

# How to start writing a hobby operating system

Manuel Hohmann

Laboratory of Theoretical Physics - Institute of Physics - University of Tartu  
Center of Excellence "The Dark Side of the Universe"



Hardware programming and operating systems workshop  
22. April 2019

- 1 Preliminary questions and general advice
- 2 Resources and required knowledge
- 3 Necessary and helpful tools
- 4 Some advice on getting started

1 Preliminary questions and general advice

2 Resources and required knowledge

3 Necessary and helpful tools

4 Some advice on getting started

# Reasons *not* to write an OS

I want to write an operating system because. . .

# Reasons *not* to write an OS

I want to write an operating system because. . .

- . . . it will make me rich and / or famous like Bill Gates, Steve Jobs and Linus Torvalds, with millions of people using my OS.  
⇒ That is very, very unlikely. . .

# Reasons *not* to write an OS

I want to write an operating system because. . .

- . . . it will make me rich and / or famous like Bill Gates, Steve Jobs and Linus Torvalds, with millions of people using my OS.  
⇒ That is very, very unlikely. . .
- . . . I don't like the way things look.  
⇒ The “look and feel” is one of the most marginal things that come with an OS. Try writing a UI for another OS instead.

# Reasons *not* to write an OS

I want to write an operating system because. . .

- . . . it will make me rich and / or famous like Bill Gates, Steve Jobs and Linus Torvalds, with millions of people using my OS.  
⇒ That is very, very unlikely. . .
- . . . I don't like the way things look.  
⇒ The “look and feel” is one of the most marginal things that come with an OS. Try writing a UI for another OS instead.
- . . . I want to unify Windows, Linux, Mac OS and run all their programs on one computer at the same time.  
⇒ Getting an OS to run even programs for *one* other OS is a major task, let alone several of them.

# Reasons *not* to write an OS

I want to write an operating system because. . .

- . . . it will make me rich and / or famous like Bill Gates, Steve Jobs and Linus Torvalds, with millions of people using my OS.  
⇒ That is very, very unlikely. . .
- . . . I don't like the way things look.  
⇒ The “look and feel” is one of the most marginal things that come with an OS. Try writing a UI for another OS instead.
- . . . I want to unify Windows, Linux, Mac OS and run all their programs on one computer at the same time.  
⇒ Getting an OS to run even programs for *one* other OS is a major task, let alone several of them.
- . . . I want it to be more secure than any other OS written before.  
⇒ Writing anything secure is highly non-trivial and any subtle bug may become a security risk. Consider joining e.g. Linux kernel development to fix security holes instead.



# Reasons to write an OS

I want to write an operating system because. . .

# Reasons to write an OS

I want to write an operating system because...

- ... my teacher / professor wants me to.

# Reasons to write an OS

I want to write an operating system because...

- ... my teacher / professor wants me to.
- ... I want to learn and understand how computers and operating systems work.

# Reasons to write an OS

I want to write an operating system because. . .

- . . . my teacher / professor wants me to.
- . . . I want to learn and understand how computers and operating systems work.
- . . . it is fun.

# Reasons to write an OS

I want to write an operating system because. . .

- . . . my teacher / professor wants me to.
- . . . I want to learn and understand how computers and operating systems work.
- . . . it is fun.

⇒ Decide what you want to achieve, and try to set a realistic goal what your OS should be able to do.

# Design considerations

- General design considerations:
  - Target a single architecture / platforms or several of them?
  - Which kernel model - monolithic, microkernel?
  - Which task model - monotasking, multitasking, real time?
  - Single processing or multiprocessing?

# Design considerations

- General design considerations:
  - Target a single architecture / platforms or several of them?
  - Which kernel model - monolithic, microkernel?
  - Which task model - monotasking, multitasking, real time?
  - Single processing or multiprocessing?
- Choosing how to implement things:
  - Memory management and protection
  - Scheduling of processes / threads (if multitasking).
  - Synchronization and inter-process communication.
  - Supported executable formats, file systems. . .

# Design considerations

- General design considerations:
  - Target a single architecture / platforms or several of them?
  - Which kernel model - monolithic, microkernel?
  - Which task model - monotasking, multitasking, real time?
  - Single processing or multiprocessing?
- Choosing how to implement things:
  - Memory management and protection
  - Scheduling of processes / threads (if multitasking).
  - Synchronization and inter-process communication.
  - Supported executable formats, file systems. . .
- Hardware specific design choices:
  - Interface between kernel and user programs (system calls).
  - Possible workarounds for CPU bugs.
  - Security measures.
  - Supported hardware.



# Which programming language to use

- C
  - + Easy to write low level code.
  - + Almost no support code / runtime environment required.
  - + Good optimizing C compilers available (fast code).
  - Not much language support for structured programming.

# Which programming language to use

- C
  - + Easy to write low level code.
  - + Almost no support code / runtime environment required.
  - + Good optimizing C compilers available (fast code).
  - Not much language support for structured programming.
- C++
  - + Most of the advantages of C.
  - + Language supports structured programming (object oriented etc.).
  - Requires slightly more support code.
  - May obscure things and accidentally create bloat if used wrongly.
  - Some features must be disabled or implemented by hand.
  - Steep learning curve (especially newer standards).

# Which programming language to use

- C
  - + Easy to write low level code.
  - + Almost no support code / runtime environment required.
  - + Good optimizing C compilers available (fast code).
  - Not much language support for structured programming.
- C++
  - + Most of the advantages of C.
  - + Language supports structured programming (object oriented etc.).
  - Requires slightly more support code.
  - May obscure things and accidentally create bloat if used wrongly.
  - Some features must be disabled or implemented by hand.
  - Steep learning curve (especially newer standards).
- Other languages (Rust, Haskell, D, C#, Objective C, Pascal. . . )
  - + May allow for better structure in high level parts.
  - May require more runtime support for low level code.

# Which programming language to use

- C
  - + Easy to write low level code.
  - + Almost no support code / runtime environment required.
  - + Good optimizing C compilers available (fast code).
  - Not much language support for structured programming.
- C++
  - + Most of the advantages of C.
  - + Language supports structured programming (object oriented etc.).
  - Requires slightly more support code.
  - May obscure things and accidentally create bloat if used wrongly.
  - Some features must be disabled or implemented by hand.
  - Steep learning curve (especially newer standards).
- Other languages (Rust, Haskell, D, C#, Objective C, Pascal. . . )
  - + May allow for better structure in high level parts.
  - May require more runtime support for low level code.
- Assembler
  - + Full control over CPU, optimization. . .
  - Tedious to write everything by hand.
  - Code is not portable to other architectures.

# Outline

1 Preliminary questions and general advice

2 Resources and required knowledge

3 Necessary and helpful tools

4 Some advice on getting started

- Andrew S. Tanenbaum, “Modern Operating Systems”.

- Andrew S. Tanenbaum, “Modern Operating Systems”.
- Andrew S. Tanenbaum, “Operating Systems: Design and Implementation”.

- Andrew S. Tanenbaum, “Modern Operating Systems”.
- Andrew S. Tanenbaum, “Operating Systems: Design and Implementation”.
- Mark Russinovich, David A. Solomon, and Alex Ionescu, “Windows Internals”.



- Andrew S. Tanenbaum, “Modern Operating Systems”.
- Andrew S. Tanenbaum, “Operating Systems: Design and Implementation”.
- Mark Russinovich, David A. Solomon, and Alex Ionescu, “Windows Internals”.
- Scott Andrew Maxwell, “Linux Core Kernel Commentary”.

- Andrew S. Tanenbaum, “Modern Operating Systems”.
- Andrew S. Tanenbaum, “Operating Systems: Design and Implementation”.
- Mark Russinovich, David A. Solomon, and Alex Ionescu, “Windows Internals”.
- Scott Andrew Maxwell, “Linux Core Kernel Commentary”.
- . . . **and many other books.**

# Hardware manuals and standards

- CPU architecture manuals:
  - x86:
    - Intel 64 and IA-32 Architectures Software Developer Manuals
    - AMD64 Architecture Programmer's Manual
  - ARM (AArch32, AArch64): [ARM Information Center](#)
  - [MIPS32 & MIPS64 Instruction Set Architecture manuals](#)

# Hardware manuals and standards

- CPU architecture manuals:
  - x86:
    - Intel 64 and IA-32 Architectures Software Developer Manuals
    - AMD64 Architecture Programmer's Manual
  - ARM (AArch32, AArch64): [ARM Information Center](#)
  - [MIPS32 & MIPS64 Instruction Set Architecture manuals](#)
- Platform documentation:
  - Buses: ISA, PCI, USB...
  - Storage devices.
  - Human interface devices.
  - Video graphics adapters.
  - Network cards.

# Hardware manuals and standards

- CPU architecture manuals:
  - x86:
    - [Intel 64 and IA-32 Architectures Software Developer Manuals](#)
    - [AMD64 Architecture Programmer's Manual](#)
  - ARM (AArch32, AArch64): [ARM Information Center](#)
  - [MIPS32 & MIPS64 Instruction Set Architecture manuals](#)
- Platform documentation:
  - Buses: ISA, PCI, USB...
  - Storage devices.
  - Human interface devices.
  - Video graphics adapters.
  - Network cards.
- Common standards:
  - Low level hardware interfaces: ACPI, UEFI...
  - File systems.
  - Executable formats.

- **Bare Bones**

How to write a simple “Hello world” on bare metal, without any OS or runtime support.

- **Bare Bones**

How to write a simple “Hello world” on bare metal, without any OS or runtime support.

- **Meaty Skeleton**

How to organize and structure your project, create a source tree. . .

- **Bare Bones**

How to write a simple “Hello world” on bare metal, without any OS or runtime support.

- **Meaty Skeleton**

How to organize and structure your project, create a source tree. . .

- Common kernel tutorials for x86 architecture **(be aware of bugs!)**:

- James A. Molloy’s tutorial (bugs)
- BrokenThorn tutorial (bugs)
- Bran’s kernel development tutorial (bugs)



- **Bare Bones**

How to write a simple “Hello world” on bare metal, without any OS or runtime support.

- **Meaty Skeleton**

How to organize and structure your project, create a source tree. . .

- Common kernel tutorials for x86 architecture **(be aware of bugs!)**:

- James A. Molloy’s tutorial (bugs)
- BrokenThorn tutorial (bugs)
- Bran’s kernel development tutorial (bugs)

- **Many, many other tutorials. . .**

# Operating system internals

Take a look at books on operating system internals, developer documentation and open source operating systems, such as...

- Teaching operating systems:
  - xv6
  - Minix

# Operating system internals

Take a look at books on operating system internals, developer documentation and open source operating systems, such as...

- Teaching operating systems:
  - xv6
  - Minix
- Major operating systems:
  - Linux
  - FreeBSD

# Operating system internals

Take a look at books on operating system internals, developer documentation and open source operating systems, such as...

- Teaching operating systems:
  - xv6
  - Minix
- Major operating systems:
  - Linux
  - FreeBSD
- Interesting alternative operating systems / kernels:
  - ReactOS
  - seL4
  - Haiku
  - Fuchsia / Zircon

# Operating system internals

Take a look at books on operating system internals, developer documentation and open source operating systems, such as...

- Teaching operating systems:
  - [xv6](#)
  - [Minix](#)
- Major operating systems:
  - [Linux](#)
  - [FreeBSD](#)
- Interesting alternative operating systems / kernels:
  - [ReactOS](#)
  - [seL4](#)
  - [Haiku](#)
  - [Fuchsia / Zircon](#)
- Successful hobby operating systems:
  - [ToaruOS](#)
  - [Sortix](#)

OSDev.org operating system developer community:



- [Forum](#)
- [Wiki](#)
- [IRC channel #osdev on freenode](#)

# Outline

1 Preliminary questions and general advice

2 Resources and required knowledge

**3 Necessary and helpful tools**

4 Some advice on getting started

- GCC

- + Comes with compilers for C, C++, Ada, Fortran.
- + Can target many different architectures.
- + Actively developed, supports up to date standards.
- + Widely supported standard among OSDev community.
- Must be compiled for every architecture separately.



- GCC

- + Comes with compilers for C, C++, Ada, Fortran.
- + Can target many different architectures.
- + Actively developed, supports up to date standards.
- + Widely supported standard among OSDev community.
  - Must be compiled for every architecture separately.

- Clang

- + Targets many different architectures by command line parameter.
- + Actively developed, supports up to date standard.
  - Less widely used.

- GCC

- + Comes with compilers for C, C++, Ada, Fortran.
- + Can target many different architectures.
- + Actively developed, supports up to date standards.
- + Widely supported standard among OSDev community.
  - Must be compiled for every architecture separately.

- Clang

- + Targets many different architectures by command line parameter.
- + Actively developed, supports up to date standard.
  - Less widely used.

- Microsoft Visual C++

- + Comes with development environment.
  - Designed to produce Windows executables and libraries.
  - Requires some tweaking to be used for OSDev.

- Why do I need a cross compiler?
  - You might want to compile code for a different architecture (e.g., compile ARM code on a x86 computer).
  - Compilers distributed with your operating system (e.g., Linux) are designed / configured to produce programs running under that OS, and getting them to target bare metal is at least tricky.
  - Setting up a cross compiler gives you a standard development environment that is used by other people, who can give advice.

- Why do I need a cross compiler?
  - You might want to compile code for a different architecture (e.g., compile ARM code on a x86 computer).
  - Compilers distributed with your operating system (e.g., Linux) are designed / configured to produce programs running under that OS, and getting them to target bare metal is at least tricky.
  - Setting up a cross compiler gives you a standard development environment that is used by other people, who can give advice.
- How do I get a cross compiler?
  - Clang: Already works as a cross compiler.
  - GCC: Build from source and configure for bare metal target.

- GNU Assembler (GAS):
  - + Targets many different architectures.
  - + Widely used standard tool.
  - + Well integrated with GCC and GNU toolchain.
  - Claimed to be harder to use because of AT&T syntax on x86.

- GNU Assembler (GAS):
  - + Targets many different architectures.
  - + Widely used standard tool.
  - + Well integrated with GCC and GNU toolchain.
    - Claimed to be harder to use because of AT&T syntax on x86.
- Netwide Assembler (NASM):
  - + Claimed to be easier to use because of Intel syntax on x86.
  - + Well developed macro capabilities.
    - Targets only x86 architectures.
    - Less well integrated into common toolchains.

- GNU Assembler (GAS):
  - + Targets many different architectures.
  - + Widely used standard tool.
  - + Well integrated with GCC and GNU toolchain.
    - Claimed to be harder to use because of AT&T syntax on x86.
- Netwide Assembler (NASM):
  - + Claimed to be easier to use because of Intel syntax on x86.
  - + Well developed macro capabilities.
    - Targets only x86 architectures.
    - Less well integrated into common toolchains.
- Flat Assembler (FASM):
  - + New version supports different target architectures.
  - + Rather easy to use.
    - Less well integrated into common toolchains.

- GNU Linker (LD):
  - + Well integrated with GCC and GNU toolchain.
  - + Supports many input and output formats.
  - + Highly configurable output via scripting language.



- GNU Linker (LD):
  - + Well integrated with GCC and GNU toolchain.
  - + Supports many input and output formats.
  - + Highly configurable output via scripting language.
- gold:
  - + Faster than LD.
  - Supports only ELF format.

- GNU Linker (LD):
  - + Well integrated with GCC and GNU toolchain.
  - + Supports many input and output formats.
  - + Highly configurable output via scripting language.
- gold:
  - + Faster than LD.
  - Supports only ELF format.
- LLD:
  - + Well integrated with LLVM toolchain.
  - + Faster than GNU linkers.

- GNU Make

- + Well supported standard tool.
- + Known to work “out of the box” for OS development.
- Can become messy for large projects, multiple architectures. . .

- GNU Make
  - + Well supported standard tool.
  - + Known to work “out of the box” for OS development.
    - Can become messy for large projects, multiple architectures. . .
- Autotools (**Automake** & **Autoconf**)
  - + Creates “configure” script + makefiles for configurable building.
  - + Avoids code duplication.
    - Sometimes considered problematic for bare metal targets.

- **GNU Make**
  - + Well supported standard tool.
  - + Known to work “out of the box” for OS development.
    - Can become messy for large projects, multiple architectures. . .
- **Autotools (Automake & Autoconf)**
  - + Creates “configure” script + makefiles for configurable building.
  - + Avoids code duplication.
    - Sometimes considered problematic for bare metal targets.
- **CMake:**
  - + Popular alternative to Autotools.
    - Designed for applications, not for kernel development.

- **Bochs:**

- + Provides rather accurate emulation.
- + Highly configurable virtual hardware.
- + Highly configurable and verbose log output.
- + Integrated debugger (both CLI and GUI).
- Supports only x86 platforms.

# Emulators and virtual machines

- **Bochs:**

- + Provides rather accurate emulation.
- + Highly configurable virtual hardware.
- + Highly configurable and verbose log output.
- + Integrated debugger (both CLI and GUI).
- Supports only x86 platforms.

- **QEMU:**

- + Emulates many CPU architectures and machines.
- + Highly configurable virtual hardware.
- + Integrated monitor allows inspecting machine state.
- Some targets are not yet fully implemented.

# Emulators and virtual machines

- **Bochs:**

- + Provides rather accurate emulation.
- + Highly configurable virtual hardware.
- + Highly configurable and verbose log output.
- + Integrated debugger (both CLI and GUI).
- Supports only x86 platforms.

- **QEMU:**

- + Emulates many CPU architectures and machines.
- + Highly configurable virtual hardware.
- + Integrated monitor allows inspecting machine state.
- Some targets are not yet fully implemented.

- Virtual machines (**VirtualBox**, **VMware**, **Virtual PC**...):

- + Provide additional virtual hardware to test your kernel.
- Less features to debug or inspect virtual machine state.
- Support only x86 platforms.



- Bochs integrated debugger:
  - + Very powerful debugger with many features.
  - + Allows inspecting and manipulating full machine state.
  - + Scriptable for automated debugging / generating reports.
  - Supports only x86 architectures.

- Bochs integrated debugger:
  - + Very powerful debugger with many features.
  - + Allows inspecting and manipulating full machine state.
  - + Scriptable for automated debugging / generating reports.
  - Supports only x86 architectures.
- GNU Debugger (GDB):
  - + Very powerful debugger with many features.
  - + Supports many different target architectures.
  - + Can connect to a “GDB stub”, which is integrated in...
    - Bochs
    - QEMU
    - your kernel (if you implement it, for remote debugging).
  - Steep learning curve.
  - Cannot access full machine state on all architectures.

- Bochs integrated debugger:
  - + Very powerful debugger with many features.
  - + Allows inspecting and manipulating full machine state.
  - + Scriptable for automated debugging / generating reports.
  - Supports only x86 architectures.
- GNU Debugger (GDB):
  - + Very powerful debugger with many features.
  - + Supports many different target architectures.
  - + Can connect to a “GDB stub”, which is integrated in...
    - Bochs
    - QEMU
    - your kernel (if you implement it, for remote debugging).
  - Steep learning curve.
  - Cannot access full machine state on all architectures.
- VirtualBox debugger:
  - + Comes integrated with VirtualBox.
  - Very limited features.

# Other tools you might need / want

- Tools for working with **disk images**:
  - Emulators use disk images as virtual storage media.
  - Images can be copied on physical storage media.
  - Images provide a convenient way to distribute your OS binary.

# Other tools you might need / want

- Tools for working with **disk images**:
  - Emulators use disk images as virtual storage media.
  - Images can be copied on physical storage media.
  - Images provide a convenient way to distribute your OS binary.
- Boot loader:
  - Loads the kernel (and other files) from disk to memory.
  - Sets up a basic execution environment.
  - Allows installing / booting more than one OS.

# Other tools you might need / want

- Tools for working with **disk images**:
  - Emulators use disk images as virtual storage media.
  - Images can be copied on physical storage media.
  - Images provide a convenient way to distribute your OS binary.
- Boot loader:
  - Loads the kernel (and other files) from disk to memory.
  - Sets up a basic execution environment.
  - Allows installing / booting more than one OS.
- Version control systems:
  - You might accidentally delete / overwrite some code.
  - It might turn out that a change you did made things worse.
  - You might want to fork off branches to work on specific features.
  - You might want to have a look at earlier versions and compare.
  - You might want to maintain a changelog and version history.

# Other tools you might need / want

- Tools for working with **disk images**:
  - Emulators use disk images as virtual storage media.
  - Images can be copied on physical storage media.
  - Images provide a convenient way to distribute your OS binary.
- Boot loader:
  - Loads the kernel (and other files) from disk to memory.
  - Sets up a basic execution environment.
  - Allows installing / booting more than one OS.
- Version control systems:
  - You might accidentally delete / overwrite some code.
  - It might turn out that a change you did made things worse.
  - You might want to fork off branches to work on specific features.
  - You might want to have a look at earlier versions and compare.
  - You might want to maintain a changelog and version history.
- Continuous integration:
  - You might want to test your code in several configurations.
  - Use automatized builds and tests on push to repository.

# Outline

- 1 Preliminary questions and general advice
- 2 Resources and required knowledge
- 3 Necessary and helpful tools
- 4 **Some advice on getting started**



# How to actually get started

- Make sure you have gained the required knowledge.

# How to actually get started

- Make sure you have gained the required knowledge.
- Choose your design - kernel model, target platform(s)...

# How to actually get started

- Make sure you have gained the required knowledge.
- Choose your design - kernel model, target platform(s)...
- Choose a programming language for your implementation.

# How to actually get started

- Make sure you have gained the required knowledge.
- Choose your design - kernel model, target platform(s)...
- Choose a programming language for your implementation.
- Get the tools you need for your platform(s) and language.

# How to actually get started

- Make sure you have gained the required knowledge.
- Choose your design - kernel model, target platform(s) . . .
- Choose a programming language for your implementation.
- Get the tools you need for your platform(s) and language.
- Make sure your tools work, compile some “Hello world” example.

# How to actually get started

- Make sure you have gained the required knowledge.
- Choose your design - kernel model, target platform(s) . . .
- Choose a programming language for your implementation.
- Get the tools you need for your platform(s) and language.
- Make sure your tools work, compile some “Hello world” example.
- Keep the hardware manuals ready at hand.

# How to actually get started

- Make sure you have gained the required knowledge.
- Choose your design - kernel model, target platform(s) . . .
- Choose a programming language for your implementation.
- Get the tools you need for your platform(s) and language.
- Make sure your tools work, compile some “Hello world” example.
- Keep the hardware manuals ready at hand.
- Make a structured plan of your implementation.

# How to actually get started

- Make sure you have gained the required knowledge.
- Choose your design - kernel model, target platform(s) . . .
- Choose a programming language for your implementation.
- Get the tools you need for your platform(s) and language.
- Make sure your tools work, compile some “Hello world” example.
- Keep the hardware manuals ready at hand.
- Make a structured plan of your implementation.
- Create a **structured directory tree** for your source code.



# How to actually get started

- Make sure you have gained the required knowledge.
- Choose your design - kernel model, target platform(s)...
- Choose a programming language for your implementation.
- Get the tools you need for your platform(s) and language.
- Make sure your tools work, compile some “Hello world” example.
- Keep the hardware manuals ready at hand.
- Make a structured plan of your implementation.
- Create a **structured directory tree** for your source code.
- Start coding!

# Typical ingredients for an OS kernel

- Linker script:
  - Determines where your kernel is in memory.
  - Defines sections and their properties (read/write, executable...).

# Typical ingredients for an OS kernel

- Linker script:
  - Determines where your kernel is in memory.
  - Defines sections and their properties (read/write, executable...).
- Assembler stubs:
  - Need to set up environment for high level language (stack...).
  - Entry points for interrupts, syscalls, CPU mode switches.
  - CPU specific system instructions<sup>1</sup>.

---

<sup>1</sup>One may also use inline assembly code here.

# Typical ingredients for an OS kernel

- Linker script:
  - Determines where your kernel is in memory.
  - Defines sections and their properties (read/write, executable...).
- Assembler stubs:
  - Need to set up environment for high level language (stack...).
  - Entry points for interrupts, syscalls, CPU mode switches.
  - CPU specific system instructions<sup>1</sup>.
- High level language (C, C++, ...) sources:
  - Probably the largest part of your code.
  - Possible to port parts of it to other architectures.
  - Reduces code duplication.
  - Easier to maintain than assembly.

---

<sup>1</sup>One may also use inline assembly code here.

# Typical ingredients for an OS kernel

- Linker script:
  - Determines where your kernel is in memory.
  - Defines sections and their properties (read/write, executable...).
- Assembler stubs:
  - Need to set up environment for high level language (stack...).
  - Entry points for interrupts, syscalls, CPU mode switches.
  - CPU specific system instructions<sup>1</sup>.
- High level language (C, C++, ...) sources:
  - Probably the largest part of your code.
  - Possible to port parts of it to other architectures.
  - Reduces code duplication.
  - Easier to maintain than assembly.
- Makefiles / build script:
  - Compilation instructions for your OS.
  - Determine in which order to build, dependencies...

---

<sup>1</sup>One may also use inline assembly code here.

# Which things should I implement next?

- Printing strings and integers (decimal and hexadecimal):  
Essential for any debug output. You know a lot more about your kernel's state and what it does if it can tell you that.

# Which things should I implement next?

- Printing strings and integers (decimal and hexadecimal):  
Essential for any debug output. You know a lot more about your kernel's state and what it does if it can tell you that.
- Interrupt / exception handling:  
Your kernel *will* have bugs that trigger CPU faults / exceptions. Dumping the machine state and possible even cause of the exception is crucial in debugging.

# Which things should I implement next?

- Printing strings and integers (decimal and hexadecimal):  
Essential for any debug output. You know a lot more about your kernel's state and what it does if it can tell you that.
- Interrupt / exception handling:  
Your kernel *will* have bugs that trigger CPU faults / exceptions. Dumping the machine state and possible even cause of the exception is crucial in debugging.
- Memory management:  
Almost anything you do will require memory management. Memory management tasks (in the kernel) can usually be split in three categories:
  - Physical memory management - which pages are free, which are used? Finding free pages, allocating & freeing pages. . .
  - Virtual memory management - how to map physical pages into virtual memory, managing paging structures, address spaces. . .
  - Kernel heap - memory used by the kernel for dynamic allocation / deallocation (`malloc()`, `free()` or `new`, `delete`).