CS 453/698: Software and Systems Security

Module: Background

Lecture: Abstractions in OS, PL, and SE

Meng Xu *(University of Waterloo)*

Spring 2025

Outline

Layered abstraction

Modern computing systems are among the most complex systems ever built.

One of the key engineering techniques that enables the construction of such complex systems is the use of layered abstractions:

Layered abstraction

Modern computing systems are among the most complex systems ever built.

One of the key engineering techniques that enables the construction of such complex systems is the use of layered abstractions:

- the system is designed as a stack of layers, where
- each layer hides implementation details of lower layers.

The hello-world example

```
#include <stdio.h>

int main(void) {
    printf("Hello World");
    return 0;
}
```

The hello-world example

```c
#include <stdio.h>

int main(void) {
    printf("Hello World");
    return 0;
}
```

Compile:

```
cc hello-world.c
```

The hello-world example

```c
#include <stdio.h>

int main(void) {
    printf("Hello World");
    return 0;
}
```

Compile:

`cc hello-world.c`

Execute:

`./a.out`

The hello-world example

```c
#include <stdio.h>

int main(void) {
    printf("Hello World");
    return 0;
}
```

Compile:

`cc hello-world.c`

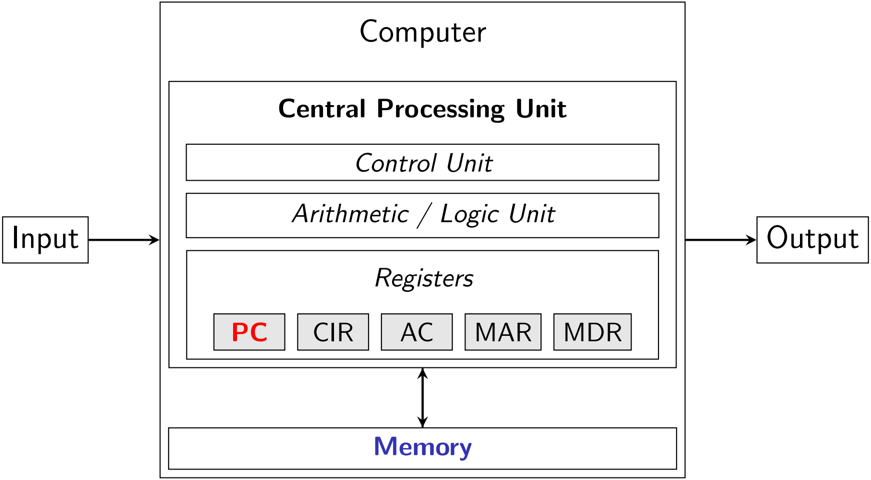
Execute:

`./a.out`

Q: What happens behind the scenes exactly?

Von Neumann architecture

Von Neumann architecture



Program the low-level machine

Suppose there is a CPU instruction called `output <char>`, with opcode `0B <char>`, which sends a single character `<char>` to the output device.

Q: How to display "Hello World" in the output device?

Program the low-level machine

Suppose there is a CPU instruction called
`output <char>`, with opcode `0B <char>`,
which sends a single character `<char>` to the output device.

Q: How to display "Hello World" in the output device?

A: This is a multi-step process:

Step 1: Find a suitable memory location (e.g., address `0x0010`)

Step 2: Put the following bytes into this memory location

```
0B 48 // ASCII code for 'H'
0B 65 // ASCII code for 'e'
...
0B 64 // ASCII code for 'd'
```

Step 3: Put value `0x0010` into the PC register.

A simplified view of compilation and loading

In this overly simplified example, we consider

- getting the bytes `0B 48 0B 65 ... 0B 64` from source code as **compilation**, and

A simplified view of compilation and loading

In this overly simplified example, we consider

- getting the bytes `0B 48 0B 65 ... 0B 64` from source code as **compilation**, and
- the rest as **loading**, including
  1. Find a suitable memory location (e.g., address `0x0010`)
  2. Put the bytes `0B 48 0B 65 ... 0B 64` into this memory location
  3. Put value `0x0010` into the `PC` register.

Reality is more complicated

However, in reality, things are way more complicated. But the operating system, compiler, and software engineering practices abstract the complications away.

Outline

A simple C program

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

int main(void) {
  int pass = 0;
  char buff[8];

  printf("Enter the password: ");
  gets(buff);

  if(strcmp(buff, "warriors")) {
    printf("Wrong password\n");
  } else {
    printf("Correct password\n");
    pass = 1;
  }

  if(pass) {
    printf ("Root privileges granted\n");
  }
  return 0;
}
```

A simple C program

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

int main(void) {
  int pass = 0;
  char buff[8];

  printf("Enter the password: ");
  gets(buff);

  if(strcmp(buff, "warriors")) {
    printf("Wrong password\n");
  } else {
    printf("Correct password\n");
    pass = 1;
  }

  if(pass) {
    printf ("Root privileges granted\n");
  }
  return 0;
}
```

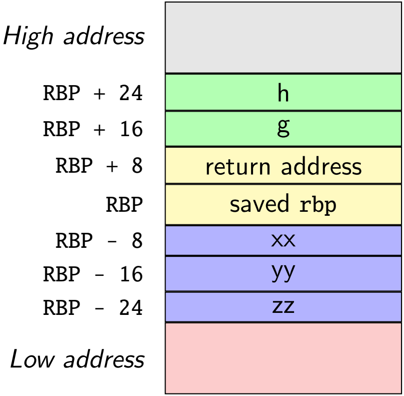
Try with

`gcc -m64 -fno-stack-protector`

And password "golden-hawks"

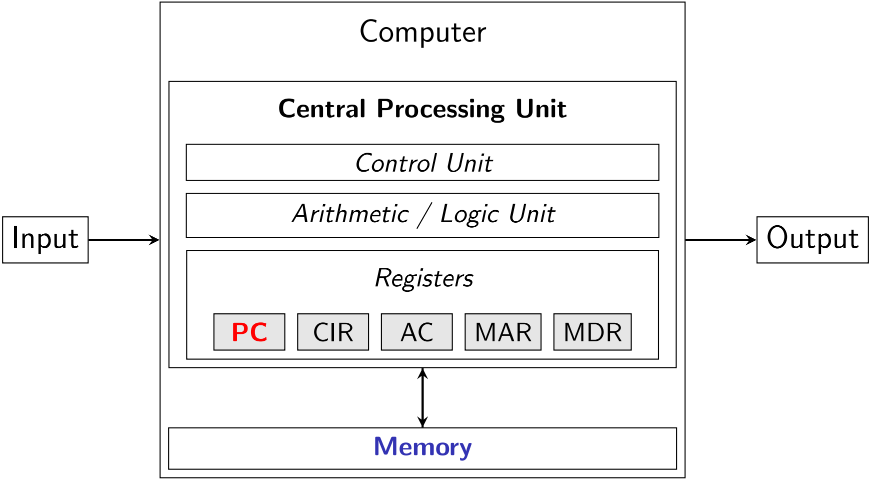
Stack layout (Linux x86-64 convention)

```c
long foo(
    long a, long b, long c,
    long d, long e, long f,
    long g, long h)
{
    long xx = a * b * c;
    long yy = d + e + f;
    long zz = bar(xx, yy, g + h);
    return zz + 20;
}
```



Argument a to `f` passed by registers.

Von Neumann architecture



Implications of the Von Neumann architecture

- Code and data reside in the same memory space and can be addressed in a unified way
  - If you manage to get the PC register to point to a memory address contains your logic, you have effectively hijacked the control flow.

Implications of the Von Neumann architecture

- Code and data reside in the same memory space and can be addressed in a unified way
  - If you manage to get the PC register to point to a memory address contains your logic, you have effectively hijacked the control flow.
- There is only one unified memory. It is the job of the compiler / programming language / runtime to find a way to utilize the memory efficiently.
  - Variables declared in a program (e.g., `int i = 0;`) need to be mapped to an address in the memory, and the mapping logic needs to be (ideally) consistent on the same architecture.

Definition: memory

Q: What is a conventional way of dividing up the “memory”?

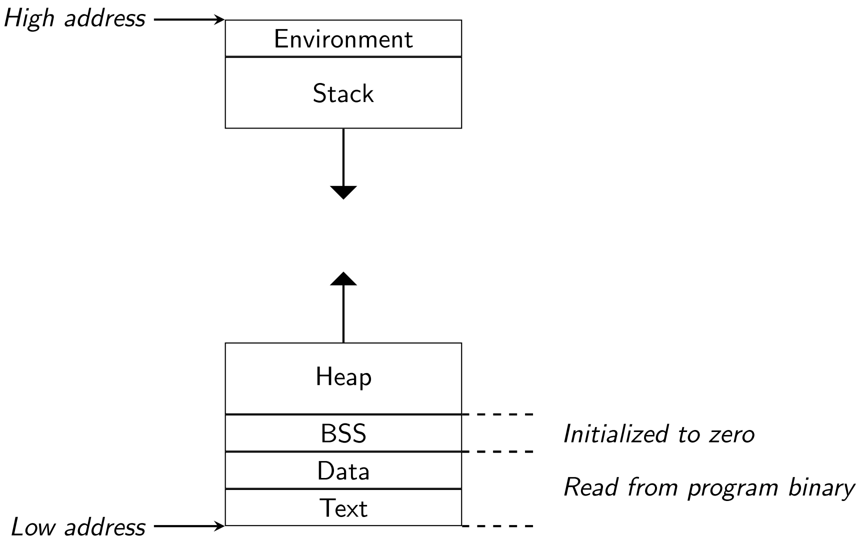
Definition: memory

Q: What is a conventional way of dividing up the "memory"?

A: Four types of memory on a conceptual level:

- Text (where program code is initially loaded to)
- Stack
- Heap
- Global (a.k.a., static)

Memory layout (Linux x86-64 convention)



Example

```c
#include <stdlib.c>

//! where is this variable hosted?
const char *HELLO = "hello";

//! where is this variable hosted?
long counter;

void main() {
    //! where is this variable hosted?
    int val;

    //! where is this variable hosted?
    //! where is its content allocated?
    char *msg = malloc(120);

    //! what is freed here?
    free(msg);

    //! what is freed here (at end of function)?
}

//! what is freed here (at end of execution)?
```

Example (and answers)

```c
#include <stdlib.c>

// this is in the data section
const char *HELLO = "hello";

// this is in the BSS section
long counter;

void main() {
    // this is in the stack memory
    int val;

    // the msg pointer is in the stack memory
    // the msg content is in the heap memory
    char *msg = malloc(120);

    // msg content is explicitly freed here
    free(msg);

    // the val and msg pointer is implicitly freed here
}

// the global memory is only destroyed on program exit
```

What is heap and why do we need it?

In C/C++, the heap is used to manually allocate (and free) new regions of process memory during program execution.

Heap vs stack

```c
typedef struct Response {
  int status;
  char message[40];
} response_t;

response_t *say_hello() {
  response_t* res =
    malloc(sizeof(response_t));
  if (res != NULL) {
    res->status = 200;
    strncpy(res->message, "hello", 6);
  }
  return res;
}
void send_back(response_t *res) {
  // implementation omitted
}
void process() {
  response_t *res = say_hello();
  send_back(res);
  free(res);
}
```

Heap vs stack

```c
typedef struct Response {
  int status;
  char message[40];
} response_t;

response_t *say_hello() {
  response_t* res =
    malloc(sizeof(response_t));
  if (res != NULL) {
    res->status = 200;
    strncpy(res->message, "hello", 6);
  }
  return res;
}
void send_back(response_t *res) {
  // implementation omitted
}
void process() {
  response_t *res = say_hello();
  send_back(res);
  free(res);
}
```

```c
typedef struct Response {
  int status;
  char message[40];
} response_t;

void say_hello(response_t *res) {
  res->status = 200;
  strncpy(res->message, "hello", 6);
}
void send_back(response_t *res) {
  // implementation omitted
}
void process() {
  struct Response res;
  say_hello(&res);
  send_back(&res);
}
```

A stack-based implementation of (roughly) the same functionality

⟨ End ⟩