Advanced Operating Systems

Niccoló Didoni

September 2022

Contents

I Introduction **1**

1 Introduction **2**
1.1 Basic concepts . . . 2
1.1.1 Program and process . . . 2
1.1.2 User and super mode . . . 2
1.2 Operating system . . . 3
1.2.1 Resource management . . . 3
1.2.2 Isolation . . . 6

II Linux processes **9**

2 Tasks **10**
2.1 Tasks description . . . 10
2.1.1 Task Control Block . . . 11
2.1.2 Task state . . . 12
2.2 Task hierarchy . . . 15
2.2.1 Kernel initialisation . . . 15
2.3 Task scheduling . . . 16
2.3.1 Run queues . . . 16
2.3.2 Completely Fair Scheduler . . . 17
2.3.3 Deadline Scheduler . . . 19

3 Synchronisation **21**
3.1 Processes . . . 21
3.1.1 Process creation . . . 21
3.1.2 Executing a program . . . 22
3.2 Process communication . . . 22
3.2.1 Waiting for the child . . . 22
3.2.2 Signal communication . . . 23
3.2.3 Pipes . . . 26
3.2.4 Messages . . . 29
3.2.5 Shared memory . . . 32
3.3 Process synchronisation . . . 33
3.3.1 Semaphores . . . 33

4 Task scheduling **35**
4.1 Problem model . . . 35
4.1.1 Task lifecycle . . . 35
4.1.2 Model . . . 36
4.2 Scheduling . . . 37
4.2.1 Optimisation metrics . . . 38
4.2.2 Algorithm classification . . . 38
4.2.3 Scheduling algorithms . . . 39
4.2.4 Priority scheduling . . . 41
4.2.5 Multi-processor scheduling . . . 43

5 Operating systems verification **46**
5.1 Software verification . . . 46
5.1.1 Model checking . . . 46
5.1.2 Automated theorem proving . . . 48

6 Memory safety **50**
6.1 Memory safety in C++ . . . 50
6.2 Rust . . . 52
6.2.1 Single ownership . . . 52
6.2.2 Move semantics . . . 53
6.2.3 Multiple ownership . . . 53

III Concurrency **55**

7 User space concurrency **56**
7.1 Concurrency execution models . . . 56
7.1.1 Thread execution models . . . 56
7.1.2 Lightweight execution models . . . 57
7.1.3 Event-based execution models . . . 57
7.2 Properties of concurrent programs . . . 59
7.2.1 Safety . . . 59
7.2.2 Liveness . . . 59
7.3 Locks . . . 64
7.3.1 How does a futex work . . . 64
7.3.2 Condition variables . . . 67
7.4 Event based concurrency . . . 67
7.4.1 Event loop . . . 67
7.4.2 Network events example . . . 68

8 Kernel space concurrency **70**
8.1 Operating system concurrency . . . 70
8.2 Sources of concurrency . . . 70
8.2.1 Interrupts . . . 70
8.2.2 Multiprocessing . . . 71
8.2.3 Kernel preemption . . . 71
8.3 Single processor locking . . . 74
8.3.1 Spinning locks . . . 74

8.3.2 Sleeping locks . . . 78
8.3.3 Read copy update locks . . . 78
8.4 Multi processors locking . . . 81
8.4.1 Optimised spinning locks . . . 81
8.5 Memory models . . . 83
8.5.1 Sequential consistency . . . 84
8.5.2 Total store order . . . 85
8.5.3 Partial store order . . . 86
8.5.4 Data races . . . 87
8.5.5 Software memory models . . . 88

IV Virtualisation 90

9 Linux virtual memory 91
9.1 Virtual memory . . . 91
9.2 Virtual memory areas . . . 92
9.2.1 Kernel virtual memory . . . 92
9.2.2 User virtual memory . . . 93

10 Linux physical memory 98
10.1 Page allocation . . . 98
10.1.1 Non-Uniform Memory Access . . . 98
10.1.2 Linux memory representation . . . 98
10.1.3 Buddy algorithm . . . 101
10.1.4 Page cache . . . 102
10.2 Page frame reclaiming algorithm . . . 104
10.2.1 LRU list management . . . 105
10.3 Object allocation . . . 107
10.3.1 Quicklists . . . 107
10.3.2 Slab allocator . . . 107

11 Memory security 110
11.1 Covert channels . . . 110
11.1.1 The cr3 register . . . 111
11.1.2 Kernel Page Table Isolation . . . 111

V Virtual machines 113

12 Introduction 114
12.1 Virtual machines . . . 114
12.1.1 Hypervisors . . . 114
12.1.2 Requirements . . . 115
12.1.3 Motivations . . . 115
12.1.4 Instructions . . . 115
12.1.5 Virtual machine classification . . . 116
12.2 Software based virtualisation . . . 116
12.2.1 Shadow page table . . . 117

12.2.2 Ring aliasing problem . . . 117
12.2.3 Excessive faulting problem . . . 118
12.2.4 Binary translator . . . 118
12.3 Hardware based virtualisation . . . 118
12.3.1 Multiple privileged modes . . . 119
12.3.2 Comparison of software and hardware based virtualisation . . . 119
12.3.3 Performance improvements . . . 119

13 Real word examples **121**
13.1 KVM . . . 121
13.2 Intel x86 VT-x . . . 121

14 Paravirtualisation **122**
14.1 Paravirtualisation . . . 122
14.1.1 Virtio . . . 122
14.2 Containerisation . . . 123
14.2.1 Namespaces . . . 124
14.2.2 Control groups . . . 124

VI Input/Output and drivers **125**

15 Input and output **126**
15.1 Processor's architecture and input/output . . . 126
15.1.1 Memory mapped ports . . . 126
15.1.2 IO ports . . . 129
15.2 Device to CPU communication . . . 129
15.2.1 Polling . . . 129
15.2.2 Interrupts . . . 129
15.3 Interrupts . . . 130
15.3.1 Classification . . . 130
15.3.2 Interrupts flow . . . 131
15.3.3 Interrupt controllers . . . 131
15.3.4 Messaging signal interface . . . 132
15.3.5 Deferred interrupts . . . 132
15.3.6 Timers . . . 134

16 Linux device management **136**
16.1 Devices . . . 136
16.1.1 Dev folder structure . . . 136
16.2 Device drivers . . . 137
16.2.1 Character device drivers . . . 137
16.2.2 Block device drivers . . . 138
16.2.3 IO schedulers . . . 141

VII C++ operating system programming 146

17 C++ 147
17.1 Classes . . . 147
17.1.1 Class definition . . . 147
17.1.2 Encapsulation . . . 148
17.1.3 Inheritance . . . 148
17.1.4 Polymorphism . . . 149
17.1.5 Member qualifiers . . . 149
17.1.6 Operators . . . 150
17.2 Object creation . . . 150
17.3 Templates . . . 150
17.4 Pointers . . . 150
17.4.1 Basic pointers . . . 150
17.4.2 Smart pointers . . . 150
17.5 Function objects . . . 153
17.5.1 Bind expressions . . . 153
17.5.2 Lambda expressions . . . 154

18 Thread programming 156
18.1 Threads . . . 156
18.1.1 Thread creation . . . 157
18.2 Synchronisation . . . 157
18.2.1 Locking classes . . . 157
18.2.2 Condition variables . . . 162

19 Drivers 164
19.1 Miosix . . . 164
19.1.1 Booting . . . 164
19.1.2 Concurrency . . . 170
19.1.3 The file-system . . . 172
19.2 Linux kernel . . . 172
19.2.1 Concurrency . . . 173
19.2.2 The Linux filesystem . . . 175

Part I

Introduction

Chapter 1

Introduction

1.1 Basic concepts

1.1.1 Program and process

Before diving into an overview over the main concepts over Operating Systems, we should give some basic definitions and distinguish some concepts. For starters we should highlight the difference between **program** and **process**. In particular, a program can be defined as follows:

> **Definition 1.1** (Program)**.** *A program is a set of instructions stored somewhere in memory, but not in execution.*

Differently, a process is:

> **Definition 1.2** (Process)**.** *A process is an instance of program in execution and the data related to it. A process is seen from an operating system perspective and usually isn't related to the user interface used to communicate with the user.*

Another word we usually hear when talking about operating systems is thread.

> **Definition 1.3** (Thread)**.** *A thread is the smallest schedulable unit of execution. This means that we can't divide anything running on a processor in something smaller than a thread.*

1.1.2 User and super mode

Another important difference to understand when talking about operating systems is the difference between user and super-mode. These modes refer to ways in which an instruction can be executed. In particular, a program issues instructions to the processor in user mode while, to execute some privileged instruction it has to ask the operating system to execute them, in super mode. In other words:

- In **user mode**, a program can run only some basic instructions and can only use some addresses.

- In **supervised mode** (or kernel mode, or super mode), the operating system can execute all instructions, including those that require to read and write from and to peripherals and files. Moreover, in super mode, the OS can address the whole memory (i.e., all addresses).

1.2 Operating system

Since we will talk a lot about operating systems, let us understand what does an operating system do. An OS has to handle:

- **Resource management**. Many applications run on a single computer, hence many applications have to share the same processor. That being said, every program should work like it could use a resource by its own. This process is handled by the OS that manages who can access the physical resources (i.e., the processor). The goal of the operating system is to share the physical resources so that every process has a fair treatment, i.e., every process receives a fair share of the resources. Long story short, the OS assigns a resource to a single process.
- **I/O** and **peripherals**. A computer is typically connected to multiple peripherals, which are handled by the operating system. In particular, the OS mediates the access to the peripherals, allowing one program at a time to use them. As an example, let us consider the access the keyboard. Only one program can read what the user is writing, and this task is handled by the OS.
- **Isolation** and **protection**. A computer has a single physical memory, however, each program running should not access the parts of memory used by other programs. Isolating the different programs running is a task handled by the operating system. The OS has the responsibility to isolate each process and not allow a process to write and read blocks of memory used by others. More specifically, the OS uses virtual memories to handle the memory of each process independently and in isolation. Note that, isolation and protection doesn't only apply to memory but also to access control, i.e., which user (or program) can read or write which file.
- **Portability** and **extendibility**. An operating system is also responsible for abstracting the complexity of the hardware of a computer. For instance, the OS is capable of hiding the complexity of the peripherals using files. Another example is storage. The operating system should allow programs to work with the storage, independently from the specific storage technology (e.g., SSD or HDD). Basically, the OS is an intermediary between the complexity of the hardware (e.g., peripherals and storage) and the programs. Note that, portability is an advantage not only for the programs and for those who write such programs, but also for those who build the components.

Let us now analyse more in detail how an operating system might handle some of the aforementioned tasks.

1.2.1 Resource management

Multiplexing

When talking about resource management we mainly refer to how an OS assigns the processor to a program so that it can execute its instructions. Basically, how the OS multiplexes the CPU. The basic idea behind multiplexing is exploiting the CPU at its best so that

- **No computational power is wasted**. This means that if two applications are ready to run, when one stops running, the other can immediately start executing so that the CPU is always used by some process. In particular, we want to exploit all those moments in which a process stops because it's waiting for something (e.g., an input from the user), and execute another process which is ready for execution and isn't waiting. Note that, to handle this constant exchange between processes, the OS has to be aware that a process is waiting for something and when it arrives. To handle waiting and restarting, the OS uses interrupts. Interrupts are hardware messages exchanged whenever an hardware (or, for exceptions, software) event happens. For instance, an interrupt is sent every time a key is pressed on the keyboard.
- **Latency is reduced**. Consider for instance two processes P_1 and P_2 that require 80 seconds and 20 seconds respectively to complete. Now imagine that P_1 is executed before P_2 and it's executed completely before P_2 starts running. All in all, completing both process require 100 seconds, however the latter has a latency of 80 seconds because it starts executing after P_1, which requires 80 seconds before ending. On the other hand, say we interleave process P_1 and P_2 (e.g., each process is executed for 10 seconds and then the other starts executing). In this context, even if both processes still require 100 seconds, P_2 ends after 40 seconds and doesn't have to wait for P_1.

Process

Before going on analysing how does multiplexing work, we should give a look at how processes work. In particular, we can say that a process can be in one of the following states:

- **Running**. A process is in the running state when its instructions are executed by the processor.
- **Ready**. A process is in the ready state when it could be running, but another process is in the running state.
- **Blocked**. A process is in the blocked state when it is waiting for an event to arrive before going back in the ready state.
- **Terminated**. A process is in the terminated state when it has no more instructions to execute.
- **New**. A process is in the new state after being created. Immediately after being created, a process isn't immediately in the ready state because the OS has to prepare some structure to handle the process itself, hence it can't start executing the newly created process.

The finite state machine representing all the states in which a process can be is shown in Figure 1.1.

The state in which a process finds itself is saved in a structure called Process Control Block (PCB). A PCB is held and accessed only by the OS and it's therefore saved in the kernel space. A PCB doesn't only store the information regarding the state of a process but also other information about the process itself. Also remember that each process has its own PCB.

Context switch

Now that we know in which states a process can be, we should understand how a process moves from a state to another. Before describing the transitions from one state to the other we have to make a fundamental difference. In particular, we want to analyse the difference between **preemptive** and **non-preemptive** operating systems. Let's start by saying what a preemptive operating system is:

Definition 1.4 (Preemptive operating system)**.** *A preemptive operating system is an OS that can stop a process and give control to another process.*

On the other hand, a non-preemptive operating system is defined as follows:

Definition 1.5 (Non-preemptive operating system)**.** *A non-preemptive operating system is an OS in which a process holds a resource until it's terminated.*

Long story short, in preemptive operating systems, a process can be stopped even when it's not terminated or it's not waiting for some event. We will consider preemptive OSes. Since we find ourselves talking about preemptive OSes, we have to analyse how one process is stopped from executing (even when it hasn't finished or it doesn't have to wait for some event) and replaced by another process. This operation is called **context switch**. A context switch is regulated by interrupts, i.e., signals sent on the bus. In particular, when an interrupt or an exception (i.e., a software interrupt, coming from a syscall) is raised, the control is given to the OS which,

1. Saves the state of the process currently executing, say P_1, using its Process Control Block PCB_1.
2. Decides which process should be executing next, among those that are in the ready state. Say the OS chooses to execute process P_2.
3. Loads the state of P_2 using its Process Control Block PCB_2. In particular, the OS loads on the CPU's registers the values of the registers saved in PCB_2 so that, when the control is released by the OS, P_2 starts executing.

Scheduling Let us focus for a bit on the second operation done during a context switch. We've said that the OS has to choose a new process to execute (P_2 in our example). The process of choosing a new process (among those that are in the ready state) is called scheduling. The criteria with which the OS might choose what process to schedule vary from OS to OS and even the same OS can offer different modes to handle scheduling. Usually a OS considers the following parameters:

- **Fairness**. Fairness considers if all process are able to proceed without starving. Basically, the OS (or more properly, the scheduler) assigns a fair and equal share of the CPU to each process so that no program waits for a long without using the CPU.
- **Throughput**. Throughput allows to have no lag and a good overall performance. Having a good throughput means having a responsive and fluid user interaction.
- **Efficiency**. Efficiency takes into consideration how much time is needed to decide what process to schedule. Basically, for some purposes, we don't want to spend to much time deciding what process to execute next.
- **Priority**. Priority handles the fact that some processes require more CPU (i.e., have more priority).
- **Deadlines**. Deadlines consider the fact that some processes have to terminate before some time instant.

Each OS, or each mode might decide to implement a policy (i.e., a scheduling policy) to optimise one parameter or another, depending on the purpose of the OS.

Keeping in mind the aforementioned parameters that a OS can optimise when scheduling, we can divide preemptive operating systems in two categories, depending on how they handle the goals (or parameters) above:

- **General Purpose Operating Systems** (GPOS). General purpose operating systems provide, most of the time, a fair share of the processor to every program. This result is obtained using time windows. A process (i.e., the running process) is assigned a time window and, upon ending, a new process is chosen for executing in the following window. When a process is stopped not because it has terminated or it's waiting for an event, we say that it is preempted by the OS. To ensure fairness, even if a waiting process is woken up, the running process keeps executing until its window is over. To make things clearer let us consider three processes P_1, P_2 and P_3, where the first has an higher priority the the last two and P_2 has the same priority of P_3. P_1 starts executing for some time, until t_1. At t_1, P_1 is preempted (P_1 hasn't terminated nor it's waiting for an event) and P_2 is chosen for the next time window. Whilst P_2 is running, P_3 is woken up (say P_3 was waiting user's input), however P_2 keeps executing because its time window isn't over yet. At time t_2, P_2's window ends and it P_1 is scheduled for execution. At time t_3, before the end of its window, P_1 stops because it has to wait an event E and P_3 is chosen to be executed. Whist P_3 is executing, an interrupt for event E is raised, however, even if P_1 has an higher priority than P_3, P_3 isn't preempted (and replaced by P_1) until its time window is over. This example is shown in Figure 1.2. General purpose operating systems are also called best effort operating systems.
- **Real Time Operating Systems** (RTOS). Real time operating systems have to deal with strict and predictable time sequences. This is because some processes have deadlines and the OS has to try and satisfy all of them. Deadlines are enforced using priorities, which, are handled differently from GPOSes. The basic idea is that, when an high priority process P_h is ready, the running process P_r is preempted and P_h is immediately executed. The same thing is true for processes with the same priority. Long story short, when a higher priority process is awakened, it's executed (and the running process is preempted). Always keep in mind however that processes with higher priority have the precedence with respect to lower priority ones and higher priority processes are always given control. This behaviour (giving control to high priority process) is justified by the hypothesis that the OS doesn't always have to deal with high priority processes. Moreover, since RTOSes are based on priorities, they usually have more priority levels with respect to GPOSes. To recap the characteristics of a RTOS, let us consider the following example. Say we have three processes P_1, P_2 and P_3 with three different levels of priority in decreasing order (P_1 having the higher priority and P_3 the lowest). P_1 starts executing until t_1 when P_3 is woken up by an interrupt and it starts executing immediately (and P_1 is preempted). After some time, at t_2, P_2 is woken up, P_3 preempted and P_2 starts executing. A while after, P_1 is woken up again and starts executing immediately. This example is shown in Figure 1.3. As we can see, since a woken process is always executed, RTOS generate predictable scheduling sequences. Also remember that some RTOSes can use time windows, which aren't however a distinctive feature of such operating systems.

1.2.2 Isolation

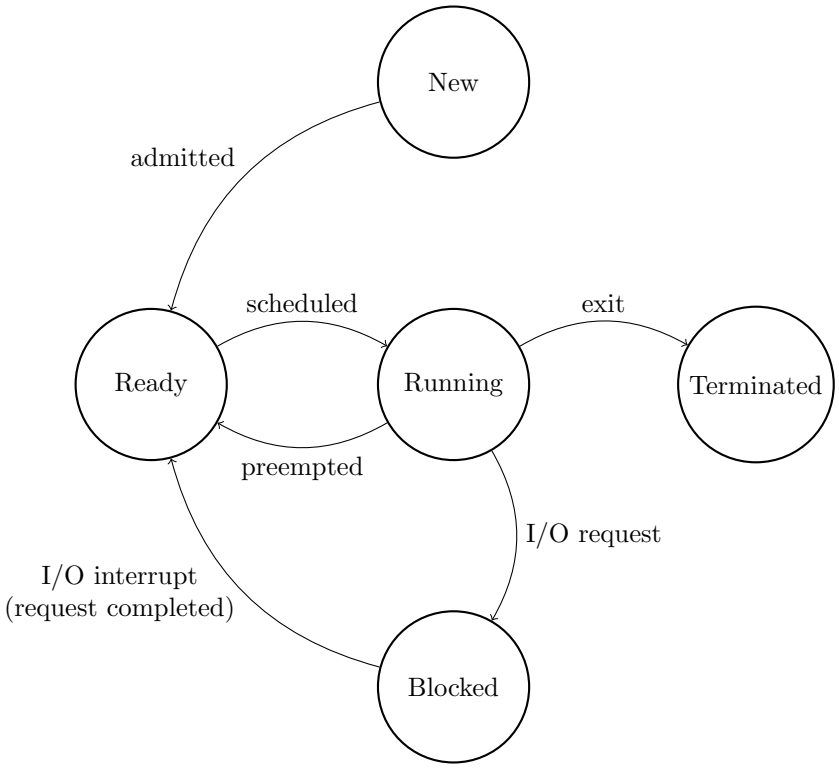


Figure 1.1: The finite state machine representing the states in which a process can be.

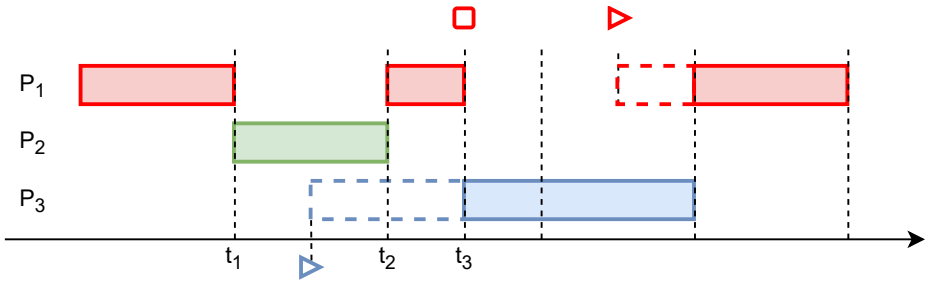


Figure 1.2: The scheduling time sequence of three processes in a general purpose operating system. The square symbol is used to indicate that a process starts waiting for an event. The triangle symbol indicates that the waited event happens.

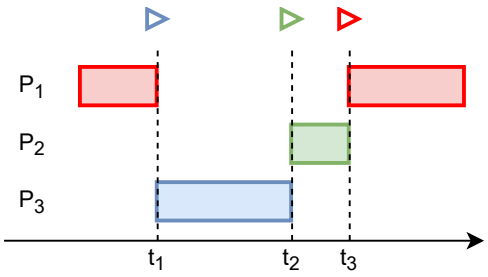


Figure 1.3: The scheduling time sequence of three processes in a real time operating system. The triangle symbol indicates that the waited event happens.

Part II

Linux processes

Chapter 2

Tasks

2.1 Tasks description

In the Linux operating system, tasks are

- A **program** and,
- a **collection of additional data**, like the program counter, the content of the stack and the register's information. In particular, each task has
  - One program counter.
  - Two stacks, one for the user space and one for the kernel space.
  - A set of registers.
  - An address space (which is optional).

In a nutshell, tasks are the basic object of activity in Linux.

In Linux, we use the word task to indicate both processes and threads. Let us highlight the difference between these words. In particular, we will use a hierarchical (like in Object Oriented Programming) description.

- A **task** is a superclass which embodies, as subclasses, processes and threads.
- A **process** is a task which doesn't share its memory with other processes. A process has its own program counter and stacks. A process can be made of one or more threads.
- A **thread** is a task which shares memory with other threads. Still, each thread has its own program counter and stack. Note that, only threads of the same process can share memory. This means that a thread of process P_1 can't share its memory with a thread of process P_2.

To make things clearer, let us consider Figure 2.1 which represents a process with its threads. As we can see, the threads share the memory (`.bss`, `.txt`, `heap`, `.data`) but they have their own stack and registers.

As we have seen, each task has two stacks. The reason is that a program can run in two different modes:

- **User mode**. In user mode, the program runs normally its instructions.

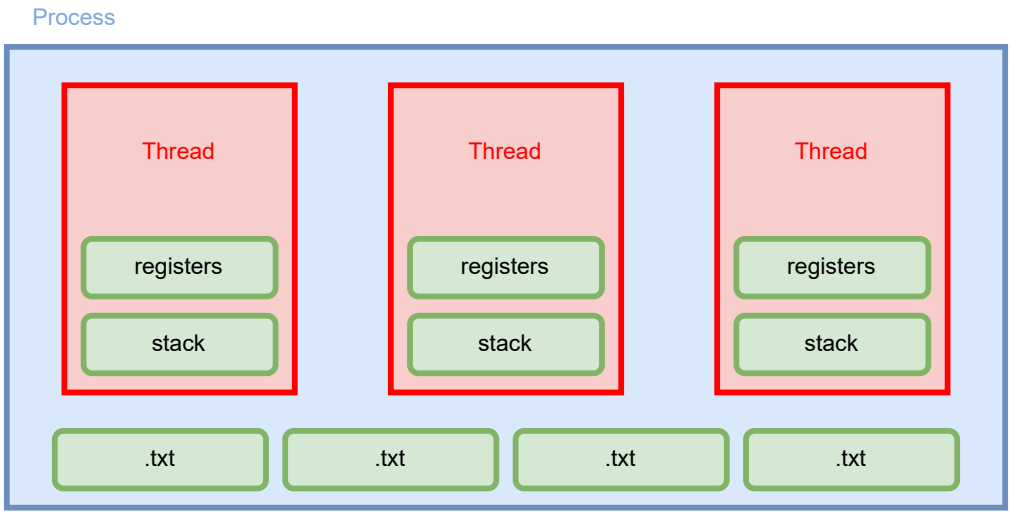


Figure 2.1: A process and its threads.

- **Kernel mode**. In some cases, a program might have to use some privileged instructions (i.e., instructions of the kernel) or it has to access some resources which are reserved to the kernel. In such cases, the program has to switch to kernel mode and use completely the Linux kernel.

2.1.1 Task Control Block

Each task has a Task Control Block which contains all the information about the task itself. In particular, Linux uses a `task_struct` structure which contains, among other things:

- The **state** of the task. A task can be **R**unning, **I**nterrupted, **U**, **D**.
- The **PID** (process id).
- A **pointer to the parent task** (i.e., the task which, using a fork created the task we are considering). Remember that tasks have a hierarchical structure and only the `init` task has no parent.
- A pointer to a **structure that describes the virtual memory address space**.
- A **pointer to the** `fs_struct` that contains the current directory and the directory in which the task has been created (i.e., the root directory).
- A **pointer to the list of open files** used by the task.
- A **pointer to a** `signal_struct` that contains the information about the signals used by the task.
- A `thread_struct` that saves the stack pointer, the registers and the callee saved registers during a context switch.

A shortened version of the `task_struct` is shown in Listing 2.1. The same structure is shown in Figure 2.2.

```
struct task_struct {
    unsigned                    int __state;
    struct pid                  *thread_pid;
    struct task_struct __rcu    *parent;
    struct mm_struct            *mm;
    struct fs_struct            *fs;
    struct files_struct         *files;
    struct signal_struct        *signal;
    struct thread_struct        thread;
}
```

Listing 2.1: The Process Control Block in Linux.

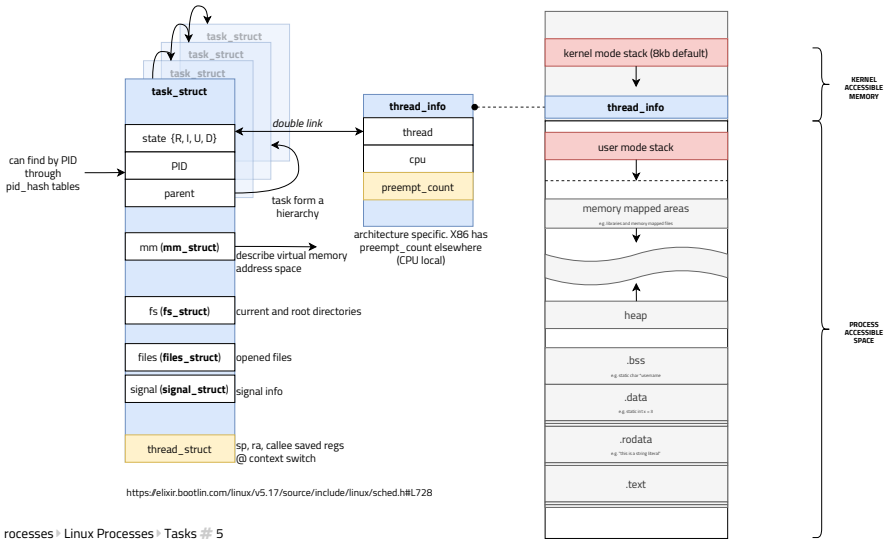


Figure 2.2: A Process Control Block of a task in Linux.

The access to a task's PCB has to be swift and efficient since tasks are continuously scheduled for execution and interrupted. To achieve such a result, a pointer to a task's `task_struct` is usually saved in a register or in a structure, called `thread_info` structure which is saved in the kernel space and contains a pointer to the task which is currently executing.

2.1.2 Task state

One of the information saved in the `task_struct` is the state of a task. A task can be in one of four states:

- **Running**. A task running is represented by the value `TASK_RUNNING`. When a task is running, the instructions of its program are executed by the CPU.
- **Ready but not running**. A task which is ready but not running is also represented by the value `TASK_RUNNING`, as for the running state. This happens because the processor knows who is running, hence it can differentiate between processes which are running or ready.
- **Blocked**. The blocked state can be further divided into two states:

  - **Interruptible**. An interruptible task is represented by the value `TASK_INTERRUPTIBLE`. If a task is in the interruptible state, it can be woken up and put in the ready state upon receiving whatever signal. This means that the task has to specifically check if the signal is the one it's waiting for.
  - **Uninterruptible**. An uninterruptible task is represented by `TASK_UNINTERRUPTIBLE`. If a task is in the uninterruptible state, it only wakes up upon receiving the signal it's waiting for.

  Note that interruptable and uninterruptable refer to the fact that a task can or can't be interrupted while waiting.

After describing each state a task can be in, we have to define the transition model, namely how a process can go from one state to another.

- From the **running** state, a task can:
  - **Terminate**, exiting using the `do_exit` primitive. A terminated task is kept in memory by the OS so that the parent can check it.
  - **Go to the ready state** if it's preempted by another task.
  - **Go to the blocked state** if it has to wait for an event. The task is added to the wait queue.
- From the **ready** state, a task can:
  - **Go to the running state** if the scheduler decides so. When a task transitions to the running state a context switch has to be executed. Namely, the values of the program counter and registers of the scheduled task have to be replaced in the CPU.
- From the **blocked** state, a task can:
  - **Go to the ready state** if the event it was waiting for occurs.

A graphical representation of tasks' states and the transition model is shown in Figure 2.3.

Wait queues

When a task is blocked waiting for an event it's added to a wait queue. More specifically, the OS has a wait queue, represented by a structure, for each event. An event's wait queue stores all the tasks (i.e., all the `task_structs`) that are waiting for such an event. Each entry in the wait queue contains:

- The `task_struct` of the task which is waiting. Tasks in a wait queue are added by calling the primitives `wait_event` or a `wait_event_interruptible` which change the task's state to `TASK_UNINTERRUPTIBLE` or to `TASK_INTERRUPTIBLE`, respectively.
- A function `func` that changes the task's state to `TASK_RUNNING`.
- A flag.

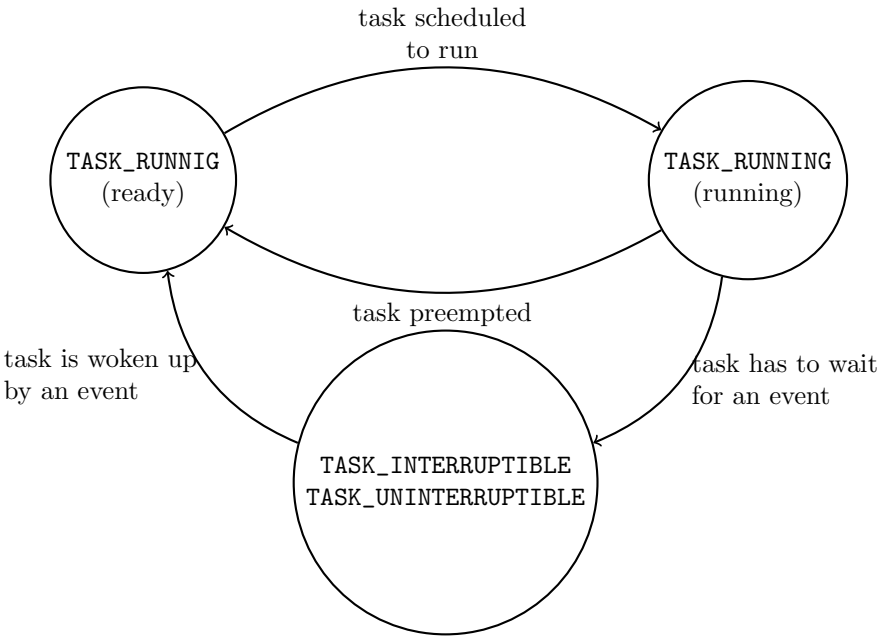


Figure 2.3: The transition model of a task's state.

The queue itself is represented by the `wait_queue_head` which contains the first entry (represented by `wait_queue_entry`) of the queue. Each entry of the queue has a pointer to the next entry of the queue.

When an event E occurs, the kernel calls the `wake_up` function which wakes up every task in the queue of E. In some cases the event can be consumed only by one task, hence every other task should be put back in the blocked state and only one task (the consumer) is put in the ready state. This behaviour wastes a lot of resources and it's called **thundering herd**. To solve the problem, we can divide the tasks in a wait queue into

- **Normal** or **non-exclusive**. Non-exclusive tasks are always woken up.
- **Exclusive**. Exclusive tasks are marked with the `WQ_FLAG_EXCLUSIVE` flag in the `wait_queue_entry` and only the first exclusive task in the queue is woken up. Exclusive tasks are always put at the end of the queue.

Long story short, when an event E occurs:

- All normal tasks in E's wait queue are woken up.
- Only the first exclusive task in E's wait queue is woken up.

```
1 struct wait_queue_entry {
2     unsigned int flags;
3     void *private;
4     wait_queue_func_t func;
5     struct list_head entry;
6 };
7
```

```
8  struct wait_queue_head {
9      spinlock_t lock;
10     struct list_head head;
11 };
12
13 typedef struct wait_queue_head wait_queue_head_t;
14
15 struct task_struct;
```

Listing 2.2: The structures used to handle a wait queue in Linux.

2.2 Task hierarchy

Tasks are ordered with a hierarchical structure. The root task is called `init` and every other task is created by cloning another task (initially, cloning the `init` task). Let us focus on the process of cloning. To clone a task, we have to use the `fork()` function which creates a new copy of the `task_struct` of the parent process. The `fork` is a `libc` function which internally uses the `sys_clone` system call. After cloning the parent task we have two tasks, parent and child, which have identical `task_struct`s. To have two different tasks we have to change PID, signals and resources stored in the `task_struct` of the child. To optimise this process, Linux uses the **copy-on-write** mechanism. The idea is to have a single shared, read-only copy of the pages used by the parent (i.e., the address space is shared). Whenever a page is modified, we create an actual copy of the modified memory page which is exclusively held by the child. This means that we

- Keep the pages which are identical for child and parent as a single shared copy.
- Have two different pages for the parts that have been modified by the child.

2.2.1 Kernel initialisation

When the computer starts:

1. The `start_kernel()` routine (in `init/main.c`) creates the first task with PID 1.
2. The same routine creates another task with PID 0 which runs `cpu_idle`. This program is used to put the processor in low power mode when no other task is running.
3. The task with PID 1 executes the `kernel_init()` routine which mounts the image of the initial (RAM) disk in memory. This will provide necessary modules and commands (e.g., bash).
4. The task with PID 1 executes the command `kernel_execve` to finally run `/bin/init`.

The first task created, namely `init`, has the job of setting up every other task required by the kernel. In particular, there exist two methods:

- `SystemV`. This method uses a single thread to sequentially initiate every task. Note that `SystemV` is a legacy method hence it isn't used anymore.
- `SystemD`. This method is able to parallelise the creation of each task. In particular, to create a task we have to write a file (usually a `.service` file) in which we specify what program to run, after and before what task it should be run and some other information about the task itself.

SystemD

Let us further analyse `systemd`, which is a really important software in Linux since it is used during boot and handles services. `systemd` is a daemon that manages other daemons. In particular, `systemd` is the first daemon to start during booting (and it's assigned pid 1) and the last daemon to terminate during shutdown. `systemd` is configured exclusively via plain-text files. Whenever we want to define a service, we have to write a configuration file, called **unit file**, that uses a declarative language to describe the service. A unit file contains, for instance, after which service a service should be run and what program the service should execute.

Note that `systemd` handles the system and the services, however we can't interact directly with it. Instead, we have to use `systemctl` which allows to start, stop and analyse services.

The main advantage of `systemd` with respect to `systemv` is that it executes elements of its startup sequence in parallel.

2.3 Task scheduling

Scheduling is a fundamental part of an operating system and allows to define what task should be running whenever the running task is preempted. Tasks are taken from the list of ready tasks using a specific policy. We can define the policy for scheduling a task using a scheduling class, i.e., an API which can be used to specify a policy to take a task from the ready queue. The scheduling defines a set of functions which do the most basic operations required during scheduling and allow developers to define their own scheduling algorithm. Table 2.1 contains some functions belonging to Linux 5.7's scheduling class. Scheduling classes can be divided in

- **Normal scheduling policies**.
- **Real time scheduling policies**.

2.3.1 Run queues

The central data structure of the core scheduler that is used to manage active processes is known as run queue. A run queue is stored as a `rq` structure (one for each cpu), which is shown in Listing 2.3.

```
1  struct rq {
2
3    /* runqueue lock: */
4    raw_spinlock_t    __lock;
5
6    /*
7     * nr_running and cpu_load should be in the same cacheline because
8     * remote CPUs use both these fields when doing load calculation.
9     */
10   unsigned int    nr_running;
11
12   struct cfs_rq    cfs;
13   struct rt_rq    rt;
14   struct dl_rq    dl;
15
16   /*
17    * This is part of a global counter where only the total sum
18    * over all CPUs matters. A task can increase this counter on
```

```
19     * one CPU and if it got migrated afterwards it may decrease
20     * it on another CPU. Always updated under the runqueue lock:
21     */
22    unsigned int    nr_uninterruptible;
23
24    struct task_struct __rcu  *curr;
25    struct task_struct  *idle;
26    struct task_struct  *stop;
27    unsigned long   next_balance;
28    struct mm_struct  *prev_mm;
29
30    /* CPU of this runqueue: */
31    int     cpu;
32    int     online;
33 };
```

Listing 2.3: A run queue structure (some fields are omitted).

The main scheduler function (i.e., `schedule()`) is invoked directly in many points in the kernel to allocate the CPU to a process other than the currently active one, e.g., after returning from system calls or interrupts or when a thread does some explicit blocking (e.g., uses mutexes, semaphores or waitqueues). When the `schedule()` function is called, the scheduler:

1. Retrieves the run queue and current task.
2. If the current task was in interruptible sleep and received a signal then the scheduler must promote the task to a runnable task, in any other case the scheduler deactivates the task (this function is implemented by the scheduling class).
3. Gets the next high priority task and clears `TIF_NEED_RESCHED`.
4. Executes the context switch, if needed.

2.3.2 Completely Fair Scheduler

Linux uses the Completely Fair Scheduler. The idea behind the Completely Fair Scheduler (CFS) is to divide a schedule latency τ (usually 6 milliseconds long) in a weighted way among all processes which are ready. Every process is assigned a weight λ_i and the latency τ is divided by the sum of the weights. The result is multiplied by λ_i to find the time task i should run. Formally, the time slice of task p is obtained as

$$\tau_p = \max\left(\lambda_p \frac{\tau}{\sum \lambda_i}, \mu\right)$$

where

- λ_i are the weights associated to tasks.
- τ is the latency.
- μ is the minimum granularity. This parameter can be configured and has a default value of 0.75 ms.

Say for instance we have three processes P_1, P_2 and P_3 with weights $\lambda_1 = 2$, $\lambda_2 = 1$, $\lambda_3 = 1$, respectively. The sum of weights is

$$\lambda_1 + \lambda_2 + \lambda_3 = 4$$

and if we divide τ for the number we just obtained we get the basic unit of time.

$$\frac{\tau}{\lambda_1 + \lambda_2 + \lambda_3} = \frac{6}{4} = 1.5 \text{ ms}$$

To find out how much execution time is assigned to each process we have to multiply the unit time by the weight. Namely, process 1 gets $1.5 \cdot \lambda_1 = 3$ ms and processes 2 and 3 get $1.5 \cdot \lambda_2 = 1.5$ ms.

Virtual runtime

Applying this algorithm, CFS tries to be as fair as possible. To improve fairness, CFS uses virtual runtime, which is a measurement of the time a task has run. Virtual runtime is used to understand which task has been left behind (a small virtual runtime means a task has had small slices of τ).

Control groups

Another way to improve fairness is by using control groups. Control groups allow categorising resources, called controllers. In particular, each controller is divided into groups and each task is assigned to a group. This technique allows to:

- Isolate core workload from background resource needs.
- Handle web server and system processes (eg. Chef, metric collection, etc).
- Handle time-critical work against long-term asynchronous jobs.
- Don't allow one workload to overpower the others.

Load balancing on multiple cores

When a machine has multiple cores, applying fair scheduling and balancing load on the cores can become tricky, in fact we can't:

- Balance on the same number of threads because threads might have different nature. Namely, one processor might be executing high priority threads while the other low-priority ones. If we have the same number of threads in both processors' queues we execute in parallel high priority threads with low priority ones, leaving some high priority threads in the queue.
- Balance on the sum of each queue q's task weights, $\lambda_q = \sum_i \lambda_{i,q}$. Say a processor has only one high-priority thread which sleeps for a long time. This core would have to frequently steal work from the other core's queue to keep itself busy with sub-optimal performance.

To solve this problem we have to balance over the average load Ω_q of a run queue, where the load of task i in queue q is

$$w_{i,q} = \lambda_{i,q} \cdot \gamma_{i,q}$$

and $\gamma_{i,q}$ is the CPU usage. The operation of balancing is done periodically on a processor. Memory is organised in a hierarchy depending on how close cores are. The leaves of the tree are the cores and cores that share the same parent P_1 are, for instance connected to the same cache, then the parent P_2 of P_1 can reach every core connected to the same memory bank and so on. Usually, at

- Level 0 we have the cores.

- Level 1 we join cores that share functional units and first level caches.
- Level 2 we join cores that share the last level of cache and a memory bank, namely, the cores in the same NUMA node.
- Level 3 we join cores one NUMA hop away one from the other.
- Level 4 we join all cores in the machine.

The task that has to balance the load walks the memory hierarchy and moves tasks from one core to another whenever it sees some imbalance starting from the bottom of the hierarchy. Namely, it starts checking if it's necessarily to balance load in level 1 nodes and then moves to higher levels.

2.3.3 Deadline Scheduler

Linux's deadline scheduler is used for situations in which the system has to meet some deadlines. It's based on three parameters:

- The **runtime**.
- The **period** between tasks.
- The **deadline**.

These parameters are used by the scheduler to assign a runtime to each task every period making sure that the task terminates before the deadline. This idea is used in Earliest Deadline First scheduling algorithms. Note that the runtime must be estimated and typically the period coincides with the deadline.

Function	Description
`enqueue_task(rq, t)`	Add thread `t` to runqueue `rq`.
`dequeue_task(rq, t)`	Remove thread `t` from runqueue `rq`.
`yield_task(rq)`	Yield the currently running thread on the CPU of `rq`.
`check_preempt_curr(rq, t)`	Check if the currently running thread of `rq` should be preempted by thread `t`.
`pick_next_task(rq)`	Return the next thread that should run on `rq`.
`put_prev_task(rq, t)`	Remove the currently running thread `t` from the CPU of `rq`.
`set_next_task(rq, t)`	Set thread `t` as the currently running thread on the CPU of `rq`.
`balance(rq)`	Run the load balancing algorithm for the CPU of `rq`.
`select_task_rq(t)`	Choose a new CPU for the waking up/newly created thread `t`.
`task_tick(rq)`	Called at every clock tick on the CPU of `rq` if the currently running thread is in this scheduling class.
`task_fork(t)`	Called when thread `t` is created after a `fork()` or `clone()` system call.
`task_dead(t)`	Called when thread `t` terminates.

Table 2.1: Linux 5.7's scheduling class.

Chapter 3

Synchronisation

As already mentioned, processes don't share memory, hence it's impossible for a process to communicate with another one using memory. Luckily, the operating system gives us ways to let processes communicate.

3.1 Processes

3.1.1 Process creation

The life of a process begins with a `fork` that allows cloning an existing process, called parent, to create a new process, called child. The `fork` function, which internally calls the `sys_clone` system call,

1. Clones the program running in the parent process in a separate address space. Namely, we get two processes with different address spaces but the same program (i.e., the same code).
2. Starts executing the program from the instruction after the call to `fork`.
3. Returns the PID of the process created to the parent and 0 to the child. If the child wants to check the parent id, it can use the `getppid()` function.

Note that parent and child processes can't communicate, even if one is the parent of the other and they both execute the same code. An example of usage of the fork function is shown in Listing 3.1. If we execute such a program multiple times, we notice that there is no specific order in which the two processes should execute.

```
1  #include <unistd.h>
2  #include <stdio.h>
3
4  int main() {
5      printf("Starting with PID %d!\n", getpid());
6
7      int pid = fork();
8      printf("Both prints this message\n");
9
10     if (ret == 0) {
11         // Child
12         printf("I'm the child[%d] and my parent is %d!\n", getpid(), getppid());
```

```
13      } else {
14          // Parent
15          printf("I'm the parent[%d] and the child has pid %d!\n", getpid(), ret);
16      }
17
18      return 0;
19  }
```

Listing 3.1: Usage of the fork function.

The library `sys/types.h` contains the type `pid_t` (32 bit integer) used to save a PID. The number of processes is a limited value (contained in `/proc/sys/kernel/pid_max`) and if such a number is reached, the fork returns -1 indicating that it couldn't create a new child.

3.1.2 Executing a program

When a process calls the `fork` function, a new process is created, however, the child executes the same program as the parent. To change the program executed by a process we have to use one of the functions of the `exec` family. If we weren't able to change the program in execution, the `init` process wouldn't be able to create all the other processes. Consider for instance a shell, which is a program that can execute other programs. A shell is in execution in a process with its own PID. When we want to execute a new program we type the name of the program and:

1. The shell calls the `fork` function to create a new process (i.e., to get a new address space).
2. The child shell calls the `execve` function and replaces the shell program with the program the user wants to run.

The `exec` function comes in different flavours. In particular, depending on the letters after the word `exec`, we get different behaviours:

- If the letter **l** is present, the `exec` function accepts a list of NULL-terminated parameters.
- If the letter **v** is present, the `exec` function accepts an array of NULL-terminated strings.
- If the letter **p** is present, the `exec` function searches the program to execute using the PATH environment variable.
- If the letter **e** is present, the `exec` function allows to specify new environment variables.

3.2 Process communication

Among all libraries that offer functions that allow tasks to communicate, the POSIX library is by far the most used, hence we will refer to it.

3.2.1 Waiting for the child

The most basic method to synchronise a parent process with its child is the `wait()` function, which internally calls a system call. The `wait` function stops the execution of the current process waiting for one child to terminate (or change its state). When using the `wait` function, we can either wait for the first child that terminates or specify, using the `waitpid(pid_t pid)` function the process id of the process to wait for.

The wait function is fundamental for avoiding zombies. A zombie is a process that is consuming resources but that the parent forgot to sync with. When the child terminates, if its parent doesn't synchronise correctly, it (the child) is dead but it's still there occupying resources. When a child ends its execution, its state is changed to zombie waiting for the parent to collect it. If the parent doesn't collect it, using a `wait`, the process remains a zombie.

3.2.2 Signal communication

The `wait` function can be used only to synchronise a parent with its child, however, we would like also arbitrary processes to communicate. The most basic ways processes can use to communicate is by using signals. A signal

- Is **unidirectional** since it's sent from a source to a destination but there is no answer.
- Involves **no data transfer**, in fact all the information is embedded in the type of the signal itself.
- Provides **no way of synchronising**.
- Is **exchanged only between processes** but not between threads. More specifically, when a process receives a signal, all threads of that process are effected.
- Can be sent by the operating system or by a process.

Some examples of signals are:

- `SIGCHLD` sent by a terminating process to its parent.
- `SIGINT` sent by the operating system when we press Ctrl+C on the keyboard.
- `SIGILL` sent by the operating system when a process wants to execute an illegal instruction.
- `SIGSEGV` sent by the operating system when a Segmentation Fault verifies (e.g., a program tries to access a reserved memory area).

Sending a signal

A process can send a signal using the

```
int
kill(pid_t pid, int sig);
```

function and specifying

- The PID of the process we want to send the signal to.
- The identifier of the signal.

The same function can also be invoked from the command line.

Receiving a signal

When a process receives a signal, it can specify what to do. In particular, a process can define a signal handler for each type of signal, which overrides the default behaviour. A process can define a signal handler using the function

```
int
sigaction(int sig, const struct sigaction *restrict act, struct sigaction *restrict
    oact);
```

which requires:

- The identifier of the signal for which we want to define the action.
- A pointer to a `sigaction` structure which contains a pointer to the function that should be executed.
- A pointer to a `sigaction` structure which is used to save the old action associated to the `sig` signal.

The `sigaction` structure is defined as in Listing 3.2 and contains:

- The pointer to the main handler.
- The pointer to an alternative handler.
- A mask used to block some signals. Basically, we say that we don't want to handle a specific signal since we don't want to be interrupted.
- Flags to specify various options.

```
1  struct  sigaction {
2      union __sigaction_u __sigaction_u;  /* signal handler */
3      sigset_t sa_mask;                   /* signal mask to apply */
4      int      sa_flags;                  /* see signal options below */
5  };
6
7  union __sigaction_u {
8      void    (*__sa_handler)(int);                          /* primary handler */
9      void    (*__sa_sigaction)(int, siginfo_t *, void *);   /* alternative handler
       */
10 };
```

Listing 3.2: The structure used to define an action.

Listings 3.3 and 3.4 contain two processes, the first being the sender of a signal and the second being the receiver which defines a custom handler for the signal sent by the former process.

```
1  #include <signal.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <sys/types.h>
5  #include <unistd.h>
6
7
8  int main(int argc, char *argv[]) {
9
10     if (argc < 2) {
11         printf("Ehi! Give me a pid!\n");
```

```
12          return 1;
13      }
14
15      pid_t target_pid = atoi(argv[1]);
16
17      kill(target_pid, SIGUSR1);
18
19      return 0;
20  }
```

Listing 3.3: A process sending a signal.

```
1  #include <signal.h>
2  #include <stdio.h>
3  #include <string.h>
4  #include <sys/types.h>
5  #include <unistd.h>
6
7  void my_handler(int sig_num) {
8      printf("Received %d\n", sig_num);
9  }
10
11 int main() {
12     struct sigaction sa;
13     memset(&sa, 0, sizeof(sa));
14
15     sa.sa_handler = &my_handler;
16     sigaction(SIGUSR1, &sa, NULL);
17
18     while(1) { sleep(1); }
19     return 0;
20 }
```

Listing 3.4: A process sending a signal.

Synchronous handler

Defining handlers, allows processes to handle signals asynchronously. This means that a process keeps executing instructions, until a signal is received and the respective handler is executed. In other cases, one might want to synchronously wait for a signal, namely, stop its execution until a specific signal is received. The POSIX library offers three different functions:

- `sigqueue` is used to send a queued signal.
- `sigwaitinfo` is used to synchronously wait a signal.
- `sigtimedwait` is used to synchronously wait a signal for a given time.

Masking signals

In some cases we don't want the process to handle a specific signal since we don't want the execution to be interrupted. The function

```
int
sigprocmask(int how, const sigset_t *restrict set, sigset_t *restrict oset);
```

allows to change the mask used to block signals. In particular, we have to specify:

- If we want to add add a signal to the mask (`how=SIG_BLOCK`), remove a signal from the mask (`how=SIG_UNBLOCK`) or replace the mask (`how=SIG_SETMASK`).
- The set of signals to add, remove or set.
- A variable where the previous set of signals is saved.

Signal library function

When working with signals, it's important to use the library functions offered by POSIX to handle the structures used to handle signals. For instance, if we have a `sigset_t` structure and we want to initialise it, we should use `sigemptyset` instead of using `memset`, since the former is more specific to the structure we are considering.

3.2.3 Pipes

Signals are not made for exchanging data, thus we have to use other technologies if we want to exchange some data. Pipes are the easiest way to achieve this goal. Pipes:

- Are FIFOs.
- Offer implicit synchronisation. Namely, multiple processes can safely read and write the same pipe.

Unnamed pipes

A pipe involves

- A **producer** that adds data to the tail of the pipe.
- A **consumer** that consumes (i.e., removes) data from the head of the pipe.

The first thing we have to do to use an unnamed pipe is to create it. In particular, we can use the function

```
int
pipe(int fildes[2]);
```

that requires an array of two file descriptors and allocates

- The read end of the pipe in the file descriptor in position 0.
- The write end of the pipe in the file descriptor in position 1.

Data written to `fildes[1]` appears on (i.e., can be read from) `fildes[0]`. This allows the output of one program to be sent to another program: the source's standard output is set up to be the write end of the pipe; the sink's standard input is set up to be the read end of the pipe. The pipe itself persists until all of its associated descriptors are closed.

Once we have obtained the write and read end of a pipe, we can:

- Use the `write` and `read` functions, using directly `fildes[0]` and `fildes[1]`.
- Transform the file descriptor in a file pointer (`FILE *`), using `fdopen`, and use C's library function for reading and writing files.

An example of a program that creates a child and uses a pipe to communicate with it is shown in Listing 3.5.

```c
#include <fcntl.h>
#include <math.h>
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>

int main() {

    float number;
    printf("Please insert a number: ");
    scanf("%f", &number);

    int fd[2];
    pipe(fd);

    int id = fork();
    if (id == 0) {
        // Child
        close(fd[0]);
        float result = sqrt(number);
        write(fd[1], &result, sizeof(result));
        close(fd[1]);

    } else {
        // Parent
        close(fd[1]);
        float result = sqrt(number);
        float result_from_child;
        read(fd[0], &result_from_child, sizeof(result_from_child));
        close(fd[0]);
        wait(NULL);

        if (result_from_child == result) {
            printf("sqrt(%f)=%f\n", number, result);
        } else {
            printf("Incorrect computation? %f != %f\n", result_from_child, result);
        }

    }

    return 0;
}
```

Listing 3.5: A program in which a parent communicates with its child with a pipe.

Named pipes

Pipes can also be associated with a name so that unrelated processes can talk. One process can create a pipe with a name and then other processes can retrieve that pipe, if they know its name. Named pipes are sometimes called FIFO pipes (not that unnamed pipes aren't FIFO, it's just a way of calling them). When a named pipe is created, a special file, used as pipe, is created. The function used to create the pipe is

```c
int
mkfifo(const char *path, mode_t mode);
```

which requires:

- The path and filename of the file to create. This path can be used by any process to retrieve the FIFO.
- The file permissions for the pipe file (remember: 1 for execution, 2 for writing, 4 for reading).

Once the file has been created, we can use the `open` function to open the file (passing the same path and filename we passed to the `mkfifo` function) and the `read` and `write` functions to interact with the queue. An example of a program that writes to the pipe and another that reads from it is shown in Listings 3.6 and 3.7. When a program doesn't require the pipe anymore, it has to use the `unlink` function, passing the filename to it, which removes a reference to the pipe file (and eventually deletes it if the number of references is 0).

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main () {

    const char * myfifo = "/tmp/myfifo";
    if (mkfifo(myfifo, 0666) != 0) {
        printf("Cannot create fifo.\n");
        return 1;
    }

    int fd = open(myfifo, O_WRONLY);
    if (fd <= 0) {
        printf("Cannot open fifo\n");
        unlink(myfifo);
        return 1;
    }

    float data = 10;
    int nb = write(fd, &data, sizeof(data));
    if (nb == 0) {
        printf("Write error\n");
    }

    printf("OK\n");

    close(fd);
    unlink(myfifo);
    return 0;
}
```

Listing 3.6: A program that writes to a named pipe.

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>

int main () {

    const char * myfifo = "/tmp/myfifo";
```

```
10      int fd = open(myfifo, O_RDONLY);
11      if (fd <= 0) {
12          printf("Cannot open fifo\n");
13          return 1;
14      }
15
16      float data;
17      int nb = read(fd, &data, sizeof(data));
18      if (nb == 0) {
19          printf("Read error\n");
20          return 1;
21      }
22
23      printf("Read: %f\n", data);
24
25      close(fd);
26      return 0;
27  }
```

Listing 3.7: A program that writes to a named pipe.

3.2.4 Messages

Even if pipes implicitly handle synchronicity, they aren't well suited for working with multiple consumers and producers. Moreover, pipes don't have the concept of priority. We can use messages to solve the aforementioned problems. More precisely, we can use a message queue to exchange prioritised messages among multiple writers and readers. A message queue is a queue with priority, this means that messages with an higher priority are delivered before messages with lower priority. Moreover, readers can also filter messages by priority (e.g., receive messages only with a certain priority).

As for named pipes, we can create a message queue with a name so that multiple unrelated processes can use the same message queue. Creating a queue means creating a file in the directory `/dev/mqueue/`. Moreover, the internal state of the file, hence the internal state of the queue, can be observed.

The first step for using a message queue is to create it. In particular, we can use the function

```
mqd_t
mq_open(const char *name, int oflag, mode_t mode, struct mq_attr *attr);
```

which requires

- The name of the queue, starting with `/`.
- A flag that specifies the mode in which the file should be open (write-only, read-only, ...).
- The file permission to give to the file created.
- Some attributes.

and returns a message queue descriptor, as a `mqd_t` structure, which can be used to use the queue.

After obtaining a reference to a message queue we can use

- The function

  ```
  int
  mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len, unsigned int
  msg_prio);
  ```

to send a message `msg_ptr` of length `msg_len` (bytes) with priority `msg_prio` on queue `mqdes`. The function

```
int
mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len, unsigned int *
msg_prio);
```

to get a message of length `msg_len` from queue `mqdes`. The message and its priority are saved in `msg_ptr` and `msg_prio`, respectively.

Note that messages with higher priority have a bigger priority number and the maximum priority number is implementation dependent (but there must be at least 32 levels, from 0 onward). Messages are ordered, with some logic, using priority and messages with the same priority are delivered in FIFO order.

When a program doesn't require a message queue anymore, it can use:

- The function

```
int mq_close(mqd_t mqdes);
```

which closes the queue only for the caller. The queue is still there.

- The function

```
int mq_unlink(const char* name);
```

which closes and destroys the queue.

An example of two programs, one writer and one reader, using messages for communication is shown in Listing 3.8 and 3.9.

```
1  #include <errno.h>
2  #include <mqueue.h>
3  #include <fcntl.h>
4  #include <stdlib.h>
5  #include <stdio.h>
6  #include <string.h>
7  #include <sys/stat.h>
8  #define MSG_SIZE    100
9  #define MSGQUEUE_NAME  "/bazinga"
10 #define NR_MESSAGES 3
11
12 char messages[3][MSG_SIZE];
13
14 unsigned int priorities[] = { 3, 0, 1 };
15
16 int main()
17 {
18     strcpy(messages[0], "Scissors Cuts Paper.");
19     strcpy(messages[1], "Rock Crushes Lizard.");
20     strcpy(messages[2], "Paper Covers Rock.");
21
22     struct mq_attr attr;
23     attr.mq_maxmsg = 10;
24     // maxvalue: cat /proc/sys/fs/mqueue/queues_max
25     attr.mq_msgsize = MSG_SIZE; // maxvalue: cat /proc/sys/fs/mqueue/msgsize_max
```

```
26      attr.mq_flags = 0;
27      attr.mq_curmsgs = 0;
28
29      mqd_t mqd = mq_open(MSGQUEUE_NAME, O_CREAT | O_WRONLY, 0664, &attr);
30      if (mqd < 0) {
31          printf("Error - open\n");
32          return 1;
33      }
34      for (int i = 0; i < NR_MESSAGES; ++i) {
35          printf("%s\n", messages[i]);
36          ssize_t ret = mq_send(mqd, messages[i], MSG_SIZE, priorities[i]);
37          if (ret < 0) {
38              printf("Error - send\n");
39              return 1;
40          }
41          printf("Sent a message with priority=%d\n", priorities[i]);
42      }
43      mq_close(mqd);
44      return 0;
45  }
```

Listing 3.8: A program that communicates using messages.

```
1  #include <errno.h>
2  #include <mqueue.h>
3  #include <fcntl.h>
4  #include <stdlib.h>
5  #include <stdio.h>
6  #include <string.h>
7  #include <sys/stat.h>
8  #define MSG_SIZE 100
9  #define MSGQUEUE_NAME "/bazinga"
10
11
12 int main()
13 {
14      char buf[MSG_SIZE];
15      unsigned int prio;
16      mqd_t mqd = mq_open(MSGQUEUE_NAME, O_RDONLY, 0664, NULL);
17      if (mqd < 0) {
18          printf ("error - mq_open()");
19          return 1;
20      }
21      struct mq_attr attr;
22      mq_getattr(mqd, &attr);
23
24      printf ("Nr. messages in the queue: %ld / %ld [maxsize=%ld]\n",
25          attr.mq_curmsgs,
26          attr.mq_maxmsg,
27          attr.mq_msgsize);
28
29      while (attr.mq_curmsgs != 0) {
30          ssize_t size = mq_receive(mqd, buf, MSG_SIZE, &prio);
31          if (size < 0) {
32              printf("main - mq_receive()");
33              return 1;
34          }
35          printf("Received a message len=%ld with priority=%d: %s\n",
36              size, prio, buf);
37
```

```
38          mq_getattr (mqd, &attr);
39      }
40      printf("No messages left in the queue.\n");
41      mq_close(mqd);
42      return 0;
43
44  }
```

Listing 3.9: A program that communicates using messages.

3.2.5 Shared memory

Messages are really good ways for communicate, however they are good only for exchanging messages with a limited length. If we were to exchange a larger amount of data, we should use other techniques. In particular, we can map a virtual memory area to the same physical memory area so that multiple processes can write and read on the same memory. Namely, we want to ask the OS to create a new set of physical pages, to which more processes can map their virtual pages.

The first thing we have to do is creating a sharing memory area by using the function

```
int
shm_open(const char *name, int oflag, ...);
```

which requires:

- A name, starting with `/`. The name is used to create a special file in `/dev/shm/`.
- A flag that specifies the mode for opening the file (read-only, write-only, ...).
- The file permissions of the file created.

The `shm_open` returns the file descriptor of the shared memory. Once the file has been created, we have to specify its size using the function

```
int
ftruncate(int fd, off_t length);
```

that truncates the file described by `fp` to `length` bytes. This process has to be done only by the process that initially creates the shared memory, while the other processes that use the same shared area don't have to create and truncate the file.

After creating the shared memory, we have to map it to the virtual address space. Namely, the function

```
void *
mmap(void *addr, size_t len, int prot, int flags, int fd, off_t offset);
```

places the memory represented by the file descriptor `fd` to the address `addr` (or to an address decided by the function if its value is `NULL`). The `mmap` returns the address of the virtual memory area, which can be used as a normal pointer. More precisely, the `mmap` causes the pages starting at `addr` and continuing for at most `len` bytes to be mapped from the object described by `fd`, starting at byte offset `offset`. If `offset` or `len` is not a multiple of the page size, the mapped region may extend past the specified range. Any extension beyond the end of the mapped object will be zero-filled. Note that, if we want the memory area to be available to everyone, we have to use the `SHARED` flag. The `SHARED` flag makes updates visible to other processes and carried out through the underlying file (i.e., the one described by `fd`).

When a process doesn't require a memory area anymore it has to use:

- The function

```
int
munmap(void *addr, size_t len);
```

  to delete the mapping for a specific address space.

- The function

```
int
shm_unlink(const char *name);
```

  to remove the `shm` object created by the `shm_open`.

3.3 Process synchronisation

Scheduling is unpredictable, hence we have to find ways to control the order in which some instructions are executed. Among all libraries that offer functions for synchronising tasks in Linux, the POSIX library is by far the most used, hence we will refer to it.

3.3.1 Semaphores

Among all techniques used to synchronise processes, semaphores are the most basic ones. A semaphore is a counter where

- 0 represents a red and the process has to wait.
- A number greater than 0 represents a green semaphore and the process can continue its execution.

As always, to work with semaphores, we have to create them. In particular, the function

```
int
sem_init(sem_t *sem, int pshared, unsigned int value);
```

creates allocates a semaphore in the pointer `sem` with initial value `value`. If the semaphore can be shared among different threads, `pshared` has to be 0, otherwise it has to be whatever number different from 0. On the other hand, we can use the function

```
int
sem_destroy(sem_t *sem);
```

to destroy the semaphore `sem`.

We can also create semaphores with a name so that they can be used by different processes using the function

```
sem_t *
sem_open(const char *name, int oflags);
```

Once we have obtained a pointer to a semaphore, we can interact with the semaphore using two atomic functions (i.e., that are executed without being interrupted by other processes):

- The `wait` function family, that includes:
  - `int sem_wait(sem_t *sem);`

  - `int sem_trywait(sem_t *sem);`
  - `int sem_timedwait(sem_t *sem, const struct timespec *timeout);`

  and waits for the semaphore to be greater than 0. When the semaphore is greater than 0, the process continues and the semaphore is decreased.

- The function

```
int
sem_post(sem_t *sem);
```

  that increments the value of the semaphore pointed by `sem`.

Chapter 4

Task scheduling

4.1 Problem model

4.1.1 Task lifecycle

To explore the world of task scheduling, we can start by analysing what is the life cycle of a task. The lifecycle of a task is made of the following steps:

1. At some point in time a task is created and **activated**.
2. Upon activating, a task is added to a ready queue where it waits to be executed.
3. Some time in the future, the operating system will decide to execute the task, hence the task is **dispatched**.
4. When the task is in execution, it can **terminate** or **be preempted** and sent back to the waiting queue.

Let us pause for a moment and clarify again what does preemption mean. When a task is running, the operating system can decide (according to some logic and some policies) replace the running task by

1. Suspending the running task.
2. Saving the status (i.e., registers, program counter, ...) of the task that was running.
3. Picking the next task to execute.
4. Loading the status of the picked task.
5. Starting the execution of the new task.

This procedure is called **context switch**.

Preemption isn't the only way a task can be replaced, in fact a task can block itself even when it has to wait some event or resource. When a task is blocked waiting for something, it's added to a wait queue and a new task is picked for execution.

The life cycle we have just described is represented in Figure 4.1.

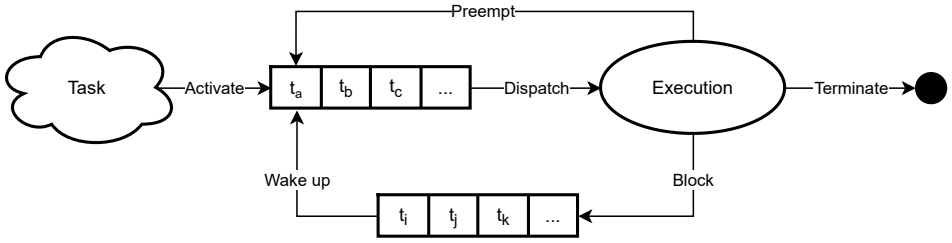


Figure 4.1: The life cycle of a task.

4.1.2 Model

To understand what are the techniques used to schedule tasks, we have to model the problem first.

Task model

First we have to model a task, hence defining the **task model**. Let us consider a set of tasks

$$T = \{\tau_1, \tau_2, \ldots, \tau_i, \ldots, \tau_n\}$$

each of which has its own set of properties. If we consider a task τ_i, we can define:

- The **arrival time** a_i as the time instant in which the task is ready for execution and it's put into the ready queue.
- The **start time** s_i as the time instant in which the task starts its execution for the first time. In other words, it's the time at which the processor is assigned the task to be executed.
- The **finish time** f_i as the time instant in which the task is completed and terminated.
- The **waiting time** $W_i = s_i - a_i$ as the time spent waiting in the ready queue (i.e., before being scheduled).
- The **computation time** C_i as the time spent executing. Note that in general, this time isn't the difference between f_i and s_i because a task may be preempted by other tasks or may block itself waiting for some event. This means that in general C_i is greater or equal to $f_i - s_i$.
- The **turnaround time** $Z_i = f_i - a_i$ as the time that elapsed from when the process is added to the wait queue, to when it's completed.
- The **response time** R_i is the time between the arrival time and the instant at which the process is preempted for the first time.

Boundness

Let us focus on the time in which the task is in execution. Tasks can be classified by a parameter, called boundness, which tells where does a task spend its time. Tasks can be:

- **CPU-bound**. A CPU-bound task spends its time mostly in execution. For CPU-bound tasks, the turnaround time can be approximated with the sum of waiting and computation time (if we neglect preemptions). Namely,

$$Z_i \approx W_i + C_i$$

- **I/O-bound**. An I/O-bound task is waiting for some input/output operations most of the time. This is the case, for instance, of text editors. In this case, the turnaround time is much bigger than the sum of waiting and computation time, since after s_i, the task is blocked during most of the time that elapses until its termination. Namely,

$$Z_i \gg W_i + C_i$$

Platform model

After introducing the model of a single task, we want to define the machine on which tasks have to run. We can see a computing system as a machine made of

- m processing elements, i.e., CPUs

$$CPU = \{CPU_1, CPU_2, \dots, CPU_m\}$$

- s additional resources, like registers (but not only),

$$R = \{R_1, R_2, \dots, R_s\}$$

We will use the notation

$$A_{cpu}(CPU_k, t) = \tau_i$$

to say that CPU_k at time instant t is executing task τ_i. This means that a task τ_i can be in execution at time t only if it exists a CPU CPU_k which has been assigned to τ_i at time t, namely if

$$\exists CPU_k : A_{cpu}(CPU_k, t) = \tau_i$$

If no task is in execution we replace τ_i with $\emptyset$. A similar notation can be used to say which tasks are using a certain resource R_k. Since some resources can be shared, the function A_{res} returns a set of tasks, which represents the tasks that are sharing the resource at time t. Namely,

$$A_{res}(R_k, t) = \{\tau_i, \tau_j, \dots\}$$

For those resources which are exclusive, we have to impose that the cardinality of $A_{res}(\cdot, \cdot)$ is less or equal to 1. As before, we can say that a task τ_i can be executed at time t only if

$$\tau_i \in A_{res}(R_k, t)$$

for all resources R_k required to run task τ_i.

4.2 Scheduling

Now that we know the model we'll use, we have to state the scheduling problem. Given

- A set of n tasks $T = \{\tau_1, \tau_2, \dots, \tau_n\}$.
- A set of m processing elements $CPU = \{CPU_1, CPU_2, \dots, CPU_m\}$.
- A set of s resources $R = \{R_1, R_2, \dots, R_s\}$

- Optionally, a set of precedence relationship between tasks, specifying which task has to be executed before another.

we want to compute an optimal schedule, i.e., an order of execution of the tasks and an allocation to resources and processors, such that the constraints are satisfied and each process is executed only if all resources it needs are available. This problem is usually NP-complete, hence it has to be reduced to easier problems. We can also used heuristics or algorithms which are optimal only for specific scenarios.

4.2.1 Optimisation metrics

When solving an optimisation problem we have to decide a metric to optimise. Schedulers can optimise different metrics and, depending on the field of application, one metric is better than the other. The most common metrics are:

- **Processor utilisation**. The scheduler tries to make processors as busy as possible.
- **Throughput**. The scheduler tries to maximise the number of tasks completing their execution time, per time unit.
- **Waiting time**. The scheduler tries to minimise the time spent in the ready queue.
- **Fairness**. The scheduler tries to schedule tasks trying not to leave tasks behind.
- **Overhead**. The scheduler tries to minimise the time spent picking the next process to schedule and doing context-switches.

Usually, either we optimise only one metric or we have to trade-off among multiple metrics since it's not possible to optimise multiple metrics altogether.

When one task is never scheduled because all the other tasks are always dispatched before it, we say that we are in a condition of **starvation**. Starvation is never desirable and, independently from the optimisation metric, a scheduler should always avoid starvation.

4.2.2 Algorithm classification

Scheduling algorithms can be classified in different ways. The first distinction we can do in between:

- **Preemptive**. Preemptive algorithms can interrupt the task running and replace it with another (ready) one. Such algorithms have a set of rules which tell when a ready task is preferable to a running task. These type of algorithms are required in responsive systems, namely a system which has to swiftly handle events and replace the running task when an event occurs.
- **Non-preemptive**. Non-preemptive algorithms do not interrupt tasks, which can execute until their completion. For this reason, non-preemptive scheduling algorithms are also called Run-To-Completion algorithms. Non-preemptive algorithms work fine when the system doesn't require responsiveness to events. Furthermore they ensure minimum overhead since, not having to change tasks frequently, the scheduling decision is taken more rarely.

Another way of classifying scheduling algorithms is between:

- **Static**. Static algorithms take their scheduling decisions using only static information about the task (i.e., known before activation). In this case we need strong assumptions about the tasks.
- **Dynamic**. Dynamic algorithms use dynamic information, i.e., that change during the execution of the task. In this case we need to observe the task and collect information about its state (i.e., we need a run-time feedback mechanism), hence increasing the algorithm's overhead.

Modern general purpose operating systems usually mix static and dynamic parameters. Namely, they do some assumptions based on static properties, and then they adjust such assumptions using dynamic properties. Scheduling algorithms can also be divided into:

- **Online**. Online schedulers decide what task has to be executed at runtime and can expect new task to be activated during the lifetime of the system.
- **Offline**. Offline schedulers decide the order of tasks in advance, namely before their activation. For this reason, offline schedulers are also static. An example could be scheduling the tasks required to launch a spaceship. Offline schedulers output the sequence of task to execute, called schedule.

Finally, we can distinguish scheduling algorithm which are

- **Optimal**. Optimal algorithms try to schedule tasks optimising a specific metric. Such algorithms might be very resource-intensive hence requiring a lot of overhead.
- **Heuristic**. Heuristic algorithms try to optimise the scheduling process using some heuristics. The result tends to an optimal result, but without guarantees of being really optimal. On the plus side, these algorithms are much faster than optimal ones (i.e., they have a smaller overhead).

4.2.3 Scheduling algorithms

Scheduling algorithm have to choose the next task (or the sequence of tasks) to execute depending on some policy. Some of the best known policies are:

- **First-In-First-Out** (FIFO).
- **Shortest Job First** (SJF).
- **Shortest Remaining Time First** (SRTF).
- **Highest Response Ratio Next** (HRRN).
- **Round Robin** (RR).

First-In-First-Out

The FIFO algorithm is a **non-preemptive** scheduling algorithm which schedules tasks in the same order in which they are activated. For this reason, this algorithm is also called First Come First Served. The main advantages of this algorithm are:

- It's **very simple**, which translates in a very small overhead.
- It **doesn't require any knowledge** of the tasks that have to be scheduled.

The main disadvantages are:

- It's **not good for responsiveness**, in fact we have no guarantees regarding when a task will be executed. Moreover, as we have already said, non-preemptive algorithms don't fit well with responsive systems.
- **Starvation**. Long tasks can use most of the time the processor (since they are not preempted) and small tasks might starve waiting for a long task to terminate.

Shortest Job First

Shortest Job First is a **non-preemptive** scheduling algorithm that schedules (i.e., picks for execution) the task that requires the smallest computation time. The main advantage of this algorithm is:

- It's **optimal among non-preemptive algorithms**, with respect to the minimisation of the average waiting time.

The main disadvantages are:

- **Starvation**, in fact tasks that require a long time could be left behind.
- The **scheduler needs to know the computation time** C_i in advance.

Shortest Remaining Time First

Shortest Remaining Time First is the **preemptive** version of Shortest Job First. Namely, this algorithm chooses the next task to schedule based on the least remaining execution time. Note that the scheduler might decide to change task whenever a new task is activated, or a new task ends its execution. The main advantage of this algorithms is:

- It improves Shortest Job First's **responsiveness**.

The main disadvantages are:

- **Starvation**, since mostly-completed and short tasks are always preferred to long tasks.
- The algorithm still **has to know the computing time** C_i in advance, in fact it works on the fraction of C_i which hasn't been done yet.

Highest Response Ratio Next

The Highest Response Ratio Next algorithm is a **non-preemptive** algorithm that schedules the task with the highest response ratio, computed as

$$RR_i = \frac{W_i + C_i}{C_i}$$

The main advantage is:

- It **prevents starvation** by considering also the time a task has been waiting.

The main disadvantage is:

- The scheduler must **know the computation time** C_i.

Round Robin

The round robin algorithm is a **preemptive** algorithm that puts in execution a task for a certain period of time q, called quantum. When a task's quantum ends, the running task is preempted and the scheduler picks another task from the ready queue (in a FIFO fashion) which is executed for another quantum. More precisely,

- The next task to execute is taken from the head of the queue.
- When a task is preempted it's added to the tail of the queue.
- When a task is activated, it's added to the tail of the queue.

We can fine tune the quantum q to find the right responsiveness for our system. In particular,

- A big quantum favours CPU-bound tasks and tends to a FIFO scheduler. Moreover, with a big quantum, we have a smaller overhead because we have to deal with fewer context switches.
- A small quantum favours I/O-bound tasks and reduces the average waiting time. Since we have more context switches, the scheduler's overhead is higher but the system is more responsive and the scheduling is more fair.

This algorithm is used in real life implementation and in particular in real time systems where we must ensure that a task doesn't go beyond a certain deadline. The main advantages of the Round Robin algorithm are:

- It has a **known maximum waiting time**, which is $(n-1) \cdot q$.
- The scheduler **doesn't have to know the computation time** C_i.
- It can achieve **good fairness and responsiveness**, since every task has the same amount of CPU time.
- **We never reach starvation** since each task is dispatched after some time.

The main disadvantage is:

- It has a **worst turnaround time** than SJF.

4.2.4 Priority scheduling

Now that we know the basics of scheduling, we can add the concept of priority and study algorithms to schedule tasks with a priority. A priority usually is a numerical value p used to say that a task, having higher priority, should be executed before another. Priorities can be

- **Fixed** if they are known and assigned at design time.
- **Dynamic** if they are assigned and changed at run-time.

Usually, the lower the value of p, the higher the priority, where 0 is the highest priority. Real time systems usually have negative priorities to indicate real time tasks (whilst best effort tasks are assigned positive values).

Multi-level queue scheduling

A priority based scheduler is usually implemented as a multi-level queue scheduler. This type of scheduler has multiple ready queues, each associated with a different priority level. When a task τ_i is activated, it's added to the queue with τ_i's priority and can't be moved to any other queue. The scheduler picks one task from first non-empty queue with higher priority. Each queue can also use a different algorithm to choose what task to schedule. Long story short, the scheduler:

- Chooses the non-empty queue with higher priority.
- Chooses a task from the chosen queue using a specific scheduling algorithm (e.g., one of those we have seen previously), which might be different for each queue.

Note that we have only one I/O queue which stores processes of every priority level. Also note that multi-level queue scheduling is **preemptive**.

One problem of this algorithm is that it may lead to starvation since if we have many tasks with high priority, the tasks with low priority will wait for a long time before being executed.

Multi-level queue scheduling with Round Robin

A way to solve the starvation problem could be to use the Round Robin algorithm on each ready queue, using small quanta for high priority levels and big ones for low priority queues. This partially solves the problem because we compensate the fact that a low priory task rarely gets executed giving it a long execution time. This means that when it's executed, a low priority task is executed for a long time. To achieve the best result we can

- Assign a low priority (i.e., a high priority value) to CPU-bound tasks so that they can be executed for a long time.
- Assign a high priority (i.e., a low priority value) to I/O-bound tasks so that they can wait.

This is because we want to exploit the long quanta for CPU-bound tasks (which rarely stop) and use small quanta for I/O-bound task which will naturally stop. This scheme ensures a good **responsiveness** and reduces starvation, without eliminating it completely. Note that boundness can either be specified by the user or be provided by the program itself.

Multi-level feedback queue scheduling with Round Robin

Multi-level feedback queue schedulers are an evolution of multi-level queue schedulers in which the boundness of a program (and consequently its priority) can be inferred dynamically. If the the scheduler notices that a task assigned to a big quantum usually blocks itself before the end of the quantum, then it marks it as an I/O-bound task. On the other hand, if a task is always preempted by the operating system, then it might be a CPU-bound task (since it stopped only because the scheduler asked to do so). More precisely:

- Upon being activated, a task is put in the highest priority queue (i.e., the one with shorter quantum).
- When the running task is preempted (i.e., the quantum expires), it is moved to the queue with lower priority with respect to the one in which it was before.

This means that CPU-bound tasks are progressively moved to low-priority queues while I/O-bound tasks are kept in high priority queues.

Solving starvation

Dynamic priority schedulers can mitigate the phenomenon of starvation, but they can't eliminate it. To definitively solve the problem we can use two techniques:

- **Time slicing**. With this technique we assign to each queue a percentage, called quota, of CPU time to use (usually higher to higher priority queues). If the quota expires, a task is picked from the next queue with lower priority (even if some task with the same priority is available). If the sum of the quota of each queue is smaller than the period (i.e., the time between preemptions), then we are guaranteed to have no starvation. Still, depending on the specific scheduling policy of each queue, we could have starvation.
- **Ageing**. The idea of ageing is to change the priority (increasing it) of tasks that are have been in the ready queue for a long time. Namely, we are changing the priority dynamically. This allows tasks to avoid starvation, since they will eventually be prioritised and executed.

These techniques can also be combined.

Linux multi-level queue scheduling

Linux uses a 5 level multi-level queue scheduler. The 5 queues used are:

- The **scheduler deadline queue**.
- The **scheduler FIFO queue**.
- The **scheduler round robin queue**.
- The **scheduler other queue**. This is where most of the task in a normal system are queued and this queue is handled by the standard CFS Linux scheduling algorithm.
- The **scheduler idle queue**.

The first three queue are reserved for real time tasks, if the system has some.

4.2.5 Multi-processor scheduling

Most of the devices on the market have multiple processors, thus we have to understand how schedulers behave when they have to feed multiple processing units. In case of multi-processor systems, a scheduler doesn't only have to choose one task to execute but also the processor to which the task should be assigned. For this reason, multi-processor scheduling is even harder than single-processor scheduling (which already was an NP-complete problem). The complexity of multi-core scheduling algorithms comes from the fact that

- We have to synchronise the execution of the tasks (task τ_i might have to execute before τ_j).
- We have to manage the simultaneous access, by different processors, to some shared resources (e.g., cache). Namely, we have to manage concurrency.

Tasks usually have access to the same cache (e.g. L2 cache) but they need different data, hence some task has to wait. To mitigate this problem, the scheduler can migrate one task from one processor to another. Mitigation works similarly to preemption, but the preempted task is assigned to another processor (which doesn't conflicts with other running tasks) instead of being sent in the ready queue.

When building a multi-processor scheduling algorithm we have to do two different design choices. The first choice is between:

- **Single queue** schedulers.
- **Multiple queue** schedulers.

If we choose multiple queue schedulers, we also have to choose between:

- **Single** scheduler. In this case, each queue uses the same algorithm to choose the next task to execute.
- **Multiple per-processor** schedulers. In this case, each queue can use a different scheduling algorithm.

Single queue schedulers

In single queue schedulers, every task waits in the same global queue. Having a single queue, the scheduling algorithm is very simple and the overhead is small. The main advantages of single queue schedulers are:

- They are **very simple**.
- They can achieve **good fairness**.
- They can manage **CPU utilisation** without many efforts from the algorithm point of view.

The main disadvantage is:

- They **don't scale well**, in fact having a single queue, the scheduler has to manage synchronisation between different CPUs and when the system has many of them, it might get messy. For these type of schedulers, mutexes and semaphores are required to correctly handle the preemption mechanism. Usually, the operating system task accessing the queue to pick a task has to wait the other running tasks that are writing the queue.

Multiple queue schedulers

Multiple queue schedulers use a different ready queue for each processor. Having decided that we use multiple queues, we have to define if:

- Each queue is assigned to a priority level.
- Each queue is a more complex data structure that can be ordered.

Independently from the approach we choose, the main advantages of multiple queue schedulers are:

- They are **more scalable** then the single queue case.
- They have an **higher overhead** since we have more data structures to handle the queues.
- They can **exploit more easily data locality** since every processor works on its queue. Moreover, say task τ_i has been preempted from CPU_k. If the schedulers lately reschedules τ_i on CPU_k, the data used by τ_i might be still there, effectively decreasing the number of cache misses.

However, we need to handle load balancing. More precisely, if we have multiple queues, we want all of them to have roughly the same number of tasks waiting or we want tasks to spend the same amount of time on each CPU. Unbalanced ready queues may lead to:

- **CPU utilisation problems**. Usually, one of the goals of a scheduler is to exploit a resource as much as possible since its useless to have one CPU processing a lot of tasks and one almost idling (due to an empty queue). Ideally we would want every processor to process the same number of tasks.
- **Performance issues**. If one processor has to handle most of the tasks, the system might be slow and unresponsive.
- **Thermal problems**. If one processor handles many tasks, it will be more stressed and generate more heat. Heat doesn't only impact the power consumption (which increases non linearly with temperature), performance and reliability of the core that generated it but also of the other cores. In fact, the heat can increase the temperature of other cores which will perform poorly as well.

The scheduler can move tasks in less crowded queues to solve both problems. As for priority scheduling, we can use **task migration** to handle unbalanced queues, improve data locality and reduce cache conflicts. In particular, we have two different approaches:

- The **push model**. In the push model, we have a dedicated task in the operating system, whose job is to monitor the queues and move tasks from one queue to another.
- The **pull model**, which is used when each queue has its own schedule. In the pull model, we have one task for each queue which looks in its and others queues to check if it can steal tasks from other queues (if its queue is empty). This model distributes the complexity of the whole scheduling algorithm on each queue, since we divide the algorithm is smaller algorithms (i.e., one for each queue, run by the scheduler) with simpler logic. In theory this mechanism should be scalable, however we always have to consider concurrent access to the ready queues (which is always complex) since different queues' scheduler might want to access the same queue and steal a task. Another problem regards how the scheduler should choose the queue from which a task should be stolen. These issues can also increase the scheduling overhead.

Hierarchical queues

In practice, we can use a hierarchical approach (which mixes the previous approaches). In particular, hierarchical queue schedulers have

- A **global queue** (at the top of the hierarchy) handled by a global scheduler which moves tasks from the global queue to one of the second level queues.
- A set of **second level queues**, each handled by a different scheduler. Second level queues can even be organised using priorities.

The main advantages of this approach are:

- It has a **better control over the CPU utilisation**.
- It **handles well load balancing**.
- It has a **good scalability**.

The main drawback is:

- It's **complex** to implement since we have to deal with more schedulers that have to synchronise correctly.

Chapter 5

Operating systems verification

5.1 Software verification

Software verification aims at checking if a program behaves like it's supposed to (i.e., how it's written in the specifications). We can't verify if a program is correct in an automatic way since it's an undecidable problem. However we still want to check, especially for safety-critical systems, if a program is correct or not. The closest we can get to this result is by using testing which, differently from what we want, tells us if a software has some errors, not if the program is correct. Namely, we can say if a program is not correct, but we can't say if a program is not correct because we might not have searched enough to find errors or miss-behaviours.

Operating systems are one of the most critical software in a system since everything else has to run on the OS, hence it's very important to correctly test and verify an OS. Most of the times, when verifying an operating system, we want to focus on:

- **Safety**. We want to verify if the OS is reliable and responsive since a crash or an unresponsive system might cause harm, especially in real-time systems (e.g., a plane).
- **Security**. We want to verify if the OS is not vulnerable to security attacks that might steal data or make the system crash.

There exist two main families of approaches for software verification:

- **Model checking**. Model checking is totally automatically, however it can't check Turing complete programs (since the problem is NP-complete). In particular, we have to use less powerful expressions that allow the model checker to automatically tell if the program, expressed in a less expressive language, is correct or not.
- **Automated theorem proving**. Automated theorem proving is very general but it requires some human intervention to write the proofs.

5.1.1 Model checking

Model checking represents a system with a model M, i.e., a simplified abstraction of a system, and proves if the model M has a certain property φ. Namely, we want to check if a property φ holds (is true) in model M and we write

$$M \models \varphi$$

Usually,

- Models are represented as finite state automata.
- Properties are usually expressed in Linear Temporal Logic (LTL). Note that LTL corresponds to star-free languages (i.e., regular expressions without the star operator). In general, we use a regular language, which is not as powerful as a Turing complete one.

Since the model and the property are written in different languages, we have to translate one so that we have a uniform language. Usually, we use some algorithms that can translate the negation of the property in an automaton A_φ. This step is very expensive since it generates an exponential number of states. Moreover, we have to consider also the translation from non-deterministic automata to deterministic ones, which exponentially increments the number of states.

Once we have the model M and the property's automaton A_φ we only have to build the intersection between M and A_φ, which has a polynomial complexity, and check if it's empty. Note that in this case we get useful information either if the intersection is empty or not. In particular:

- If the intersection is empty, i.e., $M \cap A_\varphi = \varnothing$, then we have proved that $M \models \varphi$.
- If the intersection is not empty, i.e., $M \cap A_\varphi = \varnothing$, we have found a string of the intersection language, which is a counter-example. Counter-examples are very useful since they tells us what's wrong in the system. For instance, if the property is related to security, the counter-example could be a vulnerability or an attack that can be mounted on the system.

In general, model checking is PSPACE-complete, meaning that it's polynomial deterministic in space but usually exponential in time. Model checking works, albeit its complexity, because the property we want to verify is usually simple, hence the translation to an automaton can be executed in a reasonable amount of time.

Improvements

Starting from the basic algorithms we can improve different parts of it. In particular, we can work on:

- Making it more expressive. Namely, we want to cover more expressive notations for properties.
- Making it faster or making it use less memory.

Usually, when improving the speed or the expressiveness of a model checker we end up with automata that can't fit in memory, hence we have to find ways to build only portions of the automata and visit the states on the fly.

Bounded model checking

Bounded model checkers try to improve the performance of a traditional model checker using a weaker logic, such as propositional logic or Satisfiability Modulo Theory, and adding a temporal bound. Namely, we can represent our program in a finite time window and use loops to represent infinite time.

Push-down automata

Some model checkers use push-down automata instead of finite state automata to represent the model and the properties. This is because, for instance, functions use stacks which are better represented by a PDA. In particular, many sub-classes of PDA have been developed for model checking. One of the most successful is **visibly push-down automata**, which are a sub-class of deterministic context-free languages.

Some extensions of visibly push-down automata have been created to obtain a more expressive language without losing the properties of vPDAs. An example is Operator Precedence that, together with the classical stack operations (push and pop) implements operations for handling exceptions or interrupts.

5.1.2 Automated theorem proving

Automated theorem proving aims at building a formal proof that the system has a property. In general the logic used in theorem proving tools are quite expressive, for this reason, it would be impossible to automatically build the proof. Instead, we have to build the proof and the tool only checks if the proof is correct.

Dependently Typed Functional Programming Languages

The languages used for automated theorem proving are based on functional programming languages. In particular, Dependently Typed Functional Programming Languages which are very popular because they allow to write values at the type level. These languages allow to write complex properties in the type itself of the functions. For instance, if we can write a function `f` with return type φ (which is also the property we want to prove), we have proved that `f` satisfies the property φ. This is also true for standard programming languages, but in this case the language for expressing types is much weaker (usually propositional logic-weak).

The connection between proving theorems and writing programs is called **Curry-Howard isomorphism** and defines a correspondence between **intuitionistic logic** (or constructive, where the law of excluded middle and double negation elimination are banned) and **typed lambda calculus**. The Curry-Howard isomorphism defines the following associations:

- False is isomorphic to the empty type.
- True is isomorphic to the Unit Type (`()` in Haskell).
- The conjunction $\wedge$ is isomorphic to Product Types.
- The disjunction $\vee$ is isomorphic to Sum Types.
- The implication $\Rightarrow$ is isomorphic to Function Types.
- The existence quantifier $\exists$ is isomorphic to the Dependent Pair Type (or Dependent Sum Type). Note that, in the Dependent Pair Type, the type of the 2nd component depends on the value of the first component
- The universal quantifier $\forall$ is isomorphic to the Dependent Function Type (or Dependent Product Type). Note that, in the Dependent Function Type, the return type depends on the value of the input value

Some of the most used dependently typed functional programming languages are:

- **Coq**, which is mainly a proof checker.
- **Agda**, which is used both for programming and proof checking.
- **Idris**, which is mainly for programming but can be used as proof checker.

Chapter 6

Memory safety

6.1 Memory safety in C++

Memory safety is a property of a program in which allocation and deallocation on the heap doesn't create any crash. For instance, we don't want that a program deallocates twice some data. In other words, memory access is always well defined in a memory safe program. Memory safety bugs can be classified in

- **Temporal memory safety bugs**. These errors are originated by the order in which operations are done.
- **Spatial memory safety bugs**. These errors are generated by a program that is able to access memory outside its bounds.

The problem with C++ and Rust is that they can't use a garbage collector (which will ensure safe access to memory) hence programmers have to explicitly and correctly allocate and deallocate data. Differently from C++, Rust has some compile and run-time mechanism for ensuring memory safety. Rust can deal both with temporal and spatial memory safety bugs.

Memory safety bugs are always related to an ownership problem. When a variable which has a pointer to an area of memory we say that the variable owns that area of memory. Having multiple variables that own a region of memory (i.e., have a pointer to it) might cause some problems. For instance, we might deallocate twice the same region of memory or use a region which has been deallocated by another owner. Another problem is when a region of memory, which has not been deallocated, has no owner.

Single ownership

Let us first analyse the problems we can have when we have only one owner of a region of memory. The main problem in this scenario is a memory leak, namely a region of memory which has been allocated but has no owner.

```
1 int leak_function() {
2     char *str = (char *) malloc(20);
3     str = (char *) malloc(40);
4     free(str)
5     return;
```

```
6 }
```

Listing 6.1: An example of memory leak.

C++ solves this problem using smart pointers, in particular, `unique_ptr` pointers. These pointers automatically call the destructor of the object when the variable with the pointer is deleted. For instance, in Listing 6.2 the `Rectangle` created at line 2 is deleted as soon as the variable `P1` is deleted (i.e., when the main returns, in this case).

```
1 int main(){
2     unique_ptr<Rectangle> P1(new Rectangle(10, 5));
3
4     // This'll print 50
5     cout << P1->area() << endl;
6
7     // here the smart pointer destructor will deallocate the rectangle object
8     return 0;
9 }
```

Listing 6.2: Unique pointers for solving a memory leak.

Move semantics

In a program different variables might have a pointer to the same region of memory. Namely, a region of memory might have multiple owners. To avoid accessing deallocated regions in this scenario, we have to ensure that only the last owner that accesses the memory can deallocate it. This can be achieved with a ownership movement. Namely, we can say that at any instant in time there is only a single variable that has a valid pointer to the original object (i.e., area of memory). This behaviour is obtained using `unique_ptr` pointers and the `move` function which takes a pointer `P1`, revokes its ownership and gives the ownership to another pointer. An example of ownership movement is shown in Listing 6.3.

```
1 int main(){
2     unique_ptr<Rectangle> P1(new Rectangle(10, 5));
3
4     // This'll print 50 unique_ptr<Rectangle> P2;
5     cout << P1->area() << endl;
6
7     // P1 becomes invalid here
8     P2 = move(P1);
9
10    cout << P2->area() << endl;
11
12    // only deallocates the pointer in P2
13    return 0;
14 }
```

Listing 6.3: An example of ownership movement. The ownership of a Rectangle object is moved from P1 to P2.

Note that in this case we still have only one variable that owns a region of memory since there is no time instant in which both `P1` and `P2` have a reference to the `Rectangle`.

Multiple ownership

In a program, multiple variables might own the same object (i.e., region of memory). This phenomenon is called aliasing and happens frequently, for instance when threads use a shared resource.

Double-free The first problem that can occur when multiple variable own the same area of memory is freeing (i.e., deallocating) a region of memory twice. For instance, the program in Listing 6.4 shows this behaviour.

```
int main(){
    char* s1 = (char *) malloc(20);
    char* s2 = s1;

    free(s1);
    free(s2);

    return 0;
}
```

Listing 6.4: An example of double-free.

The double free problem is solved in C++ using shared pointers. Shared pointers are smart pointers that keep a counter of how many variables are using the same pointer. This allows a shared pointer to deallocate an area of memory only when the counter reaches 0.

Not every problem can however be solved. Consider for instance the program in Listing 6.5. This is an example of use-after-free which we can't solve (we just shouldn't write programs like this). The problem is generated by the fact that `vptr` contains the address of a region handled by `v`, which is a `std::vector`. After the `push_back` operation, the `v` might free the memory pointed to by `vptr`, because of the internal mechanism that a `std::vector` uses for handling its calues, hence `vptr` might contain a reference to an area of memory which is deallocated.

```
int main(){
    std::vector<int> v{ 10, 11 };
    int *vptr = &v[1];
    v.push_back(12);
    std::cout << *vptr;
}
```

Listing 6.5: An example of use-after-free.

6.2 Rust

Rust is able to natively deal with ownership. In particular, Rust ensures that at any point in time there is only a single owner for every allocated region of memory. When we must use multiple ownership, Rust uses borrows (i.e., aliases) which are strictly controlled in terms of creation and destruction. The part of the Rust infrastructure that handles borrowing is called the **borrow checker**.

6.2.1 Single ownership

Rust handles single ownership with Resource Acquisition Is Initialisation (RAII) everywhere. This means that, if a variable owns a region of memory, when the variable is deallocated, the region of memory pointed to by the variable is allocated, too. Rust also allows to allocated objects on the heap and have them pointed by a variable on the stack using Box objects. For instance, in Listing 6.6 the variable `v` is on the stack and has a reference to a region of memory that contains the value 5. Whenever the main terminates, `b` is deleted and the region of memory pointed to by `b` is deallocated, too.

```rust
fn main() {
    let b = Box::new(5);
    println!("b = {}", *b);
}
```

Listing 6.6: An example of usage of the Box object.

6.2.2 Move semantics

Rust also natively offers the move semantics. This means that, given a variable `v1`, if we copy a reference to a region of memory contained in `v1` in another variable `v2`, then `v1` can't be used anymore.

```rust
fn main() {
    let s1 = String::from("Hello");
    let s2 = s1;
}
```

Listing 6.7: An example of automatic move semantics in Rust.

The same behaviour is obtained when passing a pointer to a function. Say we have a variable `s` on the stack which has a reference to a string on the heap. If we pass `s` to a function, we can't use `s` anymore, because ownership has been moved to the function. An example is shown in Listing 6.8.

```rust
fn consume(r: String) {
    // after using r, the variable is destroyed and the reference is deallocated
}

fn main() {
    let s = String::from("Hello");

    // s looses ownership
    consume(s);

    // compile error
    println!("{}", s);

}
```

Listing 6.8: An example of automatic move semantics for functions in Rust.

6.2.3 Multiple ownership

Rust has an equivalent of C++ shared pointers called `Arc` type, which stands for Atomic reference counting type. As the name suggests, an `Arc` variable is a pointer which has a counter of the number of owners of a variable. We can create a copy of an `Arc` variable using its method `clone` to which we have to pass the address of the variable with the reference to copy. An example is shown in Listing 6.9. As for C++, the `Arc` pointer deallocates the object only when its counter reaches 0.

```rust
fn main() {
    let s1 = Arc::new("Hello");
    let s2 = s1.clone(&s1);
}
```

Listing 6.9: An example usage of the Arc type in Rust.

Using reference counting is very expensive and wasted a lot of resources. For this reason, Rust supports borrowing, which allows to handle multiple ownership without overheads. Let us consider for intance the same example we saw for C++, rewritten in Listing 6.10, to show how Rust automatically deals with multiple ownership and solves the use-after-free problem.

```
1 int main(){
2     let mut v = vec![ 10, 11 ];
3
4     // let vptr: &'a mut i32 = &mut v[1];
5     let vptr: & mut i32 = &mut v[1];
6     v.push(12);
7     println!("{}", *vptr);
8 }
```

Listing 6.10: An example of borrowing in Rust.

In this example, the variable `vptr` is a mutable borrow. This means that when the pointer is assigned to `vptr`, all the other variables having a pointer to the same area of memory (i.e., `v[1]`) can't modify that area of memory until `vptr` hasn't released its pointer. Thanks to this behaviour, the compiler recognises that `vptr` accesses the shared region after `v` accesses it at line 4, hence it throws an error because that area shouldn't be used by `v` since it has been borrowed to `vptr`. In particular, the compiler throws the following error:

```
    error[E0499]: cannot borrow ‘v‘ as mutable more than once at a time
 --> src/main.rs:4:5
  |
3 |     let vptr: & mut i32 = &mut v[1];
  |                                - first mutable borrow occurs here
4 |     v.push(12);
  |     ^^^^^^^^^^ second mutable borrow occurs here
5 |     println!("{}", *vptr);  // compiler error
  |                    ----- first borrow later used here
```

Part III

Concurrency

Chapter 7

User space concurrency

7.1 Concurrency execution models

First of all, let us define concurrency, so that we know what we have to deal with.

> **Definition 7.1** (Concurrency)**.** *A program has to deal with concurrency when it is composed by activities where one activity can start before the previous one has finished.*

Concurrency is therefore a possibility which verifies if a program's instructions can potentially overlap in time. This means that at a certain execution time t, the program could be executing instruction i_i or instruction i_j. If concurrency is handled correctly, the order in which the instructions are executed (i.e., if we execute i_i before i_j or the opposite) doesn't impact the correctness of the program itself. Programs with overlapping activities can be handled by the operating system in different ways, called **execution models**.

7.1.1 Thread execution models

One of the easiest way of handling concurrent activities is using threads. The execution models that leverage threads are called **thread execution models**. In particular, the operating system can use:

- **Sequential execution**. Even if some activities are meant to execute together, the operating system might decide to run one after the other. In this way any problem is avoided.
- **Timesharing with threading**. An operating system can exploit threading to handle concurrency. More precisely, concurrent activities of a program are run on different threads and, in turns, the operating system assigns one thread to the CPU (assuming that only one processor is available). In other words, the OS assigns activity A_1 to the CPU, then, after some time, the state of A_1 is saved and the thread is stopped. The OS can now execute A_2 for some time, then stop it, and execute again A_1 (from where it stopped). This process can go on until all activities are over. In this case we aren't actually executing both activities at the same time.
- **Multiprocessor threading**. If the system has multiple processors available, the operating system can assign each activity to a different processor and effectively run multiple activities simultaneously.

These execution models are usually provided by the OS using the same interface (i.e., threads) and then the operating system decides whether to use timesharing or to assign activities to different processors. Threads can also be used by programmers. For this reason we can distinguish between:

- **Execution models**. The operating system decides how to handle concurrent activities using threads.
- **Programming models**. Developers can specify what portion of a program should be executed in a thread.

Importance of the multiprocessor thread execution model

In the evolution of computers we notice that at a certain point in time, we couldn't improve the chips' frequency, since, to produce faster processors we also had to increase their power consumption (which is a good thing, especially for some applications). The only way we can keep improving the overall performance (the single-thread performance is still impacted by the reduced frequency of a single core) is by using multiple cores in a single processor. This is why the multiprocessor thread execution model is so important for modern systems.

7.1.2 Lightweight execution models

When we use thread execution models, especially in the case of timesharing, we have to deal with context switches because the operating system has to continuously change the thread in execution. In some cases, as in real-time applications, this isn't acceptable hence we have to use other techniques, which reduce the overhead of context switches and fall under the name of **lightweight execution models**. The most used lightweight execution models are:

- **Co-routines** offer ed by programming languages. Co-routines allow to specify, in the program itself, at which point an activity can be preempted and removed from the CPU. More precisely, a function `f1`, when it has to be preempted, calls another function `f2` which starts executing (without context switches). Usually, using co-routines, we can achieve a smaller overhead with respect to timesharing.
- **Generators** offer ed by programming languages (e.g., Python). Generators allow to continuously generate values for the called of the generator, without being preempted. For instance, in a for loop a generator might return all the values in a certain interval without preempting the caller.

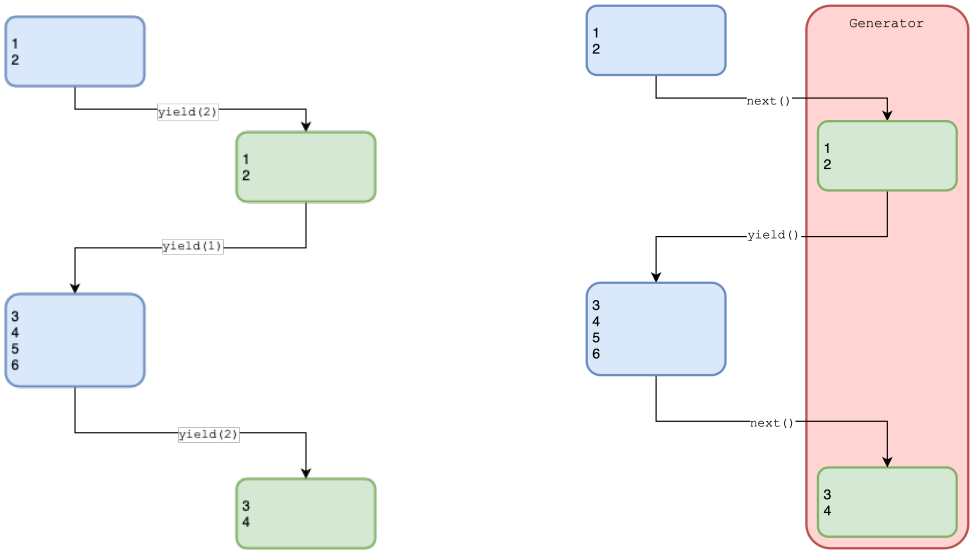
A graphical representation of co-routines and generators is shown in Figure 7.1.

7.1.3 Event-based execution models

If we have a program whose behaviour is defined by external events (e.g., a server), we can also use another set of execution models, which are called **event-based execution models**. Event-based execution models exploit:

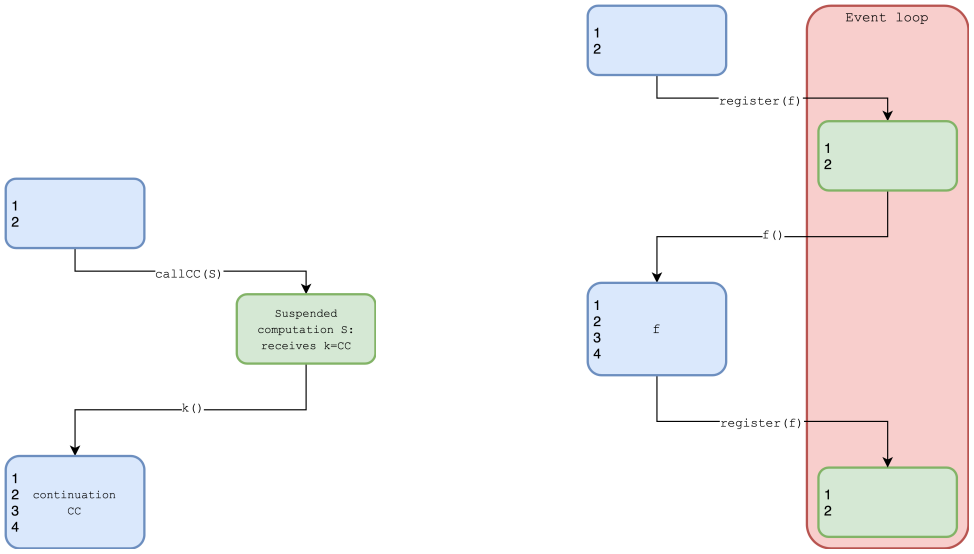
- **Continuation passing**, which works like callbacks. Using continuation passing, we incur in even less overhead then lightweight execution models.
- **Async and await constructs**, which allow to specify event-based activities.

A graphical representation of continuation passing and callbacks is shown in Figure 7.2.



(a) Co-routines used in lightweight execution models. (b) A generator used in lightweight execution models.

Figure 7.1: Lightweight execution models.



(a) A continuation in event-based execution models. (b) Callbacks in event-based execution models.

Figure 7.2: Event-based execution models.

7.2 Properties of concurrent programs

A concurrent-execution program can be characterised with two properties:

- **Safety**, or correctness. If we see a program as a state machine, safety means that the program doesn't reach an error state. Safety also means that whenever the program accesses some shared data, it doesn't change it improperly.
- **Liveness**, or progress. If we see a program as a state machine, liveness means that the program will eventually reach a final state. This means that the program will never reach a state from which it's impossible to exit , i.e., it will never reach a **deadlock**. Liveness is also effected by the fact that an high priority thread could be delayed by a lower priority thread. This problem is usually referred to as **priority inversion**.

7.2.1 Safety

The issues that might threat the safety of a program can be divided into:

- **Data races**. A data race happens when two instructions i_a^1 and i_b^2, belonging to two threads T_1 and T_2, access the same variable v without any synchronisation mechanism. Remember that a synchronisation mechanism is a way to ensure that only one thread at a time accesses the variable, or, equivalently that either T_1 accesses v before T_2, or vice versa. In general, a data race isn't a problem, but only when it might cause an error.
- **Atomicity violations**. An atomicity violation happens when an operation, that was meant to be executed without being interrupted (i.e., atomically), is interleaved with another operation that directly modifies the data on which the former is working. Consider for instance the code

  ```
  if (thd->proc_info) {
      fputs(thr->proc_info, ...);
  }
  ```

  executed by thread 1, and the code

  ```
  thd->proc_info = NULL;
  ```

  executed by thread 2. If after doing the check we allow the instruction of thread 2 to be executed, then thread 1 will work on data that's being changed (and that doesn't satisfy the condition anymore).

Note that, the problems we have described can be solved by the programmer using, among other things, locks and semaphores. However, even the operating system has techniques to avoid them.

7.2.2 Liveness

Deadlocks

One of the main threats to liveness is given by deadlocks, thus let us formally define them.

Definition 7.2 (Deadlock)**.** *A deadlock is a situation in which no task $t_i \in T = \{t_0, \ldots, t_n\}$ can take action because it's waiting for another task $t_j \in T$ to take action.*

Deadlocks can happen for different reasons:

- **Circular wait**. Thread t_1 is waiting for thread t_2 while t_2 is waiting for t_1.
- **Mutual exclusion**. Two threads t_1 and t_2 can't access the same variable at the same time.
- **Hold and wait**. A thread is holding a resource R_1 (e.g., a shared variable) and asks for another resource R_2 at the same time (i.e., it wants to hold more than one resource). Basically, the thread holds a resource R_1 while waiting for other resources R_2, R_3.
- **No preemeption**. We can't take away from a thread the resources it has acquired so far. Usually, this condition goes together with hold and wait because a thread is holding some resources, it can't be deprived of, and wants to obtain new ones.

If one or more of these conditions are met, then a program will most likely incur in a deadlock. Since the aforementioned conditions are key for having a deadlock, we should explore some ways to avoid them.

Avoiding mutual exclusion To avoid mutual exclusion, we can use some language constructs that allow to execute some operations in a safe way, without using locks. Say for instance we want to create a function (that will be executed by a thread) that takes a shared integer value and increments it by a certain amount. To execute this operation atomically, we can write the following function

```
1 void add_atomically(int *value, int amount) {
2     int old;
3     do {
4         old = *value;
5     } while (_atomic_compare_xchg(value, old, old+amount) == 0);
6 }
```

In particular, it's interesting to understand what the `_atomic_compare_xchg` function does. It atomically tries to put the value `old+amount` in the variable pointed to by `value`, only if the variable pointed by `value` is equal to `old`. If the function doesn't succeed, it returns 0 and we try again. This means that if another thread modified the value pointed by `value` between lines 4 and 5, the function would fail and the while loop will try to execute the increment again. Note that we aren't using locks, hence this approach to programming is called **lockless programming**.

Avoiding no preemption and hold-and-wait To avoid waiting for a resources while holding some others, we apply simple rules which allow threads to have all the resources they need, or no resource. Say for instance that we have a program with two locks `M1` and `M2` associated to resource $R1$ and $R2$, respectively. If a thread requires both resources then it can

1. Lock resource `R1`.
2. Try to lock resource `R2` and
   - If it can lock it, it goes on and uses the resources.
   - If it can't lock it, it releases `R1`.

The same example can be translated in (pseudo)code as follows.

```
while(!done){
    lock(M1);
    if (!try_lock(M2)) {
        unlock(M1);
    } else {
        update(R1);
        update(R2);
        unlock(M1);
        unlock(M2);
        done = true;
    }
}
```

However, if we consider another thread executing the same instructions, but switching `R1` and `R2`, then we get to a situation in which the two programs will keep failing the `try_lock` check and get stuck in a circular wait. This means that resources should be acquired always in the same order by all threads using them.

Avoiding circular waits To understand what's the problem with circular waits and how to solve it, let us consider the following example. Say two people are having dinner and, if they want to eat, they have to

1. Pick the fork on their left.
2. Pick the fork on their right.
3. Eat.
4. Put back the fork on their left.
5. Put back the fork on their right.

Note that there are only two forks on the table. This can be translated, as a program, as

```
while (1) {
    get_forks();
    eat();
    put_forks();
}

void get_forks(int p) {
    sem_wait(&forks[left(p)]);
    sem_wait(&forks[right(p)]);
}

void put_forks(int p) {
    sem_post(&forks[left(p)]);
    sem_post(&forks[right(p)]);
}
```

where the functions `left(p)` and `right(p)` return the correct index to use in the array `forks` (i.e., `left(p)=1` for person `p=0`, and `left(p)=0` for person `p=1`). The problem here is that person $p = 0$ can take `fork[1]` and person `p=1` can take `fork[0]` (i.e., both take the left fork), but then we reach a deadlock because both people have a fork and won't put it back before obtaining the other fork.

To solve this problem we have to specify an order with which the forks should be taken. If both try to obtain `fork[0]` in the first place, then only one of them obtains it and the same person can also get `fork[1]`, eat and put both forks back for the other person to eat.

In general, to avoid deadlocks because of circular waits, we should define a total ordering with which resources should be locked. In some cases even a partial order might be used, but we have to be sure that the total order relationship holds for resources that might cause a deadlock. Namely, we can enforce total ordering only on related resources.

Priority inversion

Before getting confused, let us clarify some key concepts about priorities. When dealing with priorities, we will talk about priorities and priority values. The priority is the level of importance, hence a high priority process is more important than a low priority one. Priorities are represented by priority values, i.e., numbers. In particular:

- A small priority value, i.e., a small number, represents high priority.
- A high priority value, i.e., a big number, represents low priority.

Let us now consider the following scenario to describe the priority inversion problem. Say we have three threads T_1, T_2 and T_3 with decreasing priority (T_1 being the one with higher priority). Also assume that the scheduler is preemptive and threads T_1 and T_3 access a resource R through mutual exclusion. Let us consider the following sequence of events (also represented in Figure 7.3):

1. Thread T_3 starts executing and immediately locks the shared resource R.
2. After a while, T_3 is preempted and T_1 starts executing.
3. T_1 wants to access the shared resource R, which is however locked by T_3, hence T_1 goes to sleep.
4. The scheduler picks T_3, which starts executing.
5. T_3 is preempted and, since T_1 is waiting, T_2 is scheduled for execution.
6. T_2 is preempted and, since T_1 is waiting, T_3 is scheduled for execution.
7. T_3 doesn't need the shared resource R anymore, hence it releases it.
8. T_1 is woken up and is scheduled for execution.

As we can see, even if task T_1 had higher priority, it has been delayed because it was waiting a resource held by a low priority process, i.e. T_3. The waiting time has been ever bigger because of T_2 (which also has priority lower than T_1). Note that, if T_1 is a critical thread, it could miss its deadline and cause problems.

We can use different tools for solving this problem, some of which are even available in the POSIX `pthread` library. The main techniques used are:

- **Highest Locker Priority** (HLP). If using this technique, the priority of the low priority thread is changed to the one of the high priority thread with which the locked resource is shared. In our example, when T_3 locks R, T_3's priority is changed to T_1's priority. In formulas, the priority of a process i is computed as

  $$p_i(R) = \min\{p_h : T_h \text{ is using resource } R\}$$

  Remember that we take the minimum because low-priority values represent a high priority. This means that T_3 isn't blocked by T_1 or T_2, since they don't have a higher priority. The

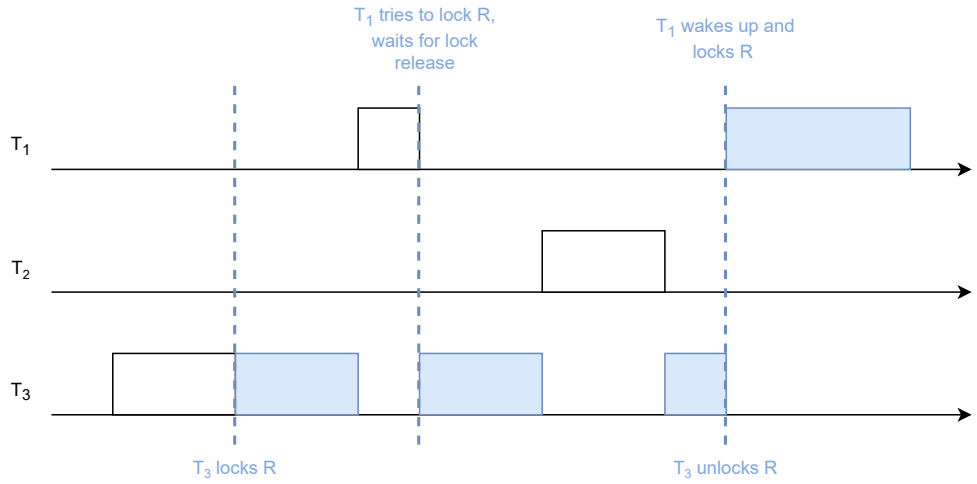


Figure 7.3: An example of priority inversion. The critical section in which the shared resource R is used is shown in blue.

priority of T_3 is changed back to its original value as soon as the shared resource is released so that T_1 can immediately start its execution. Note that this mechanism requires the scheduler to know the priorities of the other threads that use R.

- **Priority Inheritance** (PI). The main problem with higher locking priority is that T_1 can't execute the code before the critical section (i.e., the part in which the shared resource is used). Priority inheritance solves this problem by changing T_3 priority only when T_1 tries to access R. So the main difference with respect to HLP stands in the fact that priority is inherited only considering those threads that are currently trying to access the resources, whilst in the former protocol it was inherited beforehand and it was mandatory for the scheduler to know in advance what resources are going to be accessed by each thread. Still, it is left unsolved unsolved inversions that come from the inheritance of multiple threads.

- **Priority Ceiling** (PC). The priority ceiling protocol solves the priority inversion problem when multiple resources are shared (e.g., we add a shared resource R' between T_1 and T_2). If T_3 locks R first and T_2 wants to lock R' afterwards, then the protocol doesn't allow T_2 to lock R' because it's shared with a thread which is also sharing another resource R, which is already locked. In practice, given a resource R, we define the priority ceiling $C(R)$ of that resource as the lowest priority (i.e., the highest priority value), of the resources that have access to it. In formulas, we compute $C(R)$ as:

$$C(R) = \max\{p_i : R \in \sigma_i\}$$

where σ_i is the set of resources to which process i has access. A task T_i is allowed to enter a critical section only if its priority is lower (i.e., if the priority value is higher) than all the priority ceilings of the resources currently locked by other tasks. Namely, T_i can enter a critical section if

$$p_i > \max_i\{C(R_i) : R_i \text{ in use}\}$$

The `pthread` library allows us to specify which protocol we want to use when working with threads. In particular, we can use the `pthread_mutexattr_setprotocol` function to set the protocol we want to use and the following macros to specify the protocol:

- `PTHREAD_PRIO_NONE` if we don't want any protocol.
- `PTHREAD_PRIO_INHERIT` if we want to use priority inheritance.
- `PTHREAD_PRIO_PROTECT` if we want to use priority ceiling. The ceiling priority can be set with the `pthread_mutex_setprioceiling` function.

An example is shown in Listing 7.1.

```
#include <pthread.h>
int main () {
    pthread_mutex_t mutex;
    pthread_muteaxattr_t mattr;
    pthread_mutexattr_setprotocol(&mattr,
    PTHREAD_PRIO_INHERIT);
    pthread_mutex_init(&mutex, &mattr);
    return 0;
}
```

Listing 7.1: Setting the protocol for solving the priority inversion problem in the POSIX library.

7.3 Locks

Locks can be used to handle concurrency in the user space (i.e., in instructions executed directly by the program). In particular, user space locks in Linux are based on a concept called futex, which is compression of Fast User-Level Lock. The idea behind futexes is to avoid

- As many **system calls** as possible because they induce state saving overhead for switching to kernel mode.
- As many **context switches** as possible (i.e., change of thread in execution).
- The **thundering herd** problem.

A futex is implemented in Linux's `pthread` library or in POSIX `NPTL` library.

7.3.1 How does a futex work

Classical implementation

First, let us analyse what happens when we don't use the futex implementation for locks. In the classical implementation of locks, a lock is a variable in the kernel to which we associate a queue of tasks that are waiting to lock the variable. The lock is accessed (i.e., requested to lock) with the `semop` system call. Let us consider, for instance, two threads T_1 and T_2 that want to lock a lock L using the function `lock`. When one of the two threads, say T_1, calls the `lock` function:

1. Some code in the `pthread` library is executed.

2. A `semop()` system call is executed to call the kernel, enter supervised mode and acquire the lock variable in the kernel. Note that this transition introduces some overhead (we are switching from user to kernel mode) which is however necessary since the lock variable is in the kernel space.
3. The lock variable in the kernel space is locked for T_1.

When the other thread, T_2 in our example, tries to lock L:

1. It executes some code in the `pthread` library (i.e., in user space).
2. A `semop()` system call is executed.
3. Since the lock variable is already taken by T_1, T_2 is added to the waiting queue (which is also in the kernel).
4. When the lock L is unlocked by T_1, thread T_2 is taken from the waiting queue and L is locked and assigned to T_2.

Futex implementation

The classical lock implementation requires a thread to always switch to kernel mode, which is expensive and adds a significant overhead. The futex implementation, instead, tries to classify the situations in which a lock can be. More precisely, a lock can be:

- **Uncontended**. A lock L is uncontended when the threads that use it never overlap when using L. Namely, no two thread try to use L in the same interval of time. In other words, in this case, when a thread wants to lock L, it is unlocked. **Whenever a lock in uncontended, the lock operation does not involve a system call**.
- **Contended**. A lock L is contended when a thread tries to lock L which is (or might be) locked by another thread. **Whenever a lock in contended, the lock operation can involve a system call** so that threads can be put in the kernel's wait queue.

Note that we don't know the actual frequency with which we can have these types of locks since it depends on the application. Practically, since we have to lock L without going in kernel mode, the lock variable L must be in the user space and a thread is moved to the kernel space only when required. On the other hand, the wait queue is always in the kernel space.

The lock variable is a 32 bit integer accessed and modified only using atomic operations, namely operations that can't be preempted. The lock variable is divided in two parts:

- The 31 least significant bits encode the number of waiters.
- The most significant bit says if the variable is locked or unlocked.

This means that, every time a thread T_1 wants to lock a mutex L (calling `lock(L)`):

1. It checks the most significant bit of the lock variable and tries to set it to 1 with an atomic operation.
2. If the lock is set, i.e., another thread has locked it:
   (a) T_1 increases atomically the number of waiters.

(b) T_1 triggers an operation, namely the system call `futex` with parameter `FUTEX_WAIT`, to switch to kernel mode since we are in a the contested case. This operation allows T_1 to be put in a wait queue in the kernel so that it can be woken up later on, when L is unlocked.

3. If the lock is not set, i.e., is unlocked:

   (a) T_1 sets the lock bit (i.e., the most significant one).

Linux stores, for every futex that has been declared and for which there is at least one thread waiting, a structure that keeps which threads are waiting for which lock. The data structure used for storing waiting threads is optimised for this task and in particular Linux uses a collection of lists (or queues) and an hash function that, given a lock returns a certain list. Every queue can be shared across different locks.

Listing 7.2 shows what is the runtime operation done when a thread tries to lock a mutex implemented with a futex. Listing 7.3 shows the unlock operation instead. Note that the `mutex` variable passed to both functions is a pointer to the 32 bit lock variable we have described before.

```
1  void futex_based_lock(int *mutex) {
2      int v;
3      if (atomic_bit_test_set(mutex, 31) == 0) return;
4      atomic_increment(mutex);
5      while (1) {
6          if (atomic_bit_test_set(mutex, 31) == 0) {
7              atomic_decrement(mutex);
8              return;
9          }
10         v = *mutex;
11         if (v >= 0) continue;
12         futex(mutex, FUTEX_WAIT, v); /* sleeps only if mutex still has v */
13     }
14 }
```

Listing 7.2: User space implementation of the lock operation using futexes.

```
1  void futex_based_unlock(int *mutex) {
2      /* unlock and if the mutex is zero return */
3      if (atomic_add_zero(mutex, 0x80000000))
4          return;
5      futex(mutex, FUTEX_WAKE, 1); // wake up only one thread
6  }
```

Listing 7.3: User space implementation of the unlock operation using futexes.

Let us highlight some important features and characteristics of these pieces of code so that we can understand it better, since it applies some not-straightforward optimisations:

- The function `atomic_bit_test_set(mutex, 31)` atomically sets the 32nd bit of the value pointed to by `mutex` and returns the previous value. If the old value was 0 (i.e., it was unlocked), we have already locked it with this operation and we can return, having locked the mutex.
- After trying and failing to acquire the lock, we try again to lock the mutex (line 6) just in case another thread has freed the lock between lines 3 and 6.
- At line 11, the code checks if the value pointed to by `mutex` is non negative. This is because the most significant bit is used to tell if a number is positive or not, hence we are checking if

the last bit is 0 (`v` positive) or 1 (`v` negative). Namely, we are checking if the mutex is unlocked or not. If the mutex is unlocked, i.e. `v` is positive, we can execute the while loop again and try to acquire it, without going calling the `futex` system call, otherwise we must execute the system call and go in kernel space to add the current thread to the wait queue.

7.3.2 Condition variables

Note that futexes can also be used for condition variables.

7.4 Event based concurrency

When building an application, we aren't forced to use the thread-lock model, especially in cases in which the progress of the application doesn't depend on the amount of computation that the application did but rather on external events that might or might not arrive. In these cases, we must switch to the event-based concurrency model. This programming paradigm is used especially in

- GUI-based applications.
- Internet servers (e.g., Nodejs allows to write internet servers using event-based concurrency).

In an event-based application, each activity waits for an event to occur. When the event occurs, i.e., it's posted by an external entity, the application checks the type of the event and identifies which activity it belongs to (i.e., which activity was waiting for that event). Once the activity has been identified, the application invokes that activity which can execute its code and progress its execution. Schematically, the lifetime of an activity can be seen as follows:

1. An activity sleeps, waiting for an event.
2. An event is posted and the application checks to which activity it belongs.
3. An activity is invoked and it executes some code.
4. The activity goes back to sleep, waiting for another event.

7.4.1 Event loop

The problem with this model is that we could, in principle, use one thread for each activity so that, when an activity is invoked, the thread is woken up. This implementation is simple and straightforward, however it doesn't produce the best throughput. This is because, whenever we switch from one activity to another, the operating system has to execute a context switch, which costs time and adds a significant overhead.

A good solution for optimising the throughput of the application is using an **event loop**. This implementation uses a single thread of execution that executes the main function, which essentially executes the event loop. In particular, the event loop.

1. Repeatedly checks for incoming events. Effectively, the main function is the only interface through which external entities can post events.
2. When an events arrives, the main loop can invoke the correct activity. Usually, the main function has a table that associates, for each event, the activity that should be invoked. The activity isn't invoked in a new thread but simply it's called like if it were a function.

The event loop we just described can be implemented as in Listing 7.4.

```
while(1) {
    events = getEvents();
    for (e in events) {
        processEvent(e);
    }
}
```

Listing 7.4: An event loop.

Note that this implementation doesn't require locking, even if multiple activities work with shared data. This is because every activity modifies the data inside the `processEvent` call.

This implementation works fine, however it has a problem. When an activity terminates and, after some time, it is invoked again, it should be able to restore its previous state. To handle this problem, we have to use callbacks. The callback mechanism uses two functions:

- `waitfor(event, callback, state)`. This functions allows an activity to specify that it's waiting for an event `event` and when it happens it should execute the function `callback` using `state` as current state. Basically, this function is used by an activity to request to be registered for an event.
- `callback(state)`. This is the function registered by the activity and it's called by the event loop passing the `state` passed when registering the callback via the `waitfor` function. Upon terminating the `callback`, the activity has to register again (if it wants to) using the `waitfor` function. This allows to automatically handle state changes and restores.

More modern application also use the async-await model that allows the main loop to understand where an invocation ends and where it restarts so that the state can be restored. Another event-based implementation is `libuv`.

7.4.2 Network events example

Let us describe an API to practically understand how event based programming works. In particular, we will consider the API for handling network input/output events.

First, let us introduce the functions we can use to check if an event has been posted. The API provides us with the functions `select` and `poll` that check if there is any incoming network event that should be handled. For instance, the `select` function,

```
int select(int nfds, fd_set* readfds, fd_set* writefds, fd_set* errorfds, struct
    timeval* timeout);
```

takes:

- The file descriptors `* readfds` that should be checked for any request incoming.
- The file descriptors `* writefds` for any outgoing write request completed.
- The file descriptors `* writefds` for any error.

where `fd_set` is a structure that holds the stream that should be checked. The `select` returns how many events have been posted and rewrites the variables passed to identify on which streams the application has received some requests. For instance, if the `readfds` contains the values 2, 5 and 10 (which represents some streams) and, after returning, it contains 2 and 5, it means that some requests have been posted on stream 2 and stream 5.

An example of utilisation of the `select` function is shown in Listing 7.5. More precisely:

- At line 3, the application defines the stream number it's interested in.
- At lines from 4 to 8, the application allocates a set of descriptors `readFDs` which are initially set to 0 and then set to the values in `wfd`.
- At line 9, the `select` function is invoked. Note that the select blocks the execution until some event arrives.
- At lines 12, 13 and 14 the application checks if a particular stream has received a request and, if so, it executes the request using the `processFD` function.

```
1  void main() {
2      while(1) {
3          int wfd[] = {10, 20, 30};
4          fd_set readFDs;
5          FD_ZERO(&readFDs);
6          for (int i=0; i < 3; i++) {
7              FD_SET(wdf[i], &readFDs);
8          }
9          int rc = select(3, &readFDs, NULL, NULL, NULL);
10         for (int i=0; i<3; i++ {
11             if (FD_ISSET(wdf[i], &readFDs))
12                 processFD(wfd[i])
13         }
14     }
15 }
```

Listing 7.5: An event loop in C.

The problem with this implementation is that an activity can't block the entire event loop, otherwise we would block the would block the whole application.

Chapter 8

Kernel space concurrency

8.1 Operating system concurrency

Until now we have analysed how concurrency is handled by application (i.e., in the user space), however the operating system might execute task concurrently, too. Note that the sources of concurrency in operating systems and the techniques for handling them are different than those of a normal application, hence we have to analyse them more in depth.

8.2 Sources of concurrency

Concurrency can be introduced in kernel activities by three different sources:

- **Kernel preemption**. Kernel preemption happens when multiple threads in the kernel can share the same resources. For instance, we can have two system calls, done by different applications that try to read the same file.
- **Interrupts**. An interrupt might share data with some activities in the kernel. For instance it might share data with a process context routine in the kernel (i.e., a system call).
- **Multiple processors** (and the associated memory models). If we have multiple processors each could execute a kernel activity which might share some data with another kernel activity in execution in parallel on another processor.

8.2.1 Interrupts

Interrupts can occur asynchronously with respect to an application and can even happen in kernel mode. When, during the execution of a kernel activity, an interrupt is received, the kernel activity is stopped and the interrupt is handled. In particular, we call

- Process context the kernel activity initiated by a process in user mode. For instance, when an application executes a syscall, it's executed in the process context in the kernel.
- Interrupt context the kernel activity used to handle an interrupt. For instance, as soon as an interrupt is received, it's handled in the interrupt context.

What typically happens is:

1. A process in user mode executes a system call, switches to kernel mode and the OS executes a kernel activity in the process context.
2. An interrupt arrives, the kernel activity is stopped and the interrupt is handled in the interrupt context. During this phase, the interrupt handler might even be interrupted by other higher priority interrupts.
3. The interrupt routine terminates.
4. The kernel activity resumes its execution.

Since the interrupt executes some code, too, we don't want the two activities to corrupt any shared data on which they are working concurrently. If both activities can access and modify the shared data without doing any damage, we say that the code is **interrupt-safe**.

A practical example of a shared variable that could be corrupted is the `jiffies` variable, stored in the kernel, that saves the current up-time (i.e., the time since boot) of the machine. Typically, this variable is written by the interrupt context associated with the system clock but it's accessed by other kernel activities. For this reason we have to handle this variable only with interrupt-safe code.

8.2.2 Multiprocessing

When the machine has multiple processors, we have to deal with **true concurrency**. Namely, we have to ensure that the kernel code that simultaneously runs on two or more processors can safely access and modify any shared data. The code that is safe from concurrency on symmetrical multiprocessing machines is said to be **SMP-safe**. Note that it might be hard to write SMP-safe code if different processors have different views of the memory. Luckily we have techniques to let processors have the same view of the memory.

8.2.3 Kernel preemption

As we have already seen, when a process P, running in user mode, wants to execute some privileged functions, it has to do a system call. The system call allows the operating system to execute some code on behalf of P in kernel mode. When P is in kernel mode, some interrupt I_1 might rise; this forces P to be stopped for executing the interrupt routine. While handling the interrupt, another higher priority interrupt I_0 might rise and, as before, we have to stop I_1 and handle I_0. When I_0's routine finishes, the control is given back to I_1 which terminates its routine, too. At this point we can have two different behaviours, depending on the type of kernel. In particular:

- In a **non-preemptive kernel** the control is given back to process P, in kernel mode, which terminates the execution of the code in the kernel mode and goes back in user mode. In a non-preemptive kernel, **a context switch from kernel to user mode is executed only when the kernel activity is over**.
- In a **preemptive kernel** the control can be given back to P, in kernel mode, or to another process P_2 in user mode. This means that **we can switch from kernel to user mode even before terminating the kernel task**. Namely, in preemptive kernels, we can have a context switch even not at the end of a kernel activity. Put in another way, we can switch from a process P_1 in kernel mode to another process P_2 in user mode way before P_1 goes back to user mode. This creates some concurrency between processes P_1 and P_2. An example of preemptive kernel is shown in Figure 8.1.

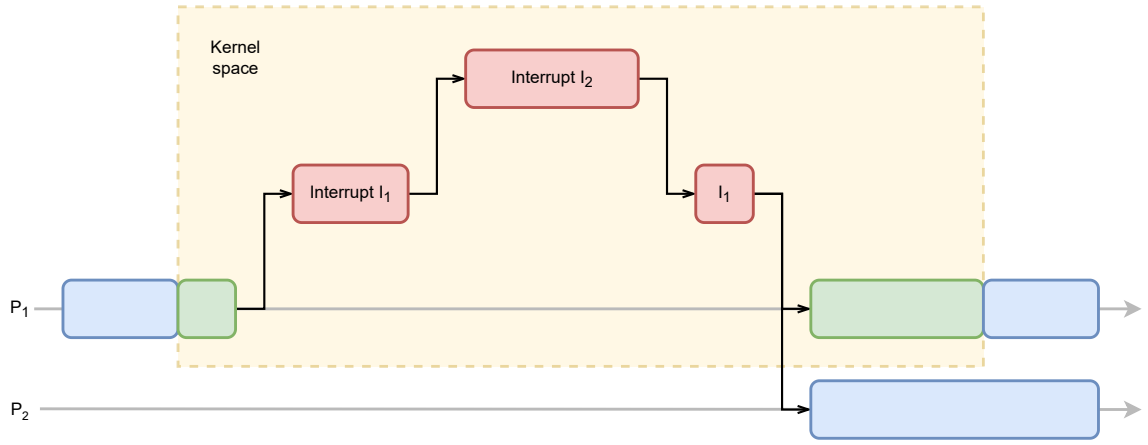


Figure 8.1: The execution scheme of a preemptive kernel.

The kernel in Linux has become preemptive since Linux 2.6 and the preemption points, that is the points in which we can choose to execute a process in user mode, are:

- At the end of interrupt/exception handling, when the `TIF_NEED_RESCHED` flag in the thread descriptor has been set (forced process switch).
- If a task in the kernel explicitly blocks, which results in a call to `schedule()`, (planned process switch). It is however always assumed that the code that explicitly calls `schedule()` knows it is safe to reschedule.

Preemption count

Since preemptive kernels are used in modern Linux distributions, and they can cause concurrency problems, we have to ensure that processes do a context switch only when it's safe. Every process has a variable `preempt_count` that used to understand if it can safely give control to another process (even in user mode) or it must keep executing. In particular, the `preempt_count`:

- Is set to 0 whenever a process enters kernel mode.
- Is increased whenever a process enters a critical section or an interrupt executes in the process' context.
- Is decreased whenever an interrupt or critical section ends its execution.

If the variable is different from 0, the kernel can't give control to another thread in user mode. Moreover, as soon as the `preempt_count` is equal to 0, the kernel can choose to

- give control to another process in user mode (even if current process in kernel mode hasn't finished), or
- continue executing the process in kernel mode.

An example of preemptive kernel using the `preempt_count` variable is shown in Figure 8.2.

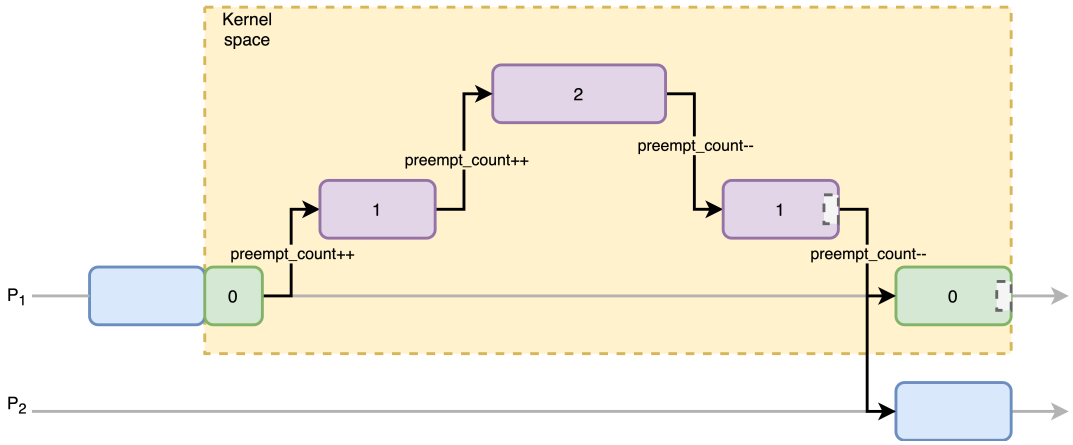


Figure 8.2: The execution scheme of a preemptive kernel with the preempt count variable. The moments in which the kernel can choose to execute another process in user mode are shown with a grey dotted rectangle.

Real-time patch

The kernel can be compiled without preemption support, however preemption allows the kernel to be more responsive. In real-time systems, the responsiveness given by a preemptive kernel isn't enough, hence we need to apply the `PREEMPT_RT` patch which maximises the preemptable code in kernel mode. Note that this patch is best-effort, hence it doesn't provide any guarantees on whether the deadlines will be met.

The `PREEMPT_RT` patch maximises preemptable code allowing critical sections in kernel mode to be preempted. This means that the kernel can stop a process in the middle of its critical section (i.e., when it has a lock) and give control to another process in user mode. This behaviour can lead to concurrency problems if the preempted process (i.e., the one in kernel mode) had a lock that is also required by the scheduled process. The kernel mush therefore be able to choose a process that has no lock in common with the preempted process. Say we call P_1 the preempted process in kernel mode and P_2 the new process. Differently from the simple preemptive kernel, we can have the following sequence of actions:

1. P_1 starts its execution in kernel mode.
2. P_1 locks a lock L.
3. An interrupt I rises and it's handled.
4. Interrupt I's routine ends.
5. Since the kernel is using the `PREEMPT_RT` patch, it can do two things:
   - Keep executing P_1.
   - Preempt P_1 and execute another process P_2. When P_1 is blocked, it's put in the state `TASK_RUNNING_MUTEX`.

   This is the main difference with the kernel without patch. Without the patch, the kernel has to wait the end of the critical section (i.e., it has to wait P_1 releasing the lock). On the other

hand, with the `PREEMPT_RT` patch, the kernel can preempt P_1 even if it's executing a critical section.

This sequence of events is shown in Figure 8.3.

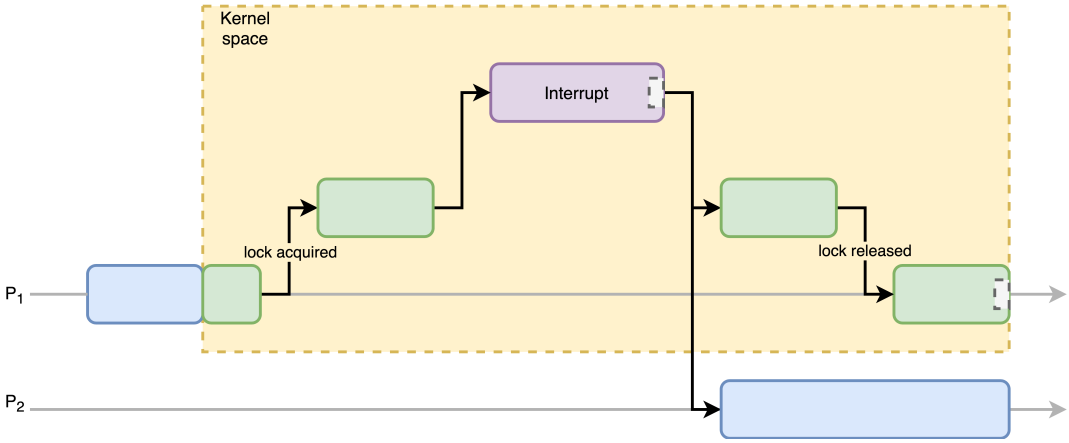


Figure 8.3: The execution scheme of a preemptive kernel with the PREEMPT RT patch. The moments in which the kernel can choose to execute another process in user mode are shown with a grey dotted rectangle.

Moreover, say an interrupt I has to be handled by process P. Instead of executing I's handler, the interrupt routine is scheduled in another task that can be executed later on. This is done to further improve reactiveness. This also means that the interrupt handler is much shorter (it only consist in creating a new task to execute the actual handler), hence we have to wait less time before deciding whether to give control back to P or to another process.

8.3 Single processor locking

Locks in kernel mode are different than locks in user mode. This is because kernel code has different requirements with respect to user code. Let us now analyse in details how locking in kernel mode works.

8.3.1 Spinning locks

Spinning locks have a different approach than mutexes (which we can call sleeping locks). In particular, a spinning lock uses a busy wait that continuously checks the if the lock is free, until the lock becomes free and can be acquired. This approach is radically different from the one used in user code, in which we prefer to put the process to sleep until the required lock is released. Spinning locks can be implemented in different ways:

- **Spinlocks**.
- **Readwrite locks**.
- **Seqlocks**.

Spinning locks are preferred in the kernel over sleeping locks since they don't require the complex infrastructure of a sleeping lock and we don't have to handle waits and wake-ups.

Spinlocks

A spinlock is implemented as a variable that can be either 0 or 1. The principle of working of spinlocks is fairly simple: an active loop keeps checking if the variable is equal to 1. When the variable is equal to 0, the process can acquire the lock putting it to 1. Upon releasing the lock, the variable is put to 0. A spinlock isn't handled directly but thanks to an API that offers the functions:

- `void spin_lock(spinlock_t *lock)` that blocks the process, using an active loop, until the lock `lock` can be acquired.
- `void spin_unlock(spinlock_t *lock)` that releases the lock `lock`.

A spinlock is usually used as in Listing 8.1.

```
spinlock_t mr_lock;
DEFINE_SPINLOCK(mr_lock);
spin_lock(&mr_lock);
/* critical region ... */
spin_unlock(&mr_lock);
```

Listing 8.1: Usage of a spinlock.

The spinlock API offers also the functions (one of which is a macro)

```
static __always_inline void spin_unlock_irqrestore(spinlock_t *lock, unsigned long
    flags);
#define spin_lock_irqsave(lock, flags)
```

that can be used to save the current interrupt state. In particular, if the process in kernel mode is sharing some data with an interrupt handler we can

1. Acquire the lock and save the current interrupt state in `flags`. Note that we don't have to pass a pointer to `spin_lock_irqsave` since it's a macro and it can save the state directly in the `flags` variable.
2. Execute the critical section.
3. Release the lock and restore the interrupt state before acquiring the lock.

This is required because a spin lock, when acquired during the handling of an interrupt, disables local interrupts. When unlocking we want to restore the previous state so that when the interrupt is enabled, it will start from where it had been stopped.

A basic implementation of a spinlock is shown in Listing 8.2

```
lock:
... ; prepare '1' in %edx (new)
... ; prepare '0' in %eax (expected)
... ; prepare 'ptr' as &lock
; the actual atomic exchange is called cmpxchg
; (%eax == *ptr) ? {*ptr = %edx, %eax = *ptr} : {%eax = *ptr}

lock cmpxchg %edx, ptr
test %eax, %eax               ; is %eax 0 ?
jnz lock
```

Listing 8.2: Basic implementation of a spinlock in assembly.

Readwrite locks

Readwrite locks are another implementation of spinning locks. The distinctive characteristic of readwrite locks is that they distinguish between readers and writers. In particular, readwrite locks allow multiple readers to read some shared data, but only one writer to modify it.

A readwrite lock is implemented with a counter that counts the number of readers and, when a writer wants to modify the data, it has to check the value of the counter. If the counter is bigger than 0, it has to wait, otherwise it can write. The same happens for the readers. A reader can take the lock if no writer is modifying the data. Readwrite blocks are internally implemented using spinlocks.

Practically, the readwrite API offers different functions for locking and unlocking, depending on whether a process wants to lock for reading or for writing. A reader can use the functions

- `read_lock` to acquire a lock for reading.
- `read_unlock` to release a lock used for reading.

A typical use of readwrite locks for reading is shown in Listing 8.3.

```
1 DEFINE_RWLOCK(mr_rwlock);
2 read_lock(&mr_rwlock);
3 /* critical section (read only) ... */
4 read_unlock(&mr_rwlock);
```

Listing 8.3: Readwrite locks for reading.

Similarly, a writer can use the function

- `write_lock` to acquire a lock for writing and reading.
- `write_unlock` to release a lock acquired with the function `write_lock`.

A typical use of readwrite locks for reading is shown in Listing 8.4.

```
1 DEFINE_RWLOCK(mr_rwlock);
2 write_lock(&mr_rwlock);
3 /* critical section (read and write) ... */
4 write_unlock(&mr_lock);
```

Listing 8.4: Readwrite locks for writing.

Seqlocks

Seqlocks try to minimise the effort that readers must do to access data. Readers are not blocked but at the same time we want to avoid writers to be left starving because blocked by the readers. That is to say, if readers are allowed to read no matter what, a writer might always find a resource occupied by a reader thus being forced to wait. This scenario usually happens when we have plenty of readers and only some writers. The idea behind seqlocks is to block a few readers and still allow some writer to write. For instance, seqlocks are particularly useful to update a global variable called `jiffies`. This variable stores the time elapsed from boot, hence it's updated by a single process (the timer interrupt handler) but can be read by a large number of processes (theoretically by every process in the system). Moreover, `jiffies` is read much more frequently than written. Since we want this variable to be as precise and updated as possible, we can't block the writing process from updating it, hence we have to use seqlocks that block the readers instead. An example of code used to read the value of `jiffies` is shown in Listing 8.5.

```
u64 get_jiffies_64(void) {
    unsigned long seq; u64 ret;
    do {
        seq = read_seqbegin(&xtime_lock);
        ret = jiffies_64;
    } while (read_seqretry(&xtime_lock, seq));
    return ret;
}
```

Listing 8.5: Code to get the value of the variable jiffies.

Having understood that seqlocks are useful in many occasions, let us understand how they work in practice. Sequlocks are implemented with a sequential counter (initially set to 0) that:

- Is incremented when a writer acquires the lock, using the function `write_seqlock`.
- Is incremented when a writer releases the lock, using the function `write_sequnlock`.

Practically, a writer would write something like the code in Listing 8.6 to acquire a lock for writing.

```
write_seqlock(&mr_seq_lock);        // increment seq. counter
/* write lock is obtained... */
write_sequnlock(&mr_seq_lock);      // increment seq. counter
```

Listing 8.6: Code to acquire a seqlock for writing

On the other hand, readers don't have an explicit function to acquire the lock but they have to use a loop in which

1. The reader checks if the counter of the seqlock is equal to 1 (i.e., it's odd) and:
   - If the counter is odd, it start spinning, until the value is even.
   - If the counter is even, the reader can execute its code. The value of the counter is saved in a variable, say `seq`.
2. The reader executes the code it has to execute.
3. The reader reads the value of the sequential counter and checks if it's the same as the one obtain in step 1 (stored in `seq`) and:
   - If the value of the sequential counter is the same, it can go on and the operations it had to do are completed.
   - If the value is different, the operations should be executed again by going back to step 1.

An example of code used to read some data is shown in Listing 8.7. As we can notice, in this implementation of spinning locks, writers are not blocked and they force readers to redo their operations if, in the meanwhile, a writer has modified the data the readers were working on.

```
do {
    // read_seqbegin loops if sequence counter odd
    seq = read_seqbegin(&mr_seq_lock);      // ^
    /* read/copy data here ...                  | check if seq. counter equal. */
} while (read_seqretry(&mr_seq_lock, seq)); // v
```

Listing 8.7: Code to acquire a seqlock for reading

8.3.2 Sleeping locks

Spinning locks (also called active locks) are usually preferred in kernel code over sleeping locks. That being said, sleeping locks can be used, too. In particular, sleeping locks in kernel code are mutexes implemented with semaphores. As always the semaphore

- is decremented when it's locked.
- is incremented every time it's unlocked.

A process can acquire a lock only when its value is greater than 0. As always we have an API that handles this complexity for us, in particular, we can use:

- The functions `down` or `down_interruptible` to acquire the semaphore. The latter allows the sleeping process to be awakened by a generic signal while sleeping while the former is woken up only when the lock is freed.
- The function `up` to release the semaphore.

An example of kernel code used to acquire a semaphore is shown in Listing 8.8.

```
1 /* define and declare a semaphore, named mr_sem, with a count of one */
2 static DECLARE_MUTEX(mr_sem);
3 /* attempt to acquire the semaphore ... */
4 if (down_interruptible(&mr_sem)) {
5     /* signal received, semaphore not acquired ... */
6 }
7 /* critical region ... */
8 /* release the given semaphore */
9 up(&mr_sem);
```

Listing 8.8: Code to acquire a semaphore

Note that, semaphores must be obtained only in process context because the interrupt context is not schedulable.

8.3.3 Read copy update locks

Read-copy-update locks are lockless synchronisation objects. In particular, read-copy-update locks are an evolution of the seqlocks, used when some data is rarely written, in which:

- Readers are not blocked.
- Writers don't disrupt readers.

This means that, differently from seqlocks, readers don't have to repeat an operation if the value of the sequence counter changed while doing such operation. Moreover, writers can work simultaneously with readers. The byproduct of this locking technique is that readers might read old data (i.e., the old value).

Read-copy-update locks allow writers to try and do their updates offline (i.e., not interacting directly with the data) and then atomically update the actual data only at the end. In the meanwhile, readers can read the data modified by the writers. In short, readers:

- Don't use locks.
- Tolerate concurrent writes.

- Can see old data.

Instead, writers:

- Update the data offline using a local copy.
- Atomically update the shared data when finished.

Let us consider an example to better understand this mechanism. Say we have three processes:

- P_1 is a reader.
- P_2 is a reader.
- P_3 is a writer.

Such processes have access to a linked-list L with three elements A, B and C. Consider now the following operations:

- P_1 starts scanning the list at time t_1, following the `next` pointer in the first node.
- P_2 starts scanning the list at time t_0, following the `next` pointer in the first node.
- P_3 wants to remove node B at time t_1. Note that this operation requires two operations:
  1. Move A's `next` pointer to C.
  2. Free B.

Let us now consider the sequence of events that happens when the processes are executing their operations:

1. P_2 scans the list starting from the head, i.e., from node A. At time t_1, after executing this operation, P_2 has a reference to A.
2. P_2 scans the following node, i.e., B and P_1 scans A. At time t_2, after executing these operations, P_2 has a reference to B and P_1 has a reference to A.
3. P_3 atomically modifies the list so that A points to C and B is effectively removed from the list (but not freed). At time t_3, after executing these operations, P_2 has a reference to B and P_1 has a reference to A.
4. If P_3 would have freed B, P_2 would be working on a node which is not valid anymore. This would be wrong, hence P_3 has to wait some time, called **grace period** before freeing B.
5. At the end of the grace period, P_3 can free B hence completing its job.

To sum things up, when a writer has to do some operations:

- It must change the values with atomic operations so that the readers don't see intermediate results.
- It must commit the changes only after the grace period, namely only when all readers have finished reading.

One might now ask how does a process know how long should the grace period last. Linux somehow knows how to compute the grace period since a the scheduler can't switch to another task before ending a read. This means that the grace period should end when the readers are preempted.

A writer can execute an operation at the end of the grace period by specifying a callback which is executed at the end of the grace period.

More schematically, a reader in the example we did before has to

1. Specify it's a reader using the function `rcu_read_lock`.
2. Perform its operations on the elements on the list.
3. Inform the system that it has terminated its operations using the function `rcu_read_unlock`.

What we have just said in words can be seen in code in Listing 8.9.

```
void manipulate_task_list(...) {
    rcu_read_lock(); // inform the reclaimer and disable preemption
    // cannot issue any blocking (sleeping) actions that might switch the context
    for_each_process(p) {
        /* Do something with p */
    }
    rcu_read_unlock();
}
```

Listing 8.9: A read operation using a read-copy-update lock.

A writer in the example we did earlier on has to:

1. Get the readwrite lock to make sequential writes.
2. Delete an element of the list (or in general do some operation).
3. Release the readwrite lock.
4. Register a callback function to say what to do with the element removed from the list. This operation is done using the function `call_rcu`.

This sequence of operations is shown in Listing 8.10.

```
void release_task(struct task_struct *p) {
    write_lock(&tasklist_lock);
    list_del_rcu(&p->tasks); // removal phase. Must allow concurrent read/write
    access
    write_unlock(&tasklist_lock);
    call_rcu(&p->rcu, delayed_put_task_struct); // update phase
}
```

Listing 8.10: A write operation using a read-copy-update lock.

Pros and cons The main advantage of read-copy-update locks is:

- They avoid deadlocks and make 1 read lock acquisition very easy.

However, the main disadvantage is:

- They might delay the completion of destructive operations.

8.4 Multi processors locking

8.4.1 Optimised spinning locks

Say we have 4 CPUs C_0, C_1, C_2 and C_3. Assume now that locking is handled thanks to a lock variable L in memory, initialised with value 0 and shared among the processors. If C_0 sets the variable to 1, the data is updated in the local low level cache of C_0 and it's not updated directly in main memory (usually memory update is handled with a write-back mechanism). This generates an inconsistency between what C_0 and the other processors see. In fact, when C_1 wants to read the lock, it should check if the latest value is in memory or in a cache and, since the value is in a cache, a write-back should be triggered. After the write-back, C_1 can read the value of the lock and put it in its cache. At this point C_0 and C_1's caches are consistent with the value in main memory. Since C_1 read $L = 1$, it must spin until L is equal to 1. The main problem with this algorithm for handling the lock is that, as we've seen in Listing 8.2, the spinning lock uses the `cmpxg` instruction, which is a read and write operation. This means that every time C_1 spins, it must access main memory and write-back the value to other caches (since `cmpxg` also writes). This is very expensive and can worsen the system's performance significantly.

Mellor-Crummey-Scott algorithm

Mellor, Crummey and Scott proposed the following algorithm to solve the aforementioned problem. Say we want to build a lock L shared between 4 processors. The MCS algorithm uses a data structure, called `mcs_spinlock`. Each processor has a local `mcs_spinlock` structure that handles the lock L and can be modified only by the owner CPU. These structures are used to create a queue (implemented as a linked list) where

- The head of the queue points to the `mcs_spinlock` of the processor whose process is holding the lock.
- The tail is made of the `mcs_spinlock`s of the CPUs whose processes are waiting for the lock.

In practice, a `mcs_spinlock` stores:

- A flag `locked` that says if the lock is taken or not.
- A pointer to the next `mcs_spinlock` in the queue.

The implementation of a `mcs_spinlock` is shown in Listing 8.11.

```
1 struct mcs_spinlock {
2   struct mcs_spinlock *next;
3   int locked;                    /* 1 if lock acquired */
4   int count;                     /* nesting count, see qspinlock.c */
5 };
```

Listing 8.11: The definition of the qspinlock structure.

Moreover, the MCS algorithm uses another data structure, called `qspinlock` that has

- A flag `taken` that tells the processes if the lock is taken or not.
- A pointer to the `mcs_spinlock` of the last processor that requested to acquire the lock. Basically, the pointer points to the tail of the queue.

This data structure is shared among all processors and can be accessed by whatever process to check if the lock is free or not.

Let us now simulate two processes that try to acquire a lock to see how the data structures are handled.

1. Initially the lock is free.
   (a) `qspinlock` has `taken=0` and the tail points to `NULL`.
   (b) Every processor's `mcs_spinlock` is initialised with `next=NULL` and `locked=0`.
2. C_1 takes the lock.
   (a) C_1 atomically sets `qspinlock.tail` to its `mcs_spinlock` structure and reads the old value of the `last` pointer.
   (b) The flag `qspinlock.taken` is set to 1 and the tail points to C_1's private `mcs_spinlock`.
   (c) C_1 sees that the old value of `qspinlock.tail` was null, hence it can take the lock.
3. C_3 wants to acquire the lock but the lock is taken.
   (a) C_3 atomically sets `qspinlock.tail` to C_3's `mcs_spinlock` with an exchange instruction that returns the old value of `qspinlock.tail`.
   (b) Since `qspinlock.tail` is not null, it has to set `qspinlock.tail.next` to its `mcs_spinlock` struct. Namely,C_1's `mcs_spinlock.next` points to C_3's `mcs_spinlock`, which has been added to the queue.
   (c) C_3 spins on the `locked` value of its `mcs_spinlock`.
4. C_1 releases its lock.
   (a) C_1 sets its `mcs_spinlock.locked` to 0.
   (b) C_1 follows the `next` pointer inside its `mcs_spinlock`, which points to C_3's `mcs_spinlock`.
   (c) C_1 sets C_3's `mcs_spinlock.locked` to 1 so that C_3 can use the lock.

What follows is a more general analysis of the Mellor-Crummey-Scott algorithm.

Lock When a process running on processor C_1 wants to acquire a lock L it:

1. Checks if the value `qspinlock.taken` is set:
   - If `qspinlock.taken=0`:
     (a) C_1 sets `qspinlock.taken=1`.
     (b) C_1 sets `qspinlock.tail` to its `mcs_spinlock`.
     (c) C_1 sets its `mcs_spinlock.locked=1`.
     (d) C_1 sets its `mcs_spinlock.next=NULL`.
   - If `qspinlock.taken=1`:
     (a) C_1 sets `qspinlock.tail.next` to its `mcs_spinlock`.
     (b) C_1 sets `qspinlock.tail` to its `mcs_spinlock`.
     (c) C_1 sets its `mcs_spinlock.locked=0`.
     (d) C_1 sets its `mcs_spinlock.next=NULL`.
2. Spins on its `mcs_spinlock.locked`. If the `qspinlock` was not taken, it simply executes some operations.

Unlock When a process running on processor C_1 wants to release a lock L it:

1. C_1 sets its `mcs_spinlock.locked=0`.
2. C_1 sets its `mcs_spinlock.next.locked=1`.
3. C_1 sets its `mcs_spinlock.next=NULL`.

8.5 Memory models

In multi-threading multi-core systems, each processor can have a different view of what's happening on the other processors. This is especially true if we consider a NUMA architecture (cores divided in groups and each group has privileged access to a bank of RAM, more will be explained later on) with multiple caches and several optimisations like write buffers and speculation. The behaviour of a system with respect to what each processor is ensured to see is called memory model. In other words, a memory model defines the behaviour of the visibility (and consistency) of the operations done by one thread from another thread.

Let us consider the following example to better understand memory models, why they are relevant and what makes a memory model strong. Say we have two threads that execute the functions in Listing 8.12. In this example, thread 1 writes values to memory (i.e. to the global variables `x` and `y`) while thread 2 reads values from memory. We will indicate these operations as

$$W_p(x)n$$

to say that processor p has written the value n to variable x and

$$R_p(x)n$$

to say that processor p has read the value n from variable x. In our example we can drop the processor number p, since it's clear that processor 1, which executes thread 1, only writes and processor 2, which executes thread 2, only reads.

```
1  int x;
2  int y;
3
4  void thread1() {
5      x = 1;
6      y = 1;
7  }
8
9  void thread2() {
10     int r1 = y;
11     int r2 = x;
12 }
```

Listing 8.12: Threads used for the initial example of memory models.

Depending on the memory model, each thread can see different things than what is actually happening. For instance, thread 1 might execute the first two instructions but thread 2 might still see `x==0` and `y==0`. Defying a memory model means defining all the possible interleaving of operations and states seen by each thread. The higher the number of combinations, the weaker the model, since we have many different possible interleaving that each thread can see (and we don't know which it might see). On the other hand, if each thread could see only one sequence of operations, we would have the best memory model, since we are always sure about what each thread sees.

Before presenting different memory models, let us clarify some terminology and notation. We say that an instruction I_1 precedes (or happens-before) in program order an instruction I_2, and we write

$$I_1 <_p I_2$$

if I_1 is written before I_2 in the source code. We say that an instruction I_1 precedes (or happens-before) in memory order an instruction I_2, and we write

$$I_1 <_m I_2$$

if the effects of instruction I_1 are seen in memory before those of instruction I_2. In some architectures, like x86, if an instruction comes before in program order, it also comes before in memory order. This isn't however true in all cases, in fact, in ARM architectures, program order and memory order might differ.

8.5.1 Sequential consistency

The stricter and stronger memory model is sequential consistency.

> **Definition 8.1** (Sequential consistency)**.** *A multi-core processor is sequentially consistent if and only if, for all pair of instructions $(I_{p,i}, I_{p,j})$, program order implies memory order. Namely a multiprocessor is sequentially consistent iff,*
>
> $$I_{p,i} <_p I_{p,j} \implies I_{p,i} <_m I_{p,j} \quad \forall (I_{p,i}, I_{p,j})$$
>
> *where p is the index of the processor that executes the instruction and i, j are instructions' indexes.*

In practice, the operations of each processor appear in memory and are seen by other processors in the order specified by its program. The possible sequences of instructions are all the interleaving that preserve each threads' sequence of instructions. That is to say, if $I_{1,1} <_p I_{1,2}$, every other process has to see $I_{1,1} <_m I_{1,2}$. Note however that other threads' instructions might be inserted between $I_{1,1}$ and $I_{1,2}$, hence we can have

$$I_{1,1} <_m I_{2,1} <_m I_{1,2}$$

Going back to the example in Listing 8.12, sequential consistency admits only 6 sequences, which are:

- $W(x)1 <_m W(y)1 <_m R(y)1 <_m R(x)1$
- $W(x)1 <_m R(y)0 <_m W(y)1 <_m R(x)1$
- $W(x)1 <_m R(y)0 <_m R(x)1 <_m W(y)1$
- $R(y)0 <_m W(x)1 <_m W(y)1 <_m R(x)1$
- $R(y)0 <_m W(x)1 <_m W(y)1 <_m R(x)1$
- $R(y)0 <_m R(x)0 <_m W(x)1 <_m W(y)1$

Note that, this model is purely theoretical and can be applied only when all processors are directly attached to the same shared memory.

8.5.2 Total store order

The total store order model, as shown in Figure 8.4 for two processors, uses write buffers instead of writing directly to the shared memory. In particular, each thread is connected to a write buffer

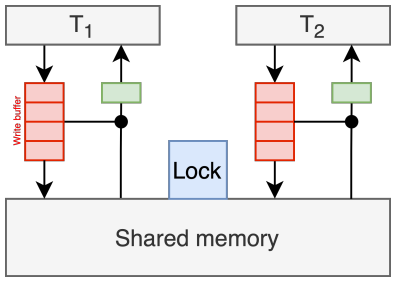


Figure 8.4: The total store order memory model.

that speeds up the writing process for a thread but that delays the writing process to memory. Each processor writes to its write buffer which will asynchronously write to the shared memory. The write buffer can is also used to read values. More precisely, a processor checks if the buffer contains the data it has to read and then, if not in the buffer, it checks in the shared memory. Moreover, instructions that require synchronisation have to acquire a global lock. Because of the write buffer:

- A thread might not see the result of a write instruction because it's still in the write buffer, even if the instruction has been executed.
- A thread might see the result of its own writes before the other threads.

Since the write buffer is a FIFO buffer, the order of the writes is seen correctly by every thread, however, different threads might read different values from memory. For instance, we can obtain a sequence

$$W(x)1 <_m R(y)0 <_m R(x)0 <_m W(y)1$$

because the first write is added to the writing queue and, before it's actually propagated to the shared memory, the second processor reads the old value from shared memory. Moreover, because of the latency introduced by the write buffer, the memory might see that a thread writes before reading, even if the program contained first a write and then a read. The program in Listing 8.12 doesn't create many problems, however it isn't the same for the program in Listing 8.13, which highlights the problem just discussed.

```
1  int x, y;
2
3  void thread1() {
4      x = 1;
5      int r1 = y;
6  }
7
8  void thread2() {
9      y = 1;
10     int r2 = x;
11 }
```

Listing 8.13: Threads used for the example on the total store order memory model.

If we run this problem on a sequentially consistent processor, we can't have both `r1` and `r2` equal to 0. On TSO processors this is instead possible. In particular we can have a sequence

$$W_1(x)1 <_m W_2(y)1 <_m R_1(y)0 <_m R_2(x)0$$

This happens because of write buffers which delay the writes, hence, from the memory point of view, a processor first reads some values and then writes them (because of the delay of the buffer). In practice:

1. Thread 1 executes `x=1` and the new value of `x` is in the write buffer.
2. Thread 2 executes `y=1` and the new value of `y` is in the write buffer.
3. Thread 1 reads `y`, whose value is still `0` because the value written by thread 2 is still in thread 2's write buffer.
4. Thread 2 reads `x`, whose value is still `0` because the value written by thread 1 is still in thread 1's write buffer.
5. Thread 1's write buffer writes $x = 1$ to the shared memory.
6. Thread 2's write buffer writes $y = 1$ to the shared memory.

8.5.3 Partial store order

The Partial Store Order (PSO) memory model is the weakest among those we have analysed. In this case, there is no shared memory and each processor reads from and writes to its own copy of the memory. Writes are then propagated independently (i.e., one processor sends to the other independently) to the other memory copies through write buffers. Note that the writes done by a single processor can be reordered when sent to other processors' memories. This means that other processors can see writes in a different order with respect to the one in which they were issued. Reads might be delayed after writes, too. Hardware threads can perform writes out-of-order, or even speculatively, i.e., before preceding conditional branches have been resolved. Any local reordering is allowed unless otherwise specified. This memory model is shown in Figure 8.5 for a dual-processor system.

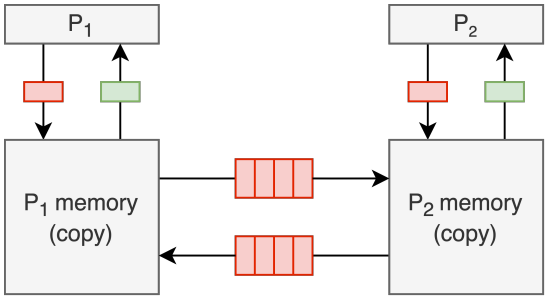


Figure 8.5: The partial store order memory model.

If we consider the program in Listing 8.12, contrary to sequentially consistent models, we can obtain the following sequence of instructions

$$W(y)1 <_m R(y)1 <_m R(x)0 <_m W(x)1$$

because the first and second instruction of thread 1 can be swapped when sent from thread 1 memory to thread 2's.

ARM

This memory model is used by ARM processors. In particular, for ARM processors:

- Within a single thread, there are (implicit) dependencies to enforce orderings (address, control and data dependencies) that make programming more intuitive.
- The memory system does not guarantee that a write becomes visible to all other hardware threads simultaneously. This behaviour is called non-multicopy- write-atomicity.

8.5.4 Data races

Some behaviour of TSO and PSO memory models are undesired hence it'd be useful to tame them. Luckily, there exist mechanism that can be enforced to partially solve the ordering problems caused by such memory models.

Memory barriers

Memory barriers are synchronisation instructions that allow enforce an order in which writes have to be issued, hence solving the write delay of the TSO memory model. More precisely, **barriers are synchronisation instructions that force memory order to coincide with program order by flushing the write buffer** (hence making memory up to date). Adding a memory barrier after every instruction is the same as enforcing sequential consistency, however it will be too costly. The best way to use barriers is therefore to put them only when it's fundamental to enforce a specific sequence of instructions.

Note that if we want to use memory barriers, we have to disable compiler optimisations which can swap instructions. In fact, we have to ensure that the barrier instruction is placed in the correct order.

Synchronisation instructions

If we want to enforce some order of instructions without disabling optimisations, we have to introduce the idea of ownership of an instruction. Before analysing this concept and how it's implemented, we have to define what is a data race and why it's relevant.

> **Definition 8.2** (Data race)**.** *A data race is a condition in which the correctness of a program depends uncontrollably on the memory model and on the scheduling of instructions.*

In other words, when we have a data race, depending on how instructions are scheduled (i.e., what instruction is executed before), we might have have a correct or incorrect behaviour. Consider for the program in Listing 8.14 and the PSO memory model.

```
1 int data, ready;
2
3 void thread1() {
4     data = 1;
5     ready = 1;
```

```
6 }
7
8 void thread2() {
9     if (ready == 1) {
10        r1 = 3 / data;
11    }
12 }
```

Listing 8.14: Threads that can lead to a data race.

In the program above, if the order in which the instructions `ready=1` and `data=1` are issued is inverted, we might get an error, in fact we can get the following sequence of instructions

$$W(ready)1 <_m R(ready)1 <_m R(data)0 <_m W(data)1$$

which results in executing the body of thread 2's `if` block and executing a division by 0.

Data races can be solved using synchronisation operations that can be used even with compiler and hardware optimisations. These operations solve TSO and PSO problems, too, since data races are generated by swapping instructions (and the PSO data model allows it to happen). A synchronisation operation is an instruction, which can also be a normal instruction, that enforces a certain memory model. A synchronisation instruction is an instruction such that, if a thread sees the effect of the synchronisation instruction, then it's sure that everything that's happened before has been issued in program order. If the synchronisation instruction is a write operation, it's called **release operation**. The dual of the release operation is called **acquire operation** and it's a read operation. An acquire operation must be used by an observer thread to observe the release operation. In particular, the acquire operation has to be a read or test-and-set instruction that must have the acquire semantics (i.e., that is able to observe the release operations around the system). The release and acquire instructions are used to acquire and release ownership of a sequence of instructions. If an acquire operation is used after a release operation, then we are sure to establish an happens-before relationship between the release and the acquire operation. Since happens-before relationship remove data races, we can use release and acquire operations to synchronise sequences of instructions. Moreover, we have the following result:

Theorem 8.1. *If a program has no data races, then the program will appear as if it was sequentially consistent.*

On the other hand, any program with a race anywhere in it falls into undefined behaviour (DRF-SC or catchfire).

8.5.5 Software memory models

Compilers can reorder instructions, for performance reasons, such that it might appear that the machine had a weaker memory model. This happens because the compiler might not know on that the program is running on a multi-core machine. The languages that are able to reorder instructions are called language memory models. More precisely, a language memory model is an higher level language that must give to the programmer a way to enforce happens-before relations just as it is done at the ISA level.

C++11 memory model

The C++11 memory model provides functions to manipulate synchronisation data (atomics) enforcing:

- **Strong synchronisation** (sequentially consistent).
- **Weak synchronisation** (acquire/release, enforces coherence-only).
- **No synchronisation** (relaxed, for hiding races).

```
int done;
atomic_store(&done, 1);

/* code */

while(atomic_load(&done) == 0) { /* loop */ }
```

Listing 8.15: C++11 memory model example.

The Linux Kernel Memory Model

The Linux Kernel Memory Model (LKMM) is a memory model for the C programming language used in the Linux kernel. When compiling Linux, some optimisations might be used, hence we have to be able to enforce happens-before relationships. Since Linux works on different architectures, the LKMM is the weakest memory model in common among the memory models of all architectures. This means that LKMM can't offer more than a PSO machine, since Linux also runs on ARM processors, which have a PSO memory model.

As we have seen, we can use the `smp_store_release` and `smp_load_acquire` instructions to enforce the happens-before relationship between two instructions. The LKMM also introduces two macros `WRITE_ONCE(var, value)` and `READ_ONCE(var)` which prevent the compiler from:

- Reordering reads or writes.
- Omitting reads for known values.
- Doing too many reads when register spilling is needed.

Linux also has the `atomic_t` (32bit) and `atomic64_t` types. Operations on these types are always guaranteed to be not interruptible.

```
atomic_t v = ATOMIC_INIT(0);
atomic_set(&v, 4);
atomic_add(2, &v);
atomic_inc(&v);
```

Listing 8.16: Operations on atomic types.

RMW instructions on `atomic_t` are implemented through lockless techniques (which might involve loops), and/or instructions with LOCK prefix.

Part IV

Virtualisation

Chapter 9

Linux virtual memory

9.1 Virtual memory

Modern operating systems like Linux use virtualisation to handle memory in different processes. The idea is to let every process think that it can use the whole physical memory. Each process is therefore assigned a virtual memory space in which each page of the virtual space has a number. Virtual pages that are actually used by a process are then mapped by the operating system to physical pages. Virtual memory mapping has many advantages, some of which are:

- Virtual pages can contain read-only data shared by multiple processes on the same physical page. For instance, we can put the `libc` in a fixed location in physical memory and then map the virtual pages of each process to that physical area.
- We allow processes to think independently from the environment they are in.

Memory mapping is handled by a data structure in the kernel. In particular, the task structure of every process has a pointer `mm` to the page table, formally to a `mm_struct`, used for that process. The structure that handles mapping, i.e., the one pointed to by `mm` is called **page directory** or **page table**.

```
struct mm_struct {
    /* omitted fields */

    pgd_t * pgd;  /* a pointer to the page table */

    /* omitted fields */
}
```

When a process is put in execution, the value of `mm` is stored in a special register, called virtual memory base register, which is used for faster page translation. For instance, in x86, this register is `CR3`. Note that a process's page table contains both user-space and kernel-space pages and, as we should expect, the last ones can only be accessed when in kernel space. The page directory can be seen as an array even if it's usually implemented as a hierarchical structure.

The standard page size is 4 KB, however, processes might require allocating even pages with bigger sizes like 2 MB or 1 GB. In this case, we obtain a better hit rate in the Translation Look-aside Buffer.

9.2 Virtual memory areas

Virtual memory works and is efficient because a process doesn't usually use the whole virtual memory. This means that only some virtual pages have to be mapped to physical pages. Linux uses Virtual Memory Areas (VMAs) to understand if a page, or better a Virtual Page Number (VPN), is valid and mapped. Virtual memory areas are ranges of addresses, with common restrictions (read-/write/execute). Before addressing VMAs, let us say that virtual memory can be split into two parts:

- The **kernel** part. This part, which is usually located in the upper part of virtual memory, contains everything related to kernel-space code. This means that the kernel works with virtual pages, too. In fact, it works only on rare occasions (e.g., during startup) directly with physical pages.
- The **user** part. This part, which is usually located in the lower part of virtual memory, contains everything related to user-space code.

Note that during a context switch, only the user part of virtual memory is changed, in fact, the kernel part is the same for every process, hence the OS doesn't have to change it. The structure of a process' virtual memory is shown in Figure 9.1.

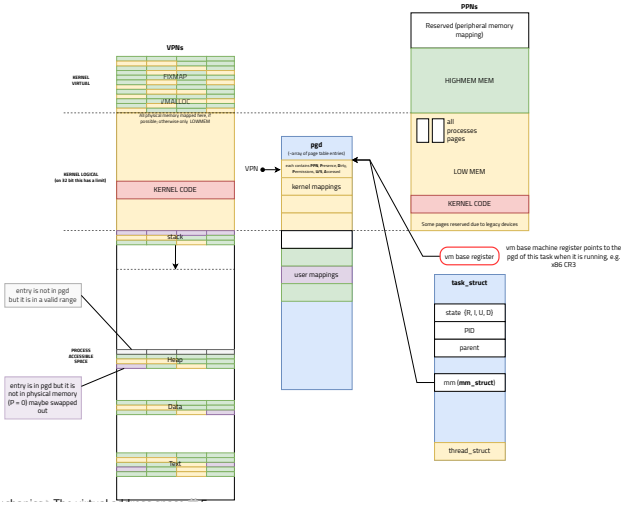


Figure 9.1: The structure of a process' virtual memory.

9.2.1 Kernel virtual memory

The kernel virtual memory can be divided into two parts:

- **Kernel virtual memory**.
- **Kernel logical memory**.

Note that the former is placed on top of the latter.

Logical memory

The logical memory area is used for remapping the pages of physical memory. In other words, this is basically a copy of the pages of kernel memory that are in the physical memory. Note that this is true only for 64-bit processors (for 32-bit processors we wouldn't have enough memory for storing a copy of the kernel pages). This area of memory is contiguous in physical memory and in virtual memory. This is very important when we want to allocate pages here. In fact, we can't use the same primitives and syscalls we use in user space but we have to use the function

```
void * kmalloc (size_t size, gfp_t flags);
```

This area of virtual memory contains, among other things:

- Kernel code.
- Memory areas used for peripherals.

Virtual memory

The kernel virtual memory area is used when the kernel needs some kernel pages from physical memory and no contiguous pages are available. In other words, this is used when the kernel needs some pages that have to be contiguous in virtual memory but that are not in physical memory. This area should be used when big allocations are required, since the kernel can't ensure that such blocks are contiguous in physical memory, too. As for logical memory, we have a specific function to allocate pages in this section, which is

```
void * vmalloc (unsigned long size);
```

9.2.2 User virtual memory

The lower part of virtual memory is dedicated to user space. This section is divided into multiple parts, called Virtual Memory Areas (VMAs). The main VMAs are:

- **Stack**. This is the area of memory dedicated to function activation records. This area grows towards lower addresses.
- **Memory mapped areas**. This area contains libraries and memory-mapped files.
- **Heap**. This is the area of memory in which processes can allocate blocks smaller than a page using the `malloc` function. This area grows towards higher addresses.
- **Bss**. This section contains uninitialised static data, both variables and constants used by the process's code.
- **Data**. This section contains global variables and static local variables used by the process's code.
- **Rodata**. This section contains the read-only variables used by the process's code.
- **Text**. This section contains the code executed by the process.

Note that these aren't the only VMAs, but are the most common. VMAs are used to understand in what state a page is. In particular, a virtual page can be:

- In the page directory with a physical address mapping.
- In the page directory but without physical address mapping. Albeit it might seem wrong, this state is valid and in particular, it represents a page that has been moved to disk because of lack of memory.
- Not in the page directory but still valid. These are temporary situations that happen, for instance, when increasing the heap.

Virtual memory areas are used just for distinguishing among these cases. More precisely, each VMA is represented by a `vm_area_struct` that stores:

- The **permission** of that area (`VM_READ`, `VM_WRITE`, `VM_EXEC`).
- The address where the area **starts**.
- The address where the area **ends**.
- A pointer to a file, called **backing store**, if it exists, from where the data has been taken. For instance, the `.txt` area has a pointer to the source code. Some areas don't have a source file, hence they are called anonymous VMAs. Even these areas might be removed from memory (for lack of memory) and written to disk (i.e., used as swap memory), however in these cases the file where they are written doesn't have a specific name. In particular, anonymous areas (e.g., heap, stack and bss) are originally mapped into a zero page maintained by the OS and mapped as read-only. A `VMA_GROWSDOWN` area is initialised with a certain number of zero pages and it can grow when the last page is accessed. This technique is used, for instance, for the stack but, for other anonymous VMAs, we have to use explicitly `brk` or `sbrk`.

A list of `vm_area_structs`, one for each VMA of the process, is stored in the process `task_struct`, in particular, in the field `mm` (which points to a `mm_struct`). When a process requests a page, the OS can look into the list of `vm_area_structs` to find the right VMA in which the page is and understand if the page should be searched in memory or in storage. Note that Linux actually uses a hierarchical structure to organise `vm_area_structs`, since it allows to search for VMAs in logarithmic time. In particular, Linux uses a red and black tree which uses the end address as the key.

```
1  struct vm_area_struct {
2    /* The first cache line has the info for VMA tree walking. */
3
4    unsigned long vm_start;   /* Our start address within vm_mm. */
5    unsigned long vm_end;   /* The first byte after our end address
6               within vm_mm. */
7
8    /* linked list of VM areas per task, sorted by address */
9    struct vm_area_struct *vm_next, *vm_prev;
10
11   /* Second cache line starts here. */
12
13   struct mm_struct *vm_mm;  /* The address space we belong to. */
14
15   /*
16    * Access permissions of this VMA.
17    * See vmf_insert_mixed_prot() for discussion.
18    */
19   pgprot_t vm_page_prot;
20   unsigned long vm_flags;   /* Flags, see mm.h. */
21
```

```
22   /*
23    * A file's MAP_PRIVATE vma can be in both i_mmap tree and anon_vma
24    * list, after a COW of one of the file pages.  A MAP_SHARED vma
25    * can only be in the i_mmap tree.  An anonymous MAP_PRIVATE, stack
26    * or brk vma (with NULL file) can only be in an anon_vma list.
27    */
28   struct list_head anon_vma_chain; /* Serialized by mmap_lock & * page_table_lock
       */
29   struct anon_vma *anon_vma;  /* Serialized by page_table_lock */
30
31   /* Function pointers to deal with this struct. */
32   const struct vm_operations_struct *vm_ops;
33
34   /* Information about our backing store: */
35   unsigned long vm_pgoff;   /* Offset (within vm_file) in PAGE_SIZE units */
36   struct file * vm_file;    /* File we map to (can be NULL). */
37   void * vm_private_data;   /* was vm_pte (shared mem) */
38
39     /* Other missing fields */
40 } __randomize_layout;
```

Listing 9.1: The structure used to handle a VMA.

A graphical representation of the virtual memory areas in a process' virtual memory and their backing store (or swap files) is shown in Figure 9.2.

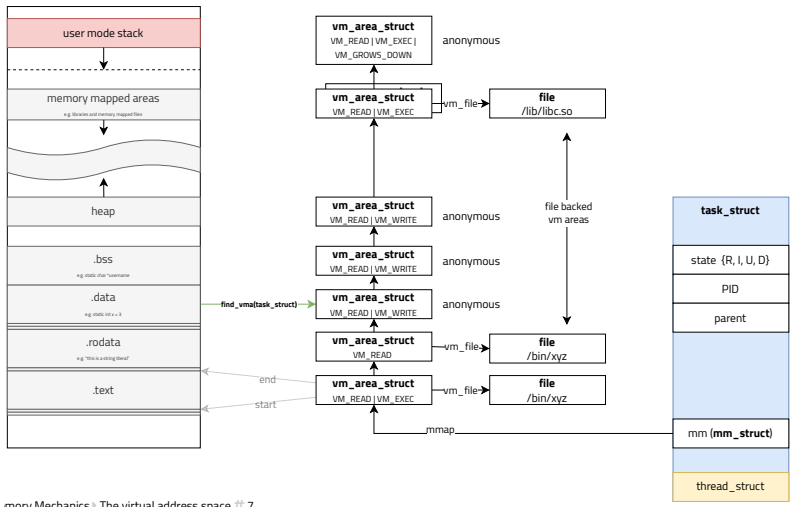


Figure 9.2: A process' virtual memory areas.

Page faults

Now that we know the structure of a process memory we can describe how VMAs are used to understand the state of a page (i.e., valid but not mapped to physical, valid but in a backing store, invalid). Note that we aren't considering the case in which the page is valid and in memory since the OS simply checks that the virtual page is mapped to a physical page. To understand what state a page is in, we have to analyse what happens when the OS can't find a page in the page directory

either because invalid, not mapped or in the backing store (remember that the backing store is on disk, hence isn't mapped in the page directory). Say a process wants to execute the instruction

```
mv rax, [0x4555ad34]
```

but the page at address `0x4555ad34` isn't in the page directory. In these cases, we talk about page faults. When a page fault occurs, the OS:

1. Scans the list of `vm_area_structs` looking for the VMA where the faulty address is. This action is executed using the

   ```
   extern struct vm_area_struct *
   find_vma(struct mm_struct * mm, unsigned long addr);
   ```

   function which returns a pointer to the `vm_area_struct` where address `addr` belongs.

   - If the address isn't found in any `vm_area_struct`, then the address must be wrong (e.g., belonging to an area that can't be accessed) and a `SEGFAULT` is returned by the page fault handler.
   - If the address is found in a `vm_area_struct`, then it can continue allocating the page.

   Note that, if the page is valid, it's for sure in the list of `vm_area_structs`.

2. Checks the permissions of the faulty page:

   - If the permissions are not ok (e.g., an instruction wants to write to a page which is read-only), a `SEGFAULT` is returned by the page fault handler.
   - If the permissions are fine, it goes on.

3. Calls the `handle_mm_fault` function. This function walks the page directory to check if the page is already there. If not, the function creates an entry, hence a mapping from a virtual page to a physical page. Namely, this function tries to understand if the page fault occurred because of a valid page in the backing store (if an entry is found in the page directory) or because of a valid page not mapped (if no entry is found).

4. Invokes the `handle_pte_fault` function which:

   (a) initiates a read from the backing store and, if necessary, flushes the TLB. This operation is done if the page fault occurred because of a valid page in the backing store.

   (b) provides an anonymous page or recognises that the VMA is writable but the page isn't; this means copy on write hence it duplicates the physical page.

Let us understand this mechanism through an example. In particular, we will analyse what happens when a program tries to enlarge its heap, hence allocating a virtual page which hasn't been mapped to a physical page, yet. The program:

1. Calls the `brk` function to enlarge the heap.

2. The `brk` enlarges the VMA but does not allocate pages in physical memory.

3. The process tries to access the newly created memory (whose pages haven't been mapped to a physical page). In this phase, the `find_vma` is invoked to find the `vm_area_struct` to which the heap address belongs (i.e., it has to look for the `vm_area_struct` of the heap).

4. Calls the `handle_mm_fault` and `handle_pte_fault` to build the mapping from the virtual page to the physical page.

User space memory mapping

Virtual memory areas can be explicitly created by a process, too. A process can use the

```
void *
mmap (void *addr, size_t length, int prot, int flags, int fd, off_t offset);
```

function on an already opened file descriptor `fd` which creates a mapping from the process's virtual address space to the physical memory area where the file described by `fd` is.

Chapter 10

Linux physical memory

10.1 Page allocation

10.1.1 Non-Uniform Memory Access

After having analysed how Linux handles the virtual address space of a program, let us describe how physical memory is handled. First, let us describe the physical architecture of RAM. In particular, we will focus on the Non-Uniform Memory Access RAM, which defines how RAM is accessed by different processors in a multi-core system. In multi-core systems, RAM is organised in banks. Each bank is close to a set of cores which have faster access to that bank. Such a set of cores is called a NUMA node. Let us consider an example to make things clearer. Consider a system with 32 cores and 4 memory banks. Cores are divided into 4 NUMA nodes, with 8 cores each, directly connected to a memory bank. Since memory isn't unified (i.e., we don't have a single bank of RAM, but it's distributed in different banks), the memory access time varies depending on the distance between a core and the node where the data is. The architecture in this example is shown in Figure 10.1. To handle this issue, Linux uses a node-local allocation policy to allocate memory from the node closest to the running CPU. This comes from the fact that a process will likely run on the same core, hence it's better to put the data it uses on the closest node available.

10.1.2 Linux memory representation

Linux represents the memory structure (i.e., NUMA nodes) using a `pgdat_list` structure which contains a list of `pg_data_t` structures, each of which represents a NUMA node. Each `pg_data_t` contains different information about a node.

```
1 typedef struct pglist_data {
2     struct zone node_zones[MAX_NR_ZONES];
3     int nr_zones;
4     struct page *node_mem_map;
5     unsigned long node_start_pfn;
6     unsigned long node_spanned_pages;
7     struct lruvec __lruvec;
8 } pg_data_t;
```

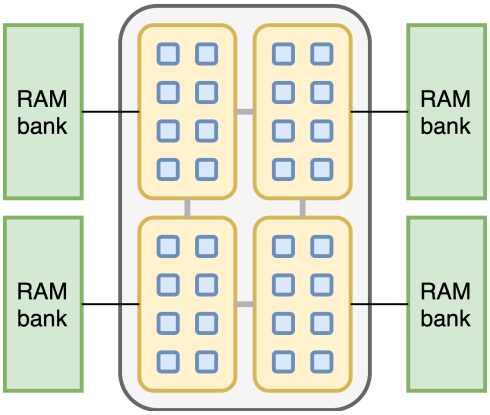


Figure 10.1: An example of NUMA RAM architecture with 32 cores, divided into 4 NUMA nodes (with 8 cores each) and 4 memory banks.

Zones

One thing Linux does to handle NUMA nodes is splitting the RAM bank directly attached to a NUMA node into zones, in fact, the `pg_data_t` structure has an attribute `node_zones` which stores a list of `zone` structures, each representing a memory zone. In particular, we can define three different zone types:

- `ZONE_DMA`, which is a zone that can be used for peripherals.
- `ZONE_NORMAL`, which is a zone that can be used for normal memory allocations.
- `ZONE_HIGHMEMORY`, which is not directly mapped by the kernel.

Each zone `zone` stores a list of `free_area` structures.

```
struct zone{
    unsigned long       _watermark[NR_WMARK];
    unsigned long       zone_start_pfn;
    unsigned long       spanned_pages;
    const char          *name;
    struct free_area    free_area[MAX_ORDER];
};
```

This list is used to understand which pages are free. Before understanding how `free_area`s are used to manage free pages, let us define some terms. If we have n contiguous free physical pages, we call them a n page block. In other words, a n page block is a sequence of free pages without any allocated page in between. Let us now go back to `free_area`s. The list of `free_area`s contained in a `pg_data_t` has 10 `free_area`s, numbered from 1 to 2^9.

```
struct free_area {
  struct list_head  free_list[MIGRATE_TYPES];
  unsigned long    nr_free;
};
```

The `free_area` with number n has a list whose values are references to the beginning of a n page block. This means that taking a `free_area` with number n, we can access all the blocks of pages with n free contiguous pages. In some cases we might have blocks with $n \neq 2^i$; this isn't however a

problem since they are always spitted into blocks with 2^i pages. The reason why this happens will be clear when the Buddy algorithm is presented. For now, we have to remember that a `free_area` with the number n has a list of all blocks of n contiguous free pages.

Moreover, each zone contains one or more watermarks. A watermark represents how many free pages are available in that zone. In particular, a watermark can take the values:

- **high** if there are a lot of free pages.
- **low** if the number of free pages is getting low.
- **min** if the number of free pages is critically low.

The mechanism with which this structure is handled will be explained later on.

Lruvec

Other than zones, the `pg_data_t` structure of a NUMA node contains also a `lruvec` structure.

```
struct lruvec {
  struct list_head    lists[NR_LRU_LISTS];
  /* per lruvec lru_lock for memcg */
  spinlock_t      lru_lock;
  /*
   * These track the cost of reclaiming one LRU - file or anon -
   * over the other. As the observed cost of reclaiming one LRU
   * increases, the reclaim scan balance tips toward the other.
   */
  unsigned long     anon_cost;
  unsigned long     file_cost;
  /* Non-resident age, driven by LRU movement */
  atomic_long_t     nonresident_age;
  /* Refaults at the time of last reclaim cycle */
  unsigned long     refaults[ANON_AND_FILE];
  /* Various lruvec state flags (enum lruvec_flags) */
  unsigned long     flags;
#ifdef CONFIG_LRU_GEN
  /* evictable pages divided into generations */
  struct lru_gen_struct   lrugen;
  /* to concurrently iterate lru_gen_mm_list */
  struct lru_gen_mm_state   mm_state;
#endif
#ifdef CONFIG_MEMCG
  struct pglist_data *pgdat;
#endif
};
```

This structure is used by the algorithm that manages page swaps from memory to disk to keep track of the activity of the physical pages in memory. In particular, it must distinguish between active and inactive pages since inactive pages should be removed from memory before active ones. Formally, the `lruvec` contains lists of pages categorised by their activity (i.e., we have a list of all active pages, one of all inactive pages and so on). Pages can be in the following states:

- `NACTIVE_ANON`
- `ACTIVE_ANON`
- `INACTIVE_FILE`

- `ACTIVE_FILE`
- `UNEVICTABLE`

10.1.3 Buddy algorithm

Now that we know how memory is represented by Linux, we can describe how it handles memory. In particular, we have to understand how physical pages are allocated and how they are removed (and stored in the backing store) when the memory is full.

Allocation algorithm

Assume that memory always has free pages and that we never have to swap pages in memory to their swap space or backing store. We will consider page removal later on.

The algorithm that handles physical page allocation is called the buddy algorithm. The idea behind the buddy algorithm is to never split blocks of contiguous pages. If that's impossible and a block has to be split, two buddy blocks are created so that they can be rejoined when both buddies are freed. In other words, when needed, a block is divided into two sub-blocks, called buddies. When both buddies are free, they are rejoined to recreate the original block. It's important to notice that, when two buddies are free, Linux doesn't see them as two free small blocks but as one single block.

When the allocator needs a block of n pages, it checks, using the list of `free_area`s if there exists any, otherwise, it splits a block of more pages until it gets a block of the size it needs.

Let us now consider an example to better understand the buddy algorithm. Let us consider a zone made of one block of 16 pages and the following sequence of events:

1. The program requests 1 page. The allocator splits the block of 16 pages into two buddies B_1^8 and B_2^8 of 8 pages each. The first buddy is further divided into two buddies of 4 pages each, hence obtaining B_1^4 and B_2^4. The first block of 4 pages is divided into two B_1^2 and B_2^2 blocks of 2 pages. The first block of 2 pages is divided into two buddies of 1 page, namely B_1^1 and B_2^1. The zone is now
$$B_1^1 \ B_2^1 \ B_2^2 \ B_2^4 \ B_2^8$$
and block B_1^1 is allocated (while the others are free).

2. The program requests 2 pages. Since there exists a free block of two pages, namely B_2^2, it's allocated.

3. The program requests 2 pages. Since no free block of 2 pages is available, the allocator has to split a block with more pages. In particular, block B_2^4 is split into B_3^2 and B_4^2, which are buddies, and B_3^2 is allocated. The current situation in memory is
$$B_1^1 \ B_2^1 \ B_2^2 \ B_3^2 \ B_4^2 \ B_2^8$$

Watermarking

Let us now analyse how Linux estimates the number of free pages available in a zone. As we have seen, each zone has a `_watermark` that is an estimate of the number of free pages. Let us understand how Linux reacts to changes in the watermark of a zone:

1. Say we start from a high watermark. In this situation, the page allocator can freely allocate pages since there are plenty of them.

2. When the zone goes beyond the high watermark, the system starts having fewer free pages, but the allocator can still work without many concerns.
3. As soon as the low watermark is passed, the `kswapd` demon is awakened by the allocator. This demon has to asynchronously swap inactive pages from memory to their backing store (or swap spaces for anonymous pages) since the system is getting low on free memory pages.
4. When the min watermark is passed, the allocator takes the place of the `kswapd` and swaps the pages themselves.
5. Page swaps go on until the system goes above the high watermark and the `kswapd` demon is put to sleep by the allocator.

10.1.4 Page cache

Say that multiple processes access the same file. The file is in the same physical pages (but not necessarily in the same virtual pages of the processes that use it) hence it would be useful to know if a page of that file requested by a process is already in memory (because requested by another process). The solution to this problem is using **per-file page caches**. In particular, a file page cache is the set of physical pages belonging to regular files that have been read or written. The page cache of a file is handled by a `address_space` structure that stores the numbers of a file's pages (i.e., the Physical Page Numbers used by a file). The page cache is able to:

- Map a file descriptor plus an offset to a physical page. This allows obtaining the physical page associated with a certain part of a file.
- Map a physical page to a virtual memory area. This is useful when we want to invalidate page table entries of shared pages in different processes.

Mapping a file to the physical page

Let us focus on the first functionality. Given the file descriptor `fd` of a file and an `offset`, we want to know the page where `fd+offset` is. Each file is represented in Linux through a `file` structure that contains a pointer to an `address_space` structure. The `address_space` structure organises the file pages as a tree and maps all pages of one object (e.g., an inode) to another concurrency (e.g., a physical disc block).

```
1 struct file {
2   /* other fields */
3
4   struct address_space  *f_mapping;
5
6   /* other fields */
7 };
```

This means that when a process wants to work with a file using file descriptor `fd`, the kernel can access the corresponding file structure through its descriptor `fd`. The kernel can then search the `file`'s `address_space` structure for the page corresponding to a given offset using the function

```
static inline struct page *find_get_page(struct address_space *mapping, pgoff_t
    offset);
```

This operation can be done for every process since the `file` structure is managed by the kernel and it's in common among all processes.

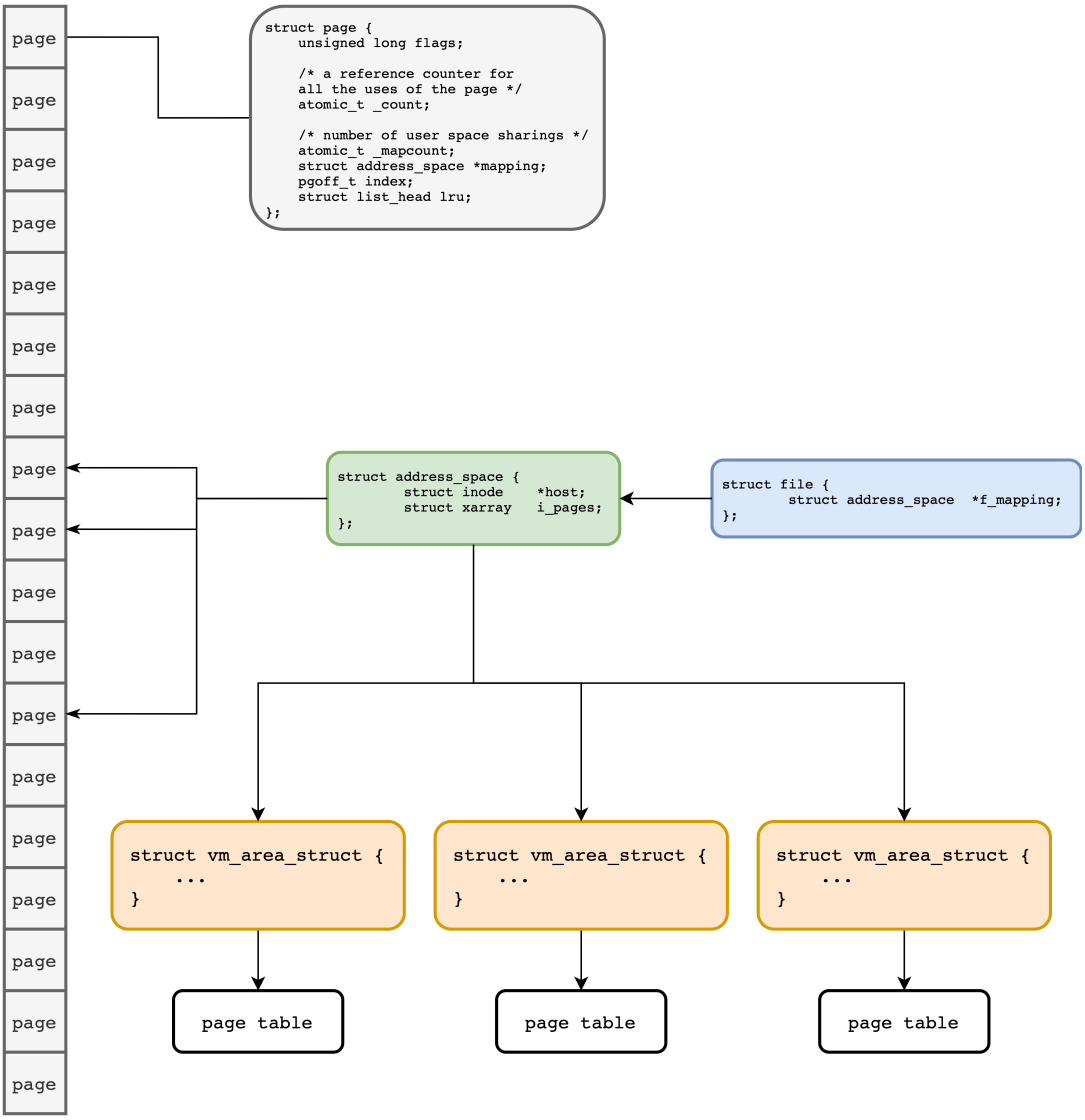


Figure 10.2: A summary of the structures used for page caching.

```
struct address_space {
  struct inode        *host;
  struct xarray       i_pages;
  struct rw_semaphore invalidate_lock;
  gfp_t               gfp_mask;
  atomic_t            i_mmap_writable;

  struct rb_root_cached   i_mmap;
  struct rw_semaphore     i_mmap_rwsem;
  unsigned long           nrpages;
  pgoff_t                 writeback_index;
  const struct address_space_operations *a_ops;
  unsigned long           flags;
  errseq_t                wb_err;
  spinlock_t              private_lock;
  struct list_head        private_list;
  void                    *private_data;
};
```

Mapping physical pages to virtual memory areas

Given a page, we can understand which file it belongs to and to which virtual memory area it's mapped, for each process using the file. This is possible thanks to the `page` structure that stores the information about a page.

```
struct page {
    unsigned long flags;

    /* a reference counter for
    all the uses of the page */
    atomic_t _count;

    /* number of user space sharings */
    atomic_t _mapcount;
    struct address_space *mapping;
    pgoff_t index;
    struct list_head lru;
};
```

In particular, if the kernel has to invalidate a page, it can go into the page's `address_space` and then into each `pdg` pointed by VMAs.

10.2 Page frame reclaiming algorithm

As we have seen, memory can get filled with inactive pages that have to be moved to swap or backing storage. It's now time to understand how this mechanism is implemented. The algorithm that chooses which pages have to be moved is called **Page Frame Reclaiming Algorithm** (PFRA). The page frame reclaiming algorithm uses the `lruvec` in the `pg_data_t` structure of a NUMA node. As we have seen, the `lruvec` structure contains lists (LRU lists), one for each state in which a page can be, that store all the pages in a certain state. We will focus mainly on two lists:

- The `INACTIVE_ANON` list that stores all inactive pages not related to a file. These pages are those the reclaiming algorithm considers for removal.
- The `ACTIVE_ANON` list that stores all active pages not related to a file.

The LRU lists implement the least recently used policy for removing pages. In particular:

- The `INACTIVE_ANON` list contains pages that are inactive. Pages in the tail of this list are those that have been inactive for longer hence those that are removed by the reclaim algorithm. When a page in this list is accessed, it's moved towards the head of the list so that it's not considered for removal.
- The `ACTIVE_ANON` list contains pages that are active. The pages at the tail are those that haven't been accessed in a long time, hence they can be moved to the inactive list while at the head we find the most recently active pages. When a page is accessed, it's put in the head of the active list.

Periodically, the page frame refrain algorithm

- Removes the pages that are in the tail of the inactive list, if not dirty.
- Moves the pages from the tail of the active list to the head of the inactive list.

10.2.1 LRU list management

Let us now analyse more in-depth how LRU lists are managed. Each page has

- An **access** bit in the TLB which is used to understand if the page has been recently accessed.
- A **ref** flag.

This information is used to move pages from one list to another. In particular, the PFRA scans both lists. While scanning the active list, for each page:

- If the access flag is 1:
  1. The access flag is set to 0.
  2. If the ref flag 1, the page is moved to the head of the active list.
  3. If the ref flag 0, it's set to 1.
- If the access flag is 0:
  1. If the ref flag is 1, it's set to 0.
  2. If the ref flag is 0, it's set to 1 and the page is moved to the head of the inactive list.

While scanning the inactive list, for each page:

- If the access flag is 1:
  1. The access flag is set to 0.
  2. If the ref flag is 1, the page is moved to the tail of the active list and the ref flag is set to 0.
  3. If the ref flag is 0, it's set to 1.
- If the access flag is 0:
  1. If the ref flag is 1, it's set to 0.
  2. If the ref flag is 0, the page is moved to the tail of the inactive list.

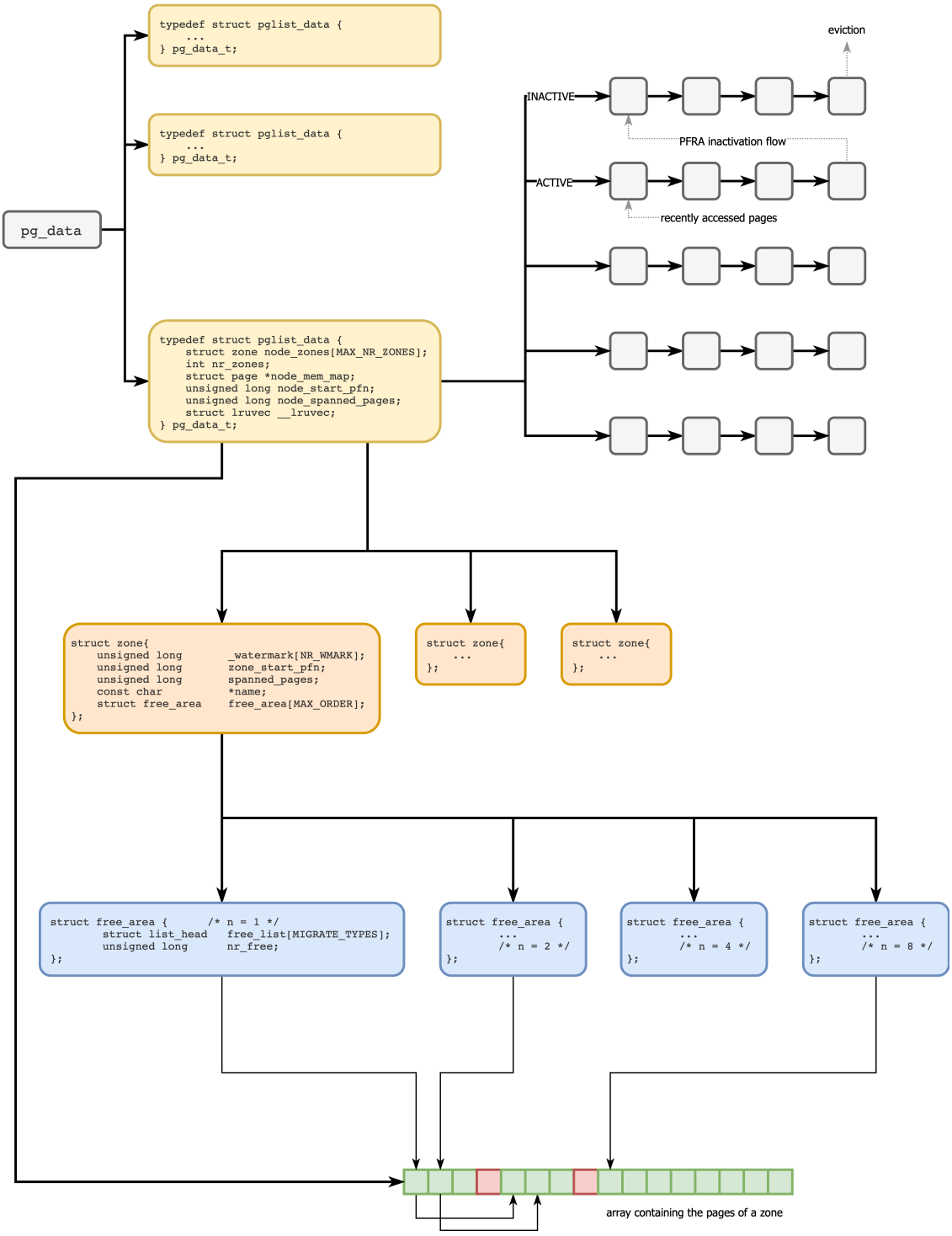


Figure 10.3: A summary of the data structures used to handle physical memory.

10.3 Object allocation

Let us now analyse how Linux handles frequent page allocation and memory allocation that require less than a page. We can't use the buddy algorithm since its smallest allocation block is a page and using a page for a small structure wastes a lot of time. Moreover, the buddy system on each NUMA node is protected by a spin-lock. The algorithms used for object allocation, i.e., for allocating memory smaller than a page are called fast allocators. The Linux kernel uses two different fast allocators:

- **Quicklists**. This fast allocator is used only for paging.
- The **Slab allocator**.

10.3.1 Quicklists

Quicklists are data structures used for implementing the page table cache. Namely, quicklists are used for pages that store page directory data itself. Quicklists are lists of pages immediately available. Each core has its own quicklist, hence there is no interference between cores and no synchronisation is necessary. Quicklists are typically implemented as stacks of pages. In particular, during the allocation, one page is popped off the list, and during free, one is placed as the new head of the list. This is done while keeping a count of how many pages are used in the cache.

Note that quicklists only allow optimising frequent accesses to some pages, but not allocations of less than a page.

10.3.2 Slab allocator

The slab allocator allows obtaining pieces of memory smaller than a page. Linux implementation of the slab allocator is called SLUB allocator. To implement this functionality, the kernel (i.e. the slab allocator) has caches, implemented with a `kmem_cache` structure, of commonly used objects kept in an initialised (but free) state available for use by the kernel.

```
struct kmem_cache {
  struct kmem_cache_cpu __percpu *cpu_slab;

  unsigned long min_partial;
  unsigned int size;       /* The size of an object including metadata */
  unsigned int object_size;/* The size of an object without metadata */

  /* omitted fields */

  const char *name;     /* Name (only for display!) */
  struct list_head list;  /* List of slab caches */

    /* omitted fields */

  struct kmem_cache_node *node[MAX_NUMNODES];
};
```

The slab allocator is basically a way in between the page allocator (i.e., the buddy allocator) and the functions used by programs to allocate some objects (e.g., `malloc`). Note that, since the slab allocator needs some pages, it will use the page allocator to obtain them.

Each cache manages objects of a particular type (e.g., `pg_data_t`) and is implemented as a doubly linked list called **cache chain**. Each cache also maintains a block of contiguous pages in

memory. More precisely, a `kmem_cache` has a list of `kmem_cache_cpu`, one for each core. Each `kmem_cache_cpu` has a reference to the page, called **slab**, that will be used to allocate the objects.

```
struct kmem_cache_cpu {
  void **freelist;  /* Pointer to next available object */
  unsigned long tid;  /* Globally unique transaction id */
  struct slab *slab;  /* The slab from which we are allocating */
#ifdef CONFIG_SLUB_CPU_PARTIAL
  struct slab *partial; /* Partially allocated frozen slabs */
#endif
  local_lock_t lock;  /* Protects the fields above */
#ifdef CONFIG_SLUB_STATS
  unsigned stat[NR_SLUB_STAT_ITEMS];
#endif
};
```

Each page is filled with the objects (e.g., `pg_data_t`) left uninitialised so that, when a process requests an object, one is picked from the page and returned to the process. This allows to speed up the process of retrieving chunks of memory for a data structure. When the data structure is released by the process, the slab allocator leaves it in an uninitialised state so that it can be immediately given to another process. To further increase retrieval speed, a list of free object addresses is kept in the field `freelist` of a `kmem_cache_cpu`. This field points to the first free structure in the slab and then each free structure in the slab contains the address of the next free structure of the slab. Whenever allocates memory for a structure handled by the slab allocator,

1. The address pointed to by `freelist` is returned.
2. The field `freelist` is updated with the address found at the memory pointed to by `freelist`.

Whenever a slab of a `kmem_cache_cpu` becomes full, it's replaced by another page and the former is stored in a list of full pages. The list of full pages is stored in the `kmem_cache`, which is a pointer to a list of `kmem_cache_node`s, one for each full slab. Note that a full page might become partially allocated if some objects are released by the process that requested them.

```
struct kmem_cache_node {
  spinlock_t list_lock;

#ifdef CONFIG_SLAB
  struct list_head slabs_partial; /* partial list first, better asm code */
  struct list_head slabs_full;
  struct list_head slabs_free;
  unsigned long total_slabs;  /* length of all slab lists */
  unsigned long free_slabs; /* length of free slab list only */
  unsigned long free_objects;
  unsigned int free_limit;
  unsigned int colour_next; /* Per-node cache coloring */
  struct array_cache *shared; /* shared per node */
  struct alien_cache **alien; /* on other nodes */
  unsigned long next_reap;  /* updated without locking */
  int free_touched;   /* updated without locking */
#endif

#ifdef CONFIG_SLUB
  unsigned long nr_partial;
  struct list_head partial;
#ifdef CONFIG_SLUB_DEBUG
  atomic_long_t nr_slabs;
  atomic_long_t total_objects;
```

```
  struct list_head full;
#endif
#endif
};
struct kmem_cache_node {
  spinlock_t list_lock;

  struct list_head slabs_partial; /* partial list first, better asm code */
  struct list_head slabs_full;
  struct list_head slabs_free;
  unsigned long total_slabs;  /* length of all slab lists */
  unsigned long free_slabs; /* length of free slab list only */
  unsigned long free_objects;
  unsigned long nr_partial;
  struct list_head partial;
};
```

Slab allocator for kmalloc

The `kmalloc` system call uses the slab allocator, in fact, it tries to figure out which `kmem_cache` has objects of the right size for the requested chunk of memory. In general, we have two different classes of caches:

- **Dedicated caches**, which are used for a specific object. Namely, we have a `kmem_cache` dedicated to a type of object (this is what we have analysed so far).
- **Generic caches**, which are general purpose and in most cases are of sizes corresponding to powers of two.

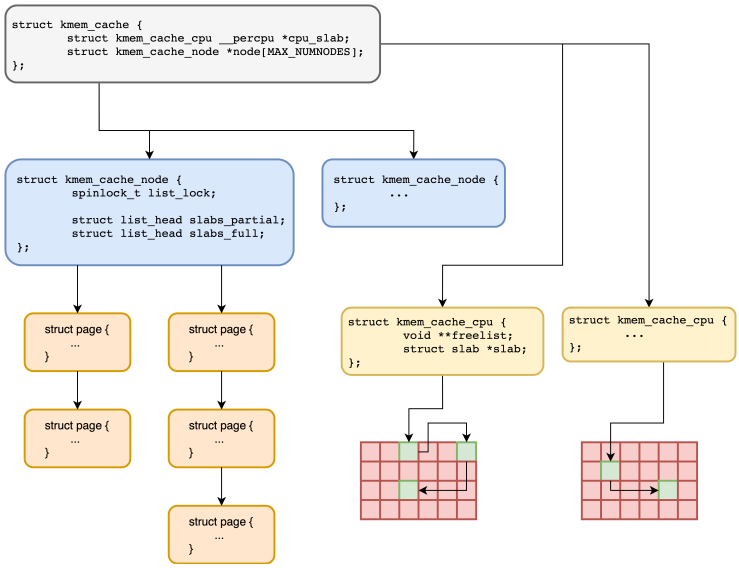


Figure 10.4: A summary of the structures used by a slab allocator.

Chapter 11

Memory security

11.1 Covert channels

Kernel memory is shared among all processes, hence it might contain sensitive data about one process even when it's not in execution (this is not true for user memory since the page directory is replaced during the context switch). Recent attacks like Meltdown have shown that it's possible to use processor speculation (i.e., the ability of a process to predict and pre-fetch the next instruction to execute after a conditional jump, before having evaluated the jump condition) to leak memory into user-mode long enough to be captured by a side-channel cache attack. Let us consider the example in Listing 11.1 to better understand how cache attacks works. The program is created by an attacker that wants to leak kernel memory data when executed on the target (victim's) machine. First, the attacker declares a buffer of characters called `my_userpace_buffer` and uses the custom function `clear_up_cache` to ensure that the buffer isn't cached. Then, we want to exploit the ReOrder Buffer (ROB), which is a buffer that allows instructions to be executed out of order but committed in order at the end of the pipeline. This can be done by executing a long instruction which forces the processor to do out of order execution. In particular, in this phase, the processor starts loading the address `kernel_address_to_be_analysed` while executing the long instruction (because of out of order execution). Since this address is in kernel space, it can't be loaded by user code, however, the fault associated with this load is triggered only when the instruction is committed by the ROB. Before throwing the exception, the `kernel_address_to_be_analysed` is used to load characters from memory to fill `my_userpace_buffer`. As last step we have to examine the cache state, i.e.,measure how much time it gets to load `my_userpace_buffer[0]` and `my_userpace_buffer[BUFSIZE]`. If `my_userpace_buffer[0]` is faster, then low bit of `kernel_address_to_be_analysed` was 0 and we have obtained a leak of kernel memory.

```
1 // buffer created by the attacker
2 char my_userpace_buffer[BUFSIZE];
3
4 clear_up_cache();
5 long_instruction; // e.g., divide
6 char t = my_userpace_buffer[ ((*kernel_address_to_be_analysed) & 1) << BUFSIZE];
7 // measure time to load my_userpace_buffer[0] and my_userpace_buffer[BUFSIZE]
```

Listing 11.1: An example of side-channel cache attack that exploits speculation.

11.1.1 The cr3 register

Before analysing how covert channels attacks can be mitigated or solved, we have to understand how Intel processors handle page tables. The `cr3` register is used in Intel machines to access the page table using a hierarchical set of paging structures. More precisely,

1. The content of `cr3` is summed with the bits from 39 to 47 of the virtual address to access the PML4 structure.
2. The value accessed in the PML4 structure is summed with the bits from 30 to 38 of the virtual address to access the PDP structure.
3. The value accessed in the PDP structure is summed with the bits from 21 to 29 of the virtual address to access the PD structure.
4. The value accessed in the PD structure is summed with the bits from 12 to 20 of the virtual address to access the PT structure.
5. The value accessed in the PT structure is the physical page number of the virtual page where the virtual address is. The last 12 bits of the virtual address are the offset from the beginning of the physical page and allow to obtain the physical address to which the virtual address maps.

This mechanism is graphically shown in Figure 11.1.

11.1.2 Kernel Page Table Isolation

Kernel Page Table Isolation solves, for old Intel CPUs, covert channels attacks. In particular, KPTI splits the page directories of user and kernel-space (which were one page directory before KPTI). This means that we have:

- A page directory that includes both kernel and user-space pages. This page directory is used only when in kernel mode, hence it's not accessed by the program in Listing 11.1. This page directory also implements protection against execution (SMEP) and access invalid references (SMAP).
- A page directory that includes only user-space pages and a minimal set of kernel-space mappings that provide the information needed to enter or exit system calls, interrupts and exceptions. In fact, for handling interrupts, there must be enough kernel code mapped in user mode to switch back to the kernel page directory and make the rest available. This page directory is used only in user-mode.

Thanks to this distinction, the program in Listing 11.1 wouldn't have access to kernel space since it's working in user-space and uses the second page directory. In particular, when specifying the kernel address, not only the process can't access it but also it can't find a mapping to the actual physical page, hence it can't even speculative-load the data in `my_userspace_buffer`.

In practice, KPTI is implemented changing the `cr3` register. More precisely, when working in user-space the register is set to a value that allows to obtain only the address of the user page directory. Whenever the program goes to kernel-space, the value in `cr3` is changed so that the kernel can access the whole page table. This means that we don't really have two tables, but we just give a limited view of the whole table to the user.

Albeit solving covert channels attacks, KPTI comes at a cost. In particular, it's estimated that this mechanism reduces performance by 5 %.

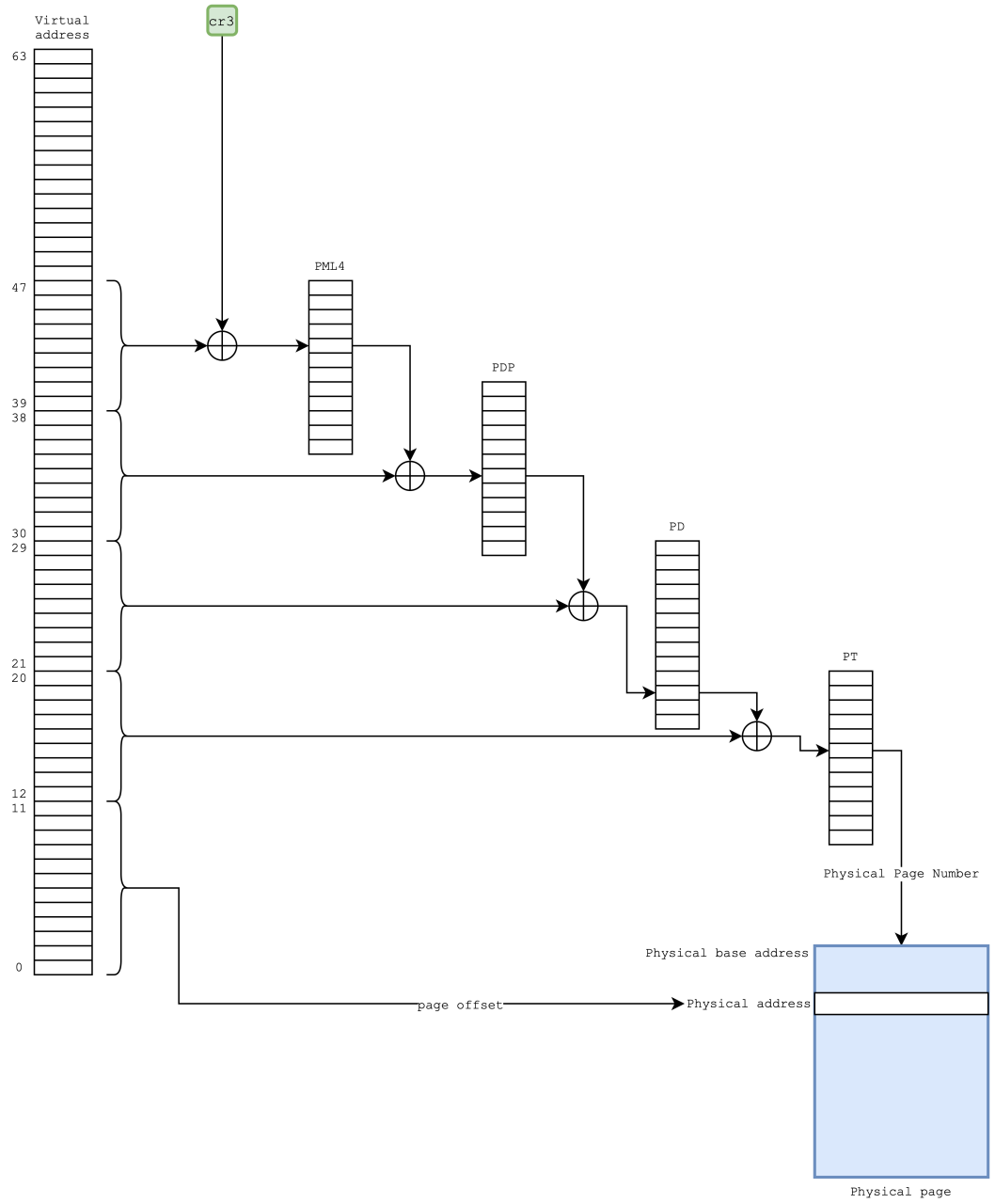


Figure 11.1: The schema representing the mechanism to obtain the physical address to which a virtual address maps.

Part V

Virtual machines

Chapter 12

Introduction

12.1 Virtual machines

> **Definition 12.1** (Virtual machine)**.** *A virtual machine is an efficient, isolated duplicate of a real machine.*

This means that we have a **host machine** which is the real, physical machine on which a **guest machine**, i.e., the virtual one, runs. The virtual machine can run on top of the host operating system, i.e., the operating system that runs on the host machine or directly on the host machine. The communication between host and guest is handled by a program called **virtual machine monitor** (VMM) or hypervisor. There exist different type of hypervisors, each having a different approach with respect to how guest instructions are handled and translated. Virtual machines bring the concept of virtualisation to a whole new level, in fact, it allows to let the entire OS think that it has the whole resources of a physical system allocated for it.

12.1.1 Hypervisors

Hypervisors are programs that allow virtual machines to work and that handle the access of guest machines to physical resources. In particular, hypervisors create an environment on which the host machine or OS runs. Moreover, hypervisors have complete control of the physical resources, in particular an hypervisor:

- Has a view of the physical resources.
- Must enforce security.
- Has to be fast.

Hypervisors can be divided into:

- **Type 1** hypervisors, which work directly on the physical machine without any intermediate OS. This means that when the machine is turned on, the hypervisors starts running immediately.
- **Type 2** hypervisors, which work on top of a host OS.

Note that Linux has a module, called **KVM**, which allows to run Linux as an hypervisor. This means that KVM can be considered wither a type 1 or 2 hypervisor.

12.1.2 Requirements

A virtual machine has to ensure:

- **Fidelity**. The virtual machine should behave like a physical machine.
- **Safety**. The virtual machine cannot override the VMM's control of virtualised resources.
- **Efficiency**. Programs running on a virtual machine should only display a small decrease in speed.

12.1.3 Motivations

Virtual machines are used to:

- **Consolidate and partition hardware**. If we need two different machines we can use a physical machine at 100% with two VMs instead of two used at 50%. (consolidation).
- **React to variable workloads**. Reduced hardware and administration costs for data-centres. Horizontal scalability.
- **Standardise infrastructures**. Simplifies software distribution for complex environments. Isolated network and storage.
- Do **security sandboxing** and fault tolerance through checkpointing and rollback.

12.1.4 Instructions

Virtual machine monitors have to monitor instructions that the guest machine wants to execute so that they can be executed on the physical machine. Instructions can be

- **Privileged**. Privileged instructions, like a system call, trigger a switch to supervised mode (i.e., kernel mode).
- **Unprivileged**. Unprivileged instructions can be executed in user mode.

The idea behind virtualisation is to execute privileged instructions of the guest machine in user mode (since the hypervisor runs in user mode). Namely, we want to run privileged instructions in an unprivileged mode. Moreover, instructions can be **virtualisation-sensitive**. In particular, an instruction is virtualisation-sensitive if it is:

- **Control sensitive**. An instruction is control sensitive if it modifies directly the machine status (e.g., enabling or disabling interrupts, modifying the interrupt vector table).
- **Behaviour sensitive**. Instructions that behave differently when used in either user or supervisor mode. These might affect fidelity.

Virtualisation-sensitive instructions have to be run in unprivileged mode, too and that's what makes virtualisation hard to achieve. Sensitive instructions are fundamental for building a virtual machine. In particular, we have the following result:

Theorem 12.1 (Popek and Goldberg)**.** *For any conventional computer, if the set of sensitive instructions for that computer is a subset of the set of privileged instruction then a virtual machine monitor can be built.*

Basic x86 processors do not comply with Theorem 12.1. For this reason, Intel has developed the VT-x technology on its processors, which allows to satisfy Theorem 12.1. However, the theorem is a sufficient condition, not a necessary one as some mechanisms exist to achieve virtualisation for x86 processors not equipped with VT-x.

12.1.5 Virtual machine classification

Virtual machines virtualisation can be divided into:

- **Software based virtualisation**. In software based virtualisation, the user and kernel code of the guest machine is run in the host machine's user space. This means that kernel code that was meant to be run in kernel mode has actually to be executed in user mode.
- **Hardware based virtualisation**. In hardware based virtualisation we try to minimise the intervention of the virtual machine monitor by creating a duplicate of the original user and supervised mode where the guest machine can run without triggering the VMM. The guest user and supervised modes are a new set of mode that are added to the host processor itself. Namely, with one guest, the processor can run in four different modes: host-user, host-supervised, guest-user, guest-supervised. Note that, when the processor is in a guest mode (even in guest-supervised), the instructions can't directly access the hardware, however, differently from software based virtualisation, privileged instructions like system calls can be executed in guest-supervised mode without passing through the VMM. Because the interaction with the VMM is limited, we can reach almost native speed.

In both cases a part of the VMM has to run on the host's kernel space since it has to catch and execute the privileged instructions issued by the guest.

12.2 Software based virtualisation

Software based virtualisation is based on two key ideas:

- **Deprivileging**. Deprivileging allows to run in user mode some instruction that were meant for supervised mode. Note that, kernel-mode instructions, if executed in user mode, might display a different behaviour than what's expected (e.g., interrupts).
- **Shadowing**. The VMM has to shadow the page directory used by the host. In particular, a request by the guest to the page directory is intercepted by the VMM which changes the page table and presents to the guest a page that is not the actual one. In other words, the VMM presents to the guest a page table which is not the real one (i.e., the one used by the host). The real page table is called shadow page table and is not visible to the guest machine.

Let us now consider an x86 machine to further analyse the two main technologies that have to be implemented in software based virtualisation. We will consider both technologies together. To handle deprivileging, x86 machines have two user modes (called ring 1 and ring 3) that can be used to run the guest user (ring 3) and the guest supervisor (ring 1). This means that when a guest

privileged instruction is executed (i.e., when the guest switches to kernel mode), the host is still in user mode since the guest supervisor is run in ring 1. Even with this distinction, since we are still talking about software based virtualisation, a privileged instruction triggers the VMM which handles it. In particular, the VMM has to shadow all the structures like the page directory that can be accessed and modified by the guest. Consider for instance a guest that wants to set up an interrupt. This is done, in kernel-space, writing the interrupt descriptor table (IDT). A write to the IDT by the guest is intercepted by the VMM which writes on a local virtual IDT, which is used specifically and only for the guest. The original IDT is therefore a shadow structure which is never accessed or modified by the guest since a write or a read will always trigger the VMM that will modify or show its local virtual IDT. The same concept is applied to the page directory.

Note that the VMM might still have to use the shadow IDT. Consider for instance a timer interrupt. If the VMM has to simulate it, it must add a timer interrupt to the shadow IDT (i.e., the real one) and when the timer fires the interrupt is received by the VMM which forwards it to the guest like if it came from the virtual IDT.

12.2.1 Shadow page table

One of the most important shadow structures is the page table. Say we have a guest machine on a x86 host which wants to write in the `cr3` register, which is the one that enables the processor to translate virtual addresses into physical addresses by locating the page directory and page tables. When the guest OS tries to write its page directory, the VMM intercepts the write and builds in the VMM memory a virtual page table that is used only by the guest. This means that:

- The virtual `cr3` register and the virtual page directory are used to map guest's virtual pages to guest's physical pages (which actually correspond to some virtual pages for the host OS).
- The `cr3` register is then set by the VMM so that the guest's physical pages are mapped to the actual physical pages in the actual physical memory. This means that the `cr3` register maps the guest virtual pages directly to the actual physical pages. In other words, the VMM creates a shadow page table, mapped to some actual physical page, to which the `c3` register points to in such a way that the actual physical mapping remains transparent from the guest OS's point of view. Moreover, such mechanism allows to avoid direct changes of the guest page table since it is marked as read-only by the VMM and if the guest OS tries to modify it, it gets intercepted by the VMM which is going to properly update the shadow page table, since it is the only one knowing the actual physical mapping.

Note that, the guest will only see the mapping from its virtual to physical pages.

12.2.2 Ring aliasing problem

Pure software based virtualisation not only can be slow but also might require some hardware modification to run fully software based virtualisation. For instance, some architectures have virtualisation-sensitive unprivileged instructions (e.g., those for manipulating the interrupt flags and those reading and writing segment descriptors and registers). As an example, the low level `pushf` and `popf` instructions, used to manipulate the interrupt flags, don't trigger any trap if executed in user mode. This is a problem because the VMM has to catch the changes in the actual interrupt flag to update the virtual interrupt flags, otherwise the guest environment differs from what the guest OS expects. This problem is called ring aliasing problem and allows the guest OS to understand that it's not running in the desired mode (i.e., not in kernel mode). In general, this problem verifies with four classes of instructions:

- The `pushf`, `popf`, `iret` instructions manipulating the interrupt flag (`%eflags.if`) are `NOP`s if executed in user mode. This means that they do not trap and the guest OS would think it has disabled interrupts when it hasn't.
- `lar`, `verr`, `verw`, `lsl` provide visibility into segment descriptors in the global or local descriptor table. These instructions would access the VMM's tables (rather than the ones managed by the operating system), thereby confusing the software.
- `pop <seg>`, `push <seg>`, `mov <seg>` manipulate segment registers. This is problematic since the privilege level of the processor is visible in the code segment register. For example, `push %cs` copies the `%cpl` as the lower 2 bits of the word pushed onto the stack. Software in a virtual machine that expected to run at `%cpl=0` could have unexpected behaviour if `push %cs` were to be issued directly on the CPU.
- `sgdt`, `sldt`, `sidt`, `smsw` provide read-only access to privileged registers such as `%idtr`. If executed directly, such instructions return the address of the VMM structures, and not those specified by the virtual machine's operating system.

The ring aliasing problem is a correctness problem since the guest OS might not work properly.

12.2.3 Excessive faulting problem

When a program running on the guest machine does a lot of system calls, it invokes the hypervisor very often. This problem, called excessive faulting problem, can slow down the guest machine, hence it is a performance problem.

12.2.4 Binary translator

The ring aliasing and excessive faulting problems can be solved using a binary translator working on each basic block and translating the instructions that cause the ring aliasing problem. Namely, the hypervisor, during translation looks for these instruction and converts them to code that can manage correctly the action that was supposed to be executed. For instance, we have seen that the `pushf` instruction causes the ring aliasing problem since it's replaced by a `NOP`. The VMM replaces it with a piece of code that correctly executes what the `pushf` should have done, hence a piece of code that actually modifies the interrupt flag.

The binary translator is invoked at runtime and it translates the basic block that has to be executed.

Translation can be sped up using a translation cache which stores, for each code block the translated code block. During translation, the virtual `eip` (i.e., the `eip` of the instruction to be translated) is used to access the translation cache and get the `eip` of the translated instruction.

12.3 Hardware based virtualisation

Hardware assisted virtualisation allows to make all sensitive instructions privileged. Then all instructions that are not privileged become privileged and the hypervisor catches instructions only when strictly necessarily.

Hardware virtualisation also allows to save and restore the state of the virtual machine with explicit operations that take the state of the guest machine and save it into an explicit data structure in memory that the hypervisor can use. This is not something that software can do by itself but it has to be supported by hardware. Hardware virtualisation is used to:

- Satisfy (or at least get closer to satisfying) the requirements of Theorem 12.1.
- Improve performance by reducing the number of privileged translations and avoiding the overhead associated with shadow pages.

12.3.1 Multiple privileged modes

Hardware based virtualisation is implemented adding new privileged modes. Different architectures decided to implement this idea in different ways, for instance:

- ARM added a mode used just for the VMM. This means that we have three modes: VU (user mode), VS (supervisor mode) and HS (hypervisor mode).
- x86 added two orthogonal modes that mirror the original modes.
- RISCV added a mode for the VMM and a mode that mirrors the host user mode, hence obtaining an hybrid between the ARM and x86 solutions. Namely, we have 4 modes: U (user), VU (guest user), VS (guest supervisor) and HS (hypervisor). Note that the hypervisor mode is used as supervisor mode by the host.

12.3.2 Comparison of software and hardware based virtualisation

Software and hardware based virtualisation have their own advantages and disadvantages. In particular:

- Hardware based virtualisation has no overhead since it invokes the VMM fewer times.
- In hardware based virtualisation it's easier to recover the state of the guest machine.
- Hardware based virtualisation has less need of binary translation.
- In hardware based virtualisation it's harder to handle input/output operations since they have to go to the VMM, then to the driver of the host and back to the guest machine. On the other hand, in software based virtualisation, we can just translate the instruction. Basically, in software based virtualisation we are working on a more optimised version since the host knows the peripherals because of shadowing.
- In hardware based virtualisation it's harder to work with page tables.

12.3.3 Performance improvements

The main problems of hardware based virtualisation are page table shadowing and I/O management. Let us analyse how to solve these issues.

Extended page tables

We can improve hardware based virtualisation performance by avoiding page table shadowing. In particular, in x86, we can avoid shadowing page tables by using **extended page tables**. Extended page tables give more authority to the guest's `cr3` register (which handles page tables) by allowing the guest to specify its own translation. In particular, the host's `cr3` isn't used anymore to translate from guest virtual pages to physical pages but only from guest physical pages to actual physical pages. Namely, the host's `cr3` register stores only the translation from guest's physical pages (or

host's virtual pages, since a guest physical page is a host virtual page) to host's physical pages (i.e., actual physical pages). Schematically, we have two mappings:

- One mapping, done by `guest.cr3` from guest virtual pages to guest physical pages (which are host virtual pages).
- One mapping, done by `cr3` from host virtual pages (which also include guest physical pages) to host physical pages.

Thanks to this mechanism, when the guest wants to access a page, the hardware composes the host and guest tables to obtain the right page. Whenever the guest wants to modify its page tables, it doesn't trigger the VMM and modifies directly its page table (i.e., the mapping from guest virtual pages to guest physical pages). This is only possible if the hardware knows that there is a guest and a host page directory.

IO pass-through

Whenever a guest wants to access a peripheral (e.g., write a memory mapped register), it has to go through the VMM, access the host driver and go back to the guest machine. To solve this problem we should allow the guest to access directly the peripheral. This result can be achieved only with an hardware mechanism. Solving this problem doesn't only involve writing from the guest OS to the peripheral but also from the peripheral to the guest machine memory. This is because peripheral work thanks to memory registers (i.e., areas of memory that work as registers) that can also be written by peripherals to communicate with the guest (e.g., sending some data from the peripheral is done by writing on the guest's memory). In particular, when the guest OS asks for some data from a peripheral we make it so that it's not possible for the peripheral to write somewhere else. This prevents the peripheral from accessing data in the guest memory.

In practice, this is achieved by introducing a IO Memory Management Unit (IOMMU), which can be thought of as a table that, for each device, says where it can write and what is the translation to the actual physical memory where it should write. When the guest OS wants to read something from a peripheral, the IOMMU has to enable the guest OS to specify its virtual address and the IOMMU will translate the data so that is written to the correct address. Note that this means revoking the peripheral from the host OS.

In general, this mechanism is called virtual function IO. The idea behind this mechanism is that a guest OS can work directly with device drivers assuming that it's possible to create a translation of addresses.

The same idea applies to **user-mode device drivers**. In this case the guest OS is thought of as a user-space application which has direct access to device drivers and address translation is handled by the IOMMU.

Chapter 13

Real word examples

13.1 KVM

KVM is a Linux kernel module that allows to write a custom VMM. We can start, stop and interact with a virtual machine so that all traps hidden to the hypervisor guest interaction are exposed to the user space virtual machine monitor.

KVM is possible because the hardware provides a way to restore the state of a guest virtual machine into a control block which is used by KVM and the user space VMM to stop the guest OS and save its state. The state of the host OS can also be saved. In fact, we can use the state of the guest OS to understand why the VMM has been triggered and react appropriately.

Moreover, the virtual machine is run as a kernel thread.

```
int kvm_fd = open("/dev/kvm", O_RDWR);
int vm_df = ioctl(kvm_fd, KVM_CREATE_VM, 0);
char buf[] = { /* memory initialised data */ };

ioctl(vm_fd, KVM_SET_USER_MEMORY_REGION, {&buf});
int vcpu_fd = ioctl(vm_fd, KVM_CREATE_VCPU, 0);

kvm_run *kr = mmap(..., vcpu_fd, ...);

ioctl(vcpu_fd, KVM_RUN, 0);
cout << "Exit reason: " << kr->exit_reason << endl;

kvm_regs regs;
ioctl(vcpu_fd, KVM_GET_REGS, &regs);
cout << regs.rax << endl;
```

Listing 13.1: Code to set up a KVM

13.2 Intel x86 VT-x

Chapter 14

Paravirtualisation

14.1 Paravirtualisation

Paravirtualisation is a virtualisation technique that aims at reducing the number of instruction executed in the virtual environment by the guest operating system. This can be achieved by changing the guest OS so that it can directly invoke the hypervisor whenever it has to do some privileged operation. This means that the guest OS must know that there is an hypervisor and must know what part of the hypervisor to invoke. For instance, when modifying the page directly, instead of writing in the page directory and being trapped by the hypervisor, the guest OS directly invokes the hypervisor which handles the write to the page tables. Paravirtualisation also allows to take a OS that was not meant to run on a VM and run it in a virtual environment, in fact we only have to add some code to let the operating system directly invoke the VMM for some specific operations. This means that we can take an already solid kernel and run it on a virtual environment.

Note that this technique is also used by non-paravirtualisation hypervisors. In general, hypervisors don't require the guest OS to know that there is an hypervisor, however we can modify the guest OS and make it call the hypervisor when it wants to access a peripheral. That is to say, when the guest OS wants to access a peripheral, it can call directly the hypervisor which will handle the operation on behalf of the guest OS. This implies that the guest OS knows that it's run on a virtual environment and that there is an hypervisor.

Paravirtualisation is a must when dealing with high performance virtual machines because it can speed up access to peripherals invoking directly the hypervisor.

14.1.1 Virtio

Virtio is a specification for writing guest device drivers that access directly the host. Let us consider, for instance, a network card to understand how virtio works. Let us first consider what would have happened with normal virtualisation (i.e., without the virtio specification). The guest driver of the network card writes into some memory mapped registers that are the interface to the card. The registers point to the queue of packets sent and received by the network card. Since we are talking about the guest driver, whenever the guest writes to the memory mapped registers, the write is trapped by the hypervisor. The hypervisor (e.g., KVM) has then to use the actual driver to send or receive the data. However, since the registers are read and written a lot (the card might have to send and receive a lot of data from and to the Internet), a lot of traps have to occur. Long story short,

every time the guest accesses a memory mapped register, the operation is trapped and executed by the hypervisor. This means that, each operation requires:

1. Exiting from the VM context.
2. Handling the trap and sending or receiving data.
3. Entering in the VM context.
4. Continue executing the guest.

The main issue with this solution is that the guest will spend a lot of time in hypervisor mode for handling the trap and using the host device driver. The virtio specification solves this problem by using a device driver that doesn't access the memory mapped registers but uses an interface, which communicates with the corresponding interface on the host OS. The host interface writes and reads data from a shared memory queue (i.e., the queue of network packets to send and received). Namely, we are replacing traps for each read and write with a write into a queue and trap. This allows the guest to spend way less time in hypervisor mode since it only has to trap for writing into the queue (i.e., only the write or read operation is trapped) but the the thread on which the guest is running can immediately return to user-guest mode. The interaction with the peripheral is deferred and handled by another thread. This means that each operation requires:

1. Exiting from the VM context.
2. Handling the trap and writing to a queue.
3. Entering in the VM context.
4. Continue executing the guest.

14.2 Containerisation

Containers are a way to isolate a set of processes and make them think that they are the only ones running on the machine. The machine they see may feature only a subset of the resources actually available on the entire machine (e.g., less memory, less disk space, less CPUs, less network bandwidth). Even if they look similar, containers are not virtual machines, in fact:

- Processes running inside a container are normal processes running on the host kernel.
- There is no guest kernel running inside the container
- You can't run an arbitrary operating system in a container, since the kernel is shared with the host.

The most important advantage of containers with respect to virtual machines is performance in fact, there is no performance penalty in running an application inside a container compared to running it on the host.

Linux Containers are implement using two distinct kernel features:

- **Namespaces**.
- **Control groups**.

14.2.1 Namespaces

Namespaces provide a mean to segregate system resources. In particular, for each process, the kernel remembers the inode of the process root director. This directory is used as a starting point whenever the process passes to the kernel (e.g.,in a `open()`) a filesystem path that starts with "/". Whenever the kernel walks through the components of any filesystem path used by the process and reaches the process root directory, a subsequent ".." path element is ignored. This process is similar to the `chroot()` syscall, in fact by using `chroot()`, we can make a subset of the filesystem look like it was the full filesystem for a set of processes.

However, not everything is a file in Unix. For instance we can have

- We can only have one port 80 throughout the system (thus, only one web server).
- Only one process with pid 1 (thus, only one `init` process).

Thus, for example, a process running in a `chroot` environment will still be able to see all the processes running in the system, and it will be able to send signals to all the processes belonging to any user with the same user id as its own. Namespaces have been introduced to create something similar to `chroot()` environments for all these other identifiers. Each process in Linux has its own network namespace, pid namespace, user namespace and a few others. Network interfaces and ports are only defined inside a namespace, and the same port number may be reused in two different namespaces without any ambiguity. The same holds true for processes and users. Normally, all processes share the same namespaces, but a process can start a new namespace that will be then inherited by all its children, grandchildren, and so on. This is done when the process is created using the `clone()` system call. This same system call has been extended to implement namespaces, essentially by adding flags for the sharing or copying of the network, pid, user namespaces and so on.

14.2.2 Control groups

Processes may use too many system resources, e.g., allocating to much memory, using to much CPU time, or disk and network bandwidth. Control groups allow to solve this problem by creating groups of task and assigning to each group a part of the system's resources. In practice, control groups are used for:

- Isolating core workload from background resource needs.
- Differentiating between web server and system processes (eg. Chef, metric collection, etc).
- Differentiating between time critical work and long-term asynchronous jobs.
- Not allowing one workload to overpower the others.

Control groups are defined by the administrator and their directory hierarchy at is specified at `/sys/fs/cgroup`. This directory contains all groups, represented as a directory. This means that if we have a `background` group in our system, we can find its configuration files in the `/sys/fs/cgroup/background` folder. For each group, we have to specify which part of each resource is assigned to that group. This is done by writing in appropriate files in the hierarchy, for instance in `/sys/fs/cgroup/background/cpu.weight` to define the percentage of cpu used by the `background` group. A task can be added to a group by adding its pid to the `*/cgroup.procs` file. The group directory also contains a `cgroup.subtree` file which is used to check what resources are accessible by that group.

Part VI

Input/Output and drivers

Chapter 15

Input and output

15.1 Processor's architecture and input/output

Before analysing how input and output is handled in Linux, let us describe a reference architecture. Let us consider a x86 CPU. The processor is connected to

- Memory with a dedicated interface, usually DDR.
- Graphic cards with a PCI (or PCIe) interface. The PCI interface is serial bus usually divided in lanes.
- The IO chipset with a dedicated interface, usually the DMI (Direct Media Interface). This chipset manages the connections to the external devices (e.g., storage devices, network interfaces). The external devices are connected to the IO chipset with all kind of interfaces, like PCIe, USB or SATA.

An example of this architecture is shown in Figure 15.1.

This is in contrast with the theoretical architecture which uses a single shared bus used for memory and devices. The proposed architecture is rather faster since we have a dedicated bus for each main functionality instead of a single bus which would allow only one device to communicate with the CPU and block the others.

The CPU writes to and reads from devices (like GPUs, storage and network cards) by writing to and reading from **device ports** which can be:

- Memory addresses from which the CPU and the devices can write and read with the `load` and `store` instructions.
- Specific IO ports addresses that are accessed by the CPU with special instructions.

15.1.1 Memory mapped ports

Memory mapped ports are used especially in microcontrollers and are used by devices that communicate through the serial interface. The serial interface is exposed to the CPU by a set of registers, which are memory regions. Namely, the CPU knows that some words (or bytes) in memory can be written or read to communicate with the device. Usually, the registers can be written serially by the device and read in parallel by the CPU. For instance, if a register is at address `0xc0100` then we can define the address of the register as

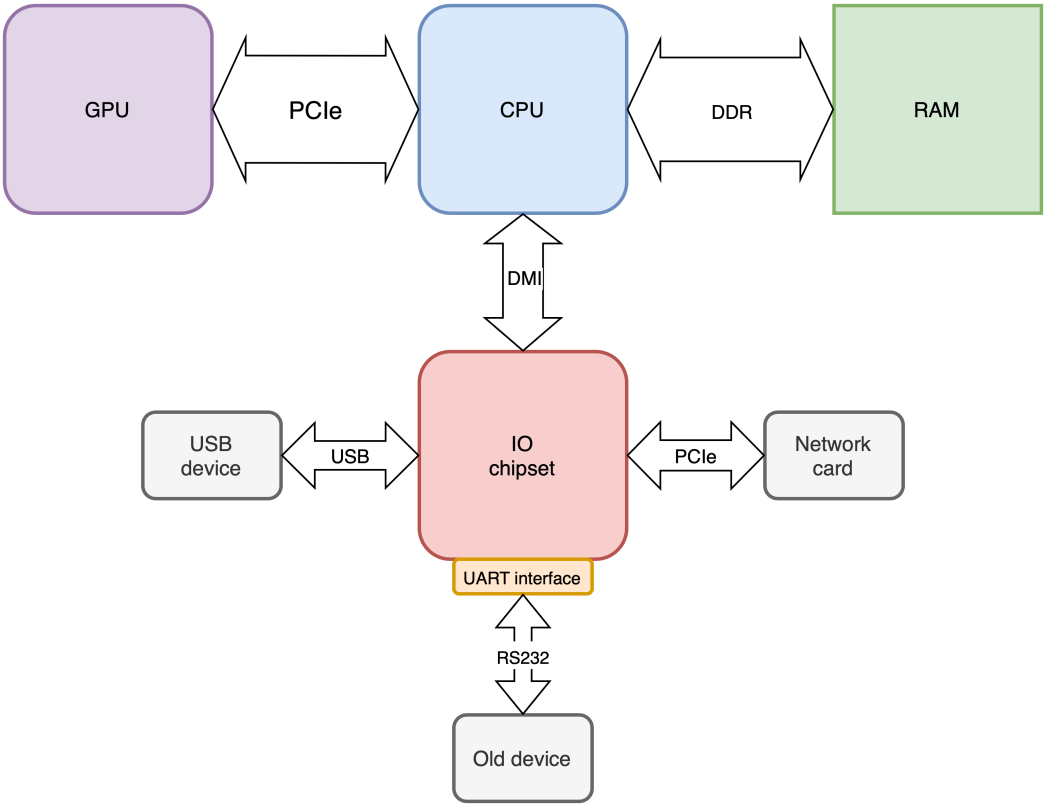


Figure 15.1: The architecture of a system with IO devices.

```
volatile int32_t device_control = *((int32_t *) (0xc0100 + PHYS_BASE));
```

Note that the variable used to access the memory register is declared as `volatile` since we don't want its value to be cached. Then we can write some value into the register by writing in the area of memory we have just defined.

```
device_control = 0x80;
```

Finally, we can read from the register by reading from memory.

```
int32_t status = device_control;
```

In some cases, a device is controlled by a set of memory mapped registers, hence it's useful to define the registers as a structure. Say for instance we have a LED which is controlled by

- A 16 bit control register called `CTRL` at address `0x3450`.
- A 8 bit data register called `DATA` at address `0x3468`.

The registers can be stored in the following structure:

```
#define LED ((LED_struct*) 0x3450)

struct LED_struct {
    volatile short CTRL;
    char gap;
    volatile char  DATA;
};
```

Let us understand why this structure makes sense:

- The `CTRL` register is 16 bit long, hence it can be stored in a `short int` variable (volatile for the same reason we've seen before).
- The `CTRL` register is put as as first field of the structure because it has the lowest address. In particular, if we define `LED` as `0x3450` casted as a pointer to a `LED_struct`, then we can access `CTRL` with the following instruction:

  ```
  LED->CTRL = 3;          /* Assign value 3 to the register CTRL */
  int ctrl = LED->CTRL;   /* Read the current value of CTRL */
  ```

  `LED` is in fact a pointer to a `LED` structure, whose address is `0x3450`. If we access the first field of `LED`, we get `CTRL`, which is at address `0x3450`.
- The `DATA` register is at address `0x3468` but, since the `CTRL` is 2 bytes long, we can't put `DATA` immediately after `CTRL`, otherwise it would be at address `0x3460`. This is why we have to add another field in between which accounts for the gap between the address of the two registers. This gap has to be 1 byte long and doesn't have to be volatile since it's not used.
- The `DATA` register can be accessed with the following instruction:

  ```
  LED->DATA = 3;          /* Assign value 3 to the register DATA */
  int data = LED->DATA;   /* Read the current value of DATA */
  ```

15.1.2 IO ports

IO ports are a characteristic of x86 processors. Peripherals are accessed using the `in` and `out` privileged instructions. Each device has a port number and the instructions manipulate the registers associated to a certain device using its port number. It's like having a different address space, called IO address space, which is used for the `in` and `out` instructions. As for memory mapped ports, IO ports are also mapped to some memory addresses.

15.2 Device to CPU communication

We have analysed how the CPU communicates with a device (i.e., how it reads from or writes to the device). Let us now analyse how the device communicates with the CPU. In particular, we want to understand how the CPU can read values written by a peripheral. There exist two methods for solving this problem:

- **Polling**.
- **Interrupts**.

15.2.1 Polling

Polling uses a loop that actively waits for the interface to be ready. Namely, the CPU keeps checking if the interface is ready for sending or receiving some data and then sends or reads the data. As an example (using the `inb` and `outb` functions for IO ports),

```
#define LSR 0x1cd
#define THR 0x1dd

// wait until the interface is ready for sending actively checking the LSR register
while((inb(LSR) & (1<<5)) == 0);

// send the character 'c' to the register THR
outb('c', THR);
```

Polling wastes a lot of clock cycles because of the busy loop but it's very useful and cheap when dealing with fast devices.

15.2.2 Interrupts

Interrupts avoid using a busy loop to check if the interface is ready. When a process wants to read some data, it issues a request for an interrupt and goes to sleep. Whenever the peripheral has some data ready, it raises an hardware interrupt which triggers the execution of the interrupt handler. In this case we don't have a busy loop, however in case of frequent communications, the OS has to handle many interrupts hence do many context switches. The code used to register an interrupt handler is shown in Listing 15.1.

```
static irqreturn_t handler(int irq, void *mydata) {
    // acquire locks on shared data
    // read/write from peripherals through MMIO // defer work
    // release lock
    return IRQ_HANDLED;
}

static int __init mydriver_init_module(void) {
```

```
9       // allocate space for mydata
10      ret = request_irq(irqnum, handler, IRQF_SHA, mydata);
11  }
```

Listing 15.1: A snippet of code used to register an interrupt and its handler.

Direct memory access

Direct memory access can solve the issue of frequent interrupts. Consider for instance a network card used as follows:

1. A process sends a byte by writing to a memory register.
2. The card sends the data.
3. The card raises an interrupt to notify the process that the data has been sent.

After writing the data, the process can go to sleep until the interrupt is raised. If the process has to send a lot of bytes, the sequence of operations above is repeated multiple times and the CPU has to execute a lot of context switches just to execute a simple operation (i.e., writing some data into memory). Direct memory access (DMA) solves this problem by adding an intermediate device, called DMA, that handles the data transfer and communication with the peripheral on behalf of the process. Thanks to the DMA, a process can send some data as follows:

1. The process sends all the data to the DMA.
2. The DMA communicates with the network card to send one byte at a time. During this phase, the DMA handles the interrupts raised by the card when a byte has been sent without stopping the process.
3. The DMA can optionally communicate with the process some intermediate results (e.g., half of the data has been successfully sent).
4. The DMA communicates to the process that the data has been transferred successfully.

15.3 Interrupts

15.3.1 Classification

Interrupts can be classified into:

- **Asynchronous interrupts** which are raised asynchronously with respect to the CPU clock. Asynchronous interrupts can in turn be:
  - **Maskable**. Maskable interrupts can be disabled and ignored by the CPU.
  - **Non-maskable**. Maskable interrupts can't be disabled.
- **Exceptions** (or synchronous interrupts) which are raised after the execution of an instruction. Exceptions can in turn be:
  - **Fault exceptions**. Fault exceptions correct and re-execute the faulty instruction.
  - **Traps**. Traps don't re-execute the faulty instruction.

Note that, the synchronous and asynchronous interrupts aren't called that way because of when they are handled but because of when they are generated. In particular, an asynchronous instruction can be raised even during the execution of an instruction while exceptions are raised only after an instruction. When dealing with peripherals we use almost only asynchronous interrupts (which makes sense since interrupts are issued by a peripheral, which has no knowledge of the instructions that are executing on the CPU and of the clock).

15.3.2 Interrupts flow

Let us consider the x86 architecture. An external device raises an interrupt sending an interrupt request (IRQ), which is an interrupt signal that contains the identification of the peripheral, to the Peripheral Interrupt Controller (PIC). The PIC uses an interrupt table, which in x86 is pointed to by a register called `IDTR`, to map the interrupt to the handler of that interrupt. For each entry in the interrupt table, Linux installs a routine called `do_irq` that invokes the actions that device drivers have registered to be executed whenever the interrupt request comes. More precisely, the `do_irq` routine:

1. Disables all the interrupts.
2. Enables all interrupts apart from the one currently handled. This is done to avoid nesting the same type of interrupt and to acknowledge to the peripheral that the interrupt has been received.
3. Handles the actions registers by the processes.
4. Enables the interrupt handled.

Interrupts are handled in supervised (kernel) mode, hence the interrupt handler, also called interrupt service routine (ISR), is run in supervised mode.

15.3.3 Interrupt controllers

The programmable interrupt controller (PIC) is the chip used to handle incoming interrupt requests. Initially, communication between the PIC and the CPU was done through a dedicated interface. The PIC had 8 pins and could receive up to 8 interrupts (one for each pin). The CPU would receive a single interrupt signal and it had to use a specific bus to check on the PIC the data related to the received interrupt.

Advanced programmable interrupt controller

The advanced programmable interrupt controller (APIC) has from 24 to 256 input pins to handle from 24 to 256 different interrupts. The APIC is connected to multiple processors, each with its local programmable interrupt controller (LPIC). Each LPIC is connected to the APIC though a APIC bus. The bus is used to handle the interrupt on the correct CPU and to balance the load. In fact, a process can specify if an interrupt can be executed on any processor or on a specific processor. In this context, Linux allows inter-processor interrupt which allows to execute an interrupt in multiple processors by using a message-passing technique.

15.3.4 Messaging signal interface

Note that, having introduced a PIC, we now have two interfaces to communicate with a peripheral:

- The PIC used to receive interrupts.
- The memory registers to write and read data.

It'd be better to have a unique interface through which a process can communicate with peripherals. In particular, we can use the PCIe interface to handle interrupts as messages on the PCIe interface. This interface is called messaging signal interface (MSI) and allows devices to write interrupt messages over the PCIe interface, which are sent to the CPUs' LPIC. Note that having removed the APIC, we have removed some circuits, hence we can handle more interrupts.

15.3.5 Deferred interrupts

The behaviour of the `do_irq` function allows to handle one interrupt synchronously, however we would like to schedule the interrupt handle so that it can be executed later on. This can be achieved dividing the `do_irq` function in two parts:

1. The **top half**. During the top half, executed by the interrupt handler routing, disable the interrupt that is handling (to avoid nesting) and schedules the work that has to be done by putting the functions to execute in some queues. The top half is executed as soon as the interrupt arrives and it's executed in a non-interruptable environment.
2. The **bottom half**. The bottom half is executed periodically, in a period called reconciliation points, in which functions are picked from the deferring queues and executed.

Linux uses multiple deferring queues, each containing different types of functions, each having different characteristics. The main types of functions are:

- SoftIRQs.
- Tasklets.
- Works.

SoftIRQs

SoftIRQs are simple non-blocking functions. When picking functions from softIRQ queues, the OS might decide to put two instances of the same function in execution on two different processors. This means that a function is executed in parallel on two processors, which might cause problems. Moreover, a softIRQ, can never be interrupted if on one processor only.

Tasklets

Tasklets solve the problem with more instances of a softIRQ running in parallel. In particular, only one tasklet is allowed to run on the processor at any time. In practice, tasklets are handled like softIRQs however, if the OS schedules multiple instances of the same tasklet, we are sure that each instance is run alone. Say, for instance, we have a tasklet to handle a key press. This tasklet might have to be executed multiple times since multiple processes might want to handle a key-press.

Thanks to tasklets, we are sure that each instance of the handler is executed alone (i.e., the handlers are serialised).

Tasklets are functions, which can't block or sleep, and some data. In practice, tasklets are represented by a `tasklet_struct` structure.

```
struct tasklet_struct {
    struct tasklet_struct *next;
    unsigned long state;
    /* other fields */
    void (*func)(unsigned long);
    unsigned long data;
};
```

Work queues

Work queues are used when we want to do some deferred interrupt work that has to sleep or block (e.g., acquire a semaphore). As for tasklets and softIRQs, work queues are made of functions to be executed and some data. The main advantage is that the functions can:

- Block.
- Sleep.
- Allocate memory.

The functions in a work queue are executed by a kernel thread called `events/n` (where `n` is the number of the processor that executes the work). A process that wants to declare some work that has to be done can use the `DECLARE_WORK` macro and signals the OS with the `schedule_work` function.

An example

Let us now consider an example to understand how Linux deals with deferred interrupts.

1. A process P enters kernel mode.
2. An interrupt I_1 is received.
3. The interrupt I_1 is handled, hence process P is preempted and the interrupt handler is executed.
4. An interrupt I_2, with higher priority is received. I_2 can be received since it's not of the same type as I_1.
5. I_2's handler executes the top half of the interrupt handler (disables all interrupts and enables all but the one that's handling).
6. I_2's handler schedules the actual work it has to do and calls the `raise_softirq` function which signals that some work has to be done later on.
7. I_2's handler terminates.
8. I_1's handler continues its execution.

9. Before terminating, I_1's handler calls the `do_softirq` function which checks if all previous interrupts have registered some work and executes some of the work with all interrupts enabled.
10. Process P is executed.
11. If some interrupt work is left to do, a kernel thread called `ksoftirqd` (one for each CPU) checks if some work is pending and invokes `do_softirq`.

15.3.6 Timers

Timers can be divided into two classes:

- **System timers**. The system timer is an hardware component that is built to generate a timer interrupt with a certain frequency called thick rate. The thick rate might change with the architecture and goes from 50 Hz to 1 kHz. Time sharing preemption relies on the system timer. Note that the system timer interrupt is different from the real time clock (RTC), which is the object that stores in memory the number of hours, minutes and seconds elapsed.
- **Dynamic timers**. Dynamic timers allow to dynamically register a timer. This is useful when a process wants to execute an action with a certain frequency or after some time, in fact the process can register a timer and the action that has to be executed whenever the timer expires.

System timer

The system timer invokes the `thick_periodic` function which

- Updates the `jiffies` variable.
- Handles the execution time for all the running processes. This means that if a process has been in execution for too long, it calls the scheduler.

The system timer's thick rate might change with the architecture and goes from 50 Hz to 1 kHz and can be changed at compile time. Increasing the frequency of the timer improves the precision of any activity involving time.

Dynamic timers

Dynamic timers can be registered at run-time by any process. In particular, a process has to

1. Declare a `struct timer_list` structure.

```
struct timer_list my_timer;
```

2. Initialise the timer.

```
init_timer(&my_timer);
my_timer.data = 0;
```

3. Define when the timer has to expire by setting the `expires` field.

```
int delay = 40;
my_timer.expires = jiffies + delay;
```

4. Define the function that has to be executed when the timer expires.

```
my_timer.function = my_function;
```

5. Call the `add_timer` to add the timer just created.

```
add_timer(&my_timer);
```

Chapter 16

Linux device management

16.1 Devices

Linux manages devices as if they were special files. In particular, each devices is given a file name and its stored under the `/dev/` directory. Devices can be divided into:

- Character devices, which are character streams. Writing and reading has direct impact on the device itself and no buffering is performed.
- Block devices, which are a sequence of blocks. Each block can be individually addressed and accessed through a cache using random access.

Depending on the device type, we have to use different techniques to work with the device. In particular:

- With character devices we can write and read characters (i.e., bytes) to and from the file.
- With block devices we have to do more complex operations.

Each device is controlled by a driver that manages the access to it. Each driver is identified by a **major device number**. If a driver handles multiple devices, each device is assigned also a **minor device number**. For instance, a driver might handle `dev/sda`, `dev/sda1` and `dev/sda2` which have the same major device number but different minor device number (i.e., 0, 1 and 2 respectively).

16.1.1 Dev folder structure

Legacy

In old systems, the `dev` directory contained every device that could possibly be attached to the system, independently from the fact that it was connected or not.

Devfs

Devfs is the evolution of `/dev/` that contains only the devices plugged into the system. The problem with this technology is that it forced some constraints on the devices' names. In particular, the names would change depending on the port to which the device was connected. This means that the system must have a database to store each connection and the relationships between device names and ports.

Udev

Udev provides a way of seeing each device with a name that is configurable. In particular, we can assign a name to a device and use it to assign the device to the correct file. The name can be configured in a configuration file as follows:

```
LABEL,BUS="usb",serial="HJASGTRYQTUIN",name="device_name"
```

In practice, every name represents both the major and minor numbers. If we attach a device to a different port, the system is able to recognise the correct file to which it should be mapped thanks to its custom name.

Sysfs

Udev allows to customise device names however it doesn't take into account how devices are connected to the system (i.e, their hierarchy). Sysfs has been introduced to solve this issue. Sysfs is a new part of the filesystem, hence it's not `/dev/`, but `/sys/` which shows how the devices are connected. This view is orthogonal to the one in `/dev/`. For instance, under `/sys/` we can check if multiple USBs are connected to the same PCI port.

16.2 Device drivers

Device drivers control access to a device through its file. In Linux, device drivers have to follow a specific pattern, in particular, we have to provide a data structure which is connected to the device file.

16.2.1 Character device drivers

Say we want to write a driver for a character device that sends and receives one character at a time. The first thing we have to do is creating a device under the `/dev/` directory, for instance `tty`. This operation can be done with the `mknod` command line command,

```
mknod /dev/<name> c <major> <minor>
```

or using device classes,

```
// create special file in /sys/class/sample
sample_class = class_create(THIS_MODULE, "sample");

// create special file /dev/sample_cdev0
device_create(sample_class, NULL, sample_dev_t, NULL, "sample_cdev%d", MINOR(
    sample_dev_t));
```

Then we have to initialise a `file_operations` data structure which defines, for each operation that can be done on a device, which function has to be called. In other words, we define a callback for each operation.

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    int (*open) (struct inode *, struct file *);
} __randomize_layout;
```

For instance, the program in Listing 16.1 shows how to create a simple character device driver.

```
/* Invoked on interrupt */
static irqreturn_t serial_irq(int irq, void *arg)

/* These two functions are our actual driver entry point for the application */
static ssize_t serialaos_read(struct file *f, char __user *buf, size_t size, loff_t
     *o) /* ... */
static ssize_t serialaos_write(struct file *f, const char __user *buf, size_t size,
     loff_t *o) /* ... */

/* The driver functions have to be associated to a file_operation structure */
static const struct file_operations serialaos_fops = {
    .write=serialaos_write,
    .read=serialaos_read,
};

static int __init serialaos_init(void) {
    /* Ask for access to the IO ports */
    if(!request_region(PORT_BASE, PORT_SIZE, "serialaos")) {
        /* manage error */
    }

    /* Ask for registering serial_irq interrupt */
    result=request_irq(PORT_IRQ, serial_irq, IRQF_SHARED, "serialaos", THIS_MODULE)
    ; if(result < 0) /* manage error */

    /* Ask for registering our driver (write and read ops) and get a major */
    major=register_chrdev(0, "serialaos", &serialaos_fops);
}
```

Listing 16.1: The code to initialise a character driver.

16.2.2 Block device drivers

With block devices, drivers don't directly interact with the device but, when they want to register a structure, they have to initialise two different structures:

- A `block_device_operations` structure that lists the callback for each operation that can be done on a block device.

```
struct block_device_operations {
    void (*submit_bio)(struct bio *bio);
    int (*open) (struct block_device *, fmode_t);
    /* other operations */
};
```

- A `gendisk` structure. This structure creates the actual block device since by creating a `gendisk` we also provide a way to register requests to the device itself for reading and writing blocks. In particular, the `gendisK` structure allows to declare, as one of its field, a function `queue_rq` that is invoked by the kernel when there is data to read or write. The `gendisk` structure is the kernel's representation of an individual disk device. It provides the capacity in 512 Kb sectors and pointers to the actual request queue that must be used for sending reading and writing commands.

A process accesses a disk with a `read()` or `write()` operation of a certain number of bytes. VFS is a module that handles two different aspects:

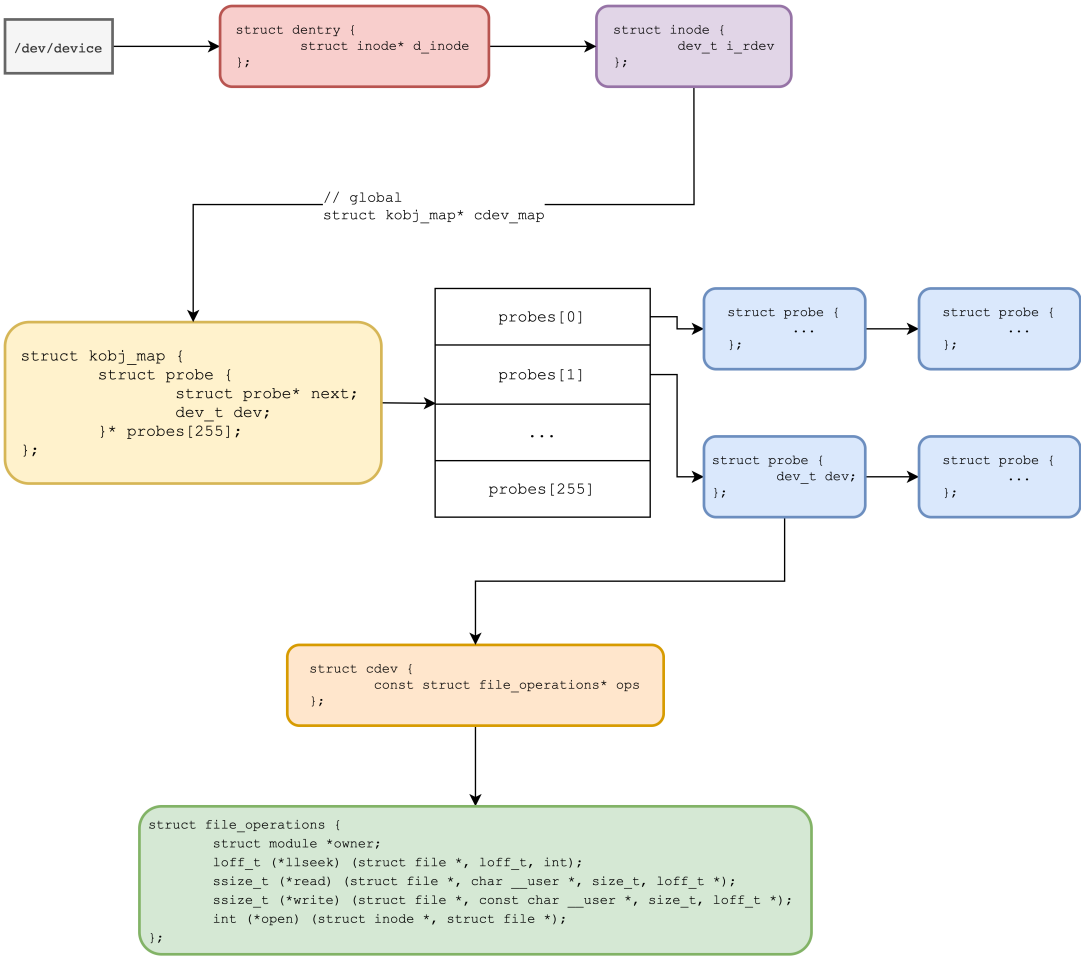


Figure 16.1: A summary of the structures used by a character device driver.

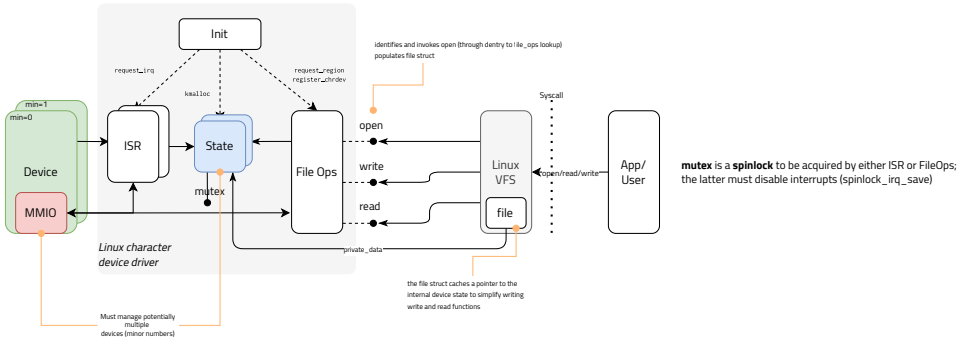


Figure 16.2: A summary of how character device drivers are used.

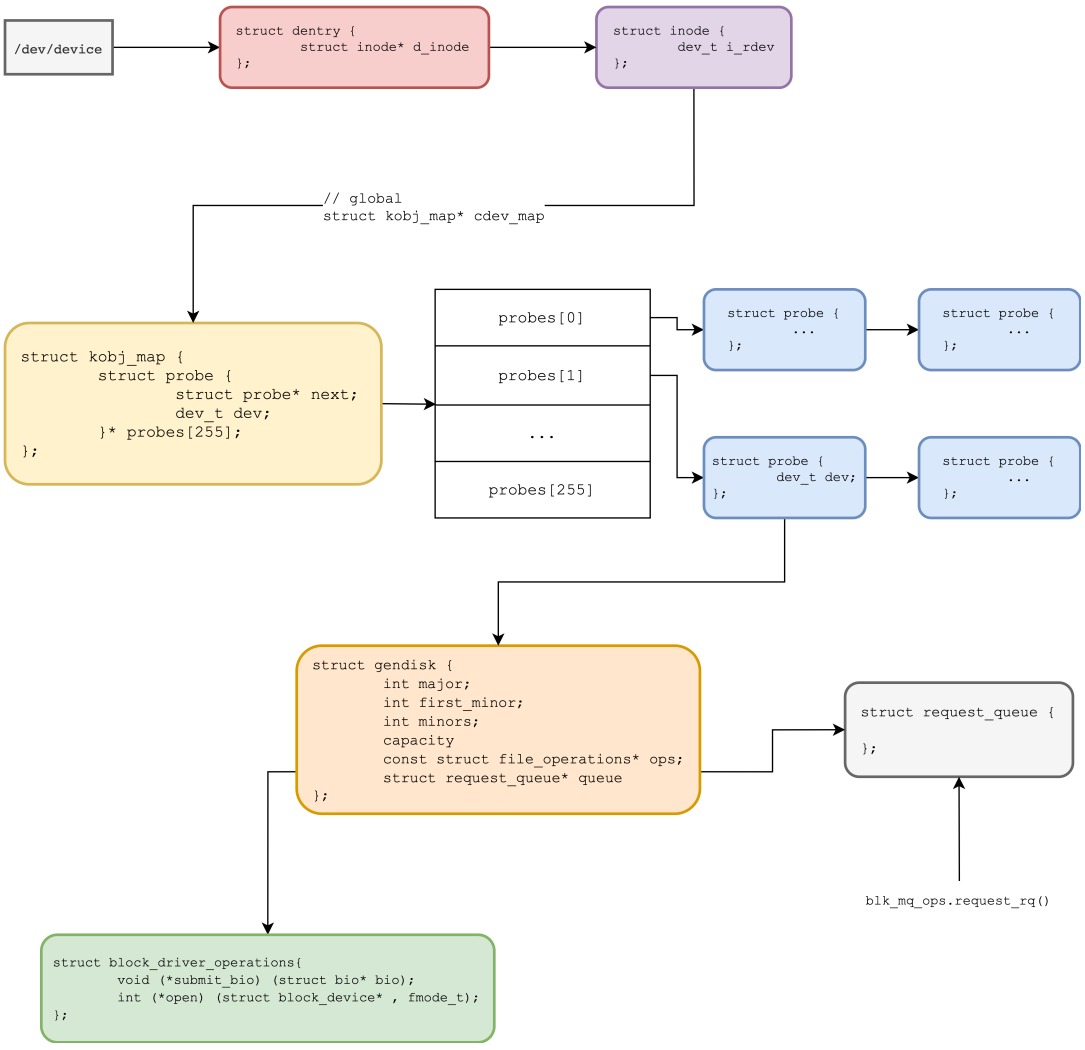


Figure 16.3: A summary of the structures used by a block device driver.

- It verifies if the data is already in memory (that is, resides in the kernel's page cache). If not, it sends the request for data to the mapping layer. The mapping layer accesses the file descriptor and pieces together the logical-to-physical mapping, therefore getting, at the end, the position of the actual disk blocks. At the end, it creates a request to the block layer through a structure called bio (block I/O).
- The block I/O layer tries to merge the list of bios into the least amount of actual requests to the driver.

Page cache

The page cache is implemented as a red-black tree that, for each offset in the file, tells us what page contains that offset of the file. Typically a page contains multiple blocks which are a multiple of a disk sector (512 b).

If the page cache does not contain the data, the mapping layer

1. Identifies the segments of a page that corresponds to contiguous sectors on disk. Internally the mapping layer works with multiples of sectors called blocks; for simplicity assume that 1 sector is 1 block.
2. Collects requests for segments that map to contiguous sectors on disk in a structure called `bio`. A `bio` references back to the original segments through a `bio_vec` pointer and contains data to iterate over it.
3. Once created, one or more `bio`s are sent to the block layer which will create the actual request to be sent to the device driver.
4. Before sending it, the block layer tries to merge several `bio` requests into single requests whenever possible. To do this, it uses a staging (software) request queue:
   (a) Initially, the generic block layer creates a request including just one `bio`.
   (b) Later, the I/O scheduler may extend the request either by adding a new segment to the original `bio`, or by linking another `bio` structure into the request. This is possible when the new data is physically adjacent to the data already in the request.
5. After a while, requests are moved from the staging queue to the actual hardware queue that will be read by the device driver and Once in a while, the `queue_rq` is called by the block layer and the device can fetch requests from the hardware queue and execute them.
6. The driver's `queue_rq` is invoked through a mechanism called plugging that adjusts the rate at which requests are dispatched to the device driver. Under a low load, operations to the driver are delayed allowing the block layer to perform more merges.

16.2.3 IO schedulers

I/O schedulers define what request should be handled next. I/O schedulers can have many purposes depending on the goals. Some examples are:

- To minimise time wasted by hard disk seeks (still relevant in some cases).
- To prioritise a certain processes' I/O requests.

- To give a share of the disk bandwidth to each running process.
- To guarantee that certain requests will be issued before a particular deadline.

Every IO scheduler has its goal and way to achieve it.

NOOP IO scheduler

The goal of NOOP IO schedulers is efficiency, hence to do as little as possible. To achieve this goal:

- Global I/O requests queue ordered in a First-In-First-Out (FIFO) fashion.
- The scheduler just merges requests to adjacent sectors to maximise the throughput.
- No sorting operations are executed since there is no seek latency minimisation goal.

This IO scheduler is suitable for storage devices not including mechanical parts since it

- Merely maintains the request queue in near-FIFO order.
- Only checks, when a request arrives, whether it can be merged with adjacent ones.

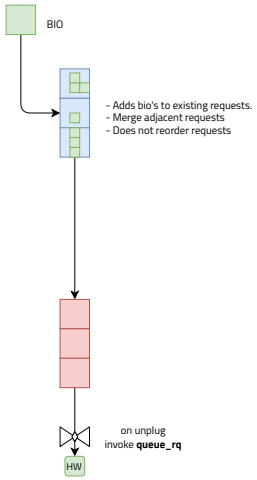


Figure 16.4: A NOOP IO scheduler.

Budget Fair Queuing IO scheduler

The goal of Budget Fair Queuing IO schedulers is to assign a fair amount of disk bandwidth. This is achieved by:

- Assigning a I/O budget to each process, i.e., the number of sectors each process is allowed to transfer.
- Once a process is selected, it has exclusive access to the storage device until it has transferred its budgeted number of sectors.

Budget Fair Queuing tries to preserve fairness overall, so a process getting a smaller budget now will get another turn at the drive sooner than a process that was given a large budget. The calculation of the budget is complicated and it's based on each process's I/O weight and observations of the process's past behaviour. The I/O weight functions like a priority value. It's set by the administrator (or by default) and is normally constant. Processes with the same weight should all get the same allocation of I/O bandwidth. Budget Fair Queuing IO schedulers have:

- Good responsiveness with slow devices.
- High per-operation overhead.

hence they are useful when I/O operation take a lot of time for instance, in mechanical disks.

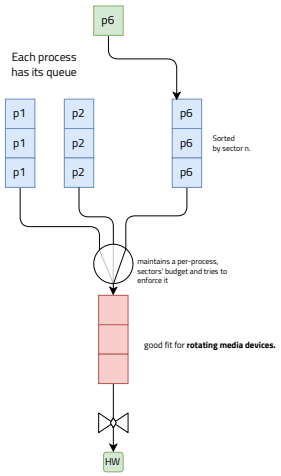


Figure 16.5: A Budget Fair Queuing IO scheduler.

Kyber IO scheduler

The goal of Kyber IO schedulers is to maximise throughput. This is achieved by using a simpler scheduler that allows for request merging and some simple policies, but mostly stays out of the way. Requests are split into primary queues (one for reads and one for writes) and no expiration for writes is scheduled (they are just throttled). This means that reads are favoured over writes since writes can stay in the queue without expiring. The administrator can configure indirectly the size of the hardware queues, in fact he or she specifies a latency target and the system measures how much current requests take to be executed and modifies the size of the queue accordingly. Note that, lower latency means smaller queues.

Kyber IO scheduler are intended for fast multi-queue devices (Flash) and lacks much of the complexity found in BFQ.

MQ-Deadline IO scheduler

The goal of MQ-Deadline IO schedulers is to try to do some merges, but then prioritise long starving reads. This is achieved by:

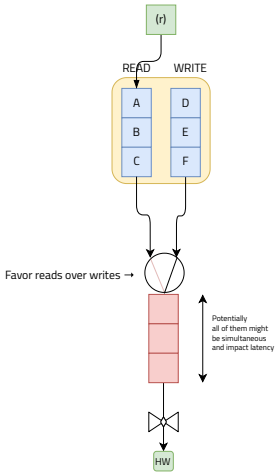


Figure 16.6: A Kyber IO scheduler.

- Assigning an expiration time to any incoming I/O request. The expiration time is shorter for reads (the default value is 500ms) longer for writes (the default is 5000ms).
- Using four queues: two separate FIFO queues read and write requests and two sector-wise sorted queues for READ and WRITE for merging requests.

Deadlines are just timers attached to each request that tick down to 0:

- The scheduler normally processes a burst of requests from the sorted queue (i.e., tries to execute requests for close sectors) and then checks if there is any expired request in the READ, first, and then the WRITE queue.
- Read latency is important to the performance of the system.

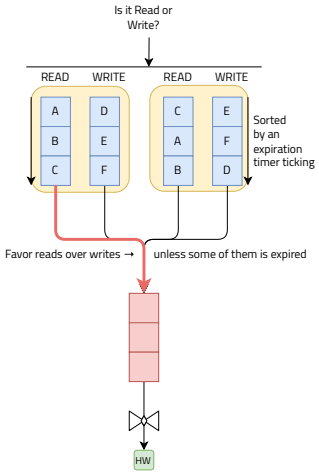


Figure 16.7: A MQ-Deadline IO scheduler.

Part VII

C++ operating system programming

Chapter 17

C++

17.1 Classes

17.1.1 Class definition

C++ is an object-oriented programming language. A class is defined using the `class` keyword. Namely, we write

```
class Car {
};
```

Usually a class is defined in an header file, i.e., a file with `.h` or `.hh` extension. **Constructors** are defined as a function with the same name of the class, no return type and a possibly empty list of parameters. If not defined, the compiler will generate a default constructor for us. In practice, we define a constructor as follows:

```
class Car {
    /*
     * A constructor with no parameters.
     */
    Car() {

    }
};
```

A C++ class also has a **copy constructor**, which is the constructor called when we declare an object and initialise it starting from another object. Since it's used for initialisation from another object, it takes the object (of the same class as argument) which is used for initialisation. As for the normal constructor, if not defined, the compiler will generate a default one.

```
class Car {
    /*
     * The copy constructor.
     */
    Car(const Car & c) {

    }
};
```

C++ classes also have a method which is called when an object is destructed. This method is called **destructor**. It does not take any parameter and should include cleanup and resource release code.

As always, if not defined, the compiler will generate a default one for us. To define the destructor of a class we have to write a method with the same name of the class, no return value or parameter and a ~ before the name of the method. In practice, we write:

```
class Car {
    /*
     * The destructor.
     */
    ~Car() {

    }
};
```

17.1.2 Encapsulation

C++ classes support encapsulation. In particular, we can use three different access modifiers:

- **Public**. Public members can be accessed by code outside of the object.
- **Private**. Private members can be accessed only by code inside the object.
- **Protected**. Protected members.

Members with the same access are specified after declaring the access modifier. In particular, we have to write the access modifier followed by a colon and the attributes or methods.

```
class Car {
public:
    int colour;

private:
    float fuel_level();

protected:

};
```

17.1.3 Inheritance

C++ classes support inheritance. We can specify that a class inherits from another class by writing, after the name of the class, the name of the class from which we want to inherit. The child class includes the public (and protected) members of the base class. Note that, if we want to redefine the child constructor, we have to call the parent's constructor.

```
class Vehicle {
public:
    Vehicle(int wheels): nr_wheels(wheels) {

    }

protected:
    int nr_wheels;
};

class Car: public Vehicle {
public:
    Car(): Vehicle(2) {} // invocation of the parent class constructor
};
```

17.1.4 Polymorphism

C++ classes support polymorphism. Member functions that can be overridden by derived classes are defined with the `virtual` specifier keyword in the parent class. In the child class we can use (it's not mandatory but strongly recommended) the `override` keyboard after the function's parameters.

```
class Vehicle {
public:
    virtual move_one_step_forward() {
        this.x += 1;
    }
};

class Car: public Vehicle {
public:
    void move_one_step_forward() override {
        this.x += 100;
        this.fuel_level -= 0.02;
    }
};
```

A method can also be **pure virtual** meaning that the children must provide an implementation. A pure virtual member function is defined using the keyword `virtual` and terminating the definition of the function with the keywords `= 0;`

```
class PowerManager {
public:
    virtual float get_temperature() = 0;
    virtual float get_power() = 0;
    virtual float get_current() = 0;
};
```

A class with all pure virtual methods is called **abstract**.

17.1.5 Member qualifiers

Members qualifiers are used to define the behaviour of a member function and are added after declaring the parameters of the function.

The `const` qualifier can be used to define read-only member functions, i.e., functions whose invocation will not change the status of the class instance.

```
class Car{
    int get_nr_wheels() const; // it will not affect the internal state
};
```

The `noexcept` specifier defines a non-throwing function, i.e., a function that doesn't throws exceptions.

```
class Car{
    int get_nr_wheels() noexcept;
};
```

17.1.6 Operators

17.2 Object creation

17.3 Templates

Templates provide a way to implement functions and classes without considering the type of data. Data type is then deducted at compile-time and the compiler generates code for us to cover the different cases. Template definition code is usually placed in header files. Albeit having many advantages, templates also have disadvantages like:

- Longer compilation times and bigger executable size.
- Modifying a template class often leads to a re-build of big part of the project.
- Debugging gets more complicated by length of the compiler's error messages.

For instance, we can define a class that can has a `add` method which can add either integers of floating point numbers.

17.4 Pointers

17.4.1 Basic pointers

C++ allows to allocate objects in the heap. The most basic way of allocating an object is by using the `new` keyword. Deleting the object (and freeing the memory) is instead done with the `delete` keyword.

```
1 class Singer {
2 };
3
4 int main() {
5     Singer * sing_rs = new Singer("Robert", "Smith");
6
7     delete sing_rs;
8 }
```

17.4.2 Smart pointers

C++ introduces, with respect to C, some pointers, which allow to avoid memory leaks (e.g., when returning from a function without having released the pointer). In particular, C++ has three types of smart pointers:

- **Unique pointers**.
- **Shared pointers**.
- **Weak pointers**.

Unique pointers

Unique pointers allow to have only one owner and use the Resource Acquisition Is Initialisation design pattern. This means that

- The resource ownership is acquired in the constructor by creating the object.
- The resource ownership is released in the destructor by destroying the object.

In practice, a unique pointer is represented by a `unique_ptr` template class that has as template type `T` the type of the pointer we want to initialise. A basic implementation of the unique pointer class is shown in Listing 17.1.

```
template <typename T>
class unique_ptr {
    T* ptr_ = nullptr;
public:
    unique_ptr() { ptr_ = new T; }
    ~unique_ptr() { if (ptr_) delete ptr_; }

}
```

Listing 17.1: Basic implementation of a unique pointer.

In many programs, a pointer is owned by different variables. Since unique pointers are exclusively hold by only one variable, we have to pass ownership from one variable to another. This is done in C++ using the `move` function,

```
template< class T >
constexpr std::remove_reference_t<T>&& move( T&& t ) noexcept;
```

This function is passed a unique pointer and it

1. Destroys the pointer passed, i.e, it revokes the pointer ownership from the variable that had that pointer.
2. Initialises a new pointer which points to the same object pointed to by the passed pointer.
3. Returns the new pointer.

Unique pointers are created using the `make_unique` function of the `memory` library. The program in Listing 17.2 shows an example of how unique pointers are used.

```
#include <memory>

/* Unique pointers are defined in the std namespace */
using namespace std;

int function() {
    unique_ptr<Singer> singer1;
    unique_ptr<Singer> singer1;

    // singer1 has exclusive ownership of the Singer object
    singer1 = make_unique<Singer>("Robert", "Smith");

    // singer2 obtains the ownership, which is revoked from singer1
    singer2 = move(singer1);
}

int main() {
```

```
18      function();
19      return 0;
20  }
```

Listing 17.2: An example of usage of unique pointers.

Shared pointers

Shared pointers allow to share ownership among multiple variables. Shared pointers are implemented using a reference counter that counts how many pointers are pointed to the same object, namely, how many owners an object has. The counter is incremented every time a pointer is copied and decremented every time a reference is removed. This allows to automatically destroy an object as soon as the counter reaches 0. This means that we can't have memory leaks since the object is immediately de-allocated when it has no owners. Similarly to unique pointers, shared ones are built using the `make_shared` function. The code in Listings 17.4.2 and 17.4.2 show how to use shared pointers.

```
1  #include <memory>
2
3  using namespace std;
4
5  shared_ptr create_singer(const string & name, const string & surname) {
6      shared_ptr<Singer> s = make_shared<Singer>(name, surname);
7      int err = check_something(s);
8
9      if (err != 0) {
10         cout << "An error occurred!" << endl;
11         /*
12         When the function returns, s is destroyed, hence the shared pointer has 0
       as counter and the memory it points to is automatically deallocated.
13         */
14         return nullptr;
15     }
16     return s;
17 }
```

```
1  class Person {
2  public:
3      virtual ~Person() {
4          // no more explicit delete
5      }
6
7      void SetChild(shared<Person> c) {
8          this->child = c;
9      }
10 private:
11     shared<Person> child;
12 }
13
14 int main() {
15     auto p1 = make_shared<Person>{"Alice"};
16     auto p2 = make_shared<Person>{"Bob"};
17     auto p3 = make_shared<Person>{"Carl"};
18
19     /*
20     When we pass a pointer to a function we know it will be handled correctly in
       all cases since it is a shared pointer
21     */
```

```
22      p1->SetChild(p3);
23      p2->SetChild(p3);
24  }
```

17.5 Function objects

The C++ provides the concept of function objects which are functions seen as objects so that the program can dynamically interact with them (e.g., dynamically pass parameters). In general a function object is a structure or class data type for which the `operator()` function call has been provided. An example of structure that is a function object is shown in Listing 17.3.

```
1  struct double_value {
2      int operator()(int value) {
3          return value * 2;
4      }
5  };
6
7  int main(int argc, const char *argv[]) {
8      double_value multiply_by_2;
9      int dv = multiply_by_2(20);
10     return 0;
11 }
```

Listing 17.3: A struct used as function object.

Additionally, the header `functional` defines the `function` class type which is a polymorphic function wrapper which allows us to store, copy and invoke any `CopyConstructibleCallable` object. An example of creation of a function object as `functional` class is shown in Listing 17.4.

```
1  #include <functional>
2
3  int double_value(int value) {
4      return value * 2;
5  }
6
7  struct DoubleValue {
8      int operator()(int value) {
9          return value * 2;
10     }
11 };
12
13 int main(int argc, const char *argv[]) {
14     // Initialized by a function
15     std::function<int(int)> fn1(double_value);
16
17     // Initialized by a function object
18     DoubleValue mul_by_2;
19     std::function<int(int)> fn2(mul_by_2);
20 }
```

Listing 17.4: An example of usage of the functional class.

17.5.1 Bind expressions

The `functional` library also provides bind expressions which allow to define a function and dynamically pass parameters to it. The idea is to use the `bind` function to pass some parameters to a

certain function without actually executing the function. The return value is a function object which can be called later on using as parameters the values passed to the `bind`. The nice part about bind expression is that we can also pass a `std::placeholder` to the `bind` function so that we can pass the actual value when we execute the function object. Basically, the `bind` function allows to define some fixed values which are always passed to the function object and some dynamic values which are passed only when executing the function object. The code in Listing 17.5 shows this mechanism.

```
#include <functional>

using namespace std::placeholders;

int multiply(int x, int y) {
    return x * y;
}

int main(int argc, char * argv[]) {
    /*
    Bind the placeholder _1 and the value 2 to the function multiply.
    */
    std::function<int (int, int)> multiply_by_2 = std::bind(multiply, _1, 2);

    /*
    Call and execute the function multiply_by_2 passing 100 as first parameter
    */
    std::cout << multiply_by_2(100) << std::endl;
}
```

Listing 17.5: Bind expressions.

17.5.2 Lambda expressions

In C++ we can also define anonymous function objects called lambdas. The syntax for creating a lambda expression is

```
[] (/* parameters */) -> <return type> { /* function body */ }
```

where

- `[]` is the capture clause, which is optional.
- `()` is the list of parameters to pass to the function.
- `{}` is the function body.

A lambda expression can be assigned to a variable so that we can call it later on. An example is shown in Listing 17.6

```
auto f = [](int a, int b) -> int { return a * b % 2; };
cout << "3 * 4 % 2 == " << f(3, 4) << endl;
```

Listing 17.6: A lambda expression.

One of the most distinctive features of a lambda expression is capture clause. The capture clause "captures" the variables from the outer scope making them visible to the body of the expression. The capture can be performed by value or by reference. In particular, in the capture we can write:

- `[&]` to capture by reference any variable in the local block scope.

- `=` to capture by copy any variable in the local block scope.
- `[&var_name]` to capture by reference only the variable `var_name` (and no other variable).
- `[var_name]` to capture by copy only the variable `var_name` (and no other variable).
- `[this]` to capture the `this` pointer of the enclosing object.

This expressions can be combined to obtain complex capture expressions, for instance we can write `[=, &var]` to say that we want to capture all variables by copy and the variable `var` by reference. Capture expressions are used, for instance, in Listing 17.7.

```
#include <algorithm>
#include <iostream>
#include <vector>

using namespace std;

class Scale {
public:
    Scale(int scale) : _scale(scale) {}

    void apply(const vector<int>& v) const {
        for_each(v.begin(), v.end(), [this](int n) { cout << n * this._scale <<
    endl; });
    } private:

    int _scale;
};

int main() {
    vector<int> values = { 1, 2, 3, 4 }; Scale s(3);
    s.apply(values);
}
```

Listing 17.7: A capture expression used for lambdas.

Chapter 18

Thread programming

18.1 Threads

A thread is defined in C++ as an instance of the `thread` class included in the `thread` library under the `std` namespace. To use this class we have to add the following flag when compiling:

```
-std=c+11 -pthread
```

The `thread` classes offers, among others, the following member functions:

- ```
  std::thread::id get_id() const noexcept;
  ```

  which returns the identifier of the thread.

- ```
  void detach();
  ```

  which separates the thread of execution from the thread object, allowing execution to continue independently. Any allocated resources will be freed once the thread exits.

- ```
  void join();
  ```

  which blocks the current thread until the thread identified by `*this` finishes its execution. The completion of the thread identified by `*this` synchronises with the corresponding successful return from `join()`. No synchronisation is performed on *this itself. Concurrently calling `join()` on the same thread object from multiple threads constitutes a data race that results in undefined behaviour.

- ```
  bool joinable() const noexcept;
  ```

  which checks if the `std::thread object` identifies an active thread of execution. Specifically, returns true if `get_id() != std::thread::id()`. So a default constructed thread is not joinable.

- ```
  static unsigned int hardware_concurrency() noexcept;
  ```
  ```

which returns the number of concurrent threads supported by the implementation. The value should be considered only a hint.

- `thread& operator=( thread&& other ) noexcept;`

which, if `*this` still has an associated running thread (i.e. `joinable() == true`), calls `std::terminate()`. Otherwise, assigns the state of `other` to `*this` and sets `other` to a default constructed state. After this call, `this->get_id()` is equal to the value of `other.get_id()` prior to the call, and other no longer represents a thread of execution.

C++ also defines the `std::this_thread` namespace that contains a group of functions to access the current thread:

- `std::thread::id get_id() noexcept;`

which returns the id of the current thread.

- `void yield() noexcept;`

which suspends the current thread, allowing other threads to be scheduled to run.

- ```
  template< class Rep, class Period >
  void sleep_for(const std::chrono::duration<Rep, Period>& sleep_duration);
  ```

which blocks the execution of the current thread for at least the specified `sleep_duration` (or more if required by scheduling).

- ```
  template< class Clock, class Duration >
  void sleep_until(const std::chrono::time_point<Clock,Duration>& sleep_time
  );
  ```

which blocks the execution of the current thread until specified `sleep_time` has been reached.

18.1.1 Thread creation

We can create a thread simply initiating a `thread` object. One thread constructor (for the other check here) is

```
template< class Function, class... Args >
explicit thread( Function&& f, Args&&... args );
```

hence we have to pass a pointer to a function object `f` and all the values that have to be passed as parameters to the function `f`.

18.2 Synchronisation

18.2.1 Locking classes

Using standard mutexes can lead to bug and errors. For instance, if one forgets to unlock a mutex before returning in case of error, the program might end in a deadlock. The C++ programming

language offers classes that implement the Resource Acquisition Is Initialisation paradigm and automatically lock and unlock mutexes. They are in fact wrappers of mutexes and are able to call the `unlock` member function whenever the locking class is destroyed (e.g., at the end of a function that returns). In particular, there exists two different locking classes:

- `lock_guard<>`.
- `unique_lock<>`.

In both cases, a locking class is a template class that takes a `Mutex` and handles its locking. The class locks the mutex as soon as it is instantiated and unlocks it when it's destroyed (i.e., when the program goes out of the lock's scope).

Lock guards

A `lock_guard` has only a constructor, which is used to lock the mutex passed as argument and a destructor, which unlocks the mutex. This means that, if we want to use a `lock_guard`, we have to:

1. Declare a mutex `m` and a `guard_lock` `lock`.
2. Initialise the `lock` using the mutex `m`.

Unlocking is handled as soon as `lock` goes out of scope and is destroyed. The code in Listing 18.1 shows what we have just described.

```
mutex m;
int shared_var;

void thread_func() {
    // obtain the lock
    guard_lock<mutex> lck(m);
    if (shared_var > 1000) {
        // the mutex is unlocked here, too since after the return lck is destroyed.
        return;
    }
    // access the shared variable
    shared_var++;

    // the lock is released after the return as soon as lck is destroyed
    return;
}
```

Listing 18.1: Basic usage of a guard lock.

Since a lock guard releases the mutex as soon as it (the guard) goes out of scope, we can use a code block to specify the part of code that has to be handled by the lock guard. An example is shown in Listing 18.2

```
mutex m;
int shared_var;

void thread_func() {
    cout << "Thread started" << endl;

    {
        // obtain the lock
        guard_lock<mutex> lck(m);
        if (shared_var > 1000) {
```

```
11              // the mutex is unlocked here, too since after the return lck is
    destroyed.
12              return;
13          }
14          // access the shared variable
15          shared_var++;
16          // the mutex is released here, immediately after going out of the scope in
    which the guard has been declared
17      }
18
19      cout << "Returning" << endl;
20      return;
21  }
```

Listing 18.2: A lock guard used only on a limited function scope

Unique locks

Unique locks do everything lock guards do and they add some some functionalities to implement ownership. In particular, a unique mutex can be owned by only one unique lock and ownership is moved from one lock to another. Moreover, locking can be deferred, namely, the mutex is locked when necessarily and not when the lock is initiated. Differently from lock guards, unique locks can own `mutex`es, `recursive_mutex`es or `timed_mutex`es. Unique locks can also explicitly lock and unlock mutexes using the following member functions:

- The

  ```
  void lock();
  ```

  function locks (i.e., takes ownership of) the associated mutex.

- The

  ```
  bool try_lock();
  ```

  function tries to lock (i.e., takes ownership of) the associated mutex without blocking.

- The

  ```
  template< class Rep, class Period >
  bool try_lock_for( const std::chrono::duration<Rep,Period>&
  timeout_duration );
  ```

  function tries to lock (i.e., takes ownership of) the associated mutex. It blocks until the specified `timeout_duration` has elapsed or the lock is acquired, whichever comes first. On successful lock acquisition it returns true, otherwise it returns false.

- The

  ```
  template< class Clock, class Duration >
  bool try_lock_until( const std::chrono::time_point<Clock,Duration>&
  timeout_time );
  ```

function tries to lock (i.e., takes ownership of) the associated mutex. It blocks until the specified `timeout_time` has been reached or the lock is acquired, whichever comes first. On successful lock acquisition it returns true, otherwise it returns false. The function may block for longer than `timeout_time` until has been reached.

- The

```
void unlock();
```

function unlocks (i.e., releases ownership of) the associated mutex and releases ownership.

- The

```
mutex_type* release() noexcept;
```

function breaks the association of the associated mutex, if any, and `*this`.

- The

```
unique_lock& operator=( unique_lock&& other );
```

operator allows to move ownership of the mutex taking it from `other`.

Scoped locks

Scoped locks are lock classes that allow to lock multiple mutexes. Scoped locks solve the problem of deadlocks caused by locking mutexes in different order (e.g., T_1 locks M_1 and then M_2 whilst T_2 locks M_2 and then M_1). In particular:

- The `scoped_lock`'s constructor takes all the mutexes that have to be locked and locks them in a specific order.
- The `scoped_lock`'s destructor unlocks the mutex in inverse order with respect to the one in which they have been locked.

An example is shown in Listing 18.3.

```
1  mutex m1;
2  mutex m2;
3
4  void function1() {
5      scoped_lock lck(m1, m2);
6      /* code to synchronise */
7  }
8
9  void function2() {
10     scoped_lock lck(m1, m2);
11     /* code to synchronise */
12 }
```

Listing 18.3: Usage of scoped locks

Shared locks and mutexes

Shared mutexes allow to define two types of ownership. A mutex can be:

- Shared. In this case it can be owned by multiple threads (i.e., multiple threads can lock the same mutex).
- Exclusive. In this case the lock can be owned by only one thread.

This type of mutex is used to implement the read-write lock, in fact the mutex can be locked

- By multiple readers using the shared ownership.
- By only one writer using the exclusive ownership.

In practice, a shared mutex has the same method of a normal mutex (which are used for exclusive ownership) and the following member functions:

- The function

  ```
  void lock_shared();
  ```

  which locks the mutex using shared ownership.

- The function

  ```
  void unlock_shared();
  ```

  which unlocks the mutex using shared ownership.

- The function

  ```
  bool try_lock_shared();
  ```

  which tries to lock the mutex using shared ownership and, if the operation is successful, it returns true, otherwise it returns false.

The `shared_lock` class is just a wrapper for the `shared_mutex` that implements the RAII paradigm. A `shared_lock` has the same member functions of a `unique_lock` which are wrappers of the shared lock functions of the mutex. This means that we can implement read-write locks using

- A `unique_lock` with a `shared_mutex` for writing. The unique lock calls the mutex's exclusive locking functions, hence it actually implements exclusive locking.
- A `shared_lock` with a `shared_muted` for reading. The shared lock calls the mutex's shared locking functions, hence it actually implements exclusive locking.

18.2.2 Condition variables

Condition variables synchronise threads in which one is waiting for some data produced by another. In this case, differently from mutexes, a thread is blocked not because it wants to concurrently access some data but because it has to wait some data which hasn't been produced yet. A condition variable allows threads to wait on a queue. Other threads can then wake up (or notify) the threads waiting in the queue. Because a condition variable works on shared data (between producer and consumer), a condition variable also has to use a mutex to manage concurrent access to the shared data. In particular, we have to acquire a `unique_lock` before using a condition variable and then pass the lock when waiting.

A condition variable is implemented in C++ as an instance of the `condition_variable` class which exposes the following member functions:

- The function

  ```
  void wait( std::unique_lock<std::mutex>& lock );
  ```

  blocks the current thread until the condition variable is notified or a spurious wake-up occurs. The `unique_lock` object is unlocked for the duration of the wait.

- The function

  ```
  template< class Rep, class Period >
  std::cv_status wait_for( std::unique_lock<std::mutex>& lock, const std::
  chrono::duration<Rep, Period>& rel_time)
  ```

  blocks the current thread until another thread wakes it up, or a time span has passed.

- The function

  ```
  void notify_one() noexcept;
  ```

  wakes up one of the waiting threads.

- The function

  ```
  void notify_all() noexcept;
  ```

  wakes up every waiting thread.

A practical example of how condition variables are used is shown in Listing 18.4

```
1  #include<mutex>
2  #include<condition_variable>
3
4  char shared_char = 255;
5  std::mutex m;
6  std::condition_variable cond_var;
7
8  void produce_char() {
9      char c;
10
11     std::cin >> c;
12     {
```

```
13          std::unique_lock<mutex> lock(m);
14          shared_char = c;
15          cond_var.notify_one();
16      }
17  }
18
19  void consume_char() {
20      char c;
21      {
22          std::unique_lock<mutex> lock(m);
23          while(shared_char > 127)
24              cond_var.wait(lock);
25          c = shared_char;
26      }
27
28      std::cout << "Received: " << c << std::endl;
29  }
```

Listing 18.4: An example of a program using condition variables.

Condition variables are generalised using the `condition_variable_any` class, which can use any type of lock. This class has the same member functions of the `condition_variable` class and, albeit it can be used with all locks (even `shared_locks`), it has worst performance than a simple `condition_variable`.

Chapter 19

Drivers

19.1 Miosix

Miosix is a real-time operating system optimised for microcontrollers. User-space is optional and applications can be run directly in kernel-space.

19.1.1 Booting

Linker file

Miosix works directly with physical memory. This means that, when booting, we have to specify at what address each section has to be loaded. This is done through a linker script (in a `.ld` file) that specifies, to which physical address each memory section (e.g., stack, bss, text and so on) has to be mapped. Physical memory can be divided in three regions:

- Peripherals.
- RAM.
- Flash.

For each region, we specify in the linker file:

- At which address that region starts and ends.
- Which sections are placed in that region.

The first thing we have to specify when writing a linker file is specifying what is the first function that should be executed when booting the system. This is done using the `ENTRY` command to which we specify the name of the function to execute.

```
ENTRY(Reset_Handler)
```

Next, we have to define the layout of the memory, hence what regions are present, how big they are and at what address they start. This is done in the `MEMORY` section. At the end of this section we also have to define at what address the stack ends writing the `_stack_top` variable.

```
MEMORY
{
    flash(rx)   : ORIGIN = 0x08000000, LENGTH =   1M
    ram(wx)     : ORIGIN = 0x20000000, LENGTH = 128K
}

_stack_top = 0x20000000 + (128 * 1024);
```

Now we can write the mapping between sections (text, bss, ...) and memory regions. This is done in the `SECTIONS` section. In this section, the variable `.`, which is called location counter, stores how much memory has been mapped, in fact it's incremented by the section size after each section.

```
SECTIONS {
    . = 0;
}
```

In this section we can map a memory section to a memory region using the syntax

```
.section :
{
    /* define regions to map */
} > .memory_region
```

Inside the curly brackets we can specify what sections should be mapped to the memory region `memory_region` using the syntax `*(.section_name)`. This means that, if we want to map the text section to flash memory we have to write:

```
SECTIONS
{
    . = 0;

    .text :
    {
        /* Startup code must go first */
        KEEP(*(.isr_vector))
        *(.text)
        . = ALIGN(4);
        *(.rodata)
    } > flash
}
```

When writing a section, we might want to align it at a certain number of bytes, depending on the Application Binary Interface. This can be specified assigning the result of `ALIGN(align_bytes)` to the location counter. If we want to use C++, we also have to include the `.init_array` section in the text section. This means that, if using C++, we have to write:

```
SECTIONS
{
    . = 0;

    .text :
    {
        /* Startup code must go first */
        KEEP(*(.isr_vector))
        *(.text)
        . = ALIGN(4);
        *(.rodata)

        . = ALIGN(4);
        __init_array_start = .;
        KEEP(*(.init_array))
```

```
        __init_array_end = .;
    } > flash
}
```

In some cases, we might want to put some data in a region of memory (e.g., in flash), which is called shadow region, so that it can be initialised there and be kept there for future boots but then map the section where that data is to another region. Basically, we put the initialised data in a non-volatile part of the memory and then, at booting, we copy it to another section. This can be done using the syntax

```
.section :
{
    /* define regions to map */
} > .memory_region AT > shadow_region
```

If we want to map the data section to ram but store it also in flash for initialisation we have to write:

```
SECTIONS
{
    . = 0;

    .text :
    {
        /* Startup code must go first */
        KEEP(*(.isr_vector))
        *(.text)
        . = ALIGN(4);
        *(.rodata)
    } > flash

    .data :
    {
        _data = .;
        *(.data)
        . = ALIGN(8);
        _edata = .;
    } > ram AT > flash
    _etext = LOADADDR(.data);
}
```

Adding the data section we have also set some variables:

- The `_data` variable stores the counter of the first byte of the data section.
- The `_data` variable stores the counter of the last byte of the data section.

Moreover, using the `LOADADDR` function we can store a pointer of the first byte of the shadow copy in the variable `_etext` which marks the end of the text section.

Finally, we can map the bss section to the ram region. In this case we don't have initialised data that has to be preserved between sessions, hence we don't have to use the shadow copy syntax. What we get is:

```
SECTIONS
{
    . = 0;

    .text :
    {
        /* Startup code must go first */
```

```
        KEEP(*(.isr_vector))
        *(.text)
        . = ALIGN(4);
        *(.rodata)
    } > flash

    .data :
    {
        _data = .;
        *(.data)
        . = ALIGN(8);
        _edata = .;
    } > ram AT > flash
    _etext = LOADADDR(.data);

    _bss_start = .;
    .bss :
    {
        *(.bss)
        . = ALIGN(8);
    } > ram
    _bss_end = .;

    _end = .;
}
```

Even in this case, we store in the variables `_bss_start` and `_bss_end` a pointer to the start and end of the bss section. The complete linker script is shown in Listing 19.1.

All the variables we have defined in the linker scripts (e.g., `etext`) can be used at runtime to initialise sections.

```
1  ENTRY(Reset_Handler)
2
3  MEMORY
4  {
5      flash(rx)    : ORIGIN = 0x08000000, LENGTH =   1M
6      ram(wx)      : ORIGIN = 0x20000000, LENGTH = 128K
7  }
8
9  _stack_top = 0x20000000 + (128 * 1024);
10
11 SECTIONS
12 {
13     . = 0;
14
15     .text :
16     {
17         /* Startup code must go first */
18         KEEP(*(.isr_vector))
19         *(.text)
20         . = ALIGN(4);
21         *(.rodata)
22
23         . = ALIGN(4);
24         __init_array_start = .;
25         /* Include if using C++
26         KEEP(*(.init_array))
27         */
28         __init_array_end = .;
29
```

```
30      } > flash
31
32      .data :
33      {
34          _data = .;
35          *(.data)
36          . = ALIGN(8);
37          _edata = .;
38      } > ram AT > flash
39      _etext = LOADADDR(.data);
40
41      _bss_start = .;
42      .bss :
43      {
44          *(.bss)
45          . = ALIGN(8);
46      } > ram
47      _bss_end = .;
48
49      _end = .;
50  }
```

Listing 19.1: The linker script used to boot the Miosix kernel

Start-up code

At boot time, we have to tell the processor what is the instruction pointer of the first instruction that has to be executed. How this operation is implemented depends on the processor itself. On a PC, at booting we start executing the BIOS/EFI code that loads in RAM a second-level bootloader like GRUB. In microcontrollers, we have to set the program counter to an address of the flash memory (which is not volatile). The program counter can be set in two ways:

- It's set to a predefined address and it starts executing from there.
- It loads the address of the first instruction from a predefined memory address and starts executing from the loaded address.

In any case, the first instruction can't be the instruction of a C function since C makes some assumptions about the environment in which it runs which are not satisfied during booting. This means that the first part of the start-up code has to be written in assembler. In this initial part we setup the environment so that we can execute C code. In particular, we have to

- Set the stack pointer register to the top of a suitable memory area.
- Set static and global initialised variables to their initial value.
- Set static and global uninitialised variables to 0.

Moreover, if we are using C++, we have to call the constructors of the global objects. Since we are considering microcontrollers, we will use ARM assembler code. The code in Listing 19.2 shows the start-up code for Miosix.

```
1       .syntax unified
2       .cpu cortex-m4
3       .thumb
4
```

```
5     .section .isr_vector
6     .global __Vectors
7 __Vectors:
8     .word _stack_top
9     .word Reset_Handler
10
11    .section .text
12    .global Reset_Handler
13    .type  Reset_Handler, %function
14 Reset_Handler:
15 /* Copy .data from FLASH to RAM */
16    ldr r0, =_data
17    ldr r1, =_edata
18    ldr r2, =_etext
19    cmp r0, r1
20    beq  nodata
21 dataloop:
22    ldr  r3, [r2], #4
23    str  r3, [r0], #4
24    cmp  r1, r0
25    bne  dataloop
26 nodata:
27 /* Zero .bss */
28    ldr r0, =_bss_start
29    ldr r1, =_bss_end
30    cmp r0, r1
31    beq nobss
32    movs r3, #0
33 bssloop:
34    str  r3, [r0], #4
35    cmp  r1, r0
36    bne  bssloop
37 nobss:
38    /* Call global contructors for C++
39        Can't use r0-r3 as the callee
40        doesn't preserve them          */
41    ldr  r4, =__init_array_start
42    ldr  r5, =__init_array_end
43    cmp  r4, r5
44    beq  noctor
45 ctorloop:
46    ldr  r3, [r4], #4
47    blx  r3
48    cmp  r5, r4
49    bne  ctorloop
50 noctor:
51 /* Jump to kernel C entry point */
52    bl kernel_entry_point
53    .size Reset_Handler, .-Reset_Handler
```

Listing 19.2: Start-up code for executing C code.

Let us comment what this code does:

- Lines `1` to `3` define the assembler file characteristics.
- Lines `7` is the first instruction executed and places the pointer `_stack_top` at the first address of flash (i.e., where code is executed at booting).
- Lines `8` places the pointer `Reset_Handler` at flash's `ORIGIN` + `0x4` (i.e., at `0x08000004` for what we have defined in the linker script). This is done because the bootloader will load the

address `FLASH_ORIGIN+0x4` in the instruction pointer and start executing the instruction at that address.

- At line `13` we define the `Reset_Handler` function, which is the first function executed, as defined in the linker script.
- From lines `13` to `24` we define a loop that is used to initialise the data in the `data` section. In particular, we want to copy initialised values from flash to RAM. This operation is possible thanks to the variables `_data` and `_etext`, defined in the linker script, that point to the beginning of the data section in RAM and flash, respectively. This is because `_etext` is the end of the text section, which is in flash memory, hence where the text section, it starts the data section since we put it immediately after it. The actual data copy is done in the `dataloop` loop.
- From lines `26` to `35` we initialise the data in the `bss` section by putting all variables to 0.
- From lines `38` to `49` we call the constructors of the C++ global objects.
- Finally, we jump to the C kernel entry point.

19.1.2 Concurrency

Miosix supports C/C++ standard libraries and POSIX also in kernel-space, hence we can use everything C++ offers to handle synchronisation.

Interrupt synchronisation

In a single core system, an interrupt can freely access a shared variable without locks since an interrupt can't be blocked, hence when it's executing, no other process can access the shared data.

On the other hand, normal code has to disable all interrupts when accessing a variable that can be accessed by an interrupt. This is because if it didn't, normal code could be blocked by an interrupt while accessing a shared variable. Miosix offers the following privimitives for disabling interrupts:

- The class

  ```
  class FastInterruptDisableLock
  ```

  disables interrupts in the constructor, enables them in the destructor. When instantiating this class, we must be sure that interrupts were enabled prior to calling this. Conceptually, this class is equivalent to a regular mutex.

- The class

  ```
  class InterruptDisableLock
  ```

  is conceptually equivalent to a `recursive_mutex`, counts number of times called and enables interrupt back at the right time.

- The class

  ```
  class FastInterruptEnableLock
  ```

allows to temporarily enable back interrupts in a scope where they were disabled. Constructor takes the lock disabling interrupts as parameter.

- The class

```
class InterruptEnableLock
```

allows to temporarily enable back interrupts in a scope where they were disabled. Constructor takes the lock disabling interrupts as parameter.

Normal code also has to be able to wait for an interrupt. Miosix provides the member functions `IRQwait()` and `IRQwakeup` of the class `Thread` (since every non-interruptable code is an instance of `Thread`). In particular:

- The member function

```
IRQwait();
```

marks the calling thread as blocked (i.e., it will no longer be scheduled). This function must be called with interrupts disabled, and immediately afterwards it is necessary to enable interrupts and call the `yield()` member function of the thread to complete the blocking.

- The member function

```
IRQwakeup();
```

marks this thread as ready (i,e, it will be scheduled). If this is the highest priority thread, the scheduler can be called to schedule it immediately.

This interface is similar to the one provided by condition variables. Miosix also provides a class `Queue` which implements a producer-consumer with interrupts, in the form of a synchronised queue. In particular, the `Queue` class has:

- A member function

```
bool IRQget(T& elem, bool& hppw);
```

which allows to get an element from the queue saving it in `elem`. If the queue is empty, the caller is blocked like when using the `IRQwait` member function.

- A member function

```
bool IRQput(const T& elem, bool& hppw);
```

which allows to add the element `elem` to the queue. After calling the put function, waiting threads are awakened (like when using `IRQwakeup`).

The class is shown in Listing

```
1 //A synchronized queue with len elements of type T
2 template<typename T, unsigned len>
3 class Queue{
4 public:
5     //To be called from the normal code side (can block if queue empty/full)
6     void get(T& elem);
7     void put(T& elem);
8     //To be called from the interrupt side (these never block)
9     //return true if operation was not successful (queue was not empty/full)
10    //optional parameter hppw will be set to true if the operation caused the
      wakeup
11    //of a higher priority thread
12    bool IRQget(T& elem, bool& hppw);
13    bool IRQput(const T& elem, bool& hppw);
14 };
```

Listing 19.3: The queue class used for waiting interrupts in the Miosix kernel. label

19.1.3 The file-system

Devices have to be registered under the `/dev` directory to be used. This is achieved by creating a subclass of the `Device` class and override its member functions, which define the methods for interacting with the device file. The `Device` class contains, for instance, the member functions for reading from and writing to the device file and by overriding them we define how to read and write the device file.

```
class Device
{
public:
    virtual ssize_t readBlock(void *buffer, size_t size, off_t where);
    virtual ssize_t writeBlock(const void *buffer, size_t size, off_t where);
    virtual int ioctl(int cmd, void *arg)
};
```

After having defined the `Device` class, we have to actually register the device and the driver we have written by overriding its member functions. This can be achieved using the member function `addDevice` to the `devFs` in the board suppport package as follows:

```
devFs->addDevice("device_name", intrusive_ref_ptr<Device>(new MyDriver));
```

19.2 Linux kernel

IO ports used to interact with peripheral are at physical addresses in the Linux kernel, hence we need some functions to obtain access to the peripherals and to their physical addresses. In particular, we have to use the macro

```
request_region(peripheral_base_addr, peripheral_size, str_drivername);
```

to request access for the physical address `peripheral_base_addr` and the macro

```
release_region(peripheral_base_addr, peripheral_size);
```

to release the physical address `peripheral_base_addr`. After having obtained access to the IO ports of a peripheral (i.e., to the physical memory addresses) we can use the functions

```
extern u8       inb(unsigned long port);
extern u16      inw(unsigned long port);
extern u32      inl(unsigned long port);
```

to read a byte, a word (2 bytes) or double word (4 bytes) from port `port` and the functions

```
extern void   outb(u8 b, unsigned long port);
extern void   outw(u16 b, unsigned long port);
extern void   outl(u32 b, unsigned long port);
```

to write the byte, word or double word `b` to port `port`.

19.2.1 Concurrency

Linux doesn't support POSIX inside the kernel, hence we can't use mutexes, POSIX threads and other POSIX constructs to handle concurrency in the kernel. This means that we have to redefine:

- How kernel threads are created and handled.
- How kernel threads are synchronised.

Kernel threads

Kernel threads are created using the `kthread_run` macro, defined as follows:

```
#define kthread_run(threadfunc, data, namefmt, ...)({                  \
    struct task_struct *__k                                            \
        = kthread_create(threadfunc, data, namefmt, ## __VA_ARGS__);   \
    if (!IS_ERR(__k))                                                  \
        wake_up_process(__k);                                          \
    __k;                                                               \
})
```

This macro creates a thread, which is stored in a pointer to a `task_struct` that executes the function `threadfunc`. Note that this macro also executes the thread (or at least awakens it so that it's in the ready state). After creating a thread we have to be able to stop it, using a join function. This functionality is implemented by the function

```
int kthread_stop(struct task_struct *k);
```

which synchronises the thread pointed to by `k` with the current thread. Finally, we can use the function

```
bool kthread_should_stop(void);
```

which returns true if the current kernel thread should stop. When someone calls `kthread_stop()` on a kernel thread T, it will be woken and the function will return true. T should then return, and its return value will be passed through to `kthread_stop()`.

Synchronisation

Mutexes in the Linux kernel are instances of `struct mutex`. After having declared a `struct mutex`, we can use the following functions to handle it:

- `void mutex_init(struct mutex *mtx);`
- `void mutex_lock(struct mutex *mtx);`
- `void mutex_unlock(struct mutex *mtx);`

Interrupt synchronisation

Interrupts are handled in kernel-space, hence we have to be able to synchronise threads even when dealing with interrupts. When dealing with interrupts we can't use mutexes since they are blocking locks but interrupts can't block. Linux uses spinlocks instead, which are instances of the `struct spinlock_t` structure. After having declared a `struct spinlock_t` we can use the following functions (or macros):

- The `spin_lock_init` initialises the spinlock and must be called before using it.

```
# define spin_lock_init(_lock)              \
do {                                        \
  spinlock_check(_lock);                   \
  *(_lock) = __SPIN_LOCK_UNLOCKED(_lock); \
} while (0)
```

- The

```
static __always_inline void spin_lock(spinlock_t *lock);
```

  function must be called by the interrupt code to lock the spinlock pointed to by `lock` (or spin if the lock is taken).

- The

```
static __always_inline void spin_unlock(spinlock_t *lock);
```

  function must be called by the interrupt code to unlock the spinlock pointed to by `lock`.

- The

```
static __always_inline void spin_lock_irq(spinlock_t *lock)
```

  function must be called by the normal (i.e., non-interrupt) kernel code to lock the spinlock pointed to by `lock` (or spin if the lock is taken).

- The

```
static __always_inline void spin_unlock_irq(spinlock_t *lock);
```

  function must be called by the normal (i.e., non-interrupt) kernel code to unlock the spinlock pointed to by `lock`.

When dealing with interrupts we might also want to block norml kernel code and wait for some interrupt. This is achieved using a `struct wait_queue_head_t` structure. After having declared a `struct wait_queue_head_t` we can use the following macros:

- The

```
#define init_waitqueue_head(wq_head)                        \
do {                                                        \
    static struct lock_class_key __key;                     \
    __init_waitqueue_head((wq_head), #wq_head, &__key);     \
} while (0)
```

macro initialises the structure.

- The

```
#define wait_event_interruptible_lock_irq(wq_head, condition, spinlock)
```

macro allows normal kernel code to atomically unlock the `spinlock` and block until `condition` becomes true.

- The

```
define wake_up(&waitqueue)
```

macro allows an interrupt routine to wake one thread from the wait queue .

19.2.2 The Linux filesystem

Linux represents peripherals as files, which are handled by structures which contain pointers to the functions that have to be executed by the kernel when reading to and writing from a file. If we consider a character device, we have to initialise a `struct file_operations` structure and then assign to each of its fields a pointer to a function. For instance, we have to assign to the `write` field a pointer to the function we shall use to write some character to the file. This means that the first step for writing a driver is writing the functions that have to be used to interact with the device file.

```
ssize_t driver_write(struct file *f, const char __user *buf, size_t size, loff_t *o
    ) {
    /* code for writing buf in file f*/
}
ssize_t read(struct file *f, char __user *buf, size_t size, loff_t *o) {
    /* code for reading from file f */
}
```

Note that when working with device drivers, Linux requires us to explicitly copy data from and to user-space. This can be done calling:

- The

```
unsigned copy_from_user(void *to, const void __user *from, unsigned n);
```

function that copies `n` bytes from `from` (in user-space) to `to` (in kernel-space). This function is typically used when writing to the device file since we want to copy the data from the user-space to the kernel space and then to the file.

- The

```
unsigned copy_to_user(void __user *to, const void *from, unsigned n)
```

function which copies `n` bytes from `from` (in kernel-space) to `to` (in user-space). This function is typically used when reading from the device file since we want to copy to user-space the data read from the file (which is in kernel-space).

Then, we have to create a `struct file_operations` structure and assign to each of its field a pointer to the functions we have defined above.

```
struct file_operations driver_fops = {
    .owner=THIS_MODULE,
    .write=driver_write,
    .read=driver_read,
};
```

Finally we have to register the driver by using the `register_chrdev` function to which we have to pass the `struct file_operations` we have defined and a name for the device (i.e., a name we give to the device).

```
int major=register_chrdev(0, "device_name", &driver_fops);
```

The `register_chrdev` function returns the major number of the file, which can be used to unregister the device if passed to the `unregister_chrdev` function.

```
unregister_chrdev(major, "device_name")
```

Definitions

Concurrency, 56

Data race, 87
Deadlock, 60

Non-preemptive operating system, 5

Preemptive operating system, 5

Process, 2
Program, 2

Sequential consistency, 84

Thread, 2

Virtual machine, 114