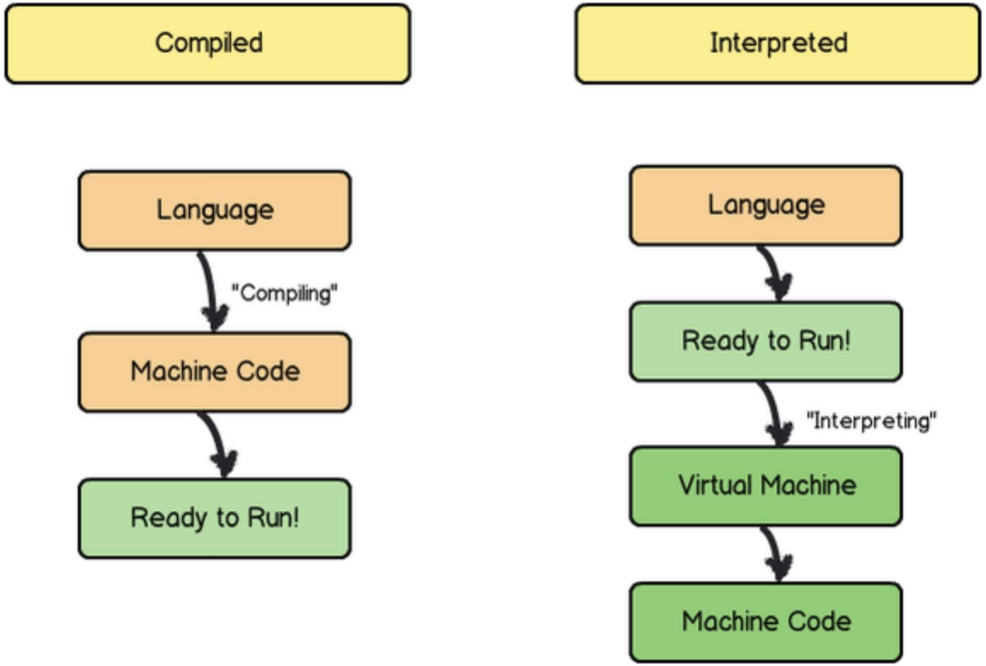
  ```
  sw x8, 0(x9)
  ```

  Stores the value at register x9 to register x8
  In C: a = b[0]

- For a full list of RISC-V Instructions, refer to the Green Sheet

Miscellaneous

Side Note: Compiled vs. Interpreted



- C is *compiled*
- Python is *interpreted*
- Primary reason why people say C runs “faster” than Python