Operating Systems and Concurrency

Introduction 2

COMP2007

Dan Marsden
(Geert De Maere)
{Geert.DeMaere,Dan.Marsden}@Nottingham.ac.uk

University Of Nottingham
United Kingdom

2023

Recap

Last Lecture

1. The **code** we write heavily **uses operating system functionality**
2. The **operating systems provides abstractions** (e.g., it hides hardware details) and **manages resources**
3. **Resources must be carefully managed** in a multi-programming environment

Goals for Today

Overview

- **Hardware** from an operating system's point of view
- **Address spaces**

Quick Lecturer Introduction

Name:	Dan Marsden
Email:	`dan.marsden@nottingham.ac.uk`
Interests:	Theoretician interested in category theory, logic and the foundations of computer science, working in the functional programming lab

Quick Lecturer Introduction

Name:	Dan Marsden
Email:	`dan.marsden@nottingham.ac.uk`
Interests:	Theoretician interested in category theory, logic and the foundations of computer science, working in the functional programming lab
Previous life:	Approximately 12 years working as a professional developer in industry. Co-author of two libraries in the Boost C++ library project `www.boost.org`

Computer Hardware

Computer hardware in a nutshell

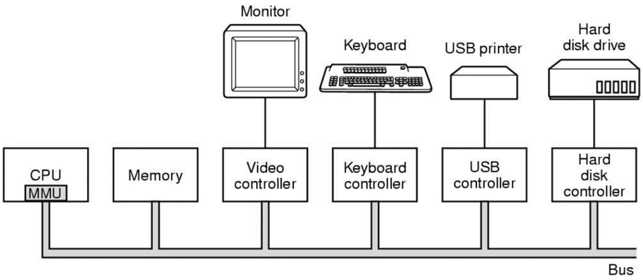


Figure: Simplified computer model (Tanenbaum, 2014)

Computer Hardware

CPU Design: Instruction evaluation

- Naively, CPU's execute a sequence of instructions one at a time
- A CPU's basic cycle consist of **fetch**, **decode**, and **execute** running in a **pipeline**
- A **superscalar** CPU provides **instruction level parallelism**, evaluating multiple instructions in parallel

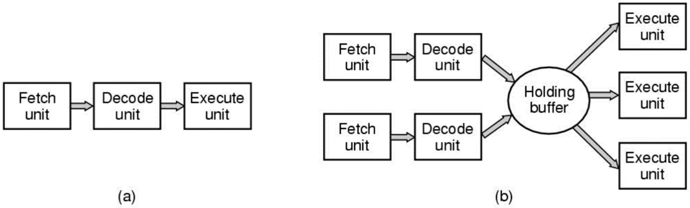


Figure: Processor Architectures (Tanenbaum, 2014) [1]

[1] Computerphile info: `https://www.youtube.com/watch?v=_qvOlL8nhN4`

Computer Hardware

CPU Design: Advanced instruction evaluation

The search for further speed

- **Out-of-order execution** - instructions may not be evaluated in the order they appear in your code
- **Speculative evaluation** - speculative evaluation of instructions
- ...

Computer Hardware

CPU Design: Advanced instruction evaluation

The search for further speed

- **Out-of-order execution** - instructions may not be evaluated in the order they appear in your code
- **Speculative evaluation** - speculative evaluation of instructions
- ...

Take home message

You have to be very careful about the assumptions you make about CPU behaviour![a]

[a]Compiler optimizations and memory architecture further complicate the picture

Computer Hardware

CPU Design: Registers

- Registers are small fast element of memory close to the CPU
- Registers have multiple purposes:
  - **program counter (PC)** - which instruction should be run next
  - **program status word (PSW)** - flags configuring the state of the CPU
  - **general purpose registers** - storing operands for CPU instructions
- The **compiler/programmer** decides what to keep in the registers
- **Registers** are part of the state of a running program

Computer Hardware

CPU Design: Moore's "law"

Moore's law

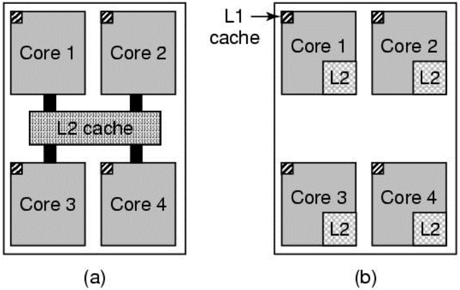
"The number of transistors on an integrated circuit (chip) doubles roughly every two years"

- Closely linked, but **not necessarily related to performance** (performance roughly doubles until 2003)
- Moore's still continuing, but the **"power wall"** slows performance improvements of single core/single processor systems
  - A few cores for multiple "programs" is easy to justify
  - How to use **massively parallel** computers/CPUs/many core machines?

Computer Hardware

CPU Design: Multiple cores and threads

- Modern CPUs contain **multiple cores** and are often support multiple hardware threads within a core
- **Evolution in hardware** has implications on **operating system design**
  - Older operations systems such as Windows XP did not support multi processor architectures
  - Modern operating systems, along with other infrastructure such as compilers, **need to make good use of hardware concurrency** to fully exploit hardware advances



Computer Hardware

CPU Design: Memory Management Unit

```
#include <stdio.h>
#include <unistd.h>

int iVar = 0;

int main() {
  while(iVar++ < 10) {
    printf("Addr:%p; Val:%d\n", &iVar, iVar);
    sleep(1);
  }
}
```

Computer Hardware

CPU Design: Memory Management Unit

```
#include <stdio.h>
#include <unistd.h>

int iVar = 0;

int main() {
  while(iVar++ < 10) {
    printf("Addr:%p; Val:%d\n", &iVar, iVar);
    sleep(1);
  }
}
```

Computer Hardware

CPU Design: Memory Management Unit

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

int iVar = 0;

int main() {
  while(iVar++ < 10) {
    printf("Addr:%p; Val:%d\n", &iVar, iVar);
    sleep(1);
  }
}
```

Computer Hardware

CPU Design: Memory Management Unit

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

int iVar = 0;

int main() {
  while(iVar++ < 10) {
    printf("Addr:%p; Val:%d\n", &iVar, iVar);
    sleep(1);
  }
}
```

Computer Hardware

CPU Design: Memory Management Unit

```
#include <stdio.h>
#include <unistd.h>

int iVar = 0;

int main() {
  while(iVar++ < 10) {
    printf("Addr:%p; Val:%d\n", &iVar, iVar);
    sleep(1);
  }
}
```

Computer Hardware

CPU Design: Memory Management Unit

```
#include <stdio.h>
#include <unistd.h>

int iVar = 0;

int main() {
  while(iVar++ < 10) {
    printf("Addr:%p; Val:%d\n", &iVar, iVar);
    sleep(1);
  }
}
```

Computer Hardware

CPU Design: Memory Management Unit

```
#include <stdio.h>
#include <unistd.h>

int iVar = 0;

int main() {
  while(iVar++ < 10) {
    printf("Addr:%p; Val:%d\n", &iVar, iVar);
    sleep(1);
  }
}
```

- If we run this multiple times, will the **same or different addresses** be displayed for `&iVar`?

Computer Hardware

CPU Design: Memory Management Unit

```
#include <stdio.h>
#include <unistd.h>

int iVar = 0;

int main() {
  while(iVar++ < 10) {
    printf("Addr:%p; Val:%d\n", &iVar, iVar);
    sleep(1);
  }
}
```

- If we run this multiple times, will the **same or different addresses** be displayed for `&iVar`?
- If we run two copies simultaneously, will the **same or different addresses** be displayed for `&iVar`?

Computer Hardware

CPU Design: Memory Management Unit

Instance one output

```
Addr:0x10eac2000; Val:1
```

Instance two output

```
...
```

Computer Hardware

CPU Design: Memory Management Unit

Instance one output

```
Addr:0x10eac2000; Val:1
Addr:0x10eac2000; Val:2
```

Instance two output

```
...
...
```

Computer Hardware

CPU Design: Memory Management Unit

Instance one output

```
Addr:0x10eac2000; Val:1
Addr:0x10eac2000; Val:2
Addr:0x10eac2000; Val:3
```

Instance two output

```
...
...
Addr:0x10eac2000; Val:1
```

Computer Hardware

CPU Design: Memory Management Unit

Instance one output

```
Addr:0x10eac2000; Val:1
Addr:0x10eac2000; Val:2
Addr:0x10eac2000; Val:3
Addr:0x10eac2000; Val:4
```

Instance two output

```
...
...
Addr:0x10eac2000; Val:1
Addr:0x10eac2000; Val:2
```

Computer Hardware

CPU Design: Memory Management Unit

Instance one output

```
Addr:0x10eac2000; Val:1
Addr:0x10eac2000; Val:2
Addr:0x10eac2000; Val:3
Addr:0x10eac2000; Val:4
...
```

Instance two output

```
...
...
Addr:0x10eac2000; Val:1
Addr:0x10eac2000; Val:2
...
```

Computer Hardware

CPU Design: Memory Management Unit

Instance one output

```
Addr:0x10eac2000; Val:1
Addr:0x10eac2000; Val:2
Addr:0x10eac2000; Val:3
Addr:0x10eac2000; Val:4
...
```

Instance two output

```
...
...
Addr:0x10eac2000; Val:1
Addr:0x10eac2000; Val:2
...
```

- Both processes display `0x10eac2000`, the same address.
- The value of `iVar` is different in each run of the program.

Computer Hardware

CPU Design: Memory Management Unit

Instance one output

```
Addr:0x10eac2000; Val:1
Addr:0x10eac2000; Val:2
Addr:0x10eac2000; Val:3
Addr:0x10eac2000; Val:4
...
```

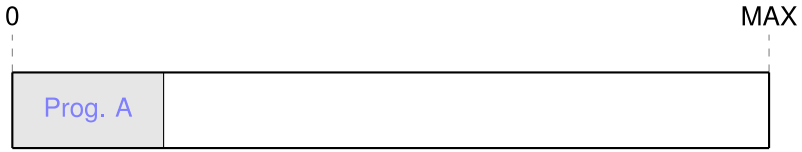
Instance two output

```
...
...
Addr:0x10eac2000; Val:1
Addr:0x10eac2000; Val:2
...
```

- Both processes display `0x10eac2000`, the same address.
- The value of `iVar` is different in each run of the program.

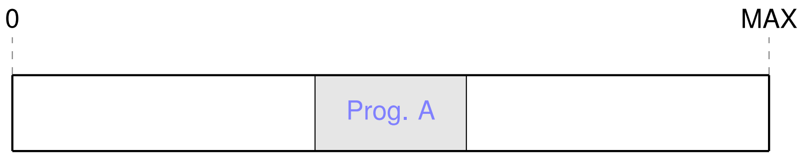
Different behaviour on Mac due to **Address Space Layout Randomization (ASLR)**.

Computer Hardware
CPU Design: Memory Management Unit



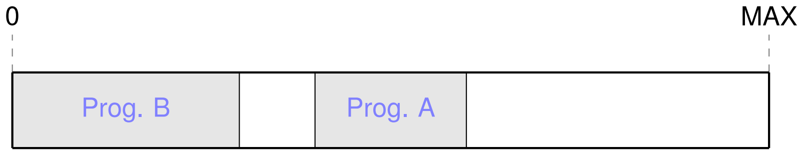
Computer Hardware

CPU Design: Memory Management Unit



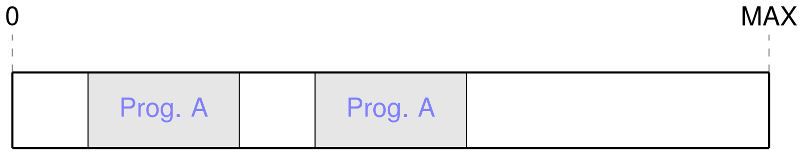
Computer Hardware

CPU Design: Memory Management Unit



Computer Hardware

CPU Design: Memory Management Unit



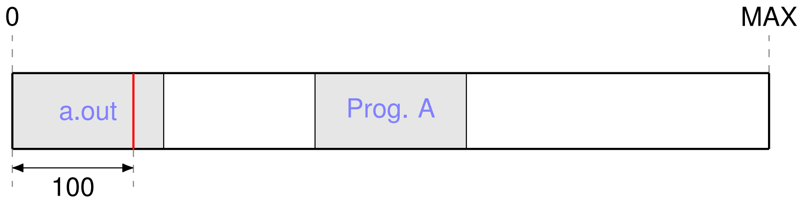
Computer Hardware

Memory Management Unit

- One cannot know **where in memory an executable will run**
    - Might be influenced by what other code is running on the machine
    - Multiple instances of the same program require distinct portions of memory
- When coding: variables have memory associated with them - they **need to have an address**
    - But the exact **physical addresses** cannot be known at compile time
    - The compiler just **assumes** that the program will **start at address 0**
    - If the program does not run at physical address 0 but at, e.g. 1000, just **add the offset** (1000) to every “compiled address”

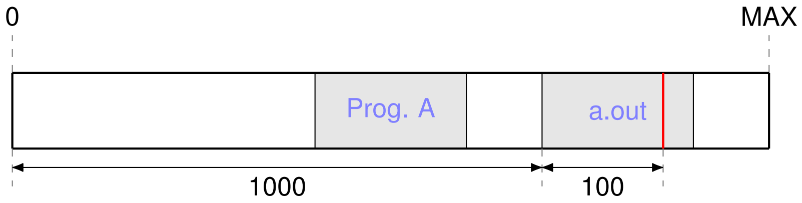
Computer Hardware

CPU Design: Memory Management Unit



Computer Hardware

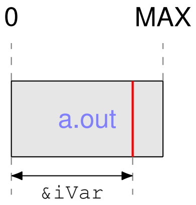
CPU Design: Memory Management Unit



Computer Hardware

Memory Management Unit

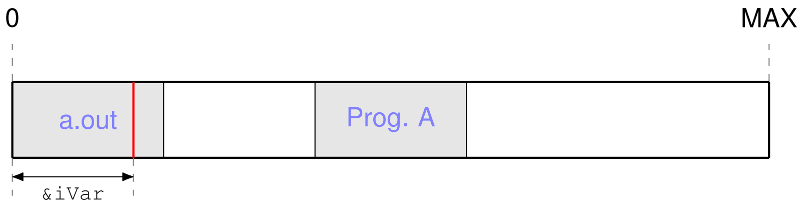
- There are two different address spaces:
    - the **logical address space** seen by the program and used by the compiler
    - the **physical address space** seen by the hardware/OS
- When compiling code, **memory addresses must be assigned to variables** and instructions, the compiler does not know what **memory addresses will be available** in physical memory
- It will just assume that the code will **start running at address 0** when generating the machine code



Computer Hardware

Memory Management Unit

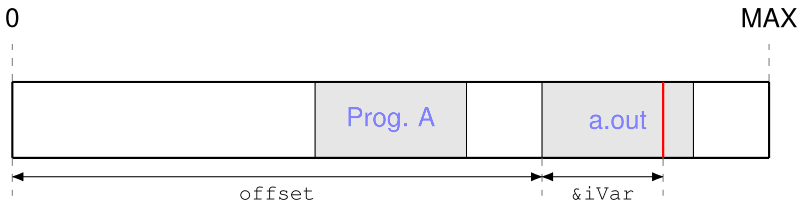
- There are two different address spaces:
  - the **logical address space** seen by the program and used by the compiler
  - the **physical address space** seen by the hardware/OS
- When compiling code, **memory addresses must be assigned to variables** and instructions, the compiler does not know what **memory addresses will be available** in physical memory
- It will just assume that the code will **start running at address 0** when generating the machine code



Computer Hardware

Memory Management Unit

- There are two different address spaces:
    - the **logical address space** seen by the program and used by the compiler
    - the **physical address space** seen by the hardware/OS
- When compiling code, **memory addresses must be assigned to variables** and instructions, the compiler does not know what **memory addresses will be available** in physical memory
- It will just assume that the code will **start running at address 0** when generating the machine code



Computer Hardware

Memory Management Unit

- On some rare occasions, the program **may run at physical address 0**
  - `physical address = logical address + 0`
- On other occasions, it will be running at a **completely different location** in physical memory and an offset is added
  - `physical address = logical address + offset`
- The **memory management unit (MMU)** is **responsible for address translation** (“adding the offset”)
  - Different program instances require **different address translation**
  - The state of the MMU is part of the state of a running program.

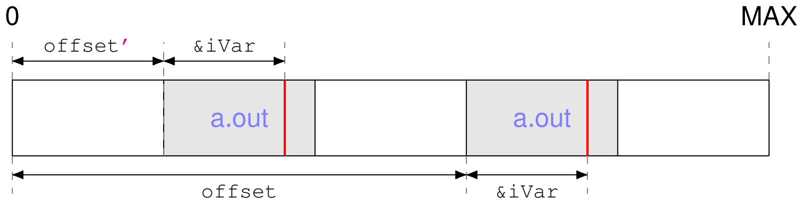
Computer Hardware

CPU Design: Memory Management Unit

- In the case of our example:
  - The **address printed** on the screen is the **logical address**
  - The logical address is translated into **two different physical addresses** using **different offsets**
  - The code

```
printf("Addr:%p; Val:%d\n", &iVar, iVar);
```

was printing the logical address of `iVar`.



Computer Hardware

Memory Management Unit

- Our discussion has omitted many details, address translation may be more complex than applying a single offset uniformly. More detail in memory management lectures later.
- The key point is that the data required by the MMU to translate addresses is part of the state of a running program

Recap

Take-Home Message

- Operating Systems design is heavily coupled to hardware design and capabilities
- Registers contents and data needed by the MMU for address translation are **part of the state of a running program**
- The MMU abstracts the specifics of the actual memory hardware

Test your understanding

1. Can a user program choose to write to a specific location in *physical memory*?
2. Can a user program find out where a variable lives in *physical memory*?
3. Are the answers to these questions a good thing or a bad thing?