Operating Systems and Concurrency

Introduction 3

COMP2007

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

University Of Nottingham
United Kingdom

2023

Recap

Last Lecture

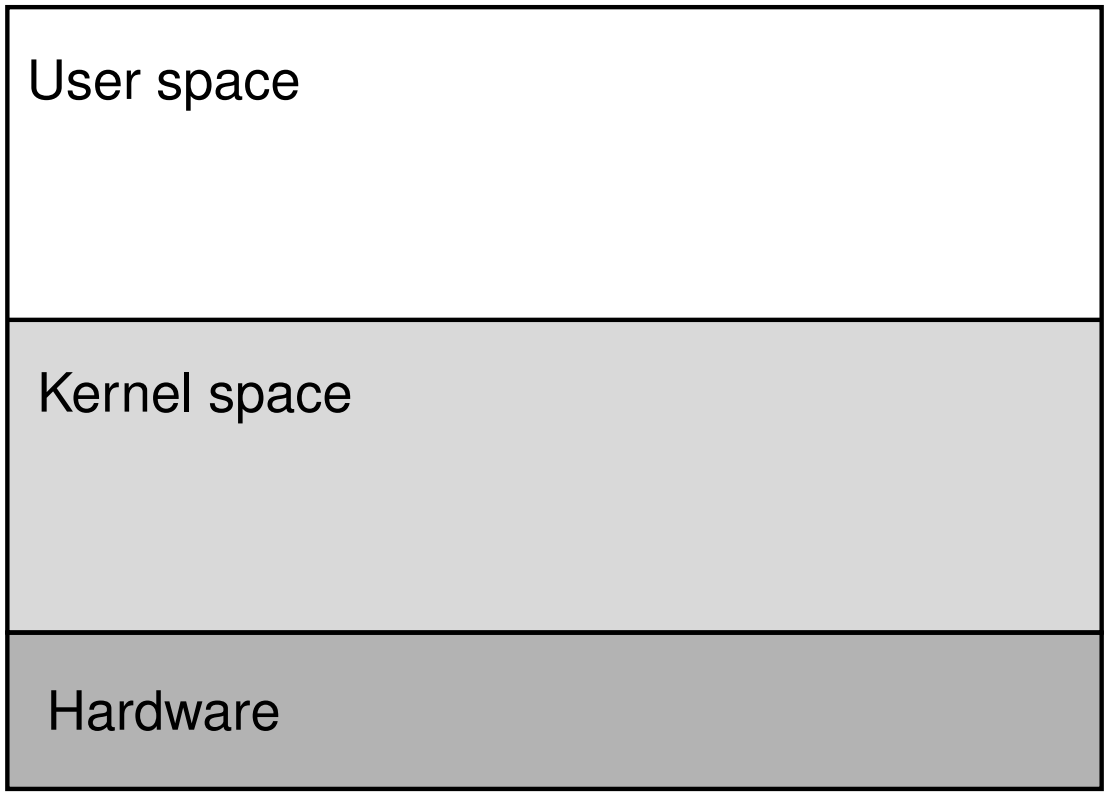
1. Operating system **kernel** design closely linked to hardware capabilities
2. **Registers** provide data consumed by the CPU
3. The **MMU** translates **logical** to **physical addresses** abstracting the details of the memory hardware

Goals for Today

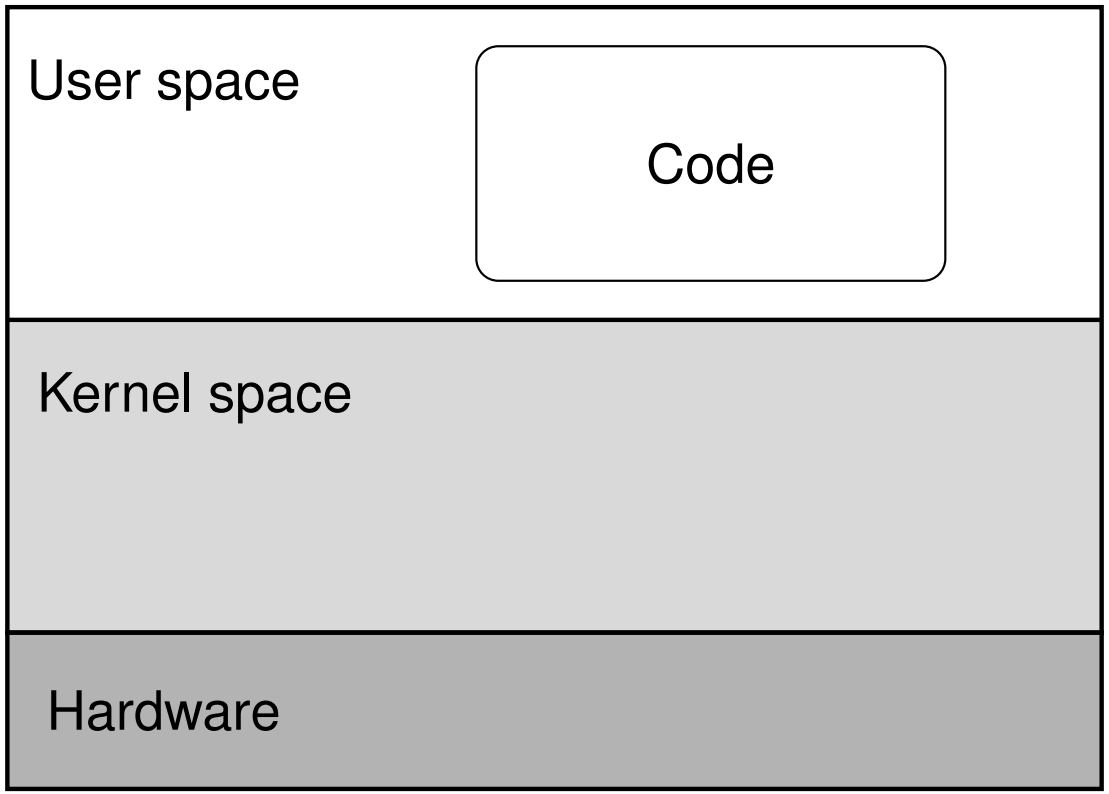
Overview

- **Kernel and user space**
- **Interrupts**
- **System calls**
- **The C programming language** - an operating systems perspective.

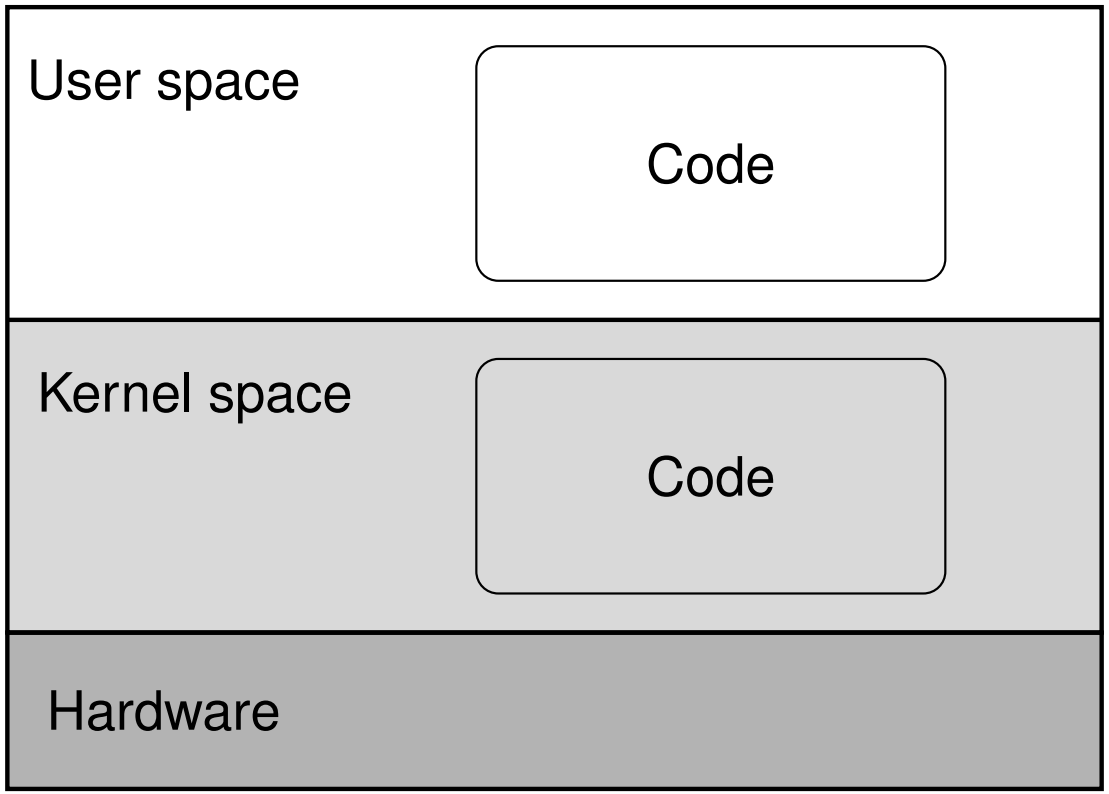
Kernel and user space



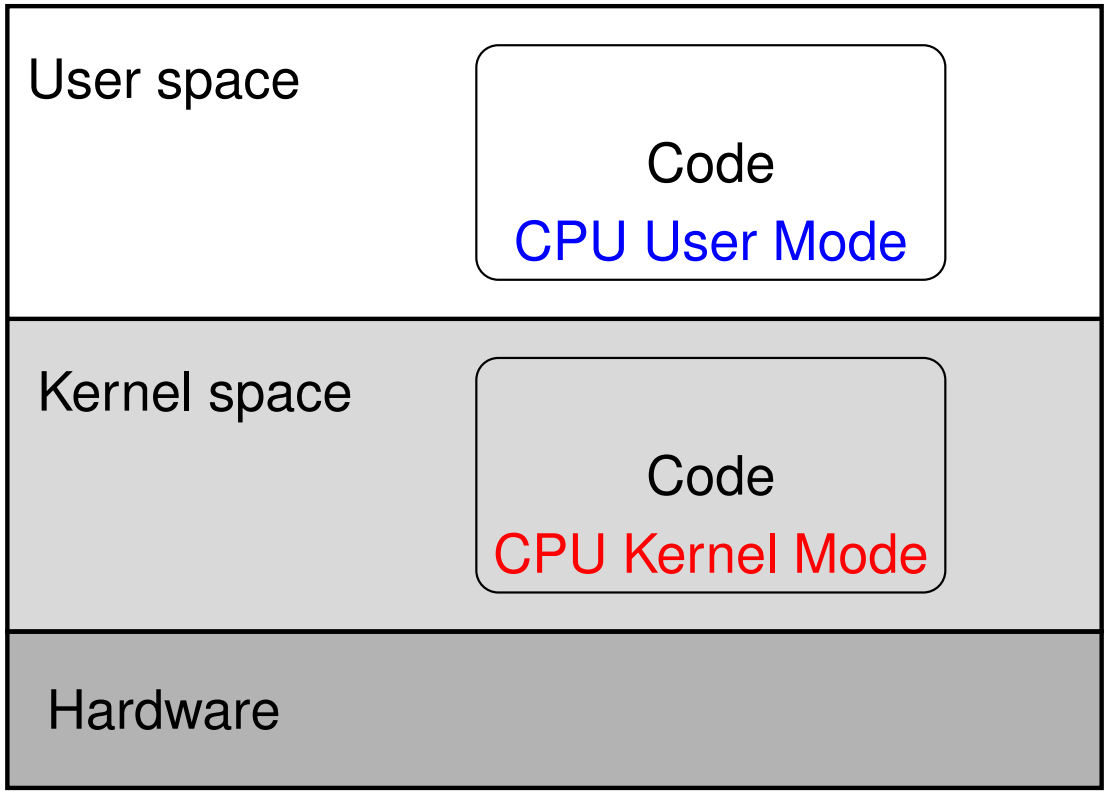
Kernel and user space



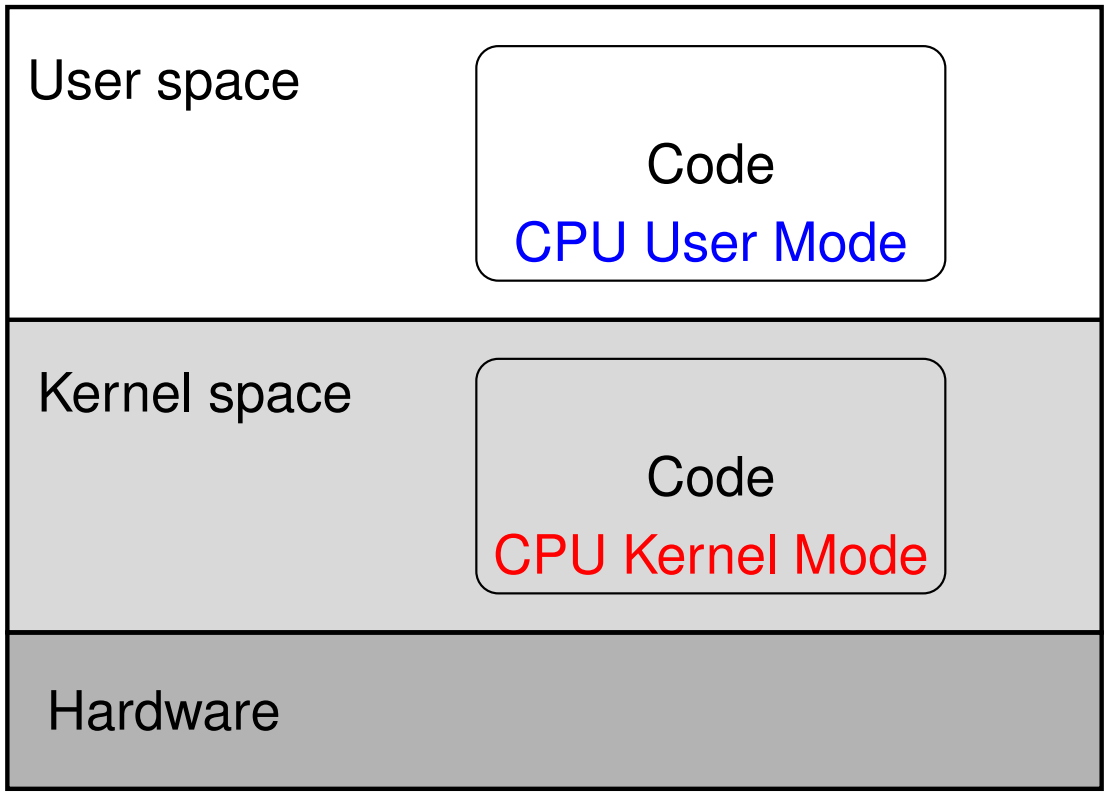
Kernel and user space



Kernel and user space



Kernel and user space



Terminology

The code running in kernel mode is often referred to as the **kernel**. Although this is the core of an operating system, typically they will also have user processes for user interfaces, scheduling daemons, ...

Interrupts

Entering the kernel to respond to events

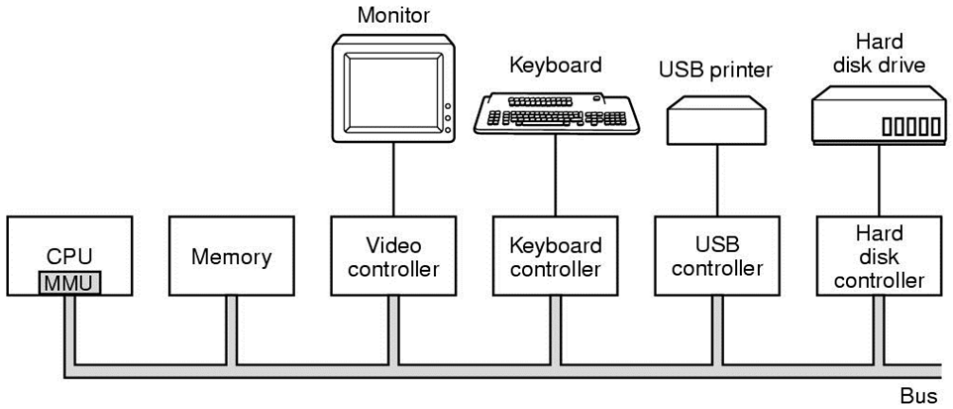


Figure: Simplified computer model (Tanenbaum, 2014)

The CPU cannot usefully live in isolation

Programs, and therefore the CPU, must be able to respond to various events.

For example:

- Connected hardware devices might need to communicate
- The passing of time
- Bad things happening - division by zero, hardware faults...

Interrupts

Entering the kernel to respond to events

Interrupts are the mechanism by which such events are handled.

- Abstractly, **an interrupt is a mechanism for changing the normal flow of execution**
- Can happen **asynchronously**, triggered by unpredictable factors external to the CPU, such as user input
  - Sometimes the term **interrupt** is reserved for this class
- Can also happen **synchronously**, triggered directly by the CPU executing an instruction
  - Sometime the term **exception** is used for this class

Interrupts

Entering the kernel to respond to events

Interrupt mechanism (sketch)

Interrupts

Entering the kernel to respond to events

Interrupt mechanism (sketch)

1. The CPU is doing some work - for example running a user process doing some calculations

Interrupts

Entering the kernel to respond to events

Interrupt mechanism (sketch)

1. The CPU is doing some work - for example running a user process doing some calculations
2. An interrupt is signalled by a hardware device - for example to indicate some IO data is available

Interrupts

Entering the kernel to respond to events

Interrupt mechanism (sketch)

1. The CPU is doing some work - for example running a user process doing some calculations
2. An interrupt is signalled by a hardware device - for example to indicate some IO data is available
3. The CPU records aspects of its current state, **switches to kernel mode**, and runs code in a **handler** to service the interrupt

Interrupts

Entering the kernel to respond to events

Interrupt mechanism (sketch)

1. The CPU is doing some work - for example running a user process doing some calculations
2. An interrupt is signalled by a hardware device - for example to indicate some IO data is available
3. The CPU records aspects of its current state, **switches to kernel mode**, and runs code in a **handler** to service the interrupt
4. Once completed, the CPU is returned to processing other tasks

Interrupts

Entering the kernel to respond to events

Challenges

Handling interrupts is one of the more challenging tasks of operating system implementation.

Interrupts

Entering the kernel to respond to events

Challenges

Handling interrupts is one of the more challenging tasks of operating system implementation.

- Interrupts can come at any time
  - Ideally handling interrupts should not take long
  - Handlers may split work into a **top** component dealt with immediately, and a **bottom** component scheduled to be dealt with later

Interrupts

Entering the kernel to respond to events

Challenges

Handling interrupts is one of the more challenging tasks of operating system implementation.

- Interrupts can come at any time
  - Ideally handling interrupts should not take long
  - Handlers may split work into a **top** component dealt with immediately, and a **bottom** component scheduled to be dealt with later
- Interrupts may be interrupted by other interrupts - it must be possible to **nest** interrupt handlers

Interrupts

Entering the kernel to respond to events

Challenges

Handling interrupts is one of the more challenging tasks of operating system implementation.

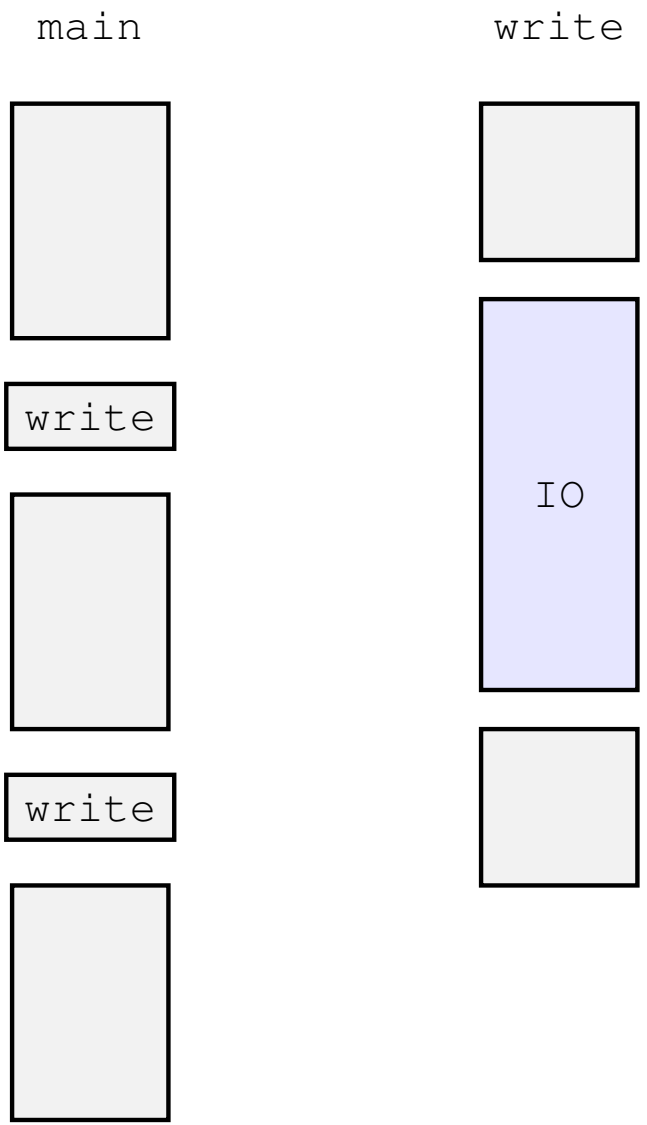
- Interrupts can come at any time
  - Ideally handling interrupts should not take long
  - Handlers may split work into a **top** component dealt with immediately, and a **bottom** component scheduled to be dealt with later
- Interrupts may be interrupted by other interrupts - it must be possible to **nest** interrupt handlers
- Sometimes critical code cannot be interrupted and they must be temporarily disabled

Interrupts

CPU utilisation

Blocking IO

Naive IO which wastes CPU cycles waiting for slow devices to respond.

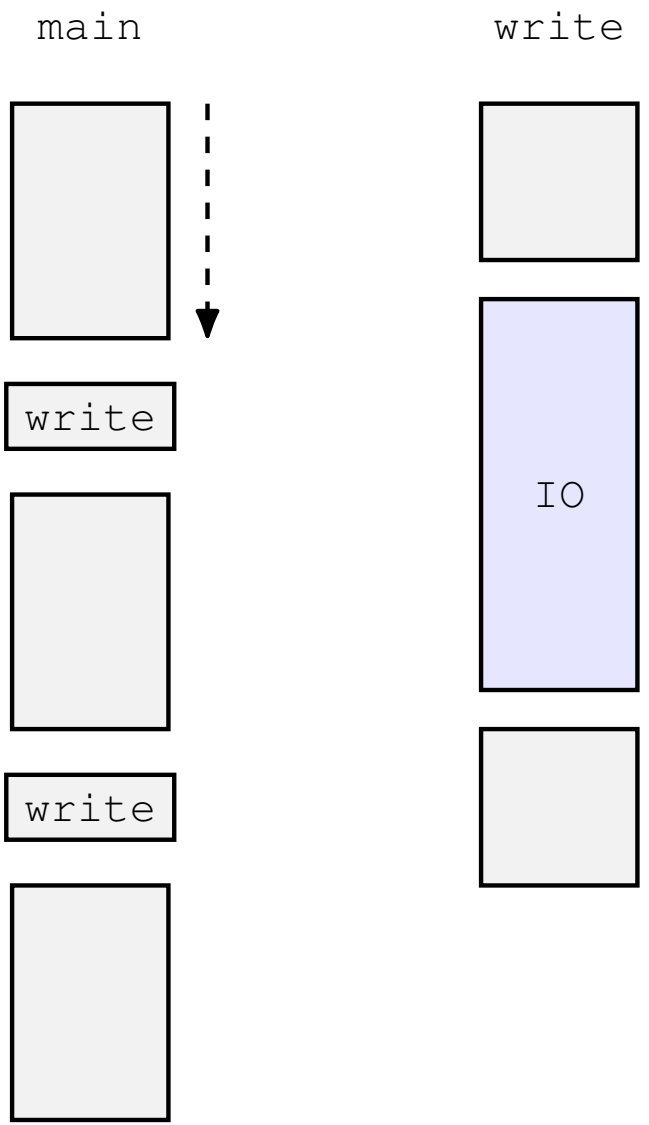


Interrupts

CPU utilisation

Blocking IO

Naive IO which wastes CPU cycles waiting for slow devices to respond.

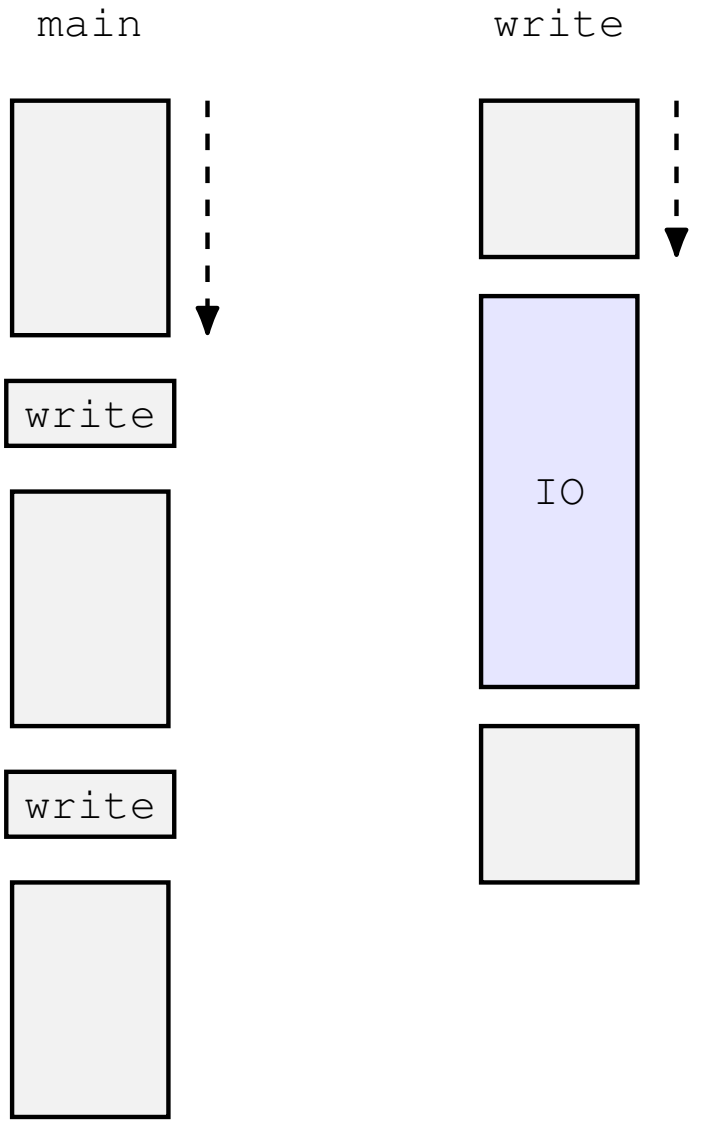


Interrupts

CPU utilisation

Blocking IO

Naive IO which wastes CPU cycles waiting for slow devices to respond.

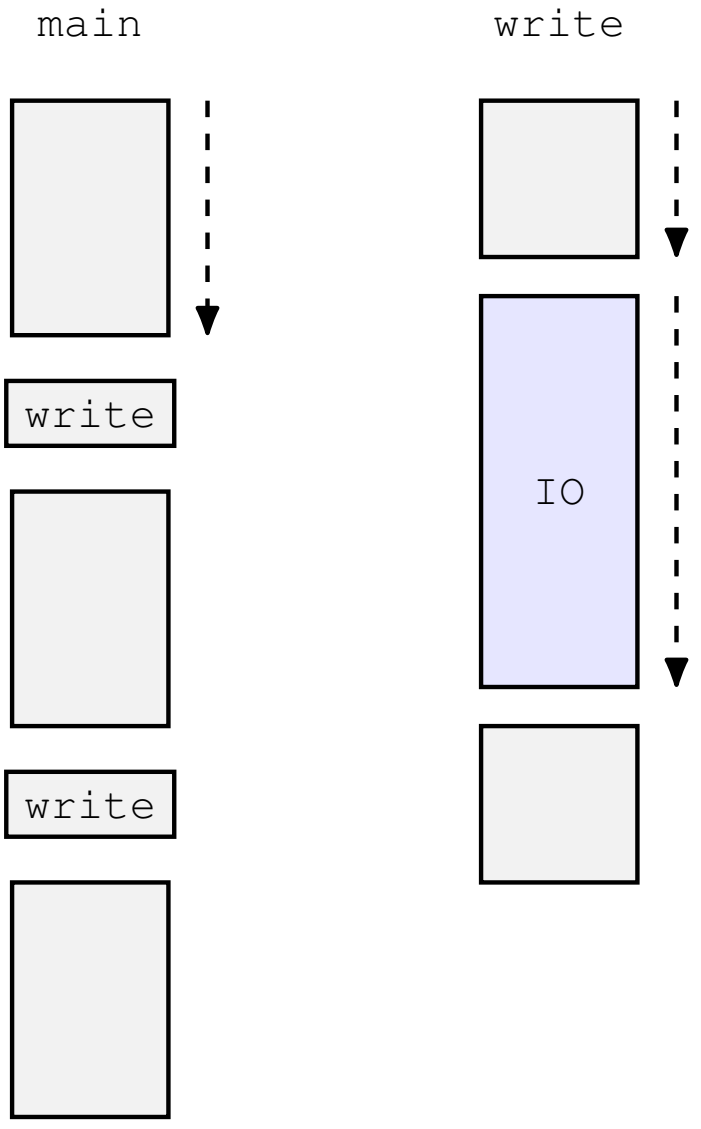


Interrupts

CPU utilisation

Blocking IO

Naive IO which wastes CPU cycles waiting for slow devices to respond.

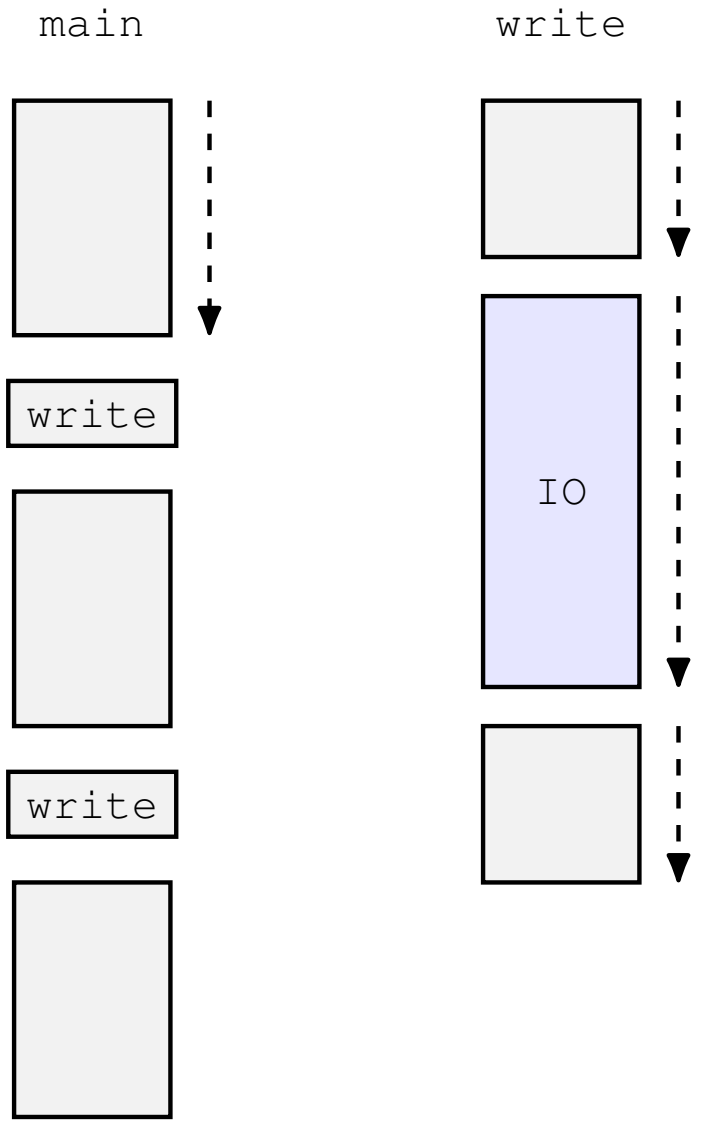


Interrupts

CPU utilisation

Blocking IO

Naive IO which wastes CPU cycles waiting for slow devices to respond.

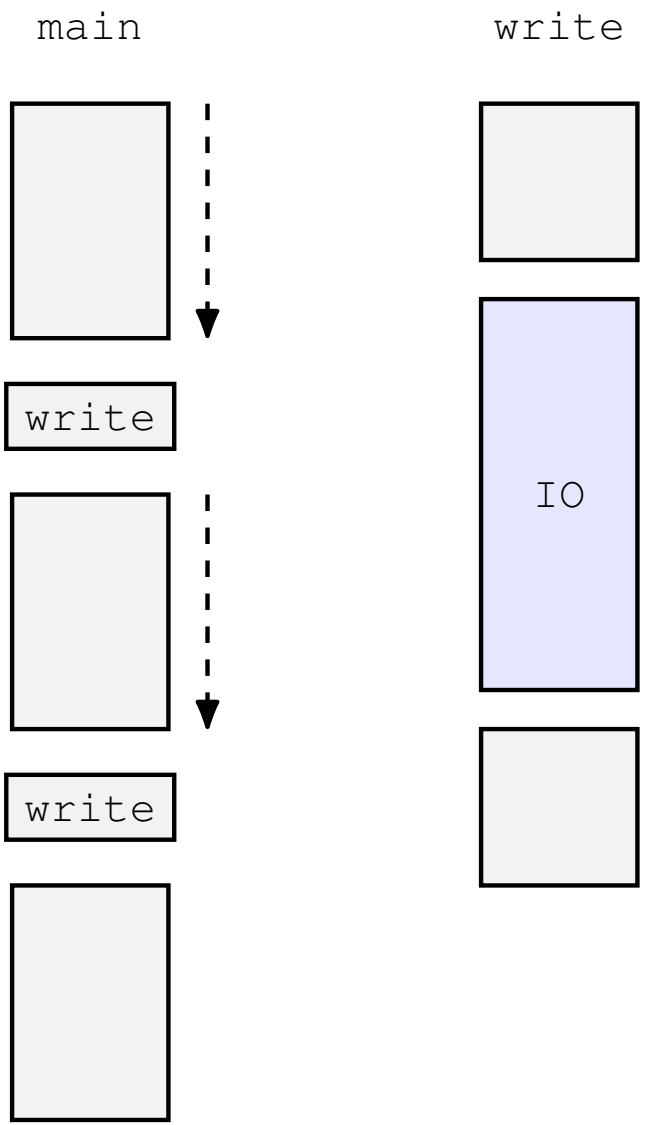


Interrupts

CPU utilisation

Blocking IO

Naive IO which wastes CPU cycles waiting for slow devices to respond.

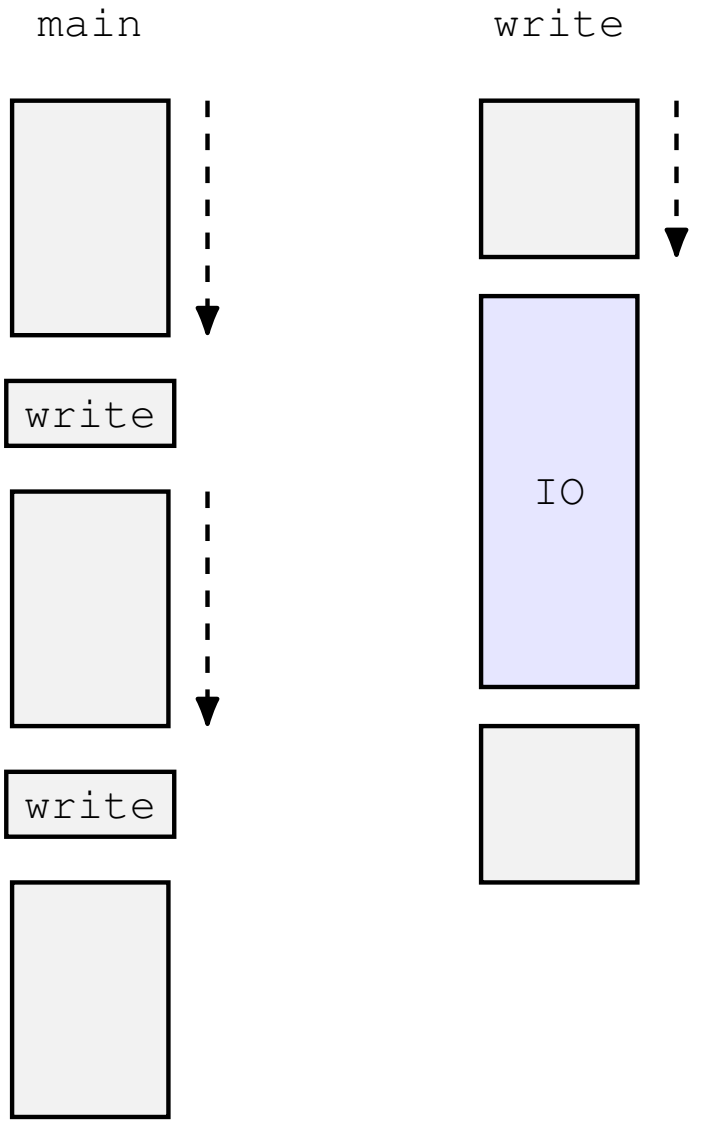


Interrupts

CPU utilisation

Blocking IO

Naive IO which wastes CPU cycles waiting for slow devices to respond.

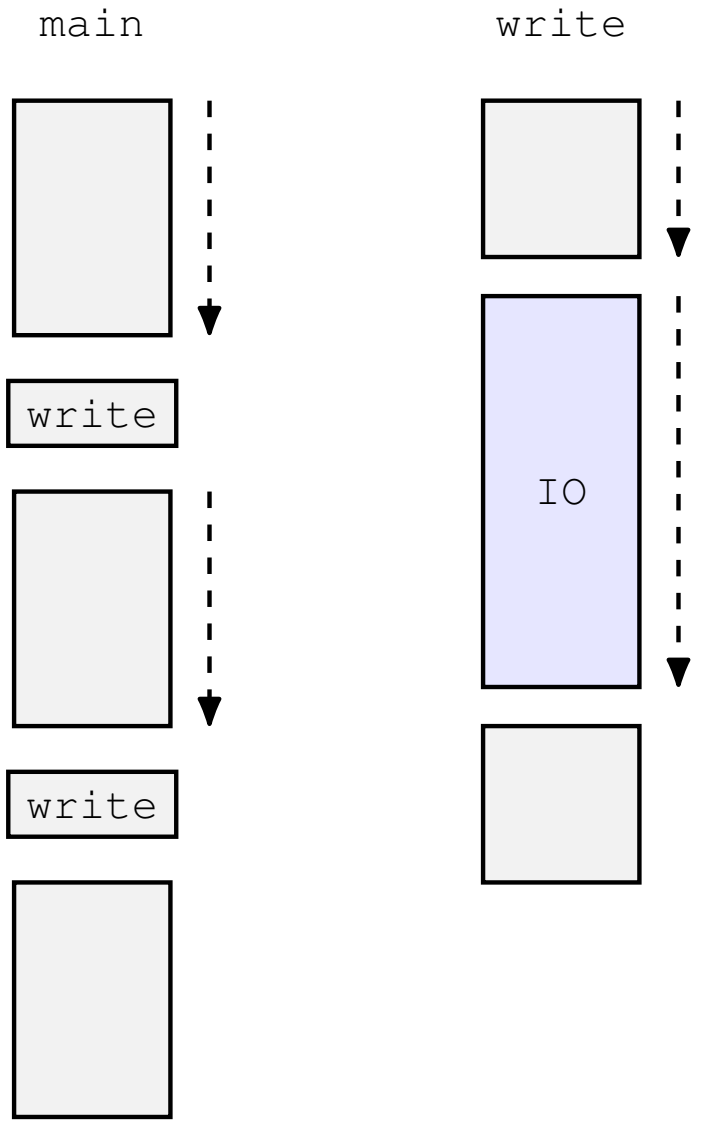


Interrupts

CPU utilisation

Blocking IO

Naive IO which wastes CPU cycles waiting for slow devices to respond.

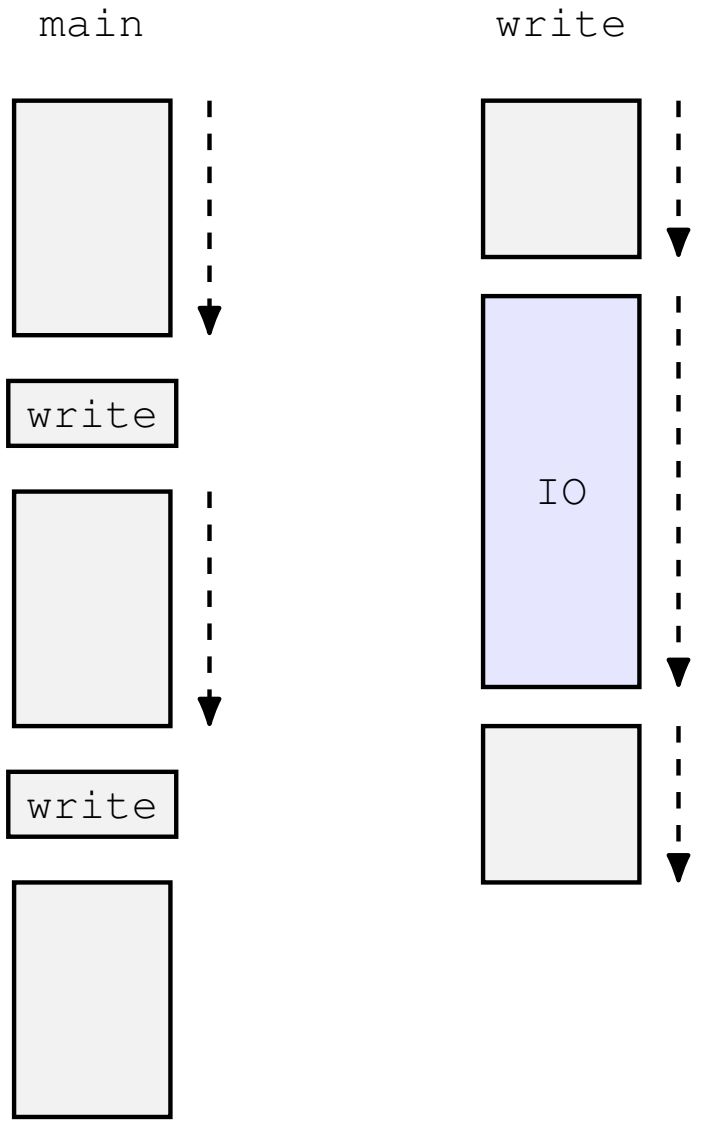


Interrupts

CPU utilisation

Blocking IO

Naive IO which wastes CPU cycles waiting for slow devices to respond.

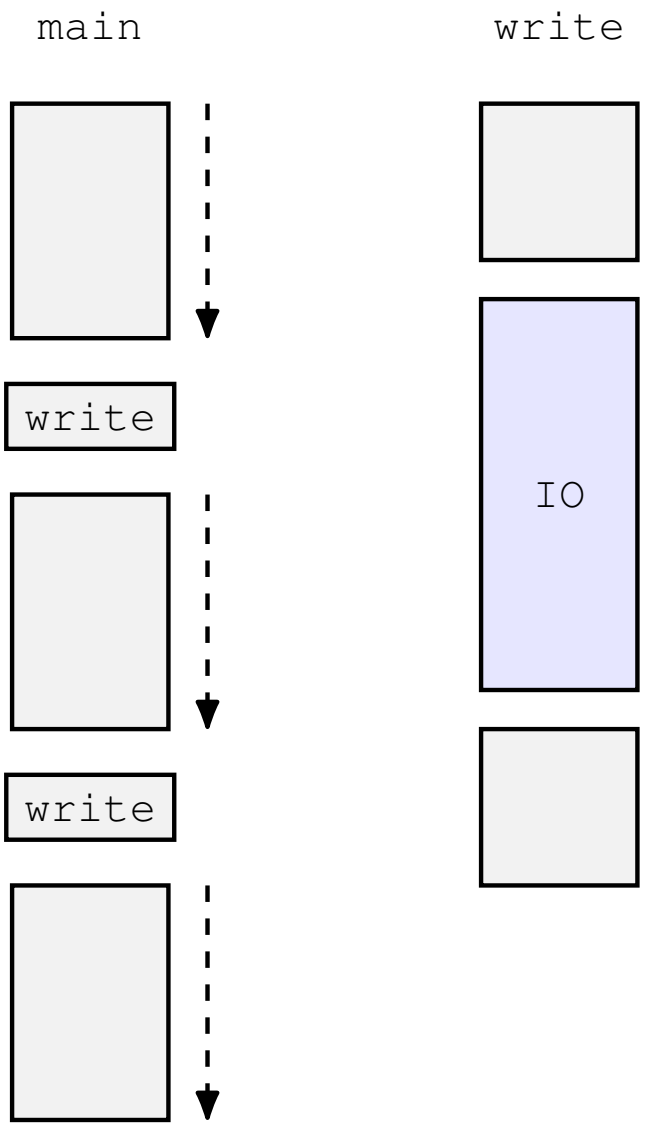


Interrupts

CPU utilisation

Blocking IO

Naive IO which wastes CPU cycles waiting for slow devices to respond.

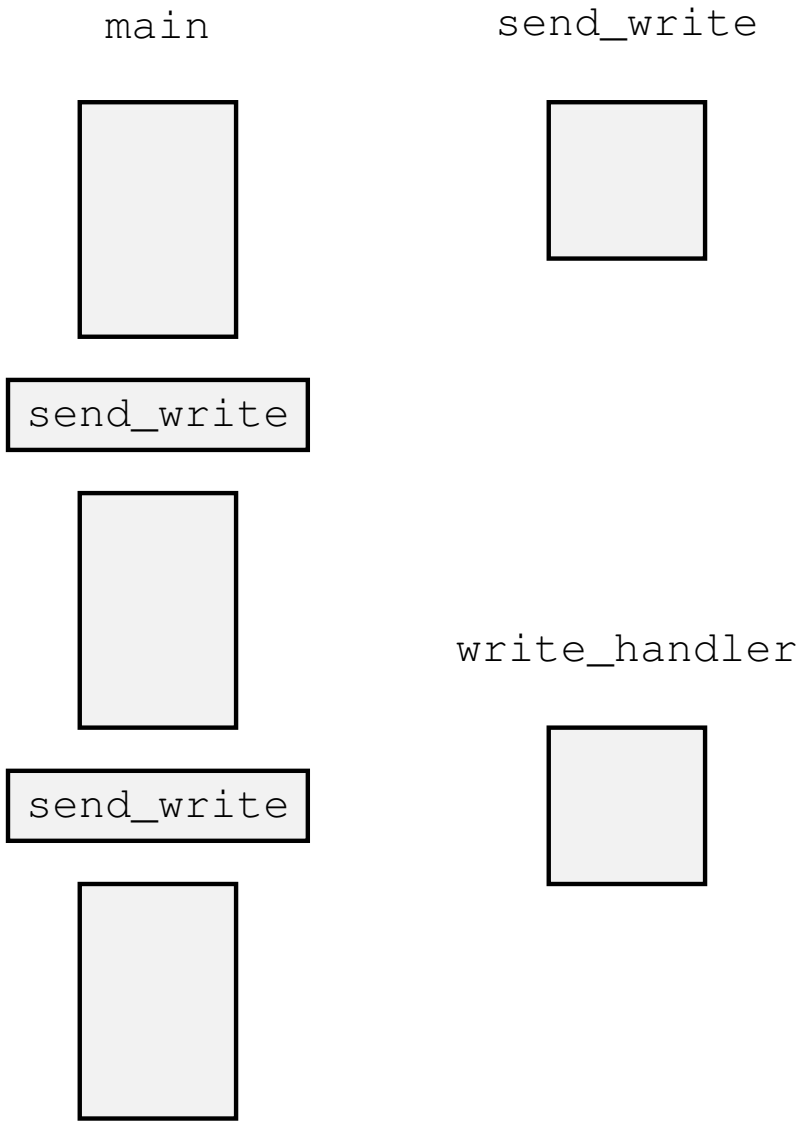


Interrupts

CPU utilisation

Interrupts

Better CPU usage supported by interrupts “calling back” when there is work to do.

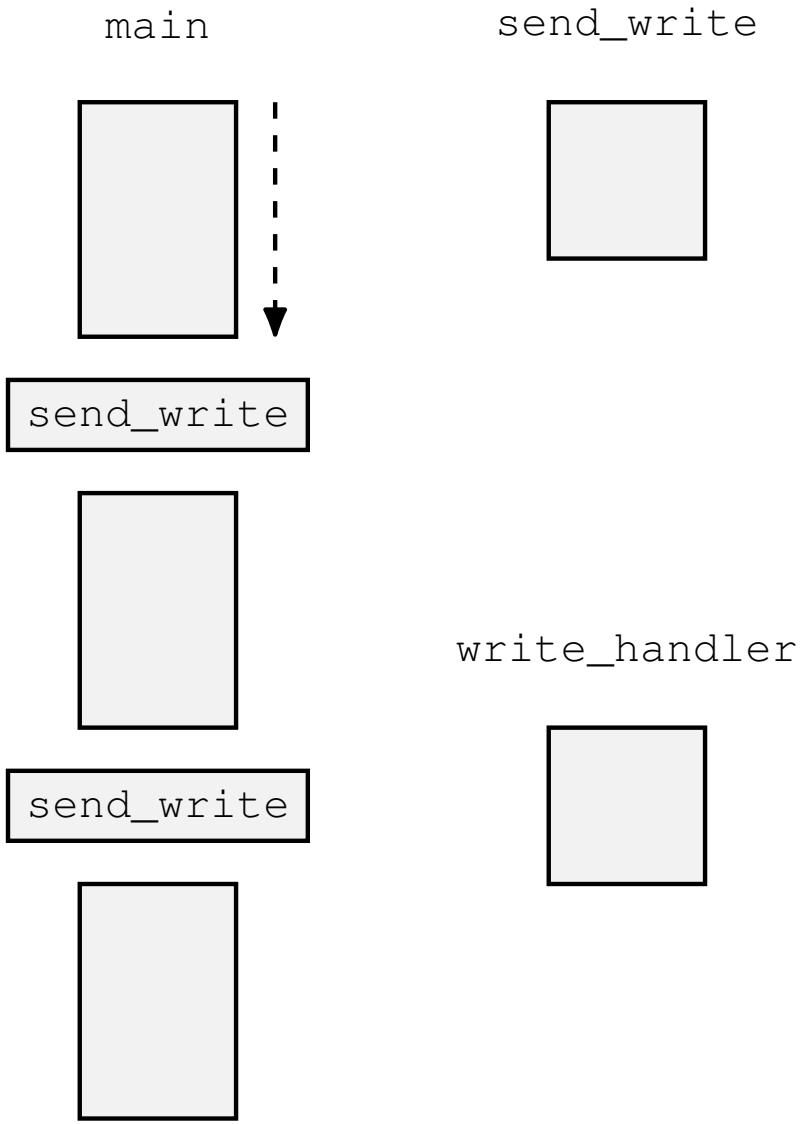


Interrupts

CPU utilisation

Interrupts

Better CPU usage supported by interrupts “calling back” when there is work to do.

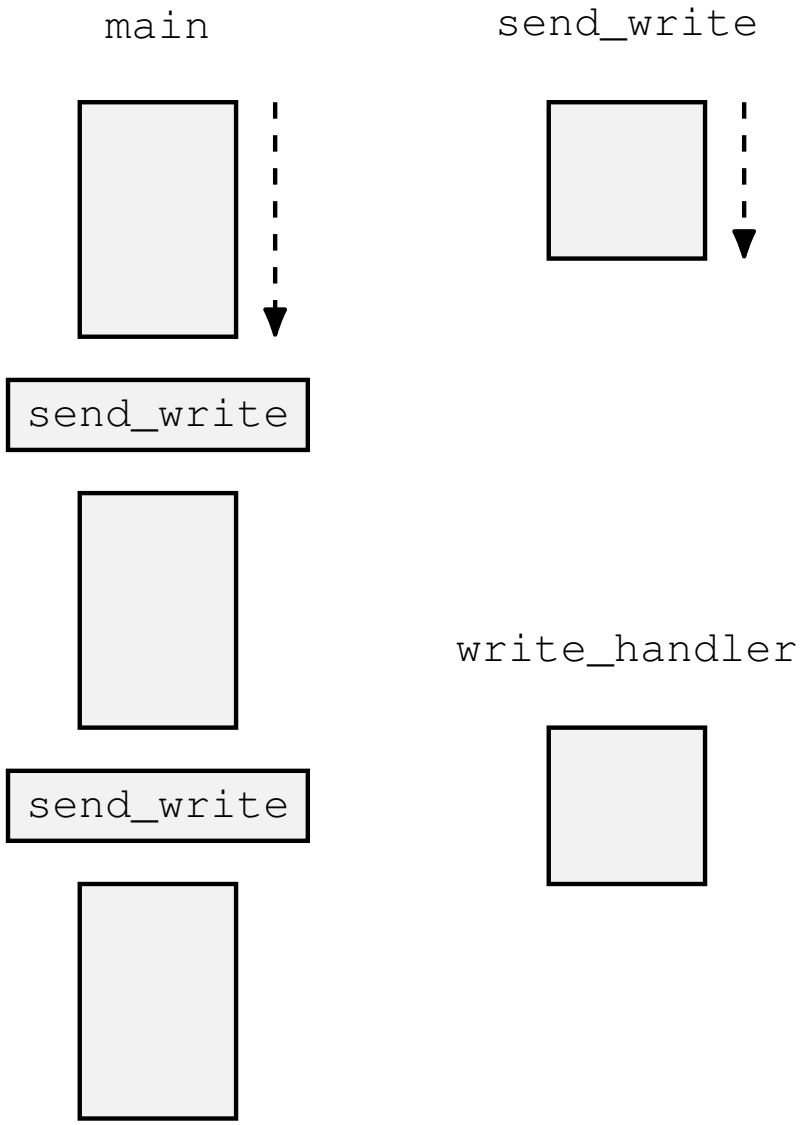


Interrupts

CPU utilisation

Interrupts

Better CPU usage supported by interrupts “calling back” when there is work to do.

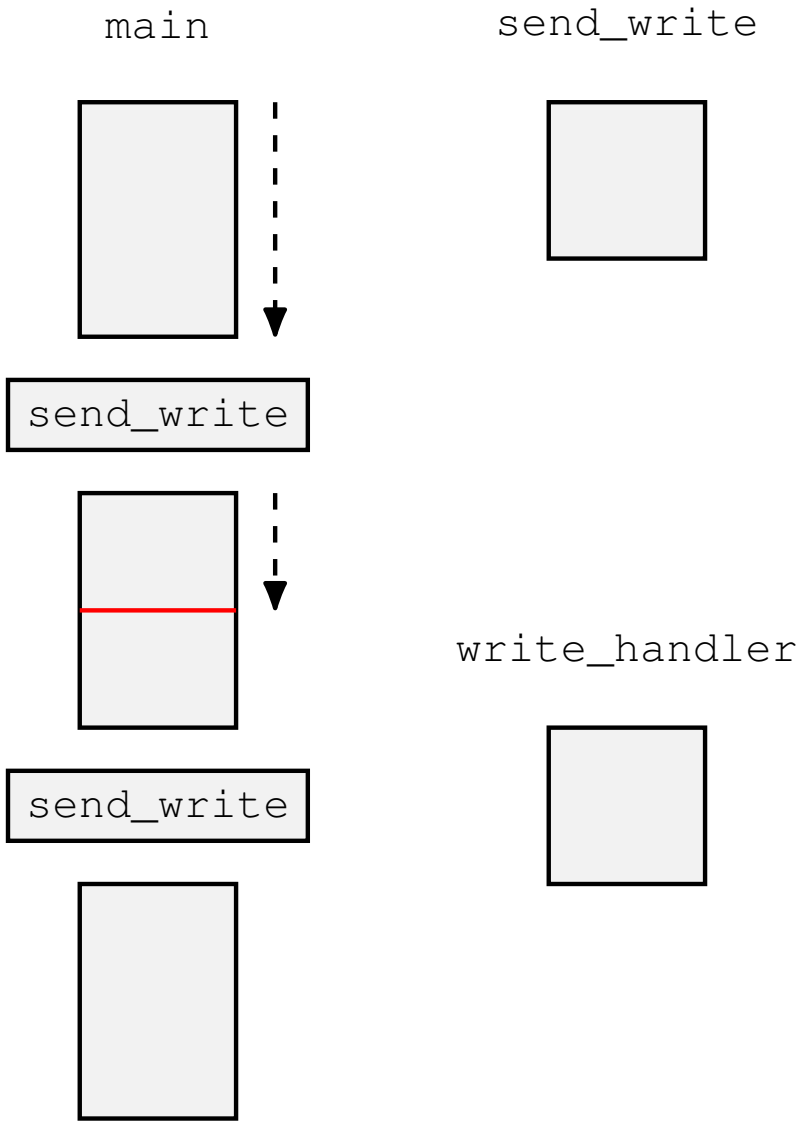


Interrupts

CPU utilisation

Interrupts

Better CPU usage supported by interrupts “calling back” when there is work to do.

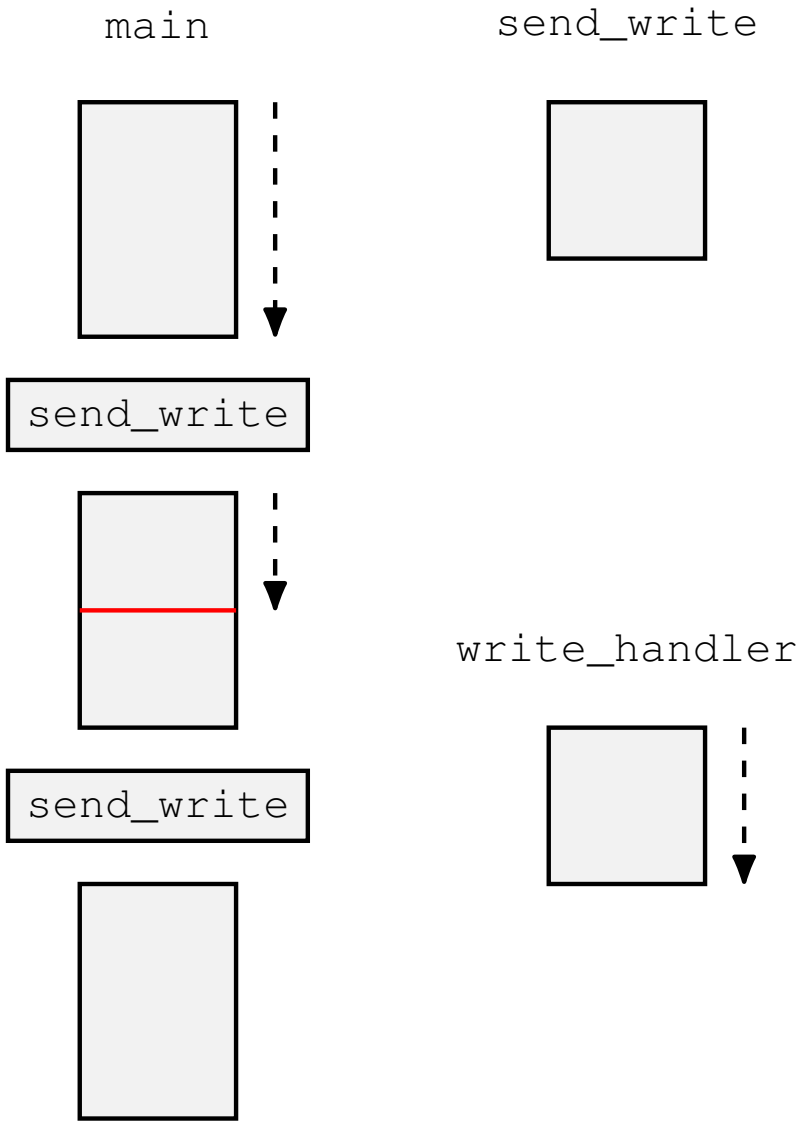


Interrupts

CPU utilisation

Interrupts

Better CPU usage supported by interrupts “calling back” when there is work to do.

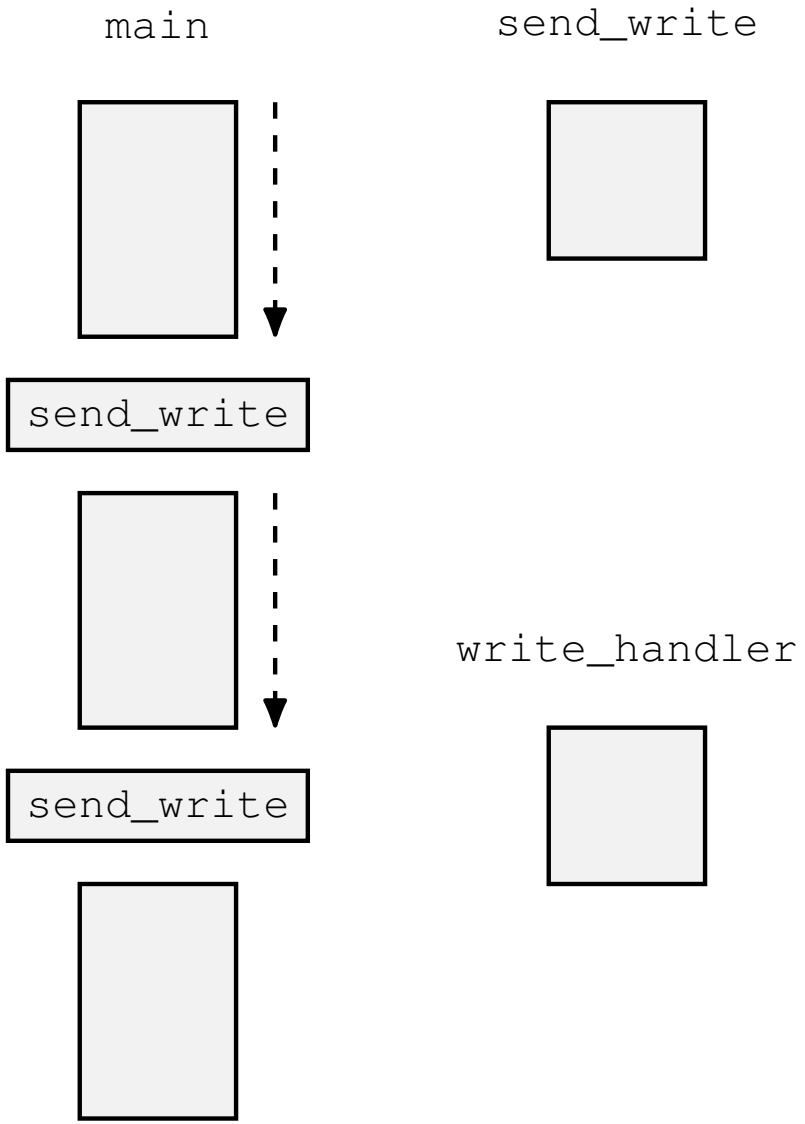


Interrupts

CPU utilisation

Interrupts

Better CPU usage supported by interrupts “calling back” when there is work to do.

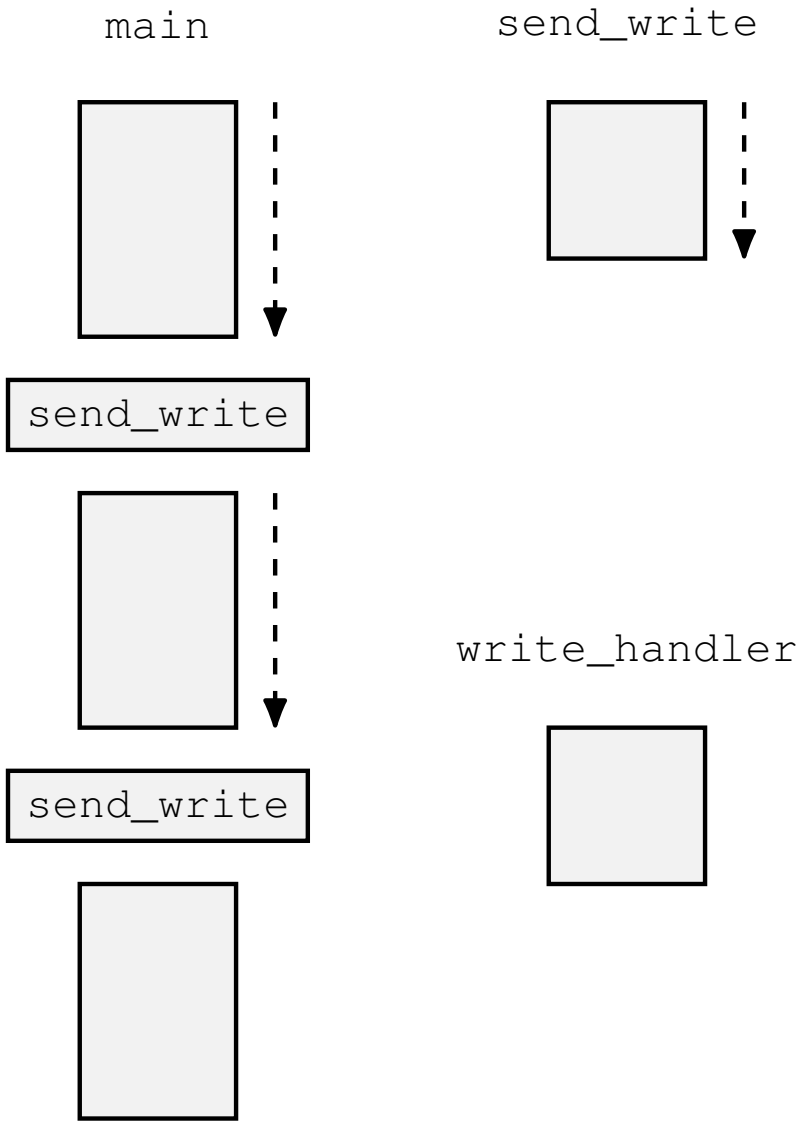


Interrupts

CPU utilisation

Interrupts

Better CPU usage supported by interrupts “calling back” when there is work to do.

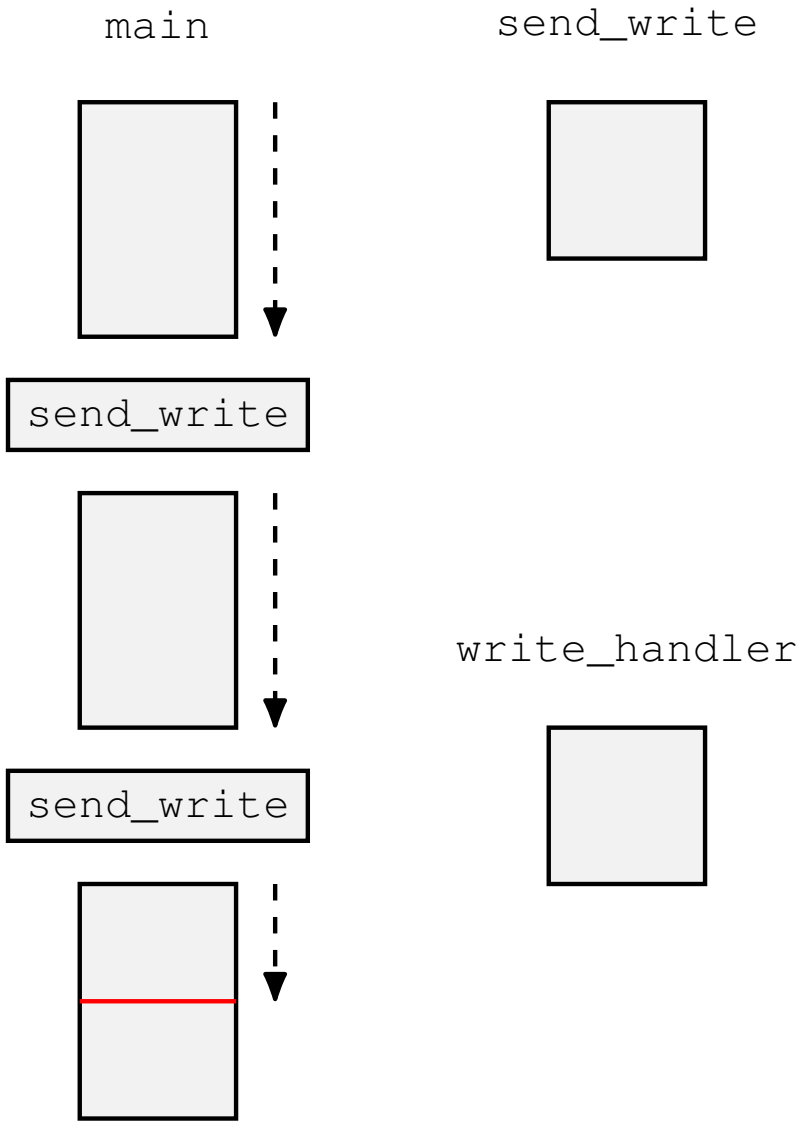


Interrupts

CPU utilisation

Interrupts

Better CPU usage supported by interrupts “calling back” when there is work to do.

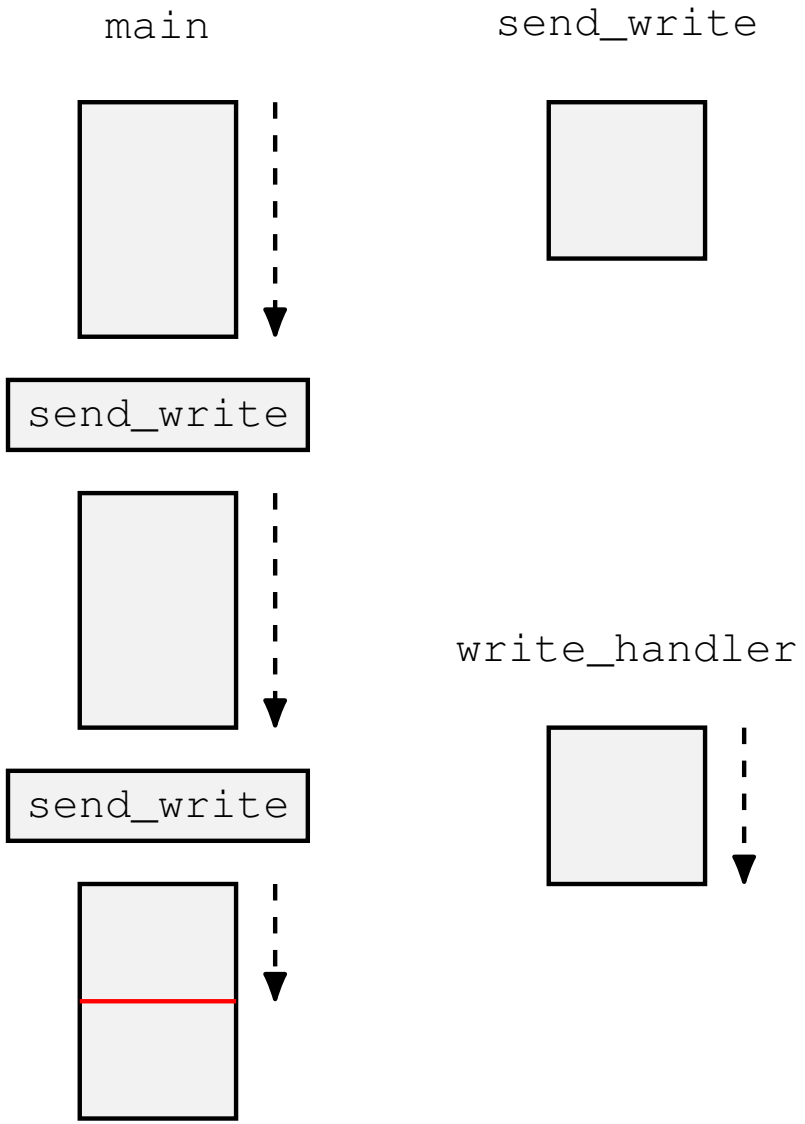


Interrupts

CPU utilisation

Interrupts

Better CPU usage supported by interrupts “calling back” when there is work to do.

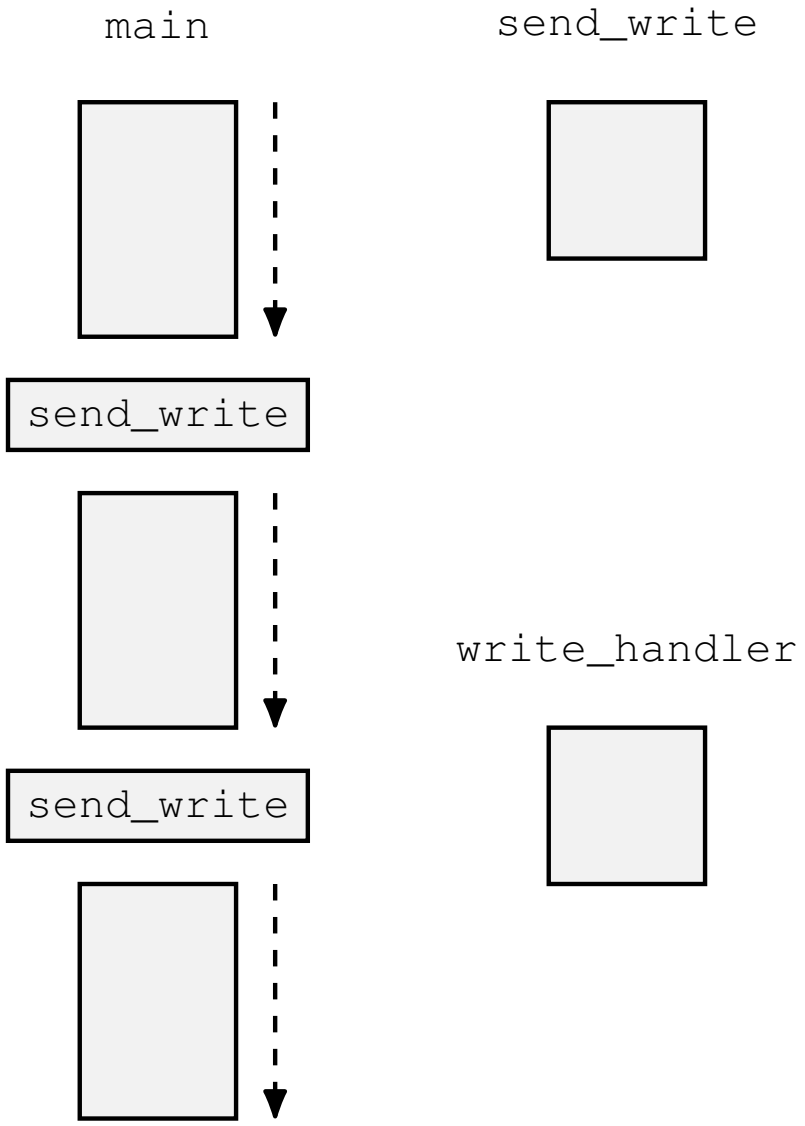


Interrupts

CPU utilisation

Interrupts

Better CPU usage supported by interrupts “calling back” when there is work to do.



System calls

Entering the kernel by software request

System calls are how programs request services from the operating system.

Example (Typical system calls)

- Requesting memory
- Accessing files
- Running programs (processes)
- Accessing concurrency features

System calls

Entering the kernel by software request

We can distinguish between an API and systems calls.

- An API is a programming interface - typically a **library of functions that run in user space**, for example pthreads
- To provide the required features **an API may need to make system calls to access the required functionality**
- The relationship is *not* necessarily one-to-one - a **single API function may invoke zero, one or many system calls**
- Often programmers use less fussy terminology, and simply refer to the API functions as system calls

System calls

Entering the kernel by software request

Question

How can a system call allow a user process to run kernel space code?

System calls

Entering the kernel by software request

System call mechanisms (naive sketch)

System calls

Entering the kernel by software request

System call mechanisms (naive sketch)

1. Some user code is running

System calls

Entering the kernel by software request

System call mechanisms (naive sketch)

1. Some user code is running
2. A system call is required - each has a **unique system call number**

System calls

Entering the kernel by software request

System call mechanisms (naive sketch)

1. Some user code is running
2. A system call is required - each has a **unique system call number**
3. The system call number is stored in a designated register

System calls

Entering the kernel by software request

System call mechanisms (naive sketch)

1. Some user code is running
2. A system call is required - each has a **unique system call number**
3. The system call number is stored in a designated register
4. The system call parameters are stored in designated registers

System calls

Entering the kernel by software request

System call mechanisms (naive sketch)

1. Some user code is running
2. A system call is required - each has a **unique system call number**
3. The system call number is stored in a designated register
4. The system call parameters are stored in designated registers
5. A synchronous interrupt (exception) is triggered by an instruction referred to as a **trap**

System calls

Entering the kernel by software request

System call mechanisms (naive sketch)

1. Some user code is running
2. A system call is required - each has a **unique system call number**
3. The system call number is stored in a designated register
4. The system call parameters are stored in designated registers
5. A synchronous interrupt (exception) is triggered by an instruction referred to as a **trap**
6. The interrupt is handled by kernel mode code, which calls a **system call service routine** which delivers the required functionality

System calls

Entering the kernel by software request

System call mechanisms (naive sketch)

1. Some user code is running
2. A system call is required - each has a **unique system call number**
3. The system call number is stored in a designated register
4. The system call parameters are stored in designated registers
5. A synchronous interrupt (exception) is triggered by an instruction referred to as a **trap**
6. The interrupt is handled by kernel mode code, which calls a **system call service routine** which delivers the required functionality
7. The operating system continues running the calling code

System calls

Entering the kernel by software request

System call mechanisms (slightly less naive)

- The system call service routine **may not actually service the request immediately** - for example if it must wait on some resource such as IO
- The operating system **may not continue running the original caller**
- **System calls are the kernel's big chance to get work done for the benefit of everything running on the system**

System calls

Entering the kernel by software request

System call mechanisms (fussier details)

- Details vary by operating system
- Modern approaches may avoid interrupt based mechanisms and use other CPU support for efficiency reasons

The C Programming Language

Criteria for an OS Implementation Language

Question

Why would we choose such an old language like C for OS implementation?

```
1 #include <stdio.h>
2
3 int main() {
4   printf("Hello Operating Systems and Concurrency\n");
5 }
```

Performance

As general purpose software, an OS must be fast enough for as many users as possible.

Performance

As general purpose software, an OS must be fast enough for as many users as possible.

- A slow OS slows every program that it runs.

Performance

As general purpose software, an OS must be fast enough for as many users as possible.

- A slow OS slows every program that it runs.
- Mobile hardware must also consider energy efficiency.

Performance

As general purpose software, an OS must be fast enough for as many users as possible.

- A slow OS slows every program that it runs.
- Mobile hardware must also consider energy efficiency.
- Bias toward performance over simplicity, elegance and maintainability.

Performance

As general purpose software, an OS must be fast enough for as many users as possible.

- A slow OS slows every program that it runs.
- Mobile hardware must also consider energy efficiency.
- Bias toward performance over simplicity, elegance and maintainability.
- Cannot sacrifice correctness.

Portability

As OS development is difficult, reuse is desirable.

Portability

As OS development is difficult, reuse is desirable.

- Must run directly on the hardware - no interpreted languages.

Portability

As OS development is difficult, reuse is desirable.

- Must run directly on the hardware - no interpreted languages.
- Ideally must be able to compile for many different hardware platforms.

Predictability

The behaviour of an OS must be predictable for user programs.

Predictability

The behaviour of an OS must be predictable for user programs.

- Example - games cannot tolerate unpredictable delays slowing frame rates and responsiveness.

Predictability

The behaviour of an OS must be predictable for user programs.

- Example - games cannot tolerate unpredictable delays slowing frame rates and responsiveness.
- Unpredictable behaviour such as garbage collection is inappropriate.

Recap

Take-Home Message

- Kernel mode code has more privileges than user code
- **Interrupts change the normal flow** of execution to invoke kernel code
- **System calls allow us to run kernel code** to access services of the operating system

Test your understanding

- What would be the advantages and disadvantages of a small kernel with a limited collection of system calls?
- What would be the advantages and disadvantages of a large kernel with a rich collection of system calls?
- How much harm can a bug in a user process cause?
- How much harm can a bug in the kernel cause?