CS 453/698: Software and Systems Security

Module: Background

Lecture: Abstractions in OS, PL, and SE

Meng Xu *(University of Waterloo)*

Winter 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

```c
#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`

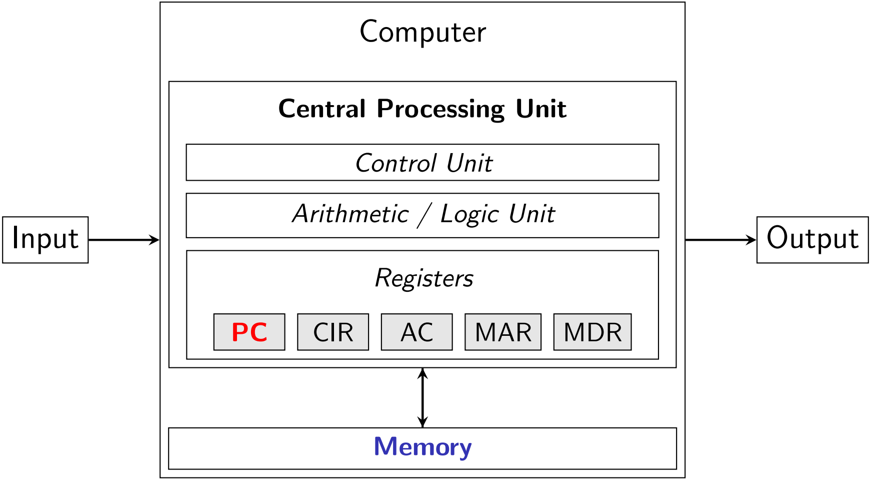
Execute:

`./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

Bus systems

Bus systems

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

Bus systems

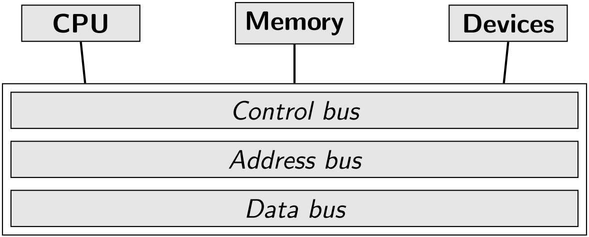
Suppose there is a CPU instruction called `output <char>`*, with opcode* `0B <char>`*, which sends a single character* `<char>` *to the output device.*

There is no such instruction in most modern CPUs. Instead, we have bus systems.

Bus systems

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

There is no such instruction in most modern CPUs. Instead, we have bus systems.



Interacting with the bus systems

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

Interacting with the bus systems

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

To send a character to the I/O device (e.g., a serial device for display), `write <control-address> <char>`.

Interacting with the bus systems

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

To send a character to the I/O device (e.g., a serial device for display), `write <control-address> <char>`.

Q: How do you get the `<control-address>`?

Interacting with the bus systems

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

To send a character to the I/O device (e.g., a serial device for display), `write <control-address> <char>`.

Q: How do you get the `<control-address>`?

Q: How do you know that the `<control-address>` works on all computing systems?

Abstraction: device drivers

Device driver: manages most if not all interactions with a device such that users of this device can access hardware functions without needing to know precise details about the hardware being used.

- Probe the bus system and devices during initialization.
- Proxy requests between application and device.

Abstraction: device drivers

Device driver: manages most if not all interactions with a device such that users of this device can access hardware functions without needing to know precise details about the hardware being used.

- Probe the bus system and devices during initialization.
- Proxy requests between application and device.

To send a character to the I/O device ~~instead of CPU instruction `write <control-address> <char>`~~, we can now use a function call `device_driver_function(WRITE_CHAR_COMMAND, <char>)`.

Alternative: getting the <control-address> directly?

To send a character to the I/O device we can first use a function call to get the <control-address>, `device_driver_function(GET_ADDRESS)`, and then continue with `write <control-address> <char>`.

Alternative: getting the <control-address> directly?

To send a character to the I/O device we can first use a function call to get the <control-address>, `device_driver_function(GET_ADDRESS)`, and then continue with `write <control-address> <char>`.

Q: Will this work?

Alternative: getting the <control-address> directly?

To send a character to the I/O device we can first use a function call to get the <control-address>, `device_driver_function(GET_ADDRESS)`, and then continue with `write <control-address> <char>`.

Q: Will this work?

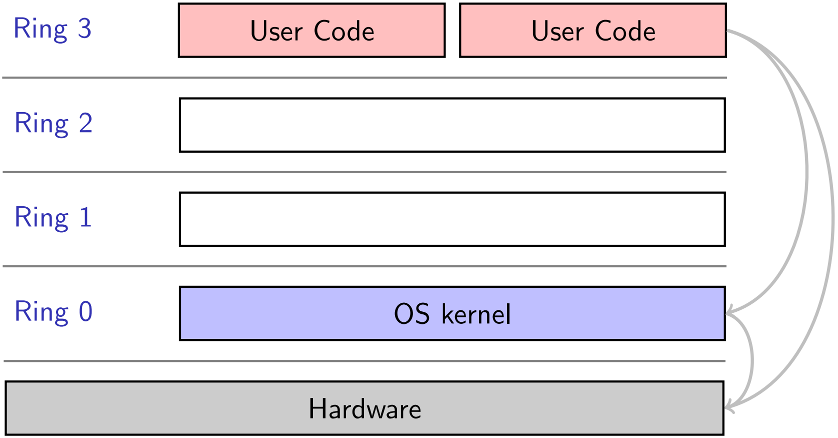
A: In most cases, it won't work, because <control-address> is in a privileged address space and only privileged instructions can read or write to it.

Abstraction: address space and privileged instructions

Now we know that there is certain code in the computing system that runs in a **privileged** mode, typically we call the code kernel.

Abstraction: address space and privileged instructions

Now we know that there is certain code in the computing system that runs in a **privileged** mode, typically we call the code kernel.



Interacting with kernels

In mainstream operating systems (i.e., Linux, Windows, and MacOS), the kernel is in charge of:

- Device drivers
- Networking
- Filesystems
- Virtual memory management
- Threading and scheduling
- Inter-process communication (IPC)
- ... *many more* ...

Interacting with kernels

In mainstream operating systems (i.e., Linux, Windows, and MacOS), the kernel is in charge of:

- Device drivers
- Networking
- Filesystems
- Virtual memory management
- Threading and scheduling
- Inter-process communication (IPC)
- ... *many more* ...

Q: What if I need these functionalities provided in kernel?

Abstraction: system calls (syscalls)

System calls: intermediate most if not all interactions with the kernel such that programs running on top of the kernel can access kernel functions without needing to know precise details on how these functions are implemented.

Abstraction: system calls (syscalls)

System calls: intermediate most if not all interactions with the kernel such that programs running on top of the kernel can access kernel functions without needing to know precise details on how these functions are implemented.

Linux system calls
XNU system calls
Windows system calls (unofficial)

Where is the syscall?

But wait..., I don't see a syscall here?

```c
#include <stdio.h>

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

Outline

Modularization and decoupling

Modularization and decoupling

Modularization: The process of breaking down a complex system into smaller, independent modules. Each module is responsible for a specific function and can be developed, tested, and maintained separately.

Modularization and decoupling

Modularization: The process of breaking down a complex system into smaller, independent modules. Each module is responsible for a specific function and can be developed, tested, and maintained separately.

Decoupling: The practice of minimizing dependencies between modules so that changes to one module don't directly affect others. Components in a decoupled system communicate through well-defined interfaces and/or protocols.

Abstraction: declaration vs definition

Declaration: introduces a symbol and briefly describes its semantics (e.g., input types, return types).

- Sometimes a declaration is also called an "interface" (e.g., "API", "ABI") or a "function signature", etc.

Definition: actually implements/instantiates this symbol.

- Sometimes a definition is also called an "implementation" or the "function body", etc.

Separation of libc and the program

```c
#include <stdio.h>

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

In this example, the program and libc is decoupled and the program only invoke libc functions via well-defined interfaces, i.e., interfaces included in <stdio.h>.

Separation of `libc` and the program

```c
#include <stdio.h>

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

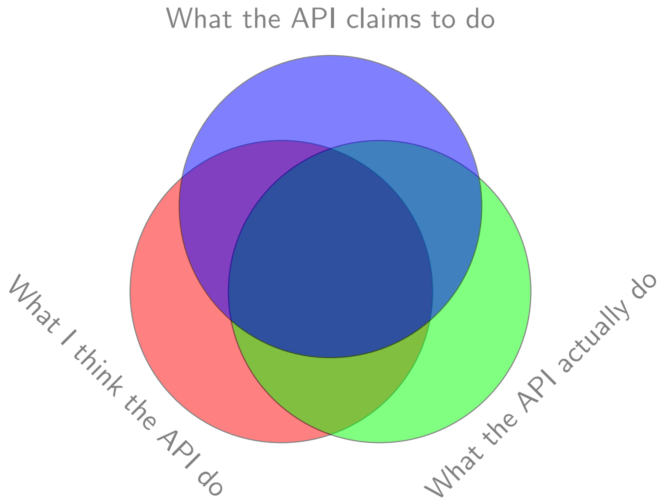
In this example, the program and `libc` is decoupled and the program only invoke `libc` functions via well-defined interfaces, i.e., interfaces included in `<stdio.h>`.

You can actually further replace the `#include <stdio.h>` directive with the actual function signature for `printf`:

```c
int printf(const char *fmt, ...);

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

Issues with this abstraction



Hidden dependency management details

For example, both `gcc` and `clang`

- finds `stdio.h` through a pre-defined search path
- finds `libc` through a pre-defined set of locations

The loader `ld` also searches for `libc` (if needed) through a pre-defined set of locations, which may or may not be the same as the compiler's search locations.

Hidden dependency management details

For example, both `gcc` and `clang`

- finds `stdio.h` through a pre-defined search path
- finds `libc` through a pre-defined set of locations

The loader `ld` also searches for `libc` (if needed) through a pre-defined set of locations, which may or may not be the same as the compiler's search locations.

Abstraction: dependency management

Hiding dependency management details is a common practice in almost all programming language toolchains.

Hidden dependency management details

For example, both `gcc` and `clang`

- finds `stdio.h` through a pre-defined search path
- finds `libc` through a pre-defined set of locations

The loader `ld` also searches for `libc` (if needed) through a pre-defined set of locations, which may or may not be the same as the compiler's search locations.

Abstraction: dependency management

Hiding dependency management details is a common practice in almost all programming language toolchains.

→ Upon loading, the program may or may not be linked to the same dependency it is compiled against!

Outline

A simple C program

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

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

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

  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;
}
```

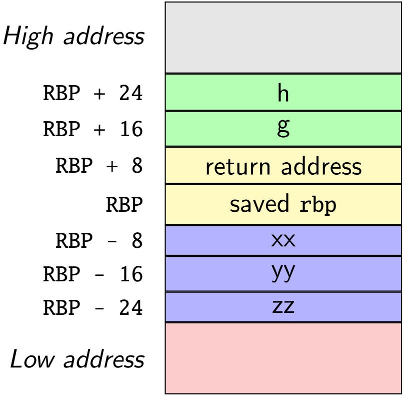
Try with

`gcc -m64 -fno-stack-protector`

And password “golden-hawks”

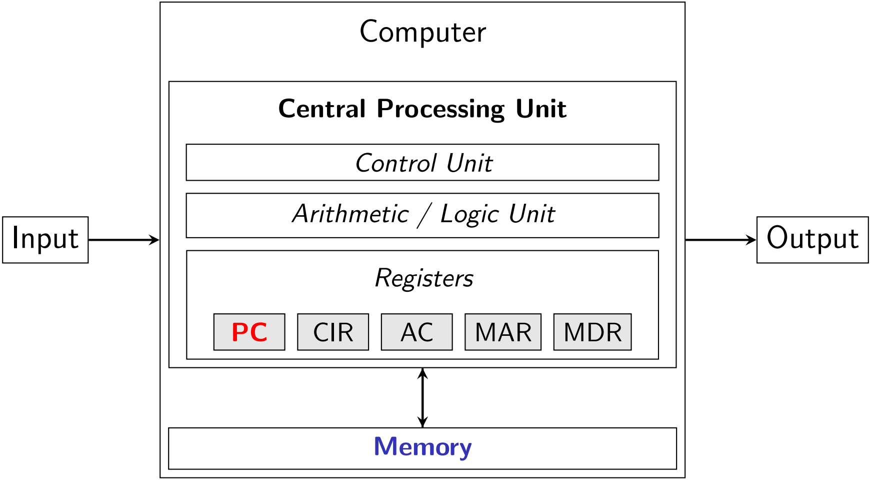
Stack layout (Linux x86-64 convention)

```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"?

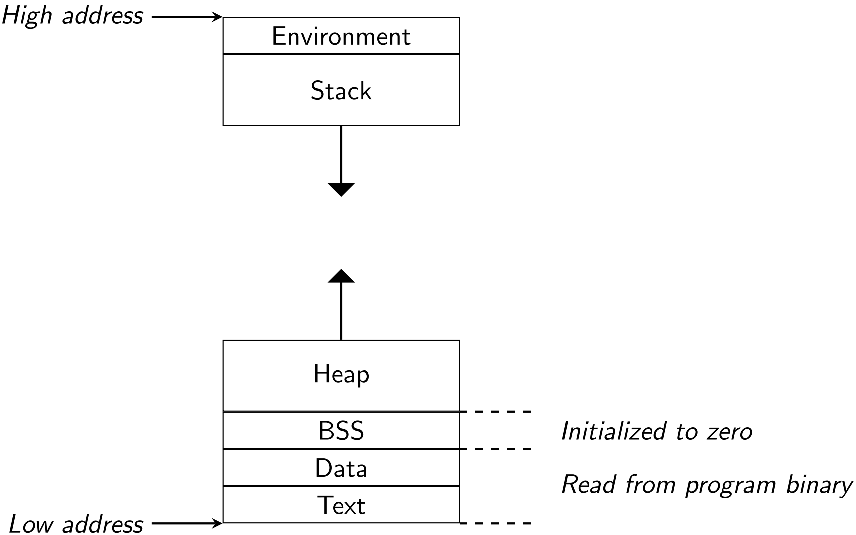
Definition: memory

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

Outline

A (simplified) stack of abstraction layers

Program source code

API declarations of dependencies

Dependencies implementation

Programming language specification

Compiler / interpreter

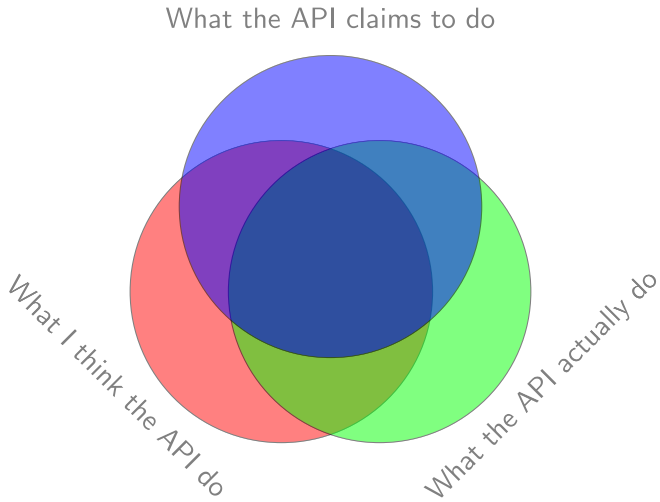
Kernel interfaces

Kernel implementation

Instruction Set Architecture (ISA)

Actual hardwarel

Issues with this abstraction



$\langle$ **End** $\rangle$