F28HS Hardware-Software Interface: Systems Programming

Hans-Wolfgang Loidl

School of Mathematical and Computer Sciences,
Heriot-Watt University, Edinburgh

HERIOT WATT UNIVERSITY

Semester 2 — 2024/25

[0]No proprietary software has been used in producing these slides

---

Outline

1. **Lecture 1: Introduction to Systems Programming**
2. Lecture 2: Systems Programming with the Raspberry Pi
3. Lecture 3: Memory Hierarchy
   - Memory Hierarchy
     - Principles of Caches
4. Lecture 4: Programming external devices
   - Basics of device-level programming
5. Lecture 5: Exceptional Control Flow
6. Lecture 6: Computer Architecture
   - Processor Architectures Overview
     - Pipelining
7. Lecture 7: Code Security: Buffer Overflow Attacks
8. Lecture 8: Interrupt Handling
9. Lecture 9: Miscellaneous Topics
10. Lecture 10: Revision

Lecture 1:
Introduction to Systems Programming

Introduction to Systems Programming

- This course focuses on **how hardware and systems software work together** to perform a task.
- We take a **programmer-oriented view** and focus on software and hardware issues that are relevant for developing **fast, secure, and portable** code.
- **Performance** is a recurring theme in this course.
- You need to grasp a lot of low-level technical issues in this course.
- In doing so, you become a **"power programmer"**.

Why is this important?

You need to understand issues at the hardware/software interface, in order to

- understand and improve performance and resource consumption of your programs, e.g. by developing cache-friendly code;
- avoid progamming pitfalls, e.g. numerical overflows;
- avoid security holes, e.g. buffer overflows;
- understand details of the compilation and linking process.

Questions to be addressed

For each of these issues we will address several common questions on the hardware/software interface:

- **Optimizing program performance:**
  - ▶ Is a switch statement always more efficient than a sequence of if-else statements?
  - ▶ How much overhead is incurred by a function call?
  - ▶ Is a while loop more efficient than a for loop?
  - ▶ Are pointer references more efficient than array indexes?
  - ▶ Why does our loop run so much faster if we sum into a local variable instead of an argument that is passed by reference?
  - ▶ How can a function run faster when we simply rearrange the parentheses in an arithmetic expression?

## Questions to be addressed

- **Understanding link-time errors:**
  - ▶ What does it mean when the linker reports that it cannot resolve a reference?
  - ▶ What is the difference between a static variable and a global variable?
  - ▶ What happens if you define two global variables in different C files with the same name?
  - ▶ What is the difference between a static library and a dynamic library?
  - ▶ Why does it matter what order we list libraries on the command line?
  - ▶ Why do some linker-related errors not appear until run time?
- **Avoiding security holes:**
  - ▶ How can an attacker exploit a buffer overflow vulnerability?

Compilation of hello world
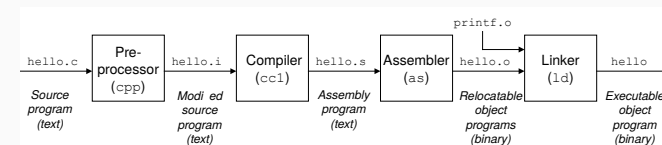
- We have seen individual phases in the compilation chain so far (e.g. assembly)
- Using `gcc` on top level picks the starting point, depending on the file extension, and generates binary code
- You can view the intermediate files of the compilation using the `gcc` flag `-save-temps`
- This is useful in checking, e.g. which assembler code is generated by the compiler
- We will be using `-D` flags to control the behaviour of the pre-processor on the front end

The Shell

Your window to the system is the **shell**, which is an interpreter for commands issued to the system:

```
host> echo "Hello world"
Hello world
host> ls
...
```

The Linux Introduction in F27PX-Praxis gave you an overview of what you can do in a shell. In this course, we make heavy usage of the shell. Check the later sections in the on-line Linux Introduction, which explain some of the more advanced concepts.

Hardware organisation of a typical system
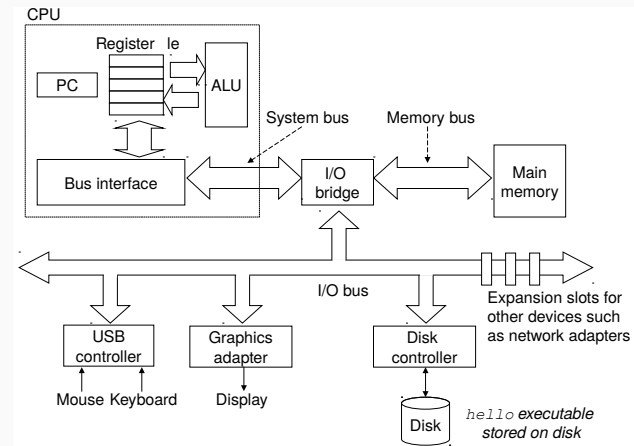
[0]From Bryant and O'Hallaron, Ch 1

Components

The picture on the previous slide, mentions several important concepts:

- **Processor:** the Central Processing Unit (CPU) is the engine that executes instructions; modern CPUs are complicated in order to provide additional performance (multi-core, pipelining, caches etc);
- **Main Memory:** temporary storage for both program and data; arranged as a sequence of dynamic random access memory (DRAM) chips;
- **Buses** transmit information, as byte streams, between components of the hardware; the Universal Serial Bus (USB) is the most common connection for external devices;
- **I/O devices** are in charge of input/output and represent the interface of the hardware to the external world

---

The Hello World Program

```c
#include <stdio.h>

int main()
{
    printf("hello, world\n");
}
```

What happens when we compile and execute this **hello world** program?

Compiling Hello World

When we compile the program by calling

```
gcc -o hello hello.c
```

the compilation chain is executed. Note:

- The source code of Hello World is represented in ASCII characters and stored in a file.
- The contents of the file is just a sequence of bytes
- The **context** determines whether these bytes are interpreted as text or as graphics etc.

When we execute the resulting binary, the next slides show what's happening

```
./hello
```

1. Reading the `hello` program from the keyboard
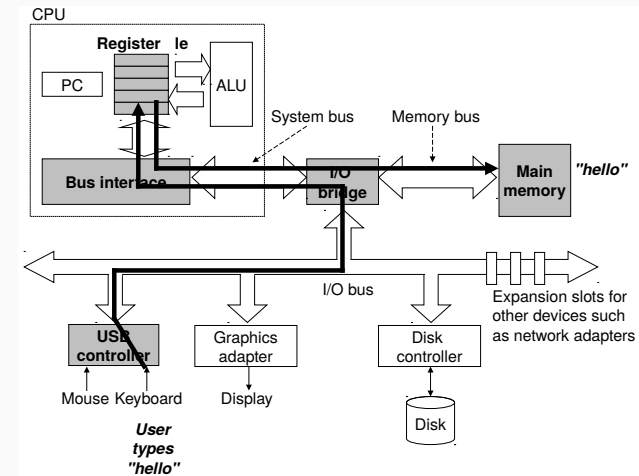
The shell reads `./hello` from the keyboard, stores it in memory; then, initiates to load the executable file from disk to memory.
[0]From Bryant and O'Hallaron, Ch 1

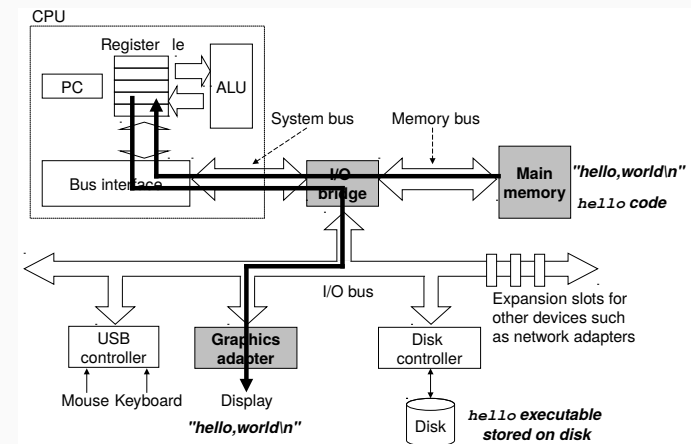## 2. Reading the executable from disk to main memory
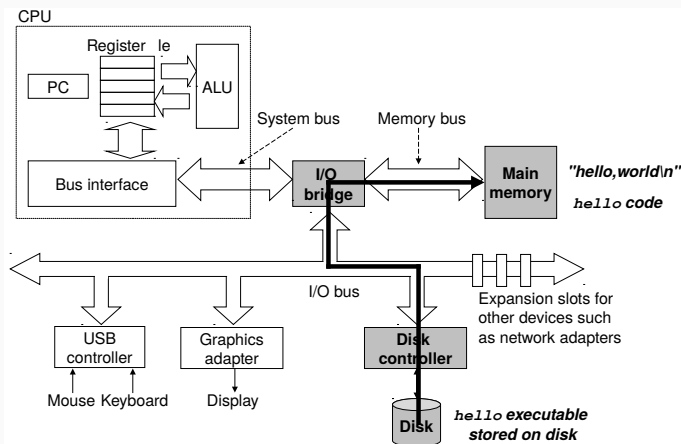
Using direct memory access (DMA) the data travels from disk directly to memory.
[0]From Bryant and O'Hallaron, Ch 1

3. Writing the output string from memory to display

Once the code and data in the hello object file are loaded into memory, the processor begins executing the machine-language instructions in the hello program's main routine.
[0]From Bryant and O'Hallaron, Ch 1

Caches

- Copying data from memory to the CPU is slow compared to performing an arithmetic or logic operation.
- This difference is called **processor-memory gap** and it is increasing with newer generations of processors.
- Copying data from disk is even slower.
- On the other hand, these slower devices provide more capacity.
- To speed up the computation, smaller faster storage devices called **cache memories** are used.
- These cache memories (or just **caches**) serve as temporary staging areas for information that the processor is likely to need in the near future.

Cache memories
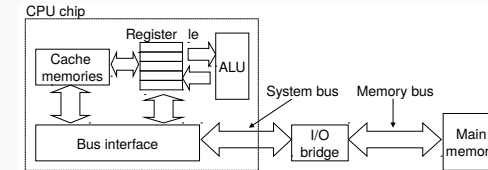
- An **L1 cache** on the processor chip holds tens of thousands of bytes and can be accessed nearly as fast as the register file.
- A larger **L2 cache** with hundreds of thousands to millions of bytes is connected to the processor by a special bus.
- It might take 5 times longer for the process to access the L2 cache than the L1 cache, but this is still 5 to 10 times faster than accessing the main memory.
- The L1 and L2 caches are implemented with a hardware technology known as static random access memory (SRAM). Newer systems even have three levels of cache: L1, L2, and L3.

Caches and Memory Hierarchy
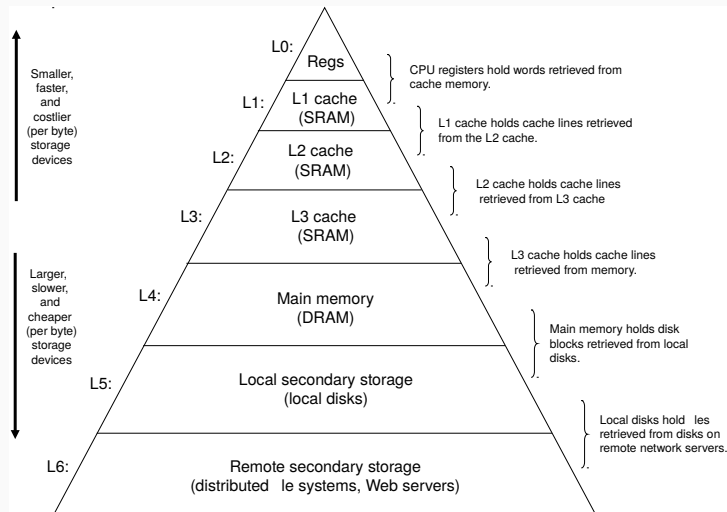
The Role of the Operating System
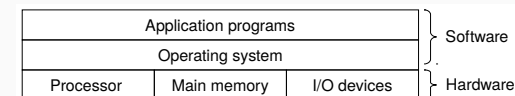
- We can think of the **operating system** as a layer of software interposed between the application program and the hardware.
- All attempts by an application program to manipulate the hardware must go through the operating system.
- This enhances the security of the system, but also generates some overhead.
- In this course we are mainly interested in the **interface between the Software and Hardware layers** in the picture above.

[0]From Bryant and O'Hallaron, Ch 1

Goals of the Operating System

The operating system has two primary purposes:

- to protect the hardware from misuse by runaway applications, and
- to provide applications with simple and uniform mechanisms for manipulating complicated and often wildly different low-level hardware devices.

The operating system achieves both goals via three fundamental abstractions: **processes, virtual memory, and files.**
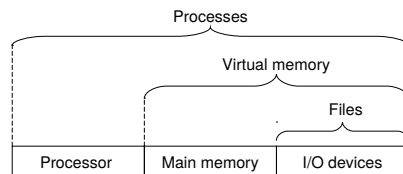
Basic Concepts

In this overview we will cover the following basic concepts:

- Processes
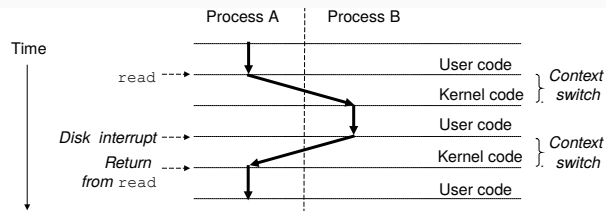- Threads
- Virtual memory
- Files

Processes

- A **process** is the operating system's abstraction for a running program.
- It provides the illusion of having exclusive access to the entire machine.
- Multiple processes can run concurrently.
- The OS mediates the access to the hardware, and prevents processes from overwriting each other's memory.

Concurrency vs Parallelism vs Threads

- **Concurrent execution** means that the instructions of one process are interleaved with the instructions of another process.
- The operating system performs this interleaving with a mechanism known as context switching.
- The context of a process consists of: the program counter (PC), the register file, and the contents of main memory.
- They appear to run simultaneously, but in reality at each point the CPU is executing just one process' operation.
- On **multi-core** systems, where a CPU contains several independent processors, the two processes can be executed **in parallel**, running on separate cores.
- In this case, both processes are genuinely running simultaneously.
- The main goal of parallelism is to make programs run faster.
- A process can itself consist of multiple **threads**.

Example of Context Switching

This example shows the context switching that is happening between the **shell** process and the `hello` process, when running our hello world example.

Different Forms of Concurrency

Concurrency can be exploited at different levels:

- **Thread-level concurrency:** A program explicitly creates several threads with independent control flows. Each thread typically represents a large piece of computation. Shared memory, or message passing can be used to exchange data.
- **Instruction-Level Parallelism:** The components of the CPU can be arranged in a way so that the CPU executes several instructions at the same time. For example, while one instruction is performing an ALU operation, the data for the next instruction can be loaded from memory ("pipelining").
- **Single-Instruction, Multiple-Data (SIMD) Parallelism:** Modern processor architectures provide **vector-operations**, that allow to execute an operation such as addition, over a sequence of values ("vectors"), rather than just two values. Graphic cards make heavy use of this form of parallelism to speed-up graphics operations.

Categorizing different processor configurations
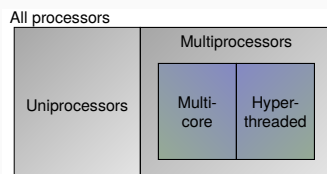
- **Uniprocessors**, with only one CPU, need to context-switch in order to run several processes seemingly at the same time
- **Multiprocessors** replicate certain components of the hardware to genuinely run processes at the same time:
  - ▶ **Muticores** replicate the entire CPU, as several "cores", each of can run a process.
  - ▶ **Hyperthreaded** machines replicate hardware to store the context of several processes to speed-up context-switching.

Virtual Memory

Virtual memory is an abstraction that provides each process with the illusion that it has exclusive use of the main memory. Each process has the same uniform view of memory, which is known as its virtual address space.
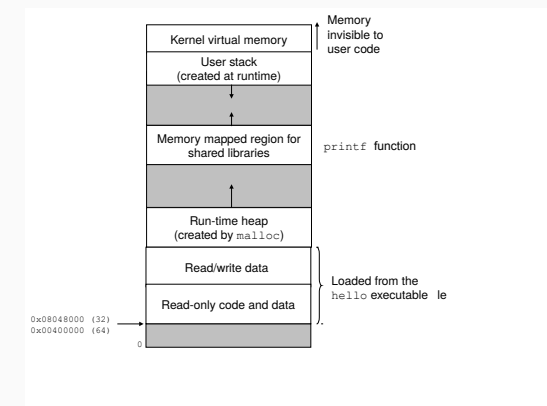
NB: The top region of the address space is reservered for the OS. User processes are not allowed to write into this area!

Virtual Memory

The lower region holds the data for the user.

The user space is separated into several areas, with different roles:

- The **code and data area:** contains the progam code and initialised data, starting at a fixed address. The program code is read only, the data is read/write.
- The **heap** contains dynamically allocated data during the execution of the program. In high-level languages, such as Java, any `new` will allocate in the heap. In low-level languages, such as C, you can use the library function `malloc` to dynamically allocate data in the heap.
- The **shared data** section holds dynamically allocated data, managed by shared libraries.
- The **stack** is a dynamic area at the top of the memory, growing downwards. It is used to hold the local data of functions whenever a function is called during program execution.
- The topmost section of the virtual memory is allocated to **kernel virtual memory**, and only accessible to the OS kernel.

Virtual Memory

- **Virtual memory** gives the illusion of a continuous address space, exceeding main memory, with exclusive access.
- It abstracts over the limitations of physical main memory and allows for several parallel threads to access the same address space.
- We will discuss this aspect in more detail in the Lecture on "Memory Hierarchy".

Aside   The Linux project

In August 1991, a Finnish graduate student named Linus Torvalds modestly announced a new Unix-like operating system kernel:

```
From: torvalds@klaava.Helsinki.FI (Linus Benedict Torvalds)
Newsgroups: comp.os.minix
Subject: What would you like to see most in minix?
Summary: small poll for my new operating system
Date: 25 Aug 91 20:57:08 GMT

Hello everybody out there using minix -
I'm doing a (free) operating system (just a hobby, won't be big and
professional like gnu) for 386(486) AT clones. This has been brewing
since April, and is starting to get ready. I'd like any feedback on
things people like/dislike in minix, as my OS resembles it somewhat
(same physical layout of the file-system (due to practical reasons)
among other things).

I've currently ported bash(1.08) and gcc(1.40), and things seem to work.
This implies that I'll get something practical within a few months, and
I'd like to know what features most people would want. Any suggestions
are welcome, but I won't promise I'll implement them :-)

Linus (torvalds@kruuna.helsinki.fi)
```

Files

- A **file** is a sequence of bytes.
- A file can be used to model any I/O device: disk, keyboard, mouse, network connections etc.
- Files can also be used to store data about the hardware (`/proc/` filesystem), or to control the system, e.g. by writing to files.
- Thus, the concept of a **file** is a very powerful abstraction that can be used for many different purposes.

External Devices

- An important task of the OS/code is to interact with external devices.
- We will see this in detail on the Rpi2
- From the OS point of view, external devices and network connections are files that can be written to and read from.
- When writing to such a special file, the OS sends the data to the corresponding network device
- When reading from such a special file, the OS reads data from the corresponding network device
- This file abstraction simplifies network communication, but is also a source of additional communication overhead.
- Therefore, high performance libraries tend to avoid this "software stack" of implementing file read/write in the OS, but rather directly read to and write from the device (in the same way that we will be using these devices)

A network is another I/O device
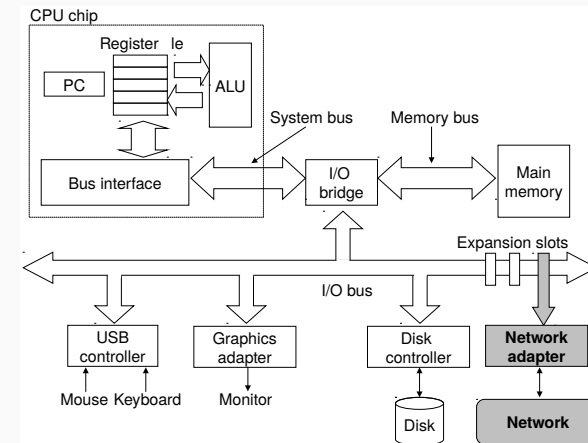
The network can be viewed as just another I/O device.

The Role of Abstraction

- In order to tackle system complexity **abstraction** is a key concept.
- For example, an application program interface (API), abstracts from the internals of an implementation, and only describes its core functionality.
- Java class declaration or C prototypes are programming language features to facilitate abstraction.
- The instruction set architecture abstracts over details of the hardware, so that the same instructions can be used for different realisations of a processor.
- On the level of the operating system, key abstractions are
  - ▶ processes (as abstractions of a running program),
  - ▶ files (as abstractions of I/O), and
  - ▶ virtual memory (as an abstraction of main memory).
- A newer form of abstraction is a **virtual machine**, which abstracts over an entire computer.

Some abstractions provided by a computer system
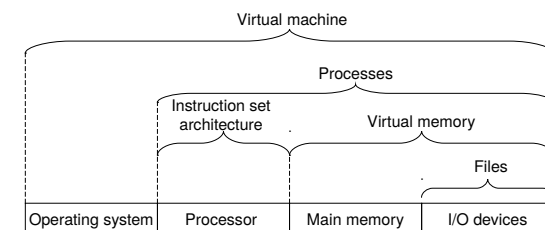
A major theme in computer systems is to provide abstract representations at different levels to hide the complexity of the actual implementations.

Reading List: Systems Programming

📕 David A. Patterson, John L. Hennessy. *"Computer Organization and Design: The Hardware/Software Interface"*, **ARM edition**, Morgan Kaufmann, Apr 2016. ISBN-13: 978-0128017333.

📕 Randal E. Bryant, David R. O'Hallaron *"Computer Systems: A Programmers Perspective"*, 3rd edition, Pearson, 7 Oct 2015. ISBN-13: 978-1292101767.

📕 Bruce Smith *"Raspberry Pi Assembly Language: Raspbian"*, CreateSpace Independent Publishing Platform; 2 edition, 19 Aug 2013. ISBN-13: 978-1492135289.

Other Online Resources

📕 Gordon Henderson et al *"WiringPi library: GPIO Interface library for the Raspberry Pi"*, https://github.com/WiringPi/WiringPi

📕 Valvers *"Bare Metal Programming in C"*, http://www.valvers.com/open-software/raspberry-pi/step01-bare-metal-programming-in-cpt1/

📕 Alex Chadwick, Univ of Cambridge *"Baking Pi"*, https://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os